TUGAS AKHIR - KI141502

PENERAPAN KONSEP DSU ON TREE DAN STRUKTUR DATA SEGMENT TREE PADA RANCANG ALGORITMA: STUDI KASUS SPOJ KLASIK LIS AND TREE

DIMAS HIRDA PRATAMA
NRP 05111440000147

Dosen Pembimbing 1
Rully Soelaiman, S.Kom., M.Kom.

Dosen Pembimbing 2
Abdul Munif, S.Kom., M.Sc.

DEPARTEMEN INFORMATIKA
Fakultas Teknologi Informasi dan Komunikasi
Institut Teknologi Sepuluh Nopember
Surabaya, 2018

Halaman ini sengaja dikosongkan

TUGAS AKHIR - KI141502

PENERAPAN KONSEP DSU ON TREE DAN STRUKTUR DATA SEGMENT TREE PADA RANCANG ALGORITMA: STUDI KASUS SPOJ KLASIK LIS AND TREE

DIMAS HIRDA PRATAMA
NRP 05111440000147

Dosen Pembimbing 1
Rully Soelaiman, S.Kom., M.Kom.

Dosen Pembimbing 2
Abdul Munif, S.Kom., M.Sc.

DEPARTEMEN INFORMATIKA
Fakultas Teknologi Informasi dan Komunikasi
Institut Teknologi Sepuluh Nopember
Surabaya, 2018

Halaman ini sengaja dikosongkan

UNDERGRADUATE THESES - KI141502

IMPLEMENTATION OF DSU ON TREE CONCEPT AND SEGMNT TREE DATA STRUCTURE ON ALGORITHM DESIGN ON: CASE STUDY IN SPOJ CLASSIC PROBLEM LIS AND TREEE

DIMAS HIRDA PRATAMA
NRP 05111440000147

Supervisor 1
Rully Soelaiman, S.Kom., M.Kom.

Supervisor 2
Abdul Munif, S.Kom., M.Sc.

INFORMATICS DEPARTMENT
Faculty of Information Technology and Communication
Institut Teknologi Sepuluh Nopember
Surabaya, 2018

Halaman ini sengaja dikosongkan

PENERAPAN KONSEP DSU ON TREE DAN STRUKTUR DATA SEGMENT TREE PADA RANCANG ALGORITMA: STUDI KASUS SPOJ KLASIK LIS AND TREE

TUGAS AKHIR

Diajukan Guna Memenuhi Salah Satu Syarat
Memperoleh Gelar Sarjana Komputer
pada
Bidang Studi Algoritma Pemrograman
Program Studi S-1 Departemen Informatika
Fakultas Teknologi Informasi dan Komunikasi
Institut Teknologi Sepuluh Nopember

Oleh:
Dimas Hirda Pratama
NRP: 05111440000147

Disetujui oleh Dosen Pembimbing Tugas Akhir

Rully Soelaiman, S.Kom., M.Kom. .....
NIP: 197002131994021001 (Pembimbing 1)

Abdul Munif, S.Kom., M.Sc. .....
NIP: 198608232015041004 (Pembimbing 2)

SURABAYA
JULI 2018

vii

Halaman ini sengaja dikosongkan

PENERAPAN KONSEP DSU ON TREE DAN STRUKTUR DATA SEGMENT TREE PADA RANCANG ALGORITMA: STUDI KASUS SPOJ KLASIK LIS AND TREE

Nama	: DIMAS HIRDA PRATAMA
NRP	: 05111440000147
Departemen	: Informatika FTIF-ITS
Pembimbing I	: Rully Soelaiman, S.Kom., M.Kom.
Pembimbing II	: Abdul Munif, S.Kom., M.Sc.

Abstrak

Permasalahan LIS and TREE merupakan sebuah permasalahan yang melibatkan sebuah struktur data tree. Dimana pada tree tersebut akan dicari LIS terpanjang dari seluruh simple path yang ada. Untuk menangani berbagai permasalahan pada permasalahan tersebut dibutuhkan struktur data yang mampu mendukung operasi-operasi tersebut dengan efisien.

Pada Tugas Akhir ini akan dirancang penyelesaian permasalahan LIS and TREE antara lain operasi pencarian nilai LIS pada node dan subtree saat ini, operasi update nilai LIS pada node dan subtree saat ini dan menggabungkan serta memindahkan nilai pada dua subtree yang berbeda. Struktur data klasik yang biasa digunakan dalam penyelesaian permasalahan ini merupakan salah satu jenis stuktur data Tree yaitu Segment Tree dengan menggabungkan konsep Disjoint Set Union.

Pada Tugas Akhir ini digunakan struktur data Segment Tree dan konsep Disjoint Set Union untuk menyelesaikan operasi-operasi tersebut.

Kata Kunci: Disjoint Set Union on Tree, Segment Tree, Longest Increasing Subsequence

Halaman ini sengaja dikosongkan

IMPLEMENTATION OF DSU ON TREE CONCEPT AND SEGMNT TREE DATA STRUCTURE ON ALGORITHM DESIGN ON: CASE STUDY IN SPOJ CLASSIC PROBLEM LIS AND TREEE

Name	: DIMAS HIRDA PRATAMA
NRP	: 05111440000147
Major	: Informatics Department Faculty of IT-ITS
Supervisor I	: Rully Soelaiman, S.Kom., M.Kom.
Supervisor II	: Abdul Munif, S.Kom., M.Sc.

Abstract

LIS and TREE is a problem which involves tree data structure. From that tree a LIS should be found from every simple path that exist. To handle various problem, an efficient data structure is needed to support the operations.

This undergraduate thesis will be designed problem solving for LIS and TREE such as operation to find LIS value in a subtree and node, operation to update value of LIS in a subtree and node, and also combine and transfer value of LIS between different subtree and node. Well known data structures, e.g tree specifically segment tree with combination of Disjoint Set Union concept is able to answer the problem efficiently.

In this undergraduate thesis Segment Tree data structure and Disjoint Set Union concept will be used to solve those operations.

Keywords: Disjoint Set Union on Tree, Segment Tree, Longest Increasing Subsequence

Halaman ini sengaja dikosongkan

KATA PENGANTAR

Puji syukur Penulis panjatkan kepada Allah SWT. atas pimpinan, penyertaan, dan karunia-Nya sehingga Penulis dapat menyelesaikan Tugas Akhir yang berjudul :

PENERAPAN KONSEP DSU ON TREE DAN STRUKTUR DATA SEGMENT TREE PADA RANCANG ALGORITMA: STUDI KASUS SPOJ KLASIK LIS AND TREE.

Penelitian Tugas Akhir ini dilakukan untuk memenuhi salah satu syarat meraih gelar Sarjana di Jurusan Teknik Informatika Fakultas Teknologi Informasi Institut Teknologi Sepuluh Nopember.

Dengan selesainya Tugas Akhir ini diharapkan apa yang telah dikerjakan Penulis dapat memberikan 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:

- Papa, Mama dan keluarga Penulis yang selalu memberikan perhatian, dorongan dan kasih sayang yang menjadi semangat utama bagi diri Penulis sendiri baik selama penulis menempuh masa perkuliahan maupun pengerjaan Tugas Akhir ini.
- Bapak Rully Soelaiman, S.Kom., M.Kom. selaku Dosen Pembimbing yang telah banyak meluangkan waktu untuk memberikan ilmu, nasihat, motivasi, pandangan dan bimbingan kepada Penulis baik selama Penulis menempuh

masa kuliah maupun selama pengerjaan Tugas Akhir ini.

- Bapak Abdul Munif, S.Kom., M.Sc. selaku dosen pembimbing yang telah memberikan ilmu, dan masukan kepada Penulis.
- Seluruh tenaga pengajar dan karyawan Jurusan Teknik Informatika ITS yang telah memberikan ilmu dan waktunya demi berlangsungnya kegiatan belajar mengajar di Jurusan Teknik Informatika ITS.
- Seluruh teman Penulis di Jurusan Teknik Informatika ITS yang telah memberikan dukungan dan semangat kepada Penulis selama Penulis menyelesaikan Tugas Akhir ini.
- Teman-teman, Kakak-kakak dan Adik-adik *administrator* Laboratorium Algoritma dan Pemrograman yang selalu menjadi teman untuk berbagi ilmu.

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, Juni 2018

Dimas Hirda Pratama

DAFTAR ISI

Halaman ini sengaja dikosongkan

DAFTAR TABEL

Halaman ini sengaja dikosongkan

DAFTAR GAMBAR

Halaman ini sengaja dikosongkan

DAFTAR KODE SUMBER

Halaman ini sengaja dikosongkan

BAB I
PENDAHULUAN

Pada bab ini akan dijelaskan latar belakang, rumusan masalah, batasan masalah, tujuan, manfaat, metodologi dan sistematika penulisan Tugas Akhir.

Latar Belakang

Di dalam ilmu komputer, terdapat permasalahan pengaplikasian struktur data yang tepat, guna memecahkan permasalahan yang ada. Hal ini dikarenakan banyaknya jenis struktur data dan kegunaannya yang beragam. Peneliti mencoba mencari beberapa permasalahan yang berkaitan dengan pengaplikasian struktur data dalam dunia pemrograman dan menemukan permasalahan di situs *online* SPOJ. Di dalam situs tersebut terdapat sebuah permasalahan dengan nama permasalahan klasik *LIS and TREE*[1].

Pada permasalahan ini diberikan masukan berupa bilangan bulat yang disimbolkan N, yang merepresentasikan jumlah *node* di sebuah *tree*. Dari *tree* tersebut diminta untuk mencari *Longest Increasing Subsequence* (LIS).
Hasil Tugas Akhir ini diharapkan dapat memberi gambaran mengenai penerapan *disjoint set union* pada struktur data *tree*.

Rumusan Masalah

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

1. Bagaimana menyelesaikan permasalahan klasik *LIS and*

TREE[1] pada situs penilaian SPOJ?
2. Bagaimana menerapkan konsep *disjoint set union* pada struktur data *segment tree*?
3. Bagaimana kinerja algoritma *disjoint set union* pada struktur data *segment tree* dalam menyelesaikan permasalahan klasik *LIS and TREE*[1]?

Batasan Masalah

Permasalahan yang dibahas pada Tugas Akhir ini memiliki beberapa batasan, yaitu sebagai berikut:

1. Implementasi algoritma menggunakan bahasa pemrograman C++.
2. Besar ukuran berkas masukkan dalam sekali uji maksimal 2 MB.
3. Banyak kasus uji disimbolkan T, dengan rentang 1 - 1000.
4. Jumlah *node* yang disimbolkan N berkisar antara 1 - 100.000.
5. Berikutnya sebanyak N-1 baris diinputkan 2 angka disimbolkan a dan b dengan syarat a $\geq$ 1 dan b $\leq$ N, yang merepresentasikan bahwa vertex a terhubung dengan vertex b.
6. Kombinasi dari vertex a dan b haruslah unik.
7. Penyimpanan yang dibutuhkan dalam 1 kali percobaan kurang dari 1536 MB.
8. Batas waktu pada program untuk setiap percobaan kurang dari 1 detik.

Tujuan

Tujuan dari Tugas Akhir ini adalah sebagai berikut:

1. Melakukan desain dan implementasi penyelesaian permasalahan klasik *LIS and TREE*[1] pada situs penilaian SPOJ.

2. Menganalisis hasil kinerja penyelesaian permasalahan klasik *LIS and TREE*[1].

Metodologi

Ada beberapa tahap dalam proses pengerjaan tugas akhir ini, yaitu sebagai berikut:

1. Studi Literatur
   Pada tahap ini, dilakukan studi mengenai pengimplementasian *disjoint set union* pada struktur data *segment tree* serta algoritman penyelesaian *Longest Increasing Subsequence*. Sumber studi antara lain dari buku-buku literatur, situs-situs pembelajaran *online*, dan materi kuliah yang bersangkutan.
2. Implementasi
   Pada Tahap ini, akan dilakukan pembangunan dari algoritma yang telah dipelajari menjadi sebuah sistem yang dapat digunakan. Bahasa pemrograman yang digunakan adalah C++ dengan menggunakan IDE(*Integrated Development System*) Dev-C++.
3. Uji Coba
   Tahap ini merupakan tahap pengujian aplikasi dengan data masukan yang telah ditentukan untuk menguji kebenaran hasil implementasi algoritma serta menguji waktu eksekusi aplikasi untuk data masukan yang telah ditentukan. Pada tahap ini juga dilakukan optimasi dari hasil implementasi apabila aplikasi masih kurang efisien.
4. Penyusunan buku tugas akhir
   Tahap ini merupakan tahap penyusunan laporan berupa buku tugas akhir sebagai dokumentasi pelaksanaan tugas akhir yang mencakup seluruh teori, implementasi serta hasil pengujian yang telah dikerjakan.

Sistematika Penulisan Laporan

Berikut adalah sistematika penulisan buku Tugas Akhir ini:

1. BABI: PENDAHULUAN
   Bab ini berisi latar belakang, rumusan masalah, batasan masalah, tujuan, manfaat, metodologi dan sistematika penulisan Tugas Akhir.
2. BAB II: DASAR TEORI
   Bab ini berisi dasar teori mengenai permasalahan dan algoritma penyelesaian yang digunakan dalam Tugas Akhir
3. BAB III: DESAIN
   Bab ini berisi desain algoritma dan struktur data yang digunakan dalam penyelesaian permasalahan.
4. BAB IV: IMPLEMENTASI
   Bab ini berisi implementasi berdasarkan desain algortima yang telah dilakukan pada tahap desain.
5. BAB V: UJI COBA
   Bab ini berisi uji coba dan evaluasi dari hasil implementasi yang telah dilakukan pada tahap implementasi.
6. BAB VI: PENUTUP
   Bab ini berisi kesimpulan dan saran yang didapat dari hasil uji coba yang telah dilakukan.

BAB II
DASAR TEORI

Pada bab ini akan dijelaskan mengenai dasar teori yang menjadi dasar pengerjaan Tugas Akhir ini.

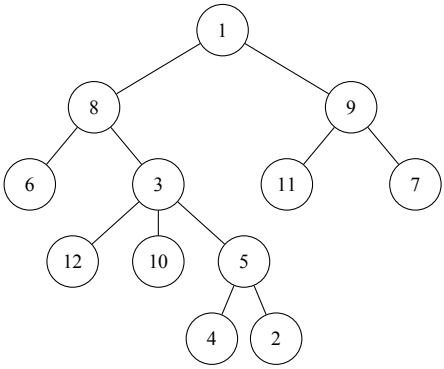
Deskripsi Umum Permasalahan

Permasalahan yang diangkat pada Tugas Akhir ini adalah mencari *Longest Increasing Subsequence* (LIS) dari sebuah *tree*, dimana *node* dalam *tree* tersebut sesuai dengan masukan yang diberikan, dan terdapat *edge* yang saling menghubungkannya. Contoh masukan yang diberikan pada daring terlihat pada Tabel 2.1.

Tabel 2.1 Contoh masukan dari daring

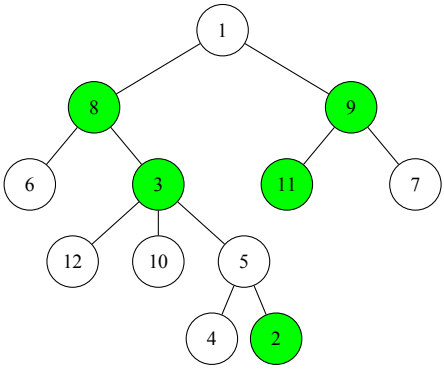
2
12
1 8
8 6
8 3
3 12
3 10
3 5
5 4
5 2
1 9
9 11
9 7

Dari masukan tadi akan terbentuk sebuah *tree* sesuai seperti angka yang telah dimasukan, visualisasi dari *tree* dapat dilihat di Gambar 2.1.



Gambar 2.1 Bentuk akhir tree sesuai masukan dari daring

Dari *tree* tersebut akan dicari LIS sesuai dengan permasalahan soal, untuk contoh masukan tersebut keluaran LIS adalah 4. Visualisasi jawaban terlihat pada Gambar 2.2.



Gambar 2.2 *Vertex* dengan warna hijau menunjukkan *node* yang menjadi anggota LIS.

Longest Increasing Subsequence

Dalam ilmu komputer, permasalahan *Longest Increasing Subsequence* atau yang biasa disingkat menjadi LIS, merupakan permasalahan yang bertujuan untuk menemukan *subsequence* terpanjang dari sebuah *sequence* dimana *subsequence* tersebut memiliki elemen yang sudah disortir dari nilai terkecil ke terbesar. Untuk dapat menyelesaikannya algoritma yang dibutuhkan adalah sebagai berikut:

Dimisalkan terdapat sebuah *array* bernama $A[]$ yang memiliki panjang tertentu, dan penulis juga memiliki LIS terpanjang sementara atau disebut *active list* dengan panjang tertentu. Kemudian penulis ingin menambahkan elemen ke-i atau $A[i]$ maka harus memperhatikan beberapa hal sebagai berikut[2]:

1. jika $A[i]$ adalah elemen terkecil dari *active list* maka buat *active list* baru dengan panjang 1.
2. Jika $A[i]$ adalah elemen terbesar dari *active list* maka lakukan duplikasi terhadap *active list* terpanjang dan tambahkan elemen tersebut di akhir.
3. Jika $A[i]$ bukan elemen terbesar ataupun terkecil maka cari *active list* dengan elemen terakhir yang terbesar dan lebih kecil dari $A[i]$, lakukan duplikasi dan tambahkan elemen tersebut, dan hapus *list* lain dengan panjang yang sama.

Contoh

Untuk menemukan LIS dari array $A[0, 8, 4, 12, 2, 10, 14]$

1. $A[0] = 1$, karena merupakan elemen pertama dan belum ada *active list* maka masuk ke *case* 1.

   $[0]$

2. $A[1] = 8$, karena angka 8 lebih besar daripada 0, maka masuk ke *case* 2.

[0]
[0, 8]

3. $A[2] = 4$, karena angka 4 berada di antara 0 dan 8, maka akan masuk ke *case* 3.

[0]
[0, 4]
~~[0, 8]~~ (dihilangkan)

4. $A[3] = 12$, karena angka 12 lebih besar daripada 4, maka masuk ke *case* 1.

[0]
[0, 4]
[0, 4, 12]

5. $A[4] = 2$, karena angka 2 berada di antara 0 dan 12 maka akan masuk ke *case* 3.

[0]
[0, 2]
~~[0, 4]~~ (dihilangkan)
[0, 4, 12]

6. $A[5] = 10$, karena angka 10 berada di antara 0 dan 12 maka akan masuk ke *case* 3.

[0]
[0, 2]
[0, 2, 10]

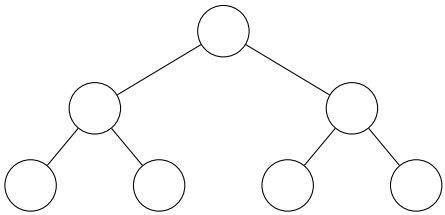
~~[0, 4, 12]~~ (dihilangkan)

7. $A[6] = 14$, karena angka 14 lebih besar daripada 10 maka akan masuk ke *case* 1.

   $[0]$
   $[0, 2]$
   $[0, 2, 10]$
   $[0, 2, 10, 14]$
   merupakan LIS dari array tersebut.

Binary Tree

Binary Tree merupakan salah satu jenis struktur data *Tree*, yang memiliki maksimal hanya dua *child* untuk setiap *node* yang ada, kedua *child* tersebut biasa disebut dengan istilah *left child* dan *right child*. Gambar 2.3 menunjukkan contoh dari *binary tree*.



Gambar 2.3 Visualisasi dari bentuk struktur data *Binary Tree*.

Segment Tree

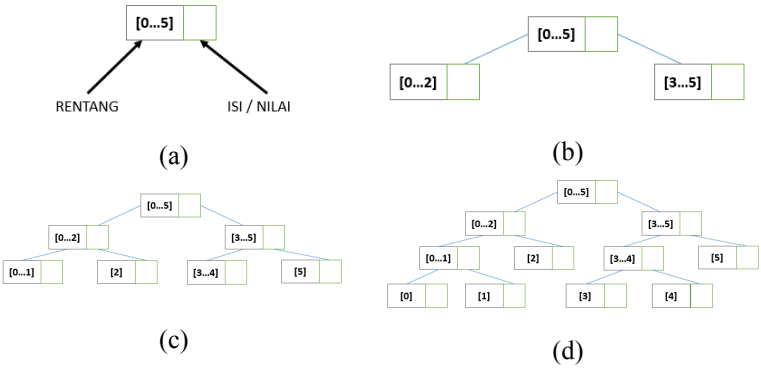
Segment Tree merupakan salah satu jenis struktur data yang merepresentasikan suatu nilai yang terhubung dengan suatu jeda atau interval tertentu di dalam *array*[3].

Dimisalkan terdapat sebuah struktur data *segment tree* yang menyimpan interval untuk suatu *array* $A[]$ dengan panjang n.Maka

rumusan algoritma yang dapat digunakan untuk konstruksi *segment tree* sebagai berikut[4]:

1. Buat node yang berisikan seluruh elemen *array* $A[]$.
2. Jika elemen di dalam node beranggotakan lebih dari satu, maka buat 2 *child node* baru yang beranggotakan elemen $A[0, ..., n/2]$ dan $A[(n/2) + 1, ..., n]$.
3. Jika elemen di dalam node hanya beranggotakan 1 elemen maka proses dihentikan.

Pada Gambar 2.4a menunjukkan *node* yang menyimpan nilai dari seluruh rentang yang ada menjadi *root* dari *segment tree*.



Gambar 2.4 Proses pembentukan *Segment Tree*

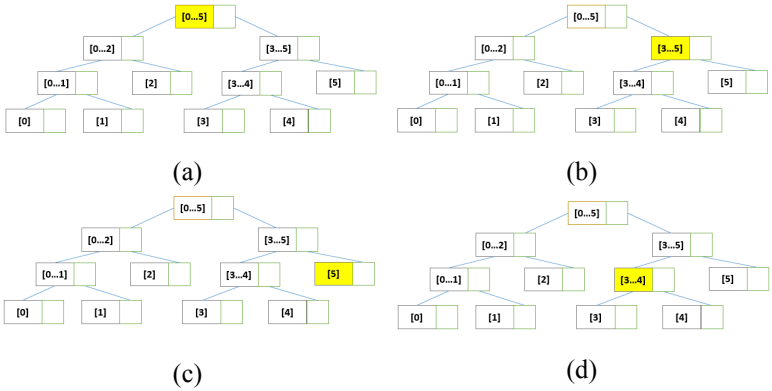
Pada Gambar 2.4b menunjukkan proses rekursi untuk membuat dua *child* baru, *child* kiri menyimpan nilai dari rentang 0 hingga 2, *child* kanan menyimpan nilai dari rentang 3 hingga 5.

Pada Gambar 2.4c menunjukkan lanjutan proses rekursi lagi, namun untuk *node* yang menyimpan hanya satu rentang saja, yaitu *node* yang menyimpan rentang 2 dan 5 proses dihentikan, bentuk akhir dari *segment tree* terlihat pada Gambar 2.4d

Selain itu pada umumnya struktur data *segment tree* juga memiliki fungsi $query()$ untuk mengambil nilai dari suatu rentang dan juga $update()$ untuk merubah nilai pada suatu rentang. Algoritma untuk melakukan fungsi $query()$ sebagai berikut:

1. Cek apakah nilai dari rentang yang diinginkan berada dalam rentang saat ini.
2. Jika iya ambil nilai yang berada di rentang tersebut.
3. Jika tidak lakukan rekursi ke *left child* dan *right child*.

Untuk visualisasi proses $query()$ akan diambil nilai pada rentang 1 sampai 4. Pada Gambar 2.5a proses dimulai dengan melakukan cek pada *root* apakah rentang yang diinginkan masih termasuk atau tidak, *root* menyimpan nilai dari rentang 0 hingga 5, maka rentang yang diinginkan masih termasuk didalamnya, dilakukan proses rekursi ke *child*.

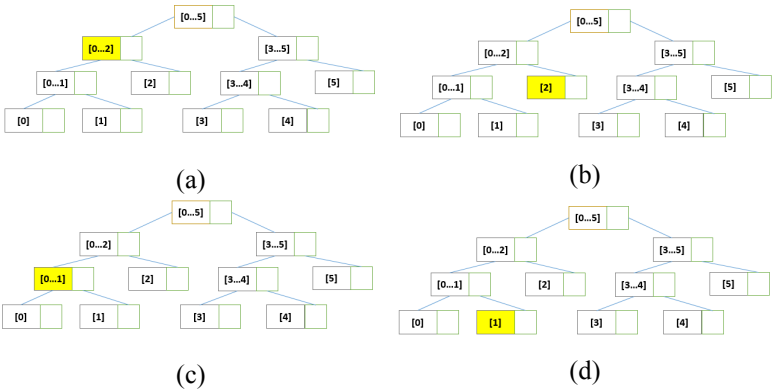


Gambar 2.5 Proses rekursi $query$ pada *right child* dari *Segment Tree*

Pada Gambar 2.5b dilakukan pengecekan pada *node* yang menyimpan rentang nilai 3 hingga 5, karena masih mencakup rentang yang dicari maka dilanjutkan proses rekursi pada *child node* tersebut.

Pada Gambar 2.5c proses dilakukan pada *node* yang menyimpan rentang nilai 5 saja, karena rentang tersebut tidak dicari, maka proses pada *node* tersebut dihentikan. Pada Gambar 2.5d dilakukan pengecekan pada *node* yang menyimpan rentang nilai 3 hingga 4, dimana *node* tersebut menyimpan nilai sesuai dengan rentang yang diinginkan, maka akan diambil nilainya dan dibandingkan dengan *sibling* dari *node* tersebut, untuk kemudian diambil nilai maksimum atau minimum sesuai kebutuhan.

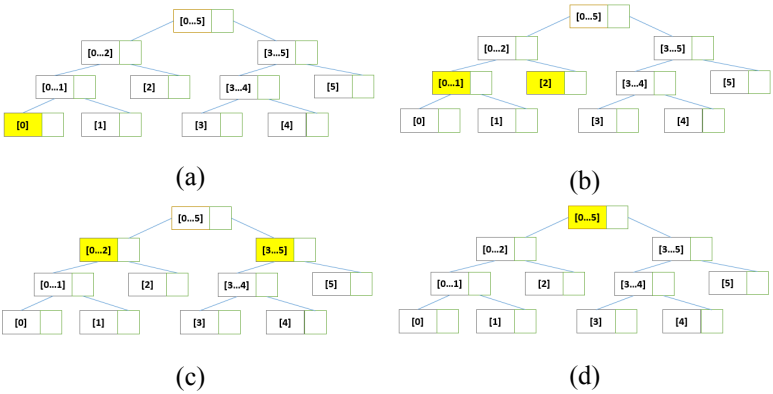
Rekursi dilanjutkan ke *child* yang berada pada sisi kiri dari *root*. Pada Gambar 2.6a proses dilakukan pada *node* yang menyimpan rentang nilai dari rentang 0 hingga 2, dikarenakan rentang tersebut mencakup rentang yang diinginkan didalamnya, maka dilakukan proses rekursi ke *child node* tersebut.



Gambar 2.6 Proses rekursi $query$ pada *left child* dari *Segment Tree*

Pada Gambar 2.6b, karena *node* tersebut termasuk pada rentang yang diinginkan maka nilai didalamnya akan diambil, untuk dibandingkan dengan *sibling* dari *node* tersebut nantinya. Proses dilanjutkan ke *node* yang menyimpan rentang 0 hingga 1, seperti pada Gambar 2.6c.

Pada Gambar 2.6d dilakukan pengecekan pada *node* yang menyimpan rentang nilai 1, dimana *node* tersebut menyimpan nilai sesuai dengan rentang yang diinginkan. Sedangkan pada Gambar 2.7a, nilai pada rentang 0 tidak termasuk nilai yang diinginkan.



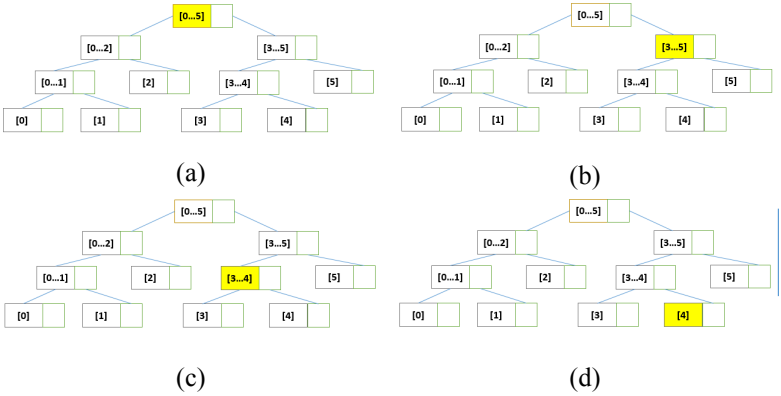
Gambar 2.7 Proses pengembalian nilai $query$ ke *root* pada *Segment Tree*

Sebelum mengembalikan nilai ke atas, akan dilakukan perbandingan nilai, dengan *sibling node* tersebut, untuk mengambil nilai yang maksimum atau minimum sesuai dengan kebutuhan, proses ini terlihat pada Gambar 2.7b dan 2.7b. Pada akhirnya nilai yang dicari akan berada pada *root*, terlihat pada Gambar 2.7d, menandakan bahwa proses $query()$ telah selesai.

Algoritma untuk melakukan fungsi $update()$ sebagai berikut:

1. Cek apakah nilai dari rentang yang diinginkan berada dalam rentang saat ini.
2. Jika iya ambil *update* nilai yang berada di rentang tersebut.
3. Jika tidak lakukan rekursi ke *left child* dan *right child*.

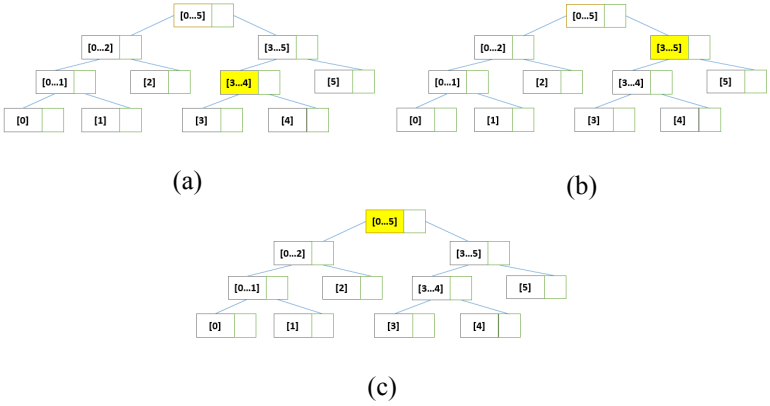
Sebagai contoh akan dilakukan $update()$ pada *node* yang menyimpan nilai rentang 4. Awal proses dimulai dengan melakukan pengecekan pada *root*, seperti yang terlihat pada Gambar 2.8a. Karena di dalam *root* mencakup rentang yang ingin diupdate, maka dilakukan rekursi ke *child*.



Gambar 2.8 Proses pencarian *node* untuk melakukan $update$ pada *Segment Tree*

Karena *left child* dari *root* tidak menyimpan nilai dari rentang yang diinginkan maka proses pada *left child* tidak dilanjutkan. Sedangkan di dalam *right child* dari *root* menyimpan rentang yang diinginkan, maka akan dilakukan rekursi pada *right child*, seperti pada Gambar 2.8b. Pada Gambar 2.8c proses dilanjutkan, karena *node* tersebut masih menyimpan rentang yang diinginkan, maka proses rekursi dilanjutkan ke *child*nya. Pada Gambar 2.8d proses telah mencapai *node* dengan rentang diinginkan, maka akan dilakukan perubahan pada nilai yang disimpan didalamnya.

Setelah nilai telah diubah, maka sebelum mengembalikan nilai ke *parent*nya akan dilakukan perbandingan nilai dengan *node* yang menjadi *sibling*nya untuk mengambil nilai maksimum atau minimum sesuai dengan kebutuhan, proses ini terlihat pada Gambar 2.9a dan 2.9b, hingga akhirnya nilai kembali pada *root* seperti pada Gambar 2.9c.



Gambar 2.9 Proses pengembalian nilai hasil $update$ pada *Segment Tree*

Heavy-Light Decomposition

Heavy-Light Decomposition adalah teknik dekomposisi sebuah *tree* menjadi sebuah *disjoint chains* (tidak ada 2 *chain* yang memiliki *node* yang sama). Disebut *Heavy-Light* karena dekomposisi dilakukan berdasarkan kriteria yang telah ditentukan untuk membedakan apakah *chain* tersebut merupakan *node heavy* atau *light*.

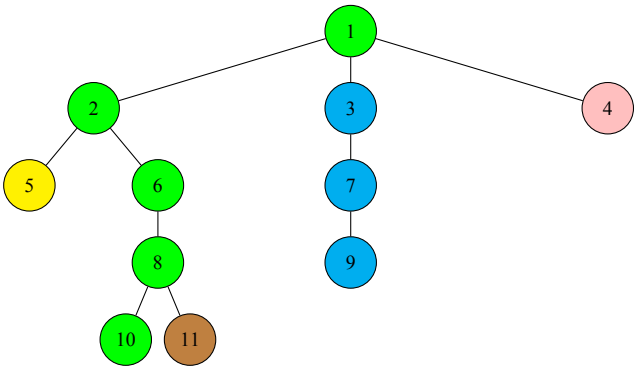
Penggunaan konsep *Heavy-Light Decomposition*, yakni mengumpamakan sebuah *node* mempunyai satu *special child* yang merupakan sebuah *subtree* dengan ukuran paling besar di antara *child* lainnya. sehingga dapat diumpamakan *tree* yang telah dibuat terdiri dari beberapa *chain* dimana setiap *chain* mempunyai

head yang bukan merupakan *special child* dari *parent child* tersebut.

Setelah membentuk *tree*, selanjutnya dicatat *edge* mana yang merupakan *heavy edge* dan *light edge*. Dengan aturan berikut:

1. *Heavy edge* adalah *edge* dengan jumlah *child* lebih besar atau sama dengan setengah jumlah *child* dari *parent child* tersebut.
2. *Light edge* adalah *edge* dengan jumlah *child* kurang dari setengah jumlah *child* dari *parent child* tersebut.

Pada Gambar 2.10 dapat dilihat terdapat banyak warna berbeda, setiap warna menggambarkan *chain* yang berbeda, dan *heavy chain* untuk *tree* tersebut ditandai dengan warna hijau.



Gambar 2.10 Contoh penggambaran *Heavy-Light Decomposition*

Disjoint Set Union

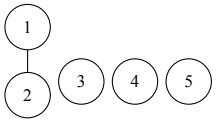
Struktur data *disjoint set* adalah sebuah struktur data yang menyimpan sekumpulan himpunan atau *set*. Dua himpunan bisa disebut *disjoint* jika kedua himpunan tersebut tidak memiliki perpotongan, atau perpotongannya sama dengan *null*. Sebagai contoh himpunan $\{1, 2\}$ dan $\{3, 4\}$ merupakan himpunan yang

disjoint karena tidak memiliki elemen yang sama diantara keduanya.

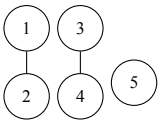
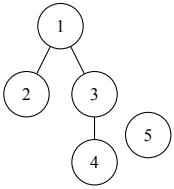
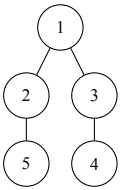
Disjoint set union sendiri merupakan suatu algoritma yang digunakan untuk menyatukan himpunan yang *disjoint* dengan himpunan lainnya. Hal ini dapat dilakukan dengan melakukan operasi $merge(a, b)$, dimana a dan b adalah dua himpunan yang saling *disjoint*.

Contoh:
Dimisalkan terdapat 5 buah himpunan bagian yang memiliki anggota berjumlah 1 yaitu: $\{1\}$, $\{2\}$, $\{3\}$, $\{4\}$, dan $\{5\}$. Kemudian dilakukan operasi seperti pada Tabel 2.2 dan 2.3.

Tabel 2.2 Visualisasi Himpunan bagian dan proses $merge()$ bagian 1.

No	Operasi	Visualisasi Himpunan	Keterangan
1	Kondisi Awal		Kondisi awal himpunan.
2	$merge(2, 1)$		Kondisi setelah $merge(2, 1)$.

Tabel 2.3 Visualisasi Himpunan bagian dan proses $merge()$ bagian 2.

3	$merge(4, 3)$		Kondisi setelah $merge(4, 3)$
4	$merge(3, 1)$		Kondisi setelah $merge(3, 1)$
5	$merge(5, 2)$		Kondisi setelah $merge(5, 2)$

Preorder Numbering

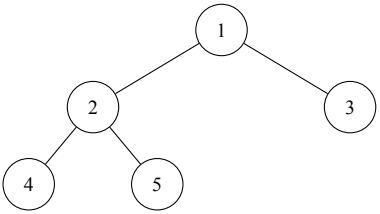
Preorder merupakan salah satu metode *traversing* sebuah *tree*, yang juga merupakan bagian dari algoritma *Depth-First Search* (DFS), yaitu sebuah algoritma yang mencari *depth* atau kedalaman sedalam mungkin pada setiap *child* sebelum melanjutkan ke *sibling* selanjutnya.

Algoritma dari *preorder* adalah sebagai berikut:

1. Cek apakah *node* saat ini kosong
2. Tampilkan nilai/data dari *node* saat ini.

3. Lakukan *traverse* ke *child* sebelah kiri secara rekursif.
4. Lakukan *traverse* ke *child* sebelah kanan secara rekursif.

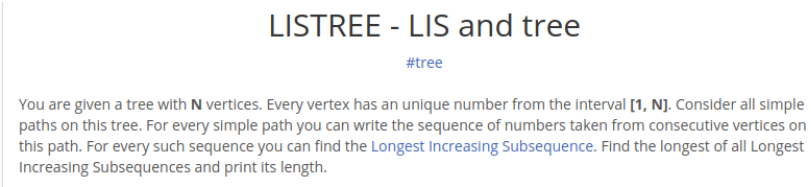
Visualisasi dari *preorder* dapat dilihat pada Gambar 2.11.



Gambar 2.11 Visualisasi dari *preorder* urutan yang sesuai menjadi 1,2,4,5,3

Permasalahan *LIS and TREE* pada SPOJ

Pada subbab ini akan dijelaskan mengenai permasalahan yang unik, untuk membuktikkan kebenaran implementasi *disjoint set union* pada struktur data *tree*. Contoh permasalahan tersebut adalah permasalahan pada situs penilaian daring SPOJ dengan judul permasalahan *LIS and TREE* dengan kode soal *LISTREE* deskripsi soal ditunjukkan oleh Gambar 2.12.


LISTREE - LIS and tree

#tree

You are given a tree with **N** vertices. Every vertex has an unique number from the interval **[1, N]**. Consider all simple paths on this tree. For every simple path you can write the sequence of numbers taken from consecutive vertices on this path. For every such sequence you can find the Longest Increasing Subsequence. Find the longest of all Longest Increasing Subsequences and print its length.

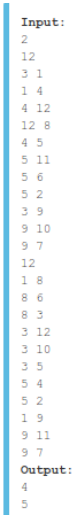
Gambar 2.12 Deskripsi soal *LIS and TREE* pada situs penilaian daring SPOJ

Di dalam soal tersebut dideskripsikan terdapat sebuah *tree* dengan N *node*. Di mana setiap *node* memiliki angka yang unik,

dengan interval 1 sampai N. Dalam soal ini di deskripsikan bahwa setiap *path* merupakan *simple path*, yang memiliki arti bahwa tidak ada *node* yang berulang. Untuk setiap *path* dapat dituliskan serangkaian angka yang diambil secara berurutan dari setiap *node*. Untuk setiap rangkaian tersebut dapat dicari LIS nya masing-masing. Tujuan dari soal ini adalah untuk menemukan LIS terpanjang dari semua LIS yang terbentuk dari setiap *path*.

Format masukan diawali dengan sebuah bilangan bulat T dimana $T(1 \leq T \leq 1000)$ yang menggambarkan banyaknya *testcase* atau *data sets*. Kemudian untuk setiap T terdapat sebuah bilangan bulat N dimana $N(1 \leq N \leq 100000)$ yang merepresentasikan jumlah *node* untuk setiap *testcase* yang nantinya akan membentuk *tree*. Setelah itu terdapat $N-1$ baris dimana terdapat dua bilangan bulat a dan b dimana $(1 \leq a, b \leq N)$ yang menunjukkan terdapat sebuah *edge* yang menghubungkan *node* a dan *node* b.

Format keluaran berupa sebuah bilangan bulat yang menunjukkan panjang dari LIS terpanjang di antara semua *simple path*. Contoh format masukan dan keluaran dapat dilihat pada Gambar 2.13.

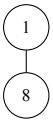
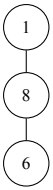
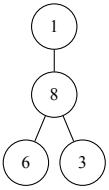


```
Input:
2
12
3 1
1 4
4 12
12 8
4 5
5 11
5 6
5 2
3 9
9 10
9 7
12
1 8
8 6
8 3
3 12
3 10
3 5
5 4
5 2
1 9
9 11
9 7
Output:
4
5
```

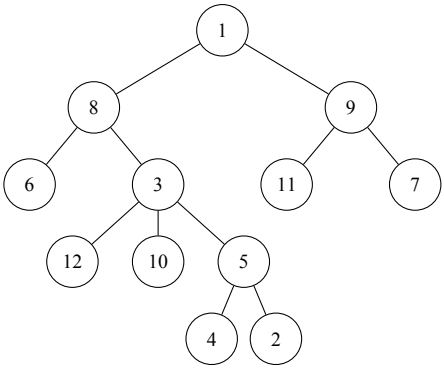
Gambar 2.13 Contoh format masukan pada situs penilaian daring SPOJ

Dari contoh format masukkan pada Gambar 2.13 diminta jumlah kasus uji (T) sebanyak 2, kemudian pada kasus uji kedua diberi masukkan 12 yang menggambarkan jumlah *node* (N). Kemudian terdapat 11 baris yang berisikan dua bilangan bulat yaitu a dan b, yang mengindikasikan bahwa kedua *node* saling terhubung, dan *node* a menjadi *parent* dari *node* b, dimana pada akhirnya akan membentuk sebuah *tree*. Proses pembentukkan tersebut dapat dilihat di Tabel 2.4.

Tabel 2.4 Visualisasi proses pembentukkan *tree* dari masukan pada daring.

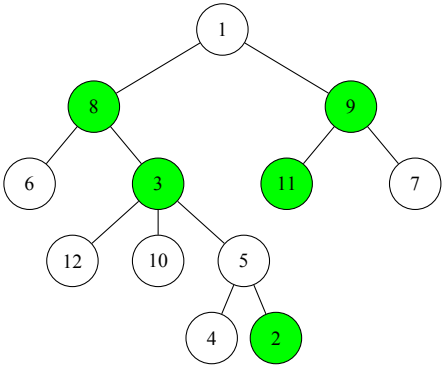
No	Masukan a dan b	Bentuk Tree	Keterangan
1	$a = 1$ $b = 8$		Masukan baris ke-1
2	$a = 8$ $b = 6$		Masukan baris ke-2
3	$a = 8$ $b = 3$		Masukan baris ke-3

Proses berlanjut hingga masukan terakhir, hingga membentuk *tree* seperti pada Gambar 2.14.



Gambar 2.14 Bentuk akhir tree sesuai masukan ke-2 pada contoh masukan

Dari contoh masukan kedua di situs penilaian daring SPOJ menunjukkan keluaran LIS yaitu 5, visualisasi untuk jawaban tersebut bisa dilihat di Gambar 2.15.



Gambar 2.15 *Vertex* dengan warna hijau menunjukkan LIS dari contoh masukan kedua dari situs penilaian daring SPOJ

Beberapa batasan yang terdapat pada permasalahan *LIS and TREE* adalah sebagai berikut:

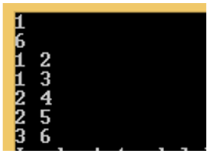
1. Besar ukuran berkas masukkan dalam sekali uji maksimal 2 MB.
2. $(1 \leq T \leq 1000)$.
3. $N(1 \leq N \leq 100000)$
4. $(1 \leq a, b \leq N)$
5. Kombinasi dari node a dan b haruslah unik.
6. Batas memori: 1536MB
7. Batas sumber kode 50000B
8. Batas waktu 2 detik.

Penyelesaian Permasalahan *LIS and TREE* pada SPOJ

Pada subbab ini akan dijelaskan bagaimana menyelesaikan permasalahan *LIS and TREE* dengan penjelasan permasalahan seperti pada subbab sebelumnya.

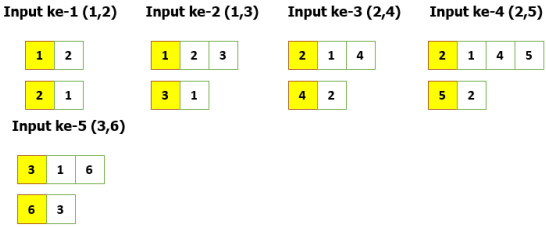
Sesuai pada deskripsi masalah, dapat diketahui bahwa setiap *node* memiliki *parent* yang telah ditentukan sesuai dengan masukan, sesuai dengan penjelasan pada subbab 2.8.

Untuk mengatasi hal tersebut maka digunakanlah konsep *Disjoint set union* dengan menggunakan *array* 2 dimensi, sebagai contoh akan digunakan masukan seperti pada Gambar 2.16.



Gambar 2.16 Contoh masukan yang digunakan.

Visualisasi memasukan masukan pada contoh ke *array* divisualisasikan pada Gambar 2.17.



Gambar 2.17 Visualisasi pengolahan masukan

Dari pengolahan masukan tersebut terlihat bahwa elemen pertama untuk setiap *array* menandakan sebagai *parent* dari *node* tersebut, kecuali untuk *node* 1 hal ini disebabkan *node* tersebut akan menjadi *root*. Maka dapat terbentuklah *tree* seperti pada Gambar 2.18.

Gambar 2.18 Visualisasi bentuk *tree* dari contoh masukan

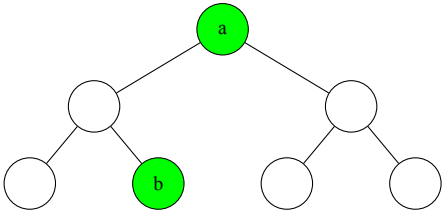
Dalam persoalan *LIS and TREE* LIS yang dicari merupakan sebuah *path* dari *tree* itu sendiri, dan *path* ini merupakan sebuah *simple path* seperti yang telah dijelaskan pada subbab 2.7. Maka untuk mendapatkan nilai LIS tersebut, secara umum terdapat beberapa *case* agar mendapatkan nilai LIS dari setiap *path* yang ada.

Case yang ada adalah sebagai berikut:

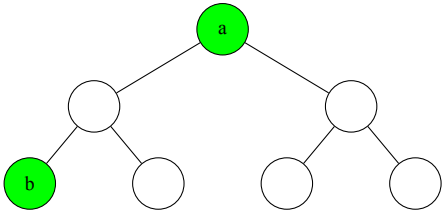
1. *Case* yang pertama adalah jika nilai *node* yang lebih besar berada di atas *node* tersebut, seperti pada Gambar 2.19.

Menunjukkan bahwa nilai *node* $a > b$, sehingga arah *path* dari bawah menuju ke atas.

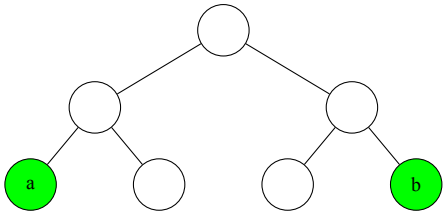
2. *Case* yang kedua adalah jika nilai *node* yang lebih besar berada di bawah *node* tersebut, seperti pada Gambar 2.20. Menunjukkan bahwa nilai *node* $a < b$, sehingga arah *path* dari atas menuju ke bawah.
3. *Case* yang kedua adalah jika nilai *node* yang lebih besar berada di *depth* yang sama dari *node* tersebut, seperti pada Gambar 2.21. Menunjukkan bahwa nilai *node* $a > b$, sehingga arah *path* dari bawah menuju ke atas kemudian menuju ke bawah.



Gambar 2.19 Visualisasi *case* ketika nilai *node child* >*parent*.



Gambar 2.20 Visualisasi *case* ketika nilai *node parent* >*child.*



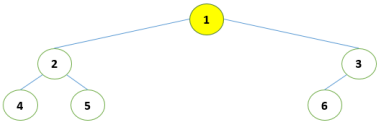
Gambar 2.21 Visualisasi *case* ketika nilai *node* yang lebih besar merupakan *sibling*

Untuk dapat menyimpan hasil LIS dari setiap *path* maka diperlukan dua *segment tree* yang akan menyimpan nilai dari tiap interval. *Segment tree* pertama akan menyimpan nilai LIS itu sendiri dan yang kedua akan menyimpan nilai LDS (Longest Decreasing Subsequence) kemudian nantinya jawaban akhir merupakan hasil perbandingan nilai dari dua *segment tree* tersebut.

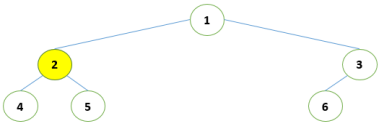
Secara umum algoritma penyelesaian yang dilakukan untuk setiap *node* adalah sebagai berikut:

1. Tentukan *child* yang menjadi *big child* atau *small child* menggunakan konsep *Heavy-light Decomposition*.
2. Lakukan rekursi ke *small child*
3. Lakukan perulangan untuk mencari LIS dan LDS dari *parent* pada seluruh *child* yang bukan merupakan *bigchild*.
4. Simpan hasil LIS dan LDS tersebut ke *Segment Tree*.
5. Cari nilai LIS dan LDS dari *child* ke *parent*.
6. Simpan hasil LIS dan LDS tersebut ke *Segment Tree*.
7. Setelah selesai, kembalikan *Segment Tree* ke kondisi semula.
8. Lakukan rekursi ke *big child*.
9. Ulangi proses ke 3 hingga 6.
10. Setelah selesai, biarkan kondisi *Segment Tree* karena akan digunakan kembali.
11. Lakukan proses hingga semua *node* terlewati.

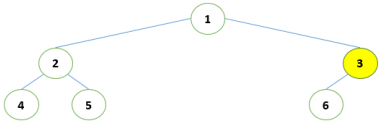
Proses diawali dari *root* yaitu melakukan pengecekan *child* mana yang merupakan *big* atau *small child* seperti pada Gambar 2.22a, 2.22b, dan 2.22c.



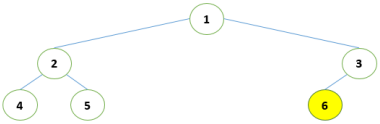
(a) Memeriksa *child* dari root



(b) *Big child* dari *root*



(c) *Small child* dari *root*

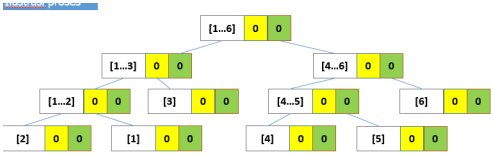


(d) *Big child* dari *node* 3

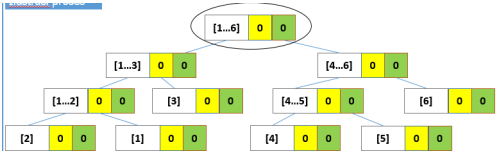
Gambar 2.22 Penentuan *big child* dan *small child* dari *root*

Dari proses sebelumnya diketahui bahwa *small child* dari *root* adalah *node* 3, maka proses akan dilanjutkan pada *node* 3. Pada *node* tersebut dilakukan kembali pencari *big* dan *small child*. Karena hanya memiliki satu *child* saja, maka proses akan langsung dilanjutkan pada *big child*, yaitu *node* 6, ditunjukkan oleh Gambar 2.22d.

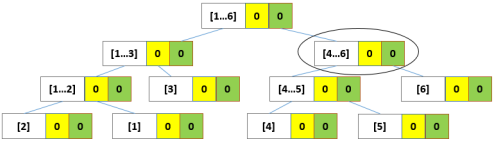
Pada *node* 6 dilakukan pencarian LIS dan LDS seperti yang tertulis pada algoritma penyelesaian sebelumnya, pada Gambar 2.23 hingga 2.24c dilakukan proses $query()$ untuk mencari LIS pada *segment tree* dari rentang 1 hingga 5. Proses yang terjadi seperti yang telah dijelaskan pada subbab 2.4.



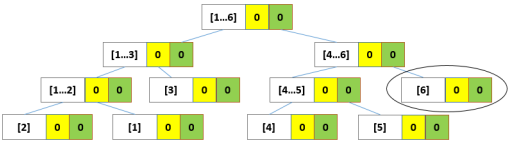
(a) Kondisi awal *segment tree*



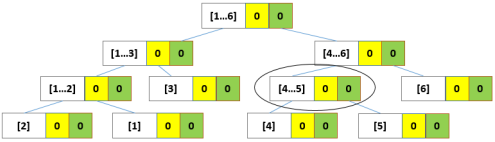
(b) Memeriksa rentang 1 hingga 6



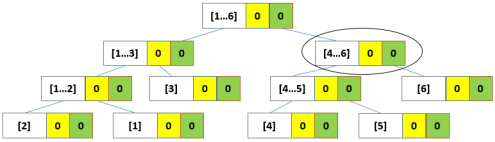
(c) Memeriksa rentang 4 hingga 6



(d) *Out of range*

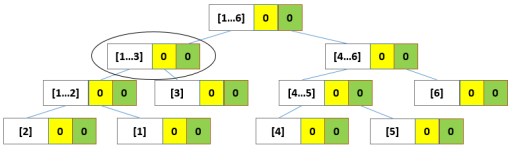


(e) Mengambil nilai rentang 4 hingga 5

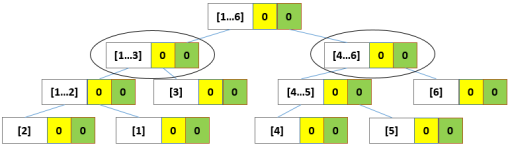


(f) Mengembalikan nilai ke atas

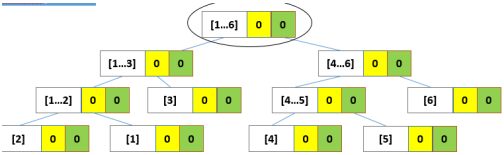
Gambar 2.23 Proses *query* nilai LIS *node* 6 pada *right child* dari *segment tree*



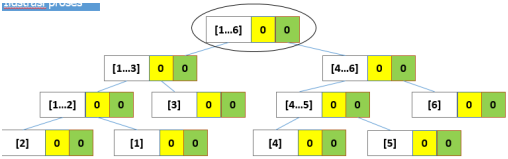
(a) Mengambil nilai rentang 1 hingga 3



(b) Membandingkan nilai dengan *sibling*



(c) Mengembalikan nilai ke *root*

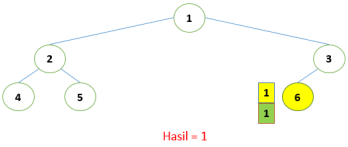


(d) *Out of range*

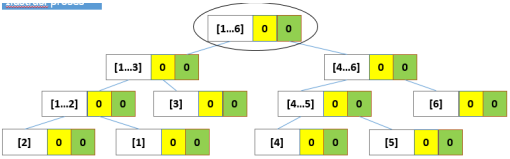
Gambar 2.24 Proses *query* nilai LIS *node* 6 pada *left child* dari *segment tree* dan pengembalian nilai ke *root*

Pada Gambar 2.24d dilakukan pencarian LDS untuk *node* 6, dengan melakukan *query*() rentang 7 pada *segment tree*, karena hanya tersedia hingga rentang 6 maka langsung mengembalikan nilai.

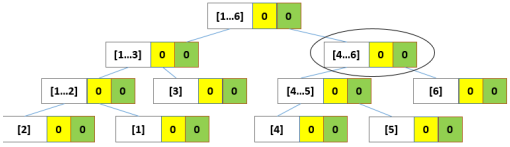
Kemudian hasil $query()$ disimpan pada variabel sementara milik *node* 6, terlihat pada Gambar 2.25a nilai LIS dan LDS sementara adalah 1, maka hasil sementara saat ini adalah 1.



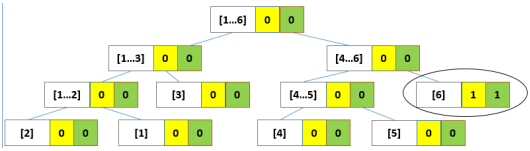
(a) Kondisi setelah proses $query$ *node* 6



(b) Memeriksa *root*



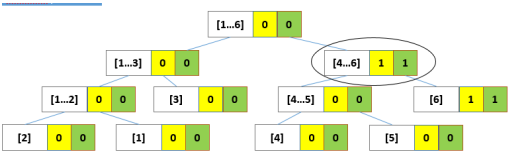
(c) Memeriksa rentang 4 hingga 6



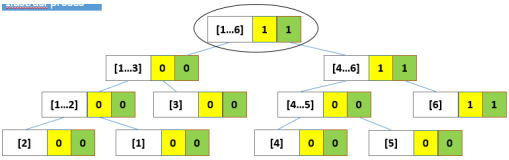
(d) Memperbarui nilai rentang 6

Gambar 2.25 Penyimpanan nilai untuk LIS dan LDS *node* 6 dan $update()$ pada *segment tree*

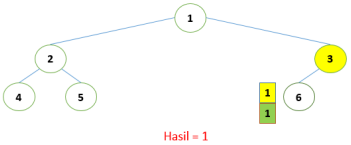
Dikarenakan *node* 6 merupakan *big child* maka perlu dilakukan $update()$ pada *segment tree*, *update* akan dilakukan pada *node* yang menyimpan nilai dari rentang 6. Proses fungsi $update()$ sendiri seperti yang telah dijelaskan pada subbab 2.4. Gambar 2.25b hingga Gambar 2.26b.



(a) Mengembalikan nilai ke atas



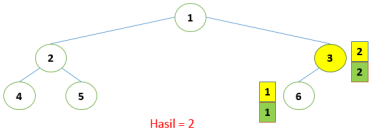
(b) Mengembalikan nilai ke *root*



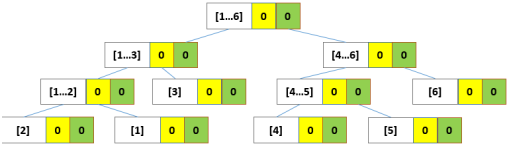
(c) Melanjutkan proses ke *node* 3

Gambar 2.26 Proses $update()$ pada *segment tree* dan rekursi ke *parent*

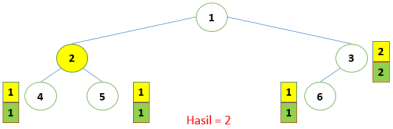
Proses dilanjutkan ke *parent* dari *node* 6, yaitu *node* 3, divisualkan pada Gambar 2.26c. Proses yang dilakukan sama yaitu menjalankan $query()$ untuk mencari nilai LIS dan LDS pada *segment tree*. Nilai yang didapatkan adalah 2 untuk LIS dan LDS, yang disimpan pada variabel sementara milik *node* 3 seperti pada Gambar 2.27a.



(a) Kondisi setelah memproses *node* 3



(b) Kondisi *segment tree* seperti semula

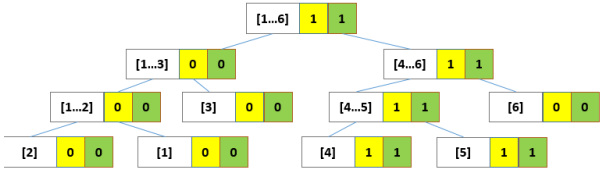


(c) Kondisi setelah selesai memproses *node* 4 dan 5

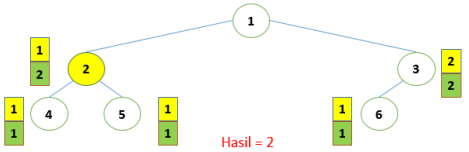
Gambar 2.27 Pengembalian kondisi *segment tree* seperti semula

Karena *node* 3 merupakan *small child*, maka sebelum kembali ke *parent*nya, kondisi *segment tree* perlu dikembalikan seperti semula, yaitu menjadikan seluruh nilai yang ada didalamnya menjadi 0. Seperti pada Gambar 2.27b.

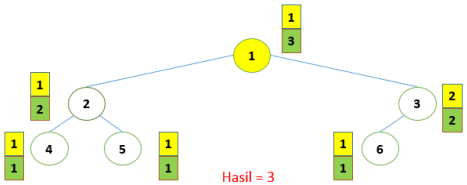
Proses dilanjutkan ke *big child* dari *root* yaitu *node* 2 seperti pada Gambar 2.27c. Tahapan yang dilakukan sama seperti pada *node* 3, dengan mencari *big child* dan *small child* kemudian menyimpan nilai pada variabel sementara untuk setiap *node*. Setelah selesai memproses seluruh *child* dilakukan *query* untuk LIS dan LDS pada *node* 2, dari kondisi *segment tree* seperti pada Gambar 2.28a, dan akan menyimpan hasilnya pada variabel sementara *node* 2, seperti pada Gambar 2.28b. Kondisi akhir dari keseluruhan proses ditunjukan oleh Gambar 2.28c dengan keluaran akhir yaitu 3.



(a) Kondisi *segment tree* pada *node* 2



(b) Kondisi setelah proses pada *node* 2



(c) Kondisi akhir dari keseluruhan proses

Gambar 2.28 Proses pencarian LIS dan LDS pada *big child* dari *root* dan pada *root*

BAB III
DESAIN

Pada bab ini akan dibahas tentang desain dan algoritma untuk menyelesaikan permasalahan pada Tugas Akhir ini.

Desain Penyelesaian Permasalahan *LIS and TREE by value*

Pada subbab ini akan dijelaskan desain penyelesaian permasalahan *LIS and TREE* dengan pendekatan *Segment Tree*.

Definisi Umum Sistem

Sistem akan menerima masukan berupa sebuah bilangan bulat T yang mewakili banyaknya kasus uji. Selanjutnya sistem menerima masukan sebuah bilangan bulat n yang mewakili jumlah *node* untuk kasus uji tersebut. Kemudian sistem akan menerima masukan berupa dua buah bilangan bulat a dan b yang merepresentasikan nilai dan hubungan antar *node* seperti yang telah dijelaskan pada subbab 2.6. Kemudian sistem akan menggambarkan masukkan tersebut sebagai sebuah *tree* dengan memanggil fungsi *init* setelah *tree* terbentuk akan melakukan pencarian LIS yang akan memenuhi persoalan tersebut, yang pada akhirnya akan menjadi keluaran akhir dari program. Pseudocode fungsi *main* ditunjukkan oleh Gambar 3.1.

Pada fungsi *main* terdapat variabel T yang merupakan jumlah kasus uji dan n yang merupakan jumlah *node*. Setiap akan menerima masukan baru, fungsi *main* akan mengosongkan terlebih dahulu isi daripada masukan yang sebelumnya, dilakukan dengan

```
Main()
1 input T
2 while T − −
3   input N
4   for i = 1 to N
5     Clear(inp[i])
6   for i = 1 to N-1
7     input a
8     input b
9     Push(inp[a],b)
10      Push(inp[a],b)
11   ans = 0
12   cnt = 0
13   init(1,-1)
14   Dfs(1,-1,0)
15   Output ans
```

Gambar 3.1 *Pseudocode* Fungsi Main

menggunakan fungsi *clear* (baris ke 5). Kemudian fungsi akan memasukkan *push* (baris ke 8 dan 9) nilai masukan yang baru. Lalu akan dipanggil fungsi *init* dan *dfs* yang akan dijelaskan pada subbab berikutnya.

Desain Algoritma

Sistem terdiri dari 4 fungsi utama, yaitu fungsi *init*, *dfs*, *query*, *update*. Pada subbab ini akan dijelaskan desain algoritma keempat fungsi tersebut.

Desain Fungsi *init*

Fungsi *init* digunakan untuk membentuk *tree* dari masukan pada fungsi *main*, ditunjukkan oleh Gambar 3.2. Variabel now merepresentasikan nilai *node* saat ini, dan $parent$

```
init(now,parent)
1 size[now]  = 1
2 in[now]    = ++cnt
3 num[cnt]   = now
4 for i = 0 to Sizeof(inp[now]-1)
5   if inp[now][i] = parent  continue
6   init(inp[now][i],now)
7   size[now]  = size[now] + size[inp[now][i]
8 out[now]   = cnt
```

Gambar 3.2 *Pseudocode* Fungsi init

merepresentasikan nilai *node* yang menjadi *parent* dari *node* saat ini. Pada fungsi ini dilakukan iterasi untuk mendapatkan total jumlah *child* dari keseluruhan *tree* untuk setiap *node* dan dilakukan *preorder numbering* yang diwakili oleh variabel $in[]$ dan $out[]$.

Desain Fungsi *Dfs*

Fungsi DFS merupakan fungsi utama untuk menemukan LIS dari *path* yang ada ditunjukkan oleh Gambar 3.3. Parameter variabel yang dibutuhkan adalah now sebagai representasi nilai *node* saat ini, $parent$ sebagai representasi nilai dari *parent* untuk *node* saat ini, dan $flag$ sebagai variabel penanda apakah *node* tersebut merupakan *bigchild* atau *smallchild*. Di awal fungsi (baris kode 1-5) dilakukan penentuan *node* mana yang menjadi *big child* atau *small child*, hal ini ditentukan dengan cara membandingkan ukuran *node* tersebut dengan *sibling*nya. Kemudian fungsi akan melakukan rekursi ke *node* yang menjadi *smallchild* (baris kode 6-9) dan dilanjutkan rekursi ke *node* yang menjadi *bigchild* (baris kode 10-11). Setelah itu dilakukan perulangan untuk setiap *child* dari *node* kecuali *bigchild* dalam perulangan ini dilakukan pencarian LIS dan LDS serta melakukan *update* pada *segment tree* untuk nilai LIS dan

LDS di *subtree* sebelumnya (baris kode 12-31). Lalu jika *node* yang telah diproses merupakan *smallchild* maka isi *segment tree* akan di kembalikan seperti semula, jika tidak isi *segment tree* akan dipertahankan (baris kode 32-36). Pada prosesnya fungsi *dfs* juga memanggil fungsi lain yaitu $query$ dan $update$.

Fungsi $query$ ditunjukkan oleh Gambar 3.4. Terdapat 6 variabel yang menjadi parameter untuk fungsi ini, idx merepresentasikan posisi *node* saat ini, l dan r merupakan representasi rentang index yang diwakili oleh *node* saat ini, x dan y merupakan representasi rentang index yang akan diambil nilainya, $flag$ merupakan penanda apakah harus mengambil nilai di *segment tree* yang berisikan LIS atau LDS. Fungsi ini akan melakukan rekursi ke *left child* kemudian dilanjutkan ke *right child* hingga pada akhirnya mendapatkan nilai maksimal dari kedua rentang tersebut.

Fungsi *update* yang ditunjukkan oleh Gambar 3.5. Terdapat 6 variabel yang menjadi parameter untuk fungsi ini, idx merepresentasikan posisi *node* saat ini, l dan r merupakan representasi rentang index yang diwakili oleh *node* saat ini, x merupakan representasi index yang ingin di*update*.

Desain Penyelesaian Permasalahan *LIS and TREE by reference*

Pada subbab ini akan dijelaskan desain penyelesaian permasalahan *LIS and TREE* dengan pendekatan *pointer*.

Definisi Umum Sistem

Sistem akan menerima masukan berupa sebuah bilangan bulat T yang mewakili banyaknya kasus uji. Selanjutnya sistem menerima masukan sebuah bilangan bulat n yang mewakili jumlah *node* untuk kasus uji tersebut. Kemudian sistem akan menerima masukan berupa dua buah bilangan bulat a dan b yang merepresentasikan

```
Dfs(now, parent, flag)
1 for i = 0 to Sizeof(inp[now]-1)
2   if temp = parent continue
3   if size[temp] > maks
4     maks = size[temp]
5     bigchild = temp
6 for i = 0 to Sizeof(inp[now]-1)
7   temp = inp[now][i]
8   if temp != parent and temp != bigchild
9     Dfs(temp,now,0)
10 if bigchild = 0
11   Dfs(big,now,1)
12 for i = 0 to Sizeof(inp[now]-1)
13   temp = inp[now][i]
14   if temp = parent and temp = bigchild
15     continue
16   foreach child in temp
17     if child < now
18       maks=query(0,1,N,now+1,N,1)
19       ans=max(ans,maks+1+dpUp[child])
20     else
21       maks=query(0,1,N,1,now-1,0)
22       ans=max(ans,maks+1+dpDec[child])
23   maks=query(0,1,N,1,child-1,0)
24   ans=max(ans,maks+1+dpDec[child])
25   maks=query(0,1,N,child+1,N,1)
26   ans=max(ans,maks+1+dpUp[child])
27   foreach child in temp
28     update(0,1,N,child,dpUp[child],dpDec[child])
29 dpUp[now]=query(0,1,N,1,now-1,0)+1
30 dpDec[now]=query(0,1,N,now+1,N,1)+1
31 ans=max(ans,max(dpUp[now],dpDec[now]))
32 if flag = 0
33   foreach child in now
34     update(0,1,N,child,0,0)
35 else
36   update(0,1,N,now,dpUp[now],dpDec[now])
```

Gambar 3.3 *Pseudocode* Fungsi Dfs

```
query(idx,l,r,x,y,flag)
1 if l > y or r < x
2     return 0
3 if x <= l and r <= y
4     if flag = 0
5         return inc[idx]
6     else
7         return dec[idx]
8 left = query((idx*2)+1,l,(l+r)/2,x,y,flag)
9 right = query((idx*2)+2,((l+r)/2)+1,r,x,y,flag)
10 return max(left,right)
```

Gambar 3.4 *Pseudocode* Fungsi query

```
update(idx,l,r,x,a,b)
1 if l = r
2     inc[idx] = a
3     dcr[idx] = b
4     return
5 if x <= (l+r)/2
6     update((idx*2)+1,l,(l+r)/2,x,a,b)
7 else
8     update((idx*2)+2,((l+r)/2)+1,r,x,a,b)
9 inc[idx] = max(inc[(idx*2)+1],inc[(idx*2)+2])
10 dcr[idx] = max(dcr[(idx*2)+1],dcr[(idx*2)+2])
```

Gambar 3.5 *Pseudocode* Fungsi update

```
Main()
1 input T
2 while T − −
3   input N
4   for i = 1 to N
5     Clear(inp[i])
6   for i = 1 to N-1
7     input a
8     input b
9     Push(inp[a],b)
10      Push(inp[a],b)
11    result = 0
12    init(1,-1)
13    Dfs(1,-1,&a,&b)
14    Output result
```

Gambar 3.6 *Pseudocode* Fungsi Main

nilai dan hubungan antar *node* seperti yang telah dijelaskan pada subbab 2.6. Kemudian sistem akan menggambarkan masukkan tersebut sebagai sebuah *tree* dengan memanggil fungsi *init* setelah *tree* terbentuk akan melakukan pencarian LIS yang akan memenuhi persoalan tersebut, yang pada akhirnya akan menjadi keluaran akhir dari program. Pseudocode fungsi *main* ditunjukkan oleh Gambar 3.6.

Pada fungsi *main* terdapat variabel T yang merupakan jumlah kasus uji dan n yang merupakan jumlah *node*. Setiap akan menerima masukan baru, fungsi *main* akan mengosongkan terlebih dahulu isi daripada masukan yang sebelumnya, dilakukan dengan menggunakan fungsi *clear* (baris ke 5). Kemudian fungsi akan memasukkan *push* (baris ke 8 dan 9) nilai masukan yang baru. Lalu akan dipanggil fungsi *init* dan *dfs* yang akan dijelaskan pada subbab berikutnya.

```
init(now,parent)
1 for i = 0 to Sizeof(inp[now]-1)
2   if inp[now][i] = parent  continue
3   init(inp[now][i],now)
4   size[now] = size[now] + size[inp[now][i]
```

Gambar 3.7 *Pseudocode* Fungsi init

Desain Algoritma

Sistem terdiri dari 3 fungsi utama, yaitu fungsi *init*, *dfs*, *addto*, *combine*. Pada subbab ini akan dijelaskan desain algoritma keempat fungsi tersebut.

Desain Fungsi init

Fungsi *init* digunakan untuk membentuk *tree* dari masukan pada fungsi *main*, ditunjukkan oleh Gambar 3.7. Variabel now merepresentasikan nilai *node* saat ini, dan $parent$ merepresentasikan nilai *node* yang menjadi *parent* dari *node* saat ini. Pada fungsi ini dilakukan iterasi untuk mendapatkan total jumlah *child* dari keseluruhan *tree* untuk setiap *node*.

Desain Fungsi *Dfs*

Fungsi DFS merupakan fungsi utama untuk menemukan LIS dari *path* yang ada ditunjukkan oleh Gambar 3.8 dan 3.9. Parameter variabel yang dibutuhkan adalah $index$ sebagai representasi nilai *node* saat ini, $parent$ sebagai representasi nilai dari *parent* untuk *node* saat ini, *vector of pointer* $*lis$ yang menyimpan nilai untuk LIS, dan *vector of pointer* $*lds$ yang menyimpan nilai untuk LDS. Pada awal fungsi (baris kode 1-5) dilakukan penentuan *node* mana yang menjadi *big child* atau *small child*, hal ini ditentukan dengan cara membandingkan ukuran *node* tersebut dengan *sibling*nya. Kemudian fungsi akan melakukan rekursi ke *node* yang menjadi *smallchild* (baris kode 8-14), dilanjutkan dengan pengecekan

apakah *node* tersebut hanya memiliki satu *child* (baris kode 15-20), dan dilanjutkan rekursi ke *node* yang menjadi *bigchild* (baris kode 21-29). Pada prosesnya fungsi *dfs* juga memanggil fungsi lain yaitu $addto$ dan $combine$.

Fungsi *addto* yang ditunjukkan oleh Gambar 3.10. Terdapat 2 parameter pada fungsi ini, yaitu *vector of int* $*vec$ yang merupakan pointer yang merujuk ke pada nilai LIS atau LDS saat ini dan val yang merupakan nilai dari *node* saat ini. Fungsi ini akan mencari nilai terkecil dari di dalam *vector* vec yang lebih besar dari val, jika tidak ada maka nilai val akan dimasukkan kedalam *vector*, namun jika ada maka nilai didalam *vector* akan diganti oleh val.

Fungsi *combine* yang ditunjukkan oleh Gambar 3.11. Terdapat 2 parameter pda fungsi ini, yaitu *vector of int* $*a$ dan *vector of int* $*b$ dimana keduanya merujuk pada *vector* LIS atau LDS saat ini. Fungsi ini akan membandingkan nilai di tiap index yang sama dari kedua *vector* yang ada dan mencari *vector* manakah yang lebih panjang.

```
Dfs(index, parent, *lis, *lds)
1 for i = 0 to Sizeof(inp[index]-1)
2   if temp = parent continue
3   if size[temp] > maks
4     maks = size[temp]
5     bigchild = temp
6 addto(lis, index)
7 addto(lds, -index)
8 if bigchild == false
9   uplis[index] = new vector
10   push(uplis[index], index)
11   uplds[index] = new vector
12   push(uplis[index], index)
13   result = max(result, sizeof(lis))
14   result = max(result, sizeof(lds))
15 else if(parent != -1 and sizeofinp[index] == 2)
16   dfs(big, index, lis, lds)
17   uplis[index] = uplis[big]
18   uplds[index] = uplds[big]
19   addto(uplis[index], index)
20   addto(uplds[index], -index)
21 else
22   a[] = *lis
23   b[] = *lds
24   dfs(big, index, &a, &b)
25   uplis[index] = uplis[bigchild]
26   uplds[index] = uplds[bigchild]
27   addto(uplis[index], index)
28   combine(lis, uplis[index])
29   combine(lis, uplds[index])
30   for i = 0 to Sizeof(inp[index]-1)
31     temp = inp[index][i]
32     if temp == parent or temp == bigchildcontinue
33     a = *lis
34     b = *lds
35     dfs(temp, index, &a, &b)
```

Gambar 3.8 *Pseudocode* Fungsi Dfs bagian 1.

```
36      addto(uplis[temp], index)
37      addto(uplds[temp], -index)
38      if i+1 < sizeofinp[index]
39          combine(lis, uplis[temp])
40          combine(lds, uplds[temp])
41      delete uplis[temp]
42      delete uplds[temp]
```

Gambar 3.9 *Pseudocode* Fungsi Dfs bagian 2.

```
addto(*vec, val)
1 p = lowerbound(vec->begin, vec->end, val)
2 if p == vec->end
3    push(vec,val)
4 else
5    *p = val
```

Gambar 3.10 *Pseudocode* Fungsi *addto*.

```
combine(*a, *b)
1 p = 0
2 while true
3    if p >= sizeof(b) break
4    else if p >= sizeof(a)
5       push(a,(*b)[p])
6    else if p < sizeof(b)
7       *a[p] = min((*a)[p], (*b)[p])
8    p++
```

Gambar 3.11 *Pseudocode* Fungsi *combine*.

Halaman ini sengaja dikosongkan

BAB IV
IMPLEMENTASI

Pada bab ini dijelaskan mengenai implementasi dari desain dan algoritma penyelesaian permasalahan klasik *LIS and TREE*.

Lingkungan Implementasi

Lingkungan implementasi dalam pembuatan Tugas Akhir ini meliputi perangkat keras dan perangkat lunak yang digunakan untuk penyelesaian permasalahan klasik *LIS and TREE* adalah sebagai berikut:

1. Perangkat Keras:
    - *Processor* AMD A8-6410 4 Cores 2.0 Ghz up to 2.4 GHz.
    - Memori 8 Gb.
2. Perangkat Lunak:
    - Sistem Operasi: Windows 8.1 64 bit.
    - IDE: Orwell Bloodshed Dev-C++ 5.11.
    - Compiler: g++ (TDM-GCC 4.9.2 64-bit).
    - Bahasa Pemrogramman: C++.

Implementasi Penyelesaian Permasalahan *LIS and TREE by value*

Pada subbab ini akan dijelaskan mengenai implementasi dari penyelesaian permasalahan *LIS and TREE* sesuai dengan desain pada subbab 3.1.

Penggunaan *Library*, Konstanta dan Variabel *Global*

Pada Subbab ini akan dibahas penggunaan *library*, *template*, konstanta dan variabel global yang digunakan dalam sistem. Pada Kode sumber 4.1 terdapat dua *library* yang digunakan yaitu: iostream dan vector. Didefinisikan konstanta N bernilai 100001 merupakan jumlah node maksimal. Pada Tabel 4.1 dan 4.2 dijelaskan mengenai variabel-variabel global yang akan digunakan dalam implementasi program.

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4  const int N = 100001;
```

Kode Sumber 4.1 Potongan kode *library* dan konstanta

Tabel 4.1 Daftar variabel global bagian 1.

No	Nama Variabel	Tipe	Penjelasan
1	ans	int	Digunakan untuk menyimpan jawaban soal
2	cnt	int	Digunakan sebagai penghitung jumlah *child*.
3	size	int []	Digunakan untuk menyimpan ukuran tiap *subtree*.
4	num	int []	Digunakan untuk menyimpan urutan dari *node* masukan.
5	in	int []	Digunakan untuk perulangan pada subtree.

Tabel 4.2 Daftar variabel global bagian 2.

No	Nama Variabel	Tipe	Penjelasan
6	out	int[]	Digunakan untuk perulangan pada subtree.
7	dpUp	int []	Digunakan untuk menyimpan nilai LDS pada suatu *node*.
8	dpDec	int []	Digunakan untuk menyimpan nilai LIS pada suatu *node*.
9	inc	int []	Digunakan sebagai representasi *segment tree* untuk nilai LIS.
10	dcr	int []	Digunakan sebagai representasi *segment tree* untuk nilai LDS.
11	inp	vector<int> []	Digunakan untuk menyimpan *disjoint set union* untuk tiap *node* masukan.

Implementasi Fungsi *Main*

Fungsi Main diimplementasikan sesuai dengan *pseudocode* pada bab sebelumnya.

```
1  int main()
2  {
3          int T, n;
4          cin>>T;
5          while(T--)
6          {
7                  cin>>n;
8                  for(int i=1;i<=n;i++)
9                  {
10                         inp[i].clear();
11                 }
12                 for(int i=1;i<=n-1;i++)
13                 {
14                         int a, b;
15                         cin>>a;
16                         cin>>b;
17                         inp[a].push_back(b);
18                         inp[b].push_back(a);
19                 }
20                 ans = 0;
21                 cnt = 0;
22                 init(1,-1);
23                 dfs(1,-1,0);
24                 cout<<ans<<endl;
25         }
26         return 0;
27 }
```

Kode Sumber 4.2 Potongan kode fungsi main

Fungsi main yang diimplementasikan seperti pada 4.2 digunakan untuk mengosongkan terlebih dahulu vector yang akan digunakan (baris kode 8 hingga 11) dan membaca masukan pada sistem (baris kode 12 hingga 19).

Implementasi Fungsi *Init*

Fungsi Init diimplementasikan pada Kode sumber 4.3 sesuai dengan *pseudocode* pada bab sebelumnya.

```
1  void init(int now, int par)
2  {
3          size[now] = 1;
4          in[now] = ++cnt;
5          num[cnt] = now;
6          for(int i=0;i<=(int)inp[now].size()-1;i++)
7          {
8                  if(inp[now][i] == par) continue;
9                  init(inp[now][i],now);
10                 size[now] += size[inp[now][i]];
11         }
12         out[now] = cnt;
13 }
```

Kode Sumber 4.3 Potongan kode fungsi init

Implementasi Fungsi Dfs

Fungsi Dfs diimplementasikan pada Kode sumber 4.4, 4.5, 4.6 sesuai dengan *pseudocode* pada bab sebelumnya.

```
1  void dfs(int now, int par, int flag)
2  {
3          int maks = -1, big = -1;
4          for(int i=0;i<=(int)inp[now].size()-1;i++)
5          {
6                  int temp = inp[now][i];
7                  if(temp == par) continue;
8                  if(size[temp] > maks)
9                  {
10                         maks = size[temp];
11                         big = temp;
12                 }
13         }
14         for(int i=0;i<=(int)inp[now].size()-1;i++)
15         {
16                 int temp = inp[now][i];
```

Kode Sumber 4.4 Potongan kode fungsi dfs bagian 1

```
17                  if(temp != par && temp != big)
18                  {
19                          dfs(temp,now,0);
20                  }
21          }
22          if(big > 0)
23          {
24                  dfs(big,now,1);
25          }
26          for(int i=0;i<=(int)inp[now].size()-1;i++)
27          {
28          int temp = inp[now][i];
29          if(temp == par || temp == big)
30          continue;
31          for(int i=in[temp];i<=out[temp];i++)
32          {
33                  int val = num[i], maks;
34
35                  if(val < now)
36                  {
37                  maks = query(0,1,n,now+1,n,1);
38                  ans = max(ans,maks+1+dpUp[val]);
39                  }
40                  else
41                  {
42                  maks = query(0,1,n,1,now-1,0);
43                  ans = max(ans,maks+1+dpDec[val]);
44                  }
45
46          maks = query(0,1,n,1,val-1,0);
47          ans = max(ans,maks+dpDec[val]);
48
49          maks = query(0,1,n,val+1,n,1);
50          ans = max(ans,maks+dpUp[val]);
51          }
```

Kode Sumber 4.5 Potongan kode fungsi dfs bagian 2

```
52          for(int i=in[temp];i<=out[temp];i++)
53          {
54          int val = num[i];
55          update(0,1,n,val,dpUp[val],dpDec[val]);
56          }
57          }
58          dpUp[now] = query(0,1,n,1,now-1,0) + 1;
59          dpDec[now] = query(0,1,n,now+1,n,1) + 1;
60
61          ans = max(ans, max(dpUp[now], dpDec[now]));
62
63          if(!flag)
64          {
65          for(int i=in[now]+1;i<=out[now];i++)
66          {
67                  update(0,1,n,num[i],0,0);
68          }
69          }
70          else
71          {
72          update(0,1,n,now,dpUp[now],dpDec[now]);
73          }
74  }
```

Kode Sumber 4.6 Potongan kode fungsi dfs bagian 3

Implementasi Fungsi Query

Fungsi Dfs diimplementasikan pada Kode sumber 4.7 dan 4.8 sesuai dengan *pseudocode* pada bab sebelumnya.

```
1  int query(int idx, int l, int r, int x, int y, int flag)
2  {
3          if (l>y || r<x)
4          {
5                  return 0;
6          }
```

Kode Sumber 4.7 Potongan kode fungsi query bagian 1

```
 7          if (x<=l && r<=y)
 8          {
 9                  if (!flag)
10                  {
11                          return inc[idx];
12                  }
13                  else
14                  {
15                          return dcr[idx];
16                  }
17          }
18          int mid = (r+l)>>1, lc = (idx<<1)+1
19          int rc = (idx<<1)+2;
20          int a = max(query(lc,l,mid,x,y,flag),
21          query(rc,mid+1,r,x,y,flag));
22          return a;
23  }
```

Kode Sumber 4.8 Potongan kode fungsi query bagian 2

Implementasi Fungsi Update

Fungsi update diimplementasikan pada Kode sumber 4.7 dan 4.8 sesuai dengan *pseudocode* pada bab sebelumnya.

```
 1  void update(int idx, int l, int r, int x, int a, int b)
 2  {
 3          if (l==r)
 4          {
 5                  inc[idx] = a; dcr[idx] = b;
 6                  return;
 7          }
 8          int mid = (l+r)>>1, lc = (idx<<1)+1
 9          int rc = (idx<<1)+2;
10          if (x <= mid)
11          {
12                  update(lc, l, mid, x, a,b);
13          }
```

Kode Sumber 4.9 Potongan kode fungsi update bagian 1

```
14          else
15          {
16                  update(rc, mid+1, r, x, a,b);
17          }
18          inc[idx] = max(inc[lc], inc[rc]);
19          dcr[idx] = max(dcr[lc], dcr[rc]);
20  }
```

Kode Sumber 4.10 Potongan kode fungsi update bagian 2

Implementasi Penyelesaian Permasalahan *LIS and TREE by reference*

Pada subbab ini akan dijelaskan mengenai implementasi dari penyelesaian permasalahan *LIS and TREE* sesuai dengan desain pada subbab 3.2.

Penggunaan *Library*, Konstanta dan Variabel *Global*

Pada Subbab ini akan dibahas penggunaan *library*, *template*, konstanta dan variabel global yang digunakan dalam sistem. Pada Kode sumber 4.11 terdapat dua *library* yang digunakan yaitu: iostream dan vector. Didefinisikan konstanta N bernilai 100001 merupakan jumlah node maksimal. Pada Tabel 4.3 dijelaskan mengenai variabel-variabel global yang akan digunakan dalam implementasi program.

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4  const int N = 100001;
```

Kode Sumber 4.11 Potongan kode *library* dan konstanta

Tabel 4.3 Daftar variabel global.

No	Nama Variabel	Tipe	Penjelasan
1	size	int []	Digunakan untuk menyimpan ukuran tiap *subtree*.
2	result	int	Digunakan untuk menyimpan hasil akhir LIS.
3	inp	int []	Digunakan untuk menyimpan *disjoint set union* untuk tiap *node* masukan.
4	uplis	vector*<int> []	Digunakan untuk menyimpan hasil LIS sementara.
5	uplds	vector*<int> []	Digunakan untuk menyimpan hasil LDS sementara.

Implementasi Fungsi *Main*

Fungsi Main diimplementasikan sesuai dengan *pseudocode* pada bab sebelumnya ditunjukkan pada Kode sumber 4.12 dan 4.13.

```
1  int main()
2  {
3       int T, n,
4       cin>>T;
5       while(T--)
6           {
7                   cin>>n;
```

Kode Sumber 4.12 Potongan kode fungsi main bagian 1

```
8           for(int i=1;i<=n;i++)
9                   {
10                  inp[i].clear();
11          }
12          for(int i=1;i<=n-1;i++)
13                  {
14              int a,b;
15              cin>>a;
16              cin>>b;
17              inp[a].push_back(b);
18              inp[b].push_back(a);
19          }
20          init(1, -1);
21          ans = 0;
22          vector<int> a,b;
23          dfs(1, -1, &a, &b);
24          ans = max(ans, (int)uplis[1]->size());
25          ans = max(ans, (int)uplds[1]->size());
26          cout<<ans<<endl;
27      }
28      return 0;
29  }
```

Kode Sumber 4.13 Potongan kode fungsi main bagian 2

Implementasi Fungsi *Init*

Fungsi Init diimplementasikan pada Kode sumber 4.14 sesuai dengan *pseudocode* pada bab sebelumnya.

```
1   void init(int index, int parent)
2   {
3           size[index] = 1;
4       for(int i=0;i<=(int)inp[index].size()-1;i++)
5           {
6           if(inp[index][i] == parent)continue;
7           init(inp[index][i], index);
8           size[index] += size[inp[index][i]];
9       }
10  }
```

Kode Sumber 4.14 Potongan kode fungsi init

Implementasi Fungsi Dfs

Fungsi Dfs diimplementasikan pada Kode sumber 4.15, 4.16, 4.17 sesuai dengan *pseudocode* pada bab sebelumnya.

```
1  void dfs(int index, int parent, vector<int>* lis,
2      vector<int>* lds){
3
4      int maks = -1, big = -1;
5
6          for(int i=0;i<=(int)inp[index].size()-1;i++)
7          {
8                  int temp = inp[index][i];
9          if(temp == parent)continue;
10
11                 if(size[temp] > maks)
12                 {
13             maks = size[temp];
14             big = temp;
15         }
16     }
17     addto(lis, index);
18     addto(lds, -index);
19
20     if(big == -1)
21         {
22         uplis[index] = new vector<int>();
23         uplis[index]->push_back(index);
24
25         uplds[index] = new vector<int>();
26         uplds[index]->push_back(-index);
27
28         ans = max(ans, (int)lis->size());
29         ans = max(ans, (int)lds->size());
30     }
```

Kode Sumber 4.15 Potongan kode fungsi dfs bagian 1

```
31          else if(parent != -1 && inp[index].size()
32          == 2)
33          {
34          dfs(big, index, lis, lds);
35
36          uplis[index] = uplis[big];
37          uplds[index] = uplds[big];
38
39                  addto(uplis[index], index);
40                  addto(uplds[index], -index);
41      }
42          else
43          {
44          vector<int> a = *lis;
45          vector<int> b = *lds;
46
47                  dfs(big, index, &a, &b);
48
49          uplis[index] = uplis[big];
50                  uplds[index] = uplds[big];
51
52          addto(uplis[index], index);
53          addto(uplds[index], -index);
54
55          combine(lis, uplis[index]);
56          combine(lds, uplds[index]);
57
58          for(int i=0;i<=(int)inp[index].size()-1;i++)
59                  {
60              int temp = inp[index][i];
61              if(temp == parent || temp == big)
62                  continue;
63
64              a = *lis;
65              b = *lds;
66
67              dfs(temp, index, &a, &b);
```

Kode Sumber 4.16 Potongan kode fungsi dfs bagian 2

```
68
69              addto(uplis[temp], index);
70              addto(uplds[temp], -index);
71
72              combine(uplis[index], uplis[temp]);
73              combine(uplds[index], uplds[temp]);
74
75                          if(i+1 < inp[index].size())
76                          {
77                  combine(lis, uplis[temp]);
78                  combine(lds, uplds[temp]);
79              }
80
81              delete uplis[temp];
82              delete uplds[temp];
83          }
84      }
85  }
```

Kode Sumber 4.17 Potongan kode fungsi dfs bagian 3

Implementasi Fungsi *Addto*

Fungsi Addto diimplementasikan pada Kode sumber 4.18, sesuai dengan *pseudocode* pada bab sebelumnya.

```
 1  void addto(vector<int>* vec, int val){
 2      vector<int>::iterator p = lower_bound(vec->begin(),
 3          vec->end(), val);
 4      if(p == vec->end())
 5      {
 6                  vec->push_back(val);
 7          }
 8      else
 9          {
10          *p = val;
11          }
12  }
```

Kode Sumber 4.18 Potongan kode fungsi addto

Implementasi Fungsi *Combine*

Fungsi Combine diimplementasikan pada Kode sumber 4.19, sesuai dengan *pseudocode* pada bab sebelumnya.

```
1  void combine(vector<int>* a,vector<int> *b)
2  {
3          int p = 0;
4          while(1)
5          {
6          if(p >= b->size()) break;
7          else if(p >= a->size())
8                  {
9                          a->push_back((*b)[p]);
10                 }
11         else if(p < b->size())
12                 {
13                         (*a)[p] = min((*a)[p],
14                                 (*b)[p]);
15                 }
16         p++;
17     }
```

Kode Sumber 4.19 Potongan kode fungsi combine

Data Generator

Kode sumber 4.20, 4.21, dan 4.22 untuk membuat data yang akan digunakan pada uji coba.

```
1  #include<bits/stdc++.h>
2
3  using namespace std;
4
5  vector<int> a, b, temp_a, temp_b, full_num;
6  int x, y, temp;
```

Kode Sumber 4.20 Potongan kode data generator

```
 7
 8  int main()
 9  {
10          int testcase,num;
11          cin>>testcase;
12          cin>>num;
13          srand(time(NULL));
14          for (int j=1;j<=testcase;j++)
15          {
16          full_num.clear();
17          a.clear();
18          b.clear();
19          for(int i=1;i<=num;i++)
20          {
21                  full_num.push_back(i);
22          }
23          cout<<full_num.size()<<endl;
24          //system("pause");
25          int flag = 0;
26          x = rand() % num;
27          temp = x;
28          x = full_num.at(x);
29          a.push_back(x);
30          full_num.erase(full_num.begin()+temp);
31          y = rand() % num;
32          temp = y;
33          y = full_num.at(y);
34          a.push_back(y);
35          b.push_back(y);
36          full_num.erase(full_num.begin()+temp);
```

Kode Sumber 4.21 Potongan kode data generator bagian 2

```
37          cout<<x<<"␣"<<y<<endl;
38          for(int i=1;i<num-1;i++)
39          {
40          x = rand() % a.size();
41          x = a.at(x);
42          y = rand() % full_num.size();
43          temp = y;
44          y = full_num.at(y);
45          full_num.erase(full_num.begin()+temp);
46          a.push_back(y);
47          b.push_back(y);
48          cout<<x<<"␣"<<y<<endl;
49          }
50          }
51          return 0;
```

Kode Sumber 4.22 Potongan kode data generator bagian 3

Program akan memasukan nilai 1 hingga num pada $array$, terlihat pada baris kode 19 hingga 20 pada kode sumber. Kemudian akan membangkitkan bilangan secara acak dari $array$ tersebut, yang akan merepresentasikan a dan b sesuai permintaan soal. Agar bilangan yang sama tidak muncul kembali maka bilangan tersebut akan dihilangkan dari $array$ terlihat pada baris kode 26 hingga 27. Kemudian untuk nilai dari variabel a berikutnya akan diambil dari nilai yang sudah dihilangkan, terlihat pada baris kode 38 hingga 50. Sehingga semua nilai dipastikan keluar dan unik, seperti permintaan dari soal.

Halaman ini sengaja dikosongkan

BAB V
UJI COBA

Pada bab ini dijelaskan tentang uji coba dan evaluasi dari implementasi yang telah dilakukan pada Tugas Akhir ini.

Lingkungan Uji Coba

Perangkat keras yang digunakan adalah *Processor* AMD A8-6410 4 Cores 2.0 Ghz up to 2.4 GHz dengan *random access memory*(RAM) sebesar 8 Gb. Dengan sistem operasi Windows 8.1 64 bit. Bahasa yang digunakan adalah bahasa C++ dengan bantuan *integrated development environment*(IDE) Orwell Bloodshed Dev-C++ 5.11 serta compiler g++ (TDM-GCC 4.9.2 64-bit).

Skenario Uji Coba

Pada subbab ini akan dijelaskan skenario uji coba yang dilakukan.

Uji Coba Kebenaran

Uji Coba Kebenaran Penyelesaian Permasalahan *LIS and TREE by value*

Uji coba kebenaran dilakukan dengan mengirimkan kode ke situs penilaian daring SPOJ sebanyak 20 kali pengiriman, ditunjukkan pada Gambar 5.1. Dari hasil uji yang dilakukan dapat dilihat bahwa kode sumber yang dikirimkan mendapat keluaran *Accepted* dengan waktu tercepat program adalah 1.81 detik dan memori yang dibutuhkan konstan pada besaran 41 MB.

Gambar 5.1 Hasil uji coba permasalahan *LIS and TREE* pada situs penilaian daring SPOJ

Uji Coba Kebenaran Penyelesaian Permasalahan *LIS and TREE by reference*

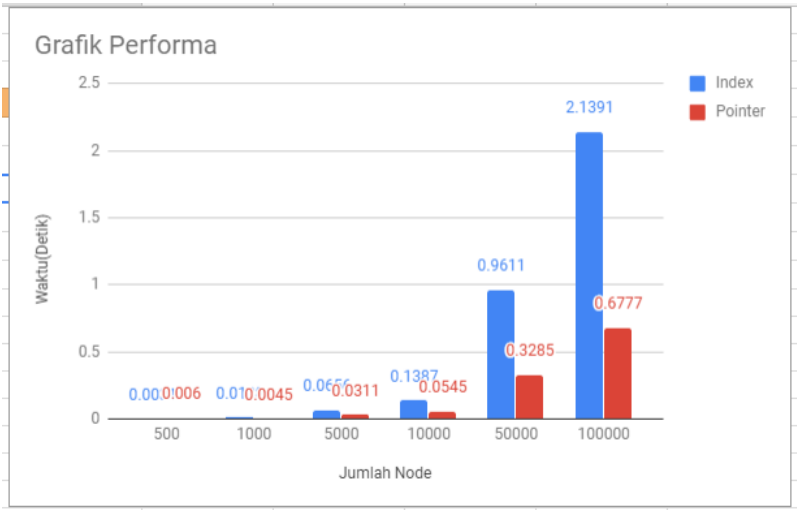
Uji coba kebenaran dilakukan dengan mengirimkan kode ke situs penilaian daring SPOJ sebanyak 20 kali pengiriman, ditunjukkan pada Gambar 5.2. Dari hasil uji yang dilakukan dapat dilihat bahwa kode sumber yang dikirimkan mendapat keluaran *Accepted* dengan waktu tercepat program adalah 0.66 detik dan memori yang dibutuhkan rata-rata sebesar 41 MB.

Gambar 5.2 Hasil uji coba permasalahan *LIS and TREE* pada situs penilaian daring SPOJ

Uji Coba Kinerja

Uji Coba Kinerja Pengaruh Jumlah Node Permasalahan *LIS and TREE*

Pada uji coba yang ditunjukkan Gambar 5.3 terlihat performa program dengan input jumlah kasus uji satu, dan jumlah *node* yang bertambah hingga 100000.

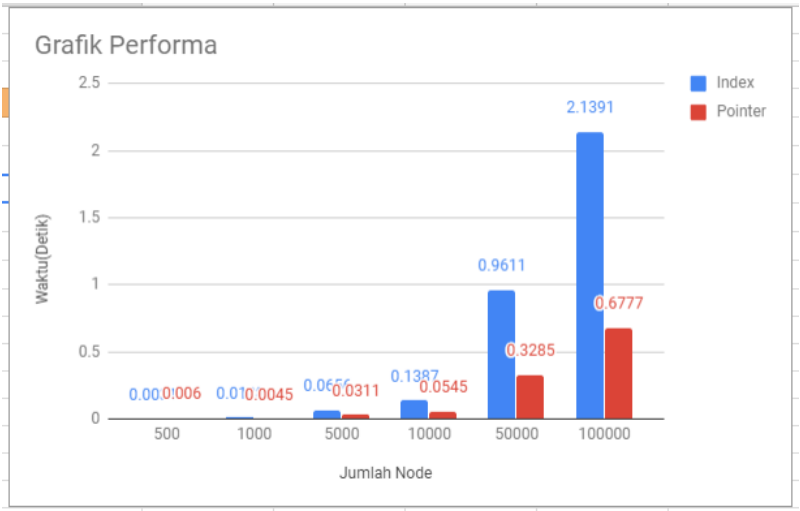


Gambar 5.3 Hasil uji coba program menggunakan data dari generator

Pada gambar grafik berwarna merah menunjukkan performa dari implementasi menggunakan pointer dan warna biru menunjukkan performa dari implementasi menggunakan index. Terlihat perbedaan yang cukup besar ketika masukan mencapai 100000 *node*, hal ini dikarenakan banyaknya bagian *segment tree* yang kosong yang harus dicek pada saat menggunakan implementasi index.

Uji Coba Kinerja Pengaruh Jumlah Kasus Uji Permasalahan *LIS and TREE by reference*

Pada uji coba yang ditunjukkan Gambar 5.4 terlihat performa program dengan input jumlah kasus uji yang bertambah hingga 1000, dan jumlah *node* yang konstan yaitu 100000.



Gambar 5.4 Hasil uji coba program menggunakan data dari generator

Pada gambar grafik berwarna merah menunjukkan performa dari implementasi menggunakan pointer dan warna biru menunjukkan performa dari implementasi menggunakan index. Waktu disini bertambah seiring dengan banyaknya kasus uji secara konstan.

BAB VI
KESIMPULAN DAN SARAN

Pada bab ini dijelaskan mengenai kesimpulan dari hasil uji coba yang telah dilakukan.

Kesimpulan

Dari hasil uji coba yang dilakukan terhadap implementasi penyelesaian permasalahan *LIS and TREE* dapat diambil kesimpulan sebagai berikut:

1. Implementasi algoritma *Disjoint Set Union on Tree* dengan struktur data *Segment Tree* dapat menyelesaikan permasalahan *LIS and TREE* dengan benar.
2. Implementasi algoritma *DSU on TREE* pada struktur data *Segment Tree* dapat menghasilkan kecepatan proses yang berbeda dengan menggunakkan pendekatan yang berbeda yaitu (*by value & by reference*).
3. Kompleksitas waktu yang dibutuhkan untuk seluruh sistem adalah $O(n \log^2 n)$, dengan n merupakan banyaknya *node* yang ada pada *tree* tersebut.

Halaman ini sengaja dikosongkan

DAFTAR PUSTAKA

[1] Bartek, ”LIS and TREE” [Online]. Available: http://www.spoj.com/problems/LISTREE/ . [Accessed 17-May-2018].

[2] Geeksforgeeks, ”Longest Increasing Subsequence” [Online]. Avaiable: http://www.geeksforgeeks.org/longest-monotically-increasing-subsequence-size-n-log-n [Accesed 17-May-2018].

[3] **Understanding Segment Trees** - Understanding segment tree · CodingHigway [Online]. Avaiable: http://codinghighway.com/2014/09/13/understanding-segment-trees/ [Accessed 17-May-2018].

[4] **Segment Tree | Set 1 (Sum of given range)** - Segment Tree | Set 1 (Sum of given range) - GeeksforGeeks [Online]. Avaiable: http://www.geeksforgeeks.org/segment-trees-set-1-sum-of-given-range/ [Accessed 17-May-2018].

Halaman ini sengaja dikosongkan

LAMPIRAN A
HASIL UJI COBA KEBENARAN PADA SITUS SPOJ

ID		DATE	PROBLEM	RESULT	TIME	MEM	LANG
21793114	☐	2018-06-06 22:54:40	LIS and tree	accepted edit ideone it	0.68	42M	CPP14
21793113	☐	2018-06-06 22:54:33	LIS and tree	accepted edit ideone it	0.69	41M	CPP14
21793112	☐	2018-06-06 22:54:28	LIS and tree	accepted edit ideone it	0.67	41M	CPP14
21793109	☐	2018-06-06 22:53:50	LIS and tree	accepted edit ideone it	0.66	42M	CPP14
21793106	☐	2018-06-06 22:53:45	LIS and tree	accepted edit ideone it	0.70	41M	CPP14
21793103	☐	2018-06-06 22:53:05	LIS and tree	accepted edit ideone it	0.66	42M	CPP14
21793101	☐	2018-06-06 22:52:59	LIS and tree	accepted edit ideone it	0.66	41M	CPP14
21793093	☐	2018-06-06 22:51:17	LIS and tree	accepted edit ideone it	0.69	41M	CPP14
21793092	☐	2018-06-06 22:51:10	LIS and tree	accepted edit ideone it	0.70	42M	CPP14
21793091	☐	2018-06-06 22:51:03	LIS and tree	accepted edit ideone it	0.66	42M	CPP14
21793086	☐	2018-06-06 22:50:04	LIS and tree	accepted edit ideone it	0.73	41M	CPP14
21793083	☐	2018-06-06 22:49:55	LIS and tree	accepted edit ideone it	0.67	42M	CPP14
21793082	☐	2018-06-06 22:49:48	LIS and tree	accepted edit ideone it	0.67	41M	CPP14
21793081	☐	2018-06-06 22:49:41	LIS and tree	accepted edit ideone it	0.66	42M	CPP14
21793080	☐	2018-06-06 22:49:31	LIS and tree	accepted edit ideone it	0.68	41M	CPP14
21793079	☐	2018-06-06 22:49:04	LIS and tree	accepted edit ideone it	0.67	42M	CPP14
21793078	☐	2018-06-06 22:48:26	LIS and tree	accepted edit ideone it	0.70	42M	CPP14
21793077	☐	2018-06-06 22:48:18	LIS and tree	accepted edit ideone it	0.66	42M	CPP14
21793076	☐	2018-06-06 22:48:10	LIS and tree	accepted edit ideone it	0.70	42M	CPP14
21793000	☐	2018-06-06 22:22:38	LIS and tree	accepted edit ideone it	0.68	41M	CPP14

Gambar A.1 Hasil Pengujian Implementasi Kode by Reference Sebanyak 20 Kali pada Situs Penilaian Daring SPOJ

ID		DATE	PROBLEM	RESULT	TIME	MEM	LANG
21792977	☐	2018-06-06 22:15:59	LIS and tree	accepted edit ideone it	1.82	41M	CPP14
21792975	☐	2018-06-06 22:15:45	LIS and tree	accepted edit ideone it	1.87	41M	CPP14
21792967	☐	2018-06-06 22:14:14	LIS and tree	accepted edit ideone it	1.87	41M	CPP14
21792961	☐	2018-06-06 22:11:47	LIS and tree	accepted edit ideone it	1.85	41M	CPP14
21792958	☐	2018-06-06 22:11:18	LIS and tree	accepted edit ideone it	1.86	41M	CPP14
21792953	☐	2018-06-06 22:10:43	LIS and tree	accepted edit ideone it	1.86	41M	CPP14
21792949	☐	2018-06-06 22:10:13	LIS and tree	accepted edit ideone it	1.87	41M	CPP14
21792948	☐	2018-06-06 22:09:43	LIS and tree	accepted edit ideone it	1.87	41M	CPP14
21792946	☐	2018-06-06 22:09:09	LIS and tree	accepted edit ideone it	1.81	41M	CPP14
21792945	☐	2018-06-06 22:08:35	LIS and tree	accepted edit ideone it	1.82	41M	CPP14
21792941	☐	2018-06-06 22:08:02	LIS and tree	accepted edit ideone it	1.84	41M	CPP14
21792940	☐	2018-06-06 22:07:33	LIS and tree	accepted edit ideone it	1.84	41M	CPP14
21792938	☐	2018-06-06 22:06:48	LIS and tree	accepted edit ideone it	1.87	41M	CPP14
21792936	☐	2018-06-06 22:06:13	LIS and tree	accepted edit ideone it	1.85	41M	CPP14
21792932	☐	2018-06-06 22:05:38	LIS and tree	accepted edit ideone it	1.81	41M	CPP14
21792928	☐	2018-06-06 22:04:44	LIS and tree	accepted edit ideone it	1.89	41M	CPP14
21792923	☐	2018-06-06 22:04:09	LIS and tree	accepted edit ideone it	1.87	41M	CPP14
21792919	☐	2018-06-06 22:03:32	LIS and tree	accepted edit ideone it	1.82	41M	CPP14
21792917	☐	2018-06-06 22:03:00	LIS and tree	accepted edit ideone it	1.84	41M	CPP14
21792915	☐	2018-06-06 22:02:26	LIS and tree	accepted edit ideone it	1.87	41M	CPP14

Gambar A.2 Hasil Pengujian Implementasi Kode by Index Sebanyak 20 Kali pada Situs Penilaian Daring SPOJ

BIODATA PENULIS

Dimas Hirda Pratama, lahir di Surabaya tanggal 3 Agustus 1996. Penulis merupakan anak pertama dari 2 bersaudara. Penulis telah menempuh pendidikan formal TK Muhajirin Surabaya, SDI Luqman Al-Hakim Surabaya (2002-2008), SMP Negeri 6 Surabaya (2008-2011) dan SMA Negeri 5 Surabaya (2011-2014). Penulis melanjutkan studi kuliah program sarjana di Jurusan Teknik Informatika ITS.

Selama kuliah di Teknik Informatika ITS, penulis pernah menjadi asisten dosen dan praktikum untuk mata kuliah Dasar Pemrogramman (2015). Selama menempuh perkuliahan penulis juga aktif di kegiatan organisasi dan kepanitiaan diantaranya menjadi staff Departemen Minat Bakat HMTC ITS (2015),staff ahli Departemen Minat Bakat HMTC ITS (2016), BPH III Hubungan Masyarakat Schematics 2015, BPH I Hubungan Masyarakat Schematics 2016, panitia Pemusatan Latihan Nasional 2 TOKI 2017 di ITS, dan panitia Pemusatan Latihan Nasional 2 TOKI 2018 di ITS. Penulis dapat dihubungi melalui surel di
dimas0308@gmail.com.