TUGAS AKHIR – KI141502

Penerapan Teknik Heavy-Light Decomposition dan Struktur Data Fenwick Tree pada Rancangan Algoritma: Studi Kasus SPOJ Klasik 28079 Maximum Child Sum

PRASETYO NUGROHADI
NRP. 5114100070

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

Dosen Pembimbing II
Dwi Sunaryono, S.Kom., M.Kom.

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

TUGAS AKHIR – KI141502

Penerapan Teknik Heavy-Light Decomposition dan Struktur Data Fenwick Tree pada Rancangan Algoritma: Studi Kasus SPOJ Klasik 28079 Maximum Child Sum

PRASETYO NUGROHADI
5114100070

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

Dosen Pembimbing II
Dwi Sunaryono, S.Kom., M.Kom.

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

[*Halaman ini sengaja dikosongkan*]

FINAL PROJECT - KI141502

Implementation of Heavy-Light Decomposition Technique and Fenwick Tree Data Structure on Algorithm Design of Clasical SPOJ Problem 28079 Maximum Child Sum

PRASETYO NUGROHADI
5114100070

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

Supervisor II
Dwi Sunaryono, S.Kom., M.Kom.

DEPARTMENT OF INFORMATICS
FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY
Sepuluh Nopember Institute of Technology
Surabaya, 2018

[*Halaman ini sengaja dikosongkan*]

LEMBAR PENGESAHAN

Penerapan Teknik Heavy-Light Decomposition dan Struktur Data Fenwick Tree pada Rancangan Algoritma: Studi Kasus SPOJ Klasik 28079 Maximum Child Sum

TUGAS AKHIR

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

Oleh:
Prasetyo Nugrohadi
NRP: 5114 100 070

Disetujui oleh Dosen Pembimbing Tugas Akhir:

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

Dwi Sunaryono, S.Kom., M.Kom.
NIP. 197404031999031002
....
(Pembimbing 2)

KEMENTERIAN RISET, TEKNOLOGI DAN PENDIDIKAN TINGGI
INSTITUT TEKNOLOGI SEPULUH NOPEMBER
FAKULTAS TEKNOLOGI INFORMASI DAN KOMUNIKASI
DEPARTEMEN
TEKNIK INFORMATIKA

SURABAYA
JANUARI 2018

[*Halaman ini sengaja dikosongkan*]

PENERAPAN TEKNIK HEAVY-LIGHT DECOMPOSITION DAN STRUKTUR DATA FENWICK TREE PADA RANCANGAN ALGORITMA: STUDI KASUS SPOJ KLASIK 28079 MAXIMUM CHILD SUM

Nama Mahasiswa	**: Prasetyo Nugrohadi**
NRP	**: 5114 100 070**
Departemen	**: Informatika FTIK - ITS**
Dosen Pembimbing 1	**: Rully Soelaiman, S.Kom., M.Kom.**
Dosen Pembimbing 2	**: Dwi Sunaryono, S.Kom., M.Kom.**

Abstrak

Permasalahan SPOJ Klasik 28079 berjudul "Maximum Child Sum" mengangkat pencarian penyelesaian cara mencari child dari suatu node yang memiliki struktur tree dengan jumlah maksimal di antara child lainnya.

Dengan struktur tree yang dinamis dan query yang dilakukan harus diselesaikan dengan batasan waktu yang ditentukan, maka perlu diterapkan teknik khusus dalam perancangan algoritma penyelesaian permasalahan tersebut. Beberapa hal yang harus diperhatikan adalah mengenai cara mengatasi operasi yang dilakukan pada dua perintah yang berbeda, yaitu query dan insert. Operasi query akan mengeluarkan index node yang merupakan child dengan jumlah subtree maksimal dari suatu node, sedangkan operasi insert akan menambahkan node baru dengan nilai dan parent tertentu.

*Tugas akhir ini telah berhasil menerapkan struktur data Fenwick Tree dan teknik Heavy-Light Decomposition untuk menyelesaikan permasalahan tersebut dengan kompleksitas O(Q*sqrt(N)*log(N)), dimana Q adalah banyaknya operasi dan N adalah banyaknya vertex.*

Kata kunci: Acyclic Graph, Fenwick Tree, Heavy-Light Decomposition, Tree

[*Halaman ini sengaja dikosongkan*]

IMPLEMENTATION OF HEAVY-LIGHT DECOMPOSITION TECHNIQUE AND FENWICK TREE DATA STRUCTURE ON ALGORITHM DESIGN OFSPOJ CLASSICAL PROBLEM 28079 MAXIMUM CHILD SUM

Nama Mahasiswa	**: Prasetyo Nugrohadi**
NRP	**: 5114 100 070**
Departemen	**: Informatika FTIK - ITS**
Dosen Pembimbing 1	**: Rully Soelaiman, S.Kom., M.Kom.**
Dosen Pembimbing 2	**: Dwi Sunaryono, S.Kom., M.Kom.**

Abstract

Classical SPOJ problems 28079 entitled "Maximum Child Sum" raised the issue of settlement by finding child of a node that has a tree structure with the maximum number among the other child.

With a dynamic tree structure and queries that must be done with the specified time limit, it is necessary to apply a special technique in designing the algorithm to solve the problem. Some things to note are on how to solve the operations performed on two different commands, namely query and insert. The query operation will output the index node which is the child with the maximum number of subtrees of a node, while the insert operation will add a new node with certain values and parent.

*This final project has successfully applied Fenwick Tree Data Structure and Heavy-Light Decomposition technique to solve the problem with the complexity of O (Q * sqrt (N) * log (N)), where Q is the number of operation(s) and N is the number of vertice(s).*

Keywords: Acyclic Graph, Fenwick Tree, Heavy-Light Decomposition, Tree

[*Halaman ini sengaja dikosongkan*]

KATA PENGANTAR

Puji syukur penulis panjatkan kepada Tuhan Yesus Kristus atas pimpinan, penyertaan, dan karunia-Nya sehingga penulis dapat menyelesaikan Tugas Akhir yang berjudul :

Penerapan Teknik Heavy-Light Decomposition dan Struktur Data Fenwick Tree pada Rancangan Algoritma: Studi Kasus SPOJ Klasik 28079 Maximum Child Sum

Penelitian Tugas Akhir ini dilakukan untuk memenuhi salah satu syarat meraih gelar Sarjana di Departemen Informatika, Fakultas Teknologi Informasi dan Komunikasi, 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:

1. Bapak Rully Soelaiman, S.Kom., M.Kom. selaku dosen pembimbing yang telah banyak meluangkan waktu untuk memberikan ilmu, nasihat, motivasi, bimbingan dan didikan kepada penulis dengan sabar selama perkuliahan maupun pengerjaan Tugas Akhir ini.
2. Bapak Dwi Sunaryono, S.Kom., M.Kom. selaku dosen pembimbing yang telah memberikan masukan dan bimbingan kepada penulis selama pengerjaan Tugas Akhir ini.
3. Bapak (Dr. Hari Basuki Notobroto, dr.M.Kes.), ibu (Dr. Dwi Winarni, M.Si.), dan kakak (Prima Widiani, S.Hum.) penulis yang selalu memberikan dukungan, perhatian, dan kasih sayang bagi penulis yang menjadi semangat selama perkuliahan maupun pengerjaan Tugas Akhir ini.
4. Seluruh tenaga pengajar dan karyawan Departemen Informatika ITS yang telah memberikan tenaga dan

waktunya demi kelancaran proses belajar mengajar penulis selama perkuliahan.

5. Teman-teman dari RMP (Andy, Rey, Arya, Anggit, Jarjit, Kurkur, Duhus) yang memberikan dukungan dan semangat dan senantiasa menemani penulis selama pengerjaan Tugas Akhir ini.
6. Anindita Larasati yang telah menemani penulis dan membantu penulis dalam pengerjaan Tugas Akhir ini.
7. Teman-teman angkatan 2014 Departemen Informatika ITS yang telah menemani perjuangan penulis selama kurang lebih 4 tahun masa perkuliahan.
8. 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, Januari 2018

Prasetyo Nugrohadi

DAFTAR ISI

LEMBAR PENGESAHAN........... Error! Bookmark not defined.
Abstrak .... **vii**
Abstract.. **ix**
KATA PENGANTAR .... **xi**
DAFTAR ISI.. **xiii**
DAFTAR GAMBAR .... **xvi**
DAFTAR TABEL.. **xviii**
DAFTAR KODE SUMBER... **xix**
BAB I PENDAHULUAN **1**
1.1 Latar Belakang 1
1.2 Rumusan Permasalahan 1
1.3 Batasan Permasalahan 2
1.4 Tujuan Pembuatan Tugas Akhir 2
1.5 Manfaat Tugas Akhir 2
1.6 Metodologi 2
1.6.1 Penyusunan Proposal Tugas Akhir 3
1.6.2 Studi Literatur 3
1.6.3 Implementasi Perangkat Lunak 3
1.6.4 Pengujian dan Evaluasi 3
1.6.5 Penyusunan Buku Tugas Akhir 3
1.7 Sistematika Penulisan 4
BAB II DASAR TEORI **7**
2.1 Deskripsi Umum Permasalahan 7
2.2 Deskripsi Umum Struktur Data Dasar yang Digunakan 9
2.2.1 *Vector* 9
2.2.2 *Tree* 9
2.3 *Offline Query* 10
2.4 *Preorder Numbering* 12
2.5 *Heavy-Light Decomposition* 13
2.6 *Fenwick Tree* 16
2.7 *Sistem Penilaian Daring* 19
BAB III ANALISIS DAN PERANCANGAN **23**

3.1 Analisis dan Observasi Permasalahan 23
3.1.1 Penyelesaian Menggunakan Solusi Naive........................ 23
3.1.2 Cara Penyimpanan Operasi... 25
3.1.3 Cara Penyimpanan Elemen Tree...................................... 25
3.1.4 Pemrosesan Berdasarkan Tipe Operasi 27
3.2 Perancangan Program... 27
3.2.1 Deskripsi Umum Program.. 27
3.2.2 Desain Fungsi DFS 28
3.2.3 Desain Fungsi HLD.. 30
3.2.4 Desain Fungsi UPDATE 31
3.2.5 Desain Fungsi TOTAL... 31
3.2.6 Desain Fungsi JUMLAH.. 32
3.2.7 Desain Fungsi UPPAR 32
3.2.8 Desain Fungsi HITUNG... 33
BAB IV IMPLEMENTASI... 35
4.1 Lingkungan Implementasi 35
4.2 Pendefinisian *Preprocessor Directives*............................ 35
4.3 Implementasi Fungsi Main 35
4.4 Implementasi Variabel Global... 37
4.5 Implementasi Fungsi DFS 37
4.6 Implementasi fungsi HLD 38
4.7 Implementasi fungsi UPDATE.. 38
4.8 Implementasi Fungsi TOTAL 39
4.9 Implementasi Fungsi JUMLAH 39
4.10 Implementasi Fungsi UPPAR 40
4.11 Implementasi Fungsi HITUNG 40
BAB V UJI COBA DAN EVALUASI 41
5.1 Lingkungan Uji Coba.. 41
5.2 Uji Coba Menggunakan Solusi Naive 41
5.3 Uji Coba Menggunakan Offline Query, HLD, dan Fenwick Tree 42
5.3.1 Menerima Masukan 43
5.3.2 Membentuk Tree... 44
5.3.3 Melakukan Dekomposisi 45

5.3.4 Eksekusi Operasi 46
BAB VI KESIMPULAN 51
6.1 Kesimpulan 51
6.2 Saran 51
A DAFTAR PUSTAKA 53
B LAMPIRAN 55
BIODATA PENULIS 57

DAFTAR GAMBAR

Gambar 2.1 Algoritma Preorder 12
Gambar 2.2 *Pseudocode Preorder* 13
Gambar 2.3 Visualisasi *Preorder* 13
Gambar 2.4 Perubahan *Index* pada *Tree* 14
Gambar 2.5 *Tree* sebelum Dekomposisi 15
Gambar 2.6 Hasil Dekomposisi 15
Gambar 2.7 Struktur *Fenwick Tree* 17
Gambar 2.8 *Pseudocode* Operasi *Update* pada *Fenwick Tree* 17
Gambar 2.9 *Pseudocode* Operasi *Read* pada *Fenwick Tree* 17
Gambar 3.1 *Pseudocode* Fungsi Main pada Solusi *Naive* 23
Gambar 3.2 Pseudocode Fungsi findMaksSum pada Solusi *Naive* 24
Gambar 3.3 *Pseudocode* Fungsi sums pada Solusi *Naive* 24
Gambar 3.4 Diagram Alur Program 27
Gambar 3.5 *Pseudocode* Fungsi Main 28
Gambar 3.6 *Pseudocode* Fungsi DFS 29
Gambar 3.7 Visualisasi Fungsi DFS 29
Gambar 3.8 *Pseudocode* Fungsi HLD 30
Gambar 3.9 *Pseudocode* Fungsi UPDATE 31
Gambar 3.10 *Pseudocode* Fungsi TOTAL 32
Gambar 3.11 *Pseudocode* Fungsi JUMLAH 32
Gambar 3.12 *Pseudocode* Fungsi UPPAR 33
Gambar 3.13 *Pseudocode* Fungsi HITUNG 34
Gambar 5.1 Hasil Uji Kebenaran Solusi *Naive* permasalahan *MAXCHILDSUM* pada Situs SPOJ 42
Gambar 5.2 Nilai pada Variabel a, b, c, dan pos setelah Menerima Masukan pada Kasus Uji Coba 44
Gambar 5.3 Struktur *Tree* yang Dibentuk pada Kasus Uji Coba 44
Gambar 5.4 Nilai pada Varibel in, out, dan sum setelah Membentuk *Tree* pada Kasus Uji Coba 45
Gambar 5.5 Hasil Dekomposisi pada Kasus Uji Coba................ 45
Gambar 5.6 Nilai pada Variabel hc dan sc setelah Melakukan Dekomposisi pada Kasus Uji Coba 46

Gambar 5.7 Hasil Uji Kebenaran Solusi Gabungan *Offline Query*, *HLD*, dan *BIT* pada Permasalahan *MAXCHILDSUM* pada Situs SPOJ..49
Gambar 5.8 Peringkat Waktu Eksekusi Program *Maximum Child Sum* pada SPOJ 50

DAFTAR TABEL

Tabel 2.1 Contoh Masukan pada Daring 7
Tabel 2.2 Visualisasi Pemrosesan Operasi 8
Tabel 2.3 Istilah-Istilah Umum dalam *Tree* 9
Tabel 2.4 Visualisasi Perbedaan *Offline Query* dan *Online Query* 11
Tabel 2.5 Visualisasi Operasi pada *Fenwick Tree* 18
Tabel 3.1 Variabel Global 25
Tabel 3.2 Proses Fungsi DFS 28
Tabel 3.3 Proses Fungsi HLD...30
Tabel 3.4 Proses Fungsi UPDATE 31
Tabel 3.5 Proses Fungsi TOTAL..31
Tabel 3.6 Proses Fungsi JUMLAH...32
Tabel 3.7 Proses Fungsi UPPAR..33
Tabel 3.8 Proses Fungsi HITUNG...34
Tabel 4.1 Spesifikasi Lingkungan Implementasi........................35
Tabel 5.1 Spesifikasi Lingkungan Uji Coba 41
Tabel 5.2 Visualisasi Eksekusi Operasi pada Kasus Uji Coba....46

DAFTAR KODE SUMBER

Kode Sumber 4.1 Implementasi *Preprocessor Directive*............35
Kode Sumber 4.2 Implementasi Fungsi Main 36
Kode Sumber 4.3 Implementasi Variabel Global 37
Kode Sumber 4.4 Implementasi Fungsi DFS 37
Kode Sumber 4.5 Implementasi Fungsi HLD..............................38
Kode Sumber 4.6 Implementasi Fungsi UPDATE 39
Kode Sumber 4.7 Implementasi Fungsi TOTAL.........................39
Kode Sumber 4.8 Implementasi Fungsi JUMLAH......................39
Kode Sumber 4.9 Implementasi Fungsi UPPAR 40
Kode Sumber 4.10 Implementasi Fungsi HITUNG....................40

[*Halaman ini sengaja dikosongkan*]

BAB I
PENDAHULUAN

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

1.1 Latar Belakang

Permasalahan SPOJ Klasik 28079 berjudul "*Maximum Child Sum*" mengangkat pencarian solusi untuk mencari *child* dari suatu *node* yang memiliki *subtree* dengan jumlah nilai maksimal diantara *child* lainnya.

Dengan struktur *tree* yang dinamis dan *query* yang dilakukan harus diselesaikan dengan batasan waktu yang ditentukan, maka perlu diterapkan teknik khusus dalam perancangan algoritma penyelesaian permasalahan tersebut.

Hasil tugas akhir ini diharapkan dapat menentukan algoritma dan struktur data yang tepat sehingga dapat menyelesaikan permasalahan di atas dengan optimal dan diharapkan dapat memberikan kontribusi pada perkembangan ilmu pengetahuan dan teknologi informasi.

1.2 Rumusan Permasalahan

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

1. Bagaimana penerapan teknik *Heavy-Light Decomposition* dan Struktur Data *Fenwick Tree* dalam menyelesaikan permasalahan SPOJ Klasik 28079 *Maximum Child Sum*?
2. Bagaimana hasil dari kinerja teknik *Heavy-Light Decomposition* dan Struktur Data *Fenwick Tree* dalam menyelesaikan permasalahan SPOJ Klasik 28079 *Maximum Child Sum*?

1.3 Batasan Permasalahan

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

1. Implementasi algoritma menggunakan bahasa pemrograman C++.
2. Batas maksimum jumlah kueri (Q) adalah 200.000.
3. Batas maksimum nilai pada node (Y) adalah 10^9.
4. Batas maksimum waktu kompilasi adalah 2 detik.
5. Batas maksimum penggunaan memori adalah 1.536MB
6. *Dataset* yang digunakan untuk pengujian kriteria algoritma adalah *dataset* pada permasalahan SPOJ Klasik 28079 *Maximum Child Sum*

1.4 Tujuan Pembuatan Tugas Akhir

Tujuan dari pembuatan Tugas Akhir ini adalah sebagai berikut:

1. Menerapkan teknik *Heavy-Light Decomposition* dan Struktur Data *Fenwick Tree* dalam menyelesaikan permasalahan SPOJ Klasik 28079 *Maximum Child Sum*.
2. Mengevaluasi hasil dan kinerja teknik *Heavy-Light Decomposition* dan Struktur Data *Fenwick Tree* dalam menyelesaikan permasalahan SPOJ Klasik 28079 *Maximum Child Sum*.

1.5 Manfaat Tugas Akhir

Tugas akhir ini diharapkan dapat membantu menyelesaikan permasalahan komputasi perhitungan bobot *edge* maksimum pada sejumlah permutasi dengan optimal dan efisien.

1.6 Metodologi

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

1.6.1 Penyusunan Proposal Tugas Akhir

Penyusunan proposal merupakan tahap awal untuk memulai pengerjaan Tugas Akhir. Pada tahap ini, penulis mengajukan gagasan untuk menyelesaikan permasalahan mencari *subtree* dengan jumlah maksimum pada studi kasus SPOJ Klasik 28079 *Maximum Child Sum*.

1.6.2 Studi Literatur

Tahap kedua adalah mencari informasi dan studi literatur yang relevan untuk dijadikan referensi dalam melakukan pengerjaan Tugas Akhir. Informasi dan studi literatur ini didapatkan dari buku, *internet*, dan materi-materi kuliah yang berhubungan dengan metode yang akan digunakan.

1.6.3 Implementasi Perangkat Lunak

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

1.6.4 Pengujian dan Evaluasi

Tahap berikutnya adalah melakukan uji coba menggunakan *dataset* pada *Online Judge* SPOJ Klasik 28079 *Maximum Child Sum* untuk mengetahui hasil dan performa dari metode dan struktur data yang dibangun. Evaluasi dan perbaikan juga akan dilakukan pada *Online Judge* hingga perangkat lunak yang diuji mengeluarkan hasil dan performa yang sesuai dengan data uji pada *Online Judge* SPOJ.

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 ini merupakan laporan 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 antara lain:

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 juga terdapat di dalamnya.

Bab II Dasar Teori

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

Bab III Analisis dan Perancangan Sistem

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 sebelumnya. 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 menyampaikan kesimpulan dari hasil uji coba yang

dilakukan dan saran untuk pengembangan perangkat lunak ke depannya.

[*Halaman ini sengaja dikosongkan*]

BAB II
DASAR TEORI

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

2.1 Deskripsi Umum Permasalahan

Permasalahan yang diangkat pada Tugas Akhir ini adalah mencari *child* dari suatu *node* yang memiliki *subtree* dengan jumlah maksimal diantara *child* lainnya.

Permasalahan utama pada persoalan ini adalah mencari sebuah teknik dan struktur data yang tepat untuk menyelesaikan operasi *insert/update* dan *query* dengan batasan waktu yang ditentukan pada struktur *tree* yang dinamis.

Operasi *insert*¸ dilambangkan dengan angka 1, berarti program menambahkan sebuah *node* baru pada *tree* dengan nilai X dan *parent* Y. Operasi *query*, dilambangkan dengan angka 2, berarti program menampilkan *child* dari *node* X yang memiliki bobot (jumlah nilai dalam *subtree*) maksimum diantara *child* lainnya.

Contoh masukan yang diberikan pada daring terdapat pada Tabel 2.1.

Tabel 2.1 Contoh Masukan pada Daring

7
1 1 3
2 1
2 2
1 2 5
2 1
1 1 4
2 1

Visualisasi dari operasi yang dilakukan pada permasalahan ini dapat dilihat pada Tabel 2.2.

Tabel 2.2 Visualisasi Pemrosesan Operasi

Operasi	Tree
Inisialisasi	
1 1 3	
2 1	Output: 3
2 2	Output: 0
1 2 5	
2 1	Output: 8
1 1 4	
2 1	Output: 8

2.2 Deskripsi Umum Struktur Data Dasar yang Digunakan

2.2.1 *Vector*

Vector adalah sebuah struktur data yang dapat menyimpan sebuah susunan objek secara berurutan. *Vector* sama seperti sebuah tipe data *array*, hanya saja ukuran *vector* dapat berubah.

Sama seperti *array*, *vector* menggunakan lokasi penyimpanan bersebelahan untuk elemennya, hal ini menunjukkan bahwa elemen *vector* juga dapat diakses menggunakan *offset* pada *pointer* reguler terhadap elemen-elemennya, dan memiliki efisiensi yang sama dengan *array*. Tidak seperti *array*, ukuran *vector* bisa berubah secara dinamis, dengan penyimpanan yang ditangani secara otomatis oleh *container*.

2.2.2 *Tree*

Tree merupakan salah satu bentuk struktur data tidak linear yang menggambarkan hubungan yang bersifat hirarkis (hubungan *one to many*) antara elemen-elemen. *Tree* bisa didefinisikan sebagai kumpulan simpul/*node* dengan satu elemen khusus yang disebut *root*, sedangkan *node* lainnya terbagi menjadi himpunan-himpunan yang saling tak berhubungan satu sama lainnya (disebut *subtree*). Istilah-istilah umum dalam *tree* dapat dilihat di Tabel 2.3

Tabel 2.3 Istilah-Istilah Umum dalam *Tree*

Istilah	Pengertian
Prodecessor	*node* yang berada diatas *node* tertentu
Successor	*node* yang berada di bawah *node* tertentu
Ancestor	seluruh *node* yang terletak sebelum *node* tertentu dan terletak pada jalur yang sama
Descendant	seluruh *node* yang terletak sesudah *node* tertentu dan terletak pada jalur yang sama
Parent	*predecessor* satu level di atas suatu *node*

Child	*successor* satu level di bawah suatu *node*
Sibling	*node-node* yang memiliki *parent* yang sama dengan suatu *node*
Subtree	bagian dari *tree* yang berupa suatu *node* beserta *descendant*-nya dan memiliki semua karakteristik dari *tree* tersebut
Size	banyaknya *node* dalam suatu *tree*
Height	banyaknya tingkatan/*level* dalam suatu *tree*
Root	satu-satunya *node* khusus dalam *tree* yang tak punya *predecessor*
Leaf	node-node dalam *tree* yang tidak memiliki *successor*
Degree	banyaknya *child* yang dimiliki suatu *node*
Path	Urutan *nodes* dan *edges* yang menghubungkan sebuah *node* dengan *descendant*

2.3 *Offline Query*

Ada dua cara dalam memproses *query* pada sebuah permasalahan, yaitu *online solution* dan *offline solution*. *Online solution* berarti solusi dari *problem* tersebut dapat memproses setiap *query* sebelum memproses *query* selanjutnya, sedangkan *offline solution* berarti solusi dari *problem* tersebut akan membaca semua *query* terlebih dahulu lalu memprosesnya secara berurutan.

Offline query diperlukan dalam penyelesaian masalah ini dalam efisiensi pembentukan *tree* dan mempercepat eksekusi pada tiap operasi yang akan dilakukan. Dengan menyimpan seluruh operasi yang akan dilakukan, maka program dapat mengetahui ukuran dan struktur dari *tree* yang akan dibangun, nilai dari seluruh *node* yang ada pada *tree*, dan seluruh operasi beserta jenisnya.

Perbandingan penggunaan *online query* dan *offline query* dengan operasi yang sama, yaitu [1 1 3, 1 2 5, 11 4, 1 3 5, 1 1 6, 1 6 7] dapat dilihat pada Tabel 2.4.

Tabel 2.4 Visualisasi Perbedaan *Offline Query* dan *Online Query*

Operasi	*Offline Query*	*Online Query*
Inisialisasi		
1 1 3		
1 2 5		
1 1 4		
1 3 5		

Operasi	*Offline Query*	*Online Query*
1 1 6		
1 6 7		

Penggunaan *offline query* dalam permasalahan ini adalah implikasi dari penggunaan *heavy-light decomposition* dan *fenwick tree*, karena perlu mengetahui ukuran dan struktur *tree* sebelum melakukan dekomposisi serta memiliki kompleksitas lebih rendah jika dibandingkan dengan *online query*.

2.4 *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* sedalam mungkin pada setiap *child* sebelum lanjut ke *sibling* selanjutnya.

Algoritma dari *preorder* dapat dilihat pada Gambar 2.1.

1	Cek apakah *node* saat ini kosong
2	Menampilkan nilai/data dari *node* saat ini
3	*Traverse* ke *subtree* sebelah kiri dengan memanggil fungsi *preorder* secara rekursif
4	*Traverse* ke *subtree* sebelah kanan dengan memanggil fungsi *preorder* secara rekursif

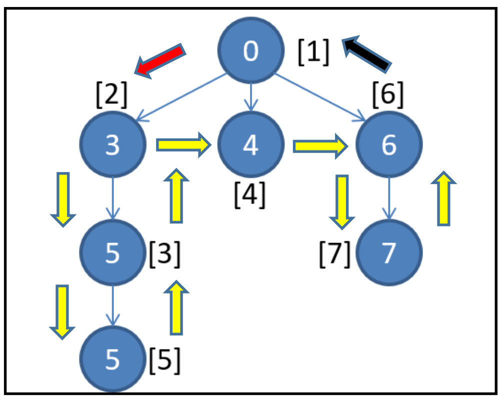
Gambar 2.1 Algoritma *Preorder*

Pseudocode dari preorder dapat dilihat pada Gambar 2.2.

1	preorder (node)
2	if node == null
3	return
4	visit(node)
5	preorder(node.left)
6	preorder(node.right)

Gambar 2.2 *Pseudocode Preorder*

Visualisasi dari *preorder* dapat dilihat pada Gambar 2.3



Gambar 2.3 Visualisasi *Preorder*

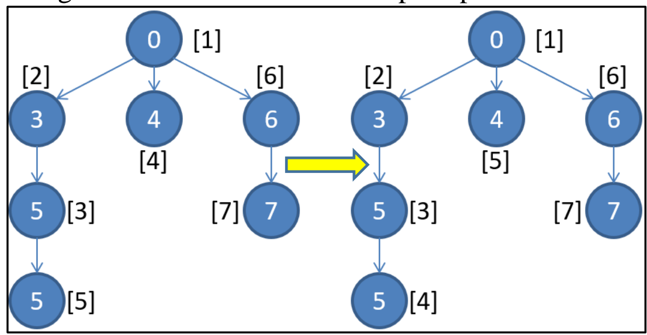
Dari visualisasi tersebut, *tracing* pada *preorder* adalah 1-2-3-5-4-6-7.

2.5 *Heavy-Light Decomposition*

Heavy-Light Decomposition adalah teknik dekomposisi sebuah *graph* (*tree*) menjadi sebuah *disjoint chains* (tidak ada 2 *chain* yang memiliki *node* yang sama). Disebut *heavy-light* karena dekomposisi dilakukan berdasarkan kriteria yang kita

tentukan untuk membedakan apakah *chain* tersebut merupakan *node heavy* atau *light*.

Dari data *tree* yang telah dibentuk, maka perlu dilakukan pengubahan indeks menggunakan *DFS* sesuai dengan bobot *node* tersebut, apakah *heavy* atau *light*, sesuai dengan dekomposisi yang akan dilakukan, serta untuk membentuk struktur *tree* yang sesuai dengan struktur *Fenwick Tree* seperti pada Gambar 2.4.



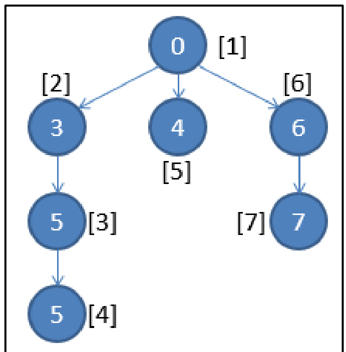
Gambar 2.4 Perubahan *Index* pada *Tree*

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

Setelah membentuk *tree*, selanjutnya dicatat *node* mana yang merupakan *heavy vertex* atau *light vertex*. Klasifikasi *heavy* dan *light vertex* pada permasalahan ini adalah sebagai berikut:

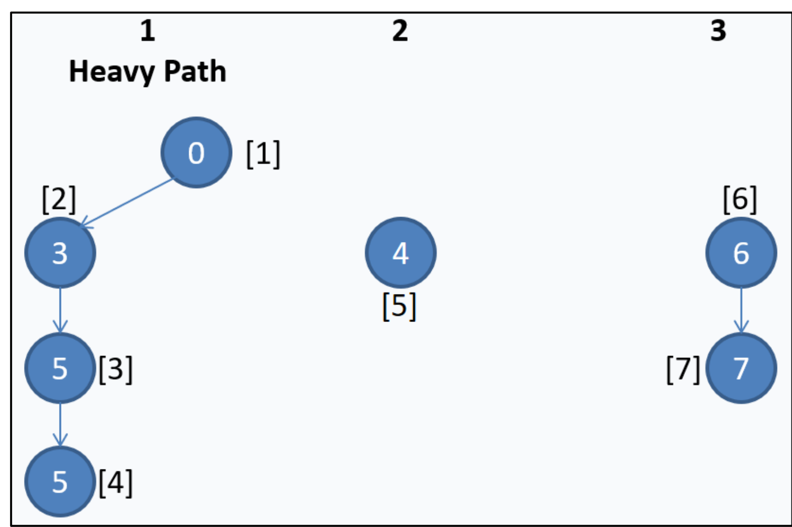
Light Vertex : *Node* dengan jumlah *child* ≤ sqrt(N)

Heavy Vertex : *Node* dengan jumlah *child* > sqrt(N)



Gambar 2.5 *Tree* sebelum Dekomposisi

Pada Gambar 2.5, tidak ditemukan sebuah *node* yang memenuhi syarat sebagai *heavy vertex*¸ maka dapat diasumsikan *heavy vertex* adalah *node* yang merupakan *child* sebelah kiri dari *parent*. Hasil dekomposisi dapat dilihat pada Gambar 2.6.



Gambar 2.6 Hasil Dekomposisi

Heavy path pada *tree* tersebut adalah 1-2-3-4, sedangkan dua *path* lainnya merupakan *light path*, yaitu 4 dan 6-7. Teknik ini digunakan untuk mengoptimalkan proses operasi *query*. Operasi *query* dibagi menjadi dua berdasarkan *light vertex* dan *heavy vertex*. Untuk *light*, maka program hanya perlu melakukan *loop* sebanyak N *child* dari *node* tersebut, dengan kompleksitas O(log N) karena bisa dipastikan bahwa hanya ada maksimal log N *child* pada *light vertex*. Untuk *heavy*, tinggal keluarkan nilai maksimum yang sudah dicatat ketika *update*.

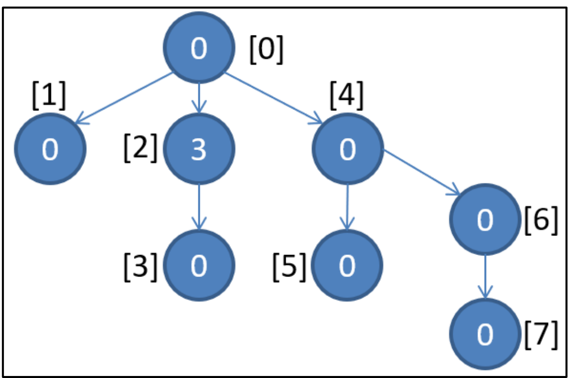
2.6 *Fenwick Tree*

Fenwick Tree atau yang juga biasa disebut *Binary Indexed Tree (BIT)* adalah sebuah struktur data yang dapat memperbarui elemen dan menghitung jumlah *prefix* secara efisien dalam tabel yang berisi angka-angka. Struktur data ini diusulkan oleh Peter Fenwick pada 1994 dengan tujuan meningkatkan efisiensi algoritma kompresi aritmatika.

Bila menggunakan algoritma dengan *flat arrays*, perhitungan jumlah *prefix* dan pembaruan elemen membutuhkan waktu linear (O(n)), sedangkan *Fenwick Tree* mampu menyelesaikan kedua operasi tersebut dalam waktu logaritmik (O(log n)).

Penggunaan struktur data *Fenwick Tree* pada permasalahan ini adalah untuk *tracing* pada *tree*, yaitu mencari *ancestor/parent* dari tiap *node* dengan *preorder numbering* pada *tree* yang telah dibangun. *Preorder Numbering* mempermudah proses *read* dan *update* pada *tree*.

Dengan data masukan pada *tree* adalah [1 1 3, 1 2 5, 11 4, 1 3 5, 1 1 6, 1 6 7], maka struktur *BIT* yang terbentuk adalah seperti pada Gambar 2.7.



Gambar 2.7 Struktur *Fenwick Tree*

Operasi dalam *Fenwick Tree* ada dua macam, yaitu *read/tracing* dan *insert/update*. Operasi tersebut dapat dilakukan dengan melakukan *preorder numbering*. Algoritma untuk operasi *read* dapat dilihat pada Gambar 2.8 dan operasi *update* pada Gambar 2.9.

1	*update current_node*
2	*comp = 2's complement of current_index*
3	*and = comp & current_index*
4	*new_index = current_index + and*

Gambar 2.8 *Pseudocode* Operasi *Update* pada *Fenwick Tree*

1	*read current_node*
2	*comp = 2's complement of current_index*
3	*and = comp & current_index*
4	*new_index = current_index - and*

Gambar 2.9 *Pseudocode* Operasi *Read* pada *Fenwick Tree*

Dengan data masukan seperti yang telah disebutkan sebelumnya, maka dapat dilakukan operasi pada *Fenwick Tree* seperti pada Tabel 2.5.

Tabel 2.5 Visualisasi Operasi pada *Fenwick Tree*

Operasi	Tree	Keterangan
1 1 3		*Update node 2* *Comp 2 => **14*** *14 & 2 => **2*** *2 + 2 => **4***
		Update node 4 *Comp 4 => **12*** *12 & 4 => **4*** *4 + 4 => **8 (X)***
1 2 5		*Update node 3* *Comp 3 => **13*** *13 & 3 => **1*** *3 + 1 => **4***
		Update node 4 *Comp 4 => **12*** *12 & 4 => **4*** *4 + 4 => **8 (X)***
1 1 4		*Update node 5* *Comp 5 => **11*** *11 & 5 => **1*** *5 + 1 => **6***

Operasi	Tree	Keterangan
		Update node 6 *Comp 6 => **10*** *10 & 6 => **2*** *6 + 2 => **8 (x)***
1 3 5		*Update node 4* *Comp 4 => **12*** *12 & 4 => **4*** *4 + 4 => **8 (X)***
1 1 6		*Update node 6* *Comp 6 => **10*** *10 & 6 => **2*** *6 + 2 => **8 (X)***
1 6 7		*Update node 7* *Comp 7 => **9*** *9 & 7 => **1*** *7 + 1 => **8 (X)***

Penggunaan struktur data *Fenwick Tree* pada permasalahan ini adalah untuk efisiensi *tracing* pada *tree* dengan kompleksitas (O(log n)).

2.7 Sistem Penilaian Daring

Sistem penilaian daring atau *online judge system* adalah sistem yang bekerja untuk melakukan pengujian program. Pada sistem penilaian daring terdapat berbagai soal yang masing-

masing terdiri atas deskripsi soal, berkas data masukan, dan berkas data keluaran. Deskripsi soal berisi deskripsi tugas yang harus dilakukan oleh program milik pengguna, batasan waktu eksekusi program, batasan memori program, serta batasan dan format masukan yang diterima dan format keluaran yang dihasilkan. Berkas data masukan berisi data masukan yang akan diberikan pada program, sesuai dengan batasan pada deskripsi soal. Berkas data keluaran berisi data keluaran yang akan dicek kesamaannya dengan hasil keluaran program pengguna.

Sistem penilaian daring menerima program pengguna berupa berkas kode sumber yang diunggah oleh pengguna melalui antarmuka sistem, kemudian melakukan kompilasi dan menjalankan program yang dihasilkan. Program hasil kompilasi dari kode pengguna dijalankan dan diuji dengan berkas data masukan soal. Kemudian sistem melakukan pengecekan hasil keluaran program dengan berkas data keluaran soal. Berkas data masukan dan data keluaran soal merupakan hasil studi pembuat soal dan telah diuji kebenarannya oleh tim ahli, sehingga kesesuaian antara deskripsi soal dengan berkas data masukan dan data keluaran dapat dipertanggungjawabkan.

Pada tahap pengujian program, pengguna tidak dapat mengetahui isi berkas data masukan dan berkas data keluaran. Pengguna hanya mendapat umpan balik dari sistem berupa total waktu eksekusi program dan frase yang menunjukkan kebenaran program pengguna. Program pengguna dianggap benar jika dan hanya jika hasil keluaran program pengguna sama persis dengan berkas data keluaran soal. Jika program pengguna benar, maka sistem akan memberikan umpan balik berupa frase "Accepted". Sedangkan jika program pengguna tidak benar, maka sistem akan memberi umpan balik berupa salah satu dari frase-frase berikut:

1. "Wrong Answer", merupakan umpan balik yang didapat jika hasil keluaran program pengguna tidak sama persis dengan berkas data keluaran soal.

2. "Time Limit Exceeded", merupakan umpan balik yang didapat jika waktu eksekusi program pengguna melebihi batasan waktu yang diberikan pada deskripsi soal.

3. "Memory Limit Exceeded", merupakan umpan balik yang didapat jika memori yang dibutuhkan program pengguna melebihi batasan waktu yang diberikan pada deskripsi soal.

4. "Runtime Error", merupakan umpan balik yang didapat jika terjadi galat ketika program pengguna sedang berjalan dan menyebabkan program berhenti sebelum selesai dieksekusi.

5. "Compile Error", merupakan umpan balik yang didapat ketika sistem tidak dapat melakukan kompilasi pada berkas kode sumber dari pengguna.

Pada awal pengembangannya, sistem penilaian daring hanya digunakan pada kontes pemrograman untuk menguji jawaban peserta. Namun seiring perkembangan kuantitas kontes pemrograman di dunia, sistem penilaian daring dapat digunakan di luar kontes pemrograman melalui situs penilaian daring. Pada situs penilaian daring terdapat *server* yang berisi sistem penilaian daring. Pengguna dapat mengunggah berkas kode sumber melalui web untuk diuji di *server*, kemudian mendapat umpan balik dari *server* pada situs yang sama. Beberapa contoh situs penilaian daring yang masih aktif ketika tugas akhir ini disusun adalah *Sphere Online Judge* (SPOJ), *UVa Online Judge* (UVa OJ), *TopCoder*, *USACO Training Gate*, dan *Timus Online Judge* (Timus OJ).

[*Halaman ini sengaja dikosongkan*]

BAB III
ANALISIS DAN PERANCANGAN

Pada bagian ini dijelaskan analisis dan desain sistem yang digunakan untuk menyelesaikan permasalahan pada Tugas Akhir ini.

3.1 Analisis dan Observasi Permasalahan

Pada bagian ini diberikan beberapa observasi dan analisis yang akan membantu jalannya implementasi.

3.1.1 Penyelesaian Menggunakan Solusi Naive

Permasalahan ini dapat diselesaikan menggunakan solusi *naive*, yaitu dengan menggabungkan penggunaan *online query* dan *vector* untuk menyimpan *tree* dengan *pseudocode* seperti pada Gambar 3.1.

```
1   T = Input() //total operasi
2   x = 2
3   for i=1 to T
4      scan(operation)
5      if operation = 1
6         scan(index, value)
7         node[index].add(x)
8         values.add(value)
9      else if operation = 2
10        scan(index)
11        if node[index].size = 0
12           print(0)
13        else
14           print(findMaksSum(index))
```

Gambar 3.1 *Pseudocode* Fungsi Main pada Solusi *Naive*

Solusi naive memerlukan 2 fungsi selain main, yaitu findMaksSum dengan *pseudocode* seperti pada Gambar 3.2 dan sums dengan *pseudocode* seperti pada Gambar 3.3.

1	`maks = 0`
2	`for i=0 to node[index].size`
3	`   maks = max(maks, sums(node[index][i]));`
4	`return maks`

Gambar 3.2 Pseudocode Fungsi findMaksSum pada Solusi *Naive*

1	`x = values[index];`
2	`if node[index].size = 0`
3	`   return x`
4	`for i=0 to node[index].size`
5	`   x += sums(node[index][i]);`
6	`return x`

Gambar 3.3 *Pseudocode* Fungsi sums pada Solusi *Naive*

Kompleksitas dari solusi *naive* adalah O(Q*N) sedangkan kompleksitas dari penggunaan *offline query, heavy-light decomposition,* dan *fenwick tree* adalah O(Q*sqrt(N)*log(N)). Perubahan kompleksitas dari O(N) pada solusi *naive* menjadi O(sqrt(N)*log(N)) disebabkan oleh perubahan penggunaan *array* menjadi proses dekomposisi *tree* dengan kompleksitas O(sqrt(N)) dan *fenwick tree* dengan kompleksitas O(log(N)).

Dengan asumsi ada 100.000 *query* dan 100.000 *update* sebagai masukan pada program, maka dapat dibandingkan perbedaan perintah yang dijalankan pada program adalah:

$$Naive = 100.000 \times 100.000 = 10^{10}$$

$$Solusi = 100.000 \times 100 \times 2 = 2 \times 10^{7}$$

Dari perbandingan diatas, dapat diperoleh perbedaan sebesar 10^3 dan *time limit* dari permasalahan adalah 2 detik atau kurang lebih setara dengan 2×10^6 operasi, sehingga penggunaan solusi dengan menggabungkan *offline query, heavy-light decomposition,* dan *fenwick tree* merupakan solusi yang lebih baik daripada solusi *naïve* dan memenuhi kualifikasi SPOJ.

3.1.2 Cara Penyimpanan Operasi

Seluruh operasi yang dilakukan pada permasalahan *Maximum Child Sum* akan dilakukan secara *offline*, yaitu seluruh *state* operasi akan disimpan terlebih dahulu pada sebuah variabel dengan tipe *vector*. Penyimpanan dilakukan agar ukuran dan struktur *tree* yang dibentuk sesuai dengan situasi akhir *problem*, sehingga tidak diperlukan lagi adanya *update* dinamis pada struktur *tree*, melainkan hanya pada nilai dari sebuah *node* dan seluruh *parent* dari *node* tersebut.

3.1.3 Cara Penyimpanan Elemen Tree

Tree yang disusun menggunakan struktur data *Binary Indexed Tree* dengan konsep *preorder numbering* untuk mencari jumlah total dalam suatu *subtree*. Dalam penyimpanan elemen *tree*¸ maka dapat juga menyimpan nilai-nilai lain yang akan digunakan pada operasi-operasi yang akan dijalankan, baik *update* maupun *query*.

Dalam penyelesaian masalah *Maximum Child Sum* ini, diperlukan setidaknya 12 variabel untuk menyimpan nilai-nilai yang diperlukan dalam program, dengan kegunaan yang berbeda. Daftar nama variabel, tipe, dan kegunaannya dapat dilihat pada Gambar 3.5.

Tabel 3.1 Variabel Global

Tipe Variabel	Nama Variabel	Deskripsi
int	a[maxn]	Menyimpan urutan operasi yang akan dilakukan. Jika operasi tersebut adalah *insert*, maka variabel menyimpan nilai *parent* dari *node* yang baru. Jika operasi adalah *query,* maka variabel menyimpan *index* dari *node* yang akan dicari nilai maksimum dari *subtree child*-nya.

Tipe Variabel	Nama Variabel	Deskripsi
int	b[maxn]	Menyimpan nilai *node* baru jika operasi yang dilakukan adalah *insert,* menyimpan nilai 0 jika operasi yang dilakukan adalah *query*.
int	c[maxn]	Menyimpan *index node* baru jika operasi yang dilakukan adalah *insert,* menyimpan nilai 0 jika operasi yang dilakukan adalah *query*.
int	in[maxn]	Menyimpan *index* dari suatu *node* pada *tree* baru
int	out[maxn]	Menyimpan *index* terakhir dari chain suatu *node* pada *tree* baru
int	sum[maxn]	Menyimpan banyaknya *node* pada *tree* dengan *index* ke-*in* hingga ke-*out*
int	hc[maxn]	Menyimpan *index head* dari *chain* tempat *node* tersebut berada
int	prevs[maxn]	Menyimpan nilai maksimum *subtree child* pada suatu *node* sebelum dilakukan operasi
int	sc[maxn]	Menyimpan *index special child* dari *node* tersebut
long long	bit[maxn]	*Binary Indexed Tree*
long long	ans[maxn]	Menyimpan nilai akhir dari maksimum *subtree child* pada suatu *node*
vector<int>	pos[maxn]	Menyimpan *index child* dari setiap *node*

3.1.4 Pemrosesan Berdasarkan Tipe Operasi

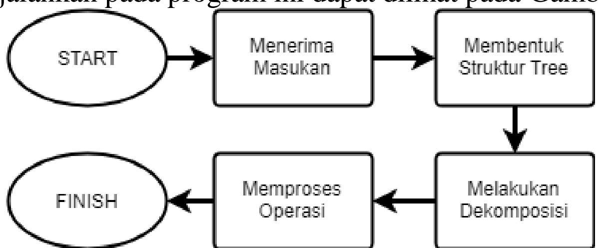
Operasi dibagi menjadi 2 bagian, yaitu *update* dan *query* dengan deskripsi seperti yang telah dijelaskan pada subbab 2.3.2.

3.2 Perancangan Program

Pada bagian ini dijelaskan mengenai desain sistem dalam membantu proses implementasi.

3.2.1 Deskripsi Umum Program

Program menerima masukan berupa sebuah bilangan yang menyatakan jumlah operasi yang akan dijalankan. Urutan proses yang dijalankan pada program ini dapat dilihat pada Gambar 3.4.



Gambar 3.4 Diagram Alur Program

Selanjutnya, program menerima masukan berupa sebuah bilangan 1 atau 2, yang menandakan operasi yang akan dijalankan. Operasi *update* dijalankan ketika masukan berupa angka 1, yang kemudian menerima dua buah bilangan, masing-masing adalah *X* dan *Y*, yang menandakan penambahan *node* dengan nilai *Y* dengan *parent X*. Operasi *query* dijalankan ketika masukan berupa angka 2, yang selanjutnya menerima sebuah angka *X*, yang menandakan program harus mengeluarkan *child* dengan total *sub-tree* maksimum dari *node X.* Kemudian dilakukan inisialisasi *tree* dan menjalankan operasi sesuai dengan tipenya.

Pseudocode Fungsi Main dari sistem ditunjukkan pada Gambar 3.5.

1	`T = Input() //total operasi`
2	`cnt=0, hd=-1`
3	`dfs(1), hld(1)`
4	`for i=1 to T`
5	`   if operasi = 1`
6	`      update(in[c[i]], b[i])`
7	`      uppar(c[i])`
8	`   else print(hitung(a[i]))`

Gambar 3.5 *Pseudocode* Fungsi Main

3.2.2 Desain Fungsi DFS

Fungsi DFS berfungsi sebagai penginisialisasi dari variabel yang diperlukan sebelum operasi dimulai, yang terdiri dari *in, sum,* dan *out,* dan membangun *tree* dengan metode *Depth-First Search*.

Fungsi ini memproses *tree* yang sudah dibentuk sebelumnya, untuk kemudian menghasilkan *tree* dengan *index* yang sesuai dengan struktur *Fenwick Tree* yang kemudian dijelaskan pada Tabel 3.2.

Tabel 3.2 Proses Fungsi DFS

Input	Proses	Output
Tree dengan nilai kosong untuk semua *node* dan struktur yang sudah terbentuk	Mengubah *index node* pada *tree* yang tidak sesuai dengan struktur *Fenwick Tree* menjadi sesuai dengan struktur *Fenwick Tree*	*Tree* dengan *index* yang sesuai dengan struktur *Fenwick* Tree

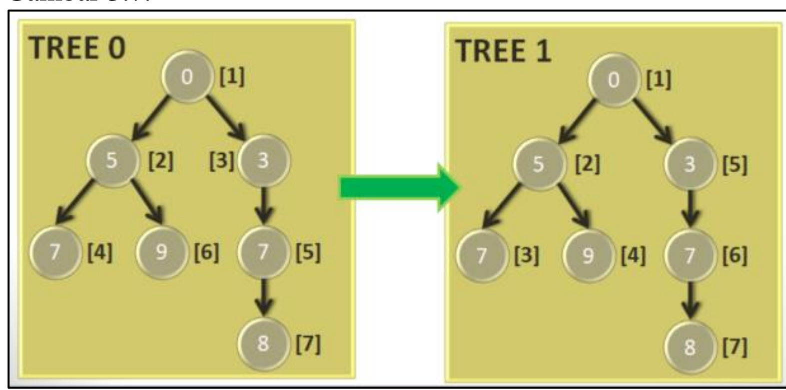
Sebelum operasi dimulai, variabel *in* berisi *index* pada *tree* baru dari setiap *node*, variabel *sum* berisi banyaknya *node* pada *tree* dari *index* ke-*in* sampai *index* ke-*out*, dan variabel *out* berisi

index terakhir dari *chain node* pada *tree* yang baru. *Pseudocode* Fungsi DFS ditunjukkan pada Gambar 3.6.

1	in[now] = ++cnt; sum[now] = 1;
2	for i=0 to sizeOf(pos[now])
3	dfs(pos[now][i])
4	sum[now] += sum[pos[now][i]]
5	out[now] = cnt;

Gambar 3.6 *Pseudocode* Fungsi DFS

Tree baru yang disusun tidak berubah secara struktur, namun hanya memberikan *index* baru dengan metode *DFS*. Visualisasi perubahan *index* pada *tree* tersebut dapat dilihat pada Gambar 3.7.



Gambar 3.7 Visualisasi Fungsi DFS

Pada perubahan dari TREE 0 ke TREE 1, ada 4 *node* yang berubah *index*, yaitu node 4, 6, 3, dan 5. Variabel *in* akan menyimpan nilai dari *index* baru *node* tersebut, dan variabel *out* akan menyimpan *index leaf* dari *chain node* tersebut.

Sebagai contoh, *node* dengan *index* 3. Pada *tree* baru, *index* berubah menjadi 5. *Chain* dari *node* tersebut adalah *index* 5-6-7, sehingga variabel *in[3]* = 5 dan variabel *out[3]* = 7.

3.2.3 Desain Fungsi HLD

Fungsi HLD berfungsi untuk melakukan dekomposisi pada *tree*. Fungsi ini memproses *tree* hasil dari proses di fungsi *DFS* yang kemudian akan dibagi menjadi beberapa *chain* berdasarkan klasifikasi apakah *node* tersebut merupakan *node heavy* atau *light*. Proses fungsi *HLD* dapat dilihat pada Tabel 3.3.

Tabel 3.3 Proses Fungsi HLD

Input	Proses	Output
Tree keluaran dari fungsi *DFS*	Melakukan dekomposisi pada *tree* tersebut sesuai dengan aturan pembagian *heavy* dan *light node*	*Tree* yang sudah di dekomposisi

Fungsi ini akan membentuk *chain* dengan mencari *heavy-vertex* terlebih dahulu, dengan algoritma seperti pada subbab 2.5, Fungsi ini kemudian mencari *head* dari setiap *chain* yang dibentuk dan *special child* dari setiap *node*.

Pseudocode fungsi HLD ditunjukkan pada Gambar 3.8.

```
1   if (hd==-1) hd = now;
2   hc[now] = hd;
3   int mks = -1, id = -1;
4   for i=0 to sizeof(pos[now])
5      int nex = pos[now][i];
6      if sum[nex] > mks
7         mks = sum[nex];
8         id = nex;
9
10  sc[now] = id;
11  if id > 0
12     hld(id)
13  for i=0 to sizeof(pos[now])
14     int nex = pos[now][i];
15     if nex != id
16        hd = -1; hld(nex);
```

Gambar 3.8 *Pseudocode* Fungsi HLD

3.2.4 Desain Fungsi UPDATE

Fungsi UPDATE merupakan fungsi dari *Fenwick Tree* yang digunakan untuk mengupdate seluruh nilai pada *chain* dari suatu *node*. Fungsi ini akan melakukan *tracing* pada *Binary Indexed Tree* dan mengupdate nilai mulai dari *node* yang diupdate menuju ke *ancient* melalui *parent* dari masing-masing *node* sesuai dengan aturan *preorder*. Proses pada fungsi UPDATE dapat dilihat pada Tabel 3.4.

Tabel 3.4 Proses Fungsi UPDATE

Input	Proses	Output
Index node yang akan diperbarui, nilai dari *node* yang akan diperbarui	Melakukan pembaruan dengan metode *preorder numbering* pada *Fenwick Tree*	*BIT* yang sudah diperbarui

Pseudocode dari fungsi update dapat dilihat pada Gambar 3.9.

```
1   For(i=x; i<=n; i+=(i&-i)) bit[i]+=val;
```

Gambar 3.9 *Pseudocode* Fungsi UPDATE

3.2.5 Desain Fungsi TOTAL

Fungsi TOTAL merupakan fungsi untuk mendapatkan nilai dari seluruh *node* pada sebuah *chain*. Fungsi ini akan melakukan *tracing* pada *BIT* dan menjumlahkan nilai mulai dari *node* yang dituju menuju ke *ancient* melalui *parent* dari masing-masing *node*. Proses pada fungsi TOTAL dapat dilihat pada Tabel 3.5.

Tabel 3.5 Proses Fungsi TOTAL

Input	Proses	Output
Index dari *node* yang akan dicari bobotnya	Melakukan perhitungan jumlah *chain* dari suatu *node* dengan metode *tracing* pada *BIT*	Jumlah *chain* dari suatu *node*

Pseudocode dari fungsi total dapat dilihat pada Gambar 3.10.

1	`For(i=x; i>0; i-=(i&-i)) ret+=bit[i];`

Gambar 3.10 *Pseudocode* Fungsi TOTAL

3.2.6 Desain Fungsi JUMLAH

Fungsi JUMLAH merupakan fungsi untuk menghitung jumlah nilai dari *subtree* suatu *node*. Nilai dari *subtree* suatu *node* didapatkan dengan mengurangkan jumlah nilai dari *chain* X, yaitu jumlah nilai *node* tersebut sampai *leaf,* dengan *chain* Y, jumlah nilai dari *root* sampai *node* dengan *index* sebelum *node* tersebut. Fungsi akan menerima masukan berupa 2 *index* yang akan dicari nilai dari *chain*-nya, kemudian mencari bobot dari *subtree* dengan cara mengurangi nilai *chain* kedua *node*, lalu memberikan keluaran berupa jumlah *subtree* dari *node* tersebut dan dapat dilihat pada Tabel 3.6

Tabel 3.6 Proses Fungsi JUMLAH

Input	Proses	Output
Index dari kedua *node* yang akan dicari bobotnya	Mengurangi nilai 2 *node* hasil dari operasi fungsi TOTAL	Jumlah *subtree* dari suatu *node*

Pseudocode dari fungsi total dapat dilihat pada Gambar 3.11.

1	`return total (b) - total(a-1);`

Gambar 3.11 *Pseudocode* Fungsi JUMLAH

3.2.7 Desain Fungsi UPPAR

Fungsi UPPAR merupakan fungsi yang digunakan untuk mengupdate nilai pada variabel *ans* yang kemudian digunakan untuk melakukan perhitungan nilai maksimum pada *subtree*.

Fungsi ini mengambil nilai maksimum dengan membandingkan nilai pada variabel *ans* sebelumnya dengan jumlah dari *chain* setelah ada *node* yang di-*update*. Proses pada fungsi ini dapat dilihat pada Tabel 3.7.

Tabel 3.7 Proses Fungsi UPPAR

Input	Proses	Output
Index dari *node* yang akan diupdate nilainya	Memperbarui nilai *BIT* dari *node* yang akan diperbarui	*BIT* dengan nilai yang telah diperbarui

Pseudocode dari fungsi uppar dapat dilihat pada Gambar 3.12.

```
1   if prevs[x] > 0
2       ans[prevs[x]] = max(ans[prevs[x]],
                            jumlah(in[x], out[x])
3       uppar(prevs[x])
```

Gambar 3.12 *Pseudocode* Fungsi UPPAR

3.2.8 Desain Fungsi HITUNG

Fungsi HITUNG merupakan fungsi yang menghasilkan keluaran berupa total nilai *subtree* terbesar dari *node* yang dicari.

Pada awalnya, fungsi ini akan mengecek apakah *node* tersebut memiliki *special child* (memiliki setidaknya 1 *child*). Jika *node* tersebut memiliki *child,* maka dilakukan perhitungan dengan mencari nilai maksimal antara nilai yang tersimpan di variabel *ans* dengan total nilai *sub-tree* dari *node* tersebut. Nilai yang tersimpan di variabel *ans* akan menjadi nilai maksimal ketika *node* tersebut memiliki lebih dari 1 *child*.

Jika *node* tersebut tidak memiliki *child*, maka fungsi akan langsung mengembalikan nilai yang tersimpan di variabel *ans* dengan *value* 0.

Fungsi menerima masukan sebuah *node* yang akan dicari nilai maksimum *subtree*-nya. Kemudian dilakukan pencarian nilai

maksimal, lalu mengeluarkan nilai maksimal. Proses dapat dilihat pada Tabel 3.8.

Tabel 3.8 Proses Fungsi HITUNG

Input	Proses	Output
Index dari node yang akan dicari nilai *subtree*-nya	Membandingkan nilai dari *subtree* maksimum yang sudah disimpan dengan *looping* ke seluruh *child*	Nilai maksimum *subtree* dari *node*

Pseudocode dari fungsi hitung dapat dilihat pada Gambar 3.13.

1	`if sc[now] > 0`
2	`    ans[now] = max(ans[now],`
	`                jumlah(in[sc[now]],`
	`                out[sc[now]])`
3	`return ans[now]`

Gambar 3.13 *Pseudocode* Fungsi HITUNG

BAB IV
IMPLEMENTASI

Pada bab ini dijelaskan mengenai implementasi dari algoritma dan struktur data berdasarkan desain yang telah dilakukan.

4.1 Lingkungan Implementasi

Lingkungan implementasi menggunakan sebuah komputer dengan spesifikasi perangkat lunak dan perangkat keras seperti tercantum pada Tabel 4.1.

Tabel 4.1 Spesifikasi Lingkungan Implementasi

No.	Jenis Perangkat	Spesifikasi
1	Perangkat Keras	• *Processor* Intel Core i7-2670QM CPU @ 2.20GHz • *Memory* 8GB 1333MHz DDR3
2	Perangkat Lunak	• Sistem operasi Windows 10 • *Integrated Development Environment* Dev-C++ 5.7.1

4.2 Pendefinisian *Preprocessor Directives*

Pada bagian ini dijelaskan beberapa pendefinisian dari *preprocessor directives* yang akan digunakan dalam program ini. Penggunaan *preprocessor directive* ditujukan untuk mempermudah dan mempersingkat implementasi. Implementasi dari *preprocessor directive* ditunjukkan pada Kode Sumber 4.1 .

```
1  #define setmin(x) memset((x), -1, sizeof((x)))
2  #define setnul(x) memset((x), 0, sizeof((x)))
```

Kode Sumber 4.1 Implementasi *Preprocessor Directive*

4.3 Implementasi Fungsi Main

Fungsi Main diimplementasikan sesuai *pseudocode* pada subbab 3.2.1. Pada awalnya sistem akan menerima semua

masukan berupa banyaknya kasus uji, yaitu banyaknya permainan yang akan dijalankan pada sekali eksekusi program dan operasi yang dilakukan. Selanjutnya, fungsi Main menginisialisasi variable *cnt* dan *hd* dan *tree* yang akan digunakan pada program. Kemudian untuk setiap operasi yang telah disimpan, fungsi akan mengoperasikan sesuai dengan tipe operasinya. Implementasi dari fungsi Main dapat dilihat pada Kode Sumber 4.2.

```
1   int main() {
2      gi(q);
3      setmin(prevs);
4      for (int i=0; i<q; i++) {
5         gi(kd[i]);
6         if (kd[i]==1) {
7            gi(a[i]); gi(b[i]);
8            pos[a[i]].push_back(++n);
9            c[i] = n;
10           prevs[n] = a[i];
11        } else gi(a[i]);
12     }
13
14     cnt = 0; dfs (1);
15     hd = -1; hld (1);
16     setnul(ans);
17
18     for (int i=0; i<q; i++) {
19        if (kd[i]==1) {
20           update(in[c[i]], b[i]);
21           uppar(c[i]);
22        } else printf ("%lld\n",
                hitung(a[i]));
23     }
24     return 0;
25  }
```

Kode Sumber 4.2 Implementasi Fungsi Main

4.4 Implementasi Variabel Global

Variabel dengan *scope* global dibutuhkan untuk kemudahan pengaksesan sebuah variabel oleh setiap fungsi yang ingin mengaksesnya. Penggunaan variabel global pada implementasi ini dapat dilihat pada Tabel 3.1. Implementasi dari variabel global dapat dilihat pada Kode Sumber 4.3.

1	`const int inf = (1<<30);`
2	`const int mod = 1e9 + 7;`
3	`const int maxn = 200002;`
4	`int q, kd[maxn], a[maxn], b[maxn],`
	`    c[maxn], in[maxn], out[maxn], n = 1,`
	`    cnt, hd;`
5	`int sum[maxn], hc[maxn], prevs[maxn],`
	`    sc[maxn];`
6	`long long bit[maxn], ans[maxn];`
7	`vector<int> pos[maxn];`

Kode Sumber 4.3 Implementasi Variabel Global

4.5 Implementasi Fungsi DFS

Fungsi DFS akan berfungsi sebagai penginisialisasi dari variabel yang diperlukan sebelum operasi dimulai dan pembentukan struktur *tree*, seperti yang telah dijelaskan pada subbab 3.2.2. Fungsi ini menggunakan 2 variabel *array* satu dimensi, in dan out, serta 2 variabel integer, cnt dan nex. Implementasi dari fungsi DFS terdapat pada Kode Sumber 4.4.

1	`void dfs (int now) {`
2	`  in[now] = ++cnt; sum[now] = 1;`
3	`  for (int i=0; i<pos[now].size(); i++)`
4	`    int nex = pos[now][i];`
5	`    dfs (nex);`
6	`    sum[now] += sum[nex];`
7	`  }`
8	`  out[now] = cnt;`
9	`  return;`
10	`}`

Kode Sumber 4.4 Implementasi Fungsi DFS

4.6 Implementasi fungsi HLD

Fungsi HLD berfungsi untuk melakukan dekomposisi pada *tree* seperti yang telah dijelaskan pada 3.2.3. Fungsi ini akan melakukan dekomposisi dan menyimpan *head* dan *special child* dari masing-masing *node* ke dalam 2 buah *array* satu dimensi, hc dan sc. Implementasi dari fungsi HLD dapat dilihat pada Kode Sumber 4.5.

```
1   void hld (int now) {
2      if (hd==-1) hd = now;
3      hc[now] = hd;
4      int mks = -1, id = -1;
5      for (int i=0; i<pos[now].size(); i++) {
6         int nex = pos[now][i];
7         if (sum[nex]>mks) {
8            mks = sum[nex];
9            id = nex;
10        }
11     }
12     sc[now] = id;
13     if (id>0) hld(id);
14     for (int i=0; i<pos[now].size(); i++) {
15        int nex = pos[now][i];
16        if (nex != id) {
17           hd = -1; hld(nex);
18        }
19     }
20     return;
21  }
```

Kode Sumber 4.5 Implementasi Fungsi HLD

4.7 Implementasi fungsi UPDATE

Fungsi UPDATE mengubah nilai *chain* sebuah *node* dan menjalankan algoritma seperti pada subbab 3.2.4. Fungsi ini menggunakan 1 variabel *array* 1 dimensi, yaitu *BIT* yang kemudian akan diperbarui nilainya sesuai dengan nilai terbaru dari *node* x. *Index* pada *BIT* yang akan diperbarui adalah sesuai

dengan *index* dari *chain node* yang diperbarui. Implementasi dari fungsi UPDATE dapat dilihat pada Kode Sumber 4.6.

```
1  void update (int x, long long val) {
2     for (int i=x; i<=n; i+= i&(-i))
3        bit[i]+=val;
4     return;
5  }
```

Kode Sumber 4.6 Implementasi Fungsi UPDATE

4.8 Implementasi Fungsi TOTAL

Fungsi TOTAL akan menjumlahkan seluruh nilai *node* pada *chain* dari *node* tersebut dengan menjalankan algoritma seperti pada subbab 3.2.5. Fungsi ini menggunakan sebuah variabel dengan tipe data longlong, yaitu ret, yang berfungsi untuk menyimpan nilai jumlah dari *chain node* yang dicari dan kemudian menjadi keluaran dari fungsi ini. Implementasi fungsi TOTAL ditunjukkan pada Kode Sumber 4.7.

```
1  long long total (int x) {
2     long long ret = 0LL;
3     for (int i=x; i>0; i-=i&(-i))
4        ret += bit[i];
5     return ret;
6  }
```

Kode Sumber 4.7 Implementasi Fungsi TOTAL

4.9 Implementasi Fungsi JUMLAH

Fungsi JUMLAH merupakan fungsi untuk menghitung jumlah nilai dari *subtree* suatu *node* yang didapatkan dengan mengurangkan nilai *chain* X dan *chain* Y. Implementasi fungsi JUMLAH ditunjukkan pada Kode Sumber 4.8.

```
1  long long jumlah(int a, int b) {
2     return total (b) - total(a-1);
3  }
```

Kode Sumber 4.8 Implementasi Fungsi JUMLAH

4.10 Implementasi Fungsi UPPAR

Fungsi UPPAR merupakan fungsi yang digunakan untuk mengupdate nilai pada variabel ans yang kemudian digunakan untuk melakukan perhitungan nilai maksimum pada *subtree*. Fungsi ini mengambil nilai maksimum seperti yang telah dijelaskan pada subbab 3.2.7. Implementasi dari fungsi UPPAR dapat dilihat pada Kode Sumber 4.9.

```
1   void uppar (int x) {
2      x = hc[x];
3      if (prevs[x]>0) {
4         ans[prevs[x]] = max (ans[prevs[x]],
                jumlah(in[x], out[x]));
5         uppar(prevs[x]);
6      }
7      return;
8   }
```

Kode Sumber 4.9 Implementasi Fungsi UPPAR

4.11 Implementasi Fungsi HITUNG

Fungsi HITUNG akan mencari nilai maksimum *subtree* dari suatu *node*, seperti telah dijelaskan pada subbab 2.5. Implementasi dari fungsi HITUNG dapat dilihat pada Kode Sumber 4.10.

Fungsi HITUNG akan mencari nilai maksimum dari nilai yang tersimpan di variabel *ans* dengan jumlah dari *sub-tree* dari *node* tersebut jika *node* tersebut memiliki *child*. Jika tidak, maka langsung mengembalikan nilai yang tersimpan pada variabel *ans*.

```
1   long long hitung (int now) {
2      if (sc[now]>0) {
3         ans[now] = max(ans[now],
             jumlah(in[sc[now]], out[sc[now]]));
4      }
5      return ans[now];
6   }
```

Kode Sumber 4.10 Implementasi Fungsi HITUNG

BAB V
UJI COBA DAN EVALUASI

Pada bab ini dijelaskan tentang uji coba dan evaluasi dari implementasi sistem yang telah dilakukan pada bab 4.

5.1 Lingkungan Uji Coba

Lingkungan uji coba menggunakan sebuah komputer dengan spesifikasi perangkat lunak dan perangkat keras seperti tercantum pada Tabel 5.1.

Tabel 5.1 Spesifikasi Lingkungan Uji Coba

No.	Jenis Perangkat	Spesifikasi
1	Perangkat Keras	• *Processor* Intel Core i7-2670QM CPU @ 2.20GHz • *Memory* 8GB 1333MHz DDR3
2	Perangkat Lunak	• Sistem operasi Windows 10 • *Integrated Development Environment* Dev-C++ 5.7.1

5.2 Uji Coba Menggunakan Solusi Naive

Uji coba dilakukan dengan analisis penyelesaian sebuah contoh kasus menggunakan pendekatan *naïve* serta pengumpulan berkas kode sumber hasil implementasi ke situs sistem penilaian daring SPOJ. Permasalahan yang diselesaikan adalah *Maximum Child Sum* dengan kode *MAXCHILDSUM*. Uji coba kebenaran akan dilakukan dengan menggunakan kasus uji dari daring SPOJ.

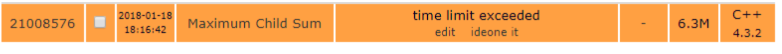
Uji coba kebenaran dilakukan dengan melakukan interaksi dengan *judge* yang terdapat pada daring SPOJ dengan kode permasalahan *MAXCHILDSUM*. Dalam melakukan uji coba, pertama dilakukan terlebih dahulu pengujian terhadap contoh kasus uji yang telah disertakan dalam permasalahan tersebut.

```
Input:

7
1 1 3
2 1
2 2
1 2 5
2 1
1 1 4
2 10
```

Dengan data masukan seperti di atas, maka didapatkan *output* yang sama dengan penggunaan solusi gabungan yaitu {3,0,8,8}.

Selanjutnya, uji coba kebenaran juga dilakukan dengan mengumpulkan berkas program ke daring SPOJ. Hasil dari pengumpulan berkas ke daring SPOJ dapat dilihat pada Gambar 5.1.



Gambar 5.1 Hasil Uji Kebenaran Solusi *Naive* permasalahan *MAXCHILDSUM* pada Situs SPOJ

Dari hasil uji coba yang telah dilakukan, kode sumber program mendapat hasil *Time Limit Exceeded,* yang berarti solusi tidak memenuhi batas waktu yang ditentukan.

5.3 Uji Coba Menggunakan Offline Query, HLD, dan Fenwick Tree

Uji coba dilakukan dengan analisis penyelesaian sebuah contoh kasus menggunakan pendekatan yang telah dijelaskan pada Bab II serta pengumpulan berkas kode sumber hasil implementasi ke situs sistem penilaian daring SPOJ. Permasalahan yang diselesaikan adalah *Maximum Child Sum*

dengan kode *MAXCHILDSUM*. Uji coba kebenaran akan dilakukan dengan menggunakan kasus uji dari daring SPOJ.

Uji coba kebenaran dilakukan dengan melakukan interaksi dengan *judge* yang terdapat pada daring SPOJ dengan kode permasalahan *MAXCHILDSUM*. Dalam melakukan uji coba, dilakukan pengujian terhadap contoh kasus uji yang telah disertakan dalam permasalahan tersebut.

```
Input:

7
1 1 3
2 1
2 2
1 2 5
2 1
1 1 4
2 10
```

Penyelesaian problem dilakukan dalam 4 langkah, yaitu menerima masukan, membentuk *tree*, melakukan dekomposisi, lalu mengeksekusi operasi.

5.3.1 Menerima Masukan

Masukan pada program diproses secara *offline* dan disimpan kedalam 4 *variabel*, yaitu A, B, C, dan pos.

Nilai pada masing-masing variabel dalam menyimpan masukan dapat dilihat pada Gambar 5.2.

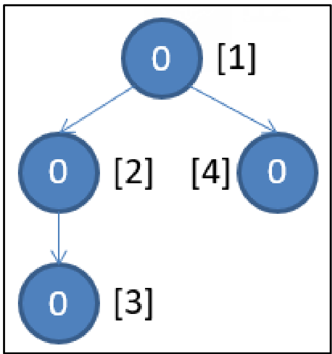
idx	0	1	2	3	4	5	6
A	1	1	2	2	1	1	1
B	3	0	0	5	0	4	0
C	2	0	0	3	0	4	0

Index	Value
1	2 4
2	3
3	-
4	-

Gambar 5.2 Nilai pada Variabel a, b, c, dan pos setelah Menerima Masukan pada Kasus Uji Coba

5.3.2 Membentuk Tree

Program akan membentuk struktur *tree* sesuai dengan masukan yang diberikan. Seluruh *node* pada *tree* diinisialisasi dengan nilai 0 dan kemudian dilakukan perubahan *index* (bila perlu) agar struktur *tree* sesuai dengan struktur *Fenwick Tree*. Struktur *tree* yang dibentuk dapat dilihat pada Gambar 5.3.



Gambar 5.3 Struktur *Tree* yang Dibentuk pada Kasus Uji Coba

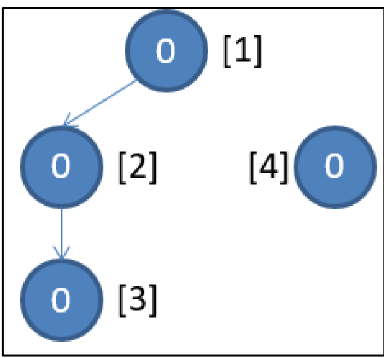
Tree tersebut dibentuk oleh 3 variabel, yaitu In, Out, dan Sum. Nilai pada masing-masing variable dapat dilihat pada Gambar 5.4.

Index	0	1	2	3	4
In	0	1	2	3	4
Out	0	4	3	3	4
Sum	0	4	2	1	1

Gambar 5.4 Nilai pada Varibel in, out, dan sum setelah Membentuk *Tree* pada Kasus Uji Coba

5.3.3 Melakukan Dekomposisi

Setelah membentuk struktur *tree*¸ maka dapat dilakukan proses dekomposisi *tree* menjadi *disjoint chains.* Dilakukan dekomposisi pada Gambar 5.3 menjadi Gambar 5.5.



Gambar 5.5 Hasil Dekomposisi pada Kasus Uji Coba

Hasil dekomposisi disimpan pada 2 variabel, yaitu hc dan sc. Nilai pada variabel tersebut dapat dilihat pada Gambar 5.6.

Index	0	1	2	3	4
Hc	0	1	1	1	4
sc	0	2	3	-1	-1

Gambar 5.6 Nilai pada Variabel hc dan sc setelah Melakukan Dekomposisi pada Kasus Uji Coba

5.3.4 Eksekusi Operasi

Dalam masukan yang telah diberikan diatas, maka ada 7 operasi yang akan dilakukan. Perbedaan perlakuan pada operasi *insert/update* dan *query* akan dilakukan sesuai dengan penjelasan pada subbab 2.5. Jalannya eksekusi melibatkan 2 variabel, yaitu bit dan ans, dan dapat dilihat pada Tabel 5.2.

Tabel 5.2 Visualisasi Eksekusi Operasi pada Kasus Uji Coba

<table>
<tr><th>Operasi</th><th>Tree dan Variabel</th><th>Keterangan</th></tr>
<tr><td>1 1 3</td><td>0 [1]
3 [2] [4] 0
0 [3]
<table><tr><th>Index</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th></tr><tr><td>BIT</td><td>0</td><td>0</td><td>3</td><td>0</td><td>3</td></tr><tr><td>ans</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table></td><td>Insert node dengan index 2, parent 1, dan nilai 3</td></tr>
</table>

<table>
<tr><th>Operasi</th><th>Tree dan Variabel</th><th>Keterangan</th></tr>
<tr><td>2 1</td><td>
<table>
<tr><th>Index</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th></tr>
<tr><td>BIT</td><td>0</td><td>0</td><td>3</td><td>5</td><td>8</td></tr>
<tr><td>ans</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
</table>
</td><td>Cari nilai subtree maksimum child dari node 1

Ans[1] = 0 → lakukan loop ke child
node 2=3, node 4=0

Output 3
Update ans[1] = 3</td></tr>
<tr><td>2 2</td><td>
<table>
<tr><th>Index</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th></tr>
<tr><td>BIT</td><td>0</td><td>0</td><td>3</td><td>5</td><td>8</td></tr>
<tr><td>ans</td><td>0</td><td>3</td><td>0</td><td>0</td><td>0</td></tr>
</table>
</td><td>Cari nilai subtree maksimum child dari node 2

Ans[2] = 0 dan node 2 tidak memiliki child

Output 0</td></tr>
<tr><td>1 2 5</td><td>
<table>
<tr><th>Index</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th></tr>
<tr><td>BIT</td><td>0</td><td>0</td><td>3</td><td>5</td><td>8</td></tr>
<tr><td>ans</td><td>0</td><td>3</td><td>0</td><td>0</td><td>0</td></tr>
</table>
</td><td>Insert node dengan index 3, parent 2, dan nilai 5</td></tr>
</table>

<table>
<tr><th>Operasi</th><th>Tree dan Variabel</th><th>Keterangan</th></tr>
<tr><td>2 1</td><td><table>
<tr><th>Index</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th></tr>
<tr><td>BIT</td><td>0</td><td>0</td><td>3</td><td>5</td><td>8</td></tr>
<tr><td>ans</td><td>0</td><td>3</td><td>0</td><td>0</td><td>0</td></tr>
</table></td><td>Cari nilai subtree maksimum child dari node 1

Ans[1] = 3 → bandingkan dengan loop seluruh child. Ans=3, subtree node 2=8

Output 8
Update ans[1]=8</td></tr>
<tr><td>1 1 4</td><td><table>
<tr><th>Index</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th></tr>
<tr><td>BIT</td><td>0</td><td>0</td><td>3</td><td>5</td><td>12</td></tr>
<tr><td>ans</td><td>0</td><td>8</td><td>0</td><td>0</td><td>0</td></tr>
</table></td><td>Insert node dengan index 4, parent 1, dan nilai 4</td></tr>
</table>

<table>
<tr><th>Operasi</th><th>Tree dan Variabel</th><th>Keterangan</th></tr>
<tr>
<td>2 1</td>
<td>0 [1]
3 [2] [4] 4
5 [3]
<table>
<tr><th>Index</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th></tr>
<tr><td>BIT</td><td>0</td><td>0</td><td>3</td><td>5</td><td>12</td></tr>
<tr><td>ans</td><td>0</td><td>8</td><td>0</td><td>0</td><td>0</td></tr>
</table>
</td>
<td>Cari nilai subtree maksimum child dari node 1

Ans[1] = 8 → bandingkan dengan loop seluruh child. Ans=8, subtree node 2 = 8

Output 8</td>
</tr>
</table>

Selanjutnya, dilakukan uji coba kebenaran dengan melakukan pengumpulan berkas program ke daring SPOJ. Hasil dari pengumpulan berkas ke daring SPOJ dapat dilihat pada Gambar 5.7.

21004530	☐	2018-01-18 04:43:05	Maximum Child Sum	accepted edit ideone it	0.29	38M	CPP14-CLANG
20962188	☐	2018-01-11 06:29:34	Maximum Child Sum	accepted edit ideone it	0.33	38M	CPP14
20962176	☐	2018-01-11 06:26:16	Maximum Child Sum	accepted edit ideone it	0.33	38M	CPP14
20962175	☐	2018-01-11 06:26:09	Maximum Child Sum	accepted edit ideone it	0.32	38M	CPP14
20962172	☐	2018-01-11 06:25:18	Maximum Child Sum	accepted edit ideone it	0.33	38M	CPP14
20962170	☐	2018-01-11 06:25:12	Maximum Child Sum	accepted edit ideone it	0.32	38M	CPP14
20962164	☐	2018-01-11 06:24:07	Maximum Child Sum	accepted edit ideone it	0.31	38M	CPP14
20962163	☐	2018-01-11 06:24:02	Maximum Child Sum	accepted edit ideone it	0.34	38M	CPP14
20962154	☐	2018-01-11 06:22:58	Maximum Child Sum	accepted edit ideone it	0.33	38M	CPP14
20650426	☐	2017-11-22 06:11:12	Maximum Child Sum	accepted edit ideone it	0.32	38M	CPP14

Gambar 5.7 Hasil Uji Kebenaran Solusi Gabungan *Offline Query*, *HLD*, dan *BIT* pada Permasalahan *MAXCHILDSUM* pada Situs SPOJ

Dari hasil uji coba yang telah dilakukan, kode sumber program mendapat hasil *Accepted*. Waktu rata-rata yang dibutuhkan program adalah 0.29 detik dan memori yang dibutuhkan adalah 38 MB. Hasil uji coba diatas membuktikan bahwa implementasi yang dilakukan telah berhasil menyelesaikan permasalahan dengan batasan-batasan yang telah ditetapkan.

Selanjutnya, kinerja implementasi pada Tugas Akhir ini dibandingkan dengan kinerja implementasi pengguna lain. Perbandingan performa dapat dilihat pada Gambar 5.8.

RANK	DATE	USER	RESULT	TIME	MEM	LANG
1	2017-03-13 08:27:55	Muhammad Salman Al-Farisi	**accepted**	0.24	37M	CPP14
2	2017-03-07 02:47:28	Muhammad Salman	**accepted**	0.25	35M	CPP14-CLANG
3	2018-01-18 04:43:05	prasetyon	**accepted**	0.29	38M	CPP14-CLANG
4	2017-03-04 08:54:25	Sergio Vieri	**accepted**	0.31	46M	CPP14-CLANG
5	2017-04-02 07:56:02	zhangqingchuan	**accepted**	0.31	38M	CPP14
6	2018-01-11 06:24:07	prasetyon	**accepted**	0.31	38M	CPP14
7	2017-03-05 18:28:49	Steven Wijaya	**accepted**	0.33	38M	CPP14-CLANG
8	2016-08-26 16:57:36	Willy	**accepted**	0.34	23M	CPP14
9	2017-11-22 06:09:26	laravelz	**accepted**	0.34	38M	CPP14
10	2017-03-05 20:26:48	Rully Soelaiman	**accepted**	0.36	20M	C++ 4.3.2

Gambar 5.8 Peringkat Waktu Eksekusi Program *Maximum Child Sum* pada SPOJ

BAB VI
KESIMPULAN

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

6.1 Kesimpulan

Dari hasil uji coba yang telah dilakukan terhadap implementasi solusi untuk permasalahan *MAXCHILDSUM* dalam mencari sebuah *child* dari sebuah *node* dengan nilai *subtree* maksimal diantara *child* lainnya, dapat diambil kesimpulan sebagai berikut:

1. Permasalahan mencari *child* dari suatu *node* yang memiliki *subtree* dengan jumlah maksimal diantara *child* lainnya telah berhasil diselesaikan dengan struktur data *Fenwick Tree* dan teknik *Heavy-Light Decomposition.*
2. Penggunaan teknik *Heavy-Light Decomposition* dan Struktur Data *Fenwick Tree* dapat menyelesaikan permasalahan operasi *query* mencari *child* dengan nilai *subtree* maksimal dengan kompleksitas O(Q*sqrt(N)*log(N)) dibandingkan dengan solusi *naïve* dengan kompleksitas O(Q*N).
3. Waktu yang diperlukan untuk menyelesaikan permasalahan *Maximum Child Sum* pada daring SPOJ adalah 0.29 detik dan memori yang digunakan adalah sebesar 38 MB.

6.2 Saran

Saran yang diberikan untuk implementasi tugas akhir ini adalah optimasi *running time* program agar mampu menjadi solusi dengan *time* paling kecil pada daring SPOJ.

[*Halaman ini sengaja dikosongkan*]

DAFTAR PUSTAKA

[1] A. Nekkanti, "**Heavy Light Decomposition - Anudeep's Blog**," 11 April 2014. [Online]. Available: https://blog.anudeep2011.com/heavy-light-decomposition/.

[2] S. O. Judge, "**SPOJ.com - Problem MAXIMUM CHILD SUM**," 22 August 2016. [Online]. Available: http://www.spoj.com/problems/MAXCHILDSUM/.

[3] Codeforces, "**What is an offline solution? - Codeforces**," 14 August 2015. [Online]. Available: http://codeforces.com/blog/entry/19771.

[4] Wikipedia, "**Fenwick Tree - Wikipedia**," 18 December 2017. [Online]. Available: https://en.wikipedia.org/wiki/Fenwick_tree.

[5] GeeksforGeeks, "**Binary Indexed Tree or Fenwick Tree - GeeksforGeeks**," [Online]. Available: http://www.geeksforgeeks.org/binary-indexed-tree-or-fenwick-tree-2/.

[6] cplusplus.com, "**vector - C++ Reference**," [Online]. Available: http://www.cplusplus.com/reference/vector/vector/.

[7] TutorialsPoint, "**Data Structure and Algorithms - Tree**," [Online]. Available: https://www.tutorialspoint.com/data_structures_algorithms/tree_data_structure.htm.

[8] E. W. Weisstein, "**Acyclic Graph -- From MathWorld--A Wolfram Web Resource**," [Online]. Available: http://mathworld.wolfram.com/AcyclicGraph.html.

[9] E. W. Weisstein, "**Tree -- From MathWorld--A Wolfram Web Resource.**," [Online]. Available: http://mathworld.wolfram.com/Tree.html.

[10] saragusti22, "**Pengantar Struktur Data : Tree dan Binary Tree**," 4 May 2015. [Online]. Available: https://saragusti22.wordpress.com/2015/05/04/pengantar-struktur-data-tree-dan-binary-tree/.

LAMPIRAN

[*Halaman ini sengaja dikosongkan*]

BIODATA PENULIS

Prasetyo Nugrohadi, lahir di Surabaya pada tanggal 12 Februari 1997. Penulis merupakan seorang mahasiswa yang sedang menempuh studi di Departemen Informatika, Fakultas Teknologi Informasi, Institut Teknologi Sepuluh Nopember (ITS) Surabaya.

Memiliki beberapa hobi antara lain olahraga dan bermain musik. Pernah menjadi asisten dosen di mata kuliah Dasar Pemrograman dan Struktur Data.

Selama menempuh pendidikan di kampus, penulis juga aktif dalam organisasi kemahasiswaan, antara lain Staff Depertemen Pengembangan Sumber Daya Manusia pada tahun pertama. Pada tahun kedua, aktif sebagai staff Kaderisasi dan Pemetaan Himpunan Mahasiswa Teknik Computer-Informatika (HMTC) dan staff Panitia Pembinaan Kerohanian Mahasiswa Baru Kristen (PKMBK). Di tahun ketiga, aktif sebagai Ketua Panitia PKMBK dan Wakil Ketua Schematics HMTC. Pada tahun terakhir, mendapat amanah sebagai Ketua Persekutuan Mahasiswa Kristen (PMK) ITS 2017/2018. Selain aktif di bidang organisasi, penulis juga menjadi pemandu untuk kegiatan Latihan Keterampilan Manajemen Mahasiswa.

Penulis dapat dihubungi melalui surel di prstyngrhd@gmail.com.