

**PERANCANGAN MODUL PENGINDEKS PADA *SEARCH ENGINE* BERUPA *GENERALIZED SUFFIX TREE* UNTUK
KEPERLUAN PEMERINGKATAN DOKUMEN**

Skripsi

**Disusun untuk memenuhi salah satu syarat
memperoleh gelar Sarjana Komputer**



**Oleh:
Zaidan Pratama
1313618013**

**PROGRAM STUDI ILMU KOMPUTER
FAKULTAS MATEMATIKA DAN ILMU PENGETAHUAN ALAM
UNIVERSITAS NEGERI JAKARTA**

2023

LEMBAR PERSETUJUAN HASIL SIDANG SKRIPSI

PERANCANGAN MODUL PENGINDEKS PADA *SEARCH ENGINE* BERUPA *GENERALIZED SUFFIX TREE* UNTUK KEPERLUAN PEMERINGKATAN DOKUMEN

Nama	: Zaidan Pratama		
No. Registrasi	: 1313618013		
	Nama	Tanda Tangan	Tanggal
Penanggung Jawab			
Dekan	: <u>Prof. Dr. Muktiningsih N, M.Si.</u>
	NIP. 196405111989032001		
Wakil Penanggung Jawab			
Wakil Dekan I	: <u>Dr. Esmar Budi, S.Si., MT.</u>
	NIP. 197207281999031002		
Ketua	: <u>Drs. Mulyono, M.Kom.</u>	17-02-2023
	NIP. 196605171994031003		
Sekretaris	: <u>Ir. Fariani Hermin Indiyah, M.T.</u>	17-02-2023
	NIP. 196002111987022001		
Penguji	: <u>Ria Arafiah, M.Si.</u>	17-02-2023
	NIP. 197511212005012004		
Pembimbing I	: <u>Muhammad Eka Suryana, M.Kom.</u>	20-02-2023
	NIP. 19851223201211002		
Pembimbing II	: <u>Med Irzal, M.Kom.</u>	20-02-2023
	NIP. 197706152003121001		

Dinyatakan lulus ujian skripsi tanggal: 14 Februari 2023

LEMBAR PERNYATAAN

Saya menyatakan dengan sesungguhnya bahwa skripsi dengan judul **Perancangan Modul Pengindeks Pada *Search Engine* Berupa *Generalized Suffix Tree* Untuk Keperluan Pemeringkatan Dokumen** yang disusun sebagai syarat untuk memperoleh gelar Sarjana komputer dari Program Studi Ilmu Komputer Universitas Negeri Jakarta adalah karya ilmiah saya dengan arahan dari dosen pembimbing.

Sumber informasi yang diperoleh dari penulis lain yang telah dipublikasikan yang disebutkan dalam teks skripsi ini, telah dicantumkan dalam Daftar Pustaka sesuai dengan norma, kaidah dan etika penulisan ilmiah.

Jika di kemudian hari ditemukan sebagian besar skripsi ini bukan hasil karya saya sendiri dalam bagian-bagian tertentu, saya bersedia menerima sanksi pencabutan gelar akademik yang saya sanding dan sanksi-sanksi lainnya sesuai dengan peraturan perundang-undangan yang berlaku.

Jakarta, 31 Januari 2023

Zaidan Pratama

LEMBAR PERSEMBAHAN



Untuk Ibu, Bapak, Adik-Adik dan Diriku

KATA PENGANTAR

Puji syukur kepada Allah Subhanahu wa Ta'alla atas berkah rahmat, hidayah, dan karunia-Nya sehingga penulis dapat menyelesaikan skripsi yang berjudul “Perancangan Modul Pengindeks Pada *Search Engine* Berupa *Generalized Suffix Tree* Untuk Keperluan *Pemeringkatan Dokumen*”. Skripsi ini disusun untuk memenuhi syarat kelulusan studi pada Program Studi Ilmu Komputer Fakultas Matematika dan Ilmu Pengetahuan Alam Universitas Negeri Jakarta.

Dalam proses penyusunan skripsi ini banyak pihak yang telah membantu dan memberikan dukungan, doa, serta bimbingannya. Oleh karena itu pada kesempatan kali ini penulis ingin mengucapkan terima kasih kepada:

1. Allah SWT atas segala rahmat dan lindungan-Nya yang telah memberikan kemudahan selama proses penyusunan skripsi.
2. Orang tua dan keluarga saya yang selalu memberikan semangat serta doa yang tidak henti-hentinya untuk kelancaran saya menempuh pendidikan dan menyelesaikan skripsi ini.
3. Ibu Ir. Fariani Hermin Indiyah, M.T., selaku Koordinator Program Studi Ilmu Komputer Fakultas Matematika dan Ilmu Pengetahuan Alam Universitas Negeri Jakarta.
4. Bapak Muhammad Eka Suryana M.Kom., selaku Dosen Pembimbing I yang telah memberikan arahan dan bimbingan dalam penulisan skripsi ini.
5. Bapak Med Irzal M.Kom., selaku Dosen Pembimbing II yang telah memberikan arahan dan bimbingan dalam penulisan skripsi ini.
6. Teman-teman dan sahabat terutama Andri Rahmanto yang telah banyak membantu dari segi diskusi dan memberi dukungan moral sehingga skripsi ini dapat selesai dengan sebaik-baiknya.
7. Semua pihak yang telah memberikan bantuan dan dukungan dalam penyusunan skripsi ini yang tidak dapat penulis sebutkan satu per satu.

Skripsi ini masih jauh dari kata sempurna dari segi isi maupun sistematika. Untuk itu kritik dan saran yang bersifat membangun sangat dibutuhkan guna

menyempurnakan skripsi ini menjadi lebih baik. Semoga skripsi ini bisa bermanfaat bagi pembaca pada umumnya.

Jakarta, 31 Januari 2023

Zaidan Pratama



ABSTRAK

ZAIDAN PRATAMA. Perancangan Modul Pengindeks Pada *Search Engine* Berupa *Generalized Suffix Tree* Untuk Keperluan Pemeringkatan Dokumen. Skripsi. Fakultas Matematika dan Ilmu Pengetahuan Alam, Universitas Negeri Jakarta. 2023. Di bawah bimbingan Muhammad Eka Suryana, M.Kom dan Med Irzal, M.Kom.

Untuk membuat *search engine* diperlukan banyak komponen yang mendukung kebutuhan arsitektur *search engine*. Pada penelitian sebelumnya, komponen *crawler* berhasil dibuat oleh Muhammad Fathan Qoriiba dalam penelitiannya yang berjudul "Perancangan *Crawler* Sebagai Pendukung Pada *Search Engine*". Salah satu komponen dari *search engine* yang belum diimplementasikan adalah modul pengindeks atau *indexer*. Penelitian ini bertujuan untuk membuat modul pengindeks menggunakan struktur data *Generalized Suffix Tree* untuk keperluan pemeringkatan dokumen sekaligus melanjutkan rangkaian penelitian *search engine*. Proses pembentukan *Generalized Suffix Tree* dan seluruh program dibuat menggunakan bahasa *Python*. Hasil akhir dari penelitian ini adalah modul pengindeks berupa program *console* yang digunakan untuk melakukan pencarian dengan nilai *Mean Average Precision* dari 3 *tester* sebesar 0.658 dan struktur data *Generalized Suffix Tree* dengan kedalaman maksimal 11 level dan total jumlah daun sebanyak 34.573 daun yang tersimpan dalam direktori yang sama.

Kata kunci: *search engine, document retrieval, modul pengindeks, generalized suffix tree, pemeringkatan dokumen*

ABSTRACT

ZAIDAN PRATAMA. *Design of a Generalized Suffix Tree based Indexing Module for Document Ranking in a Search Engine. Thesis. Faculty of Mathematics and Natural Sciences, State University of Jakarta. 2023. Supervised by Muhammad Eka Suryana, M.Kom and Med Irzal, M.Kom.*

To create a search engine, many components are needed to support the architecture of the search engine. In a previous study, Muhammad Fathan Qoriiba successfully developed the crawler component in his research entitled "Designing a Crawler as Support for a Search Engine." One component of the search engine that has not been implemented is the indexing module, or indexer. The purpose of this study is to create an indexing module using the Generalized Suffix Tree data structure for the purpose of document ranking and to continue the series of search engine research. The process of forming the Generalized Suffix Tree and all programs were developed using the Python language. The final result of this research is an indexing module in the form of a console program that is used to perform searches with a Mean Average Precision value of 0.658 from 3 testers, and a Generalized Suffix Tree data structure with a maximum depth of 11 levels and a total of 34,573 leaves stored in the same directory.

Keywords: *search engine, document retrieval, indexing module, generalized suffix tree, document ranking*

DAFTAR ISI

DAFTAR ISI	xi
DAFTAR GAMBAR	xiii
DAFTAR TABEL	xiv
I PENDAHULUAN	1
A. Latar Belakang Masalah	1
B. Rumusan Masalah	5
C. Pembatasan Masalah	5
D. Tujuan Penelitian	6
E. Manfaat Penelitian	6
II KAJIAN PUSTAKA	7
A. <i>Search Engine</i>	7
B. <i>Document Retrieval</i>	9
C. Masalah <i>Color Set Size</i>	10
D. <i>Generalized Suffix Tree</i>	11
1. Konstruksi <i>Generalized Suffix Tree</i>	12
2. <i>Suffix Range</i>	14
3. <i>Optimal Index for Colored Range Query</i>	15
4. <i>Y-fast Trie for Efficient Successor Query</i>	15
E. <i>Induced Generalized Suffix Tree</i>	15
1. <i>IGST-f: IGST Untuk Frekuensi f</i>	15
2. Representasi Array dari <i>IGST-f</i>	17
F. Indeks Efisien Untuk <i>Top-k Document Retrieval Problem</i>	19
G. Konstruksi Algoritma Pengindeksan	21
III METODOLOGI PENELITIAN	24
A. Tahapan Penelitian	24
1. <i>Pseudocode</i> Algoritma	26
B. Contoh Konstruksi <i>Generalized Suffix Tree</i>	26
C. Alat dan Bahan Penelitian	28
D. Dataset	29
E. Tahapan Pengembangan	32
F. Pengujian	32
1. Relevansi Pencarian	33
2. Waktu Pengindeksan	34
IV HASIL DAN PEMBAHASAN	35
A. Pengolahan Dataset	36

1.	Pengambilan Data dan Pembersihan Data	37
B.	Pembentukan <i>Generalized Suffix Tree</i>	39
1.	Penambahan Sufiks Ke Dalam Tree	40
2.	Penyimpanan <i>Generalized Suffix Tree</i> Sebagai Objek Menggunakan <i>Pickle</i>	42
3.	<i>Generalized Suffix Tree</i> yang Terbentuk	42
C.	Mereduksi <i>Node Redundan</i> pada Tree	44
D.	<i>Input Pola P</i>	45
E.	Mencari Nilai <i>Count</i>	46
F.	Merepresentasikan <i>Array</i> Hasil Berdasarkan Nilai <i>Count</i>	49
G.	Struktur Direktori Kode dan Tree	49
H.	Hasil Pengujian	50
1.	Pengujian <i>Mean Average Precision</i>	58
2.	Pengujian Performa Kecepatan (<i>Speed Performance Testing</i>)	59
I.	Analisis Hasil	59
J.	<i>Repository</i> Kode	60
V	KESIMPULAN DAN SARAN	61
A.	Kesimpulan	61
B.	Saran	62
	DAFTAR PUSTAKA	64
	LAMPIRAN A	65
A	Kuesioener Relevansi Hasil Pencarian Menggunakan Metode <i>Mean Average Precision</i>	65
A.	Identitas <i>Tester</i>	65
B.	Kuesioner Penilaian	65
	LAMPIRAN B	67
B	Jawaban Kuesioener Relevansi Hasil Pencarian Menggunakan Metode <i>Mean Average Precision</i>	67
A.	Identitas <i>Tester</i>	67
B.	Kuesioner Penilaian	67
	LAMPIRAN C	69
C	Jawaban Kuesioener Relevansi Hasil Pencarian Menggunakan Metode <i>Mean Average Precision</i>	69
A.	Identitas <i>Tester</i>	69
B.	Kuesioner Penilaian	69
	LAMPIRAN D	71

D Jawaban Kuesioener Relevansi Hasil Pencarian Menggunakan Metode	
<i>Mean Average Precision</i>	71
A. Identitas <i>Tester</i>	71
B. Kuesioner Penilaian	71



DAFTAR GAMBAR

Gambar 1.1	<i>High Level Google Architecture</i> (Brin dan Page, 1998)	3
Gambar 2.1	<i>High Level Google Architecture</i> (Brin dan Page, 1998)	8
Gambar 2.2	<i>CSS Example</i> (Chi dan Hui, 1992)	10
Gambar 2.3	<i>Generalized Suffix Tree</i> (Hon dkk., 2010)	12
Gambar 2.4	<i>Suffix Tree</i> (McCreight, 1976)	12
Gambar 2.5	Pembentukan <i>Suffix Tree</i> <i>ababc</i> (McCreight, 1976)	13
Gambar 2.6	<i>Suffix Range</i> (Hon dkk., 2010)	14
Gambar 2.7	<i>Pre-IGST-2</i> (Hon dkk., 2010)	16
Gambar 2.8	<i>IGST-2</i> (Hon dkk., 2010)	16
Gambar 2.9	Contoh dari penggabungan <i>list L</i> (Hon dkk., 2010)	20
Gambar 3.1	<i>Flowchart</i> Tahapan Penelitian Modul Pengindeks	24
Gambar 3.2	<i>Flowchart</i> Pencarian Nilai <i>Count</i>	25
Gambar 3.3	Contoh Konstruksi <i>Generalized Suffix Tree</i>	26
Gambar 3.4	5 Data Pertama Dataset pada Tabel <i>Page Information</i>	29
Gambar 4.1	Komponen <i>Flowchart</i> yang Diimplementasikan	35
Gambar 4.2	Jumlah Data pada <i>Database</i> Hasil <i>Crawling</i>	36
Gambar 4.3	Tampilan Data pada Tabel <i>page information</i>	37
Gambar 4.4	<i>Source Code</i> untuk Mengambil Data dari <i>Database</i>	38
Gambar 4.5	Sampel Data <i>Title</i> yang Belum Dibersihkan dan Sudah Dibersihkan	38
Gambar 4.6	<i>Source Code</i> untuk Membuat <i>Generalized Suffix Tree</i>	39
Gambar 4.7	<i>Source Code</i> untuk Menambah Sufiks pada <i>Tree</i>	40
Gambar 4.8	<i>Source Code</i> untuk Menambah Sufiks pada <i>Tree</i>	41
Gambar 4.9	<i>Source Code</i> untuk Membandingkan Sufiks dengan <i>Node</i>	41
Gambar 4.10	<i>Source Code</i> untuk Menyimpan <i>Tree</i>	42
Gambar 4.11	Ilustrasi Potongan <i>Tree</i> yang Terbentuk	43
Gambar 4.12	<i>Source Code</i> untuk Membaca <i>Tree</i>	43
Gambar 4.13	<i>Source Code</i> untuk Melihat Kedalaman dan Jumlah Daun pada <i>Tree</i>	44
Gambar 4.14	Hasil Jumlah Maksimal Kedalaman <i>Tree</i> dan Jumlah Daun pada <i>Tree</i>	44
Gambar 4.15	<i>Source Code</i> untuk Mencari Kata di <i>Tree</i>	45
Gambar 4.16	<i>Source Code</i> untuk Mencari Karakter di <i>Tree</i>	46
Gambar 4.17	Contoh Hasil Pencarian dengan <i>Query</i> Resesi 2023	46
Gambar 4.18	<i>Source Code</i> untuk Melakukan Pemeringkatan Hasil Pencarian	47
Gambar 4.19	<i>Source Code</i> untuk Melakukan Pemeringkatan Hasil Pencarian	47
Gambar 4.20	Contoh <i>Array</i> Hasil Pemeringkatan <i>Index</i> Berdasarkan Nilai <i>Count</i>	48

Gambar 4.21	<i>Source Code</i> untuk Merepresentasikan <i>Array</i> Hasil Berdasarkan Nilai <i>Count</i>	49
Gambar 4.22	Struktur Direktori Kode dan <i>Tree</i>	49
Gambar 4.23	<i>Source Code</i> Fungsi <i>Main</i>	50
Gambar 4.24	Hasil Pencarian <i>Keyword Bjorka</i>	51
Gambar 4.25	Hasil Pencarian <i>Keyword Messi</i>	52
Gambar 4.26	Hasil Pencarian <i>Keyword Masak</i>	53
Gambar 4.27	Hasil Pencarian <i>Keyword Resesi 2023</i>	54
Gambar 4.28	Hasil Pencarian <i>Keyword Bitcoin Naik</i>	55
Gambar 4.29	Hasil Pencarian <i>Keyword World Cup</i>	56
Gambar 4.30	Hasil Pencarian <i>Keyword Teknologi</i> Salah Ketik	57
Gambar 4.31	Hasil Pencarian <i>Keyword Resesi</i> Salah Ketik	58



DAFTAR TABEL

Tabel 1.1	Persentase <i>global market share search engine</i> Februari 2022 (StatCounter, 2022)	2
Tabel 2.1	<i>Array I</i> dari IGST-2 pada gambar 2.8 (Hon dkk., 2010)	18
Tabel 3.1	Deskripsi Tabel <i>Page Information</i>	30
Tabel 3.2	Deskripsi Tabel <i>Linking</i>	30
Tabel 3.3	Deskripsi Tabel <i>Style Resource</i>	30
Tabel 3.4	Deskripsi Tabel <i>Script Resource</i>	30
Tabel 3.5	Deskripsi Tabel <i>Forms</i>	31
Tabel 3.6	Deskripsi Tabel <i>Images</i>	31
Tabel 3.7	Deskripsi Tabel <i>List</i>	31
Tabel 3.8	Deskripsi Tabel <i>Tables</i>	31
Tabel 4.1	<i>Input</i> yang digunakan	51
Tabel 4.2	Hasil Pengujian <i>Mean Average Precision</i> Mengenai <i>Relevansi Query</i>	58
Tabel 1.1	Contoh Kuesioner Penilaian Relevansi Hasil Menggunakan Metode <i>Mean Average Precision</i>	65
Tabel 2.1	Tabel Penilaian Relevansi Hasil Menggunakan Metode <i>Mean Average Precision</i> Reza Nurdiansyah Firdaus	67
Tabel 3.1	Tabel Penilaian Relevansi Hasil Menggunakan Metode <i>Mean Average Precision</i> Rachel H	69
Tabel 4.1	Tabel Penilaian Relevansi Hasil Menggunakan Metode <i>Mean Average Precision</i> Lazuardy Khatulistiwa	71

BAB I

PENDAHULUAN

A. Latar Belakang Masalah

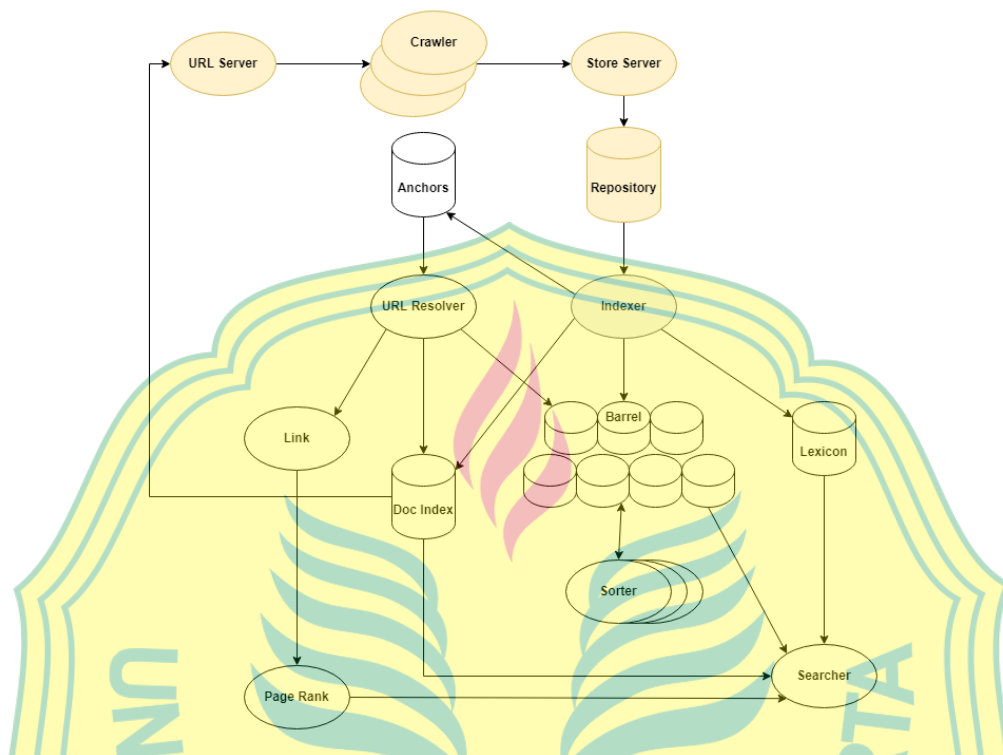
Web Search Engine adalah program perangkat lunak yang melakukan pencarian di internet yang memiliki banyak *website* dan bekerja berdasarkan kata kunci atau *query words* yang kita tentukan. *Search engine* mencari kata kunci yang kita tentukan di dalam *database* atau basis data informasi yang dimiliki. Alan Emtage beserta dua orang temannya, yaitu Bill Heelan dan J. Peter Deutsch memperkenalkan *search engine* pertama kali pada tahun 1990 dengan nama *Archie*. *Archie* mengunduh daftar direktori semua *file* yang terletak di situs anonim publik FTP (*File Transfer Protocol*) lalu membuat *database* nama *file* yang dapat dicari. Namun, *Archie* tidak mengindeks isi situs-situs tersebut karena jumlah datanya sangat terbatas sehingga dapat dengan mudah dicari secara manual. (Seymour dkk., 2011).

Di era modern ini, kegiatan kita sehari-hari tak lepas dari internet. *Search Engine* sendiri erat kaitannya dengan pengguna internet. Secara tidak langsung kita pasti menggunakan *search engine* ketika berselancar di internet. Meskipun aplikasi ponsel dengan berbagai tujuan sudah menjamur, peran peramban atau *browser* dan *search engine* masih cukup penting dalam melakukan pencarian informasi. Sistem pencarian, pemerinkatan, dan pengindeksan modern yang didukung oleh daya komputasi yang ditingkatkan, kecepatan jaringan yang cepat, dan kapasitas penyimpanan data yang hampir tak terbatas menjadikan kita memiliki akses mudah ke semua informasi yang kita butuhkan. Prinsip-prinsip yang menjadi dasar teknologi modern ini sudah ada dari sebelum tahun 1960-an. (Harman dkk., 2019). Berikut persentase *global market share* atau pangsa pasar dari beberapa *search engine* per Februari 2022.

Tabel 1.1: Persentase *global market share search engine* Februari 2022 (StatCounter, 2022)

Search engine	Market share (%)
Google	92.01
Bing	2.96
Yahoo!	1.51
Baidu	1.17
Yandex	1.06
DuckDuckGo	0.68

Dapat kita lihat dari data di atas, *Google* masih menjadi *search engine* favorit bagi para pengguna internet untuk saat ini. *Google* sendiri ditemukan dan diperkenalkan pada tahun 1998 oleh Larry Page dan Sergey Brin. Salah satu keunggulan *Google* adalah pengaplikasian *PageRank*. Manfaat terbesar dari *PageRank* adalah kehebatannya dalam mengatasi *underspecified queries* atau kueri yang kurang spesifik. Sebagai contoh, pada mesin pencari konvensional jika kita mencari "Universitas Stanford" maka hasil pencariannya dapat mengembalikan sejumlah halaman web yang mencantumkan Stanford di dalamnya. Jika menggunakan *PageRank*, maka halaman web Universitas Stanford akan menjadi halaman pertama yang dikembalikan. *Google* sebagai *search engine* terdiri dari beberapa komponen utama. Gambar 1.1 menunjukkan komponen-komponen tersebut yang termuat dalam *High Level Google Architecture*. (Brin dkk., 1998).



Gambar 1.1: *High Level Google Architecture* (Brin dan Page, 1998)

Pada penelitian sebelumnya yang dilakukan oleh Muhammad Fathan Qoriiba yang berjudul "Perancangan *Crawler* Sebagai Pendukung Pada *Search Engine*", rangkaian komponen *crawler* berhasil dibuat dan ditandai dengan warna krem pada gambar 1.1. Untuk membuat *search engine*, mulanya dibutuhkan sebuah *crawler*. *Crawler* merupakan sebuah program untuk mengambil informasi yang ada di halaman web. Dibutuhkan perancangan *crawler* yang baik untuk mendapatkan hasil yang baik. Algoritma yang digunakan mengikuti algoritma awal perkembangan *Google*, yaitu *breadth first search crawling* dan *modified similarity based crawling*. *Crawler* yang dibuat dapat dijalankan untuk situs web statis yang dibuat menggunakan *html* versi 5 atau 4. Hasil *crawling* disimpan di dalam *database MySQL*. (Qoriiba, 2021).

Rangkaian komponen *crawler* yang dibuat menghasilkan dataset berisi 8 tabel yang berukuran 176 MiB atau sekitar 185 MB. Dataset ini disusun menggunakan *tools crawling* yang dikembangkan. Situs yang digunakan untuk proses *crawling* adalah situs *indosport.com* dan *curiouscuisiniere.com*. Dataset ini akan digunakan untuk kelanjutan penelitian *search engine*. (Qoriiba, 2021).

Salah satu komponen lain yang merupakan kelengkapan dari arsitektur

search engine adalah pengindeks atau *indexer*. *Indexer* melakukan beberapa fungsi penting sebagai salah satu komponen arsitektur *search engine*. *Indexer* membaca repositori, membuka kompresi dokumen, dan menguraikan dokumen tersebut. *Indexer* lalu mendistribusikan kumpulan kata yang disebut *hits* ke dalam satu set *barrels* yang akan membuat sebagian *forward index* teratur. *Indexer* juga menguraikan semua tautan pada halaman web dan akan menyimpan semua informasi penting ke dalam *anchor file*. (Brin dan Page, 1998).

Algoritma yang terkait dengan *indexing* dan *document retrieval* sendiri telah diteliti oleh cukup banyak orang sebelumnya. Salah satu pelopornya yaitu S. Muthukrishnan pada tahun 2002 dengan memanfaatkan penggunaan *Generalized Suffix Tree*. Algoritma Muthukrishnan menekankan pada pengambilan semua dokumen yang memiliki jumlah kemunculan pola input *query* jika melebihi ambang batas tertentu. Ambang batas tersebut ditentukan oleh pencari informasi atau dengan kata lain oleh *user*. (Hon dkk., 2010).

Pada penelitian yang berjudul "*Privacy-preserving string search on encrypted genomic data using a generalized suffix tree*", penggunaan *Generalized Suffix Tree* untuk pengindeksan memberikan peningkatan yang signifikan terhadap waktu eksekusi *query*. Struktur indeks dari *generalized suffix tree* ini digunakan untuk membuat sebuah protokol yang aman dan efisien dalam melakukan pencarian *substring* dan set-maksimal pada data genom. Hasil eksperimen menunjukkan bahwa metode yang diajukan pada penelitian ini dapat mengkomputasi pencarian *substring* dan set-maksimal yang aman pada sebuah dataset *single-nucleotide polymorphism* (SNPs) dengan 2184 *records* (setiap *record* berisi 10000 SNPs) dalam 2.3 dan 2 detik. (Mahdi dkk., 2021).

Pada penelitian yang berjudul "*Parallel and private generalized suffix tree construction and query on genomic data*", konstruksi *Generalized Suffix Tree* secara paralel untuk pengindeksan dapat digunakan untuk data genom yang besar. GST menyediakan indeks yang efisien untuk data genomik yang dapat digunakan dalam banyak *query* penelusuran berbasis string, yang fundamental bagi kegunaan data genom. Konstruksi GST secara paralel dikerjakan secara terdistribusi dan terbagi. (Al Aziz dkk., 2022).

Penggunaan GST atau *Generalized Suffix Tree* sebagai struktur data pengindeksan juga dilakukan dalam penelitian yang berjudul "*Full-Text Search on Data with Access Control using Generalized Suffix Tree*". Dalam penelitian ini, alih-alih menggunakan *inverted index* yang mana lazim digunakan untuk pencarian

full-text, *Generalized Suffix Tree* digunakan karena kemampuannya dalam melakukan pencarian *substring* dalam dokumen. Hasil penelitian ini menunjukkan bahwa penggunaan GST memerlukan memori yang besar bahkan melebihi jumlah dokumen tersebut. Namun, pencarian menggunakan GST 3 kali lebih cepat dibandingkan *inverted index*. (Zaky dan Munir, 2016).

Wing-Kai Hon dan beberapa temannya berhasil mengembangkan algoritma baru dengan pemanfaatan dari *Generalized Suffix Tree* yang lebih efisien dari segi ruang dan waktu. Algoritma ini digunakan untuk mendapatkan indeks yang efisien dalam masalah pencarian dokumen. Pendekatan ini didasarkan pada penggunaan baru dari pohon sufiks atau *suffix tree* yang dinamakan *induced generalized suffix tree (IGST)*. (Hon dkk., 2010)

Di dalam penelitian ini, penulis akan membuat modul pengindeks atau *indexer* sebagai salah satu komponen arsitektur *search engine* yang ditandai dengan warna biru muda pada gambar 1.1. Penulis akan membuat *indexer* menggunakan algoritma yang sudah dikembangkan oleh Wing-Kai Hon dan teman-temannya. Dataset yang akan digunakan sebagai kumpulan dokumen adalah dataset hasil *crawling* yang sudah dibuat oleh Muhammad Fathan Qoriiba sebelumnya. Penelitian ini sekaligus akan melanjutkan rangkaian penelitian sebelumnya terkait *search engine*.

B. Rumusan Masalah

Berdasarkan uraian pada latar belakang yang ada di atas, maka rumusan masalah pada penelitian ini adalah “Bagaimana cara mengindeks dan melakukan pencarian dokumen menggunakan algoritma *Efficient Index For Retrieving Top-k Most Frequent Document?*”

C. Pembatasan Masalah

Pembatasan masalah pada penelitian ini adalah pembuatan sebagian arsitektur *search engine* yaitu modul pengindeks atau *indexer*. Kumpulan dokumen yang akan digunakan sebagai dataset adalah dataset yang telah disusun oleh Muhammad Fathan Qoriiba dalam penelitiannya yang berjudul PERANCANGAN CRAWLER SEBAGAI PENDUKUNG PADA *SEARCH ENGINE*. Judul dari halaman web akan dijadikan bahan pembentukan *tree*.

D. Tujuan Penelitian

Adapun tujuan dari penelitian ini adalah membuat modul pengindeks atau *indexer* yang akan digunakan untuk kebutuhan efisiensi pencarian yang termasuk dalam pemeringkatan dokumen pada *search engine*.

E. Manfaat Penelitian

Penelitian ini diharapkan dapat mengoptimisasi pencarian pada *search engine* dari segi waktu dan ruang serta relevansi pencarian. Penelitian ini juga diharapkan memberikan manfaat lainnya, antara lain:

1. Bagi penulis

Menambah ilmu dan pengetahuan mengenai *document retrieval* khususnya pada bagian *search engine* dan *indexing*, mempraktikkan ilmu-ilmu yang sudah didapat semasa kuliah serta mendapatkan gelar Sarjana Komputer.

2. Bagi Program Studi Ilmu Komputer

Penelitian ini merupakan salah satu rangkaian dan lanjutan dari penelitian sebelumnya mengenai *search engine*. Penelitian ini dapat memberikan gambaran dan informasi mengenai *search engine* khususnya bagian *indexer* dan dapat dijadikan acuan untuk penelitian selanjutnya terkait *search engine*.

3. Bagi Universitas Negeri Jakarta

Menjadi acuan untuk evaluasi dan pertimbangan serta pengembangan kualitas akademik di Universitas Negeri Jakarta khususnya Program Studi Ilmu Komputer.

BAB II

KAJIAN PUSTAKA

A. *Search Engine*

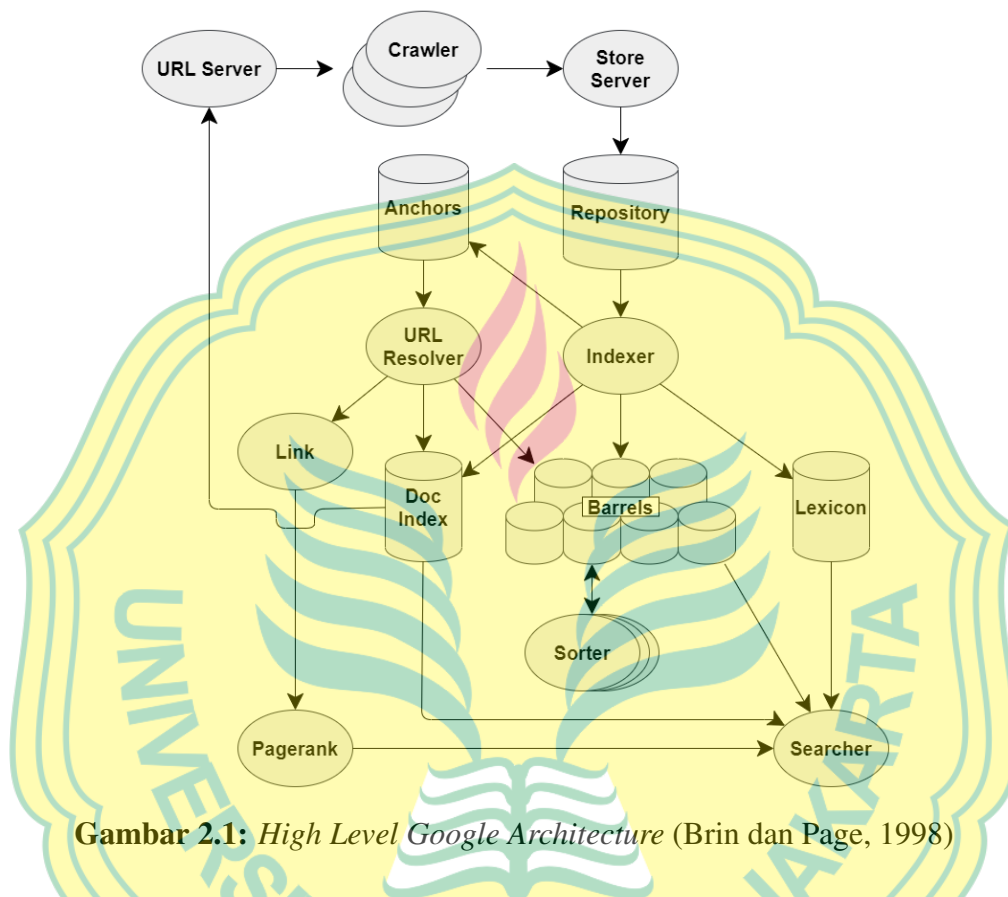
Search engine merupakan contoh dari *Information Retrieval System* berskala besar. Sejarah *search engine* bisa dibilang dimulai dari tahun 1990 ketika Alan Emtage dan kawan-kawannya membuat *Archie*. Setelah *Archie*, banyak *search engine* lain yang mulai diperkenalkan. *Gopher* dan *Veronica and Jughead* diperkenalkan pada tahun 1991. Pada tahun 1993, *W3Catalog and Wanderer*, *Aliweb*, dan *Jump Station* diperkenalkan ke publik. Menyusul di tahun 1994 ada *WebCrawler* dan pada tahun 1995 *MetaCrawler*, *AltaVista*, serta *Excite* diperkenalkan. (Seymour dkk., 2011).

AltaVista menoreh popularitas yang sangat tinggi pada waktu itu sebelum akhirnya digeser oleh kenaikan popularitas *Google*. Salah satu fitur unggulan yang ditawarkan oleh *AltaVista* adalah adanya *natural language search*. Hal ini memungkinkan pengguna untuk mencari informasi dengan mencari frase atau dengan kalimat tanya. Sebagai contoh, kita bisa mencari informasi letak kota London dengan menanyakan "*Where is London?*" dan mendapatkan jawaban yang relevan dengan tidak menyertakan kata "*where*" dan "*is*" dalam hasil pencarian. (Seymour dkk., 2011).

Google sendiri resmi diperkenalkan pada tahun 1998 oleh Sergey Brin dan Larry Page. Kesuksesan *Google* pada waktu itu sebagian besar adalah andil dari algoritma *PageRank* yang telah dipatenkan. *PageRank* sendiri bertugas untuk melakukan pemeringkatan atau *me-ranking* halaman web agar hasil informasi yang dicari oleh pengguna lebih relevan. Setelah *Google*, ada cukup banyak *search engine* yang diperkenalkan ke publik. Beberapa yang mungkin kita tahu adalah *Yahoo! Search* dan *Bing!*. *Yahoo! Search* diperkenalkan oleh *Yahoo! Inc.* pada tahun 2004 sedangkan *Bing!* diperkenalkan oleh *Microsoft* pada tahun 2009. (Seymour dkk., 2011). Sampai saat ini, *Google* masih memimpin dalam hal jumlah pengguna dan popularitas.

Sergey Brin dan Larry Page memperkenalkan arsitektur *Google* sebagai sebuah *search engine* yang dinamakan dengan *High Level Google Architecture*. Gambar 2.1 menunjukkan komponen-komponen penyusun *search engine*. (Brin dan

Page, 1998).



Gambar 2.1: *High Level Google Architecture* (Brin dan Page, 1998)

Web crawling dilakukan oleh beberapa *crawler*. *URLServer* mengirim daftar URL untuk diambil datanya oleh *crawler*. Halaman web yang sudah diambil datanya dikirim ke dalam *Store Server* lalu dikompresi dan dikirim ke dalam *Repository*. Setiap halaman web memiliki nomor identifikasi yang disebut dengan *docID*. Proses penomoran ini dilakukan ketika URL diambil dari halaman web. Selanjutnya proses pengindeksan dilakukan oleh *Indexer* dan *Sorter*. *Indexer* membaca repository, membuka kompresi dokumen, dan menguraikan dokumen tersebut. *Indexer* lalu mendistribusikan kumpulan kata yang disebut *hits* ke dalam satu set *barrels* yang akan membuat sebagian *forward index* terurut. *Indexer* juga menguraikan semua tautan pada halaman web dan akan menyimpan semua informasi penting ke dalam *anchor file*. *Anchor file* berisi informasi yang berguna untuk menentukan tautan tersebut merujuk ke mana dan juga berisi teks dari tautan tersebut.

Anchor file kemudian dibaca oleh *URLResolver* dan diubah URL nya dari URL relatif ke URL absolut serta pengubahan menjadi *docID*. *URLResolver* kemudian mengirim teks yang ada di dalam *anchor file* ke *forward index*, yang mana

berhubungan dengan *docID* yang sudah dirujuk oleh *anchor file* sebelumnya. *URLResolver* juga membuat basis data yang berisi tautan yang merupakan pasangan dari *docID*-nya masing-masing. Basis data ini digunakan untuk menghitung *PageRank* dari seluruh dokumen.

Sorter lalu mengambil *barrels* yang sudah diurutkan berdasarkan *docID* dan mengurutkannya kembali berdasarkan *wordID* untuk membuat *inverted index*. *Sorter* juga membuat daftar yang berisi *wordID* dan *offset* ke dalam *inverted index*. *DumpLexicon* mengambil daftar tadi beserta *lexicon* yang dihasilkan oleh *Indexer* lalu membuat *lexicon* baru untuk digunakan oleh *Searcher*. *Searcher* dijalankan oleh web server dan menggunakan *lexicon* serta *inverted index* dan juga *PageRank* untuk menjawab *query* atau pertanyaan. (Brin dan Page, 1998).

B. Document Retrieval

Dalam basis data yang berisi teks atau *string*, kita memiliki kumpulan beberapa dokumen yang berisi teks atau *string* alih-alih hanya satu dokumen *string* saja. Dalam kasus ini, masalah dasarnya adalah mengambil semua dokumen di mana pola *query P* berada. Masalah ini disebut juga masalah pengambilan dokumen atau *document retrieval problem*. (Hon dkk., 2010). Masalah ini sudah pernah diteliti sebelumnya oleh S. Muthukrishnan.

Muthukrishnan memperkenalkan algoritma pertama yang dikenal untuk menyelesaikan *document listing problem*. Dalam *document listing problem*, diberikan sebuah *query online* yang terdiri dari pola string *p* atau *query pattern p* dengan panjang *m*. Hasil akhirnya adalah pengembalian kumpulan-kumpulan dokumen yang mengandung satu atau lebih pola *query p*. *Document listing problem* merupakan bagian dari *document retrieval problem*. (Muthukrishnan, 2002). Diberikan sebuah kumpulan *string*, *document listing* mengacu pada masalah dari menemukan semua *string* atau dokumen di mana *query string* muncul. (Puglisi dan Zhukova, 2021).

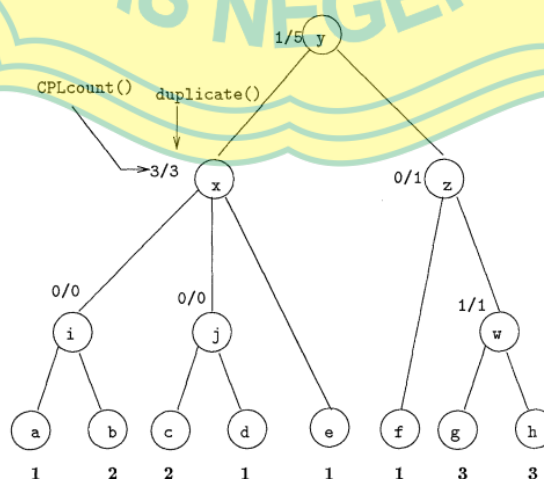
Permasalahan utama dari algoritma Muthukrishnan ini adalah bisa terjadi banyak kemunculan pola *query* lain pada seluruh koleksi dokumen. Di sisi lain, jumlah keseluruhan dokumen yang mengandung pola *query* yang dicari bisa saja jauh lebih kecil. Maka dari itu, metode yang mencari seluruh kemunculan pola *query* lalu melaporkan dokumen mana saja yang mengandung pola tersebut bisa terbilang jauh dari kata efisien. (Hon dkk., 2010).

Muthukrishnan juga memperkenalkan varian lain pada penelitiannya. Dalam varian ini, dokumen yang dikembalikan hanya dokumen yang memiliki lebih dari f kemunculan pola. Varian ini disebut *f-mine problem*. Dari segi pencarian informasi, varian ini lebih menarik karena hanya dokumen dengan relevansi yang paling tinggi yang dikembalikan. (Hon dkk., 2010).

Pada penelitian Wing-Kai Hon dan teman-temannya, masalah yang diteliti erat kaitannya dengan *f-mine problem*. Mereka langsung mencari *top-k* dokumen yang memiliki jumlah kemunculan maksimum untuk pola yang diberikan. Mereka menambahkan sesuatu yang bekerja seperti *inverse document mine query*, yang dengan cepat memungkinkan mereka menemukan ambang f_k (atau cukup f ketika konteksnya jelas) yang merupakan frekuensi pola dalam frekuensi dokumen ke- k . Berdasarkan ini kita bisa melihat hubungan yang kuat dengan *f-mine problem*. (Hon dkk., 2010).

Komponen utama dari solusi Wing-Kai Hon dan kawan-kawannya adalah *induced generalized suffix tree (IGST)*. IGST sendiri kurang lebih memiliki struktur yang sama dengan indeks yang diperkenalkan oleh Muthukrishnan. Namun, terdapat pengaplikasian baru pada IGST di mana IGST itu sendiri dilinearisasi dan dikombinasikan dengan fungsionalitas dari pencari selanjutnya untuk mendapatkan hasil yang diinginkan. (Hon dkk., 2010).

C. Masalah *Color Set Size*



Gambar 2.2: CSS Example (Chi dan Hui, 1992)

Diketahui $C = \{1, \dots, m\}$ adalah himpunan warna dan T adalah pohon berakar yang memiliki n daun di mana setiap daun memiliki warna c , $c \in C$. *Color Set Size* untuk *vertex* v , yang dinotasikan sebagai $css(v)$, adalah jumlah warna daun berbeda dalam subpohon yang berakar pada v . Masalah *Color Set Size* adalah untuk mencari *color set size* pada setiap *internal vertex* v di dalam T . Sebagai contoh pada gambar 2.2, $css(x) = 2$ dan $css(y) = 3$. (Chi dan Hui, 1992).

LeafList adalah daftar daun yang sudah terurut berdasarkan *post-order traversal* dari pohon T . Untuk v yang merupakan daun dari T yang memiliki warna c , $lastleaf(x)$ adalah daun terakhir yang memiliki warna sama yaitu c yang mendahului x pada *LeafList*. Jika tidak ada daun yang mendahului x yang memiliki warna sama dengan x maka $lastleaf(x) = Nil$. Pada gambar 2.2, $lastleaf(a)$ dan $lastleaf(b)$ adalah *Nil* karena tidak ada daun sebelumnya yang memiliki warna sama dengan daun a atau b . Untuk $lastleaf(c)$ adalah b karena b memiliki warna yang sama dengan c dan mendahului c . (Chi dan Hui, 1992).

Untuk *vertex* atau *node* x dan y dari T , $lca(x,y)$ adalah *least common ancestor* dari x,y atau *parent* yang paling tidak sama dari x,y . Subpohon dari T yang berakar pada *node* u disebut dengan *subtree*(u) dan $leafcount(u)$ adalah jumlah seluruh daun dari *subtree*(u). Sebuah *internal node* u dikatakan *color-pair-lca* atau CPL jika ada daun x yang memiliki warna c , di mana $u = lca(lastleaf(x),x)$. Sebagai contoh pada gambar 2.2, *vertex* w adalah CPL dari warna 3. Sebuah *node* bisa menjadi CPL dari beberapa warna. $CPLCount(u) = k$ jika dari semua warna, terdapat k kemunculan dari pasangan daun di mana u merupakan CPL dari pasangan daun tersebut.

$$duplicate(u) = \sum_{v \in subtree(u)} CPLCount(v) = CPLCount(u) + \sum_{w \in W} duplicate(w)$$

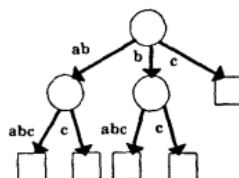
di mana W adalah himpunan anak dari u . (Chi dan Hui, 1992).

D. Generalized Suffix Tree

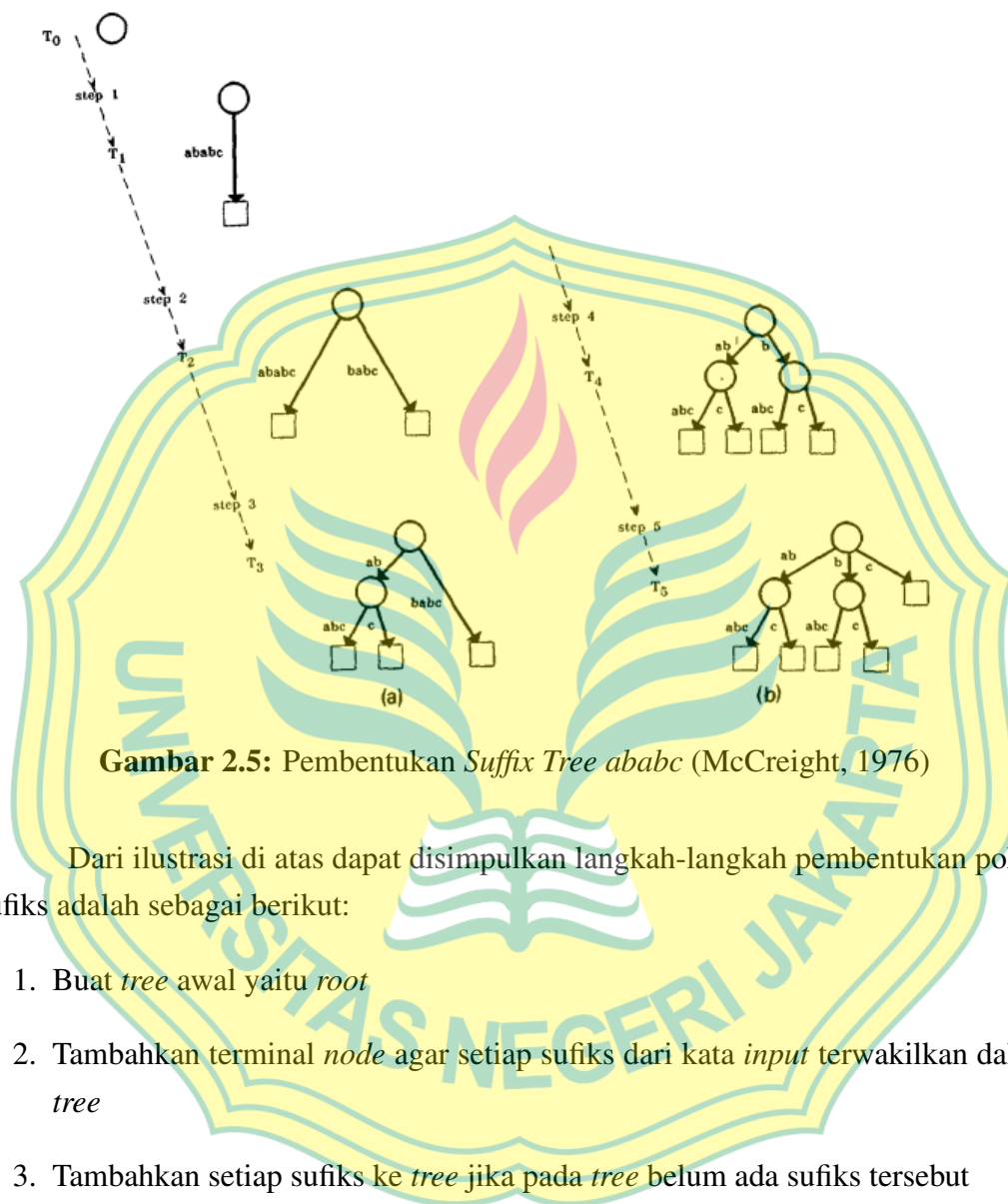
Diketahui $\Delta = \{T_1, T_2, \dots, T_m\}$ menunjukkan satu set dokumen. Setiap dokumen adalah *string* karakter dengan karakter yang diambil dari \sum alfabet umum yang ukurannya $|\sum|$ bisa tak terbatas. Kita asumsikan untuk setiap i , karakter terakhir dari dokumen T_i ditandai dengan $\$,$ yang mana bersifat *unique* untuk semua karakter pada semua dokumen. *Generalized Suffix Tree (GST)* untuk Δ adalah sebuah *compact trie* yang menyimpan semua sufiks dari dokumen T_i . Lebih tepatnya, setiap sufiks dari setiap dokumen sesuai dengan daun yang berbeda di

Gambar 2.3: Generalized Suffix Tree (Hon dkk., 2010)

Sebuah pohon sufiks dari *string* $W = w_1, \dots, w_n$ adalah sebuah pohon dengan tepi $O(n)$ dan n daun. Setiap daun di dalam pohon sufiks terhubung dengan indeks i $1 \leq i \leq n$. Tepi-tepinya dilabeli dengan karakter sehingga rangkaian tepi berlabel dari akar ke daun dengan indeks i adalah sufiks ke- i dari W . Sebuah pohon sufiks untuk *string* dengan panjang n dapat dibuat dalam $O(n)$ ruang dan waktu. Sebagai contoh *string* *ababc* akan disusun seperti gambar 2.4. (McCreight, 1976).



Proses pembentukan *suffix tree* *ababc* digambarkan pada gambar di bawah ini:



Dari ilustrasi di atas dapat disimpulkan langkah-langkah pembentukan pohon sufiks adalah sebagai berikut:

1. Buat *tree* awal yaitu *root*
2. Tambahkan terminal *node* agar setiap sufiks dari kata *input* terwakilkan dalam *tree*
3. Tambahkan setiap sufiks ke *tree* jika pada *tree* belum ada sufiks tersebut
4. Jika prefiks dari sufiks yang akan ditambahkan sudah ada maka penambahan sufiks ke *tree* menyesuaikan dengan menjadikan sufiks sebagai anak dari *prefiks* tersebut

Konsep dari pohon sufiks bisa dikembangkan untuk menyimpan lebih dari satu masukan *string*. Pengembangannya disebut juga dengan *Generalized Suffix Tree*. Terkadang dua atau lebih *input string* bisa memiliki sufiks yang sama, dalam kasus ini GST akan memiliki dua daun yang berhubungan dengan sufiks yang sama, setiap daun berkaitan dengan sebuah *input string* yang berbeda. Sebuah GST bisa dibangun

dalam $O(n)$ ruang dan waktu di mana n adalah total jumlah seluruh *input string*. (Chi dan Hui, 1992).

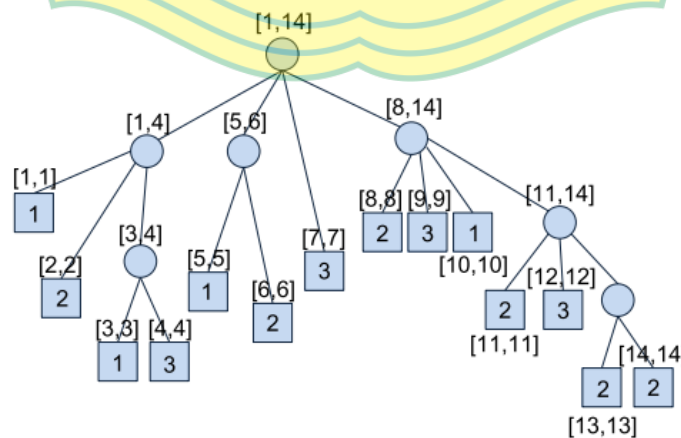
Properti dari GST adalah sebagai berikut:

1. Setiap sufiks dari setiap *input string* diwakilkan oleh sebuah daun di dalam GST
2. Panjang dari tepi berlabel di dalam pohon sufiks dapat dicari dalam $O(1)$ waktu
3. Jika sebuah daun pada pohon sufiks mewakili sebuah string V , maka setiap sufiks dari V juga diwakilkan oleh sebuah daun lain di dalam pohon sufiks
4. Jika sebuah *node* u adalah leluhur dari *node* v , maka *string* yang u wakili adalah *prefix* dari *string* yang diwakili oleh v
5. Jika *node* v_1, v_2, \dots, v_k mewakili *string* V_1, V_2, \dots, V_k , maka *least common ancestor* dari v_1, \dots, v_k merepresentasikan *longest common prefix* dari V_1, \dots, V_k

(Chi dan Hui, 1992).

2. Suffix Range

Bayangkan ada sebuah subpohon atau *subtree* dari *node* u di dalam GST. Diketahui v dan w adalah daun paling kiri dan daun paling kanan dari subpohon ini. Jika l mewakili urutan dari v dan r mewakili urutan dari w , maka v adalah daun paling kiri ke- l dan w adalah daun paling kanan ke- r di dalam GST. Jangkauan $[l, r]$ adalah yang kita sebut *suffix range* dari u . (Hon dkk., 2010).



Gambar 2.6: Suffix Range (Hon dkk., 2010)

3. *Optimal Index for Colored Range Query*

Diketahui $A[1..n]$ adalah *array* dengan panjang n , di mana setiap entri menyimpan warna yang diambil dari $C = \{1, 2, \dots, c\}$. Sebuah *colored range query* atau $CRQ(i, j)$, menerima 2 masukan integer yaitu i dan j di mana $1 \leq i \leq j \leq n$, dan memberikan hasil berupa *subarray* $A[i..j]$. Sebagai contoh, anggap A memiliki 7 entri, yang diwarnai dengan 1, 3, 2, 6, 2, 4, dan 5. Maka, $CRQ(2, 5)$ akan meminta set warna dari *subarray* $A[2..5]$ dan akan mengembalikan hasil 2,3,6. Indeks untuk *array* A dapat dibuat menggunakan $O(n)$ ruang penyimpanan, sehingga untuk i dan j apa pun, *colored range query* $CRQ(i, j)$, dapat dijawab dalam $O(\gamma)$ waktu, di mana γ menunjukkan jumlah warna berbeda pada set *output*. (Hon dkk., 2010).

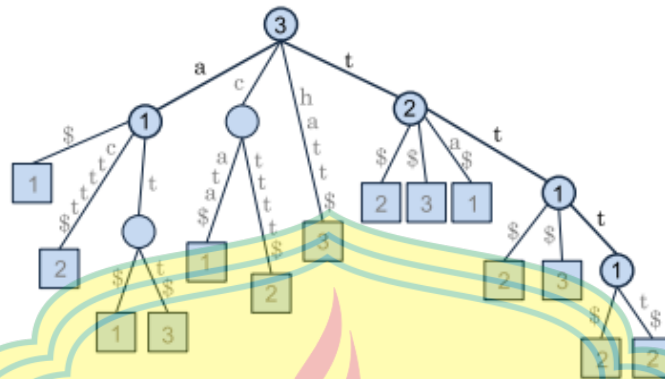
4. *Y-fast Trie for Efficient Successor Query*

Misalkan S adalah himpunan n bilangan bulat berbeda yang diambil dari $[1, D]$. Diberikan masukan x , *query* penerus atau *successor query* pada S mengembalikan bilangan bulat terkecil di S yang lebih besar dari atau sama dengan x . Sebuah index dari himpunan S yang tersusun dari n bilangan bulat dapat dibuat menggunakan $O(n)$ ruang penyimpanan, sehingga untuk setiap masukan x , *query* penerus $succ(S, x)$ dapat dijawab dengan $O(\log \log D)$ waktu, di mana D menunjukkan semesta tempat bilangan bulat S dipilih. Indeksnya bisa dibangun dalam waktu acak $O(n \log D)$. (Hon dkk., 2010).

E. *Induced Generalized Suffix Tree*

1. *IGST-f: IGST Untuk Frekuensi f*

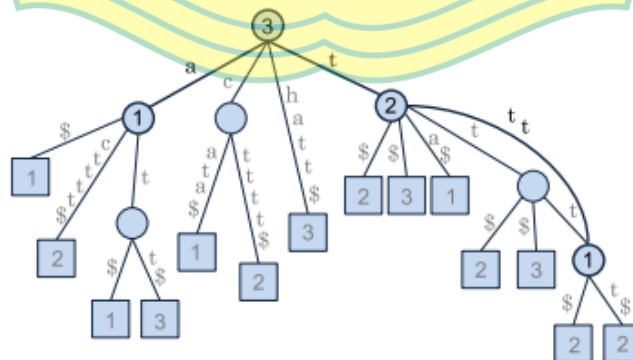
Misalkan ada GST untuk Δ dan sebuah bilangan bulat f dengan $1 \leq f \leq D$. Anggap setiap daun pada GST diberi label original dari sufiks yang sesuai. Untuk setiap *internal node* v dari GST, kita katakan v adalah f -frequent jika di dalam *subtree* yang berakar pada v , setidaknya ada f daun yang memiliki label yang sama. Subpohon terinduksi dari GST atau *induced subtree* yang dibentuk dengan mempertahankan semua *node* frekuensi- f disebut *pre-IGST-f*. Pada gambar 2.7 semua *internal node* yang memiliki setidaknya 2 daun dengan label yang sama ditandai dengan warna lebih terang atau *highlight*. Subpohon terinduksi yang terbentuk oleh *node* yang disorot ini adalah *pre-IGST-f*. (Hon dkk., 2010).



Gambar 2.7: *Pre-IGST-2* (Hon dkk., 2010)

Misalkan v adalah sebuah *node* di dalam *pre-IGST-f* dan v adalah sebuah *internal node* di dalam GST tersebut. Istilah $\text{count}(f, v)$ digunakan untuk menyatakan jumlah dokumen berbeda yang memiliki label f dalam subpohon yang berakar pada v di GST. Istilah $\text{count}(v)$ digunakan ketika konteksnya jelas. Pada *pre-IGST-2* yang ditunjukkan pada gambar 2.7, setiap *internal node* v diberi label dengan nilai $\text{count}(v)$ yang sesuai. Nilai $\text{count}(v)$ harus berada di antara 1 dan m , di mana m adalah jumlah dokumen dalam Δ . (Hon dkk., 2010).

Misalkan ada sebuah *pre-IGST-f*. Untuk setiap *internal node* v , kita katakan v redundan apabila v adalah *node* derajat-1 atau *degree-1 node* dan $\text{count}(v) = \text{count}(\text{child}(v))$ di mana $\text{child}(v)$ menyatakan *unique child* dari v . Pohon induksi atau *induced tree* yang dibentuk dengan menyusutkan atau mereduksi semua *node* redundan pada *pre-IGST-f* disebut *IGST-f*. (Hon dkk., 2010).



Gambar 2.8: *IGST-2* (Hon dkk., 2010)

Struktur dari *IGST-f* yang kita miliki ekuivalen dengan struktur indeks yang

diperkenalkan oleh Muthukrishnan yang digunakan untuk menyelesaikan masalah *document mining problem*. Di mana diberikan frekuensi f dan pola P , lalu kembalikan occ_f dokumen yang memiliki setidaknya f kemunculan dari P dalam $O(|P| + occ_f)$ waktu. Dengan menggunakan struktur ini, kita sudah bisa menjawab masalah *inverse mine* (P, k) query dengan melakukan *binary search* pada f . Di mana setiap langkah dari *binary search* memeriksa apakah f tertentu adalah jawaban yang diinginkan untuk *inverse mine* (P, k) . Perlu kita ketahui bahwa setiap langkah dari *binary search* hanya perlu mengkonfirmasi apabila $k \geq occ_f$, sehingga kita hanya membutuhkan $O(|P| + k)$ waktu. Dalam total keseluruhan, jawaban yang diinginkan dapat diambil dalam $O((|P| + k) \log D)$ waktu.

2. Representasi Array dari IGST- f

Untuk menjawab permasalahan *inverse mine query* yang harus dilakukan adalah menemukan *locus* atau lokasi dari pola P di setiap IGST selama melakukan *binary search*. Untuk memangkas waktu pencarian *locus* pada setiap langkah, penggunaan *suffix ranges* dilakukan. Pertama-tama, anggap *suffix range* dari *locus* P sudah dikomputasi. Diketahui $[l_p, r_p]$ mewakili *range* ini jika *locus*nya ada. Selanjutnya, anggap setiap *node* di dalam IGST- f terhubung dengan *suffix range* yang sesuai dengan *node* di dalam GST. Maka, *locus* dari P , jika ada, adalah *node* v sedemikian rupa sehingga *suffix range* yang berhubungan dengan keturunan dari v (termasuk v), adalah *subrange* dari $[l_p, r_p]$, dan *suffix range* yang berhubungan dengan *parent node* bukan merupakan *subrange* dari $[l_p, r_p]$. (Hon dkk., 2010).

Selanjutnya, lakukan *pre-order traversal* pada IGST- f dan hitung *suffix range* yang terhubung dengan sebuah *node* selagi *node* itu dikunjungi. Diketahui $[l(z), r(z)]$ mewakili *suffix range* dari *node* ke- z yang sudah dihitung selama proses *traversal*. Sekarang, anggap *locus* dari P dalam IGST- f (asumsikan ada) adalah *node* ke- j di dalam *pre-order traversal* dari IGST- f . Karena itu, *locus* dari P di dalam IGST- f memiliki *suffix range* yang berhubungan yang dinotasikan sebagai $[l(j), r(j)]$. Lebih lanjut, anggap kita menguji $[l(z), r(z)]$ untuk beberapa z . Ingat bahwa $[l_p, r_p]$ merupakan *suffix range* dari *locus* P di dalam GST. Maka, pernyataan di bawah ini adalah benar dan mengandung semua kemungkinan hubungan antara $l_p, r_p, l(z)$, dan $r(z)$. (Hon dkk., 2010).

1. Jika $r_p < l(z)$, maka $j < z$;
2. Jika $l_p > r(z)$, maka $j > z$;

3. Jika $[l(z), r(z)]$ adalah *subrange* dari $[l_p, r_p]$, maka $j \leq z$;
4. Jika $[l_p, r_p]$ adalah *subrange* dari $[l(z), r(z)]$, maka $j \geq z$.

Diketahui $c(z)$ mewakili nilai *count* dari *node* ke- z yang dikunjungi selama masa *pre-order traversal* dari *IGST-f*. Alih-alih menyimpan *IGST-f* sebagai struktur pohon, kita akan merepresentasikannya ke dalam *array I*. Sehingga entri ke- z dari I , $I[z]$, menyimpan 3-tuple yaitu $(l(z), r(z), c(z))$. Sebagai contoh, *array I* dari *IGST-2* pada gambar 2.8 adalah sebagai berikut:

Tabel 2.1: *Array I* dari *IGST-2* pada gambar 2.8 (Hon dkk., 2010)

k	1	2	3	4
I	(1, 14, 3)	(1, 4, 1)	(8, 14, 2)	(13, 14, 1)

Berdasarkan teorema sebelumnya, kita bisa mendapatkan nilai j menggunakan *binary search* pada *array I*, sedemikian rupa sehingga $[l(j), r(j)]$ adalah *suffix range* yang terhubung dengan *locus P*. Lalu, nilai $c(j)$ menyimpan jumlah dokumen dengan setidaknya f kemunculan dari P . Jika *locus P* dalam *IGST-f* tidak ada, maka tidak ada z sedemikian rupa dengan $[l(j), r(j)]$ yang merupakan *subrange* dari $[l_p, r_p]$ dan *binary search* akan mendeteksi ini. Dikarenakan jumlah *nodes* dalam *IGST-f* adalah $O(D)$, panjang *array I* juga $O(D)$, maka *binary search* akan memakan waktu sebanyak $O(\log D)$ waktu. (Hon dkk., 2010).

Sejatinya, kita bisa mempercepat waktu *query* dengan cara mengganti setiap *binary search* dalam *array IGST* dengan sebuah *successor query* dalam sebuah *array* yang sedikit dimodifikasi. Akibatnya, waktu yang dihabiskan pada setiap kunjungan *IGST* akan dikurangi dari $O(\log D)$ menjadi $O(\log \log D)$. Langkah pertama, kita amati bahwa setiap *node* dalam *IGST* memiliki keterhubungan natural di dalam *GST*. Lebih tepatnya, sebuah *node u* dengan *suffix range* $[l, r]$ yang berada dalam *IGST* menunjukkan bahwa sebuah *node u'* yang memiliki *suffix range* yang sama juga ada dalam *GST* original. Selanjutnya, kita lakukan *pre-order traversal* di dalam *GST* original agar setiap *node v* menerima kunjungan $\alpha(v)$ pertama kali dan $\beta(v)$ terakhir kali selama proses *traversal*. Lalu, setiap *node u* dalam *IGST* ditambah dengan informasi $\alpha(u')$ dan $\beta(u')$ yang mana u' adalah korespondensinya sendiri di dalam *GST*. Setelah itu, untuk setiap *IGST*, kita kumpulkan sebuah kumpulan nilai α dari semua *node* dan menyimpan sebuah *y-fast trie* agar *successor query* pada nilai α dapat dijawab dalam $O(\log \log D)$ waktu. (Hon dkk., 2010).

Sekarang, untuk melakukan pencarian kita harus mendapatkan *locus P* di dalam GST original, katakanlah u_p , yang waktu *pre-order traversal*nya adalah $\alpha(u_p)$ dan $\beta(u_p)$. Karena waktu traversal memiliki *nested property* yang bagus, untuk mencari *locus P* di *IGST-f* setara dengan mencari *successor* dari $\alpha(u_p)$ dalam himpunan nilai α dari *IGST-f*. Lebih tepatnya, diketahui u adalah *node* dalam *IGST-f* dengan $\alpha(u)$ menjadi *successor* dari $\alpha(u_p)$. Mudah untuk membuktikan bahwa $\beta(u) \leq \beta(u_p)$ jika dan hanya jika *locus P* ada dalam *IGST-f* dengan u sebagai *locus*. Setelah *successor query* di atas, kita dapat memeriksa *3-tuple* yang sesuai l, r, c dari u untuk menentukan berapa banyak dokumen yang mengandung setidaknya f kemunculan dari P . Dengan menyimpan GST dan semua *array I* untuk semua *IGST-1, IGST-2, ..., IGST-D, inverse mine(P,k) query* dengan P adalah pola dan k adalah bilangan bulat dapat terjawab dalam $O(|P| + \log D \log \log D)$ waktu. (Hon dkk., 2010).

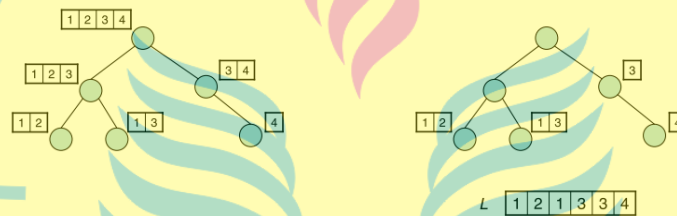
Seperti yang ditunjukkan oleh Muthukrishnan, jumlah *node* dalam *IGST-f* adalah $O(D/f)$. Secara singkat, setiap daun atau *node* derajat-1 di *IGST-f* sesuai dengan himpunan *disjoint* dari setidaknya f sufiks dari dokumen di δ , sehingga ada $O(D/f)$. Di sisi lain, jumlah *node* sisanya tidak bisa melebihi jumlah seluruh daun dan *node* derajat-1. Maka, jumlah seluruh *node* adalah $O(D/f)$. Total ruang yang digunakan oleh indeks Muthukrishnan dan Wing-Kai Hon jumlahnya sama yaitu $O(D \log D)$. (Hon dkk., 2010).

F. Indeks Efisien Untuk *Top-k Document Retrieval Problem*

Ketika kita sudah bisa menemukan indeks untuk menjawab *inverse mine query*, kita juga bisa mencari kumpulan dokumen dari *top document(P,k)* yang mana dokumen-dokumen k tersebut adalah dokumen yang memiliki kemunculan P paling banyak. Langkah-langkah menyelesaikan *top document(P,k)*:

1. Cari f^* di mana $f^* = \text{inverse mine}(P,k)$. Setidaknya ada k dokumen dengan setidaknya f^* kemunculan P , tetapi ada kurang dari k dokumen dengan setidaknya $f^* + 1$ kemunculan P .
2. Keluarkan semua dokumen dengan setidaknya $f^* + 1$ kemunculan P . Biarkan k' menjadi jumlah dari dokumen-dokumen tersebut.
3. Keluarkan $k-k'$ dokumen tambahan, berbeda dengan yang diperoleh pada langkah ke-2, yang mana memiliki setidaknya f^* kemunculan P .

Salah satu cara untuk menyelesaikan langkah 2 adalah dengan menambah setiap *node* v dari $IGST-f$ dengan daftar dokumen $count(v)$ terkait, masing-masing dari yang memiliki setidaknya f label di subpohon yang berakar pada v . Kemudian, mudah untuk melihat bahwa untuk menjawab langkah 2, kita hanya perlu mencari *locus* P dalam $IGST-(f^*+1)$ dan mengembalikan daftar dokumen dalam *locus*. Metode ini membutuhkan waktu optimal $O(k')$ untuk mengembalikan dokumen. Sayangnya, pada skenario terburuk, kita membutuhkan ruang tambahan untuk proses penambahan sebesar $O(Dm \log D)$ di mana m mewakili jumlah dokumen dalam Δ . Metode ini kita sebut dengan *Heuristic I*. (Hon dkk., 2010).



Gambar 2.9: Contoh dari penggabungan *list* L (Hon dkk., 2010)

Salah satu cara lain yang lebih baik untuk menyelesaikan langkah 2 adalah dengan menerapkan indeks Muthukrishnan yang digunakan untuk *colored range query* sebagai struktur data tambahan. Cara ini digunakan oleh Muthukrishnan dalam menyelesaikan *document mining problem*. Idennya adalah untuk setiap *node* v di $IGST-f$ kita hanya menyimpan *sublist* tereduksi dari dokumen $count(v)$ terkait, di mana setiap dokumen tersebut tidak muncul dalam daftar keturunan dari v di $IGST-f$. Selanjutnya, kita melakukan *pre-order traversal* dan menggabungkan daftar *node* yang telah dikunjungi ke dalam *list* L (lihat gambar 2.9). Kemudian, mudah untuk memeriksa bahwa setiap *node* v akan berkorespondensi ke *subrange* dalam *list* L , sehingga dokumen $count(v)$ yang terkait akan sesuai persis dengan $count(v)$ yang berbeda dokumen pada *subrange*. Sebagai contoh, perhatikan anak kiri dari akar di $IGST-f$ pada gambar 2.9, itu berkorespondensi dengan *subrange* $L[1...4]$, yang merupakan rangkaian dari *sublist* tereduksi di semua turunannya (termasuk dirinya sendiri). Kita melihat bahwa dokumen terkait-1, 2, 3-akan sesuai dengan dokumen yang berbeda di $L[1...4]$. (Hon dkk., 2010).

Jadi, jika kita menggunakan indeks Muthukrishnan untuk menyimpan L , dan untuk setiap *node*, kita menyimpan posisi awal dan akhir di L untuk *subrange* terkait, langkah 2 juga dapat dijawab secara optimal dalam $O(k')$ waktu seperti

dalam *Heuristic 1* tetapi dengan kebutuhan ruang yang lebih kecil yaitu $O(D \log D)$. (Hon dkk., 2010).

Langkah 3 dapat diselesaikan dengan cara yang sama seperti pada langkah 2. Kita amati bahwa setiap k dokumen dengan setidaknya f kemunculan P , bersama-sama dengan k' dokumen yang diperoleh dari langkah 2, harus berisi kumpulan k dokumen yang kita inginkan untuk $top\ document(P, k)$. Jadi pada langkah 3, kita akan secara sewenang-wenang memilih satu set k dokumen dengan setidaknya f kemunculan P , yang mana selanjutnya kita pilih $k-k'$ dokumen yang tidak diperoleh pada langkah 2. Cara sederhana untuk menyelesaikan bagian terakhir adalah dengan mengurutkan, yang mana membutuhkan $O(\min\{m, k \log k\})$ waktu. Untuk mempercepat, kita pertahankan *bit-vector* dari m bit, di mana bit ke- i berkorespondensi dengan dokumen T_i , dengan semua bit diset menjadi 0 pada awalan. Ketika langkah 2 selesai, kita tandai setiap bit yang berkorespondensi dengan k' dokumen dengan 1. Lalu, pada langkah 3, ketika sebuah dokumen diperiksa pada prosedur akhir, kita bisa periksa *bit-vector* untuk melihat apakah dokumen tersebut sudah diperoleh pada langkah 2, sehingga kita bisa dengan mudah mengambil set $k-k'$ dokumen yang kita inginkan. Setelah langkah 3, kita bisa me-reset *bit vector* dalam $O(k)$ waktu dengan mengacu pada *final output* atau hasil akhir. Dengan demikian, kita telah menyelesaikan permasalahan *top document query*. Sebuah indeks yang memerlukan $O(D \log D)$ ruang untuk Δ dapat dipertahankan, sehingga untuk semua masukan pola P dan semua masukan bilangan bulat k dalam $top\ document(P, k)$ dapat dijawab dalam $O(|P| + \log D \log \log D + k)$ waktu. (Hon dkk., 2010).

G. Konstruksi Algoritma Pengindeksan

Kita bisa menghitung nilai *count* lebih efisien dengan menggunakan penghitungan masalah *color set size*. Langkah pertama adalah membuat *LeafList* dan menghitung *leafcount()* dari pohon T . Selanjutnya hitung nilai *lastleaf()* dengan memanfaatkan *LeafList*. Untuk setiap warna c kita simpan indeks daun berwarna c paling terakhir yang kita temui. Nilai awalnya adalah *Nil* dan ketika kita bertemu dengan daun berwarna c kita simpan indeksnya dan mengubah nilai *lastleaf()* menjadi indeks daun tersebut. (Chi dan Hui, 1992).

Selanjutnya kita hitung *CPLCount()* dengan menginisialisasi nilainya dengan 0. Untuk setiap daun x , kita hitung $u = lca(x, lastleaf(x))$ dan menambahkan nilai

$CPLCount(u)$ dengan 1. Terakhir, kita gunakan *post-order traversal* untuk menghitung semua nilai $duplicate()$ dengan hubungan rekursif $duplicate(u) = CPLCount(u) + \sum_{w \in W} duplicate(w)$ di mana W adalah himpunan anak dari u . Setelah mendapatkan nilai $leafcount()$ dan $duplicate()$ maka menghitung nilai $count$ adalah dengan mengurangi nilai tersebut. (Chi dan Hui, 1992).

Penerapannya dilakukan misal ada sebuah $IGST-f$ tertentu dengan *sublist* tereduksi yang dibuat. Tahap *pre-processing* yang dilakukan adalah:

1. Di dalam setiap *node*, tuliskan jumlah dokumen di dalam *sublist* yang tereduksi tersebut.
2. Lakukan *bottom-up traversal* di $IGST-f$, sehingga setiap *node* v memperoleh jumlah total n_v dokumen di dalam *sublist* tereduksi dari semua turunannya (termasuk dirinya sendiri).

Nilai n_v pada setiap *node* v setelah langkah ke-2 pada tahap *pre-processing* di atas berkaitan erat dengan nilai $count$ yang diinginkan. Ketika nilai $count$ menghitung setiap dokumen berbeda dalam *sublist* turunan yang tereduksi sebanyak satu kali, nilai n_v dapat menghitung sebuah dokumen beberapa kali. Untuk memperbaiki hitungan, ada beberapa hal yang harus diperhatikan. Misalkan dokumen i muncul di *sublist* tereduksi dari *node* y . Keturunan y ini dapat dimisalkan dengan u_1, u_2, \dots, u_y terurut berdasarkan kemunculan pertamanya pada *pre-order traversal*. Lebih lanjut, misalkan dokumen i muncul di dalam *sublist* tereduksi persis di j_i keturunan dari *node* v (termasuk dirinya sendiri), maka kita memiliki:

1. Terdapat bilangan bulat z sedemikian rupa sehingga keturunan j_i dari v sama dengan $u_{z+1}, u_{z+2}, \dots, u_{z+j_i}$.
2. Nenek moyang terendah atau *lowest common ancestor* dari u_g dan u_{g+1} ada di subpohon v jika dan hanya jika $g \in \{z + 1, z + 2, \dots, z + j_i - 1\}$.

Implikasi dari hal-hal di atas adalah misalkan j'_i menyatakan jumlah g sedemikian rupa sehingga *lowest common ancestor* dari u_g dan u_{g+1} ada di subpohon v . Maka $j_i - j'_i = 1$ ketika $j_i \geq 1$ dan $j_i - j'_i = 0$ ketika $j_i = 0$. Dengan kata lain, $j_i - j'_i$ selalu mengindikasikan apakah ada tidaknya dokumen i dalam subpohon v . (Hon dkk., 2010).

Ingat bahwa n_v adalah jumlah total dokumen dalam *sublist* tereduksi di semua turunan v . Berdasarkan pernyataan di atas, kita dapat melihat bahwa nilai hitungan untuk *node* v yang merupakan jumlah dokumen yang berbeda dalam *list* yang direduksi di subpohon dari v dapat dihitung dengan $\sum_i (j_i - j'_i) = \sum_i j_i - \sum_i j'_i = n_v - \sum_i j'_i$ di mana i mencakup semua dokumen. (Hon dkk., 2010).

Nilai perbaikan dari $\sum_i j'_i$ untuk setiap *node* v pada IGST- f dapat diperoleh dengan cara yang sama seperti ketika mencari nilai n_v . Detailnya adalah sebagai berikut:

1. Tetapkan penghitung atau *counter* untuk setiap *node*, yang diinisialisasi ke 0.
2. Proses setiap dokumen i seperti di bawah ini:
 - (a) Temukan *node* u_1, u_2, \dots, u_y yang berisi dokumen i dalam *sublist* tereduksinya, di mana *node* diurutkan berdasarkan kemunculan pertama dalam *pre-order traversal*.
 - (b) Tambahkan 1 ke penghitung atau *counter* di *lowest common ancestor* dari u_g dan u_{g+1} , untuk semua g .
3. Lakukan *bottom-up traversal* pada IGST- f , sehingga untuk setiap *node* v memiliki jumlah waktu ketika turunan v menjadi *lowest common ancestor* dari beberapa pasangan pada langkah ke-2.

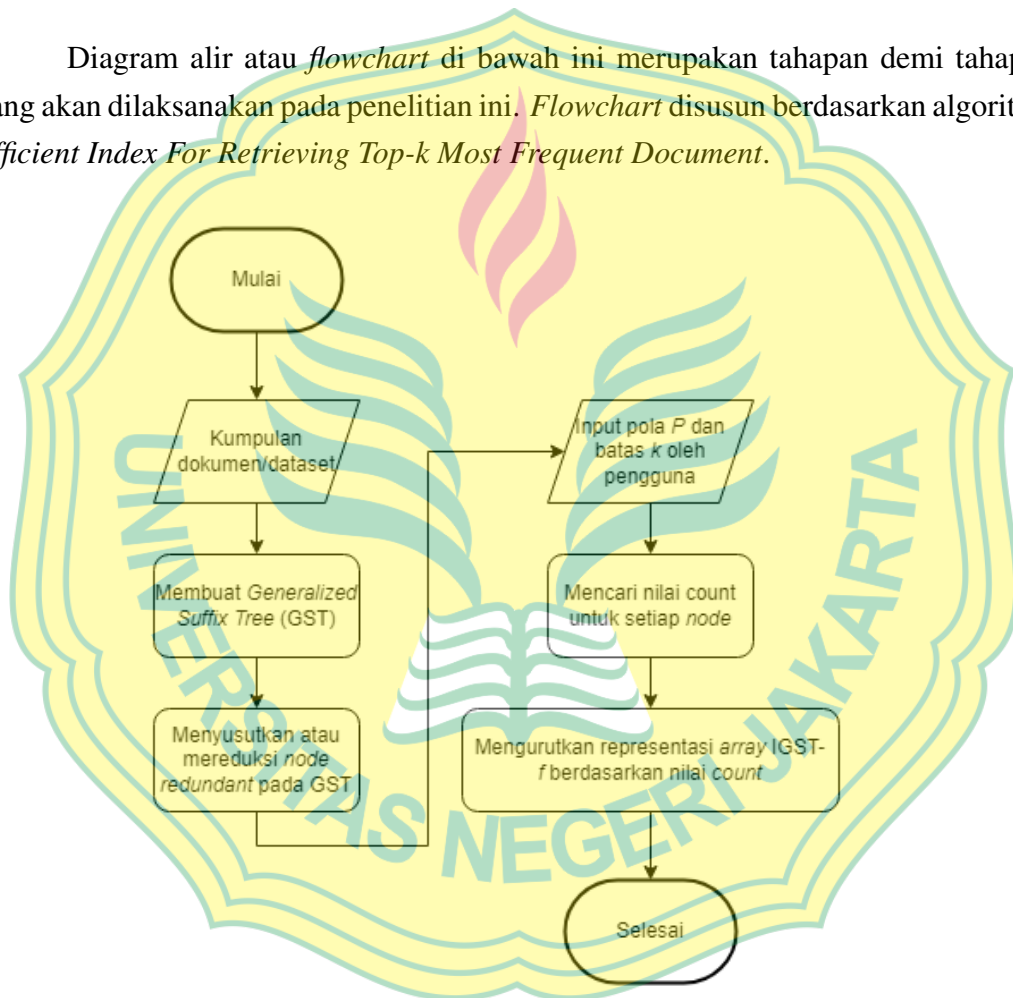
Nilai yang disimpan di setiap *node* v setelah prosedur di atas adalah nilai yang diinginkan untuk $\sum_i j'_i$. Pada akhirnya, dengan mengurangi nilai n_v dengan $\sum_i j'_i$ yang berhubungan pada setiap *node* v , kita bisa mendapatkan nilai *count* yang kita inginkan untuk setiap *node* v . (Hon dkk., 2010).

BAB III

METODOLOGI PENELITIAN

A. Tahapan Penelitian

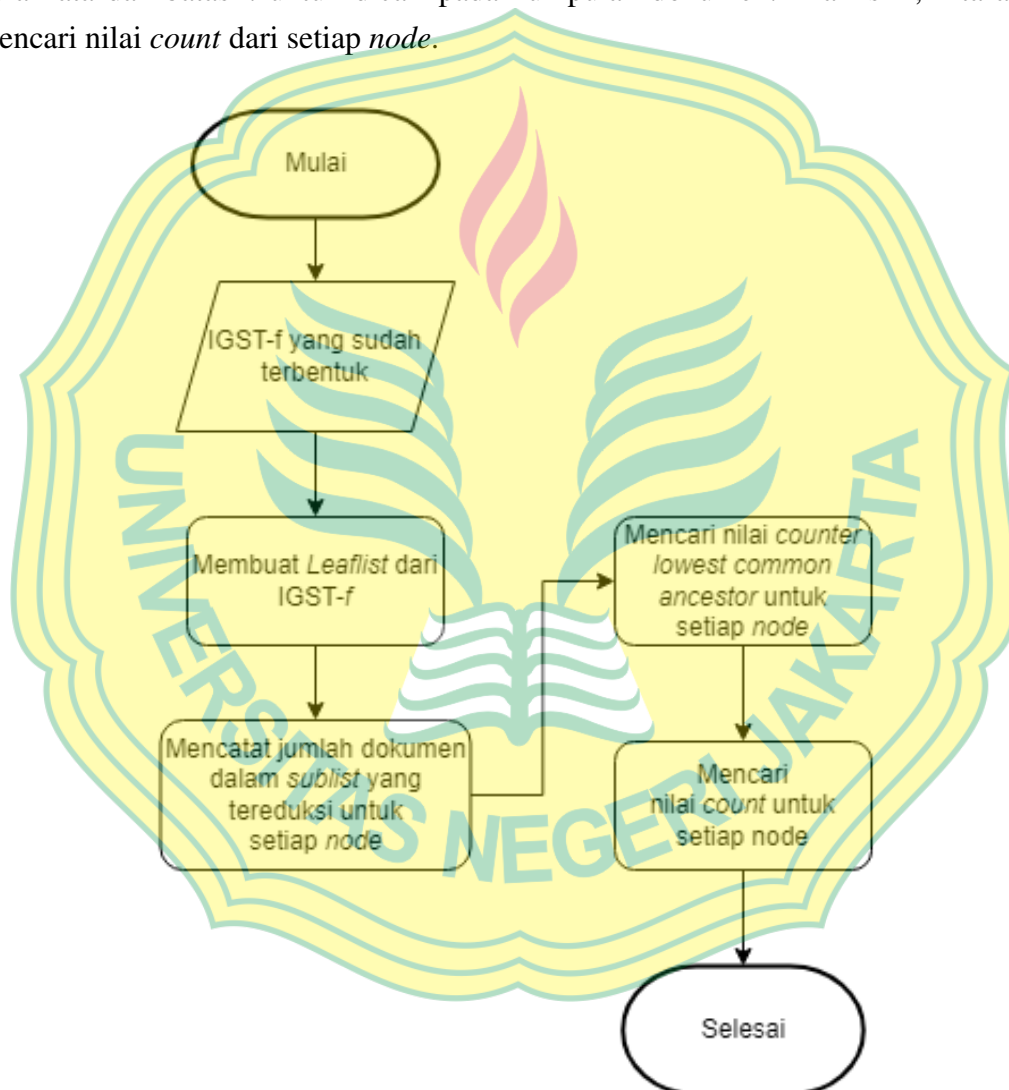
Diagram alir atau *flowchart* di bawah ini merupakan tahapan demi tahapan yang akan dilaksanakan pada penelitian ini. *Flowchart* disusun berdasarkan algoritma *Efficient Index For Retrieving Top-k Most Frequent Document*.



Gambar 3.1: *Flowchart* Tahapan Penelitian Modul Pengindeks

Diagram alir dari algoritma pengindeksan dimulai dari masukan berupa kumpulan dokumen yang akan dibuat menjadi GST. Langkah selanjutnya yaitu membuat GST dari kumpulan dokumen tersebut. Seperti yang dijelaskan pada subbab 2D yaitu *Generalized Suffix Tree*, GST menjadi struktur data awal dari kumpulan dokumen. Konstruksi dari GST juga dijelaskan pada subbab tersebut. Dari GST tersebut kemudian dilakukan penyusutan atau reduksi untuk *node* yang redundan seperti yang sudah dijelaskan pada subbab 2E sehingga akan terbentuk

pohon yang terinduksi untuk frekuensi f yang bernama *Induced Generalized Suffix Tree- f* atau *IGST- f* . *IGST- f* sendiri seperti yang dijelaskan pada bagian akhir subbab 2B mengenai *Document Retrieval* sebelumnya, menjadi kunci atau komponen utama dari pengindeksan. Langkah selanjutnya adalah program menerima masukan berupa pola kata dan batas k untuk dicari pada kumpulan dokumen. Dari sini, kita akan mencari nilai *count* dari setiap *node*.



Gambar 3.2: Flowchart Pencarian Nilai Count

Pertama-tama kita harus membuat *Leaflist* dari *IGST- f* yang sudah terbentuk. Kemudian, mencatat jumlah dokumen dalam *sublist* yang tereduksi untuk setiap *node*. Selanjutnya, mencari nilai *counter lowest common ancestor* dari setiap *node*. Dari sini, kita bisa mendapatkan nilai *count* yang merupakan jumlah dokumen berbeda yang memiliki kemunculan pola P untuk setiap *node*.

Algorithm 1 *Efficient Index For Retrieving Top-k Most Frequent Document*
Algorithm

```

p = string(patternP)
k = integer(limit)
D = documents
gst = []
nv = integer
lcaCounter = 0
countValue = 0
result = []
for all document in D do
    add documents to gst
end for
for all document in gst do
    if document is redundant then
        contracting document ▷ This is a process for making IGST
    end if
end for
for all document in gst do
    count nv
    if document is lowest common ancestor then
        lcaCounter + = 1
    end if
    countValue = nv − lcaCounter
    add document to result
end for
sort result
for i in range (0, k) do
    return result[i]
end for

```

Pada pohon tersebut terdapat 3 warna dan sufiks-sufiks yang tersusun dari pola-pola *string* yang ada. Terdapat pola *aabb* dengan warna 1, *cabc* dengan warna 2, *cdbc* dengan warna 3. Sufiks *bc* dan *c* merupakan sufiks yang paling sering ditemukan untuk warna 2 dan 3. Nantinya, GST akan memuat seluruh sufiks dari seluruh dokumen dan pengindeks atau *indexer* akan mencari pola kata kunci yang diberikan di dalam GST ini.

C. Alat dan Bahan Penelitian

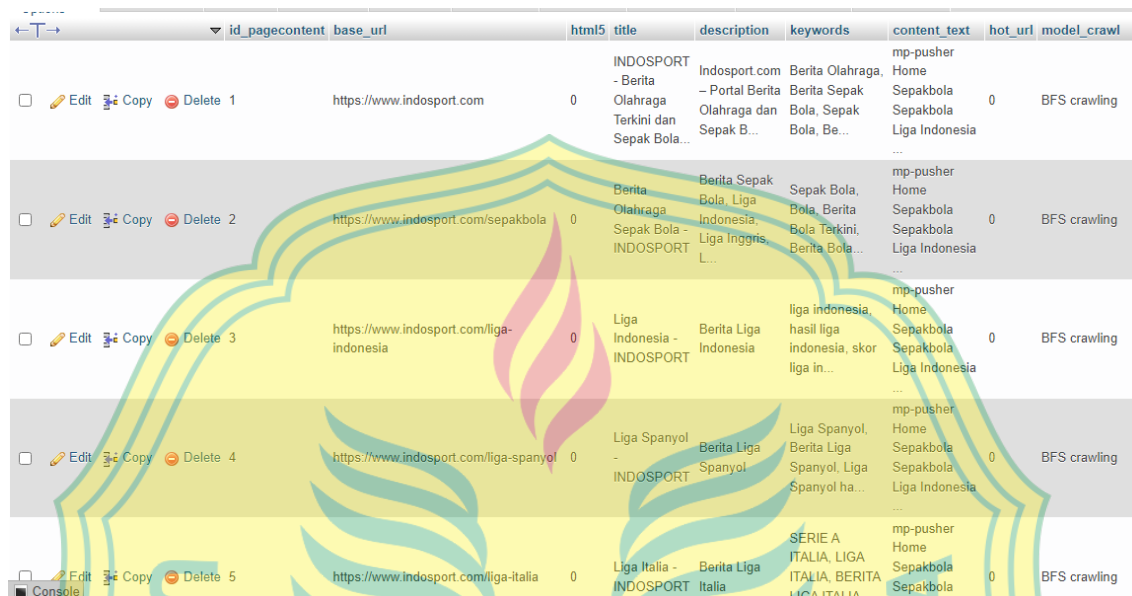
Dalam penelitian ini, beberapa alat berupa perangkat keras yang digunakan untuk menunjang pembuatan modul pengindeks adalah sebagai berikut:

1. Laptop ASUS X550D dengan CPU AMD A8 dan RAM 10GB
2. Koneksi berbasis *Wi-Fi*

Selain perangkat keras, perangkat lunak yang dipakai untuk penelitian ini adalah sebagai berikut:

1. Windows 10 64-bit Operating System
2. Visual Studio Code sebagai *code editor*
3. XAMPP untuk menjalankan *MySQL database*
4. Python 3 untuk menjalankan program Python

D. Dataset



	id_pagecontent	base_url	html5	title	description	keywords	content_text	hot_url	model_crawl
<input type="checkbox"/> Edit Copy Delete 1		https://www.indosport.com	0	INDOSPORT - Berita Olahraga Terkini dan Sepak Bola...	Indosport.com - Portal Berita Olahraga dan Sepak B...	Berita Olahraga, Berita Sepak Bola, Sepak Bola, Be...	mp-pusher Home Sepakbola Sepakbola Liga Indonesia ...	0	BFS crawling
<input type="checkbox"/> Edit Copy Delete 2		https://www.indosport.com/sepakbola	0	Berita Olahraga Sepak Bola - INDOSPORT	Berita Sepak Bola, Liga Indonesia, Liga Inggris, L...	Sepak Bola, Bola, Berita Bola Terkini, Berita Bola...	mp-pusher Home Sepakbola Sepakbola Liga Indonesia ...	0	BFS crawling
<input type="checkbox"/> Edit Copy Delete 3		https://www.indosport.com/liga-indonesia	0	Liga Indonesia - INDOSPORT	Berita Liga Indonesia	liga indonesia, hasil liga indonesia, skor liga in...	mp-pusher Home Sepakbola Sepakbola Liga Indonesia ...	0	BFS crawling
<input type="checkbox"/> Edit Copy Delete 4		https://www.indosport.com/liga-spain	0	Liga Spanyol - INDOSPORT	Berita Liga Spanyol	Liga Spanyol, Berita Liga Spanyol, Liga Spanyol ha...	mp-pusher Home Sepakbola Sepakbola Liga Indonesia ...	0	BFS crawling
<input type="checkbox"/> Edit Copy Delete 5		https://www.indosport.com/liga-italia	0	Liga Italia - INDOSPORT	Berita Liga Italia	SERIE A ITALIA, LIGA ITALIA, BERITA LIGA ITALIA	mp-pusher Home Sepakbola Sepakbola ...	0	BFS crawling

Gambar 3.4: 5 Data Pertama Dataset pada Tabel *Page Information*

Pada penelitian ini, dataset yang akan digunakan sebagai kumpulan dokumen adalah tabel *page information* khususnya kolom *title*. Dataset ini disusun oleh Muhammad Fathan Qoriiba dalam penelitiannya yang berjudul *PERANCANGAN CRAWLER SEBAGAI PENDUKUNG PADA SEARCH ENGINE*. Dataset ini disusun menggunakan *tools crawling* yang telah dikembangkan. Situs yang digunakan untuk proses *crawling* adalah situs *indosport.com* dan *curiouscuisiniere.com*. Total data yang tersimpan di dalam dataset tersebut berukuran 176 MiB atau sekitar 185 MB.

Ada 8 tabel yang tersimpan di dalam dataset ini dengan rincian sebagai berikut:

1. Tabel *page information* berisi 1060 baris halaman web dengan 8 atribut.

Tabel 3.1: Deskripsi Tabel *Page Information*

Atribut	Deskripsi
<i>base url</i>	Pranala dari halaman web
<i>html5</i>	Menyatakan halaman tersebut tersusun dari HTML5 atau bukan
<i>title</i>	Judul halaman web
<i>description</i>	Deskripsi halaman web
<i>keywords</i>	Kata kunci dari halaman web
<i>content text</i>	Isi konten dari halaman web
<i>hot url</i>	Menyatakan halaman web tersebut termasuk <i>hot url</i> atau tidak
<i>model crawl</i>	Menyatakan cara <i>crawling</i> yang digunakan pada halaman web

2. Tabel *Linking* berisi 131.581 baris *linking* dengan 3 atribut.

Tabel 3.2: Deskripsi Tabel *Linking*

Atribut	Deskripsi
<i>crawl id</i>	ID dari <i>crawl</i>
<i>url</i>	Pranala atau <i>link</i>
<i>outgoing link</i>	Pranala lain yang terhubung dengan atribut <i>url</i> .

3. Tabel *Style Resource* berisi 7.530 baris *css* dengan 2 atribut.

Tabel 3.3: Deskripsi Tabel *Style Resource*

Atribut	Deskripsi
<i>base url</i>	Pranala dari halaman web
<i>style</i>	CSS dari halaman web

4. Tabel *Script Resource* berisi 29.732 baris *js* dengan 2 atribut.

Tabel 3.4: Deskripsi Tabel *Script Resource*

Atribut	Deskripsi
<i>base url</i>	Pranala dari halaman web
<i>script</i>	<i>Script JS</i> dari halaman web

5. Tabel *forms* berisi 2.476 baris *form* dengan 2 atribut.

Tabel 3.5: Deskripsi Tabel *Forms*

Atribut	Deskripsi
<i>base url</i>	Pranala dari halaman web
<i>form</i>	<i>Script form</i> dari halaman web

6. Tabel *images* berisi 38.644 baris *image* dengan 2 atribut.

Tabel 3.6: Deskripsi Tabel *Images*

Atribut	Deskripsi
<i>base url</i>	Pranala dari halaman web
<i>image</i>	<i>Script image</i> dari halaman web

7. Tabel *List* berisi 68.341 baris *list* dengan 2 atribut.

Tabel 3.7: Deskripsi Tabel *List*

Atribut	Deskripsi
<i>base url</i>	Pranala dari halaman web
<i>list</i>	<i>Script list</i> dari halaman web

8. Tabel *Tables* berisi 75 baris *table* dengan 2 atribut.

Tabel 3.8: Deskripsi Tabel *Tables*

Atribut	Deskripsi
<i>base url</i>	Pranala dari halaman web
<i>table</i>	<i>Script table</i> dari halaman web

Nantinya akan dilakukan sedikit pembersihan pada data dengan menghilangkan isi-isi yang kurang berkaitan dengan konteks seperti simbol. Data *title* halaman web yang sudah bersih kemudian akan dijadikan acuan sebagai kumpulan dokumen yang akan digunakan untuk proses penelitian ini.

E. Tahapan Pengembangan

Pada penelitian ini, penulis akan membuat modul pengindeks atau *indexer* yang menerima *input string* berupa kata kunci atau pola kata dan *integer input* berupa batasan berapa dokumen yang dikembalikan sebagai hasil. *Indexer* akan mengembalikan beberapa dokumen sesuai dengan batasan yang diberikan dan dengan relevansi paling tinggi yang mengandung kata kunci tersebut. Algoritma yang akan digunakan adalah algoritma *Efficient index for retrieving top-k most frequent documents*.

Penelitian ini merupakan penelitian pendukung dari keseluruhan penelitian *search engine*. Penelitian induk dari rangkaian penelitian *search engine* antara lain PERANCANGAN CRAWLER SEBAGAI PENDUKUNG PADA SEARCH ENGINE oleh Muhammad Fathan Qoriiba dan PERANCANGAN ARSITEKTUR SEARCH ENGINE DENGAN MENINGTEGRASIKAN WEB CRAWLER, ALGORITMA PAGE RANKING, DAN DOCUMENT RANKING oleh Lazuardy Khatulistiwa.

Pembuatan modul pengindeks pada penelitian ini akan dilakukan bertahap sesuai dengan setiap proses pada *flowchart* 3.1. Waktu dari pembuatan setiap langkah atau proses diestimasikan 1-2 minggu dan paling maksimal 3 minggu tergantung kompleksitas dari proses tersebut. Jika penelitian ini selesai lebih dulu dibandingkan penelitian induknya yaitu PERANCANGAN ARSITEKTUR SEARCH ENGINE DENGAN MENINGTEGRASIKAN WEB CRAWLER, ALGORITMA PAGE RANKING, DAN DOCUMENT RANKING yang dibuat oleh Lazuardy Khatulistiwa, maka penelitian ini akan berjalan secara otonom dan hasil akhirnya adalah modul pengindeks. Namun, jika penelitian induknya selesai lebih dulu dibandingkan penelitian ini, maka struktur direktori serta *design class* dan fungsi dari penelitian induk akan diikuti *skemanya* atau dengan kata lain modul pengindeks akan disesuaikan dengan arsitektur *search engine* besar.

F. Pengujian

Untuk pengujian dari hasil penelitian ini, akan ada dua poin penting pengujian, yaitu:

1. Relevansi pencarian yaitu kesesuaian hasil pencarian yang dihasilkan oleh program dengan pola kata kunci yang diberikan pada saat awal memasukkan *input*. Relevansi pencarian akan diujikan dengan *mean average precision*.

Indikator modul pengindeks berjalan dengan baik adalah dokumen yang dikembalikan relevan dengan kata yang dicari oleh pengguna.

2. Waktu pengindeksan atau *indexing time* yaitu kecepatan pengindeksan dari awal proses *input* masuk hingga mengembalikan hasil. Waktu pengindeksan akan diujikan dengan *performance testing* yang berfokus pada *speed*. Indikator modul pengindeks berjalan dengan baik adalah waktu pengambilan dokumen dalam waktu singkat yaitu hitungan detik atau kurang.

1. Relevansi Pencarian

Mean Average Precision (MAP) adalah nilai rata-rata aritmatika dari nilai *average precision* pada sistem temu kembali informasi untuk sekumpulan n topik pencarian. Metrik evaluasi *Mean Average Precision* telah lama digunakan sebagai standar evaluasi sistem pengambilan informasi yang diakui secara umum pada Konferensi Pemulihan Teks NIST (TREC). Selama 25 tahun terakhir, banyak penelitian yang dipublikasikan dalam bidang pengambilan informasi mengandalkan perbedaan yang diamati pada MAP untuk menarik kesimpulan tentang efektivitas teknik atau sistem yang dipelajari. (Beitzel dkk., 2009)

$$MAP = \frac{1}{n} \sum_n AP_n \quad (3.1)$$

AP adalah *average precision* dari untuk topik tertentu dari n topik. (Beitzel dkk., 2009)

Untuk mendapatkan nilai *precision* dalam konteks pencarian informasi, rumusnya adalah sebagai berikut:

$$precision = \frac{\text{total dokumen relevan}}{\text{total dokumen hasil pencarian}} \quad (3.2)$$

Nilai *precision* untuk konteks pencarian informasi didapatkan dari hasil pembagian antara total dokumen yang relevan dengan total dokumen yang dihasilkan. (Ting, 2010)

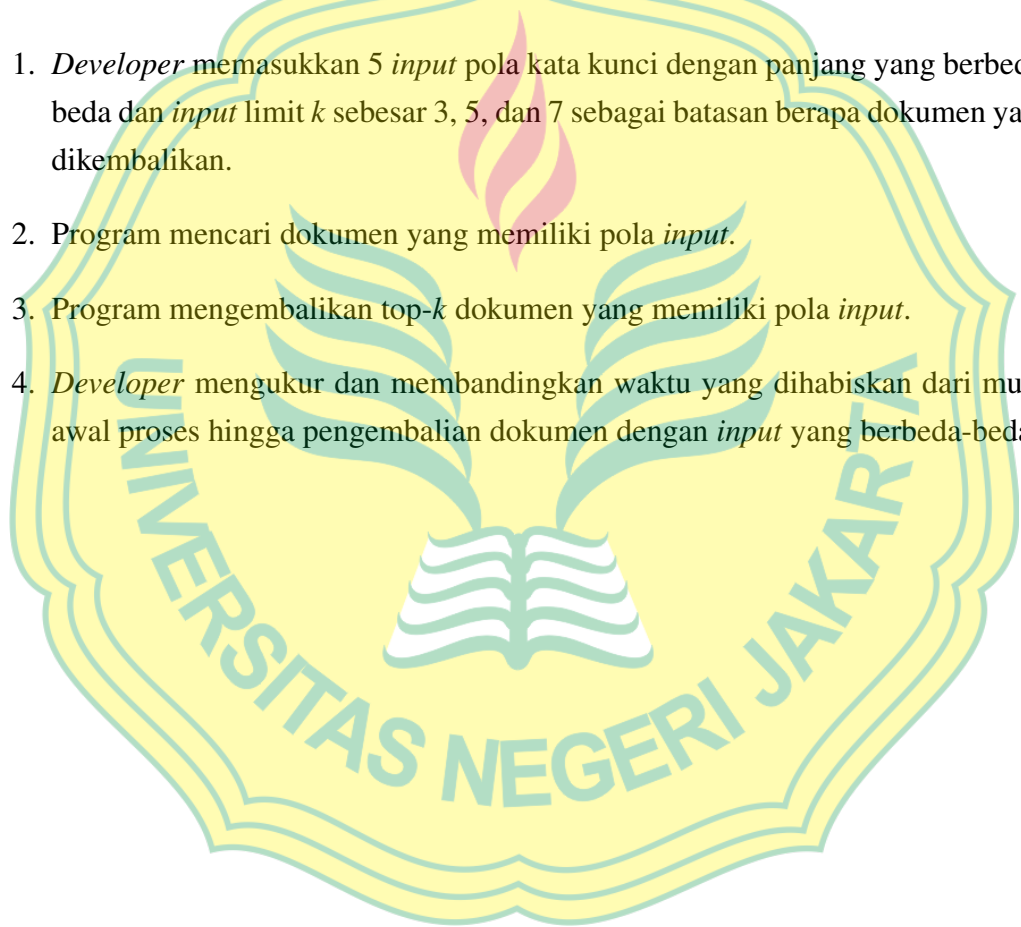
Dalam penelitian ini, 3 *tester* dengan pola *input* yang sama akan berpartisipasi dalam pengujian *mean average precision* untuk menilai apakah dokumen yang dikembalikan relevan dengan pencarian yang diinginkan. Dokumen yang akan dijadikan pengujian adalah 5 dokumen teratas pada hasil pencarian.

Penggunaan *mean average precision* menghasilkan skala hasil 0 sampai 1.

2. Waktu Pengindeksan

Performance testing merupakan pengujian yang melakukan verifikasi pada kebutuhan non-fungsional yaitu performa perangkat lunak salah satunya kecepatan atau *speed*. Untuk waktu pengindeksan, skenarionya adalah sebagai berikut:

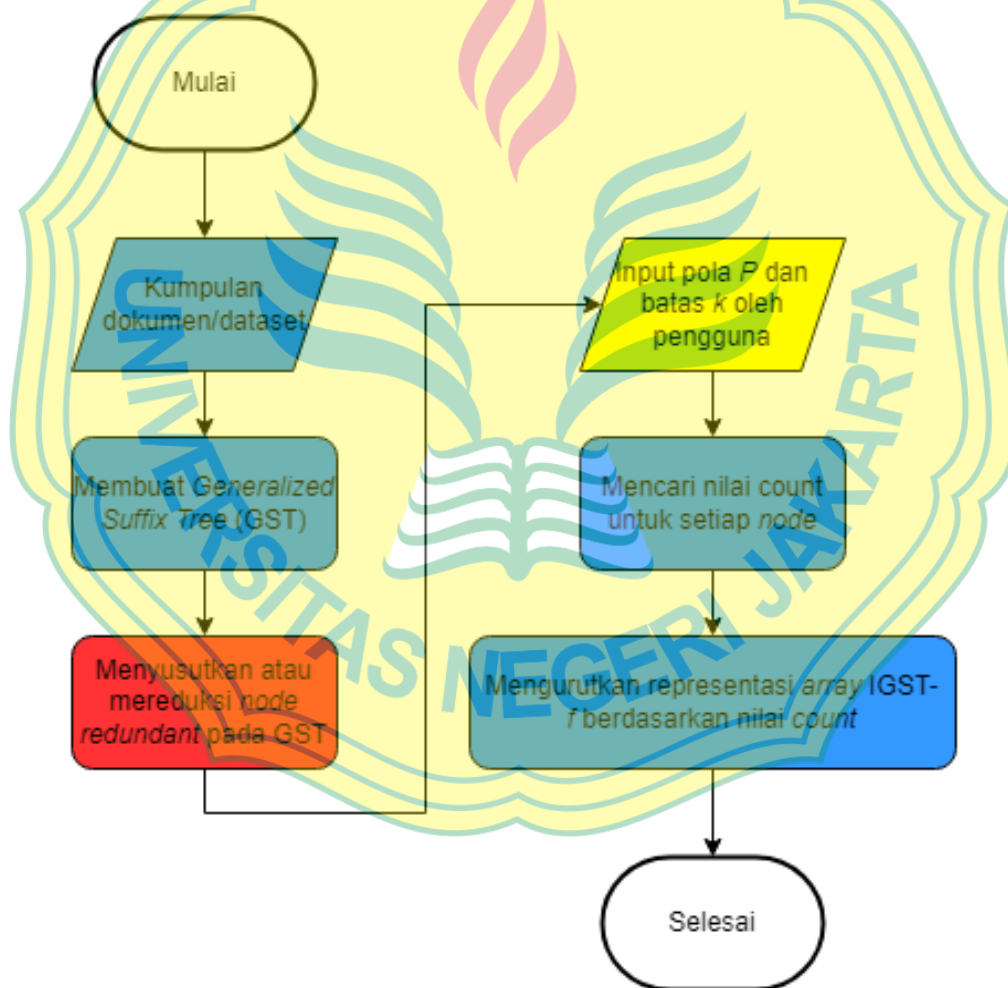
1. *Developer* memasukkan 5 *input* pola kata kunci dengan panjang yang berbeda-beda dan *input* limit k sebesar 3, 5, dan 7 sebagai batasan berapa dokumen yang dikembalikan.
2. Program mencari dokumen yang memiliki pola *input*.
3. Program mengembalikan top- k dokumen yang memiliki pola *input*.
4. *Developer* mengukur dan membandingkan waktu yang dihabiskan dari mulai awal proses hingga pengembalian dokumen dengan *input* yang berbeda-beda.



BAB IV

HASIL DAN PEMBAHASAN

Pada bab ini akan dibahas mengenai hasil penelitian dan implementasi dari rancangan yang sebelumnya telah dibuat. Rancangan yang telah dibuat diimplementasikan dengan proses *coding* menggunakan bahasa pemrograman *Python*. Berikut adalah hasil implementasi dari rancangan yang telah dibuat:



Gambar 4.1: Komponen *Flowchart* yang Diimplementasikan

Pada penelitian ini komponen *flowchart* yang ditandai dengan warna biru berhasil diimplementasikan. Komponen *flowchart* yang berwarna merah yaitu mereduksi *node* redundan belum berhasil diimplementasikan karena keterbatasan kemampuan dan waktu. Untuk bagian *input* yang ditandai dengan warna kuning

terdapat modifikasi di mana *input* yang sudah diimplementasikan hanya memerlukan pola kata yang ingin dicari dan tidak memerlukan batas k agar lebih mudah untuk digunakan dan mengikuti alur natural ketika melakukan pencarian.

A. Pengolahan Dataset

Dataset yang digunakan sebagai *input* untuk membentuk *tree* adalah tabel *page information* dan kolom *title* yang berjumlah 10.686 *rows* atau data. Dataset ini didapatkan dari hasil *crawling* pada penelitian induk yaitu PERANCANGAN ARSITEKTUR SEARCH ENGINE DENGAN MENINGTEGRASIKAN WEB CRAWLER, ALGORITMA PAGE RANKING, DAN DOCUMENT RANKING oleh Lazuardy Khatulistiwa. Dataset ini juga dapat ditemukan pada penelitian induk sebagai sumber data yang digunakan untuk perhitungan *page rank*, *TF-IDF*, dan *similarity score*. Penggunaan dataset ini adalah salah satu bentuk pengintegrasian dengan penelitian induk.

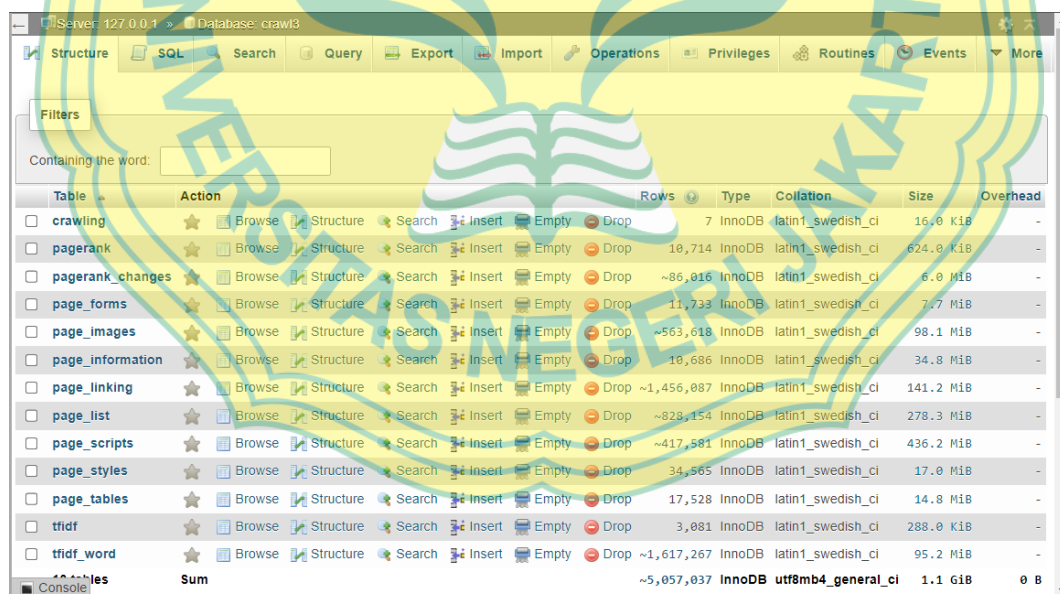


Table	Action	Rows	Type	Collation	Size	Overhead
crawling	Browse Structure Search Insert Empty Drop	7	InnoDB	latin1_swedish_ci	16.0 KiB	-
pagerank	Browse Structure Search Insert Empty Drop	10,714	InnoDB	latin1_swedish_ci	624.0 KiB	-
pagerank_changes	Browse Structure Search Insert Empty Drop	~86,016	InnoDB	latin1_swedish_ci	6.0 MiB	-
page_forms	Browse Structure Search Insert Empty Drop	11,733	InnoDB	latin1_swedish_ci	7.7 MiB	-
page_images	Browse Structure Search Insert Empty Drop	~563,618	InnoDB	latin1_swedish_ci	98.1 MiB	-
page_information	Browse Structure Search Insert Empty Drop	10,686	InnoDB	latin1_swedish_ci	34.8 MiB	-
page_linking	Browse Structure Search Insert Empty Drop	~1,456,087	InnoDB	latin1_swedish_ci	141.2 MiB	-
page_list	Browse Structure Search Insert Empty Drop	~828,154	InnoDB	latin1_swedish_ci	278.3 MiB	-
page_scripts	Browse Structure Search Insert Empty Drop	~417,581	InnoDB	latin1_swedish_ci	436.2 MiB	-
page_styles	Browse Structure Search Insert Empty Drop	34,565	InnoDB	latin1_swedish_ci	17.0 MiB	-
page_tables	Browse Structure Search Insert Empty Drop	17,528	InnoDB	latin1_swedish_ci	14.8 MiB	-
tfidf	Browse Structure Search Insert Empty Drop	3,081	InnoDB	latin1_swedish_ci	288.0 KiB	-
tfidf_word	Browse Structure Search Insert Empty Drop	~1,617,267	InnoDB	latin1_swedish_ci	95.2 MiB	-
Sum		~5,057,037	InnoDB	utf8mb4_general_ci	1.1 GiB	0 B

Gambar 4.2: Jumlah Data pada Database Hasil Crawling

Terdapat tabel-tabel lain dari dataset seperti *crawling*, *page rank*, *page rank changes*, *page forms*, *page images*, *page linking*, *page list*, *page scripts*, *page styles*, *page tables*, *tfidf*, dan *tfidf word*. Tabel yang digunakan pada penelitian ini adalah tabel *page information* dengan data sebanyak 10.868 *rows*.

Showing rows 0 - 499 (10686 total, Query took 0.0320 seconds.)

SELECT * FROM `page_information`

Profiling [Edit Inline] [Edit] [Explain SQL] [Create PHP code] [Refresh]

1 > >> Number of rows: 500 Filter rows: Search this table Sort by key: None

	id_page	crawl_id	url	html5	title	description	keywords
1	1	1	https://detik.com	1	detikcom - Informasi Berita Terkini dan Terbaru Ha...	Indeks berita terkini dan terbaru hari ini dari pe...	berita hari ini, t terbaru, i...
2	2	1	https://www.detik.com/tagfrom=framebar	1	detikcom - Informasi Berita Terkini dan Terbaru Ha...	Indeks berita terkini dan terbaru hari ini dari pe...	berita hari ini, t terbaru, i...
3	3	1	https://www.detik.com/terpopuler	1	Berita Terpopuler dan Terbaru Hari Ini di Kanal De...	Rangkuman Berita Terpopuler dan Terbaru Hari Ini d...	Berita Terpopu berita terbar...
4	4	1	https://news.detik.com/kolom/kirim	0	Detikconnect - Sign In -		
5	5	1	https://20.detik.com/live	0	Live Streaming Trans TV - Milik Kita	Tonton siaran TV online dari Trans	TransTV, My Ti

Gambar 4.3: Tampilan Data pada Tabel *page information*

Pada tabel *page information* terdapat kolom *id page* yaitu *id* dari halaman, *crawl id* yang merupakan *id dari crawl*, *url* yaitu pranala dari halaman web, *title* yaitu judul halaman web, *description* yaitu deskripsi dari halaman web, *keywords* yaitu kata kunci dari halaman web, *hot url* yaitu hot url dari halaman web, *content text* yaitu konten dari halaman web, *size bytes* yaitu ukuran dari halaman web, *model crawl* yaitu model *crawling* yang digunakan, *duration crawl* yaitu durasi dari *crawling*, dan *created at* yaitu kapan data tersebut dibuat. Kolom *title* adalah data yang akan digunakan sebagai bahan pembentukan *Generalized Suffix Tree*.

1. Pengambilan Data dan Pembersihan Data

Sebelum *title* dibentuk menjadi *tree* atau dimasukkan ke dalam *tree* perlu dilakukan pembersihan data *title* dari simbol-simbol agar data di *tree* hanya terdiri dari karakter huruf dan angka saja.

```
def getTitle():
    # Connect ke title database
    connection = pymysql.connect(host='localhost',
                                user='root',
                                password='',
                                database='crawl3',
                                charset='utf8mb4',
                                cursorclass=pymysql.cursors.DictCursor,
                                autocommit=True)

    with connection:
        with connection.cursor() as cursor:
            cursor.execute(
                "SELECT id_page, title FROM page_information")
            result = cursor.fetchall()
            for data in result:
                # menjadikan title lower case untuk dimasukkan ke tree
                data["title"] = data["title"].lower()
                # cleaning title dari simbol untuk input tree
                data["title"] = re.sub('[^A-Za-z0-9 ]+', "", data["title"])
            return result
```

Gambar 4.4: Source Code untuk Mengambil Data dari Database

Fungsi *connect* di-import dari library *pymysql* untuk menjalin koneksi dari database dan mengambil data *title* dari tabel *page information*. Selanjutnya untuk setiap data *title* diproses menjadi *lowercase* dan dihilangkan simbol-simbol selain huruf dan angka menggunakan *regex*.

```
{'id_page': 1, 'title': 'detikcom - Informasi Berita Terkini dan Terbaru Hari Ini'}
{'id_page': 2, 'title': 'detikcom - Informasi Berita Terkini dan Terbaru Hari Ini'}
{'id_page': 3, 'title': 'Berita Terpopuler dan Terbaru Hari Ini di Kanal Detikcom - de'}
{'id_page': 4, 'title': 'Detikconnect - Sign In'}
{'id_page': 5, 'title': 'Live Streaming Trans TV - Milik Kita Bersama - 20Detik'}
{'id_page': 1, 'title': 'detikcom informasi berita terkini dan terbaru hari ini'}
{'id_page': 2, 'title': 'detikcom informasi berita terkini dan terbaru hari ini'}
{'id_page': 3, 'title': 'berita terpopuler dan terbaru hari ini di kanal detikcom de'}
{'id_page': 4, 'title': 'detikconnect sign in'}
{'id_page': 5, 'title': 'live streaming trans tv milik kita bersama 20detik'}
```

Gambar 4.5: Sampel Data Title yang Belum Dibersihkan dan Sudah Dibersihkan

Pada gambar di atas terlihat sampel perbedaan di mana data judul halaman web yang belum dibersihkan masih mengandung simbol. Data judul halaman web yang sudah dibersihkan berubah menjadi *lowercase* dan bersih dari simbol.

B. Pembentukan *Generalized Suffix Tree*

Pembentukan *Generalized Suffix Tree* telah dijelaskan pada bab 2 di bagian Konstruksi *Generalized Suffix Tree*. Langkah-langkahnya sebagai berikut:

1. Buat *tree* awal yaitu *root*
2. Tambahkan terminal *node* agar setiap sufiks dari kata *input* terwakilkan dalam *tree*
3. Tambahkan setiap sufiks ke *tree* jika pada *tree* belum ada sufiks tersebut
4. Jika prefiks dari sufiks yang akan ditambahkan sudah ada maka penambahan sufiks ke *tree* menyesuaikan dengan menjadikan sufiks sebagai anak dari *prefiks* tersebut

Data *title* yang sudah dibersihkan selanjutnya akan dimasukkan ke dalam *tree* atau dengan kata lain *Generalized Suffix Tree* akan dibentuk. Pembentukan *Generalized Suffix Tree* menggunakan *naive algorithm* dengan notasi waktu $O(n^2)$ sesuai dengan ilustrasi subbab 2D. Hasil dari penelitian ini yaitu *Generalized Suffix Tree* menjadi struktur data terakhir yang digunakan sebagai modul pengindeks untuk keperluan pemeringkatan dokumen.

```
def makeTree(data):
    # inisiasi root
    root = Node("root")
    for title in data:
        # pemisahan setiap kata pada title untuk diproses
        for word in title["title"].split():
            wordIndex = title["id_page"]
            # penambahan terminal node untuk pembentukan GST
            word += "$"
            # proses memasukkan tiap sufiks dari kata pada title
            for i in range(len(word)):
                suf = word[i:]
                addChild(suf=suf, parent="root", tree=root, index=wordIndex)
    return root
```

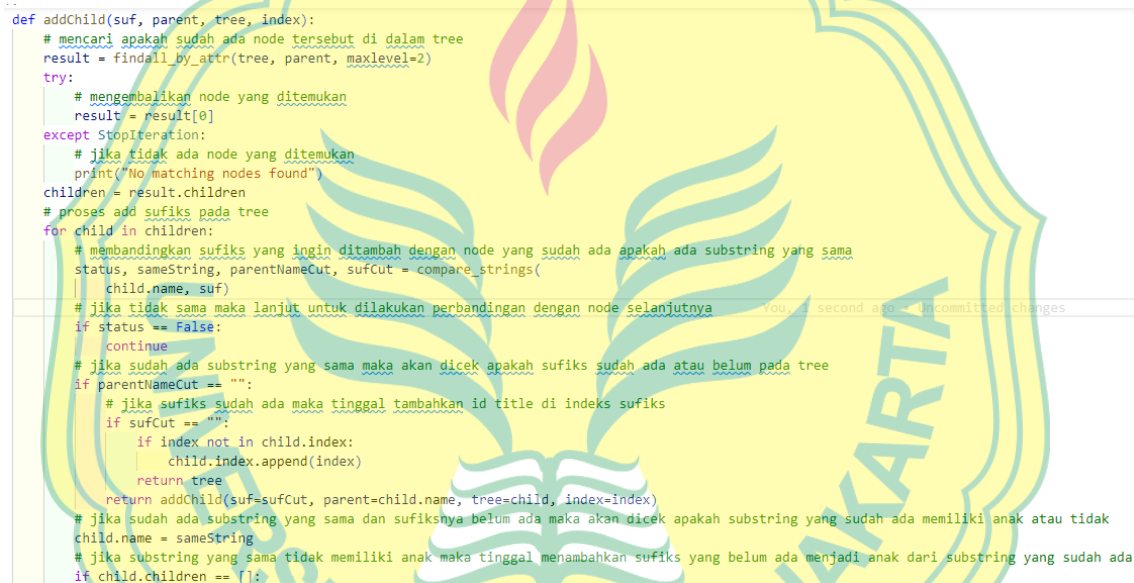
Gambar 4.6: Source Code untuk Membuat *Generalized Suffix Tree*

Fungsi *Node* dari *library anytree* digunakan sebagai struktur pohon yang akan dibentuk. Inisiasi *node root* sebagai *root* dari *Generalized Suffix Tree*. *Title* dari setiap

halaman web diproses untuk dimasukkan ke dalam *tree* dengan memasukkan sufiks satu per satu dari tiap kata yang ada pada judul halaman.

1. Penambahan Sufiks Ke Dalam Tree

Pada fungsi sebelumnya yaitu *makeTree* terdapat fungsi *addChild* yaitu fungsi untuk menambahkan sufiks ke dalam *tree*. Berikut adalah *source code* fungsi *addChild*:



```
def addChild(suf, parent, tree, index):
    # mencari apakah sudah ada node tersebut di dalam tree
    result = findall_by_attr(tree, parent, maxlevel=2)
    try:
        # mengembalikan node yang ditemukan
        result = result[0]
    except StopIteration:
        # jika tidak ada node yang ditemukan
        print("No matching nodes found")
    children = result.children
    # proses add sufiks pada tree
    for child in children:
        # membandingkan sufiks yang ingin ditambah dengan node yang sudah ada apakah ada substring yang sama
        status, sameString, parentNameCut, sufCut = compare_strings(
            child.name, suf)
        # jika tidak sama maka lanjut untuk dilakukan perbandingan dengan node selanjutnya
        if status == False:
            continue
        # jika sudah ada substring yang sama maka akan dicek apakah sufiks sudah ada atau belum pada tree
        if parentNameCut == "":
            # jika sufiks sudah ada maka tinggal tambahkan id title di indeks sufiks
            if sufCut == "":
                if index not in child.index:
                    child.index.append(index)
                return tree
            return addChild(suf=sufCut, parent=child.name, tree=child, index=index)
        # jika sudah ada substring yang sama dan sufiknya belum ada maka akan dicek apakah substring yang sudah ada memiliki anak atau tidak
        child.name = sameString
        # jika substring yang sama tidak memiliki anak maka tinggal menambahkan sufiks yang belum ada menjadi anak dari substring yang sudah ada
        if child.children == []:
```

Gambar 4.7: Source Code untuk Menambah Sufiks pada Tree

Pada fungsi *addChild* terdapat proses penambahan masing-masing sufiks dari setiap kata pada data *title*. Hal yang pertama dilakukan adalah mencari apakah sufiks tersebut sudah ada di dalam *tree* dengan membandingkan sufiks yang ingin ditambahkan dengan *node* yang sudah ada. Jika ada *substring* dari sufiks di *tree*, maka akan dicek apakah sufiks tersebut sudah ada di *tree* atau belum. Jika sufiks sudah ada maka hanya perlu menambahkan atribut *id page* sebagai indeks pada *node* tersebut. Jika sufiks belum ada di *tree* namun *substring* yang sama sudah ada, maka akan diperiksa apakah *substring* atau *node* yang sudah ada memiliki anak atau tidak. Jika *substring* yang sama tidak memiliki anak, maka hanya perlu menambahkan sufiks yang belum ada menjadi anak dari *substring* yang sudah ada.

```

# jika substring yang sama tidak memiliki anak maka tinggal menambahkan sufiks yang belum ada menjadi anak dari substring yang sudah ada
if child.children == []:
    Node(sufCut, parent=child, index=[index])
    Node(parentNameCut, parent=child, index=child.index)
    return tree
# jika substring yang sama memiliki anak maka anak dari substring yang lama nya harus dipisahkan dengan sufiks yang akan ditambahkan
nodeParentCut = Node(parentNameCut, index=child.index)
nodeParentCut.children = child.children
nodeParentCut.parent = child
Node(sufCut, parent=child, index=[index])
return tree
# penambahan sufiks pada tree jika belum ada di tree
Node(suf, parent=result, index=[index])
return tree

```

Gambar 4.8: Source Code untuk Menambah Sufiks pada Tree

Jika *substring* yang sama memiliki anak, maka anak dari *substring* yang lama harus dipisahkan dengan sufiks yang akan ditambahkan. Terakhir, jika tidak ada *substring* yang sama pada *tree* maka otomatis belum ada sufiks pada *tree* sehingga hanya perlu menambahkan sufiks ke *tree*.

```

# compare string adalah fungsi untuk membandingkan 2 string apakah memiliki substring yang sama
def compare_strings(a, b):
    # a itu node, b itu sufiks yang mau ditambah
    if a is None or b is None: # jika salah satu dari 2 string adalah none maka return False
        return False
    size = min(len(a), len(b))
    # inisiasi status awal di mana defaultnya false dan tidak ada string yang sama
    status = False
    sameString = ""
    sufCut = ""
    parentNameCut = ""
    i = 0
    while i < size and a[i] == b[i]: # membandingkan node dengan sufiks
        status = True
        sameString += a[i]
        parentNameCut = a.removeprefix(sameString)
        sufCut = b.removeprefix(sameString)
        i += 1
    # mengembalikan status perbandingan, string yang sama, node yang telah dipotong, dan sufiks yang telah dipotong
    return status, sameString, parentNameCut, sufCut

```

Gambar 4.9: Source Code untuk Membandingkan Sufiks dengan Node

Fungsi *compare strings* digunakan untuk membandingkan sufiks yang akan dimasukkan ke *tree* dengan *node* yang sudah ada di *tree*. Jika salah satu dari kedua *string* adalah *none* maka status kesamaan langsung *False*. Untuk membandingkan kedua *string* sama atau tidak maka diambil panjang *string* yang paling kecil sebagai acuan dalam perbandingan. Inialisasi status awal dilakukan di mana tidak ada *string* yang sama. Jika ada kesamaan pada *string* maka ubah status menjadi *True*, potong awalan *node* dan sufiks sehingga proses perbandingan bisa dilanjutkan hingga tidak ada huruf yang sama. Setelah semua proses selesai, kembalikan status, *string node*, *string* sufiks, dan *string* yang sama pada kedua *input*.

2. Penyimpanan *Generalized Suffix Tree* Sebagai Objek Menggunakan *Pickle*

Berikut adalah *source code* untuk menyimpan *tree* yang sudah terbentuk ke dalam file objek:

```
def storeData(tree):
    # Membuat file bernama gst
    dbfile = open('gst', 'ab')

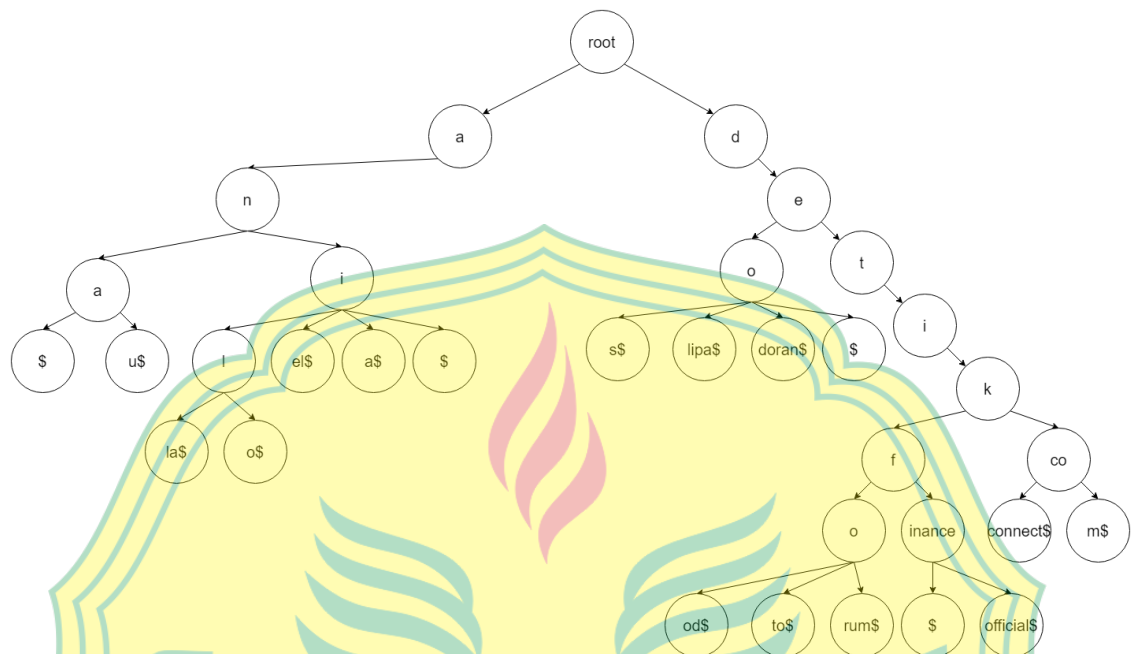
    # mengisi file gst dengan tree yang sudah jadi
    pickle.dump(tree, dbfile)
    # menutup file setelah diisi
    dbfile.close()
```

Gambar 4.10: *Source Code* untuk Menyimpan *Tree*

Generalized Suffix Tree dari *title* yang sudah terbentuk harus disimpan agar bisa dilakukan pencarian pada tahap selanjutnya. Tahap pertama adalah membuat file dengan nama "gst" sebagai wadah objek *tree* yang akan disimpan. Setelah itu *tree* disimpan ke dalam file yang sudah dibuat. *Tree* disimpan sebagai objek pada direktori yang sama dengan kode.

3. *Generalized Suffix Tree* yang Terbentuk

Berikut ini potongan ilustrasi *tree* yang sudah terbentuk dari 10.686 data *title*.



Gambar 4.11: Ilustrasi Potongan *Tree* yang Terbentuk

Ilustrasi ini merupakan sebagian kecil dari pohon yang terbentuk. Seluruh sufiks dari data *title* dimasukkan ke dalam *tree*. Sebagai contoh pada ilustrasi di atas, sufiks *detikcom*, *detikconnect*, *detikfood*, *detikfinance* dan sufiks lainnya terdapat pada *tree*.

Berikut adalah *source code* untuk melihat kedalaman *tree* dan jumlah daun pada *tree*.

```
def loadData():
    # membuka file bernama gst
    dbfile = open('gst', 'rb')
    # load data tree yang berada di dalam file
    db = pickle.load(dbfile)
    dbfile.close()
    return db
```

Gambar 4.12: *Source Code* untuk Membaca *Tree*

Pada fungsi *loadData*, *tree* yang berada di dalam file objek bernama "gst" dibuka dan dikembalikan sebagai *output* dari fungsi.

```

read = loadData()
max_depth = len(list(LevelOrderGroupIter(read)))
print("maximum depth of tree: ", max_depth)
leaves = findall(read, filter_=lambda node: node.is_leaf)
num_leaves = len(leaves)
print("Number of leaves in the tree:", num_leaves)

```

Gambar 4.13: Source Code untuk Melihat Kedalaman dan Jumlah Daun pada Tree

Variable *read* menampung data *tree* yang sudah dibentuk. Untuk mencari jumlah maksimal kedalaman dari *tree*, fungsi bawaan *anytree* yaitu *LevelOrderGroupIter* digunakan untuk mengelompokkan semua *node* berdasarkan level pada *tree*. Lalu, dengan mengukur panjang *list* dari kelompok level tersebut didapatkan jumlah maksimal kedalaman *tree*. Untuk mencari jumlah daun pada *tree*, fungsi bawaan *anytree* yaitu *findall* digunakan untuk mencari *node* dengan filter *is leaf* yang akan mendeteksi apakah *node* tersebut adalah daun atau bukan. Lalu, dengan mengukur panjang *list* dari daun tersebut didapatkan jumlah daun yang terdapat pada *tree*.

```

maximum depth of tree: 11
Number of leaves in the tree: 34573

```

Gambar 4.14: Hasil Jumlah Maksimal Kedalaman Tree dan Jumlah Daun pada Tree

Dari 10.686 data *title* didapatkan jumlah maksimal kedalaman *tree* yang terbentuk yaitu 11 level dan jumlah maksimal daun pada *tree* sebanyak 34.573 daun.

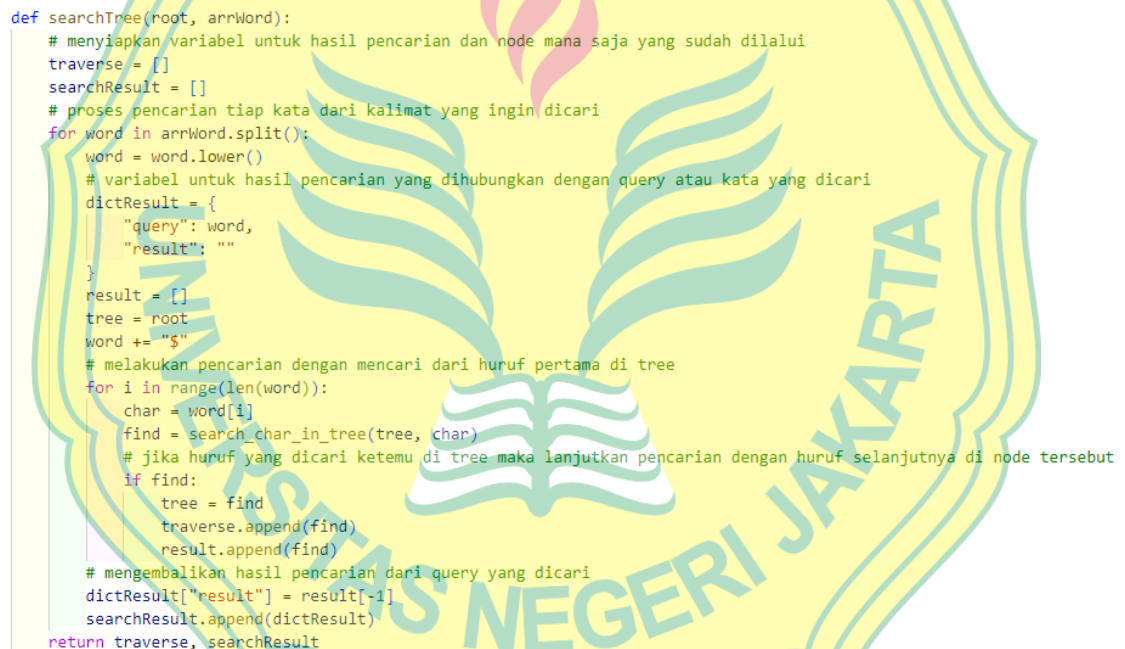
C. Mereduksi Node Redundan pada Tree

Penjelasan mengenai reduksi *node* redundan dijelaskan pada subbab 2E yaitu *Induced Generalized Suffix Tree*. Pada penelitian ini tahapan ini belum bisa diimplementasikan karena keterbatasan kemampuan dan waktu. Tahapan ini berperan dalam efisiensi segi ruang atau *space*. Seluruh tahapan *flowchart* pada penelitian ini masih bisa berjalan meskipun tahapan ini tidak terlaksana. Tahapan ini yang disebut dengan mengubah *Generalized Suffix Tree* menjadi *Induced Generalized Suffix Tree*. Karena tahapan ini tidak terlaksana maka *Generalized Suffix Tree* menjadi struktur data terakhir yang diimplementasikan pada penelitian ini.

D. Input Pola P

Terdapat sedikit perubahan pada *input* yang digunakan untuk melakukan pencarian. Pada *flowchart* terdapat *input* pola P dan batas k yaitu seberapa banyak dokumen yang ingin dikembalikan. Untuk membuat proses pencarian lebih simpel dan mengikuti alur natural pencarian maka *input* batas k dihilangkan dan dokumen yang dikembalikan adalah seluruh dokumen terkait dengan pencarian.

Setelah *tree* berhasil disimpan maka hal selanjutnya yang dilakukan adalah melakukan pencarian kata pada *tree* dengan *input* P atau kata yang dicari.



```
def searchTree(root, arrWord):
    # menyiapkan variabel untuk hasil pencarian dan node mana saja yang sudah dilalui
    traverse = []
    searchResult = []
    # proses pencarian tiap kata dari kalimat yang ingin dicari
    for word in arrWord.split():
        word = word.lower()
        # variabel untuk hasil pencarian yang dihubungkan dengan query atau kata yang dicari
        dictResult = {
            "query": word,
            "result": ""
        }
        result = []
        tree = root
        word += "$"
        # melakukan pencarian dengan mencari dari huruf pertama di tree
        for i in range(len(word)):
            char = word[i]
            find = search_char_in_tree(tree, char)
            # jika huruf yang dicari ketemu di tree maka lanjutkan pencarian dengan huruf selanjutnya di node tersebut
            if find:
                tree = find
                traverse.append(find)
                result.append(find)
        # mengembalikan hasil pencarian dari query yang dicari
        dictResult["result"] = result[-1]
        searchResult.append(dictResult)
    return traverse, searchResult
```

Gambar 4.15: Source Code untuk Mencari Kata di Tree

Untuk pola kalimat P yang dicari, pencarian akan dilakukan huruf per huruf mulai dari kata pertama dari seluruh kata yang ada di kalimat *input*. Jika huruf yang dicari ditemukan di *tree*, cari huruf selanjutnya menggunakan *node* yang ditemukan sebagai awal mula tempat dilakukan pencarian. Ulangi langkah-langkah hingga seluruh kata yang dicari ditemukan. Hasil dari fungsi *searchTree* adalah *traverse* yaitu variabel yang menampung *node* mana saja yang sudah dilalui pada pencarian dan *searchResult* yaitu variabel yang menampung *node* hasil pencarian.

```
# fungsi untuk mencari sebuah karakter huruf pada tree
def search_char_in_tree(node, char):
    # pencarian huruf pada anak dari tree
    for child in node.children:
        name = child.name
        # jika ada yang sama maka kembalikan node tersebut
        if name == char or name[0] == char:
            return child
    return False
```

Gambar 4.16: Source Code untuk Mencari Karakter di Tree

Pada fungsi *search char in tree*, pencarian huruf dilakukan dimulai dari level *tree* paling atas dan jika ada *node* yang berawalan huruf sesuai dengan yang dicari atau huruf yang dicari merupakan *node* tersebut maka *node* tersebut dikembalikan.

Berikut adalah contoh hasil pencarian untuk *query* *resesi 2023*:

```
hasil pencarian: {'query': 'resesi', 'result': Node('/root/r/e/s/i/s', index=[352, 805, 807, 813, 829, 834, 2061, 2072, 3816, 4147, 4402, 4415, 4645, 4804, 4895, 4898, 4900, 4901, 4903, 4904, 4905, 4906, 4907, 4908, 4926, 9878, 10185, 10501, 10502, 10593, 10607, 10608, 10609, 10645, 10646, 10647])}
*****
hasil pencarian: {'query': '2023', 'result': Node('/root/2/0/2/3', index=[85, 301, 316, 325, 335, 527, 566, 567, 637, 649, 660, 895, 970, 1016, 1212, 1375, 1421, 1952, 1984, 2253, 2630, 2635, 3003, 3059, 3083, 3084, 3085, 3082, 3898, 3907, 3923, 3926, 4042, 4054, 4055, 4137, 4384, 4400, 4493, 4497, 4498, 4510, 4590, 4591, 4592, 4593, 4679, 4680, 4757, 4903, 4928, 4944, 4958, 4959, 4990, 5085, 5630, 5841, 5937, 5938, 5953, 6488, 6517, 6575, 6613, 6628, 6631, 6638, 6640, 6645, 6662, 6664, 6665, 6711, 6773, 6874, 6883, 6707, 6710, 6883, 6886, 7414, 7415, 7416, 7417, 7418, 7419, 7420, 7421, 7422, 7423, 7424, 7425, 7785, 7794, 7799, 7812, 7946, 8234, 8312, 8247, 8382, 8475, 8481, 8482, 8550, 8619, 8679])}
```

Gambar 4.17: Contoh Hasil Pencarian dengan *Query* *Resesi 2023*

Untuk hasil pencarian berupa *array* berisi *dictionary*. *Key* dari *dictionary* hasil adalah *query*, *result*, dan *index*. Pada gambar di atas, elemen pertama dari *array* adalah *dictionary* hasil dari kata *resesi* dengan *result* berupa *node* dari *tree* dan *index* yang berisi *id* dokumen yang pada judulnya terdapat kata *resesi*. Elemen kedua dari *array* adalah *dictionary* hasil dari kata *2023* dengan *result* berupa *node* dari *tree* dan *index* yang berisi *id* dokumen yang pada judulnya terdapat kata *2023*.

E. Mencari Nilai Count

Pada tahapan ini nilai *count* berfungsi sebagai parameter peringkat pada tiap *node* atau hasil pencarian sesuai dengan deskripsi yang dijelaskan pada bagian 2G di bab 2 mengenai konstruksi algoritma pengindeksan. Setelah berhasil mendapatkan hasil pencarian dari kalimat yang dimasukkan, hasil pencarian perlu dilakukan pemeringkatan terkait relevansi hasil pencarian. Dengan menghitung nilai *count* atau jumlah kemunculan kata pada judul dan mengakumulasi jumlah dari tiap kata tersebut maka dokumen dengan relevansi tertinggi akan dihasilkan.

```
def rankResult(result):
    # inisiasi variabel untuk menyimpan index dari hasil pencarian untuk dihitung
    allListDocument = []
    listCount = []
    for i in result:
        allListDocument.append(i["result"].index)
    # hitung nilai count untuk setiap indeks
    for i in range(len(allListDocument)):
        for idx in allListDocument[i]:
            # cek apakah setiap kata memiliki indeks yang sama
            status, sameIdx = checkList(idx, listCount)
            # jika ada indeks yang sama maka tambahkan nilai count
            if len(listCount) > 0 and status == True:
                listCount[sameIdx]["count"] += 1
                listCount[sameIdx]["query"] += " " + result[i]["query"]
                continue
            # variabel hasil penghitungan count untuk setiap indeks hasil pencarian
            obj = {
                "index": idx,
                "count": 1,
                "query": result[i]["query"]
            }
            listCount.append(obj)
    # urutkan indeks dari jumlah count terbesar sehingga dokumen pertama adalah yang paling relevan
    rankedList = sorted(listCount, key=lambda d: d['count'], reverse=True)
    return rankedList
```

Gambar 4.18: Source Code untuk Melakukan Pemeringkatan Hasil Pencarian

Dari hasil pencarian yang belum dilakukan pemeringkatan sebelumnya, *key index* diambil untuk dilakukan perhitungan nilai *count*. Untuk setiap kata *query* akan diperiksa apabila ada *index* yang sama maka nilai *count* akan ditambah. Nilai *count* lebih dari satu pada *index* berarti bahwa ada kemunculan pola *query* lain pada judul dokumen tersebut. Setelah semua *index* hasil diberikan nilai *count*, *array index* akan diurutkan berdasarkan nilai *count* terbesar.

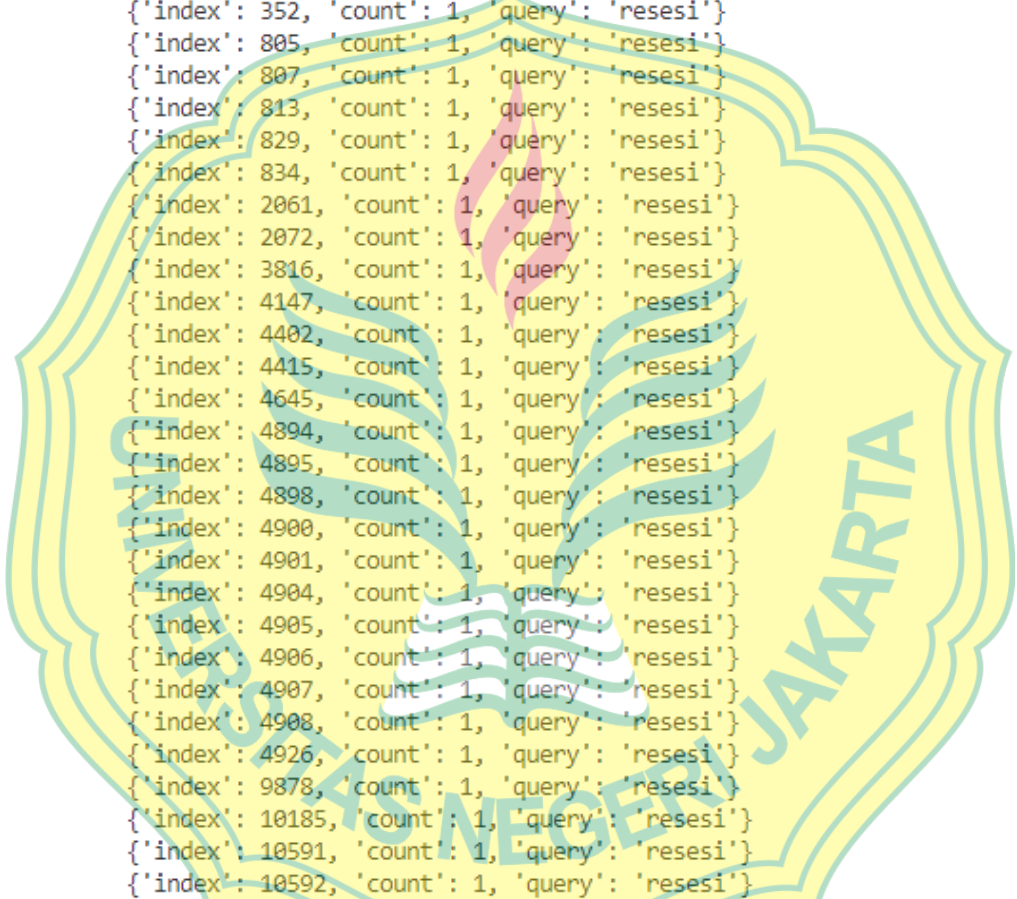
```
def checkList(index, arr):
    for i in range(len(arr)):
        if arr[i]["index"] == index:
            return True, i
    return False, 0
```

Gambar 4.19: Source Code untuk Melakukan Pemeringkatan Hasil Pencarian

Fungsi *check list* digunakan untuk memeriksa apakah suatu indeks muncul pada hasil pencarian kata yang lain. Jika pada *array* hasil lain terdapat *index* yang

sama maka status *True* dikembalikan serta *index* tersebut dikembalikan.

Berikut adalah *array* hasil perhitungan nilai *count* untuk setiap *index* yang sudah diurutkan:



```

masukkan kata yang ingin dicari: resesi 2023
{'index': 4903, 'count': 2, 'query': 'resesi 2023'}
{'index': 352, 'count': 1, 'query': 'resesi'}
{'index': 805, 'count': 1, 'query': 'resesi'}
{'index': 807, 'count': 1, 'query': 'resesi'}
{'index': 813, 'count': 1, 'query': 'resesi'}
{'index': 829, 'count': 1, 'query': 'resesi'}
{'index': 834, 'count': 1, 'query': 'resesi'}
{'index': 2061, 'count': 1, 'query': 'resesi'}
{'index': 2072, 'count': 1, 'query': 'resesi'}
{'index': 3816, 'count': 1, 'query': 'resesi'}
{'index': 4147, 'count': 1, 'query': 'resesi'}
{'index': 4402, 'count': 1, 'query': 'resesi'}
{'index': 4415, 'count': 1, 'query': 'resesi'}
{'index': 4645, 'count': 1, 'query': 'resesi'}
{'index': 4894, 'count': 1, 'query': 'resesi'}
{'index': 4895, 'count': 1, 'query': 'resesi'}
{'index': 4898, 'count': 1, 'query': 'resesi'}
{'index': 4900, 'count': 1, 'query': 'resesi'}
{'index': 4901, 'count': 1, 'query': 'resesi'}
{'index': 4904, 'count': 1, 'query': 'resesi'}
{'index': 4905, 'count': 1, 'query': 'resesi'}
{'index': 4906, 'count': 1, 'query': 'resesi'}
{'index': 4907, 'count': 1, 'query': 'resesi'}
{'index': 4908, 'count': 1, 'query': 'resesi'}
{'index': 4926, 'count': 1, 'query': 'resesi'}
{'index': 9878, 'count': 1, 'query': 'resesi'}
{'index': 10185, 'count': 1, 'query': 'resesi'}
{'index': 10591, 'count': 1, 'query': 'resesi'}
{'index': 10592, 'count': 1, 'query': 'resesi'}
{'index': 10593, 'count': 1, 'query': 'resesi'}
{'index': 10607, 'count': 1, 'query': 'resesi'}
{'index': 10608, 'count': 1, 'query': 'resesi'}

```

Gambar 4.20: Contoh Array Hasil Pemeringkatan *Index* Berdasarkan Nilai *Count*

Array berisi *dictionary* telah diurutkan berdasarkan nilai *count* dengan *key index*, *count*, dan *query*. Pada *query* resesi 2023, *index* atau *id dokumen* teratas dengan *count* 2 mengandung *query* resesi 2023. Selanjutnya, *index* dengan *count* lebih kecil ditampilkan sesuai urutan. Untuk nilai *count* yang sama atau senilai maka relevansi dokumen dinilai sama atau setara karena terdapat pola *query* pada judul sesuai nilai *count*.

F. Merepresentasikan Array Hasil Berdasarkan Nilai Count

Berikut adalah *source code* untuk menampilkan hasil pencarian:

```
def getResult(result):
    # connect ke database
    connection = pymysql.connect(host='localhost',
                                user='root',
                                password='',
                                database='crawl3',
                                charset='utf8mb4',
                                cursorclass=pymysql.cursors.DictCursor,
                                autocommit=True)

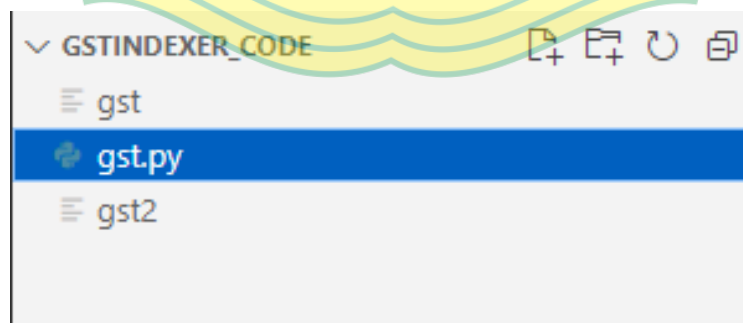
    with connection:
        for i in range(5):
            with connection.cursor() as cursor:
                cursor.execute(
                    "SELECT id_page, title, url FROM page_information WHERE id_page = %s", (result[i]["index"]))
            data = cursor.fetchall()
            # tampilkan data ke terminal
            print("query: " + result[i]["query"])
            print("count: " + str(result[i]["count"]))
            print(str(i+1) + ". " + data[0]["title"])
            print(data[0]["url"])
            print("\n")
```

Gambar 4.21: Source Code untuk Merepresentasikan Array Hasil Berdasarkan Nilai Count

Array hasil pemeringkatan yang sebelumnya sudah diurutkan berdasarkan nilai *count* ditampilkan dengan detail untuk *query*, nilai *count*, judul halaman web, dan URL dari hasil pencarian. 5 data hasil pencarian teratas ditampilkan ke *console*.

G. Struktur Direktori Kode dan Tree

Berikut adalah struktur direktori dari hasil penelitian ini:



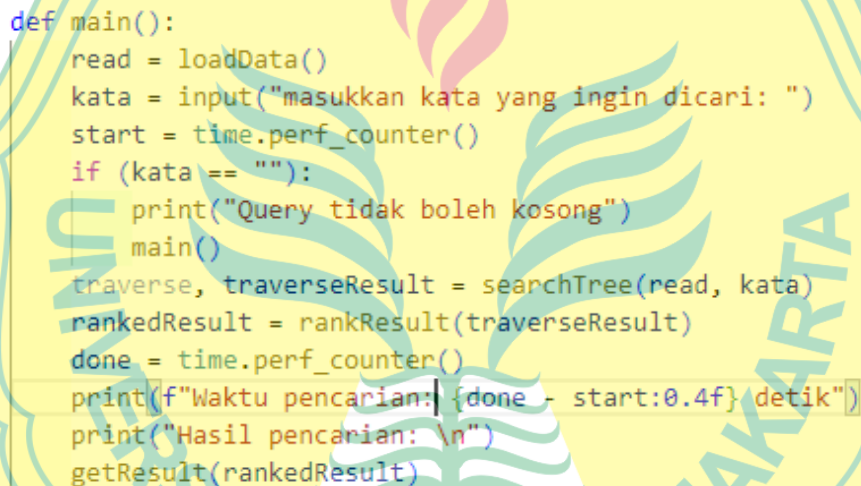
Gambar 4.22: Struktur Direktori Kode dan Tree

Struktur direktori dari kode dan *tree* mengikuti struktur dari penelitian induk yaitu PERANCANGAN ARSITEKTUR SEARCH ENGINE DENGAN

MENGINTEGRASIKAN *WEB CRAWLER*, *ALGORITMA PAGE RANKING*, DAN *DOCUMENT RANKING* oleh Lazuardy Khatulistiwa. Pada direktori atau folder *gstindexer code* tersimpan kode dengan nama file *gst.py*, *Generalized Suffix Tree* yang sudah terbentuk dalam file *gst*, dan file *gst2* berisi *tree* cadangan atau *backup*.

H. Hasil Pengujian

Setelah semua komponen sudah dijalankan, hal yang selanjutnya dilakukan adalah melakukan pencarian dengan beberapa kalimat dan panjang yang berbeda.



```
def main():
    read = loadData()
    kata = input("masukkan kata yang ingin dicari: ")
    start = time.perf_counter()
    if (kata == ""):
        print("Query tidak boleh kosong")
        main()
    traverse, traverseResult = searchTree(read, kata)
    rankedResult = rankResult(traverseResult)
    done = time.perf_counter()
    print(f"Waktu pencarian: {done - start:0.4f} detik")
    print("Hasil pencarian: \n")
    getResult(rankedResult)
```

Gambar 4.23: Source Code Fungsi Main

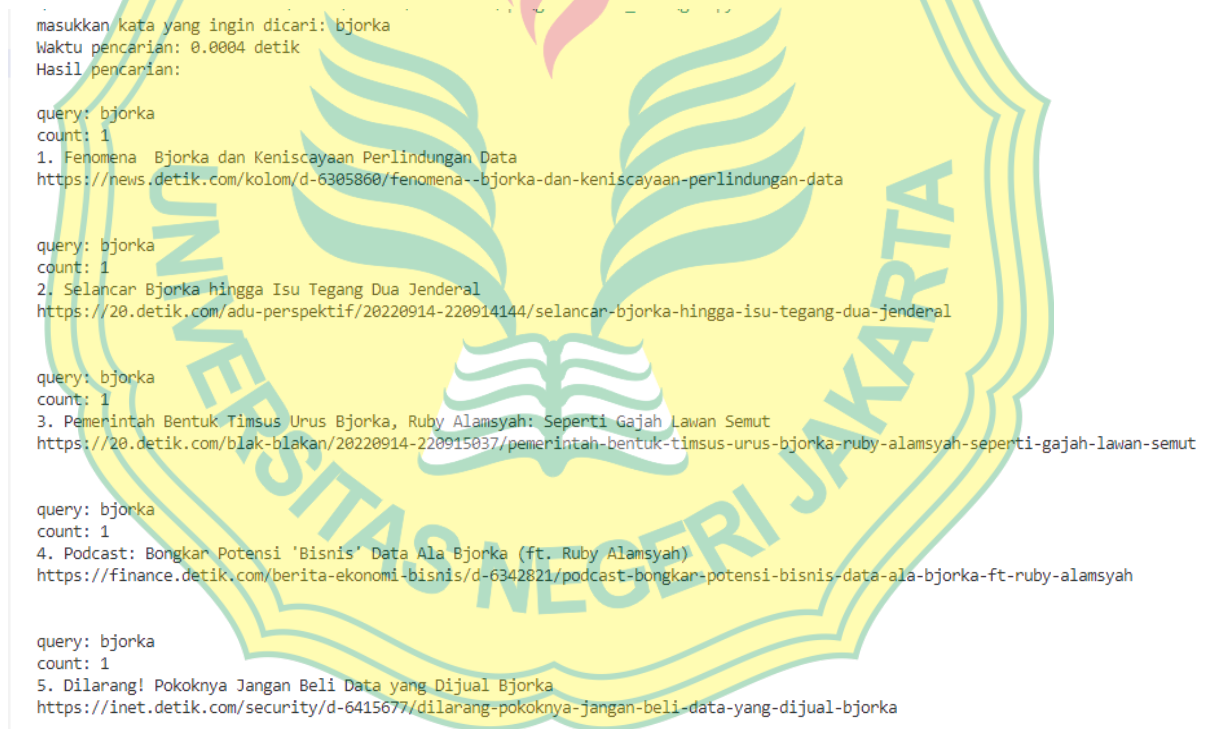
Fungsi utama atau *main* digunakan untuk membaca data *tree* pada file *gst*, menerima *input*, melakukan pencarian pada *tree*, melakukan pemeringkatan hasil pencarian dari *input* dan menampilkan hasil pencarian berupa daftar halaman web yang relevan sesuai pencarian. Urutan yang ditampilkan pada *console* merupakan hasil pencarian yang sudah dilakukan pemeringkatan kecuali nilai *count* dari hasil pencariannya sama maka relevansi atau peringkatnya sama.

Pencarian dilakukan dengan 3 skema *input* berbeda yaitu dengan panjang 1 kata, 2 kata, dan 2 *input* dengan kata yang salah ketik sehingga total ada 8 pencarian yang dilaksanakan. Seluruh *input* tersebut terdiri dari:

Tabel 4.1: *Input yang digunakan*

1 kata	2 kata
<i>bjorka</i>	<i>resesi 2023</i>
<i>messi</i>	<i>bitcoin naik</i>
<i>masak</i>	<i>world cup</i>
<i>teknolpgi</i>	
<i>reseso</i>	

Hasil pencarian dengan *keyword bjorka* dapat dilihat pada gambar berikut:

**Gambar 4.24:** Hasil Pencarian *Keyword Bjorka*

Dari gambar di atas untuk pencarian *keyword bjorka* didapatkan dalam waktu 0,0004 detik dan dokumen yang dikembalikan mengandung judul dengan *query* terkait serta nilai *count* yang sesuai.

Hasil pencarian dengan *keyword messi* dapat dilihat pada gambar berikut:

masukkan kata yang ingin dicari: messi
Waktu pencarian: 0.0005 detik
Hasil pencarian:

query: messi
count: 1

1. Cristiano Ronaldo vs Lionel Messi, Mana yang Lebih Baik?
<https://inet.detik.com/science/d-6429055/cristiano-ronaldo-vs-lionel-messi-mana-yang-lebih-baik>

query: messi
count: 1

2. Lionel Messi Dielu-elukan Netizen Bawa Asa Argentina di Piala Dunia 2022
<https://inet.detik.com/fotoinet/d-6428676/lionel-messi-dielu-elukan-netizen-bawa-asa-argentina-di-piala-dunia-2022>

query: messi
count: 1

3. Kemarin Ronaldo Bikin Rekor di Piala Dunia 2022, Sekarang Messi
<https://sport.detik.com/sepakbola/bola-dunia/d-6429267/kemarin-ronaldo-bikin-rekor-di-piala-dunia-2022-sekarang-messi>

query: messi
count: 1

4. Lionel Messi: Argentina Main Sesuai Keinginan
<https://sport.detik.com/sepakbola/bola-dunia/d-6428991/lionel-messi-argentina-main-sesuai-keinginan>

query: messi
count: 1

5. Foto: Messi Jaga Asa Tim Tango ke Babak 16 Besar Piala Dunia 2022
<https://sport.detik.com/sepakbola/foto-sepakbola/d-6428782/foto-messi-jaga-asa-tim-tango-ke-babak-16-besar-piala-dunia-2022>

Gambar 4.25: Hasil Pencarian *Keyword Messi*

Dari gambar di atas untuk pencarian *keyword messi* didapatkan dalam waktu 0,0005 detik dan dokumen yang dikembalikan mengandung judul dengan *query* terkait serta nilai *count* yang sesuai.

Hasil pencarian dengan *keyword masak* dapat dilihat pada gambar berikut:

masukkan kata yang ingin dicari: masak
Waktu pencarian: 0.0015 detik
Hasil pencarian:

query: masak
count: 1

1. Hitung-hitungan Masak Nasi Pakai Rice Cooker vs LPG, Mana yang Lebih Murah?
<https://finance.detik.com/energi/d-6426121/hitung-hitungan-masak-nasi-pakai-rice-cooker-vs-lpg-mana-yang-lebih-murah>

query: masak
count: 1

2. Video Resep & Tips Masak Memasak | detikFood
<https://food.detik.com/video>

query: masak
count: 1

3. Hobi Masak, Ini Keseruan Zaskia Sungkar Belanja di Pasar hingga Masak
<https://food.detik.com/foto-kuliner/d-6426964/hobi-masak-ini-keseruan-zaskia-sungkar-belanja-di-pasar-hingga-masak>

query: masak
count: 1

4. Kumpulan Berita Video Masak Apa Terkini dan Terbaru di 20detik
<https://20.detik.com/program/160510479/masak-apa>

query: masak
count: 1

5. Hitung-hitungan Masak Nasi Pakai Rice Cooker vs LPG, Mana yang Lebih Murah?
<https://finance.detik.com/energi/d-6426121/hitung-hitungan-masak-nasi-pakai-rice-cooker-vs-lpg-mana-yang-lebih-murah#comm1>

Gambar 4.26: Hasil Pencarian *Keyword Masak*

Dari gambar di atas untuk pencarian *keyword masak* didapatkan dalam waktu 0,0015 detik dan dokumen yang dikembalikan mengandung judul dengan *query* terkait serta nilai *count* yang sesuai.

Hasil pencarian dengan *keyword* *resesi 2023* dapat dilihat pada gambar berikut:

masukkan kata yang ingin dicari: *resesi 2023*

Waktu pencarian: 0.0046 detik

Hasil pencarian:

query: *resesi 2023*

count: 2

1. 2023, Ancaman Selain Resesi: Pandemi Lagi

<https://finance.detik.com/berita-ekonomi-bisnis/d-6416994/2023-ancaman-selain-resesi-pandemi-lagi>

query: *resesi*

count: 1

2. Berita dan Informasi Resesi Terkini dan Terbaru Hari ini - detikcom

<https://www.detik.com/tag/resesi>

query: *resesi*

count: 1

3. 4 Negara Ini Dihantui Resesi Seks, Terancam Krisis Populasi

<https://health.detik.com/fotohealth/d-6429028/4-negara-ini-dihantui-resesi-seks-terancam-krisis-populasi>

query: *resesi*

count: 1

4. Korsel Dibayangi Resesi Seks, Begini Pengakuan Warga yang Ogah Punya Anak

<https://health.detik.com/berita-detikhealth/d-6428985/korsel-dibayangi-resesi-seks-begini-pengakuan-warga-yang-ogah-punya-anak>

query: *resesi*

count: 1

5. 4 Negara Asia Ini Terancam Krisis Populasi Imbas Resesi Seks

<https://health.detik.com/berita-detikhealth/d-6428388/4-negara-asia-ini-terancam-krisis-populasi-imbis-resesi-seks>

Gambar 4.27: Hasil Pencarian *Keyword Resesi 2023*

Dari gambar di atas untuk pencarian *keyword resesi 2023* didapatkan dalam waktu 0,0046 detik dan dokumen yang dikembalikan mengandung judul dengan *query* yang berhubungan serta nilai *count* yang sesuai.

Hasil pencarian dengan *keyword bitcoin naik* dapat dilihat pada gambar berikut:

masukkan kata yang ingin dicari: bitcoin naik

Waktu pencarian: 0.0023 detik

Hasil pencarian:

query: bitcoin

count: 1

1. Info Berita Terkini Fintech, Startup, Bitcoin - 1

<https://finance.detik.com/fintech>

query: bitcoin

count: 1

2. Tech - Berita Terkini Tech, Startup, Bitcoin - CNBC Indonesia

<https://www.cnbcindonesia.com/tech>

query: bitcoin

count: 1

3. Info Berita Terkini Fintech, Startup, Bitcoin - 1

<https://finance.detik.com/fintech#major>

query: bitcoin

count: 1

4. Info Berita Terkini Fintech, Startup, Bitcoin - 1

<https://finance.detik.com/fintech#indofx>

query: bitcoin

count: 1

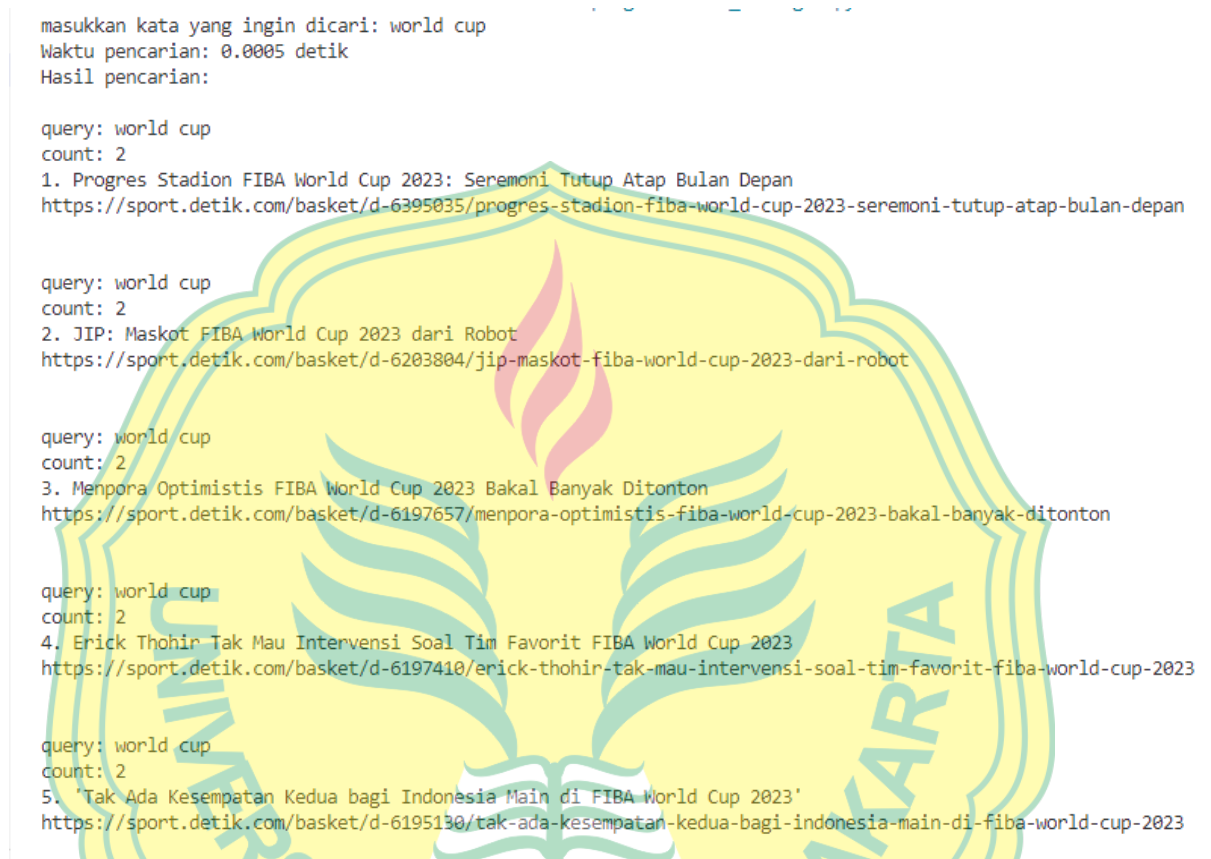
5. Info Berita Terkini Fintech, Startup, Bitcoin - 1

<https://finance.detik.com/fintech#usdfx>

Gambar 4.28: Hasil Pencarian *Keyword Bitcoin Naik*

Dari gambar di atas untuk pencarian *keyword bitcoin naik* didapatkan dalam waktu 0,0023 detik dan dokumen yang dikembalikan mengandung judul dengan *query* yang berhubungan serta nilai *count* yang sesuai.


Hasil pencarian dengan *keyword world cup* dapat dilihat pada gambar berikut:



Gambar 4.29: Hasil Pencarian *Keyword World Cup*

Dari gambar di atas untuk pencarian *keyword world cup* didapatkan dalam waktu 0,0005 detik dan dokumen yang dikembalikan mengandung judul dengan *query* yang berhubungan serta nilai *count* yang sesuai.

Hasil pencarian dengan *keyword teknologi* yang salah ketik menjadi *teknolpgi* dapat dilihat pada gambar berikut:



```

masukkan kata yang ingin dicari: teknolpgi
Waktu pencarian: 0.0012 detik
Hasil pencarian:

query: teknolpgi
count: 1
1. detikInet - Berita Teknologi Informasi Gadget Terbaru Hari ini
https://inet.detik.com

query: teknolpgi
count: 1
2. Informasi Seputar Gadget dan Teknologi Terkini - 1
https://inet.detik.com/gadget

query: teknolpgi
count: 1
3. Berita Seputar Teknologi dan Informasi Terkini - 1
https://inet.detik.com/news

query: teknolpgi
count: 1
4. Berita dan Informasi science dan teknologi informasi - 1
https://inet.detik.com/science

query: teknolpgi
count: 1
5. Berita Seputar industri Telekomunikasi dan Teknologi Terbaru - 1
https://inet.detik.com/telecommunication

```

Gambar 4.30: Hasil Pencarian *Keyword Teknologi* Salah Ketik

Dari gambar di atas untuk pencarian *keyword teknolpgi* didapatkan dalam waktu 0,0012 detik dan dokumen yang dikembalikan mengandung judul dengan *query* yang berhubungan dengan konteks asli serta nilai *count* yang sesuai.

Hasil pencarian dengan *keyword* *resesi* yang salah ketik menjadi *reseso* dapat dilihat pada gambar berikut:

```

masukkan kata yang ingin dicari: reseso
Waktu pencarian: 0.0001 detik
Hasil pencarian:

query: reseso
count: 1
1. Pimpinan DPR Upayakan Proses Cepat Calon Panglima TNI Sebelum Reses
https://news.detik.com/berita/d-6422084/pimpinan-dpr-upayakan-proses-cepat-calon-panglima-tni-sebelum-reses

```

Gambar 4.31: Hasil Pencarian *Keyword Resesi* Salah Ketik

Dari gambar di atas untuk pencarian *keyword* *reseso* didapatkan dalam waktu 0,0001 detik dan dokumen yang dikembalikan tidak mengandung judul dengan *query* yang berhubungan meskipun nilai *count* sesuai.

1. Pengujian *Mean Average Precision*

Sesuai dengan skenario pengujian di bab 3, modul pengindeks akan diuji relevansinya oleh 3 orang *tester*. Hasil pengujian relevansi menggunakan metode *Mean Average Precision* dari modul pengindeks adalah sebagai berikut:

Tabel 4.2: Hasil Pengujian *Mean Average Precision* Mengenai Relevansi *Query*

No.	Query	AP Tester 1	AP Tester 2	AP Tester 3
1	"bjorka"	1	1	1
2	"messi"	1	1	1
3	"masak"	0.4	1	1
4	"resesi 2023"	0.2	0.2	0.4
5	"bitcoin naik"	0.2	1	1
6	"world cup"	0	1	1
7	"teknolpgi"	0.8	1	1
8	"reseso"	0	0	0
	MAP	0.45	0.775	0.8

Berdasarkan hasil pengujian *Mean Average Precision* mengenai relevansi *query*, MAP *tester 1* untuk seluruh *query* adalah 0.45. MAP *tester 2* untuk seluruh *query* adalah 0.775. MAP *tester 3* untuk seluruh *query* adalah 0.8. Untuk menghitung total skor MAP dari ketiga *tester* maka seluruh nilai MAP akan

dijumlah dan dibagi dengan 3 sesuai dengan jumlah *tester*. Total skor MAP = $(0.45 + 0.775 + 0.8)/3 = 0.658$

Total skala penilaian MAP adalah dari rentang 0 sampai 1 sehingga untuk nilai 0.658 dapat disimpulkan bahwa relevansi modul pengindeks tergolong cukup relevan namun perlu ditingkatkan lagi.

2. Pengujian Performa Kecepatan (*Speed Performance Testing*)

Berdasarkan hasil pencarian dari gambar 4.24 hingga gambar 4.31 dapat dilihat bahwa waktu pencarian di bawah atau kurang dari hitungan detik. Hal ini menunjukkan bahwa waktu pengindeksan dari mulai awal proses *input* hingga pengembalian hasil terbilang cepat. Berdasarkan skenario pengujian pada bab 3 dan hasil pengujian di atas maka modul pengindeks dapat dikatakan sesuai dengan rancangan atau dengan kata lain berhasil.

I. Analisis Hasil

Berikut analisis hasil berdasarkan pengujian yang telah dilakukan:

1. Pencarian berhasil dilakukan dalam hitungan detik untuk seluruh *input* pengujian.
2. Nilai total *Mean Average Precision* dari 3 *tester* sebesar 0.658 menandakan dokumen yang ditampilkan cukup relevan dengan yang diinginkan oleh *tester* namun perlu ditingkatkan lagi.

Pencarian dengan 1 kata menampilkan dokumen dengan relevansi yang baik yaitu dokumen dengan judul yang mengandung kata tersebut dan nilai *Average Precision* dari ketiga *tester* mendekati maksimal. Pencarian dengan 2 kata pada *input resesi 2023* menghasilkan daftar dokumen dengan relevansi kurang memuaskan karena nilai *Average Precision* dari ketiga *tester* tidak ada yang maksimal. Pencarian dengan 2 kata pada *input bitcoin naik* hanya mengembalikan daftar dokumen dengan judul yang mengandung kata *bitcoin* karena pada *dataset* tidak ada dokumen dengan judul yang mencakup kedua kata bersamaan. Pencarian dengan 2 kata pada *input world cup* tergolong baik karena nilai *Average Precision* dari 2 *tester* maksimal sementara untuk 1 *tester* nilainya 0.

Pencarian dengan kata yang salah ketik pada *input teknolpgi* menghasilkan daftar dokumen dengan relevansi sangat baik di mana daftar dokumen yang

ditampilkan adalah dokumen yang mengandung kata *teknologi* dan nilai *Average Precision* dari ketiga *tester* mendekati maksimal. Dengan kata lain, dokumen yang dikembalikan sesuai dengan yang dimaksud. Pencarian dengan kata yang salah ketik pada *input reseso* hanya mengembalikan dokumen yang mencakup kata *reses* di dalamnya karena pencarian kata pada *tree* dimulai dari depan dan hanya bisa mengembalikan *longest common substring* atau *substring* terpanjang yang sesuai dengan *node* di *tree* sehingga nilai *Average Precision* dari ketiga *tester* adalah 0.

J. Repository Kode

Seluruh kode dan file dapat diakses melalui
<https://github.com/zaidanprtm/gstindexer>



BAB V

KESIMPULAN DAN SARAN

A. Kesimpulan

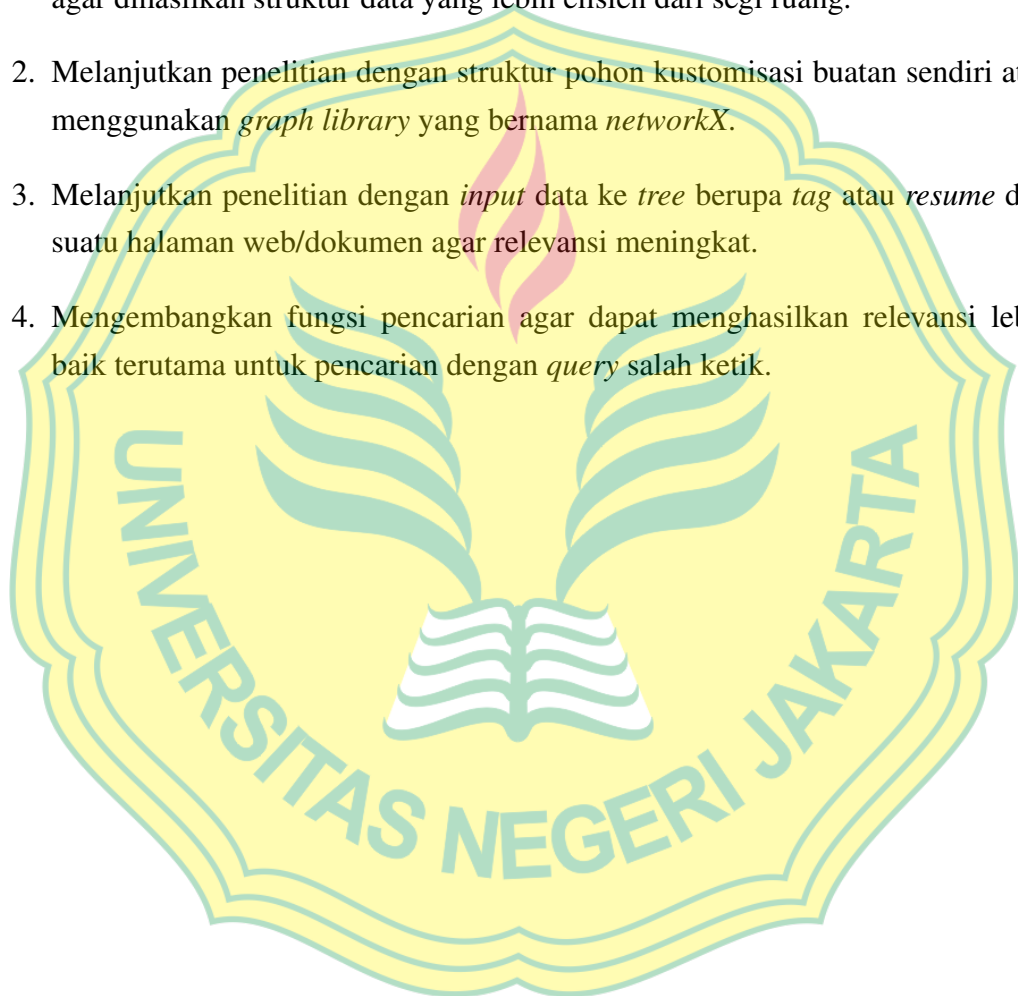
Berdasarkan hasil penelitian yang sudah dilakukan dapat ditarik kesimpulan sebagai berikut:

1. Struktur direktori serta dataset dari penelitian induk telah diikuti untuk membuat modul pengindeks berupa *Generalized Suffix Tree*.
2. Data *title* berjumlah 10.686 *rows* dari tabel *page information* berhasil digunakan untuk membuat *Generalized Suffix Tree* dengan kedalaman maksimal 11 level serta jumlah daun sebanyak 34.573 daun sebagai struktur data indeks yang dapat dimanfaatkan untuk pencarian kata atau *query*.
3. Modul pengindeks berupa *Generalized Suffix Tree* untuk keperluan pemeringkatan dokumen pada *search engine* berhasil dibuat dengan menggunakan *naive algorithm* dengan notasi waktu $O(n^2)$.
4. Pencarian dan pemeringkatan dokumen hasil pencarian berjalan dengan relevansi cukup baik dengan nilai total *Mean Average Precision* sebesar 0.658 dari ketiga *tester* di mana daftar dokumen dengan judul yang mencakup kata *query* ditampilkan dan waktu rata-rata pencarian berhasil didapatkan dalam hitungan detik.
5. Pencarian dengan *query* salah ketik bekerja cukup baik pada kata *teknolpgi* karena *substring* yang paling panjang yang bisa ditemui dan dikembalikan adalah *teknol* sehingga seluruh dokumen dengan kata *teknologi* bisa dikembalikan.
6. Pencarian dengan *query* salah ketik pada kata *reseso* tidak menghasilkan daftar dokumen yang diinginkan karena *substring* yang paling panjang yang bisa ditemui dan dikembalikan adalah *reses* sehingga hanya dokumen dengan kata *reses* yang ditampilkan.

B. Saran

Adapun saran untuk penelitian selanjutnya adalah:

1. Melanjutkan penelitian dengan struktur data *Induced Generalized Suffix Tree* agar dihasilkan struktur data yang lebih efisien dari segi ruang.
2. Melanjutkan penelitian dengan struktur pohon kustomisasi buatan sendiri atau menggunakan *graph library* yang bernama *networkX*.
3. Melanjutkan penelitian dengan *input* data ke *tree* berupa *tag* atau *resume* dari suatu halaman web/dokumen agar relevansi meningkat.
4. Mengembangkan fungsi pencarian agar dapat menghasilkan relevansi lebih baik terutama untuk pencarian dengan *query* salah ketik.



DAFTAR PUSTAKA

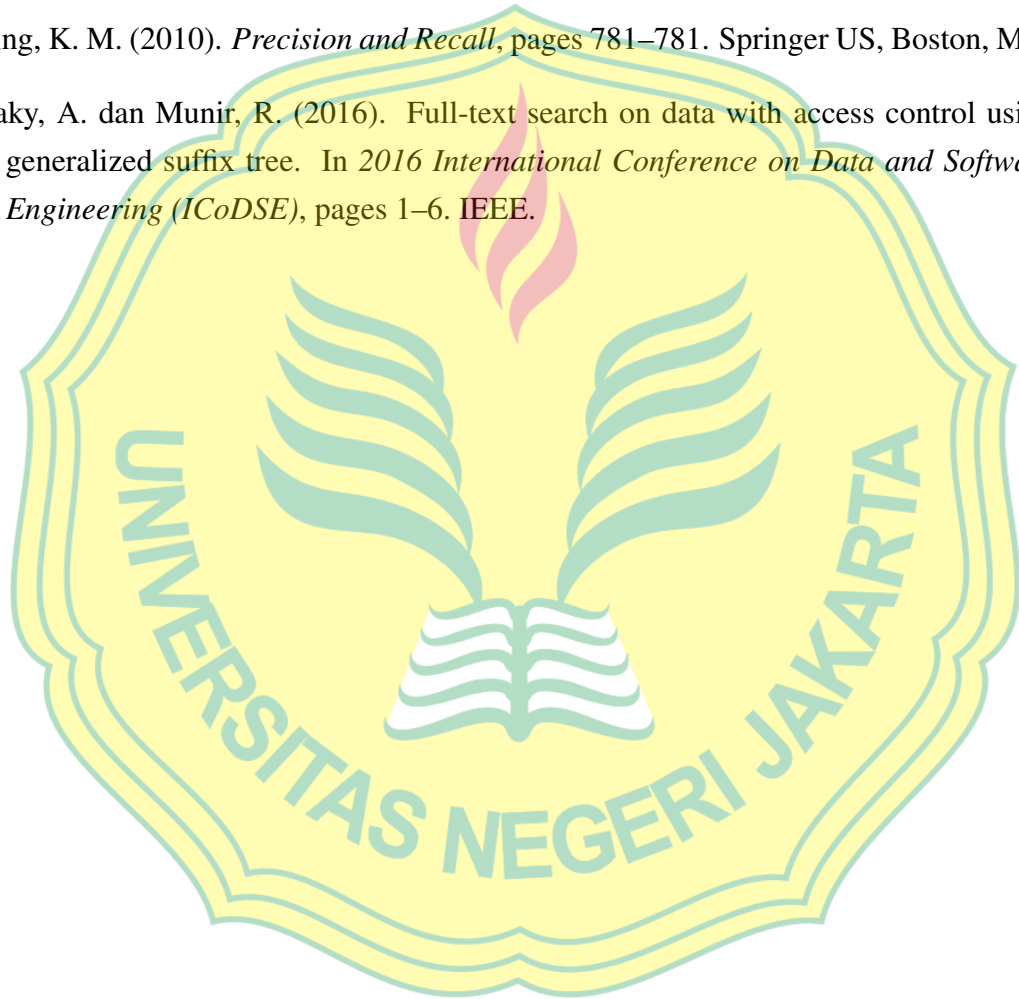
- Al Aziz, M. M., Thulasiraman, P., dan Mohammed, N. (2022). Parallel and private generalized suffix tree construction and query on genomic data. *BMC genomic data*, 23(1):1–16.
- Beitzel, S. M., Jensen, E. C., dan Frieder, O. (2009). *MAP*, pages 1691–1692. Springer US, Boston, MA.
- Brin, S., Motwani, R., Page, L., dan Winograd, T. (1998). What can you do with a web in your pocket? *IEEE Data Eng. Bull.*, 21(2):37–47.
- Brin, S. dan Page, L. (1998). The anatomy of a large-scale hypertextual web search engine.
- Chi, L. dan Hui, K. (1992). Color set size problem with applications to string matching. In *Annual Symposium on Combinatorial Pattern Matching*, pages 230–243. Springer.
- Harman, D. dkk. (2019). Information retrieval: the early years. *Foundations and Trends® in Information Retrieval*, 13(5):425–577.
- Hon, W.-K., Patil, M., Shah, R., dan Wu, S.-B. (2010). Efficient index for retrieving top-k most frequent documents. *Journal of Discrete Algorithms*, 8(4):402–417.
- Mahdi, M. S. R., Al Aziz, M. M., Mohammed, N., dan Jiang, X. (2021). Privacy-preserving string search on encrypted genomic data using a generalized suffix tree. *Informatics in Medicine Unlocked*, 23:100525.
- McCreight, E. M. (1976). A space-economical suffix tree construction algorithm. *Journal of the ACM (JACM)*, 23(2):262–272.
- Muthukrishnan, S. (2002). Efficient algorithms for document retrieval problems. In *SODA*, volume 2, pages 657–666. Citeseer.
- Puglisi, S. J. dan Zhukova, B. (2021). Document retrieval hacks. In *19th International Symposium on Experimental Algorithms (SEA 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.
- Qoriiba, M. F. (2021). Perancangan crawler sebagai pendukung pada search engine.

Seymour, T., Frantsvog, D., Kumar, S., dkk. (2011). History of search engines. *International Journal of Management & Information Systems (IJMIS)*, 15(4):47–58.

StatCounter (2022). Search engine market share worldwide.

Ting, K. M. (2010). *Precision and Recall*, pages 781–781. Springer US, Boston, MA.

Zaky, A. dan Munir, R. (2016). Full-text search on data with access control using generalized suffix tree. In *2016 International Conference on Data and Software Engineering (ICoDSE)*, pages 1–6. IEEE.



LAMPIRAN A

Kuesioner Relevansi Hasil Pencarian Menggunakan Metode *Mean Average Precision*

A. Identitas *Tester*

Nama :

Pekerjaan :

B. Kuesioner Penilaian

Tabel 1.1: Contoh Kuesioner Penilaian Relevansi Hasil Menggunakan Metode *Mean Average Precision*

No.	Hasil Pencarian <i>Query</i> "bjorka"	Relevan
1	URL Hasil Pencarian 1	
2	URL Hasil Pencarian 2	
3	URL Hasil Pencarian 3	
4	URL Hasil Pencarian 4	
5	URL Hasil Pencarian 5	
No.	Hasil Pencarian <i>Query</i> "messi"	Relevan
1	URL Hasil Pencarian 1	
2	URL Hasil Pencarian 2	
3	URL Hasil Pencarian 3	
4	URL Hasil Pencarian 4	
5	URL Hasil Pencarian 5	
No.	Hasil Pencarian <i>Query</i> "masak"	Relevan
1	URL Hasil Pencarian 1	
2	URL Hasil Pencarian 2	
3	URL Hasil Pencarian 3	
4	URL Hasil Pencarian 4	
5	URL Hasil Pencarian 5	

No.	Hasil Pencarian Query "resesi 2023"	Relevan
1	URL Hasil Pencarian 1	
2	URL Hasil Pencarian 2	
3	URL Hasil Pencarian 3	
4	URL Hasil Pencarian 4	
5	URL Hasil Pencarian 5	
No.	Hasil Pencarian Query "bitcoin naik"	Relevan
1	URL Hasil Pencarian 1	
2	URL Hasil Pencarian 2	
3	URL Hasil Pencarian 3	
4	URL Hasil Pencarian 4	
5	URL Hasil Pencarian 5	
No.	Hasil Pencarian Query "world cup"	Relevan
1	URL Hasil Pencarian 1	
2	URL Hasil Pencarian 2	
3	URL Hasil Pencarian 3	
4	URL Hasil Pencarian 4	
5	URL Hasil Pencarian 5	
No.	Hasil Pencarian Query "teknolpgi"	Relevan
1	URL Hasil Pencarian 1	
2	URL Hasil Pencarian 2	
3	URL Hasil Pencarian 3	
4	URL Hasil Pencarian 4	
5	URL Hasil Pencarian 5	
No.	Hasil Pencarian Query "reseso"	Relevan
1	URL Hasil Pencarian 1	

Tanda tangan:

LAMPIRAN B

Jawaban Kuesioener Relevansi Hasil Pencarian Menggunakan Metode *Mean Average Precision*

A. Identitas *Tester*

Nama : Reza Nurdiansyah Firdaus
Pekerjaan : Mahasiswa

B. Kuesioner Penilaian

Tabel 2.1: Tabel Penilaian Relevansi Hasil Menggunakan Metode *Mean Average Precision* Reza Nurdiansyah Firdaus

No.	Hasil Pencarian <i>Query</i> "bjorka"	Relevan
1	URL Hasil Pencarian 1	v
2	URL Hasil Pencarian 2	v
3	URL Hasil Pencarian 3	v
4	URL Hasil Pencarian 4	v
5	URL Hasil Pencarian 5	v
No.	Hasil Pencarian <i>Query</i> "messi"	Relevan
1	URL Hasil Pencarian 1	v
2	URL Hasil Pencarian 2	v
3	URL Hasil Pencarian 3	v
4	URL Hasil Pencarian 4	v
5	URL Hasil Pencarian 5	v
No.	Hasil Pencarian <i>Query</i> "masak"	Relevan
1	URL Hasil Pencarian 1	
2	URL Hasil Pencarian 2	
3	URL Hasil Pencarian 3	
4	URL Hasil Pencarian 4	v
5	URL Hasil Pencarian 5	v

No.	Hasil Pencarian Query "resesi 2023"	Relevan
1	URL Hasil Pencarian 1	
2	URL Hasil Pencarian 2	v
3	URL Hasil Pencarian 3	
4	URL Hasil Pencarian 4	
5	URL Hasil Pencarian 5	
No.	Hasil Pencarian Query "bitcoin naik"	Relevan
1	URL Hasil Pencarian 1	v
2	URL Hasil Pencarian 2	
3	URL Hasil Pencarian 3	
4	URL Hasil Pencarian 4	
5	URL Hasil Pencarian 5	
No.	Hasil Pencarian Query "world cup"	Relevan
1	URL Hasil Pencarian 1	
2	URL Hasil Pencarian 2	
3	URL Hasil Pencarian 3	
4	URL Hasil Pencarian 4	
5	URL Hasil Pencarian 5	
No.	Hasil Pencarian Query "teknolpgi"	Relevan
1	URL Hasil Pencarian 1	v
2	URL Hasil Pencarian 2	v
3	URL Hasil Pencarian 3	
4	URL Hasil Pencarian 4	v
5	URL Hasil Pencarian 5	v
No.	Hasil Pencarian Query "reseso"	Relevan
1	URL Hasil Pencarian 1	

Tanda tangan:



LAMPIRAN C

Jawaban Kuesioener Relevansi Hasil Pencarian Menggunakan Metode *Mean Average Precision*

A. Identitas *Tester*

Nama : Rachel H
Pekerjaan : Mahasiswa

B. Kuesioner Penilaian

Tabel 3.1: Tabel Penilaian Relevansi Hasil Menggunakan Metode *Mean Average Precision* Rachel H

No.	Hasil Pencarian <i>Query</i> "bjorka"	Relevan
1	URL Hasil Pencarian 1	v
2	URL Hasil Pencarian 2	v
3	URL Hasil Pencarian 3	v
4	URL Hasil Pencarian 4	v
5	URL Hasil Pencarian 5	v
No.	Hasil Pencarian <i>Query</i> "messi"	Relevan
1	URL Hasil Pencarian 1	v
2	URL Hasil Pencarian 2	v
3	URL Hasil Pencarian 3	v
4	URL Hasil Pencarian 4	v
5	URL Hasil Pencarian 5	v
No.	Hasil Pencarian <i>Query</i> "masak"	Relevan
1	URL Hasil Pencarian 1	v
2	URL Hasil Pencarian 2	v
3	URL Hasil Pencarian 3	v
4	URL Hasil Pencarian 4	v
5	URL Hasil Pencarian 5	v

No.	Hasil Pencarian Query "resesi 2023"	Relevan
1	URL Hasil Pencarian 1	v
2	URL Hasil Pencarian 2	
3	URL Hasil Pencarian 3	
4	URL Hasil Pencarian 4	
5	URL Hasil Pencarian 5	
No.	Hasil Pencarian Query "bitcoin naik"	Relevan
1	URL Hasil Pencarian 1	v
2	URL Hasil Pencarian 2	v
3	URL Hasil Pencarian 3	v
4	URL Hasil Pencarian 4	v
5	URL Hasil Pencarian 5	v
No.	Hasil Pencarian Query "world cup"	Relevan
1	URL Hasil Pencarian 1	v
2	URL Hasil Pencarian 2	v
3	URL Hasil Pencarian 3	v
4	URL Hasil Pencarian 4	v
5	URL Hasil Pencarian 5	v
No.	Hasil Pencarian Query "teknolpgi"	Relevan
1	URL Hasil Pencarian 1	v
2	URL Hasil Pencarian 2	v
3	URL Hasil Pencarian 3	v
4	URL Hasil Pencarian 4	v
5	URL Hasil Pencarian 5	v
No.	Hasil Pencarian Query "reseso"	Relevan
1	URL Hasil Pencarian 1	

Tanda tangan:



LAMPIRAN D

Jawaban Kuesioener Relevansi Hasil Pencarian Menggunakan Metode *Mean Average Precision*

A. Identitas Tester

Nama : Lazuardy Khatulistiwa
Pekerjaan : Mahasiswa

B. Kuesioner Penilaian

Tabel 4.1: Tabel Penilaian Relevansi Hasil Menggunakan Metode *Mean Average Precision* Lazuardy Khatulistiwa

No.	Hasil Pencarian Query "bjorka"	Relevan
1	URL Hasil Pencarian 1	v
2	URL Hasil Pencarian 2	v
3	URL Hasil Pencarian 3	v
4	URL Hasil Pencarian 4	v
5	URL Hasil Pencarian 5	v
No.	Hasil Pencarian Query "messi"	Relevan
1	URL Hasil Pencarian 1	v
2	URL Hasil Pencarian 2	v
3	URL Hasil Pencarian 3	v
4	URL Hasil Pencarian 4	v
5	URL Hasil Pencarian 5	v
No.	Hasil Pencarian Query "masak"	Relevan
1	URL Hasil Pencarian 1	v
2	URL Hasil Pencarian 2	v
3	URL Hasil Pencarian 3	v
4	URL Hasil Pencarian 4	v
5	URL Hasil Pencarian 5	v

No.	Hasil Pencarian Query "resesi 2023"	Relevan
1	URL Hasil Pencarian 1	v
2	URL Hasil Pencarian 2	v
3	URL Hasil Pencarian 3	
4	URL Hasil Pencarian 4	
5	URL Hasil Pencarian 5	
No.	Hasil Pencarian Query "bitcoin naik"	Relevan
1	URL Hasil Pencarian 1	v
2	URL Hasil Pencarian 2	v
3	URL Hasil Pencarian 3	v
4	URL Hasil Pencarian 4	v
5	URL Hasil Pencarian 5	v
No.	Hasil Pencarian Query "world cup"	Relevan
1	URL Hasil Pencarian 1	v
2	URL Hasil Pencarian 2	v
3	URL Hasil Pencarian 3	v
4	URL Hasil Pencarian 4	v
5	URL Hasil Pencarian 5	v
No.	Hasil Pencarian Query "teknolpgi"	Relevan
1	URL Hasil Pencarian 1	v
2	URL Hasil Pencarian 2	v
3	URL Hasil Pencarian 3	v
4	URL Hasil Pencarian 4	v
5	URL Hasil Pencarian 5	v
No.	Hasil Pencarian Query "reseso"	Relevan
1	URL Hasil Pencarian 1	

Tanda tangan:

Lazuardy

DAFTAR RIWAYAT HIDUP



Zaidan Pratama, lahir di Jakarta pada tanggal 16 September 2000. Penulis merupakan anak pertama dari tiga bersaudara yaitu dari pasangan Dedy Ariansyah dan Nariah. Penulis saat ini tinggal di Jl. Lancar Raya IV no 26 rt 014 rw 007, Sumur Batu, Kemayoran, Jakarta Pusat, 10640.

No. Ponsel: 081380111974

E-mail: zaidanpratamaa@gmail.com

Riwayat Pendidikan: Penulis mengenyam pendidikan dan lulus dari sekolah dasar di SDN Sumur Batu 12 Pagi Jakarta pada tahun 2006 – 2012, kemudian melanjutkan pendidikan ke SMPN 10 Jakarta pada tahun 2012 – 2015. Setelah itu, penulis melanjutkan pendidikan di SMAN 68 Jakarta pada tahun 2015 – 2018. Di tahun yang sama (2018), penulis diterima kuliah di Universitas Negeri Jakarta (UNJ) Fakultas Matematika dan Ilmu Pengetahuan Alam (FMIPA) dengan program studi Ilmu Komputer dengan jalur masuk melalui SBMPTN.