Penerapan *Hashing* dalam Algoritma Rabin-Karp untuk Menentukan Kata Kunci dalam Suatu Teks

Haidar Ihzaulhaq - 13520150[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13520150@std.stei.itb.ac.id

Abstract*—In a text, there are many components that build the structure of the text, one of the components is keyword. Keyword is a word or phrases that unique and appear frequently in the text. Based on that, we can determine some keywords from the text if it's same with the description, that is, it often frequently appears and unique. The way to get the frequency of occurrence of a word or a phrase in the text can be various, one of the ways is by using Rabin-Karp algorithm. Rabin-Karp algorithm is an algorithm that use hashing process to determine whether a pattern or string is exist in the text or not. With this algorithm, if we collect unique word's data from the text and check their frequency of occurrence in the text using Rabin-Karp algorithm, we will get a list of words or phrase that have the most frequency and can be used to determine the keyword in the text. Before we determine the keywords, we should 'clear' the data list by remove the words of phrase that don't have unique meaning. After that, the top list in the data list can be determine as a keyword because it has fulfilled the requirement to be a keyword (unique and frequently appears).

Keywords— hashing, keyword, Rabin-Karp algorithm, text.

I. PENDAHULUAN

Dalam suatu teks, terdapat banyak komponen yang menyusunnya, salah satunya adalah kata kunci. Kata kunci atau biasa disebut *keyword* merupakan kata-kata atau frasa yang penting dan sering muncul dalam suatu teks. Dalam suatu teks, kata kunci menjadi unsur yang penting karena kata kunci akan menjadi kata atau frasa yang mewakili isi teks secara garis besar. Kata kunci dari suatu teks tidak terbatas hanya satu kata saja, bisa jadi dalam suatu teks terdapat lebih dari satu kata kunci, bahkan hal tersebut merupakan hal yang umum.

Kata kunci teks memiliki ciri-ciri yang membedakannya dengan kata lain di dalam suatu teks. Ciri-ciri tersebut yang pertama adalah tersusun dari satu hingga dua kata yang membentuk suatu frasa. Ciri-ciri yang kedua adalah kata kunci mengandung kata yang sifatnya unik. Sifat unik disini maksudnya adalah bukan merupakan kata yang umum dipakai, seperti kata hubung, kata depan, kata kerja, dan lain sebagainya serta merupakan kata yang mudah untuk diingat oleh orang lain. Ciri-ciri yang terakhir adalah kata kunci akan sering muncul dalam setiap pembahasan di seluruh bagian teks.

Dari ciri-ciri yang telah disebutkan, secara sederhana kata kunci dapat dikatakan sebagai kumpulan beberapa kata atau frasa yang unik dan sering muncul dalam suatu teks. Dari pemahaman ini, kita dapat menentukan kata kunci yang ada di suatu teks dengan mencarinya sesuai deskripsi. Namun, mencari secara manual bukanlah hal yang mudah untuk dilakukan karena perlu melakukan pengecekan semua kata yang ada pada teks dan menghitung jumlah tiap kata dalam teks itu sendiri. Oleh karena itu, dalam makalah ini penulis akan membahas cara menentukan kata kunci dalam suatu teks dengan memanfaatkan algoritma Rabin-Karp yang mengandung fungsi *hashing* di dalamnya. Dengan algoritma ini, pengecekan kata kunci teks akan menjadi lebih mudah dilakukan dan lebih akurat dibandingkan dengan cara manual yang mungkin memiliki *miss rate* lumayan tinggi akibat *human error*.

II. LANDASAN TEORI

A. *Hashing*

Hashing adalah suatu cara untuk mentransformasi sebuah string menjadi suatu nilai yang unik dengan panjang tertentu dan panjangnya pasti yang berfungsi sebagai penanda string tersebut. Nilai unik yang dimaksud adalah suatu string akan diubah menjadi suatu nilai yang tidak mungkin sama nilainya dengan string lainnya. Dalam *hashing*, fungsi yang digunakan untuk melakukan transformasi disebut fungsi hash dan hasil yang didapat dapat disebut sebagai *hash value* (nilai hash).

Fungsi hash yang umum digunakan adalah fungsi hash satu-arah, yaitu fungsi hash yang hanya dapat mengubah string atau suatu pesan ke suatu nilai unik tertentu dan tidak dapat mengembalikan nilai uniknya ke bentuk string atau pesan semula. Fungsi hash satu arah memiliki beberapa sifat, yaitu:

1. Fungsi H dapat diterapkan pada blok data atau string atau pesan berukuran berapa saja (tidak dibatasi).
2. Fungsi H menghasilkan nilai (h) dengan panjang tetap (*fixed-length output*).
3. H(*x*) mudah dihitung *hash value*-nya untuk setiap nilai *x* yang diberikan.
4. Untuk setiap h yang dihasilkan, tidak mungkin dikembalikan ke nilai x awal sedemikian sehingga H(x) = h. Itulah sebabnya fungsi H dikatakan fungsi hash satu-arah (*one way hash function*).
5. Untuk setiap x yang diberikan, tidak mungkin mencari y ≠ x sedemikian sehingga H(y) = H(x) karena tiap pesan harus memiliki *hash value* yang berbeda.
6. Tidak mungkin mencari pasangan x dan y sedemikian sehingga H(x) = H(y).

Biasanya, fungsi hash akan memanfaatkan perhitungan modulus atau sisa pembagian dalam prosesnya. Oleh sebab itu, suatu pesan yang telah diubah dalam bentuk *hash value* tidak bisa atau kecil kemungkinan untuk dikembalikan ke pesan awal. Sebagai contoh, karakter 'A' atau dalam ASCII memiliki nilai 65 jika dilakukan *hashing* dengan melakukan operasi modulus 13, maka akan didapat *hash value* yaitu 0. Ketika *hash value* ini ingin dikembalikan ke pesan semula, hal ini akan sulit untuk dilakukan karena akan banyak kemungkinan yang tidak dapat kita prediksi mana yang merupakan pesan awal, mulai dari 0, 13, 26, 39, 52, dst.

Salah satu pemanfaatan *hashing* dalam dunia pemrograman adalah dalam pembuatan algoritma Rabin-Karp yang menggunakan dasar bahwa dua buah string yang sama akan memiliki *hash value* yang sama juga. Dengan memanfaatkan hal ini, algoritma Rabin-Karp dapat dimodifikasi dan digunakan untuk beberapa hal, seperti mencari suatu string pada teks, mendeteksi plagiarisme antara dua teks berbeda, atau seperti yang akan dibahas oleh pemakalah, yaitu menentukan kata kunci pada suatu teks.

B. Algoritma Rabin-Karp

Algoritma Rabin-Karp adalah sebuah algoritma yang diciptakan oleh Michael O. Robin dan Richard M. Karp pda tahun 1987 dengan menggunakan fungi *hashing* untuk menemukan sebuah *pattern*/string di dalam teks. Algoritma Rabin-Karp memiliki beberapa karakteristik, yaitu menggunakan sebuah fungsi *hashing*, memiliki kompleksitas waktu O(n) untuk pemrosesan stringnya dan O(mn) untuk pencarian stringnya dengan n adalah string/*pattern* yang dicari dan m adalah substring pada teks.

Konsep dasar yang digunakan pada algoritma Rabin-Karp ini adalah akan dilakukan pencocokan antara *pattern*/string yang dicari dengan substring pada teks. Apabila *pattern* dan substring memiliki *hash value* yang sama, maka akan dilakukan pencocokan terhadap karakter-karakternya. Pengecekan karakter dilakukan karena dalam penggunaan *hashing*, ada istilah kolisi (*collision*). Kolisi adalah suatu kondisi ketika dua string yang berbeda memiliki *hash value* yang sama. Hal ini mungkin saja terjadi jika fungsi hash yang dilakukan adalah fungsi yang sederhana sehingga akan memungkinkan adanya kolisi pada dua string yang berbeda. Karena alasan inilah, pengecekan kesamaan karakter akan dilakukan apabila *hash value* dari *pattern* dan substring bernilai sama untuk memastikan apakah *pattern* dan substring adalah dua string yang sama. Secara umum, langkah-langkah algoritma Rabin-Karp adalah sebagai berikut:

1. Asumsikan input dari fungsi ini adalah sebuah *pattern* P (string yang akan dicari) dengan panjang n dan sebuah teks T (dalam bentuk string) dengan panjang m.
2. Asumsikan Si adalah sebuah substring dengan panjang n, yaitu panjang yang sama dengan *pattern* yang akan dicari. Si adalah sebuah substring yang didapat secara berkelanjutan dari teks input, seperti misal, jika Si0 adalah substring dengan panjang n dan dimulai dari indeks ke-k pada teks, maka Si1 adalah substring dengan panjang n dan dimulai dari indeks ke-(k+1), begitu seterusnya hingga Si terakhir dimulai dari indeks ke-(m-n).
3. String *pattern* dibuat terlebih dahulu fungsi hashnya dengan menggunakan fungsi hash yang paling umum (memanfaatkan perhitungan modulus) berbentuk:
   $$h(x) = x \bmod p$$
   Dalam algoritma ini, x merupakan nilai ASCII dari tiap karakter yang ada pada string yang kemudian akan dimasukkan ke fungsi hash agar mendapatkan *hash value* dalam bentuk integer.
4. Dilakukan penelusuran semua kemungkinan substring dari teks yang panjangnya m dengan panjang substring adalah n dari indeks 1 sampai indeks m-n. Di setiap substring yang didapat, lakukan *hashing* pada substring kemudian bandingkan.
5. Jika h(P) = h(Si), maka lakukan pencocokan karakter antara P dengan Si. Jika cocok maka artinya Si adalah substring yang dicari, Jika h(P) ≠ h(Si), maka pengecekan karakter tidak perlu dilakukan karena sudah jelas merupakan dua string yang berbeda.
6. Jika ingin melakukan pencarian semua kemungkinan yang ada pada teks, maka iterasi substring dapat terus dilakukan setelah ditemukan substring yang mirip dengan *pattern* sambil mencatat indeks ketika P = Si.

Langkah-langkah tersebut bila diterapkan dalam pseudocode akan menjadi seperti berikut:

Algoritma Rabin-Karp

```
function rabin_karp(string T[1..m],
string P[1..n])
  hsub := hash(s[1..n])
  hpattern := hash(P[1..n])
  for i from 1 to m-n
      if hpattern = hsub
            if s[i..i+n-1] = P
                  return i
      hs := hash(s[i+1..i+n])
  return not found
```

Seperti yang telah disebutkan sebelumnya, algoritma Rabin-Karp ini didasarkan pada perbandingan antara string *pattern* dengan substring yang didapat secara iterasi sebanyak m-n kali. Itu artinya, untuk setiap pemeriksaan satu *pattern* dalam teks, akan memerlukan kompleksitas waktu sebesar O(mn) jika dilakukan secara brute force. Perhitungan O(mn) ini didapat dari iterasi sebanyak m kali untuk mencari substring, dan tiap substring dilakukan iterasi sebesar n kali untuk mengecek setiap karakter ketika membandingkan substring dengan *pattern*. Inilah kunci dari algoritma Rabin-Karp. Untuk menghindari kompleksitas yang besar, maka terdapat cara yang efektif untuk menentukan *hash value* yang digunakan. Salah satu cara tersebut adalah dengan menjadikan tiap substring sebagai suatu bilangan unik dengan basis tertentu. Biasanya basis yang

digunakan adalah bilangan prima yang berukuran besar. Dengan cara ini, kita dapat menentukan substring selanjutnya dengan hanya menghilangkan *hash value* untuk karakter pertama dan menggantinya dengan *hash value* untuk karakter selanjutnya. Sebagai contoh, kita memiliki sebuah string "23456" dan kita ingin mencari *pattern* dengan panjang 3. Dengan begitu, pertama akan ditentukan *hash value* untuk Si0, yaitu tiga karakter pertama dari string ("234"). Dengan menggunakan basis 10, kita dapat menentukan *hash value* untuk Si0 yaitu 234. Sederhananya, untuk menentukan substring selanjutnya, yaitu substring "345", kita dapat melakukan perhitungan dengan cara (234 – 2 * 100) * 10 + 5 sehingga didapatlah nilai 345 yang merupakan *hash value* dari "345" dalam basis 10. Secara umum, *hash value* dari tiap substring dapat ditentukan dengan cara berikut:

1. Menentukan *hash value* untuk substring pertama dari text dengan menggunakan algoritma berikut:

```
N := length(pattern)
t := 0
h := 1
d := 256 (banyak karakter di string)
for i from 1 to N-1
      h := (h*d) mod q
for i from 1 to N
      t := (d*t + txt[i]) mod q
*note : txt[i] dalam bentuk ASCII
```

2. Menentukan *hash value* untuk substring berikutnya dengan menggunakan algoritma berikut:

```
t := (d*(t-txt[i]*h)+(txt[i+N]) mod q
if (t < 0) :
      t = t+q
```

Dengan cara ini, maka operasi perbandingan karakter antara substring dengan *pattern* dalam algoritma brute force dapat dihilangkan dan diganti dengan pengecekan kesamaan *hash value* dari *pattern* dan substring. Apabila *hash value* keduanya sama, barulah dilakukan pengecekan tiap karakter untuk memastikan kesamaan *pattern* dengan substring. Langkah ini, secara perhitungan membuat algoritma berjalan lebih mangkus karena dia hanya memiliki kompleksitas O(m+n) karena tiap iterasi substring baru, dia hanya akan melakukan satu operasi, yaitu pengubahan nilai *hash value* dari substring sebelumnya dan operasi pembanding karakter hanya akan dilakukan sekali ketika ditemukan *hash value* yang sama.

C. *Priority Queue*

Dalam pencarian kata kunci pada suatu teks, kita perlu sebuah struktur data untuk membantu menyimpan setiap kemungkinan kata yang akan dicari dan dihitung frekuensinya dalam teks. Oleh karena itu, salah satu struktur data yang cocok digunakan untuk permasalahan ini adalah priority queue. Priority queue adalah salah satu jenis queue yang special karena setiap elemen pada queue tersebut terikat/terhubung oleh sebuah *priority value* dan elemen-elemen akan diurutkan sesuai dengan *priority value* mereka. *Priority value* yang dimaksud dapat berupa sebuah nilai yang mungkin membedakan antara elemen satu dengan yang lain sehingga dapat dilakukan pengurutan prioritas antarelemen.

Alasan *priority queue* cocok untuk proses pencarian kata kunci adalah adanya *priority value* yang dimiliki tiap kata kunci (frekuensinya) yang dapat diurutkan nantinya dari yang terbesar. Dengan *priority queue* ini, sudah pasti kata kunci yang dicari akan berada pada daftar atas prioritas karena memiliki frekuensi yang besar juga. Dalam *priority queue* ini pada dasarnya terdapat 3 proses utama, yaitu mengakses elemen pertama, enqueue, dequeue. Enqueue adalah proses memasukkan elemen ke dalam *priority queue*. Sedangkan dequeue adalah proses menghapus elemen pertama/elemen prioritas pada priority queue. Dengan menggunakan *priority queue* ini, kita dapat melakukan enqueue kata kunci baru yang kita dapat, dan pada akhir proses kita dapat melakukan dequeue untuk mendapatkan kata kunci teratas yang telah didapat.

III. PENCARIAN KATA KUNCI DALAM TEKS

A. *Bagian Penyusun Program*

Pencarian kata kunci pada teks dapat dilakukan dengan membuat sebuah program yang menerima input sebuah dokumen (memiliki format .doc atau .docx) kemudian dilakukan pengolahan untuk mencapai tujuan yang diinginkan. Dalam program pengolahan ini, akan diperlukan beberapa bagian yang Menyusun program sehingga program dapat berjalan sesuai keinginan. Bagian-bagian penyusun program tersebut yaitu:

a. Main Program

Bagian ini merupakan bagian utama program, yaitu tempat menggabungkan semua fungsi, struktur data, dan procedure yang telah dibuat. Selain itu, main program ini akan menjadi tempat menjalankan program dan menghasilkan output yang diinginkan. Secara umum, nantinya bagian ini akan berisi variable-variabel global yang akan digunakan, pemanggilan fungsi dari bagian lain, serta pemanggilan struktur data dari bagian lain. Bagian main program ini dapat dituliskan dalam pseudocode sebagai berikut:

```
                Main Program
import function search, function
read_file, function getkata
import priority_queue


myQueue.create()
text := read_file()
prev_kata := ""
idx := 1
```

```
while (not end of kata) do
      kata := getkata()
      freq := search(kata)
      myQueue.insert(<kata,freq>)
      frasa := prev_kata + " " + kata
      freq := search(frasa)
      myQueue.insert(<frasa,freq>)
      prev_kata := kata
while (not myQueue.empty) do
      print(myQueue.delete())
```

b. Search

Bagian search merupakan bagian yang berfungsi untuk mencari kata yang akan dihitung jumlahnya. Di bagian ini, akan digunakan algoritma Rabin-Karp yang telah dibahas sebelumnya dengan sedikit modifikasi pada algoritmanya. Modifikasi ini dilakukan dengan tujuan agar algoritma dapat menghitung banyak kata yang sama dengan *pattern* pada teks. Selain itu, algoritma juga diubah agar tetap dapat berjalan walaupun telah menemukan substring yang sesuai dengan *pattern* yang dicari. Dengan begini, frekuensi kemunculan *pattern* dalam teks dapat dihitung dan kemudian dijadikan sebagai nilai kembalian/*return value* dari fungsi ini. Bagian ini dapat dituliskan dalam pseudocode sebagai berikut:

Search

```
function search(string T[1..m], string
P[1..n])
  cnt := 0
  hsub := hash(s[1..n])
  hpattern := hash(P[1..n])
  for i from 1 to m-n
      if hpattern = hsub
            if s[i..i+n-1] = P
                  cnt += 1
      hs := hash(s[i+1..i+n])
  return cnt
```

c. Read File

Bagian read file adalah bagian yang membaca input, yaitu berupa teks document yang akan dibaca dan diolah untuk dicari kata kuncinya. Pada bagian ini, sebenarnya syntax yang digunakan akan memiliki perbedaan pada setiap bahasa pemrograman. Hal ini tidak menjadi masalah, yang terpenting adalah kita perlu mencari pada dokumentasi *library* bahasa pemrograman yang digunakan tentang cara membaca sebuah file berformat .doc atau .docx. Document yang telah dibaca nantinya akan diubah menjadi bentuk string. Setelah itu, di bagian ini juga dilakukan penanganan perbedaan penulisan upper case dan lower case karena dalam perhitungan hash pada algoritma Rabin-Karp, penulisan secara upper case dan lower case akan berbeda hasilnya. Hal ini disebabkan adanya perbedaan nilai ASCII antara kedua karakter. Padahal, dalam pencarian kata kunci, penulisan secara upper case ataupun lower case akan memiliki makna yang sama. Sebagai contoh, kata "sandi" dan "Sandi" akan dianggap dua kata yang berbeda ketika dilakukan *hashing*. Padahal, secara makna, kata tersebut adalah sama dan hanya dibedakan oleh gaya penulisan saja. Oleh karena itu, di bagian ini, string dari teks yang telah diperoleh akan diubah semua ke salah satu bentuk, dalam hal ini penulis memilih untuk mengubahnya ke bentuk lower case. Setelah berhasil diubah ke bentuk lower case, maka string teks dapat dijadikan sebagai nilai kembalian/*return value* dari fungsi ini. Bagian ini jika dituliskan dalam bentuk pseudocode akan sebagai berikut:

Read File

```
import document.reader


function read_file()
 word := open.document(filename)
 str_kata := convert_to_string(word)
 str_kata := str_kata.lower()
 return str_kata
```

d. Word Process

Bagian ini merupakan bagian untuk mengolah kata-kata pada string teks yang telah didapat. Bagian ini digunakan untuk mendapatkan string kata per kata yang ada pada string teks. Hal ini penting karena tujuan dari program keseluruhan adalah mencari kata kunci, artinya kita perlu mendata semua kata atau frasa yang ada pada teks. Dengan fungsi ini, program utama akan menerima kata per kata yang ada pada string untuk kemudian dijadikan *pattern* dalam fungsi pencarian dengan algoritma Rabin-Karp. Kata yang dihasilkan dari fungsi ini merupakan kata yang "bersih", yang artinya, kata-kata yang dihasilkan tidak akan tercampur oleh simbol-simbol seperti ',', '/', ':', dan lain sebagainya yang dapat mengganggu perhitungan nilai hash nantinya. Sebagai contoh, jika terdapat kalimat "sebagai berikut:" dengan symbol pada string teks, maka pada fungsi ini akan mengembalikan string dalam bentuk kata "sebagai" dan "berikut" saja dengan menghilangkan symbol yang ada. Hal ini bertujuan untuk meminimalisasi kesalahan perhitungan frekuensi kata nantinya pada fungsi search sehingga perhitungan frekuensi dapat lebih akurat. Bagian ini jika dituliskan dalam bentuk pseudocode sebagai berikut:

Word Processor

```
global idx := 0
function getkata(text)
 simbol := [':', ',', '!', '(', ')',
            '', '"']
 m := length(text)
```

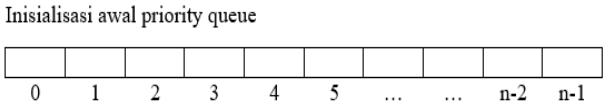
```
kata := ""
while (idx < m)and((text[idx] != " ")
or (text[idx] != "/")) do
      if (text[idx] not is simbol)
            kata += text[idx]
      idx += 1
idx += 1
return kata
```

e. Priority Queue

Bagian ini merupakan bagian opsional. Hal ini disebabkan priority queue merupakan salah satu struktur data yang tidak selalu tersedia di setiap library bahasa pemrograman. Sebagai contoh, dalam bahasa pemrograman C++, priority queue telah tersedia dalam *library*-nya dan tinggal dipanggil saja kemudian digunakan. Namun sayangnya, tidak semua bahasa pemrograman memiliki priority queue dalam *library*-nya. Hal ini tentu membuat kita harus membuat *prority queue* kita sendiri dengan memanfaatkan struktur data yang tersedia, seperti list. Pada dasarnya, dalam program ini, kita akan menggunakan *priority queue* untuk menyimpan elemen data dalam bentuk tuple <*string* kata, *int* frekuensi> dengan frekuensi akan menjadi *prority value* untuk dibandingkan. Dalam struktur data ini, fungsi yang terpenting untuk dibuat terdiri dari 3 bagian, yaitu:

1. Create Priority Queue

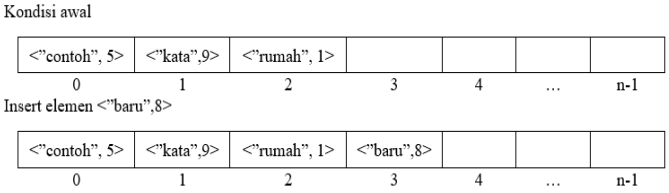
Fungsi ini digunakan untuk membuat priority queue baru dan kosong. Banyak alternatif yang dapat digunakan untuk membuat priority queue ini, bisa menggunakan list, linked list, binary heap, atau yang lain. Namun, pada program ini penulis akan menggunakan alternatif yang mudah untuk digunakan yaitu dengan menggunakan list. Dalam fungsi ini, kita hanya tinggal mendeklarasikan sebuah list kosong dengan tipe elemen berupa tuple <*string* kata, *int* frekuensi>.



Gambar 3.1 Ilustrasi Create Priority Queue

2. Insert Element

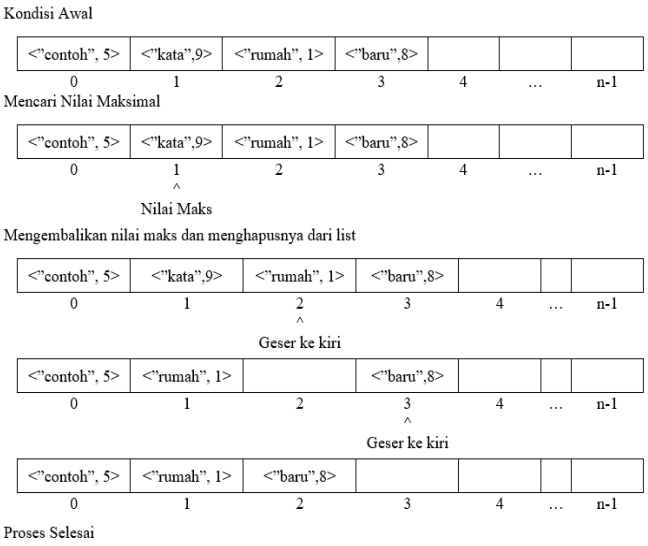
Fungsi ini digunakan untuk menambahkan elemen baru pada list. Hal ini dapat dilakukan dengan melakukan assign elemen baru ke dalam urutan terakhir list. Kita tidak perlu memerdulikan urutan elemen baru pada list yang dimiliki walaupun seperti yang diketahui, priority queue mementingkan urutan elemen berdasarkan *priority value*-nya. Pengolahan elemen yang menggunakan konsep priority queue akan dilakukan dalam proses delete elemen.



Gambar 3.2 Ilustrasi Insert Element

3. Delete Element

Fungsi ini merupakan fungsi yang paling penting yang membuat struktur data yang dibuat sesuai dengan deskripsi priority queue. Fungsi delete element adalah fungsi yang mengembalikan elemen teratas pada priority queue dan menghapus elemen tersebut dari list. Dalam struktur data buatan ini, fungsi delete element tidak akan berjalan secara mudah. Pertama, akan dilakukan pencarian *priority value* yang paling besar dengan iterasi di setiap elemen pada list. Seteleh didapat dan dicatat urutan elemen tersebut, maka elemen akan disimpan pada suatu variable dan kemudian elemen pada list akan dihapus dengan cara ditimpa dengan elemen selanjutnya pada list. Elemen selanjutnya akan digeser terus hingga sampai pada akhir elemen list. Setelah itu, kita dapat menjadikan variabel penyimpan elemen teratas tadi sebagai *return value*.



Gambar 3.3 Ilustrasi Delete Element

B. Analisis Program Utama

Dari bagian-bagian yang ada, program utama dapat dijalankan dan akan menghasilkan tujuan yang diinginkan. Namun, sebagaimana program pada umumnya, kita perlu melakukan analisis terhadap program yang telah dibuat. Salah satu analisis penting yang perlu diperhatikan adalah analisis kompleksitas waktu. Analisis kompleksitas waktu dapat dilakukan juga dengan melakukan analisis tiap bagiannya. Berikut adalah analisis kompleksitas waktu tiap bagian:

a. Search

Bagian ini merupakan bagian yang menerapkan algoritma Rabin-Karp. Secara perhitungan, fungsi search

akan melakukan iterasi sebanyak m-n, dan setiap ditemukan *hash value* yang sama akan dilakukan pengecekan string. Maka kompleksitas waktu fungsi search kurang lebih akan tetap sama dengan kompleksitas algoritma Rabin-Karp itu sendiri, yaitu sebesar O(m+n).

b. Read File

Pada bagian ini, sebenarnya agak sulit untuk ditentukan berapa kompleksitas yang diperlukan karena di bagian ini menggunakan standard library yang ada di bahasa pemrograman. Paling tidak, dari proses read file, yang dapat diidentifikasi adalah pengubahan string menjadi lower case yang memerlukan waktu O(m).

c. Word Process

Bagian word processing hanya memerlukan kompleksitas waktu sebesar O(m) karena hanya perlu melakukan iterasi string teks dari awal hingga akhir.

d. Priority Queue

Pada bagian ini, create priority queue akan memerlukan waktu sebesar O(1) karena hanya tinggal mengalokasikan list ke dalam memori. Untuk insert element juga hanya memerlukan waktu sebesar O(1) sedangkan untuk delete element memerlukan watu sebesar O(t) dengan t adalah panjang array (sebenarnya memerlukan waktu 2t, namun dalam perhitungan big O konstanta akan dihilangkan).

Dari kompleksitas waktu tiap bagian, kita dapat melakukan perhitungan kompleksitas untuk program utama. Program utama akan memanggil read_file() sekali, getkata() sekali (hal ini karena fungsi getkata mengolah variabel global sehingga tidak akan mengulang pembentukan kata dari awal), search() sebanyak t kata dalam string teks, kemudian memanggil delete element sebanyak t kata. Artinya kompleksitas waktu dari program yang dibuat sebesar $O(t^2)$ dengan t adalah banyak kata dan frasa dalam string teks. Artinya, program ini masih berjalan cukup lambat karena memiliki kompleksitas dalam bentuk pangkat.

Dalam program yang dibuat, dapat dilakukan efisiensi waktu dengan melakukan beberapa hal, seperti mengabaikan kata-kata yang telah dihitung sebelumnya dan mengabaikan kata-kata yang tidak unik (kata kerja, kata hubung, kata depan). Dengan begitu, waktu kompleksitas dapat berkurang walaupun tidak akan terlihat signifikan secara perhitungan.

IV. KASUS UJI DAN PERFORMA

Pada bagian ini, penulis melakukan uji coba terhadap teks yang telah dipersiapkan. Teks yang digunakan adalah sebuah tulisan yang ada di dapat di sini yang kemudian dipindahkan ke dalam bentuk word dan dijalankan pada program. Program yang digunakan juga dapat dilihat di repository ini. Berikut adalah hasil dari program yang telah dibuat ketika melakukan uji terhadap teks yang diberikan:

```
Panjang string teks:  4937
Banyak kata:  631
('data', 35)
('pribadi', 18)
('data pribadi', 18)
('hak', 14)
('hukum', 9)
('milik', 8)
('pengguna', 7)
('pers', 6)
('tujuan', 6)
('definisi', 6)
Program Executed in 1.228078699999969
```

Gambar 4.1 Hasil Uji Teks Pertama

Dari tangkapan layar tersebut, dapat dilihat hasil dari pengolahan teks, yaitu berupa panjang string teks yang diolah, banyak kata yang didapat (tidak ada duplikasi kata yang sama), 10 daftar teratas kata dengan frekuensi terbanyak, serta waktu eksekusi yang dibutuhkan oleh program. 10 daftar kata teratas yang ditampilkan artinya adalah rekomendasi kata yang dapat dijadikan sebagai kata kunci, dalam hal ini kita dapat memilih beberapa saja. Sebagai contoh, dalam kasus teks ini, dapat diambil kata kunci yaitu “data”, “data pribadi”, dan “hukum”. Kata kunci ini jika dicocokkan dengan teks yang diolah, maka akan relevan dan saling berhubungan. Dari hasil tangkapan layar juga disebutkan bahwa untuk mengolah 4937 karakter, diperlukan waktu sekitar 1.22 sekon, waktu yang terbilang lumayan cepat untuk sebuah teks dengan 631 kata non-duplikasi.

Hasil lain dari program adalah pengolahan dari teks yang berasal dari link di sini dengan hasil sebagai berikut:

```
Panjang string teks:  4311
Banyak kata:  473
('petani', 22)
('masa', 16)
('lahan', 14)
('salah', 14)
('masalah', 13)
('indonesia', 10)
('hal', 9)
('permasalahan', 8)
('harga', 8)
('maka', 7)
Program Executed in 0.8488710999990872
```

Gambar 4.2 Hasil Uji Teks Kedua

Dari kedua contoh kasus uji, dapat dilihat bahwa performa yang didapat terbilang cukup memuaskan dengan tidak memakan waktu yang terlalu lama. Namun, seperti yang telah diketahui, daftar kata yang dihasilkan, tidak semuanya akan menjadi kata kunci. Kita dapat memilahnya lagi kata mana yang cocok untuk dijadikan kata kunci dari daftar teratas yang telah diberikan.

Program ini juga masih memiliki beberapa kelemahan seperti masih adanya kata tidak unik yang ikut terhitung. Permasalahan ini sebenarnya dapat dilakukan dengan bantuan user dan pengembangan program. Jika program dibuat dengan lebih interaktif dan memiliki konsep *machine learning*, maka jika user merasa kata yang didapat ternyata adalah kata tidak unik,

program akan memasukkan kata tersebut ke daftar kata tidak unik yang sebelumnya telah dibuat sehingga jika dilakukan pemrosesan ulang kata tidak unik tersebut tidak akan muncul lagi.

V. KESIMPULAN

Penerapan *hashing* pada algoritma Rabin-Karp dapat dimanfaatkan untuk melakukan sesuatu yang berhubungan dengan kata pada teks, salah satunya adalah pencarian kata kunci. Dengan berbekal definisi kata kunci, yaitu kata yang sering muncul dalam teks dan unik, maka dapat dibuat sebuah program untuk mencarinya. Penggunaan algoritma Rabin-Karp juga membuat pencarian kata kunci menjadi lebih cepat jika dibandingkan dengan algoritma lain, seperti *brute force*. Program yang dihasilkan juga dapat memberikan performa yang baik walaupun masih memiliki kekurangan di dalamnya. Kekurangan tersebut jika dilakukan pengembangan lebih lanjut tentunya akan dapat diatasi dan membuat program semakin baik dan menghasilkan performa yang memuaskan.

VI. UCAPAN TERIMA KASIH

Penulis mengucapkan puji syukur kepada Tuhan Yang maha Esa atas diberinya kesempatan, kemudahan, dan kelancaran bagi penulis untuk mengerjakan makalah ini. Penulis mengucapkan terima kasih kepada pihak-pihak yang telah membantu penulis dalam menyelesaikan makalah ini. Tak lupa, penulis juga menyampaikan terima kasih kepada seluruh dosen pengampu mata kuliah Matematika Diskrit, utamanya Ibu Nur Ulfa Maulidevi sebagai dosen pengampu kelas K3 yang telah bersedia untuk menyampaikan ilmunya kepada penulis.

REFERENCES

[1] Munir, Rinaldi. 2007. *Fungsi Hash*. (diakses dari https://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi/2017-2018/Fungsi-Hash-(2018).pdf pada 10 Desember 2021)

[2] Putra, Aria Doddi, dkk. 2015. *Implementasi Algoritma Rabin-Karp untuk Membantu Pendeteksian Plagiat pada Karya Ilmiah*. (diakses dari http://66.96.237.53/perpus/peraturan/upload/Rabin%20Pararel.pdf pada 10 Desember 2021)

[3] Baedlowi, Najib, Deka Aditia Adam. 2005. *String Matching dengan Menggunakan Algoritma Rabin-Karp*. (diakses dari https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2005-2006/Makalah2006/MakalahStmik2006-15.pdf pada 10 Desember 2021)

[4] Skola, Kompas. 2021. *Pengertian dan Ciri-Ciri Kata Kunci*. (diakses dari https://www.kompas.com/skola/read/2021/09/16/144705669/pengertian-dan-ciri-ciri-kata-kunci pada 9 Desember 2021)

[5] Programiz. *Rabin-Karp Algorithm*. (diakses dari https://www.programiz.com/dsa/rabin-karp-algorithm pada 11 Desember 2021)

[6] GeeksforGeeks. 2021. *Rabin-Karp Algorithm for Pattern Seraching.* (diakses dari https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/ pada 11 Desember 2021)

[7] GeeksforGeeks. 2021. *Priority Queue | Set1 (Introduction).* (diakses dari https://www.geeksforgeeks.org/priority-queue-set-1-introduction/ pada 12 Desember 2021)

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Madiun, 14 Desember 2021

Haidar Ihzaulhaq
13520150