Aplikasi Tabel Hash dalam Pengarsipan dan Pencarian Data

Jasson Prestiliano

Fakultas Teknologi Informasi
Universitas Kristen Satya Wacana
Jl. Diponegoro 52-60, Salatiga 50711, Indonesia
Email: jasprelao@yahoo.com

Abstract

Hash table is one of the data structure algorithms that can make sorting and searching easier. It distributes data to rows and columns with certain keys. It also cuts the time cost of data searching, because it searches the data from certain row and column from the memory by using the key. Besides, it can be implemented in pattern finding in search engine, data archiving and many other applications. This article explains the basic of hash table, its advantages and how to overcome data collisions. It also provides an example of hash table implementation in pattern finding.

Key Words: Hash Table, Data Structure, Data Searching

1. Pendahuluan

Dalam dunia teknologi informasi, menyimpan dan mencari data masih menjadi suatu hal yang membutuhkan banyak waktu dan usaha. Algoritma yang paling tua dan paling lambat untuk mencari suatu data adalah *sequential search*, atau sering pula disebut dengan *brute search*, dimana algoritma ini akan melakukan penelusuran dari data asal sampai dengan data yang terakhir kemudian tiap data yang ditelusuri tersebut dibandingkan dengan data yang dicari. Algoritma ini cukup akurat dalam mencari data, namun algoritma ini membutuhkan waktu yang sangat panjang apalagi bila data yang dicari ternyata berada pada urutan terakhir. Oleh karena itu algoritma ini sudah jarang dipakai untuk melakukan pencarian data yang berskala besar. Banyak algoritma yang dikembangkan untuk meningkatkan efisiensi dan mengurangi biaya dalam penyimpanan dan pencarian data.

Algoritma pencarian *sequential search* membutuhkan waktu operasi sebesar O(n), dengan n adalah jumlah operasi yang dibutuhkan untuk menelusuri dan membandingkan data yang dicari dengan seluruh data yang ada. Sedangkan algoritma pencarian dengan *Binary Search Tree* memungkinkan waktu operasi pencarian lebih efisien sehingga mencapai O(log n). Algoritma yang dibutuhkan setidaknya dapat melakukan pencarian dalam waktu yang jauh lebih cepat, yaitu O(1) atau O(satu) yang berarti cukup satu operasi saja untuk menemukan data yang dicari [1].

Salah satu algoritma yang dapat dipergunakan untuk menyimpan dan mencari data dengan lebih cepat dan efisien adalah algoritma yang disebut dengan algoritma tabel hash atau *hash table* [2].

Algoritma tabel hash sudah banyak dikembangkan dan dipergunakan di dalam berbagai bidang karena memiliki kelebihan dalam efisiensi waktu operasi dari pengarsipan dan pencarian suatu data. Beberapa bidang yang telah menggunakannya adalah bidang jaringan komputer, dimana tabel hash yang telah dikembangkan menjadi tabel hash terbuka hibrida atau *hybrid open hash table* digunakan untuk memproses jaringan komputer [3], dan banyak digunakan pada sistem jaringan *ad hoc* yang bergerak (*mobile ad hoc networks*) [4]. Selain itu tabel hash yang dikembangkan menjadi tabel hash terdistribusi atau *distributed hash table* dapat diimplementasikan untuk mencari dan menemukan berbagai macam *web service* di Internet [5].

Bidang lain yang menggunakan algoritma tabel hash adalah bidang bioinformatika. Tabel hash yang dipadukan dengan algoritma Teiresias dapat dipergunakan sebagai algoritma untuk mencari pola pada rantai DNA yang memiliki simbol-simbol abjad, sehingga dapat menentukan sifat suatu makhluk dari banyaknya pola yang sama yang ditemukan pada rantai DNA [6]. Tabel hash juga digunakan di dalam bidang geografi, yaitu sebagai pengarsipan dan pencarian data pada sistem informasi geografis yang dikenal dengan nama *geographic hash table (GHT)* [7].

Artikel ini membahas tentang dasar-dasar tabel hash mulai dari pengertian, bagaimana cara membuat sebuah tabel hash sederhana dan kelebihan-kelebihan dari algoritma tabel hash. Di samping itu, artikel ini juga membahas metode-metode penanganan duplikasi kunci pada data atau yang dikenal dengan sebutan *collision* pada tabel hash. Artikel ini dilengkapi pula dengan sebuah contoh program yang mengimplementasikan tabel hash dalam pengarsipan atau penyimpanan kalimat dengan pemetaan ke dalam sebuah tabel hash serta bagaimana algoritma tabel hash dapat melakukan pencarian pola pada kalimat tersebut.

2. Dasar-dasar Tabel Hash

Tabel hash adalah sebuah cara untuk menyimpan suatu data pada memori ke dalam baris-baris dan kolom-kolom sehingga membentuk tabel yang dapat diakses dengan cepat [1]. Setiap sel yang merupakan pertemuan antara baris dan kolom di dalam tabel hash dimana data diarsipkan disebut dengan *slot* [2].

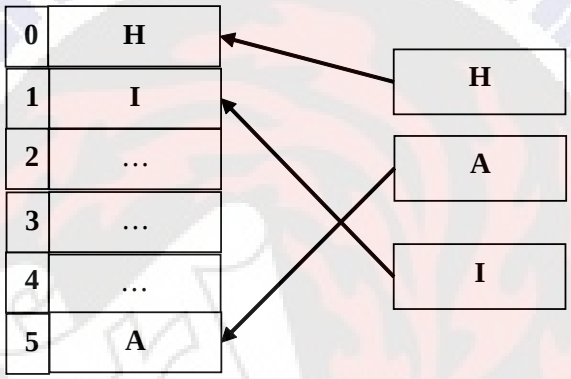
Sebuah tabel hash dapat dibuat dari dua bagian, yaitu sebuah larik atau *array* dan sebuah fungsi untuk memetakan, yang disebut dengan fungsi hash atau *hash function*. Fungsi hash adalah pemetaan ke dalam larik sesuai dengan kunci-kunci yang diberikan, dengan kata lain fungsi hash mendistribusikan data yang dimasukkan oleh pengguna ke dalam tabel atau larik yang telah disediakan.

Salah satu cara untuk menentukan kunci tabel hash adalah dengan menggunakan sistem *modulus* (sisa dari pembagian dua buah bilangan) dari

nilai integer pada data. Misalnya akan dibuat sebuah tabel hash yang akan memetakan huruf-huruf abjad dari A-Z, maka dapat dibuat sebuah kunci yang akan mengambil nilai ASCII dari huruf yang diinputkan, kemudian dibuat sebuah sistem *modulus* dari 6. Apabila masukan dari pengguna adalah kata "HAI", langkah pemetaan ke dalam tabel hash adalah:

1. Ambil nilai ASCII dari H (yaitu 72) lalu dibagi dengan 6, sisa hasil baginya (0) menjadi kunci untuk nilai H.
2. Ambil nilai ASCII dari A (yaitu 65) lalu dibagi dengan 6, sisa hasil baginya (5) menjadi kunci untuk nilai A.
3. Ambil nilai ASCII dari I (yaitu 73) lalu dibagi dengan 6, sisa hasil baginya (1) menjadi kunci untuk nilai I.

Gambar 1 memperlihatkan hasil pemetaan data masukan tersebut ke dalam tabel hash.



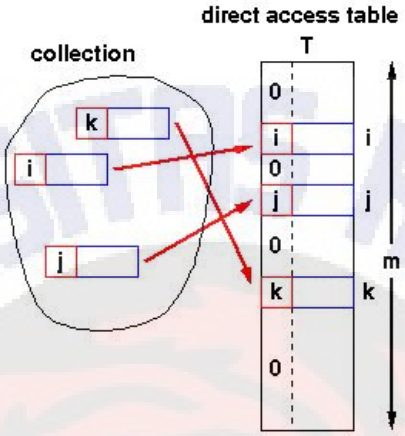
Gambar 1 Pemetaan Data ke Dalam Tabel Hash

Selain contoh seperti pada Gambar 1, terdapat pula sebuah metode tabel hash yang disebut dengan tabel akses langsung atau *direct access table*, dimana cara ini akan memetakan data dan mengarsipkannya pada memori dengan kunci yang berupa alamat memori, sehingga ketika melakukan pencarian, metode ini akan mengakses langsung alamat memori dari data yang dicari.

Sebagai contoh metode tabel akses langsung pada algoritma tabel hash adalah diketahui ada kumpulan *n* elemen yang memiliki kunci *integer* yang unik yaitu (1, *m*), dengan $m \geq n$, kemudian data tersebut disimpan ke dalam sebuah tabel alamat memori langsung atau *direct address table*, T[*m*], dimana T_i kosong atau berisi satu dari elemen yang telah dikumpulkan.

Misalnya ingin dicari elemen pada sebuah kunci, yaitu *k*, di dalam sebuah tabel hash sederhana dapat dilakukan dengan mengakses langsung alamat memori (T_k). Dengan begitu tabel hash memiliki waktu operasi O(1) atau satu operasi untuk menemukan data yang dicari. Hasil dari pencarian tersebut adalah: 1) Jika kunci *k* memiliki suatu elemen, maka fungsi hash akan mengembalikan nilai elemen tersebut, 2) Jika kunci *k* tidak memiliki suatu elemen, maka fungsi hash akan mengembalikan nilai NULL.

Tabel hash sederhana yang dibuat dengan mengakses alamat memori secara langsung tersebut dikenal dengan sebutan fungsi hash sempurna atau *perfect hash function* karena tabel hash tersebut secara sempurna memetakan setiap elemen satu demi satu dengan alamat memori yang unik, sehingga waktu operasi O(1) dapat dicapai. Hal ini diilustrasikan oleh Gambar 2.



Gambar 2 Memasukkan Data pada Tabel Alamat Memori secara Langsung

Untuk mewujudkan fungsi hash sempurna**,** diperlukan dua buah syarat krusial, yaitu: (1) Kunci yang dibuat harus unik dan (2) Jangkauan atau rentang kunci harus mencakup semua elemen yang akan dimasukkan [2].

Tabel hash sederhana yang dilakukan dengan metode pengalamatan langsung pada memori ini membutuhkan fungsi h(*k*) yang mampu melakukan pemetaan satu-satu, atau sering disebut dengan korespondensi satu-satu untuk setiap kunci *k* sampai dengan *integer* pada (1, *m*). Fungsi tersebut dikenal dengan fungsi hash sempurna (*perfect hashing function*). Fungsi hash sempurna ini memetakan setiap kunci pada *integer* yang terbatas dalam suatu jangkauan yang masih dapat diatur dan memungkinkan pengguna tabel hash untuk mencapai waktu pencarian O(1).

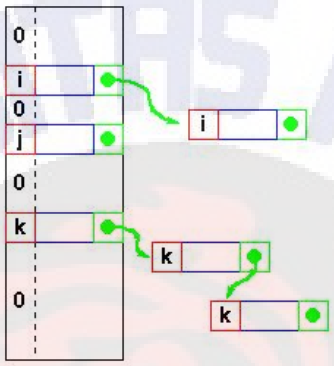
Sayangnya, merancang dan mengimplementasikan sebuah fungsi hash sempurna tidak selalu berhasil. Apabila telah dibuat sebuah sebuah fungsi hash, h(*k*), yang memetakan sebagian besar dari kunci pada *integer* yang unik, tetapi juga memetakan sejumlah kecil kunci pada *integer* yang sama.

Hal ini menimbulkan duplikasi kunci yang disebut dengan *collision*, yaitu suatu kasus dimana lebih dari satu kunci dipetakan ke dalam suatu *integer* yang sama. Apabila jumlah *collision* tidak begitu banyak, maka tabel hash masih dapat bekerja dengan baik dan tetap mencapai waktu pencarian O(1) dengan konsekuensi ada beberapa data minor yang hilang. Namun kemungkinan untuk terjadi *collision* pada fungsi hash tersebut belum tentu hanya dalam jumlah yang kecil.

Karena tidak mungkin untuk menebak berapa banyak data yang akan dimasukkan, maka tidaklah mungkin untuk menyediakan jangkauan atau

rentang kunci yang unik untuk mengolah setiap elemen yang akan dimasukkan ke dalam tabel hash.

Keterbatasan ini akan menyebabkan elemen-elemen data yang dimasukkan kemudian akan bertabrakan dengan elemen-elemen data yang dimasukkan terlebih dahulu apabila dalam penentuan kunci ternyata elemen terakhir tersebut memiliki satu macam kunci (misalnya suatu nilai *integer*) yang sama. Hal ini menyebabkan *collision.* Hal ini dapat dilihat pada Gambar 3.



Gambar 3 *Collision* antar Elemen yang Memiliki Kunci yang Sama

Jumlah *collision* yang cukup banyak tersebut dapat menjadi masalah, karena akan menyebabkan banyaknya data yang hilang. Untuk itu perlu dicari suatu cara agar jumlah *collision* tersebut dapat diminimalisir.

3. Teknik Penanganan Duplikasi Kunci (*Collision*) Tabel Hash

Pada beberapa kasus, dimana beberapa kunci memetakan data ke dalam *integer* yang sama, maka elemen-elemen tersebut akan disimpan pada *slot* yang sama pada sebuah tabel hash. Beberapa teknik yang dapat digunakan untuk mengatasi masalah tersebut. Teknik-teknik tersebut yaitu *Hash Matrix, Chaining, Re-hashing, Overflow Area, Linear Probing* (mempergunakan *slot* terdekat), dan *Quadratic Probing.*

Teknik pertama yang dapat dipergunakan untuk menangani duplikasi yaitu Teknik *Hash Matrix.* Teknik ini adalah teknik paling sederhana dalam mengatasi duplikasi kunci pada tabel hash.

Teknik ini menggunakan larik atau *array* dua dimensi dan perulangan bersarang untuk memetakan setiap data yang memiliki kunci *integer* yang sama. Contoh dari *hash matrix* adalah :

```
int hashmatrix[10][10];
```

Apabila kunci yang menunjuk pada *hashmatrix*[1][1] dan *integer* kunci berada pada indeks yang pertama sudah terisi suatu data, maka data yang memiliki kunci yang sama akan disimpan pada *hashmatrix*[1][2] dan

seterusnya. Sebaliknya bila kunci pada index kedua maka hasilnya data yang terakhir akan disimpan pada *hashmatrix*[2][1].

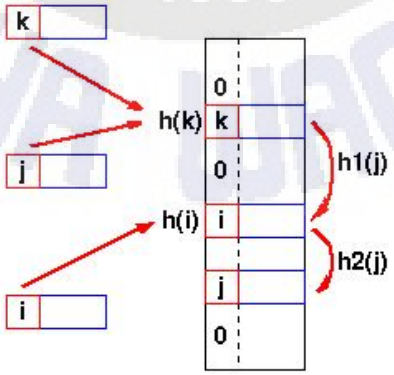
Teknik *hash matrix* tersebut memiliki kelebihan dari segi kesederhanaan program dan kemudahan dalam mempelajarinya. Namun teknik ini memiliki kelemahan, yaitu apabila alamat memori untuk menyimpan data yang ditunjukkan pada salah satu indeks sudah mencapai batas dari larik, maka data selanjutnya yang masuk pada kunci yang sama sudah tidak dapat ditangani lagi [1]. Teknik ini dipakai dalam contoh program yang dibuat.

Salah satu teknik lain yang masih cukup sederhana adalah dengan mengaitkan semua duplikasi kunci di dalam sebuah daftar berkait atau *linked-list* pada setiap data yang terdapat pada *slot* dimana terjadi duplikasi kunci. Teknik ini disebut dengan *chaining*. Kelebihan dari teknik ini yaitu teknik ini memungkinkan penanganan jumlah duplikasi kunci yang tidak terbatas dan tidak membutuhkan suatu urutan khusus dalam menentukan alamat memori yang akan dikaitkan dan berapa banyak elemen yang saling berkaitan di dalam kumpulan data. Namun, teknik ini juga memiliki kelemahan, yaitu daftar berkait dapat memperpanjang waktu pencarian, karena program harus mencari terlebih dahulu letak dari alamat memori yang ditunjukkan oleh daftar berkait tersebut.

Teknik berikutnya, yaitu teknik *re-hashing* yang digambarkan pada Gambar 4. menggunakan operasi *hash* yang kedua apabila terjadi duplikasi kunci. Apabila terdapat duplikasi kunci yang lebih lanjut, maka teknik ini akan menggunakan *re-hashing* yang ketiga, keempat dan seterusnya sampai menemukan adanya *slot* yang kosong pada tabel.

Pada Gambar 4, dapat dijabarkan bahwa h(*j*)=h(*k*), jadi fungsi berikutnya akan menggunakan h1. Jadi apabila duplikasi kunci kedua terjadi, maka h2 akan digunakan, dan begitu seterusnya.

Fungsi untuk *re-hashing* dapat dibuat baik dari fungsi yang baru ataupun reaplikasi dari fungsi aslinya. Selama fungsi yang diaplikasikan ke dalam kunci ada pada aturan yang sama, maka kunci yang terlihat akan dapat dialokasikan.

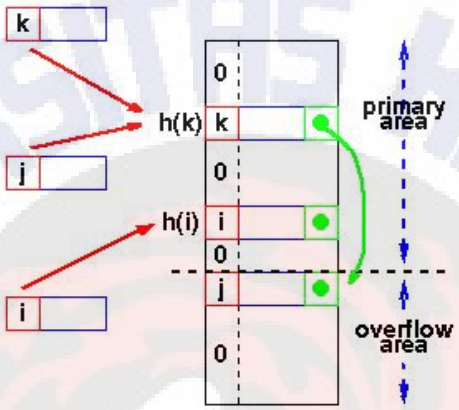


Gambar 4 Penggunaan Operasi Hash dalam Teknik *Re-Hashing*

Kelebihan teknik *re-hashing* adalah waktu yang dibutuhkan untuk melakukan fungsi *re-hashing* jauh lebih cepat dibandingkan dengan pencarian

alamat pada teknik *chaining*. Kelemahan teknik ini adalah jumlah maksimum dari elemen harus diketahui (jadi hanya berlaku untuk larik, yang jumlah maksimum indeksnya diketahui) dan dapat terjadi duplikasi kunci yang lain saat menghindari suatu duplikasi kunci tertentu [2].

Teknik lain yang dapat dilakukan adalah dengan membagi tabel pra-alokasi ke dalam dua bagian, yaitu area primer (*primary area*) untuk kunci-kunci yang telah dipetakan dan area luapan (*overflow area*) untuk menampung duplikasi kunci atau *collision*. Hal ini diperlihatkan pada Gambar 5.



Gambar 5 Penggunaan Overflow Area

Ketika sebuah *collision* terjadi, sebuah *slot* pada area luapan dipergunakan untuk elemen baru tersebut dan sebuah *link* dari slot pada area primer dibuat untuk menghubungkan antara elemen yang berada pada area primer dengan elemen yang berada pada area luapan. *Link* yang dibentuk ini menggunakan sistem seperti pada teknik *chaining*.

Teknik ini hampir serupa dengan teknik *chaining*, hanya saja area luapan telah dipraalokasikan, atau telah disiapkan sebelumnya, sehingga memungkinkan akses yang lebih cepat. Seperti pada teknik *re-hashing*, jumlah maksimum elemen harus diketahui, tetapi pada teknik area luapan ini, dua parameter harus ditentukan terlebih dahulu, yaitu ukuran yang optimal pada area primer maupun pada area luapan.

Teknik berikutnya untuk mengatasi duplikasi kunci adalah dengan melakukan fungsi *re-hashing* +1 atau -1, atau mencari *slot* terdekat dari data yang sudah ada, kemudian dilakukan sebuah perhitungan alamat baru yang cepat pada *slot* terdekat tersebut kemudian mengganti kunci dari *slot* tersebut agar sama dengan data terdekatnya. Teknik ini disebut juga dengan *Linear Probing*.

Teknik *Linear Probing* tersebut cukup efisien apabila data yang berdatangan memang memiliki kunci yang sama terus-menerus, tetapi apabila lama tidak ada data yang memiliki kunci sama, kemudian muncul data tersebut, maka waktu pemetaan akan memakan waktu lama, karena alamat-alamat terdekat telah dipakai untuk kunci yang lain.

Linear probing adalah penyebab terjadinya fenomena *clustering*. Fungsi-fungsi *re-hashing* dari satu lokasi yang mengalokasikan memori untuk sebuah blok yang terdiri dari banyak *slot* yang bertumbuh ke arah *slot* yang dimiliki oleh kunci lain. Hal ini bukannya mengatasi masalah duplikasi kunci, sebaliknya akan memperburuk keadaan dan jumlah dari fungsi *re-hashing*-pun menjadi banyak sekali.

Teknik yang lebih baik adalah *Quadratic Probing*, dimana fungsi hash sekunder bergantung pada indeks dari fungsi *re-hashing*. Hal ini dapat dinyatakan dalam rumus:

$$Alamat = h(kunci) + c\, i^2$$

Pada fungsi *re-hashing* ke-*t* maka fungsi *i* yang lebih kompleks dari dapat digunakan. Sejak kunci yang dipetakan ke dalam nilai yang sama dengan fungsi hash yang utama mengikuti urutan alamat yang sama. *Quadratic Probing* menunjukkan fenomena *clustering* sekunder, yang tidak sama dengan *clustering* yang terjadi pada *Linear Probing*.

Teknik *re-hashing* pada *Quadratic Probing* menggunakan ruang tabel yang dialokasikan secara khusus dengan menghindari terjadinya pengalokasian memori yang saling tumpang tindih dengan kunci lainnya. Oleh karena itu teknik ini membutuhkan perhitungan yang sangat rumit untuk mengetahui perkiraan jumlah data yang akan dimasukkan.

Saat ini sudah banyak pengembangan dari teknik-teknik untuk menangani duplikasi kunci, seperti teknik tabel hash terbuka hibrida atau *hybrid open hash table* [3], teknik tabel hash terdistribusi atau *distributed hash table* [5] dan masih banyak lagi teknik-teknik lainnya.

Dari pembahasan tersebut dapat dibuat suatu daftar kelebihan dan kekurangan pada setiap teknik yang telah dibahas. Tabel 1 mendaftarkan kelebihan dan kekurangan pada empat teknik yang pertama, yaitu *hash matrix, chaining, re-hashing* dan *overflow area*.

4. Aplikasi *Hash Table* dalam Pencarian Pola Kalimat

Aplikasi yang telah dikembangkan untuk menerapkan prinsip-prinsip tabel hash adalah aplikasi untuk mencari suatu pola tertentu dari kalimat yang dimasukkan oleh pengguna. Pola kalimat ini dicari berdasarkan sebuah pola yang juga dimasukkan oleh pengguna, jadi dengan pemetaan tabel hash pola yang diinginkan tersebut dicocokkan dengan kalimat yang telah dimasukkan sebelumnya. Apabila terdapat sebuah kecocokan dalam pola kalimat tersebut, maka aplikasi menelusuri kembali kalimat tersebut untuk mencari pola yang sama dengan pola yang dimasukkan tersebut sampai kalimat selesai ditelusuri. Pola yang ditemukan tersebut kemudian ditampilkan pada posisi keberapa pola tersebut muncul di dalam kalimat.

Tabel 1 Kelebihan dan Kekurangan Beberapa Teknik untuk Mengatasi *Collision*

Teknik	Kelebihan	Kekurangan
Hash Matrix	• Pra-alokasi memori sehingga akses lebih cepat • Larik dapat dibuat hingga beberapa dimensi, sehingga data yang ditampung bisa lebih banyak	• Jumlah elemen terbatas pada jangkauan data tertentu sesuai dengan pra-alokasi, jadi apabila telah melampaui batas maka data tidak dapat dialokasikan lagi
Chaining	• Dapat menangani banyak elemen yang tidak terbatas • Dapat menangani banyak collision yang tidak terbatas	• Terlalu banyak link-list sehingga memperlambat akses
Re-hashing	• Re-hashing yang cepat dimungkinkan • Akses yang cepat melalui penggunaan ruang tabel utama	• Jumlah maksimum elemen harus diketahui • *Collision* yang terjadi bisa lebih dari sekali
Overflow area	• Akses cepat • Collision tidak terjadi pada ruang tabel utama	• Dua parameter harus ditentukan di awal

Sebagai contoh terdapat kalimat “AKU BAWA PAKU”. Setiap karakter di dalam kalimat tersebut akan diberi nilai berdasarkan posisi tiap karakter dalam kalimat tersebut. Kalimat tersebut dipecah menjadi larik atau array agar setiap karakternya memiliki indeks posisi. Gambar 6 memperlihatkan susunan kalimat “AKU BAWA PAKU” dan indeks posisi tiap karakter dalam kalimat tersebut dimulai dari angka 1 yang berarti posisi pertama dan seterusnya.

Karakter	A	K	U		B	A	W	A		P	A	K	U
Posisi	1	2	3	4	5	6	7	8	9	10	11	12	13

Gambar 6 Pemberian Nilai Posisi pada Setiap Karakter

Setelah itu setiap karakter di dalam kalimat ini akan dipetakan ke dalam tabel hash yang sudah dibentuk. Data yang terdapat di dalam tabel tersebut adalah posisi dari setiap karakter di dalam kalimat dengan karakter spasi yang dihilangkan, *slot* data yang tidak terisi posisi di dalam kalimat akan diberi nilai 0. Misalnya, semua karakter A dipetakan di dalam sebuah kolom A, di

mana karakter A ada pada posisi 1, 6, 8 dan 11 dalam kalimat “AKU BAWA PAKU”, karakter B ada pada posisi 5 dan seterusnya. Penelusuran karakter-karakter pada kalimat yang dimasukkan pengguna tidak *case sensitive*, sehingga huruf kapital dan huruf kecil dianggap sama. Hasil pemetaan posisi-posisi karakter tersebut ke dalam tabel hash yang sudah terbentuk diperlihatkan oleh Gambar 7.

..	P	..	U	V	W	..	A	B	..	K	..
0	10	0	3	0	7	0	1	5	0	2	0
0	0	0	13	0	0	0	6	0	0	12	0
0	0	0	0	0	0	0	8	0	0	0	0
0	0	0	0	0	0	0	11	0	0	0	0

Gambar 7 Pemetaan Setiap Karakter ke dalam Tabel Hash

Kemudian pengguna memasukkan sebuah string sebagai pola yang ingin dicari. Misalnya pengguna memasukkan string “AKU”. Aplikasi yang dibuat pertama-tama mencari dahulu posisi setiap karakter pertama dari pola (A), kemudian baru mencari karakter-karakter selanjutnya.

Bila menemukan karakter pertama pada pola, aplikasi menyimpan posisi dari karakter tersebut ke dalam memori untuk kemudian dicari karakter kedua dalam pola yaitu karakter selanjutnya dari posisi karakter awal pada pola. Apabila karakter selanjutnya tersebut cocok dengan karakter kedua pada pola, maka aplikasi menyimpan lagi posisi dari karakter kedua untuk mencari karakter ketiga pada pola dan seterusnya. Apabila semua karakter pada pola telah ditemukan, maka aplikasi memberikan nilai dari posisi karakter awal dari pola yang ditemukan tersebut. Sebaliknya bila karakter pada posisi kedua tidak cocok dengan pola yang diinginkan, maka aplikasi meneruskan pencarian pada karakter yang lain di dalam kalimat.

Pada Gambar 7, diketahui bahwa ’A’ yang pertama ada di posisi 1, maka program akan memeriksa apakah posisi 2 adalah karakter ’K’ dan apakah posisi 3 adalah karakter ’U’, jika benar maka, kembalikan nilai posisi dari karakter pertama. ’A’ yang kedua ada pada posisi 6, kemudian diperiksa apakah posisi 7 adalah karakter ’K’, ternyata tidak karena di posisi 7 adalah karakter ’W’, maka penelusuran selesai lalu program akan mencari posisi karakter ’A’ yang lain, dan begitu seterusnya sampai semua karakter ’A’ selesai diproses. Jadi karakter yang diperiksa hanyalah pada posisi 1, 6, 8 dan 11 sesuai dengan pemetaan karakter ’A’ pada tabel hash. Hal ini mempersingkat waktu pencarian pola karena tidak perlu menelusuri karakter pada kalimat “AKU BAW PAKU” satu persatu.

Pencarian pola “AKU” pada kalimat “AKU BAWA PAKU” tersebut menghasilkan 2 (dua) kali penemuan, yaitu pada posisi 1 (satu) dan 11 (sebelas).

Untuk mewujudkan hal tersebut, dibuat sebuah program dengan menggunakan algoritma teknik tabel. Sistem kunci yang dipakai untuk pemetaan ke dalam tabel hash adalah sistem modulus dari huruf A sampai dengan Z, jadi setiap karakter masukan akan dikonversi menjadi huruf kapital atau *Uppercase*, kemudian akan diambil nilai ASCII dari karakter tersebut.

Nilai ASCII tersebut yang akan dibagi dengan nilai 26 sebagai jumlah dari karakter abjad dengan menghilangkan karakter-karakter selain huruf A sampai dengan Z.

Untuk menangani duplikasi kunci digunakan sistem *hash matrix* atau larik 2 (dua) dimensi dengan salah satu indeks dari larik menjadi kunci dan indeks yang lain akan diisi oleh data. Data yang disimpan di dalam tabel hash ini adalah posisi dari tiap karakter dalam kalimat yang dimasukkan. Pemetaan dan penyimpanan data ke dalam tabel hash ini sering pula disebut dengan pengarsipan. Penggalan program 1 memperlihatkan cara untuk memetakan data ke dalam *hash table*. Program diimplementasikan dengan menggunakan bahasa C.

Penggalan program dalam Bahasa C yang berfungsi sebagai pemetaan data ke dalam Hash Table dapat dilihat pada Kode Program 1

Kode Program 1 Pemetaan Data ke dalam Tabel Hash

```
int hashMatrix[50][26], row, column, pos;
void createHash(char sentence[]){
   int value, i;
   pos=1;
   for(i=0;i<strlen(sentence);i++){
      sentence[i]=toupper(sentence[i]);
   }
   for(i=0;i<strlen(sentence);i++){
      if(sentence[i]!=' '){
         value = (int) sentence[i];
         column = value % 26;
         row=0;
         while(hashMatrix[row][column]!=0)
            row++;
         hashMatrix[row][column]=pos;
         pos++;
      }
   }
}
```

Gambar 8 memperlihatkan masukan kalimat dari pengguna yang kemudian akan menjadi acuan untuk melakukan teknik *hash matrix*, di mana masukan kalimat tersebut adalah "Fakultas Teknologi Informasi Program Studi Teknik Informatika dan Sistem Informasi Universitas Kristen Satya Wacana" yang terdiri dari 101 karakter di luar spasi.

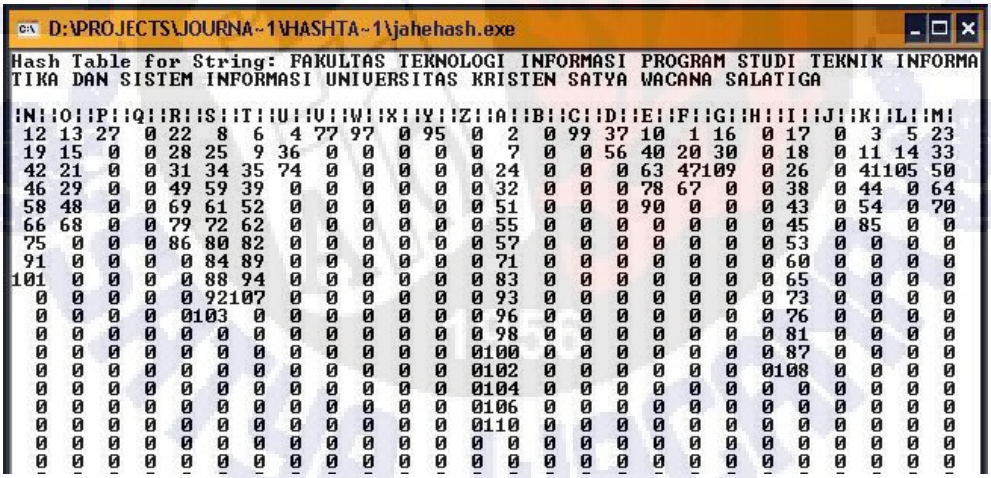
Gambar 8 Masukan Kalimat dari Pengguna

Karakter-karakter dalam kalimat yang dimasukkan pengguna seperti pada Gambar 8 tersebut didistribusikan ke dalam larik dan kemudian diberi indeks posisi mulai dari 1. Kemudian karakter-karakter tersebut dipetakan ke dalam tabel hash yang berupa matriks. Misalnya karakter 'F' berada pada posisi 1, 20, 47 dan 67, karakter 'K' berada pada posisi 3, 11, 41, 44, 54 dan 85 dan seterusnya. Pemetaan tersebut diperlihatkan pada Gambar 9.

Sebagai contoh, salah satu karakter yang dimasukkan pada kalimat adalah huruf 'I'. Huruf 'I' memiliki nilai ASCII dalam desimal yaitu 73. Angka 73 ini dibagi dengan 26 sebagai kunci dalam tabel hash. Sisa hasil pembagian dari angka 73 tersebut adalah 21. Maka setiap menemukan karakter huruf 'I', posisi karakter tersebut dipetakan ke kolom 21 dari matriks hash yang telah dibuat. Bila posisi baris pertama pada kolom tersebut sudah terisi, maka posisi karakter tersebut disimpan pada baris di bawahnya dan begitu seterusnya sampai dengan maksimal 50 karakter 'I' yang dapat dipetakan posisi karakternya.

Pada aplikasi yang dikembangkan, matriks hash yang disediakan memiliki 50 baris untuk masing-masing kolomnya yang berjumlah 26 sesuai dengan jumlah huruf di dalam abjad. Sehingga untuk setiap karakter bisa mencapai maksimal 50 buah dalam suatu kalimat.

Gambar 9 memperlihatkan hasil pemetaan tabel hash dari kalimat yang dimasukkan oleh pengguna.



Gambar 9 Hasil Pemetaan Kalimat ke dalam Tabel Hash

Untuk membuktikan bahwa data telah tersimpan dan dipetakan, dibuat sebuah sistem pencarian per karakter untuk mencari dan akan menghasilkan posisi dari huruf-huruf tersebut di dalam kalimat masukan yang diambil dari tabel hash, *source code* untuk mencari karakter pada tabel hash bisa dilihat pada Kode Program 2, dimana program ini merepresentasikan sebuah fungsi bernama *searchLetter* yang mencocokkan setiap karakter pada matriks hash dengan karakter yang dicari, karena pencarian dilakukan dengan mengambil sisa hasil bagi (modulus) dengan jumlah abjad yaitu 26, maka waktu pencarian O(1) tercapai.

Sebagai contoh, karakter yang dicari adalah huruf 'I'. Huruf 'I' memiliki nilai ASCII dalam desimal 73 dan bila dibagi dengan angka 26 sebagai kunci hash menghasilkan sisa 21. Angka 21 ini merupakan kolom yang perlu ditelusuri, sehingga tidak perlu menelusuri kolom yang lain. Untuk pencarian karakter 'I', aplikasi langsung mengambil posisi-posisi karakter dalam kalimat yang ada pada kolom ke-21 dari matriks hash saja. Hal inilah yang mempersingkat waktu operasi dari pencarian karakter.

Kode Program 2 Pencarian Karakter yang Memanfaatkan Fungsi Hash

```
int searchLetter(char letter)
{
   int find=0;
   row=0;
   column=(int) letter % COL;
   for(int i=0;i<ROW;i++)
   {
      if(hashMatrix[row][column]!=0)
      {
         find++;
         where[i]=hashMatrix[row][column];
         row++;
      }
```

Dengan menerapkan Kode Program 2 tersebut, maka hasil keluaran dari pencarian karakter dapat dlihat seperti pada Gambar 10.

Gambar 10 Hasil Pencarian Posisi Karakter dari Tabel Hash

Pencarian pola sebenarnya menggunakan prinsip yang sama pada pencarian karakter, hanya saja perlu disimpan karakter yang telah ditemukan untuk kemudian dicocokkan pada pola yang diinginkan. Caranya adalah temukan dulu karakter awal dari pola yang akan dicari, kemudian baru lakukan pencarian pada karakter-karakter berikutnya dari pola tersebut. Hasil pencarian pada aplikasi ini akan menghasilkan posisi dari pola yang dicari apabila ditemukan.

Di sini sistem pencarian dibuat mirip dengan Kode Program 2. Jumlah operasi pencarian akan sama dengan jumlah karakter pertama yang ditemukan, kemudian apabila karakter kedua dan seterusnya sama, maka akan dikembalikan nilai dari posisi karakter pertama dari pola yang dicari, sedangkan bila salah satu karakter kedua atau lanjutannya berbeda dari pola yang diinginkan, maka akan muncul pesan bahwa pola tidak ditemukan. Untuk melakukan pencarian pola, dibuat suatu fungsi *searchPattern* seperti yang tertera pada Kode Program 3.

Kode Program 3 Pencarian Pola yang memanfaatkan Fungsi Hash

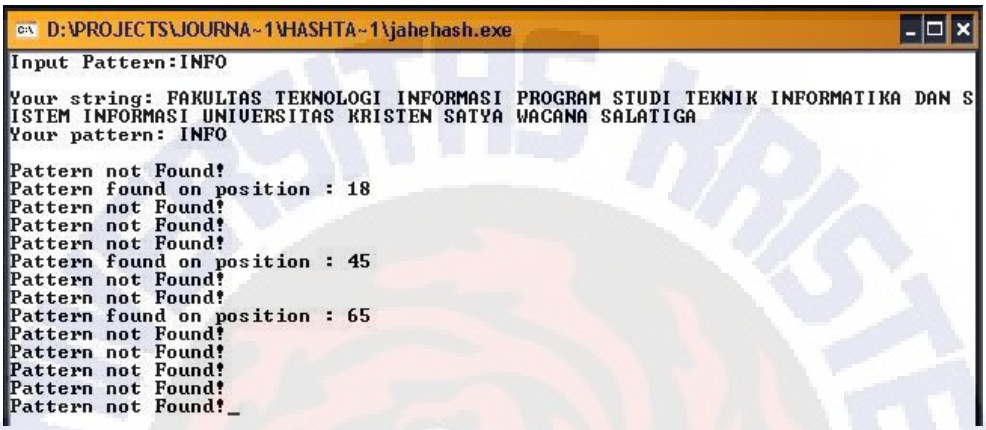
```
int searchPattern(char pattern[],char letter,int k, int
length, int pos)
{
   int findLetter=0;
   int i,state=0,next,compare;
   next=pos+1;
   state=searchLetter(letter);
   if(letter!='\0')
   {
      if(state!=0)
      {
         for(int j=0;j<ROW;j++)
         {
            compare=where[j];
            if(next==compare)
            {
             return(searchPattern(pattern,pattern[k+1],
             k+1,length,where[j])+1);
            }
         }
      }
   }
   else
   {
      return 0;
   }
}
```

Sebagai contoh, pola yang dimasukkan oleh pengguna untuk dicari yaitu pola “INFO”. Karakter awal dari pola ini adalah karakter ‘I’. Sesuai dengan perhitungan kunci hash, karakter ’I’ berada pada kolom 21 pada matriks hash. Jadi penelusuran dimulai pada posisi karakter yang terdapat pada baris pertama kolom ke-21 dari matriks hash.

Posisi karakter pertama tersebut pada contoh berada pada posisi ke-17 dari kalimat yang dimasukkan. Penelusuran dilanjutkan dengan posisi selanjutnya dari karakter ’I’ yang pertamakali ditemukan. Jika pada posisi ke-18 ditemukan karaker ’N’, maka penelusuran dilanjutkan ke karakter berikutnya pada pola yang dicari dan seterusnya sampai pola ’INFO’ terpenuhi. Jika sudah terpenuhi, kembalikan posisi karakter pertama. Bila hal tersebut tidak terjadi, maka penelusuran diubah ke baris kedua pada matriks hash, dan begitu seterusnya sampai semua karakter pertama pada pola selesai ditelusuri.

Pada contoh, saat menelusuri posisi ke-17, aplikasi tidak berhasil menemukan pola yang dicari, karena posisi ke-18 ternyata dihuni oleh karakter ’I’, baru kemudian pada penelusuran posisi ke-18 ditemukan pola yang dicari tersebut, di mana posisi ke-19 merupakan karakter ’N’, posisi ke-20 ditempati oleh karakter ’F’ dan posisi ke-21 ditemukan karakter ’O’ sehingga membentuk pola yang dicari yaitu “INFO”. Tampilan pada program adalah posisi karakter ’I’ di mana pola tersebut ditemukan, yaitu pada posisi ke-18.

Jumlah operasi pencarian akan sama dengan jumlah karakter pertama yang ditemukan, kemudian apabila karakter kedua dan seterusnya sama, maka akan dikembalikan nilai dari posisi karakter pertama dari pola yang dicari, sedangkan bila salah satu karakter kedua atau lanjutannya berbeda dari pola yang diinginkan, maka akan muncul pesan bahwa pola tidak ditemukan. Perhatikan Gambar 11 sebagai contoh hasil keluaran dari pencarian pola dalam kalimat.



Gambar 11 Hasil Pencarian Pola dari Tabel Hash

5. Simpulan

Algoritma *hash table* dapat diimplementasikan untuk mengarsipkan dan mencari suatu data dengan waktu operasi mencapai **O(1)** karena *hash table* akan memetakan data sesuai dengan kunci yang telah diberikan.*Hash table* dapat dikembangkan untuk mencari pola-pola tertentu dalam sebuah *string* atau kalimat, *hash table* berfungsi untuk memetakan urutan dari setiap karakter. Bila ada suatu pola yang dicari, sistem pencarian yang digunakan adalah mencari karakter pertama, bila karakter pertama ditemukan baru kemudian dicocokkan sesuai dengan urutan karakter pada pola yang dicari dan begitu seterusnya sampai kepada karakter terakhir pada pola.

Dengan demikian *hash table* mempersingkat waktu pencarian, karena bila ada satu karakter saja yang tidak cocok, maka pencarian langsung berpindah pada bagian selanjutnya, jadi tidak mencari secara sekuensial.

Hash table memiliki kendala, yaitu *collision*, untuk itu diperlukan penanganan secara khusus untuk mengatasi kendala tersebut. Beberapa penanganan yang dapat dilakukan antara lain menggunakan *hash matrix*, *chaining*, *re-hashing, linear probing* dan *quadratic probing*. Dalam contoh, kendala ini diatasi dengan membuat *hash matrix* dengan kolom berupa jumlah karakter-karakter pada abjad (A-Z) yaitu 26 kolom, sedangkan barisnya disiapkan untuk karakter-karakter yang akan dipetakan ke dalam *hash matrix* tersebut.

Algoritma pencarian pola di dalam *string* dapat dikembangkan ke dalam pengarsipan dan pencarian data *web* maupun *web service* di Internet pada *search engine* atau mencari pola dalam rantai-rantai DNA.

6. Daftar Pustaka

[1] Barnes & Noble, Hash Tables, *Sparknotes*, http://www.sparknotes.com/cs/searching/hashtables/section1.html.

[2] Department of Computer Science, the University of Auckland, *Data Structures and Algorithm*, http://www.cs.auckland.ac.nz/software/AlgAnim/hash_tables.html.

[3] Qing Ye, Dale Parson and Liang Cheng, 2005, Hybrid Open Hash Tables for Network Processors, *IEEE Journal*, http://www.ieee.com.

[4] Tobias Heer and Co., 2006, Adapting Distributed Hash Tables for Mobile Ad Hoc Networks, *IEEE Journal*, http://www.ieee.com.

[5] Lin, Quanhao, Ruonan Rao, Minglu Li, 2006, DWSDM: A Web Services Discovery Mechanism Based on a Distributed Hash Table, *IEEE Journal*, http://www.ieee.com

[6] Wolinsky, Murray, 1999, *An Experimental Implementation of the TEIRESIAS Algorithm*, STANFORD UNIVERSITY BIOCHEMISTRY,

[7] Ratnasamy, Sylvia and Co., 2003, Data-Centric Storage in Sensornets with GHT, a Geographic Hash Table, *Springerlink Journal Article*, 8(4) Agustus 2003, http://www.springerlink.com.