BAB X
Hashing

Tujuan

1. Menunjukkan beberapa fungsi metode Hash
2. Dapat memilah permasalahan yang dapat diselesaikan dengan metode Hashing, sekaligus dapat menyelesaikannya

---

Pada metode-metode pencarian yang telah kita pelajari, secara umum banyaknya pembandingan untuk mencari data atau rekaman yang diinginkan tergantung dari banyaknya data atau rekaman yang diketahui. Jika setiap data atau rekaman bisa ditemukan dengan sekali pemasupan terhadap tabel yang digunakan untuk menyimpan data atau rekaman tersebut, maka lokasi data atau rekaman dalam tabel hanya tergantung dari kunci yang digunakan dan tidak tergantung dari kunci yang lain, seperti dalam pohon. Cara yang paling efisien untuk mengorganisir tabel ini adalah dengan menggunakan larik. Jika kunci berupa integer, kunci tersebut sekaligus bisa digunakan sebagai subskrip dari larik yang dimaksud

Dengan mengacu pada penjelasan diatas, perhatikan conoth berikut. Sebagai contoh, suatu toko buku menjual 100 judul buku dan setiap buku mempunyai makssimum dua digit pengenal. Cara yang paling sederhana untuk menyimpan data di atas adalah

```
typedef char *NomorBuku[100];
Buku = NomorBuku;
```

Dimana `Buku[i]` menunjukkan buku yang mempunyai pengenal (nomor buku) i. Dalam contoh ini nomor buku dimanfaatkan sebagai subskrib larik `Buku`.

Dalam perkembangannya, toko buku tersebut berkembang dan menjual lebih dari 100 judul buku dan buku-buku tersebut dikelompokkan sedemikian rupa sehinggga masing-masing buku sekarang memerlukan kode dengan buah digit. Dengan menggunakan larik seperti contoh diatas, pemilik toko buku harus menyediakan larik dengan 10 juta elemen untuk menyimpan data buku yang ada. Hal ini tentu saja tidak praktis, sehingga diperlukan cara yang lebih efisien untuk mengatasi hal ini. Cara yang

dimaksud pada dasarnya adalah untuk mengkonversikan kode buku (yang memerlukan 7 digit) menjadi integer dalam batas tertentu. Secara ideal, masing-masing kode buku harus dikonversikan menjadi suatu integer yang berbeda, tetapi hal ini seringkali sukar untuk dilaksanakan.

Dengan menggunakan contoh diatas, toko buku tersebut mempunyai kurang dari 1000 judul buku, dan tetap menggunakan kode yang kurang dari 7 buah digit. Dengan demikian, pemilik toko buku tersebut cukup menyediakan larik dengan 1000 buah elemen yang nomor subskribnya 0...999, dan memanfaatkan 3 digit terakhir dari kode buku untuk menempatkan semua kode buku pada larik yang diketahui. Gambar 10.1 mengilustrasikan contoh ini. Dari gambar tersebut dapat dilihat, bahwa jarak dua buah kode buku hanya ditentukan oleh 3 digit terakhir, sehingga dua buku dengan kode 1002372 dan 1002772 akan mempunyai jarak yang lebih besar dibandingkan dua buku yang mempunyai kode 0001996 dan 5192998.

Dengan melihat pada contoh diatas, kita bisa memperhatikan bahwa pemilik toko buku tersebut memerlukan suatu fungsi untuk mengkonversikan kode buku ke nomor posisi dari larik yang diketahui. Fungsi ini disebut dengan fungsi *hash*. Metode pencarian yang memanfaatkan fungsi *hash* disebut *hashing* atau *hash addressing*. Tujuan utama dalam penentuan fungsi *hash* adalah agar dua buah kunci yang berbeda tidak mempunyai nilai *hash* yang sama. Jika hal ini terjadi, akan menyebabkan terjadinya tabrakan (*hash collision* / *hash clash*).

Posisi	Kunci	Data lain
0	102300	
1	123600	

Gambar 10.1 Contoh Konversi Kunci Menjadi Posisi

10.1 Fungsi Hash

Secara umum fungsi *hash* (*H*) adalah fungsi untuk mengkonversikan himpunan kunci rekaman (*K*) menjadi himpunan alaman pengingat (posisi subskrib dalam larik / *L*) dan bisa dituliskan dengan menggunakan notasi

$H : K \rightarrow L$

Dua aspek penting yang perlu dipertimbangkan dalam pemilihan fungsi *hash* adalah sebagai berikut. Pertama, fungsi *H* harus mudah dan cepat dicari atau dihitung. Kedua, fungsi *H* sebisa mungkin mendistribusikan posisi yang dimaksud ssecara uniform sepanjang himpunan L, sehingga banyaknya tabrakan yang mungkin terjadi bisa diminimalkan. Secara alamiah, tidak ada garansi yang memungkinkan bahwa aspek kedua bisa dipenuhi tanpa terlebih dahulu mengetahui kunci-kunci yang ada. Meskipun demikian, ada beberapa metode untuk memotong-motong kunci dalam himpunan K menjadi kombinasi tertentu yang akan dipakai sebagai fungsi *H.*

Berikut disajikan beberapa cara untuk memotong-motong kunci sehingga bisa diperoleh fungsi *hash* yang dengan mudah bisa dihitung.

10.1.1 Metode Pembagian

Dalam cara ini kita bisa memlih suatu perubah m yang nilainya lebih besar dibanding banyaknya kunci dalam K, misalnya n, dan biasanya dipilih suatu bilangan prima. Fungsi *hash*nya ditentukan sebagai :

$H(k) = k \bmod m$ atau $H(k) = k \bmod m + 1$

Persamaan pertama dipilih apabila diinginkan alamat kunci adalah 0 sampai $m - 1$. Persamaan kedua dipilih jika diinginkan alamat kunci adalah 1 sampai m.

Sebagai contoh, nomor mahasiswa terdiri dari 5 buah digit. Misalkan L terdiri dari 100 buah alamat yang masing-masing alamat terdiri dari 2 karakter : 00...99. Nomor mahasiswa yang diketahui misalnya 10347, 87492, 34212 dan 88688. Untuk menentukan alamat dari keempat nomor mahasiswa ini kita pilih suatu bilangan prima yang dekat dengan 99, misalnya $m = 97$. Dengan menggunakan fungsi $H(k) = k \bmod m$, diperoleh

$H(10347) = 65$, $H(87492) = 95$, $H(34212) = 68$, $H(88688) = 30$

Dengan demikian, nomor mahasiswa 10347 akan disimpan dalam alamat 65, nomor mahasiswa 87492 akan disimpan dalam alamat 95, nomor mahasiswa 34212 akan disimpan dalam alamat 68 dan nomor mahasiswa 88688 akan disimpan dalam alamat 30. Jika dipilih fungsi $H(k) = k \bmod m + 1$, maka keempat nomor mahasiswa diatas masing-masing akan disimpan dalam alamat 66, 96, 69 dan 31.

10.1.2 Metode Midsquare

Dalam metode ini, kunci yang diketahui dikuadratkan dan fungsi *hash* yang dipilih adalah :

$H(k) = l$

Nilai l diperoleh dengan menghapus digit-digit pada kedua sisi dari k^2, dengan catatan bahwa banyaknya digit di sebelah kiri dan sebelah kanan harus sama. Jika tidak sama, maka pada digit di sebelah kiri seolah-olah ditambahkan sejumlah *trailing zero*, sehingga akan menghasilkan alamat yang benar.

Dengan menggunakan contoh yang sama dengan diatas, maka alamat dari masing-masing nomor mahasiswa diatas adalah :

K	10347	87492	34212	88688
K^2	107060409	76548500564	1170460944	7865561344

$H(k)$	06	85	46	56

10.1.3 Penjumlahan Digit

Dalam penjumlahan digit, kunci yang diketahui bisa dipecah menjadi beberapa kelompok yang masing-masing terdiri dari beberapa buah digit, misalnya dua buah. Kemudian digit-digit dari kelompok-kelompok yang ada dijumlahkan. Pemecahan dan penjumlahan terus dilakukan jika jumlah keseluruhan kelompok yang ada masih lebih besar dari banyaknya alamat yang akan dipakai. Dengan menggunakan nomor mahasiswa diatas, maka alamat dari masing-masing nomor mahasiswa bisa ditentukan sebagai berikut (dalam hal ini digunakan kelompok dengan dua buah digit, karena alamatnya diketahui dari 00 sampai 99) :

$H(10347) = 1 + 03 + 47 = 51$

$H(87492) = 8 + 74 + 92 = 174 = 1 + 74 = 75$

$H(34212) = 3 + 42 + 12 = 57$

$H(88688) = 8 + 86 + 88 = 182 = 1 + 82 = 83$

10.2 Cara Mengatasi Tabrakan

Tujuan dari pemilihan fungsi *hash* adalah untuk menempatkan rekaman pada alamat tertentu, sehingga bisa dihindari adanya tabrakan, yaitu suatu keadaan dimana dua buah atau lebih rekaman yang mempunyai data kunci yang berbeda mampunyai alamat *hash* yang sama. Meskipun demikian, kekmungkinan adanya tabrakan selalu tetap saja terjadi, meskipun kita sudah menentukan fungsi *hash* yang cukup baik. Dengan demikian, kita harus mempunyai satu cara untuk mengatasi tabrakan yang mungkin terjadi, yang disebut dengan *collision resolution.*

Prosedur yang baik untuk mengatasi adanya tabrakan gayut antara lain terhadap perbandingan banyaknya data kunci (n) dalam K, dan banyaknhya alamat *hash* (m) dalam L. Perbandingan ini, $\lambda = n/m$, disebut faktor beban.

Lebih lanjut, efisiensi fungsi *hash* yang dilengkapi dengan prosedur untuk mengatasi tabrakan diukur dengan banyaknya pembandingan kunci (*probe*) yang diperlukan untuk mencari alamat dari rekaman yang mempunyai kunci k. Efisiensi ini gayut terhadap faktor beban dan diukur menggunakan dua besaran berikut ini :

$B(\lambda)$ = rata-rata *probe* untuk pencarian yang berhasil

$G(\lambda)$ = rata-rata *probe* untuk pencarian yang gagal

10.2.1 Pengalamatan Terbuka

Secara umum, cara mengatasi tabrakan dengan pengalamatan terbuka (*opej addressing*) bisa dijelaskan sebagai berikut. Dimisalkan sebuah rekaman dengan kunci *k* akan disisipkan ke dalam tabel alamat *hash*. Berdasarkan fungsi *hash* yang dipakai, alamat untuk kunci *k* tersebut dihitung, misalnya pada alamat *h*. Jika kemudian ternyata bhwa alamat *h* sudah terisi, maka harus dicari alamat lain yang masih kosong. Cara yang termudah adalah dengan mencari alamat berikutnya yang kosong. Cara ini disebut dengan *linear probing.*

Dari contoh diatas kita bisa melihat, bahwa untuk mencari rekaman dengan kunci *k*, harus dilakukan pencarian pada alamat *h*, *h* + 1, *h* + 2, ,,, dan seterusnya. Berdasarkan hal ini, rata-rata pencarian yang berhasil dan tidak berhasil adalah

$B(\lambda) = ½ (1 + 1/(1 - \lambda))$

$G(\lambda) = ½ (1 + 1/(1 - \lambda)^2)$

Untuk lebih memperjelas apa yang dimaksud dengan *linear probing*, berikut disajikan sebuah contoh :

Rekaman	A	B	C	K	P	Q	R	Y	Z
H(*k*)	5	6	7	5	0	1	2	9	0

Dimisalkan bahwa kesembilan rekaman diatas dimasukkan dengan urutan yang sama dengan urutan diatas. Maka rekaman-rekaman diatas akan tersimpan dalam pengingat sebagai

Rekaman	P	Q	R	Z	-	A	B	C	K	Y
H(*k*)	0	1	2	3	4	5	6	7	8	9

Dari conoth diatas kita bisa melihat bahwa rekaman A, B, C, P, Q, R dan Y menempati alamat yang tepat. Rekaman K, meskipun alamat *hash* sebenarnya adalah 5, tetapi karena alamat 5 sudah dipakai oleh A, maka diletakkan pada alamat berikutnya yang kosong. Demikian pula dengan rekaman Z, yang seharusnya menempati alamat 0, tetapi karena alamat 0 sudah dipakai oleh P, maka Z ditempatkan pada alamat 3.

Dari conoth diatas, bisa dihitung banyaknya *probe* rata-rata untuk pencarian yang berhasil dan gagal. *Probe* rata-rata untuk pencarian yang berhasil bisa dihitung dengan cara sebagai berikut. Karena A, B, C, P, Q, R dan Y, sudah menempati alamat yang seharusnya, maka untuk mencari informasi-informasi ini dan berhasil cukup dilaksanakan

dengan sebuah *probe*. Untuk mencari informasi yang lain, banyaknya *probe* yang diperlukan untuk pencarian yang berhasil adalah dengan menghitung banyaknya alamat dari alamat yang seharusnya informasi tersebut berada sampai alamat dimana informasi tersebut dicatat. Sebagai contoh, karena K sesungguhnya harus berada pada alamat 5, sedangkan kenyataannya K berada pada alamat 8, maka pencarian K yang berhsail memerlukan 4 *probe*. Dengan demikian, banyaknya *probe* rata-rata yang diperlukan untuk pencarian yang berhasil (dituliskan mulai dari A sampai Z) adalah

$$B = (1 + 1 + 1 + 4 + 1 + 1 + 1 + 1 + 4) / 9 = 15 / 9 = 1.667$$

Untuk mencari *probe* rata-rata untuk pencarian yang tidak berhasil bisa dilaksanakan dengan cara menjumlahkan probe yang diperlukan untuk mencari alamat kosong yang terdekat olehsetiap alamat yang ada. Hal ini bisa dipahami , karena jika pencarian sudah sampai pada suatu alamat yang kosong, maka data yang dicari pasti tidak akan ditemukan. Sebagai contoh, pencarian yang tidak berhasil untuk mencari P memerlukan 5 buah *probe* dihitung dari posisi P sampai tanda -, untu Q memerlukan 4 buah *probe* dan seterusnya. Dengan demikian, untuk contoh diatas, *probe* rata-rata yang diperlukan untuk pencarian yang tidak berhasil adalah

$$G = (5 + 4 + 3 + 2 + 1 + 10 + 9 + 8 + 7 + 6) / 10 = 55 / 10 = 5.5$$

Dari contoh yang telah disajikan diatas bisa dilihat satu kerugian yang utama dari *linear probing* ini adala data cenderung untuk mengumpul pada satu tempat. Hal ini bisa dipahami karena jika ada suatu data yang akan disisipkan pada suatu alamat dan alamat yang dimaksud sudah dipakai, maka data baru tersebut akan ditempatkan pada lokasi berikutnya yang letaknya berurutan. Kedua cara ini disebut dengan *quadratic probing* atau *double hashing*.

Dalam *quadratic probing*, jika alamat untuks uatu data baru yang akan disisipkan sudah dipakai (misalnya alamat h), maka data baru tersebut tidak ditempatkan pada posisi $h + 1$ atau $h + 2$ (alamat $h + 1$ juga sudah dipakai) dan seterusnya, tetapi data baru akan diletakkan pada alamat dengan urutan

h, $h + 1$, $h + 4$, $h + 9$, ...

Dengan demikian, pencarian akan dilaksanakan pada alamat diatas. Hal ini membawa keuntungan, bahwa jika banyaknya alamat yang tersedia adalah merupakan bilangan prima, cara diatas akan melakukan pencarian pada separuh dari seluruh alamat yang ada

Dalam *doubel hashing* yang digunakan dua buah fungsi *hash* untuk menghindari adanya tabrakan. Secara sederhana cara ini bisa dijelaskan sebagai berikut. Dari kunci *k* ditentukan alamat *hash*-nya yang pertama, misalnya $H(k) = h$. Kemudian ditentukan alamat *hash* yang kedua, misalnya $H'(k) = h' \neq m$ (denga *m* adalah banyaknya alamat *hash* yang dihasilkan darifungsi *hash* yang pertama). Dengan demikian, pencarian dilakukan secara urut pada alamat alamat

h, $h + h'$, $h + 2h'$, $h + 3h'$, ...

Satu kerugian yang cukup mendasar dalam sistem pengalamatan terbuka adaah sebagai berikut. Dimisalkan bahwa kita akan menyisipkan sebuah rekaman bru, misalnya *rek2*, yang akan menempati alamat *x* pada tabel *hash.* Tetapi karena alamat *x* sudah terisi, maka rekaman *rek2* ini akan ditempatkan pada lokasi kosong yang pertama sesudah alamat *x*, misalnya *x1*. Sekarang misalnya rekaman yang ada pada alamat *x* dihapus. Dengan demikian, maka alamat *x* sekarang menjadi kosong. Jika kemudian kita akan mencari rekaman *rek2* kita akan mendapatkan kenyataan bahwa program mungkin tidak akan mendapatkan kenyataan bahwa program mungkin tidak akan menemukan rekaman tersebut meskipun sesungguhnya ada. Sebabnya adalah bahwa pada saat rekaman *rek2* akan dicari, maka berdasar fungsi *hash* yang dipakai, rekaman tersebut akan menempati alamat *x.* Tetapi karena sekarang alamat *x* sudah kosong, maka program tidak akan meneruskan pencariannya ke alamat-alamat yang lain.

Salah satu cara untuk mengatasi persoalan diatas adalah dengan memberi tanda khusus pada alamat-alamat yang isi sesungguhnya sudah dihapus. Dengan demikian prgram akan meneruskan pencarian jika program membaca alamat yang diberi tandai "dihapus". Tetapi persoalan lain bisa timbul, yaitu jika hampir semua alamat diisi dengan tanda "dihapus". Dengan cara ini, maka pencarian bisa menjadi pencarian berurutan.

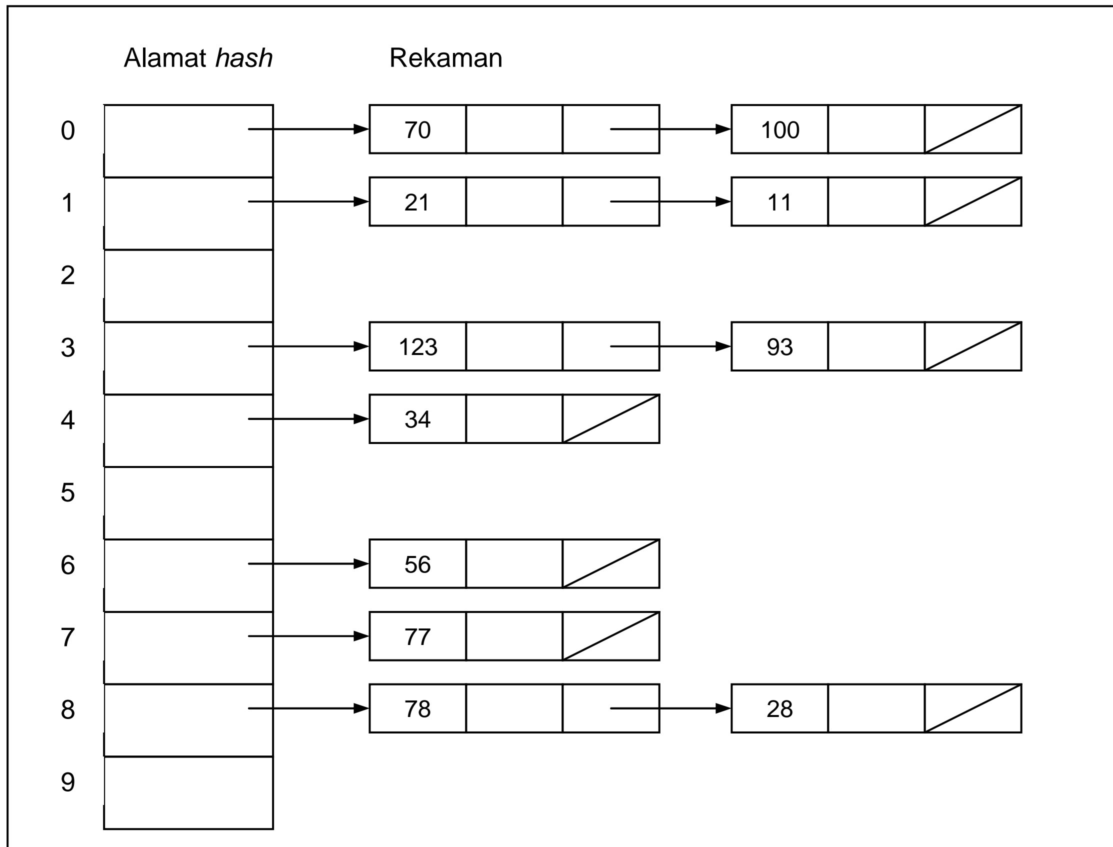
10.2.2 Penggandengan

Penggandengan (*chaining*) merupakan metode lain yang digunakan untuk mengatasi kemungkinan adanya tabrakan alamat *hash*. Metode ini pada prinsipnya memanfaatkan senarai berantai (yang juga bisa diimplementasikan menggunakan larik) yang dipasang pada setiap alamat *hash* yang diketahui. Dengan demikian, kika kita melihat pada sebuah alamat *hash* lengkap dan senarai yang menyimpan rekaman-rekaman yang mempunyai alamat *hash* yang sama, maka kita akan melihat adanya sebuah senarai berantai tunggal berkepala denga kepalanya adalah alamat *hash*.

Sebagai contoh, jika kita mempunyai rekaman-rekaman yang kunci rekamannya bisa dituliskan sebagai

34 56 123 78 93 70 100 21 11 77 28

dan fungsi hash yang dipilih adalah *k* mod 10. Dengan demikian, alamat *hash* akan terdiri dari sepuluh buah alamat yang bernomor 0 samapi 9. Gambar 10.2 menunjukkan alamat *hash* lengkap dengan senarai berantainya untuk menyimpan rekaman-rekaman diatas.



Gambar 10.2 Contoh Penggandengan Alamat *hash*

Dengan memperhatikan Gambar 10.2, kita bisa menyusun struktur data untuk menyajikan metode penggandengan dengan menggunakan link list, misalnya

```
Struct Rekaman {
   int Kunci;
   char Info;
   Rekaman *Berikut;
};
Rekaman Hash[10];
```

Program 10.1 Deklarasi Metode Penggandengan dengan Link List

10.3 Kesimpulan

1. Buatlah sebuah program untuk menyelesaikan proses mapping pada nomor telepon yang ada di Telkom dengan menggunakan metode Hashing. Data yang

ada berupa struktur yang terdiri dari no telpon, nama, alamat pelanggan. Program memberikan pilihan berupa menampilkan data, menambah data, menghapus data.

2. Buatlah sebuah program untuk menyelesaikan proses mapping pada sistem jaringan komputer yang ada di PENS dengan menggunakan metode Hashing. Data yang ada berupa struktur yang terdiri dari no IP, nama komputer, letak komputer. Program memberikan pilihan berupa menampilkan data, menambah data, menghapus data.

10.4 Latihan

1. Buatlah sebuah program untuk menyelesaikan proses mapping pada nomor telepon yang ada di Telkom dengan menggunakan metode Hashing. Data yang ada berupa struktur yang terdiri dari no telpon, nama, alamat pelanggan. Program memberikan pilihan berupa menampilkan data, menambah data, menghapus data.
2. Buatlah sebuah program untuk menyelesaikan proses mapping pada sistem jaringan komputer yang ada di PENS dengan menggunakan metode Hashing. Data yang ada berupa struktur yang terdiri dari no IP, nama komputer, letak komputer. Program memberikan pilihan berupa menampilkan data, menambah data, menghapus data.