BINUS UNIVERSITY ONLINE LEARNING

People
Innovation
Excellence

LECTURE NOTES

Data Structures

Week ke - 5

Hashing

LEARNING OUTCOMES

LO2: Illustrate any learned data structures and its usage in application

OUTLINE MATERI :

- Hashing Concept
- Hash Table
- Hash Function
- Collision
- Application of Hashing

ISI MATERI

1. *Hashing Concept*

Binary search dan *binary search tree* adalah algoritma yang efisien untuk mencari elemen. *Binary search* membutuhkan waktu O(n), tetapi bagaimana jika kita ingin melakukan operasi pencarian dalam waktu O(1)? Dengan kata lain, apakah ada cara untuk mencari *array* dalam waktu yang konstan, terlepas dari ukurannya?

Ada dua solusi untuk masalah ini. Mari kita ambil contoh untuk menjelaskan solusi pertama. Di perusahaan kecil dengan 100 karyawan, setiap karyawan diberi Emp_ID dalam kisaran 0–99. Untuk menyimpan record dalam sebuah *array*, Emp_ID setiap karyawan bertindak sebagai indeks ke dalam *array* dimana record karyawan akan disimpan seperti yang ditunjukkan pada Gambar 1.

Key	Array of Employees' Records
Key 0 → [0]	Employee record with Emp_ID 0
Key 1 → [1]	Employee record with Emp_ID 1
Key 2 → [2]	Employee record with Emp_ID 2
.....	.....
.....	.....
Key 98 → [98]	Employee record with Emp_ID 98
Key 99 → [99]	Employee record with Emp_ID 99

Gambar 1. *Records of employees*

Dalam hal ini, kita bisa langsung mengakses record pegawai manapun, begitu kita mengetahui Emp_ID-nya, karena indeks *array* sama dengan nomor Emp_ID. Namun secara praktis, implementasi ini hampir tidak mungkin dilakukan.

Mari kita asumsikan bahwa perusahaan yang sama menggunakan Emp_ID lima digit sebagai *key* utama. Dalam hal ini, nilai *key* akan berkisar dari 00000 hingga 99999. Jika ingin menggunakan teknik yang sama seperti di atas, kita memerlukan *array* berukuran

100.000, di mana hanya 100 elemen yang akan digunakan. Hal ini diilustrasikan pada Gambar 2.

Key	Array of Employees' Records
Key 00000 ⟶ [0]	Employee record with Emp_ID 00000
.....	.....
Key n ⟶ [n]	Employee record with Emp_ID n
.....	.....
Key 99998 ⟶ [99998]	Employee record with Emp_ID 99998
Key 99999 ⟶ [99999]	Employee record with Emp_ID 99999

Gambar 2 *Records of employees with a five-digit Emp_ID*

Tidak praktis membuang begitu banyak ruang penyimpanan hanya untuk memastikan bahwa setiap catatan karyawan berada di lokasi yang unik dan dapat diprediksi.

Apakah kita menggunakan *key* utama dua digit (Emp_ID) atau *key* lima digit, hanya ada 100 karyawan di perusahaan. Jadi, kita hanya akan menggunakan 100 lokasi dalam *array*. Oleh karena itu, untuk menjaga ukuran *array* ke ukuran yang sebenarnya akan kita gunakan (100 elemen), opsi bagus lainnya adalah menggunakan hanya dua digit terakhir dari *key* untuk mengidentifikasi setiap karyawan. Misalnya, pegawai dengan Emp_ID 79439 akan disimpan dalam elemen *array* dengan indeks 39. Demikian pula, pegawai dengan Emp_ID 12345 akan menyimpan catatannya dalam *array* di lokasi ke-45.

Dalam solusi kedua, elemen tidak disimpan sesuai dengan nilai *key*. Jadi dalam hal ini, kita memerlukan cara untuk mengubah nomor *key* lima digit menjadi indeks *array* dua digit. Kita membutuhkan sebuah fungsi yang akan melakukan transformasi. Dalam hal ini, kita akan menggunakan istilah *hash table* untuk *array* dan fungsi yang akan melakukan transformasi akan disebut fungsi *hash*.

2. *Hash Tables*

Tabel *hash* adalah struktur data di mana *key* dipetakan ke posisi *array* oleh fungsi *hash*. Dalam contoh yang dibahas di sini kita akan menggunakan fungsi *hash* yang mengekstrak dua digit terakhir dari *key*. Oleh karena itu, kami memetakan *key* ke lokasi

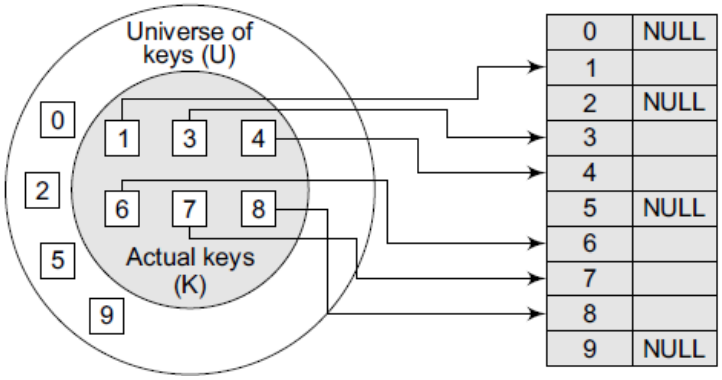
array atau indeks *array*. Nilai yang disimpan dalam tabel *hash* dapat dicari dalam waktu O(1) dengan menggunakan fungsi *hash* yang menghasilkan alamat dari *key* (dengan menghasilkan indeks *array* tempat nilai disimpan).

Gambar 3 menunjukkan korespondensi langsung antara *key* dan indeks *array*. Konsep ini berguna ketika total semesta *key* kecil dan ketika sebagian besar *key* benar-benar digunakan dari seluruh rangkaian *key*. Ini setara dengan contoh pertama, di mana ada 100 *key* untuk 100 karyawan.

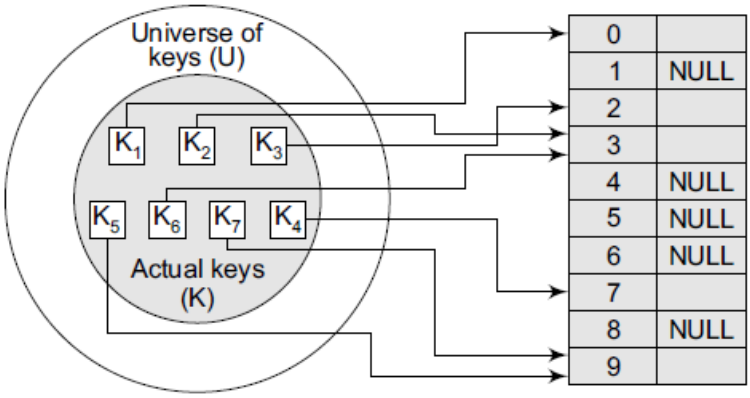
Namun, ketika set K *key* yang benar-benar digunakan lebih kecil dari semesta *key* (U), tabel *hash* mengkonsumsi lebih sedikit ruang penyimpanan. Persyaratan penyimpanan untuk tabel *hash* adalah O(k), di mana k adalah jumlah *key* yang benar-benar digunakan.

Dalam tabel *hash*, elemen dengan *key* k disimpan pada indeks h(k) dan bukan k. Artinya fungsi *hash* h digunakan untuk menghitung indeks dimana elemen dengan *key* k akan disimpan. Proses pemetaan *key* ke lokasi yang sesuai (atau indeks) dalam tabel *hash* disebut *hashing*.

Gambar 4 menunjukkan tabel *hash* di mana setiap *key* dari set K dipetakan ke lokasi yang dihasilkan dengan menggunakan fungsi *hash*. Perhatikan bahwa tombol k2 dan k6 menunjuk ke lokasi memori yang sama. Ini dikenal sebagai *collision*. Artinya, ketika dua atau lebih *key* dipetakan ke lokasi memori yang sama, *collision* dikatakan terjadi. Demikian pula, *key* k5 dan k7 juga *collision*. Tujuan utama menggunakan fungsi *hash* adalah untuk mengurangi rentang indeks *array* yang harus ditangani. Jadi, alih-alih memiliki nilai U, kita hanya membutuhkan nilai K, sehingga mengurangi jumlah ruang penyimpanan yang dibutuhkan.



Gambar 3. *Direct relationship between key and index in the array*



Gambar 4. *Relationship between keys and hash table index*

3. *Hash Function*

Fungsi *hash* adalah rumus matematika yang, ketika diterapkan pada *key*, menghasilkan bilangan bulat yang dapat digunakan sebagai indeks untuk *key* dalam tabel *hash*. Tujuan utama dari fungsi *hash* adalah bahwa elemen harus relatif, acak, dan terdistribusi secara merata. Ini menghasilkan satu set bilangan bulat unik dalam beberapa rentang yang sesuai untuk mengurangi jumlah *collision*. Dalam praktiknya, tidak ada fungsi *hash* yang menghilangkan *collision* sepenuhnya. Fungsi *hash* yang baik hanya dapat meminimalkan jumlah *collision* dengan menyebarkan elemen secara seragam di seluruh *array*.

Di bagian ini, kita akan membahas fungsi *hash* populer yang membantu meminimalkan *collision*. Namun sebelum itu, mari kita lihat dulu properti dari fungsi *hash* yang baik.

Properti Fungsi *Hash* yang Baik

Low cost. Biaya pelaksanaan fungsi *hash* harus kecil, sehingga menggunakan teknik *hashing* menjadi lebih baik daripada pendekatan lain. Misalnya, jika algoritma pencarian biner dapat mencari elemen dari tabel yang diurutkan dari n item dengan perbandingan *key* $\log_2$ n, maka fungsi *hash* harus lebih murah daripada melakukan perbandingan *key* $\log_2$ n.

Determinism Prosedur *hash* harus deterministik. Ini berarti bahwa nilai *hash* yang sama harus dihasilkan untuk nilai input yang diberikan. Namun, kriteria ini mengecualikan fungsi *hash* yang bergantung pada parameter variabel eksternal (seperti waktu) dan pada alamat memori objek yang di-*hash* (karena alamat objek dapat berubah selama pemrosesan).

Uniformity Fungsi *hash* yang baik harus memetakan *key* secara merata pada rentang outputnya. Ini berarti bahwa probabilitas menghasilkan setiap nilai *hash* dalam rentang output kira-kira harus sama. Sifat keseragaman juga meminimalkan jumlah tumbukan.

4. *Different Hash Functions*

Pada bagian ini, kita akan membahas fungsi *hash* yang menggunakan tombol numerik. Namun, mungkin ada kasus dalam aplikasi dunia nyata di mana kita dapat memiliki *key* alfanumerik daripada *key* numerik sederhana. Dalam kasus seperti itu, nilai ASCII karakter dapat digunakan untuk mengubahnya menjadi *key* numerik yang setara. Setelah transformasi ini selesai, salah satu fungsi *hash* yang diberikan di bawah ini dapat diterapkan untuk menghasilkan nilai *hash*.

4.1. Division Method

Ini adalah metode *hashing* integer x yang paling sederhana. Metode ini membagi x dengan M dan kemudian menggunakan sisa yang diperoleh. Dalam hal ini, fungsi *hash* dapat diberikan sebagai berikut

```
h(x) = x mod M
```

Division method cukup baik untuk hampir semua nilai M dan karena hanya memerlukan satu operasi pembagian, metode ini bekerja sangat cepat. Namun, perhatian ekstra harus diberikan untuk memilih nilai yang sesuai untuk M.

Contoh 1. Calculate the *hash* values of keys 1234 and 5462.
Solution Setting `M = 97`, *hash* values can be calculated as:

```
h(1234) = 1234 % 97 = 70
h(5642) = 5642 % 97 = 16
```

4.2. Multiplication Method

Langkah-langkah yang dilakukan dalam metode perkalian adalah sebagai berikut:

Langkah 1: Pilih konstanta A sehingga 0 < A < 1.

Langkah 2: Kalikan *key* k dengan A.

Langkah 3: Ekstrak bagian pecahan kA.

Langkah 4: Kalikan hasil Langkah 3 dengan ukuran tabel *hash* (m).

Oleh karena itu, fungsi *hash* dapat diberikan sebagai:

```
h(k) = ⌊m (kA mod 1) ⌋
```

di mana (kA mod 1) memberikan bagian pecahan dari kA dan m adalah jumlah total indeks dalam tabel *hash*.

Keuntungan terbesar dari metode ini adalah ia bekerja secara praktis dengan nilai A apa pun. Meskipun algoritma bekerja lebih baik dengan beberapa nilai, pilihan optimal bergantung pada karakteristik data yang di-*hash*. Knuth telah menyarankan bahwa pilihan terbaik dari A adalah

```
(sqrt5 – 1) /2 = 0.6180339887
```

Contoh 2 *Given a hash table of size 1000, map the key 12345 to an appropriate location* in the *hash* table.

Solution We will use A = 0.618033, m = 1000, and k = 12345

$$h(12345) = \lfloor 1000\ (12345 \times 0.618033 \bmod 1) \rfloor$$
$$= \lfloor 1000\ (7629.617385 \bmod 1) \rfloor$$
$$= \lfloor 1000\ (0.617385) \rfloor$$
$$= \lfloor 617.385 \rfloor$$
$$= 617$$

4.3. Mid-Square Method

Metode *mid-square* adalah fungsi *hash* yang baik yang bekerja dalam dua langkah:

Langkah 1: Kuadratkan nilai *key*. Yaitu, cari k2.

Langkah 2: Ekstrak r digit tengah dari hasil yang diperoleh pada Langkah 1.

Algoritma bekerja dengan baik karena sebagian besar atau semua digit nilai *key* berkontribusi pada hasil. Ini karena semua digit dalam nilai *key* asli berkontribusi untuk menghasilkan digit tengah dari nilai kuadrat. Oleh karena itu, hasilnya tidak didominasi oleh distribusi digit terbawah atau digit teratas dari nilai *key* asli.

Dalam metode kuadrat tengah, r digit yang sama harus dipilih dari semua *key*. Oleh karena itu, fungsi *hash* dapat diberikan sebagai:

$$h(k) = s$$

dimana s diperoleh dengan memilih r digit dari k^2.

Example 15.3 Calculate the *hash* value for keys 1234 and 5642 using the mid-square method.
The *hash* table has 100 memory locations.
Solution Note that the *hash* table has 100 memory locations whose indices vary from 0 to 99. This means that only two digits are needed to map the key to a location in the *hash* table, so r = 2.
When k = 1234, k_2 = 1522756, h (1234) = 27
When k = 5642, k_2 = 31832164, h (5642) = 21
Observe that the 3rd and 4th digits starting from the right are chosen

4.4. Folding Method

Metode melipat bekerja dalam dua langkah berikut:

Langkah 1: Bagilah nilai *key* menjadi beberapa bagian. yaitu, bagi k menjadi bagian-bagian k1, k2, ..., kn, di mana setiap bagian memiliki jumlah digit yang sama kecuali bagian terakhir yang mungkin memiliki angka lebih kecil dari bagian lainnya.

Langkah 2: Tambahkan bagian individu. Artinya, dapatkan jumlah k1 + k2 + ... + kn. Nilai *hash* dihasilkan dengan mengabaikan carry terakhir, jika ada.

Perhatikan bahwa jumlah digit di setiap bagian *key* akan bervariasi tergantung pada ukuran tabel *hash*. Misalnya, jika tabel *hash* memiliki ukuran 1000, maka ada 1000 lokasi di tabel *hash*. Untuk mengatasi 1000 lokasi ini, kita membutuhkan setidaknya tiga digit; oleh karena itu, setiap bagian dari *key* harus memiliki tiga digit kecuali bagian terakhir yang mungkin memiliki digit yang lebih kecil.

Contoh 4 Diberikan tabel *hash* dari 100 lokasi, hitung nilai *hash* menggunakan metode lipat untuk *key* 5678, 321, dan 34567.

Solution

Karena ada 100 lokasi memori yang akan dialamatkan, kami akan memecah *key* menjadi beberapa bagian di mana setiap bagian (kecuali yang terakhir) akan berisi dua digit. Nilai *hash* dapat diperoleh seperti yang ditunjukkan di bawah ini:

key	5678	321	34567
Parts	56 and 78	32 and 1	34, 56 and 7
Sum	134	33	97
Hash value	34 (ignore the last carry)	33	97

5. *Collisions*

Seperti dibahas sebelumnya dalam bab ini, *collision* terjadi ketika fungsi *hash* memetakan dua *key* yang berbeda ke lokasi yang sama. Jelas, dua catatan tidak dapat disimpan di lokasi yang sama. Oleh karena itu, diterapkan suatu metode yang digunakan untuk menyelesaikan masalah tumbukan, yang disebut juga dengan teknik resolusi tumbukan. Dua metode yang paling populer untuk menyelesaikan tabrakan adalah:

1) *Linear Probing*

2) *Chaining*

Pada bagian ini, kita akan membahas kedua teknik ini secara rinci.

5.1. Linear Probing

Mencari tempat kosong selanjutnya dan kemudian tempatkan *string* di tempat tesebut.

Contoh:

Jika terdapat *hash table* untuk strik berikut ini:

define, float, exp, char, atan, ceil, floor, acos.

Perhatikan bahwa kata “ceil” disimpan di h[6], “floor” disimpan pada h[7] dan “acos” disimpan pada h[1].

Ketika ingin menyimpan kata “ceil” dalam table telah terdapat kata – kata “char” pada h[2], sehingga dicari tempat kosong selanjutnya yaitu h[6] hal serupa terjadi Ketika memasukkan kata “floor” dan “acos”,

h[ ]	Value
0	Atan
1	**acos**
2	**char**
3	define
4	exp
5	float
6	**ceil**
7	**floor**
…	

5.2. Chaining

Menyimpan *string* dalam sebuah slot sebagai *chained list* (*linked list*). Jadi jika terjadi sebuah *collision*, data akan dimasukkan ke dalam *chain*.

h[ ]	Value
0	atan → acos
1	NULL
2	char → ceil
3	define
4	exp
5	float → floor
6	NULL
7	NULL
…	

6. Application of Hashing

Hashing digunakan untuk pengindeksan basis data. Beberapa sistem manajemen database menyimpan file terpisah yang dikenal sebagai file indeks. Ketika data harus diambil dari sebuah file, informasi *key* pertama-tama dicari dalam file indeks yang sesuai yang merujuk pada lokasi record yang tepat dari data dalam file database. Informasi *key* dalam file indeks ini sering disimpan sebagai nilai *hash*.

Dalam banyak sistem *database*, *hashing* file dan direktori digunakan dalam sistem file berkinerja tinggi. Sistem tersebut menggunakan dua teknik yang saling melengkapi untuk meningkatkan kinerja akses file. Sementara salah satu teknik ini adalah *caching* yang menyimpan informasi dalam memori, yang lain adalah *hashing* yang membuat pencarian lokasi file dalam memori lebih cepat daripada kebanyakan metode lainnya.

Teknik *hashing* digunakan untuk mengimplementasikan tabel simbol compiler di C++. Kompiler menggunakan tabel simbol untuk menyimpan catatan semua simbol yang ditentukan pengguna dalam program C++. *Hashing* memfasilitasi kompiler untuk dengan cepat mencari nama variabel dan atribut lain yang terkait dengan simbol. *Hashing* juga banyak digunakan untuk mesin pencari Internet.

SIM/Kartu Asuransi Seperti contoh CD, bahkan nomor SIM atau nomor kartu asuransi dibuat menggunakan *hashing* dari item data yang tidak pernah berubah: tanggal kelahiran, nama, dll.

Grafik Dalam grafik, masalah utama adalah penyimpanan objek dalam adegan atau tampilan. Untuk ini, kami mengatur data kami dengan *hashing*. *Hashing* dapat digunakan untuk membuat grid dengan ukuran yang sesuai, grid vertikal-horizontal biasa. (Perhatikan bahwa kisi tidak lain adalah *array* 2D, dan ada korespondensi satu-ke-satu saat kita berpindah dari *array* 2D ke *array* 1D.)

Jadi, kami menyimpan grid sebagai *array* 1D seperti yang kami lakukan dalam kasus matriks sparse. Semua poin yang jatuh dalam satu sel akan disimpan di tempat yang sama. Jika sel berisi tiga titik, maka ketiga titik ini akan disimpan dalam entri yang sama. Pemetaan dari sel grid ke lokasi memori dilakukan dengan menggunakan fungsi *hash*. Keuntungan utama dari metode penyimpanan ini adalah eksekusi operasi yang cepat seperti pencarian tetangga terdekat.

SIMPULAN

Hashing

Hashing adalah sebuah teknik pengubahan sebuah *string* menjadi sebuah nilai atau *key* yang lebih sederhana yang dapat mempresentasikan *string* yang dimaksud.

Hashing digunakan untuk mendata dan mengambil item di dalam *database* karena pencarian akan lebih mudah dilakukan dengan menggunakan *hash*ed *key* yang lebih pendek daripada mencari dengan menggunakan nilai yang sebenarnya. *Hashing* juga biasanya digunakan pada algoritma enkripsi

Hash Function

Berikut ini adalah beberapa *hash function* yang biasanya digunakan:

1. *Division method*
2. *Multiplication Method*
3. *Mid-Square Method*
4. *Digit rearrangement method*

Collision

Tedapat 2 cara utama untuk mengatasi *Collisions*

1. *Linear probing*
2. *Chaining*

DAFTAR PUSTAKA

Thareja, R. (2014). *Data Structures Using C* (second). Oxford University Press. https://doi.org/10.1136/adc.67.4.533

Data Structure and Algorithms - Hash Table https://www.tutorialspoint.com/data_structures_algorithms/hash_data_structure.htm