Struktur Hash table dengan Implemetasi Lookup Menggunakan Algoritma Binary Search

Andresta Ramadhan[1], Ibrahim Arief[2], Andree Datta Adwitya[3]

Laboratorium Ilmu dan Rekayasa Komputasi
Departemen Teknik Informatika, Institut Teknologi Bandung
Jl. Ganesha 10, Bandung

E-mail: if13030@students.if.itb.ac.id[1], if13038@students.if.itb.ac.id[2],
if13062@students.if.itb.ac.id[3]

Abstrak

Data yang disimpan di dalam memori komputer perlu ditempatkan dalam suatu cara sedemikian sehingga pencariannya dapat dilakukan dengan cepat. Struktur *hash table* merupakan struktur yang dapat mempersingkat waktu pencarian *record* (*lookup*) dalam sebuah tabel dengan cara mengasosiasikan setiap *record* dengan sebuah kunci yang unik dan menggunakan fungsi *hash* (*hash function*) untuk menentukan lokasi *record* tersebut pada tabel. Namun, karena fungsi *hash* bukanlah fungsi satu-ke-satu, dimungkinkan terjadinya bentrokan (*collision*) dalam penempatan suatu data *record*. Untuk mengatasi bentrokan diperlukan sebuah kebijakan resolusi bentrokan (*collision resolution policy*). Umumnya kebijakan ini menempatkan *record* pada posisi berikutnya dari tabel jika posisi yang ditujunya sudah terisi. Untuk kasus terbaik, kompleksitas algoritma yang dibutuhkan adalah konstan O(1), tapi untuk kasus terburuk dimana data yang hendak dicari tidak ditemukan, kompleksitasnya dapat mencapai O(n). Hal ini dikarenakan *hash table* menggunakan algoritma sequential search untuk mencari *record* pada tabel jika *record* yang hendak dicari tidak ditemukan pada posisi yang dihitung berdasar *hash function*. Dengan mengubah struktur *hash table* dengan cara mengelompokan hasil *record* dengan hasil *hash function* yang bertumpukan ke dalam sebuah struktur list berkait yang terurut berdasar kuncinya, pencarian *record* dapat menggunakan algoritma *binary* search yang lebih efisien.

***Kata kunci**: hash table, binary search, collision, collision resolution policy, hash function, lookup, hash, divide and conquer.*

1. Pendahuluan

Hash table adalah sebuah struktur data yang terdiri atas sebuah tabel dan fungsi yang bertujuan untuk memetakan nilai kunci yang unik untuk setiap *record* menjadi angka (*hash*) lokasi *record* tersebut dalam sebuah tabel. [1]

Keunggulan dari struktur *hash table* ini adalah waktu aksesnya yang cukup cepat, jika *record* yang dicari langsung berada pada angka *hash* lokasi penyimpanannya. Akan tetapi pada kenyataannya sering sekali ditemukan *hash table* yang *record-record*nya mempunyai angka *hash* yang sama (bertabrakan).

Karena pemetaan *hash function* yang digunakan bukanlah pemetaan satu-satu, (antara dua *record* yang tidak sama dapat dibangkitkan angka *hash* yang sama) maka dapat terjadi bentrokan (*collision*) dalam penempatan suatu data *record*. Untuk mengatasi hal ini, maka perlu diterapkan kebijakan resolusi bentrokan (*collision resolution policy*) untuk menentukan lokasi *record* dalam tabel. Umumnya kebijakan resolusi bentrokan adalah dengan mencari lokasi tabel yang masih kosong pada lokasi setelah lokasi yang berbentrokan.

2. *Hash table*

2.1 Struktur *Hash table*

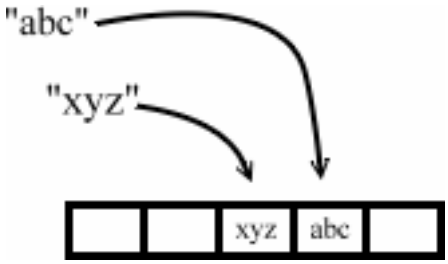
Hash table menggunakan struktur data *array* asosiatif yang mengasosiasikan *record* dengan sebuah *field* kunci unik berupa bilangan (*hash*) yang merupakan representasi dari *record* tersebut.

Misalnya, terdapat data berupa string yang hendak disimpan dalam sebuah *hash table*. String tersebut direpresentasikan dalam sebuah *field* kunci k. Cara untuk mendapatkan *field* kunci ini sangatlah beragam, namun hasil akhirnya adalah sebuah bilangan *hash* yang digunakan untuk menentukan lokasi *record*. Bilangan *hash* ini dimasukan ke

dalam *hash* function dan menghasilkan indeks lokasi *record* dalam tabel. [1]

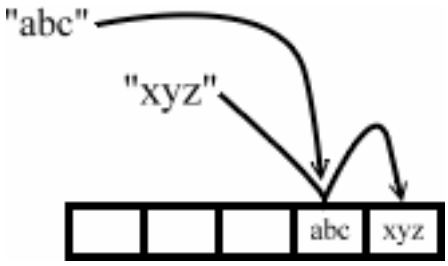
k(x) = fungsi pembangkit *field* kunci .... (1)
h(x) = *hash* function .... (2)

Contohnya, terdapat data berupa string “abc” dan “xyz” yang hendak disimpan dalam struktur *hash table*. Lokasi dari *record* pada tabel dapat dihitung dengan menggunakan h(k(“abc”)) dan h(k(“xyz”)).



Gambar 1. Penempatan *record* pada *hash table*

Jika hasil dari *hash* function menunjuk ke lokasi memori yang sudah terisi oleh sebuah *record*, maka dibutuhkan kebijakan resolusi bentrokan. Biasanya masalah ini diselesaikan dengan mencari lokasi *record* kosong berikutnya secara *incremental*.



Gambar 2. Resolusi bentrokan pada *hash table*

2.2 *Lookup* pada *Hash table*

2.2.1 Definisi *Lookup*

Salah satu keunggulan struktur *hash table* dibandingkan dengan struktur tabel biasa adalah kecepatannya dalam mencari data. Terminologi *lookup* mengacu pada proses yang bertujuan untuk mencari sebuah *record* pada sebuah tabel, dalam hal ini adalah *hash table*. [3]

2.2.2 Metode *Lookup*

Dengan menggunakan *hash* function, sebuah lokasi dari *record* yang dicari bisa diperkirakan. Jika lokasi yang tersebut berisi *record* yang dicari, maka pencarian berhasil. Inilah kasus terbaik dari pencarian pada *hash table*. Namun, jika *record* yang hendak dicari tidak ditemukan di lokasi yang diperkirakan, maka akan dicari lokasi berikutnya sesuai dengan kebijakan resolusi bentrokan. Pencarian akan berhenti jika *record* ditemukan, pencarian bertemu dengan tabel kosong, atau pencarian telah kembali ke lokasi semula.

3. *Binary Search*

3.1 Algoritma *Binary Search*

Binary Search adalah algoritma pencarian yang lebih efisien daripada algorima *Sequential Search*. Hal ini dikarenakan algoritma ini tidak perlu menjelajahi setiap elemen dari tabel. Kerugiannya adalah algoritma ini hanya bisa digunakan pada tabel yang elemennya sudah terurut baik menaik maupun menurun. [2]

Pada intinya, algoritma ini menggunakan prinsip *divide and conquer*, dimana sebuah masalah atau tujuan diselesaikan dengan cara mempartisi masalah menjadi bagian yang lebih kecil. Algoritma ini membagi sebuah tabel menjadi dua dan memproses satu bagian dari tabel itu saja.

Algoritma ini bekerja dengan cara memilih *record* dengan indeks tengah dari tabel dan membandingkannya dengan *record* yang hendak dicari. Jika *record* tersebut lebih rendah atau lebih tinggi, maka tabel tersebut dibagi dua dan bagian tabel yang bersesuaian akan diproses kembali secara rekursif.

3.2 Kompleksitas Waktu

Kompleksitas waktu terbaik algoritma ini adalah 1, sedangkan kompleksitas waktu terburuknya $^{2}\log n$. Kompleksitas waktu terburuk ini dicapai pada kasus dimana *record* tidak ditemukan dalam tabel. Pada kasus ini, algoritma melakukan pembagian tabel hingga ukuran tabel sebesar 1 elemen. Jumlah langkah tersebut adalah $^{2}\log n$. Karena pada setiap langkah dilakukan perbandingan yang merupakan basis dari penghitungan kompleksitas waktu algoritma pencarian, maka kompleksitas waktu terburuk algoritma ini adalah $^{2}\log n$. [2]

3.3 Keunggulan *Binary Search*

Keunggulan utama dari algoritma *binary search* adalah kompleksitas algoritmanya yang lebih kecil daripada kompleksitas algoritma *sequential search*. Hal ini menyebabkan waktu yang dibutuhkan algoritma *binary search* dalam mencari sebuah *record* dalam sebuah tabel lebih kecil daripada waktu yang dibutuhkan algoritma *sequential search*.

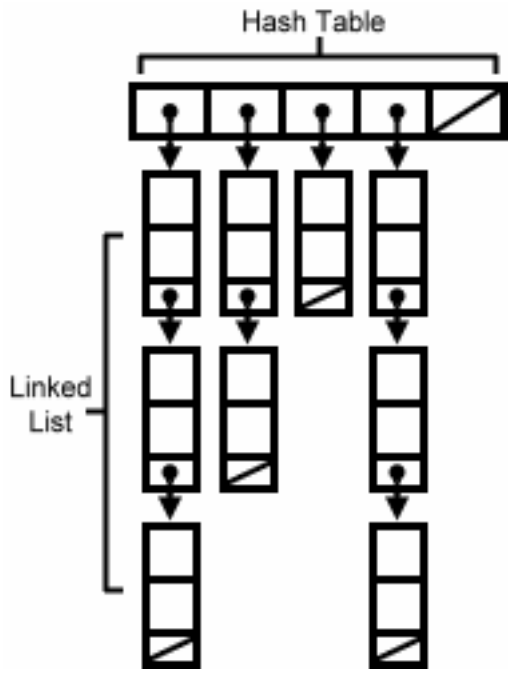
4. *Hash table* dengan Implementasi Lookup *Binary Search*

4.1 Struktur *Hash table* yang Mengimplementasikan *Binary Search*

Hash table dengan implementasi *binary search* menggunakan struktur yang hampir sama dengan

hash table standar. Struktur penyimpanan data tetap berupa sebuah tabel dengan ukuran terbatas. Perbedaanya dengan struktur *hash table* biasa adalah elemen dari tabel tersebut dan cara pengorganisasian *record* dalam tabel.

Pada *hash table* standar, elemen dari setiap tabel adalah *record* itu sendiri. Sedangkan pada *hash table* yang mengimplementasikan *binary search*, elemen dari setiap tabel adalah pointer ke sebuah struktur *linked list* yang menyimpan *record*.



Gambar 3. Struktur *hash table* dengan *linked list*

Pada struktur *linked* list, selain disimpan data *record* yang dimasukan, disimpan juga *field* kunci yang dihasilkan dari rumus (1). Metode penyimpanan pada struktur ini berupa terurut menaik berdasar *field* kunci yang disimpan.

Metode penyimpanan *record* pada struktur baru ini agak berbeda dengan metode penyimpanan pada struktur *hash table* standar, namun metode ini masih menggunakan keunggulan dari *hash table* yaitu pembangkitan kunci dan penggunaan *hash function* untuk menentukan lokasi *record* dalam tabel.

Contohnya, terdapat data berupa string “abc” dan “xyz” yang hendak disimpan dalam struktur *hash table* ini. Lokasi pada tabel yang bersesuaian dengan *record* dapat dihitung dengan menggunakan h(k(“abc”)) dan h(k(“xyz”)). Perbedaannya dengan struktur *hash* tabel standar adalah lokasi ini tidak berfungsi sebagai penyimpan *record*, namun sebagai penunjuk ke struktur list berkait yang menyimpan *record*.

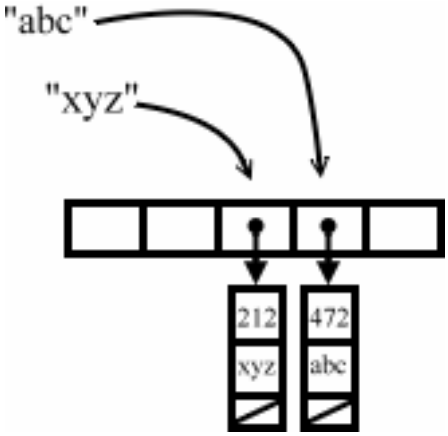
Misalnya jika dengan menggunakan rumus (1) didapat :

k(“abc”) = 472
k(“xyz”) = 212

Dan jika dengan menggunakan rumus (2) didapat :

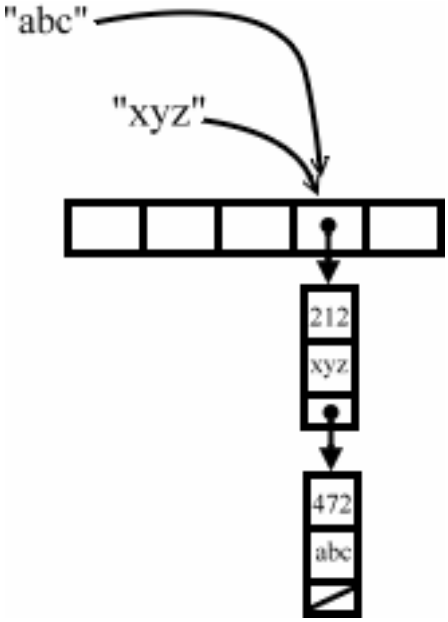
h(472) = 3
h(212) = 2

Maka penempatan *record* pada struktur *hash table* dengan *linked list* adalah sebagai berikut :



Gambar 4. Penempatan *record* pada struktur *hash table* dengan *linked list*

Jika terjadi bentrokan antara nilai *hash* dari dua buah *record*, maka kebijakan resolusi bentrokan yang diambil adalah dengan menambahkan *record* tersebut pada struktur linked list yang berlokasi di lokasi tabel yang sama secara terurut.



Gambar 5. Resolusi bentrokan pada struktur *hash table* dengan *linked list*

4.2 Implementasi *Lookup* dengan *Binary Search*

Dalam proses pencarian *record* dalam *hash table* yang mengimplementasikan *binary search*, proses akan tetap mencari angka *hash* dari *record* yang akan dicari terlebih dahulu. Runtutan aktifitas pencarian berikutnya adalah mencari lokasi penyimpanan yang sama dengan angka *hash*, lalu mengecek *list* yang ditunjuk oleh lokasi tersebut. Proses selanjutnya adalah mencari *record* didalam *list* yang mempunyai nilai kunci sama dengan nilai kunci dari *record* yang dicari. Oleh karena elemen-elemen *list* sudah terurut menaik berdasarkan kuncinya, maka proses pencarian dilakukan dengan metode *binary search*. Metode ini akan mencari elemen tengah dari *list* dan membandingkannya dengan *record* yang akan dicari, jika tidak sama maka proses akan melakukan *binary search* secara rekursif untuk kumpulan elemen yang ada di sebelah kiri atau kanan elemen tengah dari *list* tersebut.

4.3 Perbandingan dengan *Hash table* Standar

Dalam *hash table* standar waktu yang dibutuhkan untuk menyimpan dan mencari *record* di dalam *array* adalah sama. Dalam kasus terbaik proses *lookup* akan menghasilkan kompleksitas O(1) dan pada kasus normal akan menghasilkan kompleksitas O(n). Jarang sekali ditemukan *hash table* yang proses *lookup*nya menghasilkan kompleksitas waktu terbaik dikarenakan penggunaan *hash table* yang biasanya untuk jumlah *record* yang banyak. Berbeda dengan *hash table* yang menimplementasikan *binary search*. *Hash table* ini mempercepat waktu aksesnya dalam menyimpan dan mencari *record* dalam *hash table* untuk jumlah *record* yang semakin besar jika dibandingkan dengan *hash table* standar. Untuk penggunaan normal, *hash table* ini cukup mempersingkat waktu akses. *Hash table* ini pun tidak memiliki batasan jumlah *record* sebanyak lebar lokasi penyimpanan, karena *record-record* akan disimpan di dalam *list* yang berada di luar *array hash table*, tidak seperti *hash table* standar yang menyimpan *record*nya dalam *array* tersebut.

4.4 Keunggulan

Keunggulan dari implementasi *hash table* menggunakan *binary search* tidak akan terlihat dengan jelas. Akan tetapi, pada kasus dengan jumlah *record* yang sangat besar, lebar lokasi penyimpanan yang kecil, dan banyaknya *record* yang mempunyai angka *hash* yang sama, pengimplementasian *binary search* dapat mengefisienkan waktu dengan perbedaan waktu akses yang lebih cepat daripada menggunakan metode standar *hash table*.

5. Kesimpulan

Struktur *hash table* ini akan menghemat pencarian karena proses *lookup*-nya menggunakan algoritma *binary search* (pencarian biner) yang terbukti lebih cepat dari pada *sequential search* (pencarian beruntun).

Perubahan struktur membawa keuntungan tambahan dengan tidak terbatasnya jumlah record maksimum yang dapat disimpan oleh *hash table*. Selain itu partisi tabel menjadi bagian-bagian yang diproses secara terpisah dan menggunakan *hash function* sebagai selektor partisi merupakan penerapan lain dari strategi *divide and conquer*.

Dapat disimpulkan bahwa struktur *hash table* yang memanfaatkan implementasi *lookup* dengan *binary search* menggabungkan keunggulan dari struktur *hash table* dan algoritma *binary search*.

Daftar Pustaka

[1] Knuth, D. *The Art of Computer Programming*, Volume 3 : Sorting and Searching, Chapter 6.4. Addison Wesley, 1973.

[2] Munir, R, *Matematika Diskrit*, Vol 2, Penerbit Informatika Bandung, 2003.

[3] Wikipedia (http://www.wikipedia.org/) entry on *Hash table* (http://en.wikipedia.org/wiki/*Hash*_table)