BAB 8 & 9

HASH TABLE

8.1. TUJUAN PEMBELAJARAN

Tujuan dari pembelajaran pada pertemuan graph, diantaranya mahasiswa diharapkan mampu:

1. Membuat dan mendeklarasikan struktur algoritma *hash table*
2. Menerapkan dan mengimplementasikan struktur algoritma *hash table*

8.2. DASAR TEORI

Hash Table adalah sebuah struktur data yang terdiri atas sebuah tabel dan fungsi yang bertujuan untuk memetakan nilai kunci yang unik untuk setiap record (baris) menjadi angka (*hash*) lokasi record tersebut dalam sebuah tabel. Biasanya struktur data ini digunakan dalam penyimpanan data sementara. Tujuan dari penggunaan *hash table* adalah untuk mempercepat pencarian kembali dari banyak data yang disimpan. Berdasarkan prinsip kerjanyam, *hash table* menggunakan suatu teknik penyimpanan sehingga waktu yang dibutuhkan untuk penambahan data (*insertions*), penghapusan data (*deletions*), dan pencarian data (*searching*), prinsip kerjanya dapat dikatakan relatif sama dibanding struktur data atau algoritma yang lain.

Pada *hash table* terdapat beberapa operasi yang biasa digunakan, diantaranya insert (diberikan sebuah key dan nilai, insert nilai dalam tabel), find (diberikan sebuah key, temukan nilai yang berhubungan dengan key), remove (diberikan sebuah key,temukan nilai yang berhubungan dengan key kemudian hapus nilai tersebut), dan getIterator (mengambalikan iterator,yang memeriksa nilai satu demi satu)

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-recordnya mempunyai angka hash yang sama (bertabrakan). Karena pemetaan *hash function* yang digunakan bukanlah pemetaan satusatu, (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.

Memori penyimpanan utama pada *hash table* berbentuk array dengan tambahan algoritma untuk mempercepat pemrosesan data. Pada intinya *hash table* merupakan penyimpanan data menggunakan *key value* yang didapat dari nilai data itu sendiri. Dengan *key value* tersebut didapat *hash value*. Jadi *hash function* merupakan suatu fungsi sederhana untuk mendapatkan hash value dari key value suatu data. Terdapat beberapa hal yang perlu diperhatikan untuk membuat *hash function,* diantaranya ukuran array/table size(m), key value/nilai yang didapat dari data(k), *hash value/hash index*/indeks yang dituju(h).

Perhatikan contoh di bawah ini mengenai penggunaan *hash table* dengan *hash function* sederhana yaitu memodulus key value dengan ukuran array:

H(k) = k MOD m

Misal kita memiliki array dengan ukuran 13, maka hash function : h = k (mod 13). Dengan hash function tersebut di dapat :

k	H
7	7
13	0
25	12
27	1
39	0

Perhatikan range dari h untuk sembarang nilai k.

Maka data 7 akan disimpan pada index 7, data 13 akan disimpan pada index 0, dst..

Untuk mencari kembali suatu data, maka kita hanya perlu menggunakan hash function yang sama sehingga mendapatkan hash index yang sama pula.

Misal : mencari data 25 → h = 25 (mod 13) = 12

Namun pada penerapannya, seperti contoh di atas terdapat tabrakan (*collision*) pada k = 13 dan k = 39. *Collision* berarti ada lebih dari satu data yang memiliki hash index yang sama, padahal seperti yang kita ketahui, satu alamat / satu index array hanya dapat menyimpan satu data saja.

8.2.1. Collision

Collision dalam *hash table* merupakan situasi dimana dua atau lebih elemen memiliki nilai hash yang sama, sehingga elemen-elemen tersebut harus disimpan dalam lokasi yang sama dalam tabel hash. Dalam pemrograman C++, kolision ini dapat ditangani dengan beberapa teknik, seperti *Separate Chaining*, *Linear Probing, Double Hashing.*

Untuk meminimalkan *collision* dapat mengunakan hash function yang dapat mencapai seluruh indeks/alamat. Dalam contoh di atas gunakan m untuk me-modulo k. Perhatikan bila kita menggunakan angka m untuk me-modulo k maka pada indeks yang lebih besar dari dan sama dengan m di hash table tidak akan pernah terisi (memori yang terpakai semakin kecil), kemungkinan terjadi collision juga semakin besar. Berikut adalah beberapa cara yang dapat dilakukan untuk meminimalkan *collision,* diantaranya :

1. Melakukan *closed hashing (open addressing)*

   Cara kerja *closed hashing* untuk menyelesaikan *collision*, yaitu dengan menggunakan memori yang masih ada tanpa menggunakan memori diluar array yang digunakan. Kelemahan dari *closed hashing* adalah ukuran array yang disediakan harus lebih besar dari jumlah data. Selain itu dibutuhkan memori yang lebih besar untuk meminimalkan *collision*. Cara kerjanya, *closed hashing* akan mencari alamat lain apabila alamat yang akan dituju sudah terisi oleh data. Terdapat 3 cara untuk mencari alamat lain tersebut, meliputi :

   a. Linear probing/metode pembagian

      Teknik ini mencari lokasi berikutnya dalam tabel hash saat *collision* terjadi. Prinsip kerjanya dapat diilustrasikan dengan suatu kondisi apabila telah terisi, linear probing mencari alamat lain dengan bergeser 1 indeks dari alamat sebelumnya hingga ditemukan alamat yang belum terisi data, dengan rumus

(h+1) mod m

      Contoh

      Disediakan nomor mahasiswa terdiri dari 5 digit. Disediakan larik (l) dari 100 buah alamat yang masing - masing alamat terdiri dari 2 karakter : 00 … 99, nomor mahasiswa yang diketahui 10347, 87492, 34212, 88688. Untuk menentukan alamat dari ke empat nomor mhs tsb kita pilih bil prima yang mendekati 99 yaitu 97 maka m = 97. Masukan hit di atas ke dalam rumus **H(k) mod m + 1,** maka dapat dituliskan seperti berikut ini :

      H(K) 1 = 10347 MOD 97 + 1 -> 66

H(K) 2 = 87492 MOD 97 + 1 -> 96

H(K) 3 = 34212 MOD 97 + 1 -> 69

H(K) 4 = 88688 MOD 97 + 1 -> 31

H(K) 5 = 88588 MOD 97 + 1 -> 28

H(K) 6 = 87578 MOD 97 + 1 -> 85

b. Quadratic Probing / Metode Midsquare / Nilai Tengah

Penanganannya hampir sama dengan metode linear, hanya lompatannya tidak satu-satu, tetapi *quadratic. Quadratic Probing* mencari alamat baru untuk ditempati dengan proses perhitungan kuadratik yang lebih kompleks. Tidak ada formula baku pada *quadratic probing* ini, sehingga formula yang akan digunakan dapat ditentukan sendiri.

Contoh formula quadratic probing untuk mencari alamat baru:

h,(h+i2)mod m,(h-i2)mod m, … ,(h+((m-1)/2)2)mod m, (h-((m-1)/2)2)mod m dengan i = 1,2,3,4, … , ((m-1)/2)

Maksud formula di atas adalah jika alamat h telah terisi, maka alamat lain yang digunakan adalah (h+1)mod m, jika telah terisi gunakan alamat (h-1)mod m, jika telah terisi gunakan alamat (h+4)mod m, jika telah terisi gunakan alamat (h-4)mod m, dan seterusnya.

Jadi jika m=23,maka nilai maksimal i adalah : ((23-1)/2)=11.

Contoh Metode Ini , Kunci Yang Di Ketahui Dikuadratkan.

K	10347	87492
K2	01 07 06 04 09	76 54 85 00 64
H(k)	**06**	**85**

c. Double Hashing / Metode Penjumlahan Digit

Teknik ini menggunakan dua fungsi hash dan mencari lokasi berikutnya dengan mempertimbangkan kedua nilai hash. Sesuai dengan namanya, alamat baru untuk menyimpan data yang belum dapat masuk ke dalam table diperoleh dengan menggunakan hash function lagi. *Hash function*

kedua yang digunakan setelah alamat yang dihasilkan oleh *hash function* awal telah terisi tentu saja berbeda dengan hash function awal itu sendiri.

Contoh

10347 = 01 + 03 + 47 = 51

87492 = 08 + 74 + 92 = 174 = 01 + 74 = 75

Kesimpulannya, pada metode ini kunci yang diketahui bisa dipecah menjadi beberapa kelompok. Pemecahan dan penjumlahan terus dilakukan jika keseluruhan kelompok yang ada masih lebih besar dari banyaknya alamat yang akan dipakai.

2. Open hashing (Separate Chaining)

   Teknik ini menyimpan elemen dengan nilai hash yang sama dalam daftar yang terpisah. Pada dasarnya separate chaining membuat tabel yang digunakan untuk proses hashing menjadi sebuah *array of pointer* yang masing-masing pointernya diikuti oleh sebuah *Linked list*, dengan *chain* (mata rantai) 1 terletak pada *array of pointer*, sedangkan chain 2 dan seterusnya berhubungan dengan chain 1 secara memanjang. Kelemahan dari open hashing adalah bila data menumpuk pada satu/sedikit indeks sehingga terjadi Linked list yang panjang.

8.3. PERCOBAAN

1. Buatlah workspace menggunakan Replit.
2. Buatlah project baru HashTable yang berisi file C++ source untuk algoritma dijkstra
3. Cobalah untuk masing-masing percobaan di bawah

Percobaan 1 : Implementasi hashing table

```
/**
 * Program Hashtable dengan OpenAddressing
 * Penampung data menggunakan array
 * Tiga Implementasi menggunakan Linear Probing, Quadric
Probing, Double Hashing
 * Populasi data menggunakan fungsi random
 */

#include <iostream>
#include <stdio.h>
#include <math.h>
#include <ctime>

using namespace std;

// asumsikan kita menggunakan array dengan batas 1024
```

```
int storage[1024];
int i = 0;
int hdt_boundary;

void tambah_linear_probing(int n)
{
    bool inserted = false;
    int hash;
    i = 0;

    while( (!inserted) &&  (i < hdt_boundary) )
    {
        hash = (n % hdt_boundary) + i;

        if(storage[hash] == 0)
        {
            storage[hash] = n;
            inserted = true;
        }
        else
        {
            ++i;
            cout << "Terjadi tabrakan di " << hash << endl;
        }
    }

    if(i == 0)
    {
        cout << "Langsung" << endl;
    }

    cout << "Isi hash[" << hash << "] dengan " << n << endl;
```

```
    cout <<
"============================================================
======" << endl;
}

void tambah_quadric_probing(int n)
{
    bool inserted = false;
    int hash;
    i = 0;

    while( (!inserted) &&  (i < hdt_boundary) )
    {
        hash = (n % hdt_boundary) + (i*i);

        if(storage[hash] == 0)
        {
            storage[hash] = n;
            inserted = true;
        }
        else
        {
            ++i;
            cout << "Terjadi tabrakan di " << hash << endl;
        }
    }

    if(i == 0)
    {
        cout << "Langsung" << endl;
    }

    cout << "Isi hash[" << hash << "] dengan " << n << endl;
```

```
    cout <<
"============================================================
======" << endl;

}

// fungsi pembantu double_hashing
int prima_atas(int n)
{
    if (n % 2 == 0)
        n = n + 1;
    else
        n = n + 2;

    bool ketemu = false;

    while (!ketemu){
        bool prima = true;
        if (n % 2 == 0)
            prima = false;
        else {
            int i = 3;
            while (prima == true && i <=sqrt(n))
            {
                if(n % i == 0)
                    prima = false;

                i = i + 2;
            }
        }
        if (prima)
            ketemu = true;
        else
            n = n + 2;
```

```
    }
    return n;
}

int prima_bawah()
{
    int n = hdt_boundary;
    if (n % 2 == 0)
        n = n - 1;
    else
        n = n - 2;

    bool ketemu = false;
    while (!ketemu){
        bool prima = true;
        if (n % 2 == 0)
            prima=false;
        else {
            int i=3;
            while (prima == true && i <= sqrt(n)){
                if(n % i == 0)
                    prima = false;
                i = i + 2;
            }
        }
        if (prima)
            ketemu = true;
        else
            n = n-2;
    }
    return n;
}

void tambah_double_hashing(int n)
```

```
{
    //cout <<
"============================================================
======" << endl;
    //cout << "doube hashing..." << endl;
    int hash;
    int hash2 = prima_bawah();
    bool inserted = false;
    i = 0;
    while( (!inserted) &&  (i < hdt_boundary) )
    {
        hash = ( (n%hdt_boundary) + i*((n) % hash2 + 1) ) %
hdt_boundary;

        if(storage[hash] == 0)
        {
            storage[hash] = n;
            inserted = true;
        }
        else
        {
            ++i;
            cout << "Terjadi tabrakan di " << hash << endl;
        }
    }

    if(i == 0)
    {
        cout << "Langsung" << endl;
    }

    cout << "Isi hash[" << hash << "] dengan " << n << endl;
```

```
    cout <<
"============================================================
======" << endl;
}

void cetak(int n)
{
    cout << endl;
    cout << "Output program : " << endl;
    for (int a = 0; a < n; ++a)
    {
        cout << "hash[" << a << "] = " << storage[a] << endl;
    }

}

int getRandomNumberFromRange(int min, int max)
{
    return min + (rand() % (max - min ) );
}

int main()
{
    int n,random_number;
    char pilihan;
    string cara;

    // harus panggil fungsi ini agar generate random datanya
tidak sama terus.
    srand((unsigned)time(0));

    cout << "Masukan jumlah data : ";
    cin >> n;
```

```
    cout << "Kami menggunakan metode open addressing ada 3
cara, pilih salah satu dengan memilih a,b, atau c : " <<
endl;
    cout << "a. linear probing" << endl;
    cout << "b. quadric probing" << endl;
    cout << "c. double hashing" << endl;
    cout << "Pilih a,b, atau c ? " << endl;
    cin >> pilihan;
    cout << endl << endl << "Proses pemasukan data ke
hashtable " << endl;

    // tentukan batas |T|
    hdt_boundary = prima_atas(n);

    for (int a = 0; a < n; ++a)
    {
        random_number = getRandomNumberFromRange(0,n);

        // switch mau pakai cara open addressing apa?
        switch(pilihan)
        {
            case 'a':
                tambah_linear_probing(random_number);
                cara = "Linear Probing" ;
                break;
            case 'b':
                tambah_quadric_probing(random_number);
                cara = "Quadric Probing";
                break;
            case 'c':
                tambah_double_hashing(random_number);
                cara = "Double hashing";
                break;
        }
```

```
    }

    /**
     * Dibawah adalah operasi untuk cetak
     */
    cetak(n);
    cout << "Diatas adalah hasil output Hashtable dengan cara
\"" << cara << "\""<< endl;
    switch(pilihan)
    {
        case 'a':
            cout << "h(k) = (k mod "<< hdt_boundary <<") + i"
<< endl;
            break;
        case 'b':
            cout << "h(k) = (k mod "<< hdt_boundary <<") +
i*i" << endl;
            break;
        case 'c':
            cout << "h1(k) = k mod "<< hdt_boundary << endl;
            cout << "h2(k) = ( h1(k) + i*(k mod " <<
prima_bawah() << " + 1) ) mod " << hdt_boundary << endl;
            break;
    }
}
```

OUTPUT

[OPSI Metode "a"]

Masukan jumlah data : 6

Kami menggunakan metode open addressing ada 3 cara, pilih salah satu dengan memilih a,b, atau c :

a. linear probing

b. quadric probing

c. double hashing

Pilih a,b, atau c ?

a

Proses pemasukan data ke hashtable

Langsung

Isi hash[1] dengan 1

==========

=

Terjadi tabrakan di 1

Isi hash[2] dengan 1

==========

=

Terjadi tabrakan di 1

Terjadi tabrakan di 2

Isi hash[3] dengan 1

==========

=

Langsung

Isi hash[5] dengan 5

==========

=

Terjadi tabrakan di 1

Terjadi tabrakan di 2

Terjadi tabrakan di 3

Isi hash[4] dengan 1

==========

=

Terjadi tabrakan di 1

Terjadi tabrakan di 2

Terjadi tabrakan di 3

Terjadi tabrakan di 4

Terjadi tabrakan di 5

Isi hash[6] dengan 1

==========
=

Output program :

hash[0] = 0

hash[1] = 1

hash[2] = 1

hash[3] = 1

hash[4] = 1

hash[5] = 5

Diatas adalah hasil output Hashtable dengan cara "Linear Probing"

h(k) = (k mod 7) + i

[OPSI Metode "b"]

Masukan jumlah data : 6

Kami menggunakan metode open addressing ada 3 cara, pilih salah satu dengan memilih a,b, atau c :

a. linear probing

b. quadric probing

c. double hashing

Pilih a,b, atau c ?

b

Proses pemasukan data ke hashtable

Langsung

Isi hash[2] dengan 2

==========
=

Langsung

Isi hash[4] dengan 4

==========
=

Langsung

Isi hash[0] dengan 0

==========

=

Langsung

Isi hash[5] dengan 5

==========

=

Langsung

Isi hash[1] dengan 1

==========

=

Terjadi tabrakan di 1

Terjadi tabrakan di 2

Terjadi tabrakan di 5

Isi hash[10] dengan 1

==========

=

Output program :

hash[0] = 0

hash[1] = 1

hash[2] = 2

hash[3] = 0

hash[4] = 4

hash[5] = 5

Diatas adalah hasil output Hashtable dengan cara "Quadric Probing"

h(k) = (k mod 7) + i*i

[OPSI Metode "c"]

Masukan jumlah data : 4

Kami menggunakan metode open addressing ada 3 cara, pilih salah satu dengan memilih a,b, atau c :

a. linear probing

b. quadric probing

c. double hashing

Pilih a,b, atau c ?

c

Proses pemasukan data ke hashtable

Langsung

Isi hash[3] dengan 3

==========

=

Terjadi tabrakan di 3

Isi hash[4] dengan 3

==========

=

Langsung

Isi hash[2] dengan 2

==========

=

Langsung

Isi hash[0] dengan 0

==========

=

Output program :

hash[0] = 0

hash[1] = 0

hash[2] = 2

hash[3] = 3

Diatas adalah hasil output Hashtable dengan cara "Double hashing"

h1(k) = k mod 5

h2(k) = (h1(k) + i*(k mod 3 + 1)) mod 5

8.4. TUGAS DAN LATIHAN

Buatlah sebuah program sederhana dari hash table yang menggunakan metode double hashing!