

PRAKTIKUM STRUKTUR DATA DAN ALGORITME

MATERI : HASH TABLE

I. TUJUAN

1. Mahasiswa mampu menjelaskan definisi dan konsep dari Hash Code
2. Mahasiswa mampu menerapkan Hash Code kedalam pemrograman

II. Dasar Teori

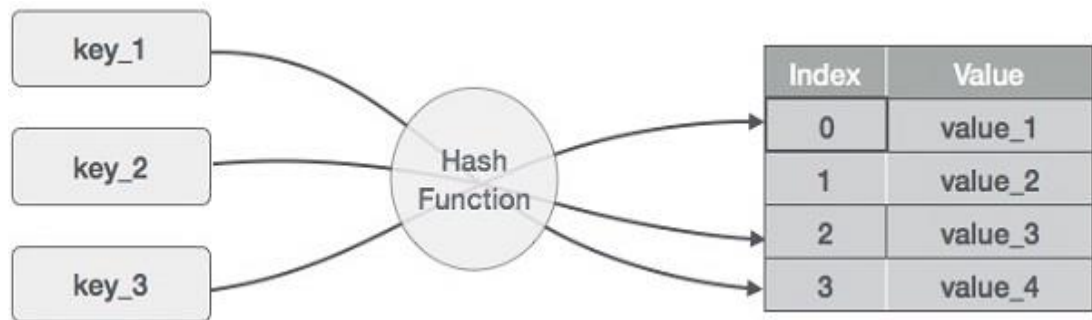
1. Pengertian Hash Table

Tabel Hash adalah struktur data yang digunakan untuk menyimpan pasangan kunci/nilai. Hash table biasanya terdiri dari dua komponen utama: array (atau vector) dan fungsi hash. Dengan menggunakan fungsi hash yang baik hashing dapat berjalan dengan baik. **Hashing** adalah teknik untuk mengubah rentang nilai kunci menjadi rentang indeks array.

Array menyimpan data dalam slot-slot yang disebut bucket. Setiap bucket dapat menampung satu atau beberapa item data. Fungsi hash digunakan untuk menghasilkan nilai unik dari setiap item data, yang digunakan sebagai indeks array. Dengan cara ini, hash table memungkinkan pencarian data dalam waktu yang konstan ($O(1)$) dalam kasus terbaik

Sistem hash table bekerja dengan cara mengambil input kunci dan memetakannya ke nilai indeks array menggunakan fungsi hash. Kemudian, data disimpan pada posisi indeks array yang dihasilkan oleh fungsi hash. Ketika data perlu dicari, input kunci dijadikan sebagai parameter untuk fungsi hash, dan posisi indeks array yang dihasilkan digunakan untuk mencari data. Dalam kasus hash collision, di mana dua atau lebih data memiliki nilai hash

yang sama, hash table menyimpan data tersebut dalam slot yang sama dengan Teknik yang disebut chaining.



Gambar 1. Struktur Data Hash

2. Fungsi Hash Table

Fungsi hash membuat pemetaan antara kunci dan nilai, hal ini dilakukan melalui penggunaan rumus matematika yang dikenal sebagai fungsi hash. Hasil dari fungsi hash disebut sebagai nilai hash atau hash. Nilai hash adalah representasi dari string karakter asli tetapi biasanya lebih kecil dari aslinya.

Contoh: Pertimbangkan sebuah array sebagai Peta di mana kuncinya adalah indeks dan nilainya adalah value pada indeks itu. Jadi untuk array A jika kita memiliki indeks i yang akan diperlakukan sebagai kunci maka kita dapat menemukan nilainya hanya dengan mencari value pada $A[i]$.

Tipe fungsi hash :

- Division Method.
- Mid Square Method.
- Folding Method.
- Multiplication Method.

Untuk mencapai mekanisme hashing yang baik, penting untuk memiliki fungsi hash yang baik dengan persyaratan dasar sebagai berikut:

- Dapat dihitung secara efisien.
- Harus mendistribusikan kunci secara seragam (Setiap posisi tabel memiliki kemungkinan yang sama untuk masing-masing.)
- Harus meminimalkan tabrakan.
- Harus memiliki faktor muatan rendah (jumlah item dalam tabel dibagi dengan ukuran tabel).

3. Operasi Hash Table

1. Insertion:

Memasukkan data baru ke dalam hash table dengan memanggil fungsi hash untuk menentukan posisi bucket yang tepat, dan kemudian menambahkan data ke bucket tersebut.

2. Deletion:

Menghapus data dari hash table dengan mencari data menggunakan fungsi hash, dan kemudian menghapusnya dari bucket yang sesuai.

3. Searching:

Mencari data dalam hash table dengan memasukkan input kunci ke fungsi hash untuk menentukan posisi bucket, dan kemudian mencari data di dalam bucket yang sesuai.

4. Update:

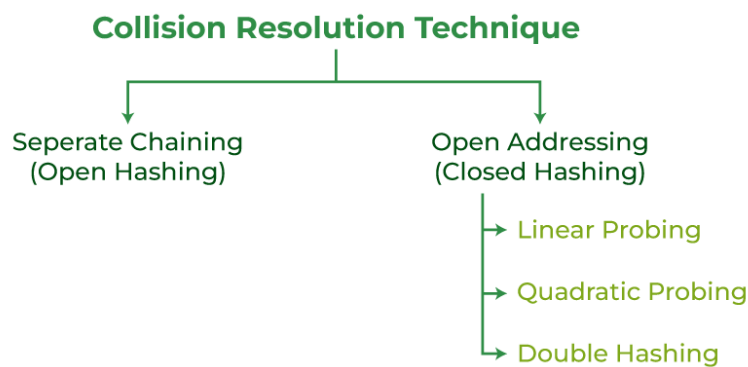
Memperbarui data dalam hash table dengan mencari data menggunakan fungsi hash, dan kemudian memperbarui data yang ditemukan.

5. Traversal:

Melalui seluruh hash table untuk memproses semua data yang ada dalam tabel

4. Collision Resolution

Keterbatasan tabel hash adalah jika dua angka dimasukkan ke dalam fungsi hash menghasilkan nilai yang sama. Hal ini disebut dengan collision. Ada dua teknik untuk menyelesaikan masalah ini diantaranya :



Gambar 2. Teknik Collision

A. Open Hashing (Chaining)

Metode chaining mengatasi collision dengan cara menyimpan semua item data dengan nilai indeks yang sama ke dalam sebuah linked list. Setiap node pada linked list merepresentasikan satu item data. Ketika ada pencarian atau penambahan item data, pencarian atau penambahan dilakukan pada linked list yang sesuai dengan indeks yang telah dihitung dari kunci yang dihash. Ketika linked list memiliki banyak node, pencarian atau penambahan item data menjadi lambat, karena harus mencari di seluruh linked list. Namun, chaining dapat mengatasi jumlah item data yang besar dengan efektif, karena keterbatasan array dihindari.

B. Closed Hashing

1. Linear Probing

Pada saat terjadi collision, maka akan mencari posisi yang kosong di bawah tempat terjadinya collision, jika masih penuh terus ke bawah, hingga ketemu tempat yang kosong. Jika tidak ada tempat yang kosong berarti HashTable sudah penuh.

2. Quadratic Probing

Penanganannya hampir sama dengan metode linear, hanya lompatannya tidak satu-satu, tetapi quadratic (12, 22, 32, 42, ...)

3. Double Hashing

Pada saat terjadi collision, terdapat fungsi hash yang kedua untuk menentukan posisinya kembali.

III. Guided

```
#include <iostream>
using namespace std;
const int MAX_SIZE = 10;

// Fungsi hash sederhana
int hash_func(int key) {
    return key % MAX_SIZE;
}

// Struktur data untuk setiap node
struct Node {
    int key;
    int value;
    Node* next;
    Node(int key, int value) : key(key), value(value), next(nullptr) {}
};

// Class hash table
class HashTable {
private:
    Node** table;
public:
    HashTable() {
        table = new Node*[MAX_SIZE]();
    }
    ~HashTable() {
        for (int i = 0; i < MAX_SIZE; i++) {
            Node* current = table[i];
            while (current != nullptr) {
                Node* temp = current;
                current = current->next;
                delete temp;
            }
        }
        delete[] table;
    }

    // Insertion
```

```

void insert(int key, int value) {
    int index = hash_func(key);
    Node* current = table[index];
    while (current != nullptr) {
        if (current->key == key) {
            current->value = value;
            return;
        }
        current = current->next;
    }
    Node* node = new Node(key, value);
    node->next = table[index];
    table[index] = node;
}

// Searching
int get(int key) {
    int index = hash_func(key);
    Node* current = table[index];
    while (current != nullptr) {
        if (current->key == key) {
            return current->value;
        }
        current = current->next;
    }
    return -1;
}

// Deletion
void remove(int key) {
    int index = hash_func(key);
    Node* current = table[index];
    Node* prev = nullptr;
    while (current != nullptr) {
        if (current->key == key) {
            if (prev == nullptr) {
                table[index] = current->next;
            } else {
                prev->next = current->next;
            }
        }
        prev = current;
        current = current->next;
    }
}

```

```

        }
        delete current;
        return;
    }
    prev = current;
    current = current->next;
}
}

// Traversal
void traverse() {
    for (int i = 0; i < MAX_SIZE; i++) {
        Node* current = table[i];
        while (current != nullptr) {
            cout << current->key << ": " << current->value << endl;
            current = current->next;
        }
    }
}

};

int main() {
    HashTable ht;
    // Insertion
    ht.insert(1, 10);
    ht.insert(2, 20);
    ht.insert(3, 30);

    // Searching
    cout << "Get key 1: " << ht.get(1) << endl;
    cout << "Get key 4: " << ht.get(4) << endl;

    // Deletion
    ht.remove(4);

    // Traversal
    ht.traverse();

    return 0;
}

```

Kode di atas menggunakan array dinamis “table” untuk menyimpan bucket dalam hash table. Setiap bucket diwakili oleh sebuah linked list dengan setiap node merepresentasikan satu item data. Fungsi hash sederhana hanya menggunakan modulus untuk memetakan setiap input kunci ke nilai indeks array.

GUIDED 2

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;
const int TABLE_SIZE = 11;

string name;
string phone_number;

class HashNode {
public:
    string name;
    string phone_number;

    HashNode(string name, string phone_number) {
        this->name = name;
        this->phone_number = phone_number;
    }
};

class HashMap {
private:
    vector<HashNode*> table[TABLE_SIZE];

public:
    int hashFunc(string key) {
        int hash_val = 0;

        for (char c : key) {
            hash_val += c;
        }
    }
};
```



```

        return hash_val % TABLE_SIZE;
    }

    void insert(string name, string phone_number) {
        int hash_val = hashFunc(name);

        for (auto node : table[hash_val]) {
            if (node->name == name) {
                node->phone_number = phone_number;
                return;
            }
        }

        table[hash_val].push_back(new HashNode(name, phone_number));
    }

    void remove(string name) {
        int hash_val = hashFunc(name);

        for (auto it = table[hash_val].begin(); it !=
table[hash_val].end(); it++) {
            if ((*it)->name == name) {
                table[hash_val].erase(it);
                return;
            }
        }
    }

    string searchByName(string name) {
        int hash_val = hashFunc(name);

        for (auto node : table[hash_val]) {
            if (node->name == name) {
                return node->phone_number;
            }
        }

        return "";
    }

```

```

    }

    void print() {
        for (int i = 0; i < TABLE_SIZE; i++) {
            cout << i << ": ";
            for (auto pair : table[i]) {
                if(pair != nullptr){
                    cout << "[" << pair->name << ", " << pair->
phone_number << "];"
                                }
                            }
                        cout << endl;
                    }
                }
            };

    int main() {
        HashMap employee_map;

        employee_map.insert("Mistah", "1234");
        employee_map.insert("Pastah", "5678");
        employee_map.insert("Ghana", "91011");

        cout << "Nomer Hp Mistah : " <<
employee_map.searchByName("Mistah") << endl;
        cout << "Phone Hp Pastah : " <<
employee_map.searchByName("Pastah") << endl;

        employee_map.remove("Mistah");

        cout << "Nomer Hp Mistah setelah dihapus : " <<
employee_map.searchByName("Mistah") << endl << endl;

        cout << "Hash Table : " << endl;
        employee_map.print();
        return 0;
    }
}

```

Pada program di atas, class HashNode merepresentasikan setiap node dalam hash table, yang terdiri dari nama dan nomor telepon karyawan. Class HashMap digunakan untuk mengimplementasikan struktur hash table dengan menggunakan vector yang menampung pointer ke HashNode.

Fungsi hashFunc digunakan untuk menghitung nilai hash dari nama karyawan yang diberikan, dan fungsi insert digunakan untuk menambahkan data baru ke dalam hash table. Fungsi remove digunakan untuk menghapus data dari hash table, dan fungsi searchByName digunakan untuk mencari nomor telepon dari karyawan dengan nama yang diberikan.

IV. Unguided

1. Implementasikan hash table untuk menyimpan data mahasiswa. Setiap mahasiswa memiliki NIM dan nilai. Implementasikan fungsi untuk menambahkan data baru, menghapus data, mencari data berdasarkan NIM, dan mencari data berdasarkan nilai. Dengan ketentuan :
 - a. Setiap mahasiswa memiliki NIM dan nilai.
 - b. Program memiliki tampilan pilihan menu berisi poin C.
 - c. Implementasikan fungsi untuk menambahkan data baru, menghapus data, mencari data berdasarkan NIM, dan mencari data berdasarkan rentang nilai (80 – 90).