LAPORAN FINAL PROJECT STRUKTUR DATA

Disusun Oleh :

Ahmad Loka Arziki (5025241044)

Reza Afzaal Faizullah Taqy (5025241051)

Ary Pasya Fernanda (5025241053)

Dosen Pengampu :

Dr. Dwi Sunaryono, S.Kom., M.Kom.

INSTITUT TEKNOLOGI SEPULUH NOPEMBER

2025

BAB I
i. Latar Belakang

Di dunia pendidikan, data mahasiswa merupakan salah satu jenis data yang sangat sering digunakan. Data seperti ID, nama, kota asal, tanggal lahir, dan email dibutuhkan dalam berbagai aktivitas, mulai dari keperluan administrasi hingga pendataan kegiatan kampus. Karena jumlah mahasiswa bisa sangat banyak, diperlukan cara yang efisien untuk mengelola data tersebut agar mudah diakses dan diproses.

Melalui proyek ini, kami mencoba membuat sebuah simulasi pengelolaan data mahasiswa dalam jumlah banyak, lalu membandingkan dua metode penyimpanan data, yaitu Hash Table dan B+ Tree. Kami ingin melihat bagaimana performa masing-masing metode dalam melakukan operasi seperti menambahkan data, mencari berdasarkan ID, menampilkan informasi, mengubah data, dan menghapus data. Khusus untuk B+ Tree, kami juga menguji pencarian berdasarkan rentang ID, yang tidak bisa dilakukan oleh Hash Table.

Data yang kami gunakan berasal dari tiga file CSV dengan jumlah data yang berbeda, yaitu 100, 500, dan 1000 mahasiswa. Dengan membandingkan hasilnya, kami bisa melihat bagaimana kedua metode tersebut bekerja dalam skala data yang berbeda-beda.

Melalui proyek ini, kami ingin memahami secara langsung bagaimana struktur data bisa mempengaruhi kecepatan dan efisiensi saat mengelola data dalam jumlah besar. Harapannya, hasil dari proyek ini bisa menjadi pengalaman belajar yang bermanfaat saat kami mengembangkan program atau aplikasi di masa depan.

ii. Tujuan

1. Mengembangkan program dalam bahasa C untuk mengelola data mahasiswa yang terdiri dari ID, nama, kota, tanggal lahir, dan email.
2. Mengimplementasikan dan membandingkan dua struktur data, yaitu Hash Table dan B+ Tree, dalam proses penyimpanan dan pengolahan data.
3. Menyediakan fitur dasar seperti penambahan, pencarian, penampilan data, pembaruan, dan penghapusan data pada kedua struktur.
4. Mengukur dan membandingkan performa kedua struktur berdasarkan waktu eksekusi dan kompleksitas saat menangani jumlah data yang berbeda.
5. Memberikan pemahaman praktis mengenai dampak pemilihan struktur data terhadap efisiensi pengelolaan data dalam skala besar.

BAB II
Hashmap

i. Struktur Data.

```
typedef struct data

{

    int id;

    char name[MAXDATA];

    char city[MAXDATA];

    char email[MAXDATA];

    char date[MAXDATE];

    struct data *next;

} data;
```

Penjelasan

Struktur Mahasiswa digunakan untuk menyimpan informasi mahasiswa. Struktur ini mencakup:

- id: Nomor identifikasi unik mahasiswa.
- name: Nama mahasiswa.
- city: Kota asal mahasiswa.
- date: Tanggal lahir mahasiswa.
- email: Alamat email mahasiswa.

Collision handling memakai linked-list chaining: setiap slot array menunjuk ke daftar berantai (next).

ii. Penjelasan Fungsi.

1. hash

```
unsigned int hash(int key)
{
    return key % TABLE_SIZE;
}
```

Penjelasan.

Menggunakan hasil dari modulo key terhadap TABLE_SIZE sebagai index hasil hashing.

2. createNode

```
data *createNode(int id, const char *name, const char *city, const char *email, const
{
    data *newNode = (data *)malloc(sizeof(data));
    newNode->id = id;
    strcpy(newNode->name, name);
    strcpy(newNode->city, city);
    strcpy(newNode->email, email);
    strcpy(newNode->date, date);
    newNode->next = NULL;
    return newNode;
}
```

Penjelasan.

Membuat data newNode sebagai tempat untuk memasukkan data baru kedalam hash_table dengan menggunakan **strcpy** untuk memasukkan data-data kedalam newNode dan mendeklarasikan next nya menjadi NULL sebagai tempat untuk menyimpan data baru nanti dengan index hasil hashing yang sama

3. id_ada

```c
int id_ada(int id)
{
    unsigned int index = hash(id);
    data *curr = hash_table[index];
    while (curr)
    {
        if (curr->id == id)
        {
            return 1;
        }
        curr = curr->next;
    }
    return 0;
}
```

Penjelasan.

Mengecek apakah id yang telah ada di hash tabel sama dengan id yang akan dimasukan dengan pengecekan curr-> id == id .

4. tambah_data

```c
void tambah_data()
{
    char idstr[100];
    int id;
    char name[MAXDATA], city[MAXDATA], email[MAXDATA], date[MAXDATE];

    printf("\nTambah Data Baru\n");

    printf("Masukkan ID : ");

    fgets(idstr, sizeof(idstr), stdin);

    id = atoi(idstr);

    if (id_ada(id))
    {
        printf("ID %d sudah ada. Gunakan ID lain.\n", id);
        return;
    }

    printf("Masukkan Nama: ");
    fgets(name, sizeof(name), stdin);
    name[strcspn(name, "\n")] = '\0';

    printf("Masukkan Kota: ");
    fgets(city, sizeof(city), stdin);
    city[strcspn(city, "\n")] = '\0';

    printf("Masukkan Email: ");
    fgets(email, sizeof(email), stdin);
    email[strcspn(email, "\n")] = '\0';

    printf("Masukkan Tanggal Lahir: ");
    fgets(date, sizeof(date), stdin);
    date[strcspn(date, "\n")] = '\0';

    unsigned int index = hash(id);
    data *newNode = createNode(id, name, city, email, date);
    newNode->next = hash_table[index];
    hash_table[index] = newNode;
    count++;

    printf("Data berhasil ditambahkan.\n");
}
```

Penjelasan.

Membaca input pengguna, memvalidasi duplikasi ID dengan **id_ada**, lalu membuat dan menyisipkan node baru ke posisi head rantai pada indeks hasil hashing, sekaligus mengincrement variabel count.

5. load_data

```c
void load_data(const char *filename)
{
    FILE *file = fopen(filename, "r");
    if (!file)
    {
        printf("File tidak ditemukan.\n");
        return;
    }
    char line[500];
    fgets(line, sizeof(line), file);

    int inserted = 0;
    while (fgets(line, sizeof(line), file))
    {
        char name[MAXDATA], city[MAXDATA], email[MAXDATA], date[MAXDATE];
        char id_str[20];
        int id;

        int fields = sscanf(line, "%[^,],%[^,],%[^,],%[^,],%[^\n]", name, id_str, city, date, email);
        if (fields != 5)
        {
            continue;
        }
        id = atoi(id_str);
        if (id_ada(id))
        {
            continue;
        }
        unsigned int index = hash(id);
        data *newNode = createNode(id, name, city, email, date);
        newNode->next = hash_table[index];
        hash_table[index] = newNode;

        count++;
        inserted++;
    }

    fclose(file);
    printf("%d data dimuat dari %s.\n", inserted, filename);
}
```

Penjelasan.

Membuka file CSV, melewati header, membaca tiap baris dengan **sscanf**, menolak baris tak valid atau id duplikat, lalu menambah node baru ke tabel serta mencatat jumlah data yang berhasil dimuat.

6. cari_data

```c
void cari_data()
{
    char input[20];
    printf("\nCari Data Berdasarkan ID \nMasukkan ID: ");
    fgets(input, sizeof(input), stdin);
    input[strcspn(input, "\n")] = 0;
    int id = atoi(input);

    unsigned int index = hash(id);
    int kompleks = 0;
    data *curr = hash_table[index];

    while (curr)
    {
        kompleks++;
        if (curr->id == id)
        {
            printf("\nData ditemukan (kompleksitas: %d)\n", kompleks);
            printf("ID              : %d\n", curr->id);
            printf("Nama            : %s\n", curr->name);
            printf("Kota            : %s\n", curr->city);
            printf("Email           : %s\n", curr->email);
            printf("Tanggal Lahir   : %s\n", curr->date);
            return;
        }
        curr = curr->next;
    }

    printf("Data dengan ID %d tidak ditemukan (kompleksitas: %d).\n", id, kompleks);
}
```

Penjelasan.

Menerima ID dari pengguna, menghitung indeks hash, lalu menelusuri rantai pada slot tersebut sambil menghitung jumlah langkah (kompleksitas) hingga data ditemukan atau habis.

7. update_data.

```c
void update_data()
{
    char input[20];
    printf("\nUpdate Data \nMasukkan ID: ");
    fgets(input, sizeof(input), stdin);
    input[strcspn(input, "\n")] = 0;
    int id = atoi(input);

    unsigned int index = hash(id);
    data *curr = hash_table[index];
    int kompleks = 0;

    while (curr)
    {
        kompleks++;
        if (curr->id == id)
        {
            printf("Data ditemukan (kompleksitas: %d).\nMasukkan data baru:\n", kompleks);

            printf("Nama baru: ");
            fgets(curr->name, MAXDATA, stdin);
            curr->name[strcspn(curr->name, "\n")] = 0;

            printf("Kota baru: ");
            fgets(curr->city, MAXDATA, stdin);
            curr->city[strcspn(curr->city, "\n")] = 0;

            printf("Email baru: ");
            fgets(curr->email, MAXDATA, stdin);
            curr->email[strcspn(curr->email, "\n")] = 0;

            printf("Tanggal lahir baru: ");
            fgets(curr->date, MAXDATE, stdin);
            curr->date[strcspn(curr->date, "\n")] = 0;

            printf("Data berhasil diperbarui.\n");
            return;
        }
        curr = curr->next;
    }

    printf("Data dengan ID %d tidak ditemukan (kompleksitas: %d).\n", id, kompleks);
}
```

Penjelasan.

Fungsi ini mencari data berdasarkan ID, lalu memperbarui isinya (nama, kota, email, tanggal lahir) jika ditemukan. Data baru langsung ditimpa ke node yang sesuai. Jika tidak ditemukan, ditampilkan pesan kegagalan beserta jumlah langkah pencarian (kompleksitas).

8. hapus_data.

```c
void hapus_data()
{
    char input[20];
    printf("\nHapus Data \nMasukkan ID: ");
    fgets(input, sizeof(input), stdin);
    input[strcspn(input, "\n")] = 0;
    int id = atoi(input);

    unsigned int index = hash(id);
    data *curr = hash_table[index];
    data *prev = NULL;
    int kompleks = 0;

    while (curr)
    {
        kompleks++;
        if (curr->id == id)
        {
            if (prev)
                prev->next = curr->next;
            else
                hash_table[index] = curr->next;

            free(curr);
            count--;

            printf("Data dengan ID %d berhasil dihapus (kompleksitas: %d).\n", id, kompleks);
            return;
        }
        prev = curr;
        curr = curr->next;
    }

    printf("Data dengan ID %d tidak ditemukan (kompleksitas: %d).\n", id, kompleks);
}
```

Penjelasan.

Menelusuri rantai pada indeks hash untuk menemukan node ber-ID sesuai. jika ketemu, menghubungkan ulang pointer next, membebaskan memori, dan melakukan decrement terhadap count.

9. tampilkan_data.

```
void tampilkan_data()
{
    for (int i = 0; i < TABLE_SIZE; i++)
    {
        data *curr = hash_table[i];
        while (curr)
        {
            printf("\n- ID              : %d\n", curr->id);
            printf("  Nama            : %s\n", curr->name);
            printf("  Kota            : %s\n", curr->city);
            printf("  Email           : %s\n", curr->email);
            printf("  Tanggal Lahir   : %s\n", curr->date);
            curr = curr->next;
        }
    }
    printf("\nSemua Data (Total: %d) \n", count);
}
```

Penjelasan.

Melintasi seluruh slot tabel dan setiap rantainya untuk mencetak semua data, lalu menampilkan total elemen yang tersimpan.

10. init.

```
void init()
{
    for (int i = 0; i < TABLE_SIZE; i++)
    {
        hash_table[i] = NULL;
    }
    count = 0;
}
```

Penjelasan.

Mengosongkan seluruh slot hash map dengan mengatur pointer ke NULL dan mereset penghitung count menjadi nol sebagai tahap inisialisasi awal program.

11. main

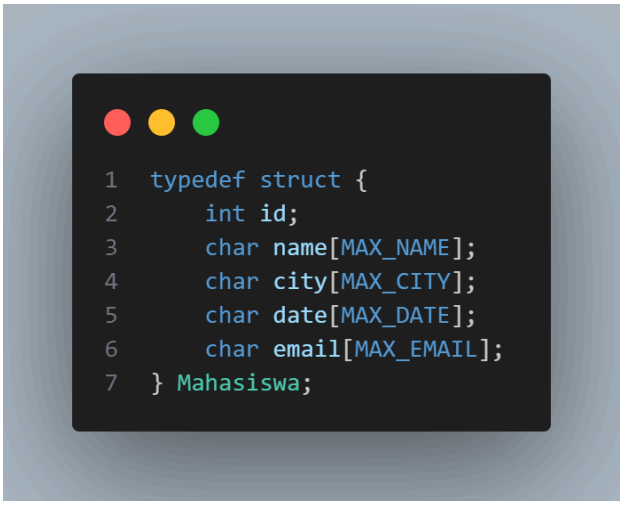
Penjelasan.

Menampilkan menu interaktif yang dapat dipilih oleh user.

B+ Tree

i. Penjelasan Struktur Data

1. Struct `Mahasiswa



```c
typedef struct {
    int id;
    char name[MAX_NAME];
    char city[MAX_CITY];
    char date[MAX_DATE];
    char email[MAX_EMAIL];
} Mahasiswa;
```

Penjelasan:

- Struct ini merepresentasikan data mahasiswa dengan field:
  - `id` (integer): ID unik mahasiswa.
  - `name` (string): Nama mahasiswa (maksimal `MAX_NAME` karakter).
  - `city` (string): Kota asal mahasiswa (maksimal `MAX_CITY` karakter).
  - `date` (string): Tanggal lahir (maksimal `MAX_DATE` karakter).
  - `email` (string): Alamat email (maksimal `MAX_EMAIL` karakter).
- Struct ini digunakan untuk menyimpan data yang akan dimasukkan ke dalam B+ Tree.

2. Struct `BPTreeNode` (Node B+ Tree)

```c
typedef struct BPTreeNode {
    int keys[2 * MIN_DEGREE - 1];
    Mahasiswa *data[2 * MIN_DEGREE - 1];
    struct BPTreeNode *children[2 * MIN_DEGREE];
    struct BPTreeNode *next;
    int leaf;
    int n;
} BPTreeNode;
```

Penjelasan:

- Struct ini merepresentasikan node dalam B+ Tree dengan properti:

    - `keys`: Array integer untuk menyimpan ID mahasiswa sebagai key.

    - `data`: Array pointer ke struct `Mahasiswa` (data aktual).

    - `children`: Array pointer ke child node (digunakan jika node bukan leaf).

    - `next`: Pointer ke node berikutnya (hanya digunakan di **leaf node** untuk traversal berurutan).

    - `leaf`: Flag untuk menandai apakah node adalah leaf (`1`) atau internal node (`0`).

    - `n`: Jumlah key/data yang saat ini tersimpan di node.

- Kapasitas Node:

    - Setiap node dapat menyimpan maksimal `2 * MIN_DEGREE - 1` key/data.

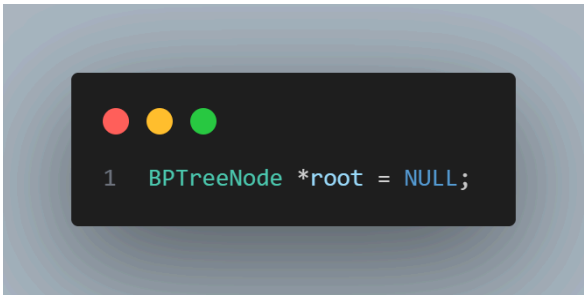
    - Setiap node (kecuali leaf) dapat memiliki maksimal `2 * MIN_DEGREE` child.

3. B+ Tree (Balanced Search Tree)

Karakteristik B+ Tree dalam Kode Ini:

1. **Order (`MIN_DEGREE`):**
   - Didefinisikan sebagai `3`, artinya
     - Setiap node (kecuali root) harus memiliki minimal `MIN_DEGREE - 1` key (yaitu 2 key).
     - Maksimal key per node adalah `2 * MIN_DEGREE - 1` (yaitu 5 key).
2. **Struktur:**
   - Leaf Node:
     - Menyimpan key (`keys`) dan data (`data`).
     - Memiliki pointer `next` untuk menghubungkan leaf node secara berurutan.
   - Internal Node:
     - Menyimpan key (`keys`) dan pointer ke child node (`children`).
     - Tidak menyimpan data aktual (hanya key sebagai pembatas).
3. **Operasi:**
   - Insert: Memastikan node tidak melebihi kapasitas, lalu memecah node jika penuh (`splitChild`).
   - Search: Pencarian dimulai dari root, lalu berpindah ke child sesuai key.
   - Range Search: Melintasi leaf node berurutan menggunakan pointer `next`.
   - Delete: Menghapus data dari leaf node dan menjaga keseimbangan tree.

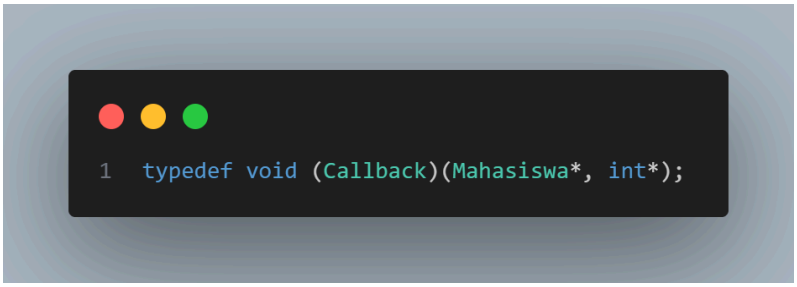
4. Variabel Global `root`



Penjelasan:

- Pointer ke root node dari B+ Tree.
- Diinisialisasi sebagai NULL saat program dimulai.
- Diperbarui saat operasi insert/split mengubah struktur tree.

5. Callback Function (`printMahasiswa`)



```
typedef void (Callback)(Mahasiswa*, int*);
```

Penjelasan:

- Fungsi callback digunakan saat traversal tree untuk mencetak data mahasiswa.
- Parameter `j` adalah counter untuk nomor urut output.

ii. Penjelasan Fungsi.

1. Fungsi `createNode`

```
BPTreeNode* createNode(int leaf) {
    BPTreeNode *node =  (BPTreeNode *)malloc(sizeof(BPTreeNode));
    node->leaf = leaf;
    node->n = 0;
    node->next = NULL;
    for (int i = 0; i < 2 * MIN_DEGREE; i++) node->children[i] = NULL;
    for (int i = 0; i < 2 * MIN_DEGREE - 1; i++) node->data[i] = NULL;
    return node;
}
```

Penjelasan:

Fungsi ini digunakan untuk membuat node baru dalam B+ Tree. Node yang dibuat dapat berupa leaf node atau internal node tergantung pada parameter `leaf`. Fungsi mengalokasikan memori untuk node baru, menginisialisasi nilai-nilai default seperti `n`

(jumlah key), `next` (pointer ke node berikutnya untuk leaf node), dan mengosongkan array `children` serta `data`.

2. Fungsi `printMahasiswa`

```c
void printMahasiswa(Mahasiswa *mhs, int *j) {
    printf("%d. ID: %d | Nama: %s | Kota: %s | Tanggal Lahir: %s | Email: %s\n",
           *j, mhs->id, mhs->name, mhs->city, mhs->date, mhs->email);
}
```

Penjelasan:

Fungsi ini digunakan untuk mencetak informasi mahasiswa dalam format yang rapi. Parameter `j` adalah nomor urut yang akan ditampilkan, sedangkan `mhs` adalah pointer ke struct `Mahasiswa` yang berisi data mahasiswa. Fungsi ini dipanggil saat traversal atau pencetakan semua data.

3. Fungsi `traverse`

```c
void traverse(BPTreeNode *node, Callback cb, int *j, int *iterations) {
    if (node == NULL) return;
    (*iterations)++;
    int i;
    for (i = 0; i < node->n; i++) {
        if (!node->leaf)
            traverse(node->children[i], cb, j, iterations);
        cb(node->data[i], j);
        (*j)++;
    }
    if (!node->leaf)
        traverse(node->children[i], cb, j, iterations);
}
```

Penjelasan:

Fungsi ini melakukan traversal pada B+ Tree secara in-order. Jika node bukan leaf node, fungsi akan rekursif menelusuri anak-anaknya. Setiap data mahasiswa diproses menggunakan callback function (`cb`), yang biasanya adalah `printMahasiswa`. Variabel `j` digunakan untuk nomor urut, dan `iterations` menghitung jumlah operasi yang dilakukan.

4. Fungsi `splitChild`

```c
void splitChild(BPTreeNode *parent, int i, int *iterations) {
    (*iterations)++;
    BPTreeNode *y = parent->children[i];
    BPTreeNode *z = createNode(y->leaf);
    z->n = MIN_DEGREE - 1;

    for (int j = 0; j < MIN_DEGREE - 1; j++) {
        z->keys[j] = y->keys[j + MIN_DEGREE];
        z->data[j] = y->data[j + MIN_DEGREE];
    }

    if (!y->leaf) {
        for (int j = 0; j < MIN_DEGREE; j++)
            z->children[j] = y->children[j + MIN_DEGREE];
    } else {
        z->next = y->next;
        y->next = z;
    }

    y->n = MIN_DEGREE - 1;

    for (int j = parent->n; j >= i + 1; j--)
        parent->children[j + 1] = parent->children[j];
    parent->children[i + 1] = z;

    for (int j = parent->n - 1; j >= i; j--) {
        parent->keys[j + 1] = parent->keys[j];
        parent->data[j + 1] = parent->data[j];
    }
    parent->keys[i] = y->keys[MIN_DEGREE - 1];
    parent->data[i] = y->data[MIN_DEGREE - 1];
    parent->n++;
}
```

Penjelasan:

Fungsi ini memisahkan child node yang penuh (jumlah key = `2 * MIN_DEGREE - 1`) menjadi dua node. Node `y` adalah child yang penuh, dan node `z` adalah node baru yang dibuat. Key dan data dari `y` dibagi dua, dengan separuh dipindahkan ke `z`. Jika `y` adalah leaf node, `z` akan menjadi node berikutnya dari `y`. Fungsi ini menjaga sifat B+ Tree setelah pemisahan.

5. Fungsi `insertNonFull`

```c
void insertNonFull(BPTreeNode *node, Mahasiswa *mhs, int *iterations) {
    (*iterations)++;
    int i = node->n - 1;
    if (node->leaf) {
        while (i >= 0 && mhs->id < node->keys[i]) {
            (*iterations)++;
            node->keys[i + 1] = node->keys[i];
            node->data[i + 1] = node->data[i];
            i--;
        }
        node->keys[i + 1] = mhs->id;
        node->data[i + 1] = mhs;
        node->n++;
    } else {
        while (i >= 0 && mhs->id < node->keys[i]) {
            (*iterations)++;
            i--;
        }
        i++;
        if (node->children[i]->n == 2 * MIN_DEGREE - 1) {
            splitChild(node, i, iterations);
            if (mhs->id > node->keys[i])
                i++;
        }
        insertNonFull(node->children[i], mhs, iterations);
    }
}

```

Penjelasan:

Fungsi ini menyisipkan data mahasiswa ke dalam node yang belum penuh. Jika node adalah leaf node, data akan disisipkan pada posisi yang tepat. Jika node adalah internal

node, fungsi akan mencari child yang sesuai untuk penyisipan rekursif. Jika child penuh, fungsi `splitChild` akan dipanggil.

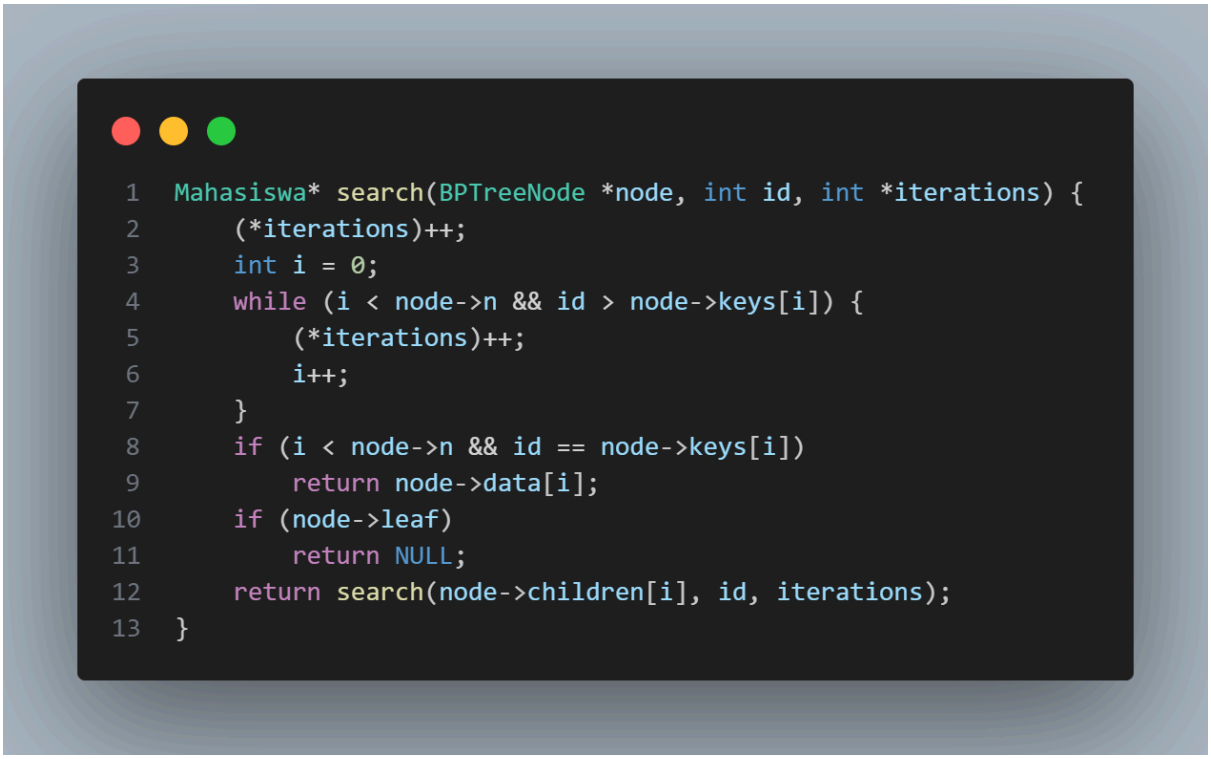
6. Fungsi `insert`

```c
void insert(Mahasiswa *mhs) {
    int iterations = 0;
    if (root == NULL) {
        root = createNode(1);
        root->keys[0] = mhs->id;
        root->data[0] = mhs;
        root->n = 1;
        iterations++;
    } else {
        if (root->n == 2 * MIN_DEGREE - 1) {
            BPTreeNode *s = createNode(0);
            s->children[0] = root;
            splitChild(s, 0, &iterations);
            int i = 0;
            if (mhs->id > s->keys[0])
                i++;
            insertNonFull(s->children[i], mhs, &iterations);
            root = s;
        } else {
            insertNonFull(root, mhs, &iterations);
        }
    }
    printf("Insert operation completed with %d iterations\n", iterations);
}
```

Penjelasan:

Fungsi utama untuk menyisipkan data mahasiswa ke dalam B+ Tree. Jika root kosong, root baru akan dibuat. Jika root penuh, root akan di-split terlebih dahulu. Fungsi memanggil `insertNonFull` untuk penyisipan data dan mencetak jumlah iterasi yang dilakukan.

7. Fungsi `search`



```
Mahasiswa* search(BPTreeNode *node, int id, int *iterations) {
    (*iterations)++;
    int i = 0;
    while (i < node->n && id > node->keys[i]) {
        (*iterations)++;
        i++;
    }
    if (i < node->n && id == node->keys[i])
        return node->data[i];
    if (node->leaf)
        return NULL;
    return search(node->children[i], id, iterations);
}
```

Penjelasan:

Fungsi ini mencari data mahasiswa berdasarkan ID. Pencarian dilakukan secara rekursif dengan membandingkan ID dengan key di setiap node. Jika ditemukan, fungsi mengembalikan pointer ke data mahasiswa. Jika tidak, mengembalikan `NULL`.

8. Fungsi `deletedata`

```c
int deletedata(BPTreeNode *node, int id, int *iterations) {
    (*iterations)++;
    for (int i = 0; i < node->n; i++) {
        (*iterations)++;
        if (node->keys[i] == id) {
            free(node->data[i]);
            for (int j = i; j < node->n - 1; j++) {
                node->keys[j] = node->keys[j + 1];
                node->data[j] = node->data[j + 1];
            }
            node->n--;
            return 1;
        }
    }
    if (node->leaf) return 0;
    for (int i = 0; i <= node->n; i++) {
        if (deletedata(node->children[i], id, iterations)) return 1;
    }
    return 0;
}
```

Penjelasan:

Fungsi ini menghapus data mahasiswa berdasarkan ID. Jika data ditemukan di leaf node, data akan dihapus dan struktur array diperbarui. Jika tidak ditemukan, pencarian dilanjutkan ke child node. Fungsi mengembalikan 1 jika berhasil dan 0 jika gagal.

9. Fungsi `loadCSV`

```c
void loadCSV(const char *filename) {
    int iterations = 0;
    FILE *file = fopen(filename, "r");
    if (!file) {
        printf("Gagal membuka file %s\n", filename);
        return;
    }

    char line[256];
    fgets(line, sizeof(line), file);

    while (fgets(line, sizeof(line), file)) {
        iterations++;
        Mahasiswa *mhs = (Mahasiswa *) malloc(sizeof(Mahasiswa));
        sscanf(line, "%49[^,],%d,%49[^,],%19[^,],%49[^\n]",
               mhs->name, &mhs->id, mhs->city, mhs->date, mhs->email);
        insert(mhs);
    }
    fclose(file);
    printf("Data berhasil dimuat! Total records processed: %d\n", iterations);
}
```

Penjelasan:

Fungsi ini membaca data mahasiswa dari file CSV dan memasukkannya ke dalam B+ Tree. Setiap baris di file CSV di-parsing menjadi struct `Mahasiswa`, lalu disisipkan menggunakan fungsi `insert`. Jumlah total record yang diproses dicetak di akhir.

10. Fungsi `searchRange`

```c
void searchRange(BPTreeNode *node, int low, int high) {
    int iterations = 0;
    if (node == NULL) return;

    int i = 0;
    while (!node->leaf) {
        iterations++;
        while (i < node->n && low > node->keys[i]) {
            iterations++;
            i++;
        }
        node = node->children[i];
        i = 0;
    }

    int done = 0;
    while (node && !done) {
        iterations++;
        for (i = 0; i < node->n; i++) {
            iterations++;
            if (node->keys[i] > high) {
                done = 1;
                break;
            }
            if (node->keys[i] >= low)
                printf("ID: %d | Nama: %s\n", node->keys[i], node->data[i]->name);
        }
        node = node->next;
    }
    printf("Range search completed with %d iterations\n", iterations);
}

```

Penjelasan:

Fungsi ini mencari semua data mahasiswa dengan ID dalam rentang `low` hingga `high`. Pencarian dimulai dari leaf node yang sesuai dengan `low`, kemudian

melanjutkan ke leaf node berikutnya hingga ID melebihi `high`. Hasil pencarian dicetak beserta jumlah iterasi.

11. Fungsi `main`

Fungsi utama yang menangani interaksi pengguna melalui menu. Menu mencakup operasi seperti memuat data dari CSV, menambah/mencari/menghapus data, dan pencarian rentang ID. Setiap operasi memanggil fungsi-fungsi yang sesuai dan menampilkan hasilnya.

BAB III

i. Pembagian Tugas Anggota Kelompok

Fitur	Deskripsi	Struktur Data	PJ
Create	Menambahkan data baru ke dalam struktur	Hash Table	Reza
	Menambahkan data baru ke dalam struktur	B+ Tree	Kaka
Read	Membaca dan memuat data dari file CSV ke dalam struktur data	Hash Table	Pasya
	Membaca dan memuat data dari file CSV ke dalam struktur data	B+ Tree	Kaka
Update	Mengubah data mahasiswa berdasarkan ID	Hash Table	Reza
	Mengubah data mahasiswa berdasarkan ID	B+ Tree	Kaka
Delete	Menghapus data mahasiswa berdasarkan ID	Hash Table	Pasya
	Menghapus data mahasiswa berdasarkan ID	B+ Tree	Kaka
Cari Data	Mencari data mahasiswa berdasarkan ID	Hash Table	Reza
Tampilkan Data	Menampilkan seluruh data mahasiswa dari struktur Hash Table	Hash Table	Pasya

ii. Analisis Perbandingan

A. Operasi Create

Pada saat proses pemasukan data, Hash Table menunjukkan performa yang sangat cepat, terutama pada dataset kecil (100.csv) dan sedang (500.csv). Dengan kompleksitas rata-rata **O(1)**, data dapat langsung disisipkan ke indeks hasil fungsi hash tanpa perlu mempertimbangkan urutan.

Namun, pada dataset besar (1000.csv), kinerja Hash Table mulai menurun. Terdapat 55 data yang gagal dimasukkan akibat kolisi hash yang tidak bisa diatasi (ID yang berbeda menghasilkan indeks hash yang sama dan sudah terisi). Hal ini menunjukkan bahwa Hash Table sangat bergantung pada kualitas fungsi hash dan load factor.

Sementara itu, B+ Tree mempertahankan performa yang konsisten dalam setiap ukuran dataset. Meskipun proses penyisipan memerlukan traversal (dengan kompleksitas **O(log n)**), struktur pohon tetap seimbang dan data tersimpan dalam urutan yang teratur, sehingga lebih stabil dan tidak mengalami masalah seperti kolisi.

B. Operasi Read

Pada tahap pembacaan (read), yang dimaksud di sini bukan sekadar mencari data, melainkan proses membaca file CSV dan memuatnya ke dalam struktur data. Dalam konteks ini, Hash Table menunjukkan performa yang sangat baik pada dataset kecil dan sedang. Proses pemuatan data berlangsung cepat karena Hash Table cukup melakukan hashing dan chaining jika terjadi konflik.

Namun, pada dataset besar, proses pemuatan Hash Table mulai terhambat karena meningkatnya jumlah kolisi. Data yang memiliki ID unik namun menghasilkan hash yang sama harus dikelola melalui linked list (chaining), yang secara tidak langsung memperlambat proses input.

B+ Tree membutuhkan waktu lebih karena setiap data harus dimasukkan ke lokasi yang tepat berdasarkan urutan, dan bila perlu dilakukan rebalancing untuk menjaga keseimbangan pohon. Namun, keunggulannya adalah data terstruktur secara terurut sejak awal, sehingga sangat menguntungkan untuk operasi selanjutnya seperti pencarian rentang.

C. Operasi Update

Pada Hash Table, update data cukup sederhana dan cepat jika data langsung ditemukan di bucket yang sesuai. Namun, saat terjadi chaining, proses pencarian data yang ingin di update bisa menjadi lebih lambat dan berisiko jika tidak dikelola dengan baik.

B+ Tree menawarkan pembaruan yang lebih terstruktur. Setelah pencarian node (O(log n)), data dapat langsung dimodifikasi tanpa mengubah struktur pohon, karena update tidak memengaruhi urutan atau pointer antar node. Ini menjadikan B+ Tree lebih stabil dan aman dalam operasi update, terutama untuk dataset besar.

D. Operasi Delete

Hash Table memungkinkan penghapusan data dengan cepat selama chaining tidak terlalu panjang. Namun, saat jumlah kolisi tinggi (seperti pada 1000.csv), penghapusan bisa menjadi kompleks dan berpotensi memengaruhi data lain yang ada dalam satu bucket.

B+ Tree membutuhkan rebalancing saat node kosong atau terlalu sedikit data, tetapi hal ini justru menjaga struktur tetap optimal. Meskipun penghapusan lebih kompleks dibandingkan Hash Table, keuntungannya adalah performa pohon tetap stabil untuk operasi-operasi berikutnya.

E. Operasi Pencarian Data

- **Pencarian Eksak :**

Hash Table memiliki keunggulan mutlak untuk pencarian jenis ini. Dengan asumsi distribusi hash yang baik, data dapat diakses langsung dengan kompleksitas **O(1)**. Ini sangat cocok untuk sistem yang sering mencari data berdasarkan ID tertentu.

- **Pencarian Rentang :**

Hash Table tidak mendukung pencarian range secara efisien karena tidak menyimpan data dalam urutan tertentu. Untuk mencari data antara dua ID, seluruh tabel harus dipindai.

Sebaliknya, **B+ Tree sangat unggul untuk pencarian rentang**, karena seluruh data disimpan secara terurut dan node daunnya saling terhubung. Cukup cari batas bawah, lalu lanjutkan traversal ke node berikutnya hingga batas atas tercapai. Operasi ini sangat efisien dan hanya memerlukan traversal searah.

iii. Kesimpulan

Pada dataset yang berukuran kecil, seperti 100.csv, Hash Table terbukti menjadi pilihan yang sangat efisien. Proses pemuatan data berlangsung cepat, pencarian berdasarkan ID dapat dilakukan hampir seketika, dan tidak terjadi banyak kolisi. Dengan kompleksitas waktu yang rendah, Hash Table sangat cocok digunakan dalam sistem yang hanya memerlukan pencarian eksak dan memiliki data yang relatif sedikit serta tidak sering berubah.

Namun, ketika jumlah data mulai meningkat, seperti pada 500.csv dan 1000.csv, efisiensi Hash Table mulai menurun. Kolisi yang terjadi mengakibatkan struktur chaining menjadi panjang, sehingga operasi pencarian, pembaruan, dan penghapusan membutuhkan waktu lebih lama dari seharusnya. Bahkan pada file 1000.csv, sebanyak 55 data gagal dimasukkan karena keterbatasan struktur dalam menangani tabrakan hash.

Sebaliknya, B+ Tree menunjukkan performa yang lebih stabil pada semua skala data. Walaupun proses insert dan update memiliki kompleksitas lebih tinggi dibandingkan Hash Table, B+ Tree mampu menjaga struktur data tetap seimbang dan terurut. Hal ini membuatnya unggul terutama pada operasi pencarian rentang yang tidak dapat dilakukan secara efisien oleh Hash Table. Selain itu, karena setiap node daunnya saling terhubung, B+ Tree sangat cocok untuk sistem yang memerlukan traversal data secara berurutan.

Dengan mempertimbangkan kebutuhan dan karakteristik masing-masing struktur, dapat disimpulkan bahwa Hash Table merupakan pilihan terbaik untuk data kecil dengan pencarian eksak, sedangkan B+ Tree lebih unggul dalam menangani data besar, terutama ketika operasi pembaruan, penghapusan, dan pencarian rentang menjadi prioritas. Maka dari itu, pemilihan struktur data sebaiknya disesuaikan dengan skenario aplikasi yang akan dikembangkan, agar kinerja sistem tetap optimal dan efisien di berbagai kondisi.