FINAL PROJECT – Struktur Data

LAPORAN PERBANDINGAN PERFORMA B+ TREE DAN HASH MAP PADA IMPLEMENTASI DATABASE

Anggota Kelompok:

Farrel Hasudungan Immanuel Limbong (5025241016)

Muhammad Nabil Fauzan (5025241024)

Muhammad Quthbi Danish Abqori (5025241036)

Dosen Pembimbing

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

Program Studi Teknik Informatika

Departemen Teknik Informatika

Fakultas Teknologi Elektro dan Informatika Cerdas

Institut Teknologi Sepuluh Nopember

Surabaya

Tahun 2025

FINAL PROJECT – Struktur Data

LAPORAN PERBANDINGAN
PERFORMA B+ TREE DAN HASH MAP

Anggota Kelompok:

Farrel Hasudungan Immanuel Limbong (5025241016)

Muhammad Nabil Fauzan (5025241024)

Muhammad Quthbi Danish Abqori (5025241036)

Dosen Pembimbing

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

Program Studi Teknik Informatika

Departemen Teknik Informatika

Fakultas Teknologi Elektro dan Informatika Cerdas

Institut Teknologi Sepuluh Nopember

Surabaya

Tahun 2025

PEMBAGIAN KERJA ANTAR ANGGOTA

Tugas	**Anggota yang mengerjakan**
Fungsi Create	Muhammad Quthbi Danish Abqori
Fungsi Search	Muhammad Quthbi Danish Abqori
Fungsi Range Search	Muhammad Quthbi Danish Abqori
Fungsi Update	Muhammad Quthbi Danish Abqori
Fungsi Remove	Muhammad Quthbi Danish Abqori
Laporan	Muhammad Nabil Fauzan & Farrel Hasudungan Immanuel Limbong

PENDAHULUAN

1.1 Latar Belakang

Dalam ranah teknik informatika dan rekayasa perangkat lunak, struktur data merupakan fondasi konseptual dan praktis yang menopang hampir semua sistem komputasi. Pilihan struktur data yang tepat bukanlah sekadar detail implementasi, melainkan sebuah keputusan arsitektural fundamental yang dampaknya merambat ke seluruh aspek aplikasi, termasuk kinerja, skalabilitas, responsivitas, dan efisiensi penggunaan sumber daya. Tantangan utama dalam manajemen koleksi data dinamis adalah mencapai keseimbangan optimal antara berbagai metrik yang seringkali saling bertentangan: latensi operasi (penyisipan, pencarian, pembaruan, dan penghapusan), penggunaan memori, dan kompleksitas implementasi. Dalam proyek ini, kami membandingkan dua struktur data populer—B+ Tree dan Hash Map—dalam hal kinerja waktu dan penggunaan memori untuk empat operasi dasar: *insert, search (exact search & range search), update,* dan *delete*. Studi ini bertujuan untuk memahami kelebihan dan keterbatasan masing-masing struktur dalam konteks aplikasi skala kecil hingga menengah, khususnya *in-memory computation.*

1.2 Tujuan

Tujuan dari laporan ini adalah untuk melakukan analisis terhadap karakteristik dua struktur data, yaitu Hashmap dan B+ Tree. Analisis ini akan melakukan perbandingan seperti kecepatan pencarian secara eksak ataupun range, penggunaan memori, dan skala data.

DASAR TEORI

2.1 B+ Tree

B+ Tree adalah evolusi dari B-Tree yang dirancang khusus untuk mengoptimalkan efisiensi, terutama dalam sistem yang menangani data dalam jumlah besar, seperti basis data dan sistem file. B+ Tree menyimpan semua nilai hanya pada node daun, sedangkan node internal hanya menyimpan kunci untuk navigasi. Semua daun berada pada tingkat yang sama dan saling terhubung melalui pointer next, menjadikan *range queries* sangat efisien.

2.1.1 Arsitektur dan Terminologi

a. Node Internal (Internal Nodes

Node-node ini berfungsi murni sebagai "papan penunjuk jalan" dalam pohon. Mereka hanya berisi nilai-nilai kunci yang bertindak sebagai pemisah dan pointer yang menunjuk ke node anak. Yang terpenting, node internal tidak menyimpan data (nilai) yang sebenarnya. Desain ini memungkinkan lebih banyak kunci untuk dimuat dalam satu node, yang secara langsung meningkatkan fan-out (jumlah anak per node).

b. Node Daun (Leaf Nodes)

Ini adalah level terendah dari pohon dan merupakan satu-satunya tempat di mana pasangan kunci-nilai (data aktual) disimpan. Semua node daun berada pada level kedalaman yang sama, sebuah properti yang menjamin bahwa pohon selalu seimbang. Keseimbangan ini memastikan bahwa waktu yang dibutuhkan untuk mencapai data apa pun dari akar adalah sama.

2.1.2 Karakteristik utama:

- Kompleksitas pencarian: $O(\log_m n)$
- Kompleksitas range search: $O(\log_m n + k)$
- Fan-out besar → tinggi pohon lebih rendah → I/O lebih sedikit (penting untuk disk-based storage)

2.2 Hash Map

Hash Map menggunakan fungsi hash untuk memetakan kunci ke indeks dalam sebuah array. C++ menyediakan *std::unordered_map* sebagai implementasi hash table yang sangat dioptimalkan.

2.2.1 Arsitektur dan Terminologi

a. Fungsi Hash (Hash Function)

Ini adalah fungsi matematis yang mengambil kunci dari tipe data apa pun (misalnya, integer, string) dan mengubahnya menjadi sebuah nilai integer dengan panjang tetap, yang disebut hash code atau hash. Fungsi hash yang baik harus deterministik (kunci yang sama selalu menghasilkan hash yang sama) dan

mendistribusikan kunci yang berbeda secara seragam ke seluruh rentang outputnya untuk meminimalkan tabrakan.

b. Tabel Hash (Hash Table / Buckets)

Ini adalah sebuah array internal. Hash code yang dihasilkan oleh fungsi hash digunakan untuk menghitung indeks dalam array ini. Indeks ini menentukan di "bucket" mana elemen akan disimpan.

c. Resolusi Kolisi (Collision Resolution)

Tidak dapat dihindari bahwa dua kunci yang berbeda dapat menghasilkan hash code yang sama setelah dimodulus dengan ukuran tabel. Peristiwa ini disebut kolisi. Implementasi *std::unordered_map* yang umum, termasuk yang digunakan oleh kompiler GCC (libstdc++), menggunakan teknik yang disebut Separate Chaining. Dalam teknik ini, setiap bucket dalam tabel hash tidak menyimpan elemen secara langsung, melainkan sebuah pointer ke *head* sebuah linked list. Semua elemen yang di-hash ke bucket yang sama akan ditambahkan ke dalam linked list tersebut.

2.2.2 Karakteristik utama:

- Kompleksitas rata-rata: O(1) untuk insert, search, delete
- Tidak mendukung range queries
- Pada kasus terburuk (*collision* tinggi): O(n)
- Menggunakan teknik chaining/open addressing dan rehashing untuk mempertahankan kinerja

CARA KERJA PROGRAM

3.1 Arsitektur Fungsi Main

Fungsi main berfungsi sebagai test harness untuk eksperimen ini. Alur kerjanya dapat diuraikan sebagai berikut:

1. Program menerima argumen baris perintah (bptree atau hashmap) untuk menentukan struktur data mana yang akan diuji.
2. Program melakukan loop melalui serangkaian ukuran data yang telah ditentukan (sizes = {100, 500, 1000}). Berdasarkan output yang diberikan, pengujian juga dijalankan untuk ukuran yang jauh lebih besar, yaitu 200,000 elemen.
3. Untuk setiap ukuran n, sebuah *std::vector<int>* dibuat. Fungsi *std::iota* digunakan untuk mengisinya dengan urutan bilangan bulat dari 1 hingga n.
4. *std::shuffle* kemudian digunakan untuk mengacak urutan elemen dalam vektor. Langkah ini sangat penting karena memastikan bahwa data yang disisipkan ke dalam struktur data tidak memiliki urutan atau pola yang dapat diprediksi. Ini menciptakan skenario pengujian kasus rata-rata yang adil untuk kedua struktur data. Secara khusus, ini menghindari kasus terburuk untuk B+ Tree (penyisipan data yang sudah terurut, yang menyebabkan pemisahan node secara konstan) dan juga menghindari kasus terburuk untuk Hash Map (data yang sengaja dirancang untuk menyebabkan kolisi hash maksimum).
5. Program memanggil fungsi testBPlusTree atau testHashMap yang sesuai, meneruskan vektor data yang telah diacak.

3.2 Implementasi BPlusTree

Implementasi B+ Tree kustom ini merupakan inti dari salah satu sisi perbandingan.

- **Struktur Node**

  Definisi struct Node mencakup anggota-anggota penting:

  a. *bool leaf* untuk membedakan node internal dan daun.
  b. *vector<int> keys* dan *vector<int> values* untuk menyimpan data.
  c. *vector<Node*>* children untuk pointer ke anak.
  d. *Node* next* untuk tautan sekuensial antar node daun.

  Penggunaan *std::vector* untuk menyimpan kunci dan anak di dalam satu node adalah pilihan desain yang menarik. Ini memberikan lokalitas data yang baik di dalam node itu sendiri. Ketika sebuah node dimuat ke dalam cache CPU, semua kuncinya berada dalam blok memori yang berdekatan, memungkinkan pencarian biner yang efisien *(std::upper_bound)* di dalam node.

- **Fungsi insertInternal**

Fungsi ini mengimplementasikan logika rekursif untuk penyisipan. Ia menavigasi ke bawah pohon ke node daun yang sesuai. Jika penyisipan menyebabkan node (baik daun maupun internal) melebihi kapasitasnya (didefinisikan oleh order), node tersebut akan dibagi. Logika pemisahan ini mengikuti pola standar B+ Tree: untuk node daun, kunci pertama dari *sibling* baru disalin ke induk; untuk node internal, kunci tengah dipromosikan ke induk. Ini adalah mekanisme yang memungkinkan pohon tumbuh secara seimbang.

- **Fungsi rangeSearch**

  Fungsi ini pertama-tama melakukan traversal logaritmik (O(log n)) untuk menemukan node daun pertama yang mungkin berisi kunci dalam rentang yang dicari. Setelah itu, ia tidak lagi perlu menavigasi pohon. Sebaliknya, ia hanya perlu melintasi *linked list* node daun menggunakan pointer node->next hingga menemukan kunci yang melebihi batas atas rentang. Ini adalah implementasi textbook dari pencarian rentang yang efisien.

- **Fungsi remove**

  Kode untuk remove hanya mencari kunci di node daun dan, jika ditemukan, menghapusnya dari *std::vector keys* dan *values* menggunakan *vector::erase*.

3.3 Implementasi Hash Map

Program ini menggunakan std::unordered_map dari Pustaka Standar C++, yang merupakan implementasi Hash Map yang sangat dioptimalkan.

- **Operasi Standar**

  Penggunaan mp[v] = v untuk penyisipan dan pembaruan, volatile int x = mp[v] untuk pencarian, dan mp.erase(v) untuk penghapusan adalah cara yang idiomatis dan efisien untuk berinteraksi dengan *std::unordered_map*. Penggunaan volatile adalah trik untuk mencegah kompiler mengoptimalkan loop pencarian, memastikan bahwa pencarian benar-benar dilakukan.

- **Analisis Kritis rangeSearch**

  Implementasi pencarian rentang untuk Hash Map sangatlah penting untuk dianalisis. Kode tersebut menggunakan loop bersarang:

```
for (auto &r: ranges) {
    for (auto &p: mp) {
        if (p.first >= r.first && p.first <= r.second) {
            //...
        }
    }
}
```

Ini adalah implementasi *brute-force*. Untuk setiap kueri rentang (di mana QUERY_RANGE = 100), ia mengiterasi **setiap elemen** di dalam unordered_map. Ini menghasilkan kompleksitas waktu total (O(M*N)), di mana M adalah jumlah kueri rentang dan N adalah jumlah total elemen dalam map. Implementasi ini dengan sempurna menggambarkan kelemahan fundamental Hash Map: karena tidak adanya keterurutan, tidak ada cara yang lebih baik untuk melakukan pencarian rentang selain memeriksa setiap elemen satu per satu.

3.4 Metodologi Pengukuran Kinerja

- **Pengukuran Waktu**

  Penggunaan *std::chrono::high_resolution_clock* ini menyediakan jam dengan resolusi setinggi mungkin yang ditawarkan oleh sistem, memungkinkan pengukuran interval waktu yang sangat singkat dengan presisi.

- **Pengukuran Memori (getCurrentRSS)**

  Fungsi ini menggunakan Windows API untuk mengukur penggunaan memori. Pengukuran ini menggunakan fungsi *GetProcessMemoryInfo* dari *psapi.h* yang kemudian akan mengambil *WorkingSetSize* dari struktur *PROCESS_MEMORY_COUNTERS*. *WorkingSetSize* merupakan jumlah memori fisik (RAM) yang saat ini digunakan oleh proses. Dengan mengukur delta WorkingSetSize sebelum dan sesudah operasi penyisipan, program ini memberikan perkiraan berapa banyak memori tambahan yang digunakan oleh struktur data untuk menyimpan elemen-elemennya.

3.5 Compile dan Run Program

Program ini dikompilasi menggunakan G++ (GNU C++ Compiler), yang umum ditemukan pada sistem operasi Windows (melalui MinGW/MSYS2), Linux, dan macOS. Gunakanlah perintah berikut di terminal untuk melakukan kompilasi program:

```
g++ compare_bptree_hashmap.cpp -o compare.exe -std=c++17 -lpsapi
```

Perintah ini masih dapat disesuaikan dengan nama kode **.cpp** yang sesuai dengan nama kode pada direktori sendiri dan **.exe** yang diinginkan. Opsi -std=c++17 menginstruksikan kompiler untuk menggunakan standar bahasa C++17. Opsi -l digunakan untuk menautkan (*link*) sebuah *library*. psapi adalah "Process Status Helper API" di Windows. *Library* ini **wajib** ditautkan karena program menggunakan fungsi GetProcessMemoryInfo untuk mengukur penggunaan memori, dan fungsi tersebut merupakan bagian dari PSAPI. Tanpa ini, akan terjadi *linker error*.

Setelah kompilasi berhasil dan file compare.exe terbentuk, program dapat dijalankan dari terminal. Program ini memerlukan satu argumen untuk memilih mode pengujian.

`.\compare.exe bptree` atau `.\compare.exe hashmap`.

HASIL PENGUJIAN

Struktur	**N**	***Insert (ms)***	***Mem (KB)***	***Search (ms)***	***Range Search (ms)***	***Update (ms)***	***Delete (ms)***
B+ Tree	100	0.1346	20	0.0269	0.5382	0.0270	0.2158
B+ Tree	500	0.7023	36	0.1986	1.6910	0.2763	0.8610
B+ Tree	1000	1.2612	48	0.4196	2.6382	0.3155	0.8589
Hash Map	100	0.0432	16	0.0052	0.1387	0.0045	0.0117
Hash Map	500	0.1446	16	0.0243	1.4232	0.0639	0.0914
Hash Map	1000	0.2898	24	0.0520	1.4837	0.0388	0.1785
B+ Tree	200K	67.3864	2744	38.1240	371962.0000	40.6410	87.7229
Hash Map	200K	32.6199	9180	10.1103	414584.0000	9.2587	14.6431

```
PS D:\kuliah\semester 2\strukdat\final project> .\compare.exe hashmap
Hash Map with 100 items:
  Insert time: 0.0176 ms, Memory delta: 12 KB
  Exact search time: 0.0026 ms
  Range search time (100 queries): 0.0774 ms
  Update time: 0.0027 ms
  Delete time: 0.0067 ms

Hash Map with 500 items:
  Insert time: 0.079 ms, Memory delta: 16 KB
  Exact search time: 0.0135 ms
  Range search time (100 queries): 0.4079 ms
  Update time: 0.0138 ms
  Delete time: 0.0318 ms

Hash Map with 1000 items:
  Insert time: 0.1807 ms, Memory delta: 28 KB
  Exact search time: 0.028 ms
  Range search time (100 queries): 1.0424 ms
  Update time: 0.0243 ms
  Delete time: 0.07 ms

PS D:\kuliah\semester 2\strukdat\final project> .\compare.exe bptree
B+ Tree with 100 items:
  Insert time: 0.0749 ms, Memory delta: 20 KB
  Exact search time: 0.013 ms
  Range search time (100 queries): 0.1825 ms
  Update time: 0.0149 ms
  Delete time: 0.0421 ms

B+ Tree with 500 items:
  Insert time: 0.4213 ms, Memory delta: 40 KB
  Exact search time: 0.0872 ms
  Range search time (100 queries): 0.6421 ms
  Update time: 0.0795 ms
  Delete time: 0.2574 ms

B+ Tree with 1000 items:
  Insert time: 1.1084 ms, Memory delta: 92 KB
  Exact search time: 0.2325 ms
  Range search time (100 queries): 1.5272 ms
  Update time: 0.2273 ms
  Delete time: 0.9855 ms
```

```
PS D:\kuliah\semester 2\strukdat\final project> .\compare.exe bptree
B+ Tree with 200000 items:
  Insert time: 67.3864 ms, Memory delta: 2744 KB
  Exact search time: 38.124 ms
  Range search time (200000 queries): 371962 ms
  Update time: 40.641 ms
  Delete time: 87.7229 ms

PS D:\kuliah\semester 2\strukdat\final project> .\compare.exe hashmap
Hash Map with 200000 items:
  Insert time: 32.6199 ms, Memory delta: 9180 KB
  Exact search time: 10.1103 ms
  Range search time (200000 queries): 414584 ms
  Update time: 9.2587 ms
  Delete time: 14.6431 ms
```

PEMBAHASAN

5.1 Insert dan Exact Search

- Hash Map unggul signifikan dalam kecepatan pada dataset kecil hingga menengah karena O(1) *complexity* dan *cache-friendliness STL*.
- B+ Tree lebih lambat karena harus *traverse node, manage pointer,* dan *vector insert*, meskipun masih cukup kompetitif.

5.2 Range Search

- Hash Map lebih cepat untuk jumlah data kecil karena overhead B+ Tree (*pointer, cache miss*) cukup besar, sedangkan scan Hash Map langsung ke array dan sangat cepat.
- Namun, pada data besar (200K items), B+ Tree menunjukkan keunggulan struktur traversal linear antar daun, menjadikan performa lebih baik dari Hash Map yang tetap harus scan seluruh elemen.

5.3 Update dan Delete

- Update dan delete juga lebih cepat di Hash Map, namun karena tidak ada *rebalancing*, hasil B+ Tree masih stabil.
- Jika implementasi *rebalancing* ditambahkan, waktu delete bisa meningkat namun menjamin struktur tetap efisien dalam jangka panjang.

5.4 Memory Usage

- Hash Map lebih boros memori saat skala besar karena alokasi *bucket array* dan *rehashing.*
- B+ Tree lebih hemat untuk data besar berkat struktur pohon dan *page-level split.*

Pada pengujian dengan volume data yang relatif kecil (misalnya, 1.000 entri), Hash Map menunjukkan performa yang lebih unggul. Alasan utamanya adalah program ini menggunakan standar *library STL* dari C++ yang mana sudah dioptimalisasi sedemikian rupa (*cache-friendly*), di mana data cenderung disimpan dalam blok memori yang berdekatan, sehingga mengurangi latensi saat akses data. Sebaliknya, B+ Tree yang digunakan dalam pengujian ini merupakan implementasi sederhana buatan sendiri. Desain ini secara inheren menggunakan banyak *pointer* untuk menghubungkan *node* satu sama lain. Struktur seperti ini lebih rentan mengalami *cache miss*, di mana CPU harus mengambil data dari memori utama yang lebih lambat karena data yang dibutuhkan tidak ditemukan di *cache*.

Meskipun Hash Map lebih cepat pada skenario di atas, keunggulan teoretis dan praktis dari B+ Tree mulai terlihat secara signifikan ketika kondisi pengujian diubah. Ketika volume data meningkat secara masif, efisiensi B+ Tree dalam mengelola data di disk atau memori yang besar menjadi lebih nyata. B+ Tree secara fundamental dirancang untuk melakukan *range search* secara efisien. Semakin besar jangkauan data yang dicari, semakin jelas keunggulannya dibandingkan Hash Map yang harus melakukan iterasi pada banyak elemen. Dengan

menaikkan *order* B+ Tree (misalnya, ke *order* 64), setiap *node* dapat menampung lebih banyak *key* dan *pointer*. Hal ini mengurangi kedalaman pohon (*tree depth*) secara keseluruhan, sehingga jumlah operasi I/O yang dibutuhkan untuk mencari data menjadi lebih sedikit.

Penting untuk dicatat bahwa implementasi B+ Tree pada sistem manajemen basis data (DBMS) di dunia nyata (misalnya, pada PostgreSQL atau MySQL) jauh lebih kompleks dan dioptimalkan secara masif dibandingkan dengan implementasi sederhana yang dibuat untuk proyek ini.

KESIMPULAN

Berdasarkan teori dan data empiris:

- Hash Map cocok untuk aplikasi kecil dan in-memory, dengan kebutuhan pencarian eksak, kecepatan tinggi, dan *overhead* minim.
- B+ Tree unggul dalam skenario database besar, khususnya untuk operasi *range query* dan data yang tidak seluruhnya di-load ke memori utama.
- Pemilihan struktur data harus mempertimbangkan skala data, jenis operasi dominan, dan lingkungan eksekusi (*in-memory vs disk-based).*

Hasil eksperimen ini menggarisbawahi bahwa pilihan antara Hash Map dan B+ Tree sangat bergantung pada konteks. Untuk data kecil dan pencarian berbasis *key* tunggal (*point query*), Hash Map dari *library* standar adalah pilihan yang sangat efisien. Namun, untuk aplikasi yang menangani data dalam skala besar dan sering melakukan *query* jangkauan, B+ Tree dengan implementasi yang matang adalah struktur data yang lebih superior.