Aplikasi Pohon dalam Struktur Data *Hash Trie*

M. Isham Azmansyah F. 13514014
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
i@adimaja.com

***Abstrak* – Salah satu struktur data yang sering digunakan dalam pengembangan perangkat lunak adalah struktur data *map*, yang digunakan untuk memetakan suatu nilai ke nilai yang lain. Namun, struktur data *map* cukup sulit diimplementasikan secara efisien. Makalah ini menjelaskan salah satu alternatif implementasi struktur data map, yaitu dengan menggunakan *hash trie*. *Trie* adalah sebuah pohon n-ary terurut dengan beberapa sifat yang memudahkan penggunaannya sebagai struktur data *map*. Implementasi struktur data map dengan menggunakan *hash trie* bisa mendapatkan waktu pencarian, penambahan, dan penghapusan yang konstan.**

***Kata kunci* – map, pohon, struktur data, hash, trie**

I. PENDAHULUAN

Struktur data *map* adalah salah satu struktur data yang paling sering dipakai. Struktur data ini berfungsi untuk memetakan suatu data yang disebut *key* ke data lain yang biasa disebut *value*.

Aplikasi *map* dalam pengembangan software sangat banyak. Struktur *map* memang merupakan struktur yang sangat alami dan banyak muncul di kehidupan kita. Contohnya, mengakses data mahasiswa berdasarkan NIMnya.

Struktur data ini cukup sulit diimplementasikan. Idealnya, semua data harus bisa diakses dalam waktu yang konstan, tidak peduli ada berapa data yang tersimpan. Penambahan dan penghapusan data juga harus bisa dilakukan dengan waktu yang konstan.

Implementasi struktur data *map* sudah banyak, namun banyak implementasi yang memiliki kekurangan. Contohnya adalah *hash table* dan *binary search tree*.

Hash table memiliki waktu akses dan penghapusan yang konstan, namun struktur data ini memiliki kapasitas yang terbatas. Bila data yang baru terus ditambahkan, harus dilakukan operasi *resize* yang relatif lambat.

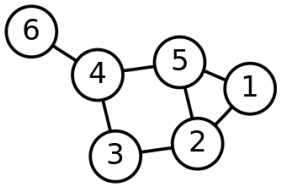
Binary search tree memiliki waktu akses, tambah, dan hapus yang tidak konstan, yaitu O(log n). Ketika penambahan dan penghapusan juga diperlukan operasi penyeimbangan.

Makalah ini akan membahas salah satu jenis implementasi map yang memiliki waktu akses, tambah, dan hapus yang konstan (O(1)), yaitu *hash trie.*

II. DASAR TEORI

Struktur data ini menggunakan beberapa konsep matematika diskrit dan struktur data, yang dijelaskan di bab ini.

A. Graf

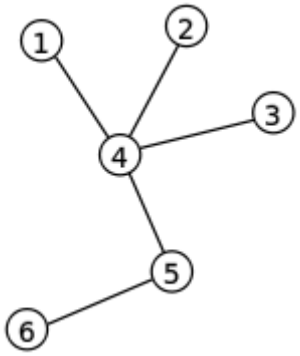


Ilustrasi graf. Sumber: https://commons.wikimedia.org/wiki/File:6n-graf.svg

Graf adalah gabungan dari himpunan simpul dan himpunan sisi. Simpul adalah objek-objek yang saling dihubungkan oleh sisi.

Graf memiliki beberapa variasi yang memiliki sifat-sifat khusus. Salah satu dari variasi tersebut adalah pohon.

B. Pohon

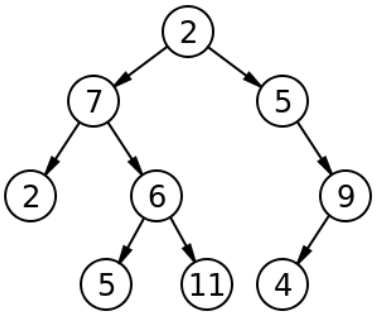


Ilustrasi pohon. Sumber: https://commons.wikimedia.org/wiki/File:Tree_graph.svg

Pohon adalah graf terhubung yang tidak memiliki sirkuit. Hal ini berarti hanya ada satu jalan di antara dua simpul sembarang.

Pohon memiliki beberapa sifat yang membuat mereka sering digunakan dalam algoritma.

C. Pohon berakar



Ilustrasi pohon berakar. Sumber: https://en.wikipedia.org/wiki/File:Binary_tree.svg

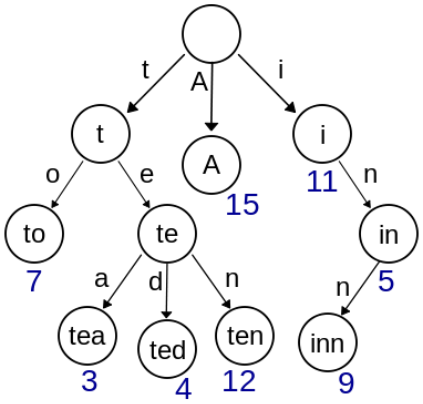
Pohon berakar adalah salah satu variasi dari pohon. Pohon berakar memiliki satu simpul yang ditentukan sebagai akar. Semua sisi pada pohon berakar adalah sisi berarah yang menjauhi akar.

D. Pohon n-ary

Pohon n-ary adalah pohon berakar yang setiap simpulnya memiliki paling banyak n anak. Contohnya, setiap simpul pada pohon biner memiliki paling banyak dua anak.

Pohon n-ary adalah jenis pohon yang sering dipakai dalam algoritma karena kemudahan implementasi dan sifat-sifatnya.

E. Trie



Ilustrasi Trie. Sumber: https://commons.wikimedia.org/wiki/File:Trie_example.svg

Trie adalah sebuah pohon n-ary terurut dengan beberapa sifat khusus. Setiap subpohon dari suatu *trie* menyimpan data dengan prefiks yang sama.

Sifat ini sangat membantu pencarian, penambahan, dan penghapusan data pada *trie*. Operasi-operasi tersebut dapat dilakukan secara rekursif. Jumlah operasi yang harus dilakukan dalam pencarian, penambahan, atau penghapusan akan dibatasi dengan ukuran *key*, dan tidak tergantung dengan jumlah elemen.

F. Hash

Hash adalah sebuah fungsi yang mengubah data menjadi data lain yang ukurannya konstan. Misalnya sebuah string yang bisa berukuran apa saja dapat diubah oleh fungsi *hash* menjadi data yang besarnya 64 bit.

Sebuah fungsi *hash* yang akan digunakan dalam implementasi struktur data harus memiliki beberapa sifat penting. Fungsi *hash* tersebut harus cepat dihitung dan tersebar merata.

Kecepatan perhitungan fungsi hash sangat penting karena fungsi ini akan dipanggil berulang kali. Hal ini berbeda dengan fungsi *hash* yang ditujukan untuk kriptografi, seperti SHA. Karena itu, biasanya algoritma hash yang digunakan untuk struktur data dan algoritma hash yang digunakan untuk kriptografi dipisahkan, karena penggunaan yang salah bisa berdampak negatif terhadap kualitas program.

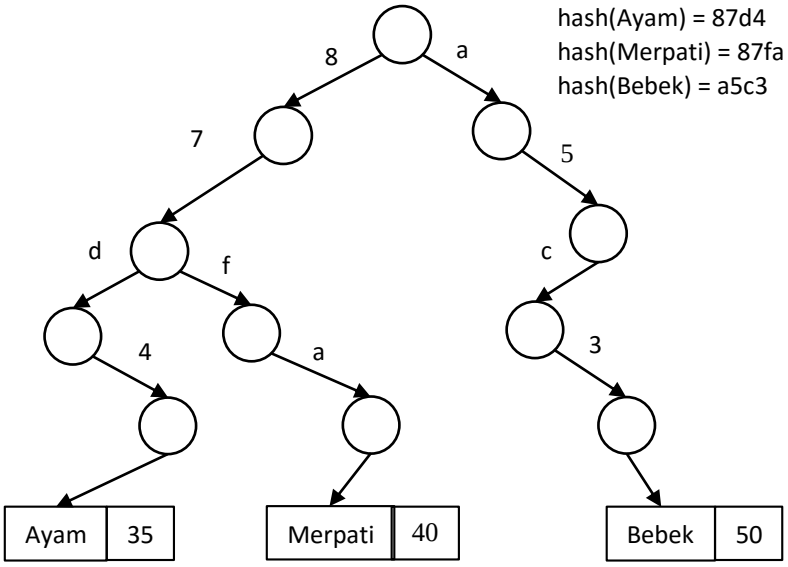
Fungsi *hash* juga harus tersebar merata, karena kebanyakan algoritma yang bekerja dengan *hash* memiliki kinerja paling baik ketika semua data menghasilkan *hash* yang berbeda.

III. ANALISIS

A. Deskripsi

Hash trie adalah *trie* yang menyimpan data menggunakan *hash* dari *key*-nya. *Hash* digunakan agar tinggi *trie* dapat diprediksi dan nilai-nilai yang disimpan tersebar merata.

Contoh *hash trie* adalah seperti di gambar berikut:



Ilustrasi hash trie.

Pada gambar di atas, sebuah *hash trie* digunakan untuk memetakan nama hewan kepada sebuah bilangan bulat. Dapat dilihat bahwa cara kerja *hash trie* sama persis dengan *trie* biasa. *Hash trie* yang terlihat di atas merupakan pohon 16-ary, dan setiap sisi menyimpan satu karakter heksadesimal.

B. Kompleksitas Algoritma

Pada *hash trie* yang optimal, waktu akses, tambah, dan hapus bisa mendekati O(1). Dapat dilihat bahwa waktu ketiganya hanya tergantung dengan ukuran hash yang digunakan. Karena ukuran hash yang digunakan adalah konstan, maka waktu aksesnya juga konstan.

Pada contoh di atas, pencarian, penambahan, atau penghapusan data hanya perlu menelusuri pointer sebanyak empat kali, tidak peduli berapa banyak data yang telah disimpan.

Salah satu implementasi hash trie yang sudah dioptimalkan berhasil mendapatkan waktu akses yang mendekati O(1), dan dengan kecepatan dua kali lipat dari hash table, seperti yang ditunjukkan pada tabel berikut.

SetSize	HAMTC	HAMTL	Hash2K	Hash64K	Hash512K
8K	0.82	0.75	1.00	1.00	1.00
16K	0.81	0.88	1.44	0.94	1.00
32K	0.93	1.00	2.53	1.03	0.94
64K	1.06	1.13	4.66	1.19	0.97
128K	1.29	1.42	9.07	1.63	1.16
256K	1.06	1.29	19.27	1.70	1.18
512K	1.16	1.34	37.90	2.13	1.49
1024K	1.26	1.41	77.32	3.27	1.46
2048K	1.37	1.52	165.00	5.83	2.04
4096K	1.45	1.63	347.00	11.28	2.49
8192K	1.35	1.73	–	–	–

Perbandingan waktu pencarian *hash trie* (HAMTC dan HAMTL) dan *hash table* (Hash2K, Hash64K, dan Hash512K) dengan jumlah elemen hingga delapan juta elemen. Waktu dengan tanda garis menandakan keterbatasan memori sistem.
Sumber: Bagwell, P., Ideal Hash Trees, 2000.

C. Kelemahan

Seperti struktur data lain yang memanfaatkan hash, hash trie juga rentan terhadap *hash collision*, yaitu kondisi di mana beberapa data memiliki nilai hash yang sama. Namun, hal ini sangat jarang terjadi sehingga dapat diabaikan dalam kebanyakan kasus.

Struktur *hash trie* juga akan menggunakan lebih banyak memori dibandingkan dengan *hash table*. Hal ini disebabkan ukuran pointer yang banyak digunakan *trie* (8 byte per pointer pada sistem 64-bit).

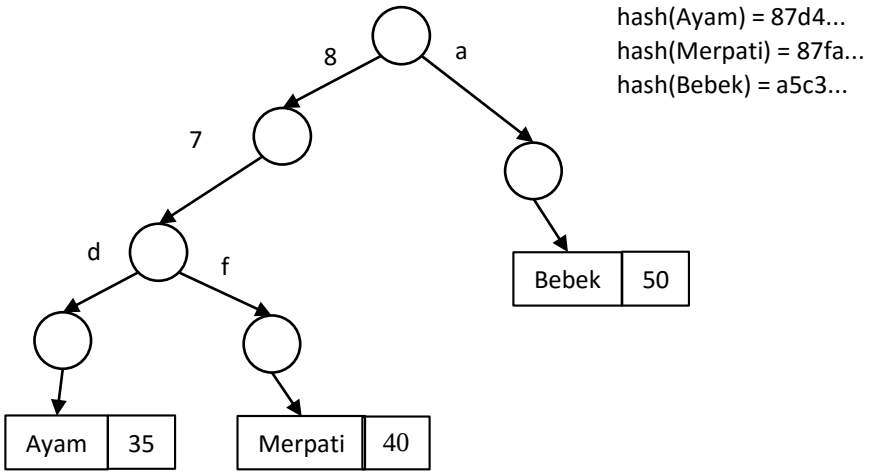
D. Pemutakhiran

Beberapa kelemahan dari *hash trie* dapat dikurangi dengan beberapa perubahan ke struktur data ini.

Masalah *hash collision* dapat ditangani dengan

menggunakan fungsi hash yang bisa menambah ukuran hash seperlunya. Misalnya, dua buah *key* memiliki hash yang sama ketika ukuran hash adalah 4 bit. Maka digunakan hash 8 bit.

Bila *hash trie* diatur untuk menggunakan ukuran hash yang paling kecil, hal ini juga bisa mengurangi penggunaan memori. Berikut ilustrasi struktur data *hash trie* yang telah dimodifikasi:



Ilustrasi hash trie yang telah dimodifikasi. Empat node berhasil dihilangkan untuk menghemat memori. Fungsi hash yang digunakan dapat menghasilkan hash yang lebih panjang jika diperlukan program.

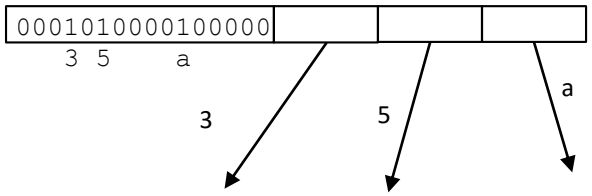
Penghematan memori lebih lanjut dapat didapatkan dengan melihat cara penyimpanan pointer ke subpohon pada *trie*.

Berikut adalah contoh kode program C yang mengimplementasikan *hash trie* ini:

```
typedef struct NodeHashTrie_t {
    NodeHashTrie* children[16];
    Data* data;
} NodeHashTrie;
```

Contoh kode program yang mengimplementasikan *hash trie*.

Penyimpanan anak seperti ini tidak efisien, karena banyak pointer yang akan bernilai 0 (null pointer). Cara yang lebih baik adalah dengan menyimpan pointer mana saja yang tidak bernilai 0 dalam sebuah bitmap, seperti yang ditunjukkan pada ilustrasi berikut:



Ilustrasi penggunaan bitmap untuk menghemat memori.

Kode program yang diperbaharui akan terlihat seperti berikut:

```
typedef struct NodeHashTrie_t {
    NodeHashTrie** children;
    Uint16_t bitmap;
    Data* data;
} NodeHashTrie;
```

Contoh kode program yang mengimplementasikan *hash trie* dengan penghematan memori menggunakan bitmap.

Dengan perubahan-perubahan tersebut, sifat boros memori dari *trie* dapat ditanggulangi.

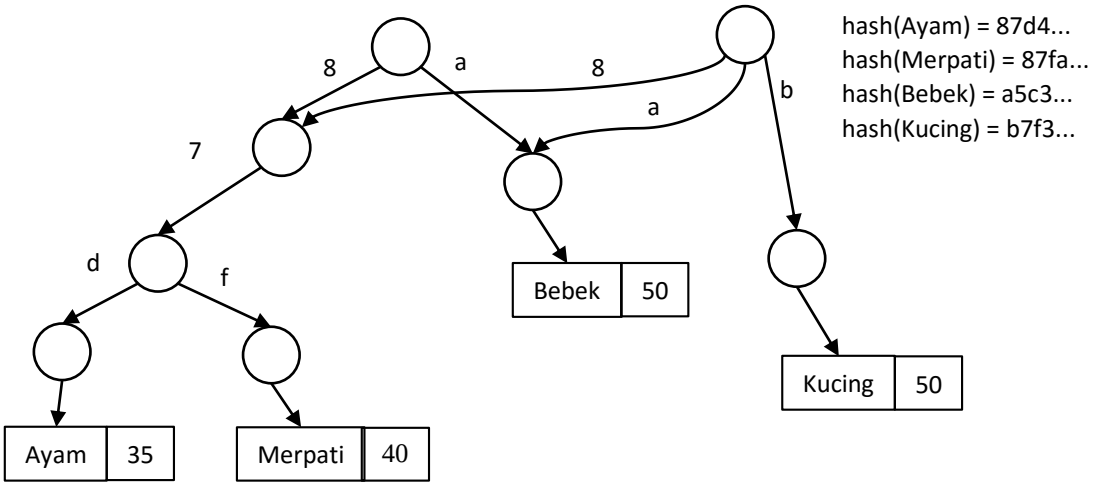
IV. Penggunaan

Hash trie merupakan struktur data yang digunakan untuk mengimplementasikan *map* pada *library* Immutable.js yang dikembangkan oleh Facebook. Hash trie juga digunakan dalam bahasa pemrograman Haskell.

Implementasi *hash trie* juga tersedia pada banyak bahasa pemrograman lainnya.

Hash trie memiliki beberapa sifat yang membuatnya mudah digunakan sebagai struktur data yang *immutable* (tidak dapat diubah). Hal ini disebabkan oleh strukturnya yang berdasarkan struktur pohon.

Contoh sifat ini adalah sebagai berikut. Misalkan hewan baru ingin ditambahkan ke contoh di atas, tetapi tanpa mengubah struktur data asli. Hal ini dapat dilakukan dengan penggunaan memori yang minim sebagai berikut:



Ilustrasi dua hash trie. Keduanya bisa digunakan secara independen, selama struktur *hash trie* tersebut didefinisikan *immutable*.

Dapat dilihat bahwa penambahan hewan baru ke *hash trie* tersebut hanya membutuhkan tambahan dua *node*. *Hash trie* yang sebelumnya tidak berubah. Hal ini hanya dapat dilakukan jika struktur data ini dideklarasikan *immutable*.

VII. Ucapan Terima Kasih

Penulis memanjatkan puji dan syukur kepada Allah SWT atas segala rahmatnya sehingga makalah ini dapat diselesaikan. Penulis juga mengucapkan terima kasih kepada dosen pengajar Matematika Diskrit, Ibu Harlili dan Bapak Rinaldi Munir untuk semua pengetahuan yang telah diberikan mengenai semua aspek Matematika Diskrit, dan khususnya tentang teori Graf dan Pohon yang mendasari makalah ini.

VIII. Daftar Pustaka

[1] Munir, Rinaldi. "Matematika Diskrit". Informatika, Bandung: 2010.

[2] Bagwell, Phil. "Ideal Hash Trees". 2000.

[3] Tibell, Johan. "Announcing unordered-containers 0.2". Diakses 10 Desember 2015. http://blog.johantibell.com/2012/03/announcing-unordered-containers-02.html

[4] Byron et al. "Immutable.js". Diakses 10 Desember 2015. https://facebook.github.io/immutable-js/

Pernyataan

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 10 Desember 2015

M. Isham Azmansyah F.
13514014