Pengenalan *Trie* dan Aplikasinya

Reinhard Denis Najogie - 13509097

Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13509097@std.stei.itb.ac.id

ABSTRAK

***Trie* adalah struktur data berupa pohon terurut untuk menyimpan suatu himpunan string dimana setiap node pada pohon tersebut mengandung awalan (prefix) yang sama, karena itulah *trie* disebut juga *pohon prefix*. *Trie* sering digunakan pada masalah komputasi yang melibatkan penyimpanan dan pencarian string. *Trie* memiliki sejumlah keunggulan dibanding struktur data lain untuk memecahkan masalah serupa terutama dalam hal kecepatan dan memori yang digunakan. Makalah ini akan membahas *trie*, implementasinya, aplikasinya dalam kehidupan sehari-hari, serta keunggulannya dibandingkan struktur data lain untuk permasalahan serupa.**

***Kata Kunci*: pohon, prefix, string, trie.**

1. PENDAHULUAN

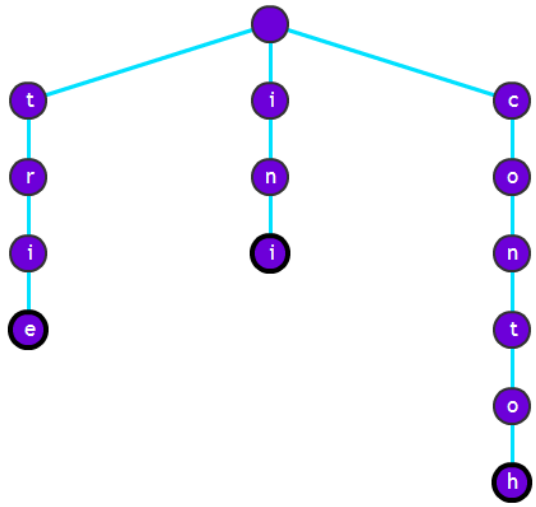
Salah satu permasalahan komputasi yang sering muncul adalah masalah pengolahan string. Banyak algoritma dan struktur data yang telah ditemukan untuk merepresentasikan dan mengolah string. Pengolahan string sendiri terdiri dari berbagai macam rutinitas, diantaranya pencarian sub-string, perbandingan string, sub deret terpanjang, dan lain-lain. Jika yang kita olah adalah himpunan atau kumpulan string maka akan ada lebih banyak lagi rutinitas yang kita perlukan, diantaranya pencarian suatu string serta srtuktur data untuk menyimpan himpunan string tersebut.

Kata "trie" berasal dari kata retrieval yang berarti pengambilan. Struktur data trie ditemukan oleh seorang professor di MIT bernama Edward Fredkin.

Permasalahan pengolahan string biasanya tidak berupa permasalahan pengolahan string murni seperti yang ada di bangku kuliah, tetapi lebih ke aplikasi string pada permasalahan kehidupan sehari-hari. Contoh aplikasi pengolahan string misalnya masalah deret DNA pada ilmu biologi, penamaan molekul pada ilmu kimia, penyimpanan dan pencarian kata-kata dalam kamus, pengecekan plagiarisme dan lain-lain.

Beberapa struktur data yang sering digunakan untuk mengolah string antara lain array of string, tabel hash, pohon pencarian biner, dan trie. Untuk penggunaan biasa dan dalam skala kecil, akan lebih mudah menggunakan array of string biasa untuk menyimpan himpunan string dan menggunakan algoritma pencarian biner untuk melakukan pencarian string. Cara ini memang terlihat sederhana, tetapi untuk skala dan keperluan yang lebih besar, struktur data ini tidak cukup mangkus baik secara waktu maupun ruang. Oleh karena itu, struktur data yang lebih sering digunakan untuk menyimpang himpunan string untuk skala lebih besar adalah binary search tree, hash table, serta trie. Dalam makalah ini kita akan menggunakan terminologi yang ada pada graf dan pohon secara langsung untuk trie.

Dalam trie, tidak ada node yang menyimpan kunci yang terkait dengan node tersebut, sebaliknya, posisinya di pohon menunjukkan kunci apa yang terkait dengannya. Setiap keturunan dari sebuah node memiliki prefix yang sama dengan string yang diwakilkan oleh node tersebut, dan akar menandakan sebuah string kosong.



Gambar 1 – Trie dengan kata "trie","ini", dan "contoh"

Dari gambar di atas kita seharusnya sudah mendapat gambaran mengenai kompleksitas waktu yang diperlukan untuk mencari sebuah kata dalam trie. Ya, jika panjang kata terpanjang dalam trie adalah L, maka untuk mencari sebuah kata dalam trie memerlukan waktu terburuk $O(L)$

Makalah ini secara khusus membahas struktur data trie, implementasinya, dan aplikasinya. Akan dibahas fungsi-fungsi apa saja yang diperlukan ketika kita mengimplementasikan sebuah trie. Aplikasi trie yang dibahas adalah pengunaannya sebagai struktur data

kamus, trie sebagai alternatif struktur data pohon biner atau tabel hash, serta penerapannya dalam masalah kesamaan prefix terpanjang.

2. TRIE

2.1 IMPLEMENTASI TRIE

Struktur data trie bisa diimplementasikan dengan banyak cara, beberapa diantaranya bisa dibuat untuk dapat mencari sejumlah kata di dalam sebuah kamus dimana setiap kata bisa berbeda sedikit dengan kata yang dicari, sementara implementasi lain dari trie hanya memberikan kita kata yang persis sama dengan kata yang ingin kita cari. Implementasi trie yang akan kita bahas dalam makalah ini hanya akan mencari kata yang persis sama dengan kata yang dicari dan menghitung jumlah kata yang memiliki sebuah prefix tertentu. Implementasi disajikan menggunakan pseudocode untuk memudahkan pengguna mengerti terlepas dari apapun bahasa pemrograman yang digunakannya

Kita akan mengimplementasikan 3 fungsi berikut:

- addWord. Fungsi ini akan menambahkan string **word** ke dalam kamus.
- countPrefixes. Fungsi ini akan menghitung jumlah kata yang mengandung string **prefix** sebagai prefixnya.
- countWords. Fungsi ini akan menghitung jumlah kata di kamus yang persis sama dengan masukan string **word**.

Kita juga perlu mengimplementasikan sebuah struktur yang mengembalikan nilai yang disimpan pada setiap node. Karena kita ingin mengetahui jumlah kata yang sesuai dengan string yang diberikan, setiap node harus memiliki elemen yang menandakan bahwa node ini merepresentaskan sebuah kata ataukah hanya sebuah prefix (agar lebih sederhana, sebuah kata yang lengkap kita anggap sebagai prefix juga) dan berapa banyak kata di kamus tersebut yang direpresentasikan oleh prefix tersebut (dalam kamus asli tidak ada kata yang berulang, namun demikian pada masalah komputasi seringkali kita harus membedakan dua kata yang sama). Hal ini dapat dilakukan dengan menambahkan satu elemen integer kata.

Karena kita ingin mengetahui jumlah kata yang memiliki prefix yang sama dari sebuah inputan string, kita memerlukan elemen integer **prefixes** yang menyimpan berapa banyak kata yang memiliki prefix node tersebut. Lalu, setiap node harus memiliki refrensi ke setiap kemungkinan anaknya (ada 26 referensi, sesuai banyak alphabet). Setelah mengetahui detil informasi ini, struktur trie kita akan terlihat seperti ini:

```
structure Trie
  integer words;
  integer prefixes;
  reference children[26];
```

Dan fungsi-fungsi yang kita perlukan antara lain:

```
initialize(node)
addWord(node, word);
integer countPrefixes(node, prefix);
integer countWords(node, word);
```

Pertama-tama, kita perlu menginisialisasi node-node dengan fungsi berikut:

```
initialize(node)
  node.words=0
  node.prefixes=0
  for i=0 to 26
    children[i]=NoChildren
```

Fungsi addWord mempunyai dua parameter, node tempat kata akan ditambahkan dan kata yang akan ditambahkan. Idenya adalah ketika string word ditambahkan ke sebuah node, kita akan menambahkan word sebagai cabang dari node, dengan cara memotong karakter paling kiri dari word. Jika cabang yang diperlukan tidak ada, kita harus membuatnya.

```
addWord(node, word)
  if isEmpty(word)
    node.words=node.words+1
  else
    node.prefixes=node.prefixes+1
    k=firstCharacter(word)
    if(notExists(children[k]))
      children[k]=createChildren()
      initialize(children[k])
    cutLeftmostCharacter(word)
    addWord(children[k], word)
```

Fungsi countWords dan countPrefixes sangat mirip. Jika kita menemukan string kosong kita hanya perlu mengembalikan jumlah kata/prefix yang tercatat di node tersebut. Jika kita mencari string yang tidak osong, kita harus mencarinya di cabang yang tepat dari trie, tapi jika tidak ada cabang yang tepat, kita hanya perlu mengembalikan nilai 0.

```
countWords(node, word)
  k=firstCharacter(word)
  if isEmpty(word)
    return node.words
  else if notExists(children[k])
    return 0
  else
    cutLeftmostCharacter(word)
    return countWords(children[k], word);

countPrefixes(node, prefix)
  k=firstCharacter(prefix)
  if isEmpty(word)
    return node.prefixes
  else if notExists(children[k])
    return 0
```

```
else
    cutLeftmostCharacter(prefix)
    return countWords(children[k], prefix)
```

2.2 ANALISIS KOMPLEKSITAS

Pada bagian pendahuluan, pernah dibahas bahwa kompleksitas waktu trie untuk mencari dan menyisipkan elemen adalah linear, tapi kita belum melakukan analisisnya. Pada penyisipan dan pencarian perhatikan bahwa menambahkan kedalaman dari sebuah pohon bisa dilakukan dalam waktu konstan, dan setiap kali program menambah kedalaman dari pohon sebanyak satu level, satu karakter dipotong dari string. Kita bisa ambil kesimpulan bahwa setiap fungsi menurunkan kedalaman dari pohon sebanyak *L* dan setiap kali fungsi menurunkan kedalaman, bisa dilakukan dengan kompleksitas waktu O(*L*). Ruang yang digunakan oleh trie bergantung pada metode untuk menyimpan sisi dan berapa banyak kata yang memiliki prefix yang sama.

3. VARIASI TRIE

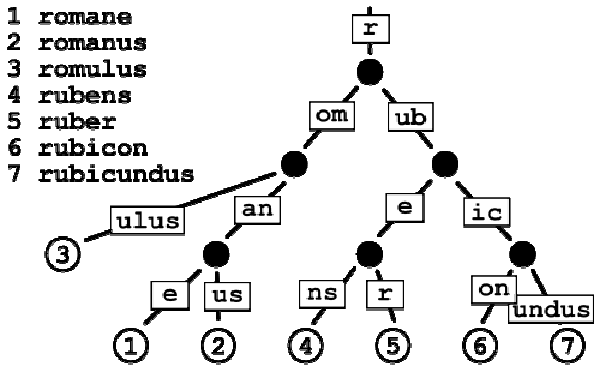
3.1 BITWISE TRIE

Bitwise trie memiliki banyak kesamaan dengan trie berbasis karakter biasa, kecuali dalam representasi denang bit individual yang biasanya digunakan untuk traversal secara efektif dan membentuk sebuah pohon biner. Secara umum, implementasinya menggunakan fungsi khusus CPU untuk dapat secara cepat mencari himpunan bit dengan panjang tertentu. Nilai ini lalu akan digunakan sebagai entri dari tabel dengan indeks 32 atau 64 yang menunujuk kepada elemen pertama dalam bitwise trie dengan sejumlah bilangan 0 di depan. Proses pencarian selanjutnya akan dilakukan dengan mengetes setiap bit dalam kunci dan memilih anak[0] atau anak[1] sesuai aturan hingga pencarian berakhir.

Walaupun proses ini mungkin terdengar lambat, tetapi sangat fleksibel karena kurangnya ketergantungan terhadap *register* dan oleh karena itu pada kenyataannya melakukan eksekusi dengan sangat baik pada CPU modern.

3.2 PATRICIA TRIE

PATRICIA merupakan singkatan dari Practical Algorithm To Retrieve Information Coded In Alphanumeric. PATRICIA trie sendiri lebih dikenal dengan sebutan pohon radix. Pohon radix sendiri bisa diartikan secara sederhana sebagai trie yang kompleksitas ruanganya lebih eisien, dimana setiap node yang hanya memiliki satu anak digabung dengan anaknya tersebut. Hasilnya adalah setiap node paling dalam paling tidak memiliki 2 anak. Tidak seperti trie biasa, anak bisa diberi label deretan karakter maupun satu karakter. Ini membuat pohon radix jauh lebih efisien untuk jumlah string yang sedikit (terutama jika stringnya cukup panjang) dan untuk himpunan string yang memiliki prefix sama yang panjang.



Gambar 2 – Pohon radix dengan 7 kata dengan prefix "r"

Pohon radix memiliki fasilitas untuk melakukan operasi-operasi berikut, yang mana setiap operasinya memiliki kompleksitas waktu terburuk O(k), dimana k adalah panajng maksimum string dalam himpunan :

- Pencarian: Mencari keberadaan suatu string pada himpunan string. Operasi ini sama dengan pencarian pada trie biasa kecuali beberapa sisi mengandung lebih dari satu karakter.
- Penyisipan: Menambahkan sebuah string ke pohon. Kita mencari tempat yang tepat di pohon untuk menyisipkan elemen baru. Jika sudah ada sisi yang memiliki prefix sama dengan string masukan, kita akan memisahkan nya menjadi dua sisi dan memprosesnya. Proses pemisahan ini meyakinkan bahwa tidak ada node yang memiliki anak lebih banyak dari jumlah karakter string yang ada.
- Hapus: Menghapus sebuah string dari pohon. Pertama kita menghapus daun yang berkaitan. Lalu, jika orangtuanya hanya memiliki satu anak lagi, kita menghapus orangtuanya dan menggabungkan sisi yang saling terhubung
- Cari anak: Mencari string terbesar yang lebih kecil dari string masukan, sesuai dengan urutan alfabet.
- Cari orangtua: Mencari string terkecil yang lebih besar dari string masukan, sesuai dengan urutan alfabet.

Pengembangan yang umum dari pohon radix yaitu menggunakan node dua warna, hitam dan putih. Untuk mengecek apakah sebuah string masukan sudah ada di dalam pohon, pencarian dimulai dari puncak, dan terus menelusuri setiap sisi sampai tidak ada lagi jalan. Jika node akhir dari proses ini berwarna hitam, berarti pencarian gagal, jika node berwarna putih berarti pencarian telah berhasil. Hal ini membuat kita bisa menambahkan string dalam jumlah banyak yang memiliki

prefix yang sama dengan elemen di pohon dengan menggunakan node putih, lalu menghapus sejumlah pengecualian untuk mengehemat memori dengan cara menambahkan elemen string baru dengan node hitam.

4. APLIKASI TRIE

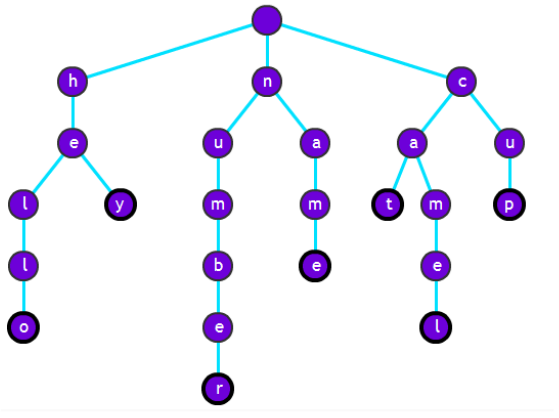
4.1 KAMUS

Kamus terdiri dari kumpulan kata-kata yang sudah terurut menaik berdasarkan urutan alfabet. Dalam perkembangannya saat ini sudah banyak kamus yang hadir dalam bentuk perangkat lunak, yang bisa digunakan di komputer ataupun di telepon genggam. Kamus dalam bentuk perangkat lunak tentunya memiliki fitur-fitur yang memudahkan pengaksesannya, antara lain pencarian kata dan penambahan kata ke kamus.

Implementasi struktur data untuk kamus akan serupa dengan fungsi-fungsi yang telah diimplementasikan pada bab 2.1, dengan tambahan fungsi find yang berguna untuk mencari suatu kata tertentu atau kata di kamus yang mirip dengan kata yang ingin dicari. Fungsi find dapat dimplementasikan seperti berikut:

```
Find (node, key)
  for each char in key
    if char not in node.children
      return None
    else
      node = node.children[char]
  return node.value
```

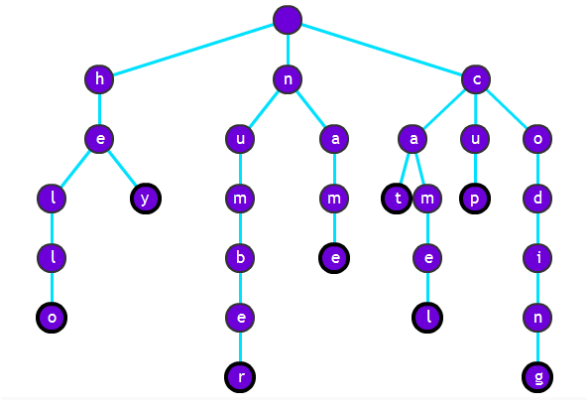
sementara itu untuk setiap node sendiri kita tambahkan elemen value yang menyimpan kata atau lebih tepatnya prefix yang diwakilkan oleh node tersebut.
Berikut ilustrasi kamus dengan struktur data trie



Gambar 3 – Trie dengan kata "hello","hey","number","name","cat","camel", dan "cup"

Bisa dibayangkan, ketika kita mencari sebuah kata dalam kamus, kita akan mulai dengan karakter pertama dari kata tersebut, jika tidak ada anak dari akar yang nilainya sama dengan karakter itu, maka langsung disimpulkan kata tidak ada di kamus. Jika ada, akan ditelusuri terus sampai ke dasar dari trie, jika kata ketemu, maka akan di kembalikan info dari node itu, sedangkan jika tidak ketemu kita bisa mengembalikan kata yang memiliki prefix sama dengan kata yang dicari sebagai saran pencarian.

Sementara itu penambahan kata akan berakibat penambahan cabang baru bila kata itu belum ada sebelumnya.



Gambar 4 – Penambahan kata "coding" pada gambar 2

Sistem yang serupa seperti pada kamus juga dipakai pada auto-complete pada address bar browser atau search suggestion pada google.

4.2 PENGGANTI STRUKTUR DATA LAIN

Trie sebagai turunan dari pohon memiliki keunggulan dibandingkan stuktur data yang sering digunakan untuk persoalan yang sama, yakni pohon pencarian biner dan tabel hash.

Berikut keunggulan utama trie dibandingkan pohon pencarian biner:

- Pencarian kunci dengan trie lebih cepat. Mencari sebuah kunci dengan panjang *m* menghabiskan waktu dengan kasus terburuk O(*m*). Pohon pencarian biner melakukan O(log(*n*)) perbandingan kunci, dimana n adalah jumlah elemen di dalam pohon, karena pencarian pada pohon biner bergantung pada kedalaman pohon, yang mana bernilai logaritmik terhadap jumlah kunci pencarian apabila pohonnya seimbang. Oleh karena itu pada kasus terburuk, sebuah pohon biner menghabiskan waktu O(*m* log *n*), yang mana pada kasus terburuk juga log *n* akan mendekati *m*. Operasi sederhana yang digunakan trie pada saat pencarian karakter, seperti penggunaan array index menggunakan karakter, juga membuat pencarian dengan trie menjadi lebih cepat.

- Trie menggunakan ruang lebih sedikit kita memuat string pendek dalam jumlah besar,

karena kunci tidak disimpan secara eksplisit dan node dipakai bersama oleh kunci yang memiliki prefix yang serupa.

- Trie bisa memiliki fitur untuk menghitung kesamaan prefix terpanjang, yang membantu untuk mencari pengunaan kunci bersama terpanjang dari karakter-karakter yang unik.

Berikut keunggulan utama trie dibanding tabel hash:

- Trie bisa melakukan pencarian kunci yang paling mirip hampir sama cepatnya dengan pencarian kunci yang tepat, sementara tabel hash hanya bisa mencari kunci yang sama tepat karena tidak menyimpan hunbungan antara kunci.

- Trie lebih cepat secara rata-rata untuk menyisipkan elemen baru dibandingkan dengan tabel hash. Hal ini terjadi karena tabel hash harus membangun ulang indeks nya ketika tabel ssudah penuh, yang mana menghabiskan waktu sangat banyak. Oleh karena itu trie memiliki kompleksitas waktu terburuk yang yang batasnya lebih konsisten, yang mana merupakan salah satu unsur penting pada jalannya sebuah program.

- Trie bisa diimplementasikan sedemikian sehingga tidak memerlukan memori tambahan. Tabel hash harus selalu memliki memori tambahan untuk menyimpan pengindeksan tabel hash.

- Pencarian kunci bisa jauh lebih cepat jika fungsi hashing dapat dihindarkan. Trie bisa menyimpan kunci bertipe integer maupun pointer tanpa perlu membuat fungsi hashing sebelumnya. Hal ini membuat trie lebih cepat daripada tabel hash pada hampir setiap kasus karena fungsi hash yang baik sekalipun cenderung overhead ketika melakukan hashing pada data yang hanya berukuran 4 sampai 8 byte.

- Trie bisa menghitung kesamaan prefix terpanjang, tetapi tabel hash tidak.

4.3 MENGHITUNG KESAMAAN PREFIX TERPANJANG

Kesamaan prefix terpanjang merupakan model permasalahan yang sering digunakan pada jaringan Internet Protocol (IP) untuk memilih entri dari tabel routing. Karena setiap entri dalam tabel routing dapat menentukan spesifikasi sebuah jaringan, satu alamat tujuan dapat memiliki kesamaan dengan lebih dari satu entri tabel routing. Isi tabel yang paling spesifik – yaitu yang memiliki subnet mask tertinggi - disebut kesamaan prefix terpanjang. Hal ini disebut demikian karena ia juga merupakan entri dimana jumlah bit terbesar alamat terkemuka di entri tabel sama dengan alamat tujuan.
Sebagai contoh, perhatikan tabel routing IPv4 berikut:

192.168.20.16/28
192.168.0.0/16

Ketika alamat 192.168.20.19 perlu dicari, kedua entri di atas cocok dengan tabel routing. Artinya, kedua entri mengandung prefix alamat yang dicari. Dalam kasus ini, prefix terpanjang dimiliki 192.168.20.16/28, karena subnet mask (/28) lebih tinggi daripada subnet mask yang lain (/16), membuatnya menjadi rute yang lebih spesifik.
Tabel routing sering mengandung default route, yang memiliki kesamaan prefix sependek mungkin, sebagai jalur kembali jika tidak ada entri yang memenuhi pencarian.

5. KESIMPULAN

Trie adalah struktur data yang sangat mangkus dalam menyelesaikan persoalan-persoalan yang berkaitan dengan string.

UCAPAN TERIMA KASIH

Terimakasih saya ucapkan kepada ilmuwan sains komputer yang telah menemukan struktur data trie dan aplikasinya. Tidak lupa saya ucapkan terimakasih juga kepada pengajar mata kuliah IF2091 Struktur Diskrit yakni Bapak Rinaldi Munir dan Ibu Harlili yang telah membagi ilmu struktur diskrit selama satu semester ini.

REFERENSI

[1] http://en.wikipedia.org/wiki/Trie - diakses tanggal 9 Desember pukul 20.32.
[2] http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=usingTries - diakses tanggal 9 Desember 2010 pukul 20.31
[3] http://en.wikipedia.org/wiki/Patricia_trie - diakses tanggal 9 Desember 2010 pukul 23.07
[4] Skiena, Steven S. *The Algorithm Design Manual*. New York: Springer, 1997, halaman 183.

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, 9 Desember 2010
ttd

Reinhard Denis Najogie/13509097