# STRUKTUR POHON SEIMBANG UNTUK JARINGAN *PEER-TO-PEER*

Muh Ikhsan Assaat – NIM : 13505042

*Program Studi Teknik Informatika*
*Sekolah Teknik Elektro dan Informatika*
*Institut Teknologi Bandung*
*Jl. Ganesha 10, Bandung*
E-mail : if15042@students.if.itb.ac.id

**Abstrak**

Makalah ini menjelaskan tentang adanya penemuan baru dalam teknologi *peer-to-peer* yang dapat menjawab masalah-masalah sebagai berikut. Dalam teknologi jaringan *peer-to-peer*, tidak ada struktur jaringan berbentuk pohon yang mangkus, alasan utamanya adalah bentuk umum struktur pohon membiarkan pengaksesan berada di sekitar akar dari pohon jaringan tersebut. Telah terdapat beberapa sistem yang menggunakan struktur pohon dengan beberapa modifikasi namun tetap mempunyai kekurang-kekurangan.

Makalah ini menawarkan suatu bentuk struktur pohon terbaru yang dapat mengatasi kelemahan-kelemahan dari struktur pohon untuk *peer-to-peer* yang sudah ada. Struktur ini dinamakan BATON (*Balanced Tree Overlay Network*). BATON menawarkan suatu struktur pohon seimbang sebagai lapisan dalam jaringan *peer-to-peer*. Walaupun struktur pohon menyebabkan terjadinya perbedaan antara tiap simpul di tiap tingkat-tingkat yang berbeda di dalam pohon, BATON memperlihatkan keluaran dari tiap simpul kurang lebih sama. Walaupun struktur pohon membuat seluruh simpul hanya mempunyai satu jalan, BATON menunjukan bahwa menggunakan rute samping dari tiap simpul membuat toleransi kesalahan yang cukup untuk di perbaiki secara mangkus. Secara khusus, untuk jaringan dengan N simpul, BATON menjamin masalah dapat dipecahkan dalam *O(log N)* langkah dan tiap operasi pembaharuan akan memakan *O(log N)* juga. Eksperimen telah dilaksanakan untuk memvalidasi proposal BATON.

## 1. Pendahuluan

### 1.1 Struktur Dasar Pohon

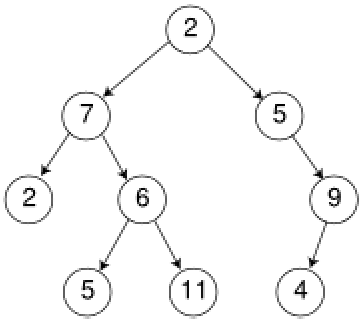
Dalam Ilmu komputer, pohon adalah struktur data yang sering dipakai. Pohon menyamai struktur pohon dengan beberapa simpul. Pohon adalah graf khusus. Tiap simpul bisa mempunyai **anak simpul (child)**, bisa juga tanpa anak simpul, yaitu simpul yang berada dibawah pohon itu sendiri. Pohon dalam struktur data tumbuh kebawah, bukan ke atas seperti pohon yang sebenarnya. Simpul yang mempunyai anak simpul disebut simpul **Orangtua (parent)**. Tiap simpul maksimal hanya mempunyai satu simpul orang tua. Simpul yang berada paling atas dinamakan **Akar (root)**. Akar tidak mempunyai simpul Orangtua karena akar adalah simpul paling atas. Simpul ini adalah tempat untuk memulai seluruh operasi. Seluruh simpul yang berada di pohon dapat dicapai dari akar, dan juga setiap simpul memiliki jalur yang unik.

Upapohon (subtree) adalah bagian dari pohon yang dianggap sebagai pohon sendiri. Seluruh simpul di pohon P, yang berada dibawahnya membentuk upapohon dari P. Setiap simpul dapat dinyatakan sebagai akar dari upapohon yang berada dibawahnya. Simpul yang berada di paling bawah dinamakan **Daun (leaf)**. Daun tidak mempunyai anak karena berada di paling bawah.

Ada 2 jenis dasar dari pohon. Yaitu, pohon tidak terurut dan pohon terurut. Pohon tidak terurut adalah pohon yang memenuhi definisi dasar struktur pohon. Sedangkan pohon terurut adalah pohon yang urutan anak-anaknya penting. Pohon terurut adalah struktur data pohon yang paling umum.

Melangkah ke tiap-tiap elemen dari pohon, yaitu dari hubungan antara orangtua dan anak, disebut 'berjalan di pohon' (*walking the tree*). Biasanya, operasi dilakukan dari suatu simpul. Berjalan mempunyai beberapa cara, yaitu dengan **pre-**

**order**, **in-order**, **post-order**. Pre-order adalah berjalan dengan melewati simpul orangtua lebih dahulu, lalu baru ke simpul-simpul anaknya. In-order adalah berjalan dengan melewati simpul anak pertama terlebih dahulu, lalu ke simpul orang tua, lalu kembali ke simpul-simpul anaknya yang tersisa. Post-order adalah berjalan dengan melewati simpul-simpul anaknya terlebih dahulu, lalu baru ke simpul orangtuanya.




**Gambar pohon sederhana**

Operasi umum yang dilakukan ke struktur pohon

- Menghitung tiap elemen
- Mencari suatu elemen
- Menambahkan elemen baru di posisi tertentu
- Menghapus elemen
- Memindahkan seluruh bagian dari pohon (pruning)
- Menambahkan seluruh bagian dari pohon (grafting)

### 1.2 Jaringan *Peer-to-Peer,* Jaringan berlapis, dan BATON

*Peer-to-peer* dalah sistem yang menjadi populer belakangan ini. Kekuatan utama dari sistem *peer-to-peer* adalah kemampuannya akan membagi sumberdaya file yang sangat besar, yang biasanya menggunakan *server* dapat diganti dengan komputer yang lebih sederhana. Tantangan terbesar dari membuat sistem *peer-to-peer* yang efektif adalah bagaimana melekatkan dengan erat komputer-komputer tersebuat menjadi sistem yang *cohesive*(erat). Biasanya masalah ini ditanggulangi dengan "*overlay network*", yaitu jaringan berlapis, yang digunakan untuk mengorganisasikan data-data yang terdapat di komputer-komputer (yang disimbolkan sebagai simpul-simpul dalam struktur pohon). Beberapa topologi yang disarankan untuk jaringan ini adalah *a ring,* dan *multi-dimensional grid*.

Dalam dunia *database*, struktur pohon banyak dipakai dan diapresiasi dengan baik. Tetapi, tidak ada jaringan berlapis yang ditawarkan menggunakan struktur pohon. Alasan dari situasi tersebut terjadi adalah bila di struktur pohon yang umum simpul yang dekat dengan akar akan sangat sering diakses, sedangkan simpul yang mendekat ke daun jarang diakses. Kondisi yang condong dalam hal ke-akses-an semacam ini tidaklah dinginkan oleh sistem *peer-to-peer*. Sedangkan makalah ini menawarkan suatu jaringan berlapis dengan struktur pohon untuk sistem *peer-to-peer* yang tidak menimbulkan yang sistem aksesnya tidak condong sebelah.

Jaringan berlapis yang ditawarkan berbasis struktur pohon biner seimbang yang tiap-tiap simpulnya diurus oleh satu *peer.* Tiap elemen *peer* mempunyai informasi jaringan, yaitu hubungan ke simpul orangtuanya, hubungan ke simpul anak kirinya, hubungan ke simpul anak kanannya, hubungan ke simpul yang berbatasan di kirinya, hubungan ke simpul yang berbatasan di kanannya, tabel rute kiri ke simpul yang dipilih yang berada di sebelah kirinya dan berada di level yang sama, dan tabel rute kanan ke simpul yang dipilih yang berada di sebelah kanannya dan berada di level yang sama. Struktur ini dinamakan BATON (*BAlanced Tree Overlay Network*). Dengan strukturnya yang berbentuk pohon biner, sistem ini mempunyai skalabilitas dan kekuatan yang sama dengan *B-tree.*

Ada beberapa cara yang dapat dipakai untuk pemeliharaan *range* dan *querying*, tapi cara tersebut berbasis jaringan lapis yang standar dan cukup kompleks. Dalam *P-tree* tiap simpul *peer-to-peer* disimbolkan sebagai simpul daun apada $B^+$*-tree*, dan tiap simpul mempunyai jalur dari indeks untuk akar sampai ke simpul daun. Simpul-simpul *peer-to-peer* ini didistribusikan dalam jaringan lapis CHORD. Pemeliharaan dari tiap jalur cukup mahal dan bila terjadi perubahan dari struktur pohon $B^+$*-tree* maka mudah terjadi ketidakakuratan. Walaupun begitu, kebanyakan jaringan, semacam CAN dan CHORD cukup bagus dalam performanya. Namun, mereka tidak efektif untuk mendukung partisi dan balikan data dengan basis jarak.

Dalam makalah ini berkontribusi :

- BATON adalah jaringan lapis untuk sistem *peer-to-peer* dengan struktur pohon biner seimbang. Konsekuensinya kesamaan yang tepat dan *queries* yang luas terdukung dengan mangkus.
- Saat simpul bergabung atau meninggalkan sistem, seperti halnya sistem *peer-to-peer* lain, sistem BATON melakukan log N langkah untuk mencari tempat untuk simpul bergabung atau mencari simpul untuk menggantikan simpul yang pergi. Walaupun begitu, untuk memperbaharui tabel rute hanya memakan O*(log N)*, yang ternyata lebih efisien dari sitem *peer-to-peer* lainnya yang biasanya memerlukan O$(log^2 N)$ untuk memperbaharui sebuah tabel rute. Perbedaan ini semakin signifikan untuk jaringan dengan jumlah N simpul yang besar.
- Menyeimbangi beban dan partisi *range*: BATON tidak memerlukan pengetahuan untuk data *range* tersimpan. Ia bisa mengatur tiap simpul dengan dinamik. Mekanisme yang sama memperbolehkan menyeimbangkan beban, dengan beban yang berlebihan dari suatu simpul kontennya ditransfer ke simpul lain.
- Toleransi kesalahan :
  Dari definisi, pohon mempunyai satu jalur unik untuk tiap-tiap simpulnya. Hubungan-hubungan tambahan yang tidak terlalu banyak disimpan ke dalam jaringan untuk menyediakan perbaikan yang mangkus bila terjadi kesalahan teknis untuk suatu simpul atau hubungannya itu sendiri. Intinya, BATON akan menunjukkan bahwa dengan adanya kesalahan dalam jumlah yang banyak, jaringan tetap terhubung.

Sisa dari makalah ini menjelaskan tentang pekerjaan-pekerjaan yang berhubungan, perkenalan dengan sistem arsitektur dan sistem operasi, serta kesimpulan.

## 2. Pekerjaan yang berhubungan

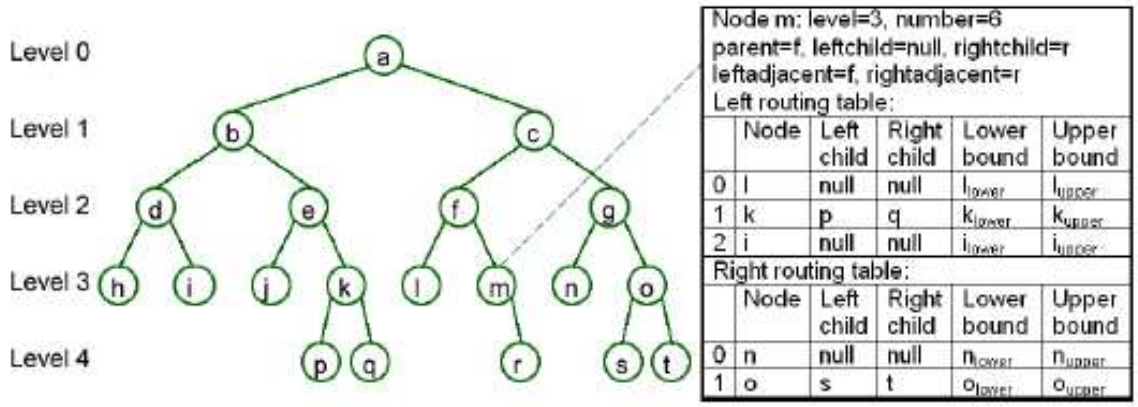
Paritisi data dan pencarian meliputi banyak tempat telah diteliti dengan baik dalam konteks distribusi data. Namun, strategi partisi dan pencarian tidak bisa diaplikasikan dengan penuh untuk distribusi sistem jaringan *peer-to-peer*. Alasannya adalah karena dalam sistem *peer-to-peer* tidak ada indeks global dan jaminan untuk sistem individualnya terhubung. Disini akan dibahas pekerjaan berhubungan yang *up-to-date* dalam sistem *peer-to-peer*.

CHORD, CAN, Pastry, dan Tapestry adalah empat dari sistem *peer-to-peer* terbaik yang pernah diketahui. Tiap-tiap sistem ini mengimplementasikan table *distributed hash*, yang sangat mangkus untuk *query* yang tepat tapi tidak terlalu cocok untuk *range queries*, karena *hashing* menghancurkan urutan dari data. Untuk menanggulangi hal ini, Gupta menawarkan sistem *peer-to-peer* berbasis *Hashing* sensitif lokal, yaitu range yang sama di-*hash* ke suatu *peer* yang sama dengan probabilitas yang besar. Namun, metode ini hanya untuk mendapatkan jawaban dengan aproksimasi. Cara yang lain adalah dengan mengikutsertakan *range* tersebut dalam fungsi *hash* yang ditawarkan oleh Sahin. Jadi, sistem dapat mengembalikan *superset* dari *range query.* Walaupun begitu, pencarian dengan tepat tidaklah mangkus, karena tidak dapat menjamin lokalnya sebuah data dan keseimbangan keluaran untuk seluruh sistem.

Pekerjaan yang paling mirip dengan BATON adalah P-Tree, P-Grid. P-Tree berbasis struktur pohon dan mengunakan CHORD sebagai arsitektur rute jaringan lapisannya. Tiap simpul dari simpulnya menyimpan jalur akar sampai daun yang paling kiri dari pohon $B^+$*-tree*-nya. Data hanya disimpan di simpul daunnya saja, dan simpul-simpul ini membentuk cincin CHORD. P-tree menjamin *logN* untuk *exact query* dan *range query.* Sewaktu suatu simpul bergabung dengan tambahan *logN* untuk pencarian predesesornya dan $log^2N$ untuk memperbaharui tabel rute, maka terdapat banyak usaha yang dilakukan untuk membuat cabang pohon untuk struktur pohon dari predesesornya tersebut. Lebih lagi, untuk mengecek kekonsistensiannya sebuah simpul yang baru tergabung memerlukan proses yang spesial yang berjalan secara periodik di simpul-simpul lain dalam sistem tersebut. Seperti sistem lain yang berbasis CHORD, performanya menurun seiring semakin condongnya sebuah data. P-Grid adalah sebuah struktur pohon biner prefix yang tiap simpulnya mempunyai referensi untuk simpul-simpul yang lain, yang mempunyai panjang $l$ untuk prefix yang sama, tapi dengan nilai posisi yang berbeda $(l+1)$ untuk kunci yang mereka tanggungkan. *Multiway Tree* juga lapisan yang

berstruktur pohon yang dimana tiap simpulnya diurus oleh tiap *peer* dan mempunyai hubungan ke simpul orangtuanya, simpul-simpul anaknya, simpul saudara kandungnya, dan simpul tetangganya. Struktur dari *Multiway tree* bukan dengan $B^+$*-tree* maupun pohon biner, adalah pohon dengan banyak lintasan tanpa batasan, dan suatu simpul boleh mempunyai anak sebanyak-banyaknya. Pencarian dilakukan dengan cara melompat dari simpul *query* le simpul yang mengandung jawaban dengan mengikuti hubungan tiap simpulnya satu demi satu. Karena setiap simpul hanya berhubungan dengan orangtua, saudara kandung, anak dan tetangganya, maka sistem ini rawan akan kesalahan atau kecelakaan. Dalam P-Grid dan *Multiway tree*, pohon tidak seimbang bila ada kecondongan, dan bahkan lebih parahnya bisa terjadi pohon menjadi struktur list linear yaitu tiap level hanya mempunyai satu simpul. Karena itu, proses pencarian tidak bisa dijamin hanya dengan *logN* langkah. Bila dibandingkan dengan struktur jaringan yang ada, BATON dapat menyesuaikan dengan sendirinya untuk kecondongan data dan 'keseimbangan ketinggian' dan mempertahankan rute vertikal dan horizontalnya untuk pencarian yang mangkus dan toleransi kesalahan.

## 3. Struktur BATON




Node m: level=3, number=6
parent=f, leftchild=null, rightchild=r
leftadjacent=f, rightadjacent=r
Left routing table:

| | Node | Left child | Right child | Lower bound | Upper bound |
|---|---|---|---|---|---|
| 0 | l | null | null | $l_{lower}$ | $l_{upper}$ |
| 1 | k | p | q | $k_{lower}$ | $k_{upper}$ |
| 2 | i | null | null | $i_{lower}$ | $i_{upper}$ |

Right routing table:

| | Node | Left child | Right child | Lower bound | Upper bound |
|---|---|---|---|---|---|
| 0 | n | null | null | $n_{lower}$ | $n_{upper}$ |
| 1 | o | s | t | $o_{lower}$ | $o_{upper}$ |

**(a) Arsitektur indeks pohon biner seimbang**

Struktur BATON adalah struktur pohon biner seimbang seperti pada gambar (a).

***Definisi 1*** **:** *Pohon seimbang jika dan hanya jika di setiap simpul dalam pohon, tinggi dari dua upapohon berbeda maksimal satu.*

Kita mengasosiasikan tiap simpul dalam pohon level dan angka. Level dari akar adalah 0, dan anak dari akar berada di level satu dan seterusnya. Level dari tiap simpul satu lebih besar dari level orangtuanya. Sehingga, level maksimal dari pohon adalah satu lebih kecil dari tinggi pohon tersebut. Di level *L*, paling banyak jumlah simpul dalam pohon biner adalah $2^L$. Kita berikan indeks angka dari kiri ke kanan dari 1 sampai dengan $2^L$, dan dalam setiap level, ada maupun tidaknya simpul disitu simpul tetap dinomori indeks angka yang sesuai. Dengan begitu, tiap simpul punya angka dan level untuk memberitahukan letak posisi mereka dengan presisi.

urutan linier dari simpul-simpul dalam pohon dan untuk tujuan ini kita menggunakan *in-order traversal* (lintasan *in-order*). Bila diberikan simpul *x*, makan kita simpulkan bahwa simpul sebelumnya adalah batas kiri simpul tersebut, dan simpul setelahnya adalah batas kanan simpul tersebut. Perhatikan bahwa batas-batas simpul bisa jadi beda level. Bahkan, dalam pohon lengkap setiap setiap simpul pengganti dalam traversal adalah simpul daun, dan setiap simpul pengganti yang lain adalah simpul interior. Bahkan saat pohon tidak lengkap, akan mudah untuk menunjukan bahwa tiap simpul interior harus memiliki paling tidak satu batas simpul yang bila bukan simpul daun dia adalah simpul interior dengan anak yang kurang dari dua.

Tiap simpul dalam pohon biasanya memberitahu letak dengan tepat satu *peer* simpul dalam sistem *peer-to-peer*. Tiap simpul *peer* mempunyai alamat IP atau ID jaringan yang berhubungan, yang bisa digunakan untuk mencari tahu letak simpul dan berkomunikasi dengannya. Maka,

kita berpikir bahwa tiap simpul mempunyai identitas yang logis, yaitu dari aturan level dan indeks angkanya, dan juga identitas fisik dari alamat IP-nya.

Tiap simpul dalam pohon mengandung hubungan ke simpul orangtuanya, anaknya, batas kanannya, batas kirinya, dan tetangga yang dipilih yang berada di level yang sama. Mempunyai hubungan ke orang tua, anak dan simpul batas kanan dan kiri dalam arti sederhananya mempunyai identitas fisik orangtua, identitas fisik anak kiri, identitas fisik anak kanan, identitas fisik batas kiri, dan identitas fisik batas kanan. Hubungan ke tetangga yang telah dipilih mempunyai arti yaitu mempunyai 2 tabel jalan samping yang spesial, yaitu tabel rute kiri dan tabel rute kanan. Masing-masing dari tabel ini berisikan hubungan ke simpul yang berada di level yang sama dan mempunyai indeks yang lebih kecil dari indeks simpul asalnya di pangkat 2. Elemen ber-indeks-kan j dalam tabel rute kiri di simpul ber-indeks-kan N mengandung hubungan ke simpul ber-indeks-kan $N - 2^{j-1}$ di level yang sama dalam pohon. Bila simpulnya tidak tersedia, maka entri tetap dilakukan di tabel rute, tapi ditandai dengan *null.* Tabel rute dinyatakan penuh bila tiap hubungan valid ke simpul tidak ada yang *null*. Sebagai contoh, perhatikan simpul *h* dalam gambar (a). Tabel rute kirinya tidak mempunyai hubungan yang valid, dan tabel rute kanannya berisi hubungan tetangga ke simpul *i, j* dan *l* yang berjarak $2^i$ simpul dari *h*. Struktur ini mempunyai kemiripan dengan CHORD, kecuali ini dalam garis lurus, bukan dalam bentuk lingkaran, dan entri tabel rute membawa informasi tambahan yaitu tidak hanya alamat IP, dan beberapa hubungan bisa berupa *null*.

***Teorema 1*** *: Pohon seimbang adalah pohon yang tiap simpulnya mempunyai satu anak, juga tabel rute kiri dan tabel rute kanannya penuh*

**Bukti :** Anggap penambahan simpul ke pohon seimbang. Biarkan simpul baru tersebut menjadi anak simpul dari simpul *x*. Biarkan simpul *x* ada di level *L*, dan simpul yang baru berada di level *L* + 1. Pohon yang dihasilkan bisa menjadi tidak seimbang bila leluhur dari simpul *x* yang mana saja, kedalaman dari upapohon kanan dan upapohon kiri berbeda lebih dari satu level akibat dari penambahan simpul. Anggap leluhur dari simpul *y* dari simpul *x* ada di level *i*, berarti *x* berada di upapohon kirinya *y*. Kedalaman dari upapohon bisa berubah dari *L* menjadi *L* + 1 akibat dari penambahan simpul. Tetapi, karena tabel rute kanan dari *x* penuh, maka pasti ada entri ke *i* dalam tabel untuk simpul dalam upapohon kanan *y*. dan simpul ini berada di level *L*. Maka, upapohon kanan dari *y* mempunyai kedalaman paling tidak di level *L*. dan perubahan dari kedalaman upapohon kiri menjadi *L* + 1 tidak mempengaruhi seimbang atau tidaknya pohon tersebut. Aplikasikan argumen tersebut di tiap leluhur dari *x* dalam pohon, maka kita bisa menyebutkan bahwa pohon tetap seimbang setelah terjadi penambahan simpul.

Sekarang anggap terjadi penghapusan dari pohon, yaitu simpul *v* yang adalah anak dari simpul *x* di level *L*. Penghapusan ini bisa menyebabkan tidak seimbangannya sebuah pohon, bila kedalaman dari upapohon *x* berubah dari *L* + 1 menjadi *L*, sedangkan upapohon yang lainnya mempunyai kedalaman *L* + 2. Anggap *x* berada di upapohon kiri dari *y*. Pasti ada simpul *z* di tabel rute kanan *x* yang berada di upapohon kanan dari *y*. Simpul *z* juga berada di level *L*. Anggap kedalaman dari upapohon kanan dari *y* adalah *L* + 1atau kurang, maka kita telah selesai, pohon tetap seimbang. Anggap kedalamannya adalah *L* + 2 atau lebih.

Kasus 1 : Ada simpul *v*, anak dari *z*, yaitu tabel rute kanan dari *u* dan mempunyai satu anak. Dari syarat dari teorema, penghapusan simpul *u* tidak diperbolehkan bila *z* mempunyai anak. Maka pohon tetap seimbang.

Kasus 2 : Tidak ada simpul *v*. Ini berarti tiap simpul yang berada di level *L* + 2 di dalam upapohon kanan dari *y*, mempunyai orangtua di level *L* + 1 yang mempunyai entri di dalam tabel rute tetangganya dari simpul w di upapohon kiri dari *y* yang berbeda dengan *u*. Simpul w berada di level *L* + 1, jadi kepergian dari simpul *u* tidak merubah kedalaman dari upapohon kiri dari *y*. Maka, pohon tetap seimbang.

Dari argumen di atas untuk tiap leluhur *y* dari *x*, kita menunjukan bahwa pohon seimbang tidak bisa di rusak oleh penghapusan simpul yang menjadi subjek kondisi teorema.

***Teorema 2 :*** *Bila suau simpul, misalkan x, berisikan hubungan ke simpul yang lain, misalkan y, dalam tabel rute kiri atau kanannya simpul orangtua dari x harus berisikan hubungan kesimpul orangtua dari y juga, kecuali*

*simpul orangtua dari x dan y adalah simpul yang sama.*

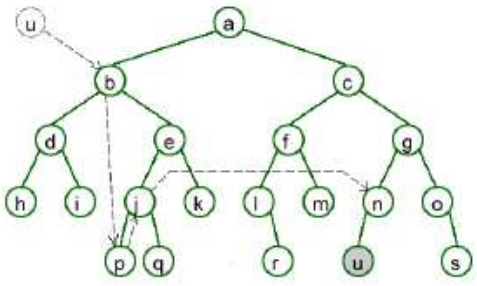
**Bukti :** anggap $N_x$ menjadi indeks angka untuk simpul *x* dan orangtua dari *x* adalah w. Tanpa mengurangi maknanya, buat *x* menjadi anak kanan dari w. Lalu $N_x$ dalah genap. Kita harus punya
$N_w = N_x / 2$. Dengan cara yang sama buat *y* anak dari *z*. Maka kita harus punya $N_z = N_y/2$ jika genap, dan $N_z = (N_y + 1)/2$ jika ganjil.

Kasus 1 : Anggap *y* paling tidak berjarak 2 dari *x*. Lalu $N_y = N_x \pm 2^k$ untuk integer $k > 1$. $N_y$ telah dijamin genap, bila $N_x$ genap. Maka $N_y = N_x \pm 2^{k-1}$ dalam arti *z* mempunyai hubungan dari w.

Kasus 2 : Anggap *y* berjarak 1 dari *x*, tapi bukan saudara kandung. Karena *x* adalah anak kanan dari orangtuannya, *y* harus menjadi tetangga kanan dari *x*. Maka $N_y = N_x + 1$, dan ganjil. Lalu kita hitung $N_z = (N_y+1)/2 = (N_x+2)/2 = N_w + 1$. Karena *z* mempunyai indeks yang lebih besar dari *w*, maka yang belakang pasti punya hubungan dari *w*.

Adalah mudah untuk melihat bahwa ada paling banyak *L* entri di tabel rute kiri di level *L*. Maka jumlah angka dari entri-entri adalah *O(log N)*.

### 3.1 Penggabungan Simpul




**(b) Suatu simpul bergabung kedalam jaringan**

Suatu simpul yang ingin bergabung ke dalam jaringan paling tidak harus tahu satu dari simpul yang berada dalam jaringan dan mengirimkan permintaan bergabung ke simpul tersebut. Ada dua fase dalam penggabungan simpul baru ke jaringan. Pertama adalah menentukan dimana simpul baru bergabung. Kedua adalah menggabungkannya kedalam jaringan di posisi yang telah dispesifikasi.

Saat suatu simpul menerima permintaan untuk bergabung, bila tabel rute kanan dan kiri simpul tersebut penuh ketika mempunyai kurang dari dua anak, maka simpul tersebut dapat menerima simpul baru yang meminta bergabung sebagai anaknya. Sebaliknya, permintaan tersebut harus di teruskan ke simpul yang lain seperti yang dijelaskan algoritma penggabungan yang diterangkan dibawah.

Sebagai contoh, anggap simpul *u* ingin bergabung ke jaringan dan ia mengirimkan permintaan gabung ke simpul *b* seperti di gambar (b). *b* lalu meneruskan permintaan tersebut ke p, yang menjadi simpul batasnya. Karena tabel rute dari p ttidak penuh maka permintaan tersebut diteruskan ke orangtuanya yaitu j. J mengecek tabel rutenya dan meneruskan permintaan ke simpul tetangga n, yang kebetulan tidak mempunyai anak yang cukup. Akhirnya n menerima *u* sebagai anaknya.

```
Algorithm: join(node n)
If (Full(LeftRoutingTable(n)) and
    Full(RightRoutingTable(n)) and
    ((LeftChild(n)==null) or (RightChild(n)==null))
   Accept new node as child of n
Else
   If ((Not Full(LeftRoutingTable(n))) or
       (Not Full(RightRoutingTable(n))))
      Forward the JOIN request to parent(n)
   Else
      m=SomeNodesNotHavingEnoughChildrenIn
          (LeftRoutingTable(n), RightRoutingTable(n))
      If (there exists such an m)
         Forward the JOIN request to m
      Else
         Forward the JOIN request to one of its
         adjacent nodes
      End If
   End If
End If
```

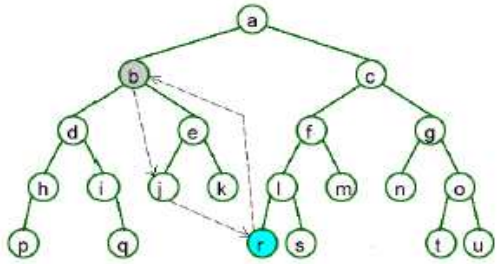
Setelah menganalisa algoritma ini, anggap bahwa hubungan dengan batasnya di-*traverse*-kan ke simpul daun w. Tidak peduli apakah w dapat menerima simpul baru menjadi anaknya, atau mempunyai tabel tetangga yang tidak lengkap. Dalam kasus sebelumnya, w meneruskan permintaan tersebut ke orangtuanya, yang bisa melokasi dari tabel teatangganya sebuah simpul *v*, yaitu adalah orangtua dengan tetangganya yang hilang dari w. Simpul *v* sekarang dapat menerima dari simpul baru sebagai anaknya, kecuali bila tabel tetangganya tidak penuh, dalam kasus ini permintaan akan diteruskan ke orangtuanya. Karena tinggi dari pohon adalah *O(logN)*, permintaan tidak bisa diteruskan lebih daari *O(logN)* kali. Semua arah

penerusan yang lain hanya menambah syarat konstan. Maka kita mempunyai loncatan dari *O(log N)* pesan untuk melokasikan tempat untuk simpul baru untuk bergabung. Labih dari itu, algoritma ini dispesialisasi untuk mencari simpul daun dan orangtua dari simpul yang memiliki tabel tetangganya yang tidak lengkap, yang semuanya semestinya adalah daun menurut teorema 1. Secara spesifik, simpul leluhur tidak pernah diperlukan dan tidak ada turut serta dari akar, tidak lebih dari hanya sekedar simpul biasa. Maka kita mengharapkan bahwa keluaran tidak diaplikasikan ke akar.

Sewaktu simpul *x* menerima simpul baru sebagai anaknya, maka konten darinya dibagi dua kepada anaknya. Dalam kata lain, *range* dari *x* menjadi terpartisi antara dia sendiri dengan anak barunya. Sebagai tambahan, bila *y* diterima sebagai anak kiri dari *x*, *x* juga mengirimkan hubungan dengan batas-batasnya yang menunjuk ke *z*, ke *y*, dan pembaharuan hubungan batas kirinya dengan *y*, lalu *y* membuat hubungan baas kirinya yang menunjuk ke *z*, dan hubungan batas kanannya menunjuk ke *x*, dan juga memberi tahu *z* bahwa *z* harus memperbaharui simpul batas kanannya dengan *y*, bukan dengan sebelumnya yaitu *x*. Sama halnya dengan bila *y* diterima sebagai anak kanannya *x*, maka hubungan batas kanan *x* di transfer ke *y*. Akhirnya, simpul *x* mengontak semua simpul tetangganya di tabel rute kiri dan kanannya, menyuruh mereka untuk memberi tahu anaknya yang relevan diberitahu tentang *y*, dan sebagai gantinya merespon dengan informasi dengan melihat anaknya yang relevan yang *y* akan butuhkan. Semua proses ini memerlukan *O(log N)* pesan dan *O(logN)* respon. Secara spesifik, *x* perlu mengirim maksimal $2L_1$ pesan ke simpul tetangganya, dimana $L_1$ adalah level dari *x*. Simpul-simpul tetangga ini perlu mengirimkan maksimal $2L_1$ kepada anaknya yang sebagai gantinya adalah mengirimkan $2L_2$ pesan untuk merespon ke simpul baru, dimana $L_2$ adalah level simpul baru. Simpul baru *y* perlu mengirimkan hanya satu pesan ke satu simpul batasnya. Maka dari itu, angka maksimal untuk jumlah pesan yang diperlukan untuk memperbaharui tabel rute adalah $2L_1+2L_2+2L_2+1<6$ *logN.*

### 3.2 Kepergian Simpul

Hanya simpul daun yang rela meninggalkan jaringan dan hanya kepergian mereka yang tidak akan mempengaruhi keseimbangan pohon. Dalam kasus yang lain, suatu simpul yang ingin meninggalkan jaringan harus mencari pengganti untuk dirinya sendiri, yaitu adalah simpul daun yang tidak memberikan efek terhadap keseimbangan pohon.




**(c) Kepergian simpul dari jaringan**

Bila simpul daun *x* ingin meninggalkan jaringan, dan tidak ada simpul tetangga yang dalam tabel rute dengan anak, maka *x* bisa meninggalkan jaringan tanpa memberikan efek terhadapa keseimbangan pohon. Karena persyaratan dari Teorema 1 masih dapat dipenuhi oleh simpul tetangganya. Dalam kasus ini, *x* harus mentransfer semua kontennya, dan *range* dari nilai indeksnya diambil alih oleh orangtuanya, adalah hubungan dengan batas kirinya bila dia adalah anak kirinya, dan adalah hubungan dengan batas kanannya bila dia adalah anak kanannya, dan mengirimkan pesan kepergiannya ke simpul-simpul tetangganya untuk mereka memperbaharui tabel rute masing-masing. Orangtua dari *x* setelah menerima konten dari *x* juga perlu mengirimkan pesan ke simpul tetangganya untuk memberi tahu mereka tentang konten barunya dan anaknya. Itu juga menjelaskan hubungan dengan simpul-simpul batas yang terkena efek untuk memperbaharui hubungan dengan batasnya yang berkorespondensi dengan dirinya sendiri. Maka, jumlah angka dari pesan yang dibutuhkan dalam kasus ini adalah $2L_1 + 2L_2 + 2 < 4$*log N,* dimana $L_1$ dan $L_2$ adalah level dari simpul orangtua *x* dan smipul *x*. Bila simpul daun ingin meninggalkan jaringan dan ada simpul tetangga dalam tabel rute dengan anak, maka ia perlu mencari simpul untuk mencari penggantinya dengan cara mengirimkan pesan cari-pengganti *(FINDREPLACEMENT)* ke simpul anak dari salah satu simpul tetangganya. Bila simpul bukan daun ingin meninggalkan jaringan, ia akan menemukan suatu simpul untuk menggantikannya dengan cara mengirimkan pesan cari-pengganti ke salah satu dari simpul batasnya yang adalah simpul daun atau sedalam

mungkin. Algoritma cari-pengganti dijelaskan dibawah.

Karena proses mencari simpul pengganti selalu ke bawah, proses memakan langkah paling banyak sebesar tinggi dari pohon yaitu *O(logN)*.

```
Algorithm: find replacement node (node n)
If (LeftChild(n)!=null)
  Forward the request to LeftChild(n)
Else If (RightChild(n)!=null)
  Forward the request to RightChild(n)
Else
  m=SomeNodesHavingChildrenIn
     (LeftRoutingTable(n), RightRoutingTable(n))
  If (there exists such an m)
    Forward the request to a child of m
  Else
    Come to replace the leave node
  End If
End If
```

Sebagai contoh, anggap simpul *b* di gambar (c). Bila ia ingin meninggalkan jaringan ia harus mencari simpul daun sebagai penggantinya. *b* membuat sebuah pesan cari-pengganti dan mengirimkan ke simpul batasnya yaitu *j*. *j* mengecek tabel rutenya dan menyadaari bahwa ada beberapa simpul tetangga dengan anak, sehingga *j* meneruskan pesan ke r, yaitu anak dari simpu ltetangga *j*. Di *r*, karena ia tidak punya anak, dan tidak ada simpul tetangga dengan anak, *r* bisa menggantikan *b* dengan aman. Seperti ilustrasi, BATON mengadaptasikan dirinya untuk kepergian simpul dan tetap mempertahankan keseimbangan tingginya.

Sebelum simpul *y* menggantikan *x* yang meninggalkan jaringan, ia perlu memberi tahu simpul-simpul tetangganya dan orangtuannya tentang kepergiannya seperti kasus sebelumnya, yaitu memakan 4*logN* langkah. Sebagai tambahan, semua simpul dengan hubungan ke *x* harus diberi tahu untuk menggnati identitas fisiknya (IP) untuk menunjuk ke *y* bukan ke *x*. Ini dengan mudah dilakukan dengan menggunakan informasi yang diterima dari *x*. Secara spesifik, orang tua aslinya dari simpul *x* (sekarang *y*) perlu mengirim $2L_1$ pesan kepada simpul-simpul tetangganya untuk memberitahu pergantian baru anaknyadi level $L_1$. *y* perlu memberikan $2L_2$ pesan ke simpul tetangga barunya, dimana $L_2$ adalah level baru dan 2 pesan untuk anak-anaknya dan 2 pesan untuk simpul batasnya. Maka, angka maksimal yang diperlukan untuk memperbaharui tabel rute untuk merefleksikan perubahan adalah sebanyak 8*logN*.

### 3.3 Kegagalan Simpul

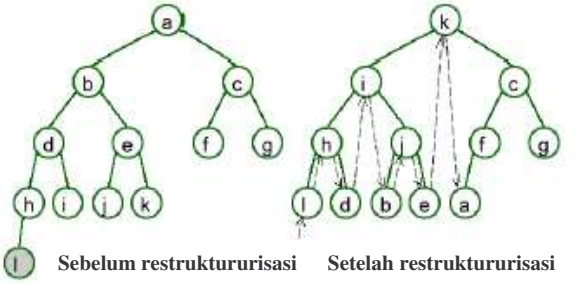
Kadang, simpul bisa gagal, atau hilang tiba-tiba. Dalm kasus ini, beberapa simpul ingin mengakses simpul yang hilang, misalkan *x*, akan mengetahui bahwa alamat yang dituju tidak dapat dicapai. Simpul ini mesti memberi laporan kegagalan ini ke simpul *y*, yaitu orangtua dari *x*, yang sekarang harus bertanggung jawab terhadap kepergian *x*. Simpul *y* menggunakan hubungan yang terdapat di tabel rute nya sendiri dan dengan cepat membentuk kembali tabel rute kiri dan tabel rute kanannya dari *x* dengan cara mengontak anak dari simpul dari tabel rutenya sendiri. Anak ini juga dapat membantu mencari tahu lokasi bila ada anak dari simpul *x*. Simpul *y*, orangtua dari *x* dapat menginisiasi kepergian untuk anak dari simpul *y* yang pergi tiba-tiba, mengikuti protokol yang dijelaskan di bagian sebelumnya. Karena informasi rute *x* telah dibuat kembali oleh *y*, maka algoritma yang dijelaskan diatas bisa dipakai dengan beberapa modifikasi.

### 3.4 Toleransi Kesalahan

Telah dijelaskan diatas bagaimana kesalahan atau kegagalan dan kepergian simpul yang tiba-tiba dapat diatasi dengan anggun. Operasi perbaikannya sama seperti kepergian simpul, hanya memerlukan *O(logN)* pesan, tapi memakan waktu yang bukan nol. Dalam bagian ini menunjukan bagaimana jaringan dapat terus beroperasi, sambil mengambil rute memutari simpul yang hilang.

Ada dua poros untuk memutarkan rute pesan dalam BATON : poros samping, melewati tabel rute kiri dan kanan, dan poros atas bawah, melewati orang tua, anak dan hubungan dengan batas-batas. Cara sebelumnya adalah toleransi kesalahan, karena ada algoritma perluasan hubungan yang mirip dengan Chord, dan maka dari itu banyak lintasan alternatif antara semua pasangan dari tiap simpul. Hal ini mengubah toleransi kesalahan karena simpul bisa pergi ke tetangga datri orangtua, mencari anak dari simpul tersebut dan kemudian menghubungkan kembali ke simpul anak, dengan demikian merekonstruksi hubungan antara orangtua dengan anak yang hilang. Sejauh ini hanya membahas kegagalan hanya di satu simpul saja.

Bila simpul yang gagal berjumlah dua atau lebih maka ada dua kemungkinan yang harus dipikirkan. Bila simpul yang gagal mempunyai hubungan oranguta-anak maka teknik diatas bisa



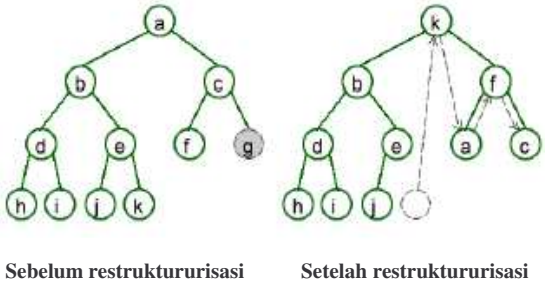

**(d) Restrukturisasi jaringan setelah simpul bergabung**

digunakan, yaitu berjalan melalui simpul tetangga. Bila simpul yang gagal tidak mempunyai hubungan orangtua-anak maka kegagalan mereka bisa dibenahi secara independen, dan tanpa komplikasi tambahan karena kegagalan sementara mereka yang serentak. Dalam kasus khusus, bahkan bila semua simpul dalam satu level gagal semua, pohon tersebut tidak terpartisi karena hubungan dengan batas simpul bisa digunakan sebagai rute untuk melewati jarak kosong tersebut. Kontras dengan kerapuhan dari sistem *multiway tree.*

### 3.5 Restrukturisasi Jaringan

Dari deskripsi di atas, menggabungkan simpul adalah memaksa keluar bagian pohon yang lain, sedangkan meninggalkan simpul harus menemukan penggantinya terlebih dahulu bila kepergiannya menyebabkan pohon menjati tidak seimbang. Kadang, sewaktu simpul bergabung adalah bagian daaari usaha menyeimbangkan keluaran., arahan seperti ini tidak diperbolehkan. Alternative lain adalah merestruktur sistem untuk mencapai keseimbangan.

Sewaktu simpul *x* menerima simpul *y* untuk bergabung sebagain anaknya dan mendeteksi pelanggaran dari Terorema 1, maka proses restrukturisasi dijalankan. Anggap restrukturisasi ini menuju ke kanan. Asumsikan *y* bergabung sebagai anak kiri dari *x*. Untuk membuat sistem menjadi seimbang kembali, *x* memberi tahu *y* untuk menggantikan posisinya, dan memberi tahu simpul batas kanannya yaitu *z* bahwa *x* akan menggantikan posisi *z, z* lalu mengecek simpul batas kananya, yaitu *t*, untuk melihat apakah anak kirinya kosong. Bila ia, dan tambah seorang anak ke *t* tidak memiliki efek untuk keseimbangan pohon, *z* mengambil posisi dari anak kiri *t* sebagai posisi barunya dan proses restrukturisasi berhenti. Bila anak kiri *t*




**(e) Restrukturisasi jaringan setelah simpul pergi**

penuh atau *t* tidak bisa menerima *x* sebagai anak kirinya tanpa merusak keseimbangan pohon, *z* mengambil posisi *t* sedangkan *t* memerlukan tempat baru untuk dirinya sendiri dengan cara melanjutkannya ke simpul batas kanannya. Anggap gambar (d) sebagai contoh. Anggap *l* bergabung dengan jaringan sebagai anak kiri dari *h*, dan penggabungan merusak keseimbangan pohon. Proses restrukturisasi diinisiasi di *h*, dimana *l* mengganti *h*, *h* mengganti *d*, *d* mengganti *i*, *i* mengganti *b*, *b* mengganti *j, j* mengganti *e*, *e* mengganti *k*, *k* mengganti *a*, dan akhirnya *a* menjadi anak kiri dari *f*, karena *f* dapat menerima satu anak tanpa membuat pohon menjadi tidak seimbang. Sekarang pohon menjadi seimbang.

Sewaktu simpul daun *x* meninggalkan jaringan dan membuat pohon menjadi tidak seimbang, orangtuanya yaitu *y* memulai proses restrukturisase. Anggap restrukturisasi condong kiri. Asumsikan bahwa *x* adalah anak kanan dari *x*. Untuk membuat pohon menjadi seimbang *y* harus menukar *x*, dan simpul batas kirinya yaitu *z* harus menukar *y*. Jika pergerakan *z* tidak mempengaruhi seimbangnya suatu pohon, proses restrukturisasi berhenti. Jika pergerakan *z* merusak keseimbangan pohon maka digunakanlah simpul bataas kiri yaitu *t* untuk menggantikan posisinya, dan secara rekursif mencari simpul pengganti untuk simpul *t*. Sebagai sontoh anggap *g* meninggalkan jaringan dan membuat sistem tidak seimbang seperti di gambar (e). Proses restrukturisasi dimulai di *c* dimana *c* mengganti *g*, *f* mengganti *c*, *a* mengganti *f*, akhirnya *k* mengganti *a*. Proses berhenti karena pergerakan *a* tidak menyebabkan sistem tidak seimbang.

Tidak ada pergerakan data yang diperlukan karena proses restrukturisasi. Bagaimanapun, beberapa simpul mengubah posisinya dalam pohon, berefek ke levelnya dan indeks angkanya, dan berefek ke tabel rutenya. Untuk tiap simpul semacam itu, menyesuaikan tabel rute memerlukan *O(log N)* usaha. Maka, semakin banyak simpul yang berpartisipasi dalam proses restrukturisasi, makin banyak pula usaha yang dibutuhkan untuk memperbaharui tabel rutenya.

## 4. Konstruksi Indeks

Dalam bagian sebelumnya telah dijelaskan jaringan lapis dengan struktur pohon biner seimbang. Bagian ini menunjukan bagaimana cara menggunakan jaringan lapis untuk membangun sebuah struktur indeks terdistribusi yang mangkus.

Kita tentukan setiap simpul, daun maupun internal dengan *range* nilai. Kita rekam untuk setiap hubungan *range* daari nilai yang diatur dari tiap simpul di sasaran dari hubungan tersebut. Manakala terjadi perubahan *range*, hubungannya pun harus diubah untuk merekam perubahannya.

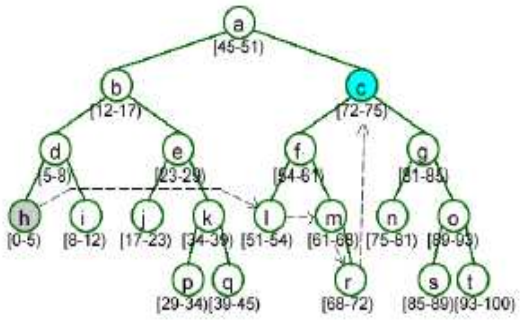
*Range* dari nilai diatur langsung dari simpul diperlukan menjadi yang paling kanan dari *range* diatur oleh upapohon kirinya dan kurang dari *range* diatur oleh upapohon kanannya. Dengan kata lain, tidak seperti $B^+$-trees, simpul internal dalam pohon mereka sendiri juga mengatur langsung *range* dari nilai data.

Dengan ini, akan menjadi mudah untuk melihat struktur lapisan BATON secara langsung bertindak seperti pohon indeks. Struktur ini sama seperti *main-memory* indeks yang bernama T-Tree, yang dirancang unruk mengurangi jumlah *pointer*.

```
Algorithm: search exact(node n, query q, value v)
If ((LowerBound(n)<=v) and (v<=UpperBound(n)))
  q is executed at x [1]
Else
  If (UpperBound(n)<v)
    m=TheFarthestNodeSatisfyingCondition
        (LowerBound(m)<=v)
    If (there exists such an m)
      Forward q to m
    Else
      If (RightChild(n)!=null)
        Forward q to RightChild(n)
      Else
        Forward q to RightAdjacentNode(n)
      End If
    End If
  Else
    //A similar process is followed towards the left
  End If
End If
```

### 4.1 *Exact Match Query*

Untuk *exact match query* diterima di simpul *x*, simpul tersebut pertama kali memeriksa *range* yang ada pada saat itu, indeks lokal dicari untuk nilainya dan pencarian berhenti. Bila tidak, *x* berjalan melalui rute dengan *query* ke simpul tujuan seperti yang dijelaskan dibawah dalam algoritma pencarian tepat.




**(f) Pencarian *Exact Match Query***

Kita akan mengiliustrasikan menggunakan gambar (f). Anggap simpul *h* ingin mencari data yang disimpan di simpul *c*. Karena sudah mencari untuk nilai yang lebih besaar dari bagian atas dari *h*, maka sistem akan mengecek tabel rute kanannya dan meneruskannya permintaan pencariaannya ke simpul *l*, yang adalah simpul yang paling kanan yang mempunyai bagian bawah lebih kecil dari nilai yang dicari, lalu *l* mengecek tabel rute kanannya dan meneruskan permintaan pencariannya ke simpul m. Di simpul m, karena tidak dapat mencari simpul tetangga untuk meneruskan permintaannya, maka permintaan akan diteruskan ke r, anak kanannnya. Akhirnya r meneruskan permintaan ke *c*, yaitu simpul yang ingin dituju.

Sewaktu simpul *x* ingin mencari untuk nilai yang tepat, bila *x* adalah akar, permintaan pencarian selalu diteruskan ke bawah sampai simpul tujuan yang nilai yang dicarinya terletak dalam indeks *range* nilai. Maka, maksimal jumlah langkah dari proses ini sebesar tinggi dari pohon tersebut. *log N*. Bila *x* bukan akar, asumsikan bahwa simpul yang diminta berada di sisi kiri dari pohon. Kita bagi dalam dua kasus. Kasus pertama, bila simpul tujuan adalah akar, mengikuti algoritma, permintaan pencarian selalu diteruskan ke simpul r paling kanan dari upapohon kiri, dan dari sana diteruskan ke akar, yang adalah simpul batas

kanan dari r. Biaya dari meneruskan permintaan samapi r adalah *log N* –1, yang adalah tinggi dari upapohon kiri karena tiap penerusan, ke simpul tetangga, anak kanan, atau simpul batas kanan, ruang pencarian selalu terbagi setengahnya. Maka angka maksimal untuk langkah yang diambil juga *logN*. Dalam kasus yang kedua, bila simpul tujuan berada di sisi kanan pohon, selama proses pencarian berlangsung, akan memakan satu langkah tiap penerusan permintaan di upapohon kiri ke simpul yang berada di upapohon kanan lewat tabel rutenya. Tergantung dari nilai yang dicari, langkah ini bisa terjadi di awal atau akhir proses pencarian. Bagaimanapun, bila terjadi pada akhir, langkah-langkah sebelumnya tetap membantu mengurangi ruang pencarian. dari upapohon kanan setengahnya. Maka, jumlah langkah juga 1 + (*logN* - 1) = *logN*, dimana *logN* adalah biaya untuk pencarian di upapohon kanan. Algoritma menunjukkan bahwa permintaan pencarian selalu diteruskan melalui simpul tetangga atau simpul anak. Permintaan hanya diperlukan untuk meneruskan ke simpul dengan level yang lebih tinggi dalam dua kasus; simpul dengan level yang lebih tinggi mengandung nilai yang dicari, atau simpul yang diproses tidak memiliki dua anak.

### 4.2 *Range Query*

*Range query* menjalankan hal yang sama dengan *point query*, dengan satu perbedaan, kita bukan mencari *range* data tiap simpul yang mengandung nilai yang dicari, tapi sebuah perpotongan dari *range* yang dicari. Begitu diketemukan, kita paling tidak mempunyai bagian dari jawaban untuk *range query*. Lalu proses diteruskan ke kiri dan atau kanan untuk menangani *range* yang telah dicari.

Sama seperti di *point query*, untuk menemukan perpotongan pertama dibutuhkan *O(logN)* langkah. Setelah itu ada biaya dari *O*(1) untuk tiap simpul yang di kunjungi. Maka dari itu, untuk menjawab *range query*, dengan *range* menangani X simpul, kita memerlukan *O(logN + X)* langkah.

### 4.3 Penyisipan Data

Sewaktu data akan dimasukkan, kita ikuti proses pencarian untuk *exact match query* untuk mencari simpul dimana data akan dimasukkan. Bagaimanapun, untuk simpul paling kiri dan paling kanan, *range* dari mereka perlu penyesuaian bila data yang dimasukkan di luar *range* yang tersedia. Bila simpul paling kiri menerima permintaan masukkan dan nilai yang dimasukkan lebih kecil dari *range* nilai, maka *range* akan melebar sehingga dapat menampung nilai yang baru dimasukkan. Begitu juga dengan simpul yang paling kanan bila nilai yang dimasukkan lebih besar dari *range* yang tersedia, maka *range* akan melebar agar dapat menampung nilai masukkan. Dalam kasus spesial seperti ini, akan menambah *logN* langkah untuk memperbaharui table rute. Biaya untuk mencari tahu lokasi simpul untuk memasukkan data baru adalah *O(logN).*

### 4.4 Penghapusan Data

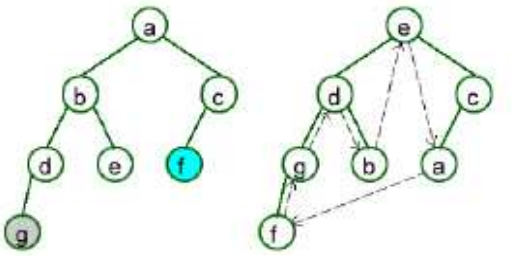
Untuk menghapus data, kita mencari lokasi simpul yang mengatur nilai data tersebut, dan menghapus datanya. Biayanya hanya untuk pencarian saja, yaitu *O(logN).*

### 4.5 Menyeimbangkan Beban

Kita ingin mendistribusikan keluaran komputasi merata untuk seluruh simpul dalam sistem *peer.* Keluaran ini diestimasi dalam syarat dari jumlah *queries* atau jumlah pesan. Biasanya, semakin besar *range* yang dapat ditampung oleh satu simpul, semakin banyak jumlah data yang ditampung oleh satu simpul maka semakin banyak keluarannya pula. Proses penyeimbang keluaran mengijinkan simpul untuk memisahkan menjadi bagian dari *range*-nya atau mengambil *range* tambahan daari simpul yang lain. Tujuannya adalah untuk menyesuaikan *range* data untuk meratakan pekerjaan keluaran.

Menyeimbangkan keluaran berbasis perpindahan data sederhana antara dua simpul yang berbatasan yang tidak dapat bekerja dengan mangkus bila data set sangat condong ke kiri atau kanan (tidak seimbang). Lebih lagi, perpindahan data bisa mlewati jaringan dan menyebabkan keluaran yang berlebihan. Maka, daripada menyeimbangkan keluaran hanya dengan simpul yang berbatasan, kita menawarkan bahwa untuk simpul akan menyeimbangkan keluaran jika di simpul yang bukan daun. Bila simpul dalah daun maka bisa keluaran dapat seimbang dari simpul-simpul yang berbatasan atau mencari simpul daun yang lain, yaitu simpul yang ringan dalam hal beban untuk dibagi beban tersebut. Secara spesiik, sewaktu simpul daun kelebihan beban, pertama-tama ia mencoba untuk membagi bebannya dengan simpul yang berbatasan dengannya. Bila

simpul yang berbatasan juga penuh dengan beban maka, ia mencari simpul yang bebannya ringan untuk menyeimbangkan beban. Anggap simpul dengan beban ringan ini berada di kanan simpul yang kelebihan beban. Simpul beban ringan ini bisa mengoper bebannya ke simpul batas kanannya lalu ia meninggalkan posisinya dan kembali bergabung sebagai anak dari simpul dengan beban lebih dengn restrukturisasi jaringan bila diperlukan.




**(g) Menyeimbangkan beban dengan restrukturisasi**

Sebagai contoh, asumsikan bahwa simpul *g* di gambar (g) kelebihan beban dan simpul *f* mempunyai beban yang ringan. simpul *f* memberi datanya ke simpul *c*, dan bergabung kembali menjadi anak dari simpul *g*. Pergerakan dari simpul ini menyebabkan struktur lapisan menjadi tidak seimbang dan restrukturisasi diperlukan. Dalam proses penyeimbangan beban simpul *f* mengganti *g*, *g* mengganti *d*, *d* mengganti *b*, *b* mengganti *e*, *e* mengganti *a*, dan *a* mengambil alih posisi asli dari *f*.

Pelajari bahwa restrukturiasi yang baru dilakukan, dan paling parah melibatkan perubahan lengkap dari posisi simpul yang kelebihan beban ke posisi simpul beban ringan. Umumnya, perubahan yang diperlukan semakin kecil, yang hanya memberikan dampak ke beberapa simpul di tiap-tiap akhir sampai menemukan tempat untuk mengakomodir kepergian dan kedatangan simpul. Kemungkinan dari perubahan yang melibatkan *k* simpul akan meningkat secara eksponensial dengan nilai *k*. Dengan sedikit analisis dapat ditunjukan biaya untuk penyeimbangan beban untuk tiap penyisipan atau penghapusan adalah hanya *O(logN).*

## 5. Kesimpulan

Terjadi berlebihannya sistem yang ditawarkan untuk jaringan lapis untuk sistem *peer-to-peer*. Namun walaupun struktur pohon banyak dipakai untuk manejemen data, namun, tidak ada struktur pohon yang dapat menjadi basis untuk sistem jaringan lapis untuk Peer-to-peer. Penemuan terbaru adalah BATON yang menjawab permasalahn yang ada, yaitu sebagai jaringan lapis berbasis struktur pohon biner seimbang untuk sistem *peer-to-peer*.

Dengan menambahkan bebearapa informasi dan hubungan tambahan, BATON dapat menghasilkan toleransi kesalahan yang baik, dan distribusi beban yang baik, yaitu tidak terpusat disekitar akar daari pohon. Makalah ini telah menjelaskan bagaimana caranya dan BATON telah mendapat pengakuan aas kompleksitasnya dengan cara eksperimen

## DAFTAR PUSTAKA

[1] Jagadish, H.V., Beng Chin Ooi, Quang Hieu Vu (2005). BATON : a Balanced Tree Structure for Peer-to-peer Networking. *http://www.vldb2005.org/program/paper/thu/p661-jagadish.pdf*. Tanggal akses: 2 Januari 2006 pukul 10:00.

[2] Munir, Rinaldi. (2004). Diktat Kuliah IF2153 Matematikan Diskrit. Edisi keempat. Departemen Teknik Informatika, Institut Teknologi Bandung.

[3] Wikipedia. *http://en.wikipedia.org/wiki/Tree_Data_Structure*. Tanggal akses : 2 Januasi 2006 pukul 10.15