Perbandingan Struktur Data B-tree dan Pohon Merah-Hitam

Muhammad Rifky Indraputra Bariansyah 13517081
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13517081@std.stei.itb.ac.id

***Abstrak*—Pada dunia pemrograman, pohon pencarian biner seimbang adalah salah satu struktur data yang seringkali digunakan. B-tree dan pohon merah-hitam adalah variasi dari pohon pencarian biner seimbang. Makalah ini bertujuan untuk memberikan gambaran ikhtisar tentang B-tree dan pohon merah hitam serta perbandingannya.**

***Keywords*—Pohon, struktur data, B-tree, pohon merah hitam**

I. PENDAHULUAN

Pohon adalah sistem hierarkis yang banyak digunakan dalam pemrograman. Pohon pencarian biner atau pohon biner terurut merupakan struktur data yang menyimpan data dalam urutan tertentu. Dalam penggunaan pohon pencarian biner, waktu yang digunakan untuk menyelesaikan suatu operasi akan proporsional terhadap tinggi pohon. Hal ini berarti semakin banyak data yang dalam sebuah pohon, dengan menggunakan algoritma penambahan pada pohon paling sederhana, maka akan meningkatkan jumlah waktu yang dibutuhkan untuk melakukan suatu operasi.

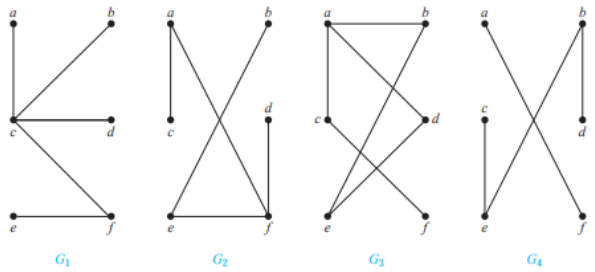
Pohon pencarian biner seimbang menggunakan operasi yang dapat menjaga tinggi pohon agar tetap proporsional. Tinggi pohon yang proporsional akan mengurangi waktu eksekusi sebuah operasi pada pohon. Pohon pencarian biner seimbang memiliki beberapa variasi yang berbeda – beda. Pada makaah ini akan dibahas penggunaan B-tree dan pohon merah hitam pada struktur data.

II. POHON

A. Definisi Pohon

Pohon adalah graf tak-berarah terhubung yang tidak memiliki simpul. Sebuah graf tak-berarah disebut pohon jika dan hanya jika terdapat lintasan sederhana antara dua tiap simpul[1]. Misalkan G = (V, E) adalah graf tak-berarah sederhana dan jumlah simpulnya n. Maka, semua pernyataan ini adalah ekivalen:

1. G adalah pohon.
2. Setiap pasang simpul di dalam G terhubung dengan lintasan tunggal.
3. G terhubung dan memiliki n - 1 buah sisi.
4. G tidak mengandung sirkuit dan memiliki n - 1 buah sisi.
5. G tidak mengandung sirkuit dan penambahan satu sisi pada graf akan membuat hanya satu sirkuit.
6. G terhubung dan semua sisinya adalah jembatan.[2]

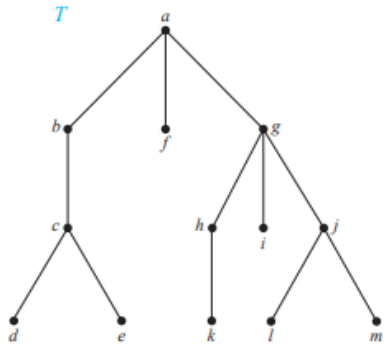


Gambar 1. Pohon dan graf yang bukan pohon
(Sumber: Discrete Mathematics and Its Applications, 5th ed., halaman 746)

G1 dan G2 merupakan pohon, G3 bukan pohon karena memiliki sirkuit sederhana, G4 bukan pohon Karena tidak terhubung.

B. Pohon Berakar

Pohon berakar adalah pohon dimana sebuah simpulnya diperlakukan sebagai akar dan setiap sisi diberi arah menjauhi dari akar[1] Terminologinya adalah sebagai berikut:



Gambar 2. Pohon berakar
(Sumber: Discrete Mathematics and Its Applications, 5th ed., halaman 747)

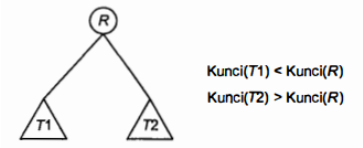
Sebuah simpul a adalah orangtua dari simpul b bila terdapat simpul berarah dari a ke b. Bila simpul a adalah orangtua dari

simpul b, maka simpul b merupakan anak dari simpul a. Simpul dengan orangtua yang sama disebut saudara kandung. b, f, dan h merupakan keturunan dari a. Simpul a merupakan leluhur dari sipul b, f, dan h. Sebuah simpul disebut daun bila tidak memiliki anak. Simpul yang memiliki anak disebut simpul dalam.

Sebuah pohon berakar disebut sebagai pohon *m-ary* bila tiap simpul dalam tidak memiliki lebih dari *m* anak. Pohon *m-ary* dengan *m* = 2 disebut pohon *binary* atau pohon biner.

III. Pohon Pencarian Biner

Pada persoalan yang melakukan operasi pencarian, penyisipan, dan penghapusan elemen pohon pencarian biner memiliki kinerja yang lebih baik daripada struktur data lain. Ketentuan pengaturan kunci pada pohon pencarian biner adalah sebagai berikut:



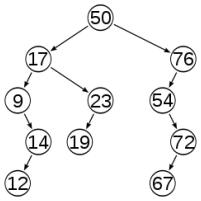
Gambar 3. Skema pohon pencarian
(Sumber: Matematika Diskrit, edisi ketiga, halaman 482)

Jika R adalah akar, dan semua kunci yang tersimpan pada setiap simpul tidak ada yang sama, maka:

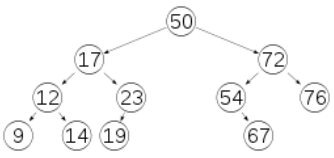
(a) Semua simpul pada upapohon kiri mempunyai kunci lebih kecil dari Kunci(*R*)
(b) Semua simpul di upapohon kanan mempunyai kunci lebih besar dari Kunci(*R*).

IV. Pohon Pencarian Biner Seimbang

Pohon penvarian biner seimbang adalah pohon pencarian biner yang secara otomatis menjaga tinggi dari pohon agar lebih kecil saat dilakukan penambahan atau pengurangan[4].



Gambar 6. Pohon tak seimbang
(Sumber: https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree)



Gambar 7. Pohon yang sama setelah seimbang
(Sumber: https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree)

Pohon pencarian biner dengan simpul *n* dan tinggi *h* memiliki persamaan:

$$n \leq 2^{h+1} - 1$$

Yang menyatakan bahwa:

$$h \geq \lceil \log_2(n+1) - 1 \rceil \geq \lfloor \log_2 n \rfloor .$$

Dalam kata lain, tinggi minimal dari sebuah pohon biner dengan simpul *n* adalah $\log_2(n)$.[4]

Pohon pencarian biner seimbang menjaga tinggi minimal ini dengan melakukan transformasi pada pohon seperti rotasi. operasi dasar pada pohon pencarian biner seimbang hanya memerlukan waktu Olog2(n).

Pada referensi [3], dijelaskan algoritma operasi pencarian pada pohon pencarian biner seimbang secara rekursif

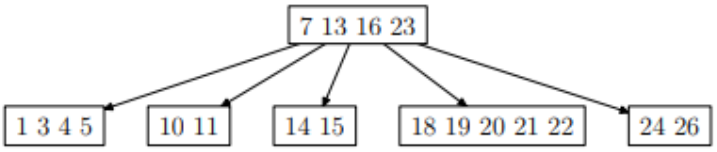
```
TREE-SEARCH(x, k)
1  if x == NIL or k == x.key
2      return x
3  if k < x.key
4      return TREE-SEARCH(x.left, k)
5  else return TREE-SEARCH(x.right, k)
```

Gambar 3. Algoritma pencarian pohon biner seimbang
(Sumber: Introduction to Algorithms, third edition, halaman 290)

V. B-Tree

B-Tree *T* dengan akar [*T*] orde *m* adalah pohon *m*-ary dengan tiap simpul *x*:

1. Jumlah kunci yang tersimpan pada simpul *x*, n[*x*].
2. Kunci n[*x*] sendiri tersimpan dalam urutan tidak menurun Kunci1[*x*] ≤ Kunci1[*x*] ≤…. ≤ Kunci(n[*x*])[*x*] .
3. Sebuah nilai Boolean, daun[*x*] true jika *x* adalah daun.
4. *Pointers* sebanyak n[*x*] + 1 c1[*x*], c2[*x*], .. , c(n[*x*]+1)[*x*] ke anaknya.
5. Kunci*i*[*x*] memisahkan jarak kunci yang tersimpan di setiap upapohon. Bila *ki* adalah kunci yang tersimpan pada upapohon dengan akar c*i*[*x*] maka: *k1*≤kunci1[*x*]≤ *k2* ≤ kunci2[x]≤..≤kunci(n[*x*]) ≤*k*(n[*x*]+1)
6. Semua daun memiliki tinggi yang sama.
7. Derajat minimum dari B-tree ($t \geq 2$):
   - Lower bound:
     Setiap simpul selain akar harus memiliki minimal kunci sebanyak *t*-1 dan minimal anak sebanyak *t*
   - Upper bound:
     Setiap simpul paling banyak memiliki kunci sebanyak 2*t* – 1 setiap simpul dalam memiliki maksimal anak sebanyak 2*t*.[5]



Gambar 8. B-tree dengan *t* = 3
(Sumber: http://www.di.ufpb.br/lucidio/Btrees.pdf)

Berikut adalah algoritma beberapa operasi umum pada B-tree:

1. Pencarian

   Dengan masukan *x*, *pointer* ke akar sebuah upapohon, *k*, kunci yang dicari pada upapohon tersebut. Pemanggilan teratas dalam bentuk B-TREE-SEARCH(*root*[*T*], *k*). Bila *k* terdapat dalam pohon tersebut akan dikembalikan nilai (*y*,*i*) mengandung simpul *y* dan indeks *i* dimana kunci*i*[*y*] = *k*. Bila tidak akan dikembalikan NIL.

```
B-TREE-SEARCH(x, k)
1  i ← 1
2  while i ≤ n[x] and k ≥ key_i[x]
3        do i ← i + 1
4  if i ≤ n[x] and k = key_i[x]
5       then return (x, i)
6  if leaf [x]
7       then return NIL
8       else DISK-READ(c_i[x])
9            return B-TREE-SEARCH(c_i[x], k)
```

Gambar 9. *Pseudocode* pencarian B-tree
(Sumber: http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap19.htm)

Jumlah disk yang diakses pada fungsi B-TREE-SEARCH adalah $\Theta(h) = \Theta(\log_t n)$, dimana *h* adalah tinggi B-tree dan *n* adalah jumlah kunci pada B-tree. Semenjak $n[x] < 2t$ waktu yang dibutuhkan pada *while loop* adalah O(*t*), maka total waktu yang dibutuhkan adalah $O(th) = O(t \log_t n)$.[5]

2. Penambahan

   B-TREE-INSERT memerlukan fungsi B-TREE-SPLIT-CHILD untuk memastikan rekursi tidak turun hingga simpul penuh. B-TREE-SPLIT-CHILD memecah simpul penuh *y* dengan kunci sejumlah $2t - 1$ disekitaran kunci median key*t*[*y*] ke dua simpul dengan $t - 1$ kunci masing-masing. Kunci median tersbut kemudian akan bergerak ke orangtua *y*, jika *y* tidak memiliki orangtua maka tinggi pohon akan bertambah satu.

   Fungsi B-TREE-SPLIT-CHILD menerima masukkan simpul dalam *nonfull x*, indeks *i*, simpul *y*. dimana *y* = c*i*[*x*] adalah anak *full* dari *x*.

```
B-TREE-SPLIT-CHILD(x,i,y)
1  z ← ALLOCATE-NODE()
2  leaf[z] ← leaf[y]
3  n[z] ← t - 1
4  for j ← 1 to t - 1
5       do key^j[z] ← key^(j+t)[y]
6  if not leaf [y]
7     then for j ← 1 to t
8               do c_j[z] ← c_(j+t)[y]
9  n[y] ← t - 1
10 for j ← n[x] + 1 downto i + 1
11      do c_(j+1)[x] ← c_j[x]
12 c_i+1[x] ← z
13 for j ← n[x] downto i
14       do key_(j+1)[x] ← key_j[x]
15 key_i[x] ← key_t[y]
16 n[x] ← n[x] + 1
17 DISK-WRITE(y)
18 DISK-WRITE(z)
19 DISK-WRITE(x)
```

Gambar 10. Pseudocode B-TREE-SPLIT-CHILD
(Sumber: http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap19.htm)

```
B-TREE-INSERT(T,k)
1  r ← root[T]
2  if n[r] = 2t - 1
3     then s ← ALLOCATE-NODE()
4          root[T] ← s
5          leaf[s] ← FALSE
6          n[s] ← 0
7          c^1[s] ← r
8          B-TREE-SPLIT-CHILD(s,1,r)
9          B-TREE-INSERT-NONFULL(s,k)
10 else B-TREE-INSERT-NONFULL(r,k)
```

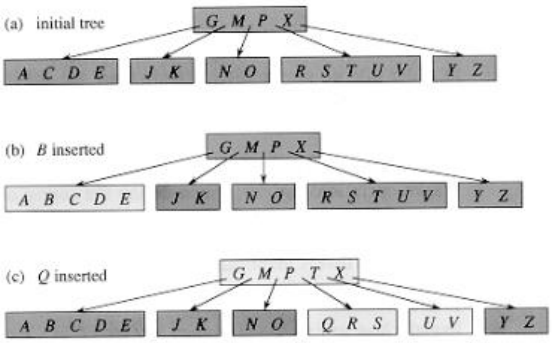
Gambar 11. Pseudocode B-TREE-INSERT
(Sumber: http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap19.htm)

Baris 3-9 menangani kasus dengan *r* sebagai simpul *full*. Akar akan dipisah dan simpul *s* menjadi akar yang baru. Memisahkan pohon adalah satu – satunya cara untuk meningkatkan tinggi dari B-tree. Tidak seperti pohon pencarian biner biasa, B-tree meningkatkan tinggi dari atas dan bukan dari bawah. Fungsi penambahan ini kemudian diselesaikan dengan pemanggilan fungsi B-TREE-INSERT-NONFULL untuk melakukan penambahan kunci *k* ke pohon yang berakar pada simpul akar *nonfull*.[5]

```
B-TREE-INSERT-NONFULL(x,k)
1  i ← n[x]
2  if leaf[x]
3     then while i ≥ 1 and k < key^i[x]
4               do key_i+1 [x] ← key^i[x]
```

```
5                 i ← i - 1
6           key_i+1[x] ← k
7           n[x] ← n[x] + 1
8           DISK-WRITE(x)
9   else while i ≥ 1 and k < key^i[x]
10                do i ← i - 1
11        i ← i + 1
12        DISK-READ(c^i[x])
13        if n[c^i[x]] = 2t - 1
14           then B-TREE-SPLIT-CHILD(x,i,c^i[x])
15                if k > key^i[x]
16                   then i ← i + 1
17        B-TREE-INSERT-NONFULL(c^i[x],k)
```

Gambar 12. Pseudocode B-TREE-INSERT
(Sumber: http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap19.htm)



Gambar 13. Penambahan pada B-Tree
(Sumber: http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap19.htm)

Gambar tersebut dijelaskan sebagai berikut:

a. Kondisi pohon awal.
b. Penambahan B pada pohon awal, ini adalah operasi penambahan sederhana.
c. Penambahan Q, simpul RSTUV dipisah menjadi dua simpul yang mengandung RS dan UV, Kunci T dipindahkan ke akar dan Q ditambahkan di paling kiri dari kedua bagian (simpul RS).

Operasi penambahan membutuhkan waktu O($t \log_t n$).

3. Penghapusan
   Penghapusan pada B-tree dibagi menjadi beberapa kasus: Missal *k* adalah kunci yang akan dihapus, *x* adalah simpul yang mengandung kunci tersebut. Maka kasus – kasusnya adalah:
   a. Jika *k* ada pada simpul *x* yang merupakan daun, langsung hapus *k* dari *x*.
   b. Jika *k* ada pada simpul *x* yang merupakan simpul dalam, ada tiga kasus yang harus dipertimbangkan:
      I. Jika anak *y* yang mendahului *k* pada simpul *x* memiliki palin tidak kunci sebanyak *t*, maka temukan kunci pendahulu *k'* dalam upapohon yang berakar pada *y*. Kemudian secara rekursi hapus *k'* dan ganti *k* dengan *k'* pada *x*.
      II. Secara simetris, jika anak *z* yang mengikuti *k* pada simpul *x* memiliki paling tidak simpul sebanyak *t*, temukan pendahulu *k'* dan hapus dan gantikan seperti sebelumnya.
      III. Jika tidak, bila *y* dan *z* hanya memiliki kunci sebanyak *t*-1, gabungkan *k* dan semua *z* ke *y*, sehingga *k* dan pointer ke *z* terhapus dari *x*. Sekarang *y* memiliki 2*t* – 1 kunci dan *k* terhapus.
   c. Bila *k* tidak terdapat pada simpul dalam *x*, tentukan akar upapohon yang harus mengandung *k*. Jika akar tersebut hanya memiliki kunci sebanyak *t*-1, maka ada dua kasus yang harus dipertimbangkan:
      I. Jika akar hanya memiliki *t*-1 kunci tapi memiliki saudara sebanyak *t* kunci, tambahkan akar sebuah kunci dari *x*, memindahkan anak dari saudara kandung kiri atau kanan akar ke *x*, dan memindahkan anak dari saudara tersebut ke *x*.
      II. Jika akar dan semua saudara kandungnya memilik kunci sebanyak *t*-1, gabungkan akar tersebut dengan satu saudara kandung. Ini melibatkan memindahkan kunci dari *x* ke simpul gabungan baru untuk menjadi kunci median simpul tersebut.[5]

```
B-TREE-DELETE-KEY(x, k)
  if not leaf[x] then
    y ← PRECEDING-CHILD(x)
    z ← SUCCESSOR-CHILD(x)
    if n[y] > t − 1 then
        k' ← FIND-PREDECESSOR-KEY(k, x)
        MOVE-KEY(k', y, x)
        MOVE-KEY(k, x, z)
        B-TREE-DELETE-KEY(k, z)
    else if n[z] > t − 1 then
        k' ← FIND-SUCCESSOR-KEY(k, x)
        MOVE-KEY(k', z, x)
        MOVE-KEY(k, x, y)
        B-TREE-DELETE-KEY(k, y)
    else
        MOVE-KEY(k, x, y)
        MERGE-NODES(y, z)
        B-TREE-DELETE-KEY(k,  y)

  else (leaf node)
    y ← PRECEDING-CHILD(x)
    z ← SUCCESSOR-CHILD(x)
    w ← root(x)
    v ← RootKey(x)
```

```
    if n[x] > t − 1 then REMOVE-KEY(k, x)
    else if n[y] > t − 1 then
        k′ ← FIND-PREDECESSOR-KEY(w, v)
        MOVE-KEY(k′, y, w)
        k′ ← FIND-SUCCESSOR-KEY(w, v)
        MOVE-KEY(k′, w, x)
        B-TREE-DELETE-KEY(k, x)
    else if n[w] > t − 1 then
        k′ ← FIND-SUCCESSOR-KEY(w, v)
        MOVE-KEY(k′, z, w)
        k′ ← FIND-PREDECESSOR-KEY(w, v)
        MOVE-KEY(k′, w, x)
        B-TREE-DELETE-KEY(k, x)
else
    s ← FIND-SIBLING(w)
    w′ ← root(w)
    if n[w′] = t − 1 then
        MERGE-NODES(w′, w)
        MERGE-NODES(w, s)
        B-TREE-DELETE-KEY(k, x)
    else
        MOVE-KEY(v, w, x)
        B-TREE-DELETE-KEY(k, x)
```

Gambar 13. Pseudocode B-TREE-DELETE-KEY
(Sumber: http://www.di.ufpb.br/lucidio/Btrees.pdf)

Preceding-Child(*x*) mengembalikan anak kiri dari kunci *x*. Move-Key(*k, n1, n2*) menggerakkan kunci k dari simpul n1 ke n2. Merge-Nodes(*n1,n2*) menggabungan kunci dari simpul *n1* dan *n2*. Find-Predecessor-Key(*n,k*) mengembalikan kunci pendahulu *k* pada anak dari simpul *n*. Remove-Key(*k,n*) menghapus kunci *k* dari simpul *n*. *n* haruslah sebuah daun.[5]

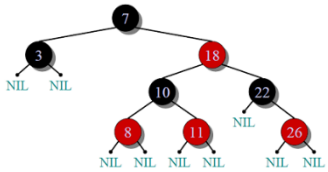
Waktu yang dibutuhkan untuk operasi ini adalah $O(t \log_t n)$.

VI. POHON MERAH-HITAM

Pohon merah-hitam adalah pohon pencarian biner dengan satu bit ekstra yang mengandung warna hitam atau merah. Dengan menghambat warna simpul dari lintasan sederhana dari akar ke daun akan menjaga pohon sehingga tidak ada lintasan yang dua kali lebih panjang dari lintasan lain, sehingga pohon tersebut seimbang.[3]

Pohon merah-hitam memiliki sifat sebagai berikut:

1. Setiap simpul pasti merah atau hitam.
2. Akar berwarna hitam.
3. Setiap daun berwarna hitam.
4. Jika simpul berwarna merah maka kedua anaknya berwarna hitam.
5. Untuk setiap simpul, semua lintasan sederhana dari simpul ke daun keturunan memiliki jumlah simpul berwarna hitam yang sama.

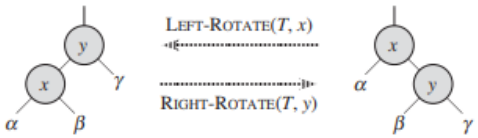


Gambar 14. Pohon merah-hitam
(Sumber: https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/)

Berikut adalah algoritma dari beberapa operasi umum pada pohon merah-hitam:

1. Rotasi

   Operasi penambahan dan pengurangan pada pohon merah-hitam dapat menghasilkan pohon yang melanggar sifat pohon merah-hitam. Untuk mengembalikan sifat tersebut, harus dilakukan perubahan warna dari beberapa simpul dan struktur *pointer* melalui rotasi. [3]



Gambar 15. Rotasi pada pohon merah-hitam
(Sumber: Introduction to Algorithms, third edition, halaman 313)

```
LEFT-ROTATE(T, x)
 1  y = x.right
 2  x.right = y.left
 3  if y.left ≠ T.nil
 4      y.left.p = x
 5  y.p = x.p
 6  if x.p == T.nil
 7      T.root = y
 8  elseif x == x.p.left
 9      x.p.left = y
10  else x.p.right = y
11  y.left = x
12  x.p = y
```

Gambar 16. Pseudocode Left-Rotate
(Sumber: Introduction to Algorithms, third edition, halaman 313)

2. Penambahan

   Kita dapat menambahkan sebuah node z ke pohon T seperti pohon biner biasa dengan kemudian mewarnai z merah. Dibutuhkan pemanggilan RB-Insert-Fixup untuk memastikan sifat pohon merah-hitam dngan melakukan rotasi dan pewarnaan.

```
RB-INSERT(T, z)
 1  y = T.nil
 2  x = T.root
 3  while x ≠ T.nil
 4      y = x
 5      if z.key < x.key
 6          x = x.left
 7      else x = x.right
 8  z.p = y
 9  if y == T.nil
10      T.root = z
11  elseif z.key < y.key
12      y.left = z
13  else y.right = z
14  z.left = T.nil
15  z.right = T.nil
16  z.color = RED
17  RB-INSERT-FIXUP(T, z)
```

Gambar 17. Pseudocode RB-Insert
(Sumber: Introduction to Algorithms, third edition, halaman 315)

Pada RB-Tree ditetapkan *z.left* dan *z.right* ke *T.nil* untuk menjaga struktur pohon. Kemudian, *z* diwarnai merah dan karena pewarnaan bisa menyebabkan pelanggaran

pada sifat dari pohon merah-hitam, dipanggil RB-Insert-Fixup.

```
RB-INSERT-FIXUP(T, z)
 1  while z.p.color == RED
 2      if z.p == z.p.p.left
 3          y = z.p.p.right
 4          if y.color == RED
 5              z.p.color = BLACK
 6              y.color = BLACK
 7              z.p.p.color = RED
 8              z = z.p.p
 9          else if z == z.p.right
10                  z = z.p
11                  LEFT-ROTATE(T, z)
12              z.p.color = BLACK
13              z.p.p.color = RED
14              RIGHT-ROTATE(T, z.p.p)
15      else (same as then clause
                with "right" and "left" exchanged)
16  T.root.color = BLACK
```

Gambar 18. Pseudocode RB-Insert-Fixup
(Sumber: Introduction to Algorithms, third edition, halaman 316)

Operasi penambahan pada pohon merah-hitam memerlukan waktu O(*log n*).

3. Penghapusan
   Ketika kita akan menghapus simpul *z* yang memiliki kurang dari dua anak, maka akan langsung dihapus dan kemudian kita akan mengubah *y* menjadi *z*. ketika *z* memiliki dua anak maka y akan menjadi pendahulu *z* dan *y* berpindah ke posisi *z*. kita harus mengingat warna y saat perpindahan pada pohon, kemudian *x* yang akan berpindah ke posisi *y* sebelumnya harus diikuti untuk memastikan sifat pohon merah-hitam tetap terjaga. Setelah menghapus simpul *z*, RB-Delete memanggil RB-Delete-Fixup yang melakukan perubahan warna dan rotasi.

```
RB-DELETE(T, z)
 1  y = z
 2  y-original-color = y.color
 3  if z.left == T.nil
 4      x = z.right
 5      RB-TRANSPLANT(T, z, z.right)
 6  elseif z.right == T.nil
 7      x = z.left
 8      RB-TRANSPLANT(T, z, z.left)
 9  else y = TREE-MINIMUM(z.right)
10      y-original-color = y.color
11      x = y.right
12      if y.p == z
13          x.p = y
14      else RB-TRANSPLANT(T, y, y.right)
15          y.right = z.right
16          y.right.p = y
17      RB-TRANSPLANT(T, z, y)
18      y.left = z.left
19      y.left.p = y
20      y.color = z.color
21  if y-original-color == BLACK
22      RB-DELETE-FIXUP(T, x)
```

Gambar 19. Pseudocode RB-Delete
(Sumber: Introduction to Algorithms, third edition, halaman 324)

Operasi pengurangan pada pohon merah-hitam memerlukan waktu O(log n).

VII. Perbandingan B-Tree dan Pohon Merah-Hitam

Jika *n* adalah jumlah simpul pada sebuah pohon.

1. Tinggi Pohon
   Tinggi pohon pada B-tree adalah log(*n*) yang lebih kecil daripada pohon merah-hitam yaitu log(*n+1*).
2. Pencarian
   Waktu pencarian pada B-tree adalah O($t \log_t n$) dimana *t* adalah jumlah derajat, sementara pada pohon merah-hitam O($\log_2 n$). Menunjukkan bahwa semakiin besar derajat pada B-tree memungkinkan waktu operasi yang lebih cepat dibanding pada pohon merah-hitam.
3. Penambahan dan pengurangan
   Waktu penambahan dan pengurangan pada B-tree adalah O($t \log_t n$) dimana *t* adalah jumlah derajat, sementara pada pohon merah-hitam O($\log_2 n$).

Pada penggunaan di kehidupan nyata, contohnya pada skema pencarian, penambahan, dan pengurangan, bila data yang akan dioperasikan sangat besar dan melebihi dari yang bisa ditangani memori maka penyimpanan data akan dilakukan di disk dan penggunaan yang cocok adalah B-tree karena akan mengurangi jumlah operasi rotasi. Bila jumlah memori mencukupi untuk mengoperasikan data maka pohon merah-hitam bisa digunakan untuk akses yang lebih cepat dan jumlah elemen yang harus dipindai pada kasus terburuk akan lebih sedikit.

VIII. Kesimpulan

B-tree dan pohon merah-hitam merupakan variasi dari pohon pencarian biner seimbang. Kedua variasi pohon pencarian biner seimbang ini memungkinkan operasi pencarian, penambahan, dan pengurangan yang cepat. Pada jumlah data yang besar atau yang tersimpan di disk lebih cocok menggunakan B-tree, sementara yang lebih kecil lebih cocok menggunakan pohon merah-hitam.

VII. Ucapan Terima Kasih

Penulis mengucapkan rasa syukur kepada Tuhan Yang Maha Esa karena telah memberikan nikmat yang cukup untuk menyelesaikan makalah ini. Penulis juga ingin mengucapkan terima kasih kepada dosen mata kuliah Matematika Diskrit, bapak Rinaldi Munir, ibu Harlili, dan khususnya kepada bapak Judhi Santoso yang telah mengajar kelas K03 selama satu semester ini. Kemudian, penulis ingin mengucapkan terimakasih kepada teman – teman dan keluarga serta seluruh pihak yang telah memberikan dukungan baik secara langsung maupun tidak langsung.

Referensi

[1] Rosen, Kenneth H., Discrete Mathematics and Its Applications (5th ed.), McGraw-Hill Higher Education, 2002.

[2] Munir, Rinaldi, *Pohon[slide PowePoint]*. http://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/20132014/Pohon%20(2013).pdf diakses pada 8 Desember 2018.
[3] Munir, Rinaldi, Matematika Diskrit, Bandung: Informatika Bandung, 2009.
[4] Knuth, Donlad, The Art of Computer Programming (2nd ed.), Addison-Wesley, 1998.
[5] http://www.di.ufpb.br/lucidio/Btrees.pdf, diakses pada 8 Desember 2018
[6] Cormen, Thomas H.; Leiserson, Charles E.; Rivest Ronald L.; Stein Clifford, Introduction to Algorithms (3rd ed.), MIT Press & McGraw Hill, 2009

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, 8 Desember 2018

M. Rifky I. Bariansyah 13517081