Variasi Pohon Pencarian Biner Seimbang

Tony 13516010
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
buddy90_lost@yahoo.co.id

Abstrak — Pohon merupakan struktur data yang sering kali digunakan dalam dunia komputer. Salah satu struktur data yang banyak digunakan karena kecepatannya dalam melakukan operasi tertentu adalah pohon pencarian biner seimbang. Pohon biner seimbang memiliki sangat banyak variasi, yaitu pohon AVL, pohon merah-hitam, pohon AA, pohon splay, pohon 2-3 dan masih banyak lagi. Pada makalah ini akan dibahas dua variasi dari pohon pencarian biner seimbang yang populer, yaitu pohon AVL dan pohon merah-hitam.

Kata kunci — Pohon AVL, pohon merah-hitam, pohon pencarian biner seimbang, struktur data.

I. PENDAHULUAN

Pohon adalah struktur data yang bersifat hirarkis. Pohon biner merupakan struktur data pohon yang banyak di gunakan di keilmuan komputer. Pohon pencarian biner adalah struktur data pohon biner yang simpul-simpulnya tersusun dengan aturan sehingga pohon pencarian biner memiliki suatu keterurutan. Pohon pencarian biner mendukung operasi penambahan, penghapusan, dan pencarian data dengan cepat, yaitu setara dengan tingginya.

Namun, dengan dilakukannya operasi penambahan dan pengurangan data, tinggi dari pohon pencarian biner bisa sampai sebanyak banyak data yang ada, sehingga pohon pencarian biner bisa saja tidak lebih sangkil dibanding struktur data tabel. Tinggi dari pohon pencarian biner dapat dipertahankan seminimum mungkin dengan menjaga keseimbangan dari pohon tersebut. Pada makalah ini akan dibahas struktur data pohon pencarian biner yang dapat menjaga keseimbangannya dan variasinya, yaitu pohon AVL dan pohon merah-hitam.

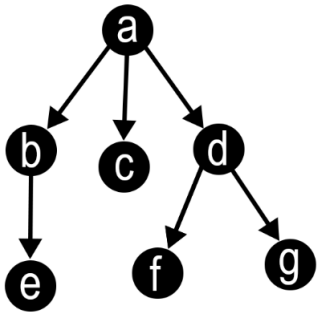
II. DEFINISI POHON

Graf terhubung yang tidak mengandung sirkuit disebut pohon[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.

III. POHON BERAKAR

Pohon berakar (*rooted tree*) merupakan pohon berarah. Pohon yang sebuah simpulnya diperlakukan sebagai akar dan sisi-sisinya diberi arah menjauh dari akar dinamakan pohon berakar[1]. Sisi pohon berakar biasanya digambar tanpa arah sebagai konvensi.



Gambar 1. Pohon berakar

Terminologi pohon berakar berdasarkan Gambar 1:

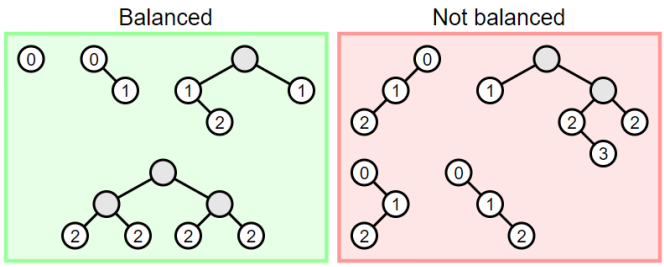
1. Anak (*child*) dan orang tua (*parent*)
   Simpul b, c, dan d merupakan anak dari simpul a serta simpul b merupakan orang tua dari simpul e.
2. Lintasan (*path*)
   Lintasan dari a ke g adalah a, d, g.
   Panjang lintasannya adalah 2.
3. Keturunan (*descendant*) dan leluhur (*ancestor*)
   Simpul b, e, dan f merupakan keturunan dari simpul a.
   Simpul a merupakan leluhur dari simpul c, d, g, dan e.
4. Saudara kandung (*sibling*)
   Simpul b merupakan saudara kandung dari simpul c dan simpul d. Simpul e bukan saudara kandung dari simpul f maupun g.
5. Upapohon (*subtree*)
   Pohon berakar dengan simpul b dan e merupakan upapohon dari pohon berakar dengan simpul a, b, c, d, e, f, dan g.
6. Derajat (*degree*)
   Derajat adalah jumlah anak yang dimiliki oleh suatu simpul. Derajat simpul a adalah 3.
7. Daun (*leaf*)
   Daun adalah simpul yang berderajat nol. Simpul c, e, f, dan g merupakan daun.
8. Simpul dalam (*internal node*)

Simpul dalam adalah simpul yang memiliki anak. Simpul b dan d adalah simpul dalam.

9. Aras (*level*) atau tingkat
   Aras suatu akar adalah 0, sedangkan aras simpul lainnya adalah panjang lintasan dari akar ke simpul tersebut. Aras simpul a adalah 0, aras simpul e adalah 2.
10. Tinggi (height) atau kedalaman (depth)
    Tinggi adalah panjang lintasan maksimum dari akar ke daun. Tinggi pohon berakar tersebut adalah 3.

IV. Pohon Biner Seimbang

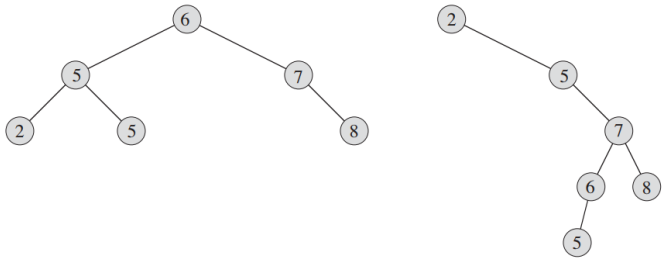
Pohon biner merupakan salah satu jenis pohon yang setiap simpulnya paling banyak memiliki dua anak. Pohon biner seimbang adalah pohon biner yang perbedaan tinggi upapohon kiri dan upapohon kanan maksimal satu serta upapohon kiri dan upapohon kanan seimbang. Pohon biner seimbang yang optimal memiliki ketinggian $\lceil \log_2(n+1) \rceil$, dengan n merupakan banyak simpul[4].



Gambar 2. Pohon biner seimbang (kiri) dan pohon biner tak seimbang (kanan).
(sumber: http://www.growingwiththeweb.com/2015/11/check-if-a-binary-tree-is-balanced.html diakses pada 3 Desember 2017 pukul 3.32 WIB)

III. Pohon Pencarian Biner

Pohon pencarian biner adalah sebuah pohon biner yang anak nilai dari anak kirinya paling besar sebesar nilainya sendiri dan nilai dari anak kanannya paling kecil sebesar nilainya sendiri. Simpul dari suatu pohon pencarian biner dapat berupa kunci dari suatu data atau data itu sendiri. Suatu himpunan data dapat memiliki lebih dari satu representasi pohon pencarian biner.

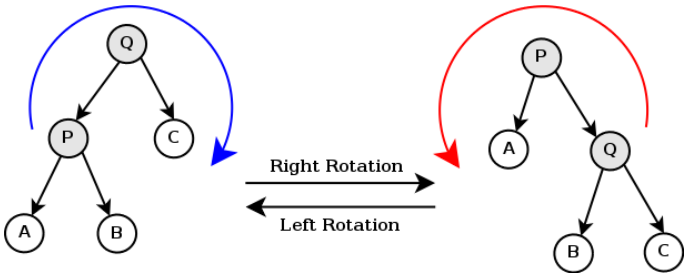


Gambar 3. Dua buah pohon pencarian biner yang berbeda, namun memiliki himpunan nilai yang sama.
(Sumber: Introduction to Algorithm 3rd edition, halaman 287)

Pada pohon pencarian biner, dapat dilakukan operasi pencarian, penambahan, dan penghapusan dengan cepat. Operasi dasar diatas memerlukan waktu setara dengan tingginya. Dalam melakukan operasi penghapusan dan penambahan, pohon pencarian biner harus tetap menjaga keterurutan simpulnya.

IV. Pohon Pencarian Biner Seimbang

Pohon pencarian biner seimbang adalah pohon pencarian biner yang seimbang. Pohon ini memililiki kelebihan, yaitu dalam melakukan operasi dasar hanya memerlukan waktu $\Theta(\log_2 n)$, dengan n adalah banyak simpul. Setelah melakukan operasi penambahan maupun penghapusan pohon pencarian biner seimbang harus melakukan rotasi jika pohon menjadi tidak seimbang. Rotasi adalah operasi pada pohon biner yang mengubah struktur tanpa mengubah keterurutannya. Ada dua macam rotasi, yaitu rotasi kanan dan rotasi kiri.



Gambar 4. Rotasi pohon
(sumber: https://en.wikipedia.org/wiki/Tree_rotation)

Dari Gambar 4, rotasi kanan adalah perubahan keadaan struktur pohon atau upapohon dari gambar sebelah kiri menjadi gambar sebelah kanan serta rotasi kiri adalah perubahan dari keadaan struktur pohon atau upapohon dari gambar sebelah kanan menjadi keadaan gambar sebelah kiri.

Algoritma operasi umum pada pohon pencarian biner seimbang:

1. Pencarian
   Berikut algoritma untuk operasi pencarian.

```
Tree-Search(T,k)
1   if x == Nil or k == T.key
2       return T
3   if k < T.key
4       return Tree-Search(T.left,k)
5   else
6       return Tree-Search(T.right,k)
```

Gambar 6. *Pseudocode* fungsi pencarian dengan rekursif
(sumber : Introduction to Algorithms (3rd ed.) halaman 290)

```
Itterative-Tree-Search(T,k)
1   while x ≠Nil nd k ≠T.key
2   if k < T.key
3       T = T.left
4   else
5       T = T.right
6   return T
```

Gambar 7. *Pseudocode* fungsi pencarian dengan iteratif
(sumber : Introduction to Algorithms (3rd ed.) halaman 291)

V. VARIASI POHON PENCARIAN BINER SEIMBANG

A. Pohon AVL

Pohon AVL diberi nama dari penemunya, yaitu G. M. Adel'son, Vel'skii, dan E. M. Landis. Pohon AVL adalah pohon biner yang selisih tinggi upapohon kiri dan kanan antara -1 dan 1[4]. Terminologi pada pohon AVL

1. Factor keseimbangan (*balance factor*) adalah nilai dari tinggi upapohon kiri dikurang tinggi upapohon kanan.

Pohon AVL dengan n simpul mempunyai tinggi antara $\log_2(n+1)$ dan $1{,}44\log_2(n+2) - 0{,}328$[4].

Setiap simpul pada pohon AVL mempunyai atribut tinggi (*height*), kunci (*key*), kiri (*left*), dan kanan (*right*).Operasi-operasi umum pada pohon AVL adalah:

1. Rotasi

   Fungsi *LeftRotation* adalah fungsi yang melakukan rotasi ke kiri dan fungsi *RightRotation* adalah fungsi yang melakukan rotasi ke kanan. Berikut implementasi dari kedua fungsi tersebut.

```
LeftRoatation(T)
1  l = T.left
2  r = T.right
3  r.left  = T
4  T.right = l
5  r.height = CalculateHeight(r)
6  T.height = CalculateHeight(T)
7  return r
RightRotation(T)
1  l = T.left
2  r = T.right
3  l.right = T
4  T.left = r
5  l.height = CalculateHeight(l)
6  T.height = CalculateHeight(T)
7  return l
```

Gambar . *Pseudocode* untuk fungsi rotasi
(sumber: https://www.thecodingdelight.com/avl-tree-implementation-java/ diakes pada tanggal 3 Desember 2017 pukul 19.06)

2. Penambahan

   Fungsi Insert melakukan penambahan nilai x ke pohon AVL T dan fungsi BalanceTree melakukan rotasi kiri atau rotasi kanan kiri, jika pohon atau upapohon AVL berat ke kanan atau melakukan rotasi kanan atau rotasi kiri kanan, jika pohon atau upapohon AVL berat ke kiri. Berikut kedua implementasi dari fungsi diatas.

```
Insert(T, x)
 1 if T == NULL
 2     T = AlocNode(x)
 3     T.height = 0
 4     return T
 5 if T.key < x
 6     T.left = Insert(T.left, x)
 7 else
 8     T.right = Insert(T.right, x)
 9 T = BalanceTree(T, x)
10 T.height = CalculateHeight(T)
11 return T
```

Gambar . *Pseudocode* untuk fungsi Insert
(sumber: https://www.thecodingdelight.com/avl-tree-implementation-java/ diakes pada tanggal 3 Desember 2017 pukul 19.06)

```
BalanceTree(T)
 1  balval = GetBalanceValue(T)
 2  if balval > 1
 3      if GetBalanceValue(T.left) < 0
 4          T.left = LeftRotation(T.left)
 5      return RightRotation(T)
 7  elseif balval < -1
 8      if GetBalanceValue(T.right) > 0
 9          T.right = RightRotation(T.right)
10      return LeftRotation(T)
11  return T
```

Gambar . Pseudocode untuk fungsi BalanceTree
(sumber: https://www.thecodingdelight.com/avl-tree-implementation-java/ diakes pada tanggal 3 Desember 2017 pukul 19.06)

3. Penghapusan

   Fungsi Remove melakukan penghapusan terhadap suatu simpul dengan nilai yang diinginkan.

```
Remove(T,x)
 1  if T == NULL
 2      return NULL
 3  if T.key == x
 4      if T.left == NULL and T.right == NULL
 5           return NULL
 6      elseif T.left == NULL
 7          return right
 8      elseif T.right == NULL
 9          return left
10      else
11          Temp = GetLargestNode(T.left)
12          T.key = Temp.key
13          T.left = Remove(T.left,Temp.key)
14  elseif x < T.key
15      T.left = Remove(T.left,x)
16  else
17      T.right = Remove(T.right,x)
18  T.height = CalculateHeight(T)
19  return BalanceTree(T)
```

Gambar . *Pseudocode* untuk fungsi Remove
(sumber: https://www.thecodingdelight.com/avl-tree-implementation-java/ diakes pada tanggal 3 Desember 2017 pukul 19.06)

Untuk setiap operasi pada Pohon AVL, kompleksitas

waktunya adalah:

1. Pencarian: $O(\log_2 N)$[4].
2. Penambahan: $O(\log_2 N)$[4].
3. Penghapusan: $O(\log_2 N)$[4].

Karena memerlukan banyak operasi rotasi untuk setiap operasi penambahan dan penghapusan, sekarang penggunaan pohon AVL sudah kurang dibanding pohon merah-hitam, salah satu aplikasi dari pohon AVL adalah struktur data set.

B. Pohon Merah-Hitam

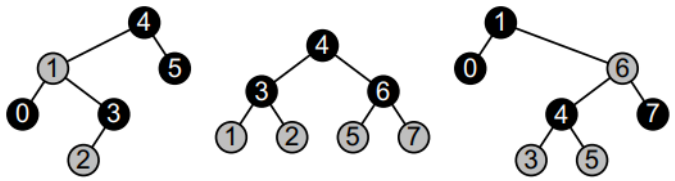
Pohom merah-hitam (*red-black tree*) ditemukan oleh R. Bayer dan dipelajari lebih lanjut oleh L. J. Guibas dan R. Sedgeeick. Operasi penambahan dan penghapusan pada pohon merah-hitam lebih kompleks dibandingkan dengan pohon AVL serta tinggi maksimum pohon merah-hitam lebih besar dibanding kan pohon AVL, namun operasi rotasi setelah penghapusan pada pohon merah-hitam tidak pernah melebihi 3 rotasi, sedangkan pada pohon AV L bisa sampai $\log_2 N$ operasi.

Berikut beberapa istilah/terminologi pada pohon merah-hitam:

1. Lintasan simpel, adalah lintasan yang hanya memiliki simpul yang unik (dalam lintasan tersebut tidak ada simpul yang mengulang).

Setiap simpul dari pohon hitam-putih memiliki atribut warna (*color*), kunci (*key*), kiri (*left*), kanan (*right*), dan p (*parent*). Jika suatu anak atau orang tua tidak ada, maka atribut penunjuk bernilai *NIL*. Berikut sifat-sifat dari pohon merah-hitam:

1. Setiap simpul adalah ‘hitam’ atau ‘merah’.
2. Akar adalah simpul hitam.
3. Setiap daun *NIL* adalah hitam.
4. Jika sebuah simpul merah, maka anaknya haruslah hitam.
5. Untuk setiap simpul, lintasan simpel dari simpul tersebut ke daun keturunannya mempunyai jumlah simpul hitam yang sama.



Gambar 5. Pohon merah-hitam yang valid. Pada gambar diatas, simpul merah berwarna abu-abu dan simpul hitam berwarna hitam.
(sumber: An Introduction to Binary Search Trees and Balanced Trees halaman 140)

Lemma 1. Sebuah pohon merah-hitam dengan N buah simpul dalam memiliki tinggi maksimum $2\log_2 N + 1$[2].

Operasi pada pohon merah-hitam:

1. Rotasi
   Dalam melakukan operasi penambahan dan penghapusan, perlu dilakukan rotasi agar pohon merah-hitam masih tetap terjaga.

```
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
11       x.p.right = y
12  y.left = x
13  x.p = y
```

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

Gambar . *Pseudocode* untuk rotasi pada pohon merah-hitam (sumber : Introduction to Algorithms (3rd ed.) halaman 313)

2. Penambahan

   Pada proses penambahan diperlukan 2 prosedur, yaitu prosedur RB-Insert melakukan penambahan simpul z ke pohon merah-hitam T dan prosedur RB-Insert-Fixup di panggil untuk menjaga keseimbangan pohon merah-hitam T, setelah penambahan simpul z.

```
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
 8            x = x.right
 9  z.p = y
10  if y == T.nil
11       T.root = z
12  elseif z.key < y.key
13       y.left = z
14  else
15       y.right = z
16  z.left = T.nil
17  z.right = T.nil
16  z.color = RED
17  RB-Insert-Fixup(T,z)
```

Gambar . *Pseudocode* untuk prosedur RB-Insert (sumber : Introduction to Algorithms (3rd ed.) halaman 315)

```
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           elseif 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
16          (same as then clause with
            'right' and 'left' exchanged)
17  T.root.color = BLACK
```

Gambar . *Pseudocode* untuk prosedur RB-Insert-Fixup
(sumber : Introduction to Algorithms (3rd ed.) halaman 316)

3. Penghapusan
   Pada proses penghapusan, diperlukan 3 prosedur, yaitu prosedur RB-Delete melakukan penghapusan terhadap simpul z dari pohon T, prosedur RB-Transplant melakukan perubahan ke pohon T dari simpul u dan v, dan prosedur RB-Delete-Fixup membuat pohon T tetap sesuai dengan aturan/sifat dari pohon merah-hitam. Berikut implementasi dari ketiga prosedur diatas.

```
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
10      y = Tree-Minimum(Z.right)
11      y-original-color = y.color
12      x = y.right
13      if  y.p == z
14          x.p = y
15      else
16          RB-Transplant(T,y,y.right)
17          y.right = z.right
18          y.right.p = y
19      RB-Transplant(T,z,y)
20      y.left = z.left
21      y.left.p = y
22      y.color = z.color
23  if y-original-color == BLACK
24      RB-Delete-Fixup(T,x)
```

Gambar . *Pseudocode* prosedur RB-Delete
(sumber : Introduction to Algorithms (3rd ed.) halaman 324)

```
RB-Transplant(T,u,v)
1   if u.p == T.nil
2       T.root = v
3   elseif u == u.p.left
4       u.p.left = v
5   else
6       u.p.right = v
7   v.p = u.p
```

Gambar . *Pseudocode* prosedur RB-Transplant
(sumber : Introduction to Algorithms (3rd ed.) halaman 323)

```
RB-Delete-Fixup(T,x)
 1  while x ≠ T.root and x.color == BLACK
 2      if x == x.p.left
 3          w = x.p.right
 4          if w.color == RED
 5              w.color = BLACK
 6              x.p.color = RED
 7              Left-Rotate(T,x.p)
 8              w = x.p.right
 9          if w.left.color == BLACK and
               w.right.color == BLACK
10              w.color = RED
11              x = x.p
12          else if w.right.color == BLACK
13              w.left.color = BLACK
14              w.color = RED
15              Right-Rotate(T,w)
16              w = x.p.right
17              w.color = x.p.color
18              x.p.color = BLACK
19              w.right.color = BLACK
20              Left-Rotate(T,x.p)
21              x = T.root
21      ele
22          (same as then clause with
            'right' and 'left' exchanged)
23  x.color = BLACK
```

Gambar . *Pseudocode* prosedur RB-Delete-Fixup
(sumber : Introduction to Algorithms (3rd ed.) halaman 326)

Setiap operasi pada pohon merah-hitam, kompleksitas waktunya adalah:

1. Penambahan: $O(\log_2 N)$[2].
2. Penghapusan: $O(\log_2 N)$[2].
3. Pencarian: $O(\log_2 N)$[2].

Sekarang pohon merah-hitam lebih banyak digunakan dibandingkan dengan pohon AVL, contoh aplikasi dari pohon ini adalah *complete fair scheduler* (CFS) di kernel linux, implementasi struktur data *map* dan *set* pada *standard template library* (STL) C++, implementasi *HashMap* pada bahasa pemrograman Java, struktur data untuk komputasi geometri, dan masih banyak lagi.

V. PERBANDINGAN

Berikut perbandingan antara pohon merah-hitam dan pohon AVL :

1. Tinggi
   Tinggi pohon AVL antara $\log_2(n+1)$ dan $1{,}44\log_2(n+2) - 0{,}328$, sedangkan tinggi pohon merah-hitam antara $\log_2(n+1)$ dan $2\log_2(n+1)$. Ini menunjukkan bahwa pohon merah-hitam lebh tinggi dibanding pohon AVL pada kasus terburuk.
2. Penambahan
   Pohon merah-hitam lebih cepat dikarenakan rata-rata operas rotasi yang di lakukan lebih sedikit dibanding pohon AVL.
3. Pencarian
   Pogon AVL lebih cepat dikarenakan tinggi maksimal pohon AVL kurang dari tinggi maksimal pohon merah-

hitam.

4. Penghapusan
   Pohon merah-hitam lebih cepat dikarenakan rata-rata operasi rotasi yang di lakukan lebih sedikit dibanding pohon AVL.
5. Implementasi
   Implementasi pohon AVL jauh lebih mudah dan sedikit dibandingkan implementasi pohon merah-hitam.

V. KESIMPULAN

Pohon pencarian biner seimbang merupakan struktur data yang cepat untuk operasi-operasi seperti pencarian, penambahan, dan penghapusan dibanding struktur data lainnya. Pohon AVL dan pohon merah-hitam merupakan salah dua dari banyak variasi pohon pencarian biner seimbang.

Pohon AVL lebih sangkil jika digunakan untuk hal yang memerlukan operasi pencarian yang banyak, sedangkan pohon merah-hiam lebih sangkil jika digunakan untuk halam memerlukan operasi pennambahan dan penghapusan yang banyak.

VII. UCAPAN TERIMA KASIH

Penulis mengucapkan terima kasih kepada semua pihak yang secara langsung maupun tidak langsung telah membantu kelancara pembuatan makalah ini.

REFERENCES

[1] Munir, Rinaldi, *Matematika Diskrit*, Bandung: Informatika Bandung, 2009.
[2] Cormen, Thomas H.; Leiserson, Charles E.; Rivest Ronald L.; Stein Clifford, Introduction to Algorithms (3rd ed.), MIT Press & McGraw Hill, 2009.
[3] Grimaldi, Ralph P., Discrete and Combinatorial Mathematics An Applied Introduction (5th ed.), Pearson Education, 2004.
[4] Pfaf, Ben, An Introduction to Binary Search Trees and Balanced Trees, Free Software Foundation, 2004.
[5] https://www.thecodingdelight.com/avl-tree-implementation-java/ diakes pada tanggal 3 Desember 2017 pukul 19.06 WIB
[6] http://www.growingwiththeweb.com/2015/11/check-if-a-binary-tree-is-balanced.html diakses pada 3 Desember 2017 pukul 3.32 WIB

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, 3 Desember 2017

Tony (13516010)