Implementasi Skema Pohon Biner yang *Persistent* dalam Pemrograman Fungsional

Azby Khilfi M. – NIM : 13506018
Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung
Jl. Ganesha 10, Bandung
E-mail : if116018@students.if.itb.ac.id

Abstrak

Makalah ini membahas kajian tentang skema pemrograman pohon biner *persistent* dalam paradigma fungsional dan peng-implementasi-annya. Skema pohon biner ini menitikberatkan kelebihannya yang *persistent*. Disebut persistent karena perubahan yang dilakukan terhadap suatu pohon biner tidak destruktif dan masih menyimpan pohon biner awal. Biasanya dalam pemrograman pohon biner dalam paradigma fungsional, *programmer* cenderung melakukan perubahan yang destruktif terhadap data (pohon biner) awal sehingga setelah dilakukan suatu fungsi atau prosedur akan mengubah data tersebut. Karena itu, di sini penulis akan mencoba menjelaskan skema yang dapat melakukan perubahan yang tidak destruktif terhadap data awal.

***Kata kunci**: programmer, persistent, file*, direktori, pohon, pohon berakar, pohon n-ary, pohon biner, pohon terurut, pohon biner penuh, pohon biner seimbang, pohon condong kiri, pohon condong kanan, traversal, preorder, inoreder, postorder, *search, insert, delete, pseudocode*, bahasa LISP, predecessor, successor, akses, konso, assign, representasi.

1. Pendahuluan

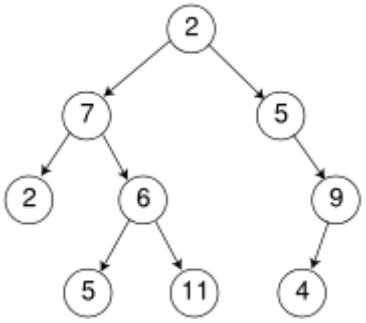
Salah satu bentuk struktur data yang sering digunakan adalah pohon. Struktur ini juga biasanya digunakan pada system operasi pada computer untuk menyimpan dan mengolah *file* dan direktori. Dari beberapa struktur pohon, yang paling penting adalah pohon biner. Dianggap penting karena mudah untuk dilakukan proses-proses tanpa harus susah-susah memikirkan pengaksesan bagiannya. Dalam makalah ini kita akan mencoba menerapkan skema pohon biner yang *persistent* atau kukuh dibandingkan skema pohon biner biasa. Mungkin skema ini tidak selalu memuaskan, di samping memang skema ini menitikberatkan pada kekukuhan struktur pohonnya. Karena itu kegunaannya tergantung tujuan *programmer* dalam memilih skema. Oleh karena itu, diperlukan pemahaman yang baik dalam memilih suatu skema yang akan digunakan untuk menyelesaikan permasalah yang kita hadapi.

2. Pohon Berakar

Definisi

Pohon yang sebuah simpulnya diperlakukan sebagai akar dan sisi-sisinya diberi arah sehingga menjadi graf berarah disebut **pohon berakar** (*rooted tree*).

Akar mempunyai derajat masuk sama dengan nol, sedangkan simpul lainnya mempunyai derajat masuk satu. Simpul yang mempunyai derajat keluar sama dengan nol disebut **daun** atau simpul terminal. Sedangkan simpul yang mempunyai derajat keluar sama dengan satu disebut simpul dalam atau simpul cabang.



Gambar 1 : Sebuah pohon biner sederhana dengan tinggi 3 dan simpul akar bernilai 2

Untuk selanjutnya sebagai perjanjian, arah sisi dalam pohon dapat dibuang, karena setiap simpul pohon harus dicapai dari akar, maka lintasan di dalam pohon berakar selalu dari atas ke bawah.

Terminologi

Anak (child) dan Orangtua (*parent*)

Sebuah simpul dikatakan sebagai **anak** dari suatu simpul jika ada sisi dari suatu simpul ke simpul tersebut. Simpul asal dari sisi tersebut dinamakan **orangtua**.
Pada gambar 1, simpul dengan nilai 2 dan 6 merupakan anak dari simpul dangan nilai 7, sedemikian rupa sehingga simpul 7 merupakan orangtua dari simpul 2 dan 6.

Lintasan (*path*)

Lintasan dari simpul v_1 ke simpul vk adalah runtunan simpul-simpul $v_1,v_2,..,v_k$ sedemikian sehingga v_i adalah orangtua dari simpul v_{i+1} untuk $1 \leq i < k$.
Panjang Lintasan adalah jumlah sisi yang dilalui suatu lintasan yaitu k-1.
Pada gambar 1, lintasan dari simpul 2 ke 11 adalah 2,7,6,11. Sedangkan panjang lintasannya 3.

Keturunan (*descendant*) dan Leluhur (*ancestor*)

Jika terdapat lintasan dari simpul x ke simpul y di dalam pohon, maka x adalah **leluhur** dari simpul y, dan y adalah **keturunan** dari simpul x.
Pada gambar 1, simpul 7 adalah leluhur dari simpul 11, dan 11 adalah keturunan dari 7.

Saudara kandung (*sibling*)

Simpul yang mempunyai orangtua sama adalah **saudara kandung** satu sama lain.
Pada gambar 1, simpul 2 dan 6 mempunyai orangtua sama yaitu simpul 7 sehingga merupakan saudara kandung. Namun simpul 2 dan 9 bukan merupakan saudara kandung karena orangtuanya berbeda.

Upapohon (*subtree*)

Misalkan *x* adalah simpul di dalam suatu pohon *T*. Yang dimaksud dengan **upapohon** dengan *x* sebagai akarnya ialah upagraf *T' = (V', E')* sedemikian sehingga *V'* mengandung *x* dan semua keturunannya dan *E'* mengandung sisi-sisi dalam semua lintasan yang berasal dari *x*.

Derajat (*degree*)

Derajat sebuah simpul dalam pohon berakar adalah jumlah upapohon (atau jumlah anak) pada simpul tersebut.
Derajat maksimum dari semua simpul pada suatu pohon berakar merupakan derajat pohon itu sendiri.
Pada gambar 1, derajat simpul 2 adalah 2, derajat simpul 9 adalah 1, dan derajat simpul 4 adalah 0. Pohon tersebut mempunyai derajat 2 (biner) karena derajat maksimum dari semua simpulnya adalah 2.

Tingkat (*level*)

Simpul akar mempunyai tingkat = 0, sedangkan simpul lainnya mempunyai tingkat = panjang lintasan dari akar ke simpul tersebut. Beberapa literatur memulai tingkat dari 0, beberapa dari 1. Sebagai perjanjian kita akan memulai penomoran tingkat dari 0.
Pada gambar 1, tingkat simpul 9 adalah 2, sedangkan tingkat simpul 11 adalah 4.

Tinggi (*height*) dan Kedalaman (*depth*)

Tingkat maksimum dari suatu pohon disebut **tinggi** dari pohon tersebut. Atau dapat dikatakan **tinggi** adalah panjang lintasan maksium dari akar ke daun.
Kedalaman suatu simpul adalah panjang lintasan dari akar ke simpul tersebut atau dapat juga disebut tingkat.
Pada gambar 1, tinggi pohon tersebut adalah 4. Sedangkan kedalaman simpul 6 adalah 2.

3. Pohon Biner

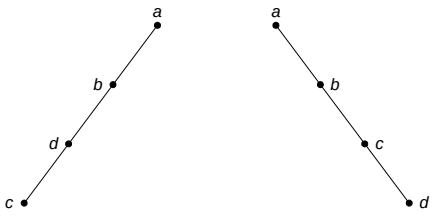
Pohon berakar yang setiap simpulnya mempunyai paling abnayak n buah anak disebut pohon *n-ary*.

Pohon biner adalah pohon *n-ary* dengan *n* = 2. Pohon biner tiap simpulnya mempunyai paling banyak 2 buah anak. Pohon ini merupakan pohon yang paling penting karena banyak aplikasinya.

Alih-alih menyebut anak-anaknya dengan sebutan anak pertama dan kedua, kita menyebutnya dengan **anak kiri** (*left child*) dan **anak kanan** (*right child*). Pohon yang akarnya adalah anak kiri disebut **upapohon kiri** (*left subtree*), sedangkan pohon yang akarnya adalah anak kanan disebut **upapohon kanan** (*right subtree*).

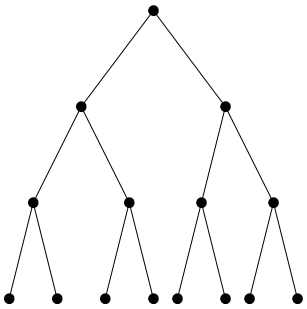
Karena ada perbedaan urutan anak, maka pohon biner merupakan **pohon terurut** (*ordered tree*). Pohon terurut sendiri adalah pohon berakar yang urutan anak-anaknya penting.

Pohon yang semua simpulnya terletak di bagian kiri saja atau di bagian kanan saja disebut **pohon condong kiri** (*skew left*) dan **pohon condong kanan** (*skew right*).



Gambar 2 : Pohon condong kiri dan Pohon condong kanan

Pohon biner yang setiap simpulnya mempunyai tepat 2 buah anak, kiri dan kanan, kecuali pada daun disebut **pohon biner penuh** (*full binary tree*).

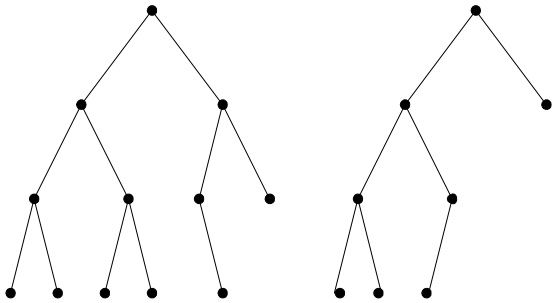


Gambar 3 : Pohon biner penuh

Pohon biner penuh dengan tinggi *h* mempunyai jumlah daun sebanyak 2h buah, sedangkan jumlah seluruh simpulnya adalah:

$$S = 2^0 + 2^1 + 2^2 + 2^3 + \ldots + 2^h = 2^{h+1} - 1$$

Pohon biner seimbang (*balanced binary tree*) adalah pohon biner yang tinggi upapohon kiri dan kanannya berbeda maksimal satu.



T1 *T2*

Gambar 4 : Pohon biner seimbang T1 dan pohon biner tidak seimbang T2

Skema Penulusuran Pohon Biner

Ada tiga macam skema penulusuran (*traversal*) pohon biner:

Misalkan T adalah pohon biner, akarnya R, upapohon kiri Left, dan upapohon kanannya Right.

1. *Preorder*
   i. Kunjungi R
   ii. Telusuri Left secara preorder
   iii. Telusuri Right secara preorder
2. *Inorder*
   i. Telusuri Left secara inorder
   ii. Kunjungi R
   iii. Telusuri Right secara inorder
3. *Postorder*
   i. Telusuri Left secara postorder
   ii. Telusuri Right secara postorder
   iii. Kunjungi R

4. Algoritma Pohon Biner *Persistent* dalam Pemrograman Fungsional

Kita akan merepresentasikan pohon biner sebagai list dengan 3 elemen, yaitu nilai simpul yang berisi integer , anak kirinya, dan anak kanannya. List kosong merepresentasikan pointer kosong (sebuah simpul kosong)

Kita akan menuliskan beberapa prosedur penting seperti *search*, *insert*, dan *delete*.

Pertama, pencarian biner (*binary search*). Kita harus mengecek apakah upapohon (*subtree*) sekarang :

1) Kosong, berarti nilai yang kita cari tidak ada di dalam upapohon.
2) Nilai simpul pada upapohon sekarang sama dengan nilai yang kita cari, sehingga kita mengembalikan true.
3) Nilai simpul pada upapohon sekarang lebih besar atau sama dengan nilai yang kita cari, sehingga kita harus melakukan pencarian rekursif pada anak kirinya.
4) Nilai simpul pada upapohon sekarang lebih kecil daripada nilai yang kita cari, sehingga kita harus melakukan pencarian rekursif pada anak kanannya.

Berikut *pseudocode* prosedur *search* yang diadopsi dari bahasa LISP namun tidak sepenuhnya memenuhi sintaks LISP.

Sebagai perjanjian, pada *pseudocode* ini akan mereferensikan **car tree** sebagai nilai akar upapohon, **cadr tree** sebagai anak kiri upapohon, dan **caddr tree** sebagai anak kanan upapohon. Selanjutnya perefeerensian ini akan terus dipakai pada *pseudocode-pseudocode* berikutnya.

```
Fungsi TreeSearch (input: T:BinaryTree,
value: integer)

Kamus:
T : BinaryTree
value : integer

Algoritma:

(define (TreeSearch T value)
   (cond  ((null T) False)
          ((equal value (car T)) True)
          ((<= value (car T))
           (TreeSearch (cadr T) value)
          )
          (else
           (TreeSearch (caddr T) value)
          )
   )
)
```

Berikutnya, fungsi penyisipan (*insertion*). Penyisipan cukup mudah karena kita hanya perlu untuk menelusuri pohon dengan menggunakan perbandingan dan melakukan penyisipan simpul baru saat kita menemukan anak kosong

```
Fungsi TreeInsertion (input: T:BinaryTree,
value: integer)

Kamus:
T : BinaryTree
value : integer

Algoritma:

(define (TreeInsertion T value)
   (cond ((null T) (list value '() '()))
          ((<= value (car T))
           (list (car T) (TreeInsertion
            (cadr T) value) (caddr T))
          )
          (else
           (list (car T) (cadr T)
            (TreeInsertion (caddr T)value))
          )
   )
)
```

Berikutnya adalah penghapusan simpul (*delete*) yang tidak sesederhana pencarian dan penyisipan.

1) Menghapus sebuah simpul yang tidak punya anak cukup sederhana, yaiut menggantinya dengan list null.

2) Menghapus sebuah simpul dengan satu anak juga cukup sederhana, yaiut dengan mengganti simpul tersebut dengan anaknya.

3) Menghapus sebuah simpul dengan 2 anak lebih rumit. Kita harus menemukan apakah simpul terurut membesar (*in-order predecessor*) atau mengecil (*in-order successor*). Kita akan mengganti nilai simpul dengan nilai *predecessor*(simpul paling kanan dalam upapohon kiri) dan menghapus predecessor tersebut sehingga simpul hanya punya satu anak karena anak kanannya menjadi kosong secara definisi

Jadi, masalah pertama yang harus kita pecahkan, diberikan sebuah upapohon, adalah menemukan simpul paling kanan, menghapusnya dan mengembalikan nilai dan upapohon barunya (dengan simpul paling kanan telah dihapus). Kita akan mengembalikan nilai dan pohon baru ini dalam pasangan konso (*cons)*.

Kita akan menggunakan sebuah kode yang dapat mengembalikan nilai simpul dan upapohon yang telah dimodifikasi dalam satu kali *traversal*. Masalah yang kita hadapi adalah: bagaimana kita dapat meng-*akses* nilai yang ditemukan di akhir rekursi tanpa menghancurkan struktur yang dibangun rekusrsi tersebut.

Sebelum melakukan konso (*cons*) pohon yang diperbarui dengan nilainya kita harus mengevaluasi pohon tersebut terlebih dahulu untuk menghindari sifat tak terdefinisi (*undefined behaviour*). Kita akan melakukan *assign* (let) terakhir untuk memastikan nilai telah di-set sebelum dilakukan konso dengan pohon yang diperbarui.

```
Fungsi ReplaceInOrderPredecessor (input:
T:BinaryTree)
Fungsi DelRightMost (input: T1:
BinaryTree)

Kamus:
T,T1,Tupdated : BinaryTree
Value : integer

Algoritma:

(define (ReplaceInOrderPredecessor T)
   (let ((value '() )))
       (define (DelRightMost T1))
         (if (null (caddr T1))
             (list value(car T1)(cadr T1))
             (list (car T1) (cadr T1)
               (DelRightMost (caddr T1)))
          )
        )
    (let ((Tupdated  (DelRightMost T))))
    (cons value Tupdated)
)
```

Sekarang kita memebuat sebuah fungsi , diberikan sebuah upapohon, yang dapat menghapus simpul akarnya. Jangan lupa tiga kasus yang telah dideskripsikan tadi, yaitu : tidak punya anak, punya satu anak, punya dua anak.

Sekarang fungsi penghapusan utama: bila telah diberikan suatu nilai, fungsi akan mencari dan menghapus simpul pertama yang ditemukan yang bernilai sama. Tidak lupa diberikan suatu permberitahuan kesalahan bila tidak ditemukan nilai yang akan dihapus.

```
Fungsi TreeDelete(input: T:BinaryTree,
value: integer)

Kamus:
T : BinaryTree
value : integer

Algoritma:

(define (TreeDelete T value)
   (cond  ((null T)
              (output "nilai tidak
                ditemukan")
          )
          ((equal value (car T))
              (TreeDeleteNode T)
          )
          ((<= value (car T))
           (list (car T)(TreeDelete(cadr T)
              value) (caddr T))
          )
          (else
           (list (car T) (cadr T)
              (TreeDelete (caddr T) value))
          )
   )
)
```

Itulah realisasi fungsi-fungsi utama. Sekarang mari kita buat sebuah fungsi list ke pohon (*list to tree*)dan perataan pohon (*tree flattening*) untuk mengimplementasikan pengurutan pohon biner (*sorting binary tree*) untuk mengetes implementasi pohon di atas. Kita akan menggunakan 1 subfungsi dan 1 fungsi pembantu untuk merealisasikan pengurutan pohon.

```
Fungsi ListToTree(input: L:List )
Fungsi Helper (input : L1:List,
T:BinaryTree)

Kamus:
T : BinaryTree
L,L1 : List

Algoritma:

(define (ListToTree L)
   (define (Helper L1 T)
      (if (null L1)
           (T)
           (Helper (cdr L1) (TreeInsert T
              (car L1)))
      )
   )
   (Helper L '() )
)


Fungsi TreeFringe(input: T:BinaryTree)

Kamus:
T : BinaryTree

Algoritma:

(define (TreeFringe T)
   (if (null T)
       ('() )
       (append (TreeFringe (cadr T)) (list
         (car T)) (TreeFringe (caddr T)))
   )
)


Fungsi TreeSort(input: T:BinaryTree,
L:List)

Kamus:
T : BinaryTree
L : List

Algoritma:

(define (TreeSort L T)
   (TreeFringe (ListToTree L))
)
```

5. Simulasi

Mari kita coba fungsi di atas dengan sebuah contoh kasus:

> (tree-sort '(2 43 7 8 5 4 23 4556 6 7 4 3 2))
(2 2 3 4 4 5 6 7 7 8 23 43 4556)

Untuk mengilustrasikan kekukuhan (*persistent*) kode di atas:

> (let tree1 (list->tree '(3 2 1 4 5)))
> tree1
(3 (2 (1 () ()) ()) (4 () (5 () ())))
> (tree-delete tree1 4)
(3 (2 (1 () ()) ()) (5 () ()))
> tree1
(3 (2 (1 () ()) ()) (4 () (5 () ())))

6. Kesimpulan

Kesimpulan yang didapat dari hasil simulasi:

- Skema pohon biner ini tidak mengubah struktur awal walaupun telah dilakukan fungsi yang menghasilkan pohon baru
- Skema ini cocok untuk permasalahan yang menuntut penyimpanan struktur awal pohon

DAFTAR PUSTAKA

[1] Munir, Rinaldi. (2005). Bahan Kuliah IF2151 Matematika Diskrit. Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung.

[2]Literate Programs, http://en.literateprograms.org/Binary_tree_%28Scheme%29" Tanggal akses: 26 Desember 2007 pukul 22:51.

[3] English Wikipedia, *http://en.wikipedia.org/wiki/* Tanggal akses: 26 Desember 2007 pukul 23:00.