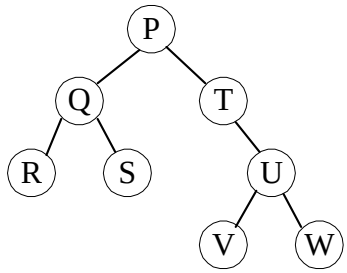
7 POHON BINAR

7.1 POHON

Pohon atau *tree* adalah salah satu bentuk *graph* terhubung yang tidak mengandung sirkuit. Karena merupakan *graph* terhubung, maka pada pohon selalu terdapat *path* atau jalur yang menghubungkan setiap dua simpul dalam pohon. Kali ini kita sampai pada pembahasan suatu bentuk pohon, yang dilengkapi dengan apa yang disebut "akar" atau "*root*". Pohon semacam ini disebut pohon berakar atau *rooted tree*. Selanjutnya, lebih khusus lagi dibahas tentang pohon berakar yang disebut "pohon binar" atau "*binary tree*". Contoh di Gambar 7.1 yang disebut pohon berakar P.



Gambar 7.1. *Contoh pohon berakar*

Sifat utama sebuah pohon berakar adalah :

1. Jika pohon mempunyai simpul sebanyak n, maka banyaknya ruas atau *edge* adalah (n-1). Pada pohon P di Gambar 7.1, banyak simpul adalah n = 8, dan banyak *edge* (n – 1) = 8 – 1 = 7
2. Mempunyai simpul khusus yang disebut "*root*," yang merupakan simpul yang memiliki derajat keluar >= 0, dan derajat masuk = 0. Simpul P merupakan *root* pada pohon di Gambar 7.1 di atas.
3. Mempunyai simpul yang disebut sebagai "daun" atau "*leaf*," yang merupakan simpul berderajat keluar 0, dan berderajat masuk = 1. Simpul-simpul R, S, V, W merupakan daun pada pohon di Gambar 7.1.
4. Setiap Simpul mempunyai tingkatan atau *level*, yang dimulai dari *root* yang *level*nya = 0, sampai dengan *level* n pada daun paling bawah.
   Pada pohon P di Gambar 7.1 :
   P ber*level* 0
   Q dan T ber*level* 1
   R, S dan U ber*level* 2
   V dan W ber*level* 3
   "Simpul yang mempunyai *level* sama disebut "bersaudara" atau "*brother*" atau "*siblings*".
5. Pohon mempunyai ketinggian atau kedalaman atau "*height*," yang merupakan *level* tertinggi + 1. Pohon di Gambar 7.1 mempunyai ketinggian atau kedalaman 3+1 = 4.
6. Pohon mempunyai berat atau bobot atau "*weight*," yang merupakan banyaknya daun pada pohon. Pohon di Gambar 7.1 mempunyai bobot = 4.

7.2 POHON BINAR (*BINARY TREE*)

Dalam struktur data, pohon memegang peranan yang cukup penting. Struktur ini biasanya digunakan terutama untuk menyajikan data yang mengandung hubungan hirarkikal antara elemen-elemen mereka. Kita lihat misalnya, data pada *record*, keluarga dari pohon, ataupun isi dari tabel. Mereka mempunyai hubungan hirarkikal.

Bentuk pohon berakar yang khusus, yang lebih mudah kita kelola dalam komputer adalah pohon binar (*binary tree*). Bentuk pohon berakar yang umum, kita kenal sebagai "pohon umum" atau "*general tree*."

Sebuah pohon binar T didefinisikan terdiri atas sebuah himpunan hingga elemen yang disebut simpul (*node*), sedemikian sehingga :

(a) T adalah hampa (disebut pohon *null*) atau;

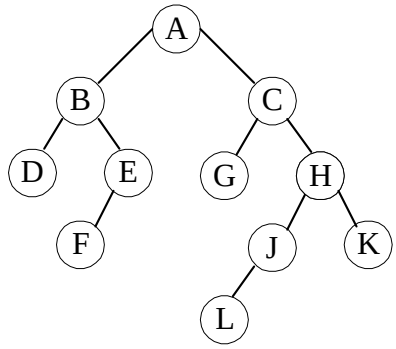
(b) T mengandung simpul R yang dipilih (dibedakan dari yang lain), disebut "akar" atau "*root*" dari T, dan simpul sisanya membentuk 2 pohon binar (subpohon kiri dan subpohon kanan dari akar R) T_1 dan T_2 yang saling lepas.

Perhatikan bahwa pendefinisian pohon binar di atas adalah rekursif. Jika T_1 tidak hampa, maka simpul akarnya disebut suksesor kiri dari R. Hal serupa untuk akar dari T_2 (tidak hampa) disebut suksesor kanan dari R.

Pohon binar acapkali disajikan dalam bentuk diagram. Perhatikan contoh Pohon binar pada Gambar 7.2 berikut. Pohon binar tersebut mempunyai 11 simpul yang diberi label huruf A sampai L (tak termasuk I). Simpul akar adalah simpul yang digambar pada bagian paling atas. Untuk mengambarkan suksesor kiri serta suksesor kanan, dibuat garis ke kiri bawah dan ke kanan bawah. Perhatikan pada Gambar tersebut bahwa B adalah suksesor kiri dari A, sedangkan C adalah suksesor kanan dari A. Subpohon kiri dari A mengandung simpul-simpul B, D, E dan F, sedangkan subpohon kanannya mengandung simpul-simpul C, G, H, J, K dan L.

Kita dapat melihat bahwa jika N adalah sebarang simpul dari pohon binar T, maka N mempunyai 0, 1 atau 2 buah suksesor. Suksesor kerap kali disebut anak atau anak lelaki (*child* atau *son*). Jadi simpul N tersebut boleh kita sebut ayah atau orang-tua (*father* atau *parent*) dari suksesornya.

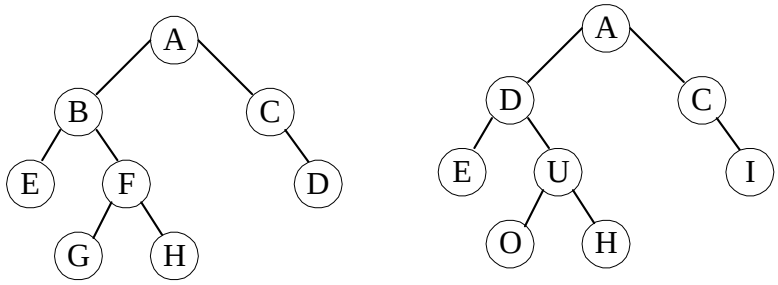
Pada contoh kita, Simpul A, B, C dan H mempunyai 2 anak, simpul E dan J mempunyai satu anak. Sementara itu simpul-simpul D, F, G, L dan K tidak mempunyai satu anakpun. Simpul yang tidak mempunyai anak disebut "daun" atau "terminal."



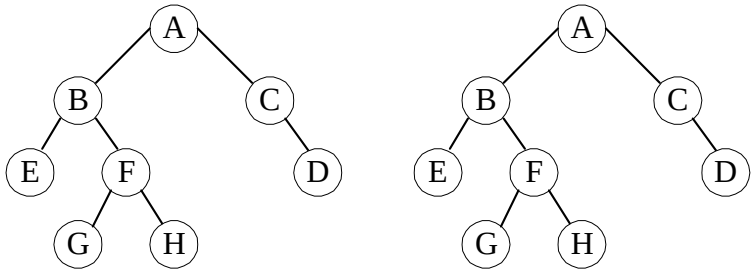
Gambar 7.2. *Contoh pohon binar dengan 11 simpul*

Sekali lagi perhatikan bahwa definisi pohon binar di atas adalah rekursif. T didefinisikan berdasarkan subpohon binar T_1 dan T_2. Ini berarti bahwa setiap simpul N dari pohon mengandung subpohon kiri dan kanan. Jika simpul N adalah daun maka kedua subpohon kiri dan kanannya adalah hampa.

Dua pohon binar T dan U disebut "*similar*" jika mereka mempunyai bangun (susunan) yang sama. Dua pohon binar pada Gambar 7.3 berikut ini adalah *similar*, sementara itu, pohon pada Gambar 7.4 menggambarkan dua pohon yang tidak saja *similar* tetapi juga sama persis antara satu dengan lainnya (baik susunan atau struktur pohon, maupun isi dari setiap simpulnya) yang disebut dengan salinan (*copy / copies*).



Gambar 7.3. *Dua pohon binar yang disebut similar*

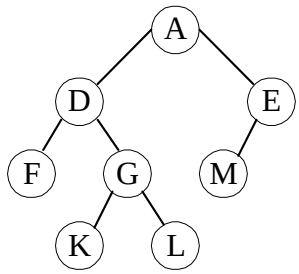


Gambar 7.4. *Dua pohon binar yang disebut copies*

7.3 TERMINOLOGI PADA POHON BINAR

Terminologi hubungan keluarga banyak digunakan dalam terminologi pada pohon binar. Misalnya istilah anak kiri dan anak kanan, untuk menggantikan suksesor kiri dan suksesor kanan, serta istilah ayah untuk pengganti predesesor. Selain itu juga istilah saudara (*brother*) yang diberikan kepada dua simpul yang mempunyai ayah yang sama. Jelas bahwa setiap simpul kecuali akar mempunyai ayah yang tunggal.

Simpul L disebut keturunan (atau *descendant*) dari simpul N dan sebaliknya N disebut moyang (atau *ancestor*) dari L, bila L diperoleh dari percabangan berturut-turut dari N. Khususnya, L disebut keturunan kiri atau kanan dari N tergantung apakah L terletak pada subpohon kiri atau kanan dari N.



Gambar 7.5. *Contoh untuk terminologi*

Pada Gambar 7.5 di atas, K misalnya adalah keturunan kanan dari D, tetapi bukan keturunan dari F, E ataupun M. Simpul G adalah ayah dari K dan L. Di sini K dan L adalah bersaudara, masing-masing anak kiri dan kanan dari G. Selain terminologi hubungan keluarga di atas, terminologi dari teori *graph* juga banyak digunakan. Garis yang ditarik dari simpul N ke suksesor disebut ruas dan sederetan ruas yang berturutan disebut jalur atau *path*. Sebuah jalur yang berakhir pada daun (simpul terminal) disebut cabang.

Sebagai contoh, pada Gambar 7.5 tersebut, garis AD, ataupun GL adalah contoh ruas. Sedangkan barisan ruas (AD, DG, GL) adalah jalur dari simpul A ke simpul L. Jalur ini sekaligus merupakan cabang, karena berakhir di Simpul terminal (daun) L. Jalur (AD, DG) bukan sebuah cabang. Setiap simpul dari pohon mempunyai nomor tingkat (*level*). Akar R dari pohon T mempunyai tingkat nol. Simpul lain, mempunyai tingkat lebih besar dari tingkat ayahnya. Di sini satu simpul yang mempunyai tingkat dikatakan berada pada satu generasi.

Kembali sebagai contoh, Pohon Binar pada Gambar 7.5 di atas, simpul A mempunyai tingkat 0. Simpul-simpul D dan E berada pada generasi dengan tingkat 1. Generasi berikutnya terdiri atas simpul-simpul F, G dan M dengan tingkat 2. Generasi terakhir, diisi oleh simpul K dan simpul L dengan tingkat = 3.

Kedalaman atau ketinggian (*depth* atau *height*) dari pohon binar T didefinisikan sebagai banyak maksimum simpul, dari cabang di T. Dengan kata lain, panjang maksimum jalur di T ditambah 1. Ketinggian juga sama dengan tingkat tertinggi dari simpul ditambah 1. Pohon pada Gambar 7.5 di atas mempunyai ketinggian 4. Perhatikan bahwa cabang (AD, DG, GK) ataupun (AD, DG, GL) mengandung simpul dengan jumlah maksimum,

yakni = 4. Cabang yang lain mengandung simpul yang lebih sedikit, cabang (AE, EM) serta (AD, DF) misalnya hanya mengandung 3 simpul.

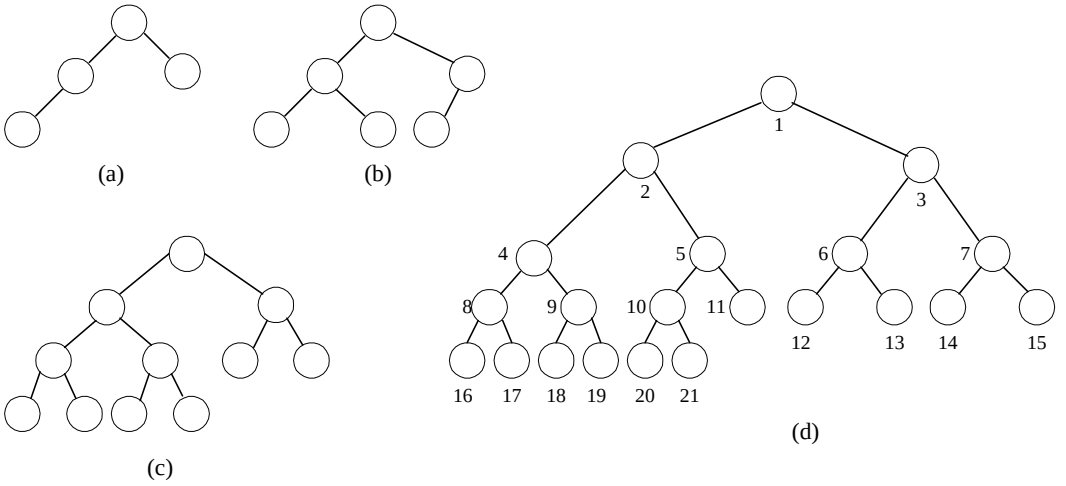
7.4 POHON BINAR LENGKAP

Setiap simpul dari pohon binar paling banyak mempunyai dua anak. Dapat kita lihat bahwa simpul akar bertingkat = 0, hanya terdiri 1 simpul. Anaknya adalah bertingkat =1, terdiri paling banyak 2 simpul. Demikian seterusnya, simpul dengan tingkat = r paling banyak ada 2^r.

Suatu pohon binar T dikatakan lengkap atau *complete*, bila setiap tingkatnya, kecuali mungkin tingkat yang terakhir, mempunyai semua simpul yang mungkin, yakni 21 simpul untuk tingkat ke-r, dan bila semua simpul pada tingkat terakhir muncul di bagian kiri pohon.

Jadi pohon binar lengkap dengan n simpul, T_n adalah tunggal (dalam hal ini dengan mengabaikan label simpul). Gambar 7.6 berikut menggambarkan berturut-turut T_4, T_6, $T_{11,}$ dan T_{21}.

Dapat dicatat bahwa beberapa buku mendefinisikan pohon binar lengkap harus mengandung semua simpul untuk semua tingkat pohon binar menurut pendefinisian kita di atas disebut pohon binar hampir lengkap atau *almost complete.*



Gambar 7.6. *Pohon binar lengkap dan pohon hampir lengkap*

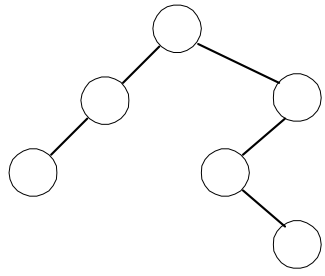
Kita dapat memberi label pohon binar lengkap menggunakan *integer* 1, 2, ..., n dari kiri ke kanan generasi demi generasi seperti pada Gambar 7.6 (d) di atas. Pemberian label seperti ini mempermudah kita untuk mengetahui ayah serta anak dari suatu simpul dalam pohon binar lengkap. Dalam hal ini anak kiri dan anak kanan dari simpul K adalah berturut-turut 2*K dan 2*K+1. Sedangkan ayah dari K adalah INT(K/2). Notasi INT(P) adalah *integer* terbesar yang lebih kecil atau sama dengan P. Jadi INT(3.8) = 3, INT(15/2) = 7, INT(4) = 4, dan sebagainya. Sebagai contoh, anak kiri dan anak kanan dari simpul 7 adalah simpul 14 dan simpul 15, sedangkan ayahnya adalah simpul INT(7/2) = 3.

Ketinggian dari pohon binar lengkap T_n diberikan oleh rumus INT(^{2}log n) +1. Sebagai contoh, ketinggian dari T_4 adalah INT(^{2}1og 4) + 1 = 3, ketinggian dari T_{21} = INT(^{2}log21) = 5 dan sebagainya. Nilai ketinggian dari T_n relatif kecil dibandingkan nilai n yang bersangkutan. Lihat misalnya ketinggian dari T_n untuk n satu juta simpul, ketinggiannya = 21.

7.5 POHON-2

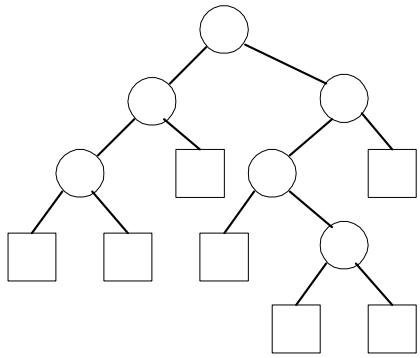
Pohon binar T dikatakan pohon-2 atau pohon binar yang dikembangkan (*extended binary tree*) bila setiap simpul mempunyai 0 atau 2 anak. Dalam kasus ini, simpul dengan 2 anak disebut simpul internal, sedangkan simpul tanpa anak disebut simpul eksternal. Dalam diagramnya, seringkali diadakan pembedaan antara simpul internal dan eksternal. Simpul internal digambar sebagai lingkaran, sedangkan simpul eksternal sebagai bujur sangkar.

Istilah "pohon binar yang dikembangkan" datang dari pengoperasian berikut. Perhatikan pohon binar pada Gambar 7.7 berikut. Ia dapat dikembangkan menjadi pohon-2 dengan mengganti subpohon hampa dengan simpul baru.



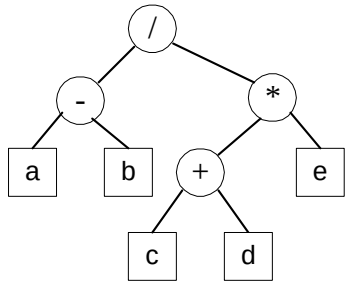
Gambar 7.7. *Pohon binar yang akan dijadikan pohon-2*

Hal ini terlihat pada Gambar 7.8. Di sini simpul yang lama adalah menjadi simpul internal, sementara itu simpul baru menjadi simpul eksternal dari pohon-2.



Gambar 7.8. *Extended binary tree*

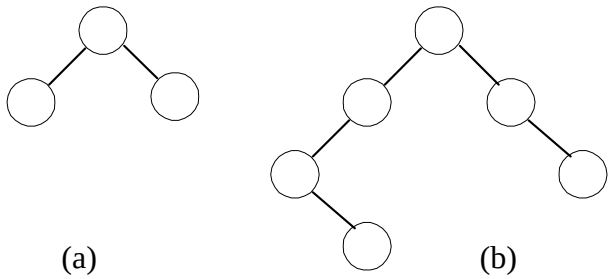
Sebuah pemakaian penting dari pohon-2 adalah untuk menyajikan suatu ekspresi aritmetik yang mengandung operasi binar. Di sini simpul eksternal menyajikan *operand* (variabel) sedangkan simpul internal menyajikan operator yang bekerja terhadap ke dua subpohonnya. Sebagai contoh adalah pohon-2 pada Gambar 7.9 berikut yang menyajikan ekspresi (a-b) / ((c+d) *e)



Gambar 7.9. *Pohon binar (a-b) / ((c+d) *e)*

7.6 POHON KETINGGIAN SEIMBANG

Pohon binar yang mempunyai sifat bahwa ketinggian subpohon kiri dan subpohon kanan dari pohon tersebut berbeda paling banyak 1, disebut pohon ketinggian seimbang atau *height balanced tree* (HBT). Gambar 7.10 menunjukkan contoh-contoh HBT.



Gambar 7.10. *Pohon ketinggian seimbang*

7.7 KETINGGIAN MINIMUM DAN MAKSIMUM POHON BINAR

Untuk menentukan ketinggian minimum, jika diberikan banyaknya simpul N, dapat digunakan rumus :

$$H_{min} = INT(^2log\ N) + 1$$

Sebagai contoh, untuk N = 80

$$H_{min} = INT(^2log\ 80) + 1$$

$$= 7$$

Banyaknya simpul pohon binar merupakan ketinggian maksimum Pohon binar tersebut.

7.8 PENYAJIA.N POHON BINAR DALAM MEMORI

Kita dapat menyajikan suatu pohon binar T dalam memori dengan dua cara. Cara pertama adalah penyajian kait (*link*). Cara ini biasa digunakan. Ia analog dengan cara *list* berkaitan (*linked list*) ketika disajikan dalam memori. Cara kedua adalah dengan menggunakan sebuah *array* tunggal disebut penyajian sekuensial dari T.

Kebutuhan utama yang harus dipenuhi pada setiap penyajian dari T adalah bahwa seseorang dapat mempunyai akses langsung ke akar R dan T, dan bila diberikan sembarang simpul N, seseorang harus dapat akses langsung ke anak dari N.

7.8.1 PENYAJIAN KAIT

Kalau tidak dinyatakan lain, suatu pohon binar T akan disimpan dalam memori secara penyajian kait. Penyajian ini menggunakan tiga *array* sejajar INFO, LEFT, dan RIGHT, serta sebuah variabel penuding ROOT. Masing-masing simpul N dari pohon T berkorespondensi dengan suatu lokasi K, sedemikian sehingga :

(1) INFO[K] berisi data pada simpul N

(2) LEFT[K] berisi lokasi dari anak kiri simpul N

(3) RIGHT[K] berisi lokasi dari anak kanan simpul N.

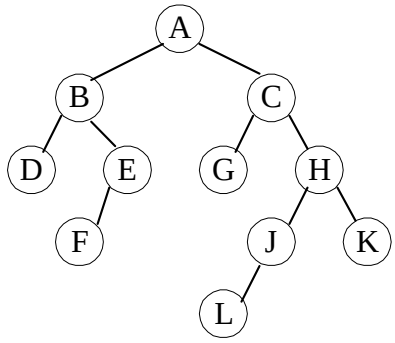
ROOT akan berisi lokasi dari akar R dari pohon T. Jika suatu subpohon hampa, maka penuding yang bersangkutan akan berisi harga nol. Jika suatu pohon T sendiri hampa, maka ROOT akan berisi harga nol.

Dapat dicatat bahwa simpul dari pohon bisa saja berisi lebih dari satu informasi. Pada prakteknya simpul biasanya berisi sebuah *record*. Jadi INFO sebenarnya berupa *linear array* dari *record* ataupun berupa sebuah koleksi *array* sejajar. Selain itu, karena suatu simpul boleh diselipkan sebagai simpul baru pohon, atau simpul lama boleh dihapus dari pohon, kita juga secara implisit mengasumsikan bahwa lokasi hampa dalam *array* INFO, LEFT dan RIGHT membentuk sebuah *list* berkaitan dengan penuding AVAIL. Kita biasanya memisalkan *array* LEFT berisi penuding untuk *list* AVAIL.

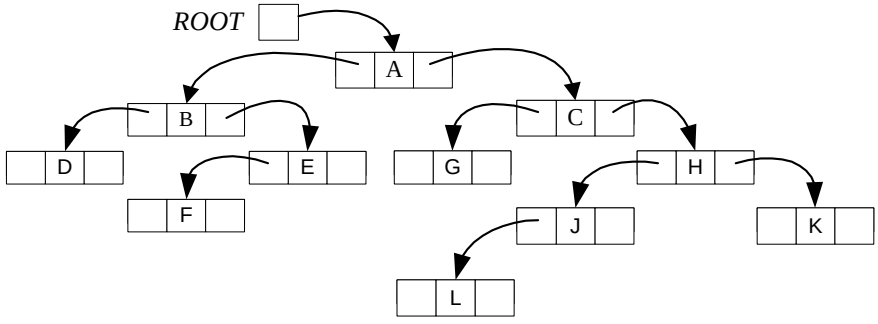
Untuk menuding ke alamat yang invalid (subpohon hampa), dipilih penuding nol yang dinyatakan sebagai NULL. Kenyataannya dalam praktek, kita gunakan 0 atau bilangan negatif sebagai isi dari NULL.

Sebagai contoh, Gambar 7.12 menggambarkan skema dari penyajian kait pohon binar Gambar 7.11. Terlihat bahwa setiap simpul digambar terdiri atas tiga *field*. Terlihat juga di sini, subpohon hampa digambar berlabel x untuk mengisi penuding nol. Selanjutnya

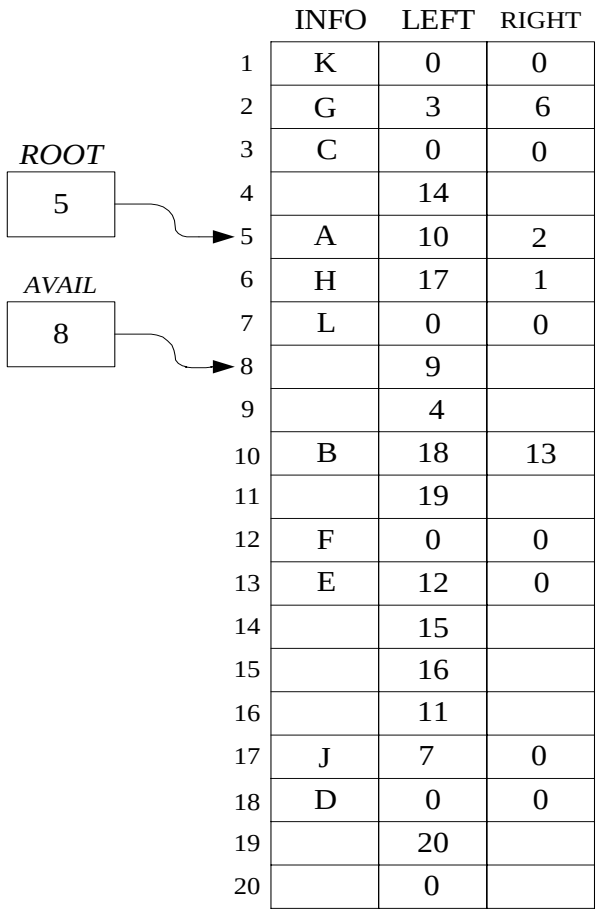
Gambar 7.13 menunjukan bagaimana penyajian kait pohon binar yang bersangkutan. Sebagai contoh, misalkan diketahui berkas personalia suatu pe-rusahaan kecil yang berisi data 9 pegawainya dengan *fields :* NAME, SOCIAL-SECURITY-NUMBER (SSN), SEX, serta SALARY. Berkas tersebut disimpan dalam memori sebagai pohon binar, seperti terlihat pada Gambar 7.15



Gambar 7.11. *Contoh sebuah pohon binar*

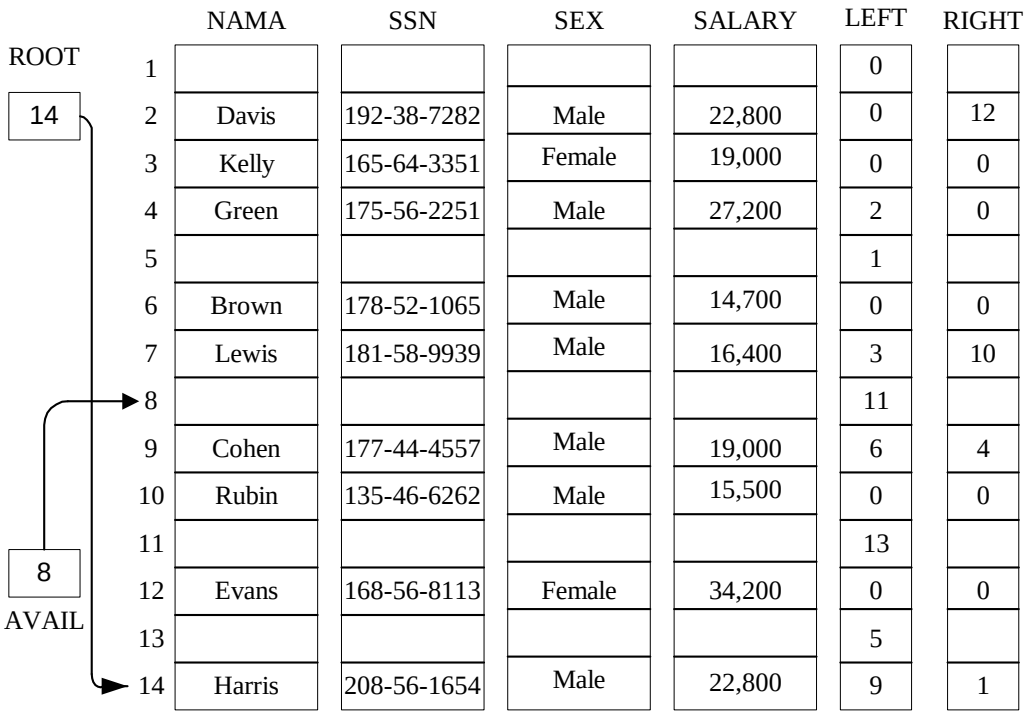


Gambar 7.12. *Penggambaran skema penyajian kait pada pohon binar*



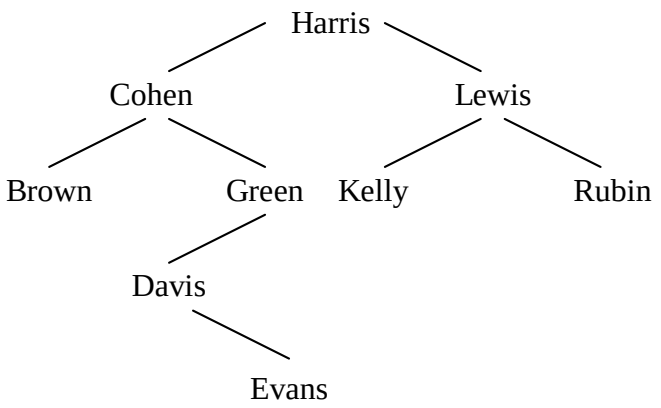
	INFO	LEFT	RIGHT
1	K	0	0
2	G	3	6
3	C	0	0
4		14	
5	A	10	2
6	H	17	1
7	L	0	0
8		9	
9		4	
10	B	18	13
11		19	
12	F	0	0
13	E	12	0
14		15	
15		16	
16		11	
17	J	7	0
18	D	0	0
19		20	
20		0	

Gambar 7.13. *Skema memori penyajian kait dari pohon binar*



Gambar 7.14. *Contoh penggunaan pohon binar pada records.*

Diagram pohon binar dari Gambar 7.14. dapat dilihat di Gambar 7.15. berikut ini :



Gambar 7.15. *Skema dari Gambar 7.14.*

Untuk memudahkan, kita menulis label dari simpul hanya berupa *field* kunci, NAME. Kita membentuk pohon pada Gambar 7.15 tersebut sebagai berikut :

(1) Harga dari ROOT = 14 menunjukkan bahwa *record* nomor 14 dengan NAME = "Harris" adalah akar dari Pohon.

(2) LEFT[14] = 9 menunjukan bahwa Cohen (*record* nomor 9) adalah anak kiri dari Harris, dan RIGHT[14] = 7 menunjukkan bahwa Lewis adalah anak kanan dari Harris.

Dengan mengulangi langkah (2) kita peroleh diagram pohon binar seperti Gambar 7.15. Ingat bila LEFT[N] atau RIGHT[N] bernilai 0 menandakan bahwa simpul N tidak mempunyai anak kiri /kanan.

7.8.2 PENYAJIAN SEKUENSIAL

Pandang pohon binar T yang lengkap atau hampir lengkap. Terdapat sebuah cara yang efisien untuk menyajikan T dalam memori. Penyajian ini disebut penyajian sekuensial. Penyajian ini hanya menggunakan sebuah *linear array* TREE sebagai berikut :

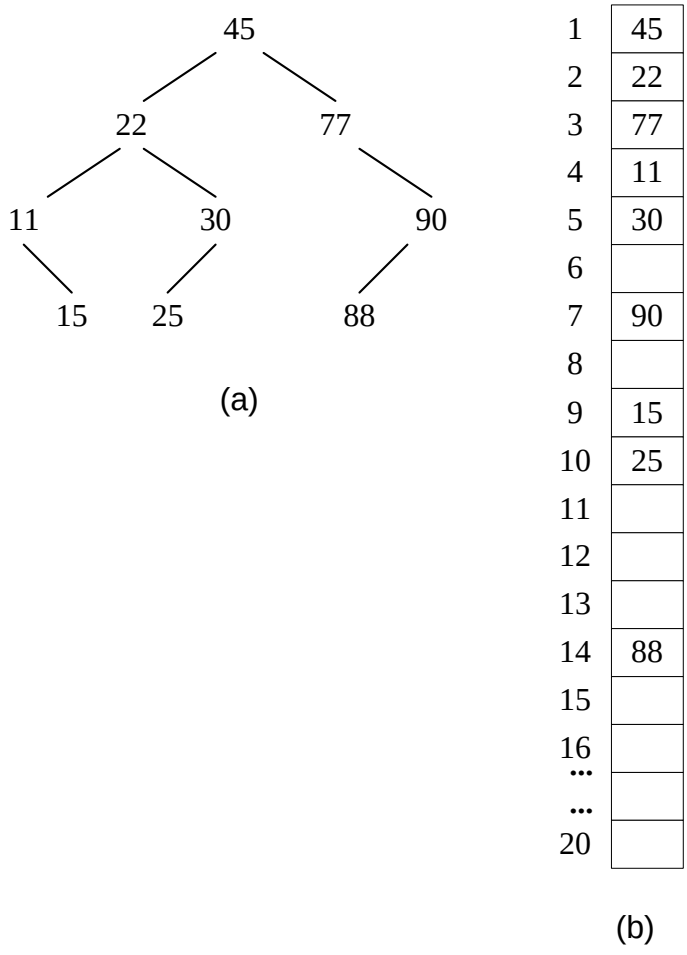
(1) Akar R dari pohon T tersimpan sebagai TREE[i]

(2) Jika simpul N menduduki TREE[K] maka anak kirinya tersimpan dalam TREE[2*K] dan anak kanannya dalam TREE[2*K+l]

Nilai penuding NULL, seperti penyajian kait, juga diperuntukkan menunjukkan subpohon hampa. Khusus bila TREE[i] = NULL, maka berarti Pohon T adalah pohon hampa. Berdasarkan kenyataan tersebut di atas, secara umum penyajian sekuensial ini menjadi tidak efisien. Penyajian sekuensial dari pohon binar pada Gambar 7.16(a) terlihat pada Gambar 7.16(b).

Dapat kita lihat bahwa penyajian membutuhkan 14 lokasi dalam *array* TREE, meskipun T hanya mempunyai 9 simpul. Kenyataannya, bila kita memasukkan elemen nol sebagai suksesor dari simpul terminal, kita akan membutuhkan TREE[29] untuk suksesor kanan dari TREE[14].

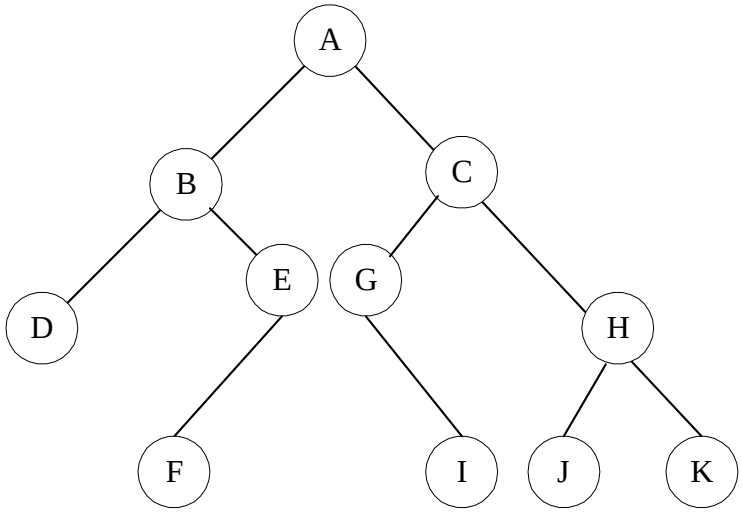
Berbicara secara umum, penyajian sekuensial dari pohon binar dengan ke-tinggian d akan membutuhkan *array* dengan banyak elemen mendekati 2^{n+1}. Hal ini tidak efisien, kecuali tentunya untuk pohon binar T yang lengkap atau hampir lengkap. Sebagai contoh, pohon binar T pada Gambar 7.11 dengan 11 simpul dan ketinggian = 5 membutuhkan *array* dengan banyak elemen mendekati $2^{5+1} = 64$ elemen.



Gambar 7.16 *Penyajian sekuensial dari pohon binar*

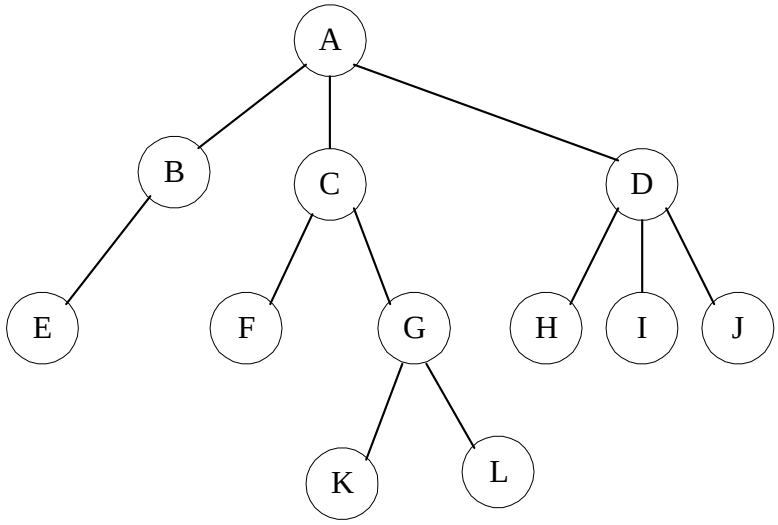
7.9 PENYAJIAN POHON UMUM SECARA POHON BINAR

Kalau kita mempunyai sebuah struktur pohon yang umum (*general tree*), maka ada sebuah algoritma yang dapat menyajikannya secara pohon binar. Kita ingat kembali bahwa pohon binar selalu terdiri atas paling banyak dua subpohon, yakni subpohon kiri dan subpohon kanan. Pendefinisian ini berlaku secara rekursif. Gambar 7.17 merupakan contoh dari pohon binar.



Gambar 7.17. *Pohon binar dengan 11 simpul*

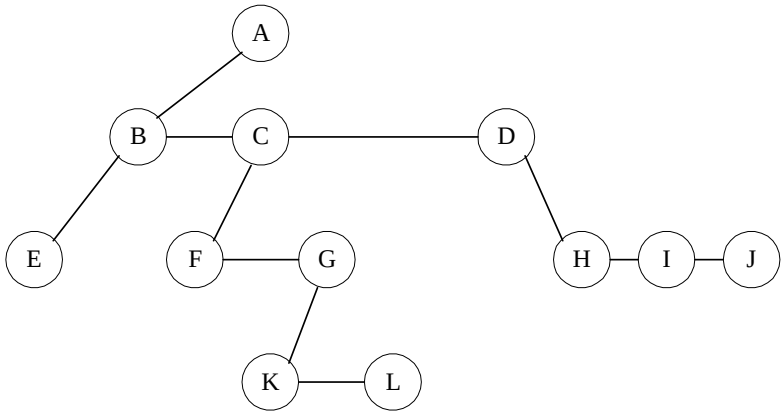
Pada pohon binar dapat kita lihat bahwa setiap simpul selalu mempunyai 0, 1 atau 2 anak, tak lebih dari itu. Gambar 7.18 menunjukkan sebuah contoh pohon umum, yang bukan pohon binar.



Gambar 7.18. *Contoh pohon umum*

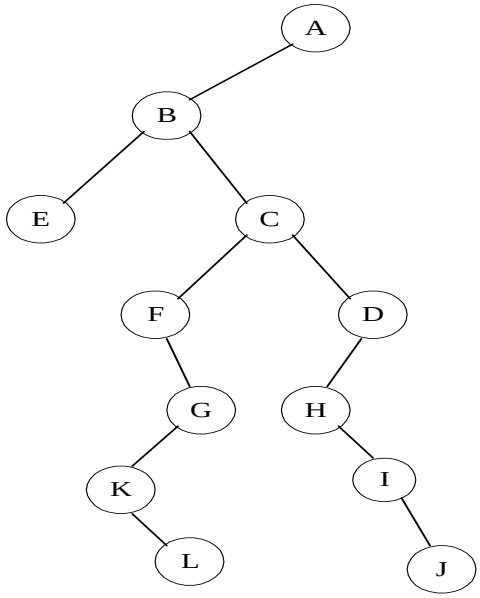
Pohon pada Gambar 7.18 tersebut bukan pohon binar, karena simpul D mempunyai 3 anak, yakni simpul-simpul H, I, dan J. Algoritma yang akan kita gunakan untuk menyajikan pohon umum secara pohon binar terdiri atas 2 langkah. Pertama, kita tambahkan ruas (*edge*) baru, menghubungkan 2 Simpul bersaudara yang berdampingan, lalu kita hapus ruas dari simpul ayah (*parent*) ke simpul anak bersaudara tersebut, kecuali ruas ke simpul anak paling kiri. Langkah kedua, kita melakukan rotasi sebesar 45^o, searah jalannya putaran jarum jam terhadap pohon hasil langkah pertama tadi.

Sebagai contoh, Gambar 7.19a dan 7.19b menunjukkan penggunaan algoritma di atas terhadap pohon umum pada Gambar 7.18. Gambar 7.19a adalah pelaksanaan langkah pertama dan Gambar 7.19b adalah pelaksanaan langkah kedua, sehingga diperoleh pohon binar yang diinginkan.



Gambar 7.19a *Langkah pertama pembentukan pohon binar*

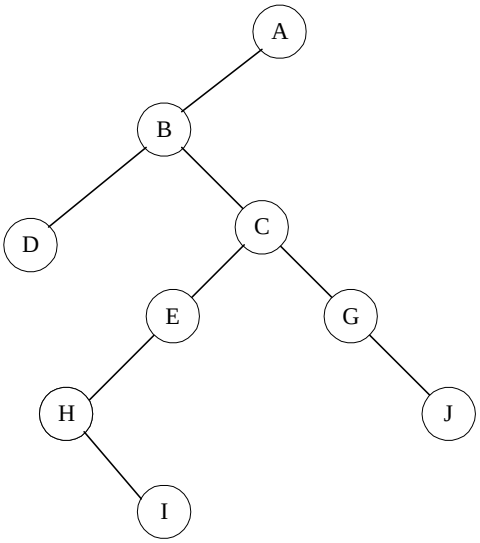
Gambar 7-19a terjadi setelah pada Gambar 7.18 ruas mendatar (B,C), (C,D), (F,G), (H,I), (I,J) kita tambahkan. Mereka menghubungkan dua simpul bersaudara yang berdampingan. Kemudian dilakukan penghapusan ruas (A,C), (A,D), (C,G), (D,I), (D,J) dan (G,L).



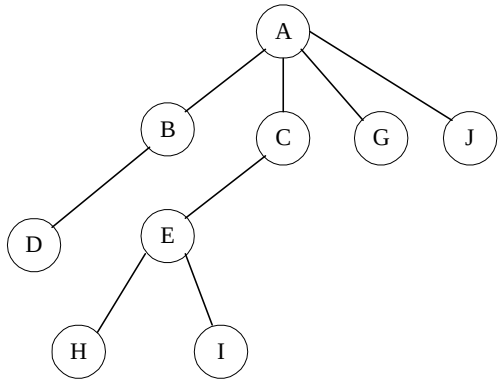
Gambar 7.19b *Langkah kedua pembentukan pohon binar*

Gambar 7-19b terjadi setelah perotasian pohon pada Gambar 7.19a. Dari hasil ini dapat kita lihat dan simpulkan bahwa ruas kiri pada pohon binar hasil merupakan penuding ke simpul anak pada 1 pohon umum semula. Sementara itu ruas kanan merupakan pertanda bahwa kedua simpul adalah saudara yang berdampingan pada pohon semula.

Dari kesimpulan kita tersebut, kita selalu dapat mengembalikan pohon binar secara penyajian pohon umum ke penyajian semula. Hal ini dapat kita lakukan dengan mudah seperti pohon binar pada Gambar 7.20 yang kita kembalikan ke pohon semula pada Gambar 7.21.



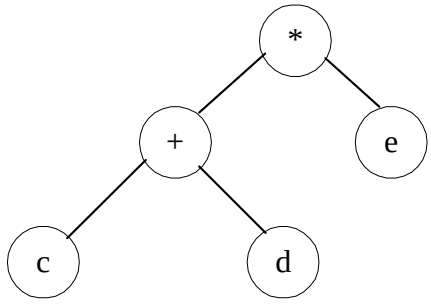
Gambar 7.20 *Pohon binar*



Gambar 7.21 *Pohon umum*

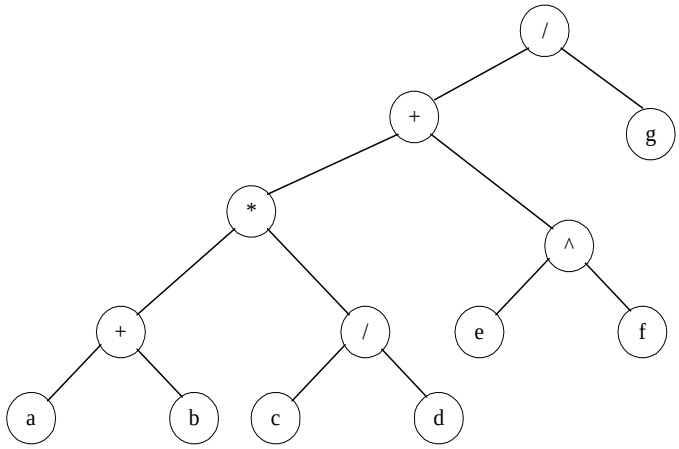
7.10 NOTASI *PREFIX*, *INFIX* DAN *POSTFIX* SERTA TRAVERSAL

Akan kita lihat bagaimana struktur pohon dipakai untuk menempatkan data, guna memudahkan pekerjaan cari (*search*). Pohon juga berguna untuk menyajikan koleksi data yang mempunyai struktur logik bercabang.



Gambar 7.22. *Penyajian notasi (c+d)*e*

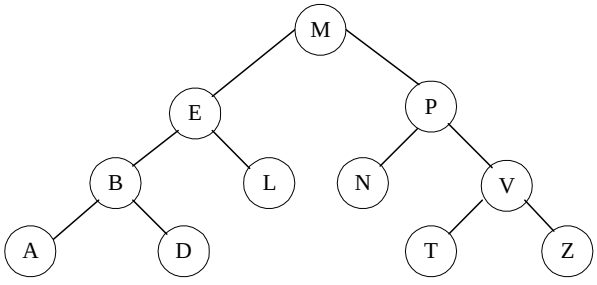
Sebagai contoh, perhatikan pohon binar pada Gambar 7.22 dan 7.23 yang menya-jikan ekspresi aritmetika (c+d)*e dan ((a+b)*(c/d)+(e^f))/g. Pada penyajian ini, masing-masing simpul yang bukan daun, mewakili operator, sedangkan subpohon kiri, dan kanannya merupakan *operand.*



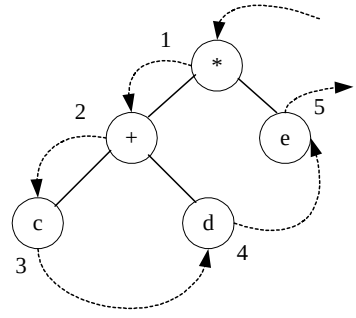
Gambar 7.23. *Penyajian notasi ((a+b) * (c/d) + (e^f)) / g*

Contoh pohon pada Gambar 7.24 menyajikan koleksi elemen data yang disusun sedemikian rupa, yakni bila K adalah label/ nama suatu simpul, maka label dari semua simpul subpohon kirinya lebih kecil atau sama dengan K (secara alfabetik), dan label semua simpul subpohon kanannya lebih besar dari K.

Proses kunjungan dalam pohon, dengan setiap simpul hanya dikunjungi tepat satu kali disebut traversal. Ketika dilakukan traversal pohon, koleksi simpul dari pohon terlihat satu persatu. Hasil dari traversal pohon adalah suatu untai simpul pohon yang urut secara *linear*. Suatu simpul dikatakan dikunjungi, bila simpul tersebut kita masuk-kan ke dalam urutan *linear* tersebut.



Gambar 7.24 *Pohon binar*



Gambar 7.25 *Traversal preorder dari Gambar 7.22*

Tiga kegiatan yang terdapat dalam traversal pohon binar adalah :

(1) mengunjungi simpul akar (*root*)

(2) melakukan traversal subpohon kiri dan

(3) melakukan traversal subpohon kanan.

Kita mengenal tiga macam traversal pohon, yang berbeda satu dengan yang lain dari cara pengurutan ketiga kegiatan di atas. Ketiga traversal tersebut adalah traversal *pre-order*, *in-order* dan *post-order*. Pada traversal *pre-order* dilakukan berturut-turut :

(1) Kunjungi simpul akar

(2) Lakukan traversal subpohon kiri secara *pre-order*

(3) Lakukan traversal subpohon kanan secara *pre-order*

Perhatikan bahwa proses berlangsung secara rekursif. Kalau kita lakukan traversal *pre-order* terhadap pohon pada Gambar 7.22, 7.23 dan 7.24 maka berturut-turut diperoleh deretan urutan *linear*

Gambar 7.22 : c * d + e

Gambar 7.23 : a / b + c * d + e / f ^ g

Gambar 7.24 : M E B A D L P N V T Z

Untuk jelasnya perhatikan Gambar 7.25, 7.26 dan 7.27 dengan arah dari traversal (dinyatakan dengan garis putus-putus).:

Traversal *in-order* berbeda urutannya, yakni :

(1) Lakukan traversal subpohon kiri secara *in-order*

(2) Kunjungi simpul akar

(3) Lakukan traversal subpohon kanan secara *in-order*

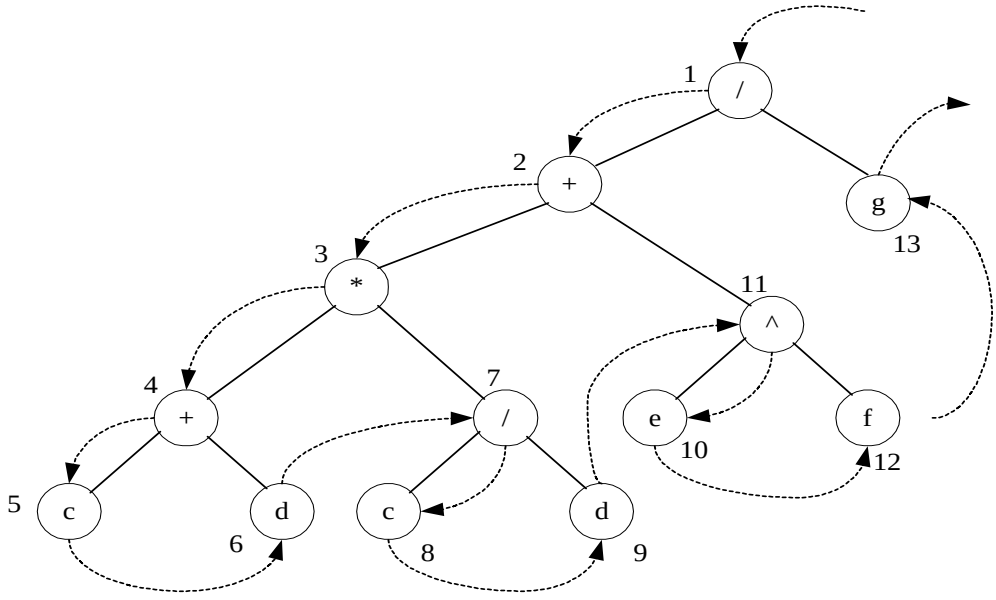
Kalau kita lakukan traversal *in-order* terhadap pohon pada Gambar 7.22, 7.23 dan 7.24 maka diperoleh deretan urutan *linear*, berturut-turut :

Gambar 7.22 : (c + d) * e

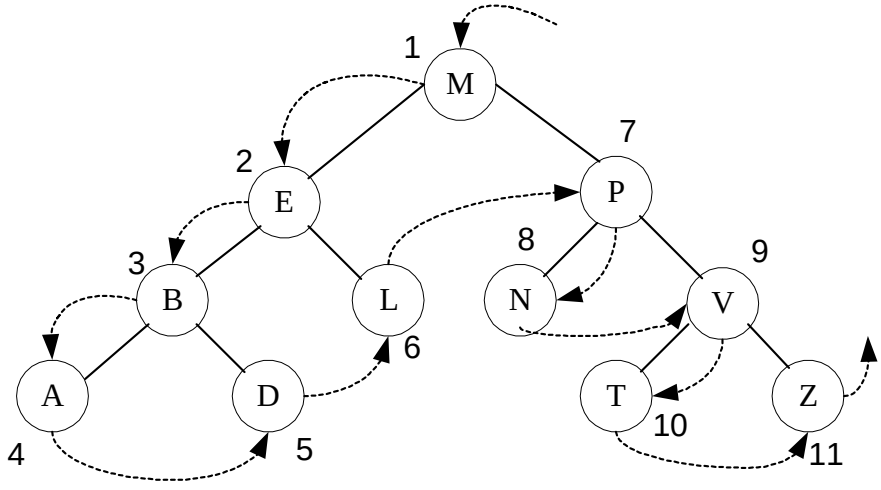
Gambar 7.23 : ((a + b) * (c / d) + (e ^ f)) / g

Gambar 7.24 : A B D E L M N P T V Z

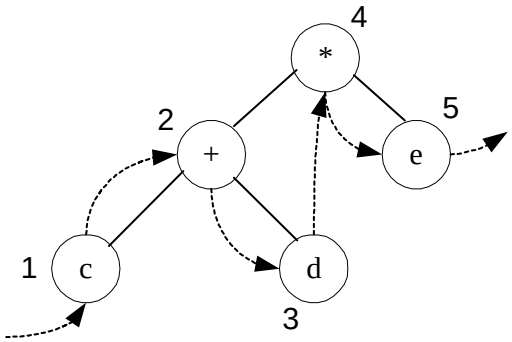
Gambar 7-28(a) menggambarkan traversal secara *in-order* pada pohon di Gambar 7.22. Sisanya dapat anda coba sendiri.



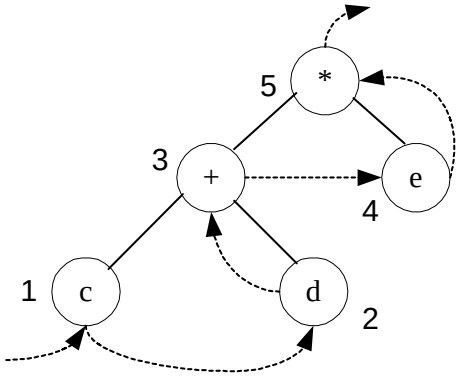
Gambar 7.26. *Traversal pre-order dari Gambar 7.23*



Gambar 7.27. *Traversal pre-order dari Gambar 7.24*



Gambar 7.28(a) *Traversal in-order*



Gambar 7.28(b) *Traversal post-order*

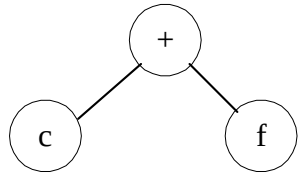
Tanda Kurung pada Notasi *Infix*

Lihat kembali gambar 7.28(a) di atas, kunjungan secara *in-order* pada pohon tersebut seharusnya menghasilkan : c + d * e. Akan tetapi, hal itu akan berbeda hasilnya jika *operand-operand* aritmetika tersebut kita beri nilai. Contoh, c kita beri nilai 2, b = 3 dan e = 4. Maka hasil perhitungan 2 + 3 * 4 adalah 14. Padahal seharusnya (2 + 3) * 4 = 20.

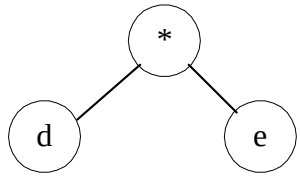
Mari kita lakukan langkah-langkah pembuatan pohon binar (1) c + d * e dan (2) (c + d) * e. Apakah kedua notasi tersebut akan memiliki struktur pohon yang sama ? Kita mulai dengan pohon binar (1).

c + d * e; maka operasi yang akan dilakukan pertama kali adalah d * e (sesuai kaidah derajat operasi matematika). Bila d * e kita anggap sebagai f, maka notasi semula bisa disederhanakan menjadi c + f.

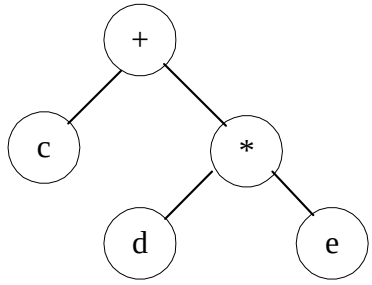
c + f; struktur pohonnya adalah :



Kita ketahui bahwa f adalah d * e yang struktur pohonnya adalah :

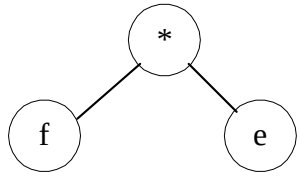


Kita ganti f di atas menjadi d * e, sehingga hasil akhirnya menjadi :

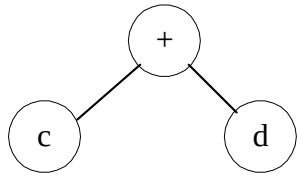


Sekarang, kita kerjakan pohon kedua (c + d) * e. Operasi yang pertama kali dilakukan adalah (c + d). Jika (c + d) kita misalkan f, maka ekspresinya akan menjadi f * e.

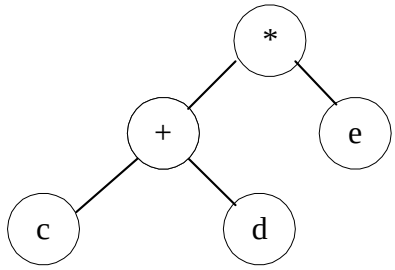
f * e akan digambarkan sebagai :



Kita tahu bahwa f adalah (c + d) yang struktur pohonnya adalah :



Maka, ketika kita masukkan ke posisi sebenarnya, struktur pohon akhirnya adalah :



Bisa Anda lihat, bahwa kedua pohon di atas berbeda strukturnya. Hal yang rumit adalah menuliskan notasi *infix* ketimbang *prefix* dan *postfix* karena pada notasi *infix* harus memperhatikan tanda kurung, sedang yang lain tidak memerlukannya. Tanda kurung itu digunakan untuk "mengurungi" setiap *substree* yang ada.

Perhatikan kembali Gambar 7.23 di atas. Notasi yang dihasilkan adalah :

Substree paling kiri : (a + b)

Substree di sebelahnya : (c / d)

Digabung menjadi (a + b) * (c / d)

Substree berikutnya : (e ^ f)

Digabung lagi menjadi ((a + b) * (c / d)) * (e ^ f)

Digabung keseluruhannya menjadi : (((a + b) * (c / d)) * (e ^ f)) / g

Operasi itu akan menghasilkan nilai yang berbeda jika tidak digunakan tanda kurung : a + b * c / d * e ^ f / g.

Traversal yang ketiga, yakni traversal *post-order* memakai urutan :

(1) Lakukan traversal subpohon kiri secara *post-order*

(2) Lakukan traversal subpohon kanan secara *post-order*

(3) Kunjungi simpul akar.

Traversal *post-order* terhadap Pohon pada Gambar 7-22, 7-23 dan 7-24 menghasilkan:

Gambar 7.22 : c + d * e

Gambar 7.23 : a + b / c * d ^ e + f / g

Gambar 7.24 : A D B L E N T Z V P M

Terlihat bahwa hasil yang diperoleh dari traversal pohon yang menyajikan ekspresi aritmetika, merupakan ekspresi aritmetika secara notasi *prefix, infix* serta *postfix*. Hanya di sini urutan operasi berdasarkan hirarki operator dan tanda kurung yang diberikan tidak terjaga pada traversal *in-order*. Keunggulan *post-order*, selain tidak memerlukan tanda kurung, juga dapat lebih mudah dikomputasi.

Operator selalu didahului oleh dua *operand*. Namun traversal *in-order* menunjuk-kan keunggulannya pada traversal pohon pada Gambar 7.24. Hasil traversal adalah urut secara alfabetik. *Pre-order* banyak digunakan dalam sistem manajemen *database* seperti *Information Management System* (IMS) dari IBM. Traversal *pre-order* ekivalen dengan *hirarchi sequence order* dari IMS.

Jadi ketiga metode traversal tersebut sama-sama penting, sehingga perlu kita ketahui dengan baik. Juga dapat dicatat bahwa pemberian penuding (*pointer*) arah tra-versal, baik secara *pre-in* atau *post-order* sangat membantu. Pohon yang telah dilengkapi dengan penuding tersebut, seperti pada Gambar 7.25 sampai 7.28(b) berupa garis putus-putus, disebut pohon binar berbenang atau *threaded*.

Berikut ini adalah rangkuman langkah-langkah membuat pohon binar dari arit-metika *infix* :

1. Susun serta beri tanda kurung ekspresi yang dimaksud
2. Tentukan hirarki atau tingkatan dari operator aritmetika yang berlaku
3. Mulailah dari tingkat tertinggi, pembentukan pohon dari bawah ke atas.

Berikut ini beberapa contoh pohon binar untuk menyajikan ekspresi secara *infix*, *prefix*, dan *postfix* yang bersangkutan.

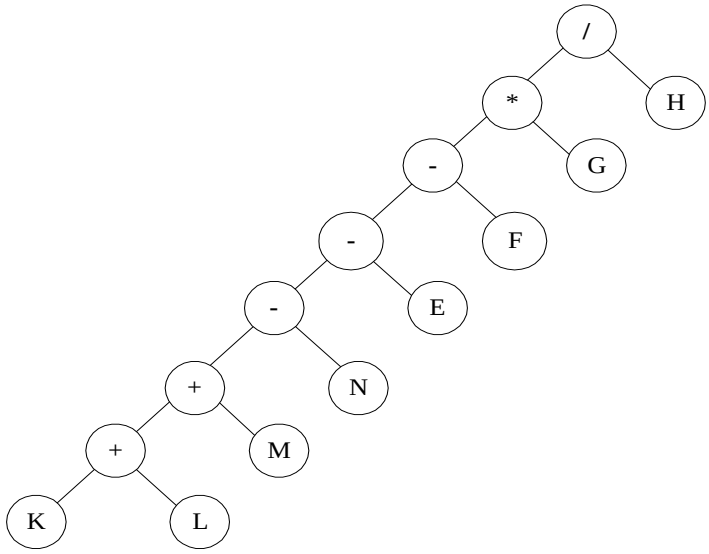
Contoh 7.1

Ekspresi *infix* : (K + L + M – N – E – F) * G / H mempunyai pohon binar seperti pada Gambar 7-29.

Notasi *infix* : (((((((K+L) +M) – N) – E) – F) *G) /H), hasil operasinya akan sama dengan notasi (K+ L + M – N – E – F) *G / H.

Prefix untuk Gambar 7.29 adalah : K / L * M – N – E – F + G + H

Postfix untuk Gambar 7.29 adalah : K + L + M – N – E – F * G / H



Gambar 7.29. *Notasi infix : (((((((K+L) +M) -N) -E) -F) *G) /H)*

Tampak bahwa *root tree* akan menjadi elemen pertama dalam *prefix* tetapi menjadi elemen terakhir dalam *postfix.*

Contoh 7.2

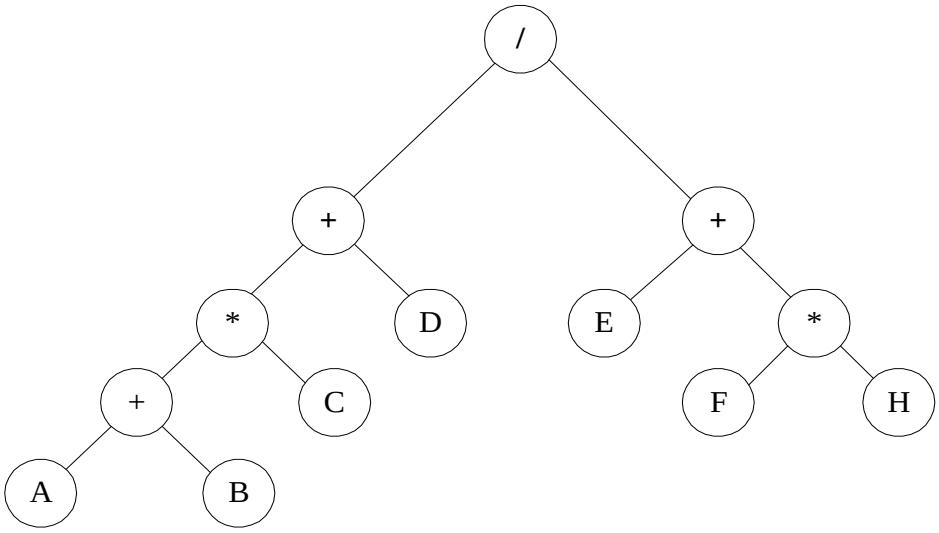
Ekspresi *infix* ((A + B) * C + D) / (E + F * H), mempunyai pohon binar seperti terlihat pada Gambar 7.30.

7.11 POHON CARI BINAR

Kali ini kita membahas sebuah struktur data bentuk khusus dari pohon binar yang teramat penting dalam ilmu komputer, yakni pohon cari binar (*binary search tree*). Ia penting untuk mengorganisasi koleksi besar data yang memerlukan kemampuan akses baik secara langsung maupun sekuensial.

Dalam struktur ini, seseorang dapat melakukan pencarian (*searching*) elemen dalam waktu pelaksanaan O(log n). Ia juga memungkinkan kita dengan mudah melakukan penyisipan (*inserting*) serta penghapusan (*deleting*) elemen. Pohon cari binar ini

dibandingkan dengan struktur lain seperti *linear array* terurut ataupun *list* berkaitan (*linked list*), mempunyai beberapa keuntungan.



Gambar 7.30. *Pohon binar : ((A + B) * C + D) / (E + F * H)*

*Prefix*nya adalah : / + * + A B C D + E * F H

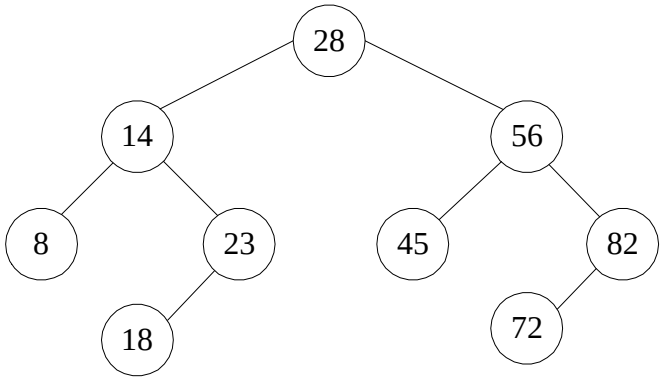
*Postfix*nya adalah : A B + C * D + E F H * + /

Pada *linear array* terurut, memang kita dapat melakukan cari dalam waktu pelaksanaan O(log n), namun proses penyisipan dan penghapusan elemen sukar serta mahal untuk dilakukan. Sementara itu pada *list* berkaitan; sungguhpun proses penyisipan dan penghapusan elemen lebih mudah dilaksanakan, namun proses cari secara *linear* membutuhkan waktu pelaksanaan yang kurang baik, yakni O(n).

Sungguhpun setiap simpul dari pohon cari binar boleh berisi *record* data, pendefinisian pohon cari binar tergantung pada *field* yang diberikan tertentu (*field* kunci) yang nilainya berbeda dan dapat diurutkan. Nilai tersebut dinamakan nama *record* atau nilai simpul.

Pandang T suatu pohon binar. Maka T disebut pohon cari binar (atau disebut juga pohon terurut binar) bila masing-masing simpul N dari T mempunyai sifat seperti berikut : "Nilai dari N selalu lebih besar dari setiap nilai simpul pada subpohon kiri dari N dan selalu lebih kecil dari setiap nilai simpul pada subpohon kanan dari N."

Definisi di atas menjamin bahwa traversal *in-order* terhadap pohon cari binar selalu menghasilkan untai yang terurut. Gambar 7.31 menggambarkan sebuah pohon cari binar T. Terlihat bahwa nilai simpul N pada pohon selalu lebih besar dari nilai setiap simpul di subpohon kanannya. Misalkan simpul bernilai 23 kita ganti nilainya dengan 35, maka T ternyata tetap merupakan pohon cari binar. Lain halnya bila 23 tersebut kita ganti dengan 40, T bukan lagi pohon cari binar.

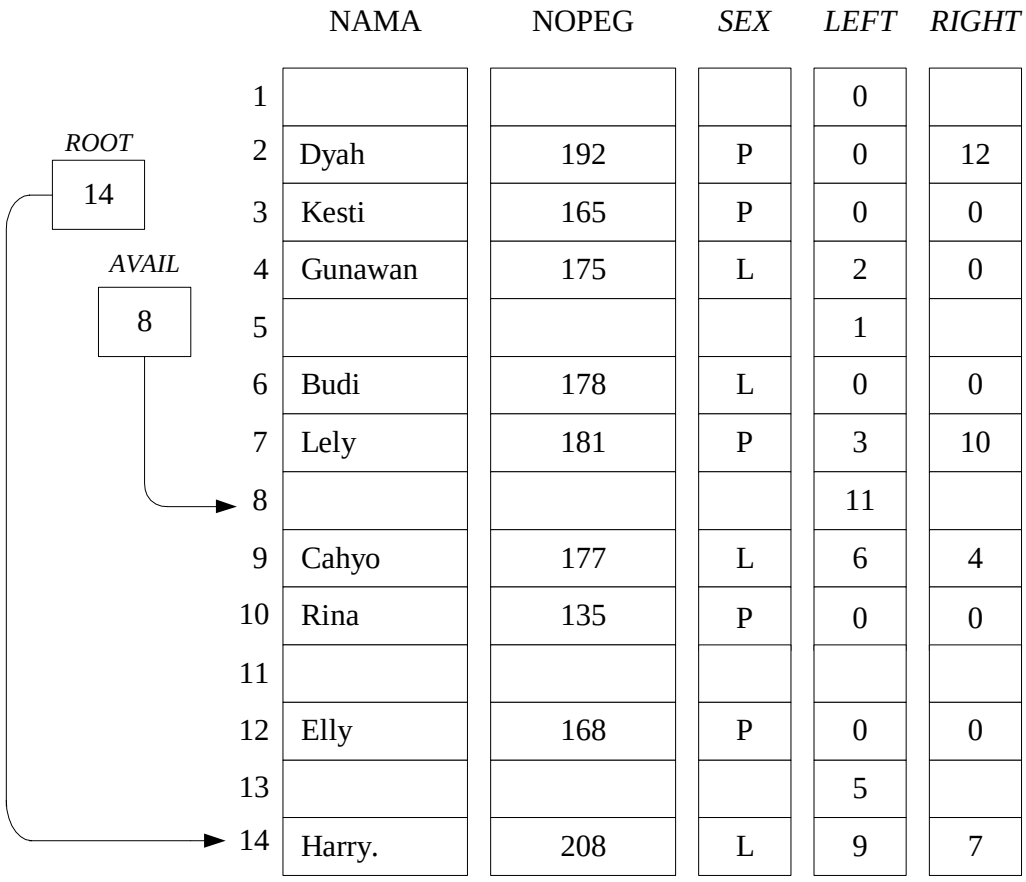


Gambar 7.31. *Contoh pohon cari binar*

Contoh berikut adalah pohon cari binar pada Gambar 7.33 untuk berkas (*file*) pada Gambar 7.32 dengan *field* kunci NAMA. Kalau *field* kunci diambil NOMOR-PEGAWAI, maka pohon binar tersebut bukan merupakan pohon cari binar.

Dapat dicatat bahwa dalam definisi pohon cari binar di atas, semua nilai/label simpul adalah berbeda. Kalau diijinkan terjadi adanya label yang sama; kita dapat melakukan sedikit modifikasi. Definisi pohon cari binar menjadi "*bila N adalah simpul dari pohon maka nilai semua simpul pada subpohon kiri dari N adalah lebih kecil atau sama dengan nilai simpul N dan nilai semua simpul pada subpohon kanan dari N adalah lebih besar dari nilai simpul N*".

Juga patut diingat, bahwa terdapat lebih dari satu pohon cari binar yang dapat dibentuk untuk menyajikan nama *record*. Perhatikan Gambar 7.34 yang menggambarkan beberapa pohon cari binar untuk nama *field* Adhini, Budiman, Cherry dan Doddy.



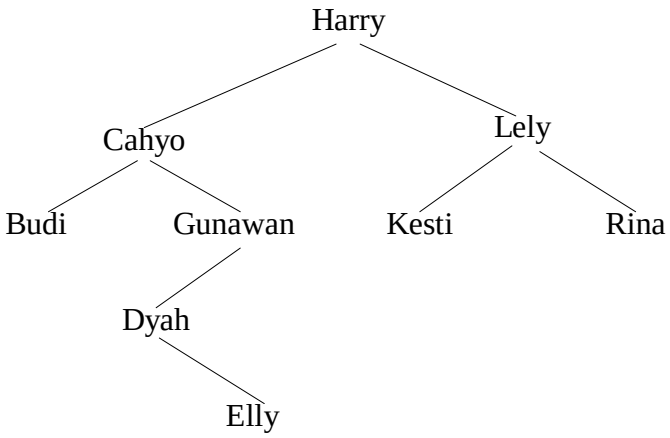
	NAMA	NOPEG	*SEX*	*LEFT*	*RIGHT*
1				0	
2	Dyah	192	P	0	12
3	Kesti	165	P	0	0
4	Gunawan	175	L	2	0
5				1	
6	Budi	178	L	0	0
7	Lely	181	P	3	10
8				11	
9	Cahyo	177	L	6	4
10	Rina	135	P	0	0
11					
12	Elly	168	P	0	0
13				5	
14	Harry.	208	L	9	7

Gambar 7.32 *Penyajian pohon cari binar dalam file.*

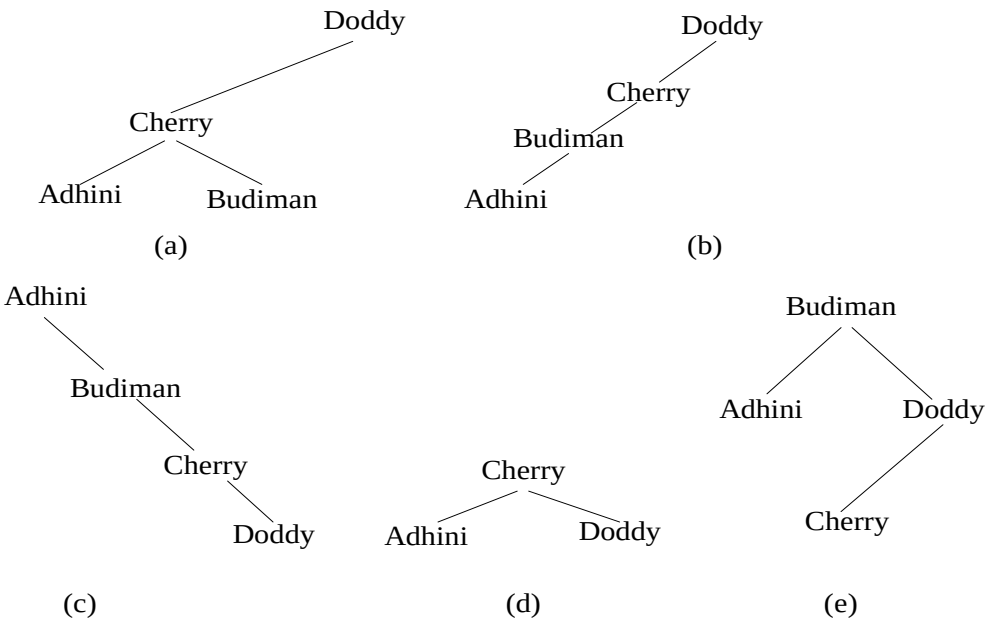
7.12 CARI DAN PENYISIPAN SIMPUL POHON CARI BINAR

Akan kita berikan sebuah algoritma sederhana untuk operasi cari dan penyisipan elemen pada pohon cari binar. Operasi penghapusan akan kita bicarakan pada sub-bab 7.13 nanti.

Pandang bahwa diberikan sebuah ITEM informasi. Algoritma di bawah ini akan menemukan lokasi dari ITEM dalam pohon cari binary T atau menyelipkan ITEM sebagai simpul baru dari T dalam posisi yang tepat.



Gambar 7.33 *Penggambaran pohon cari binar dari Gambar 7.32*



Gambar 7.34 *Contoh-contoh pohon cari binar*

ALGORITMA

Algoritma bekerja sebagai berikut :

(a) Bandingkan ITEM dengan simpul akar N dari pohon, jika ITEM < N proses subpohon kiri dari N, jika ITEM > N proses subpohon kanan dari N.

(b) Ulangi langkah (a) sampai hal berikut ditemui :

(1) Ditemukan simpul N sedemikian sehingga ITEM = N, dalam hal ini pencarian berhasil;

(2) Dijumpai subpohon hampa, ini menunjukkan bahwa pencarian tidak berhasil, dan kita selipkan ITEM mengisi subpohon yang hampa tadi.

Jadi dengan kata lain, kita memproses mulai dari simpul akar R turun sepanjang pohon T, sampai ditemukan atau dilakukan penyisipan ITEM sebagai daun dari T.

Sebagai contoh, pandang pohon cari binar pada Gambar 7.31. Misalkan diberikan ITEM = 20. Dengan algoritma di atas, kita kerjakan langkah-langkah sebagai berikut :

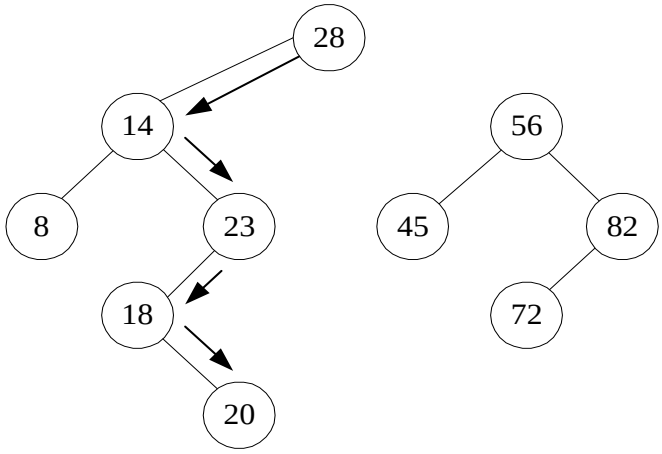
(1) Bandingkan ITEM = 20 dengan akar = 38. ITEM < akar, kita proses subpohon kiri dari 38 dengan akar = 14.

(2) Bandingkan ITEM = 20 dengan 14. Karena ITEM > akar, proses subpohon kanan dari 14. Akar dari subpohon tersebut = 23.

(3) Bandingkan ITEM = 20 dengan 23. Karena ITEM < akar, proses subpohon kiri dari 23. Akar dari subpohon tersebut = 18.

(4) Bandingkan ITEM = 20 dengan 18. Karena ITEM > akar, proses subpohon kanan dari 18. Subpohon tersebut adalah hampa, jadi algoritma selesai, diperoleh penyisipan 20 sebagai anak kanan dari 18.

Gambar 7.35 menunjukkan pohon cari binar setelah 20 diselipkan. Pada gambar itu juga ditunjukkan jalur dari simpul akar turun sampai ke elemen 20 yang diselipkan, selama berlangsungnya algoritma.



Gambar 7.35. *Penyisipan elemen 20*

Untuk lebih jelas, sekali lagi kita pakai algoritma di atas terhadap pohon cari binar pada Gambar 7.33. Sebagai ITEM diberikan Dyah. Kita peroleh langkah sebagai berikut :

(1) Bandingkan ITEM Dyah dengan akar pohon yakni Harry. Karena ITEM < akar, proses subpohon kiri dari Harry. Akar subpohon itu adalah Cahyo.

(2) Bandingkan ITEM Dyah dengan Cahyo. Karena ITEM > akar, proses subpohon kanan dari Cahyo. Akar subpohon tersebut adalah Gunawan.

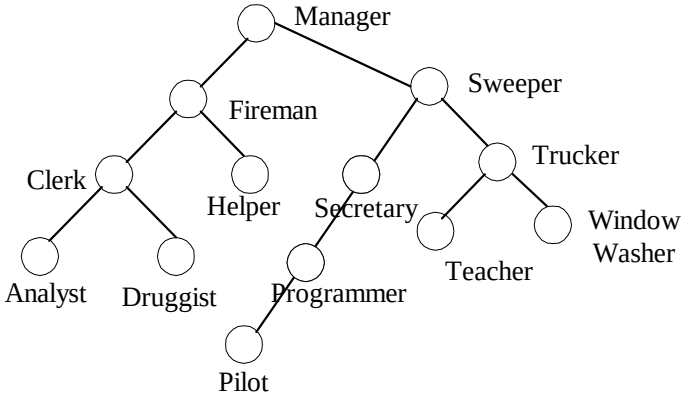
(3) Bandingkan Dyah dengan Gunawan. Karena Dyah < Gunawan, proses subpohon kiri dari Gunawan yang akarnya adalah Dyah

(4) Karena ITEM Dyah sama dengan akar maka algoritma selesai. Di sini diperoleh bahwa Dyah ada di dalam pohon cari binar tersebut.

7.13 PENGHAPUSAN SIMPUL POHON CARI BINAR

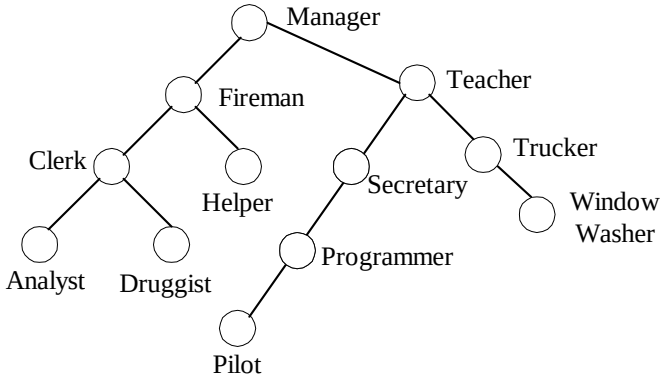
Setelah pada bagian yang lalu kita bicarakan operasi cari dan penyisipan, kali ini kita bicarakan operasi penghapusan (*deletion*) simpul pohon cari binar. Operasi penghapusan suatu simpul dari pohon cari binar bukan merupakan hal yang mudah. Pertama-tama kita harus tetapkan lebih dahulu simpul yang akan kita hapus, bila merupakan simpul daun, maka proses akan berlangsung dengan mudah, karena simpul daun tersebut akan dapat langsung kita hapuskan dari pohon cari binar yang bersangkutan.

Jika Simpul yang akan dihapuskan mempunyai hanya sebuah subpohon kanan, maka untuk menggantikan posisi simpul yang dihapuskan tersebut kita ambil simpul akar subpohon kiri dan subpohon kanan tersebut. Jika Simpul yang akan dihapuskan mempunyai dua buah subpohon kiri dan subpohon kanan, maka untuk menggantikan posisi dari simpul yang dihapus tersebut, kita tentukan simpul dari salah satu subpohon kiri atau subpohon kanan sedemikian sehingga bangun pohon yang terbentuk kembali, memenuhi sifat sebagai pohon cari binar. Perhatikan bangun pohon cari binar pada Gambar 7.36.



Gambar 7.36 *Contoh lain dari pohon cari binar*

Jika simpul yang akan dihapuskan hanya mempunyai satu subpohon, misalnya "Secretary", maka untuk menggantikan "Secretary" tersebut, dapat kita tempatkan "Programmer" secara langsung. Tetapi bila simpul yang akan dihapuskan mempunyai dua subpohon, contohnya "Sweeper," kita pilih simpul dengan nama tertentu dari subpohon kiri dan subpohon kanan, sehingga pohon cari yang terbentuk kembali memenuhi sifat sebagai "pohon cari binar." Misalkan kita pilih simpul dengan nama "Teacher" dan sekarang "Teacher" akan menempati posisi "Sweeper". Bangun pohon cari yang dibentuk kembali karena penghapusan "Sweeper" terlihat pada Gambar 7.37. Terlihat bahwa pohon cari binar yang terbentuk kembali tersebut memenuhi sifat pohon cari binar lagi.



Gambar 7.37 *Hasil penghapusan simpul "Sweeper"*

Prosedur untuk penghapusan suatu simpul dari pohon cari, dapat kita tulis sebagai berikut :

(1) Jika pohon hampa, maka penghapusan yang dilakukan gagal. Berhenti. Jika tidak, lakukan (2).

(2) Jika $n < R_i$ (akar), subpohon kiri dari R_i diselidiki sampai ditemukan simpul yang telah ditentukan untuk dihapus.

(3) Jika $n > R_i$, maka subpohon kanan dari R_i diselidiki sampai ditemukan simpul yang telah ditentukan untuk dihapus.

(4) Jika $n = R_i$ dan subpohon kiri dan subpohon kanan hampa, maka hapus R_i.

(5) Jika $n = R_i$ dan subpohon kirinya hampa, maka hapus R_i, kemudian ambil akar dari subpohon kanan untuk menggantikan posisi R_i. Pohon baru akan memenuhi sifat sebagai pohon cari lagi.

(6) Jika $n = R_i$ dan subpohon kanannya hampa, maka hapus R_i. Ambil akar dari subpohon kiri untuk menggantikan posisi R_i. Pohon baru akan memenuhi sifat sebagai pohon cari lagi.

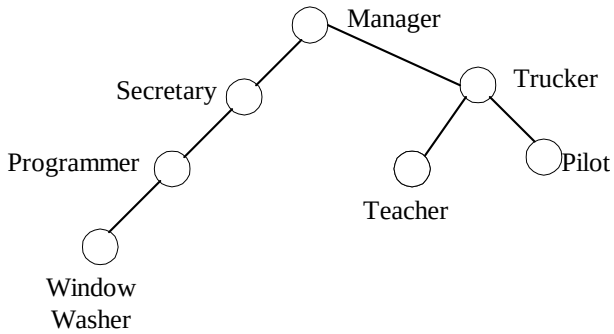
(7) Jika $n = R_i$ dan subpohon kanan tidak hampa, maka untuk menggantikan posisi R_i yang dihapus, kita tentukan suatu simpul, mungkin dari subpohon kiri atau mungkin dari subpohon kanan, sedemikian sehingga pohon yang terbentuk kembali memenuhi sifat sebagai pohon cari lagi.

Contoh 7.3

Perhatikan pohon cari binar pada Gambar 7.36 yang lalu. Misalkan simpul dengan nama "Sweeper" akan kita hapus dari pohon. Cara untuk menghapuskan "Sweeper" tersebut, sebagai berikut :

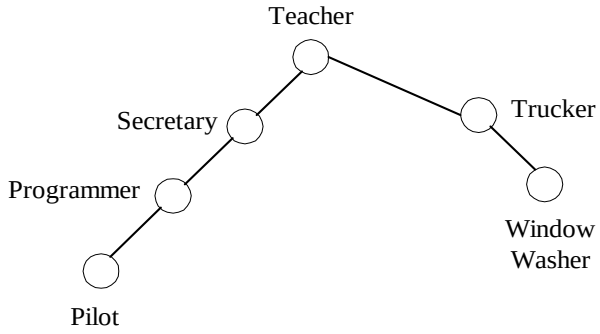
Langkah (1) : Kita periksa apakah pohon cari tersebut kosong atau tidak.

Langkah (2) : "Sweeper" > "Manager," berarti subpohon kanan dari akar R_i = "Manager" diselidiki. Subpohon kanannya terlihat pada Gambar 7.38.



Gambar 7.38.

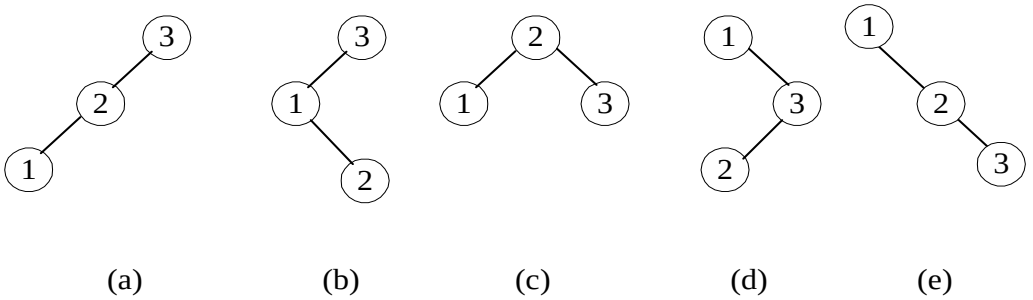
Langkah (3) : Akar R_i = "Sweeper" = n dan subpohon kiri serta subpohon kanan tidak kosong (lihat Gambar 7.39). Untuk menggantikan posisi "Sweeper", kita ambil simpul dengan nama "Teacher" dari subpohon kanan. Jika "Teacher" kita tempatkan di posisi "Sweeper", akan kita dapatkan kembali bangun pohon cari yang memenuhi sifat sebagai suatu pohon cari binar. Bangun pohon cari yang baru terlihat pada Gambar 7.39.



Gambar 7.39.

7.14 POHON CARI OPTIMAL

Telah kita ketahui dari bagian-bagian yang lalu bahwa suatu himpunan data ataupun himpunan *record* dapat disajikan dalam berbagai pohon cari binar. Besarnya upaya yang diperlukan untuk mencari suatu elemen tertentu pada sebuah pohon cari tergantung dari letak elemen tersebut. Proses pencarian untuk menemukan sebuah elemen pada pohon cari dimulai dari akar. Besarnya upaya yang diperlukan dalam pencarian tersebut diukur oleh banyaknya perbandingan yang dilakukan. Yang dimaksudkan dengan banyak perbandingan yang dilakukan adalah banyaknya simpul pada jalur cari, yaitu jalur yang berawal dari akar dan berakhir di simpul *record* yang kita cari tersebut. Sebagai contoh, perhatikan himpunan *record* 1, 2 dan 3, ketiga nama *record* tersebut dapat dibentuk menjadi 5 pohon cari yang berbeda, seperti terlihat pada Gambar 7.40.



Gambar 7.40 *Contoh lima pohon cari binar dengan simpul-simpul yang sama*

Untuk menemukan *record* 2 pada pohon cari binar di atas, banyaknya perbandingan yang dilakukan adalah berbeda-beda, seperti terlihat pada Tabel 7.1.

Tabel 7.1

Pohon	Banyak Perbandingan
a	2
b	3
c	1
d	3
e	2

Jadi untuk *record* 2 pada pohon c, banyaknya perbandingan lebih kecil dari pohon a, pohon b, pohon d dan pohon e. Masalah yang kita hadapi adalah, bagaimana menen-tukan bahwa suatu bangun pohon cari lebih baik dari pohon cari lainnya, untuk himpunan *record* yang sama dan juga bagaimana bangun suatu pohon cari yang baik.

Bangun pohon cari binar tidak dapat dievaluasi dengan baik jika hanya berdasarkan jalur cari yang didapat dari pencarian satu *record* saja. Kita harus mengikut-sertakan semua *record* pada evaluasi tersebut. Sehingga bila rata-rata panjang dan jalur cari bangun tersebut dibandingkan, barulah dapat ditentukan bangun mana yang lebih baik.

Misalkan P adalah peluang atau probabilitas akses (pengambilan), yakni peluang bahwa nama N akan dicari, N_i adalah salah satu nama *record* pada pohon cari. Jika peluang pengambilan suatu *record* tertentu N_i di atas diketahui, maka :

$$\sum_{i=1}^{n} P_i = 1$$

Pada pencarian yang dianggap selalu sukses, panjang cari yang diharapkan untuk setiap pohon cari didefinisikan, sebagai :

$$\sum_{i=1}^{n} P_i h_i$$

h, adalah banyaknya perbandingan untuk mencapai N_i.

Sebagai contoh, pandang ketiga nama 1, 2, 3 dengan 5 buah pohon cari pada Gambar 7.40. Jika peluang pengambilan nama *record* pada suatu pohon cari adalah diketahui sebagai berikut :

Untuk nama 1 = P_1 = 1/7
Untuk nama 2 = P_2 = 2/7
Untuk nama 3 = P_3 = 4/7

maka panjang cari yang diharapkan untuk masing-masing pohon cari dapat dilihat pada Tabel (2).

Tabel 7.2

Pohon Cari	nama 1	nama 2	nama 3	Panjang Cari
a	1/7.3	2/7.2	4/7.1	11/7
b	1/7.2	2/7.3	4/7.1	12/7
c	1/7.2	2/7.1	4/7.2	12/7
d	1/7.1	2/7.3	4/7.2	15/7
e	1/7.1	2/7.2	4/7.2	17/7

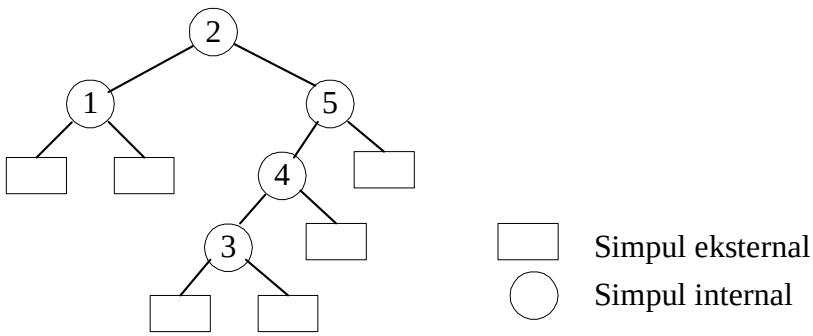
Dari Tabel (2) tersebut terlihat bahwa panjang cari bernilai minimum dari masing-masing pohon cari adalah pohon cari (a). Jadi bangun cari pohon cari (a) adalah pohon cari yang terbaik dari 5 bangun pohon cari yang ada. Agar panjang cari yang diharapkan minimum, maka pohon cari yang bersangkutan dibuat dengan cara menaruh nama yang sering diakses, sedekat mungkin dengan akar. Untuk melaksanakan hal tersebut, kita harus meletakkan nama-nama tersebut berurutan sesuai aturan pohon cari binar.

Namun, cara di atas tidak selalu menjamin bahwa pohon cari yang kita dapatkan adalah yang terbaik. Karena panjang jalur dari suatu pohon cari akan berubah, bila kita lakukan penyisipan atau penghapusan sebuah nama pada pohon yang bersangkutan. Bila kita perhatikan, bahwa untuk pencarian suatu *record* dengan nama X dalam pohon cari, dapat terjadi kegagalan, yang disebabkan *record* yang kita cari atau inginkan itu tidak terdapat pada pohon cari tersebut. Untuk keadaan tersebut, dapat kita letakkan suatu simpul bayangan pada pohon cari. Simpul bayangan disisipkan pada pohon cari, dengan maksud agar pencarian yang gagal berakhir pada simpul tersebut. Letak dari *record* dengan nama X adalah di antara $N_i < X < N_{i+1}$, i =1, 2, …, n.

Bila kita gambarkan dalam suatu bangun pohon cari binar, letak simpul bayangan tersebut terdapat di setiap bagian subpohon yang hampa. Andaikan peluang kegagalan dalam menemukan sebuah *record* tertentu pada pohon cari adalah q, maka peluang kegagalan secara keseluruhan adalah :

$$\sum_{i=0}^{n} q_i$$

Perhatikan bangun pohon cari binar pada Gambar 7.41. Di sini himpunan (N_i) adalah himpunan *record* dengan nama 1, 2, 3, 4 dan 5.



Gambar 7.41. *Simpul internal dan eksternal*

Terlihat bahwa pada setiap bagian subpohon yang hampa, terdapat simpul bayangan. Simpul bayangan tersebut dikatakan sebagai *external* simpul (digambarkan sebagai bujursangkar) dan simpul lainnya dikatakan sebagai *internal* simpul. Jika pada suatu pohon cari terdapat N buah *internal* simpul, maka banyaknya *external* simpul adalah N+1 buah. Setiap *external* simpul menampilkan pencarian yang berakhir dengan kegagalan. Jika pencarian sukses berakhir pada sebuah simpul dan h_i adalah banyaknya perbandingan yang dilakukan untuk mencapai nama N_i, maka panjang cari yang diharapkan adalah :

$$\sum_{i=1}^{n} P_i h_i$$

Jika kita perhatikan kembali pada pencarian yang gagal, maka letak dari nama X akan terdapat pada salah satu dari (N+1) buah *external* simpul. Selanjutnya (N+1) buah *external* simpul tersebut dinotasikan sebagai E_i, dengan i = 0,1,..., n.

E_0 adalah *external* simpul yang mengandung nama X untuk E_0, $E_0 < N_1$ maka X terletak pada $X < N_i$,

E_1 adalah *external* simpul yang mengandung nama X untuk Ei, $N_i < E_i < N_{i+1}$, maka X terletak pada $Ni < X < Ni+1$, dengan i = 1, 2, ..., n-1.

E_n adalah *external* simpul yang mengandung nama X, untuk X_n, $N_n < E_n$, maka X terletak pada $X > N_n$.

Jika pencarian yang gagal berakhir pada sebuah simpul dan h_i adalah banyaknya perbandingan yang dilakukan untuk mencapai nama E_i, maka panjang cari yang diharapkan adalah :

$$\sum_{i=0}^{n} q_i(h_i-1)$$

h, adalah banyaknya perbandingan untuk mencapai E. Dari uraian di atas, kita dapatkan bahwa panjang cari yang diharapkan (COST) untuk masing-masing pohon cari adalah,

$$\sum_{i=1}^{n} pihi + \sum_{i=0}^{n} q_i(h_i-1)$$

Berarti untuk setiap pohon cari binar, yang memiliki COST terendah (minimum) di antara bangun pohon cari lainnya (dengan himpunan *record* yang sama) adalah pohon cari binar yang optimal. Sebagai contoh, himpunan *record* dengan nama 1, 2 dan 3 yang lalu dapat kita buat menjadi beberapa bangun pohon cari dengan simpul eksternal, seperti terlihat pada Gambar 7.42 di bawah ini, dengan :

$P_1 = 0,5$; $P_2 = 0,1$; $P_3 = 0,05$

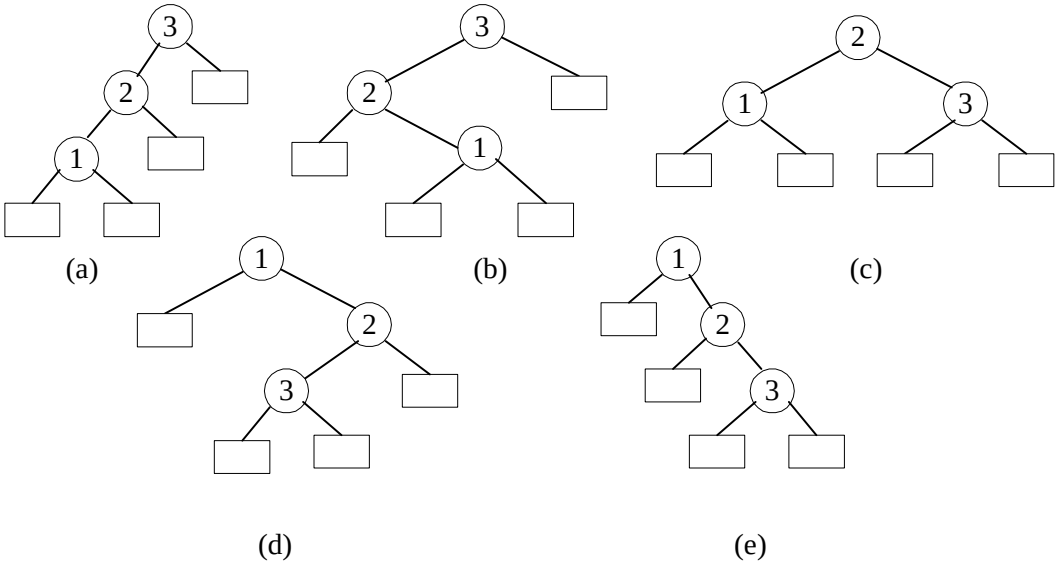
$Q_0 = 0,15$; $Q_1 = 0,1$; $Q_2 = 0,05$; $Q_3 = 0,05$

Panjang cari yang diharapkan, (COST), dari masing-masing pohon cari adalah (sambil melihat Gambar 7.42).

nama1 nama2 nama3 E0 E1 E2 E3

COST(a) = 0,5*3 + 0,1*2 + 0,05*1 + 0,15*3 + 0,1*3 + 0,05*2 + 0,05*1 = 2,65
COST(b) = 0,5*2 + 0,1*1 + 0,05*2 + 0,15*2 + 0,1*2 + 0,05*2 + 0,05*2 = 1,90
COST(c) = 0,5*1 + 0,1*2 + 0,05*3 + 0,15*1 + 0,1*2 + 0,05*3 + 0,05*3 = 1,50
COST(d) = 0,5*2 + 0,1*3 + 0,05*1 + 0,15*2 + 0,1*3 + 0,05*3 + 0,05*1 = 2,05
COST(e) = 0,5*1 + 0,1*3 + 0,05*2 + 0,15*1 + 0,1*2 + 0,05*3 + 0,05*2 = 1,60

Terlihat bahwa COST yang minimum dari pohon cari tersebut dimiliki oleh pohon cari (c). Jadi bangun pohon cari (c) adalah bangun pohon cari yang terbaik (optimal) dari 5 bangun pohon cari yang ada. Cara yang kita gunakan di atas cukup mudah dan sederhana. Di sini kita menggunakan pendekatan *try of posibilites*, maksudnya kita harus menghitung semua kemungkinan yang ada pada setiap pohon cari. Cara ini sukar sekali kita terapkan untuk mencari pohon Cari optimal pada pohon berorder besar. Pada bagian berikut ini, kita ketengahkan cara yang lebih umum, meskipun dengan pendekatan matematika yang rumit.

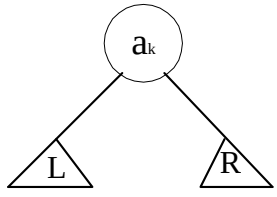


Gambar 7.42.

7.15 TAMBAHAN : LEBIH LANJUT TENTANG POHON CARI OPTIMAL

Telah kita lihat pada bagian terdahulu, bahwa pohon cari yang optimal mempunyai sifat bahwa semua subpohon yang ada harus optimal pula. Berdasarkan sifat tersebut, terdapat suatu cara untuk menyelesaikan masalah optimasi dari pohon binar, khususnya bila kita harus mengevaluasi suatu pohon cari yang besar. Karena setiap subpohon harus optimal, evaluasi pada pohon cari dimulai dari masing-masing simpul daun. Berikutnya, evaluasi dilakukan pada subpohon yang lebih besar lagi, hingga mencapai bangun pohon cari keseluruhan.

Perhatikan suatu bangun pohon cari binar T yang optimal pada Gambar 7.43 dengan akar $a_{k.}$



Gambar 7.43.

Pendekatan yang kita lakukan adalah bahwa setiap a_i dapat diangkat sebagai akar dari bangun pohon cari. Jika kita pilih a_k sebagai akar dari pohon cari, maka simpul internal a_1, a_2, a_3, ..., a_{k+1} terletak di subpohon kiri dari akar a_k. Begitu pula simpul eksternal E_1, E_2, …, E_{k+1} terletak di subpohon kiri. Sedangkan simpul lainnya yaitu a_{k+1}, ..., a_n dan E_k, E_{k+1}, ..., E_n terletak di subpohon kanan.

Ditentukan :

$$\sum_{i=1}^{k-1} p_i h_i \qquad \sum_{i=0}^{h-1} q_i (h_i - 1)$$

dan

$$COST(R) = P_i h_i + q_i(h_i - 1)$$

Diketahui bahwa COST untuk masing-masing pohon cari binar adalah :

$$\begin{aligned} COST &= P_i h_i + q_i(h_i - 1) \\ &= P_i h_i + P_k h_k + P_i h_i + q_i(h_i - 1) + q_i(h_i - 1) \end{aligned}$$

h_k adalah banyaknya perbandingan yang dilakukan untuk mencapai akar a_k. Banyaknya perbandingan untuk mencapai akar a_k adalah sebanyak 1.

$$\begin{aligned} &= P_i h_i + P_k + P_i h_i + q_i(h_i - 1) + q_i(h_i - 1) \\ &= P_k + P_i h_i + q_i(h_i - 1) + P_i h_i + q_i(h_i - 1) \\ &= P_k + COST(L) + COST(R) \end{aligned}$$

Kita dapatkan COST sebenarnya dari bangun pohon cari binar T, untuk semua simpul internal. Sedangkan COST yang dihitung, termasuk juga simpul eksternal, untuk menentukan suatu pohon cari T yang optimal. Untuk menghitung COST dari simpul eksternal, kita gunakan :

$$W(i,j) = q_i + (q+p)$$

sebagai bobot dari pohon T.

$$\text{Bobot}(L) = \text{Bobot } T(0, k-1) = W(0, k-1)$$

$$\text{Bobot}(R) = \text{Bobot } T(k,n) = W(k,j)$$

$$\begin{aligned} W(i,j) &= W(0, k-1) + W(k,n) \\ &= Q_0 + (q+p)\, q_k + (q+p) \end{aligned}$$

Kita peroleh COST dari pohon cari T,

P_k + COST(L) + COST(R) + W(0, k-1) + W(k, n) ……(1)

adalah COST keseluruhan dari pohon cari optimal T. Jika pohon cari T optimal, COST dari persamaan (1) harus minimum. Akibatnya COST(L) harus minimum pula, untuk pohon cari yang mengandung a_1, a_2, a_3, …, a_{k-1} dan E_0, E_1, E_2, …, E_{k-1}. Begitu pula COST(R) harus minimum pula untuk a_{k+1}, …, a_n dan E_k, E_{k+1}, …, E_n. Jika C(i,j) adalah COST untuk pohon cari T yang optimal, berarti COST untuk subpohon yang optimal adalah COST(L) = C(0, k-1). Sedangkan COST untuk subpohon kanan yang optimal adalah COST(R) = C(k,n). Persamaan (1) menjadi :

P_k + C(0,k-1) + X(k,n) + W(0, k-1) + W(k, n) ……(2)

Pohon cari T adalah suatu pohon cari yang optimal, kita tentukan harga k sedemikian rupa sehingga persamaan (2) minimum,

$$C(0,n) = \min_{i<=k<=n} C(0, k-1) + C(k, n) + P_k + W(0, k-1) + W(k,n) \quad \text{.....(3)}$$

Dalam bentuk umum persamaan (3) dapat dituliskan sebagai berikut :

$$C(i,j) = \min_{i<=k<=n} C(i, k-1) + C(k, j) + P_k + W(i, k-1) + W(k,j)$$
$$= \min_{i<=k<=n} C(I, k-1) + C(k,j) + W(I,j) \quad \text{……..(4)}$$

Persamaan (4) dapat diselesaikan dengan menghitung semua kemungkinan C(i,j) sedemikian sehingga j–1= 0. Selanjutnya menghitung semua kemungkinan C(i,j) sedemikian sehingga j–i = 1, kemudian j–i = 2 dan seterusnya.

Selama perhitungan kita akan mendapatkan akar R(i,j) dari setiap pohon T. Selanjutnya dapat dibentuk suatu harga bangun pohon cari binar yang optimal dari *root* R(i,j) tersebut. R(i,j) adalah suatu harga (sebut k) yang meminimumkan persamaan (4).

Contoh 7.4

Misalkan n = 4

(a_1,a_2,a_3,a_4) = (do, if, read, write)

Misalkan P(1,2,3,4) = (3,3,1,1)

Q(0,1,2,3,4) = (2,3,1,1,1)

Pada perhitungan awal, W(i,j) = Q(i)

C(i,j) = 0

R(i,j) = 0 untuk 0 <= i <= 4

Dengan mempergunakan persamaan (4) dan W(i,j) = P(j) + W(i, j–1), kita lakukan :

Langkah 1.

i = 0, Q(0) = 2; C(0,0) = 0; R(0,0) = 0

i = 1, Q(1) = 3; C(1,1) = 0; R(1,1 = 0

i = 2, Q(2) = 1; C(2,2) = 0; R(2,2) = 0

i = 3, Q(3) = 1; C(3,3) = 0; R(3,3) = 0

i = 4, Q(4) = 1; C(4,4) = 0; R(4,4) = 0

didapat :

j - i = 0	0	1	2	3	4
	0, 0 2,0,0	1, 1 3,0,0	2,2 1,0,0	3,3 1,0,0	4,4 1,0,0

Langkah 2 :

W(0,1) = P(j) + Q(j) + W(i, j-1)

= P(1) + Q(1) + W(0,0) = 3 + 3 + 2 = 8

C(0,1) = min(C(0,0) + C(1,1) + W(0,1))
0 <= k <= 1
= 0 + 0 + 8 = 8

R(0,1) = 1, harga k yang minimum C(0,1) adalah i (k=1)

W(1,2) = P(2) + Q(2) + W(1,1) = 3 + 1 + 3 = 7
1 < k <= 2
= 0 + 0 + 7 = 7

R(1,2) = 2

W(2,3) = P(3) + Q(3) + W(2,2) = 1 + 1 + 1 = 3

C(2,3) = min C(2,2) + C(3,3) + W(2,3)
2 <k <=3
= 0 + 0 + 3 = 3

R(3,4) = 4
Didapat :

	0	1	2	3
j - i = 1	0, 1 2,0,0	1, 2 3,0,0	2,3 1,0,0	3,4 1,0,0

Langkah 3

W(0,2) = P(2) + Q(2) + W(0,1) = 3 + 1 + 8 = 12
C(0,2) = min(C(0,0) + C(1,2) + W(0,2))
0 <= k <= 2
= 0 + 7 + 12 = 19
= min(C(0,1) + C(2,2) + W(0,2))
0 <= k <= 2
= 8 + 0 + 12 = 20
R(0,2) = 1
W(1,3) = P(3) + Q(3) + W(1,2) = 1 + 1 + 7 = 9
C(1,3) = min(C(1,1) + C(2,3) + W(1,3))
1 <= k <= 3
= 7 + 3 + 9 = 19
R(1,3) = 2

W(2,4) = P(4) + Q(4) + W(2,3) = 1 + 1 + 3 = 5

C(2,4) = min C(2,2) + C(3,4) + W(2,4)
2 <k <= 4
= 0 + 3 + 5 = 8

C(2,4) = min C(2,3) + C(4,4) + W(2,4)
2 <k <= 4
= 3 + 0 + 5 = 8
R(,4) = 3
Didapat :

	0	1	2
j - i = 2	0, 1 8,8,1	1, 2 7,7,2	2,3 3,3,3

Langkah 4

W(0,3) = P(4) + Q(3) + W(0,2) = 1 + 1 + 12 = 14
C(0,3) = min(C(0,0) + C(1,3) + W(0,3))
0 <= k <= 3
= 0 + 12 + 14 = 26
C(0,3) = min(C(0,1) + C(2,3) + W(0,3))
0 <= k <= 3
= 8 + 3 + 14 = 25
C(0,3) = min(C(0,2) + C(3,3) + W(0,3))
0 <= k <= 3
= 19 + 0 + 14 = 33
R(0,3) = 2
W(1,4) = P(4) + Q(4) + W(1,3) = 1 + 1 + 9 = 11
C(1,4) = min(C(1,1) + C(2,4) + W(1,4))
1 <= k <= 3
= 7 + 3 + 9 = 19
C(1,4) = min(C(1,2) + C(3,4) + W(1,4))
1 <= k <= 4
= 7 + 3 + 11 = 21
C(1,4) = min(C(1,3) + C(4,4) + W(1,4))
1 <= k <= 4
= 12 + 0 + 11 = 23
Didapat :

	0	1
j - i = 3	0,3 14,25,2	1,4 11,19,2

Langkah 5

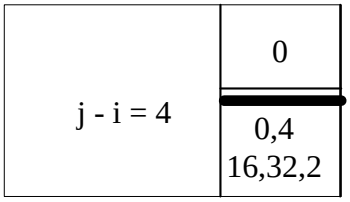
W(0,4) = P(4) + Q(4) + W(0,3) = 1 + 1 + 14 = 16
C(0,4) = min(C(0,0) + C(1,4) + W(0,4))
0 <= k <= 4
= 0 + 19 + 16 = 35
C(0,4) = min(C(0,1) + C(2,4) + W(0,4))
0 <= k <= 4
= 8 + 8 + 16 = 32
= min(C(0,2) + C(3,4) + W(0,4))
0 <= k <= 4
= 19 + 3 + 16 = 38
= min(C(0,3) + C(4,4) + W(0,4))

$0 <= k <= 4$

$= 25 + 0 + 16 = 41$

$R(0,4) = 2$

Didapat :

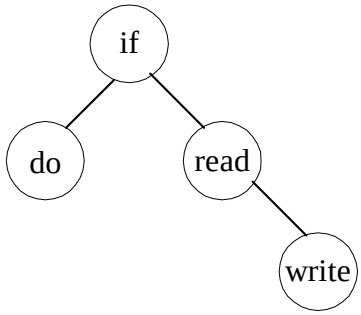


Kita dapatkan :

TABEL 7.3.

	0	1	2	3	4
0	$W_{00} = 2$ $C_{00} = 0$ $R_{00} = 0$	$W_{11} = 3$ $C_{11} = 0$ $R_{11} = 0$	$W_{22} = 1$ $C_{22} = 0$ $R_{22} = 0$	$W_{33} = 1$ $C_{33} = 0$ $R_{33} = 0$	$W_{44} = 1$ $C_{44} = 0$ $R_{44} = 0$
1	$W_{01} = 8$ $C_{01} = 8$ $R_{01} = 1$	$W_{12} = 7$ $C_{12} = 7$ $R_{12} = 2$	$W_{23} = 3$ $C_{23} = 3$ $R_{23} = 3$	$W_{34} = 3$ $C_{34} = 3$ $R_{34} = 4$	
2	$W_{02} = 12$ $C_{02} = 19$ $R_{02} = 1$	$W_{13} = 9$ $C_{13} = 12$ $R_{13} = 2$	$W_{24} = 5$ $C_{24} = 8$ $R_{24} = 3$		
3	$W_{03} = 14$ $C_{03} = 25$ $R_{03} = 2$	$W_{14} = 11$ $C_{14} = 19$ $R_{14} = 2$			
4	$W_{04} = 16$ $C_{04} = 32$ $R_{04} = 2$				

Dari tabel di atas, C(0,4)=32 adalah COST minimum yang dimiliki oleh pohon cari binar dengan (a1,a2,a3,a4) (do,if,read,while). Akar dari pohon binar T04 adalah a2. Subpohon kiri adalah T01 dan subpohon kanan adalah T24. T24 mempunyai akar a3; dengan subpohon kiri T22 dan subpohon kanan T34. Jadi dengan data dari tabel di atas, kita dapat membentuk sebuah pohon dengan akar T04. Bangun dari pohon cari binarnya terlihat pada Gambar 7.44.

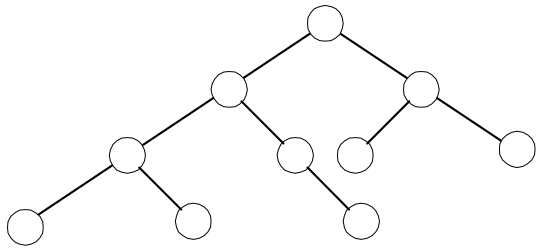


Gambar 7.44. *Contoh pohon cari binar*

Bangun pohon cari binar di atas adalah bangun pohon cari optimal yang kita dapatkan.

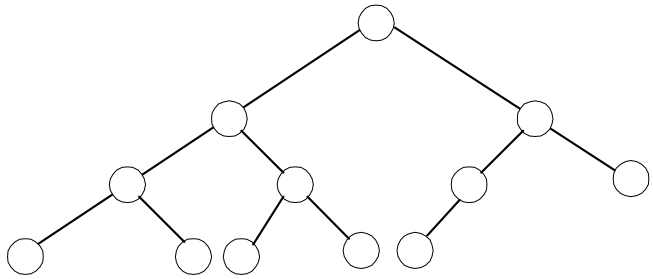
7.16 HEAP

Kita tengok kali ini, salah satu bentuk khusus dari pohon binar lengkap. Seperti telah dibahas terdahulu, pohon binar lengkap adalah pohon binar yang seluruh simpul lengkap, kecuali pada *level* terakhir, dengan catatan pula bahwa simpul pada *level* terakhir tersebut terletak sekiri mungkin dari pohon. Gambar 7.45 menunjukkan contoh pohon binar lengkap.



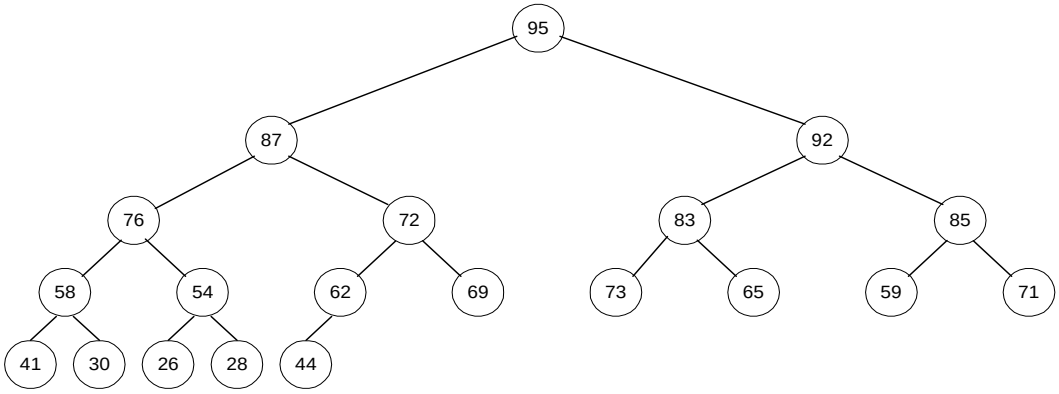
Gambar 7.45. *Pohon binar lengkap*

Pada Gambar 7.45, pohon binar ber*level* 0, 1, 2 dan 3. Hanya 4 simpul terdapat pada *level* 3. Di sini ke empat simpul tersebut berada di bagian kiri. Gambar 7.46 merupakan contoh pohon binar tidak lengkap



Gambar 7.46. *Pohon binar hampir lengkap*

Pada pohon binar yang simpulnya berisi elemen anggota himpunan terurut total, seperti himpunan bilangan atau untai alfabet, kita dapat mendefinisikan struktur *heap*. Suatu pohon binar adalah *heap*, jika nilai setiap simpul lebih besar atau sama dengan nilai anaknya. *Heap* seperti didefinisikan di atas disebut juga "*maxheap*". Kalau nilai setiap simpul lebih kecil atau sama dengan nilai anaknya maka pohon tersebut disebut "*minheap*". Gambar 7.47 menunjukkan contoh suatu *heap*.

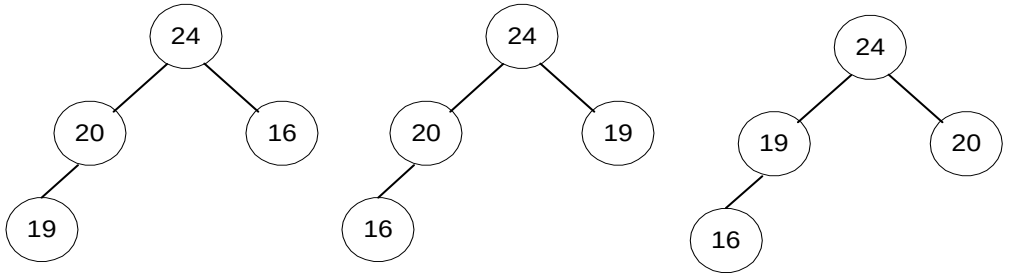


Gambar 7.47. *Contoh sebuah heap*

Ke-20 elemen *heap* tersebut dapat kita sajikan sebagai daftar berurutan TREE[ ], sebagai berikut :

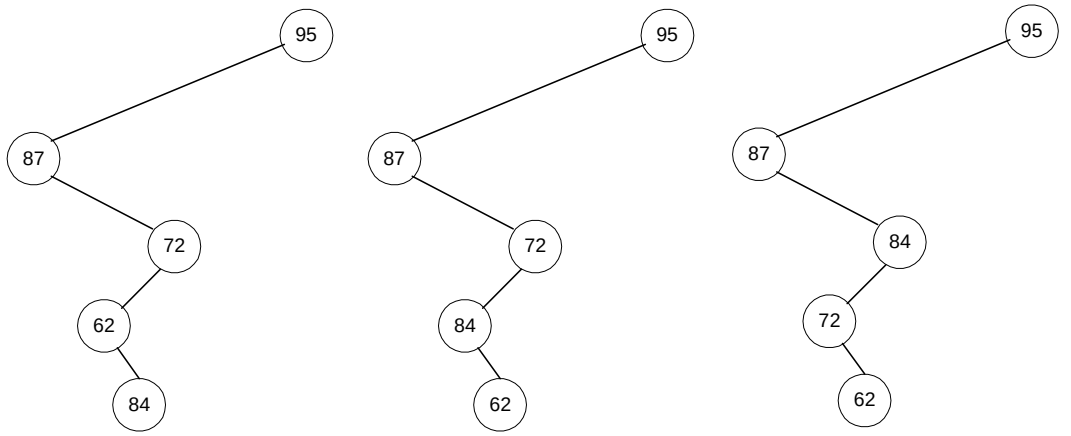
Posisi ke	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
data :	95	87	92	76	72	83	85	58	54	62	69	73	65	59	71	41	30	26	28	44

Urutan dari elemen TREE[ ] adalah berdasarkan *level* dari simpul, dibaca dari kiri ke kanan, *level* demi *level*. Dapat dicatat bahwa bila kita memiliki himpunan data, *heap* yang dapat kita bentuk adalah tidak tunggal. Perhatikan saja himpunan data kita [20,16,19,24]. Tiga buah *heap* yang dapat kita bentuk terlihat pada gambar 7.48.



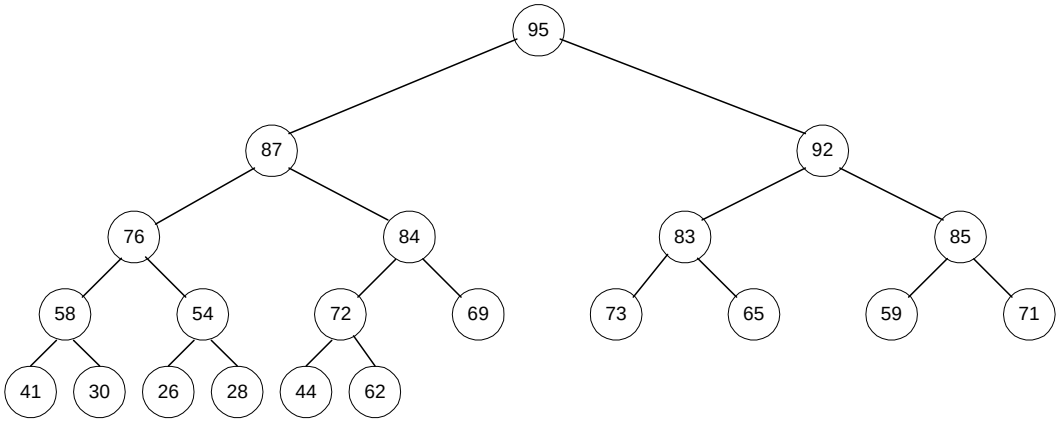
Gambar 7.48.

Kita dapat memasukkan elemen baru ke dalam sebuah *heap*. Misalkan elemen 84 hendak kita masukkan ke dalam *heap* pada gambar 7.47. Mula-mula elemen tersebut kita letakkan sebagai elemen terakhir dari *heap*, dalam hal ini sebagai elemen TREE[21] pada daftar berurutan. Langkah selanjutnya adalah memeriksa apakah elemen tersebut lebih besar dari ayahnya. Bila lebih besar, kita lakukan penukaran; elemen tersebut sekarang menjadi ayah. Demikian seterusnya sampai kondisi ini tidak tercapai. Gambar 7.49 menunjukkan proses pemasukan elemen 84 ke dalam *heap.*



Gambar 7.49.

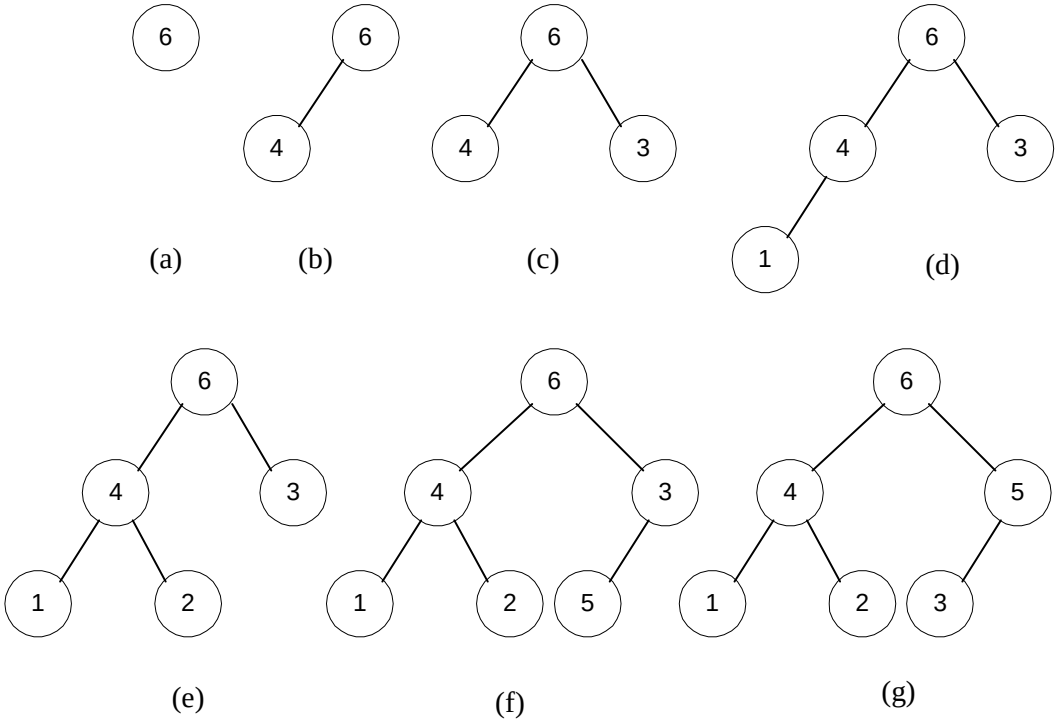
Proses kita hentikan, karena 84 < 87, sehingga *heap* setelah memasukkan elemen 84 menjadi seperti gambar 7.50.



Gambar 7.50

Proses pemasukkan satu simpul seperti dijelaskan di atas dapat kita manfaatkan pula untuk membentuk sebuah *heap* dari himpunan data yang diberikan. Proses dimulai dari pohon hampa. Sebagai contoh kita hendak membentuk *heap* dari himpunan [6,4,3,1,2,5]. Prosesnya sebagai berikut :

* Masukkan elemen 6
* Masukkan elemen 4 sebagai anak kiri. Karena 4 < 6, tidak dilakukan pertukaran
* Masukkan elemen 3 sebagai anak kanan dari 6. Karena 3 < 6, tidak dilakukan pertukaran
* Masukkan elemen 1 sebagai anak kiri elemen 4. Karena 1<4 tidak dilakukan pertukaran.



Gambar 7.51. *Proses pembentukan heap*

7.17 HEAPSORT

Setelah kita membahas mengenai suatu bentuk khusus dari Pohon binar, yaitu bentuk *heap*, maka pada bagian ini kita perkenalkan salah satu aplikasi dari bentuk *heap* tersebut. Aplikasi tersebut adalah dalam rangka melakukan sortir atau pengurutan suatu larik (*array*). Metode pengurutan ini, dikenal sebagai metode *heapsort*.

Pandang bahwa kita mempunyai sebuah *array* A dengan N elemen. Algoritma *heapsort* untuk melakukan pengurutan terhadap elemen dari A terdiri atas 2 fase. Pada fase pertama kita membentuk sebuah *heap* H terhadap elemen A. Selanjutnya pada fase kedua secara berulang-ulang kita menghapus akar dari *heap* H. Karena akar dari *heap* H selalu mengandung simpul dengan nilai terbesar di H, maka fase pelaksanaan fase kedua akan menghapus elemen dari A secara urutan menurun.

Secara formal algoritma ini memanfaatkan prosedur pembentukan *heap*, seperti yang telah dibicarakan pada bagian 7-16 yang lalu.

ALGORITMA HEAPSORT(A,N)

(Diberikan sebuah *array* A dengan N elemen. Algoritma ini dimaksudkan untuk melakukan pengurutan terhadap elemen A)

1. [Kita bentuk sebuah *heap* H dengan cara yang biasa dilakukan]

   Repeat untuk J = 1 sampai N-1

   Call insheap(A,J,a(j+1))

   [**end** dari *loop*]
2. [Urutkan elemen A dengan menghapus berulang-ulang akar dari H]

   Repeat while N > 1;

   a). **Call** Delheap(A,N,ITEM)

   b). A[CN+1] := ITEM

   [end dari *loop*]
3. Exit

Maksud langkah 2b dari algoritma adalah untuk mengefisienkan tempat. Artinya, seseorang dapat menggunakan *array* lain B untuk menampung elemen terurut dari A dan menggantikan langkah 2b dengan menetapkan :

B[N+1] := ITEM

Kompleksitas Algoritma Heapsort.

Sekarang kita tinjau waktu kompleksitas dari heapsort. Pandang bahwa algoritma heapsort akan kita gunakan terhadap *array* A dengan n elemen. Algoritma mempunyai 2 fase dan kita melakukan analisis kedua fase tersebut secara terpisah.

Fase pertama

Pandang sebuah *heap* H. Periksa bahwa banyaknya perbandingan untuk mencari posisi yang sesuai dari sebuah elemen baru ITEM di H tidak boleh melebihi kedalaman dari H. Karena H adalah pohon lengkap, kedalamannya terbatas oleh fungsi log2 m, di sini m adalah banyaknya elemen dari H. Karena itulah jumlah perbandingan keseluruhan g(n) untuk memasukkan n buah elemen dari A ke dalam H terbatas oleh :

$$g(n) <= n \log_2 n$$

Oleh karena itu, waktu pelaksanaan fase pertama dari heapsort adalah berbanding sebagai $n \log_2 n$

Fase Kedua

Pandang H adalah pohon lengkap dengan m elemen. Juga pandang subpohon kanan dari H, adalah *heap*, dan L adalah akar dari H. Dapat dilihat bahwa pengulangan pembentukan *heap* membutuhkan 4 perbandingan untuk memindahkan simpul L satu tingkat ke bawah dari pohon H. Karena kedalaman dari H tidak melebihi $\log_2 m$, pem-bentukan kembali *heap* membutuhkan paling banyak $4 \log_2 n$ perbandingan. Untuk mendapatkan posisi yang tepat dari L di dalam pohon H. Ini berarti bahwa jumlah perbandingan keseluruhan h(n) untuk menghapus n buah elemen *array* A dari pohon H, yang membutuhkan pengulangan pembentukan n kali, terbatas :

$$h(n) <= 4 \log_2 n$$

Oleh karena itu, waktu pelaksanaan fase kedua dari *heapsort* adalah juga berbanding sebagai $n \log_2 m$. Karena masing-masing fase membutuhkan waktu berbanding sebagai $n \log_2 m$, maka waktu pelaksanaan untuk pengurutan n elemen *array* A menggunakan *heapsort* adalah berbanding sebagai $n \log_2 n$, yaitu f(n) = O(n log2n)

Sebagai perbandingan dapat kita lihat 2 metode pengurutan yang lain, yaitu *bubble-sort* dan *quicksort*. Di sini, waktu pelaksanaan rata-rata untuk *bubble-sort* adalah lebih lambat, yakni $O(n^2)$. Sedangkan untuk *quicksort* adalah $O(n \log_2 n)$; sama seperti *heap-sort*. Namun, waktu pelaksanaan terburuk dari *quicksort* mencapai $O(n^2)$, sama seperti *bubble-sort*.

L A T I H A N 7

1. Apa hubungan pohon binar dengan *graph* ?

2. Apa yang dimaksud dengan istilah (a) *path,* (b) *root,* (c) *edge,* (d) *leaf* dan (e) *level,* dalam pohon binar ?

3. Apa pula yang dimaksud dengan istilah (a) *siblings,* (b) *height,* (c) *weight,* (d) *node* dan (e) *null tree* dalam pohon binar ?

4. Apa pula yang dimaksud dengan istilah (a) *son,* (b) *father,* (c) suksesor, (d) terminal dan (e) rekursif dalam pohon binar ?

5. Apa beda pohon yang *similar* dengan *copies* ? apa syarat keduanya ?

6. Apa pula yang dimaskud dengan istilah (a) *descendant,* (b) *ancestor,* (c) *branch,* (d) *complete* dan (e) *almost complete* dalam pohon binar ?

7. Apa pula yang dimaskud dengan istilah (a) *extended,* (b) *left substree,* (c) ketinggian seimbang, dan (d) *binary search tree* dan (e) *heap* dalam pohon binar ?

8. Ada tiga cara traversal dalam pohon binar, sebutkan berikut cara kerjanya.

9. Jelaskan (a) pembentukan, (2) penyisipan dan (3) penghapusan elemen pada pohon cari binar.

10. Buat struktur pohon binar dari notasi *postfix* berikut ini, serta tentukan notasi *prefix* dan *infix*nya : A + B – C ^D ^ E – F /G *H

11. Buat sebuah struktur pohon umum dan jelaskan bagaimana mengalihkan pohon umum tersebut menjadi pohon binar.

12. Buat suatu deret bilangan secara acak dan bentuklah struktur *heap*-nya.