BAB II

MATERI PENUNJANG

2.1. Teori Graf (*Graph Theory*)

2.1.1. Graf berarah (*Directed graph*)

Definisi 1.

Graf tak berarah G = (V,E) adalah suatu sistem yang terdiri dari himpunan V(G) berhingga tidak kosong dari elemen-elemen yang disebut node dan himpunan E(G) (mungkin kosong) dari pasangan tidak terurut antara dua node yang disebut garis. Suatu garis (v_i,v_j) atau (v_j,v_i) dikatakan menghubungkan node v_i dan node v_j.

■

Definisi 2.

Sebuah path dengan panjang n-1 dalam graf G adalah suatu susunan garis-garis berbentuk (v_1,v_2), (v_2,v_3), ... ,(v_{n-1},v_n) dimana $v_i \neq v_j$ untuk setiap i dan j. Node v_1 disebut node awal dan node v_n disebut node akhir dari path tersebut. Apabila path mempunyai node awal dan node akhir yang sama maka disebut dengan cycle.

■

Definisi 3.

Sebuah graf G dikatakan terhubung apabila setiap pasangan node v_i,v_j dalam G dihubungkan oleh sekurang-kurangnya satu path.

■

Definisi 4.

Graf berarah $G_d = (V,E)$ adalah suatu sistem yang terdiri dari himpunan $V(G_d)$ berhingga tidak kosong dari elemen-elemen yang disebut node dan himpunan $E(G_d)$ dari garis berarah. Suatu garis berarah (v_i,v_j) dikatakan berarah dari node v_i menuju node v_j.

■

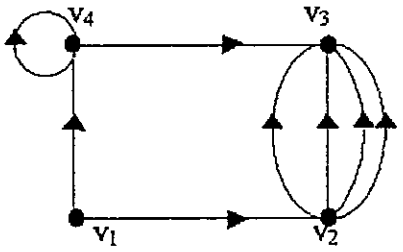
Definisi 5.

Misalkan $G_d = (V,E)$ graf berarah dan garis berarah $(v_i,v_j) \in E(G_d)$. Maka v_i disebut node awal dan v_j disebut node akhir garis tersebut. Jika $v_i = v_j$ maka garis berarah tersebut disebut loop. Sedangkan garis pararel dalam G_d adalah garis berarah dimana v_i node awal dan v_j node akhir dengan bentuk $(v_i,v_j)_1$, $(v_i,v_j)_2$, ..., $(v_i,v_j)_k$ dengan $k \geq 2$.

■

Contoh 1.

Pandang graf berarah G_d pada gambar 2.1. Garis berarah (v_4,v_4) disebut loop pada node v_4 dan $(v_2,v_3)_1$, $(v_2,v_3)_2$, $(v_2,v_3)_3$, $(v_2,v_3)_4$ disebut garis paralel.



Gambar 2.1. Graf berarah G_d dengan loop dan garis paralel.

□

Definisi 6.

Misalkan $G_d = (V,E)$ graf berarah dan $v \in V(G_d)$. Derajat masuk (*indegree*) v, ditulis $d^-(v)$, adalah banyaknya garis berarah yang masuk ke v. Derajat keluar (*outdegree*) v, ditulis $d^+(v)$, adalah banyaknya garis berarah yang keluar dari v. ■

2.1.2. Tree berarah (*Directed Tree*)

Definisi 7.

Tree berarah adalah sebuah graf berarah tanpa cycle yang mempunyai tepat satu node dengan indegree 0, yang disebut akar. Sementara node lainnya mempunyai indegree 1. ■

Definisi 8.

Dalam tree berarah, setiap node yang mempunyai outdegree 0 adalah node daun. Semua node lainnya disebut node cabang. Level dari node adalah panjang path dari akar. Panjang path adalah jumlah garis berarah yang nampak dalam barisan satu path.

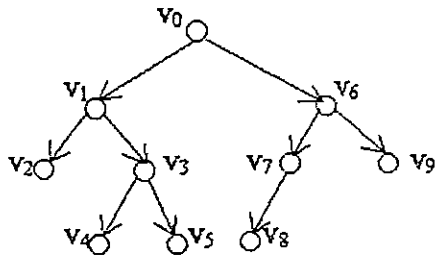
■

Level dari akar dari tree berarah adalah 0, sementara level dari node lainnya sama dengan panjang path dari akar. Susunan dari tree berarah menunjukkan bahwa setiap node dari tree adalah akar dari sejumlah subtree. Derajat dari node adalah jumlah subtree pada sebuah node, dengan demikian derajat dari node daun adalah 0.

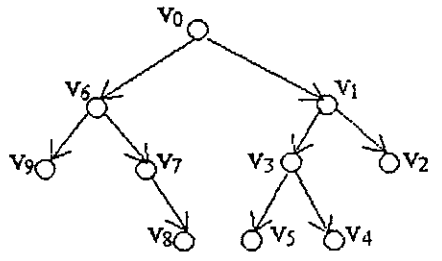
Definisi 9.

Jika dalam tree berarah, suatu pengurutan node pada tiap level ditentukan, maka tree semacam ini disebut tree terurut. ■

Node pada level tertentu dalam tree terurut adalah terurut dari kiri ke kanan. Untuk membedakan sebuah tree terurut dengan tree berarah yang lain menggunakan terminologi sebuah keluarga. Dengan demikian setiap node yang dapat dicapai dari sebuah node, misalkan u disebut turunan dari u. Sedangkan node yang dapat dicapai dari u dan terhubung oleh sebuah garis disebut anak dari u. Untuk melihat adanya urutan yang berbeda pada sebuah tree dapat dilihat pada gambar berikut ini :



(a)



(b)

Gambar 2.2. Contoh tree berarah.

Contoh 2.

Pandang gambar 2.2 menunjukkan tree berarah yang mempunyai dua node pada level 1, empat node pada level 2 dan tiga node pada level 3. Gambar 2.2. (a) dan (b) merepresentasikan tree berarah yang sama tetapi merupakan tree terurut yang berbeda. Pada gambar 2.2. terlihat bahwa node v_6 adalah akar dari $\{v_6, v_7, v_8, v_9\}$, v_1 adalah akar dari $\{ v_1, v_2, v_3, v_4, v_5\}$, v_5 adalah akar dari $\{v_5\}$, v_7 adalah akar dari $\{v_7, v_8\}$dan seterusnya. Degree dari v_3 adalah 2 karena mempunyai dua subtree yaitu $\{v_4\}$ dan $\{v_5\}$, sementara degree dari v_1 adalah 2 karena mempunyai dua subtree yaitu $\{v_2\}$ dan $\{ v_3, v_4, v_5\}$. □

Definisi 10.

Jika dalam tree berarah, outdegree dari setiap node-nodenya adalah lebih kecil atau sama dengan m , maka tree ini disebut *m-ary tree.* Jika outdegree dari setiap nodenya tepat sama dengan m atau 0, maka tree ini disebut *full* atau *complete m-ary tree. Binary tree biner* dan *full binary tree* adalah tree dengan $m = 2$. Pandang *m-ary tree* dimana m anak (atau lebih sedikit) dari satu node diasumsikan mempunyai m posisi berbeda. Jika suatu posisi yang demikian diperhitungkan maka tree ini disebut *a positional m-ary tree.* ■

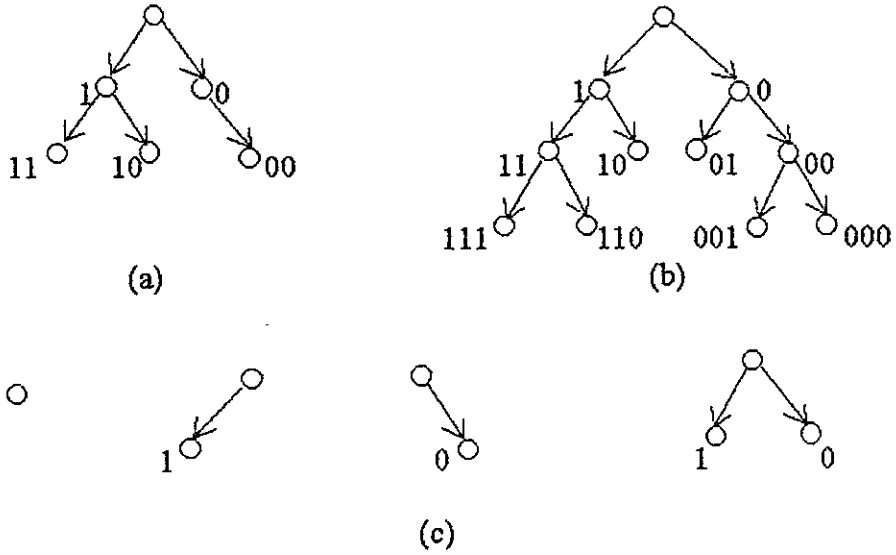
Dalam suatu posisi tree biner, setiap node direpresentasikan secara unik dengan menggunakan string pada alfabet {0,1} dan akar dinyatakan dengan string kosong. Suatu anak dari node u mempunyai string yang diawali oleh string dari u.

string dari suatu node daun tidak mengawali string dari node lainnya. Himpunan string-string yang berhubungan ke node daun membentuk *a prefix code*. Representasi yang serupa dari node-node dalam sebuah posisi *m-ary tree* dengan memakai string-string dari alfabet {0, 1, ..., *m*-1}adalah memungkinkan.

Contoh 3.

Pandang gambar 2.3 (a) dan (b) menunjukkan sebuah tree biner dengan posisi yang berbeda meskipun keduanya bukan termasuk tree terurut yang berbeda, (b) menunjukkan sebuah *full binary tree* yang prefix codenya adalah {111, 110, 10, 01, 001, 000}, dan (c) menunjukkan empat kemungkinan susunan anak dari sebuah node dalam sebuah tree biner.

□

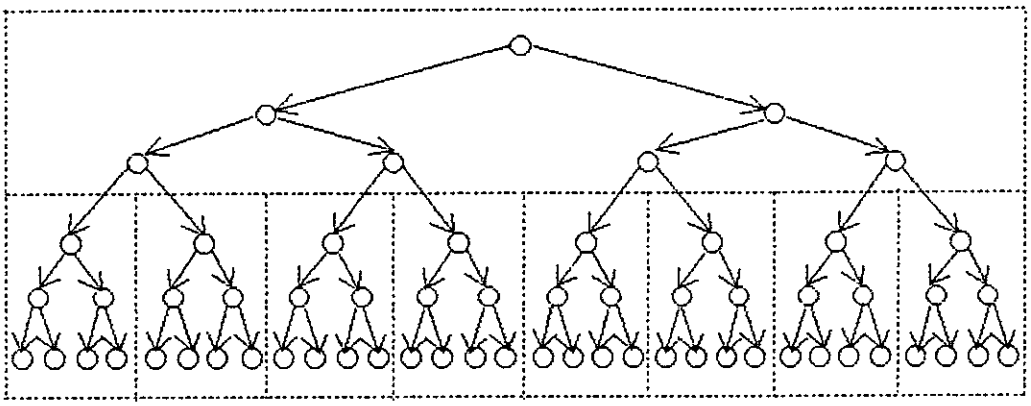


Gambar 2.3. Beberapa contoh Tree biner (*Binary tree*).

2.2. Tree Biner Banyak Cabang.

Untuk membentuk dan memelihara tree dalam skala besar yang didalamnya terjadi operasi penghapusan dan penyisipan, akan banyak membutuhkan tempat pada memori komputer, hal ini dapat diatasi dengan menggunakan tree banyak cabang. Tree banyak cabang ini merupakan perluasan dari tree biner. Dimisalkan node-node tree yang disimpan dalam media penyimpan sekunder (contohnya : *harddisk*), struktur data dinamis yang digunakan khusus untuk mengintegrasikan media penyimpan tersebut. Inovasinya terletak pada penyajian pointer sebagai alamat penyimpan sekunder bukan alamat penyimpan utama. Dengan menggunakan tree biner untuk sekumpulan data organisasi penyimpan menggunakan pengaksesan dengan waktu minimal akan sangat diperlukan. Tree banyak cabang memberikan penyelesaian akan permasalahan ini. Jika item pada penyimpan sekunder diakses, maka keseluruhan isi penyimpan dapat diakses tanpa tambahan waktu.

Hal ini memberikan pemikiran bahwa tree seharusnya terdiri atas sejumlah subtree dan subtree merupakan satu kesatuan yang dapat diakses secara bersamaan yang biasa disebut dengan halaman. Gambar 2.4 menunjukkan tree biner yang terbagi menjadi beberapa halaman, yang setiap halamannya terdiri atas 7 node maka akan terjadi penghematan dalam pengaksesan.



Gambar 2.4. Tree biner terbagi atas sejumlah "halaman".

2.3. *Trie* dan *Binary Trie*

Definisi 11.

Trie adalah *tree* yang mempunyai derajat lebih besar sama dengan dua, dimana percabangan tiap level tidak ditentukan dengan semua nilai kunci tetapi hanya oleh sebagian nilai kunci.

■

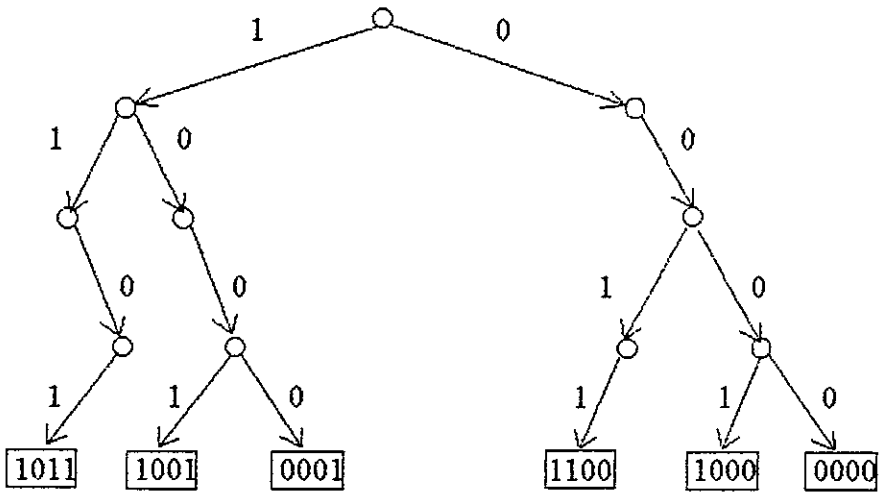
Definisi 12.

Binary Trie adalah sebuah pohon biner yang mempunyai dua macam node yaitu node elemen dan node cabang. Sebuah node cabang mempunyai dua field yaitu *LeftChild* dan *RightChild*, tidak ada field data. Sebuah node elemen mempunyai satu field data.

■

Contoh 4.

Gambar 2.5. menunjukkan contoh dari *trie* biner yang berisi 6 elemen. Elemen-elemennya adalah 1011, 1001, 0001, 1100, 1000, dan 0000.



Gambar 2.5. Contoh *Trie* biner.

Perhatikan bahwa node pada *trie* mempunyai cabang di dua arah yaitu ke kiri yang disimbolkan dengan 1 dan ke kanan yang disimbolkan dengan 0.. Hanya node daun dari *trie* yang berisi sebuah pointer ke sebuah halaman yang berisi elemen.

2.4. Tipe Data Pointer Dalam Bahasa Pemrograman Pascal

2.4.1. Pengertian Pointer

Nama peubah, yang digunakan untuk mewakili suatu nilai data, sebenarnya merupakan penunjuk suatu lokasi tertentu dalam memori komputer di mana data yang diwakili oleh nama peubah tersebut disimpan. Pada saat sebuah program dikompilasi, kompiler akan melihat pada bagian deklarasi peubah (dengan statemen *var*) untuk mengetahui nama-nama peubah sekaligus mengalokasikan tempat di memori untuk menyimpan nilai data tersebut. Sebelum program dieksekusi lokasi-lokasi dalam memori sudah ditentukan dan tidak bisa

diubah selama proses eksekusi. Peubah-peubah seperti yang diterangkan di atas disebut dengan peubah statis.

Suatu lokasi memori jika sudah ditentukan untuk suatu nama peubah maka dalam program tersebut peubah yang dimaksud akan tetap menempati memori yang telah ditentukan dan tidak bisa diubah. Dengan melihat sifat-sifat peubah statis dapat di katakan bahwa data yang akan diolah menjadi terbatas. Karena itu pascal juga menyediakan satu fasilitas disebut peubah dinamis, yaitu suatu peubah yang akan dialokasikan pada saat diperlukan yaitu setelah program dieksekusi. Dengan kata lain, pada saat program di kompilasi, lokasi untuk peubah tersebut belum ditentukan. Kompiler hanya akan mencatat bahwa ada suatu peubah akan diperlakukan sebagai peubah yang dinamis.

Pada peubah statis, isi memori pada lokasi tertentu (nilai peubah) adalah data sesungguhnya yang akan diolah. Nilai peubah pada peubah dinamis adalah alamat lokasi lain yang menyimpan data yang sesungguhnya. Dengan demikian data yang sesungguhnya tidak bisa dimasukkan secara langsung. Peubah dinamis akan menunjukkan lokasi yang berisi data sesungguhnya yang akan diproses. Pointer adalah peubah dinamis yang berarti menunjuk ke sesuatu. Definisi ini sangat abstrak untuk itu kongkretnya dengan membuat interpretasi geometri yaitu berupa gambar. Gambar ini tersaji berupa kotak yang terhubung dengan tanda panah. Dan pointer disajikan dengan tanda panah. Dalam peubah dinamis nilai data yang ditunjuk oleh pointer disebut dengan node.

2.4.2. Deklarasi Pointer dan Alokasi Tempat

Deklarasi pointer seperti halnya deklarasi tipe data lain adalah di bagian deklarasi type, dengan bentuk umum deklarasi pointer adalah:

```
type pengenal = ^node ;
    node      = tipe;
```

dengan *pengenal* : nama pengenal yang menyatakan data bertipe pointer
node : nama node
tipe : tipe data dari node

Tanda ^ di depan nama node harus ditulis karena menunjukan bahwa pengenal adalah tipe data pointer. Tipe data dari node yang dinyatakan dalam statemen *type* bisa berupa sebarang tipe data, misalnya *char, integer, real.* Misalkan deklarasinya sebagai berikut :

```
type cacah = ^integer;
```

dengan deklarasi diatas menunjukkan bahwa cacah merupakan tipe data yang bertipe pointer, dalam hal ini akan menunjuk ke data yang bertipe integer dan misalnya dipunyai deklarasi peubah sebagai berikut:

```
var x,y : cacah;
```

yang menunjukan bahwa *x* dan *y* adalah peubah bertipe pointer yang hanya bisa mencakup data yang bertipe integer. Dapat pula sekumpulan data yang dikumpulkan dalam sebuah rekord sehingga akan banyak menunjukan tipe data pointer yang elemennya (data yang ditunjuk) adalah sebuah rekord.

Contoh 5.

Sebagai contoh deklarasi dari pointer yang berisi sekumpulan data adalah :

```
type str30 = string[30];
```

```
point = ^data;
data = record
          Nama_peg : str30;
          alamat   : str30;
          pekerjaan : str30;
       end;
```

dengan deklarasi seperti diatas bisa dideklarasikan peubah sebagai berikut:

```
var p1,p2 : point;
    a,b,c  : str30;
```

□

Pada contoh diatas *p1* dan *p2* masing-masing bertipe pointer *a,b,c* merupakan peubah statis yang bertipe *string*. Saat dikompilasi peubah *p1* dan *p2* akan menempati lokasi dalam memori dan belum menunjuk ke suatu node. Suatu pointer yang belum menunjuk ke suatu node disebut **nil** (kosong).

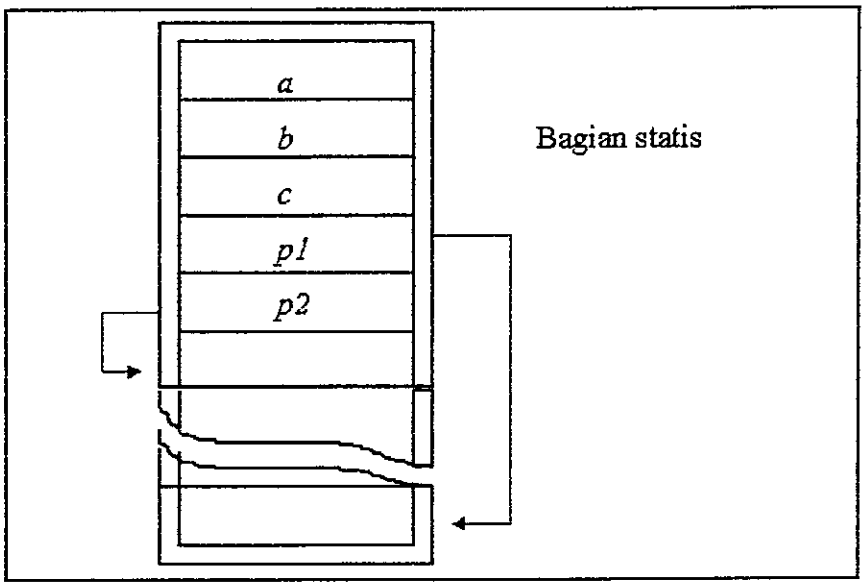
Untuk mengalokasikan node dalam memori statemen yang digunakan adalah *new* dengan bentuk umum :

```
new (peubah) ;
```

dengan peubah yang bertipe pointer. Dari deklarasi dan statemen sebagai berikut:

```
new (p1);
new (p2);
```

maka sekarang dipunyai dua node yang ditunjuk oleh *p1* dan *p2* . Jika dilihat dalam memori maka alokasi peubah-peubah diatas adalah :



Gambar 2.6. Contoh alokasi dalam memori.

□

Dalam ilustrasi diatas banyaknya memori yang diperlukan pada dasarnya juga tetap untuk lebih lanjut jika statemen :

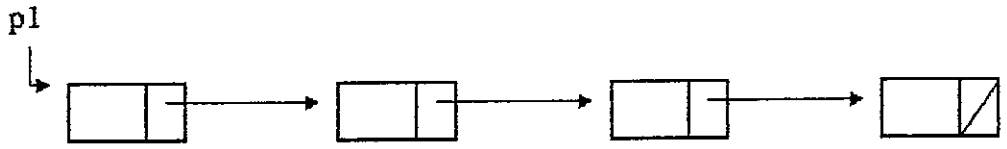
new(p1)

diulang beberapa kali maka hanya node terakhir yang dapat dimasukan , hal ini terjadi karena setiap kali diberikan statemen *new(p1)*, maka nilai *p1* yang lama akan terhapus. Dengan terhapusnya nilai *p1* otomatis node yang ditunjuk oleh *p1* tidak ditunjuk oleh pointer yang lain, sehingga karena alamatnya tidak jelas maka node tersebut tidak bisa digunakan lagi.

Dengan melihat penjelasan diatas dapat diperhatikan bahwa deklarasi diatas kedinamisannya masih tersamar, alasannya karena jika menginginkan node aktif dalam memori maka harus disediakan sejumlah pointer yang sesuai dengan

demikian seolah-olah tidak ada perbedaan yang nyata antara peubah statis dan pointer.

Jika menginginkan peubah yang benar-benar dinamis maka diharuskan mampu untuk memasukkan sejumlah lokasi tertentu dengan hanya menggunakan sebuah pointer awal, misalnya seperti gambar 2.7. berikut:



Gambar 2.7. Node-node yang membentuk senarai berantai.

Pada gambar diatas, *p1* adalah peubah yang bertipe pointer dengan nodenya yang bertipe rekord dan salah satu field dalam node tersebut juga bertipe pointer dengan tipe data yang sama dengan *p1*. Sehingga dengan memanfaatkan deklarasi tipe pointer ini, bentuk senarai berantai seperti gambar 2.7. dapat diperoleh secara mudah dengan mengubah deklarasi pointernya menjadi:

```
type   peubah = ^node ;
       node   = record
                  info    : tipe;
                  berikut : peubah
                end;
```

dengan
peubah : nama peubah yang bertipe pointer
node : nama node
info : nama field dari data yang bertipe sebarang
tipe : tipe data dari masing-masing field
berikut : nama field yang bertipe data pointer.

Jika dipunyai deklarasi data seperti dibawah ini:

```
type   str30 = string[30];
       point = ^data;
       data  = record
                 nama_mhs : str30;
                 nim      : str10;
                 alamat   : str30;
                 berikut  : point;
               end;
var    p1, p2 : point;
```

maka setelah statemen:

```
new (p1);
new (p2);
```

dieksekusi akan dipunyai dua buah node yang apabila digambarkan akan tampak seperti gambar 2.8. sebagai berikut:

Gambar 2.8. Node yang berisi field yang bertipe pointer.

Perhatikan bahwa, karena field nama, nim, nilai, dan alamat belum mempunyai nilai maka nilainya ditunjukan dengan tanda ?. Sedangkan pada field yang bertipe pointer (field berikut), karena ia tidak menunjuk ke node lain maka nilainya adalah **nil**, yang disimbolkan seperti gambar di atas. Dengan keadaan seperti di atas maka akan lebih mudah untuk menggandeng node-node yang ditunjuk oleh *p1* dan *p2*.

2.4.3. Operasi Dasar Pada Pointer

2.4.3.1. Mengkopi Pointer

Operasi dasar pada pointer ada dua yaitu operasi mengkopi pointer, sehingga sebuah node akan ditunjuk lebih dari sebuah pointer ,dan operasi mengkopi isi pointer, sehingga dua atau lebih node yang ditunjuk oleh pointer yang berbeda mempunyai isi yang sama. Untuk memahaminya perhatikan ilustrasi sebagai berikut.

Pertama kali deklarasikan tipe pointernya, yaitu:

```
type node = ^data;
    data  = record
              nama    : string;
              alamat  : string;
              berikut : node
            end;
var t1, t2  : node;
```

Pada deklarasi diatas pointer *t1* dan *t2* mempunyai deklarasi yang sama, sehingga mempunyai syarat untuk kedua operasi diatas. Setelah itu berikan statemen sebagai berikut:

```
new (t1);
new (t2);
```

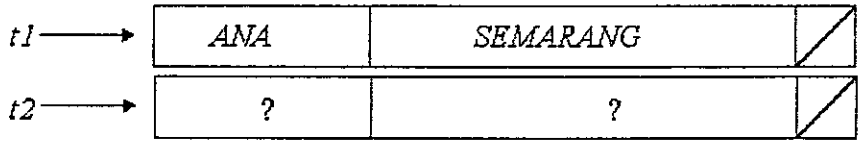
dipunyai dua node, yaitu:

Gambar 2.9. Dua node dari pointer *t1* dan *t2*.

dengan menggunakan statemen:

```
t1^.nama  := 'ANA';
t1^.alamat := 'SEMARANG';
```

maka keadaan node menjadi :

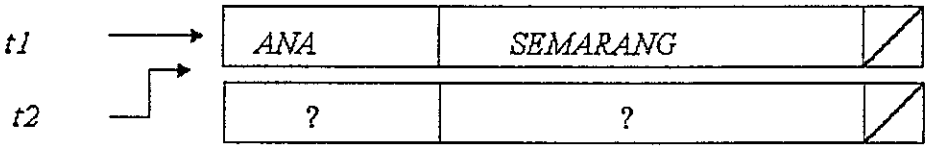


Gambar 2.10. Node *t1* setelah adanya pemasukan data.

jika memberikan statemen:

```
t2 := t1;
```

maka gambar di atas akan menjadi :



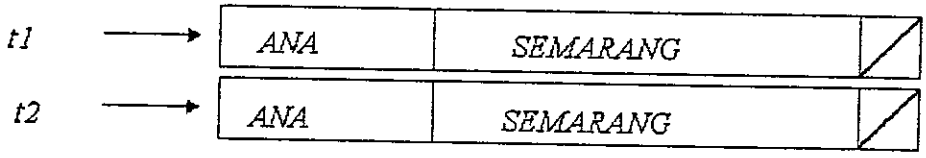
Gambar 2.11. Node *t1* dan *t2* setelah adanya statemen *t2 := t1;*

Perhatikan bahwa sekarang pointer *t2* juga menunjuk ke node yang di tunjuk oleh pointer *t1*; node yang semula ditunjuk oleh *t2* menjadi terlepas dalam keadaan seperti ini karena tidak ditunjuk oleh pointer lain, maka node tersebut tidak bisa digunakan lagi karena lokasi node tersebut dalam memori tidak diketahui lagi (kecuali apabila node ini ditunjuk oleh pointer lain). Operasi inilah yang disebut dengan operasi mengkopi pointer.

Kembali lagi pada gambar 2.10., jika statemen yang diberikan adalah:

t2^ := t1^;

maka hasil yang diperoleh adalah:



Gambar 2.12. Node *t1* dan *t2* setelah adanya statemen *t2^ := t1^;*

Dari ilustrasi diatas, jelaslah apa yang dimaksud dengan operasi mengkopi pointer dan operasi mengkopi isi node. Sebagai ringkasan, maka:

- jika dalam statemen pemberian tanda ^ tidak ditulis, operasinya disebut dengan operasi mengkopi pointer dengan konsekuensi node yang semula ditunjuk oleh suatu pointer akan bisa terlepas dan tidak berfungsi lagi.
- jika dalam statemen pemberian tanda ^ ditulis operasinya disebut operasi mengkopi isi node pointer dengan konsekuensi isi dua node atau lebih akan sama.

Perlu diingat bahwa statemen pemberian hanya bisa dilaksanakan untuk peubah-peubah yang bertipe sama. Dengan demikian statemen yang salah adalah sebagai berikut :

```
t1  := t2^;
t2^ := t1;
```

Untuk pointer, karena hanya berurusan dengan alamat tetentu dalam memori maka hanya bisa dioperasikan dengan dua operasi yaitu = dan <>. Sebagai contoh, misalnya:

```
if t1 = t2 then
if t1 = nil then
while t2 <> nil do
```

Perhatikan bahwa kondisi *t1 = t2* menanyakan apakah kedua pointer ini menunjuk ke lokasi yang sama. Untuk node maka semua operator relasi bisa digunakan. Kondisi-kondisi seperti:

```
if t1^ = t2^ then
```

dan

```
while t1^ <> t2^ do
```

akan membandingkan isi node yang ditunjuk oleh pointer yang ditunjuk oleh pointer *t1* dan *t2*.

2.4.3.2. Menghapus Pointer

Pointer yang telah dibentuk dapat dihapus kembali saat program dieksekusi. Setelah pointer dihapus, maka lokasi semula yang ditunjuk oleh pointer tersebut akan bebas, sehingga bisa digunakan oleh peubah lain.

Statemen untuk menghapus pointer adalah statemen *dispose*, yang mempunyai bentuk umum:

```
dispose (peubah);
```

dengan *peubah* adalah sebarang peubah yang bertipe pointer.

2.5. Algoritma

Algoritma adalah himpunan instruksi-instruksi yang menjelaskan langkah-langkah yang teratur dan berhingga untuk menyelesaikan suatu permasalahan. Sehingga dengan langkah-langkah yang terperinci dan terurut akan mempermudah pembuatan logika program sebagai dasar penulisan program.

2.5.1. Kriteria Algoritma yang Baik

1. Adanya Input

   Suatu algoritma sekurang-kurangnya mempunyai sejumlah nol atau lebih input yang disuplai secara eksternal.

2. Adanya Output

   Input suatu algoritma harus memberikan output sekurang-kurangnya sejumlah satu output .

3. Kepastian

   Tiap-tiap instruksi dalam algoritma harus jelas dan tidak mempunyai arti ganda.

4. Keberhinggaan

   Apabila instruksi-instruksi suatu algoritma ditelusuri, maka untuk semua kasus, algoritma berhenti setelah sejumlah berhingga langkah-langkah.

5. Keefektifan

   Setiap instruksi harus cukup mendasar untuk dilaksanakan (langsung mengarah ke sasaran yang dituju). Tidaklah cukup bahwa setiap operasi

adalah definite seperti dalam (3), tetapi juga harus fisibel yaitu dapat dikerjakan dengan mudah.

2.5.2. Cara Menyatakan Algoritma

1. Dengan bahasa alamiah.

   Algoritma ditulis dengan ungkapan bahasa sehari-hari

2. Dengan sandi semu (*Pseudocode*).

   Algoritma yang telah ditulis dengan menggunakan perintah bahasa pemrograman tertentu ditambah dengan bahasa alamiah.

3. Dengan Bagan Alir (*flowchart*).

   Algoritma ditulis dalam bentuk simbol/bentuk diagram.

2.6. Istilah-istilah Dalam Basis Data (*Database*)

2.6.1. Entity, Atribut dan Data Value

Definisi 13.

Entity adalah orang, tempat, kejadian atau konsep yang informasinya disimpan. Atribut adalah sebutan untuk mewakili suatu entity. Data Value (nilai atau isi data) adalah data aktual atau informasi yang disimpan pada tiap data elemen atau atribute.

■

Contoh 6.

Dalam bidang administrasi mahasiswa misalnya, entity adalah mahasiswa, nilai ujian, indeks prestasi. Entity mahasiswa dapat mempunyai atribut misalnya nama,

nim, alamat, nama orang tua. Atribut juga disebut sebagai data elemen, data field, data item. Atribut nama mahasiswa menunjukkan tempat dimana informasi nama mahasiswa disimpan, sedang data value adalah Ana Yuli Astutik, Tohir, merupakan isi nama mahasiswa tersebut.

□

2.6.2. Rekord

Definisi 14.

Rekord adalah kumpulan elemen-elemen yang saling berkaitan menginformasikan suatu entity secara lengkap. Satu rekord mewakili satu data atau informasi.

■

Contoh 7.

Rekord dari suatu entity mahasiswa adalah nim, nama, alamat, nama orang tua, nilai mata kuliah, indeks prestasi kumulatif. □

2.6.3. File

Definisi 15.

File adalah kumpulan rekord-rekord sejenis yang mempunyai panjang elemen yang sama, atribut yang sama, namun berbeda-beda data valuenya. ■

2.6.4. Basis Data dan DBMS

Definisi 16.

Database adalah kumpulan file-file yang mempunyai kaitan antara satu file dengan file yang lain sehingga membentuk satu bangunan data untuk menginformasikan satu perguruan tinggi, instansi dan lain-lain. DBMS adalah pengelola kumpulan file yang berkaitan bersama dengan program. Database adalah kumpulan datanya, sedang program pengelolanya berdiri sendiri dalam satu paket program untuk membaca data, mengisi data, menghapus data, melaporkan data dalam database. ■

2.7. Fungsi Hashing

2.7.1. Kepadatan Pengenal

Definisi 17.

Kepadatan pengenal (*the identifier density*) dari suatu tabel hash adalah perbandingan n/T, dengan n adalah jumlah pengenal dalam tabel dan T adalah total jumlah pengenal yang mungkin. ■

2.7.2. Kepadatan beban atau faktor beban

Definisi 18.

Kepadatan beban (*the loading density*) atau faktor beban (*the loading factor*) dari suatu tabel hash adalah

$$\alpha = n / (sj)$$

dengan j jumlah halaman suatu tabel hash dan masing-masing halaman mampu menampung s slot.

■

Contoh 8.

Perhatikan tabel hash T dengan $j = 26$ halaman dan $s = 3$. Diasumsikan $n = 9$ pengenal berbeda dan tiap pengenal dimulai dengan suatu huruf. Faktor beban untuk tabel ini adalah $\alpha = 9/(26.3) = 9 / 78 = 0{,}12$.

□

2.7.3. Synonim, Overflow dan Tabrakan

Definisi 19.

Dua kunci pengenal k_1 dan k_2 dikatakan synonim terhadap fungsi hash h apabila $h(k_1) = h(k_2)$. *Overflow* terjadi apabila suatu kunci pengenal k dipetakan / di-hash-kan oleh suatu fungsi hash h ke dalam halaman yang penuh.

Tabrakan (*collision*) terjadi apabila dua kunci pengenal yang tidak identik di-hash-kan ke dalam slot yang sama.

■

Dengan demikian, sinonim-sinonim yang berbeda dimasukan ke dalam halaman yang sama sepanjang slot-slot dalam halaman itu belum penuh. Apabila slot-slot suatu halaman telah penuh, penyisipan kunci pengenal baru ke dalam halaman itu mengakibatkan *overflow*. Tentu saja, apabila ukuran slot s adalah 1, tabrakan dan *overflow* terjadi bersamaan.

2.7.4. Tahap Persiapan

Penggunaan metode hashing untuk pencarian memerlukan persiapan awal sebelum fungsi hash tertentu digunakan. Hal ini dikarenakan tiap elemen kunci sering mengandung karakter-karakter alfanumerik. Beberapa karakter-karakter aritmatika ataupun logika sulit dimanipulasi, sehingga tepat untuk mengkonversi kunci-kunci yang demikian agar dapat lebih mudah dimanipulasi oleh suatu fungsi hash. Proses konversi ini sering disebut dengan persiapan (*preconditioning*).

Cara persiapan yang paling efisien dilakukan adalah dengan menggunakan perwakilan internal yang disandikan secara numerik (*the numerically coded internal representation*) (misalnya ASCII) dari tiap karakter kuncinya.

Contoh 9.

Untuk kunci A1, dalam ASCII, karakter A mempunyai nilai $65_{10} = 1000001_2$ dan karakter 1 mempunyai nilai $49_{10} = 0110001_2$. Dengan menginterprestasikan kunci-kunci sebagai bilangan basis 2 14-digit, kunci A1 mempunyai kode biner 10000010110001_2 nilai ini sama dengan 8345_{10}. □

Umumnya hasil persiapan sebuah kunci mungkin tidak cocok dengan ukuran suatu *word* memori. Dalam kejadian seperti itu, digit-digit tertentu hasil persiapan tersebut dapat dibatalkan, misalnya dengan menggunakan suatu fungsi hash yang melakukan transformasi pengurangan ukuran. Hal ini dilakukan mengingat bahwa seringkali satu fungsi hash membangun hasil persiapan itu dan kemudian fungsi hash kedua memetakan hasil ini ke dalam lokasi tabel.

Akhirnya, tentu saja bagi kunci yang tiap elemennya hanya terdiri dari karakter-karakter numerik tahap persiapan ini dapat tidak diperlukan (yaitu apabila elemen-elemen kuncinya ingin diinterprestasikan sebagai bilangan bulat).

2.7.5. Kriteria Pemilihan Fungsi Hashing

Ada beberapa kriteria yang harus diperhatikan untuk pemilihan suatu fungsi hash yang baik, yaitu :

1. Kecepatan dan kemudahan perhitungannya.

   Suatu fungsi hash yang baik haruslah sangat cepat dan mudah untuk memanipulasi suatu kunci.

2. Ketergantungan pada bit-bit kunci.

   Fungsi hash yang baik harus bergantung pada semua bit-bit kunci agar tidak ada informasi yang hilang. Kejadian kehilangan informasi ini dapat memperbesar tabrakan yang terjadi, sehingga dapat pula menyebabkan kegagalan suatu operasi tabel simbol.

3. Fungsi hash seharusnya menghasilkan nilai acak, dengan demikian akan menghindari tabrakan atau paling tidak meminimumkan.

   Fungsi hash menghasilkan sebuah elemen kunci untuk diletakkan dalam suatu tabel dan mentransformasikannya ke dalam sebarang lokasi dalam tabel itu. Fungsi hash ideal menyebarkan elemen-elemen itu (dengan menghasilkan nilai random) secara seragam diseluruh tabelnya.

Kenyataannya, tidaklah mungkin mendapatkan suatu fungsi hash yang memilih suatu lokasi random dimana kuncinya diletakkan. Hal ini dikarenakan fungsi hash *h* tidak dapat bersifat kemungkinan (*probabilistic*) tetapi ia harus bersifat menentukan (*deterministic*) yaitu menghasilkan lokasi yang sama setiap waktu diterapkan dalam elemen yang sama.

Fungsi hash yang baik memenuhi (mendekati) asumsi hashing seragam sederhana : tiap kunci berkesempatan sama (*equally likely*) untuk di-hash-kan ke sebarang *j* buah halaman.