6 SORTIR

6.1 SORTIR TERHADAP *RECORD*

Sebelum berbicara tentang sortir secara umum, kita ulang secara singkat pembicaraan tentang *file* dan *record*, yang telah kita bicarakan pada Bab 2 yang lalu. *File* adalah himpunan *record*. Misalnya suatu perusahaan mempunyai *file* yang berisi seluruh data yang diperlukan oleh perusahaan itu tentang para pegawainya. Data dari masing-masing pegawai disebut *record*. Jadi, setiap orang pegawai mempunyai satu *record*.

Record seorang pegawai dapat berisi :

- nomor pegawai
- nama pegawai
- jenis kelamin
- status perkawinan
- jabatan
- gaji pokok
- tunjangan
- jumlah anak
- ikut KB

dan lain-lain, sesuai dengan informasi yang dibutuhkan perusahaan tadi.

Jadi suatu *record* adalah himpunan elemen yang bersifat heterogen, yang dianggap sebagai satu unit struktur data. Heterogen di sini maksudnya adalah bahwa elemen dari suatu *record* boleh saja mempunyai tipe data yang berlainan. Misalnya untuk nama pegawai biasanya digunakan tipe data alfabetik yakni tipe data *string*, jumlah anak bertipe data numerik berupa *integer*, gaji pokok bertipe numerik berupa *real* dan sebagainya. Dapat dicatat bahwa *field* ikut KB dapat merupakan tipe *boolean*, yakni bernilai Y jika ikut, dan bernilai T jika tidak ikut.

Elemen dari *record* kita sebut *field*. Kalau kita lihat contoh tadi, maka nomor pegawai, nama pegawai dan seterusnya masing-masing adalah *field*. Tiap *record* dapat mempunyai banyak *field* sebarang sesuai kebutuhan. Jadi, suatu *field* adalah bagian dari suatu *record* yang berisi suatu informasi tertentu. Perhatikan penggambaran potongan suatu *record* seperti pada Tabel 6.1.

Table 6.1. *Contoh sebuah file dengan 2 records*

No pegawai	nama pegawai	jumlah anak	gaji pokok	ikut KB
012557562	Haikal Delon	3	$ 723,570	Y
032354786	Joy Tia	2	$ 625,250	T

Suatu *record* biasanya mengandung *field* penunjuk, yang biasanya digunakan sebagai kunci untuk memanggil *record* tersebut. *Field* penunjuk ini biasa kita sebut sebagai "*key*" (atribut kunci) dari suatu *record*. Pada contoh di Gambar 6.1, yang dapat kita ambil sebagai KEY misalnya adalah NO PEGAWAI (nomor induk pegawai) Jadi, jika kita ingin mengetahui data tentang "Haikal Delon," maka yang kita panggil adalah atribut kuncinya, yakni nomor induk pegawai 012557562.

Dalam suatu *file*, atribut kunci inilah yang biasanya ingin kita urutkan, boleh diurutkan dari kecil ke besar (urut menaik atau *ascending*), ataupun sebaliknya (urut menurun atau *descending*): Cara penyusunan inilah yang kita sebut sebagai sortir (dari kata *sorting*). Jadi sortir terhadap *file* adalah suatu proses pengurutan sekumpulan *record*, sedemikian sehingga :

$$KEY(I) >= KEY(J)$$

untuk setiap I < J (dalam urut *ascending*)

atau

$$KEY(I) <= KEY(J)$$

untuk setiap I > J atau J < I (dalam urut *descending*). Di sini, KEY(I) adalah nilai data (data *value*) KEY dari *record* ke I.

Secara umum, sortir dapat dilakukan terhadap suatu himpunan bilangan, ataupun terhadap himpunan *string*, ataupun himpunan lain yang bersifat ordinal. Ada 2 kategori sortir berdasar media yang digunakan

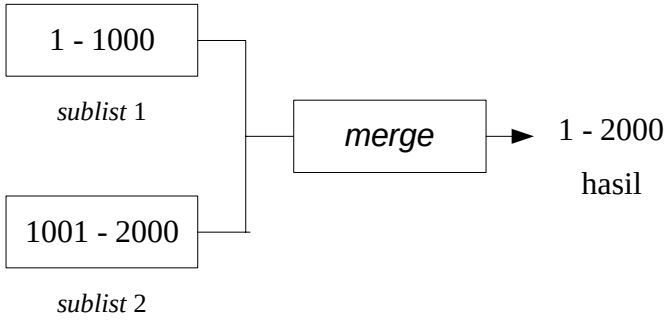
1. **Sortir internal**. Metode ini dipakai jika himpunan data yang akan disortir itu adalah kecil sehingga proses sortir tidak membutuhkan tempat yang besar di memori utama komputer.

2 . **Sortir eksternal**. Metode ini baik untuk dipakai jika himpunan data yang akan disortir cukup besar. Di sini kita membutuhkan media atau alat tambahan, seperti *magnetic tape*, disket, dan sebagainya.

Kita dapat melakukan beberapa operasi pada *record*. Kita bisa menyisipkan (*insert*) sebuah KEY, kita dapat juga menghapus (*delete*) sebuah KEY, dan kita dapat pula menukar posisi dari dua buah KEY. Pada waktu kita melakukan penyisipan, penghapusan ataupun penukaran posisi dari dua buah KEY, selain *field* KEY yang berubah, *field* lain yang terdapat pada *record* tersebut juga akan berubah.

6.1.1 METODE SORTIR GABUNG (MERGESORT)

Misalkan kita mempunyai 2000 *record* yang akan kita sortir, namun hanya 1000 *record* yang dapat disimpan di dalam memori utama. Masalah ini akan diselesaikan dengan suatu metode "Sortir Gabung," yakni dengan memisahkan mereka menjadi dua kelompok yang berdiri sendiri, yakni record 1 sampai dengan 1000, dan 1001 sampai dengan 2000. Hasil penerapan dari sortir internal terhadap masing-masing kelompok, akan berbentuk dua buah *sublist* terurut. Kemudian kedua *sublist* tersebut kita gabung (*merge*), menghasilkan *file* yang terurut yang kita inginkan. Lihat keterangan di bawah ini :



Gambar 6.2. *Skema mergesort*

6.1.2 METODE SORTIR NATURAL MERGE DAN BALANCED MERGE

Ada beberapa jenis sortir gabung, di antaranya yang akan kita bahas adalah sortir gabung natural (*natural merge*) dan sortir gabung seimbang (*balanced merge*).

Jika pada *natural merge* kita menggunakan 1 *output file*, maka pada *balanced merge* banyaknya *output file* tergantung pada *input file*nya. Bila kita gunakan 2-*way balanced merge*, maka *input file* ada 2, dan *output file* ada 2 pula, sedangkan jika dengan 3-*way balanced merge*, *input file* ada 3, dan *output file* 3 pula. Secara umum, jika digunakan M-*way balanced merge*, maka terdapat M buah *input file*, dan M buah *output file*.

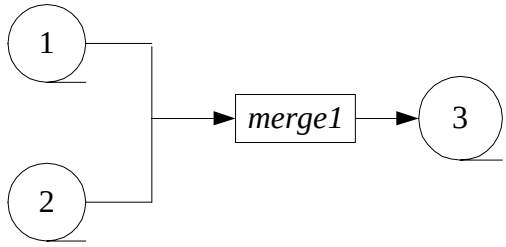
Misalkan *file* yang terdiri atas 6000 *record* dibagi menjadi 12 buah *subfile* yang masing-masing terdiri dari 500 *record*. Jika digunakan *natural merge*, maka kita memerlukan 3 buah *tape*, 2 buah untuk menampung *file input* dan sebuah untuk menampung *file output*. Prosesnya terlihat pada Gambar 6.3 berikut :

Tape T1

5001-5500	4001-4500	3001-3500	2001-2500	1001-1500	1-500

Tape T2

5501-6000	4501-5000	3501-4000	2501-3000	1101-2000	501-1000

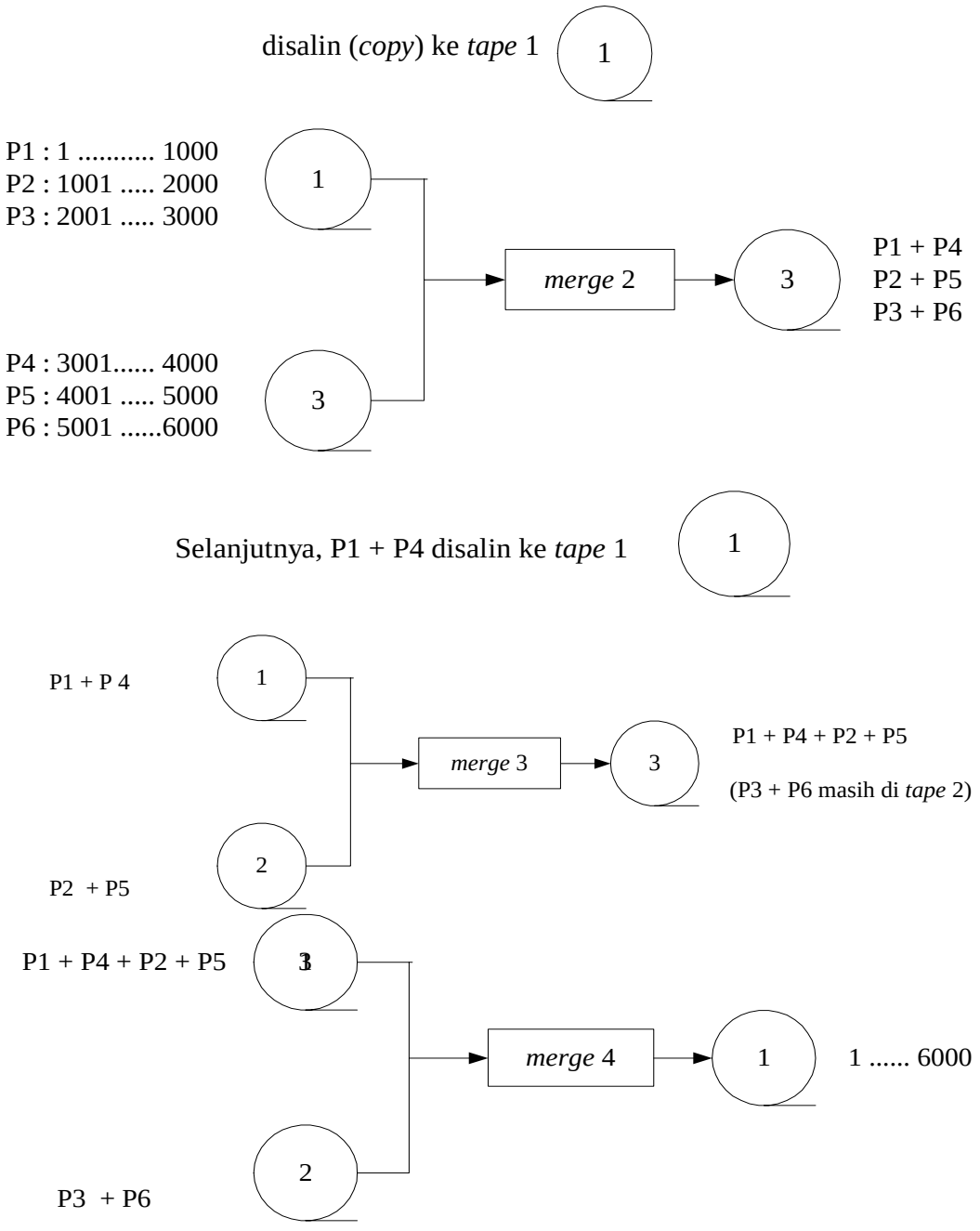


Selanjutnya, P3, P2, dan P1 :

1 …….. 1000

1001 …… 2000

2001 …… 3000



Catatan : Yang dimaksud dengan P1 + P4 adalah *merge* antara P1 dan P4

Gambar 6.3. *Proses merge sort*

6.1.3 BALANCED MERGE

Balanced merge adalah metode yang paling sederhana dan mudah untuk meng-gabungkan *file* partisi yang dihasilkan dari pengolahan suatu *file* yang tidak disortir. Metode ini benar-benar efektif dan diperbaiki dengan mengelompokkannya dalam 2-*way*, 3-*way* atau *merge* M-*way* yang urutannya lebih tinggi.

Untuk *balanced merge* dengan 2-*way* kita memerlukan *file* penyimpanan dengan 4 alat penyimpan. Biasanya terdapat sejumlah besar tahap pengolahan yang terlibat, dan 2 *file* yang digunakan untuk input dan 2 *file* untuk output. Katakanlah, kita mempergunakan 4 *tape* pada 4 buah *drive*.

Tahap *merge* 1 (pada gambar 6.4) adalah hasil pembangkitan dari 2 *file* partisi yang disortir (pada tape T3) dengan menggunakan *generator*, misalnya *generator* itu menghasilkan 8 partisi yang disortir yakni P1, P2, P3, P4 ... P8, dan secara bergantian ditempatkan pada T1 dan T2 untuk menjamin bahwa *file* ini menerima sejumlah partisi yang sama.

Pada tahap *merge* 1 tape T1 dan *tape* T'2 digabungkan, sehingga dihasilkan 2 *file* partisi, yang telah disortir, pada *tape* T3 dan *tape* T4. Tape T3 perlu diputar kembali sebelum tahap ini dapat dimulai. Untuk lebih jelasnya lihat Gambar 6.4. Pasangan pertama dari partisi T1 dan T2 digabungkan untuk memberikan partisi pertama pada T3, kemudian pasangan kedua dari partisi pada T1 dan T2 digabungkan untuk memberikan partisi pertama pada T4. Proses ini berlanjut secara berurutan sehingga gabungan T1 dan T2 menghasilkan partisi yang baru.

Pada *merge* 2 semua *tape* diputar kembali kemudian pada tahap itu T3 dan T4 diaplikasikan, hasilnya dimuat dalam 2 *tape* T1 dan T2. Dan akhirnya pada *merge* 3 dihasilkan *output file* yang telah tersortir pada *tape* T3. Berikut ini diberikan suatu contoh pelaksanaan *balanced merge*, yang kemudian selanjutnya diikuti dengan gambaran skematik proses tersebut.

Contoh 6.1

Misalkan *file* yang terdiri atas 6000 *record* dibagi menjadi 12 buah *subfile* yang masing-masing terdiri dari 500 *record*. Kita akan menggunakan *balanced merge* 2-*way*.

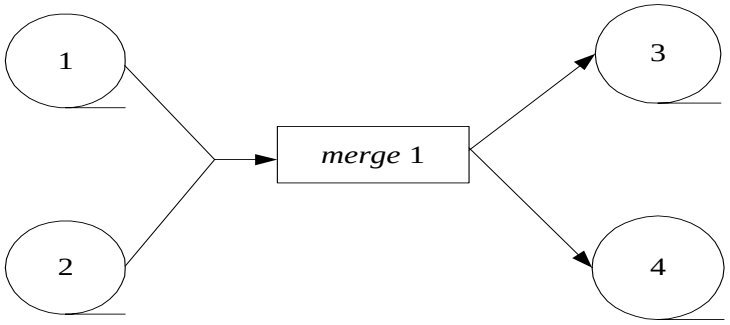
Di sini kita memerlukan 4 buah *tape*, 2 buah *tape* T1 dan T2 untuk menampung *file input* dan 2 buah *tape* T3 dan T4 untuk menampung *file output*. Namun yang perlu dicatat, bahwa sesungguhnya kita cukup menggunakan 2 buah *tape* saja, karena *tape* T3 dan T4 dapat dirangkap oleh *tape* T1 dan T2. Prosesnya adalah terlihat pada Gambar 6.4 berikut :

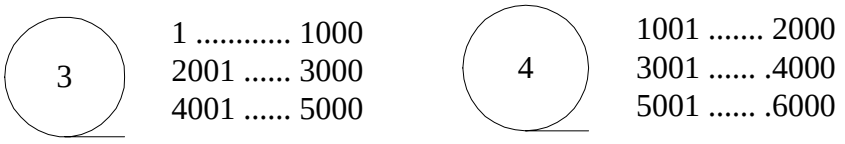
Tape T1

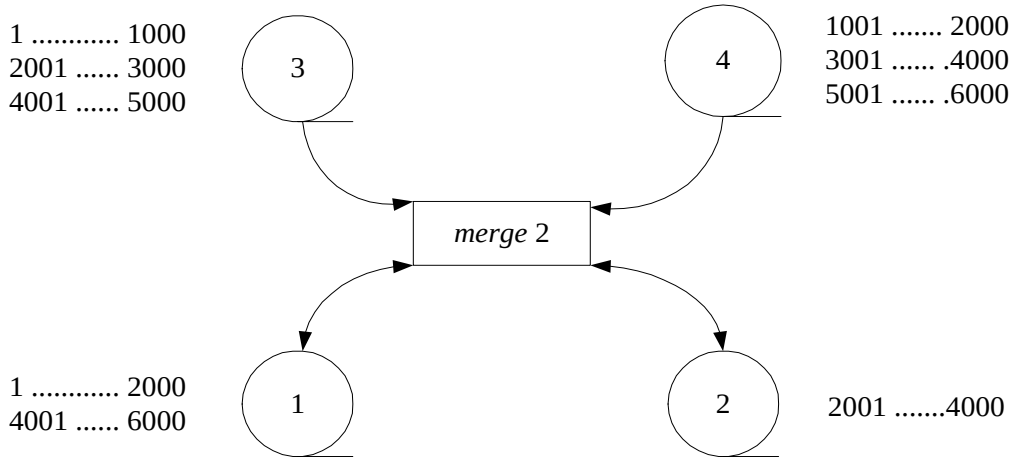
5001-5500	4001-4500	3001-3500	2001-2500	1001-1500	1-500

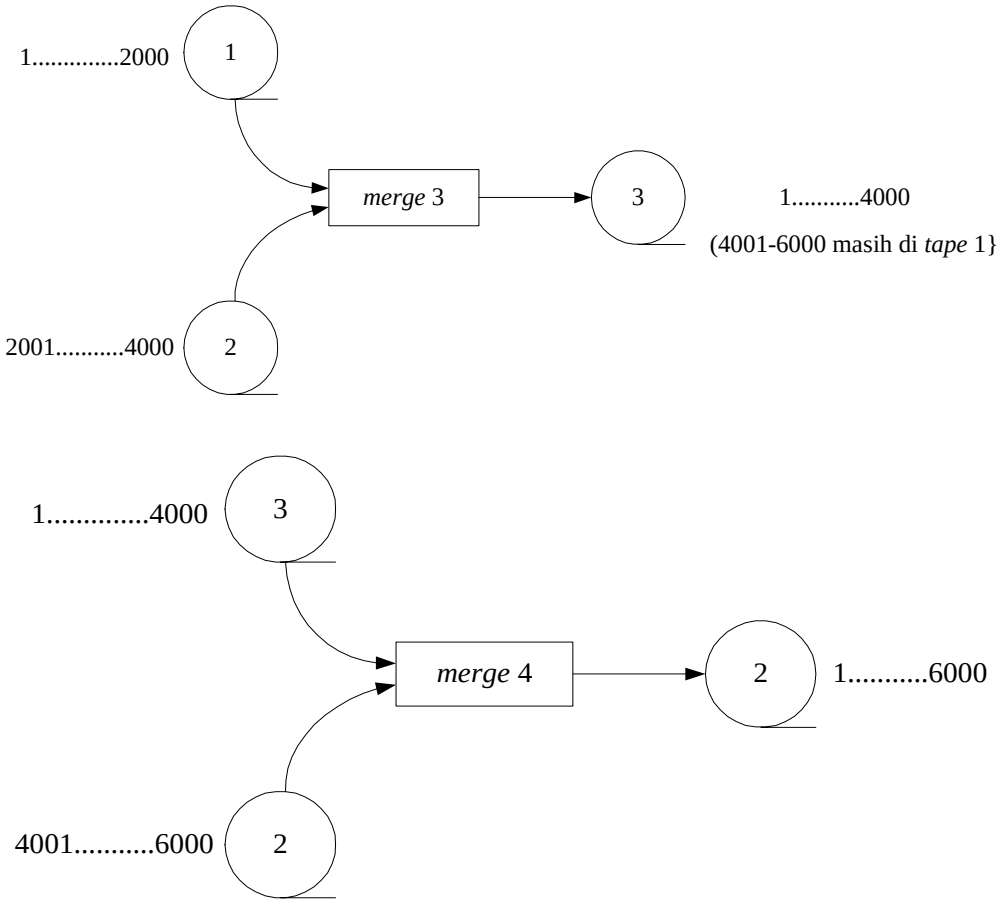
Tape T2

5501-6000	4501-5000	3501-4000	2501-3000	1101-2000	501-1000

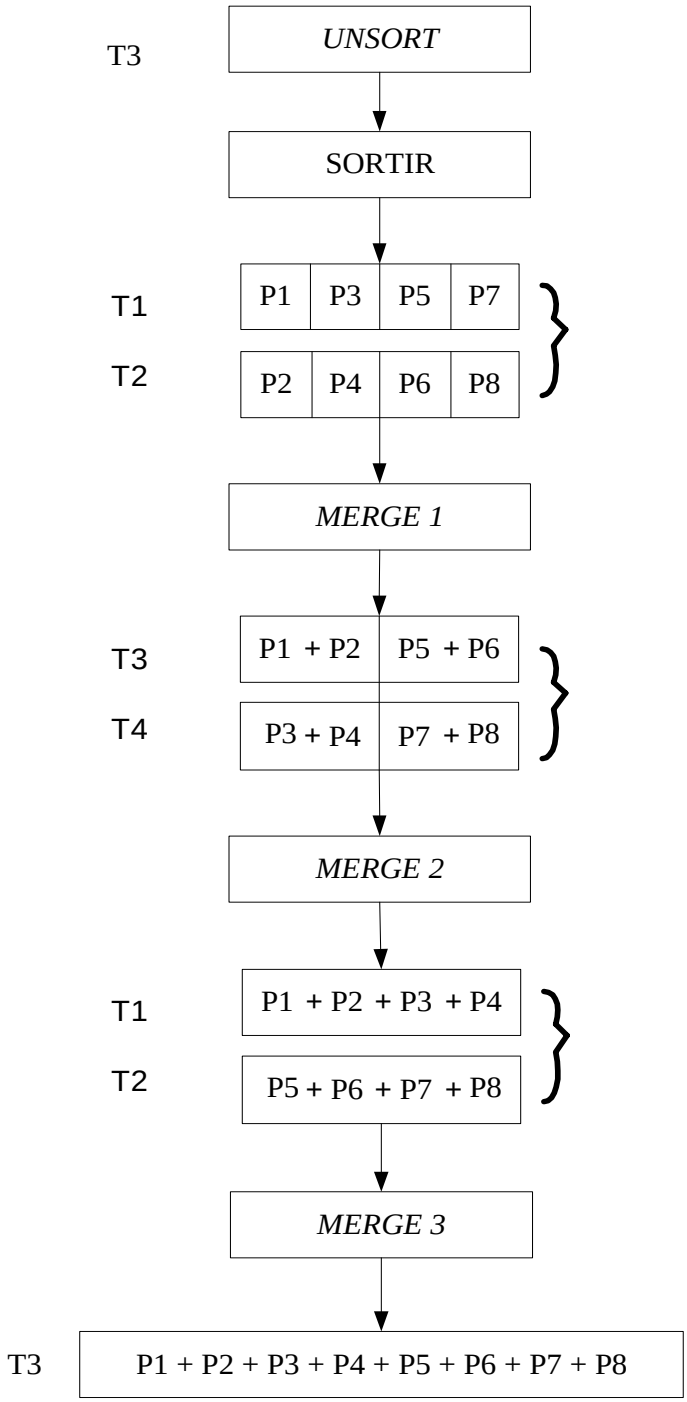








Gambar 6.4. *Gambaran skematik pendistribusian balanced merge ke tape*



Gambar 6.5. *Algoritma merge 3-way*

Kita dapat mengetahui bahwa proses mengombinasikan secara umum melibatkan pengambilan/ penggunaan 2 *tape* yakni *tape* T1 dan *tape* T2 dengan menggabungkannya. Kemudian didistribusikan pada 2 *tape* yang lain yakni *tape* T3 dan *tape* T4 dan dilanjutkan lagi dengan menggabungkannya pada *tape* T1 dan T2. Ini berlangsung secara bergantian sampai mempunyai satu partisi *output* pada satu *tape*.

Algoritma yang umum dipakai adalah :

1. Gabungkan partisi yang disortir pada TI dan T2.
2. Distribusikan secara bergantian partisi yang digabungkan pada T3 dan T4.
3. Jika hanya satu partisi yang didistribusikan, maka berhenti.
4. Jalankan semua *tape*.
5. Gabungkan partisi yang disortir pada T3 dan T4
6. Didistribusikan secara bergantian partisi yang dikombinasikan pada *tape* T1 dan T2.
7. Jika hanya satu partisi yang didistribusikan, maka berhenti.
8 Jalankan semua *tape*.
9. Ulangi langkah 1

6.2 TEKNIK SORTIR PENYISIPAN

Dua hal yang sangat mempengaruhi kecepatan algoritma sortir adalah jumlah operasi perbandingan yang dilakukan dan jumlah operasi pemindahan data dilakukan. Berlainan dengan proses pencarian data, pada proses sortir data juga harus diperhatikan jumlah pemindahan data atau *data movement* yang dilakukan. Hal ini penting sekali karena pada proses sortir, isi daftar sebagai input akan berubah menjadi output daftar yang sudah terurut. Oleh karena itu banyak proses pemindahan data yang dilakukan jelas akan mempengaruhi kecepatan algoritma.

Pada garis besarnya ada tiga teknik utama yang dapat dilakukan dalam melakukan sortir. Ketiga teknik tersebut adalah :

1. Sortir penyisipan atau *insertion sort*
2. Sortir pemilihan atau *selection sort*
3. Sortir penukaran atau *exchange sort*

Keuntungan dan kerugian dari masing-masing teknik, baik dalam hal operasi perbandingan maupun pemindahan data, akan dipelajari berikut ini. Untuk memudahkan,

maka diasumsikan bahwa urutan akhir yang dikehendaki adalah urutan dari kecil ke besar atau *ascending*.

Teknik pertama yang akan dibahas adalah teknik sortir penyisipan. Teknik ini sangat sederhana dan paling mudah untuk dimengerti maupun diterapkan. Prinsip dasar dari teknik ini adalah secara berulang-ulang memasukkan setiap kata ke dalam tempatnya yang benar. Cara ini biasanya digunakan oleh para pemain kartu pada saat mereka sedang menyusun kartu mereka.

Contoh 6.2

Urutkan 8 bilangan berikut ini :

44 55 12 42 94 18 7 67

Kita mulai dengan i = 2

i = 2

Kita bandingkan elemen ke 2, yakni 55 dengan elemen pertama, 44. Karena 55 > 44 tidak dilakukan pemindahan.

44 55 12 42 994 18 7 67

Di sini a[1] dan a[2] sudah terurut

i = 3

Kita bandingkan elemen ke 3, yakni 12 dengan elemen ke 2, 55. Tukarkan posisi mereka, sehingga a[2] = 12, a[3] = 55. Lalu perbandingkan 12 dengan 44, pertukarkan lagi. Hasilnya :

12 44 55 42 94 18 7 67

Sampai sini a[1], a[2], dan a[3] sudah terurut.

i = 4

Kita bandingkan elemen ke 4, yakni 42 dengan elemen ke 3, yakni 55. Tukarkan posisi mereka, sehingga a[3] = 12, a[4] = 55. Lalu perbandingkan 42 dengan 44, pertukaran lagi. Selanjutnya antara 42 dengan 12 tidak kita lakukan pertukaran. Sehingga :

12 42 44 55 94 18 7 67

Di sini a[1],... , a[4] sudah terurut, dan seterusnya

i = 5 hasilnya : 12 42 44 55 94 18 7 67

i = 6 hasilnya : 12 18 42 44 55 94 7 67

i = 7 hasilnya : 7 12 18 42 44 55 94 67

i = 8 hasilnya : 7 12 18 42 .44 55 67 94

Jadi pada setiap langkah ke i, subdaftar a[1],.... a[i] sudah terurut. Untuk dapat memasukkan x ke dalam tempat yang sebenarnya, maka harus dilakukan perbandingan dan pemindahan secara bergantian. Jadi x akan bergeser ke kiri dengan membandingkan nilai x dengan nilai a[j] sebelumnya, dan kemudian x disisipkan ke dalam nilai tempatnya, atau a[j] dipindahkan ke kanan. Hal ini diteruskan untuk unsur di sebelah kiri a[j]. Proses ini akan berhenti bila salah satu dari kedua hal berikut ini berlaku :

1. Salah satu unsur a[j] mempunyai *key* yang lebih kecil dari x,

2. Bagian ujung kiri daftar telah dicapai.

Untuk dapat melakukan pengecekan dengan mudah, kita tambahkan suatu unsur tambahan di sebelah ujung kiri, yakni a[0], dan diberi nilai x. Berikut ini adalah garis besar prosedur *insortion sort* :

```
Procedure  insertion sort;
var i, j, n, x : integer;
begin
for i: = 2 to n do
   begin
      x := a[i];
      a[0] = x;
      j := i – 1
      while x < a[j] do
      begin
       a[j+1] := a[j]
       j := j-1
      end;
      a[j+1] := x
   end;
end;
```

Kompleksitas Algoritma Sortir Penyisipan

Banyaknya perbandingan *f*(n) di dalam algoritma sortir penyisipan sangat mudah dihitung. Yang pertama adalah kondisi terburuk (*worst case*) yaitu ketika *array* A (susunan data yang akan diproses) terbalik (dari besar ke kecil) sehingga *looping* yang dilakukan akan sebanyak k-1 perbandingan.

$$f(n) = 1 + 2 + \ldots + (n-1) = (n^2 - n)/2 = O(n^2)$$

Jika dihitung rata-ratanya, perkiraan perbandingan yang dilakukan oleh *looping* adalah (k-1)/2, maka :

$$f(n) = 1/2 + 2/2 + \ldots + (n-1)/2 = (n^2 - n)/4 = O(n^2)$$

Apabila Algoritma di atas diperhatikan secara seksama, terlihat bahwa setelah iterasi ke-i, data yang ada dalam suatu subdaftar sudah terurut. Dalam hal ini, subdaftar a[1], a[2], ... a[i] sudah terurut. Untuk iterasi berikutnya harus dimasukkan di antara a[1] .... a[i] suatu nilai x yang baru. Untuk mencari tempat yang tepat dari x, dalam hal ini dapat dilakukan proses cari binar atau *binary search* pada subdaftar a[1], ..., a[i] yang sudah terurut. Maka kita dapatkan suatu model *insertion sort* berikut ini :

```
Procedure  binary-insertion
var i,j,l,r,m,x : integer;
  begin
     for I := 2 to n do
     begin x := a[i];
         l := 1
         r := i-1;
     while l <= r do
     begin
         m := (I+r) div 2
         if x < a[m] then r := m-1 else l := m+1
     end;
     for j := i-1 down to 1 do a[j+i]:= a[j]
     a[1] := x
  end
end
```

Dapat ditunjukkan bahwa algoritma di atas akan menurunkan jumlah perbandingan menjadi O(n log n), tetapi tidak menurunkan jumlah perpindahan data. Biasanya dalam penerapannya jumlah waktu yang dipergunakan oleh algoritma sortir lebih tergantung pada perpindahan data dari pada jumlah operasi perbandingan. Karenanya algoritma di atas tidak terlalu banyak menunjukkan kemajuan. Bahkan bila data awal sudah terurut, jumlah operasi perbandingan yang dilakukan lebih banyak dari pada algoritma sortir penyisipan yang pertama.

6.3 TEKNIK SORTIR PEMILIHAN

Algoritma sortir pemilihan atau *selection sort* bekerja berdasarkan prinsip berikut ini :

1. Pilih data dengan *key* terkecil.
2. Tukarkan data tersebut dengan elemen a[1] .

Kemudian ulangi hal tersebut dengan n-1 data yang ada kecuali a[1]. Lalu dengan n-2 data kecuali a[1] dan a[2]; dan seterusnya. Garis besar algoritmanya adalah sebagai berikut :

```
for i := 1 to n-1 do
begin
pilih elemen yang terkecil dari a[i], ..., a[n] dengan
indeks k tukarkan a[k] dan a[i].
end
```

Jadi pada setiap langkah ke-i, data a[1] sampai dengan a[i], sudah terurut dari kecil ke besar. Dengan demikian, pada langkah selanjutnya hanya diperhatikan a[i+l] sampai dengan a[n] saja.

Contoh 6.3

Urutkan : 44 55 12 42 94 18 7 67

Setelah langkah pertama, data 7 sudah menempati tempatnya dengan benar, yakni :

7 55 12 42 94 18 44 67

Setelah langkah kedua, 7 dan 12 sudah menempati tempatnya dengan benar, yakni : 7 12 55 42 94 18 44 67

Setelah langkah ketiga, data 7,12 dan 18 sudah menempati tempatnya yang benar. Proses ini diteruskan sampai dengan langkah ke i-1, sehingga diperoleh berturut-turut :

7	12	18	42	94	55	44	67
7	12	18	42	94	55	44	67
7	12	18	42	44	55	44	67
7	12	18	42	44	55	94	67
7	12	18	42	44	55	94	67
7	12	18	42	44	55	67	94

Perbedaan utama antara sortir penyisipan dan sortir pemilihan adalah sebagai berikut. Pada sortir penyisipan, pada setiap langkah hanya diperhatikan satu data saja, kemudian untuk mencari tempat data diletakkan, dilihat semua data yang akan menjadi tujuan. Sebaliknya pada *selection sort*, pada tiap langkah dipilih data dari semua barisan data, kemudian diletakkan sebagai satu data baru pada sub daftar tujuan.

Berikut ini diberikan prosedur dari *straight insertion procedure*

```
procedure  straightinsertion;
var i, j, k, x : integer;
begin
  for I := 1 to n-1 do
  begin k := I; x := a[i];
    for j := i+1 to n do
      if a[j] < x then
         begin
           k := j; x := a[j]
           end;
         a[k] := a[i] ; a[i] := x;
  end;
end.
```

Khusus untuk teknik ini, jumlah operasi perbandingan yang dilakukan tidak tergantung dari susunan data awal yang ada. Jadi untuk keadaan terbaik, terburuk maupun rata-rata jumlah operasi perbandingan adalah sama, yakni :

$$C = n(n-1)/2$$

Sedangkan untuk pemindahan, ada tiga kemungkinan :

Kemungkinan terbaik (*best case*)	M = 3(n-1)
Rata-rata (*average case*)	M = 0(n log n)
Kemungkinan terburuk (*worst case*)	M = trunc(n/4) + 3(n-1).

6.4 TEKNIK SORTIR PENUKARAN

Algoritma yang temasuk di dalam kelas ini mempunyai ciri khusus, yakni dengan membandingkan, dan apabila urutan data tidak dipenuhi, diadakan penukaran. Seperti halnya algoritma pada *selection sort* maka pada tiap iterasi, data dengan *key* terkecil dalam sisa Daftar akan bergerak ke bagian kiri dari sisa daftar tersebut. Algoritma yang paling sederhana dan termasuk dalam kelas ini adalah sortir gelembung atau *bubble sort*.

6.4.1 SORTIR GELEMBUNG (BUBBLE SORT)

Sekalipun tidak termasuk jenis sortir yang cepat, sortir ini juga bukan sortir yang paling lambat. Bagaimana caranya sortir gelembung ini melaksanakan penyortirannya? Salah satu versinya akan kita jelaskan sekarang. Untuk itu, kita melihat suatu contoh yang terdiri atas 6 bilangan seperti berikut ini :

Tabel 6.2 Bubble Sort

Sebelum disortir	9	11	12	7	31	3
Setelah disortir	3	7	9	11	12	31

Selanjutnya, demi kemudahan sebutan, letak dari kiri ke kanan, kita namakan saja sebagai letak pertama, letak kedua, sampai letak keenam. Letak itu memiliki lambang (1), (2), sampai (6). Sortir gelembung menyelesaikan penyortirannya secara letak demi letak serta dimulai dengan letak pertama.

Asal dasar dari sortir gelembung ini adalah membandingkan bilangan di antara dua letak. Misalkan saja, kita membandingkan bilangan di antara letak (2) dan letak (5). Dengan asas ini, sortir gelembung membandingkan bilangan di antara berbagai letak serta, bila perlu, memindahkan bilangan di antara letak itu. Berdasarkan asas itu, coba kita lihat kerja sortir gelembung secara langkah demi langkah.

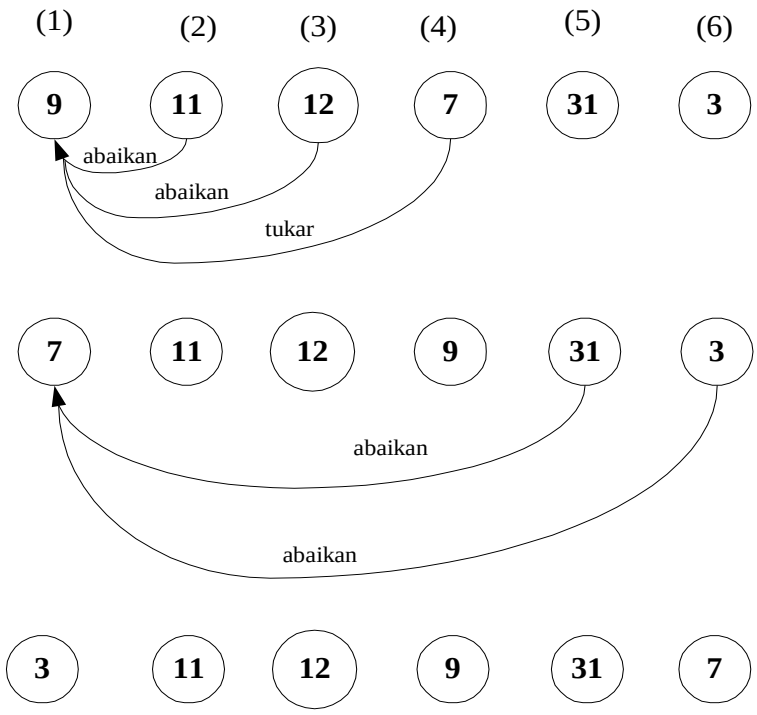
Letak Pertama

Karena sortir gelembung menyelesaikan pcnyortirannya letak demi letak dan dimulai dari letak pertama, maka kita coba mengikutinya dari letak pertama. Gambar 6.6 menunjukkan bagaimana sortir letak pertama itu dilakukan oleh sortir gelembung. Sortir pada letak pertama ini, kita tandai dengan index I = 1.

Pada sortir gelembung langkah pertama ini, letak pertama kita bandingkan dengan letak pertama (indeks J = 1). Tidak terjadi apa-apa. Setelah itu, letak pertama kita bandingkan dengan letak kedua (indeks J=2). Tidak terjadi apa-apa. Selanjutnya, letak pertama kita bandingkan dengan letak ketiga (indeks J=3). Juga tidak terjadi apa-apa. Selanjutnya lagi, letak pertama kita bandingkan dengan letak keempat (indeks J=4). Di sini terjadi pemindahan bilangan. Bilangan 9 di (I) dan bilangan 7 di (4) dipertukarkan. Kini, letak (1) memiliki bilangan 7 dan bukan lagi 9.

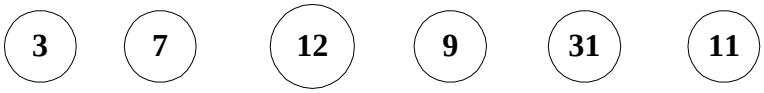
Setelah itu, letak pertama kita bandingkan dengan letak kelima (indeks J=5). Tidak terjadi apa-apa. Pada akhirnya, letak pertama kita bandingkan dengan letak ke- enam (indeks J=6). Di sini, terjadi pemindahan bilangan. Bilangan 7 di (1) dan bilangan 3 di (6) dipertukarkan. Kini, letak(1) memiliki bilangan 3.

Semua langkah ini menimbulkan satu hal. Bilangan terkecil dari kelompok bilangan itu akan berpindah ke letak pertama. Dengan kata lain, kini, letak pertama memiliki bilangan terkecil. Dengan demikian, pada langkah selanjutnya, letak pertama dapat kita tinggalkan.

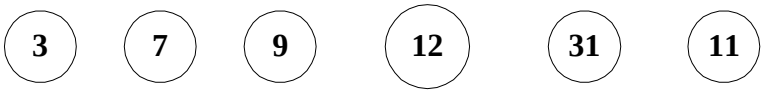


Gambar 6.6. *Penentuan letak pertama pada bubble sort*

Dengan cara yang sama (kini dimulai dari elemen ke dua) yang berisi angka 11, dibandingkan dengan angka-angka di sebelahnya. Bila ada angka di sebelahnya yang lebih kecil, saling tukar tempatnya. Pada langkah kedua, kini susunan datanya menjadi:



Begitu juga untuk menentukan elemen mana yang akan diletakkan di tempat ketiga, caranya sama. Bandingkan elemen ketiga (yang nilainya 12) dengan angka-angka di sebelahnya. Hasilnya :



Susunan elemen pada langkah untuk menentukan letak keempat adalah :

Dan yang terakhir adalah membandingkan elemen kelima dengan keenam, hasilnya adalah:

SORTIR GELEMBUNG SECARA UMUM

Secara umum, kelompok bilangan itu akan memiliki n bilangan. Dengan demikian, kita akan menemukan n-1 kali letak penyortiran. Letak pertama menggunakan indeks I=1, letak kedua menggunakan indeks I=2, dan seterusnya, sampai ke letak ke-(n-1) yang menggunakan indeks I = n-1.

Pada letak pertama, kita menggunakan indeks J = 1, J = 2 sampai ke J = n. Pada letak kedua, kita menggunakan indeks J=2, J=3, sampai ke J=n. Pada letak ketiga, kita menggunakan indeks J=3, J=4 sampai ke J=n. Dan demikian seterusnya. Atau pada umumnya, nilai indeks J bergerak dari J= 1 sampai ke J = n.

Indeks ini adalah penting. Indeks ini menunjukkan letak pada penyortiran itu. Pada penyortiran, setiap kali letak berindeks J dibandingkan dengan letak berindeks I. Berdasarkan pembandingan itulah, ditentukan ada tidaknya pertukaran di antara letak.

Dengan demikian, prosedur dapat kita tulis sebagai :

```
Procedure bubblesort1;
var i, j, x : integer;

be

gin
   for i := 1 to n-1 do
      for j := i  to n do
        if a[j] < a[i] then
        begin
           x := a[i];
           a[i] := a[j];
           a[j] := x;
        end;
end;
```

Bersama itu, kita dapat menyusun program komputer untuk melaksanakan sortir gelembung itu. Dalam bahasa BASIC, program itu tercantum berikut ini :

```
10   REM SORTIR GELEMBUNG
20   REM DISUSUN DALAM BASIC
30   REM MASUKKAN DATA YANG DISORTIR
40   CLS
50   INPUT ''BANYAKNYA DATA: “; N
60   PRINT
70   DIM X(N)
80     FOR K=1 TO N
90        PRINT ''DATA KE '';K;
100      INPUT ''NILAI: “; X(K)
120  NEXT K
130  PRINT
140  REM MULAI SORTIR
150  FOR I= 1 TO N-1
160  FOR J= I+1 TO N
170  IF X(J) >= X(l) THEN 260
180  T = X(J) : X(J) = X(l) : X(I) = T
190  NEXT J
200  NEXT I
210  REM TAMPIL HASIL SORTIR
220  FOR K=1 TO N
230  PRINT ''DATA KE '';K;
240  PRINT “NILAI : “; X(K)
250  NEXT K
```

VERSI LAIN SORTIR GELEMBUNG

Selain algoritma bubblesortl di atas, kita dapat pula melaksanakan sortir gelembung kita dengan algoritma bubblesort2 berikut nanti. Pada algoritma bubblesort2 tersebut, pada setiap iterasi diperiksa dua data yang bersebelahan. Bila urutan tidak dipenuhi, kedua data tersebut saling bertukar tempat. Pada akhir setiap iterasi, data terkecil yang ada pada sisa daftar telah bergeser ke bagian sebelah kiri dari daftar.

```
procedure bubblesort2;
      var i,j,x : integer;
      begin
        for i := 2 to n do
           for j := n down to i do
              if a[j-1] > a[j] then
              begin
                  x := a[j-1];
                  a[j-1] := a[j];
                  a[j] := x;
              end;
      end;
```

Contoh 6.4

Pandang data yang diberikan pada kolom pertama berikut ini yang belum terurut :

Tabel 6.3

data awal	i=2	i=3	i=4	i=5	i=6	i=7	i=8
44	7	7	7	7	7	7	7
55	44	12	12	12	12	12	12
12	55	44	18	18	18	18	18
42	12	55	44	42	42	42	42
94	42	18	55	44	44	44	44
18	94	42	55	55	55	55	55
7	18	94	67	67	67	67	67
67	67	67	94	94	94	94	94

Pada contoh ini terlihat bahwa pada i = 2, maka data 7 sudah benar letaknya. Pada i = 3, maka data 7 dan 12 sudah benar. Demikian seterusnya pada iterasi ke-i, data a[i] sampai dengan i – 1 sudah benar letaknya. Jadi data yang harus diperhatikan hanya data ke-i sampai dengan n. Di sini juga terlihat bagaimana unsur yang terkecil pada suatu iterasi akan timbul ke permukaan (*bubbles up*). Silakan Anda menyusun program apapun yang Anda kuasai untuk menyelesaikan algoritma ini.

6.4.2 PERBAIKAN ALGORITMA SORTIR GELEMBUNG

Dari algoritma bubblesort2, dan contoh di atas jelas dapat dilakukan perbaikan. Hal ini dapat dilihat, karena pada tiga iterasi terakhir terlihat bahwa tidak ada perubahan yang terjadi, karena pada dasarnya seluruh daftar sudah terurut.

Hal lain yang juga penting adalah mencatat di mana perubahan susunan data terjadi pada daftar. Dengan mengingat hal ini, maka data lain sesudah titik pertukaran tersebut tidak perlu diperhatikan lagi, dan sudah pasti terurut.

Perhatikan dua kemungkinan susunan data awal berikut ini. Pada kedua susu-nan tersebut, data sudah hampir terurut, kecuali ada satu data yang salah tempatnya. Pada kasus pertama, angka terkecil berada di ujung kanan, sedangkan pada kasus kedua angka terbesar berada di ujung kiri.

Kasus 1 : 12 18 42 44 53 67 94 7

Kasus 2: 94 7 12 18 42 44 55 67

Bila data di atas diurutkan dengan algoritma bubblesort2, maka untuk kasus 1 hanya diperlukan satu iterasi saja untuk menyelesaikannya, sedangkan kasus 2 memerlukan tujuh iterasi. Hal ini menimbulkan gagasan baru untuk mengubah arah sortir pada setiap iterasi. Di sini timbul apa yang dikenal sebagai *shakersort*. Pada teknik ini selalu diingat indeks di mana perubahan terakhir terjadi. Kemudian iterasi berikutnya akan dimulai dari indeks tersebut, dengan arah yang berlawanan.

```
procedure  shakersort;
var j,k,l,r,x : integer;
begin i := 2; r := n; k := n;
    repeat
       for j := r down to i do
            if a[j-1] > a[j] then begin
            k := a[j-1]; a[j-1] := a[j] a[j] := x
            k := j;
            end;
       l := k+1
       for j := 1 to r do
           if a[j-1] > a[j] then begin
           x := a[j-1];  a[j-1] := a[j]; a[j] := x
           l := j;
           end
           r = k -1;
    until  I > r
end;
```

Contoh 6.5

Pandang contoh lalu yang akan diselesaikan oleh *shakersort:*

Tabel 6.4

l	2	3	3	4	4
r	8	8	7	7	4
	44	7	7	7	7
	55	44	44	12	12
	12	55	12	44	18
	42	12	42	18	42
	94	42	55	42	44
	18	94	18	55	55
	7	18	67	67	67
	67	67	94	94	94

KOMPLEKSITAS ALGORITMA SORTIR GELEMBUNG

Jumlah perbandingan untuk algoritma *bubble sort* adalah sama untuk setiap kemungkinan, yakni n(n-1)/2. Sedangkan jumlah perpindahan data yang diperlukan adalah :

Keadaan terbaik (best case) M = 0

Rata-rata (average case) M = 3n(n-1)/ 4

Keadaan terburuk (worst case) M = 3n(n-1)/4

Untuk *shakersort* jumlah perbandingan data dapat diturunkan pada order yang sama, tetapi jumlah perpindahan data hampir tetap. Jadi sekali lagi teknik sortir ber-dasarkan pertukaran data yang bersebelahan ini kurang menguntungkan.

6.4.3 SORTIR BIASA

Sekarang kita bahas model *exchange sort* yang lain lagi, yang caranya sangat sederhana, dan biasa dilakukan orang awam, yakni yang dikenal sebagai "sortir biasa" atau "*common sort*". Misalkan kita mempunyai n buah elemen yang belum terurut. Dalam sortir ini kita mempunyai suatu indeks (I) yang menyatakan kedudukan elemen (ke-i) dari himpunan elemen, dan satu panji (P) yang menandakan terjadi atau tidaknya pertukaran posisi elemen dalam himpunan itu. Dalam keadaan awal, harga I = 1 dan P = 0. Kemudian kita lakukan langkah sebagai berikut ini :

a. Jika el(i) < el(i+1), maka posisi el(i) dibiarkan tetap. I bertambah 1, menjadi I = 2. Patokan kita sekarang adalah el(i+1). El(i+1) kita bandingkan dengan elemen berikutnya. Proses di atas dilakukan lagi sampai didapat elemen berikutnya yang > dari el(i+1). Pada saat itu dilakukan langkah b.

b. Jika el(i) > el(i+1), maka posisi el(i) dan el(i+1) dipertukarkan. Jika terjadi per-tukaran seperti di atas, P berubah dari 0 menjadi 1 (P = 1). Langkah berikut adalah membandingkan el(i+1) dengan elemen berikutnya. Jika el(i+1) < el(i+2) maka kita lakukan langkah a kembali. Jika el(i+1) > el(i+2) maka posisi el(i+1) dan el(i+2) dipertukarkan.

c. Setelah mencapai elemen terakhir, jika P = 0 maka proses sortir selesai. Jika P=1 maka proses sortir harus diulangi kembali, terhadap urutan yang baru tadi.

Demikianlah seterusnya kita lakukan langkah a dan b sampai dengan elemen ke n. Jika sampai dengan elemen ke-n harga P masih sama dengan satu (P=1), maka sortir diulangi

kembali sampai didapatkan P = 0. Pada saat pengulangan sortir, harga P dan I dibuat menjadi 0 dan 1 kembali. (P=0 dan 1=1)

Contoh 6.6

Pandang 6 buah elemen yang belum terurut sebagai berikut :

(1)	(2)	(4)	(3)	(5)	(6)
7	11	12	3	31	9

Pada contoh di atas kita mempunyai 6 buah elemen yang tidak berurutan yakni 7, 11, 12, 3, 31, 9. Pada keadaan awal harga panji P = 0 dan kita melihat elemen pertama I = 1.

Mula-mula kita bandingkan el(1) dengan el(2). Ternyata el(1) < el(2), jadi el(1) tetap dan kita beralih ke elemen kedua, I = 2. Kita lihat el(2) < el(3) maka el(2) tetap juga. Kemudian kita beralih ke elemen ketiga, I = 3, ternyata el(3) > el(4) maka terjadi pertukaran posisi el(3) dan el(4) dan P = 1.

Sekarang urutan elemen menjadi 7, 11, 3, 12, 31, 9 dengan P =1. Kita bandingkan sekarang elemen keempat I = 4 yakni el(4) dibandingkan dengan el(5), ternyata el(4) < el(5), jadi el(4) tetap. Kemudiaan kita ambil elemen kelima yakni el(5) dibandingkan dengan el(6), ternyata el(5) > el(6), maka kedua elemen tersebut dipertukarkan posisinya. Sortir telah sampai pada elemen keenam, hasilnya adalah :

7, 11, 3, 12, 9, 31 dengan P = 1

Anda dapat melihat bahwa elemen di atas berjumlah terurut. Sekarang kita ulangi kembali proses sortir, dengan harga P = 0 dan harga I = 1 kembali. Dengan P = 0, kita bandingkan el(1) dengan el(2) dan ternyata el(1) < el(2), jadi el(1) tetap. Kita beralih ke el(2), I = 2, kita bandingkan el(2) tersebut dengan el(3), ternyata el(2) > el(3), maka posisi keduanya kita pertukarkan harga P menjadi sama-dengan 1.

Sekarang yang menjadi patokan adalah el(3), I = 3. Kita bandingkan elemen ketiga dengan elemen keempat, dan ternyata el(3) < el(4) maka el(3) tetap dan harga I menjadi 4. Elemen keempat kita bandingkan dengan elemen kelima, terlihat bahwa el(4) > el(5), maka posisi keduanya kita pertukarkan. Harga P tetap sama dengan 1. Kemudian, yang menjadi patokan sekarang adalah elemen kelima yang akan dibandingkan dengan elemen keenam. El(5) < el(6), maka posisi elemen-ke 5 dan elemen ke 6 tetap. Urutan elemen sekarang menjadi sebagai berikut :

7, 3, 11, 9, 12, 31 dengan P = 1

Terlihat kembali di sini bahwa kedudukan elemen masih belum berurut dan harga P masih sama dengan 1. Kita lakukan sortir kembali dengan harga P berubah menjadi 0 kembali dan harga I menjadi 1 lagi. Kita bandingkan elemen kesatu dari urutan elemen tersebut di atas dengan elemen kedua dan terlihat el(1) > el(2), terjadi pertukaran tempat. Patokan berikutnya adalah elemen kedua, I = 2, yang akan kita bandingkan dengan elemen ketiga, I = 3. Ternyata el(2) < el(3) maka posisi kedua elemen tetap. Selanjutnya, elemen ketiga, I = 3, yang akan kita bandingkan dengan elemen keempat. Kita lihat bahwa el(3) > el(4) maka terjadi pertukaran posisi dari kedua elemen itu.

Sekarang kita beralih pada elemen keempat, I = 4. Kita bandingkan elemen keempat tersebut dengan elemen kelima. Terlihat el(4) < el(5) maka posisi kedua elemen itu tetap. Sebagai patokan sekarang adalah I = 5, yakni kita bandingkan elemen kelima dengan elemen keenam. Ternyata el(5) < el(6), jadi posisi kedua elemen tersebut tetap. Urutan dari elemen-elemen sekarang menjadi sebagai berikut :

3 7 9 11 12 31 dan P = 1

Kita lihat bahwa urutan elemen telah terurut dari kecil ke besar, tetapi harga P masih sama dengan 1. Jadi, kita lakukan sortir elemen di atas dengan harga P = 0 dan harga I = 1. Mula-mula kita bandingkan el(1) dengan el(2). Ternyata el(1) < el(2), maka posisi kedua elemen tersebut tetap. Selanjutnya, kita bandingkan el(2) dengan el(3). Karena el(2) < el(3), maka posisi keduanya tetap.

Sekarang kita bandingkan el(3) dengan el(4), dan ternyata el(3) < el(4), jadi posisi kedua elemen tersebut tetap. Pada langkah berikutnya kita membandingkan el(4) dengan el(5), dan ternyata terlihat bahwa el(4) < el(5); posisi kedua elemen tersebut tetap. Terakhir kita bandingkan el(5) dengan el(6). Kita lihat el(5) < el(6) maka posisi kedua elemen itupun tetap. Sekarang kita lihat urutan dari elemen setelah disortir. Urutan mereka menjadi sebagai berikut :

3 7 9 11 12 31 dan P = 0

Karena P = 0, maka proses sortir selesai.

6.5 SHELLSORT : MEMPERCEPAT SORTIR PENYISIPAN

Kita akan menampilkan suatu metode yang merupakan perluasan dari teknik sortir penyisipan (*insertion sort*) biasa, yakni "*Shell Sort*." Dalam hal ini data dibagi dalam beberapa kelompok yang berbeda. Pada setiap kelompok dilakukan sortir penyisipan. Kemudian banyak kelompok diciutkan, sehingga banyak data dalam masing-masing kelompok bertambah. Lalu diberlakukan lagi algoritma sortir penyisipan.

Hal ini terus dilanjutkan, sampai banyaknya kelompok yang ada hanya tinggal satu, dan mengandung seluruh data. Hal tersebut menunjukkan bahwa teknik ini lebih cepat dibandingkan dengan teknik sortir penyisipan biasa. Hal ini disebabkan karena pada tahap awal dilakukan beberapa sortir penyisipan dengan jumlah data yang sedikit karena jumlah kelompok masih cukup banyak.

Pada akhir proses harus dilakukan sortir penyisipan pada seluruh data, tetapi sortir secara partial yang dilakukan sebelumnya sudah menyebabkan data terturut sebagian-sebagian hingga pada akhirnya jumlah operasi pada sortir penyisipan yang dilakukan tidak terlalu besar.

Jadi pada teknik ini sortir pada kelompok yang kecil akan sangat mempengaruhi kecepatan sortir pada kelompok berikutnya, yang mempunyai data sedikit lebih banyak.

Contoh 6.7

Diketahui 8 buah data yang akan dibagi menjadi 4 kelompok, masing-masing dengan 2 data, lalu dibagi ke dalam 2 kelompok, masing-masing dengan 4 data, dan terakhir menjadi 1 kelompok dengan 8 data. Untuk setiap tingkat dan kelompok dilakukan sortir penyisipan.

Keadaan awal : 44 55 12 42 94 18 7 67

Kita kelompokkan menjadi 4 kelompok, lalu dilakukan sortir penyisipan partial :

Kelompok 1 : 44 94 diurutkan menjadi 44 94

Kelompok 2 : 55 18 diurutkan menjadi 18 55

Kelompok 3 : 12 7 diurutkan menjadi 7 12

Kelompok 4 : 42 67 diurutkan menjadi 42 67

Hasilnya sekarang : 44 18 7 42 94 55 12 67

Sekarang Kelompok 1 dan 3 kita gabungkan menjadi Kelompok 1-3, serta Kelompok 2 dan 4 kita gabungkan menjadi Kelompok-2-4. Kemudian dilakukan sortir penyisipan parsial.

Kelompok 1-3 : 44 94 7 12 diurutkan menjadi 7 12 44 94

Kelompok 2-4 : 18 55 42 67 diurutkan menjadi 18 42 55 67

7 18 12 42 44 55 94 67

Sekarang kedua kelompok di atas kita gabungkan menjadi satu kelompok dan kita lakukan sortir penyisipan. Diperoleh hasil akhir :

7 12 18 42 44 55 67 94

Secara umum pengelompokan berulang-ulang tersebut tidak perlu tergantung pada bilangan yang merupakan pangkat dari 2, seperti 2; 4, 8 dan seterusnya. Sembarang bilangan untuk pengelompokan dapat dilakukan, tetapi harus diakhiri dengan kelompok tunggal, yakni dalam hal ini kelompok terakhir hanya terdiri dari satu kelompok dengan seluruh elemen data. Bila digunakan sampai t kali pengelompokan yang masing-masing besarnya $h_1, h_2, \ldots, h_t$, maka harus dipenuhi : $h_t = 1$ dan $h_{i+1} < h_i$

Dalam penerapannya setiap kelompok diurutkan dengan teknik sortir penyisipan dengan menggunakan kondisi khusus untuk berhenti mencari tempat, data harus dimasukkan. Seperti pada penerapan untuk sortir penyisipan terdahulu, untuk memudahkan di bagian depan ditambahkan data a[0]. Untuk hal ini a didefinisikan sebagai :

a : array[-h_1…n] of integer

di sini h_1 adalah pertambahan kelompok pertama yang terbesar.

Pemilihan nilai t dan h yang baik sulit ditentukan dan analisis jumlah operasi perbandingan dan pemindahan kata juga sulit dilakukan.

Berikut ini adalah prosedur dari *shellsort.*

```
Procedure shellsort;
Const t := 4;
var i, j, k, s, x : integer;
   h : array [1..t] of integer;
   m : 1 .. t;
begin
   h[1] := 9; h[2] := 2; h[3] := 3; h[4] :=1;
   for m = 1 to t do begin
      X := a[i] : j := i-k;
         if s := 0 then
            S := -k;
            S := s+1;
            A[s] := x;
            While x < a[j] do begin
              A[j+k] := a[j];
              J := j=k;
            End;
    End;
End;
```

6.6 *QUICKSORT* : SUATU APLIKASI *STACK*

Kali ini kita bicarakan metode *quicksort*. *Quicksort* adalah sebuah algoritma sortir dari model atau tipe *devide-and-conquer*, sama seperti metode *shellsort* yang baru lalu. Sebelum kita berikan algoritma *quicksort* secara lebih formal, baiklah kita tengok lebih dahulu sebuah ilustrasi mengenai bekerjanya algoritma tersebut.

Misalkan, A adalah himpunan atau daftar berisi 12 bilangan yang pada mulanya mempunyai urutan :

44 33 11 55 77 90 40 60 99 22 88 66

Kita hendak mengurutkan data di atas secara naik atau *ascending*, yakni mengurutkan dari kecil ke besar. Tahap reduksi dalam algoritma *quicksort* dimulai pada posisi awal dan akhir dari daftar bilangan tersebut. Dalam contoh ilustrasi di atas, bilangan pertama adalah 44.

Dimulai dari angka terakhir 66, amati daftar dari kanan ke kiri, kemudian bandingkan setiap bilangan dengan 44 dan berhenti pada bilangan pertama yang lebih kecil dari 44. Bilangan tersebut adalah 22. Kemudian pertukarkan posisi 44 dan 22 tersebut sehingga urutan menjadi :

22 33 11 55 77 90 40 60 99 44 88 66

Perhatikan bahwa data 88 dan 66 pada sebelah kanan 44, adalah lebih besar dari 44. Kemudian dimulai dari 22, lakukan hal yang sama seperti di atas, tetapi dengan arah berlawanan (dari kiri ke kanan), bandingkan setiap bilangan dengan 44 sampai kita menemukan bilangan pertama yang lebih besar dari 44. Bilangan tersebut adalah 55. Kemudian pertukarkan posisi 44 dan 55, sehingga urutan menjadi :

22 33 11 44 77 90 40 60 99 55 88 66

Perhatikan bahwa angka 22, 33 dan 11 berada di sebelah kiri 44, dan ketiganya lebih kecil dari 44. Kemudian mulai dari 55, lakukan hal yang sama seperti di atas dengan arah dari kanan ke kiri, sampai kita menemukan angka pertama yang nilainya lebih kecil dari 44, yakni 40. Pertukarkan 44 dan 40 sehingga urutan menjadi :

22 33 11 40 77 90 44 60 99 55 88 66

Perhatikan angka-angka di sebelah kanan 44. Seluruhnya bernilai lebih besar dari 44. Mulai dari 40, lakukan pengamatan dari kiri ke kanan. Angka pertama yang lebih besar dari 44 adalah 77, kemudian pertukarkan kedua angka tersebut. Diperoleh:

22 33 11 40 44 90 77 60 99 55 88 66

Perhatikan, ternyata angka-angka di kiri 44 lebih kecil dari 44. Mulai dari 77, amati daftar dari kanan ke kiri, dicari angka yang lebih kecil dari 44. Ternyata angka tersebut tidak ada. Hal ini berarti bahwa seluruh angka telah diperbandingkan dengan 44. Semua angka yang lebih kecil dari 44 sekarang membentuk daftar sendiri, demikian pula angka-angka yang lebih besar dari 44, seperti tampak di bawah ini:

22 33 11 40 44 90 77 60 99 55 88 66

Daftar 1 Daftar 2

Jadi angka 44 pada posisi reakhir ini merupakan tempat yang tepat. Tahap reduksi seperti di atas dapat diulang terhadap masing-masing daftar yang mengandung 2 atau lebih elemen. Bila kita hanya mampu melakukan proses reduksi tersebut satu daftar dalam satu waktu, kita harus dapat mengawasi beberapa daftar untuk proses berikutnya.

Hal ini diselesaikan dengan menggunakan 2 *stack*, yang kita sebut "*lowe*r" dan "*upper*.". Elemen pertama dan terakhir dalam masing-masing daftar disebut nilai batas (*boundary values*). Elemen-elemen tersebut dimasukkan ke dalam *stack lower* dan *stack upper*.

Tahap reduksi hanya dapat diterapkan pada sebuah daftar, bila nilai-nilai batas tersebut telah dipindahkan dari *stack*.

Contoh berikut menggambarkan cara *stack lower* dan *stack upper* digunakan. Proses dimulai dengan memasukkan nilai-nilai batas 1 dan 12 dari A ke dalam *stack*, sehingga dihasilkan :

Lower = 1

Upper = 12

Agar langkah atau tahap reduksi digunakan, algoritma mula-mula memindahkan nilai paling atas (*top value*) yakni 1 dan 12 dari *stack*, sehingga :

Lower = kosong

Upper = kosong

dan kemudian gunakan langkah reduksi untuk mencocokkan daftar A[1], A[2],…, A[12]. Langkah reduksi yang dilakukan di atas akhirnya menempatkan elemen pertama, 44 dalam A[5]. Dengan demikian, algoritma memasukkan nilai-nilai batas 1 dan 4 dari daftar

pertama dan nilai-nilai batas 6 dan 12 dari daftar kedua ke dalam *stack* sehingga menghasilkan :

Lower = 1, 6

Upper = 4, 12

Agar langkah reduksi digunakan lagi, algoritma memindahkan angka paling atas 6 dan 12 dari *stack*, sehingga :

Lower = 1

Upper = 4

dan kemudian gunakan langkah reduksi untuk mencocokkan daftar A[6], A[7],…,A[12]. Langkah reduksi mengubah daftar tersebut ada di dalam Tabel berikut ini :

Tabel 6.5

A[6]	A[7]	A[8]	A[9]	A[10]	A[11]	A[12]
90	77	60	99	55	88	66
66	77	60	99	55	88	90
66	77	60	90	55	88	99
66	77	60	88	55	90	99
		Daftar I			Daftar II	

Di sini ternyata bahwa daftar kedua hanya mempunyai 1 elemen. Dengan demikian algoritma hanya memasukkan nilai-nilai batas 6 dan 10 dari daftar pertama ke dalam *stack*, sehingga dihasilkan :

Lower = 1, 6

Upper = 4, 10

dan seterusnya. Algoritma berakhir ketika stack tidak mengandung daftar untuk diproses oleh langkah reduksi.

ALGORITMA QUICKSORT

Algoritma *quicksort* dibagi dalam 2 bagian. Bagian pertama memberikan sebuah prosedur yang disebut QUICK, yang akan menjalankan langkah reduksi dari Algoritma di atas. Bagian kedua menggunakan prosedur QUICK untuk mengurutkan seluruh daftar.

Kondisi LOC <> RIGHT dalam langkah 2(a) dan kondisi LEFT <> LOC dalam langkah 3(a) dapat diabaikan. *Lower* dan *upper* adalah *stack* untuk menempatkan nilai-nilai batas dari daftar (sebagaimana biasa, kita menggunakan NULL = 0).

Procedure QUICK(A,N,BEG,END,LOC)

"Di sini A adalah barisan dengan N elemen. Parameter BEG dan END memuat nilai batas dari daftar A yang digunakan pada prosedur ini. LOC mengawasi posisi dari elemen pertama A[BEG] dari daftar selama prosedur. Variabel LEFT dan RIGHT akan memuat nilai batas dari daftar elemen yang belum diamati".

Prosedur

1. LEFT:=BEG, RIGHT:=END dan LOC:=BEG
2. Amati dari kanan ke kiri
   a) pengulangan bila A[LOC] <= A[RIGHT] dan LOC <> RIGHT; RIGHT := RIGHT-1
   b) jika LOC := RIGHT, maka return;
   c) jika A[LOC] > A[RIGHT], maka:
      i) pertukarkan A[LOC] dan A[RIGHT]; TEMP:= A[LOC]; A[LOC]: = A[RIGHT], A[RIGHT] := TEMP;
      ii) LOC := RIGHT;
      iii) kembali ke langkah 3

   Amati dari kiri ke kanan :
   a) Ulangi bila A[LEFT] <= A[LOC] dan LEFT <> LOC; LEFT:= LEFT + 1, akhir dari pengulangan
   b) jika LOC := LEFT maka return
   c). jika A[LEFT] > A[LOC], maka:
      i) pertukarkan A[LEFT] dan A[LOC]
         TEMP := A[LOC)
         A[LOC] := left
      ii) kembali ke langkah 2

Algoritma *Quicksort*

1. TOP := NULL
2. Masukkan nilai yang hingga dari A ke dalam stack. Bila A memiliki 2 elemen atau lebih :
   Jika N > 1, maka
   TOP := TOP +1

LOWER[1] := 1
UPPER[1] := N

3. Ulangi langkah 4 sampai langkah 7 ketika TOP <> NULL
4. Pindahkan / hapus daftar dari stack
   BEG := LOWER[TOP]
   END := UPPER[TOP]
   TOP := TOP – 1
5. Call QUICK(A,N,BEG,END,LOC) [sebuah prosedur]
6. Masukkan daftar sebelah kiri ke dalam stack bila mempunyai 2 elemen atau lebih
   TOP := TOP + 1; LOWER[TOP] := BEG
   UPPER[TOP] : LOC – 1
7. Masukkan daftar sebelah kanan ke dalam stack bila mempunyai 2 elemen atau lebih.
   Jika LOC+1 < END, maka :
   TOP := TOP + 1
   LOWER[TOP] := LOC + 1
   UPPER[TOP] := END
   ********** akhir perulangan langkah 3 **********

KOMPLEKSITAS ALGORITMA *QUICKSORT*

Lamanya penyortiran biasanya diukur oleh fungsi f(n) yang menunjukkan banyaknya perbandingan yang dibutuhkan algoritma untuk menangani n elemen. Algoritma *quicksort* mempunyai banyak variasi. Pada umumnya, Algoritma mempunyai kasus terburuk adalah dari n^2 . Tetapi, lama untuk kasus rata-rata adalah dari n log n.

Kasus terburuk terjadi apabila daftar sudah terurut. Kemudian elemen pertama akan membutuhkan n perbandingan untuk menandakan bahwa elemen tersebut tetap pada posisi pertama. Selanjutnya daftar pertama akan kosong, tetapi Daftar kedua akan mempunyai n–1 elemen. Dengan demikian elemen kedua akan membutuhkan n-1 perbandingan untuk menandakan bahwa elemen kedua tersebut tetap pada posisi kedua dan seterusnya. Maka keseluruhannya adalah :

$$F(n) = n + \frac{(n-1)}{2} + \ldots + 2 + 1 = n\,\frac{(n+1)}{2} = n^2 + O(n^2)$$

Perbandingan

Kompleksitas f(n) = O(n log n) dari kasus rata-rata berasal dari fakta itu, pada rata-rata setiap langkah reduksi dari algoritma menghasilkan 2 daftar. Dengan demikian :

1) Reduksi pada daftar mula-mula menempatkan 1 elemen dan menghasilkan 2 Daftar.
2) Reduksi pada 2 daftar menempatkan 2 elemen dan menghasilkan 4 daftar.
3) Reduksi pada 4 daftar menempatkan 4 elemen dan menghasilkan 8 daftar.
4) Reduksi pada 8 Daftar menempatkan 8 elemen dan menghasilkan 16 daftar dan seterusnya.

Diperhatikan bahwa tahap reduksi pada tingkat k menempatkan lokasi 2^{k-1} elemen, karena itu akan terdapat kira-kira log n tingkat tahap reduksi. Selanjutnya setiap tingkat menggunakan perbandingan n yang terbanyak, jadi f(n) = O(n log n). Pada kenyataannya baik analisis matematika maupun berbagai bukti empiris, keduanya menunjukkan bahwa :

f(n) = 1,4 n log n

adalah banyaknya perbandingan untuk algoritma *quicksort*.

6.7 SORTIR TOPOLOGIK

Pada bagian ini kita akan membicarakan salah satu jenis sortir yang dikenal dengan nama "sortir topologik" atau "*topological sorting*". Sortir ini kita jumpai misalnya dalam proses kompilasi bahasa ADA. Bahasa ADA adalah bahasa yang pertama kali dibuat untuk keperluan Departemen Pertahanan Amerika Serikat.

Sebelum melangkah kepada sortir topologik, kita perlu memahami sedikit tentang pengertian *graph* dan *digraph*, karena pengertian tersebut sangat penting dalam pembahasan sortir topologik.

6.7.1 GRAPH DAN DIGRAPH

Kata *graph* di dalam matematika mempunyai bermacam-macam arti. Biasanya kita mengenal kata *graph* atau grafik suatu fungsi, ataupun relasi. Untuk kali ini kita gunakan kata *graph* dalam arti yang lain. Suatu graph mengandung 2 himpunan :

(1) Himpunan V yang elemennya disebut simpul (atau *vertex* atau *point* atau *node* atau titik).

(2) Himpunan E yang merupakan pasangan tak urut dari simpul. Anggotanya disebut ruas (*edge*, rusuk atau sisi).

Graph seperti dimaksud di atas, kita tulis sebagai G(E,V).

Simpul u dan v disebut berdampingan bila terdapat ruas (u,v). *Graph* dapat pula disajikan secara geometrik. Untuk menyatakan *graph* secara geometrik, simpul disajikan sebagai sebuah titik, sedangkan ruas disajikan sebagai sebuah garis yang menghubungkan 2 simpul.

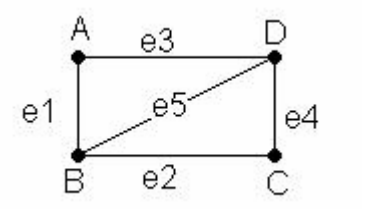
Sebagai contoh, Gambar 6.7 berikut menyatakan *graph* G(E,V) dengan :

(1) V mengandung 4 simpul, yakni simpul A, B, C, D.

(2) E mengandung 5 ruas, yakni :

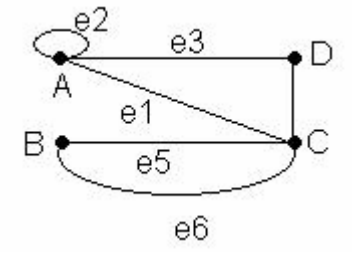
e1 = (A, B) e2 = (B, C) e3 = (A, D)

e4 = (C, D) e5 = (B, D)



Gambar 6.7. *Contoh sebuah graph*

Banyaknya simpul disebut order, sedangkan banyaknya ruas disebut *size* dari *graph*. Gambar 6.8 merupakan suatu *graph* yang lebih umum, disebut *multigraph*. Di sini, ruas e2 kedua titik ujungnya adalah satu simpul yang sama, yakni simpul A. Ruas semacam ini disebut gelung atau *self-loop*. Sedangkan ruas e5 dan e6 mempu-nyai titik ujung yang sama, yakni simpul-simpul B dan C. Kedua ruas ini disebut ruas berganda atau ruas sejajar.

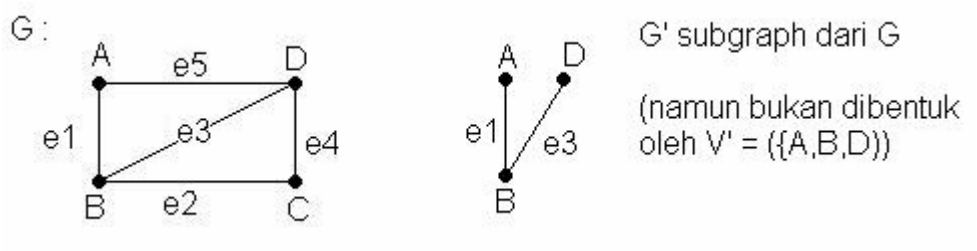


Gambar 6.8. *Contoh sebuah multigraph*

Suatu *graph* yang tak mengandung ruas sejajar ataupun *self-loop*, sering disebut sebagai *graph* sederhana atau *simple graph*. Suatu *graph* G'(E',V') disebut *subgraph* dari G(E,V), bila E' himpunan bagian dari E dan V' himpunan bagian dari V. Jika E'

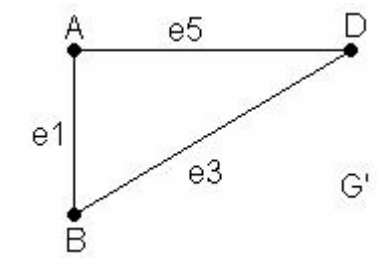
mengandung semua ruas dari E yang titik ujungnya di V', maka G' disebut *subgraph* yang direntang oleh V' (*spanning subgraph*).

Sebagai contoh :



Gambar 6.9. *Contoh graph dan subgraph*

G' *subgraph* yang dibentuk oleh V' = (A,B,D)

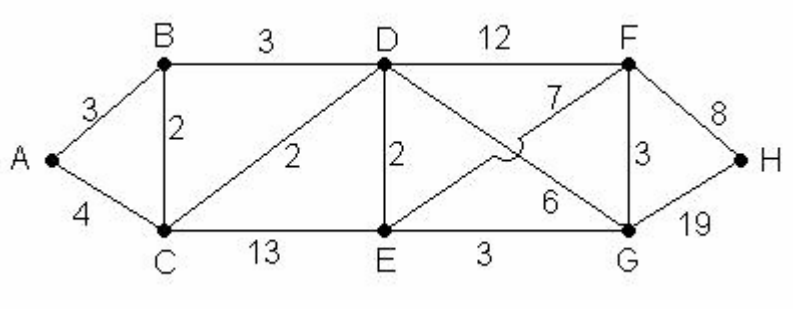


Gambar 6.10. *Contoh sub-graph*

Suatu *multigraph* disebut hingga apabila ia mempunyai sejumlah hingga simpul dan sejumlah hingga ruas. Jelas suatu *graph* dengan sejumlah hingga simpul akan mempunyai sejumlah hingga ruas.

GRAPH BERLABEL

Graph G disebut *graph* berlabel jika ruas dan atau simpulnya dikaitkan dengan suatu besaran tertentu. Khususnya, jika setiap ruas e dari G dikaitkan dengan suatu bilangan non negatif d(e), maka d(e) disebut bobot atau panjang dari ruas e. Sebagai contoh, Gambar 6.11 berikut ini menyajikan hubungan antarkota. Di sini, simpul menyatakan kota dan label d(e) menyatakan jarak antara dua kota.



Gambar 6.11 *Contoh graph berlabel*

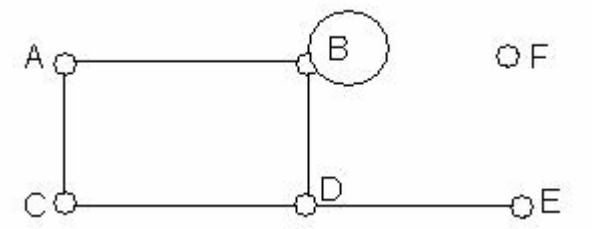
DERAJAT *GRAPH*

Derajat simpul V, ditulis d(v) adalah banyaknya ruas yang menghubungi v. Karena setiap ruas dihitung dua kali ketika menentukan derajat suatu *graph*, maka:

> "Jumlah derajat semua simpul suatu *graph* (disebut derajat) = dua kali banyaknya ruas *graph* (*size* atau ukuran *graph*)".

Suatu simpul disebut genap/ ganjil tergantung apakah derajat simpul tersebut genap/ ganjil. Kalau terdapat *self-loop*, maka *self-loop* dihitung 2 kali pada derajat simpul.

Contoh 6.8



Gambar 6.12 *Contoh untuk perhitungan derajat graph*

Di sini, banyak ruas = 7, sedangkan derajat masing-masing simpul adalah :

d(A) = 2
d(B = 5
d(C) = 3
d(D) = 3
d(E) = 1
d(F) = 0

Catatan : E disebut simpul bergantung/ akhir, yakni simpul yang berderajat satu. Sedangkan F disebut simpul terpencil, yakni simpul berderajat nol.

KETERHUBUNGAN.

Walk atau perjalanan dalam graph G adalah barisan simpul dan ruas berganti-ganti :

$$v_1, c_1. v_2, c_2, \ldots, c_{n-1}, v_n$$

di sini ruas e menghubungkan simpul v_i dan v_{i+1}. Banyaknya ruas disebut panjang *walk*. *Walk* dapat ditulis lebih singkat dengan hanya menulis deretan ruas.

$$c_1, c_2, \ldots, c_{n-1}$$

atau deretan simpul :

$$v_1, v_2, \ldots, v_{n-1}, v_n,$$

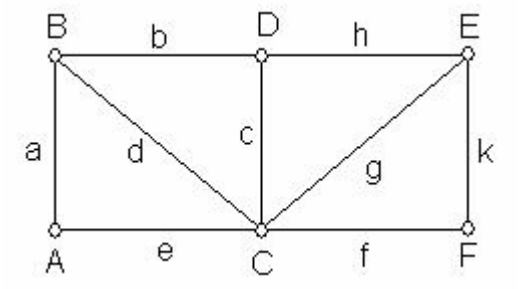
v_1 disebut simpul awal
v_n disebut simpul akhir

Walk disebut tertutup bila $V_1 = V_n$, dalam hal lain *walk* disebut terbuka menghubungi V_1 dan V_n.

Trail adalah *walk* dengan semua ruas dalam barisan adalah berbeda. *Path* atau jalur adalah *walk* yang semua simpul dalam barisan adalah berbeda. Jadi *path* pasti *trail*.

Dengan kata lain: suatu *path* adalah suatu *trail* terbuka dengan derajat setiap simpulnya 2, kecuali simpul awal V_1 dan simpul akhir V_n yang berderajat = 1.

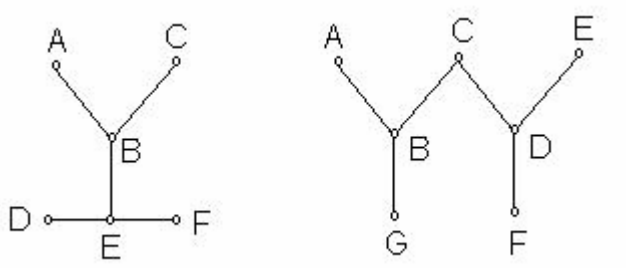
Cycle atau sirkuit adalah suatu *trail* tertutup dengan derajat setiap simpulnya = 2. *Cycle* dengan panjang k disebut k-*cycle*. Demikian pula, jalur dengan panjang k disebut k-jalur.



Gambar 6.13. *Contoh sebuah graph*

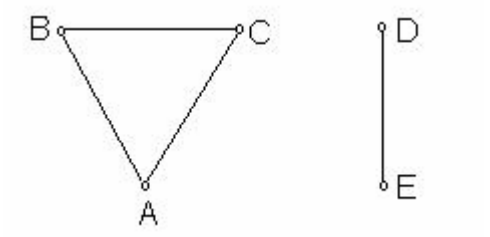
Barisan ruas a, b, c, d, b, h, g, f adalah *walk* bukan *trail* (ruas b dua kali muncul). Barisan simpul A, B, E, F bukan *walk* (tak ada ruas menghubungkan simpul B ke F). Barisan simpul A, B, C, D, E, C, F adalah *trail* bukan jalur karena C dua kali muncul. Barisan ruas a, d, g, k adalah jalur menghubungkan A dengan F dan a, b, h, g, e adalah *cycle*.

Graph yang tak mengandung *cycle* disebut *acylic*. Contoh dari *graph acyclic* adalah pohon atau *tree*. Contoh dari pohon :



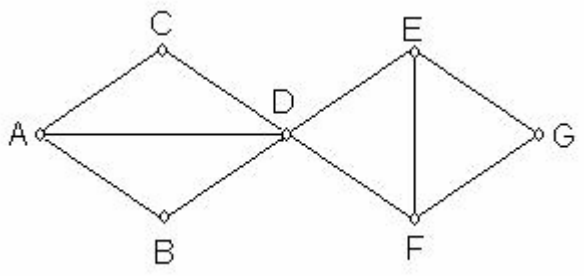
Gambar 6.14. *Contoh pohon*

Suatu g*raph* G disebut terhubung jika untuk setiap 2 simpul dari *graph* terdapat jalur yang menghubungkan 2 simpul tersebut. *Subgraph* terhubung suatu *graph* disebut komponen dari G bila s*ubgraph* tersebut tidak terkandung dalam *subgraph* terhubung lain yang lebih besar.



Gambar 6.15. *Contoh graph tidak terhubung*

Graph G pada Gambar 6.15 adalah tidak terhubung, karena simpul D dan E tidak terhubung dengan simpul A, B dan atau C. Jarak antara 2 simpul dalam *graph* G adalah panjang jalur terpendek antara kedua simpul tersebut. Diameter suatu *graph* terhubung G adalah maksimum jarak antara simpul-simpul G.



Gambar 6.16. *Contoh graph untuk menghitung diameternya*

Jarak maksimum dalam *graph* G adalah 3 (yaitu antara A-G atau B-G ataupun C-G). Jadi diameter = 3.

Kalau *order* dari G = n, ukuran dari G = e, dan banyaknya komponen = k, maka didefinisikan :

$$Rank\ (G) = n - k$$

$$Nullity\ (G) = e - (n - k)$$

Kedua *graph* pada gambar yang lalu, masing-masing mempunyai *rank* = 6 – 3 = 3, *nullity* = 4 – (6 – 3) = 1, dan *rank* = 7 – 1 = 6, *nullity* = 10 – (8 – 1) = 4.

MATRIKS PENYAJIAN *GRAPH*

Pandang bahwa G adalah *graph* dengan N simpul dan M ruas. Untuk mempermudah komputasi, *graph* dapat disajikan dalam bentuk matriks, disebut "Matriks Ruas", yang berukuran (2 x M) atau (M x 2) yang menyatakan ruas dari *graph*. Kalau *graph* mengandung simpul terpencil, matriks ini tak dapat menunjukannya, kecuali kalau jumlah simpul disebutkan. Misalnya kita menyajikan *graph* G dalam matriks ruas B ukuran (M x 2), maka setiap baris matriks menyatakan ruas. Misalnya baris (4, 7) menyatakan ada ruas menghubungkan simpul 4 dan 7. Matriks *adjacency* dari *graph* G tanpa ruas sejajar adalah matriks A berukuran (N x N), yang bersifat :

$$a_{ij} = \begin{cases} 1 \text{ bila ada ruas } (V_i, V_j) \\ 0 \text{ dalam hal lain} \end{cases}$$

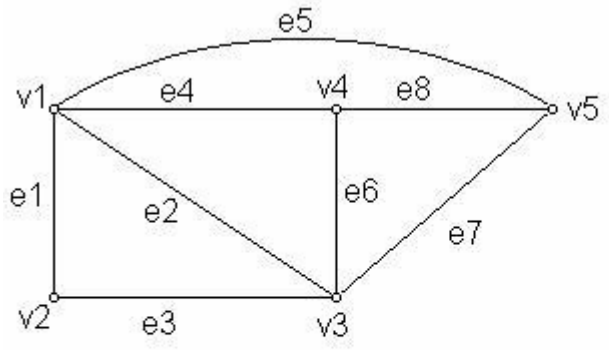
.

Matriks *adjacency* merupakan matriks simetri. Untuk *graph* dengan ruas sejajar, matriks *adjacency* didefinisikan sebagai berikut :

$$a_{ij} = \begin{cases} P \text{ bila ada p buah ruas menghubungkan } (V_i, V_j)\ (P > 0) \\ 0 \text{ dalam hal lain} \end{cases}$$

Matriks *incidence* dari *graph* G, tanpa *self-loop* didefinisikan sebagai matriks M berukuran (N x M) sebagai berikut :

$$m_{ij} = \begin{cases} 1 \text{ bila ruas } c_j \text{ berujung di simpul } v_i \\ 0 \text{ dalam hal lain} \end{cases}$$



Matriks ruas

1	2
1	3
1	4
1	5
2	3
3	4
3	5
4	5

atau

1	1	1	1	2	3	3	4
2	3	4	5	3	4	5	5

Atau secara pasangan :

{(1,2), (1,3), (1,4), (1,5), (2,3), (3,4), (3,5), (4,5)}

Matriks *adjacency* :

	v1	v2	v3	v4	v5
v1	0	1	1	1	1
v2	1	0	1	0	0
v3	1	1	0	1	1
v4	1	0	1	0	1
v5	1	0	1	1	0

Matriks *incidence* :

	e1	e2	e3	e4	e5	e6	e7	e8
v1	1	1	0	1	1	0	0	0
v2	1	0	1	0	0	0	0	0
v3	0	1	1	0	0	1	1	0
v4	0	0	0	1	0	1	0	1
v5	0	0	0	0	0	1	0	1

GRAPH BERARAH (*DIGRAPH*)

Di dalam situasi yang dinamis, seperti contohnya pada komputer digital, ataupun pada sistem aliran (*flow system*), konsep *graph* berarah lebih sering digunakan dibandingkan dengan konsep *graph* tak berarah. Suatu *graph* berarah (*directed graph* disingkat *digraph* (*D*)) terdiri atas 2 himpunan:.

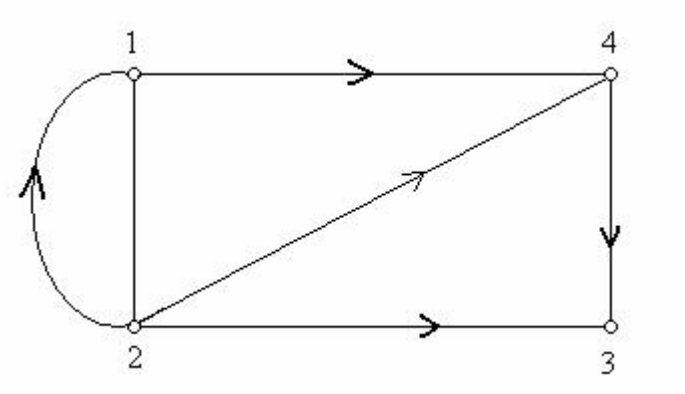
(1) Himpunan V, anggotanya disebut simpul,

(2) Himpunan A, merupakan himpunan pasangan terurut, yang disebut ruas berarah atau arkus, *graph* berarah di atas ini, kita tulis sebagai D(V,A).

Kita dapat menggambar suatu graph berarah pada suatu bidang rata. Simpul, anggota V, digambarkan sebagai titik (atau lingkaran kecil). Sedangkan arkus a = (u,v), digambarkan sebagai garis dilengkapi dengan tanda panah mengarah dari simpul u ke simpul v. Simpul u disebut titik pangkal dan simpul v disebut titik terminal dari arkus tersebut. Sebagai contoh, Gambar 6.17 di bawah ini adalah sebuah *graph* berarah D (V,A), dengan :

1. V mengandung 4 simpul, yakni 1, 2, 3 dan 4
2. A mengandung 7 arkus, yakni (1,4), (2,1), (2,1), (2,1), (2,3), (4,3) dan (2,2)

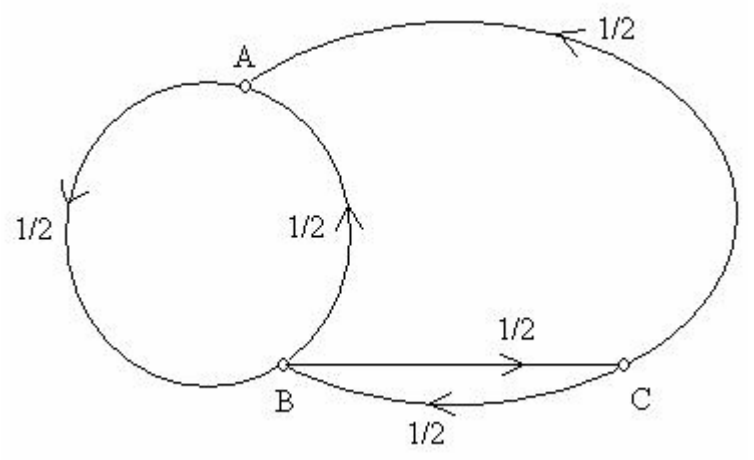
Arkus (2,2) disebut gelung (*self-loop*), sedangkan arkus (2,1) muncul lebih dari satu kali, disebut arkus sejajar atau arkus berganda.



Gambar 6.17 *Contoh sebuah graph berarah*

Apabila arkus dan atau simpul suatu *graph* berarah menyatakan suatu bobot, maka graph berarah tersebut dinamakan suatu jaringan atau *network*. *Graph* semacam itu biasanya digunakan menggambarkan situasi dinamis. Sebagai contoh, 3 orang anak A, B dan C melempar bola di antara mereka sedemikian sehingga A selalu melempar kepada B, namun B serta C melempar sekehendaknya.

Gambar berikut ini, menunjukkan situasi dinamis di atas. Di sini arkus diberi bobot yang menyatakan probabilitas, sebagai contoh, A melempar bola kepada B dengan probabilitas = 1, B melempar bola tersebut kepada C dengan probabilitas 1/2, melempar kepada A juga dengan probabilitas ½, dan sebagainya.



Gambar 6.18 *Bobot probabilitas*

Misalkan D(V,A) suatu *graph* Berarah. D disebut hingga (*finite*), jika baik V maupun A merupakan himpunan hingga. Bila V' himpunan bagian dari V serta A' himpunan bagian dari A, dengan titik ujung anggota A' terletak di dalam V', maka dikatakan bahwa D'(V',A') bahwa D'(V',A') adalah *graph* bagian (*subgraph*) dari D(V,A). Kalau A' mengandung semua arkus anggota A yang titik ujungnya anggota V', maka dikatakan bahwa D'(V',A') adalah *graph* bagian yang dibentuk atau direntang oleh V'

DIGRAPH DAN RELASI

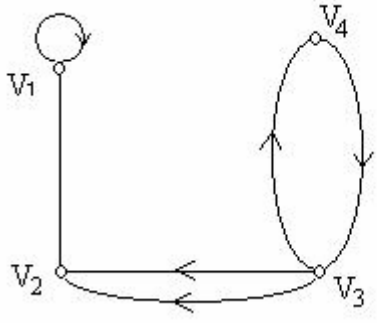
Pandang D(V,A) suatu *graph* berarah tanpa arkus sejajar. Maka A adalah himpunan bagian dari V x V (produk *cartesius* himpunan), jadi merupakan relasi pada V. Sebaliknya bila R adalah relasi pada suatu himpunan V, maka D(V,R) merupakan *graph* berarah tanpa arkus sejajar. Maka konsep relasi serta konsep *graph* berarah tanpa arkus sejajar adalah satu dan sama.

Misalkan sekarang D(V,A) suatu *graph* berarah dengan simpul V_1, V_2, ..., V_m. Matriks M berukuran (m x m) merupakan matriks *adjacency* dari D, dengan mendefinisikan sebagai berikut :

$M = (M_{ij})$, dengan M_{ij} banyaknya arkus yang mulai di V_i dan berakhir di V_j.

Bila D tidak mengandung arkus berganda, maka elemen dari M adalah 0 dan 1. Kalau *graph* mengandung arkus berganda, elemen M merupakan bilangan bulat non negatif. Jadi suatu matriks berukuran (m x m) yang elemennya bilangan bulat non negatif menyatakan secara tunggal suatu *graph* berarah dengan m simpul.

Sebagai contoh, *graph* pada Gambar 6-19 berikut mempunyai matriks M



$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Gambar 6.19 *Penggambaran relasi dalam graph dengan matriks*

6.7.2 PERSYARATAN SORTIR TOPOLOGIK

Proses di dalam sortir topologik sedikit berbeda dengan proses di dalam sortir yang lain. Pada proses sortir topologik, data yang akan disortir disajikan dalam suatu *graph* berarah atau *directed* graph (yang disingkat *digraph*). Setelah disajikan ke dalam suatu *digraph* (misalkan *digraph* tersebut kita namakan *digraph* G, maka *digraph* G tersebut kemudian harus disajikan ke dalam G*, yakni transitif-refleksif *closure* dari G. Digraph G* harus menyajikan suatu relasi *partial order*.

Suatu relasi R pada himpunan S dinamakan *partial order* pada S, jika R memenuhi 3 sifat. Ketiga sifat tersebut adalah :

(1) Refleksif, yakni untuk setiap a anggota S berlaku aRa

(2) Antisimetris, yakni untuk a, b anggota S; jika aRb, bRa, maka a = b

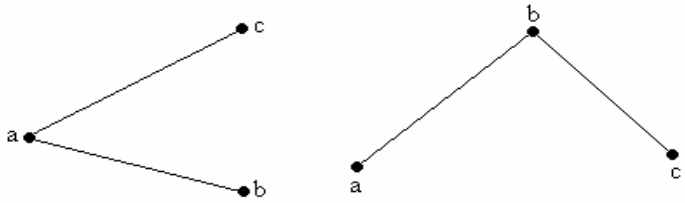
(3) Transitif, yakni untuk a, b, c anggota S; jika aRb, bRc, maka aRc.

(di sini yang dimaksud dengan xRy adalah x berelasi R dengan y).

Sortir topologik adalah proses pembentukan atau pengurutan simpul suatu *digraph*. Ruas dari *digraph* tersebut mempunyai peranan penting untuk menentukan suatu urutan tertentu dari simpul yang ada. Urutan tersebut dinamakan *topological enumeration*. Jadi pada hakikatnya *topological enumeration* merupakan hasil dari proses sortir topologik.

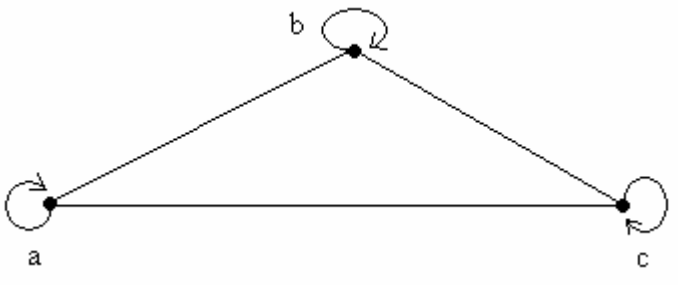
Untuk memudahkan kita dalam mengikuti algoritma sortir topologik, terlebih dahulu kita lihat contoh *digraph* yang dapat disajikan ke dalam suatu *digraph* yang *partial order*. Pada Gambar 6.20 terlihat bahwa *digraph* D_1 mempunyai sirkuit atau *cycle*, yakni sirkuit dari simpul a terdapat ruas ke Simpul b dan dari simpul b terdapat ruas ke simpul a.

Ini berarti bahwa *digraph* tersebut tidak antisimetris. Dalam menyajikan *digraph* ke *digraph* yang *partial order*, sifat transitif dan refleksif sudah terpenuhi dalam transitif-refleksif *closure*nya, tetapi sifat antisimetris harus sudah terpenuhi dalam *digraph* D_1 Jadi *digraph* D_1 tersebut tidak memenuhi syarat untuk algoritma kita.



Gambar 6.20 *Graph tanpa sirkuit*

Digraph D_2 pada Gambar 6.20 tidak mengandung sirkuit, sehingga transitif-refleksif *closure*nya, yakni D_2* merupakan *partial order*. Sifat refleksif disajikan dalam bentuk gelung (*self-loop*); sedang sifat transitif kita penuhi dengan cara yakni bila dari simpul a terdapat ruas ke simpul b, dan dari simpul b terdapat ruas ke simpul c, kita tambahkan ruas dari simpul a ke simpul c (kalau belum ada). Kita perhatikan Gambar 6.21.



Gambar 6.21 *Transitif-reflektif closure dari D_2*

6.7.3 ALGORITMA SORTIR TOPOLOGIK

Misalkan himpunan elemen yang akan disortir adalah U, dan R adalah relasi dari elemen tersebut, yang dinotasikan sebagai u_1Ru_2. Untuk menjalankan algo-ritma sortir topologik kita harus memeriksa apakah R pada U dapat disajikan dalam R* yang partial order. Dalam hal ini, bila *digraph* (U,R) mengandung satu *cycle*, maka tidak ada hasil sortir (berupa *topological enumeration*) yang lengkap. Proses tidak terselesaikan.

Jika R* adalah *partial order*, maka kita bentuk urutan terdiri dari <u1, ..., U_n>, sedemikian sehingga U = (u_1, ... , u_n.) dan setiap (u_j, u_k) ada pada R* dengan j < k (j mendahului k).

Algoritma sortir topologik mengandung input dan *output*. Input adalah sebuah *digraph* G = (V,E), dengan banyak simpul n. *Output*nya adalah *topological enumeration* S_n = <S_1, ... , S_n > dari V dengan memperhatikan setiap ruas E, asalkan E* adalah *partial order* pada V.

METODE

(1) Tentukan $U_0 = V$, $S_0 = <>$ dan $T_0(v) = (u – (u,v)$ anggota E dan u tidak sama dengan v).

(2) Ulangi untuk i = 1, 2, ..., n

(a) Pilih si dari U_{i-1}, yang memenuhi bahwa $T_{i-1}(S_i)$ HAMPA

Bila tidak ada, maka proses akan berhenti karena E* tidak antisimetris

(b) Tentukan $U_1 = U_{i-1}–(s_i)$, $S_i = S_{i-1}<>S_i>$ dan $T_1(v) = T_{i-1}(v)-(s_i)$, untuk semua v anggota V.

(3) Jika selesai, maka outputnya adalah S_n.

Keterangan Algoritma

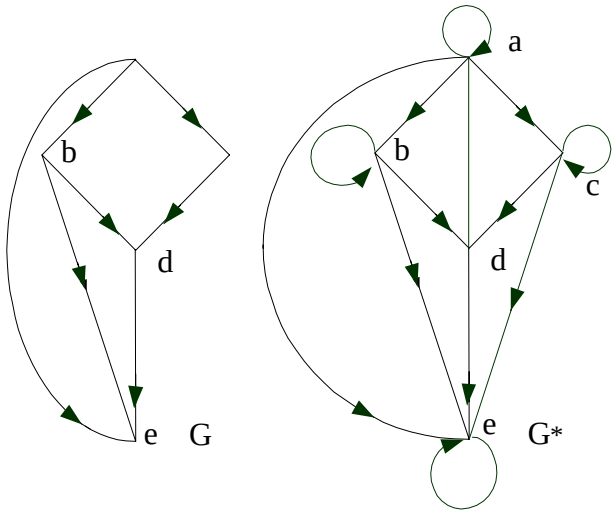
Kita tetapkan mula-mula U_0 sebagai himpunan simpul yang akan disortir. S_0 diberi harga awal HAMPA < >, dan $T_0(v)$ adalah himpunan simpul yang mempu-nyai ruas masuk ke simpul yang akan disortir (anggota U_0). Simpul u dimisalkan sebagai simpul yang menuju ke v. Di sini u tidak boleh sama dengan v.

Kemudian untuk i = 1, 2, ..., , n kita ulangi pilihan (yang akan disortir) yang diambil dari himpunan U_{ij}, yang bersifat bahwa T_i-1(s_i) HAMPA. Dengan kata lain, kita mengambil sebuah simpul dari *digraph* itu yang *in-degree*nya = 0.

Himpunan simpul pada saat itu adalah sama dengan himpunan simpul sebe-lumnya dikurangi simpul yang akan disortir (simpul yang dipilih). Himpunan simpul sudah disortir, adalah sama dengan himpunan simpul sudah disortir sebelumnya ditambah dengan simpul yang dipilih. Himpunan simpul yang menuju ke V adalah sama dengan himpunan simpul yang menuju ke V sebelumnya dikurang simpul yang dipilih. Jika prosesnya berjalan sampai selesai, maka *output*nya adalah Sn.

Contoh 6.9

Untuk menerapkan algoritma di atas, kita ambil sebuah contoh *digraph* yang terdiri dari simpul a, b, c, d, e dengan ruas (a,b); (a,e); (a,c); (b,d); (b,e), (c,d); (d,c), seperti pada Gambar 6.22.



Gambar 6.22. *Digraph G dan transitif-reflektif closurenya*

Syarat pertama terpenuhi, yakni bahwa data berupa *digraph*. Setelah itu kita lihat apakah *digraph* itu mengandung sirkuit berarah, kalau tidak mengandung, maka berarti bahwa *digraph* tersebut antisimetris, dan dapat disajikan ke dalam *digraph* yang *partial order*. Setelah kita periksa, ternyata *digraph* di atas antisimetris. Dengan demikian G* merupakan *digraph* yang *partial order*. Kalau persyaratan telah terpenuhi, maka algoritma di atas dapat dijalankan.

Kita tuliskan simpul yang akan disortir itu dalam himpunan U_0, dalam hal ini himpunan yakni (a,b,c,d,e). Setelah itu kita cari $T_0(v)$nya satu persatu, yakni :

$T_0(a)$ = HAMPA

$T_0(b)$ = (a)

$T_0(c)$ = (a)

$T_0(d)$ = (b,c)

$T_0(c)$ = (a,b,d)

Kemudian kita ambil S_i dari U_0 yang $T_0(S_i)$nya HAMPA. Dalam hal ini kita ambil simpul a sehingga U_1 = (a, b, c, d, e) - (a) = (b,c,d,e) dan S_1 = <a>. Sekarang $T_i(a)$ = HAMPA, $T_i(b)$ = HAMPA, $T_i(c)$ = HAMPA, $T_i(d)$ = (b,c), $T_i(e)$ = (b,d).

Untuk i yang kedua, kita ambil misalnya b, kita cari lagi U_2, S_2, dan T_2(v)nya. Demikian seterusnya sampai proses berakhir. Urutan proses dan hasil sortir dapat kita lihat pada tabel berikut ini :

Tabel 6.6

i	Ui	Si	Ti(a)	Ti(b)	Ti(c)	Ti(d)	Ti(e)
0	(a,b,c,d,e)	<a>	HAMPA	(a)	(a)	(b,c)	(a,b,d)
1	(b,c,d,e)	<a>	HAMPA	HAMPA	HAMPA	(b,c)	(b,d)
2	(c,d,e)	<a,b>	HAMPA	HAMPA	HAMPA	(c)	(d)
3	(d,e)	<a,b,c>	HAMPA	HAMPA	HAMPA	HAMPA	(d)
4	(e)	<a,b,c,d>	HAMPA	HAMPA	HAMPA	HAMPA	HAMPA
5	HAMPA	<a,b,c,d,e>	HAMPA	HAMPA	HAMPA	HAMPA	HAMPA

Keterangan : U_i = Himpunan simpul yang belum disortir

S_i = Himpunan simpul yang sudah disortir

T_i<v> = Himpunan simpul yang mempunyai ruas menuju v.

L A T I H A N 6

1. Apa bedanya sortir internal dan sortir eksternal, dan jelaskan cara kerja *merge sort* (sortir gabung)

2. Jelaskan perbedaan cara kerja *natural merge* dan *balanced merge*.

3. Sebutkan tiga teknik utama dalam melakukan sortir.

4. Mana dari ketiga teknik tersebut yang paling cepat melaksanakan sortir secara *ascending* jika data yang diberikan terurut secara *descending* ?

5. Teknik sortir yang mana yang ketika proses pensortiran dilaksanakan mencari elemen terkecil terlebih dulu ?

6. Bagaimana cara kerja dari *common sort* ?

7. Teknik sortir apa yang dilakukan dengan mengaplikasikan fungsi *stack* ?

8. Apa maksud dari istilah-istilah : (a) *size*, (b) *multigraph*, (c) *self-loop*, (d) *simple graph* dan (e) derajat dari suatu *graph* ?

9. Apa maksud dari istilah-istilah : (a) *walk* , (b) *trail*, (c) *cycle*, (d) *acylic* dan (e) diameter dari suatu *graph* ?

10. Apa maksud dari istilah-istilah : (a) *nullity*, (b) *rank*, (c) matriks *adjacency*, (d) matriks *incidence* dan (e) *digraph* dari suatu *graph* ?

11. Teknik sortir apa yang memanfaatkan fungsi *graph* ?

12. Buat suatu program (Pascal, FORTRAN atau BASIC) untuk mensortir secara *ascending* seratus angka yang diberikan secara acak.