Desain Test Case

- Definisi Test Case
- White Box Testing
- Black Box Testing

I. Definisi Test Case

- *Test case* merupakan suatu tes yang dilakukan berdasarkan pada suatu inisialisasi, masukan, kondisi ataupun hasil yang telah ditentukan sebelumnya.

- Adapun kegunaan dari *test case* ini, adalah sebagai berikut:
  - Untuk melakukan testing kesesuaian suatu komponen terhadap spesifikasi – *Black Box Testing*.
  - Untuk melakukan testing kesesuaian suatu komponen terhadap disain – *White Box Testing*.

II. White Box Testing

- adalah suatu metode disain *test case* yang menggunakan struktur kendali dari disain prosedural.

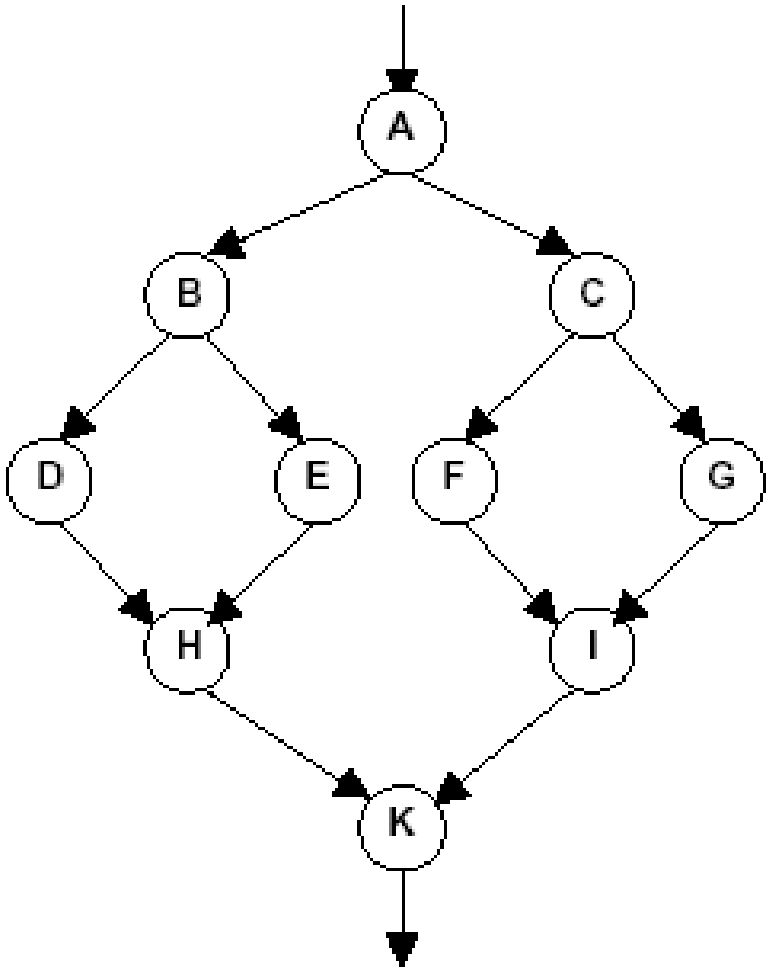
- Metode disain *test case* ini dapat menjamin:
  - Semua jalur (*path*) yang independen / terpisah dapat dites setidaknya sekali tes.
  - Semua logika keputusan dapat dites dengan jalur yang salah dan atau jalur yang benar.
  - Semua *loop* dapat dites terhadap batasannya dan ikatan operasionalnya.
  - Semua struktur internal data dapat dites untuk memastikan validitasnya.

Cakupan pernyataan, cabang dan jalur

- Cakupan pernyataan, cabang dan jalur adalah suatu teknik *white box testing* yang menggunakan alur logika dari program untuk membuat *test cases*.
- Yang dimaksud dengan alur logika adalah cara dimana suatu bagian dari program tertentu dieksekusi saat menjalankan program.
- Alur logika suatu program dapat direpresentasikan dengan *flow graph*.

Contoh *flow graph* dari suatu kode program

```
if A then
        if B then
                D
        else
                E
        end if;
        H
else
        if C then
                F
        else
                G
        end if;
        I
end if
```



Komponen Flow Graph

- ***Nodes*** (titik), mewakili pernyataan (atau sub program) yang akan ditinjau saat eksekusi program.
- ***Edges*** (anak panah), mewakili jalur alur logika program untuk menghubungkan satu pernyataan (atau sub program) dengan yang lainnya.
- ***Branch nodes*** (titik cabang), titik-titik yang mempunyai lebih dari satu anak panah keluaran.
- ***Branch edges*** (anak panah cabang), anak panah yang keluar dari suatu cabang
- ***Paths*** (jalur), jalur yang mungkin untuk bergerak dari satu titik ke lainnya sejalan dengan keberadaan arah anak panah.

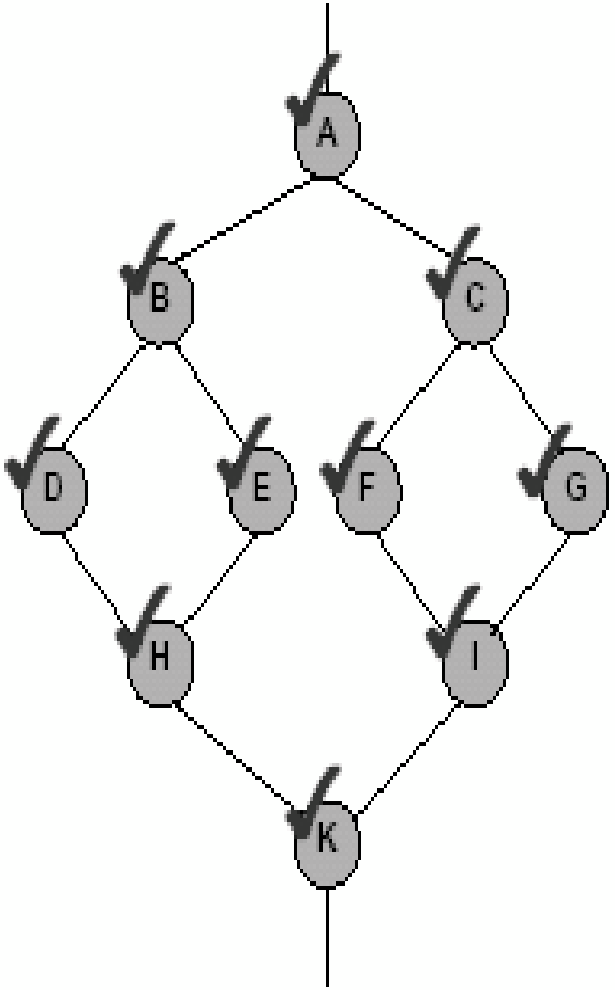
- Eksekusi suatu *test case* menyebabkan program untuk mengeksekusi pernyataan-pernyaan tertentu, yang berkaitan dengan jalur tertentu, sebagaimana tergambar pada *flow graph*.

- Cakupan cabang, pernyataan dan jalur dibentuk dari eksekusi jalur program yang berkaitan dengan peninjauan titik, anak panah, dan jalur dalam *flow graph*.

Cakupan pernyataan

- Cakupan pernyataan ditentukan dengan menilai proporsi dari pernyataan-pernyataan yang ditinjau oleh sekumpulan *test cases* yang ditentukan.
- Cakupan pernyataan 100 % adalah bila tiap pernyataan pada program ditinjau setidaknya minimal sekali tes.
- Cakupan pernyataan berkaitan dengan tinjauan terhadap titik (*node*) pada *flow graph*.
- Cakupan 100 % terjadi bilamana semua titik dikunjungi oleh jalur-jalur yang dilalui oleh *test cases*.

Contoh Cakupan Pernyataan

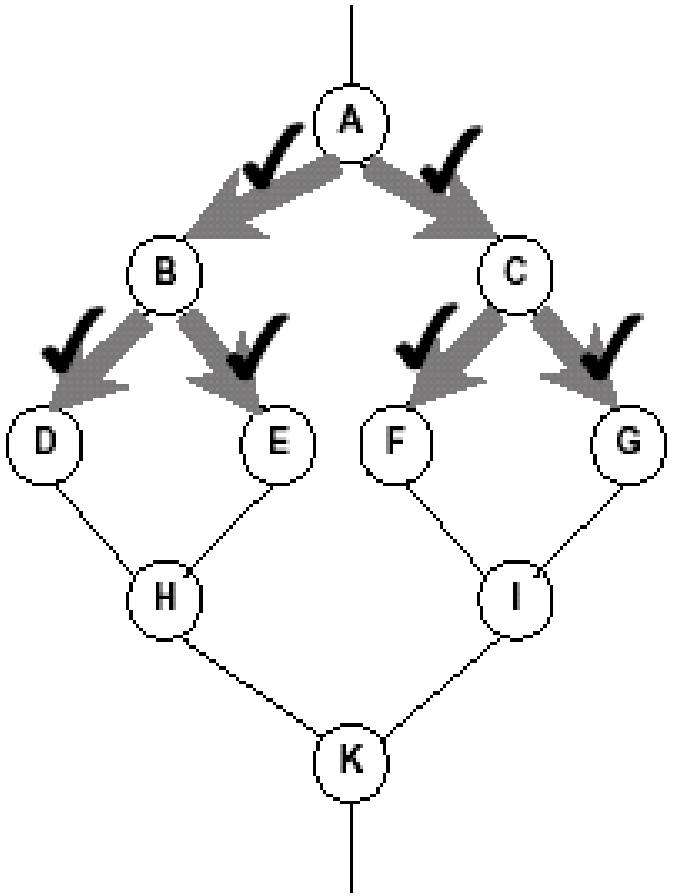


- Pada contoh gambar *flow graph* di samping terdapat 10 titik.
- Misal suatu jalur eksekusi program melewati titik-titik A, B, D, H, K.
- Berarti ada 5 titik dari 10 titik yang dikunjungi, maka cakupan pernyataan sebesar 50 %.

Cakupan cabang

- Cakupan cabang ditentukan dengan menilai proporsi dari cabang keputusan yang diuji oleh sekumpulan *test cases* yang telah ditentukan.
- Cakupan cabang 100 % adalah bilamana tiap cabang keputusan pada program ditinjau setidaknya minimal sekali tes.
- Cakupan cabang berkaitan dengan peninjauan anak panah cabang (*branch* edges) dari *flow graph*.
- Cakupan 100 % adalah bilamana semua anak panah cabang ditinjau oleh jalur-jalur yang dilalui oleh *test cases*.

Contoh cakupan cabang

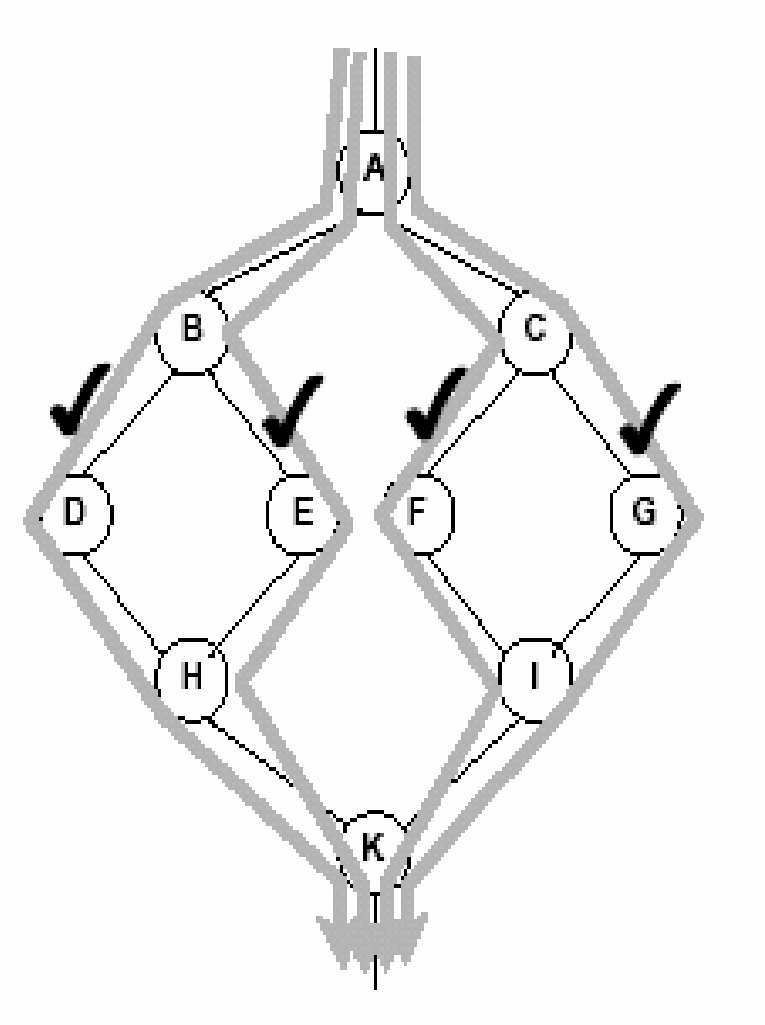


- Berdasarkan pada contoh gambar flow graph di atas, terdapat 6 anak panah cabang.
- Misal suatu jalur eksekusi program melawati titik-titik A, B, D, H, K, maka jalur tersebut meninjau 2 dari 6 anak panah cabang yang ada, jadi cakupannya sebesar 33 %.

Cakupan jalur

- Cakupan jalur ditentukan dengan menilai proporsi eksekusi jalur program yang diuji oleh sekumpulan *test cases* yang telah ditentukan.
- Cakupan jalur 100 % adalah bilamana tiap jalur pada program dikunjungi setidaknya minimal sekali tes.
- Cakupan jalur berkaitan dengan peninjauan jalur sepanjang *flow graph*.
- Cakupan 100 % adalah bilamana semua jalur dilalui oleh *test cases*.

Contoh cakupan jalur

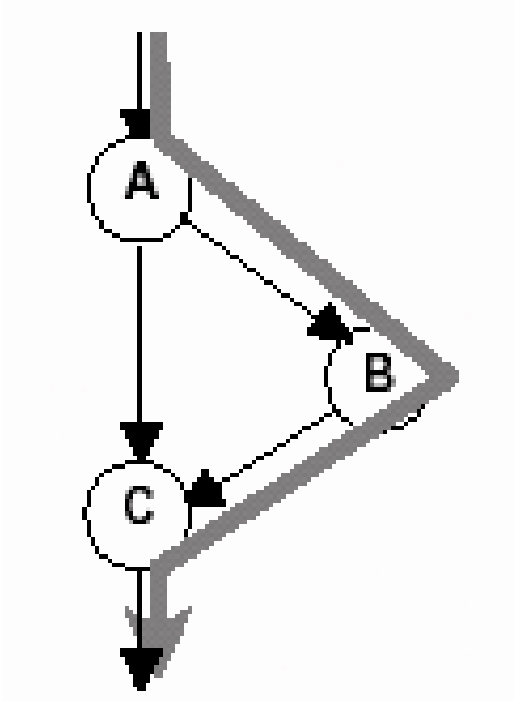


- *flow graph* di samping, terdapat 4 jalur.
- Bila suatu eksekusi jalur pada program melalui titik-titik A, B, D, H, K, maka eksekusi tersebut meninjau 1 dari 4 jalur yang ada, jadi cakupannya sebesar 25 %.

Perbedaan antara cakupan pernyataan, cabang dan jalur

- Pencapaian cakupan pernyataan 100 % dapat terjadi tanpa harus membuat cakupan cabang menjadi 100 % juga. Contoh program:

```
If A then B
C
```

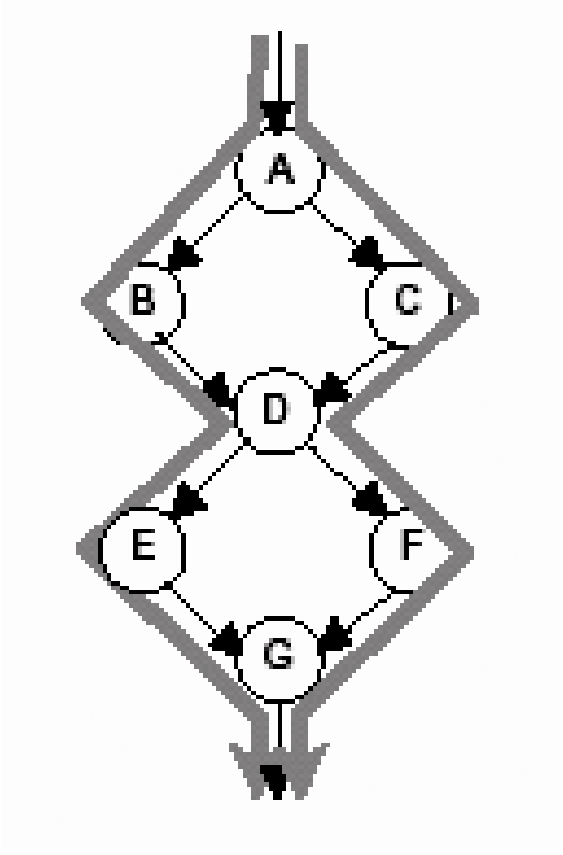


- Dapat pula membuat cakupan cabang 100 %, dengan meninjau seluruh anak panah cabang tanpa harus meninjau semua jalur yang ada (cakupan jalur 100 %). Contoh program:

```
If A then B else C

If D then E else F
```

Hanya dibutuhkan 2 jalur untuk mengunjungi semua anak panah cabang, dari 4 jalur yang ada pada *flow graph*.



Kesimpulan

- Jadi bila cakupan jalur sebesar 100 %, maka secara otomatis cakupan cabang sebesar 100% pula.
- Demikian pula bila cakupan cabang sebesar 100 %, maka secara otomatis cakupan pernyataan sebesar 100 %.

Disain cakupan tes

Untuk mendisain cakupan dari tes, perlu diketahui tahap-tahap sebagai berikut:

- Menganalisa *source code* untuk membuat *flow graph*.
- Mengidentifikasi jalur tes untuk mencapai pemenuhan tes berdasarkan pada *flow graph*.
- Mengevaluasi kondisi tes yang akan dicapai dalam tiap tes.
- Memberikan nilai masukan dan keluaran berdasarkan pada kondisi.

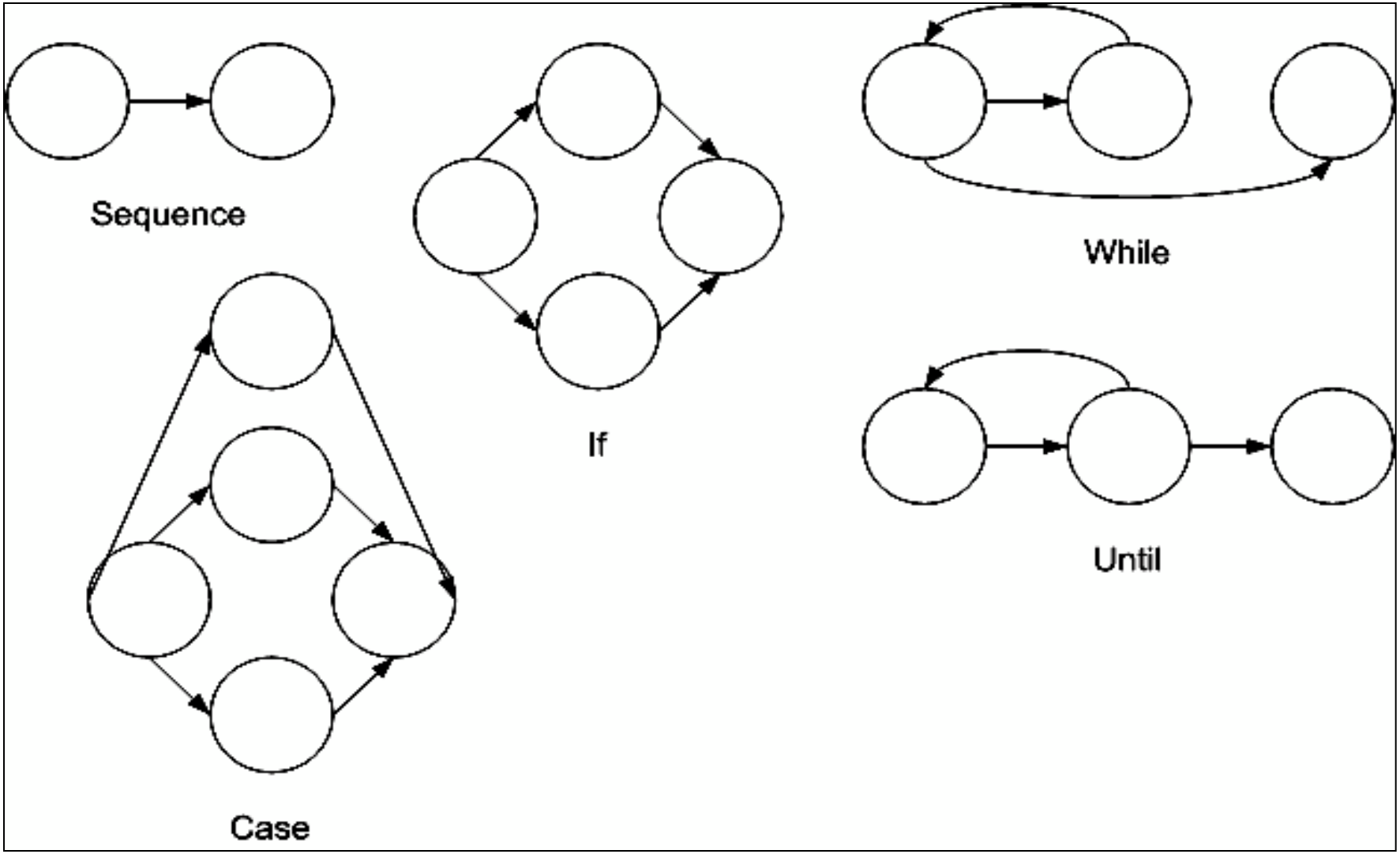
Basis Path Testing

- Merupakan teknik *white box testing*.
- Metode identifikasi yang berdasarkan pada jalur, struktur atau koneksi yang ada dari suatu sistem ini biasa disebut juga sebagai *branch testing*.
- *Basis path* hadir dalam 2 bentuk, yaitu:
  - ***Zero Path***: Jalur penghubung yang tidak penting atau jalur pintas yang ada pada suatu sistem.
  - ***One Path***: Jalur penghubung yang penting atau berupa proses pada suatu sistem.

Konsep utama *basis path*:

- Tiap *basis path* harus diidentifikasi, tidak boleh ada yang terabaikan (setidaknya dites 1 kali).
- Kombinasi dan permutasi dari suatu *basis path* tidak perlu dites.

Notasi flow graph



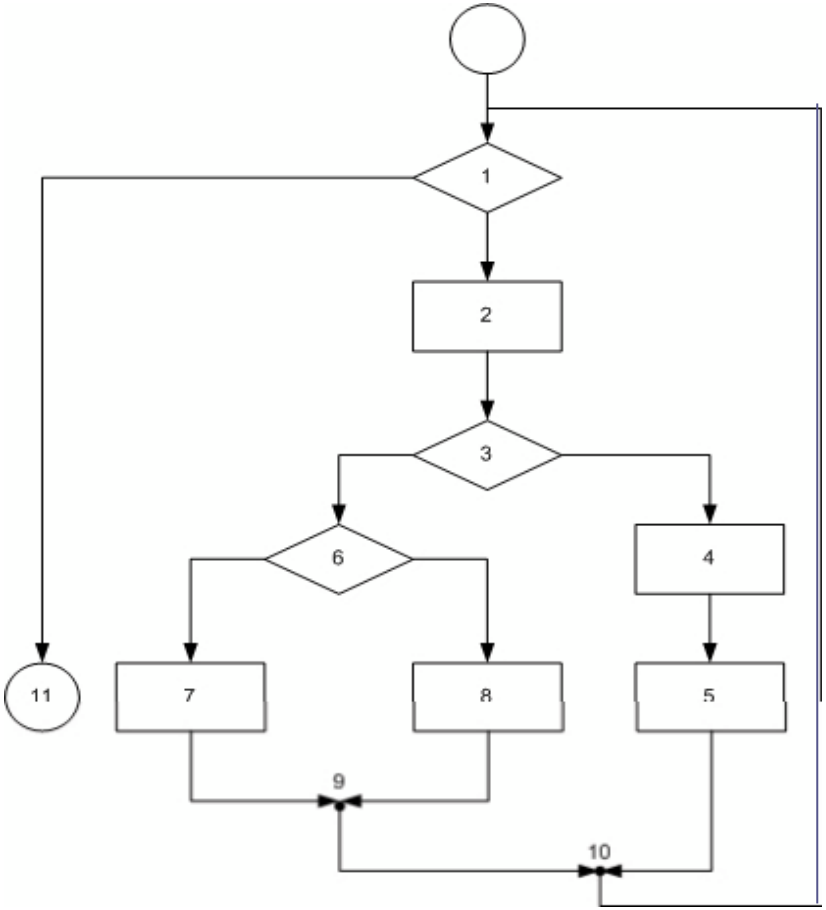
Terminologi dasar yang berkaitan dengan flow graph

- **Node**: Setiap node flow graph merupakan satu atau lebih pernyataan prosedural.
- **Node Predikat**: Setiap node yang berisi kondisi.
- **Edge**: Edge merupakan koneksi antara dua node. Edge antara node mewakili aliran kontrol. Edge harus berakhir pada node, bahkan jika node tidak mewakili pernyataan prosedural yang berguna.
- **Region**: Sebuah daerah dalam flow graph adalah area yang dibatasi oleh edge dan node.

Contoh konversi dari flow chart ke flow graph

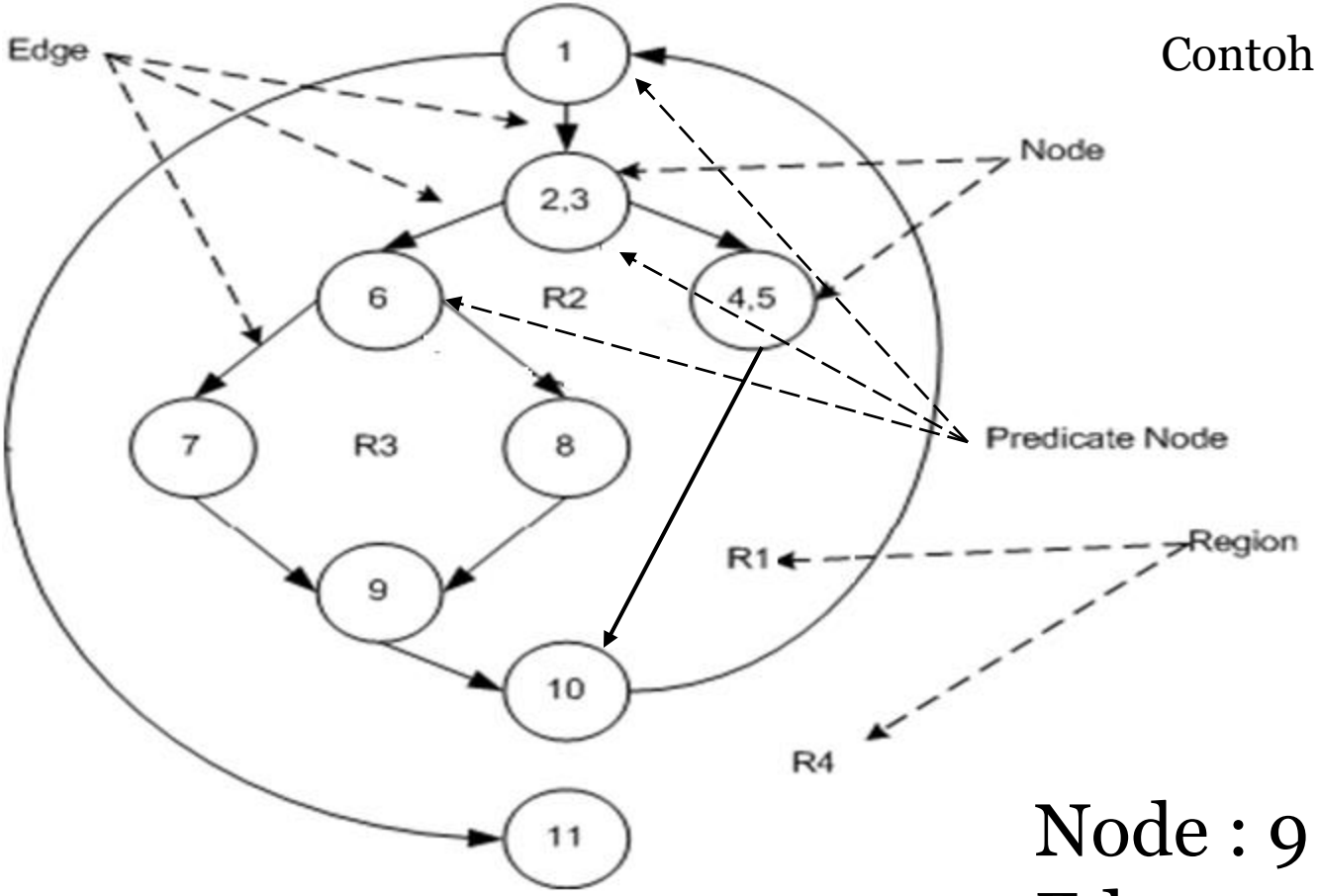
```
1 Do while records remain read record;
2         Calculate proses;
3         If record field 1 = 0
4                   Then process record;
5                   Store in buffer;
                    Increment counter;
6         Else If record field 2 = 0
7                   Then reset counter;
8                   Else process record;
                  Store in file;
9                   Endif
10         Endif
11  Enddo
    End
```

Kode program



Flow chart

Contoh 1



Flow graph

Node : 9
Edge : 11
Region : 4
Predicate node : 3

A. Cyclomatic Complexity

Kompleksitas cyclomatic untuk flow graph ditentukan dalam salah satu dari tiga cara, yaitu:

1. Jumlah region flow graph sesuai dengan kompleksitas cyclomatic.

2. Kompleksitas cyclomatic, V (G), untuk flow graph G didefinisikan sebagai

   V (G) = E - N + 2

   di mana E adalah jumlah edge flow graph dan N adalah jumlah node flow graph.

3. Kompleksitas cyclomatic, V (G), untuk aliran G grafik adalah juga didefinisikan sebagai

   V (G) = P + 1

   dimana P adalah jumlah node predikat yang terkandung dalam aliran grafik G.

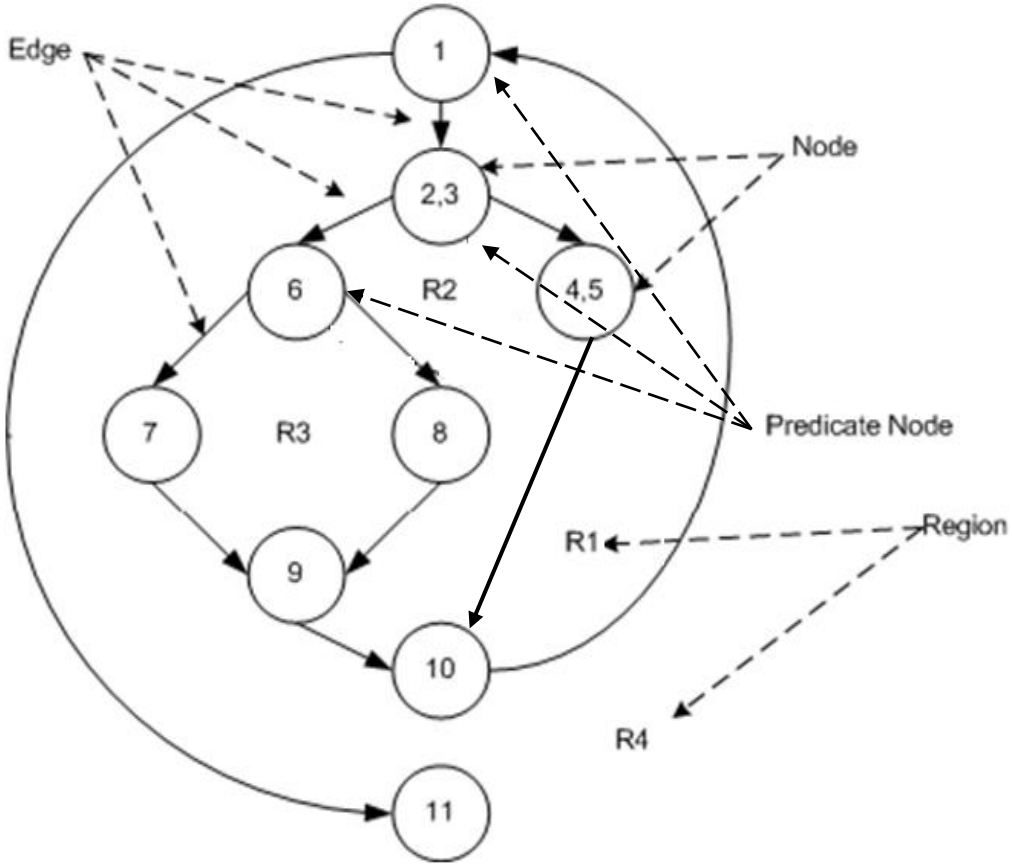
Contoh menghitung Kompleksitas cyclomatic pada contoh 1

- **Cara 1:**
  Jml Region = 4

- **Cara 2:**
  V(G) = E (*edges*) – N (*nodes*) + 2
  V(G) = 11 – 9 + 2 = 4

- **Cara 3:**
  V(G) = P (*predicate node*) + 1
  V(G) = 3 + 1 = 4

Basis path contoh 1



Berdasarkan urutan alurnya, didapatkan suatu kelompok basis *flow graph*.

- Jalur 1 : 1–11
- Jalur 2 : 1-2-3-4-5-10-1-11
- Jalur 3 : 1-2-3-6-7-9-10-1-11
- Jalur 4 : 1-2-3-6-8-9-10-1-11

Tahapan test cases

Tahapan dalam membuat *test cases* dengan menggunakan *cyclomatic complexity*:

1. Gunakan disain atau kode sebagai dasar, gambarlah *flow graph*
2. Berdasarkan *flow graph*, tentukan *cyclomatic complexity*
3. Tentukan kelompok basis dari jalur independen secara linier
4. Siapkan *test cases* yang akan melakukan eksekusi dari tiap jalur dalam kelompok basis

Contoh test cases

- *Test case* jalur (*Path*) 1
  - Nilai(record.eof) = input valid, dimana record.eof = true
  - Hasil yang diharapkan : Sistem keluar dari loop dan sub program.

- *Test case* jalur (*Path*) 2
  - Nilai(field 1) = input valid, dimana field 1 = 0
  - Nilai(record.eof) = input valid, dimana record.eof = false
  - Nilai(counter) = Nilai(counter) + 1
  - Hasil yang diharapkan : Sistem melakukan [process record], [store in buffer] dan [increment counter].

Contoh test cases

- *Test case* jalur (*Path*) 3
  - Nilai(field 2) = input valid, dimana field 2 = 0
  - Nilai(record.eof) = input valid, dimana record.eof = false
  - Nilai(counter) = 0
  - Hasil yang diharapkan : Sistem melakukan [reset counter].

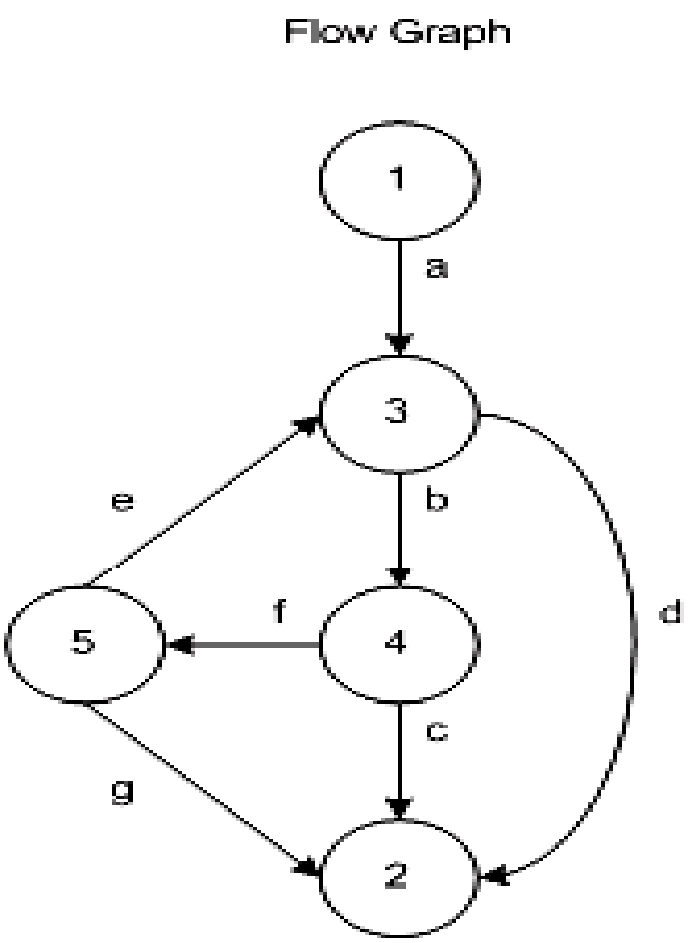
- *Test case* jalur (*Path*) 4
  - Nilai(field 2) = input valid, dimana field 2 <> 0
  - Nilai(record.eof) = input valid, dimana record.eof = false
  - Hasil yang diharapkan : Sistem melakukan [process record] dan [store in file].

B. Graph Matrix

- Adalah matrik berbentuk segi empat sama sisi, dimana jumlah baris dan kolom sama dengan jumlah *node,* dan identifikasi baris dan kolom sama dengan identifikasi *node*, serta isi data adalah keberadaan penghubung antar *node* (*edges*).

- Beberapa properti yang dapat ditambahkan sebagai pembobotan pada koneksi antar *node* di dalam *graph matrix,* sebagai berikut:
  - Kemungkinan jalur (*Edge*) akan dilalui / dieksekusi.
  - Waktu proses yang diharapkan pada jalur selama proses transfer dilakukan.
  - Memori yang dibutuhkan selama proses transfer dilakukan pada jalur.
  - Sumber daya (*resources*) yang dibutuhkan selama proses transfer dilakukan pada jalur.

Graph Matrix



Dihubungkan pada Node

Node	1	2	3	4	5	Koneksi
1			1			1 - 1 = 0
2						
3		1		1		2 - 1 = 1
4		1			1	2 - 1 = 1
5		1	1			2 - 1 = 1

Cyclomatic Complexity = 3 + 1 = 4

Connection Matrix

Control Structure Testing

Control structure testing meliputi:

a. **Testing kondisi** (*Condition Testing*)
b. **Testing alur data** (*Data Flow Testing*)
c. **Testing *loop*** (*Loop Testing*)

A. Testing Kondisi (*Condition Testing*)

Tipe elemen yang mungkin ada dalam suatu kondisi adalah:

- Operator *boolean*
- Variabel *boolean*
- Sepasang *boolean parentheses*
- Operator relasional
- Ekspresi aritmatika.

Testing Kondisi (*Condition Testing*)

Tipe *error* pada kondisi adalah sebagai berikut:

- Kesalahan operator *boolean*
- Kesalahan variabel *boolean*
- Kesalahan *boolean parentheses*
- Kesalahan operator relasional
- Kesalahan ekspresi aritmatika.

Strategi tes kondisi :

- ***Branch Testing***
- ***Domain Testing***
- ***BRO (Branch and Relational Operator) Testing***

A.1.Branch Testing

- Merupakan strategi tes kondisi yang paling sederhana.
- Sebagai contoh ilustrasi penggunaan, diasumsikan terdapat penggalan kode berikut:

```
IF (X=1) AND (Y=1) AND (Z=1) then
        [Do Something]
END IF
```

Untuk *branch testing* dibutuhkan dua tes, yaitu:

- Dengan memberikan nilai (X, Y, Z) = (1,1,1), untuk mengevaluasi dengan kondisi benar (*true*).
- Dan dengan memberikan nilai (X,Y,Z) = (2,1,1), sebagai wakil untuk mengevaluasi dengan kondisi salah (*false*).

A.2.Domain Testing (1)

- Membutuhkan tiga atau empat tes yang dilaksanakan untuk suatu ekspresi relasional. Untuk suatu ekspresi relasional dalam bentuk:

  E1<operator-relasional>E2

- Contoh:

```
If  (X + 1) > (Y – Z)  then
        [Do Something]
End if
```

A.2.Domain Testing (2)

```
If  (X + 1) > (Y – Z)  then
        [Do Something]
End if
```

Dimana E1 diwakili oleh (X + 1) dan E2 diwakili oleh (Y – Z). Ada tiga tes yang dilakukan, yaitu:

- Tes pertama dengan mewakilkan E1 dan E2 dengan nilai 5 dan 2, yang didapat dari masukan (X,Y,Z) = (4,5,3), agar E1 > E2. Dan hasil kondisi yang diharapkan adalah *true*.
- Tes kedua dengan mewakilkan E1 dan E2 dengan nilai 2 dan 2, yang didapat dari masukan (X,Y,Z) = (1,4,2), agar E1 = E2. Dan hasil kondisi yang diharapkan adalah *false*.
- Tes ketiga dengan mewakilkan E1 dan E2 dengan nilai 1 dan 2, yang didapat dari masukan (X,Y,Z) = (0,4,2), agar E1 < E2. Dan hasil kondisi yang diharapkan adalah *false*.

A.2.Domain Testing (3)

- Strategi ini dapat mendeteksi *error* dari operator dan variabel *boolean parenthesis*, namun ini hanya dipraktekkan jika n adalah kecil. Contoh:

```
IF X AND Y THEN
        [Do Something]
END IF
```

- Dimana X dan Y adalah variabel *boolean*, maka akan dilakukan tes sebanyak $2^2 = 4$, yaitu dengan memberikan nilai X dan Y:
  a. (X,Y) →(t,f) hasil yang diharapkan f
  b. (X,Y) →(f,t) hasil yang diharapkan f
  c. (X,Y) →(f,f) hasil yang diharapkan f
  d. (X,Y) →(t,t) hasil yang diharapkan t

A.2.Domain Testing (3)

- Strategi ini dapat mendeteksi *error* dari operator dan variabel *boolean parenthesis*, namun ini hanya dipraktekkan jika n adalah kecil. Contoh:

```
IF X AND Y THEN
        [Do Something]
END IF
```

- Dimana X dan Y adalah variabel *boolean*, maka akan dilakukan tes sebanyak $2^2 = 4$, yaitu dengan memberikan nilai X dan Y:
  a. (X,Y) →(t,f) hasil yang diharapkan f
  b. (X,Y) →(f,t) hasil yang diharapkan f
  c. (X,Y) →(f,f) hasil yang diharapkan f
  d. (X,Y) →(t,t) hasil yang diharapkan t

A.3.BRO (Branch and Relational Operator) Testing (ada batasan)

- Teknik ini menjamin deteksi *error* dari operator cabang dan relasional dalam suatu kondisi yang ada dimana semua variabel *boolean* dan operator relasional yang terdapat di dalam kondisi terjadi hanya sekali dan tidak ada variabel yang dipakai bersama.

- Sebagai ilustrasi diberikan contoh-contoh sebagai berikut:

Contoh 1: Suatu kondisi C1: B1 & B2

- Dimana B1 dan B2 adalah variabel *boolean*.
- Batasan kondisi C1 dalam bentuk (D1, D2), dan D1 dan D2 adalah t atau f.
- Nilai (t,f) adalah suatu batasan kondisi C1 dan dicakup oleh tes yang membuat nilai B1 menjadi *true* dan nilai B2 menjadi *false*.
- Strategi *BRO testing* membutuhkan sekumpulan batasan {(t,t), (f,t), (t,f)} dicakup oleh eksekusi dari C1.
- Jika C1 tidak benar terhadap satu atau lebih *error* operator *boolean*, setidaknya satu dari sekumpulan batasan akan membuat C1 salah.

Contoh 2: Suatu kondisi C2 : B1 &(E3 = E4)

- Dimana B1 adalah ekspresi *boolean*, E3 dan E4 adalah ekspresi aritmatika.
- Batasan kondisi C2 dalam bentuk (D1, D2), dan D1 adalah t atau *f* dan D2 adalah >, =, <.
- Bila C2 = C1, kecuali kondisi sederhana kedua pada C2 adalah ekspresi relational, dapat dibangun suatu kumpulan batasan untuk C2 dengan memodifikasi sekumpulan batasan {(t,t), (f,t),(t,f)} yang didefinisikan untuk C1.
- Dimana t untuk (E3 = E4) melambangkan = dan f untuk (E3 = E4) melambangkan < atau >.
- Dengan mengganti (t,t) dan (f,t) dengan (t,=) dan (f,=), dan dengan menggantikan (t,f) dengan (t,<) dan (*t*,>), menghasilkan sekumpulan batasan untuk C2 yaitu {(t,=), (f,=), (t,<), (t,>)}.
- Cakupan untuk sekumpulan batasan diatas akan menjamin deteksi *error* dari operator *boolean* dan relational pada C2.

Contoh 3: Suatu kondisi C3: (E1 > E2) & (E3 = E4)

- Dimana E1, E2, E3, dan E4 adalah ekspresi aritmatika.
- Batasan kondisi C3 dalam bentuk (D1, D2), dan D1 dan D2 adalah >, =, <.
- Bila C3 sama dengan C2 kecuali kondisi sederhana pertama pada C3 adalah ekspresi relational, dapat dibangun sekumpulan batasan untuk C3 dengan memodifikasi kumpulan batasan untuk C2 dengan menggantikan t dengan >, dan f dengan =, dan <, sehingga didapat {(>,=),(=,=),(<,=),(>,>),(>,<)}
- Cakupan kumpulan batasan ini akan menjamin deteksi *error* dari operator relational pada C3.

Contoh 4:

```
IF (X = TRUE) AND (Y = TRUE)  AND (Z = TRUE) THEN
        [Do Something]
END IF
```

- Dimana X, Y dan Z adalah variabel *boolean*. Maka dapat dituliskan kembali, menurut *Branch and relational operator testing (BRO),* yang diterdapat pada [TAI89]: C4: X & Y & Z
- Dengan C4 adalah identitas dari kondisi yang mewakili *predicate* dari penggalan kode di atas.
- Dibutuhkan delapan tes dengan batasan kondisi C4, sebagai berikut: {(t,f,f), (t,f,t), (t,t,f), (t,t,t), (f,f,f), (f,f,t), (f,t,f), (f,t,t)}, dengan hasil kondisi C4 yang diharapkan adalah (f, f, f, t, f, f, f, f).

B. Data Flow Testing (strategi *Definition-Use testing)*

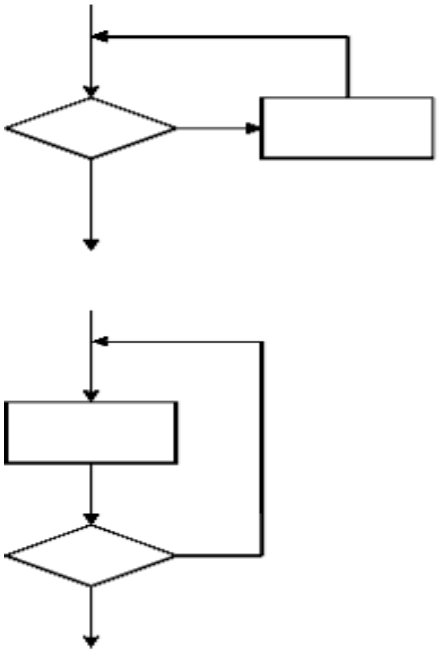
- Metode *data flow testing* memilih jalur program berdasarkan pada **lokasi dari definisi** dan penggunaan variabel-variabel pada program.
- Strategi *data flow testing* sangat berguna untuk menentukan jalur tes pada program yang berisi pernyataan *nested* if dan *loop*

```
Proc x
    B1;
    Do while C1
        If C2
            Then
                If C4
                    Then B4;
                    Else B5;
                Endif;
            Else
                If C3
                    Then B2;
                    Else B3;
                Endif;
        Endif;
    Enddo;
    B6;
End proc;
```

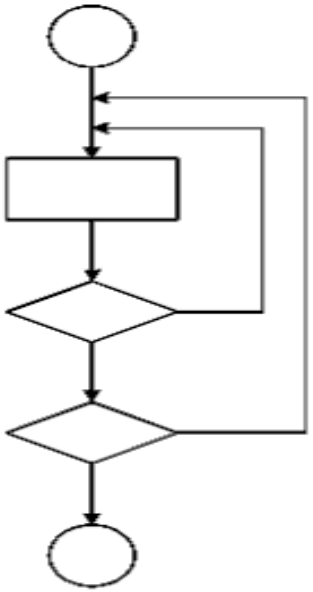
- Untuk menggunakan strategi *DU testing* dalam memilih jalur tes dari diagram *control flow*, perlu mengetahui defnisi dan penggunaan dari variabel di tiap kondisi atau blok pada PDL.

- Asumsi variabel X didefinisikan pada pernyataan akhir dari blok B1, B2, B3, B4, dan B5
- Variabel X digunakan pada pernyataan pertama dari blok B2, B3, B4, B5, dan B6.
- Strategi *DU testing* membutuhkan suatu eksekusi jalur terpendek dari tiap Bi, $1 < i \leq 5$, ke tiap Bj, $2 < j \leq 6$. (Suatu testing tertentu juga mencakup penggunaan tiap variabel dari X dalam kondisi C1,C2, C3, dan C4.)
- Walaupun ada 25 ikatan DU dari variabel X, hanya dibutuhkan lima jalur tes untuk mencakup ikatan DU ini.
- Alasannya adalah kelima jalur ini dibutuhkan untuk mencakup ikatan DU X dari Bi, $1 < i \leq 5$, ke B6 dan ikatan DU lainnya dapat di cakup dengan membuat kelima jalur ini beriterasi sesuai dengan *loop*.

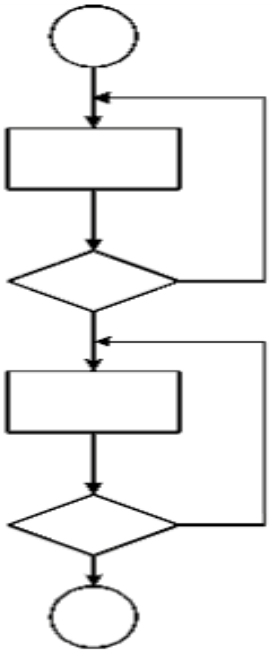
C. Loop Testing



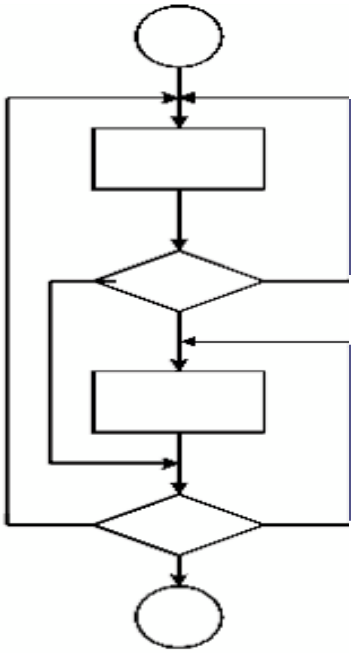
Simple Loops



Nested Loops



Concatenated Loops



Unstructured Loops

Loop Testing

- *Loop testing* adalah suatu teknik *white box testing* yang berfokus pada validitas konstruksi *loop* secara eksklusif. Empat kelas yang berbeda dari *loop*, yaitu:
  - ***Simple Loops***
  - ***Nested Loops***
  - ***Concatenated Loops***
  - ***Unstructured Loops***

Simple Loops

- Sekumpulan tes berikut ini dapat digunakan untuk *simple loops*, dimana n adalah jumlah maksimum yang dapat dilewatkan pada *loop*:

1. Lompati *loop* secara keseluruhan, tak ada iterasi / lewatan pada *loop*.
2. Lewatkan hanya satu kali iterasi pada *loop*.
3. Lewatkan dua kali iterasi pada *loop*.
4. Lewatkan m kali iterasi pada *loop* dimana m<n.
5. Lewatkan n-1, n, n+1 kali iterasi pada *loop*.

Nested Loops

- Jika pendekatan tes untuk *simple loops* dikembangkan pada *nested loops*, jumlah kemungkinan tes akan berkembang secara geometris searah dengan semakin tingginya tingkat dari *nested loops*.
  1. Mulailah dari *loop* yang paling dalam. Set semua *loops* lainnya dengan nilai minimum.
  2. Lakukan tes *simple loops* untuk *loop* yang paling dalam, dengan tetap mempertahankan *loops* yang ada di luarnya dengan nilai parameter iterasi yang minimum. Tambahkan tes lainnya untuk nilai yang diluar daerah atau tidak termasuk dalam batasan nilai parameter iterasi.
  3. Kerjakan dari dalam ke luar, lakukan tes untuk *loop* berikutnya, tapi dengan tetap mempertahankan semua *loop* yang berada di luar pada nilai minimum dan *nested loop* lainnya pada nilai yang umum.
  4. Teruskan hingga keseluruhan dari *loops* telah dites.

Concatenated Loops

- *Concatenated loops* dapat dites dengan menggunakan pendekatan yang didefinisikan untuk *simple loops*, jika tiap *loops* independen (tidak saling bergantung) antara satu dengan yang lainnya.
- Dikatakan dua *loops* tidak independen, jika dua *loops* merupakan *concatenated loops*, dan nilai *loop counter* pada *loop* 1 digunakan sebagai nilai awal untuk *loop* 2.
- Bila *loops* tidak independen, direkomendasikan memakai pendekatan sebagaimana yang digunakan pada *nested loops*.

Unstructured Loops

- Tidak dapat dites dengan efektif. Dan bila memungkinkan *loops* jenis ini harus didisain ulang

D. Lines of Code

- Pengukuran sederhana: menghitung jumlah baris kode dalam program dan menggunakan perhitungan ini untuk mengukur kompleksitas. Berdasarkan studi yang telah dilakukan:
  - Program kecil mempunyai *error* rata-rata 1,3 % sampai 1,8 %.
  - Program besar mempunyai kenaikan *error* rata-rata dari 2,7 % sampai 3,2 %.

E. Halstead's Metrics

- Halstead's metric adalah pengukuran yang berdasarkan pada penggunaan operator-operator (seperti kata kunci) dan operan-operan (seperti nama variabel, obyek database) yang ada dalam suatu program.
  - n1 = jumlah operator yang unik (*distinct*) dalam program
  - n2 = jumlah operan yang unik (*distinct*) dalam program.

Panjang program: H = n1 log2 n1 + n2 log2 n2.

  - N1 = perhitungan jumlah keseluruhan operator program.
  - N2 = perhitungan jumlah keseluruhan operan program.

Prediksi *bug*: B = (N1 + N2) log2 (n1 + n2) / 3000