Detecting and Debugging Insecure Information Flows

Wes Masri
American University of Beruit
Beruit, Lebanon

Andy Podgurski and David Leon
Electrical Engineering & Computer Science Dept.
Case Western Reserve University
Cleveland, OH

Overview

- Present new approach to *dynamic information flow analysis*
  - First precise *forward computing algorithm* for intra- and inter-procedural DIFA of both structured and unstructured programs
  - Integrated with dynamic slicing
  - Static preprocessing permits detection of *implicit flows*
- Describe DIFA tool for analyzing Java programs
  - Can be used to detect and debug insecure flows in programs
  - Actual flows checked against configurable *information flow policy*
  - Present results of empirical evaluation

Information flows in programs

- Certain information flows between objects may indicate:
  - leakage of confidential information
    - e.g., from password file to untrusted socket
  - loss of data integrity
    - e.g., corruption of critical information
  - untested interactions between components
    - e.g., one module expects pounds, another kilograms

AP4

Explicit flows

- Via data flows
  - e.g., from **x** to **z** in

    y = f(x); z = g(y);
- Via control flows
  - e.g., from **y** to **x** in

    ```
    x = 1;
    if (y == 0)
      x = 0;
    print(x);
    ```

    when **y == 0** holds

Implicit flows

- Occur when value of **x** depends on value of **y** used in branch condition, though branch taken doesn't define **x**, as with
  - e.g., flow from y to x in

```
x = 1;
if (y == 0)
  x = 0;
print(x);
```

when y != 0 holds

Information flow analysis

- Can reveal insecure flows automatically
- Based on control flow and data flow analysis
- Closely related to
  - program dependence analysis
  - program slicing
- May be done statically or dynamically
  - Dynamic version is more precise
  - Closely related to dynamic slicing

Forward computing algorithms for DIFA and dynamic slicing

- Operate in tandem with program execution
- Don’t require stored execution trace
- Can be used online if overhead permits
- Support interactive debugging

Dynamic dependences

- We characterize information flows and dynamic slices in terms of
  - *dynamic control dependences* of actions upon branch conditions
  - *dynamic data dependences* of variable uses upon definitions
- Computed from execution trace

Dynamic control dependence

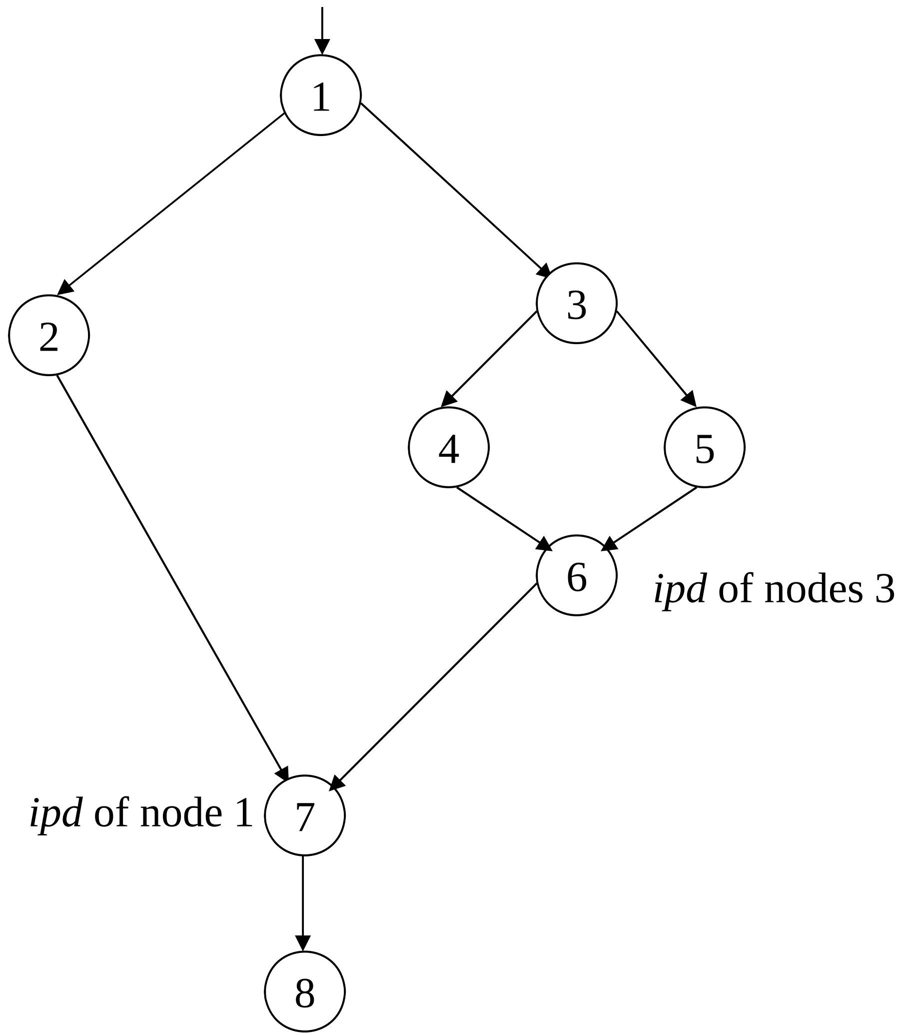
- Action *t*[*m*] is dynamically CD on action s[*k*] if s[*k*] is most recent predicate action to occur prior to *t*[*m*] such that statement *t* is statically CD on statement *s*
  - *t* is statically CD on *s* if it postdominates one successor of *s* but not the other
- Our forward computing algorithm:
  - is more precise than other proposed algorithms
  - uses constant time per action processed
  - uses bounded space
  - handles unstructured code

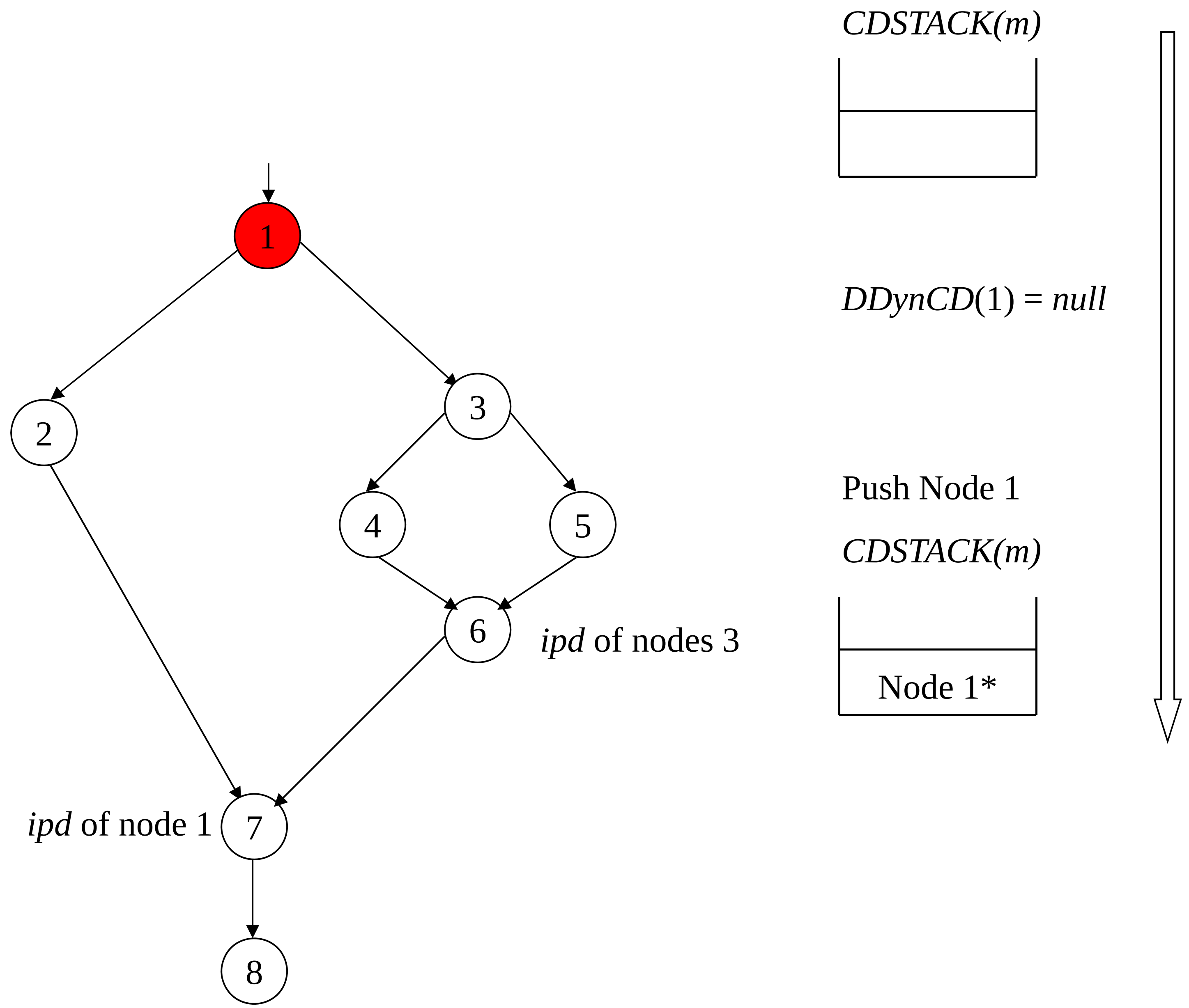
Compute Direct Dynamic Control Dependence( )

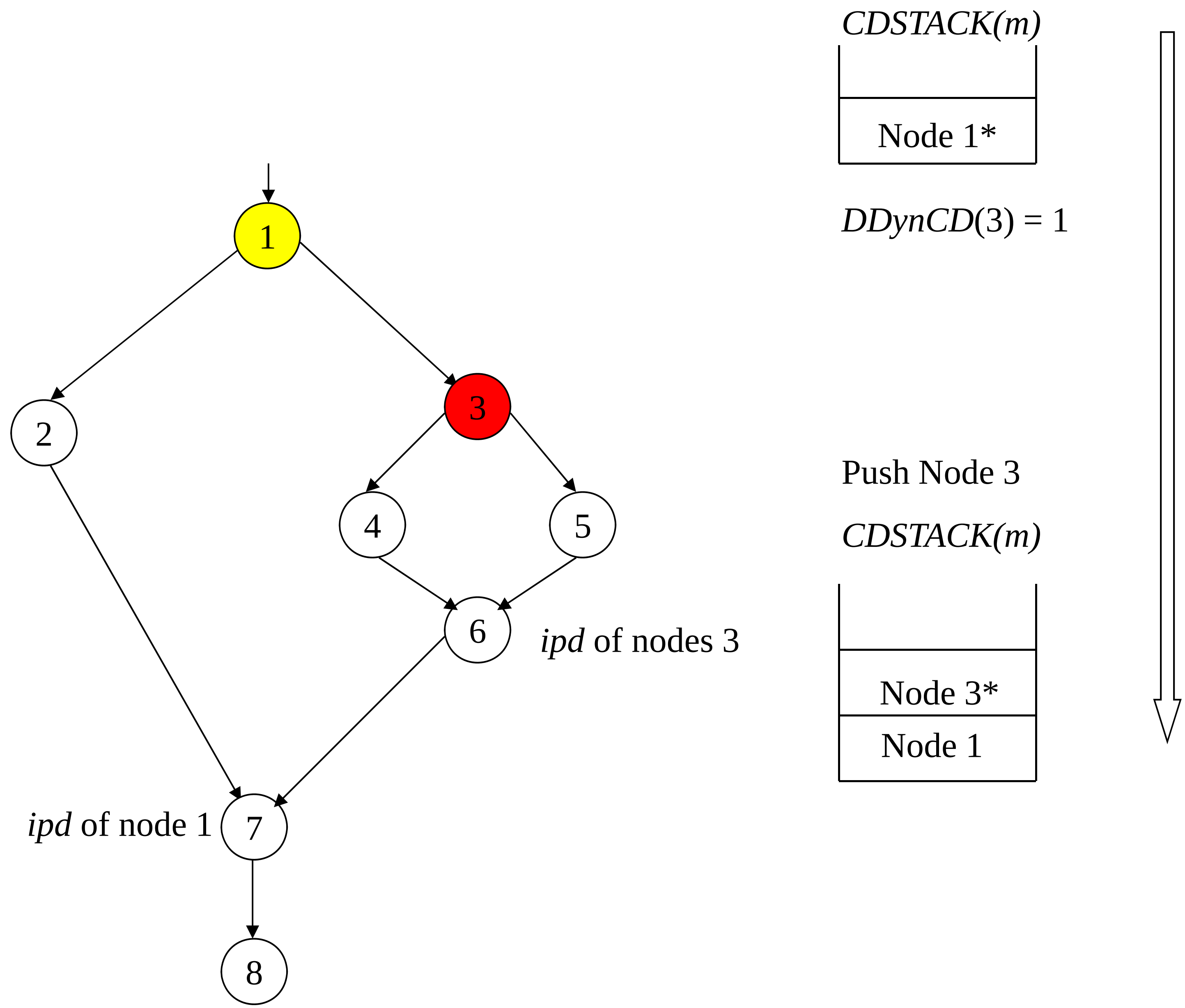
***ipd*(*s*): immediate post dominator of a statement s**

***TOS*(*m*): top of *CDSTACK*(*m*)**

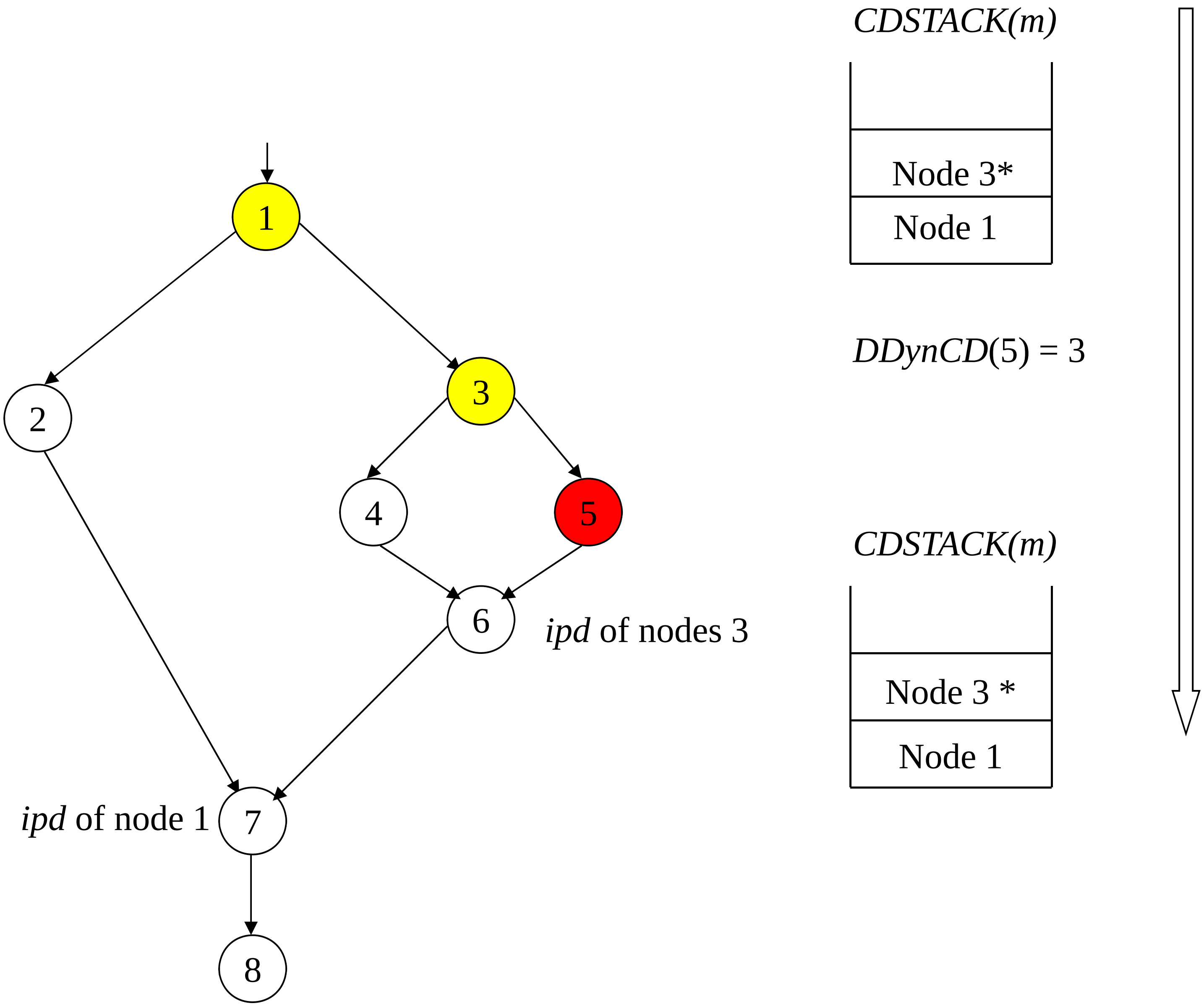
```
if  ¬Empty(CDSTACK(m)) and s = ipd(TOS(m))  then
        pop CDSTACK(m)          // inactivate dynamic control scope
Endif

if  ¬Empty(CDSTACK(m)) then
      DDynCD(sλ) = TOS(m)    // sλ DDynCD TOS(m)
else
      DDynCD(sλ) = null
Endif

if  s is a decision statement  then
        if  ¬Empty(CDSTACK(m)) and ipd(s) = ipd(TOS(m)) then
                pop CDSTACK(m)
        endif
        push s onto CDSTACK(m)  // activate dynamic control scope
endif
```


1
2
3
4
5
6
ipd of nodes 3
ipd of node 1
7
8

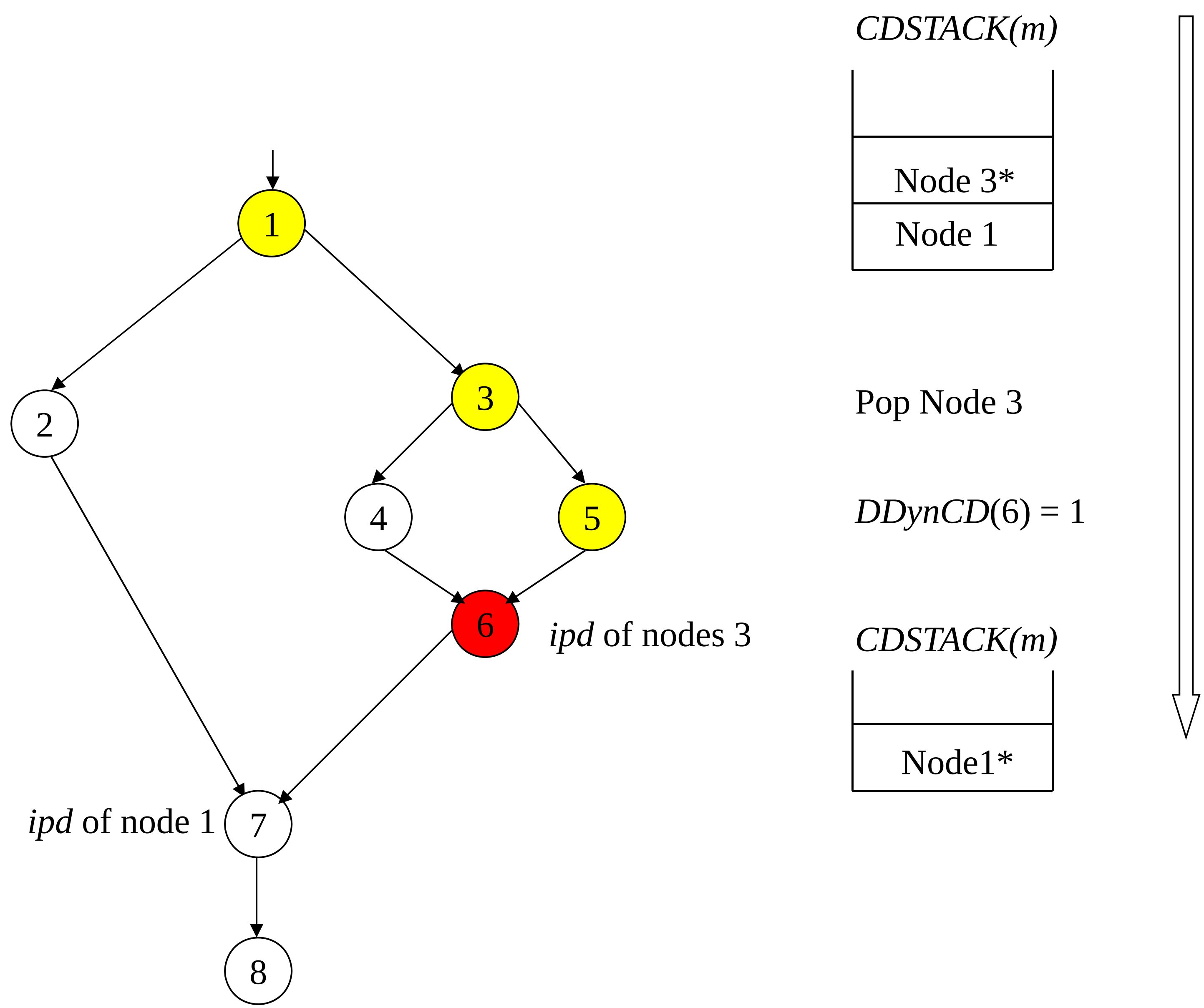

CDSTACK(m)
DDynCD(1) = null
Push Node 1
CDSTACK(m)
Node 1*
1
2
3
4
5
6
ipd of nodes 3
ipd of node 1
7
8



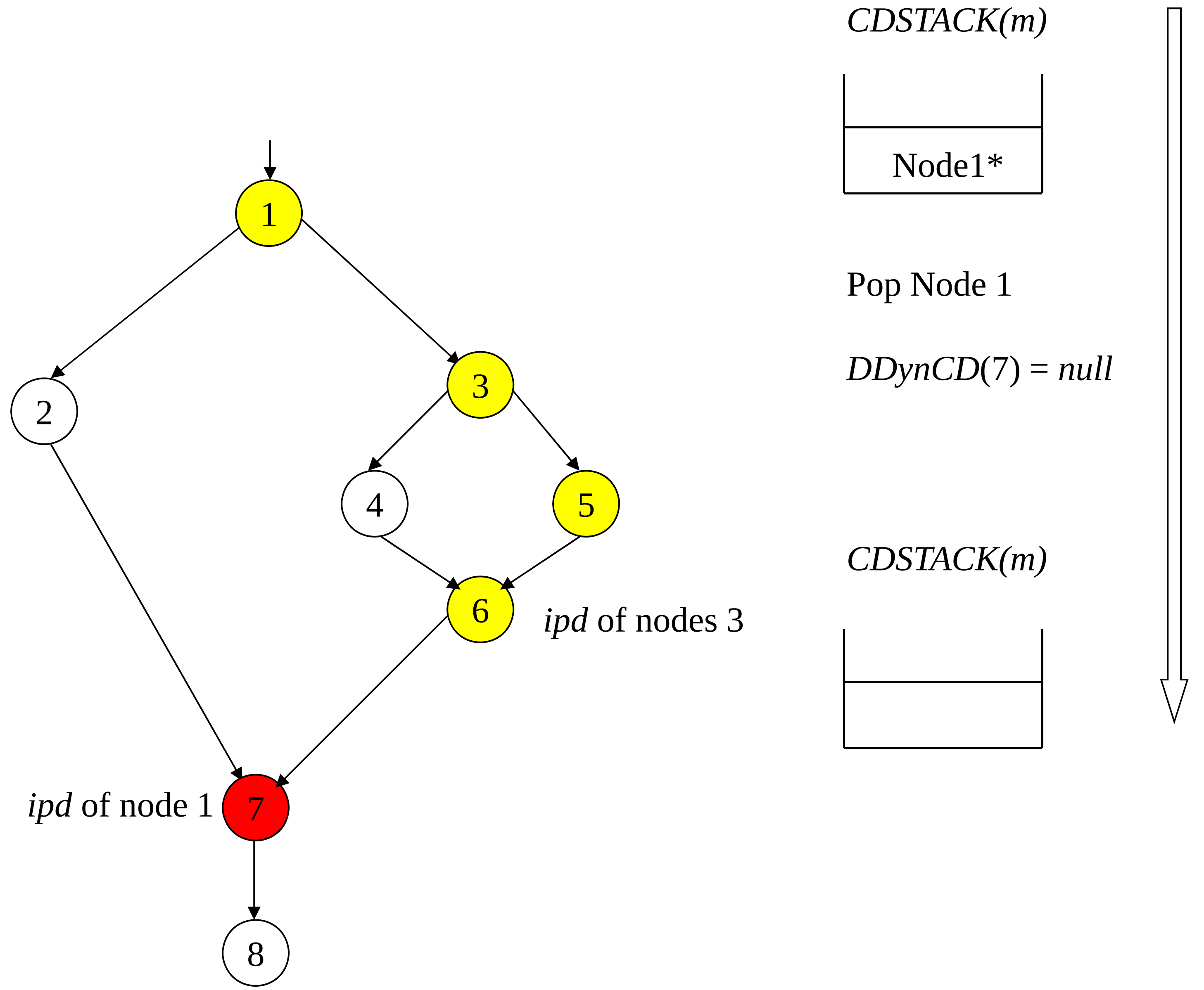
CDSTACK(m)
Node 1*
DDynCD(3) = 1
Push Node 3
CDSTACK(m)
Node 3*
Node 1
1
2
3
4
5
6
ipd of nodes 3
ipd of node 1
7
8



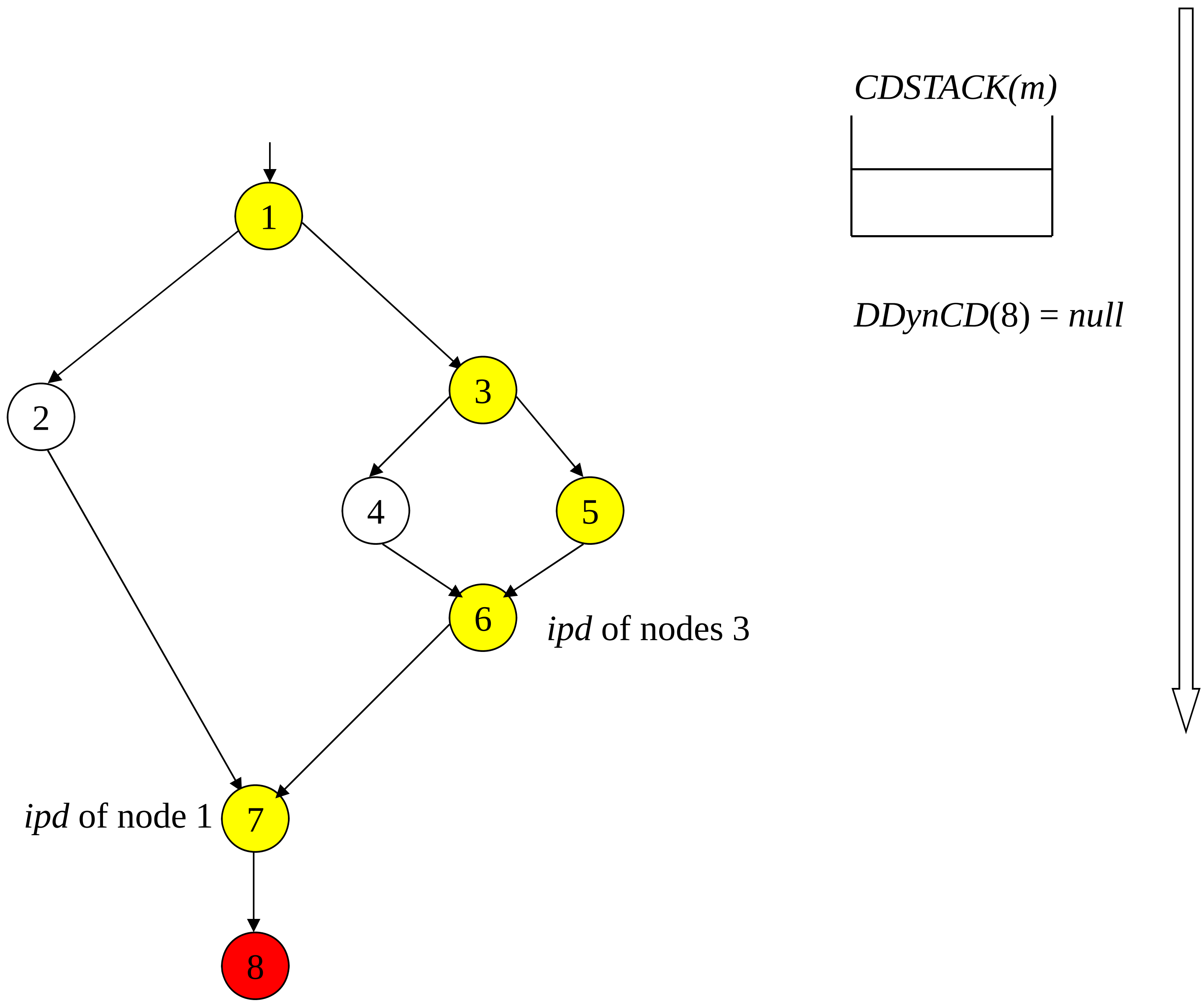
1
2
3
4
5
6
ipd of nodes 3
ipd of node 1
7
8
CDSTACK(m)
Node 3*
Node 1
DDynCD(5) = 3
CDSTACK(m)
Node 3 *
Node 1



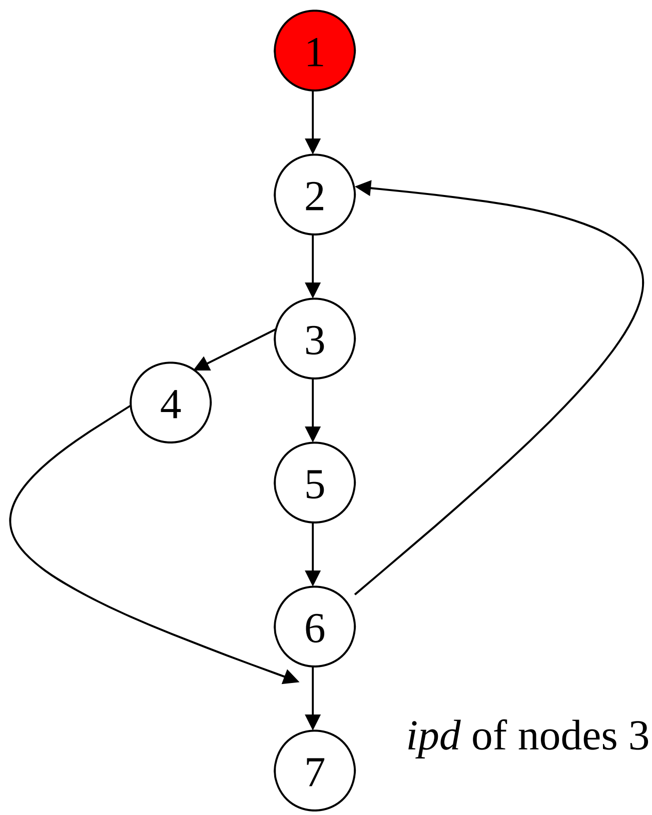
CDSTACK(m)
Node 3*
Node 1
Pop Node 3
DDynCD(6) = 1
CDSTACK(m)
Node1*
1
2
3
4
5
6
7
8
ipd of nodes 3
ipd of node 1



1
2
3
4
5
6
7
8
ipd of nodes 3
ipd of node 1
CDSTACK(m)
Node1*
Pop Node 1
DDynCD(7) = null
CDSTACK(m)

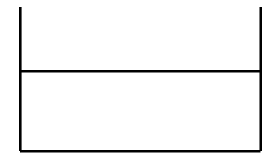

CDSTACK(m)
DDynCD(8) = null
1
2
3
4
5
6
ipd of nodes 3
ipd of node 1
7
8

Step 4 ensures that the size of *CDSTACK(m)* is bounded by the number of decision vertices in $G(m)$

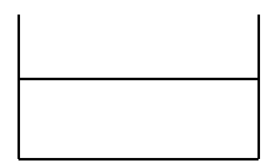


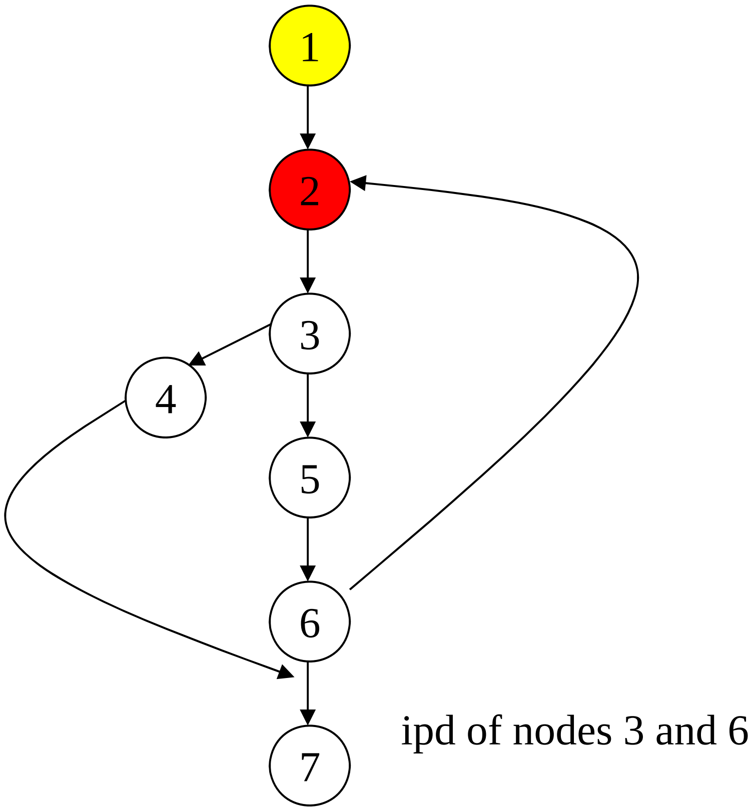
ipd of nodes 3 and 6

CDSTACK(m)

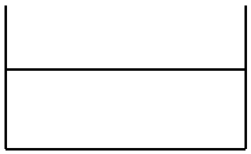


CDSTACK(m)
(omit step 4)

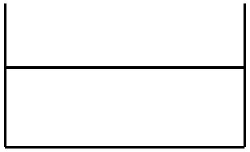


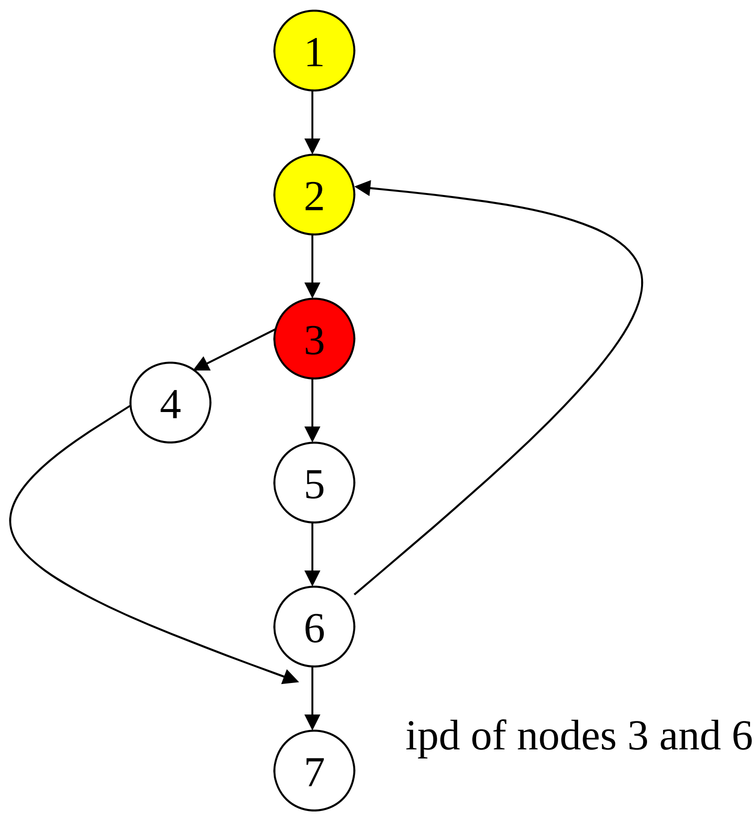


CDSTACK(m)

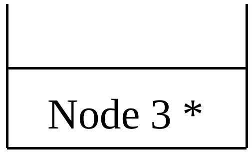


CDSTACK(m)
(omit step 4)

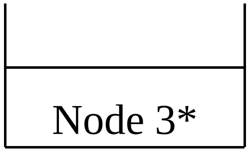


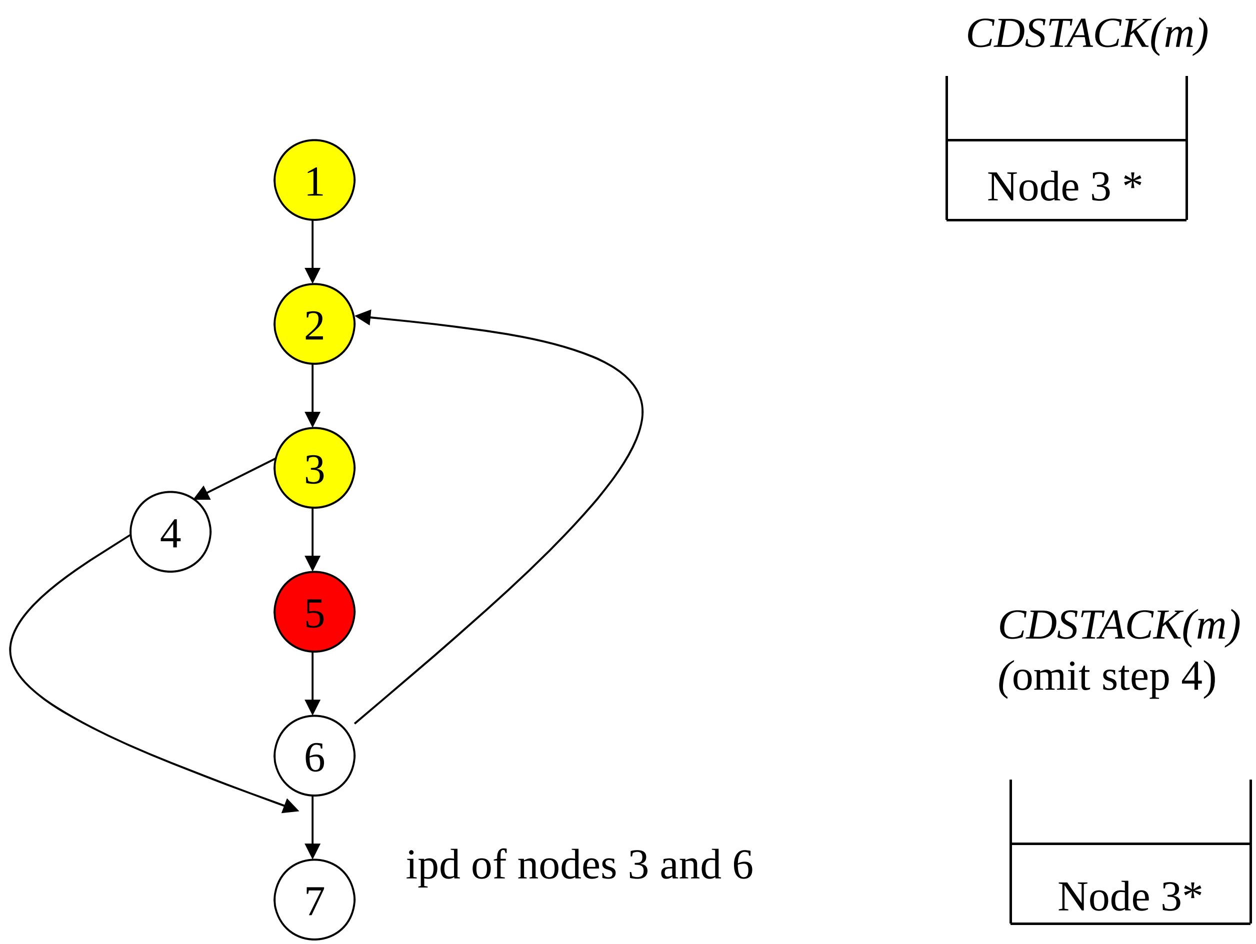


CDSTACK(m)

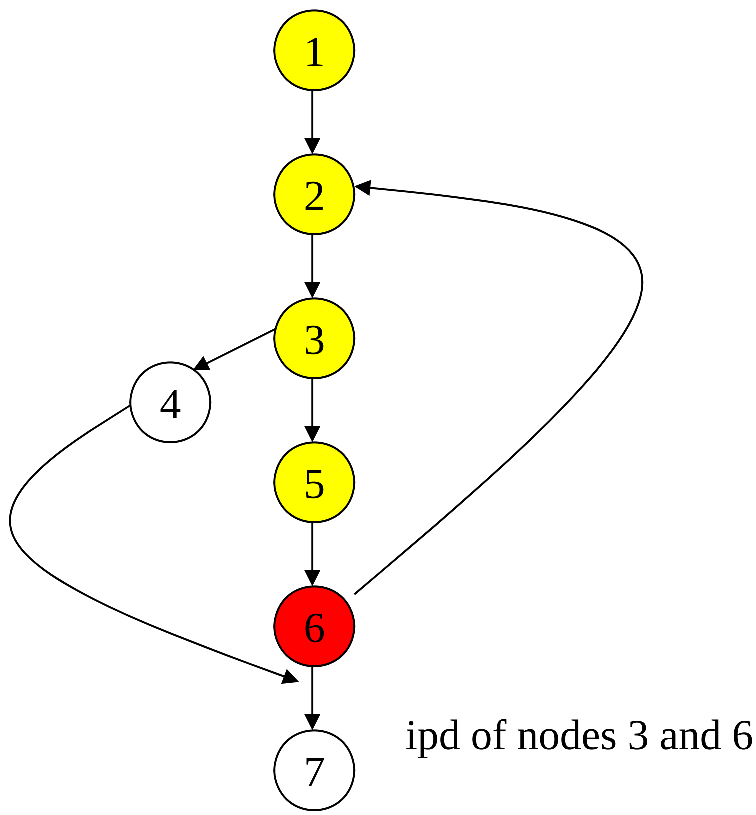


CDSTACK(m)
(omit step 4)

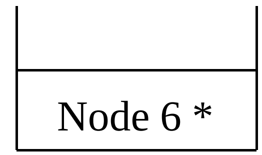




CDSTACK(m)
Node 3 *
1
2
3
4
5
6
7
ipd of nodes 3 and 6
CDSTACK(m)
(omit step 4)
Node 3*

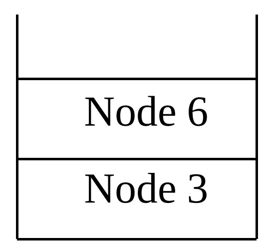


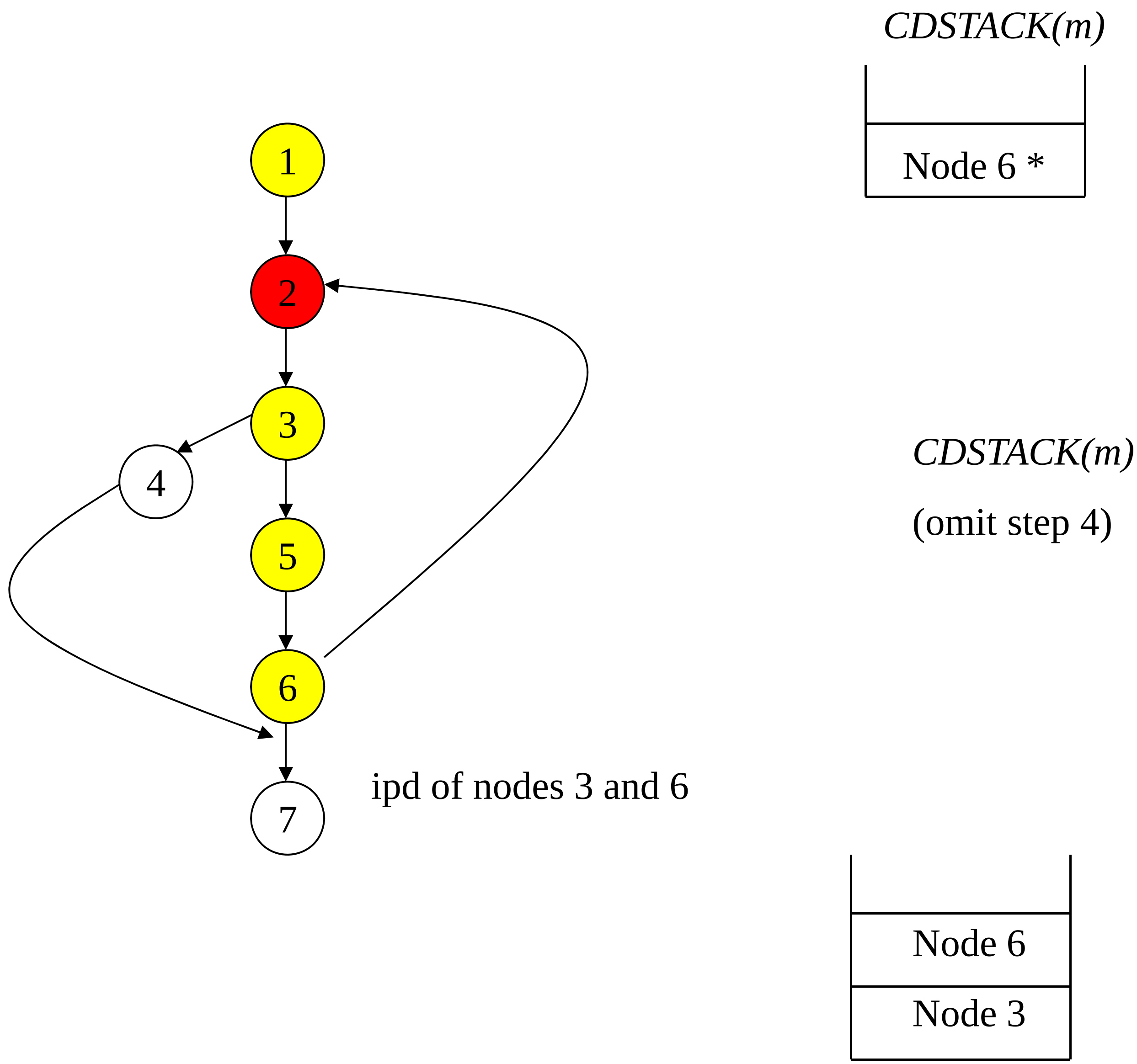
CDSTACK(m)



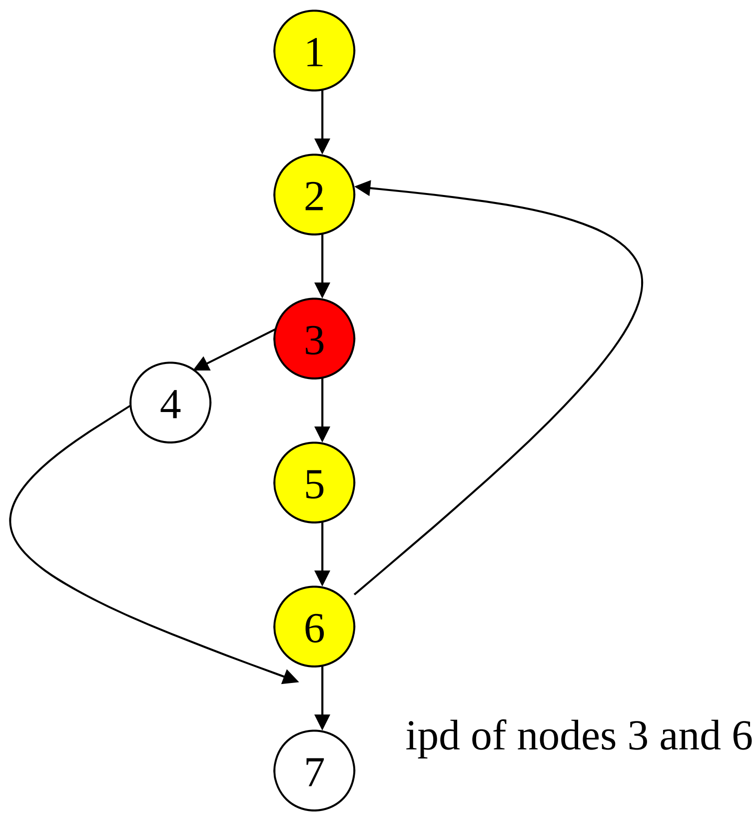
CDSTACK(m)

(omit step 4)





CDSTACK(m)
Node 6 *
1
2
3
4
5
6
7
ipd of nodes 3 and 6
CDSTACK(m)
(omit step 4)
Node 6
Node 3



CDSTACK(m)

Node 3 *

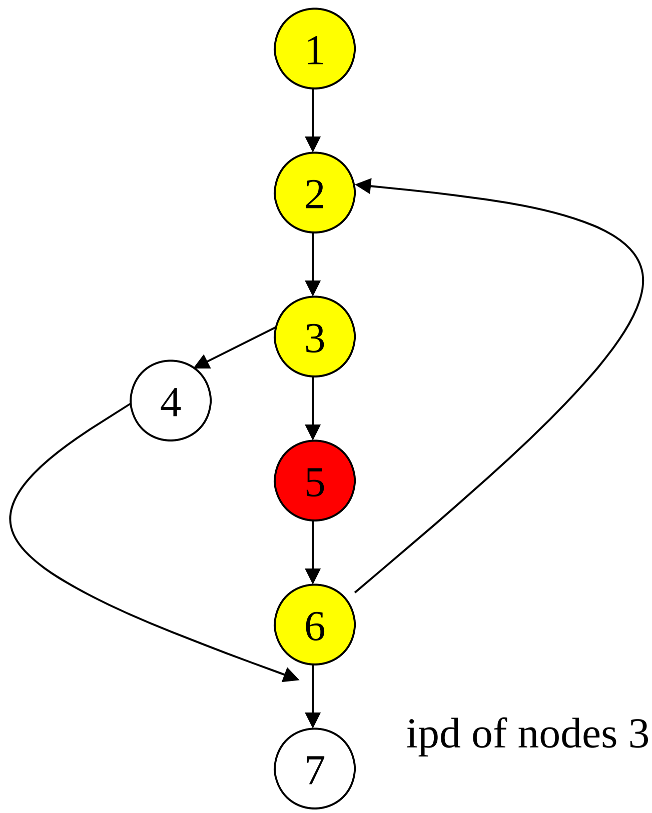
CDSTACK(m)

(omit step 4)

Node 3

Node 6

Node 3



ipd of nodes 3 and 6

CDSTACK(m)

Node 3 *

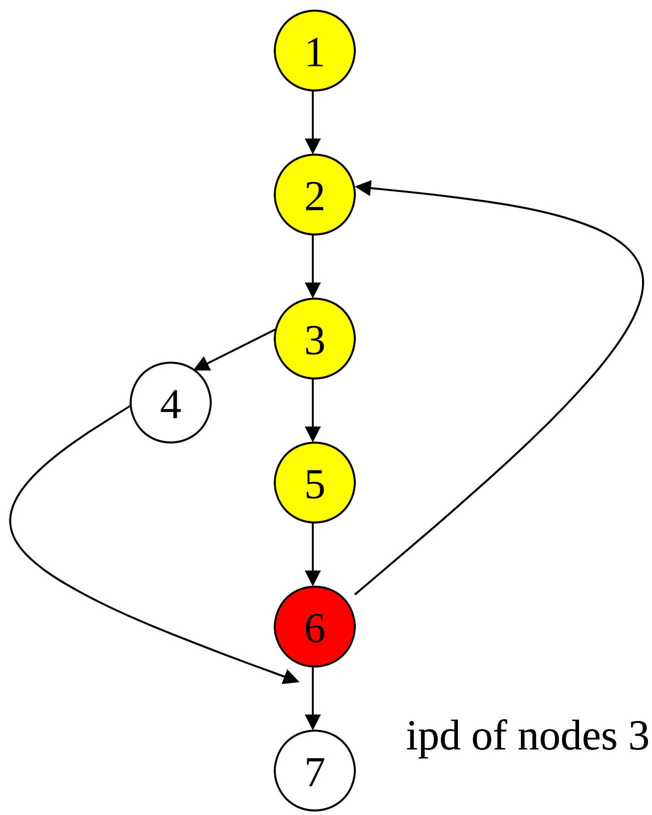
CDSTACK(m)

(omit step4)

Node 3

Node 6

Node 3



ipd of nodes 3 and 6

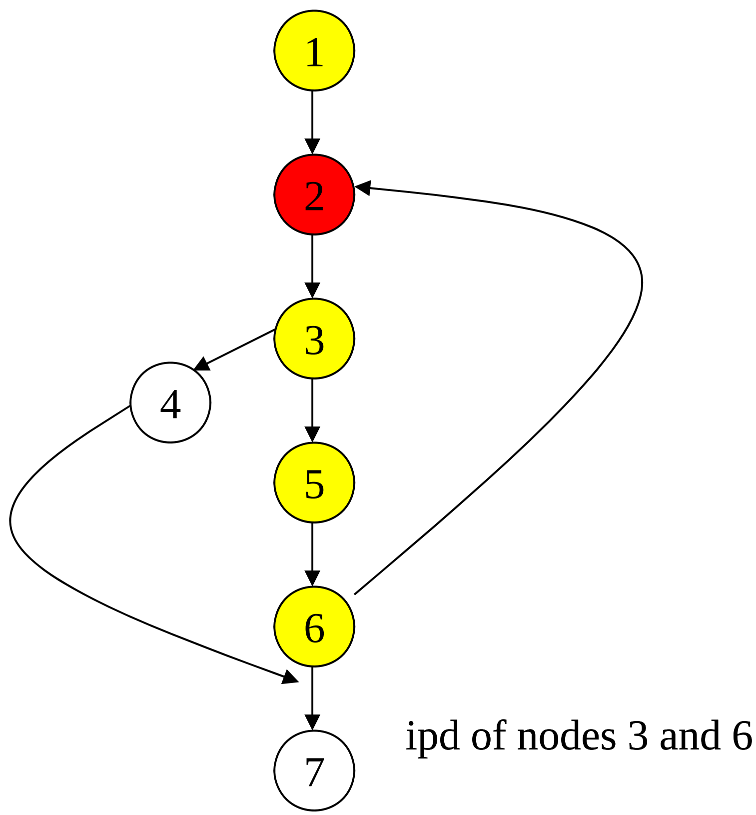
CDSTACK(m)

Node 6 *

CDSTACK(m)

(omit step 4)

Node 6
Node 3
Node 6
Node 3



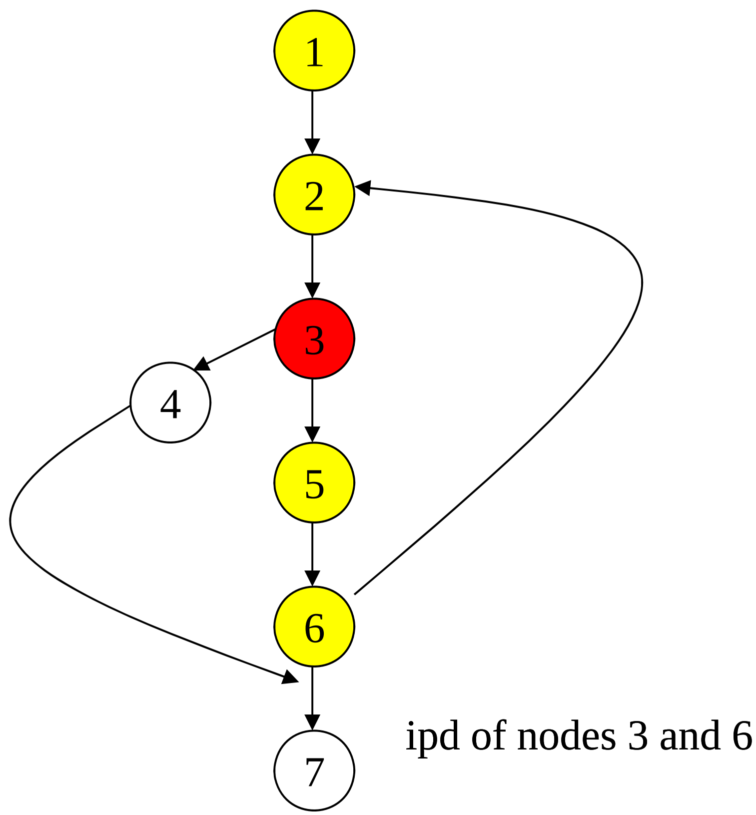
CDSTACK(m)

Node 6 *

CDSTACK(m)

(omit step 4)

Node 6
Node 3
Node 6
Node 3



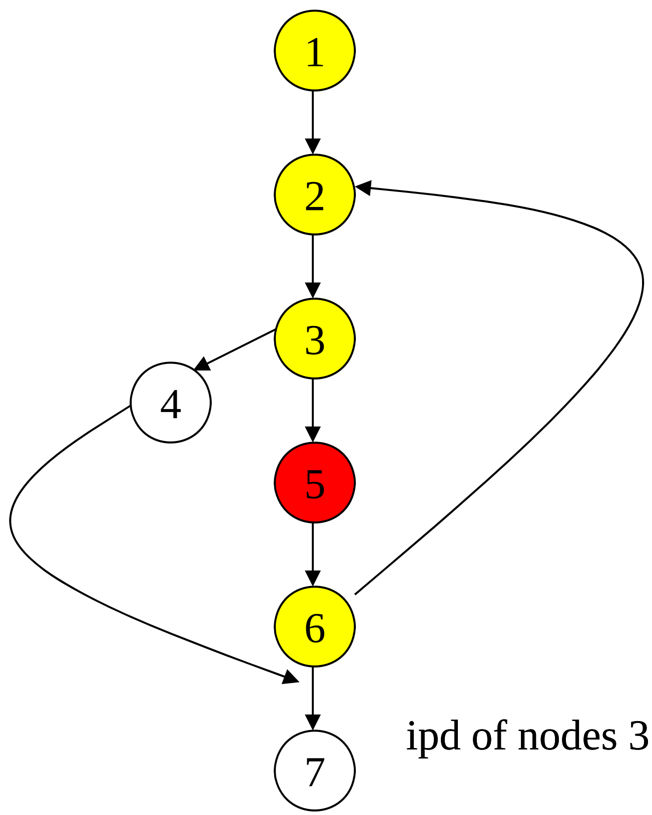
CDSTACK(m)

Node 3 *

CDSTACK(m)

(omit step 4)

Node 3
Node 6
Node 3
Node 6
Node 3



ipd of nodes 3 and 6

CDSTACK(m)

Node 3 *

CDSTACK(m)

(omit step 4)

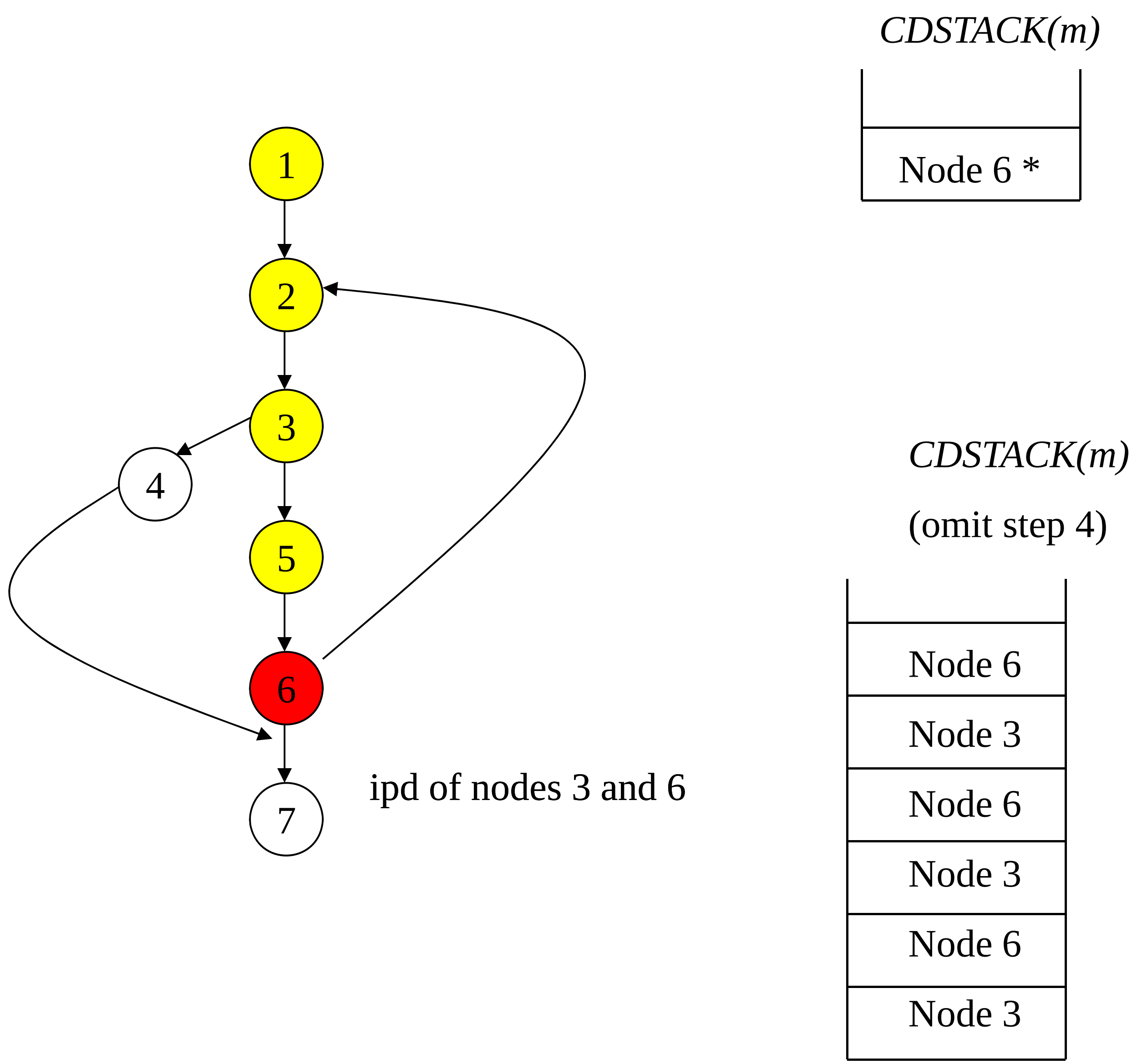
Node 3

Node 6

Node 3

Node 6

Node 3


CDSTACK(m)
Node 6 *
1
2
3
4
5
6
7
ipd of nodes 3 and 6
CDSTACK(m)
(omit step 4)
Node 6
Node 3
Node 6
Node 3
Node 6
Node 3

Dynamic data dependence

- Action *t*[*m*] is dynamically DD on action *s*[*k*] if *t*[*m*] uses variable/object last defined by *s*[*k*]

DIFA and slicing algorithms

Information Flow:

$$InfoFlow(t^m) = U(t^m) \cup \bigcup_{s^k \in DInfluence(t^m)} InfoFlow(s^k)$$

Dynamic Slicing:

$$DynSlice(t^m) = \{t\} \cup \bigcup_{s^k \in DInfluence(t^m)} DynSlice(s^k)$$

DInfluence(t_m) is the set actions that directly influence t_m through intra-procedural or inter-procedural data and control dependence.

U(t_m) is the set of variables used at t_m.

Compute InfoFlow and DynSlice (Action t^m)

$$DInfluence(t^m) = DDynDD(t^m) \cup DDynCD(t^m) \cup ReturnD(t^m) \cup ParamD(t^m) \cup InterprocCD(t^m)$$

$DynSlice(t^m) = \{t\}$
$InfoFlow(t^m) = U(t^m)$
for all $s^k \in DInfluence(t^m)$
  $InfoFlow(t^m) = InfoFlow(t^m) \cup InfoFlow(s^k)$
  $DynSlice(t^m) = DynSlice(t^m) \cup DynSlice(s^k)$
endfor

Store $InfoFlow(t^m)$ **for subsequent use**
Store $DynSlice(t^m)$ **for subsequent use**

if t^m **is a** ***sink*** **and** $InfoFlow(t^m)$ **contains sensitive sources then**
  Stop execution
  Log $InfoFlow(t^m)$ **and** $DynSlice(t^m)$

Handling implicit flows

- Problem: can’t be detected by dynamic mechanisms
- Solution: optional static transformation converts implicit flows to explicit ones
- Caveat: conservative analysis generates more false positives

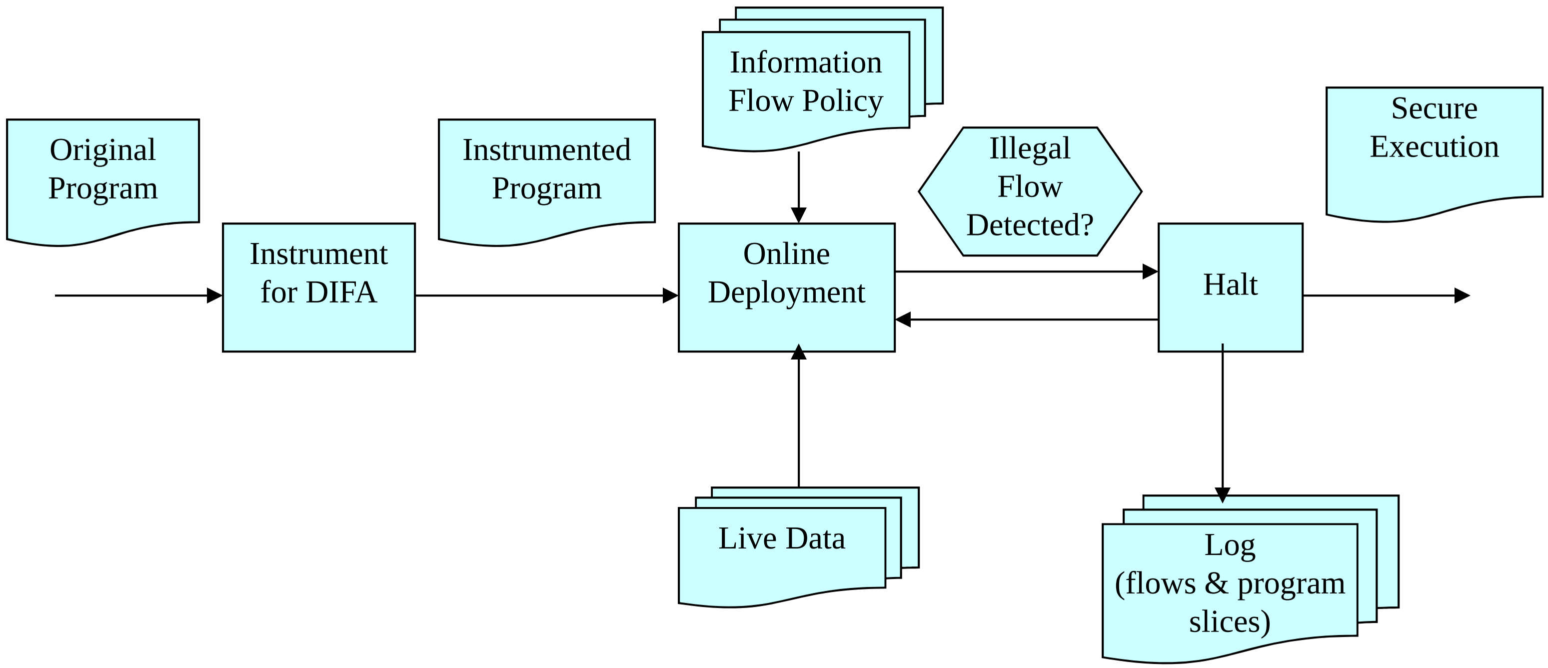
```
   x = 1;
S1 if (y == 0)
   {
     x = 0;    → explicit flow from y to x
    }
    // if y == 1 → implicit flow from y to x

   print(x);
```

```
   x = 1;
S1 if (y == 0)
   {
     x = 0;
   }
   addFlow(S1, x); // register influence of S1 on x

   print(x);
```

Prototype DIFA/Slicing tool

- Analyzes Java programs
- User defines information flow policy by identifying:
  - *sensitive* objects to monitor
  - *sink* objects at which flows are checked
- Two main components:
  - *Instrumenter*
  - *Profiler*
- Java byte code instrumented using Byte Code Engineering Library (BCEL)
- ~12,000 SLOC

Enforcing flow policies



Debugging insecure flows

- Insecure flows may be complex
- Dynamic slicing facilitates diagnosis

```
class DataH {
	public String getData() { return "some secure data"; }
}
class DataL {
	public String getData() { return "some public data"; }
}

public class DataMgr {
	DataH high = new DataH();
	DataL low = new DataL();

	String getData(boolean secureData) {
	L1:		if (secureData = true)		// BUG causing illegal flow
	L2:			return high.getData();
			else
	L3:			return low.getData();
	}

	public static void main(String[] args) {
			DataMgr dataMgr = new DataMgr();
			boolean secureData = false;
	L4:		broadcast(dataMgr.getData(secureData));		// outlet function
	}
}
```

Original Code

```
public String getData() { return "some secure data"; }
```

```
DataH high = new DataH();
```

```
L1:         if (secureData = true)
L2:                 return high.getData();
```

```
            DataMgr dataMgr = new DataMgr();
            boolean secureData = false;
L4:         broadcast(dataMgr.getData(secureData));
```

Dynamic Slice

Case studies

- Revealed *file disclosure* vulnerability in *DefaultServlet* of *Apache Tomcat*
- Revealed information leak in:
  - JConsole utility
  - “Pentagon” meeting scheduler
- Performance evaluated on 14 subject programs

DefaultServlet

- DIFA tool applied to *DefaultServlet* component of *Apache Tomcat*
  - Intended to serve public text files
  - Exhibits *file disclosure vulnerability*
  - Can be exploited to reveal JSP code
- Issued various requests, including ones for files in JSP directories
- Logged and inspected *all* flows into output methods
  - Flows occurred from 17 source objects per request
  - Slice comprised 45 byte code instructions
  - 4 source objects revealed pathname of requested file

Performance

- Substantial slowdown and storage impact
- Feasible to use in field only with non-processing-intensive applications
- Much room for improvement

Program	# lines	# callbacks	Time1 (secs)	Time2 (secs)	Time3 (secs)	Mem1 (KB)	Mem2 (KB)	Mem3 (KB)
JConsole	2,049	10,894	13	14	15	6,700	16,528	16,572
Meeting Scheduler	231	9,579	0.4	1.5	1.6	4,640	9,292	9,352
DefaultServlet	2,342	2,457	0.2	0.5	0.8	26,644	37,102	37,456
JarScan	884	274,332	2	12	25	5,840	10,300	10,420
Jally	2,309	900,045	21	35	50	12,736	24,272	24,420
JDictionary	13,423	90,688	16	22	26	24,580	52,240	54,060
JPassGen	6,673	30164	1.2	5.1	5.5	8,096	15,108	15,128
Jtidy	23,991	965,964	2	57	148	7,900	55,920	73,972
JackSum	4,017	297,365	3	11.6	11.8	5,248	12,664	12,896
IZPress	1,748	1,800,090	23	45	50	13,544	23,672	24,012
Diff	703	710,125	0.42	16	38	5,264	14,138	18,184
JavaTar	5,163	366,207	3.5	12.8	17.3	5,912	17,212	17,260
javap	1,382	11,099	0.75	3.4	3.5	7,026	20,492	20,504
javac	12,807	2,616,353	1.5	121	203	11,048	119,084	216,212

Related work

- Fenton (1974) – Data Mark Machine
- Denning (1975) – static flow certification
- Weiser (1984) first proposed static slicing
- Korel and Laski (1988) first proposed dynamic slicing
- Agrawal and Horgan (1990) used *Dynamic Dependence Graphs* and a reduced version of them
- Most proposed algorithms are based on *backward* analysis

Related work (2)

- Korel and Yalamanchili (1994) first proposed forward computing slicing
- Beszedes, Faragó et al (2002), proposed another forward computing slicing algorithm for C programs. Not precise. Statically determines the variables used by a given statement
- Zhang *et al* (2004) devided forward computing algorithm based on reduced order binary decision diagrams (roBDD), which exploits redundancy in slices improve space efficiency

Current and Future Work

- Further empirical evaluation
- Use of tool in
  - anomaly intrusion detection
  - test-case filtering
- Improvements
  - Eclipse integration
  - More precise analysis of implicit flows
  - Handling interprocedural and inter-thread control dependences