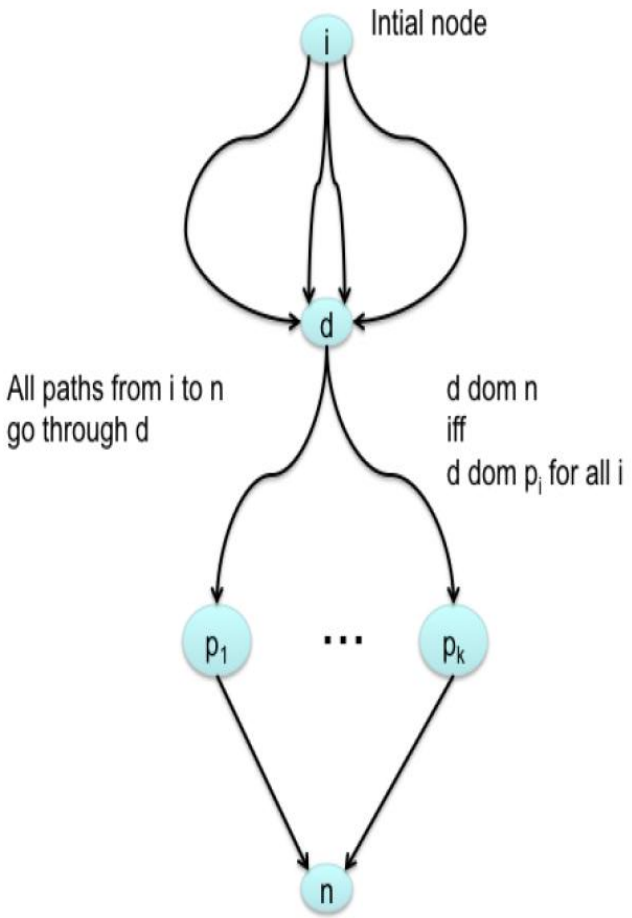
CC Lecture 9

Prepared for: 7th Sem, CE, DDU

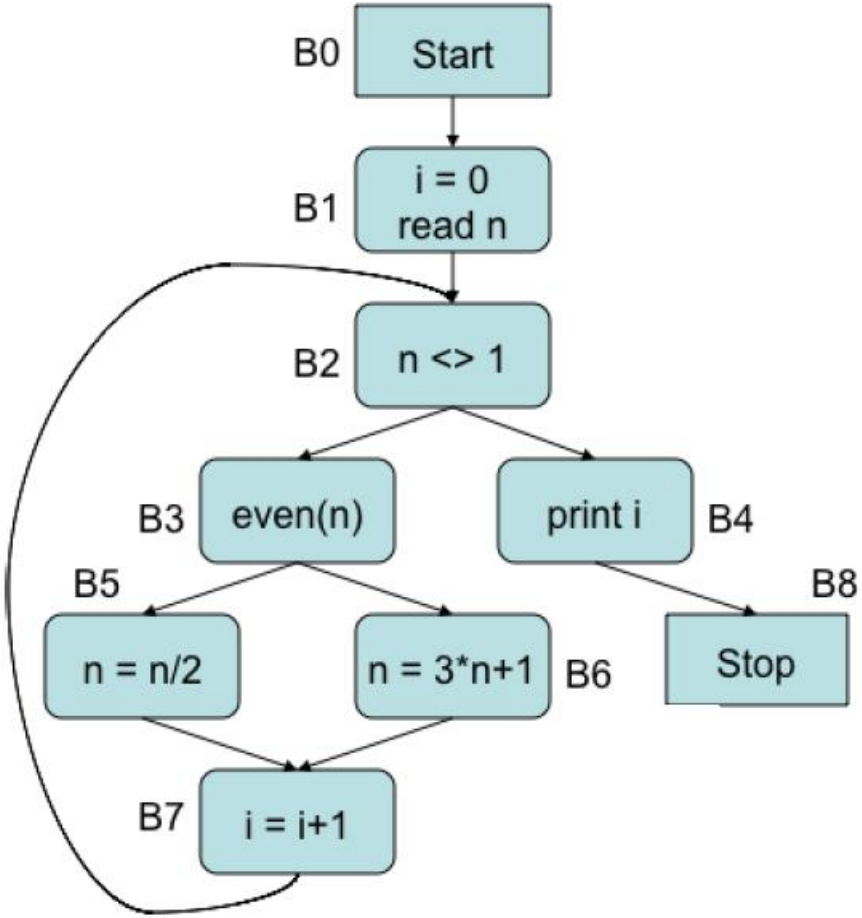
Prepared by: Niyati J. Buch

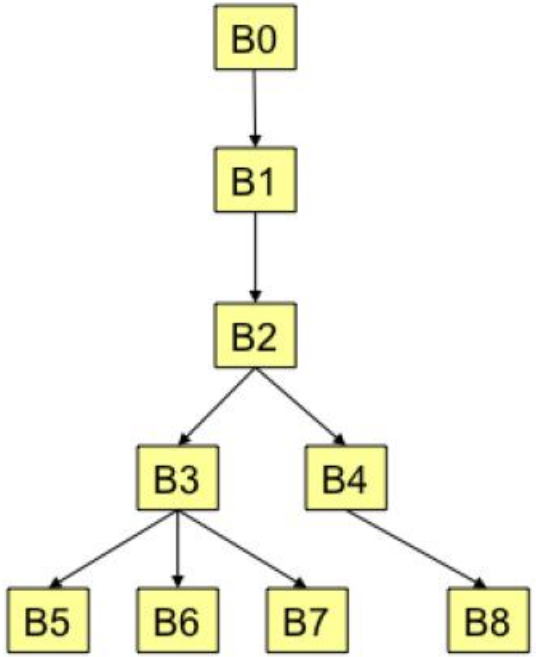
Dominators



- A node d in a flow graph is said to dominates node n, written ***d dom n***, if every path from the initial node of the flow graph to n goes through d.
- Initial node is the root, and each node dominates only its descendents in the dominator tree (including itself).
- The node x strictly dominates y, if x dominates y and $x \neq y$.
- x is the immediate dominator of y (denoted idom(y)), if x is the closest strict dominator of y.
- A **dominator tree** shows all the immediate dominator relationships not the transitive relationship.
- Principle of the dominator algorithm
  - If p1, p2, …, pk , are all the predecessors of n, and $d \neq n$, then d dom n, iff d dom pi for each i.

Dominator Example 1:





An algorithm for finding dominator:

$D(n) = OUT[n]$ for all n in N (the set of nodes in the flow graph), after the following algorithm terminates

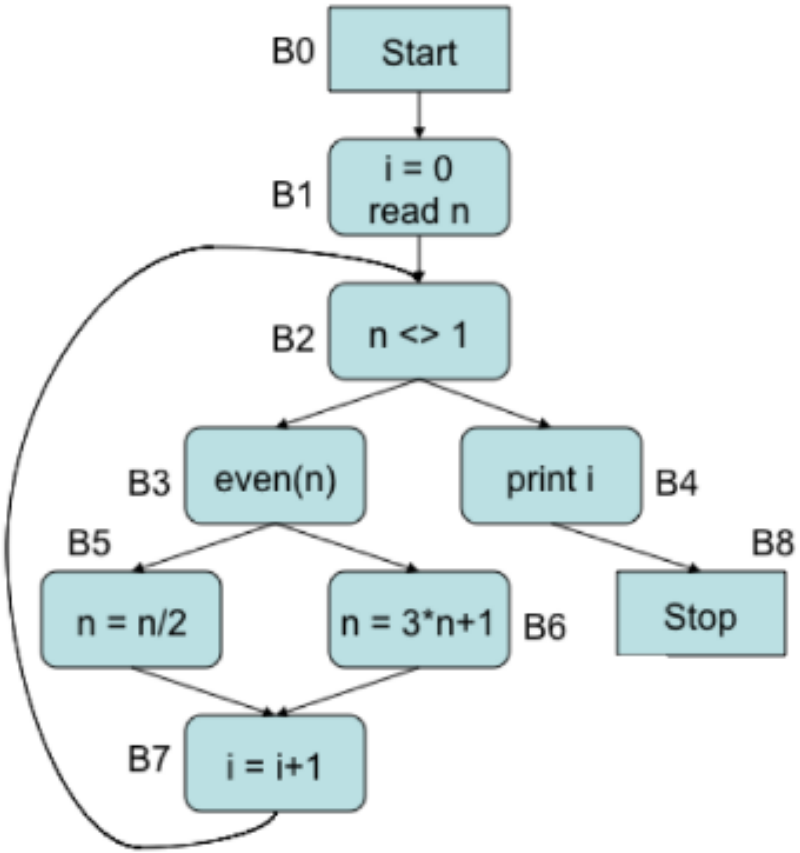
{ /* n_0 = initial node; N = set of all nodes; */
  $OUT[n_0] = \{n_0\}$;
  for n in $N - \{n_0\}$ do $OUT[n] = N$;
  while (*changes to any* $OUT[n]$ *or* $IN[n]$ *occur*) do
    for n in $N - \{n_0\}$ do

$$IN[n] = \bigcap_{P \text{ a predecessor of } n} OUT[P];$$

$$OUT[n] = \{n\} \cup IN[n]$$

}

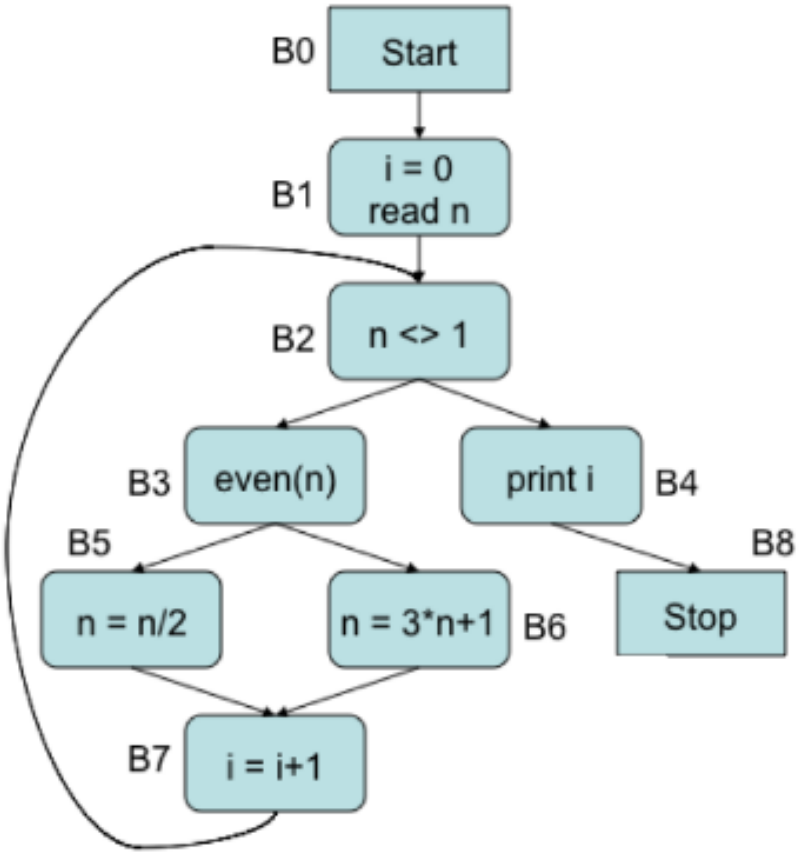
Applying the algorithm for finding dominator



- OUT[B0]={B0}
- OUT[B1] = OUT[B2] = OUT[B3] = OUT[B4] = OUT[B5] = OUT[B6] = OUT[B7] = OUT[B8] =

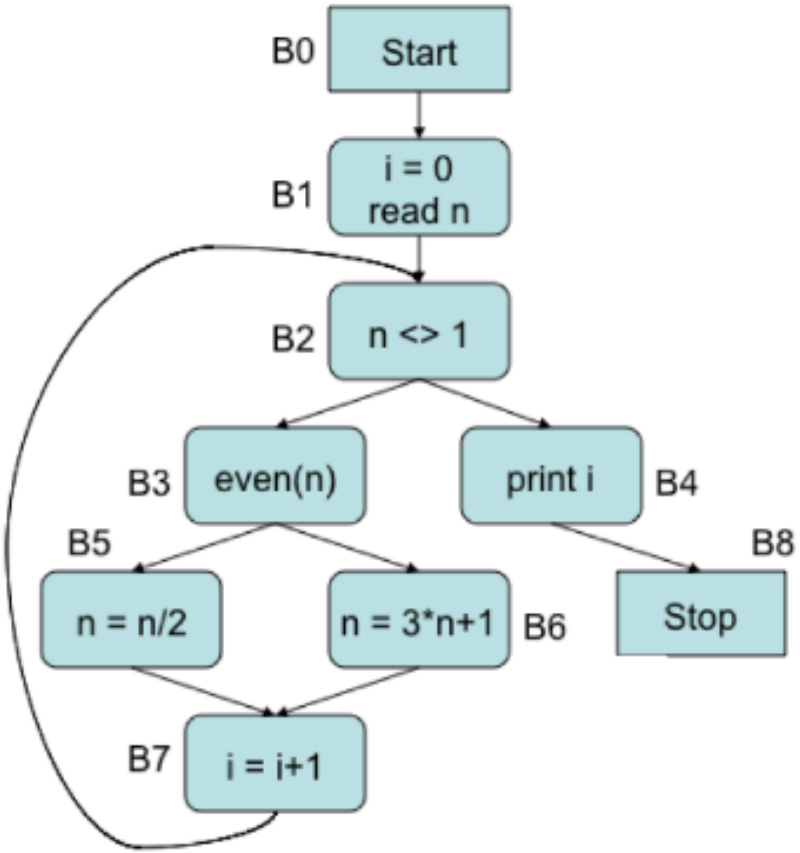
  {B0, B1, B2, B3, B4, B5, B6, B7, B8}

Applying the algorithm: B1



- IN[B1]
  = OUT[B0]
  = {B0}
- OUT[B1] = {B0, B1}

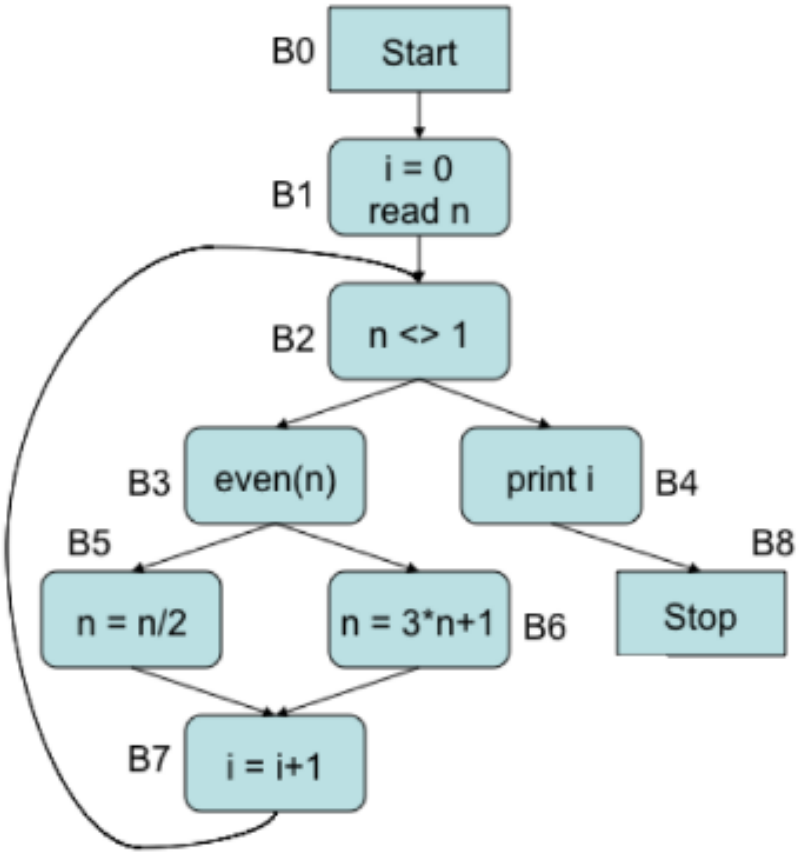
Applying the algorithm: B2



- IN[B2]
  = OUT[B1] ∩ OUT[B7]
  = {B0, B1}

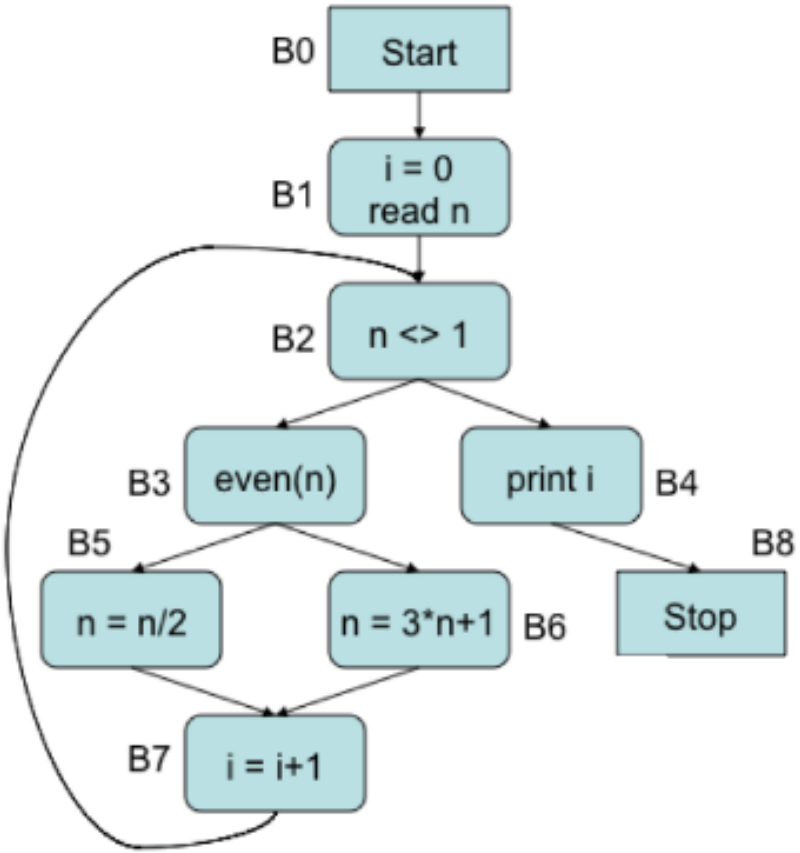
- OUT[B2] = {B0, B1, B2}

Applying the algorithm: B3



- IN[B3]
  = OUT[B2]
  = {B0, B1, B2}
- OUT[B3] = {B0, B1, B2, B3}

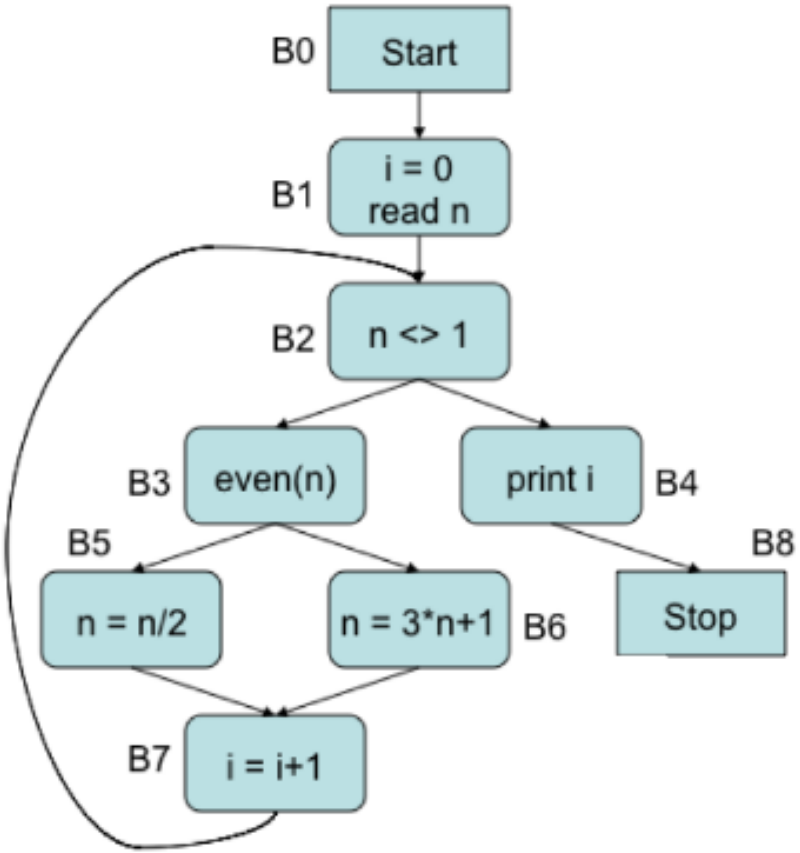
Applying the algorithm: B4



- IN[B4]
  = OUT[B2]
  = {B0, B1, B2}

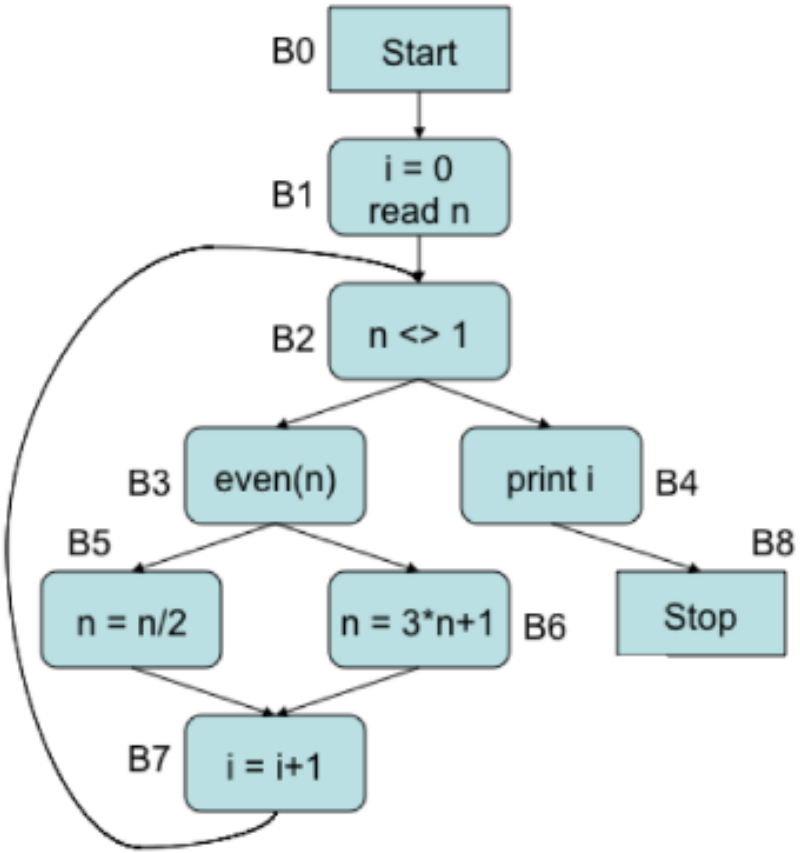
- OUT[B4] = {B0, B1, B2, B4}

Applying the algorithm: B5



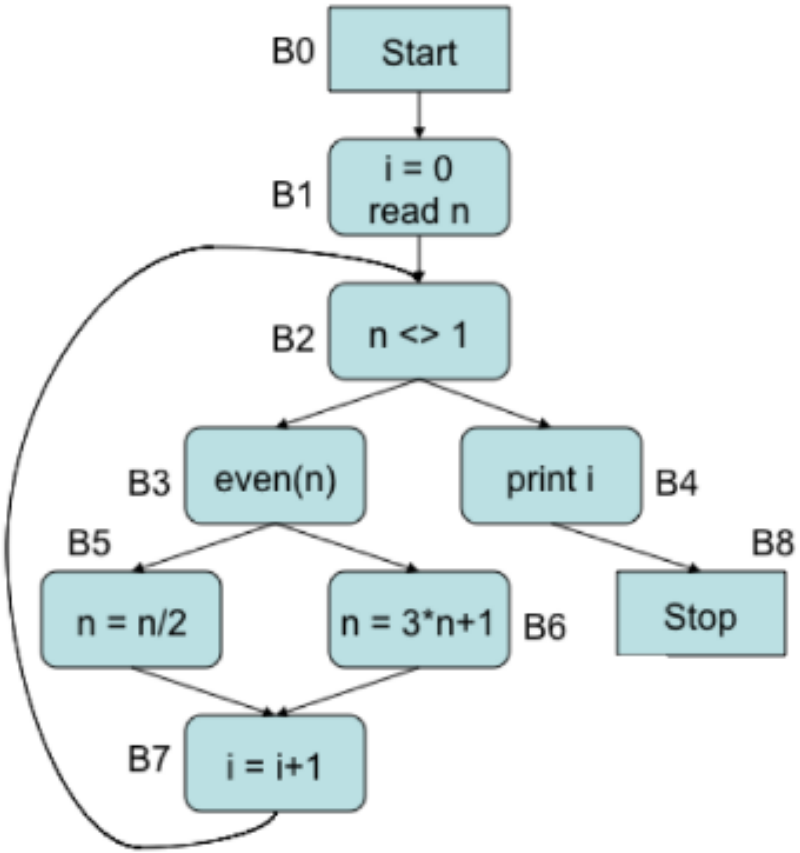
- IN[B5]
  = OUT[B3]
  = {B0, B1, B2, B3}
- OUT[B5] = {B0, B1, B2, B3, B5}

Applying the algorithm: B6



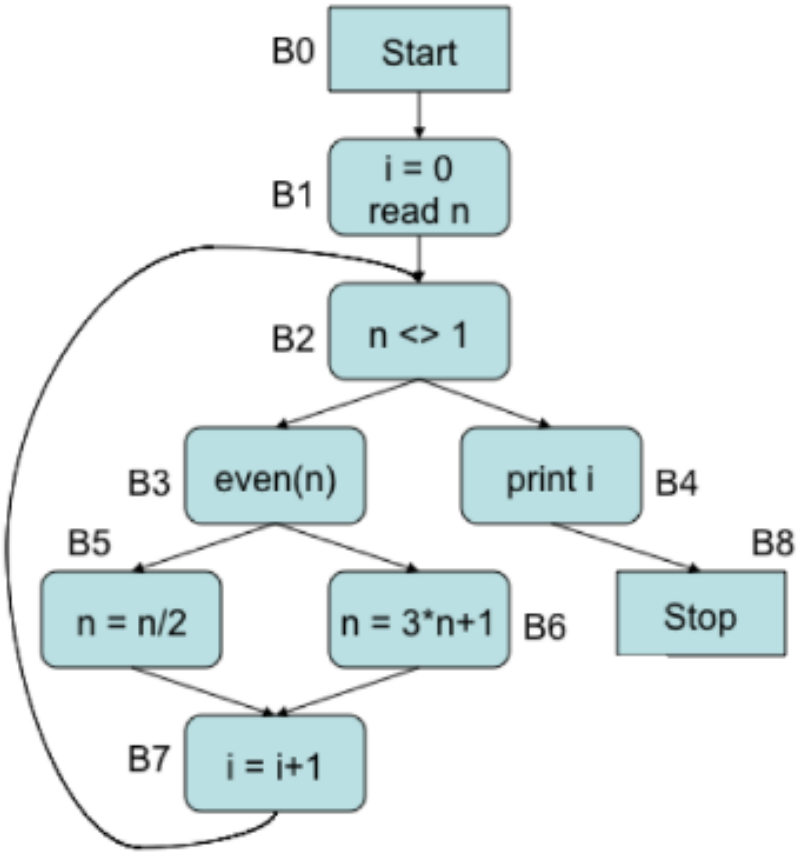
- IN[B6]
  = OUT[B3]
  = {B0, B1, B2, B3}
- OUT[B6] = {B0, B1, B2, B3, B6}

Applying the algorithm: B7



- IN[B7]
  = OUT[B5] ∩ OUT[B6]
  = {B0, B1, B2, B3}
- OUT[B7] = {B0, B1, B2, B3, B7}

Applying the algorithm: B8

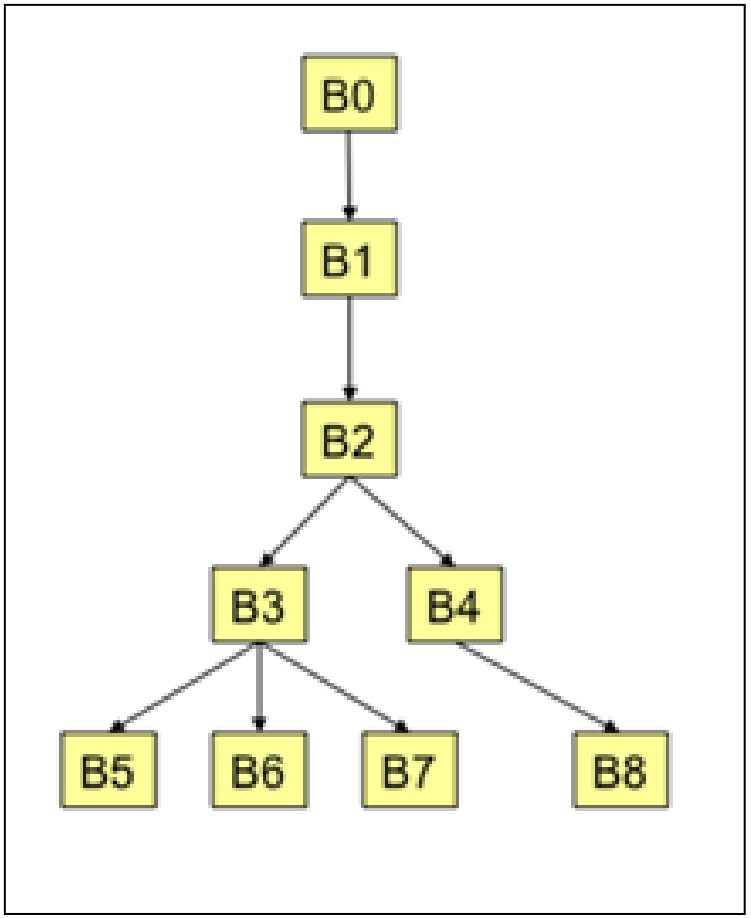


- IN[B8]
  = OUT[B4]
  = {B0, B1, B2, B4}

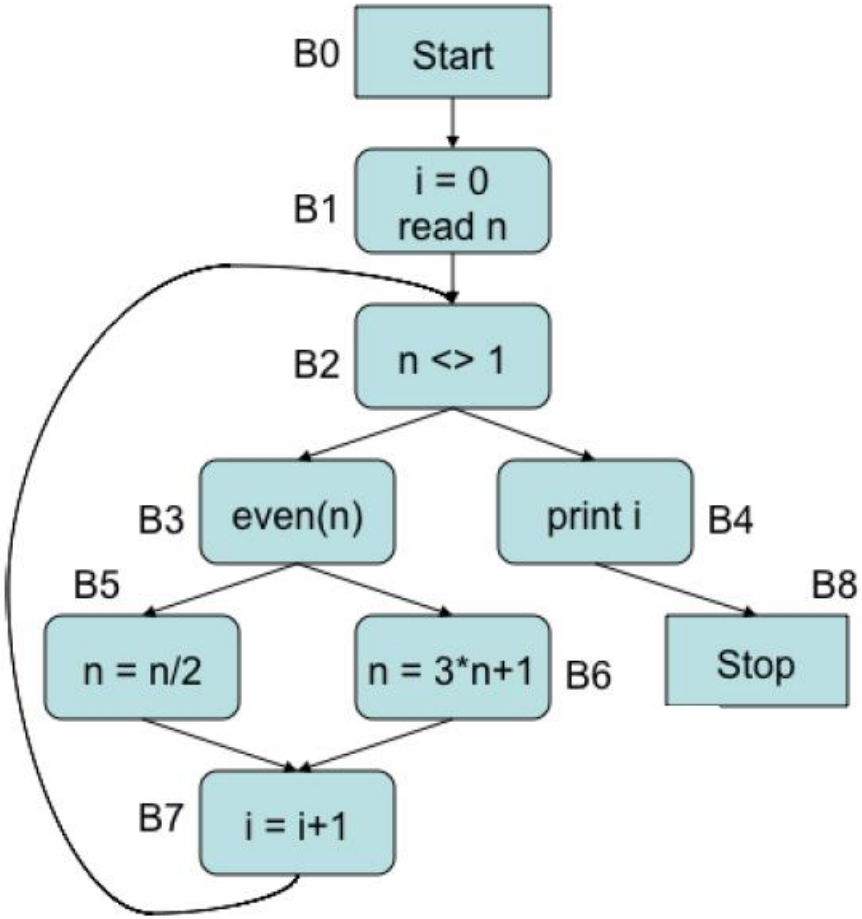
  B8 is last node so OUT[B8] not required

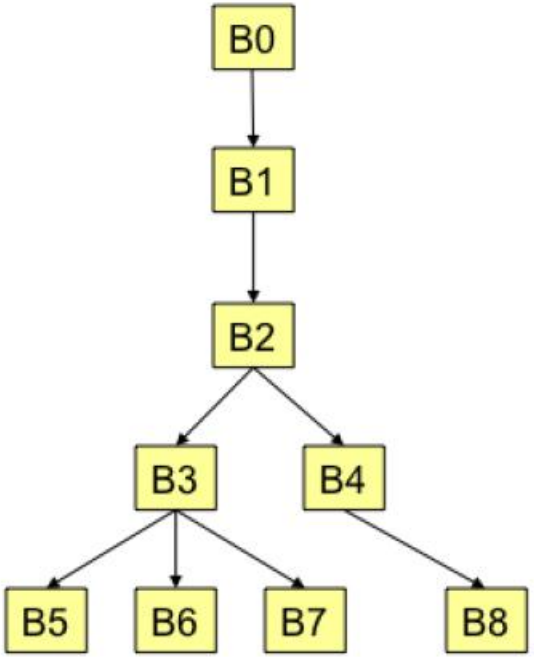
Construct Dominator Tree

- IN[B1] = {B0}
- IN[B2] = {B0, B1}
- IN[B3] = {B0, B1, B2}
- IN[B4] = {B0, B1, B2}
- IN[B5] = {B0, B1, B2, B3}
- IN[B6]= {B0, B1, B2, B3}
- IN[B7] = {B0, B1, B2, B3}
- IN[B8] = {B0, B1, B2, B4}

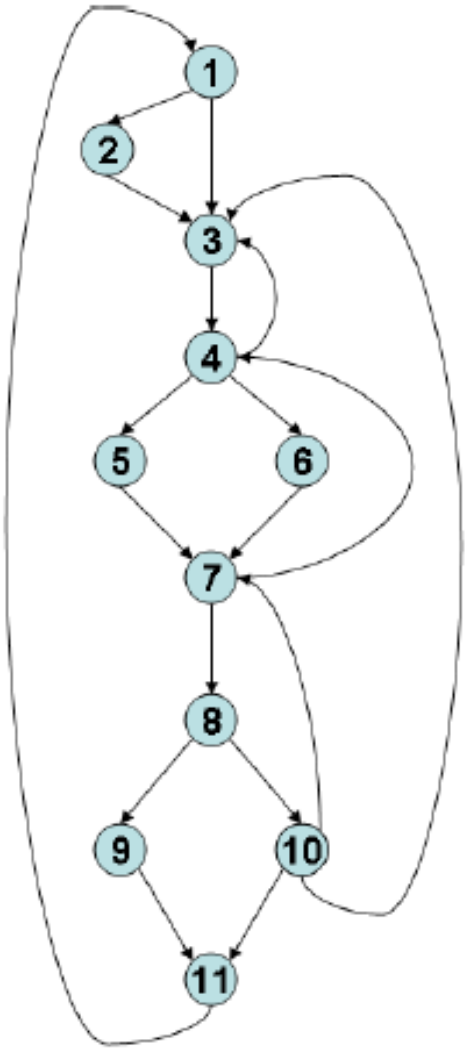


Finally, the dominator tree

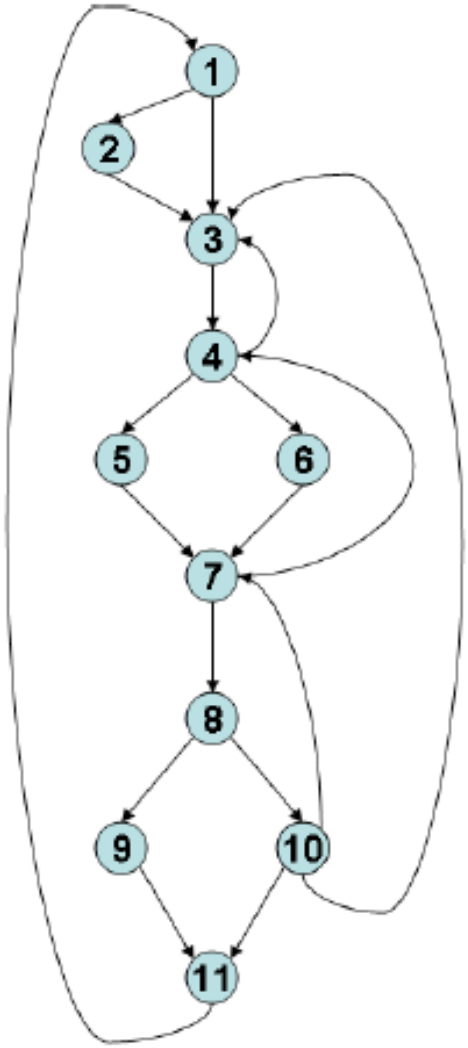


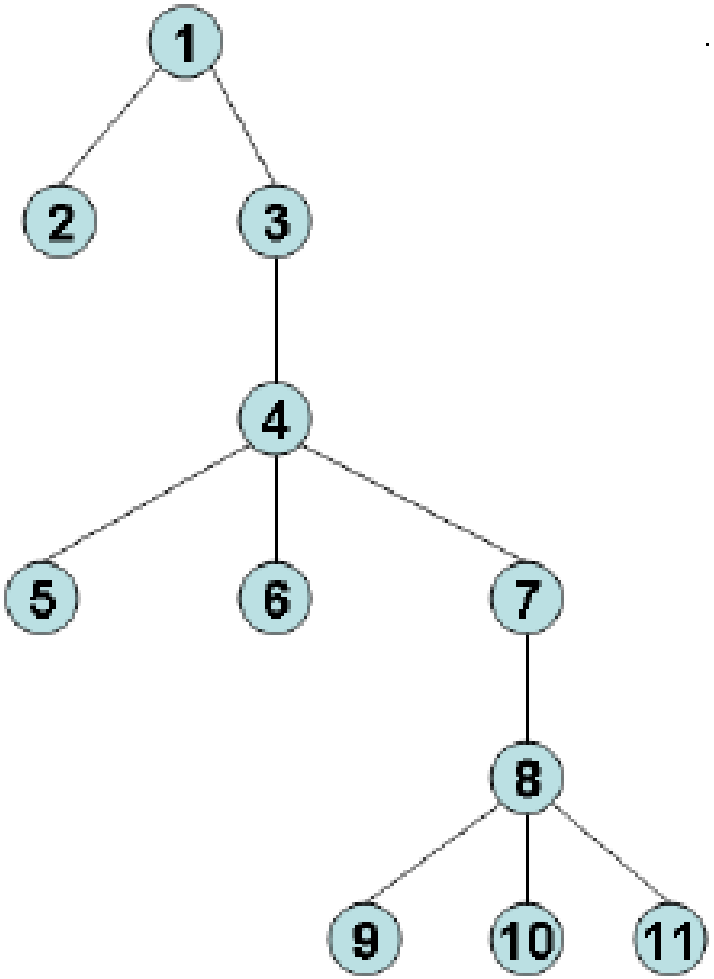


Example 2: Find dominator tree for given flow graph

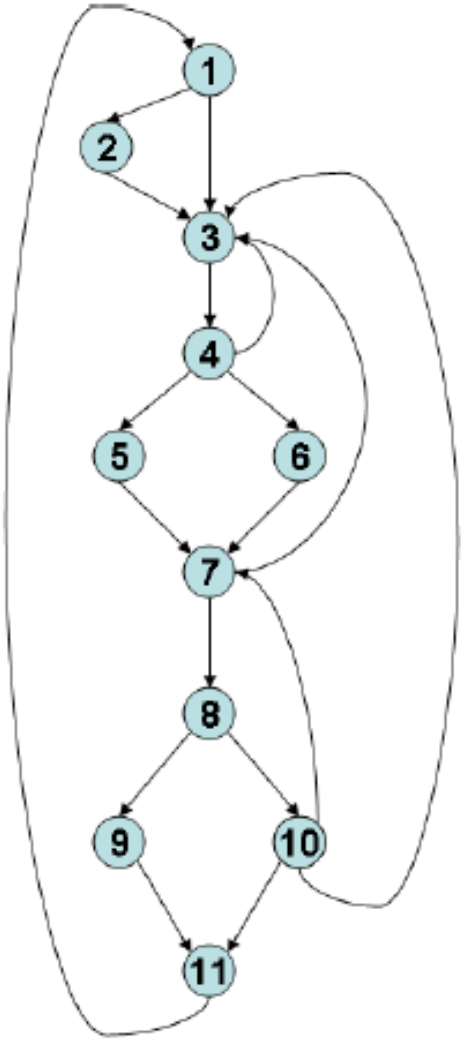


Example 2: Dominator tree for given flow graph

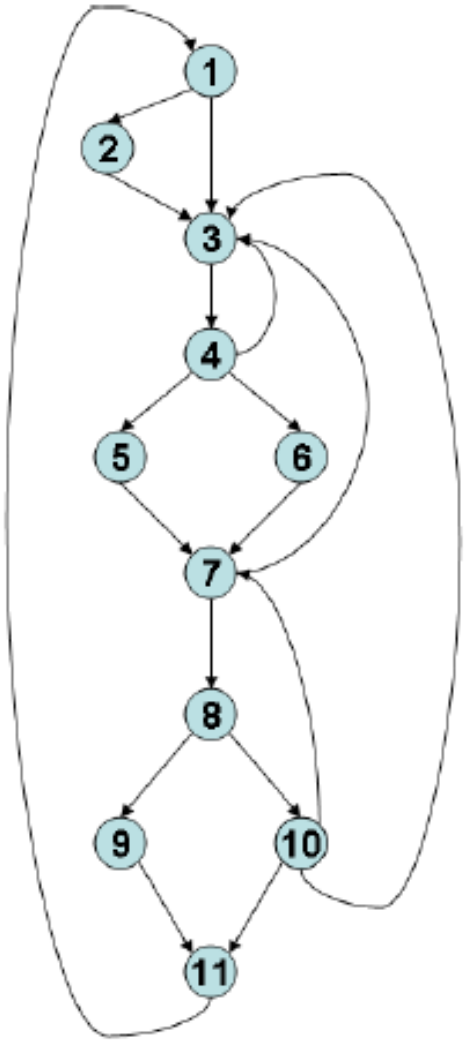


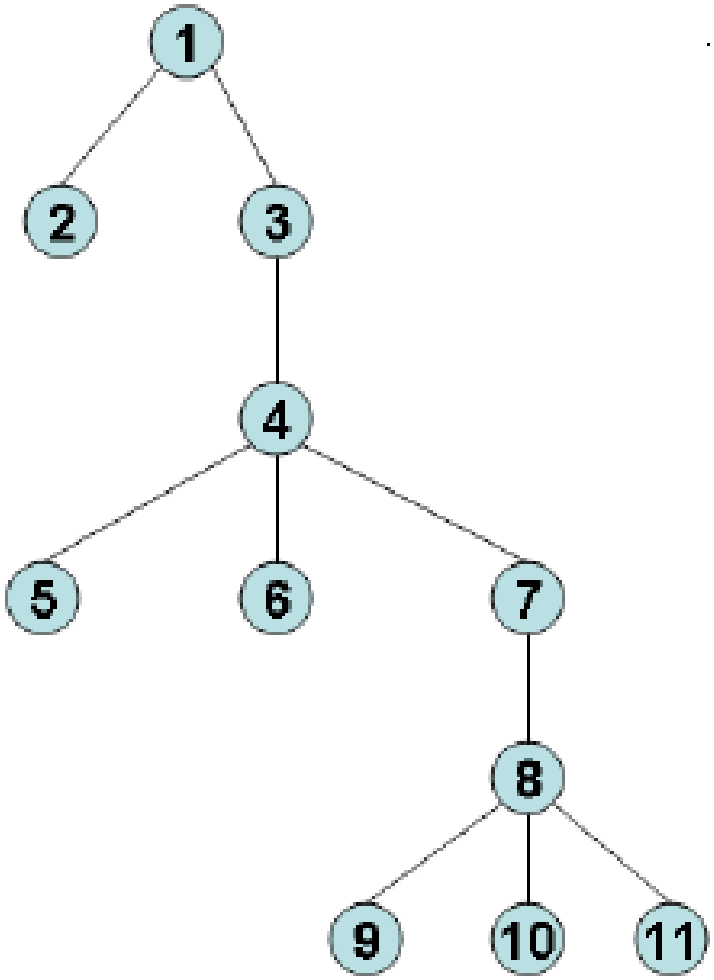


Example 3 : Find dominator tree for given flow graph



Example 3 : Dominator tree for given flow graph



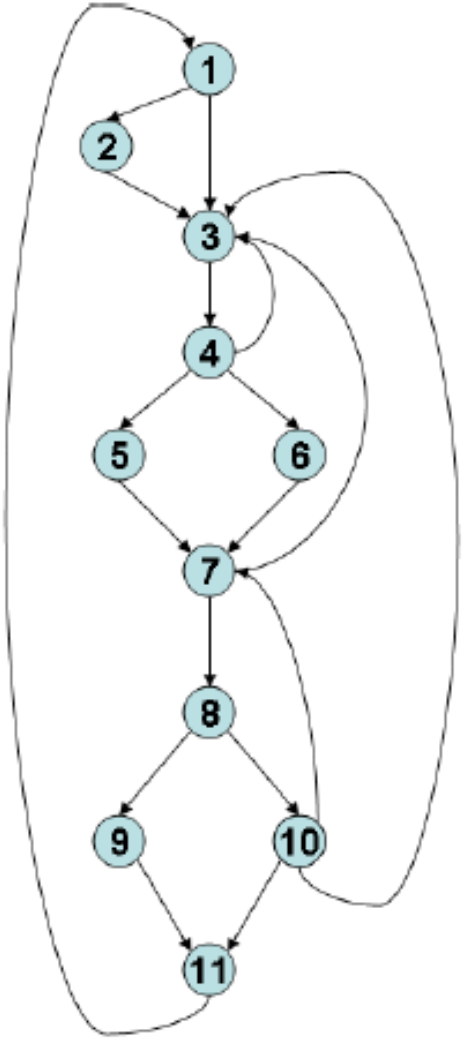


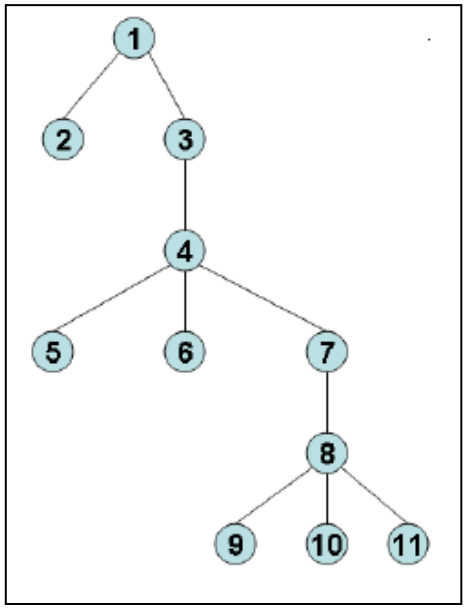
Natural loops and Back Edges

- Edges whose heads dominate their tails are called ***back edges***.

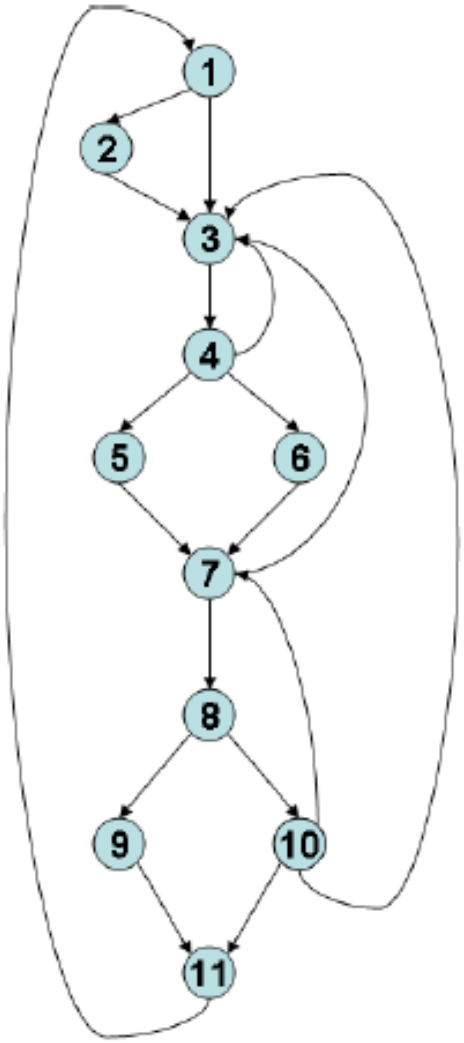
- For example, if there is an edge from a → b .
  - Then this b is called as the head and a is called as the tail.
  - So the head must dominate the tail that is called a back edge.

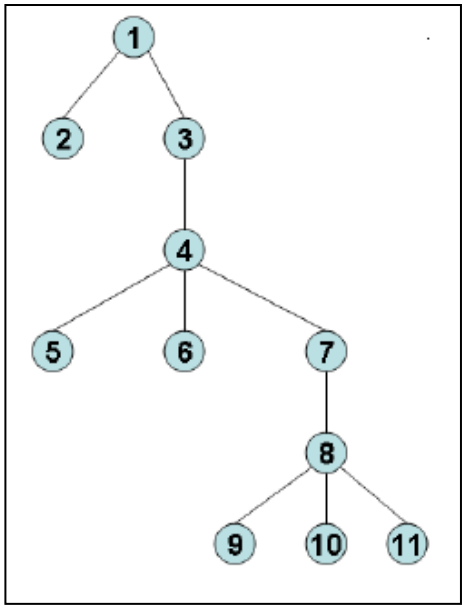
Find Back edges in the given graph





Back Edge

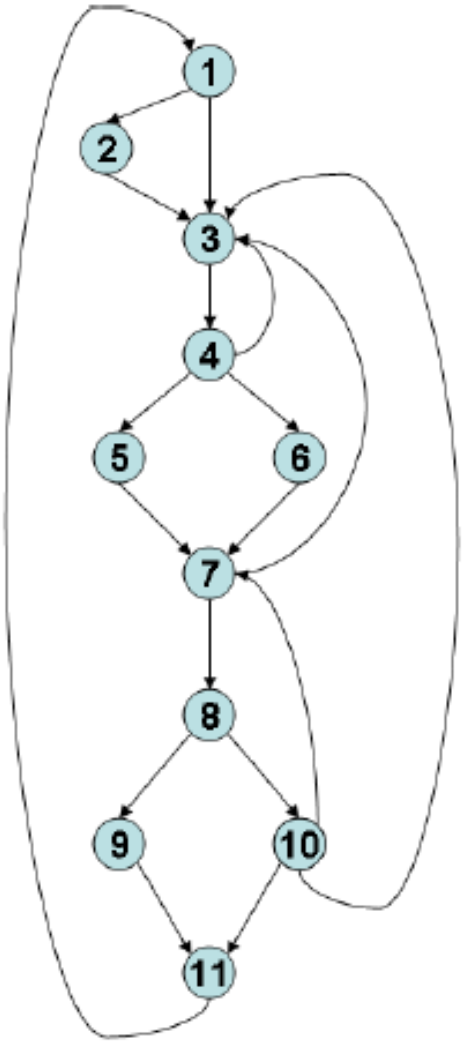


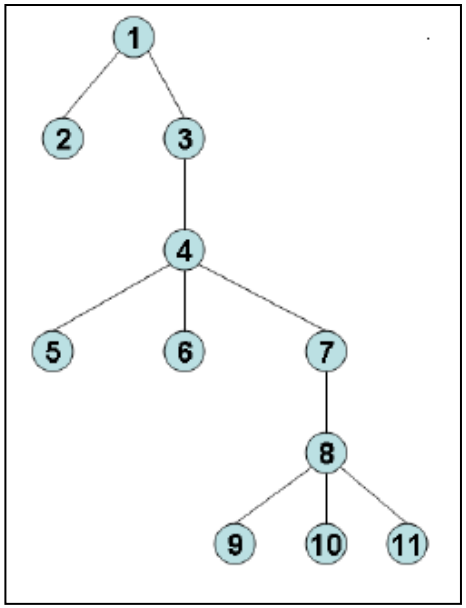


- Here is an edge from 4 to 3 and 3 dominates 4.
- Therefore, 4 to 3 is a back edge.

- Similarly, consider the edge from 10 to 7.
- 7 dominates node number 10 which is the tail.
- So, 10 to 7 is also another back edge.

Back Edge





- Now consider the edge from 7 to 3, 3 dominates 7.
- So, 7 to 3 is a back edge.
- For the edge from 10 to 3, 3 dominates 10.
- So 10 to 3 is a back edge.
- 11 to 1 is a back edge, as 1 dominates all the nodes.

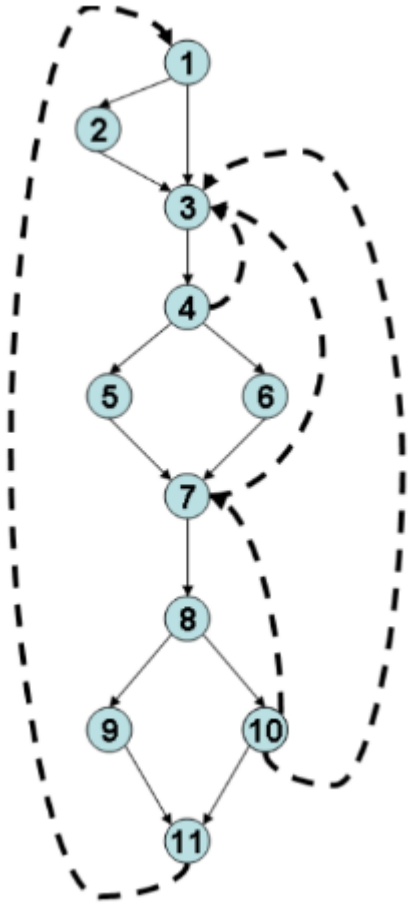
The natural loop of the edge

- Given a back edge n → d
  - The *natural loop* of the edge is d plus the set of nodes that can reach n without going through d
  - d is the header of the loop and the significance of the header is as follows:
    - A single entry point to the loop that dominates all nodes in the loop.
    - At least one path back to the header exists (so that the loop can be iterated).

Algorithm for finding Natural loop of a Back edge

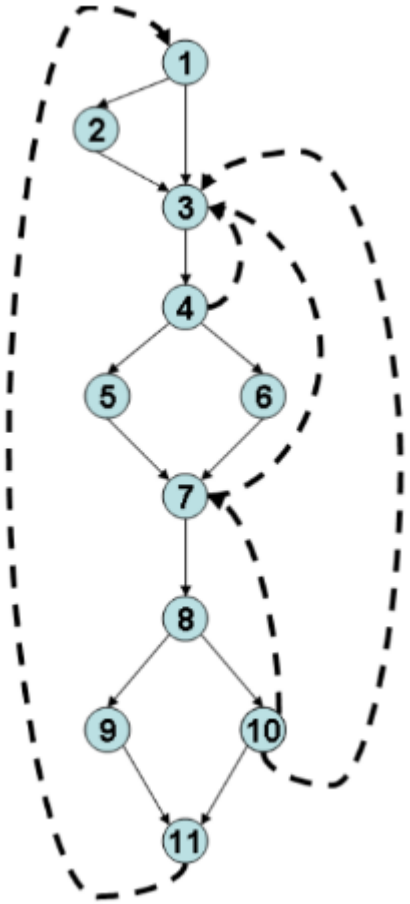
```
/* The back edge under consideration is n → d /*
{    stack = empty;
     loop = {d};
     /* This ensures that we do not look at predecessors of d */
     insert(n);
     while (stack is not empty) do {
          pop(m, stack);
          for each predecessor p of m do insert(p);
     }
}
procedure insert(m) {
     if m ∉ loop then {
          loop = loop U {m};
          push(m, stack);
     }
}
```

Back Edges for the given graph



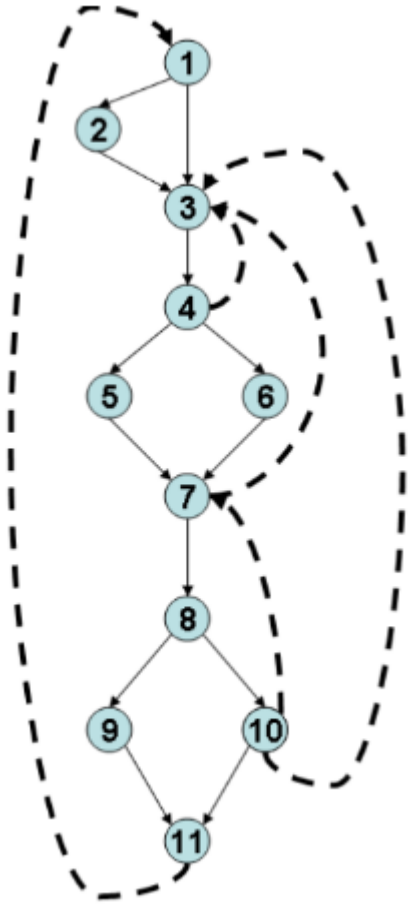
1. 4 → 3
2. 10 → 7
3. 7 →3
4. 10 → 3
5. 11 → 1

Find natural loop for the back edges



1. 4 → 3
2. 10 → 7
3. 7 →3
4. 10 → 3
5. 11 → 1

Natural loops for back edges for the given graph



- 4 → 3

  Natural loop={3,4}
- 10 → 7

  Natural loop={7,8,10}
- 7 →3

  Natural loop={3,4,5,6,7,8,10}
- 10 → 3

  Natural loop={3,4,5,6,7,8,10}
- 11 → 1

  Natural loop={1,2,3,4,5,6,7,8,9,10,11}

Note:

- The algorithm for finding a natural loop does not say how to find dominators.
- The algorithm for finding dominators does not say in which order to visit the nodes.

- Think…. Visiting the nodes in which order tends to find the dominators and expose the back edges faster???

Answer

Q. Visiting the nodes in which order tends to find the dominators and expose the back edges faster???

A. Depth First Order

Purpose of finding loops

- Loop optimization is most valuable machine-independent optimization because program's inner loop takes bulk to time of a programmer.
- If we decrease the number of instructions in an inner loop then the running time of a program may be improved even if we increase the amount of code outside that loop.
- For loop optimization the following three techniques are important:
  1. Code motion
  2. Induction-variable elimination
  3. Strength reduction

Code Motion

- Code motion is used to decrease the amount of code in loop.
- This transformation takes a statement or expression which can be moved outside the loop body without affecting the semantics of the program.

```
while (i<=limit-2)  /*statement does not change limit*/
```

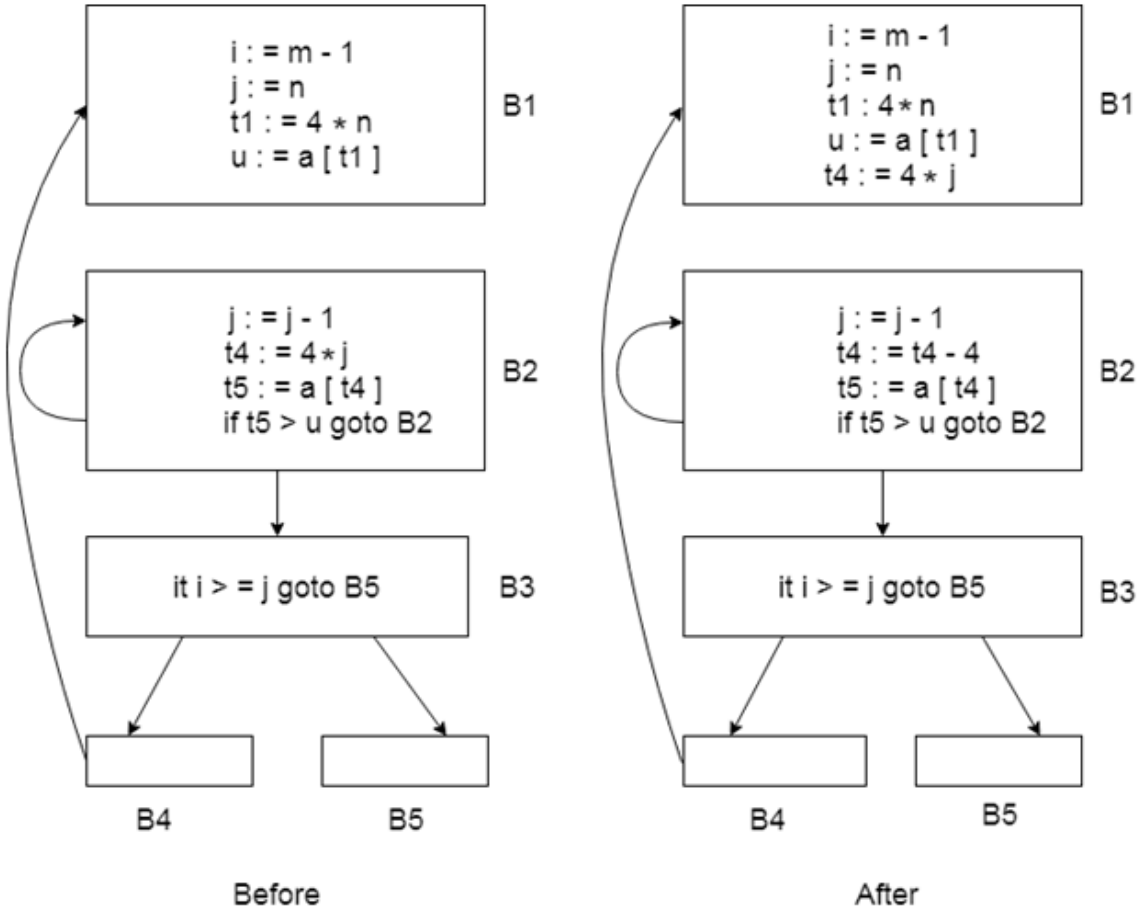
- After code motion the result is as follows:

```
a= limit-2;
while(i<=a)    /*statement does not change limit or a*/
```

Induction-Variable Elimination

- Induction variable elimination is used to replace variable from inner loop.
- It can reduce the number of additions in a loop. It improves both code space and run time performance.

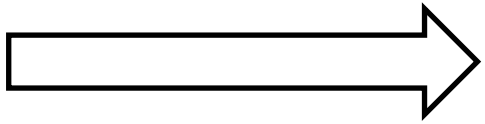
In this figure, we can replace the assignment t4:=4*j by t4:=t4-4. The only problem which will be arose that t4 does not have a value when we enter block B2 for the first time. So we place a relation t4=4*j on entry to the block B2.



Reduction in Strength

- Strength reduction is used to replace the expensive operation by the cheaper once on the target machine.
- Addition of a constant is cheaper than a multiplication. So we can replace multiplication with an addition within the loop.
- Multiplication is cheaper than exponentiation. So we can replace exponentiation with multiplication within the loop.

```
while (i<10)
      {
j= 3 * i+1;
a[j]=a[j]-2;
i=i+2;
      }
```



```
s= 3*i+1;
    while (i<10)
    {
        j=s;
        a[j]= a[j]-2;
        i=i+2;
        s=s+6;
    }
```

Here, it is cheaper to compute s=s+6 than j=3 *i