merge (SLL<T...> other) { // merge of merge sort

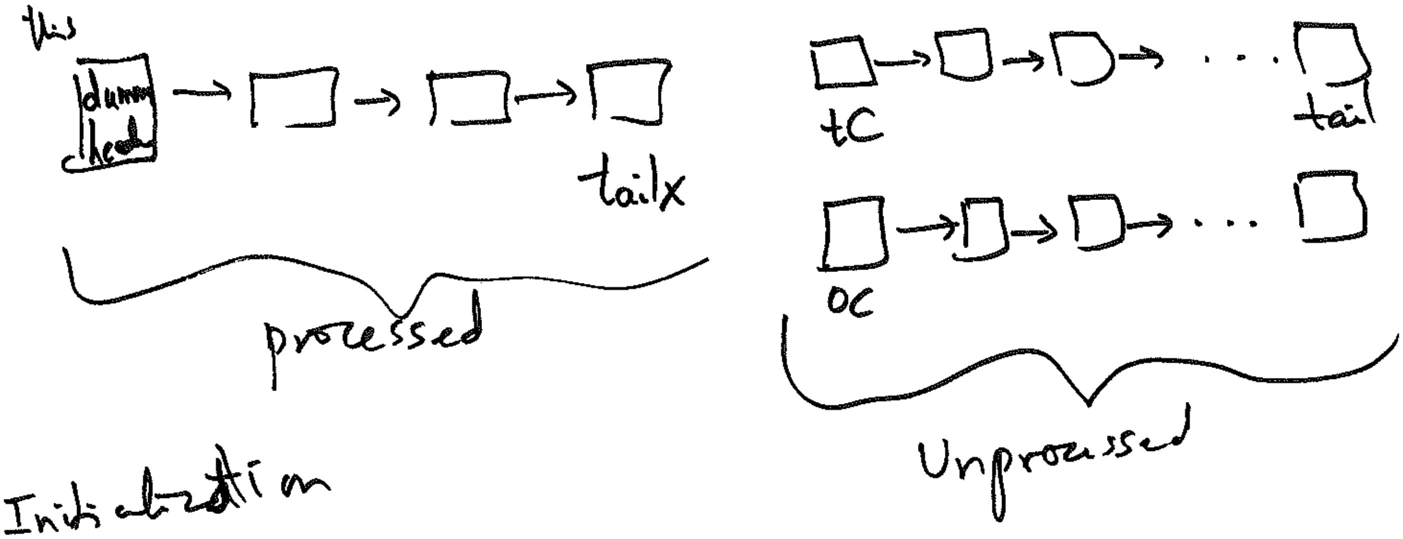
// Loop invariant: tC: this list's cursor

oC: other list's cursor

// this.head ... ~~tailx~~ tC.prev - processed

// other.head.next ... oC.prev - processed

tC, oC } next to be processed.



// Initialization

tailx ← this.head

tC ← this.head.next

oC ← other.head.next

// main loop

while (tC ≠ null and oC ≠ null) do

if tC.element ≤ oC.element then

tailx.next ← tC tailx ← tC tC ← tC.ne

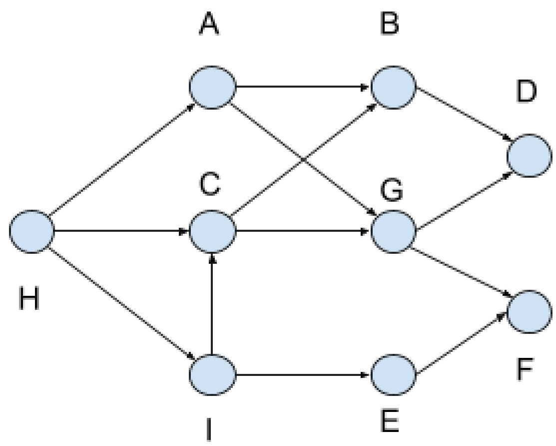
else tailx.next ← oC tailx ← oC oC ← oC.ne

if tC = null then {tailx.next ← oC ; tail ← other.t

else {tailx.next ← tC}

Applications of Depth-First Search (DFS)

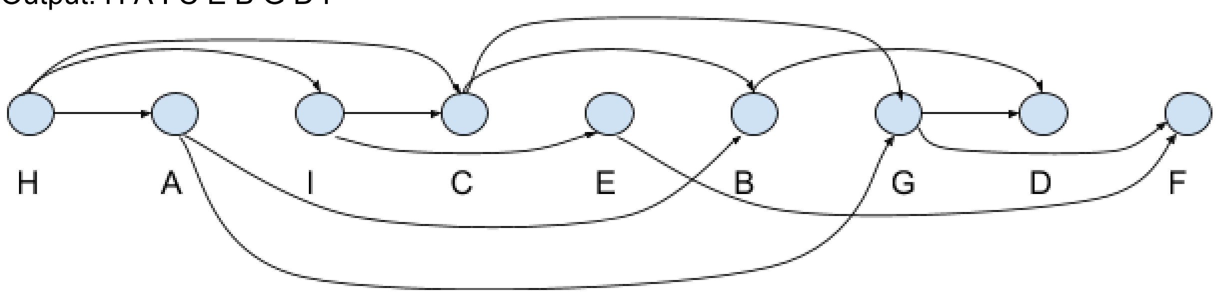
Topological order: linear ordering of vertices of a directed graph, such that all edges of G go from left to right. Only DAGs (directed, acyclic graphs) have topological orders.



求拓扑算法1

Algorithm 1: Repeat: remove node with no incoming edge, with its outgoing edges.

Output: H A I C E B G D F




```
// Input: Directed graph G=(V,E).  Output: list of vertices of G, in topological order
topologicalOrder1(G=(V,E)):
topNum ← 0
q ← new Queue(Vertex)              每次把当前入度为0的节点全放进Queue中，然后每次remove取出来放进
                                   最终结果list:topList
topList ← new List(Vertex)
for u in V do
   u.inDegree ← u.revAdj.size()    revAdj表示指向某点的点有哪些
   if u.inDegree = 0 then q.add(u) 把一开始所有入度为0的点加入queue    如H
while q is not empty do
   u ← q.remove()
   u.top ← ++topNum  标记每个vertex在拓扑序列中处于第几个
   topList.add(u)
   for each edge (u,v) going out of u do
         v.inDegree--
         if v.inDegree = 0 then q.add(v)
if topNum != |V| then raise exception "Not a DAG"    最后得到的序列覆盖不了所有的点
return topList
```

求拓扑算法2

Algorithm 2: Order nodes by decreasing finish times of DFS. Output:: H I E C A G F B D

<table>
<tr>
<td>

topologicalOrder(g)
it ← g.iterator()
DFS(it)
return decFinList

DFS(it)
topNum ← g.size()
time ← 0
cno ← 0
decFinList ← new Linked List of vertices
for u in V do u.seen ← false
while it.hasNext() do
    u ← it.next()
    if ! u.seen then
        cno++
        DFSVisit(u)

</td>
<td>

DFSVisit(u)
u.seen ← true
u.dis ← ++time
u.cno ← cno
for each edge (u,v) going out of u do 设置该点的各个出度目标点的状态
    if ! v.seen then
        v.parent ← u
        DFSVisit(v)
u.fin ← ++time
u.top ← topNum--
decFinList.addFirst(u) 头插法

</td>
</tr>
</table>

出度指向下一个点 结束时间：其所有出度节点都访问完 拓扑序列

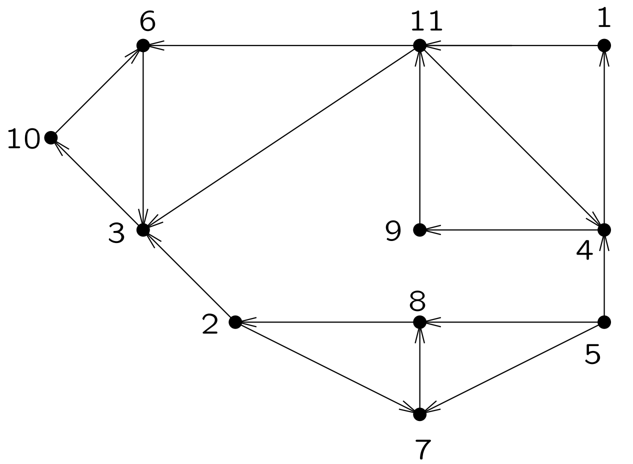
Vertex	Neighbors	dis 到底是干嘛的	fin	parent 入度节点	top
A	B G	1	10	--	5
B	D	2	5	A	8
C	B G	11	12	--	4
D	--	3	4	B	9
E	F	13	14	--	3
F	--	7	8	G	7
G	D F	6	9	A	6
H	A C I	15	18	--	1
I	C E	16	17	H	2

Strongly connected components:
Input: Directed graph G=(V,E). Any two nodes u and v are **strongly connected** if G has paths from u to v, and, from v to u. **Strongly connected components**: Partition of V into subsets of nodes such that within each subset, its nodes are strongly connected to each other.

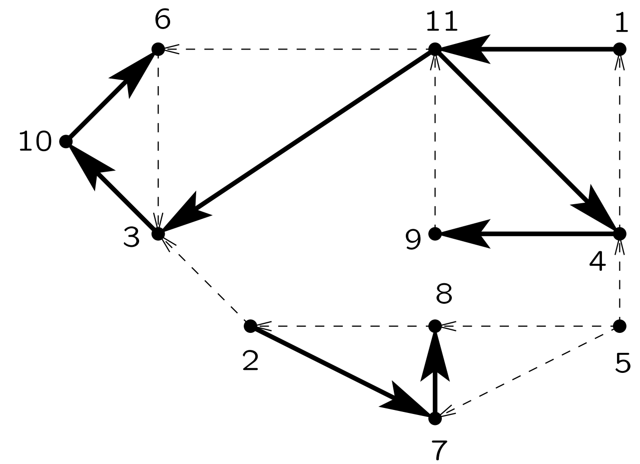
Algorithm scc(G=(V,E)):

Run DFS(G) to find finish time order
Reverse the edges of G (exchange adj and revAdj of each Vertex)
Run DFS again, going through nodes in decreasing finish time order of first DFS, by using “it ← decFinList.iterator()” instead of “it ← g.iterator()”.

Example: Input graph for SCC

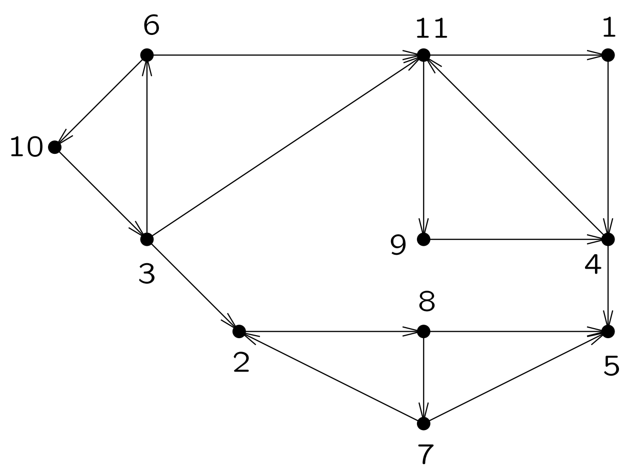


First DFS on input graph

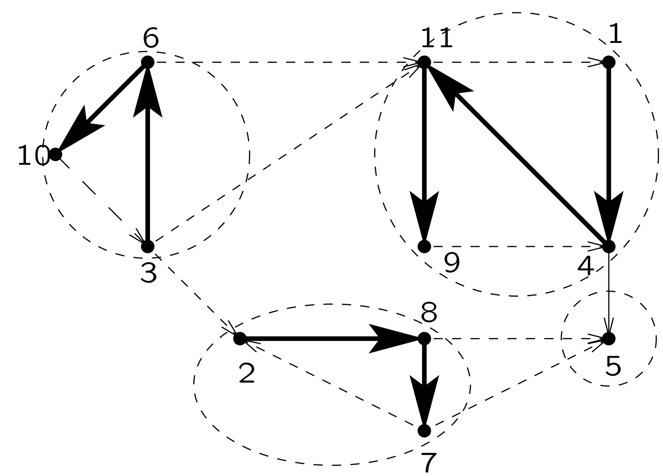


DFS finish-time order: 9, 4, 6, 10, 3, 11, 1, 8, 7, 2, 5.

Reverse of given graph, G^R



Second DFS on G^R



SCC: $\{\{5\}, \{2, 7, 8\}, \{1, 4, 9, 11\}, \{3, 6, 10\}\}$.