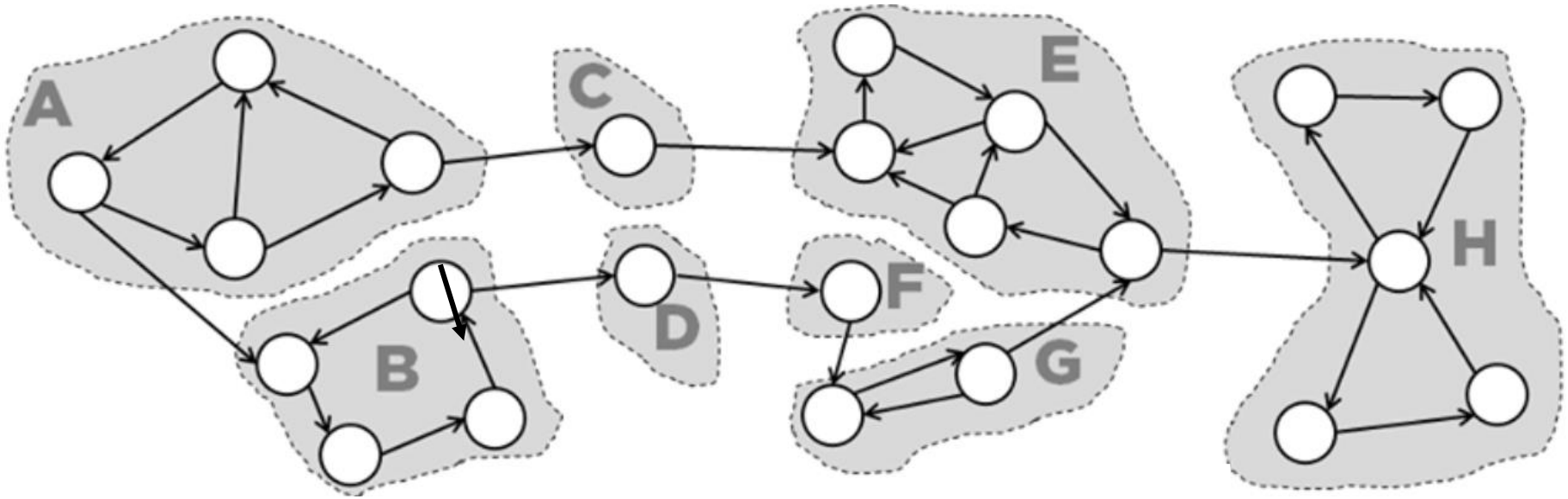
Strongly Connected Components with DFS

- Tarjan 1972 (wikipedia)
- Performs a single DFS with an additional **S**tack

Strongly connected components (SCC)

Def: Equivalence classes of the relation (u, v) iff there is a path from u to v and a path from v to u



Goal: Compute for each vertex the number of its component

Strongly connected components (SCC)

Obs: Let u and v be in a SC C. Let P be a path from u to v then every vertex on P is also in C

```
DFS(G)
For each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
For each vertex u ∈ G.V
    if u.color == WHITE
        DFS-Visit(G,u)
```

```
DFS-Visit(G,u)
time = time + 1
u.d = time
u.color = GRAY
For each vertex v ∈ G.Adj[u]
    if v.color == WHITE
        v.π = u
        DFS-Visit(G,v)
u.color = BLACK
time = time + 1
u.f = time
```

Thm 2: $[v.d, v.f] \subset [u.d, u.f]$ iff v is a descendant of u in the DFS-forest, $[v.d, v.f] \cap [u.d, u.f] = \emptyset$ iff u and v are unrelated in the DFS-forest

```
DFS(G)
For each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
For each vertex u ∈ G.V
    if u.color == WHITE
        DFS-Visit(G,u)
```

```
DFS-Visit(G,u)
time = time + 1
u.d = time
u.color = GRAY
For each vertex v ∈ G.Adj[u]
    if v.color == WHITE
        v.π = u
        DFS-Visit(G,v)
u.color = BLACK
time = time + 1
u.f = time
```

Thm 3: v is a descendant of u in the DFS-forest iff when we invoke DFS-Visit(u) there is a white path from u to v

Edge classification

- Tree edges: (u,v), DFS-Visit(v) was called from DFS-visit(u)
- Back edges: (u,v) such that v is an ancestor of u in the DFS-forest
- Forward edges: nontree edges (u,v) such that v is a descendant of u
- Cross edges: (u,v) such that $v.f < u.d$

Observe, among non-tree edges:

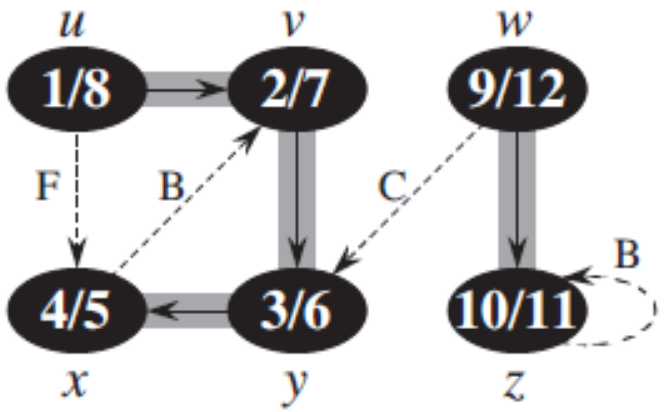
1. (Only) backward edges go to a vertex with a later finish time
2. (Only) forward edges go to a vertex with later discovery time

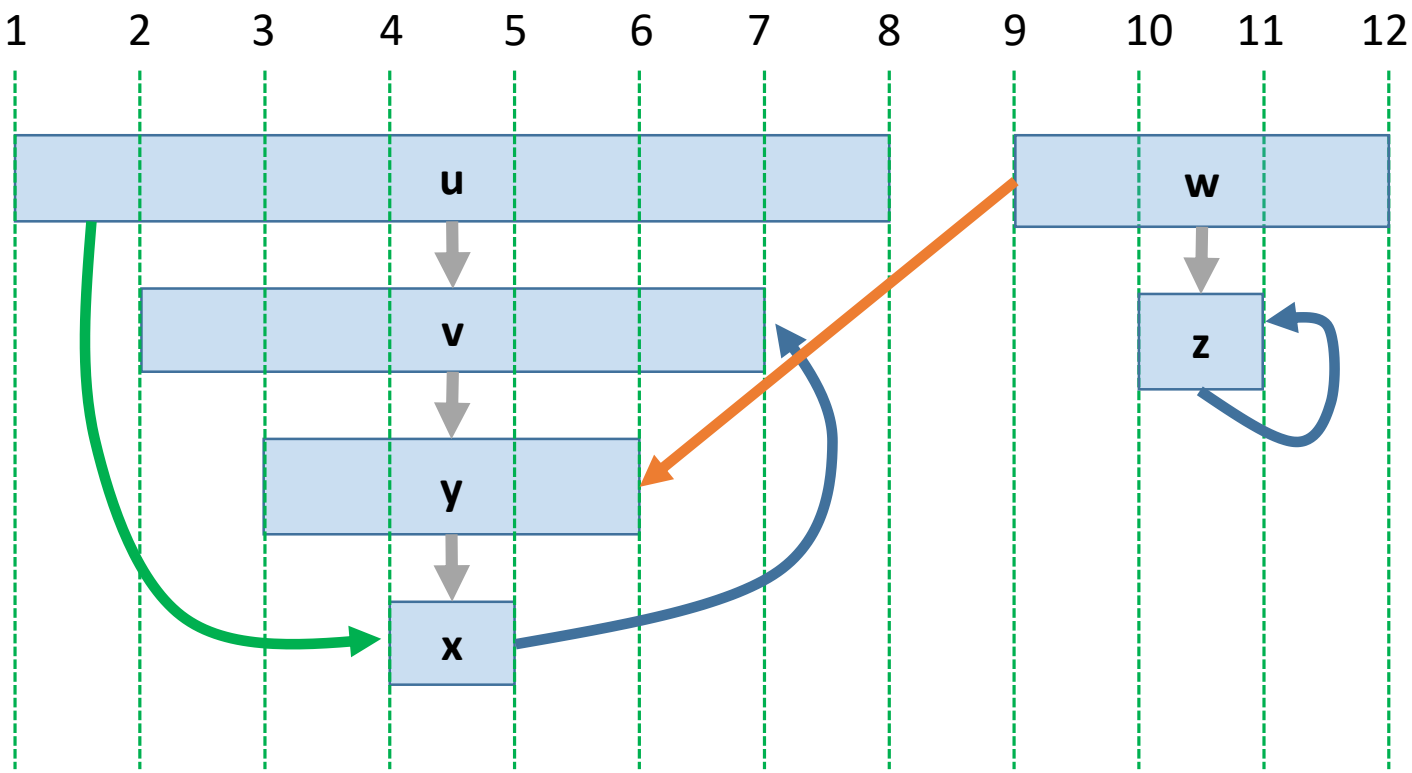
Edge classification

- Tree edges: (u,v), DFS-Visit(v) was called from DFS-visit(u)
- Back edges: (u,v) such that v is an ancestor of u in the DFS-forest
- Forward edges: nontree edges (u,v) such that v is a descendant of u
- Cross edges: (u,v) such that $v.f < u.d$

Observe, among non-tree edges:

1. (Only) backward edges go to a vertex with a later finish time
2. **(Only) forward edges go to a vertex with later discovery time**

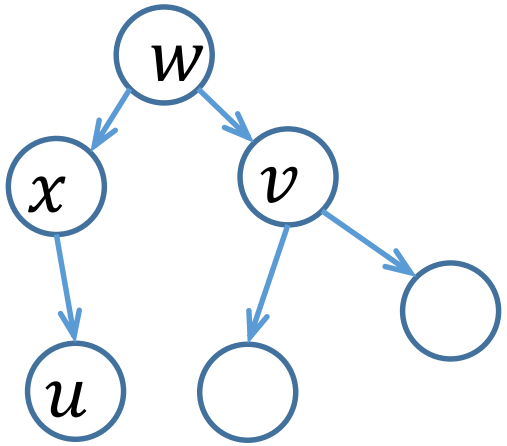

u
v
w
1/8
2/7
9/12
F
B
C
B
4/5
3/6
10/11
x
y
z


1
2
3
4
5
6
7
8
9
10
11
12
u
w
v
z
y
x

Tarjan's algorithm

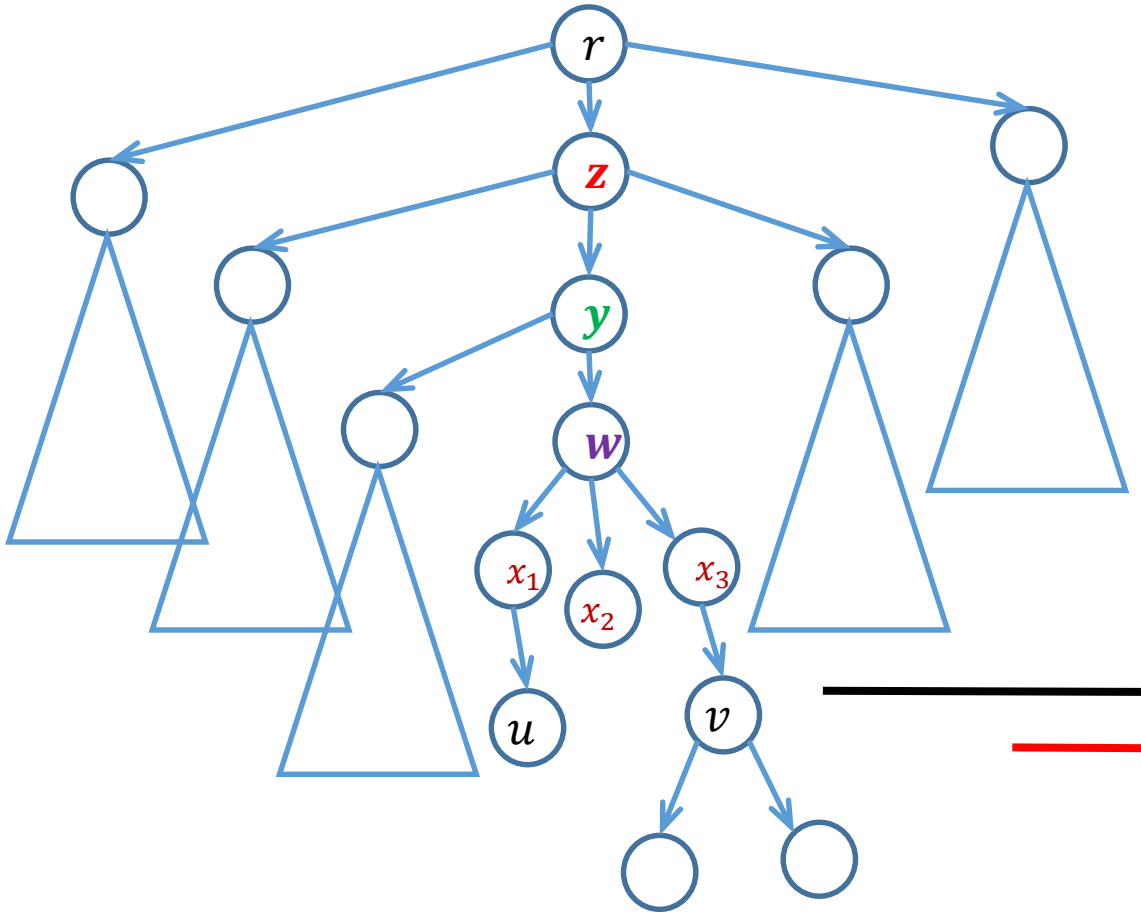
- We will find SCCs using only one DFS

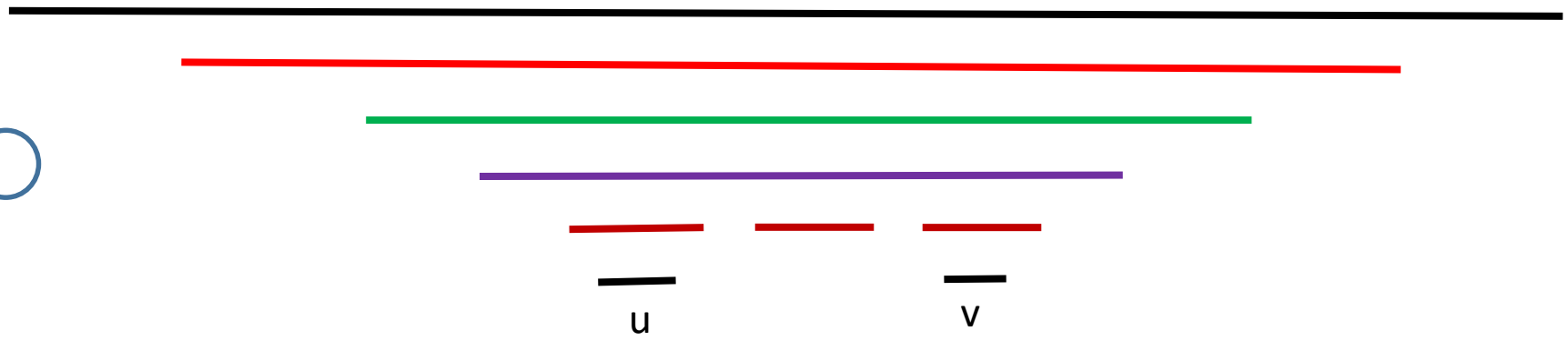
Thm 9: If u and v are in the same SC then they are in same tree of the DFS-forest and $LCA(u, v)$ is also in this component



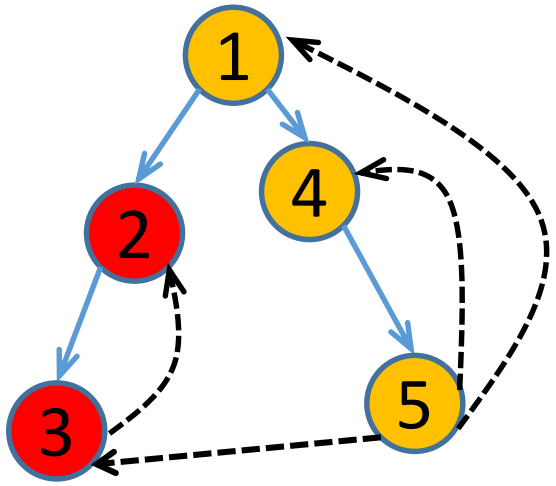
Corr: Each SC form a **connected** subtree of the DFS-forest

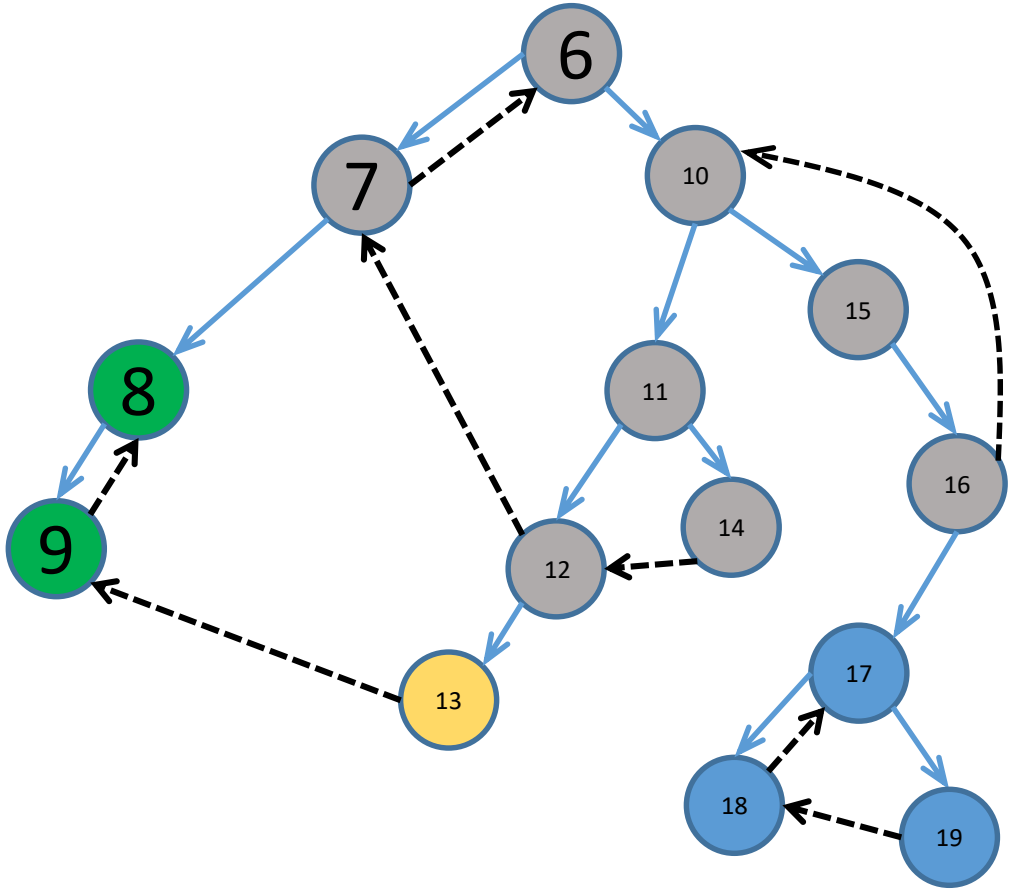
Proof: “by picture”, order the children of every node from left to right by their discovery time



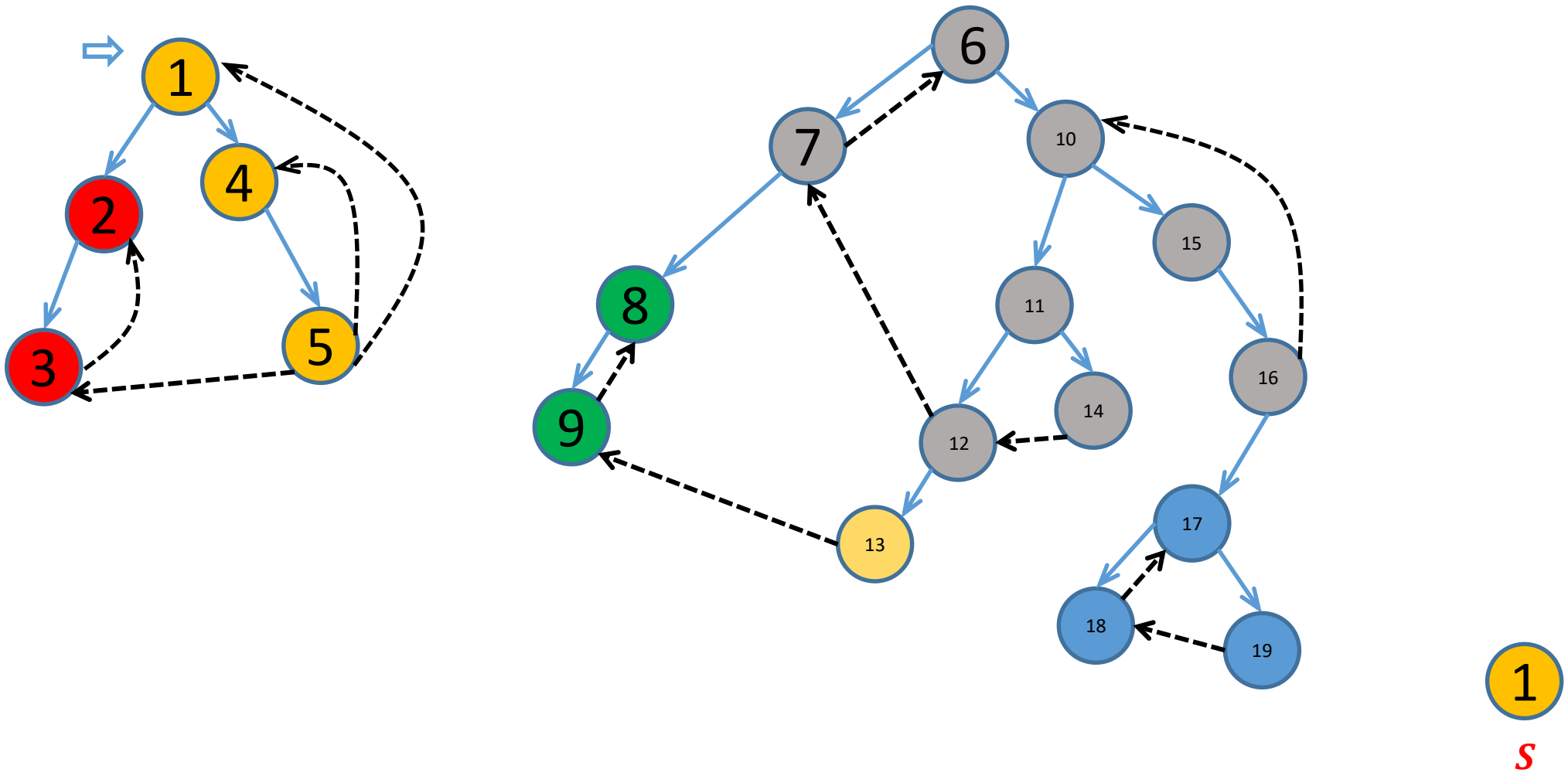


Identify the roots of these subtrees

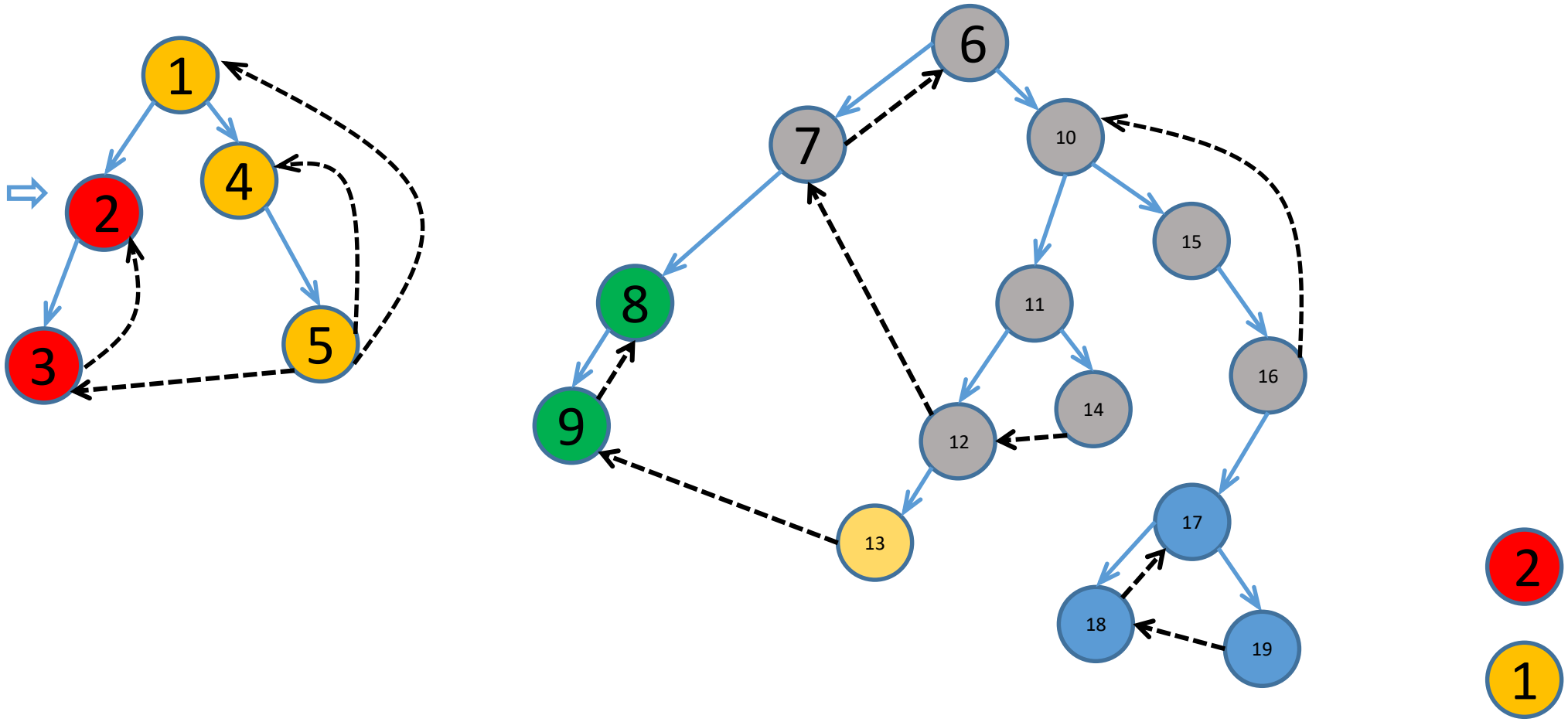




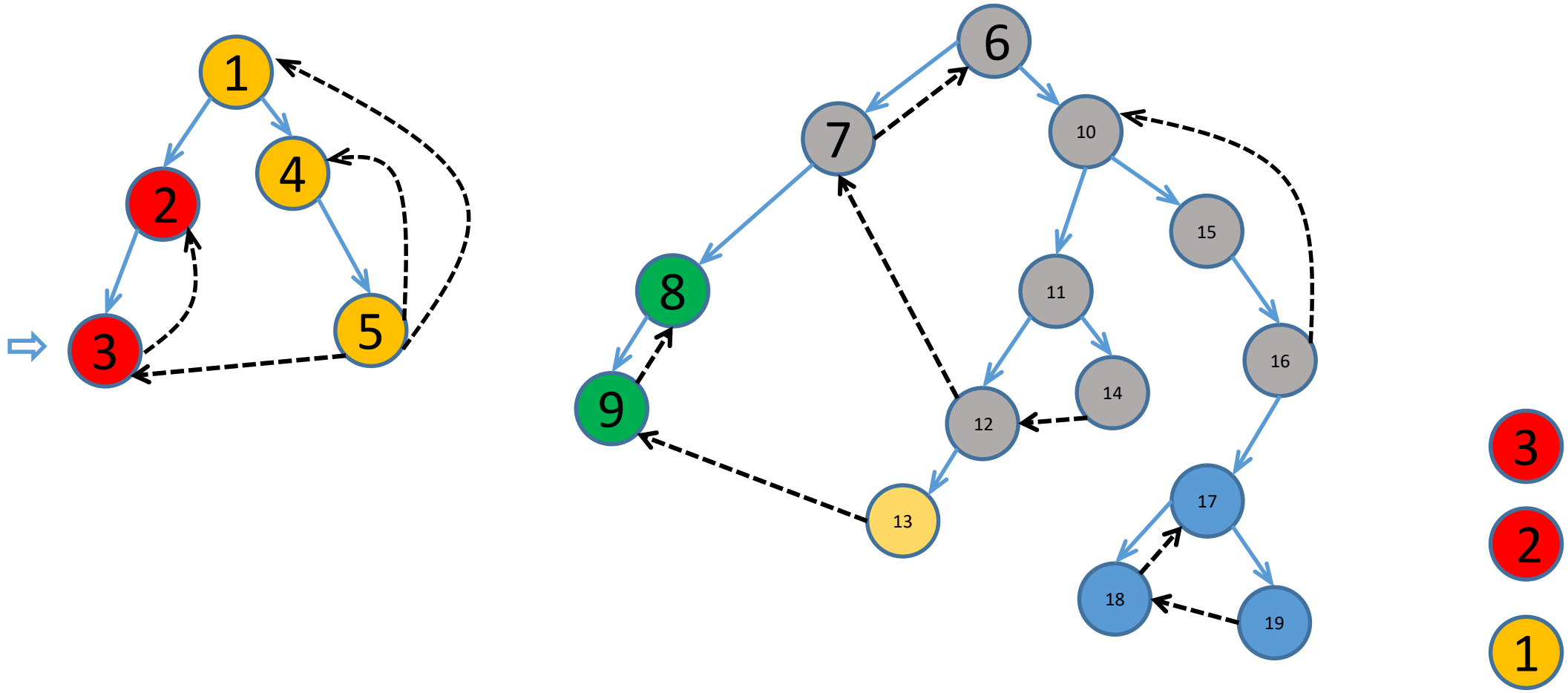
Use an additional stack (S)



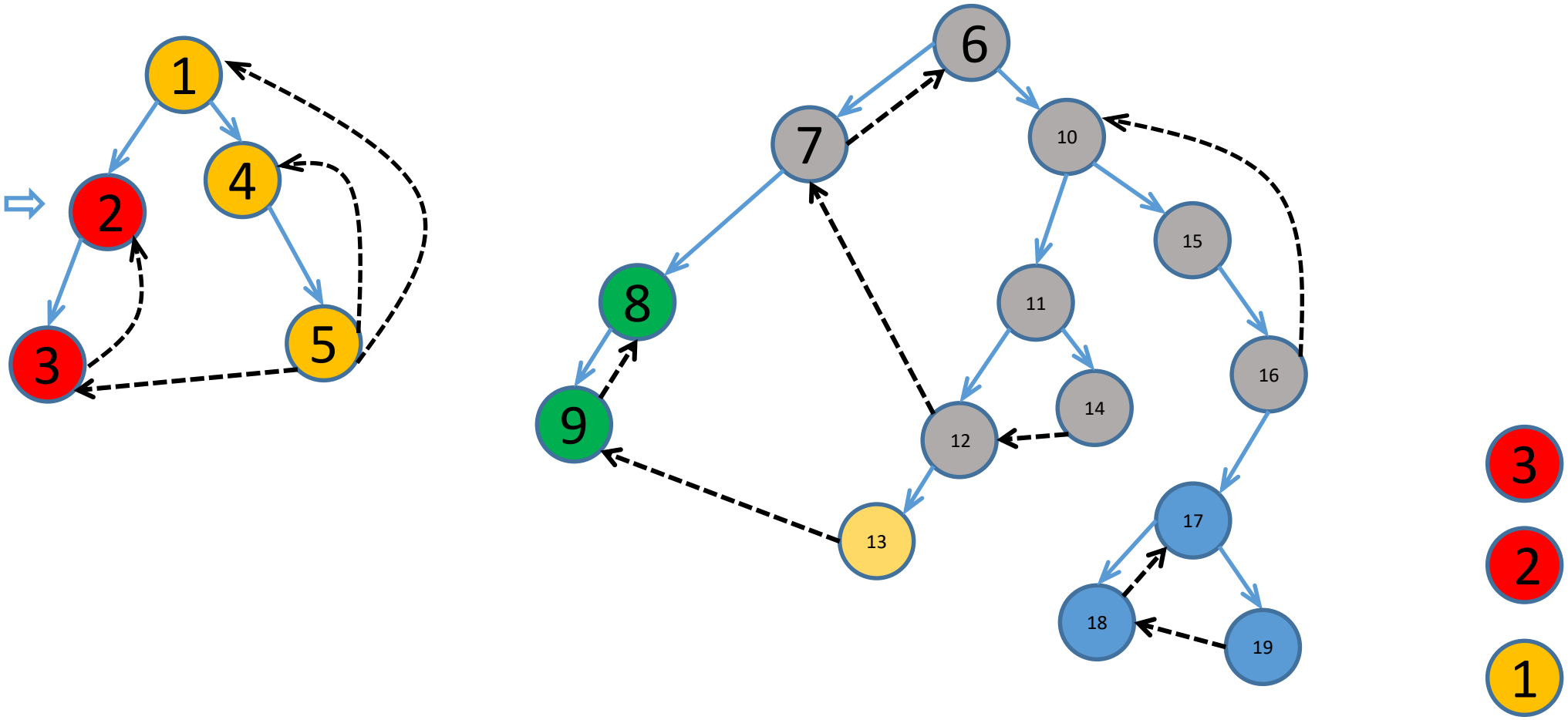
Use an additional stack



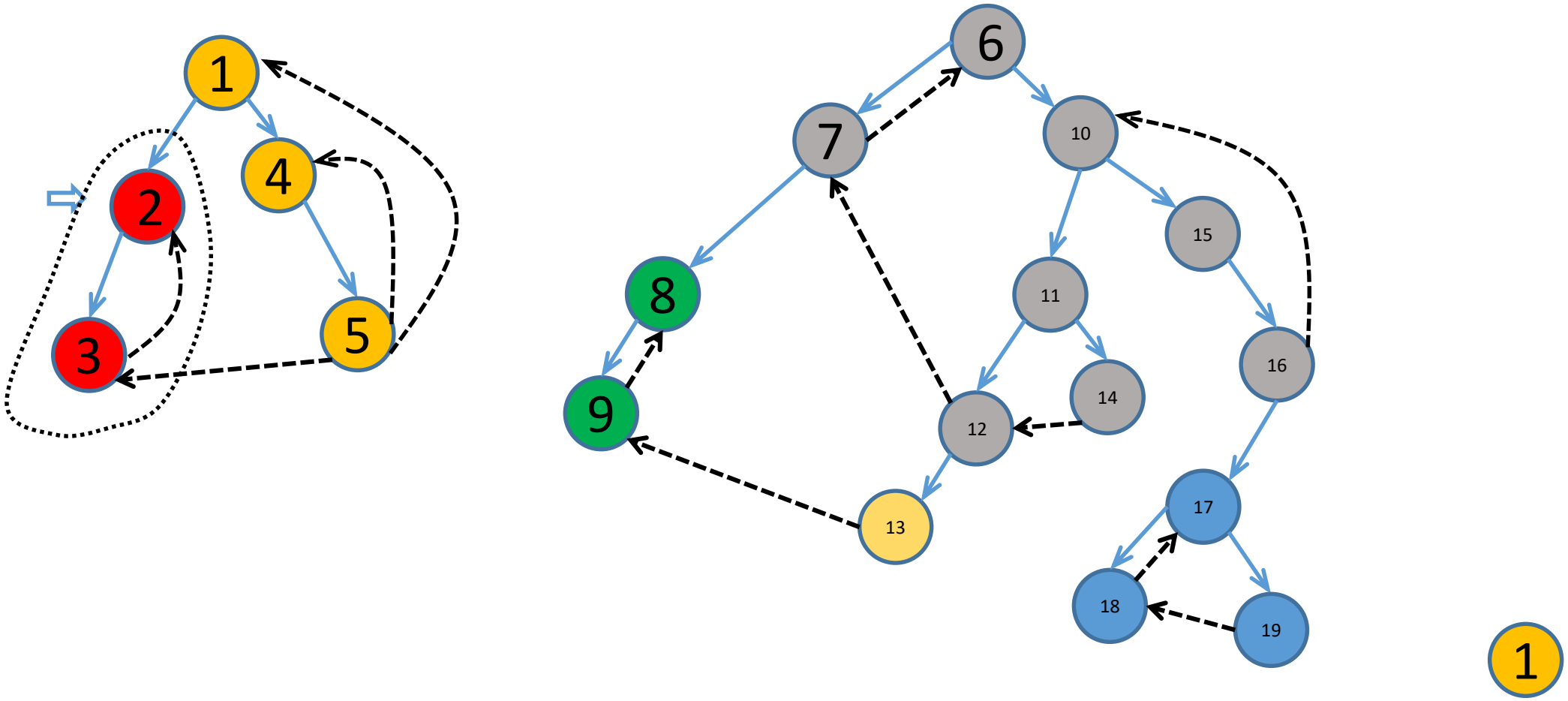
Use an additional stack



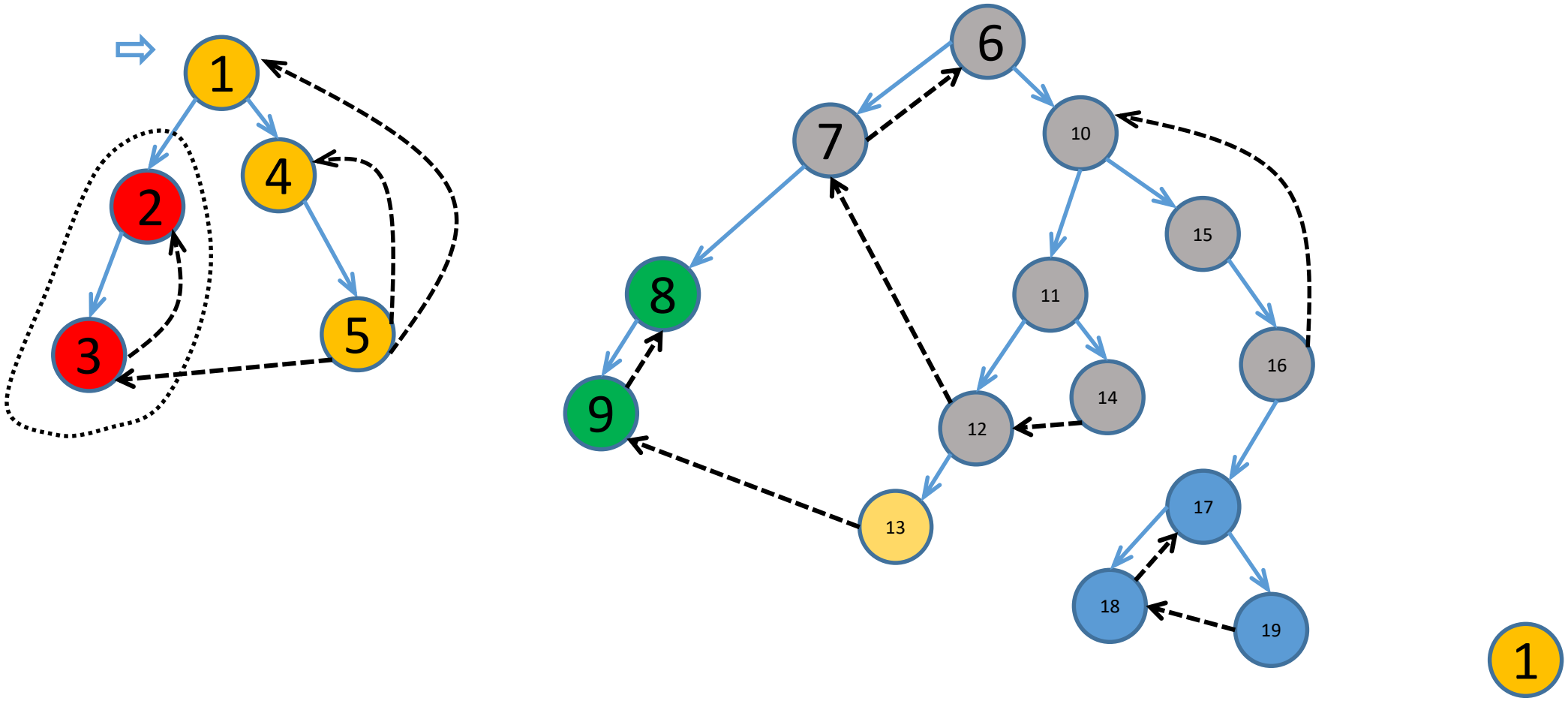
Use an additional stack



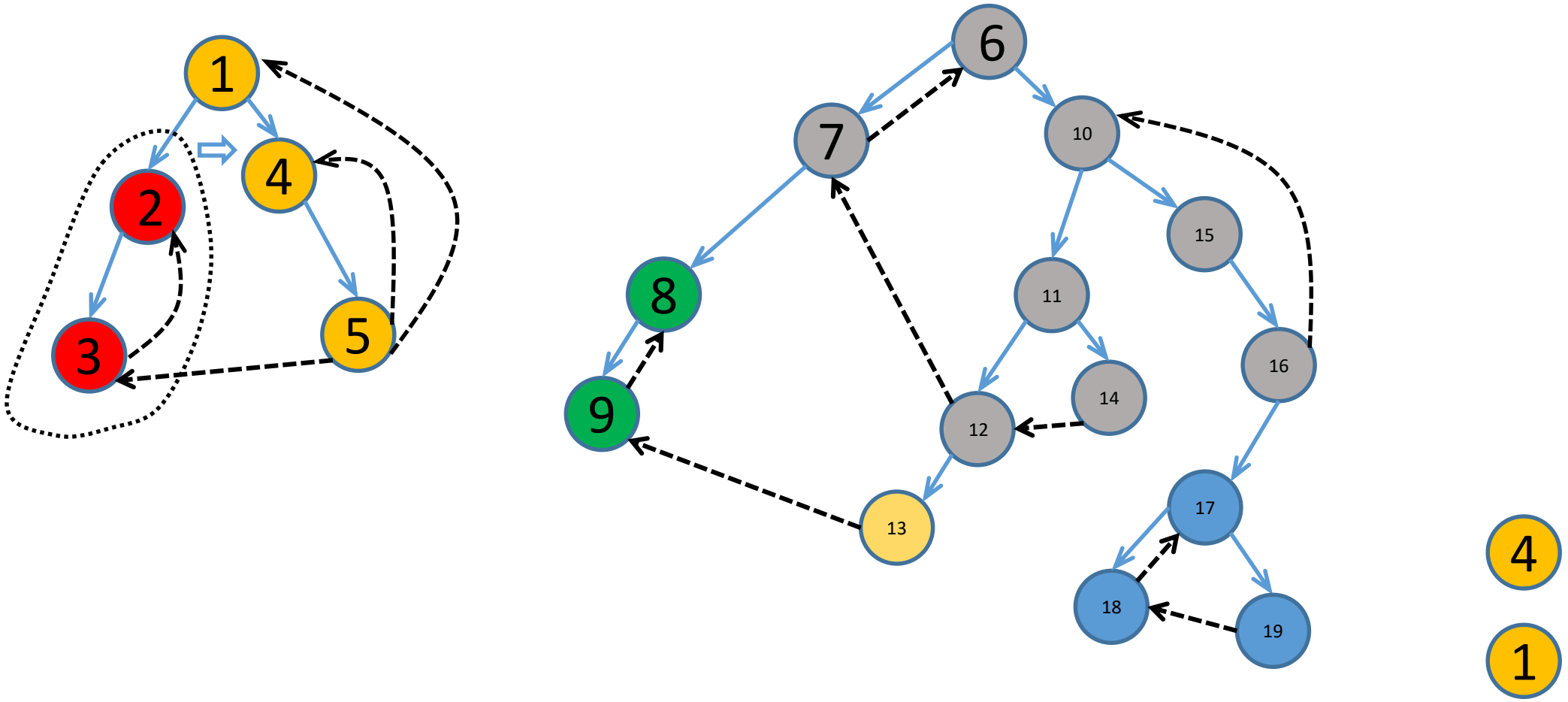
Use an additional stack



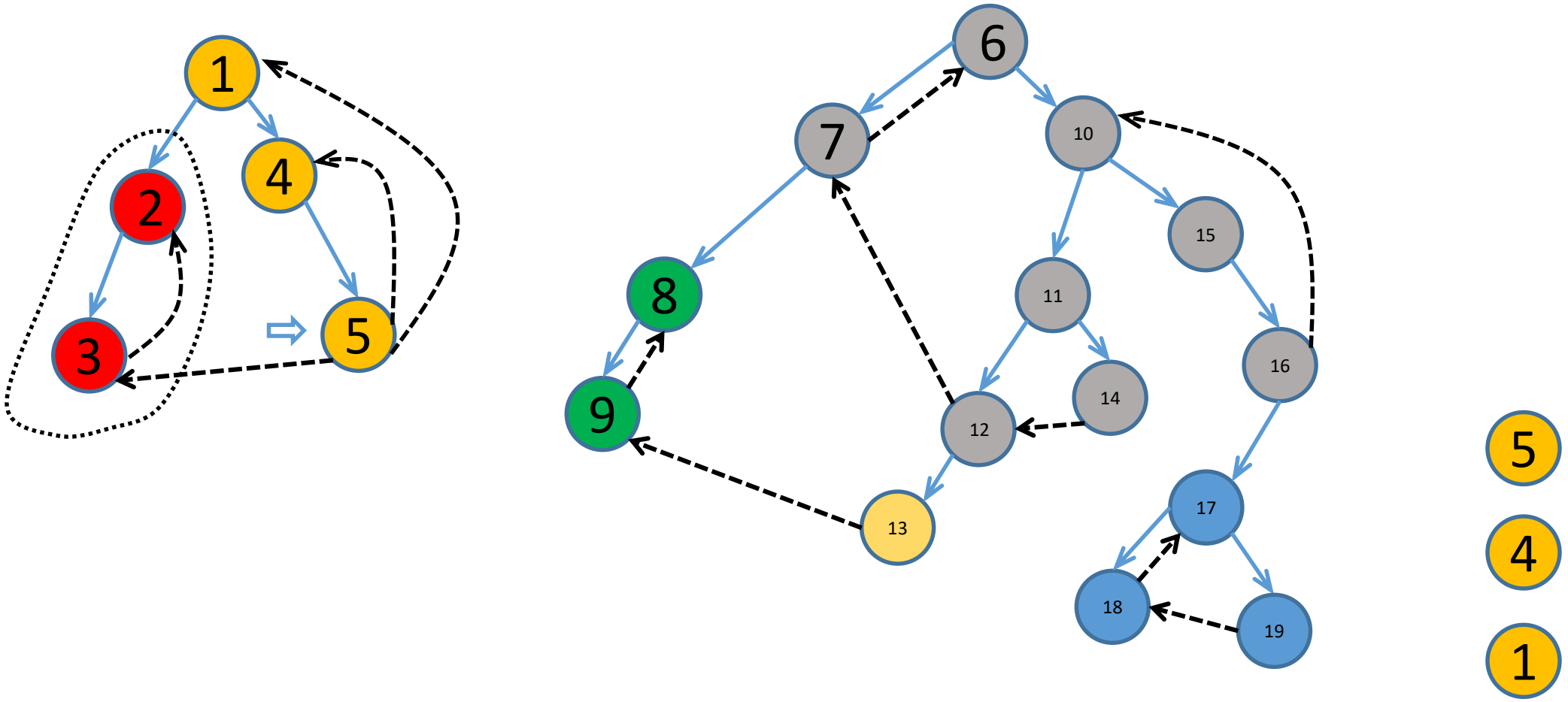
Use an additional stack



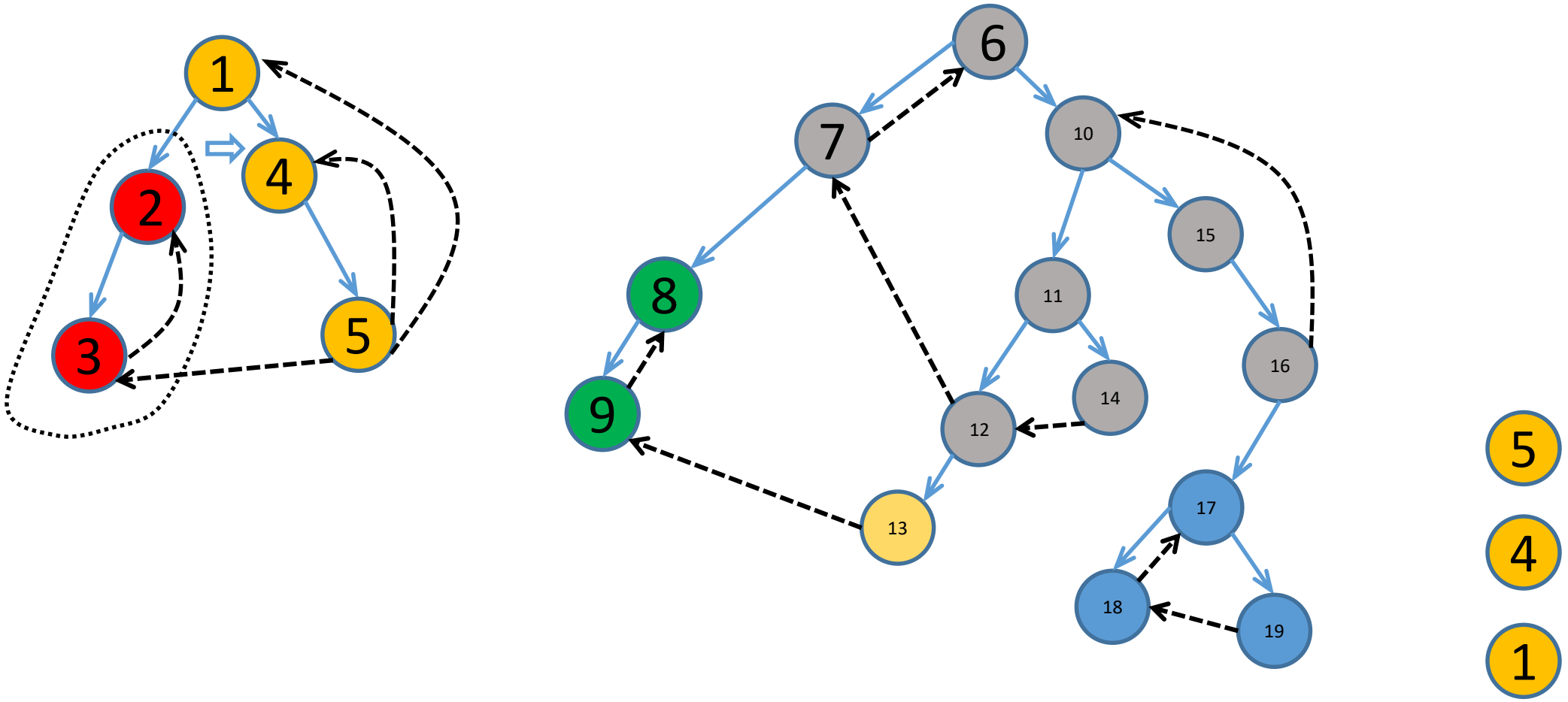
Use an additional stack



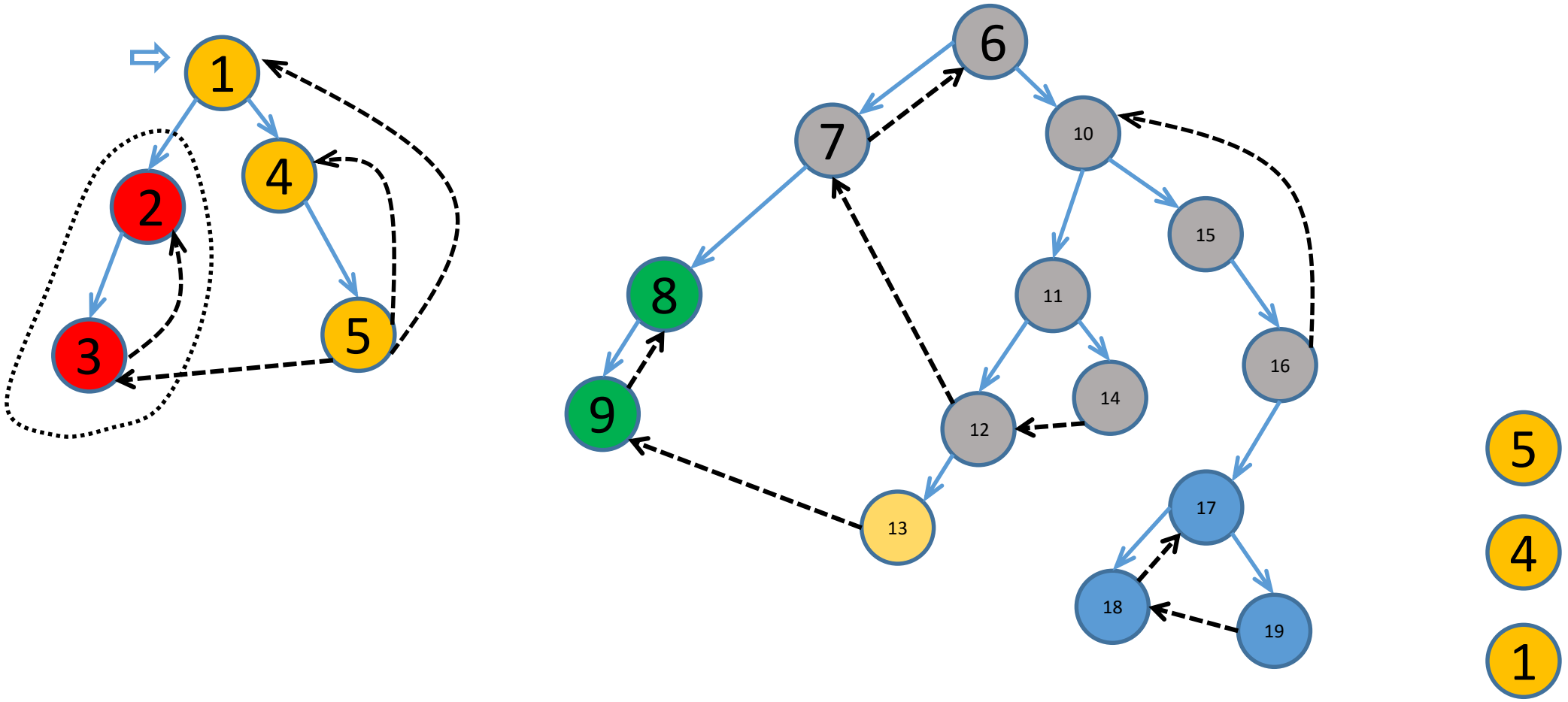
Use an additional stack



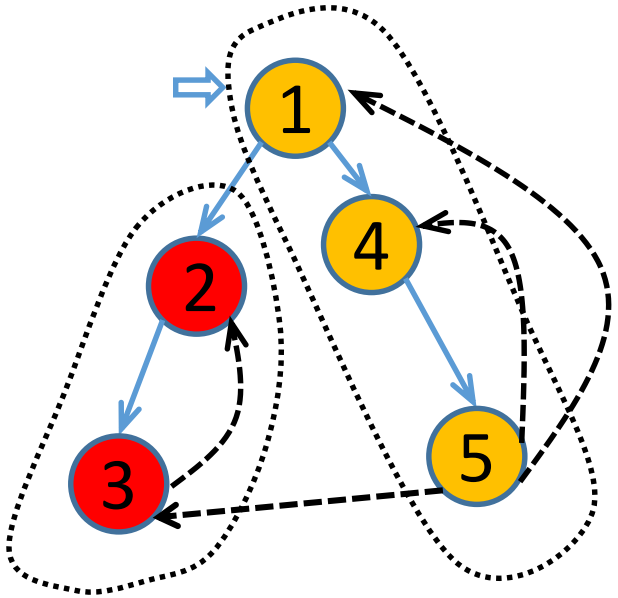
Use an additional stack

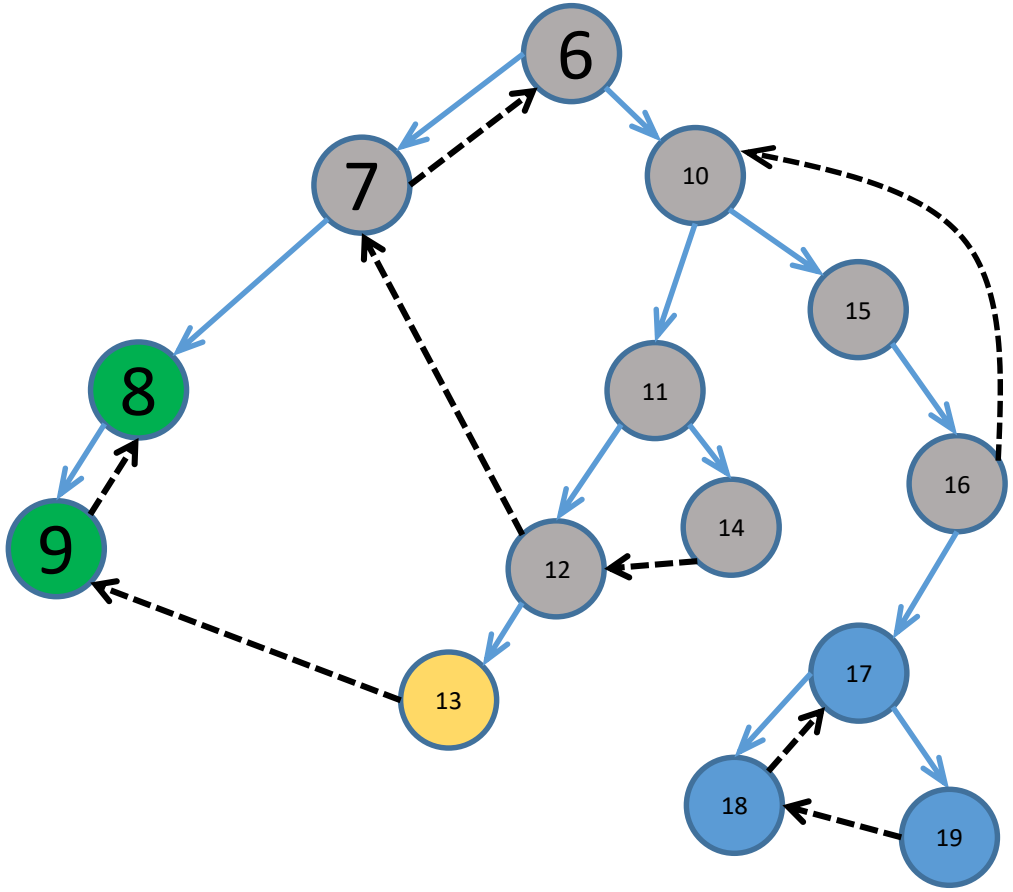


Use an additional stack

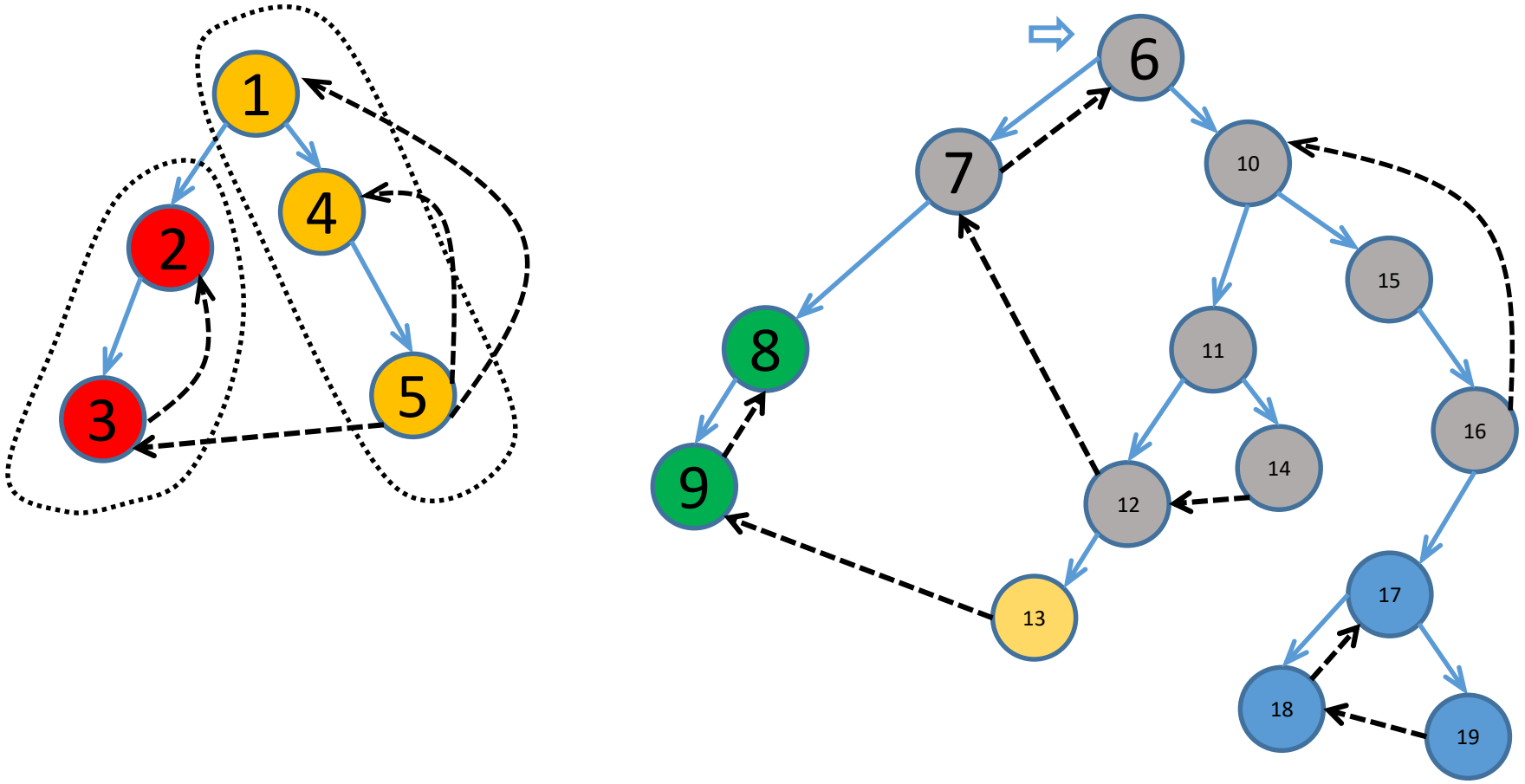


Use an additional stack

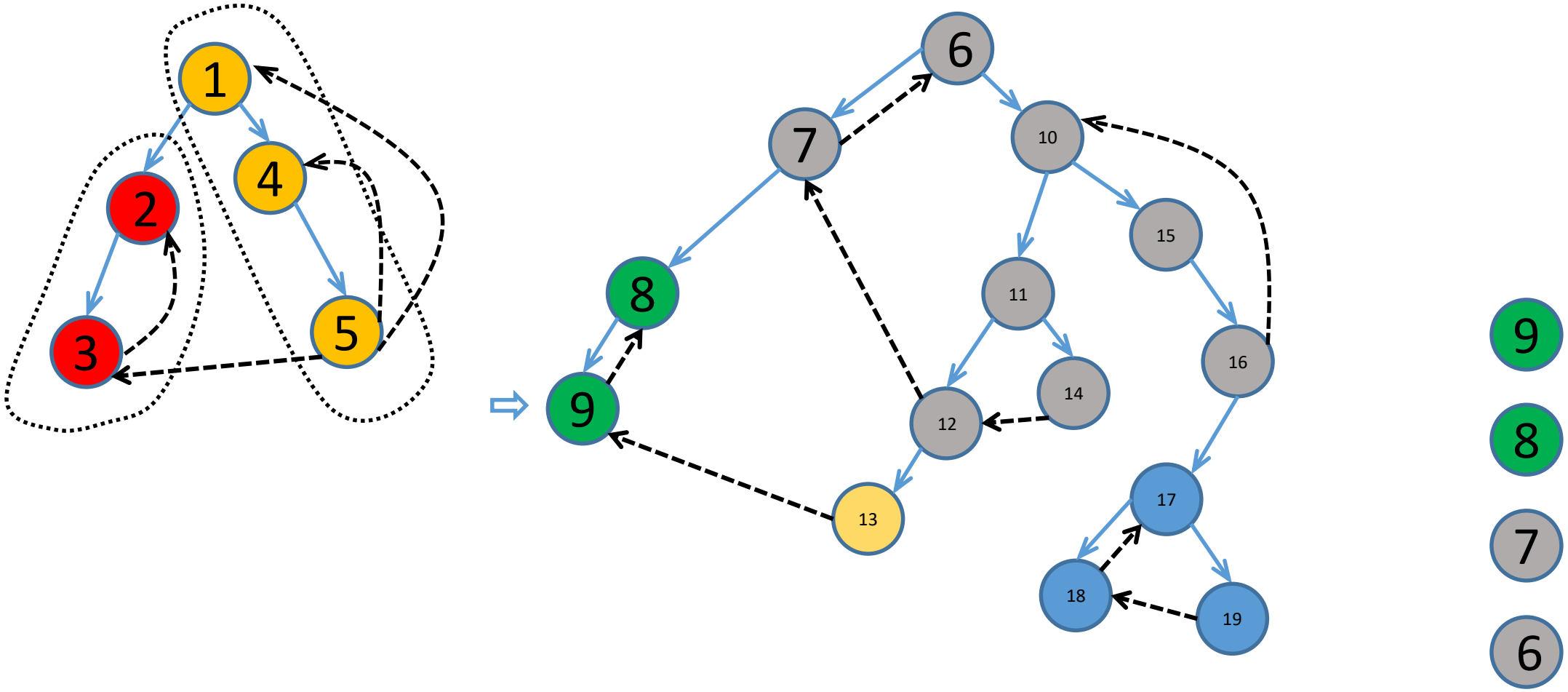




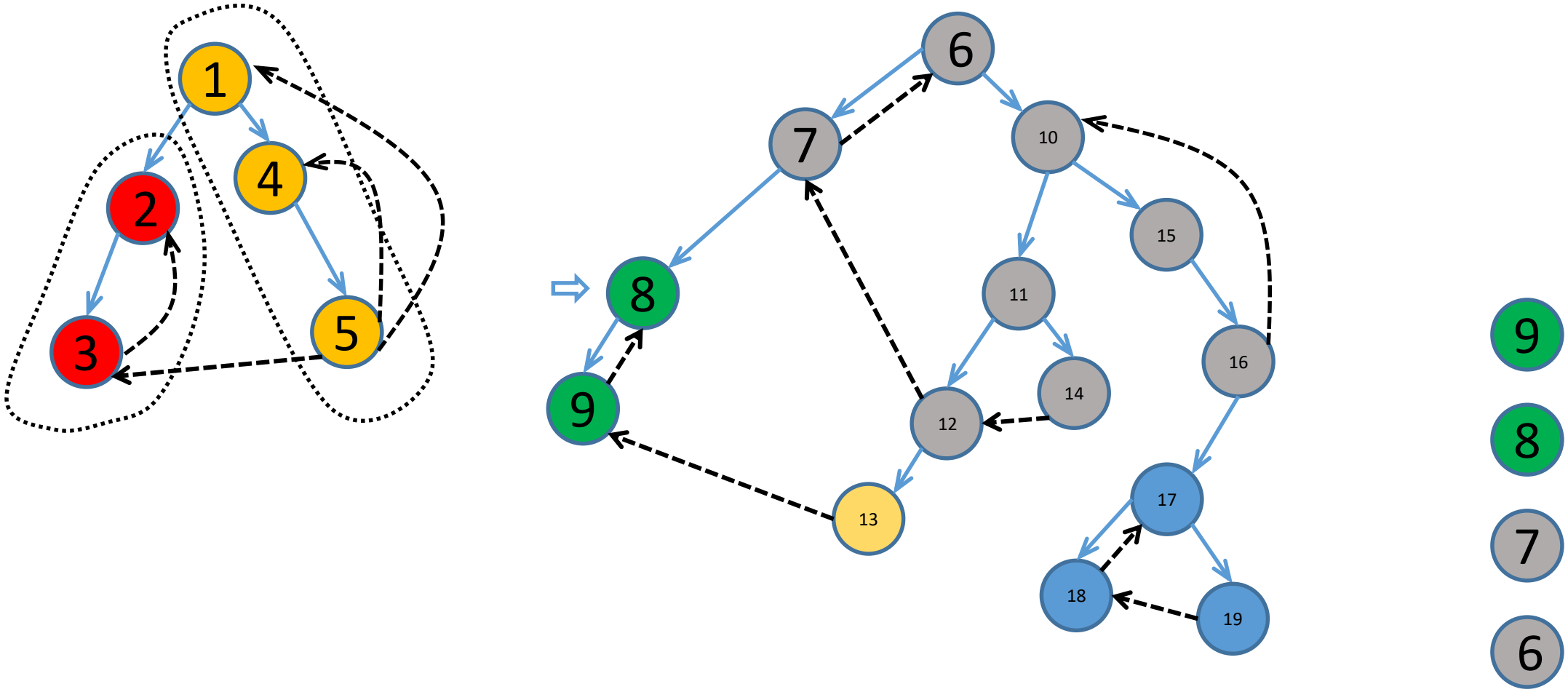
Use an additional stack



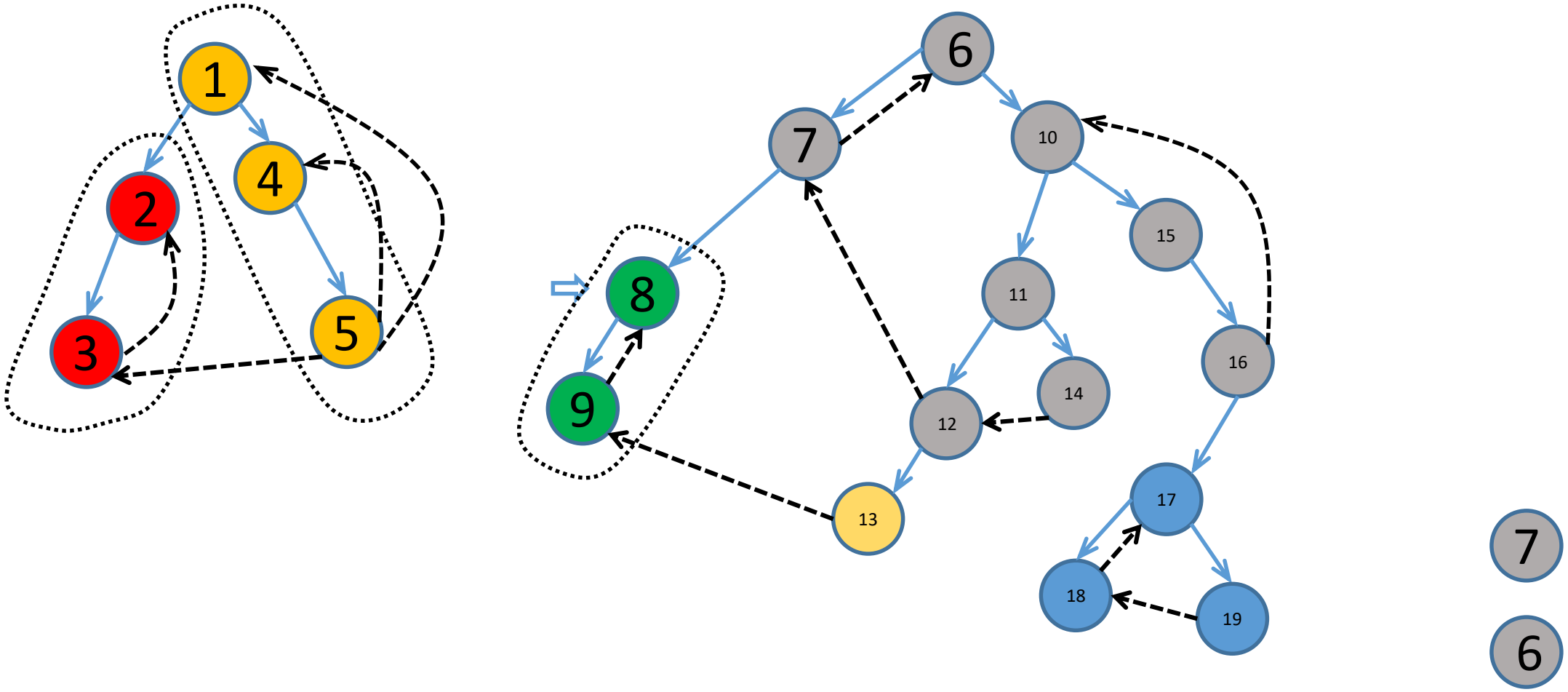
Use an additional stack



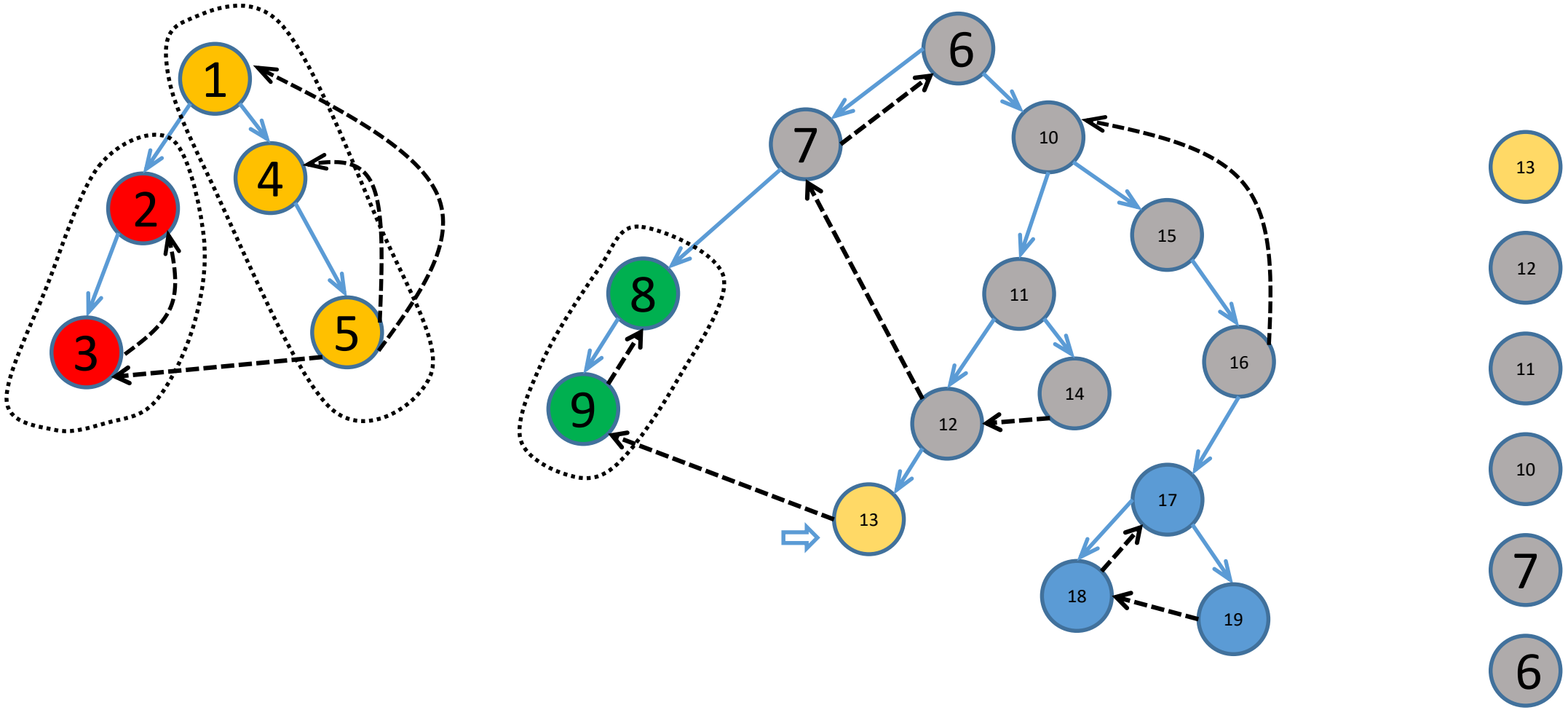
Use an additional stack



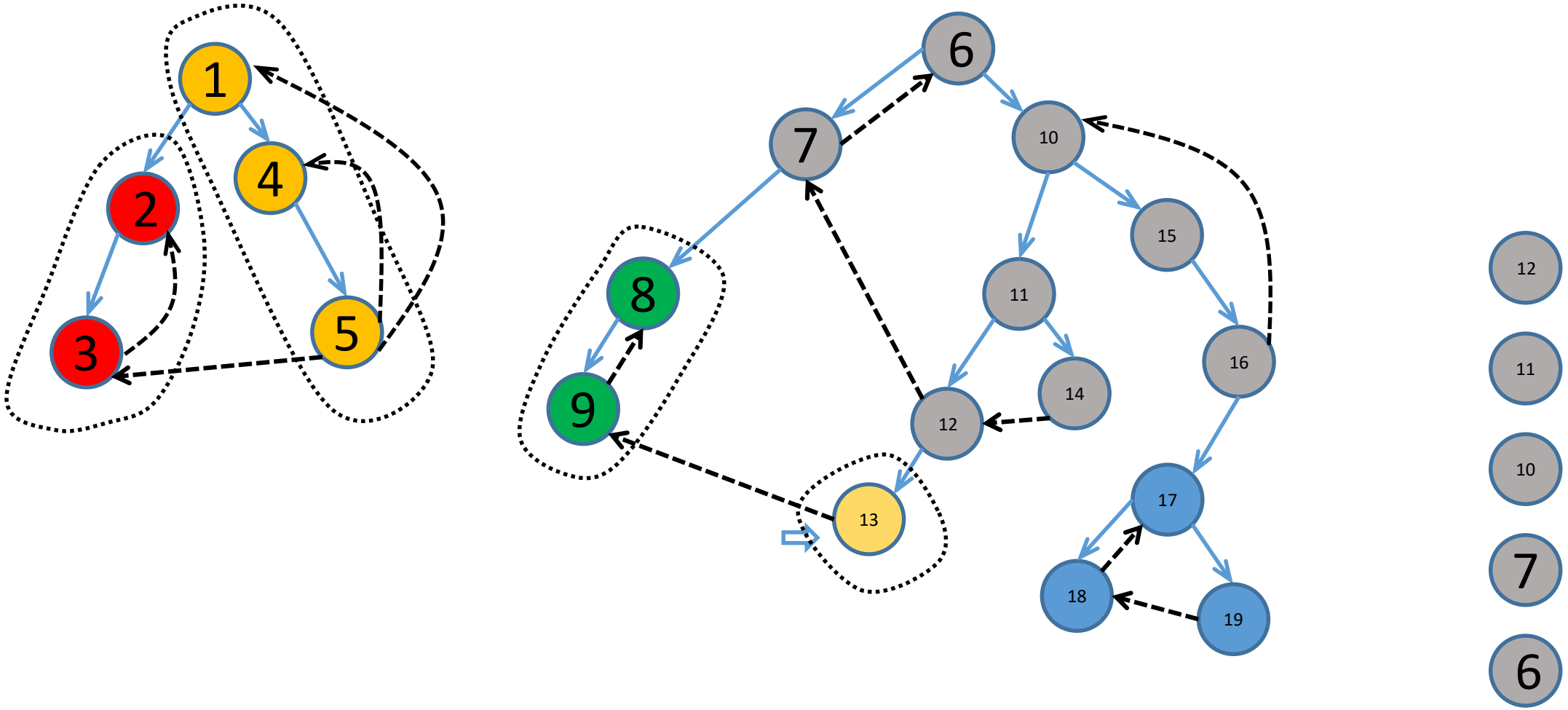
Use an additional stack



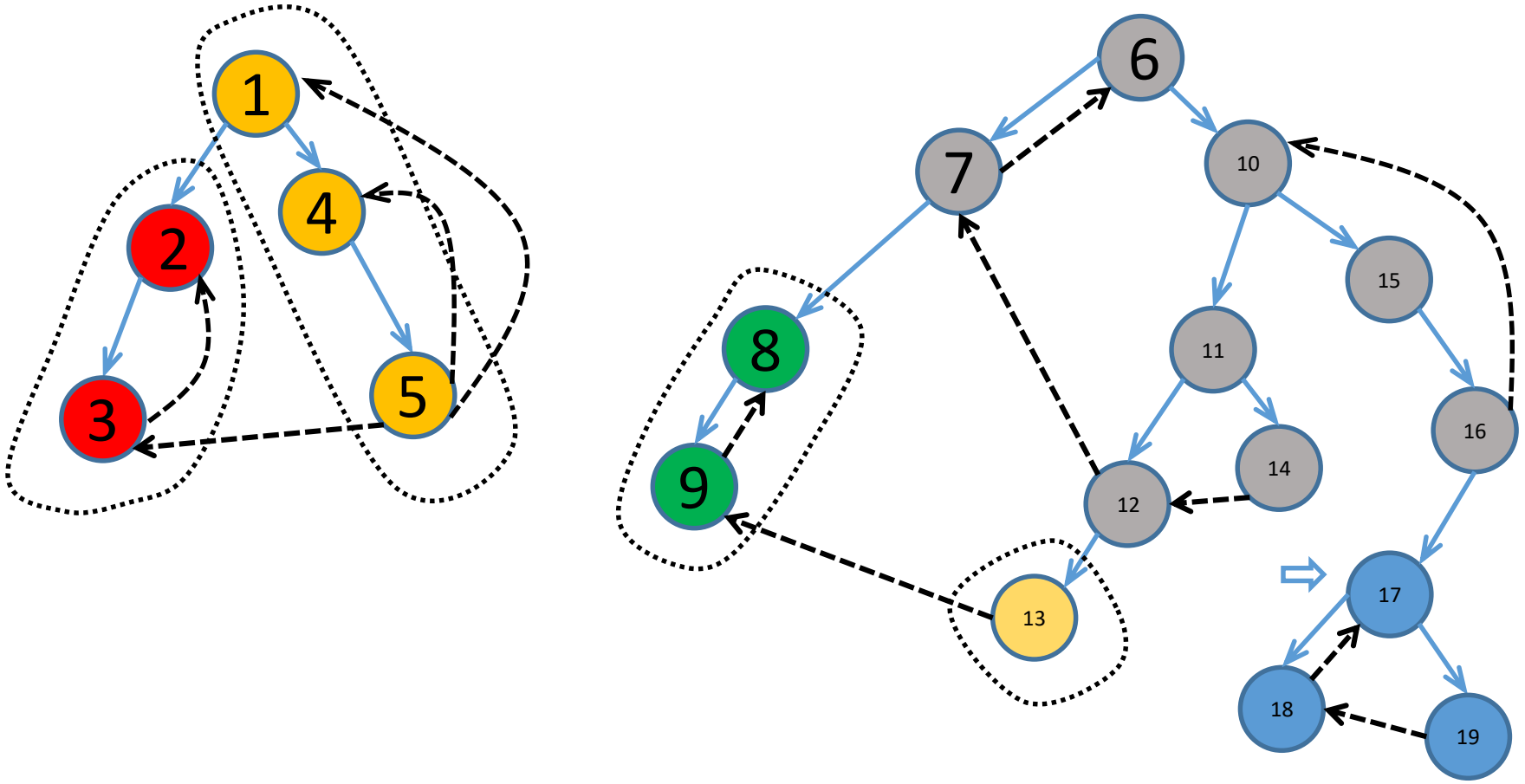
Use an additional stack



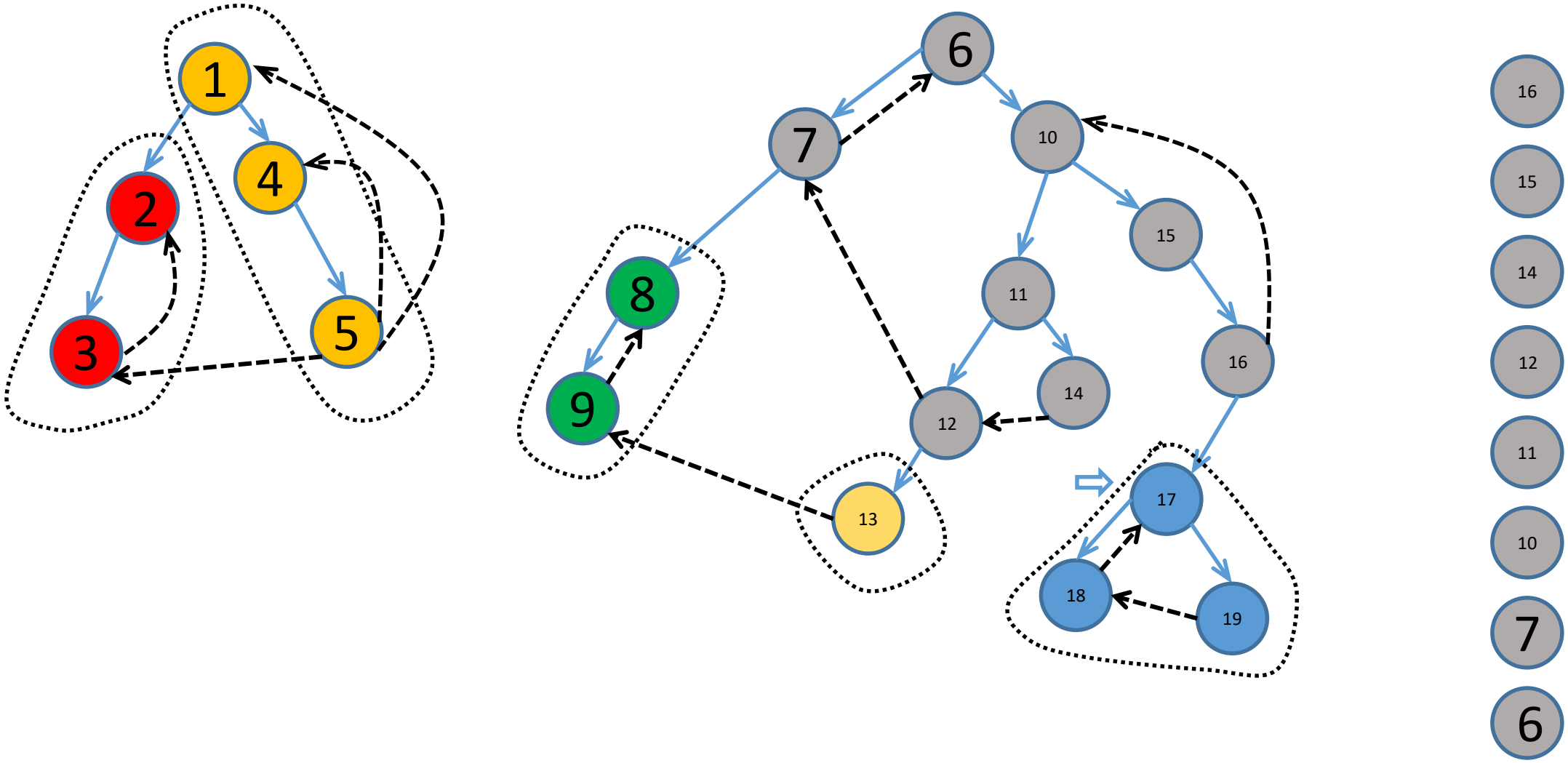
Use an additional stack



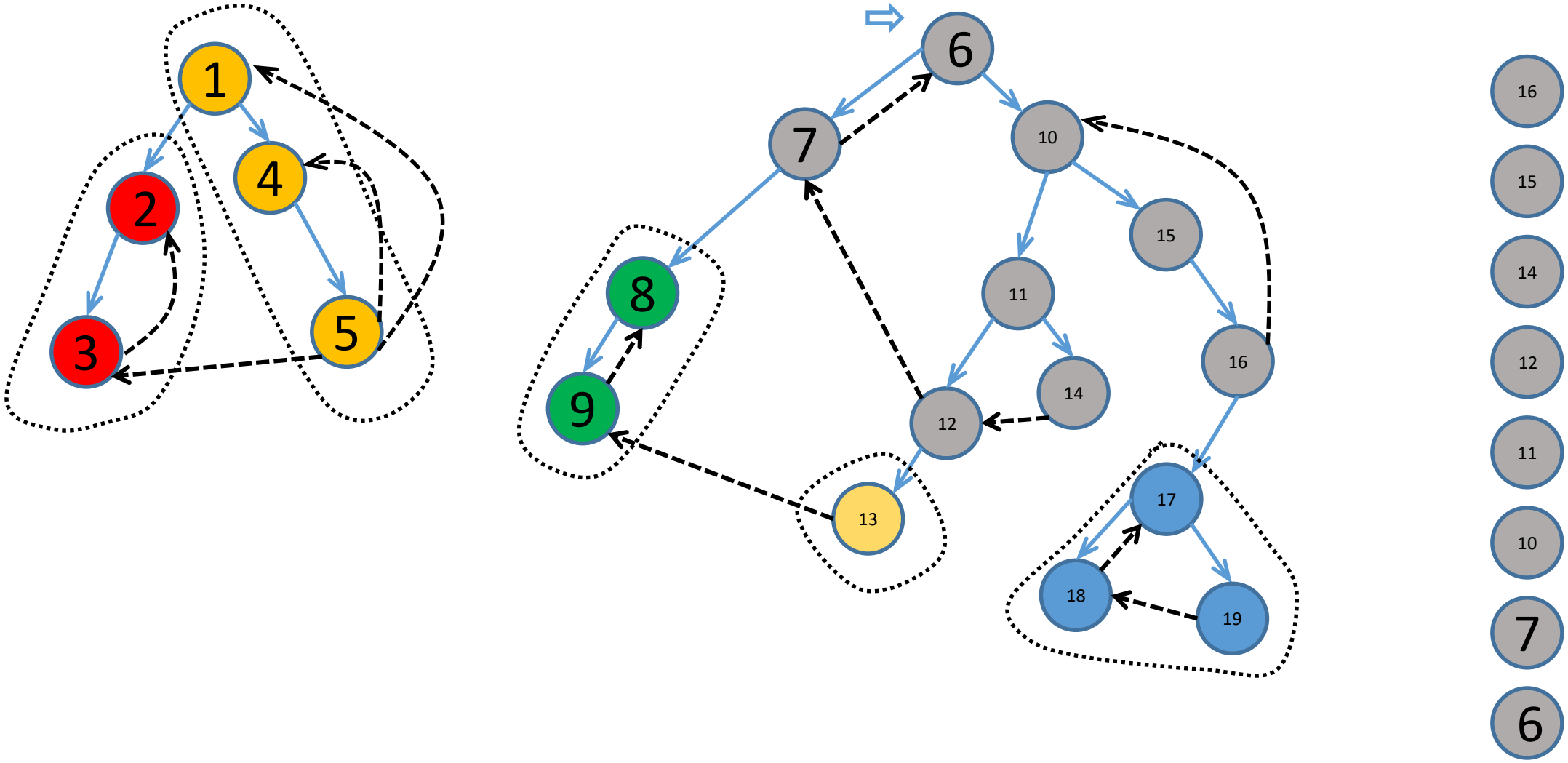
Use an additional stack



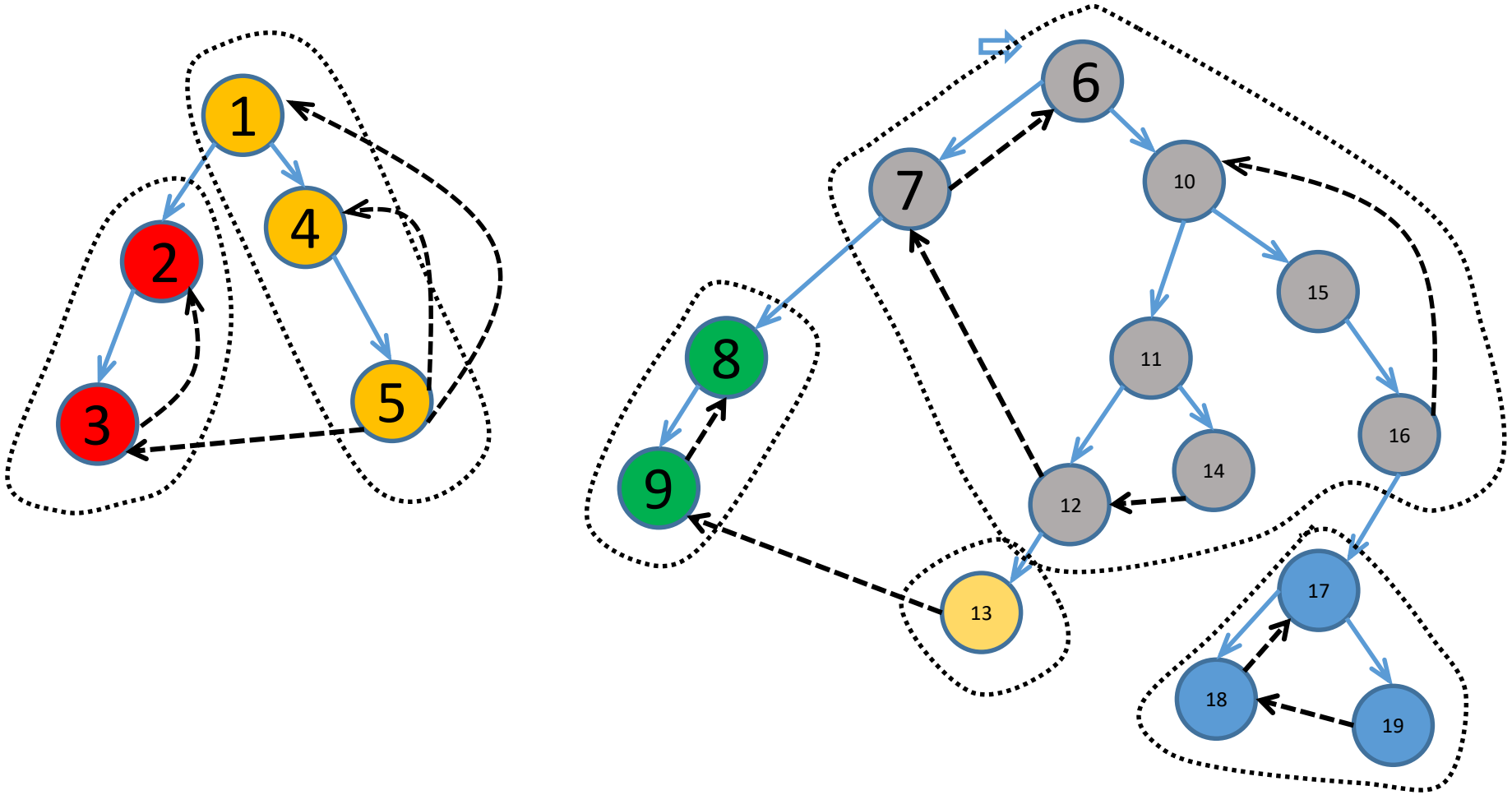
Use an additional stack



Use an additional stack



Use an additional stack



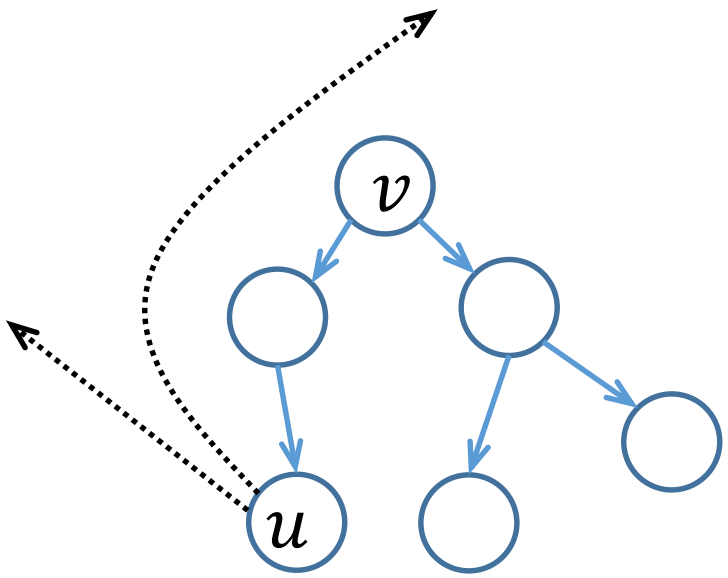
The Stack Invariant

- Let $C_1, C_2, \dots, C_i$ be the SC that intersect the path of the GRAY DFS vertices ordered top down.
- Then S contains the vertices of each C_j that we have already visited
- The vertices of C_1 are at the bottom of S and $\forall j > 1$ the vertices of C_j are consecutive and on top of the vertices of C_{j-1}
- $\forall j$ the root of C_j is the bottommost vertex among all the vertices of C_j

- Proof by induction on the steps of the algorithm

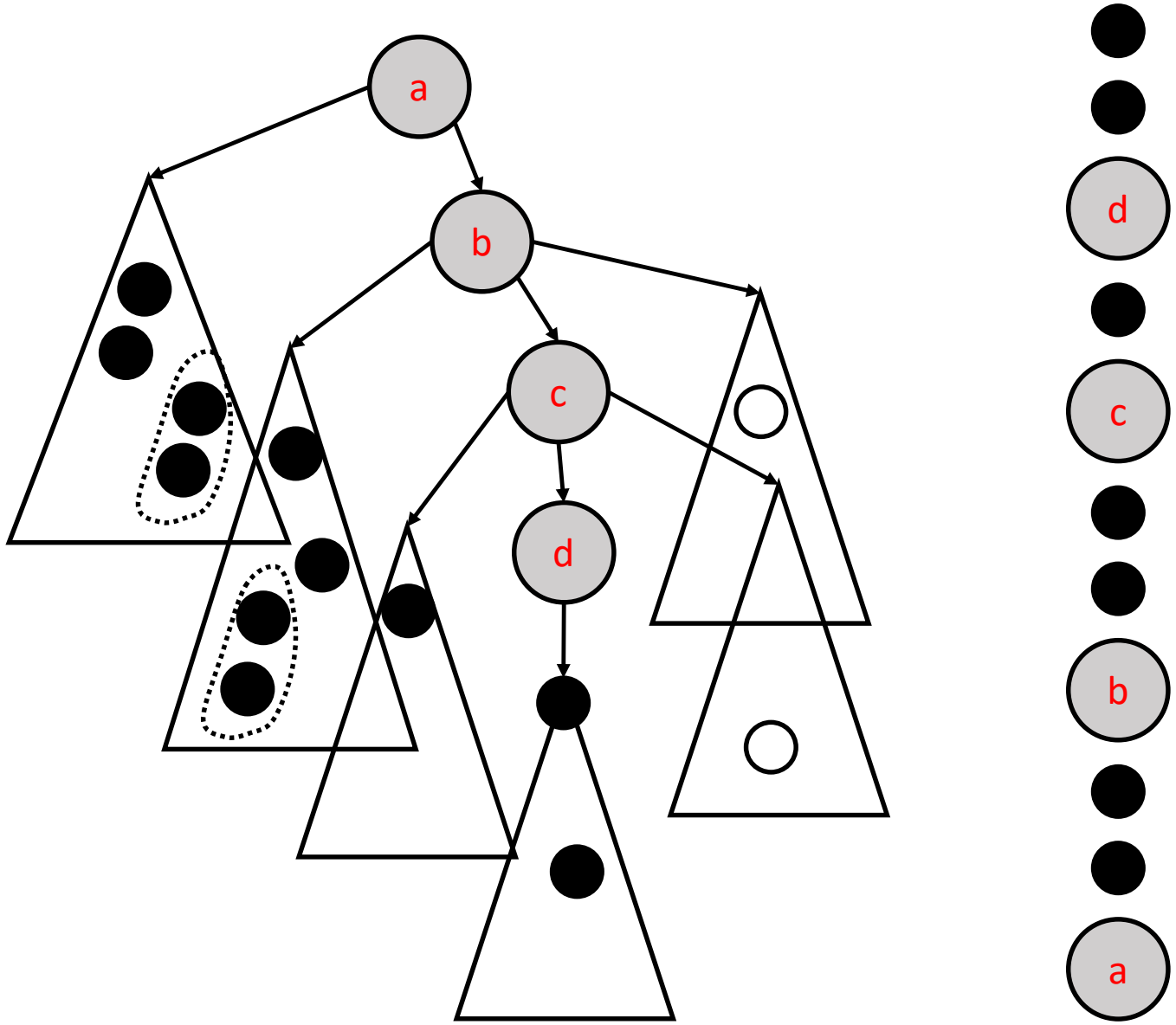
Characterize these roots ?

Thm 10: v is a **root** of a component **iff** when the DFS backtracks from v no back-edges or cross-edges going out of its subtree (in the DFS-forest) to vertices still in the stack.



Proof: Assume we apply this rule then the following Invariants hold:

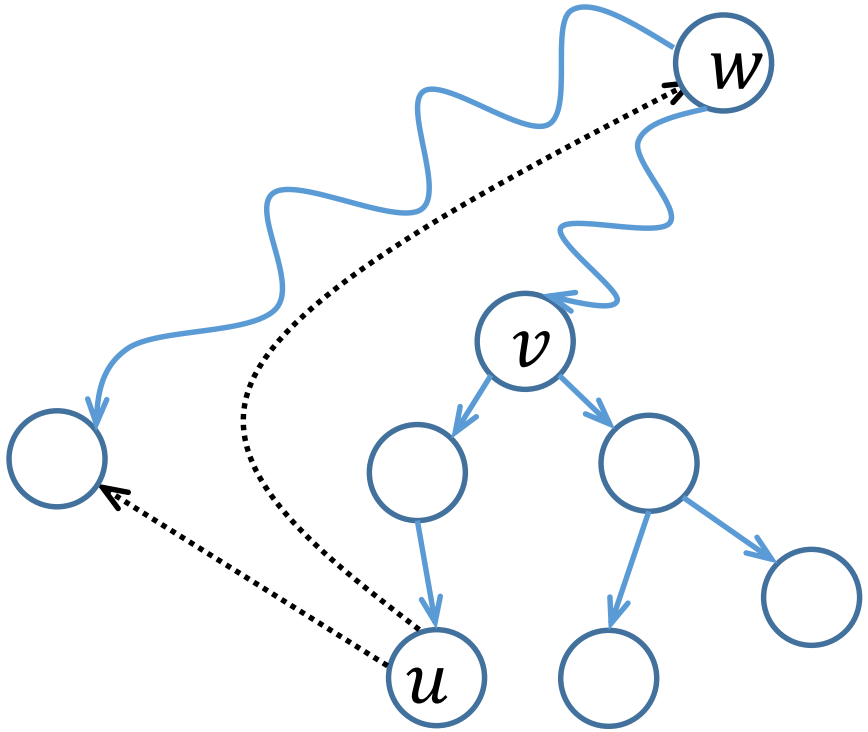
- The components we have already identified are correct
- **If we backtracked from v and v is not a root then v is in the same component as its lowest GRAY ancestor**


a
b
c
d

d
c
b
a

How do we identify roots ?

For each vertex v compute $low(v)$ = the smallest $w.d$ such that there is a cross edge or back edge to $w \in \boldsymbol{S}$ out of the subtree of v.
If there is no such edge then $low(v) \ = \ v.d$



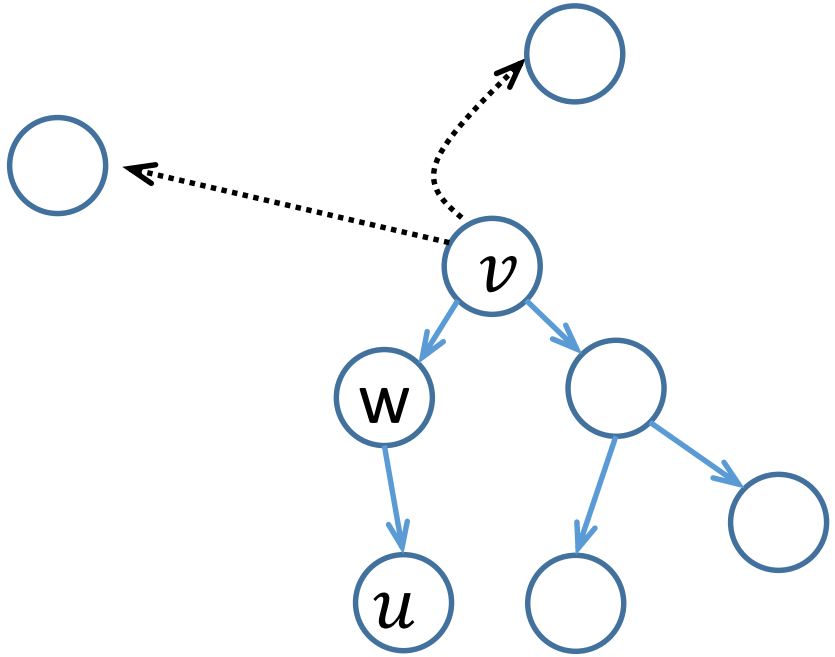
If $low(v) \ = \ v.d$
then v is the root of a component

How to compute low values ?

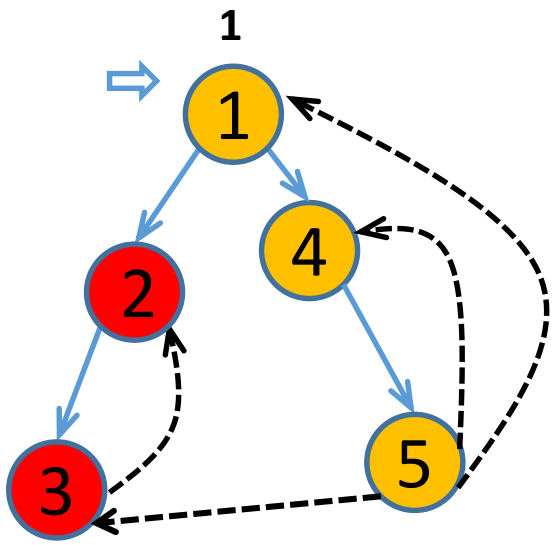
Initialize $low(v) = v.d$

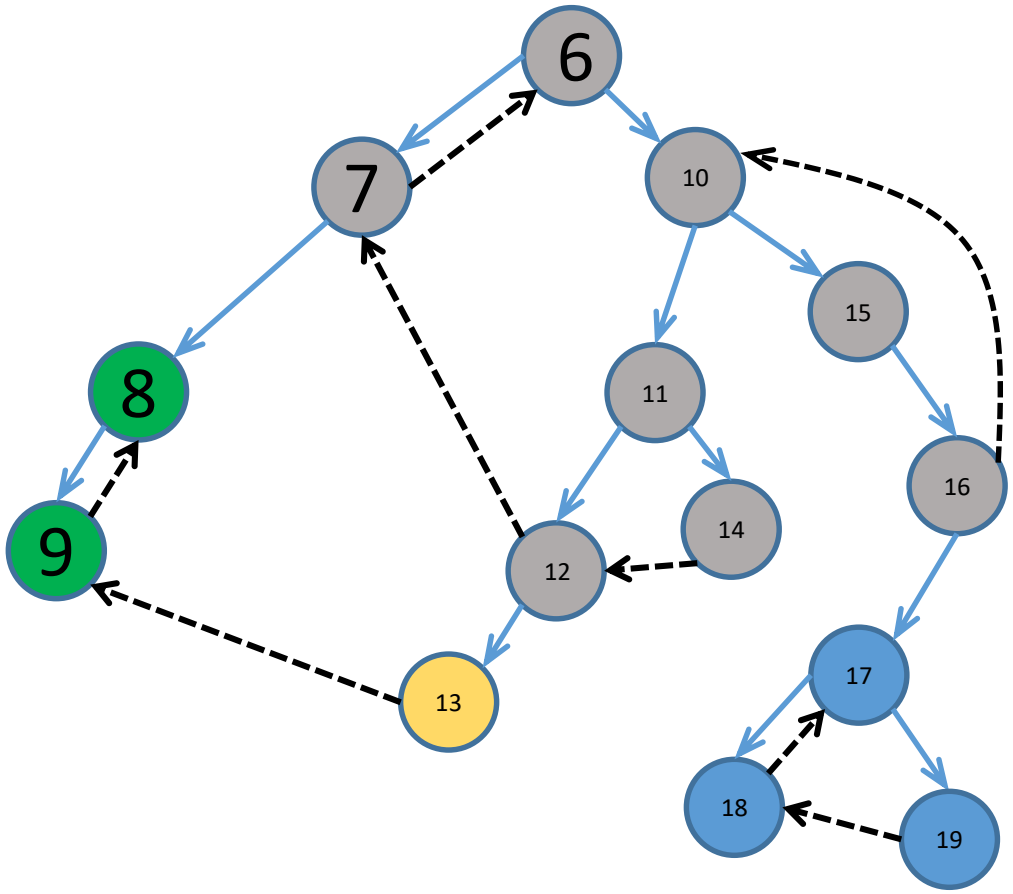
Update when you see a back edge or cross edge out of v

Update when you backtrack to v from a child w of v

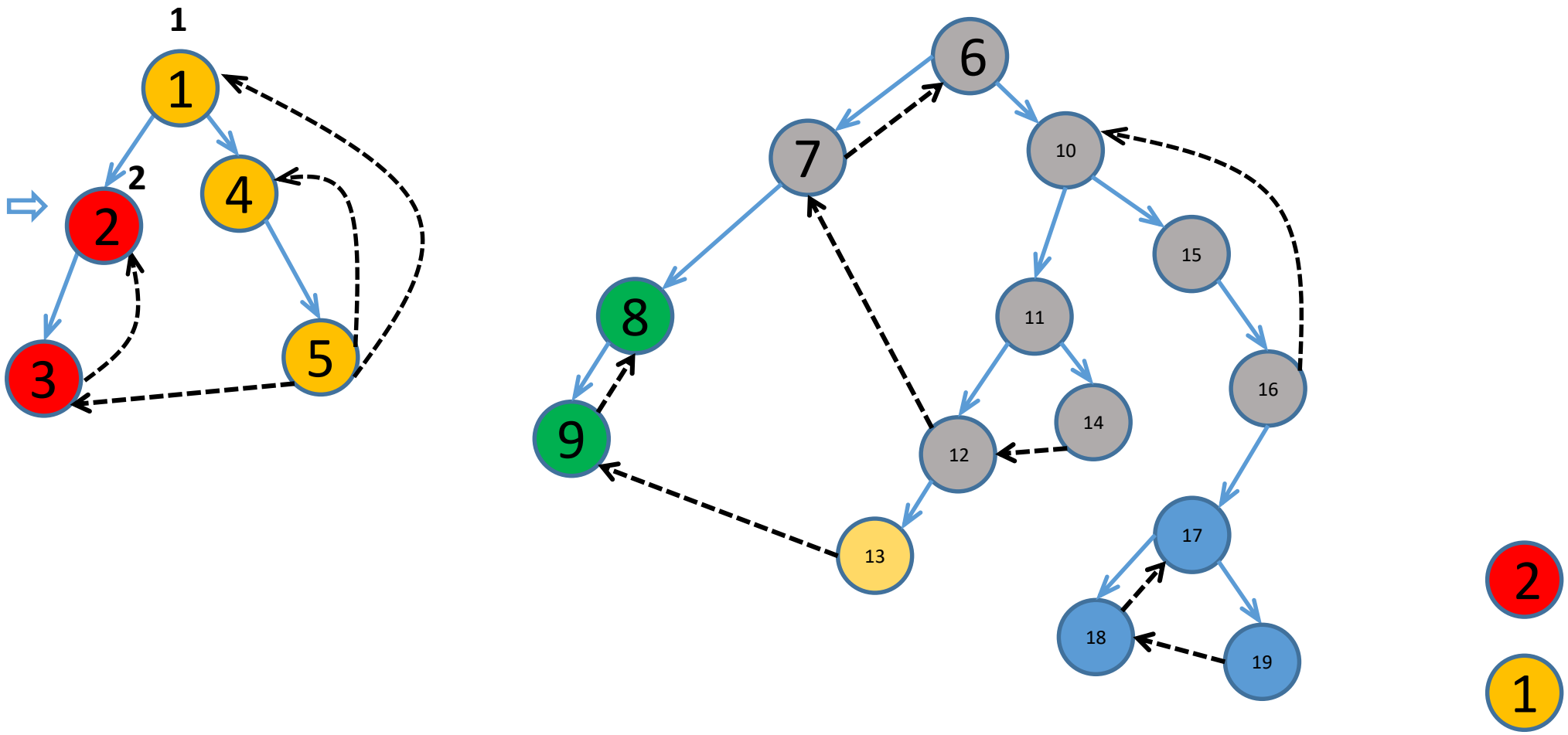


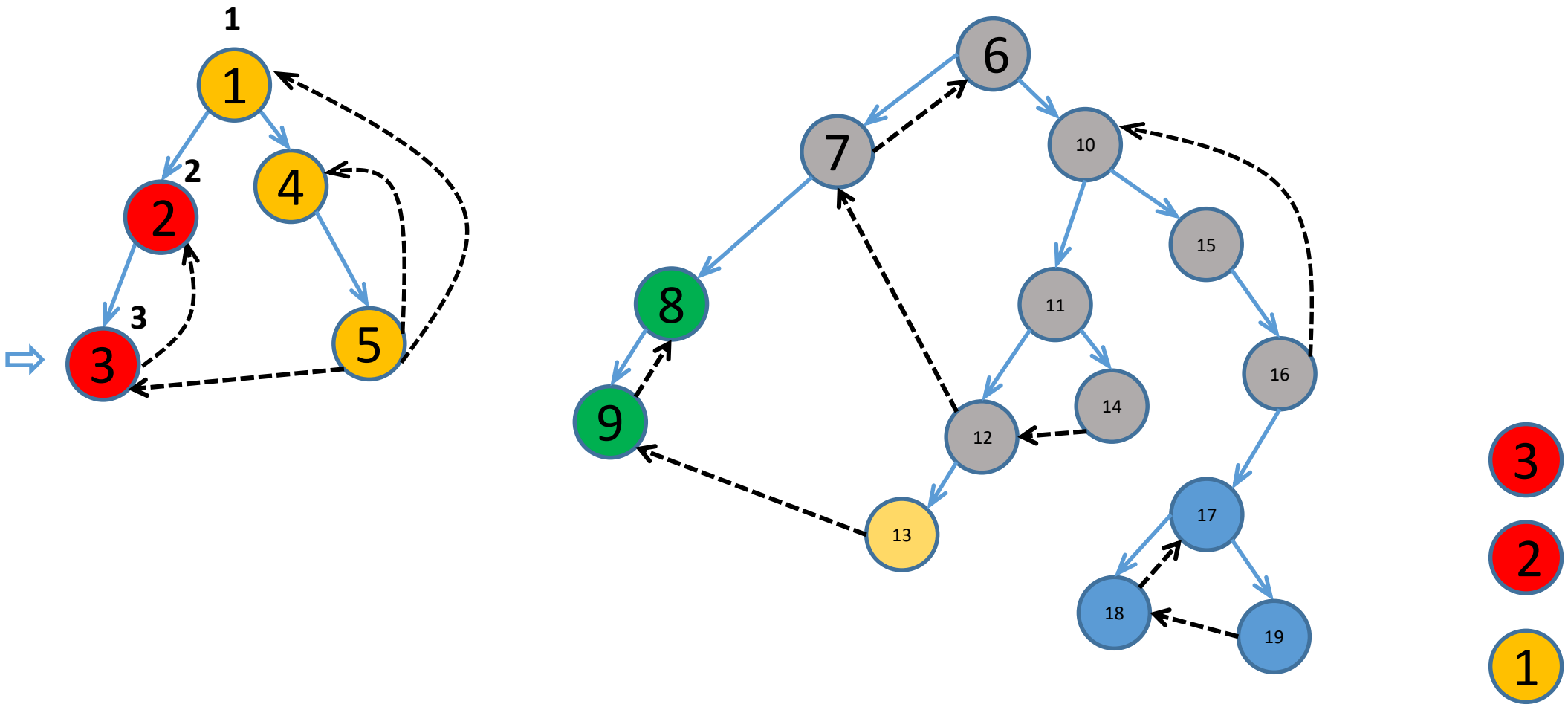
Compute SCs using low values

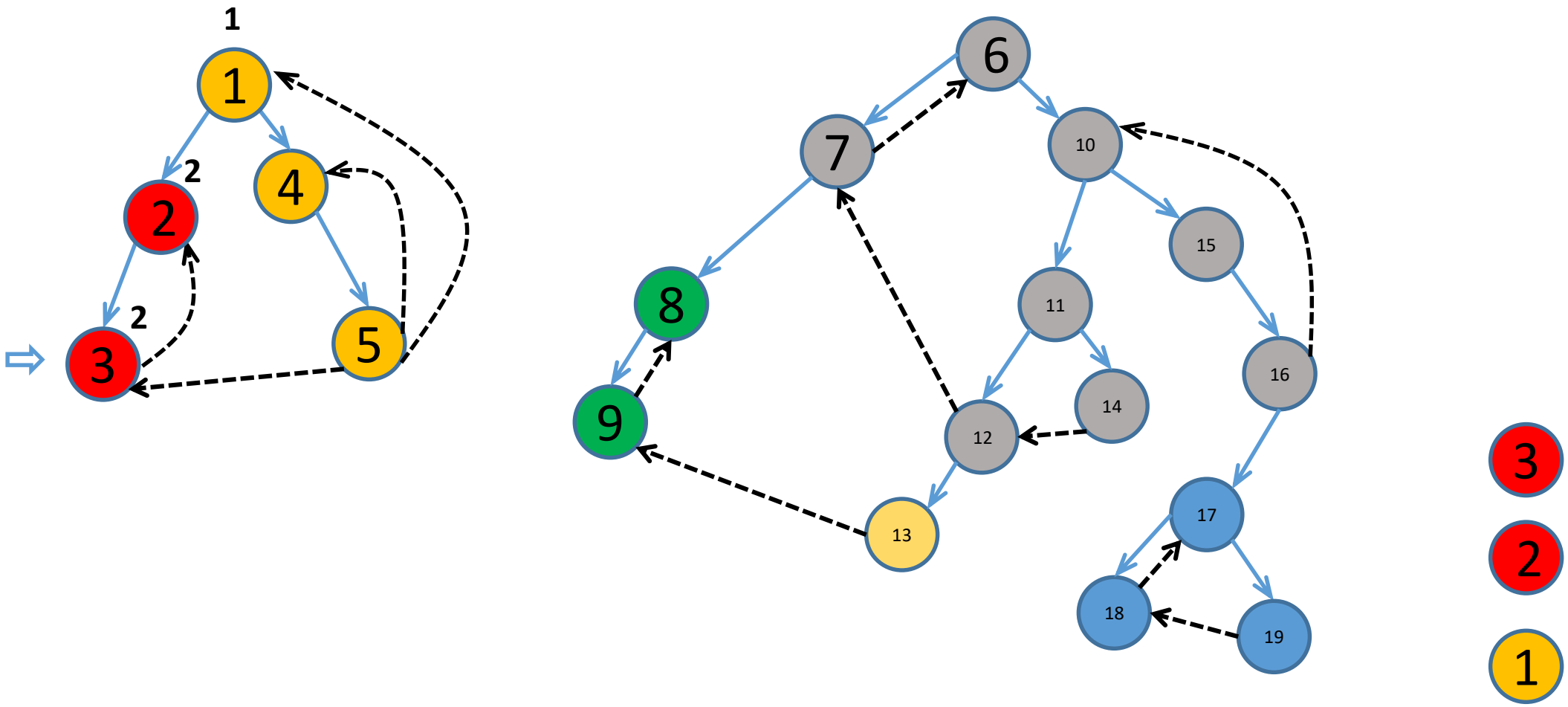


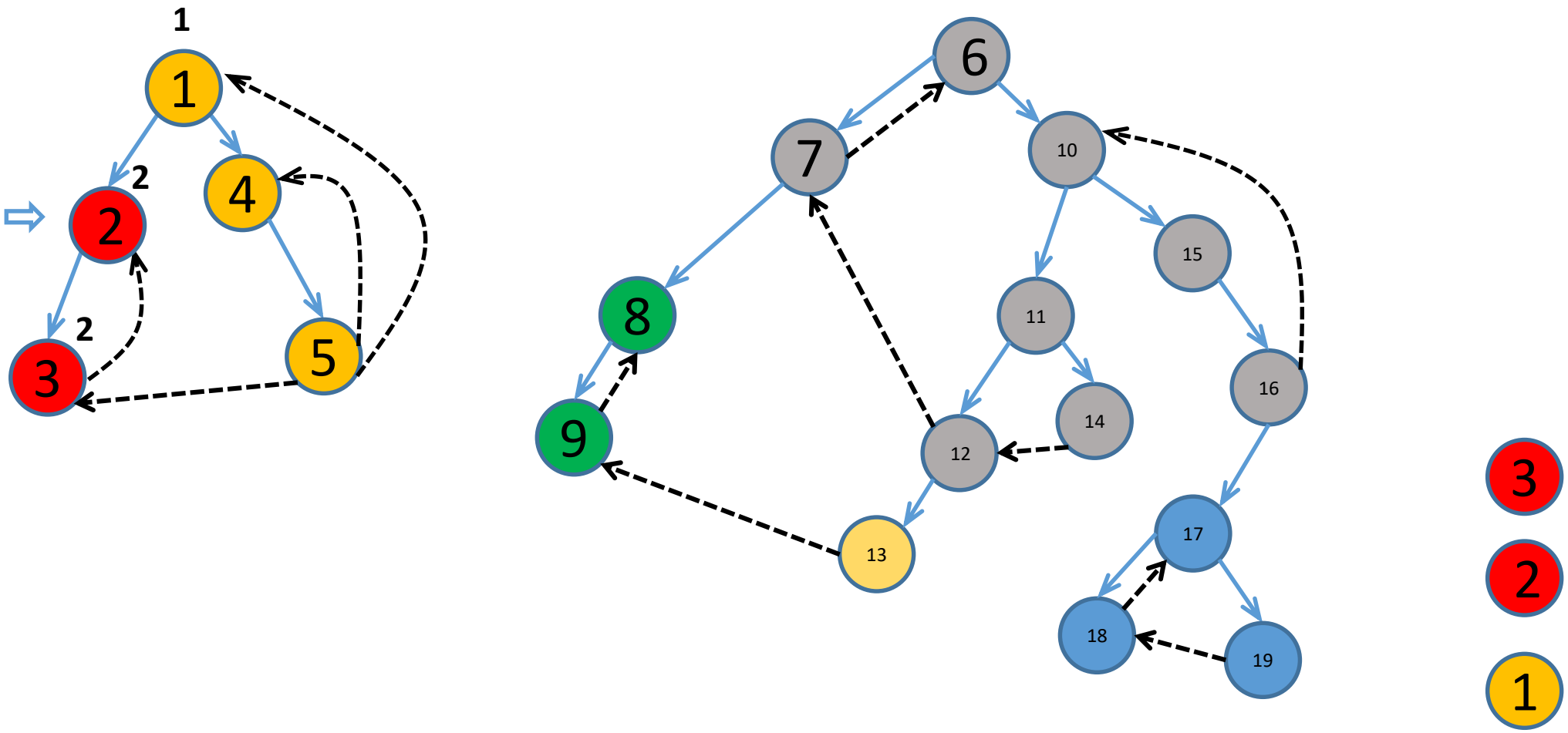


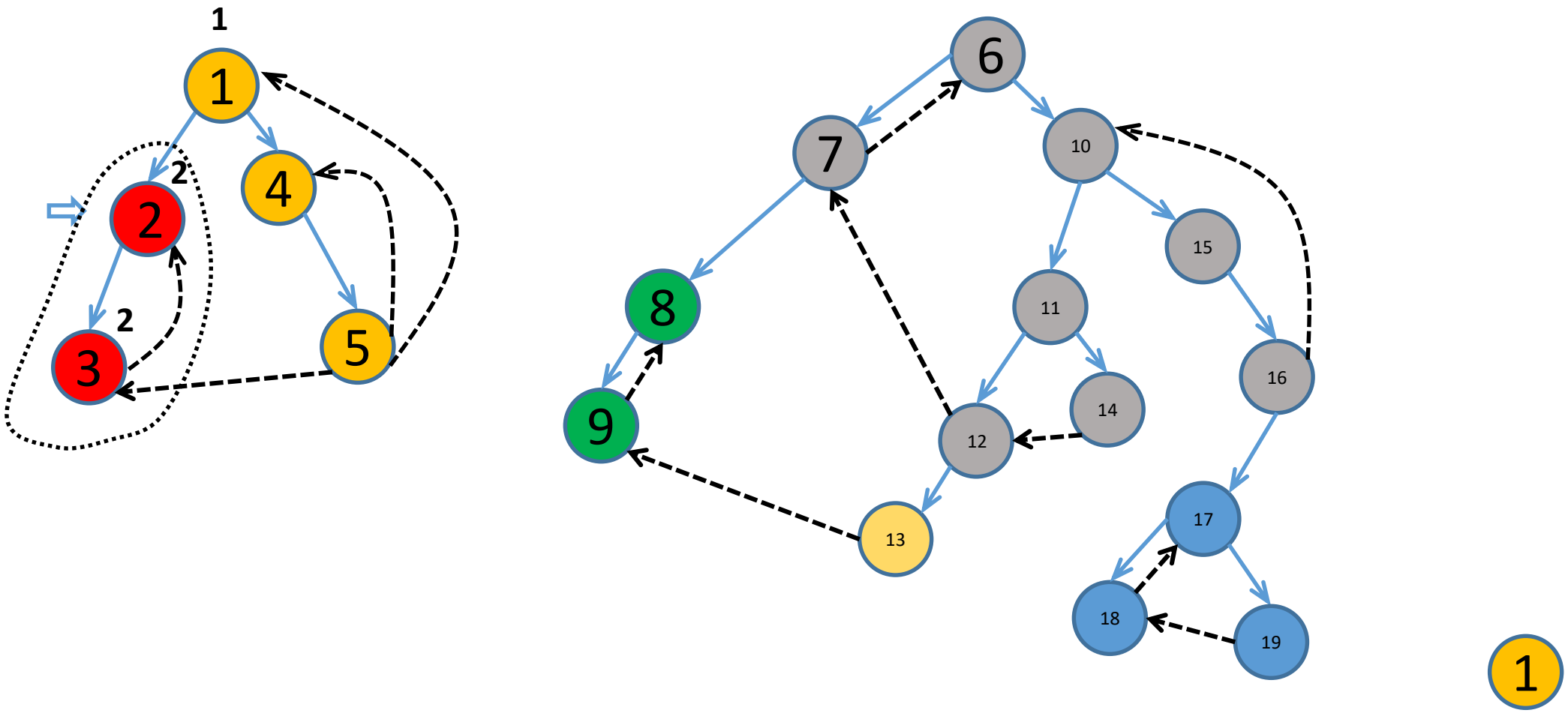
Compute SCs using low values

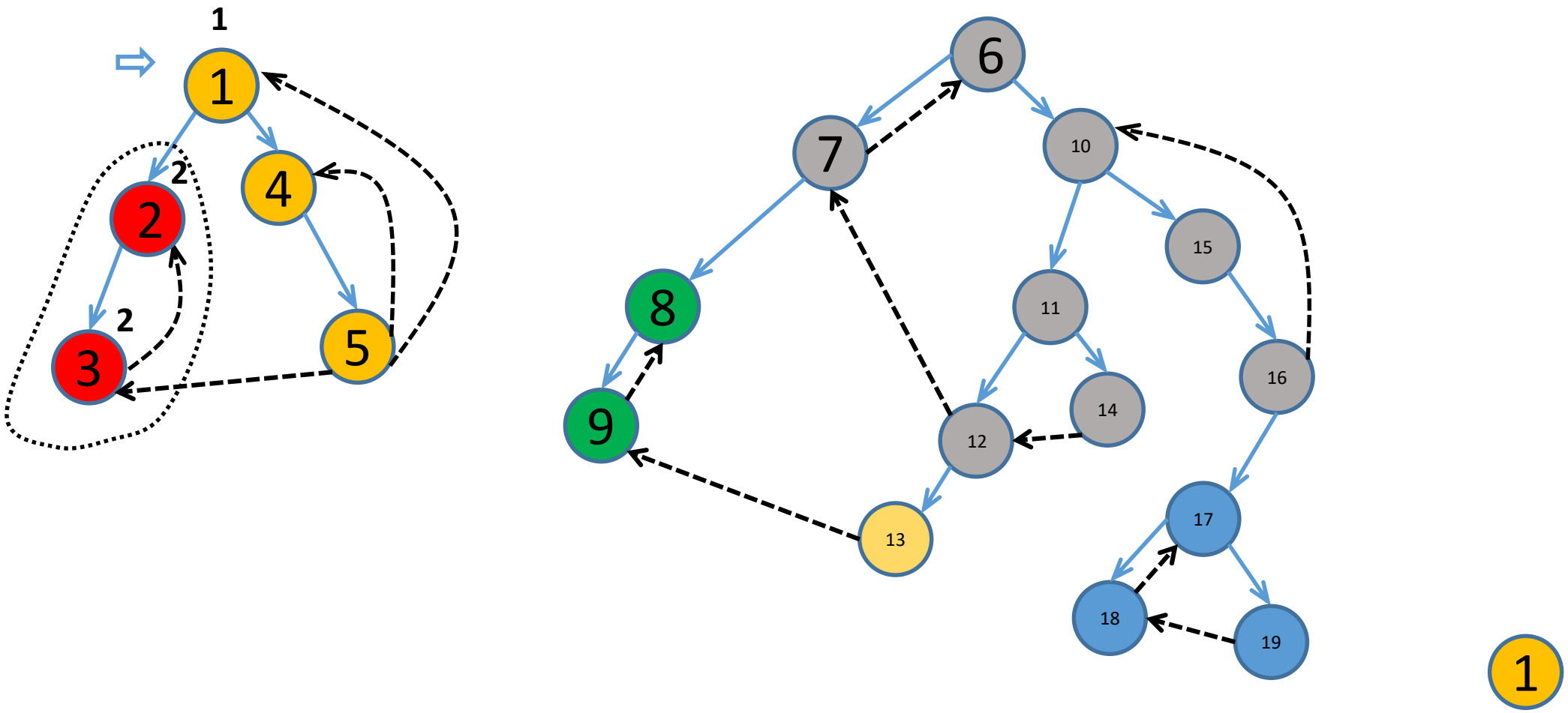


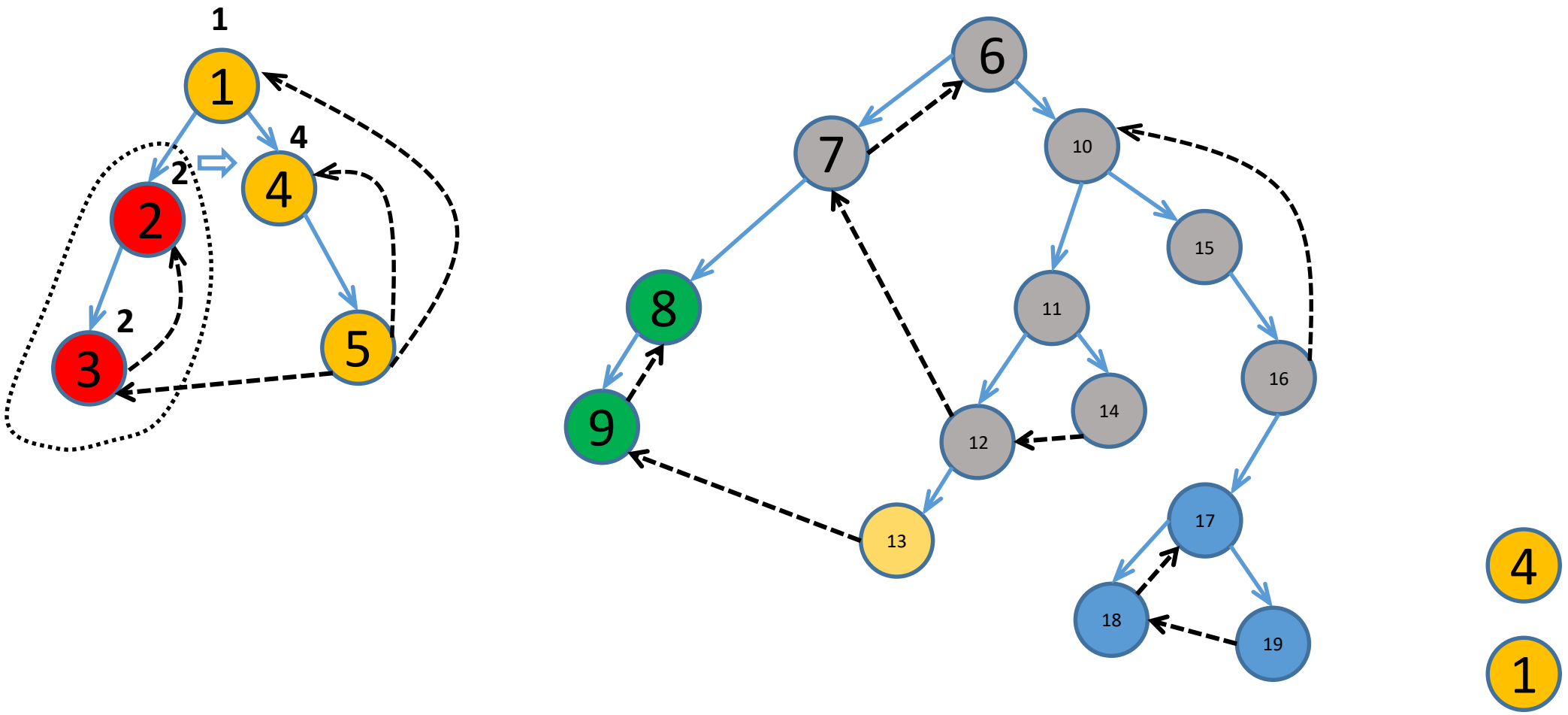

1
1
2
2
4
3
3
5
6
7
10
8
15
11
16
9
14
12
13
17
18
19
3
2
1

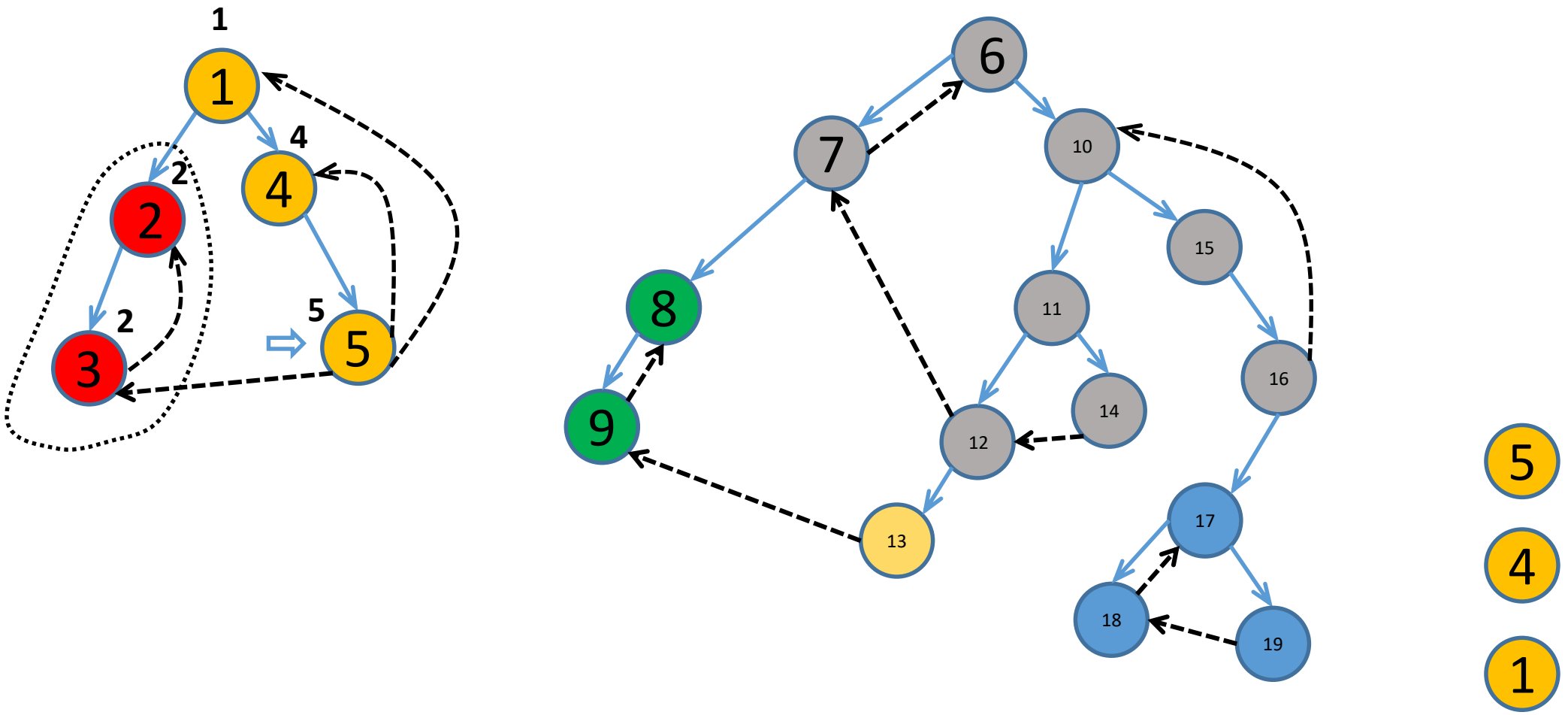

1
1
2
2
4
2
3
5
6
7
10
8
15
11
16
9
14
12
13
17
18
19
3
2
1

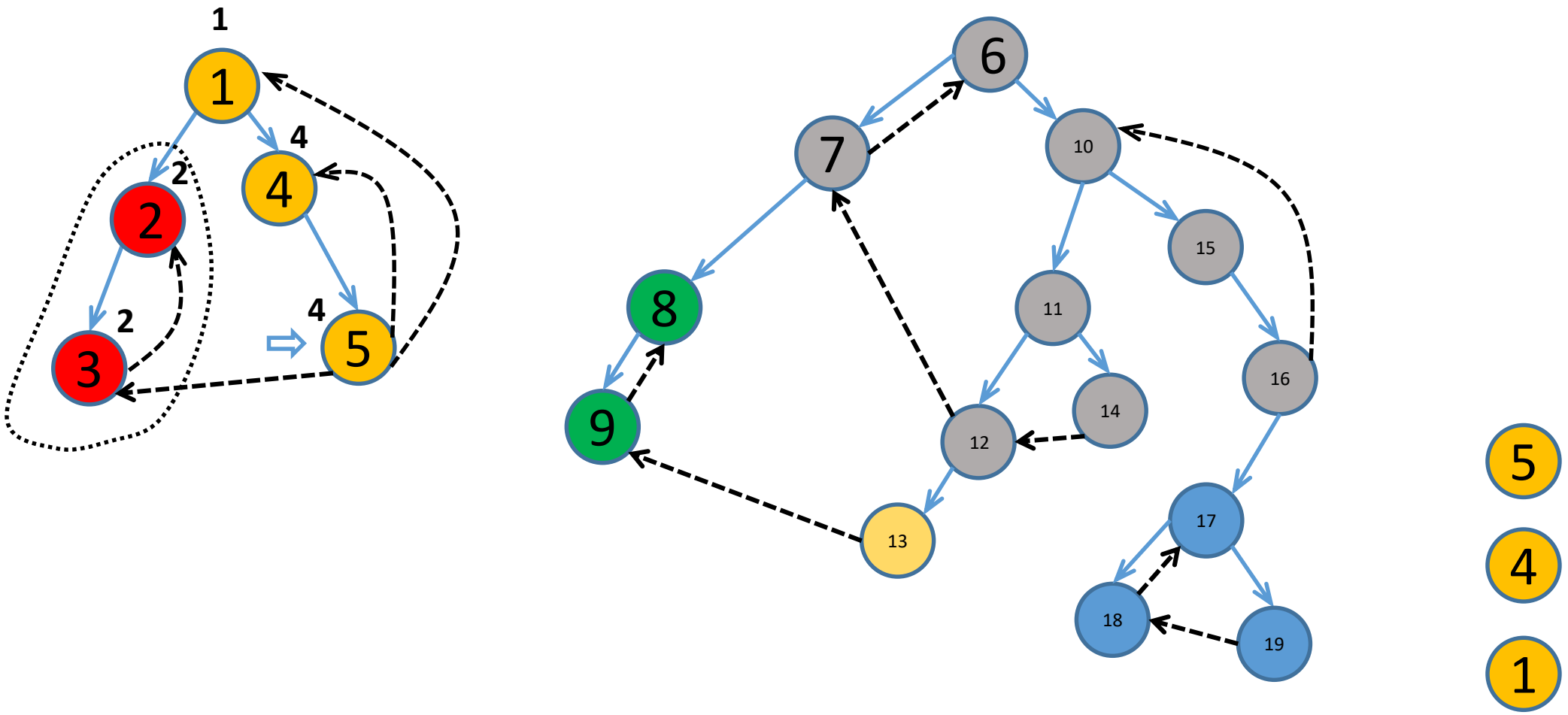

1
1
2
2
4
2
3
5
6
7
10
8
15
11
16
9
14
12
13
17
18
19
3
2
1

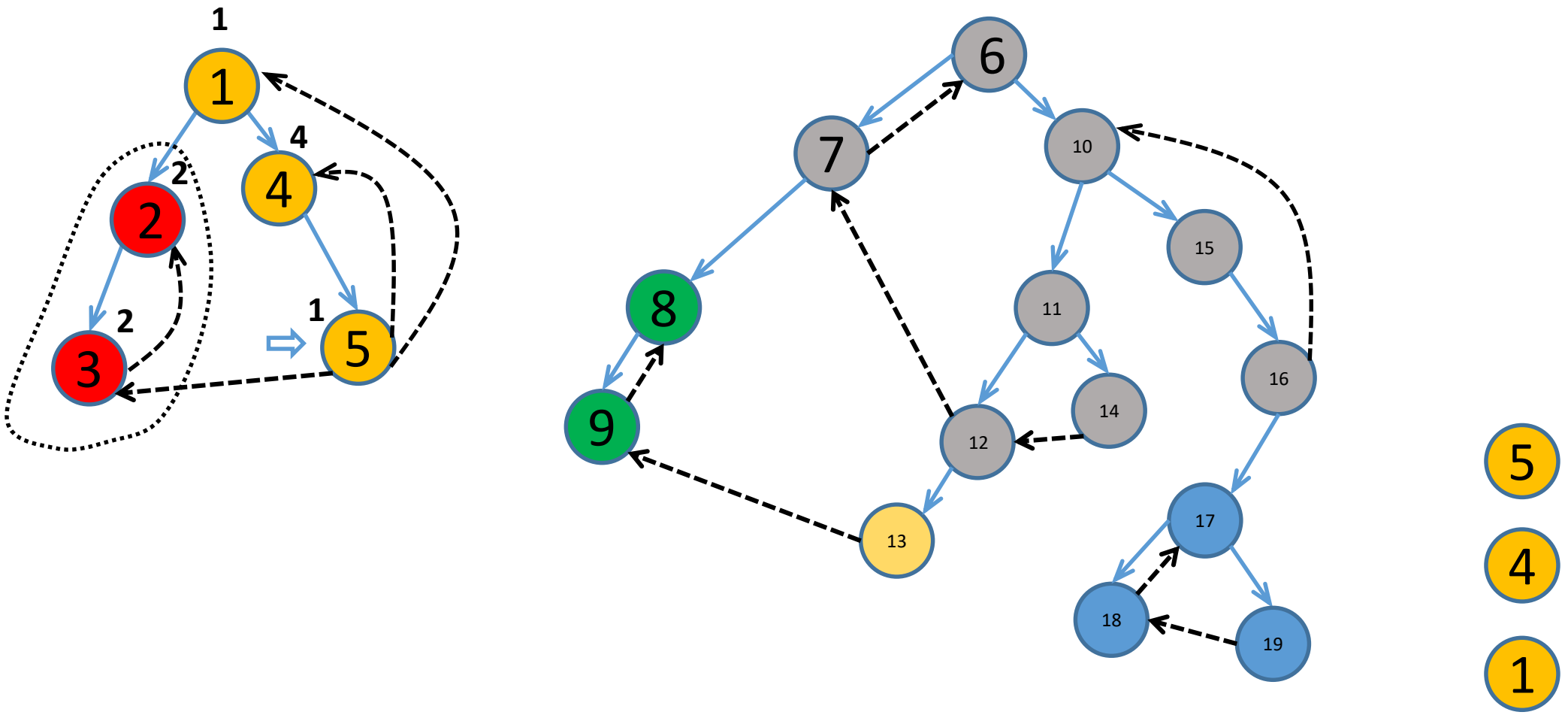

1
1
2
2
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
1

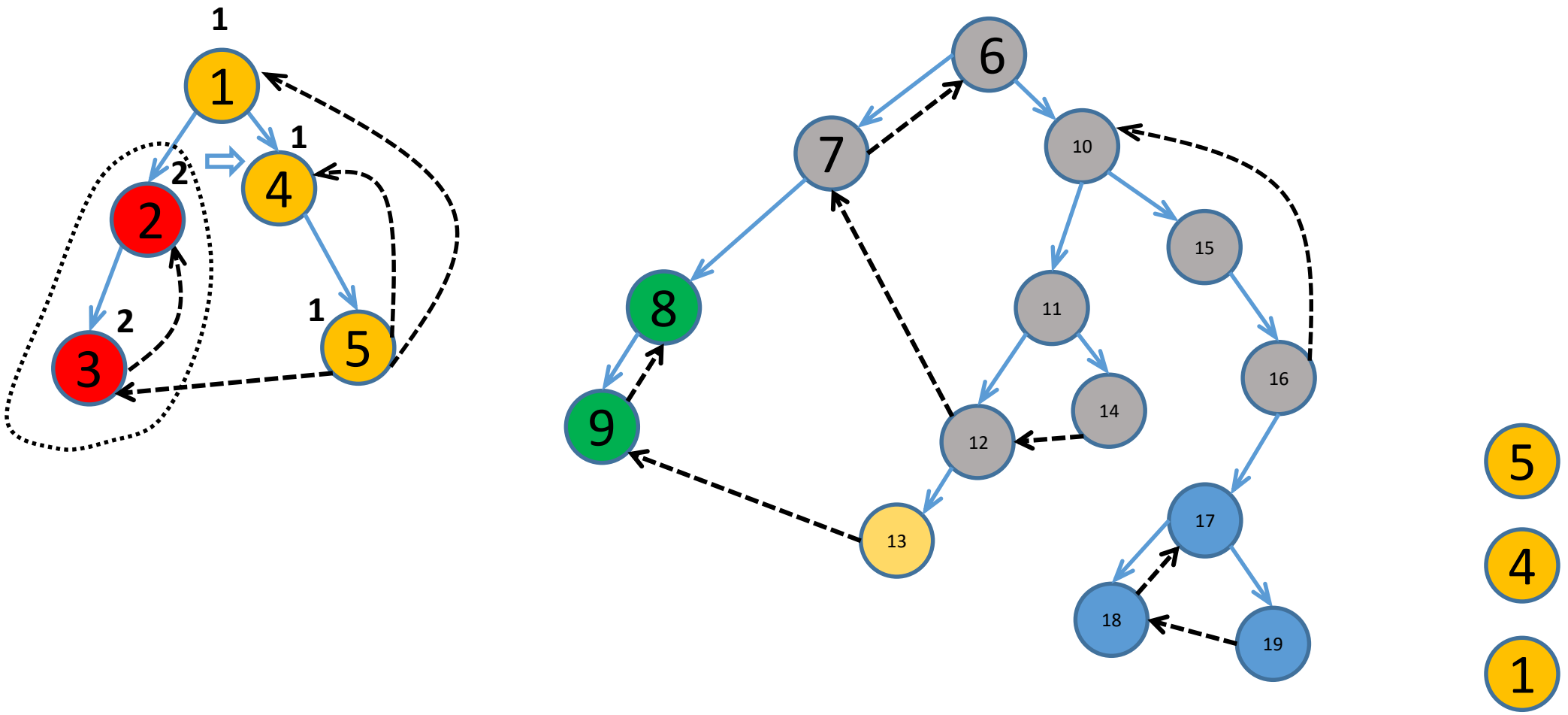

1
1
2
2
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
1

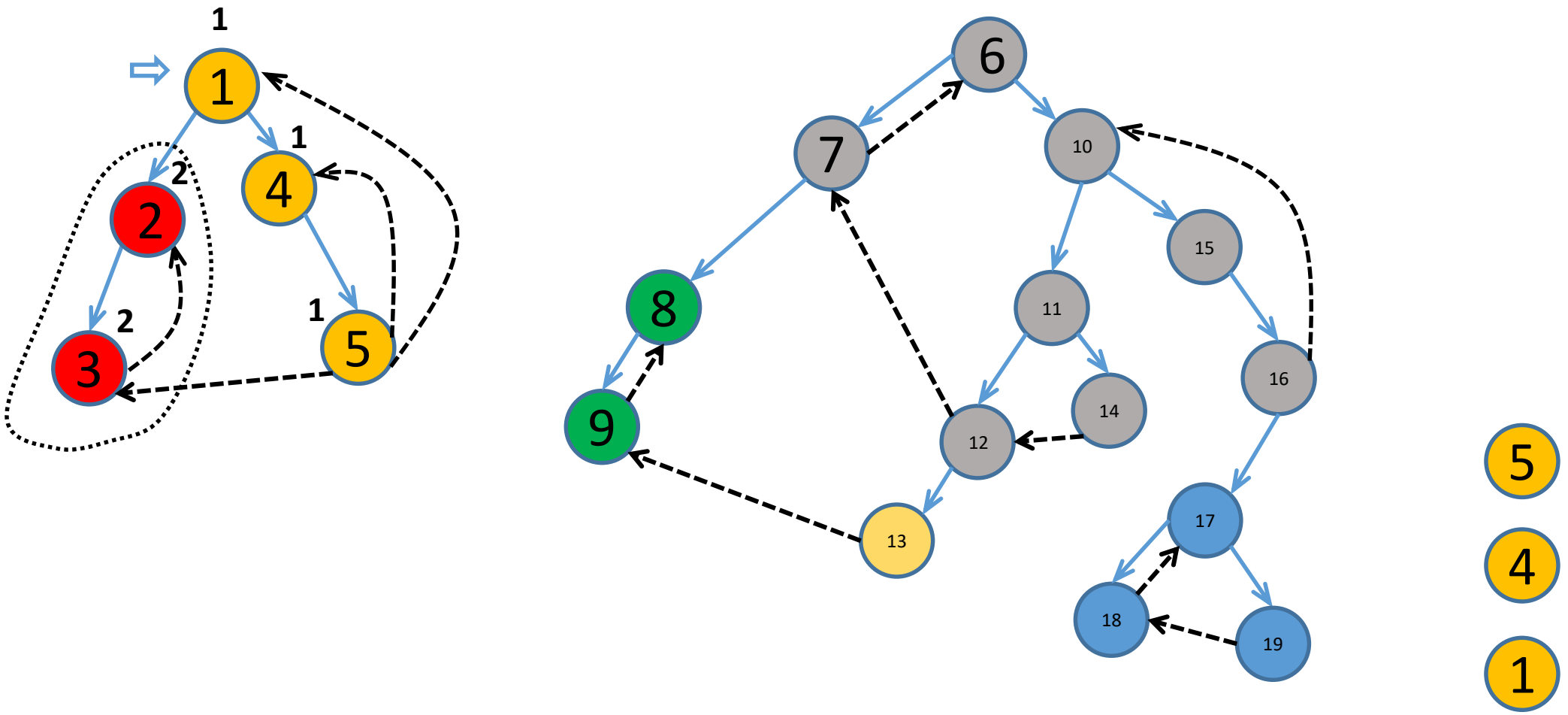

1
1
4
2
4
2
2
3
5
6
7
10
8
15
11
16
9
14
12
13
17
18
19
4
1

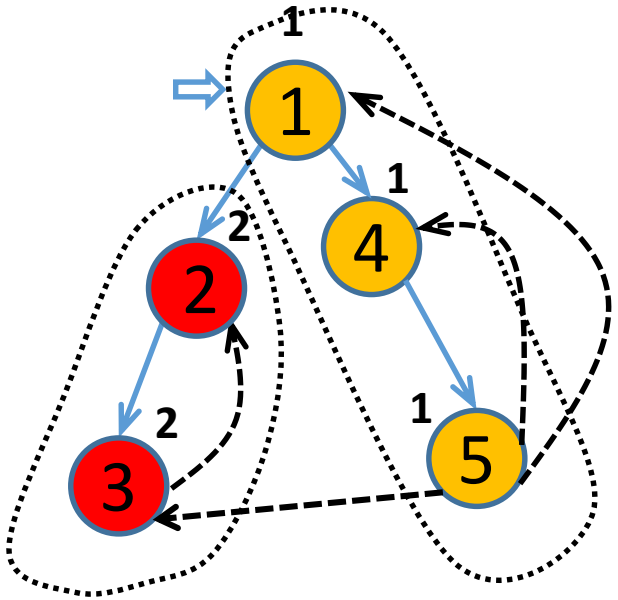

1
1
4
4
2
2
2
3
5
5
6
7
10
15
8
11
16
9
14
12
13
17
18
19
5
4
1

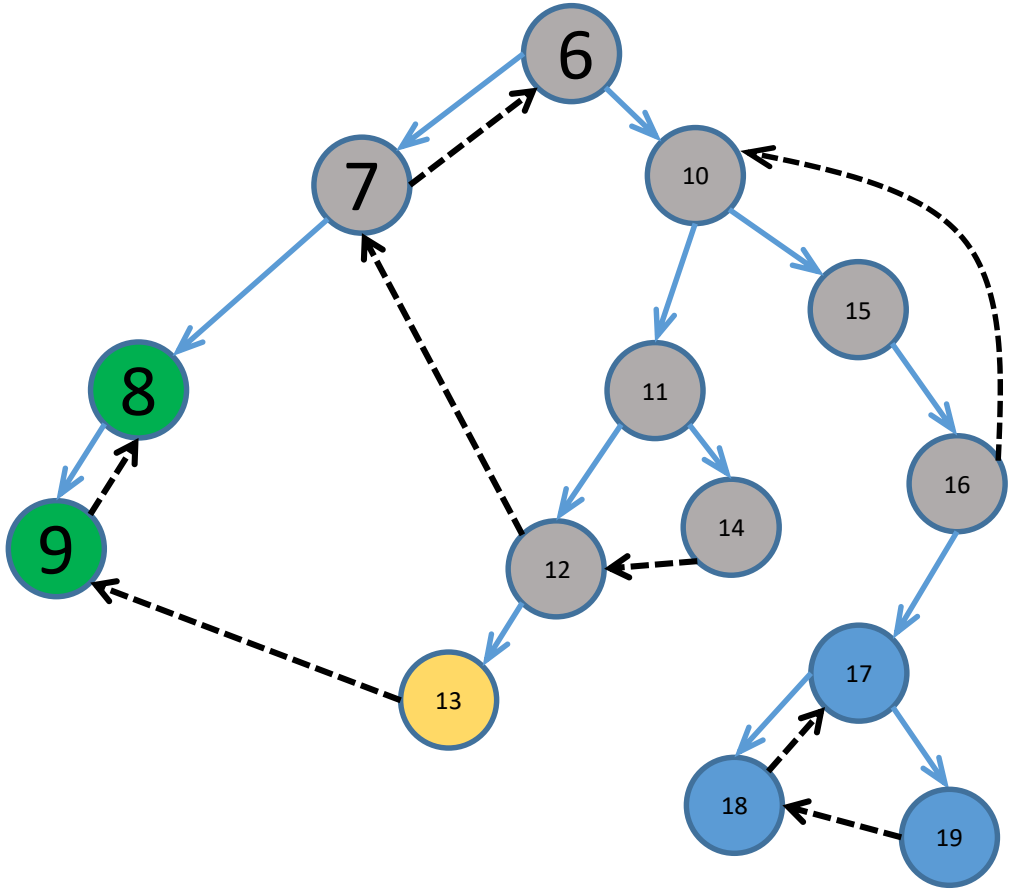

1
1
4
4
2
2
2
3
4
5
6
7
10
15
8
11
16
9
14
12
13
17
18
19
5
4
1

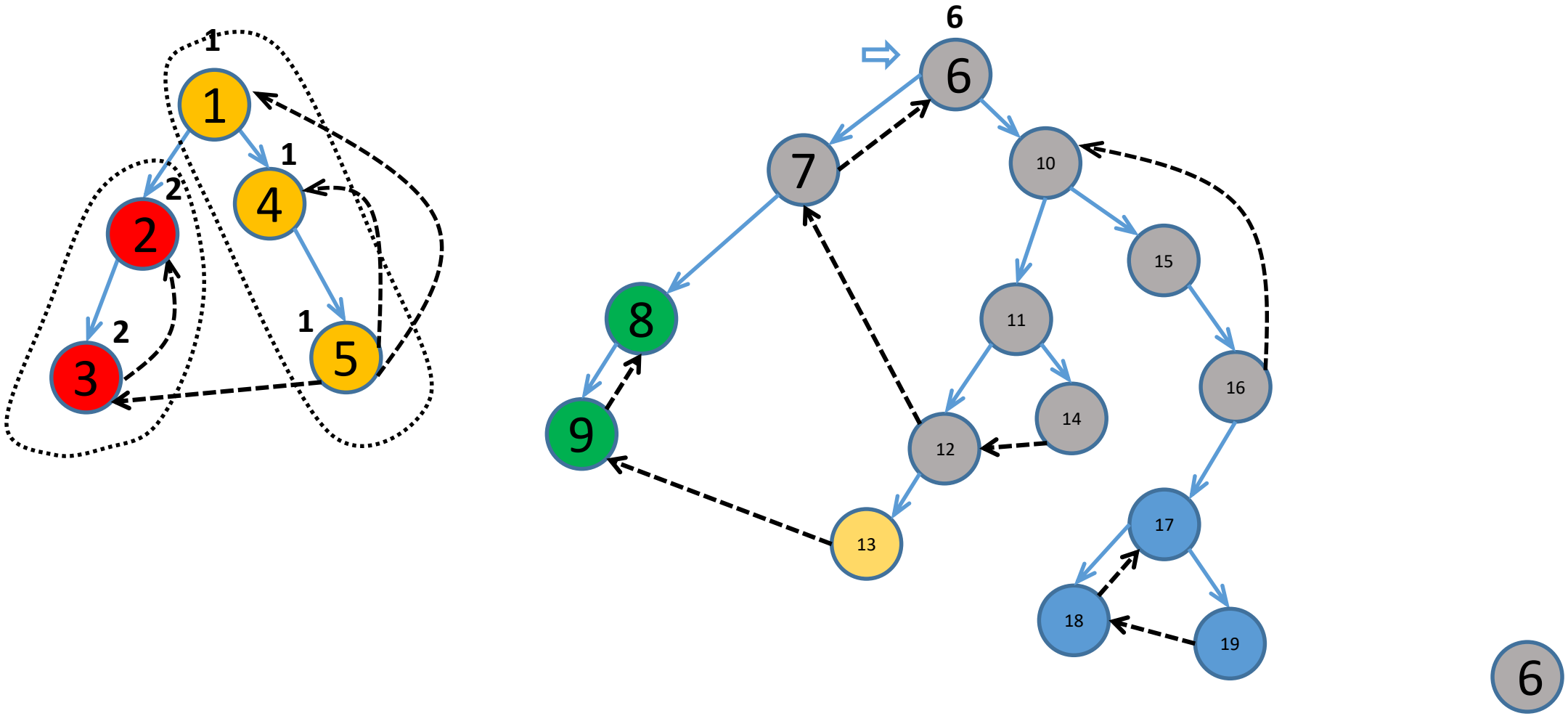

1
1
4
2
4
2
2
1
3
5
6
7
10
15
8
11
16
9
14
12
13
17
18
19
5
4
1

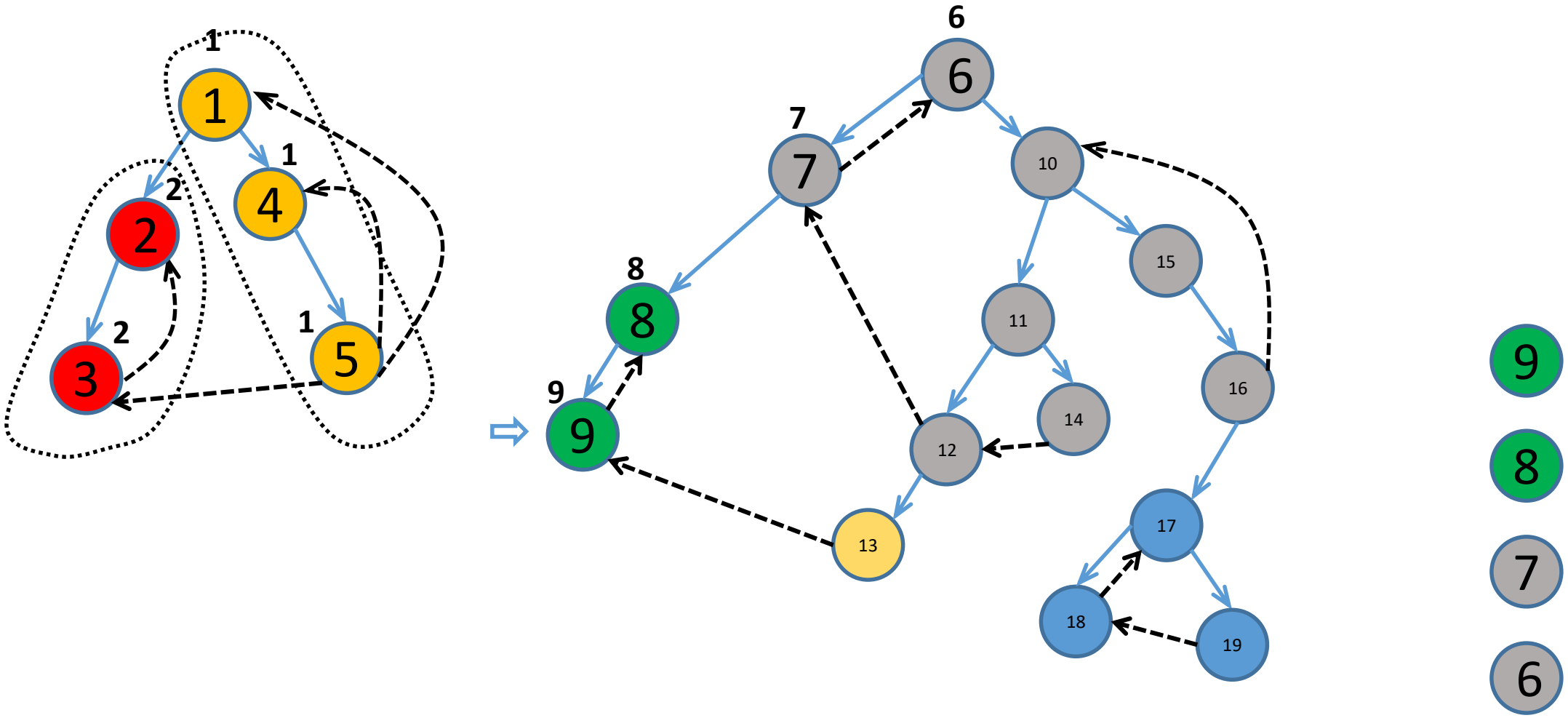

1
1
1
2
4
2
2
1
5
3
6
7
10
15
8
11
16
9
14
12
13
17
18
19
5
4
1

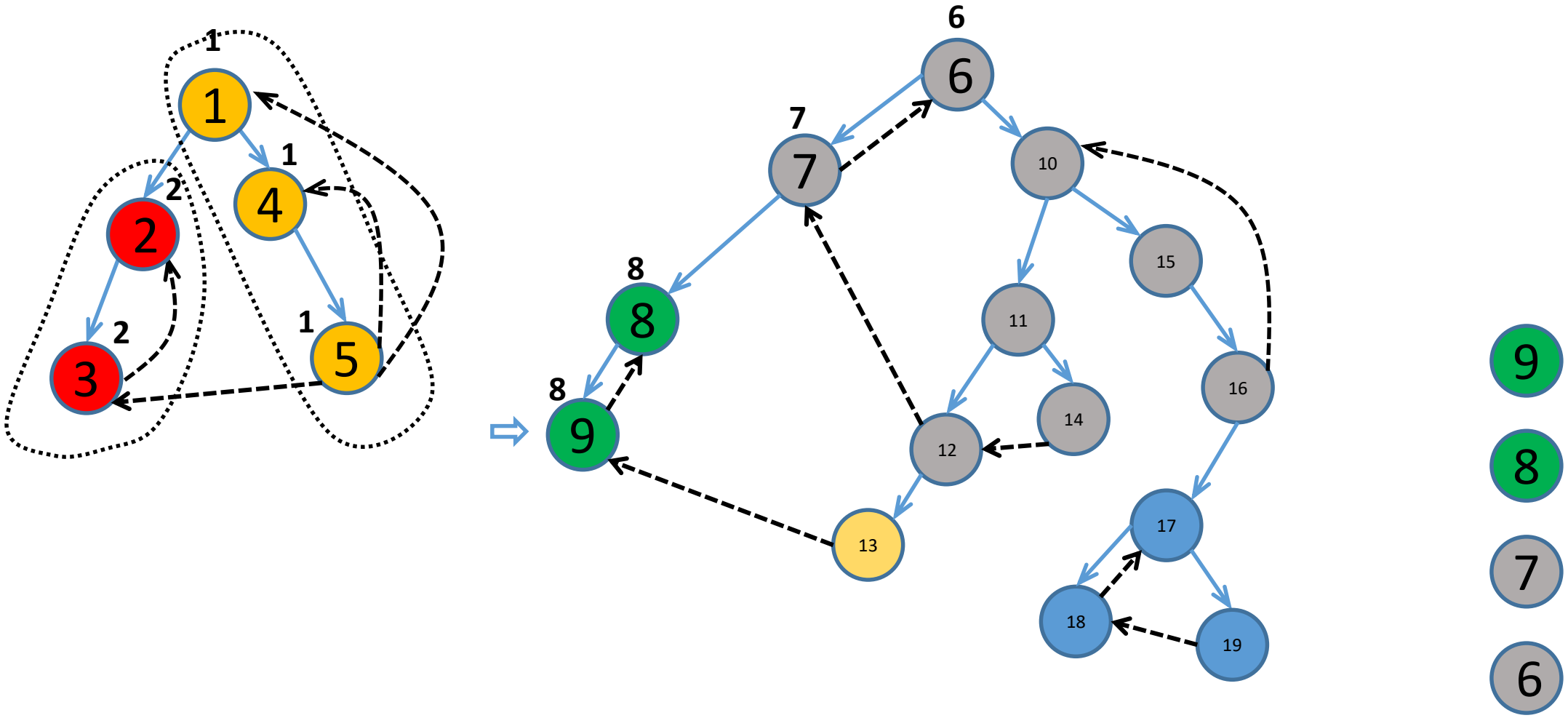

1
1
2
4
2
1
3
5
6
7
10
8
15
11
16
9
14
12
13
17
18
19
5
4
1

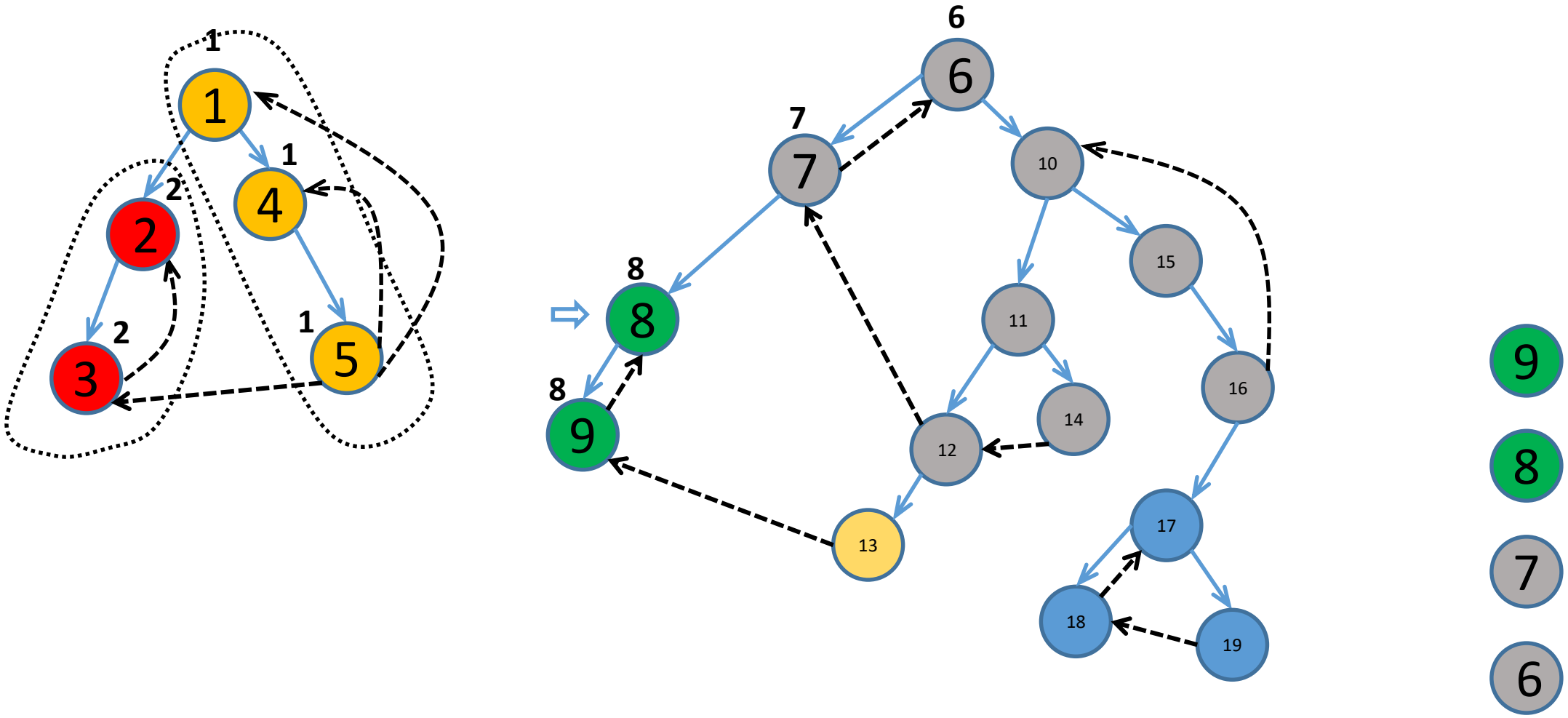

1
1
2
1
2
1
2
3
4
5

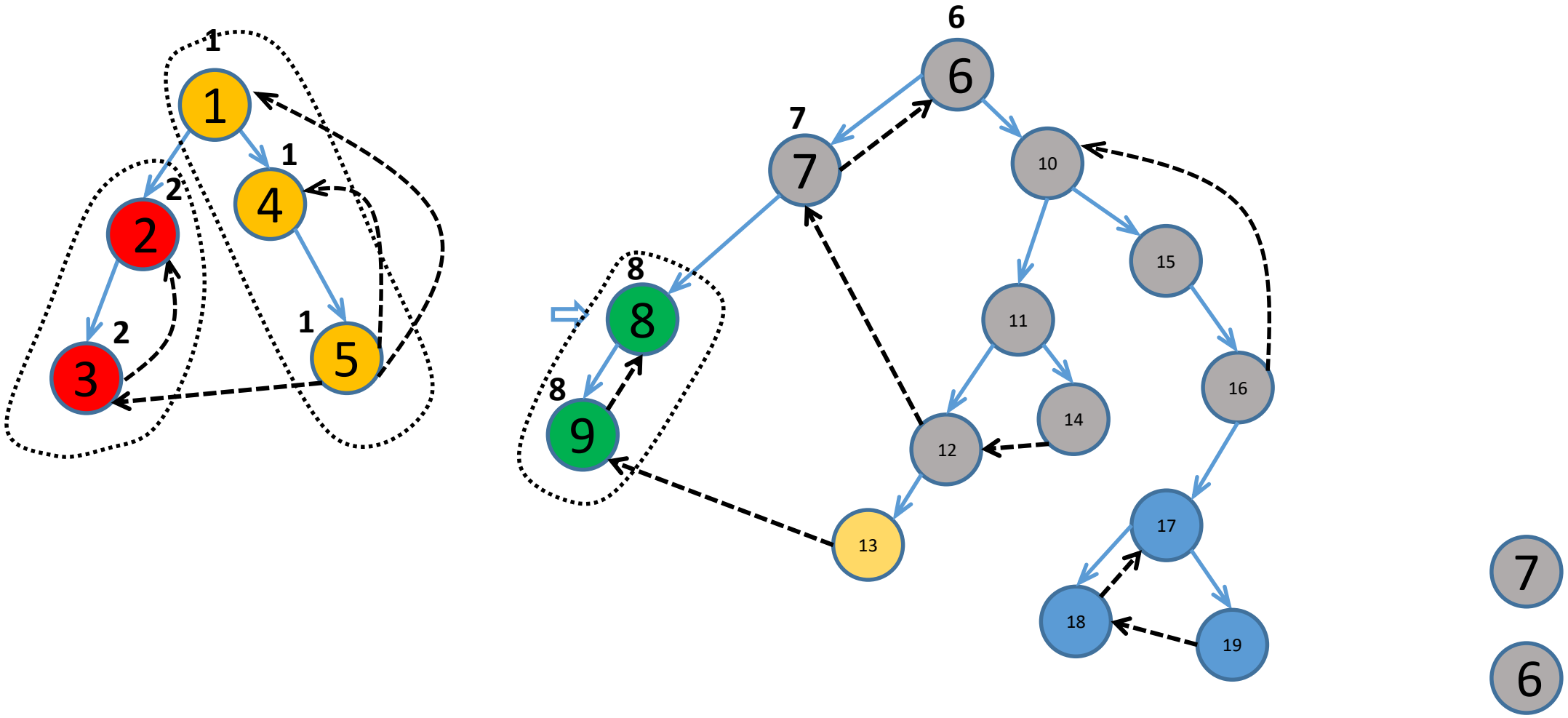

6
7
10
15
8
11
16
9
14
12
13
17
18
19

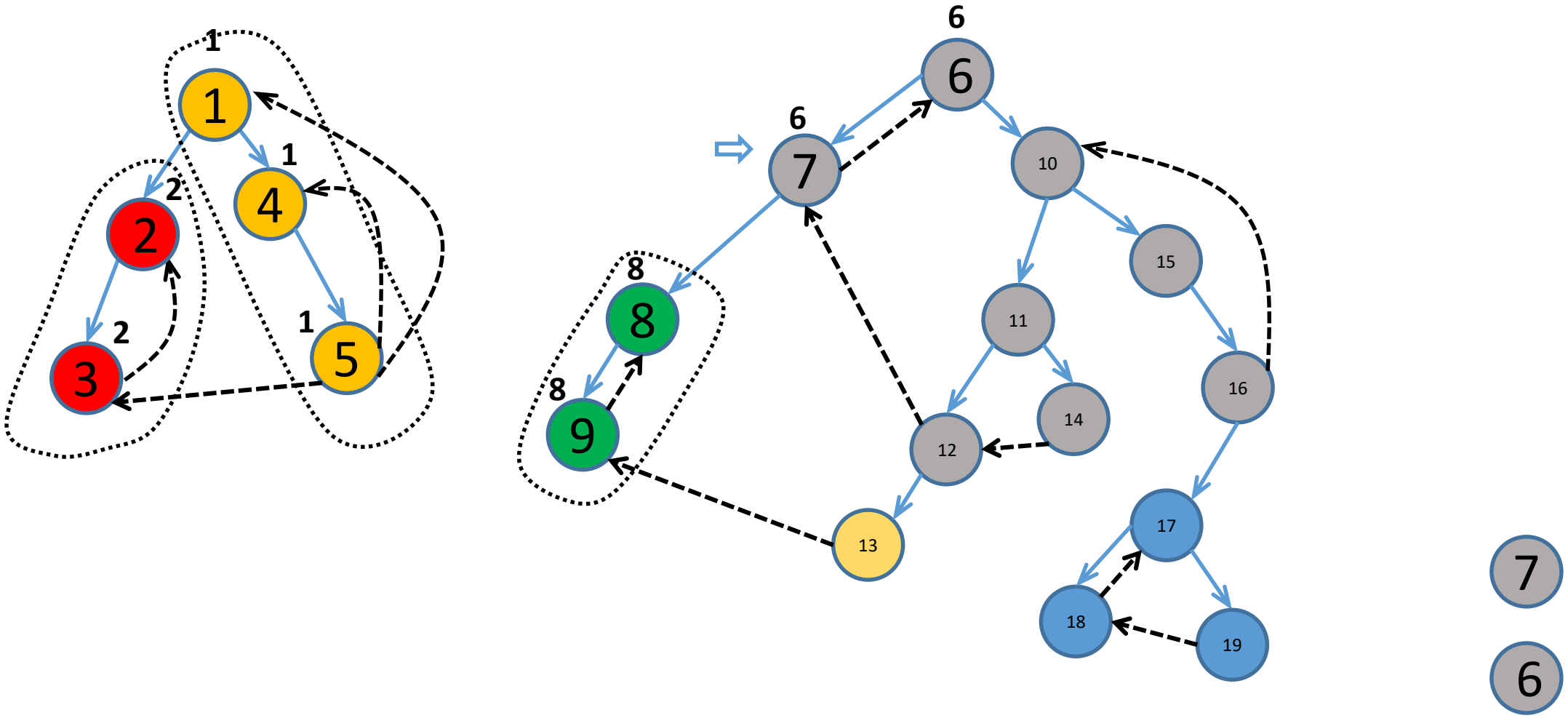

1
1
2
1
2
1
6
6
7
10
15
8
11
16
9
14
12
13
17
18
19
6

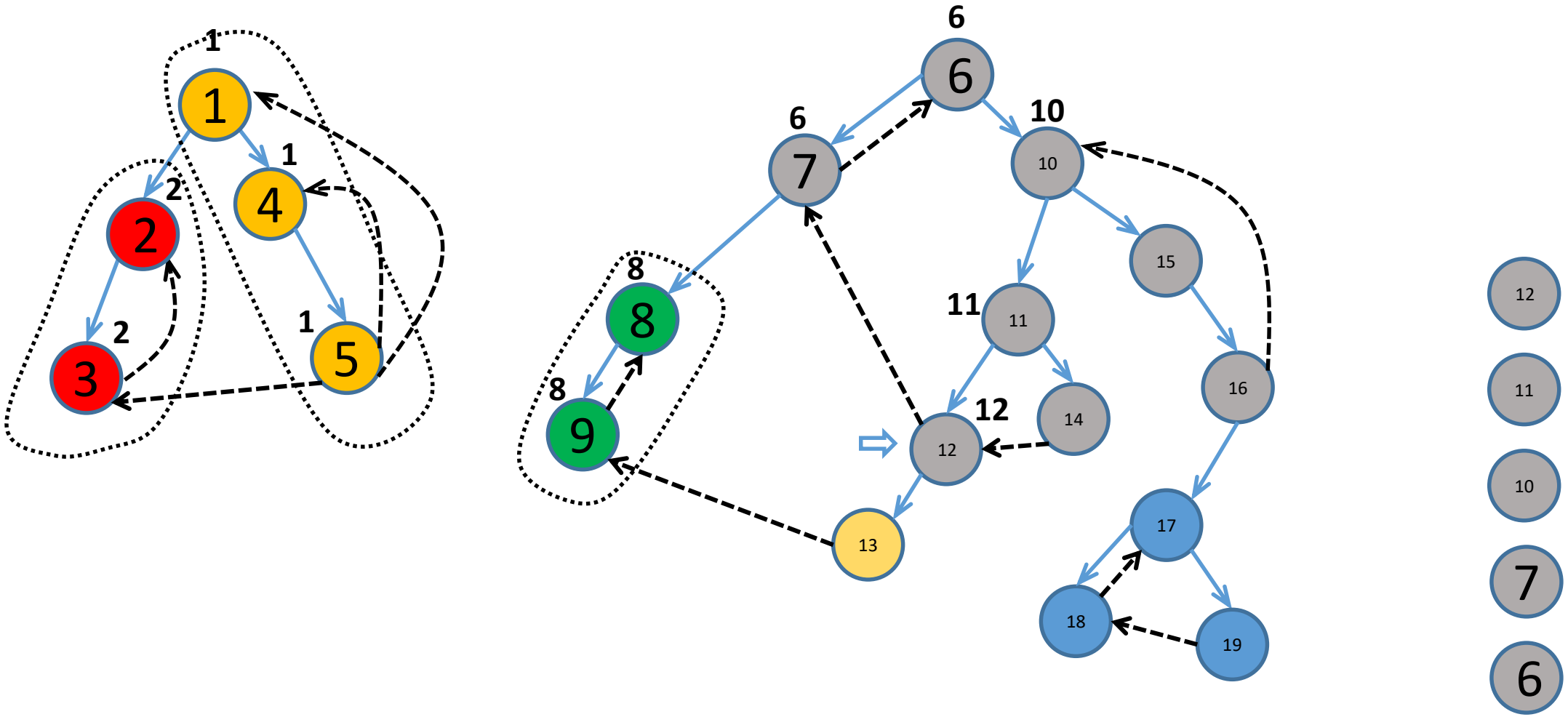

1
1
2
1
4
2
3
1
5
6
6
7
7
10
8
8
15
11
9
9
16
14
12
13
17
18
19
9
8
7
6

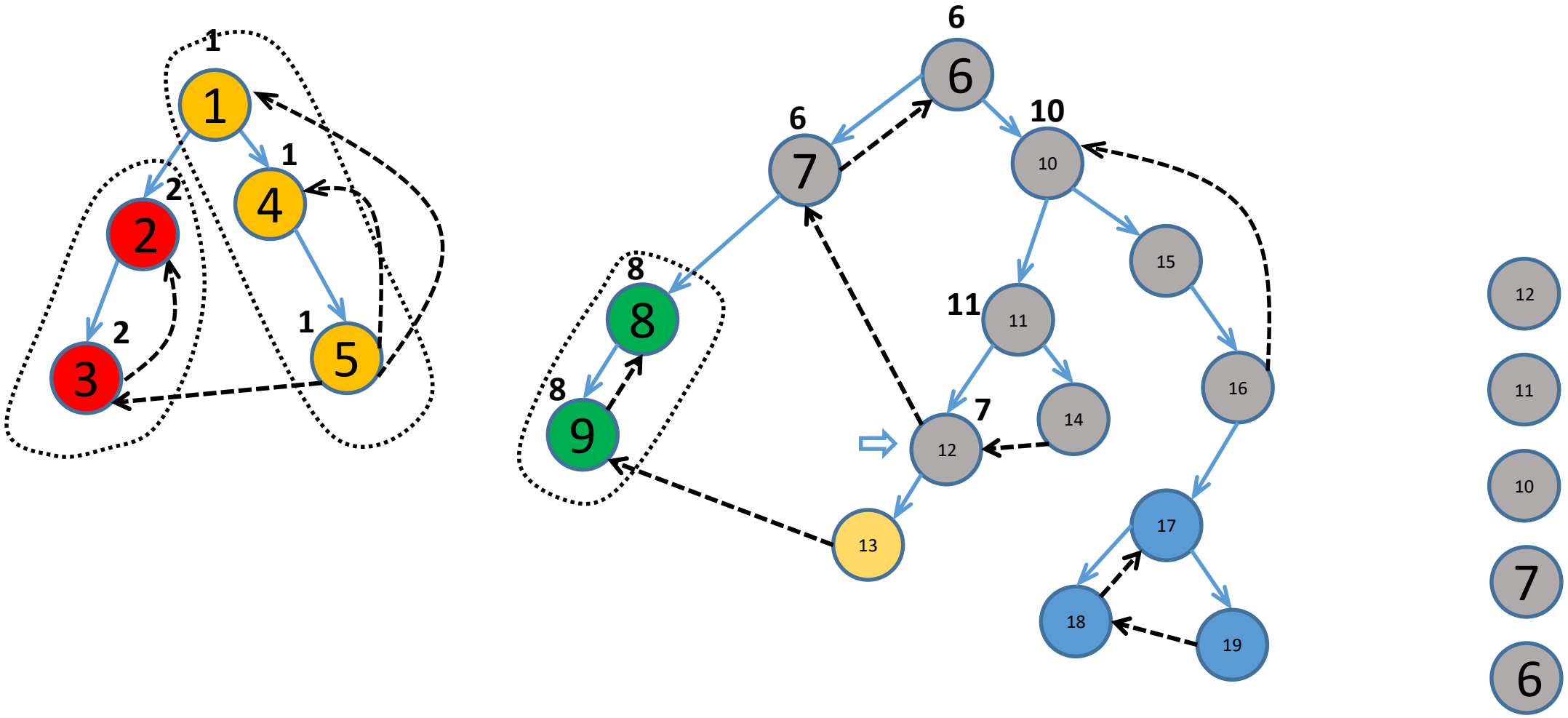

1
1
2
1
4
2
3
1
5
6
6
7
7
10
8
8
15
11
8
9
16
14
12
13
17
18
19
9
8
7
6

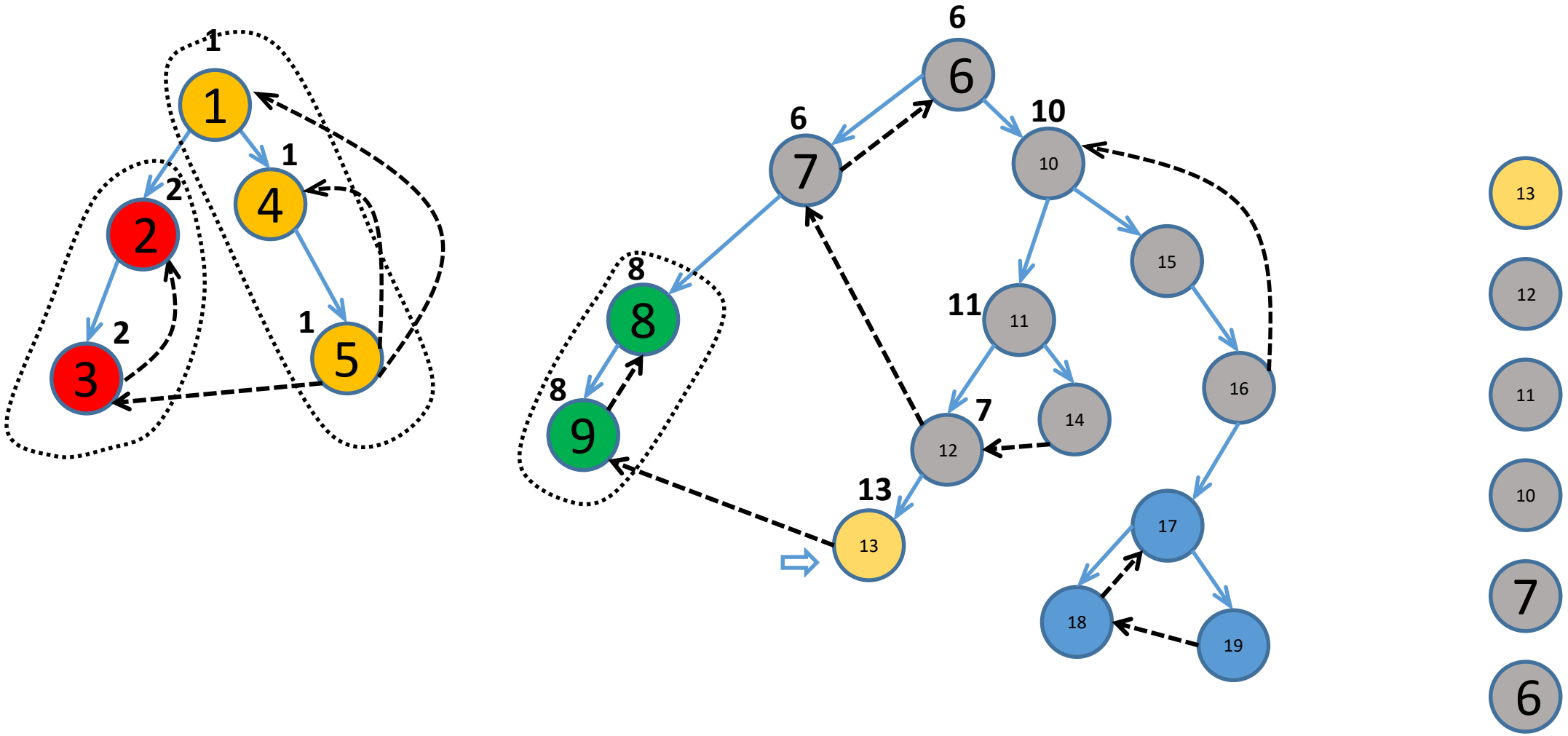

1
1
2
1
4
2
3
1
5
6
6
7
7
10
8
8
9
15
11
16
14
12
13
17
18
19
9
8
7
6

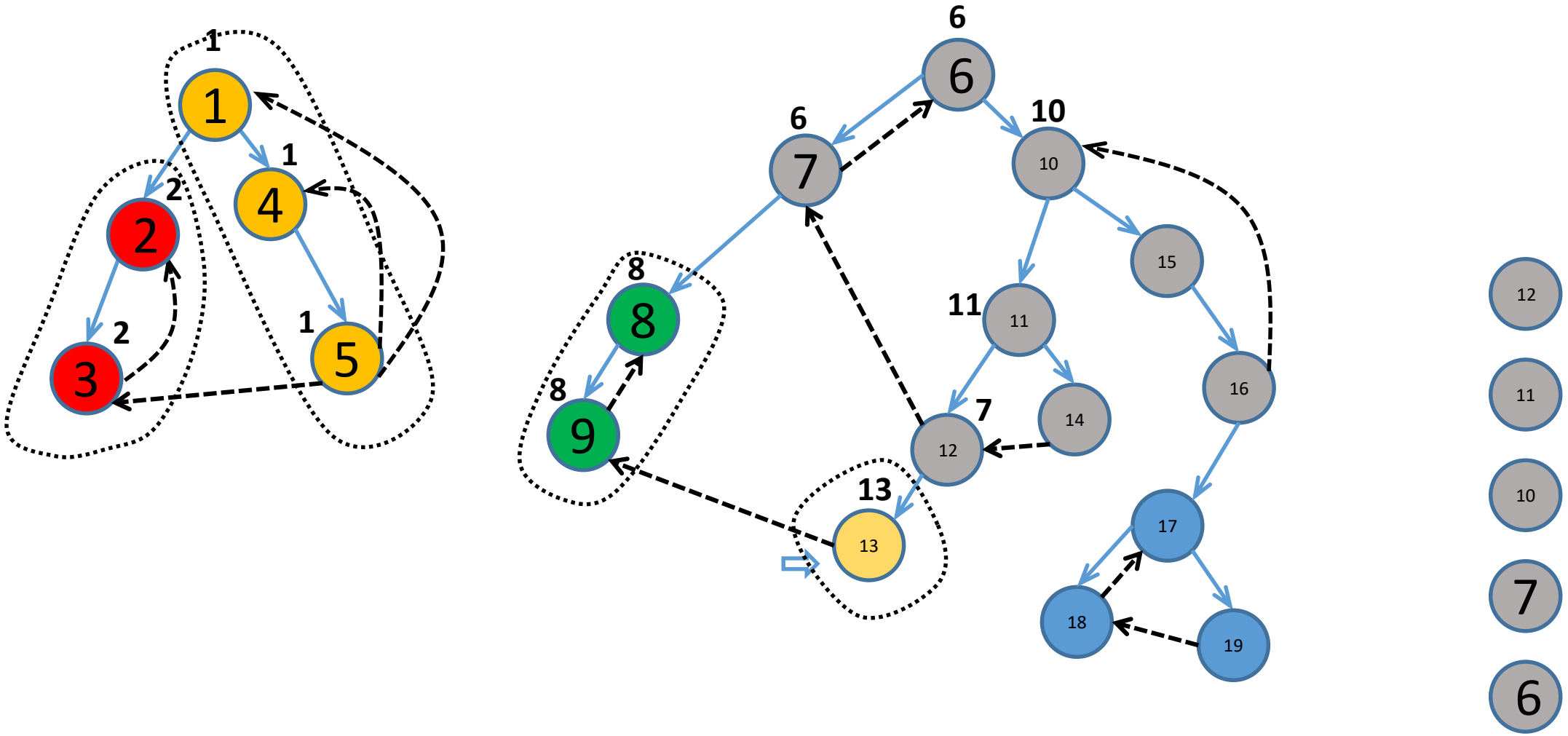

1
1
2
1
4
2
3
1
5
6
6
7
7
10
8
8
8
9
15
11
16
14
12
13
17
18
19
7
6

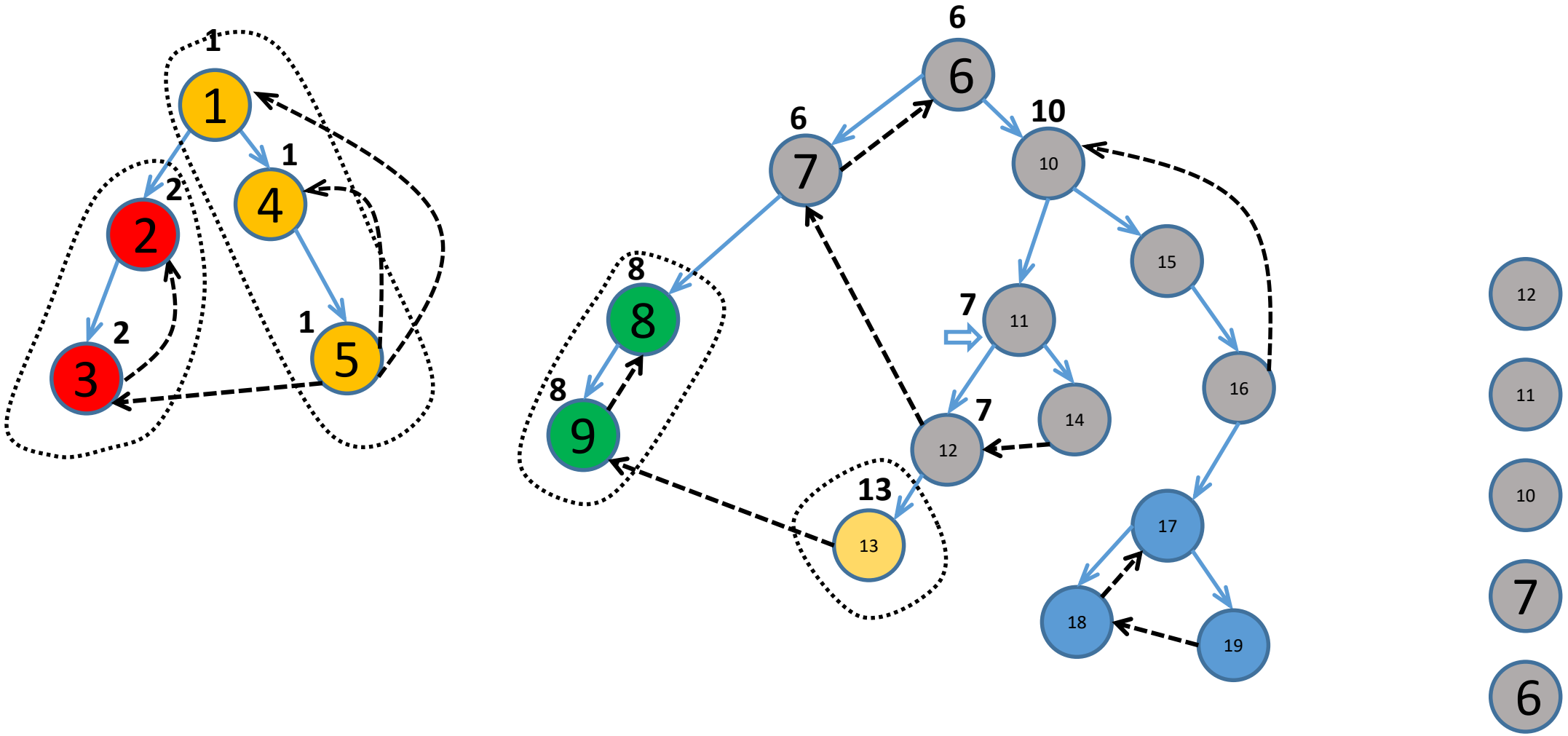

1
1
2
2
1
1
2
3
4
5
6
6
7
8
8
9
10
11
12
13
14
15
16
17
18
19
7
6

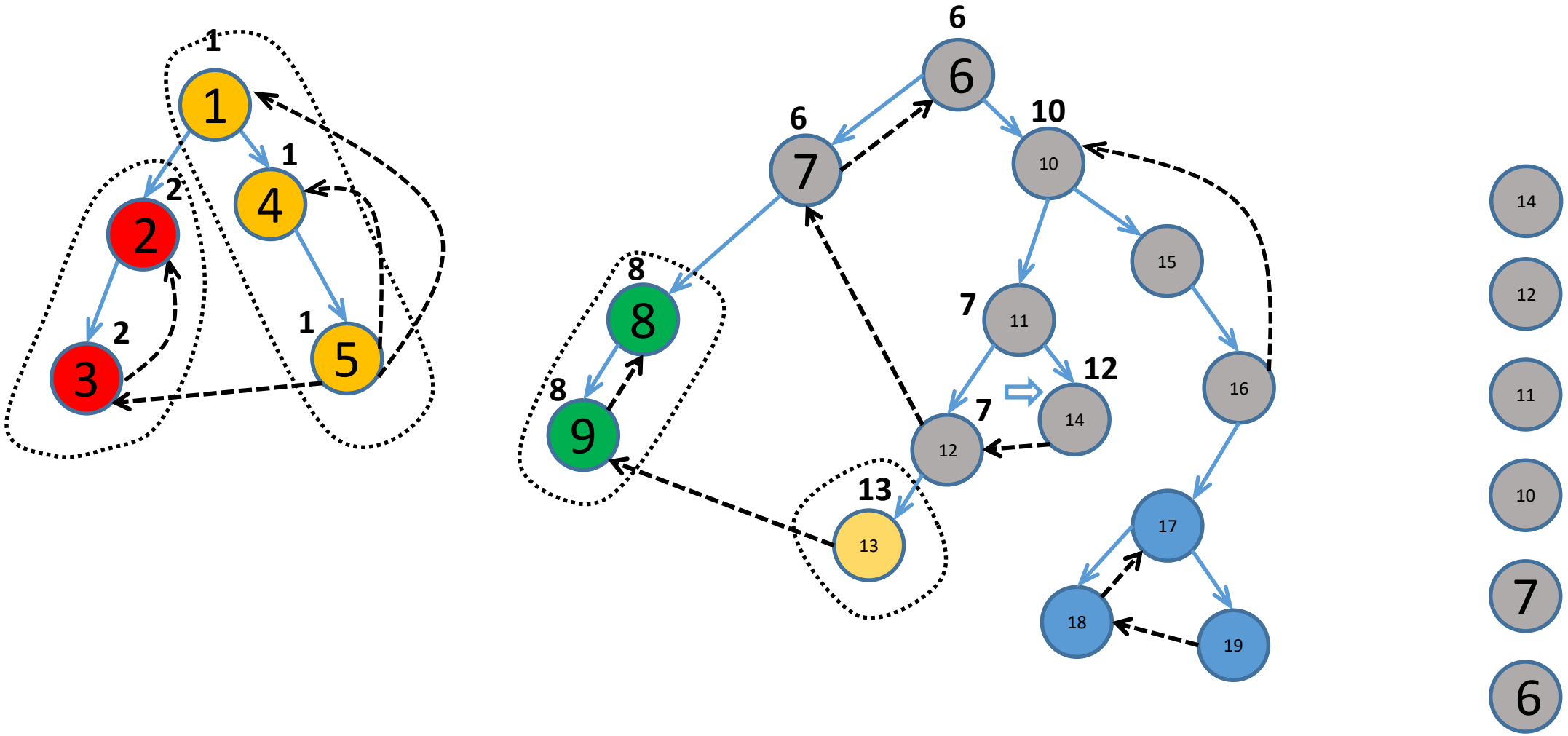

1
1
2
2
1
1
2
3
4
5
6
6
6
7
10
10
8
8
8
9
11
11
12
12
13
14
15
16
17
18
19
12
11
10
7
6

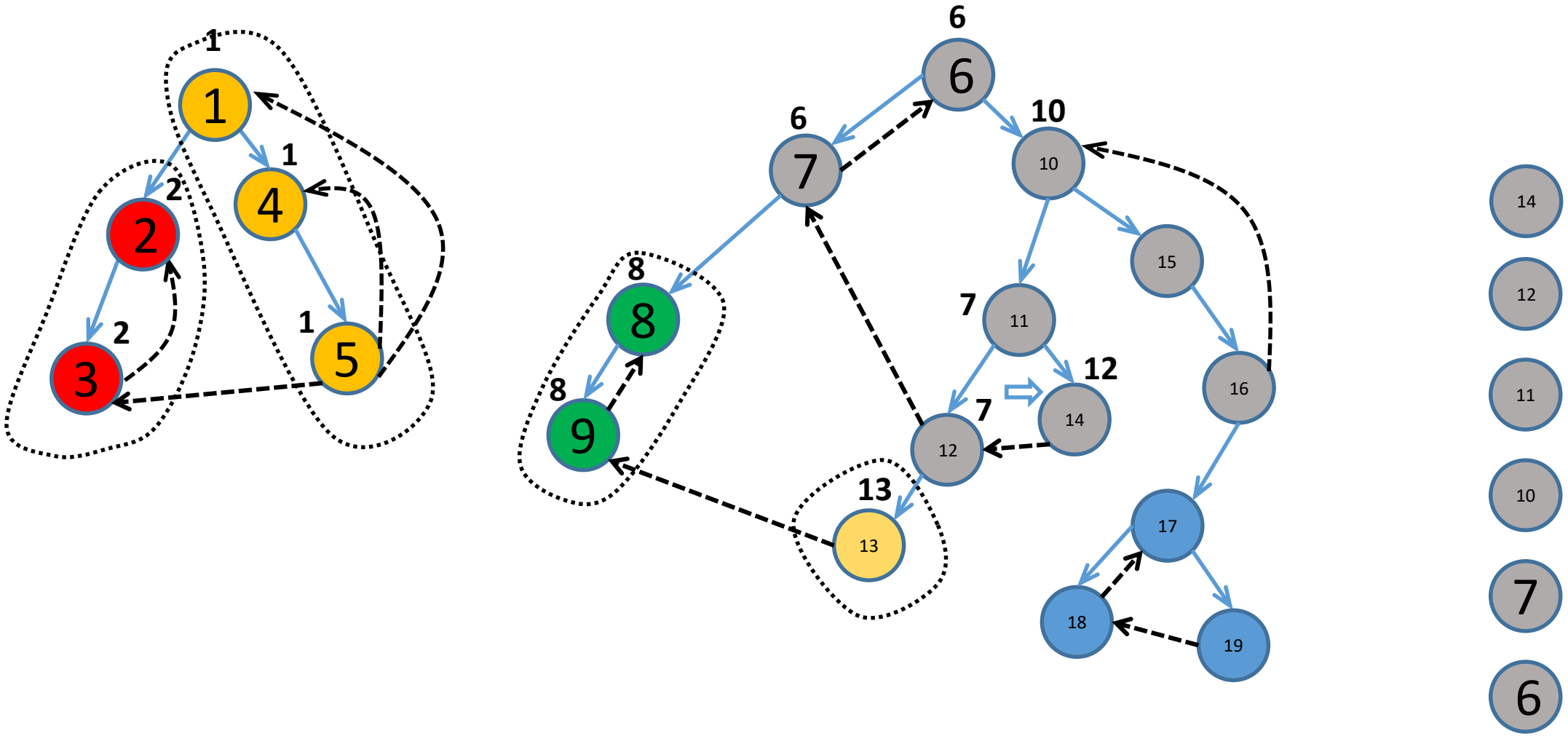

1
1
2
2
1
1
2
3
4
5
6
6
6
7
10
10
8
8
8
9
11
11
7
12
13
14
15
16
17
18
19
12
11
10
7
6

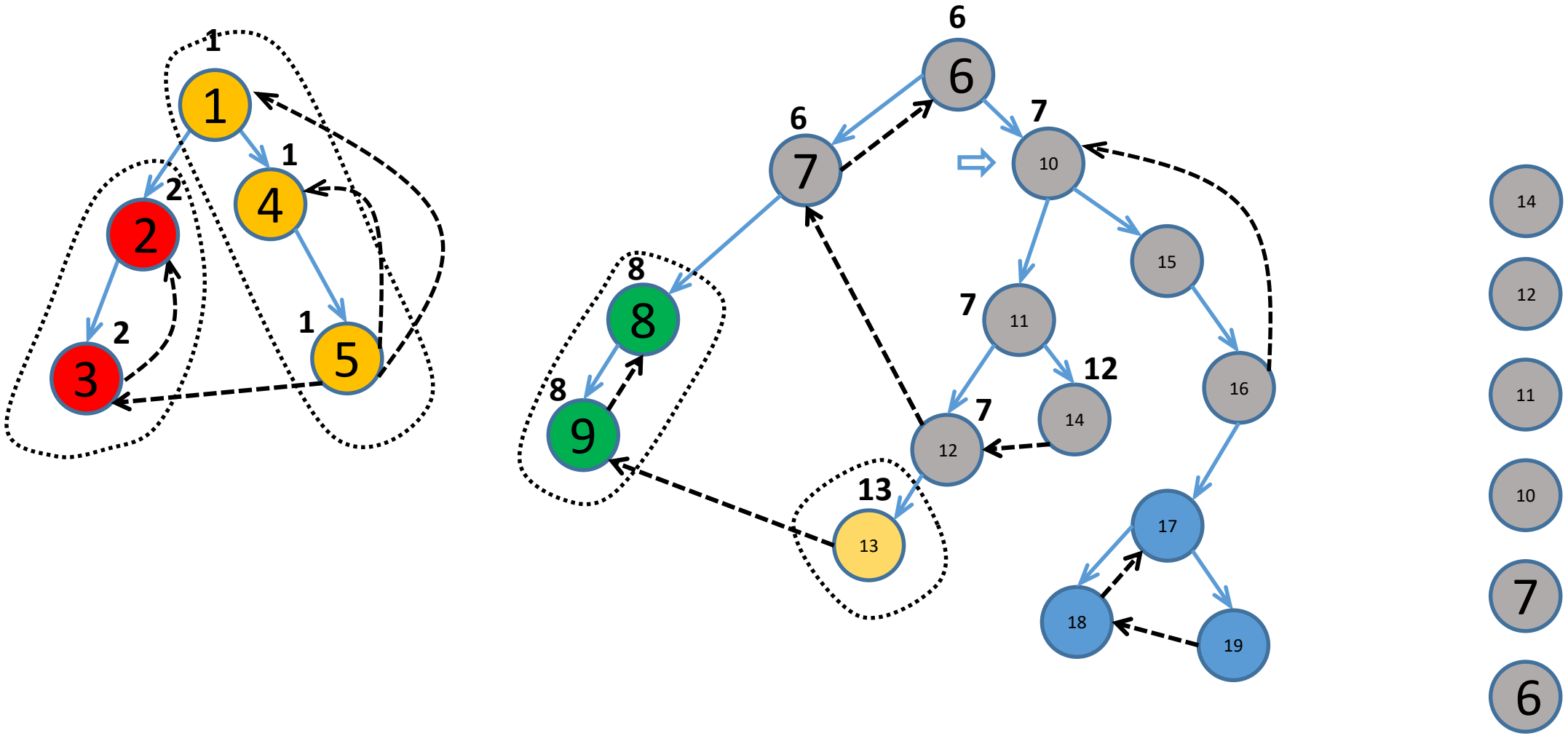

1
1
2
1
4
2
3
1
5
6
6
7
10
10
8
8
9
11
11
15
16
7
12
14
13
13
17
18
19
13
12
11
10
7
6

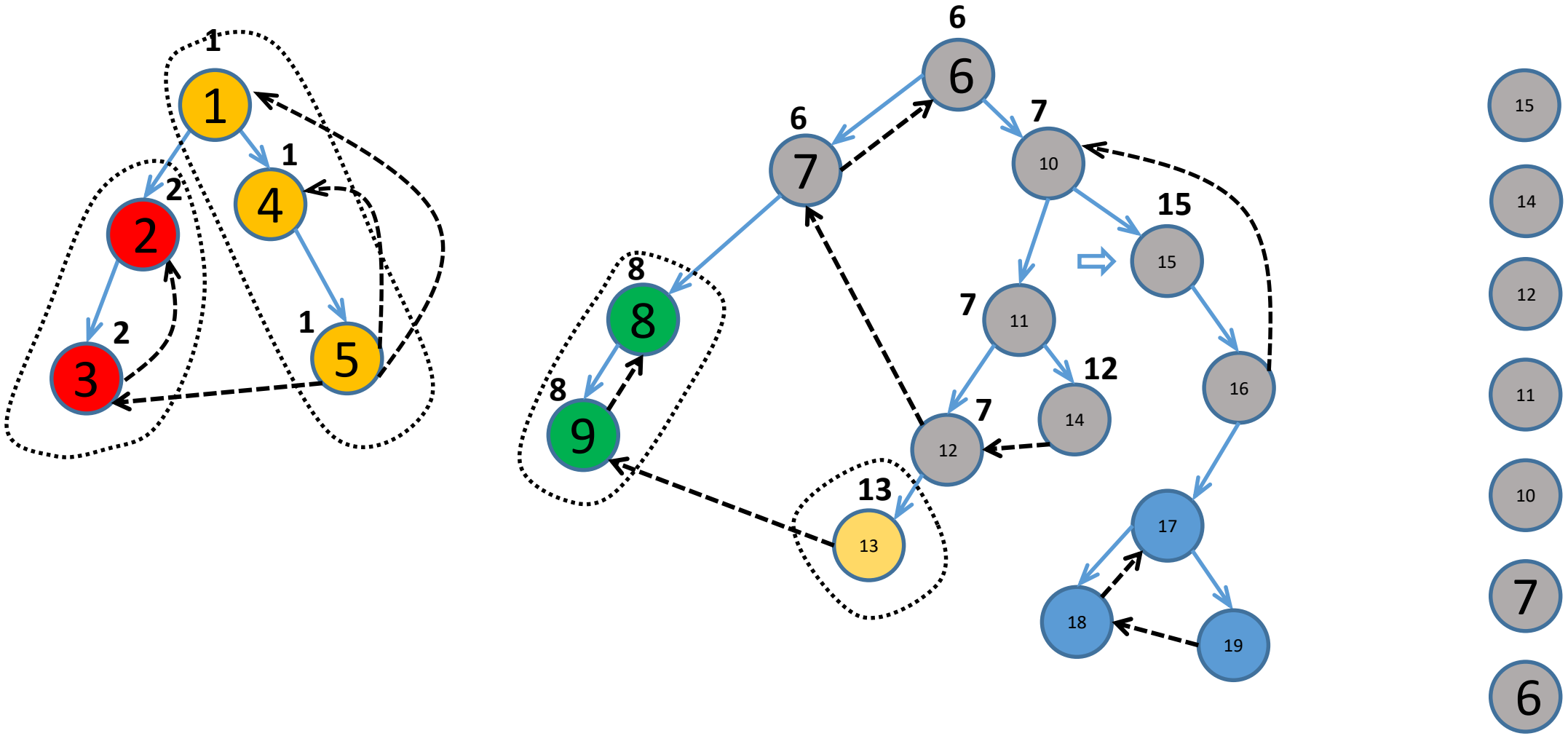

1
1
2
2
1
1
2
3
4
5
6
6
7
10
10
8
8
9
11
11
7
12
12
13
13
14
15
16
17
18
19
12
11
10
7
6

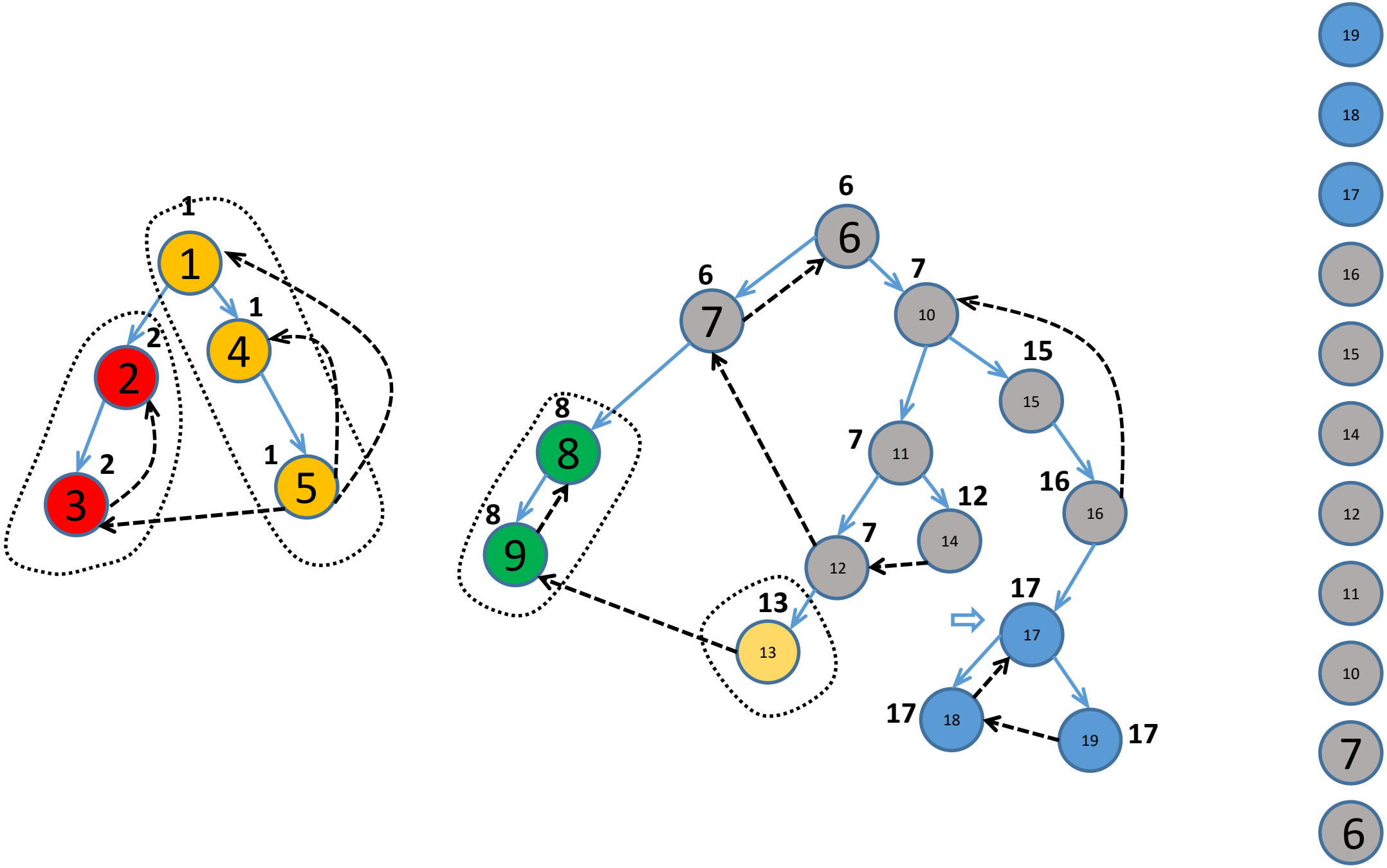

1
1
2
1
4
2
3
1
5
6
6
7
10
10
8
8
9
7
11
15
16
7
12
14
13
13
17
18
19
12
11
10
7
6

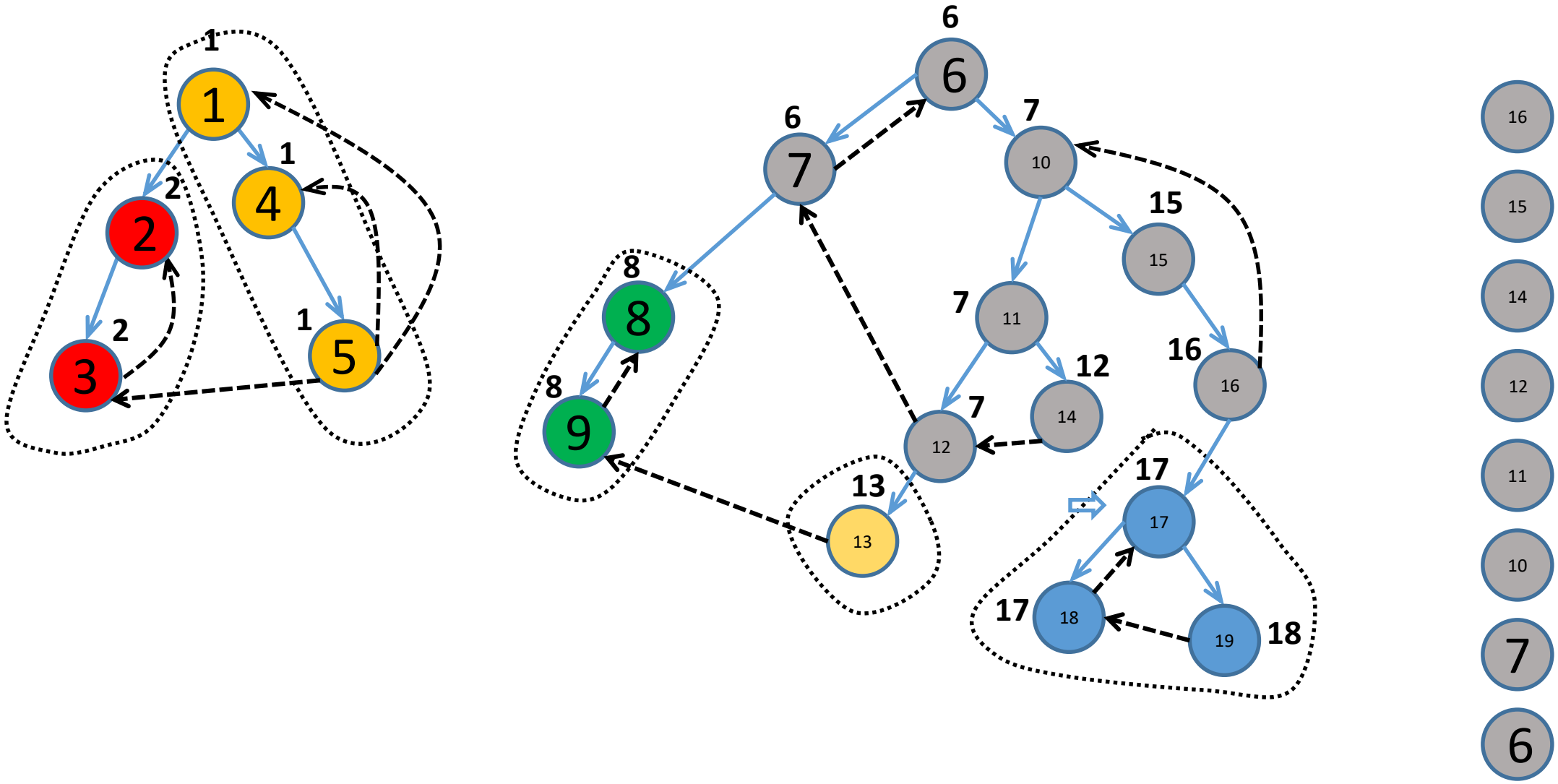

1
1
2
1
4
2
3
1
5
6
6
7
10
10
8
8
9
15
7
11
12
14
16
7
12
13
13
17
18
19
14
12
11
10
7
6


1
1
2
1
4
2
3
1
5
6
6
7
10
10
8
8
9
15
7
11
12
14
16
7
12
13
13
17
18
19
14
12
11
10
7
6

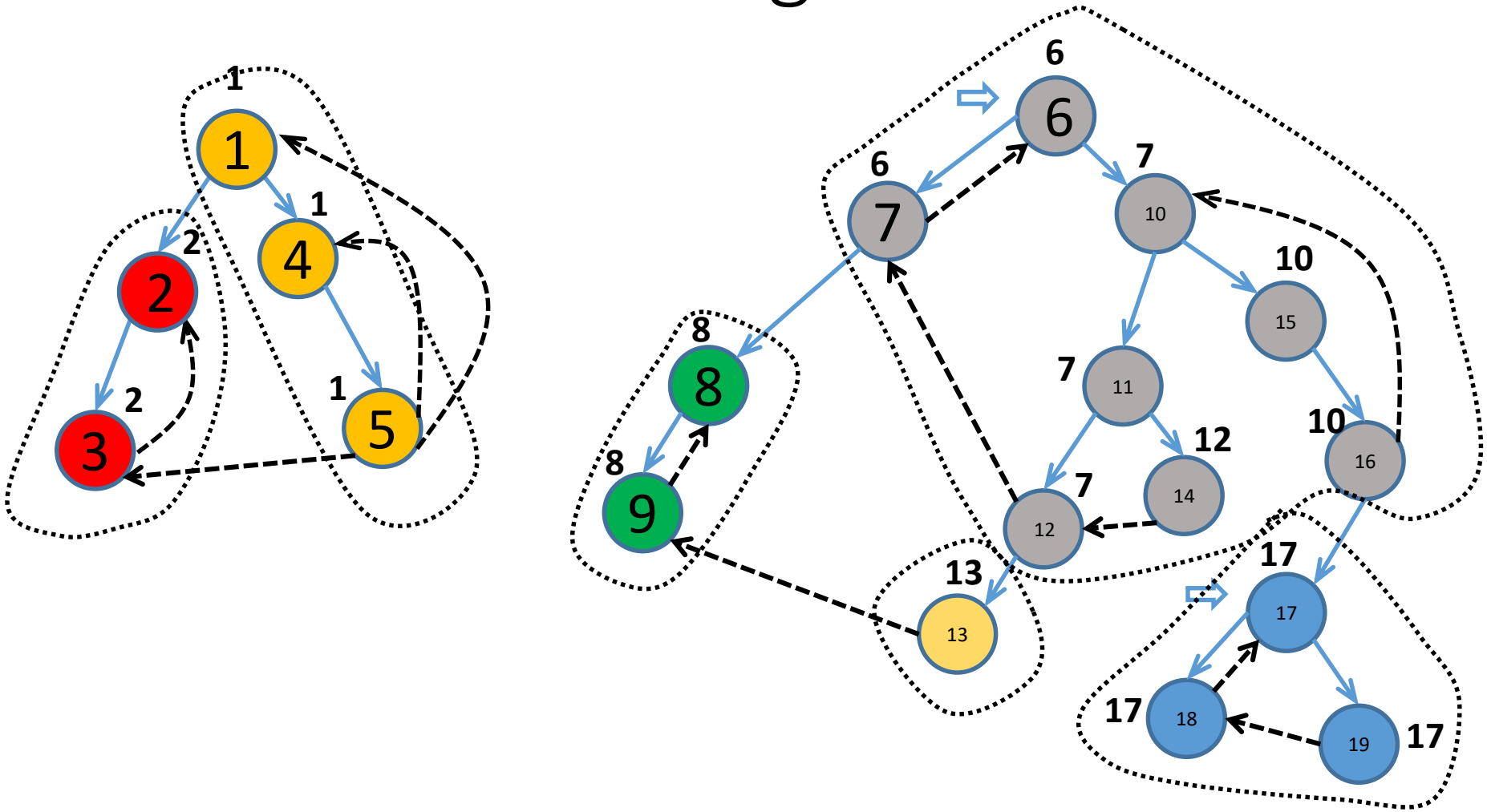

1
1
2
1
4
2
3
1
5
6
6
6
7
7
10
8
8
8
9
15
7
11
12
16
7
14
12
13
13
17
18
19
14
12
11
10
7
6


1
1
2
2
1
1
2
3
4
5
6
6
7
7
10
15
15
8
8
9
7
11
12
14
7
12
13
13
16
17
18
19
15
14
12
11
10
7
6


1
1
2
1
4
2
3
1
5
6
6
7
6
7
10
15
15
8
8
8
9
7
11
12
14
16
16
7
12
13
13
17
17
17
18
19
17
19
18
17
16
15
14
12
11
10
7
6


1
1
2
1
4
2
3
1
5
6
6
6
7
7
10
15
15
8
8
8
9
7
11
12
14
16
16
7
12
13
13
17
17
17
18
19
18
16
15
14
12
11
10
7
6

SCs using low values



Tarjan's SCC algorithm

```
SCC(G)
For each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
For each vertex u ∈ G.V
    if u.color == WHITE
        SCC-Visit(G,u)
```

```
SCC-Visit(G,u)
time = time + 1
u.d = low(u) = time
push(u,S)
u.color = GRAY
For each vertex v ∈ G.Adj[u]
    if v.color == WHITE
        v.π = u
        SCC-Visit(G,v)
        low(u) = min{low(u), low(v)}
    else if v ∈ S
        low(u) = min{v.d, low(u)}
u.color = BLACK
time = time + 1
u.f = time
if low(u) = u.d then peform pop(S) until u
  is poped, the set poped is a SC
```