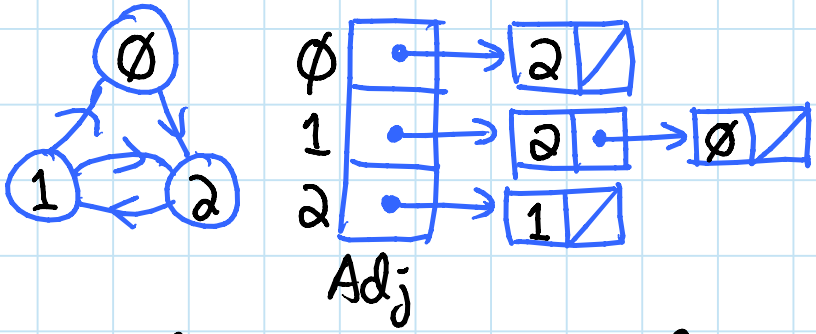
6.006	Lecture 11	Oct. 18, 2018

TODAY: DFS
- depth-first search algorithm
- edge classification
- cycle testing
- topological sort

Recall:
- adjacency-set representation of graphs: for each vertex $u \in V$, store u's neighbor set $Adj(u) = \{v \in V \mid (u,v) \in E\}$
  - just outgoing edges if directed
  - e.g. linked list DS (adjacency lists)
  - e.g. Adj is an array, $V = \{0, 1, \ldots, |V|-1\}$

e.g.

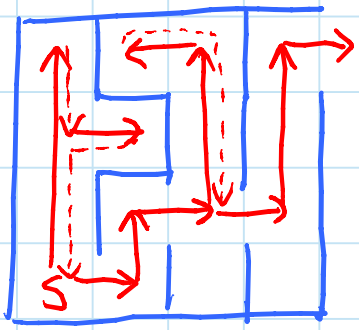


- BFS: explore graph level by level
  → breadth before depth

TODAY: DFS: natural recursive graph exploration
→ depth before breadth

Recall: in-order traversal of BST [L6]
- recurse left (if it exists)
- visit node
- recurse right (if it exists)

Depth-first search (DFS):
- for each outgoing edge (u,v):
  - recurse on v
- never revisit a vertex
  ⇒ avoid infinite recursion
- like exploring a maze: follow any path until you get stuck, backtrack along breadcrumbs until reach unexplored neighbor



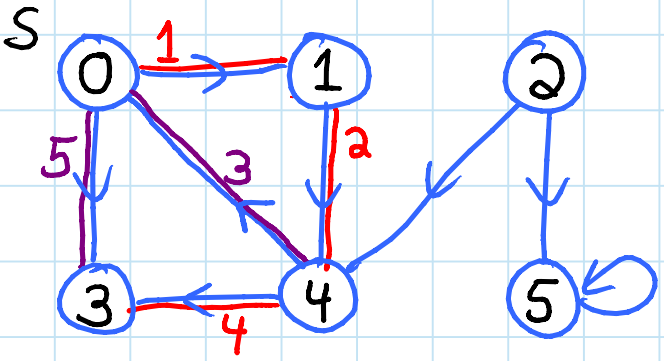

```
DFS(Adj, s):
    parent = {s: s}
    visit(u):                          ← called ≤ once/vertex
        start u
        for v in Adj[u]:               ← |Adj[u]|
            if v not in parent:
                parent[v] = u          ┐
                visit(v)               ┘ O(1) time
        finish u
    visit(s)
```

Time: $\sum_{u \in V} |Adj(u)| = O(E)$
+ $O(V)$ to init. arrays

Analysis: $O(V+E)$ time

- each $u \in V$ visited $\leq$ once (then parent[u] set)
- visit costs $\Theta(|Adj(u)|)$
- total time $= \sum_{u \in V} |Adj(u)| = O(E)$ (Handshaking, just like BFS)
  $+ O(V)$ if using arrays

Example:



tree edge (formed by parent)
nontree edge

BFS vs. DFS:

- both visit the vertices reachable from s
  - ↳ for v in Adj[u]:
    if v not in parent:
    parent[v] = u
    (for sake of this comparison ...)
- only difference is the order they visit the vertices
- can implement using a sequence DS (e.g. linked list)
  - BFS visits in queue order
    (finish level before going onto next)
  - DFS visits in stack order
    (like recursion stack)

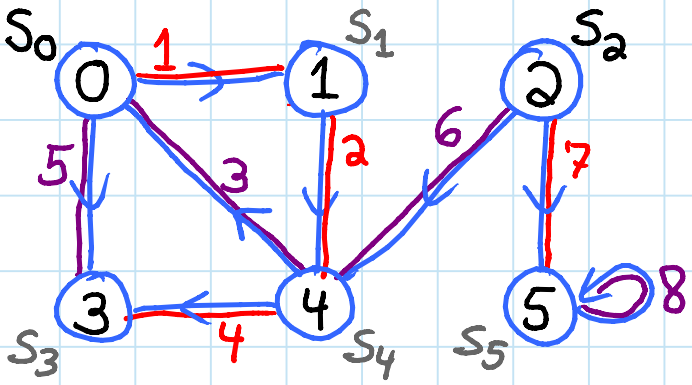
Exploring the whole graph:

(instead of just vertices reachable from s)

- try all vertices as source s
- skip if visited by a previous search

```
Full-DFS(Adj):          — usually called just "DFS"
    parent = {}
    visit(u):
        ... as above ...
    for s in range(len(Adj)):
        if s not in parent:
            parent[s] = s
            visit(s)
```

Example:



tree edge (formed by parent)

nontree edge

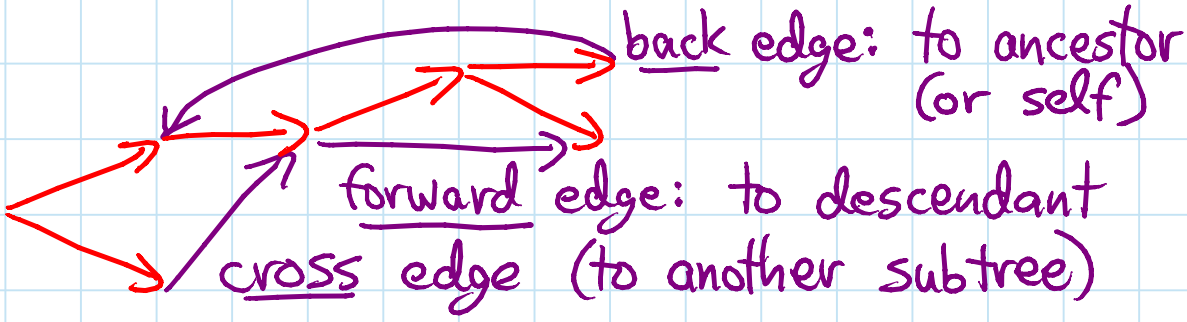
Analysis: $O(V+E)$

- outer loop adds only $O(V)$
- still visit each vertex $\leq$ once

Could do this with BFS too, but not useful (e.g. for shortest paths).

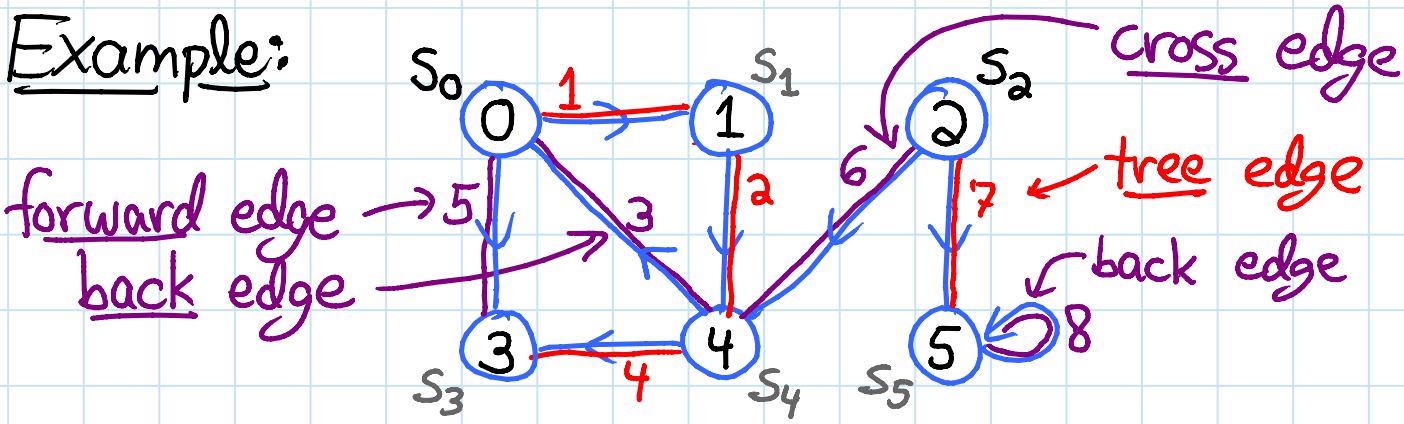
Edge classification:

tree edges (formed by parent)
nontree edges



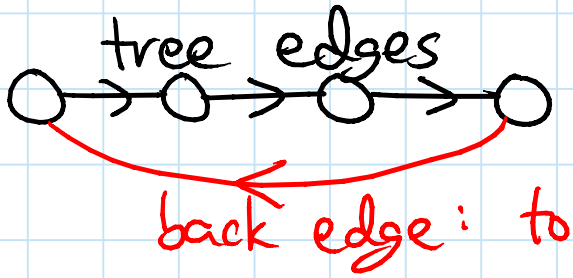
- to compute this classification, mark nodes for duration they are "on the stack" (start/end in code above)
- undirected graph has only tree & back edges

Example:



Cycle detection: graph G has a cycle $\Leftrightarrow$ full DFS has a back edge

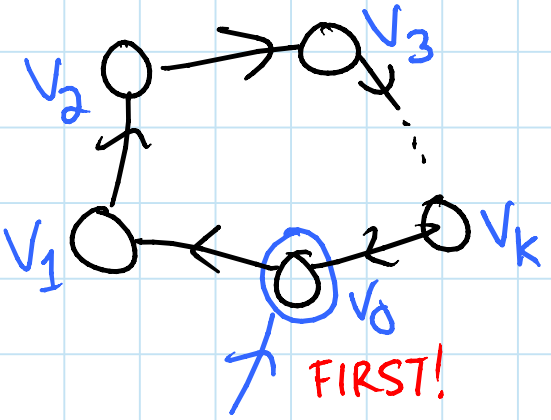
Proof: ($\Leftarrow$)



is a cycle

back edge: to tree ancestor

($\Rightarrow$) consider first visit to cycle:



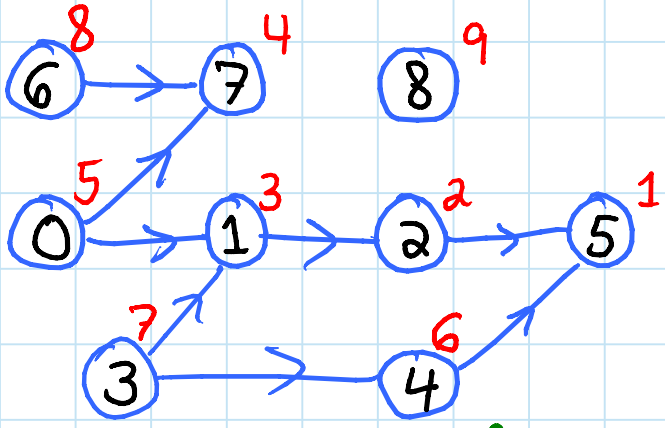
- before visit to v_i finishes, will visit v_{i+1} (& finish): will consider edge (v_i, v_{i+1}) $\Rightarrow$ visit v_{i+1} now or already did

$\Rightarrow$ before visit to v_0 finishes, will visit v_k (& didn't before by first-visit assumption)

$\Rightarrow$ before visit to v_k (or v_0) finishes, will see (v_k, v_0) as back edge.

□

Topological sorting problem: given a DAG (directed acyclic graph), order the vertices such that all edges point lower → higher



DFS finishing times

Application: job scheduling (vertices = jobs, edges = dependencies)

Source = vertex with no incoming edges
= can put at beginning (0, 3, 6, 8)

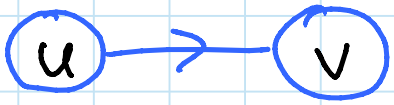
Attempt: BFS from each source:
- from 0 finds 0, 17, 2, 5
- from 3 finds 3, 14, 25 ← slow... and wrong!
- from 6 finds 6, 7
- from 8 finds 8

Topological sort: reverse of full DFS finishing times (time at which DFS visit() finishes)

```
visit(u):
    ...
    order.append(u)
order.reverse()
```

Correctness: for any edge (u, v), u ordered before v

i.e. v finished before u



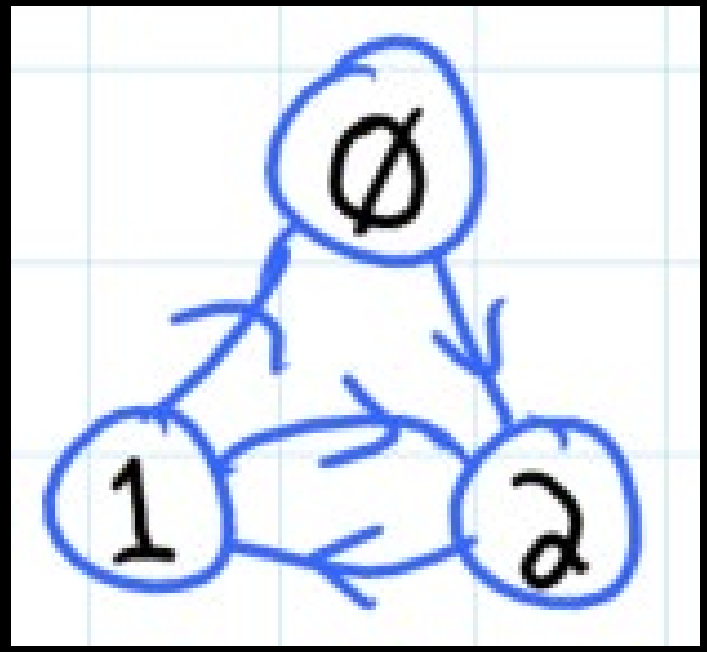
2 cases:

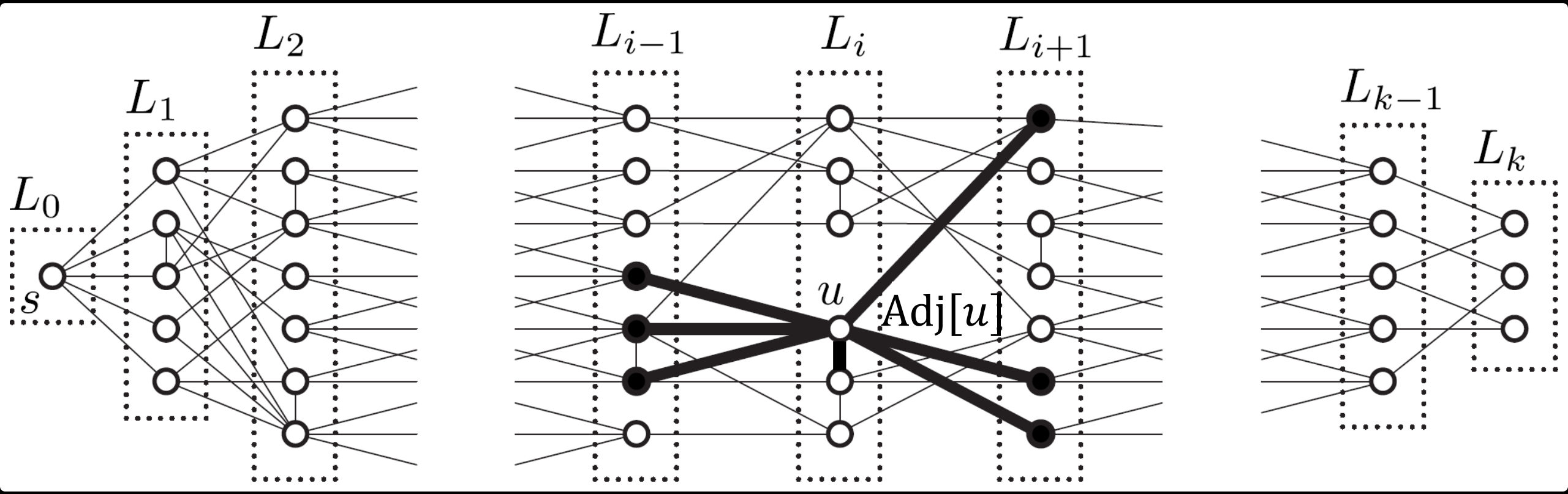
- if u visited before v:
  - before visit to u finishes, will visit v (via (u,v) or otherwise)
  - $\Rightarrow$ v finishes before u ✓

- if v visited before u:
  - graph is acyclic
  - $\Rightarrow$ u can't be reached from v
  - $\Rightarrow$ visit to v finishes before visiting u ✓

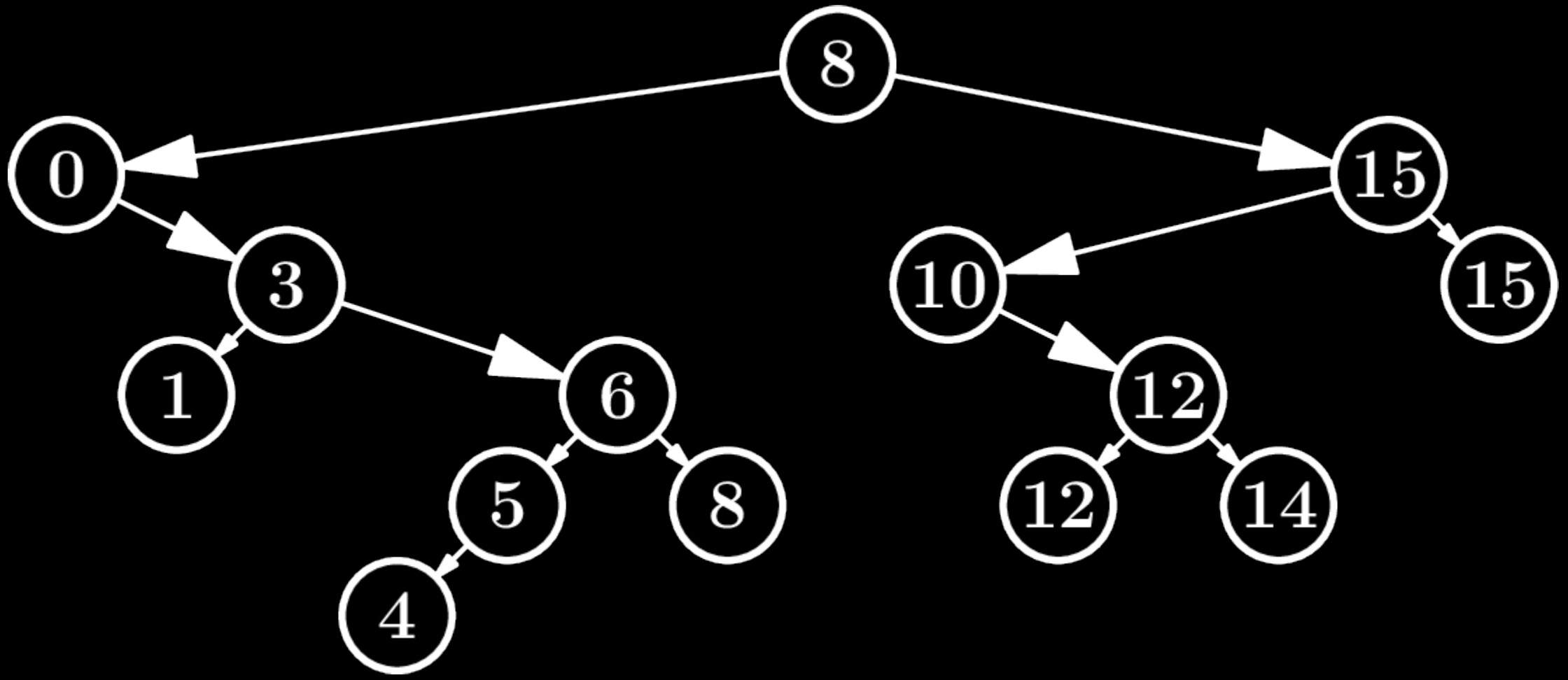
□

```
Adj = [
      [2],      # 0
      [2, 0],   # 1
      [1],      # 2
]
```




L_0
L_1
L_2
L_{i-1}
L_i
L_{i+1}
L_{k-1}
L_k
s
u
Adj[u]

In-Order Traversal
(order-iter for BSTs)



```
# In-Order Traversal
# (order-iter for BSTs)
def traversal(node):
    if node.left:
        yield from traversal(node.left)
    yield node
    if node.right:
        yield from traversal(node.right)
```

```
# In-Order Traversal
# (order-iter for BSTs)
def traversal_print(node):
    if node.left:
        traversal_print(node.left)
    print(node.item)
    if node.right:
        traversal_print(node.right)
```

```
def DFS(Adj, s): # DFS from s in graph given by Adj
    def visit(u):
        for v in Adj[u]: # edge (u,v)
            visit(v)
    visit(s)
```

INFINITE RECURSION ☹

```
def DFS(Adj, s): # DFS from s in graph given by Adj
    parent = [None] * len(Adj) # DFS tree
    def visit(u):
        for v in Adj[u]: # edge (u,v)
            if parent[v] is None:
                parent[v] = u
                visit(v)
    parent[s] = s          # s is root
    visit(s)
```

```
def BFS(Adj, s): # BFS from s in graph given by Adj
    parent = [None] * len(Adj) # shortest-path tree
    parent[s] = s              # s is root
    d = [math.inf] * len(Adj)  # d[v] = δ(s,v)
    d[s] = 0                   # δ(s,s) = 0
    level = [[s]]     # level 0 = {s}
    while level[-1]: # stop at empty level
        level.append([]) # start next level
        for u in level[-2]:
            for v in Adj[u]: # edge (u,v)
                if parent[v] is None: # v new
                    parent[v] = u
                    d[v] = d[u] + 1 #= len(level)-1
                    level[-1].append(v)
```

```
def BFS(Adj, s): # BFS from s in graph given by Adj
    parent = [None] * len(Adj) # BFS tree
    d = [math.inf] * len(Adj)  # d[v] = δ(s,v)
    d[s] = -1                  # will become 0
    queue = Queue([(s, s)]) # level 0 = {s}
    while len(queue):  # stop at empty queue
        u, v = queue.delete_left()
        if parent[v] is None: # v new
            parent[v] = u
            d[v] = d[u] + 1
            for x in Adj[v]: # edge (v,x)
                queue.insert_right((v, x))
```

```
def DFS(Adj, s): # DFS from s in graph given by Adj
    parent = [None] * len(Adj) # DFS tree
    d = [math.inf] * len(Adj)  # d[v] = tree dist.
    d[s] = -1                  # will become 0
    stack = Stack([(s, s)]) # level 0 = {s}
    while len(stack):  # stop at empty stack
        u, v = stack.delete_right()
        if parent[v] is None: # v new
            parent[v] = u
            d[v] = d[u] + 1
            for x in reversed(Adj[v]): # (v,x)
                stack.insert_right((v, x))
```

```
def DFS(Adj, s): # DFS from s in graph given by Adj
    parent = [None] * len(Adj) # DFS tree
    def visit(u):
        for v in Adj[u]: # edge (u,v)
            if parent[v] is None:
                parent[v] = u
                visit(v)
    parent[s] = s          # s is a root
    visit(s)
```

```
def full_DFS(Adj): # DFS graph given by Adj
    parent = [None] * len(Adj) # DFS tree
    def visit(u):
        for v in Adj[u]: # edge (u,v)
            if parent[v] is None:
                parent[v] = u
                visit(v)
    for s in range(len(Adj)): # try all sources
        if parent[s] is None:
            parent[s] = s      # s is a root
            visit(s)
```

```
def full_DFS(Adj): # DFS in graph given by Adj
    parent = [None] * len(Adj) # DFS tree
    start = [None] * len(Adj)
    finish = [None] * len(Adj)
    t = 0
    def visit(u):
        nonlocal t
        start[u] = t
        t += 1
        for v in Adj[u]: # edge (u,v)
            if parent[v] is None:
                parent[v] = u
                visit(v)
        finish[u] = t
        t += 1
    for s in range(len(Adj)): ... # try all sources
```

```
def topological_sort(Adj):
    parent = [None] * len(Adj) # DFS tree
    order = []
    def visit(u):
        for v in Adj[u]:   # edge (u,v)
            if parent[v] is None:
                parent[v] = u
                visit(v)
        order.append(u)
    for s in range(len(Adj)): # try all sources
        ...
    order.reverse()
    return order
```