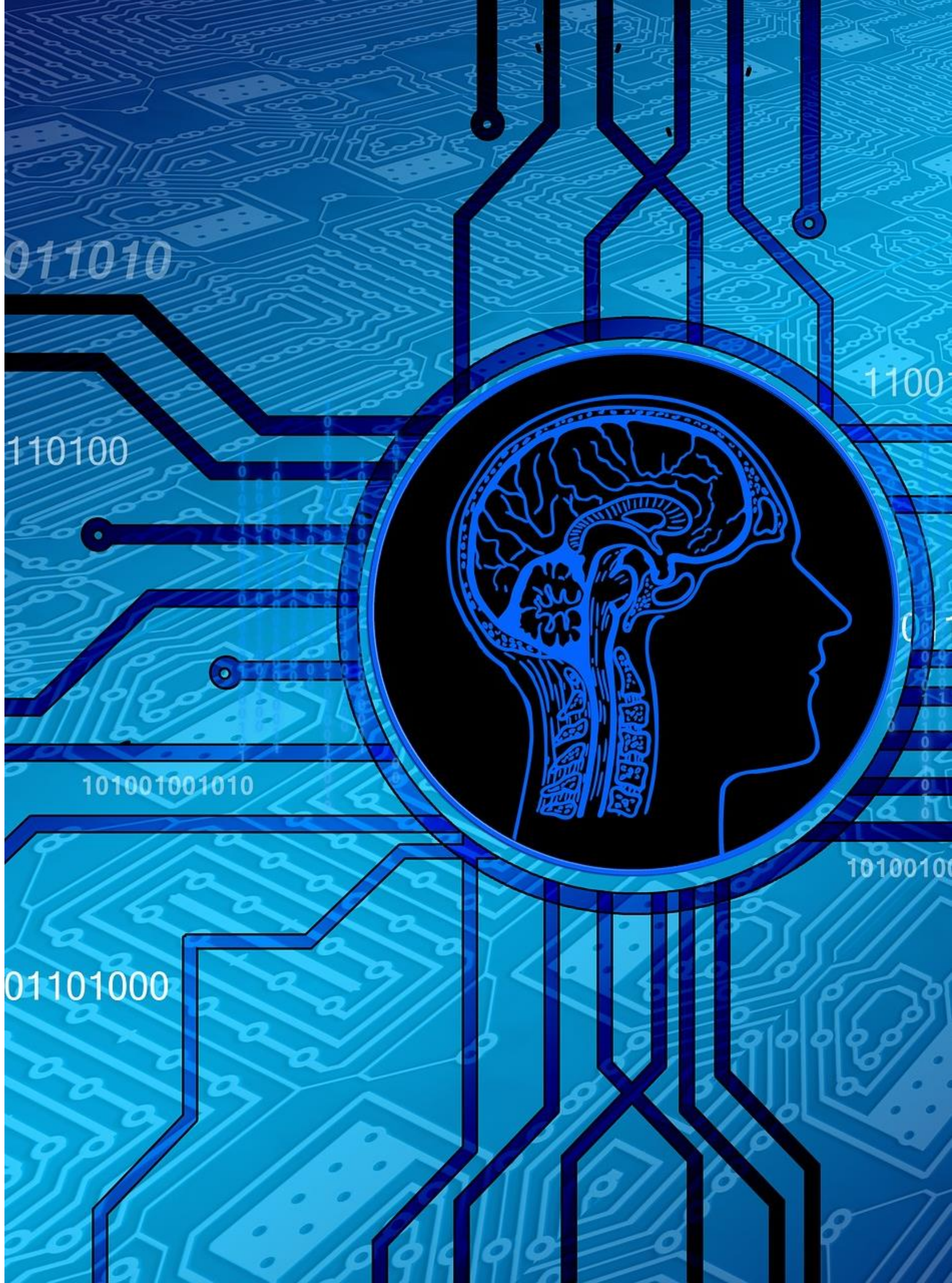
Search Strategies

Informatics 2D: Reasoning and Agents

Lecture 3

Adapted from slides provided by Dr Petros Papapanagiotou



Search strategies

A search strategy is defined by picking the order of node expansion.

- Nodes are taken from the ***frontier.***

Evaluating search strategies

completeness: does it always find a solution if one exists?

time complexity: number of nodes generated / expanded

space complexity: maximum number of nodes in memory

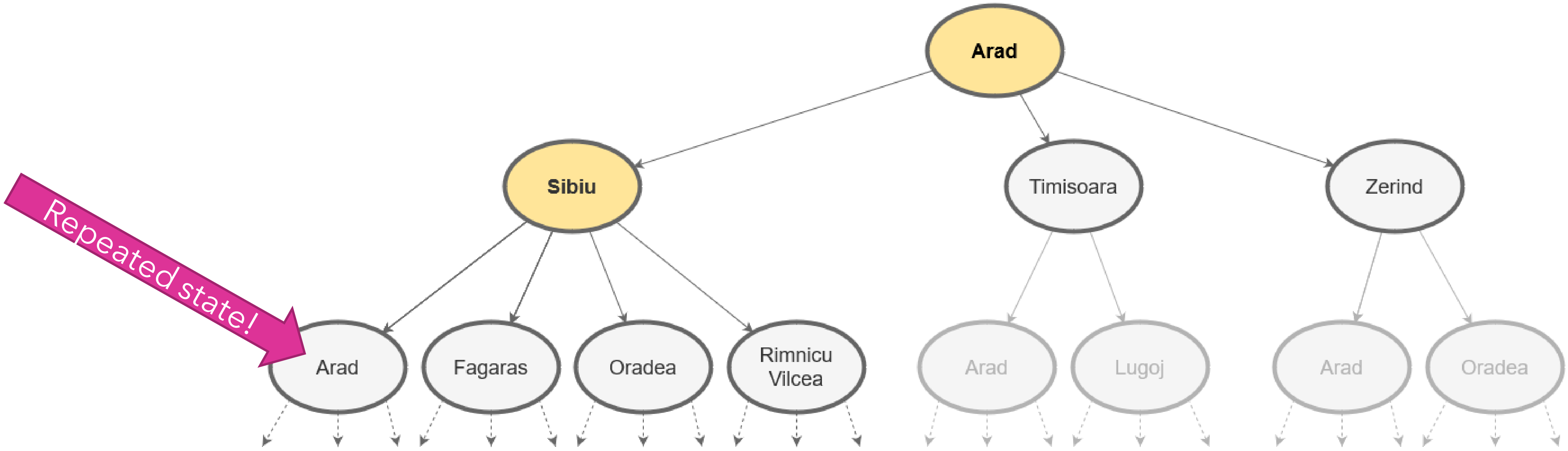
optimality: does it always find a least-cost solution?

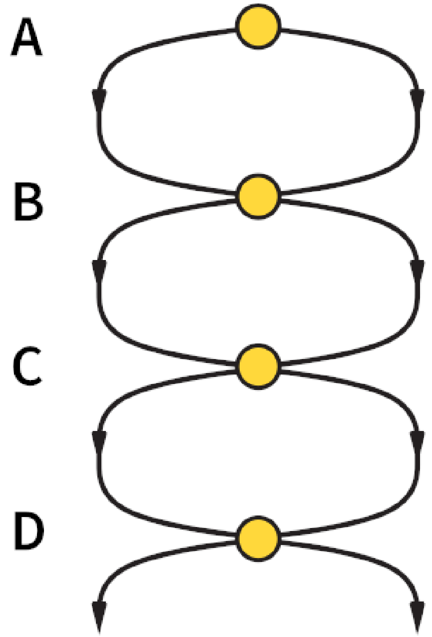
Time and space complexity are measured in terms of:

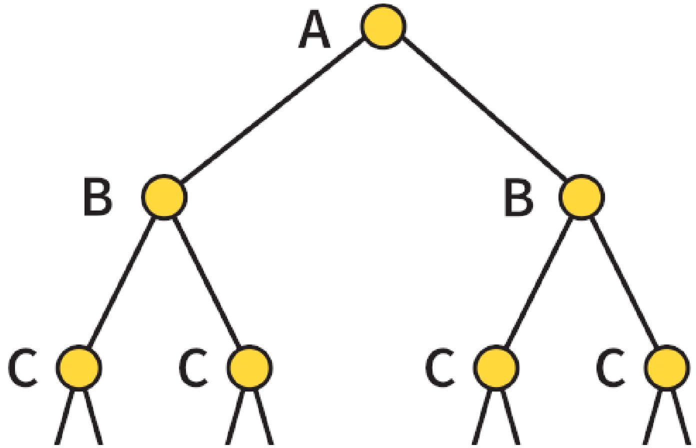
- ***b***: maximum branching factor of the search tree
- ***d***: depth of the least-cost solution
- ***m***: maximum depth of the state space (may be ∞)

Recall: Tree Search

function TREE-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier







Repeated states

Failure to detect repeated states can turn a **linear** problem into an **exponential** one!

Graph search

function GRAPH-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  initialize the explored set to be empty
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set

Augment TREE-SEARCH with a new data-structure:

- the **explored set** (closed list), which remembers every expanded node
- newly expanded nodes already in explored set are discarded

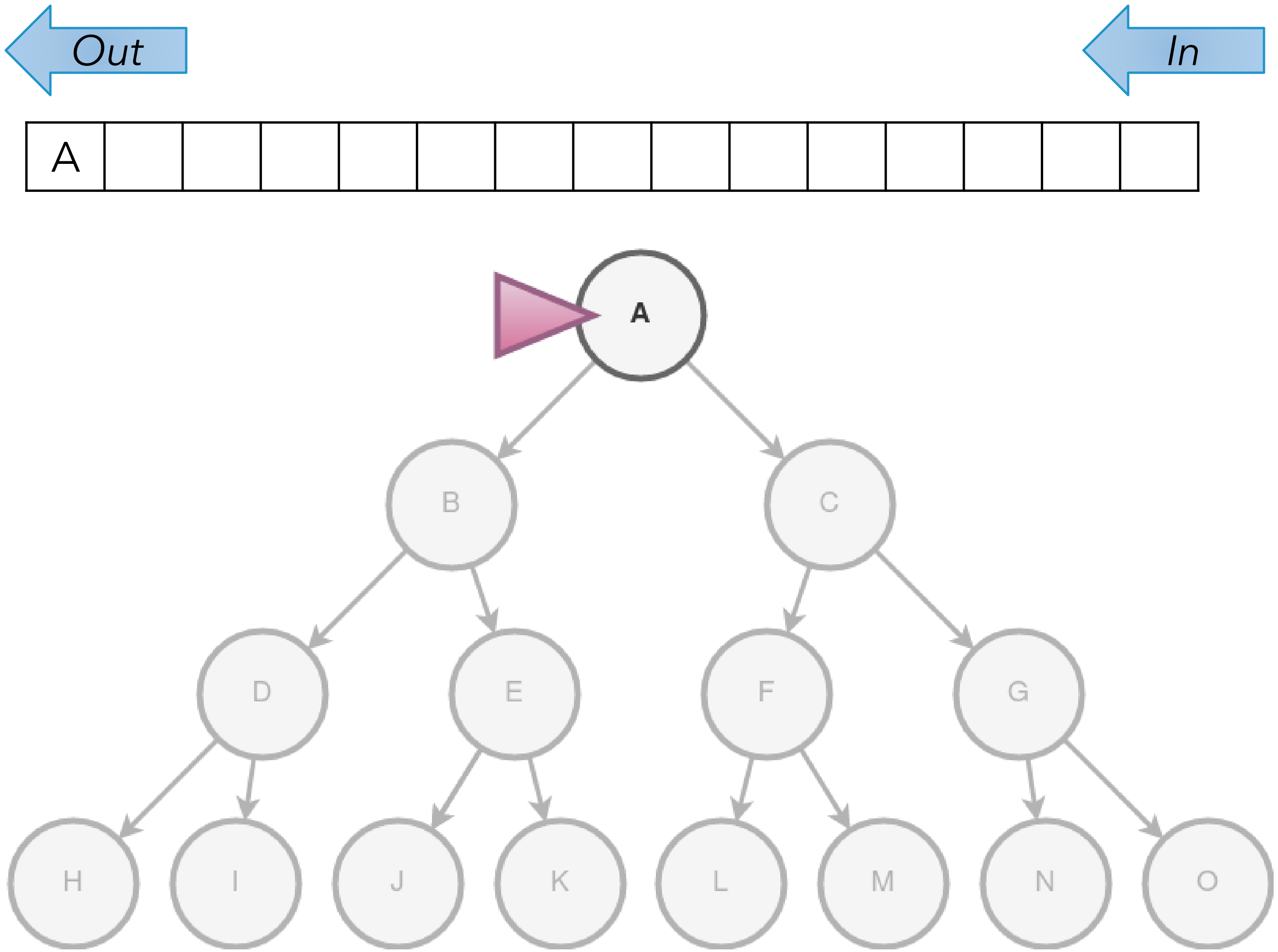
Breadth-first search

Breadth-first search

Expand **shallowest** unexpanded node

Implementation:

- *frontier* is a **FIFO** queue, i.e., new successors go at end

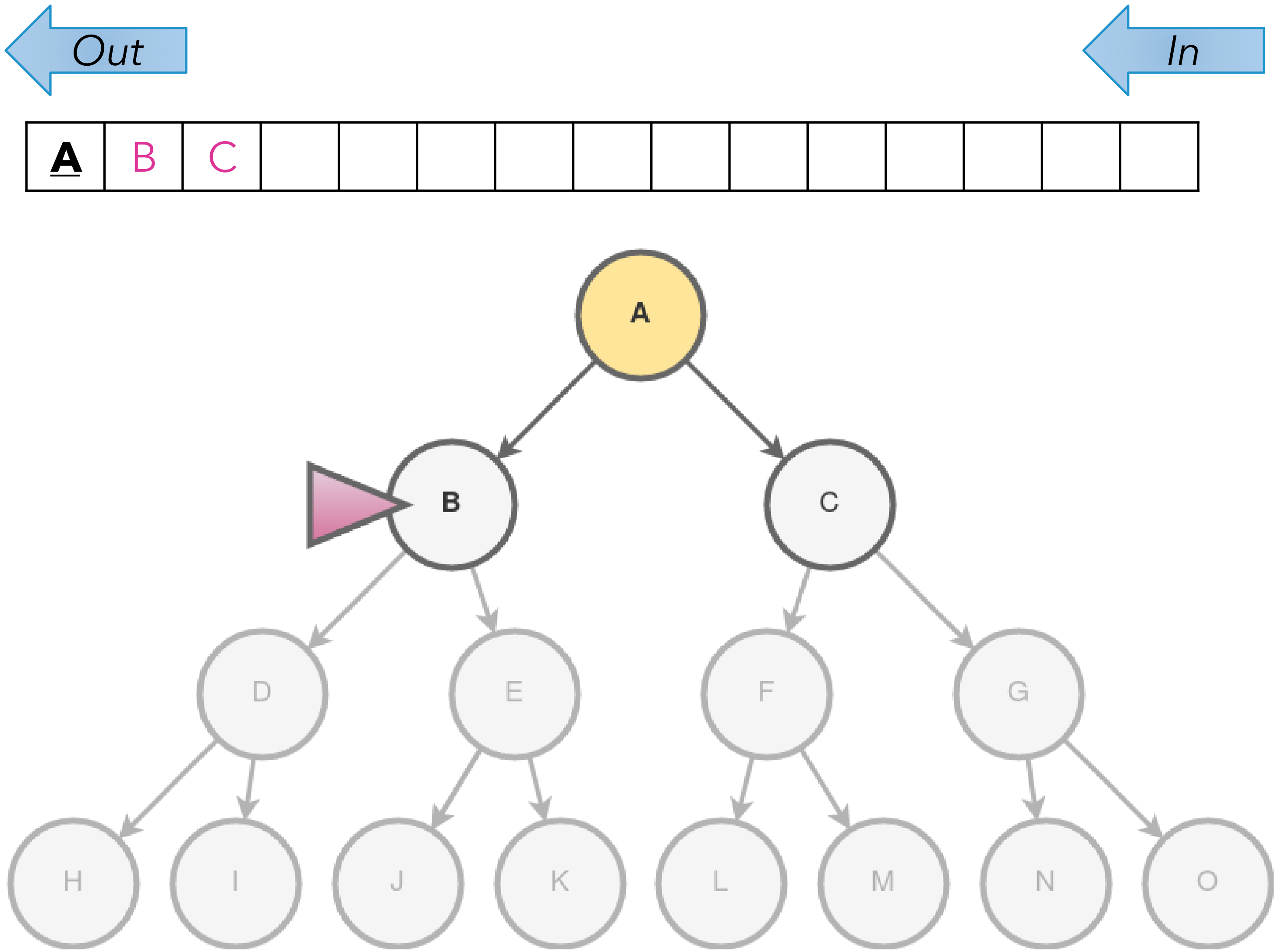


Breadth-first search

Expand **shallowest** unexpanded node

Implementation:

- *frontier* is a **FIFO** queue, i.e., new successors go at end

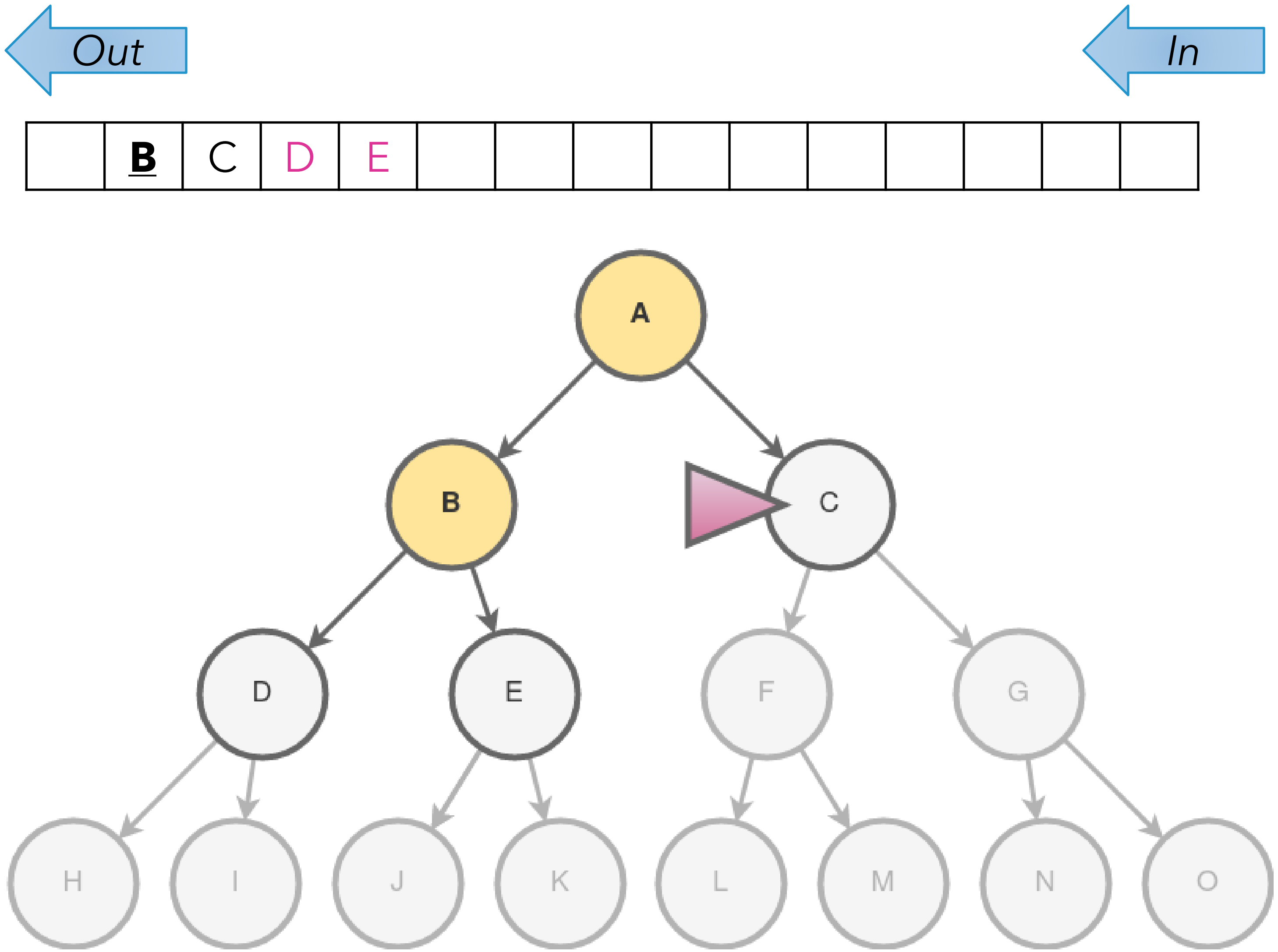


Breadth-first search

Expand **shallowest** unexpanded node

Implementation:

- *frontier* is a **FIFO** queue, i.e., new successors go at end

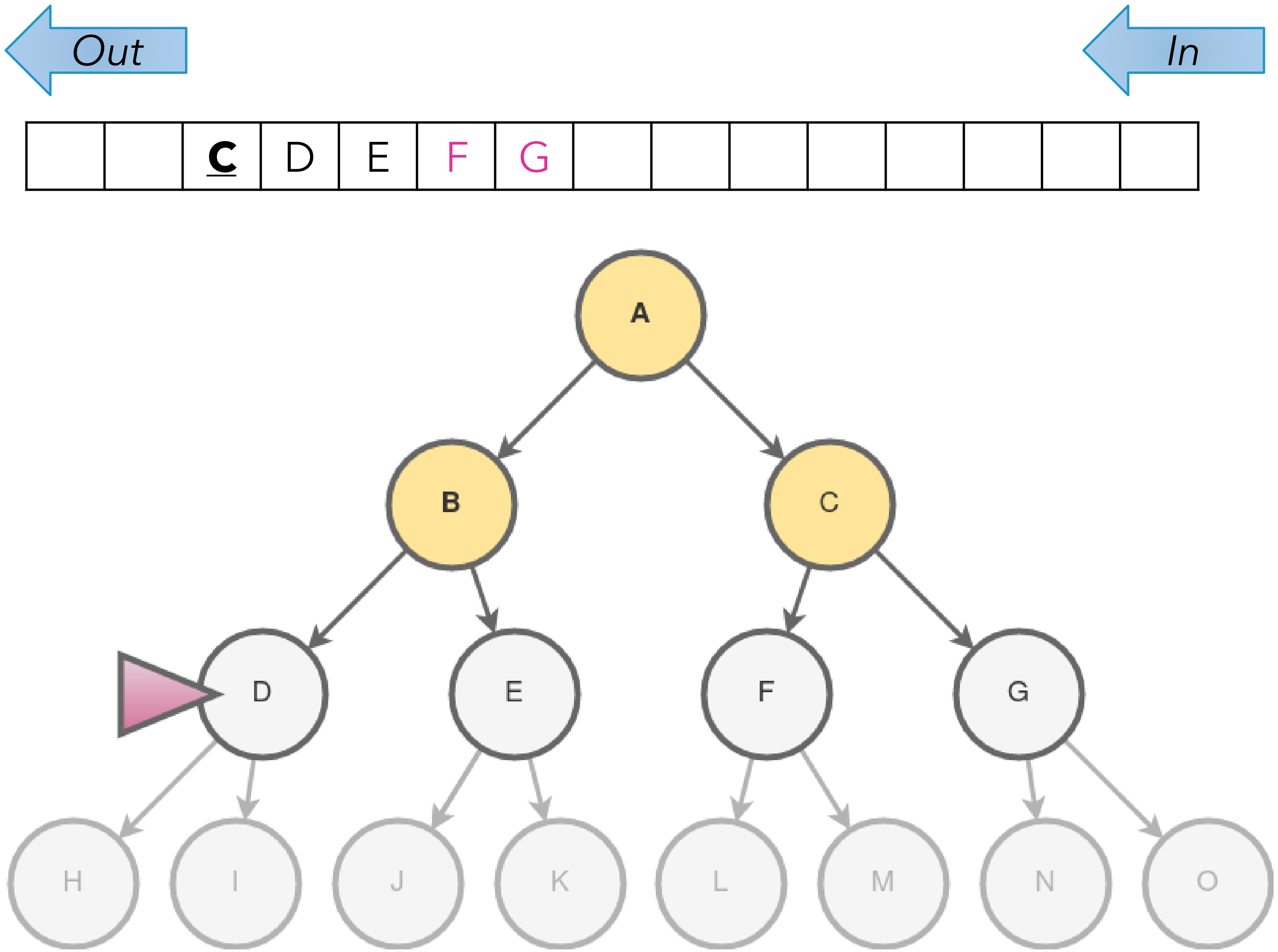


Breadth-first search

Expand **shallowest** unexpanded node

Implementation:

- *frontier* is a **FIFO** queue, i.e., new successors go at end



Breadth-first search algorithm

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)   /* chooses the shallowest node in frontier */
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
           frontier ← INSERT(child, frontier)
```

Properties of breadth-first search

Complete?

Time complexity?

Space complexity?

Optimal?

Properties of breadth-first search

Complete?
Yes (if *b* is finite)

Time complexity?

Space complexity?

Optimal?

Properties of breadth-first search

Complete?
Yes (if b is finite)

Time complexity?
$b+b^2+b^3+\ldots+b^d = O(b^d)$ (worst-case)

Space complexity?

Optimal?

Properties of breadth-first search

Complete?
Yes (if *b* is finite)

Time complexity?
$b+b^2+b^3+\ldots+b^d = O(b^d)$ (worst-case)

Space complexity?
$O(b^d)$ (keeps every node in memory)

Optimal?

Properties of breadth-first search

Complete?
Yes (if b is finite)

Time complexity?
$b+b^2+b^3+\ldots+b^d = O(b^d)$ (worst-case)

Space complexity?
$O(b^d)$ (keeps every node in memory)

Optimal?
Yes (if cost = 1 per step)

Properties of breadth-first search

Complete?
Yes (if b is finite)

Time complexity?
$b+b^2+b^3+\ldots +b^d = O(b^d)$ (worst-case)

Space complexity?
$O(b^d)$ (keeps every node in memory)

Optimal?
Yes (if cost = 1 per step)

then optimal solution is closest to start!

Properties of breadth-first search

Complete?
Yes (if b is finite)

Time complexity?
$b+b^2+b^3+\ldots+b^d = O(b^d)$ (worst-case)

Space complexity?
$O(b^d)$ (keeps every node in memory)

Optimal?
Yes (if cost = 1 per step)

Space is the bigger problem (more than time)

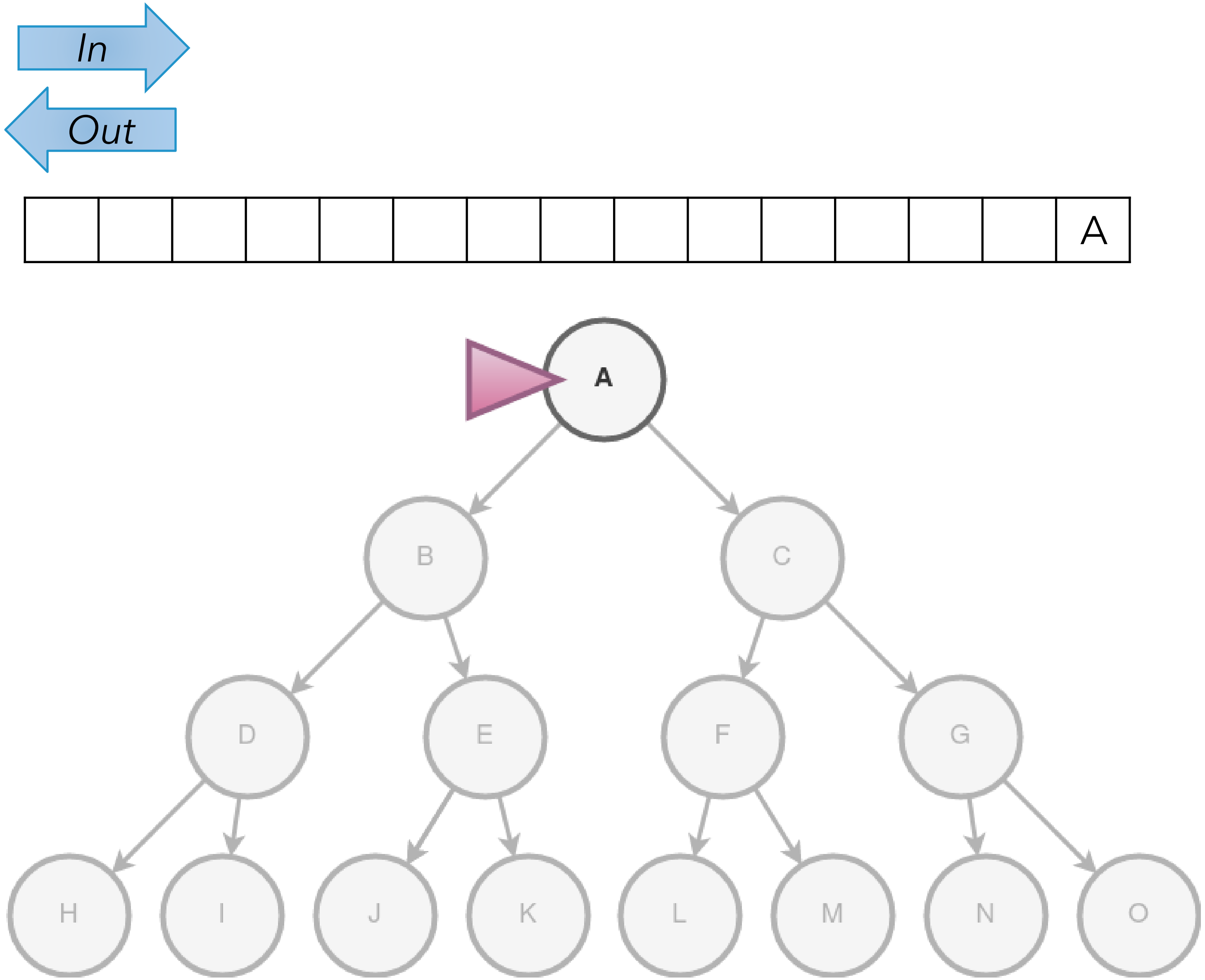
Depth-first search

Depth-first search

Expand **deepest** unexpanded node

Implementation:

- *frontier* is a **LIFO** queue, i.e., new successors go at front

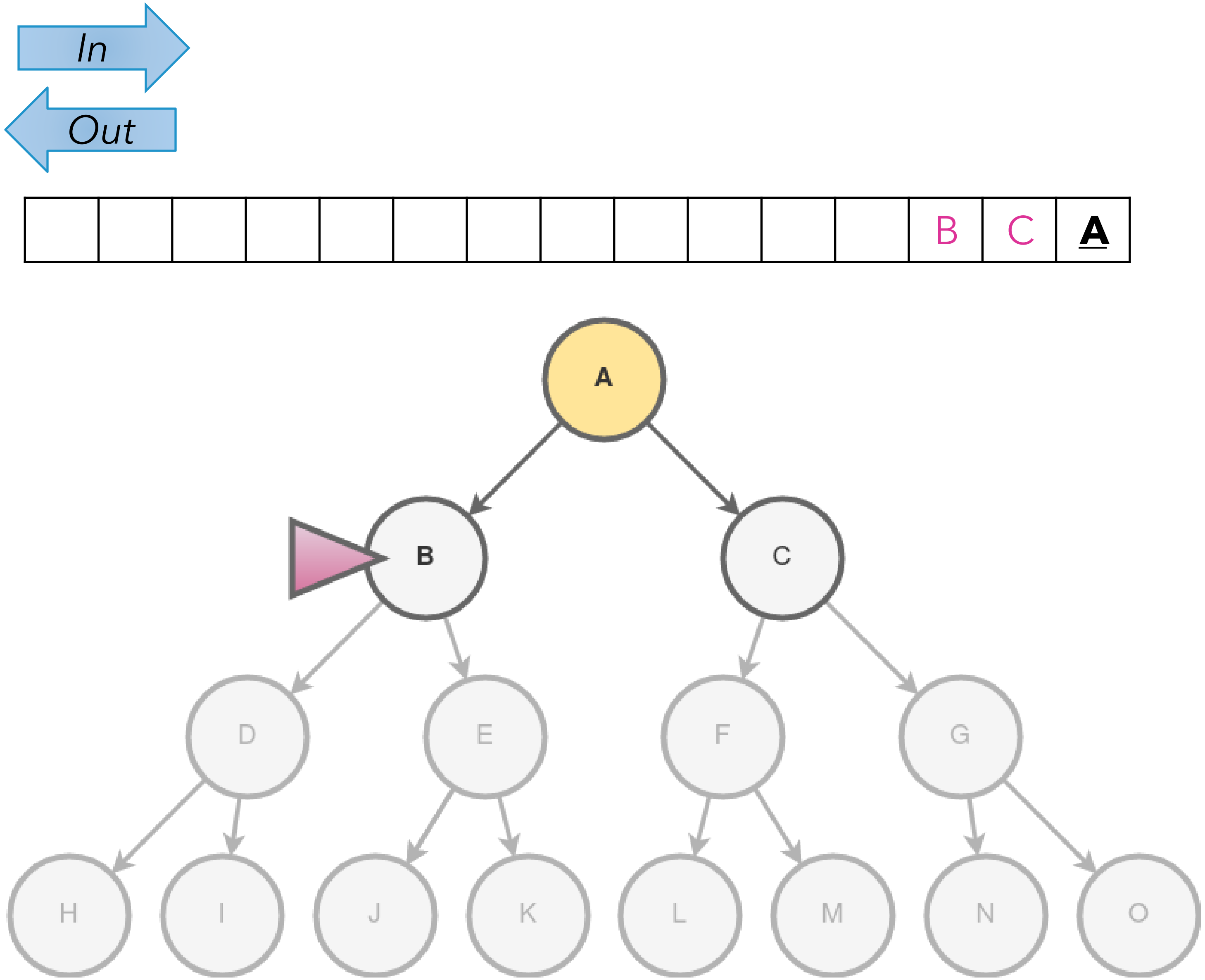


Depth-first search

Expand **deepest** unexpanded node

Implementation:

- *frontier* is a **LIFO** queue, i.e., new successors go at front

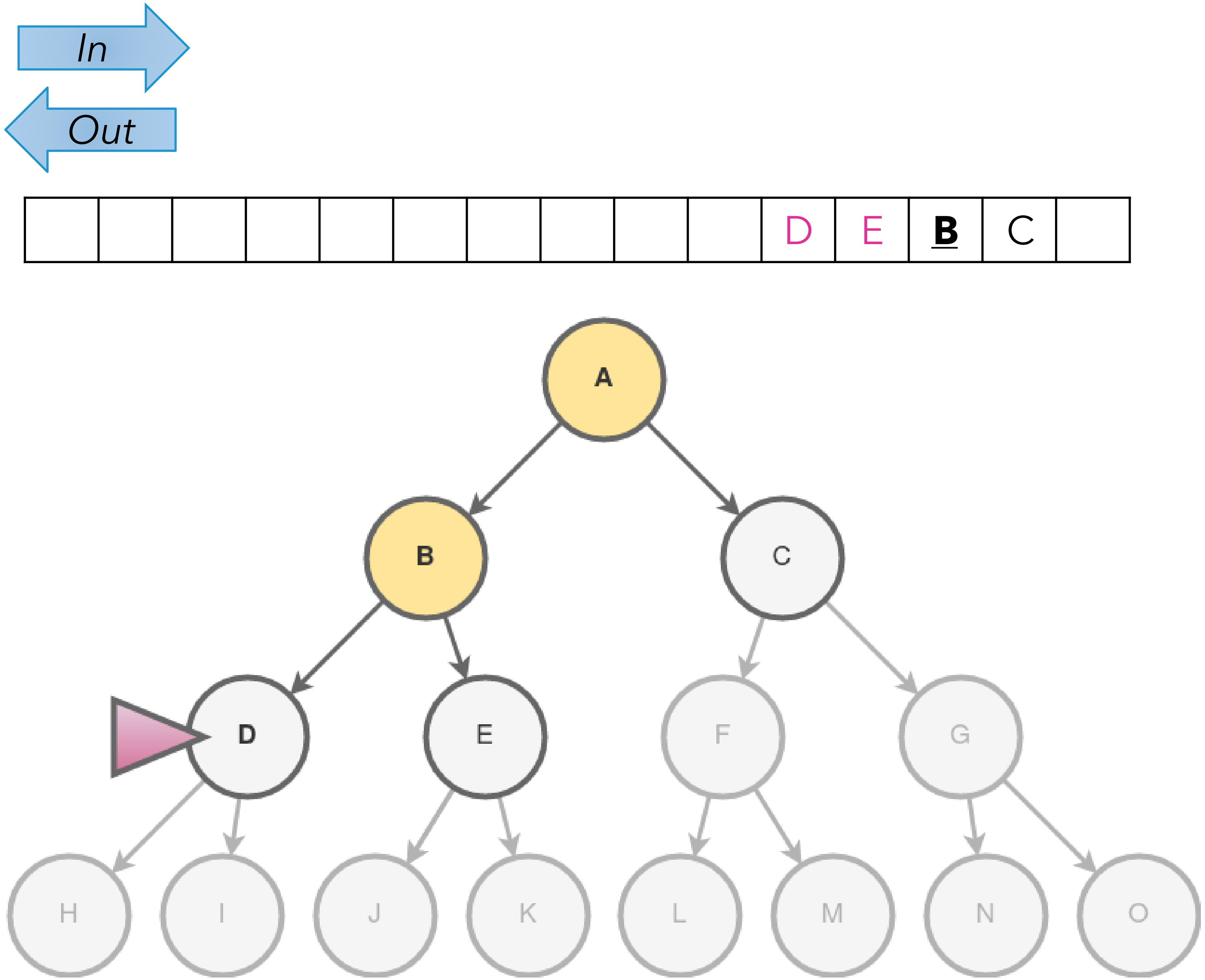


Depth-first search

Expand **deepest** unexpanded node

Implementation:

- *frontier* is a **LIFO** queue, i.e., new successors go at front

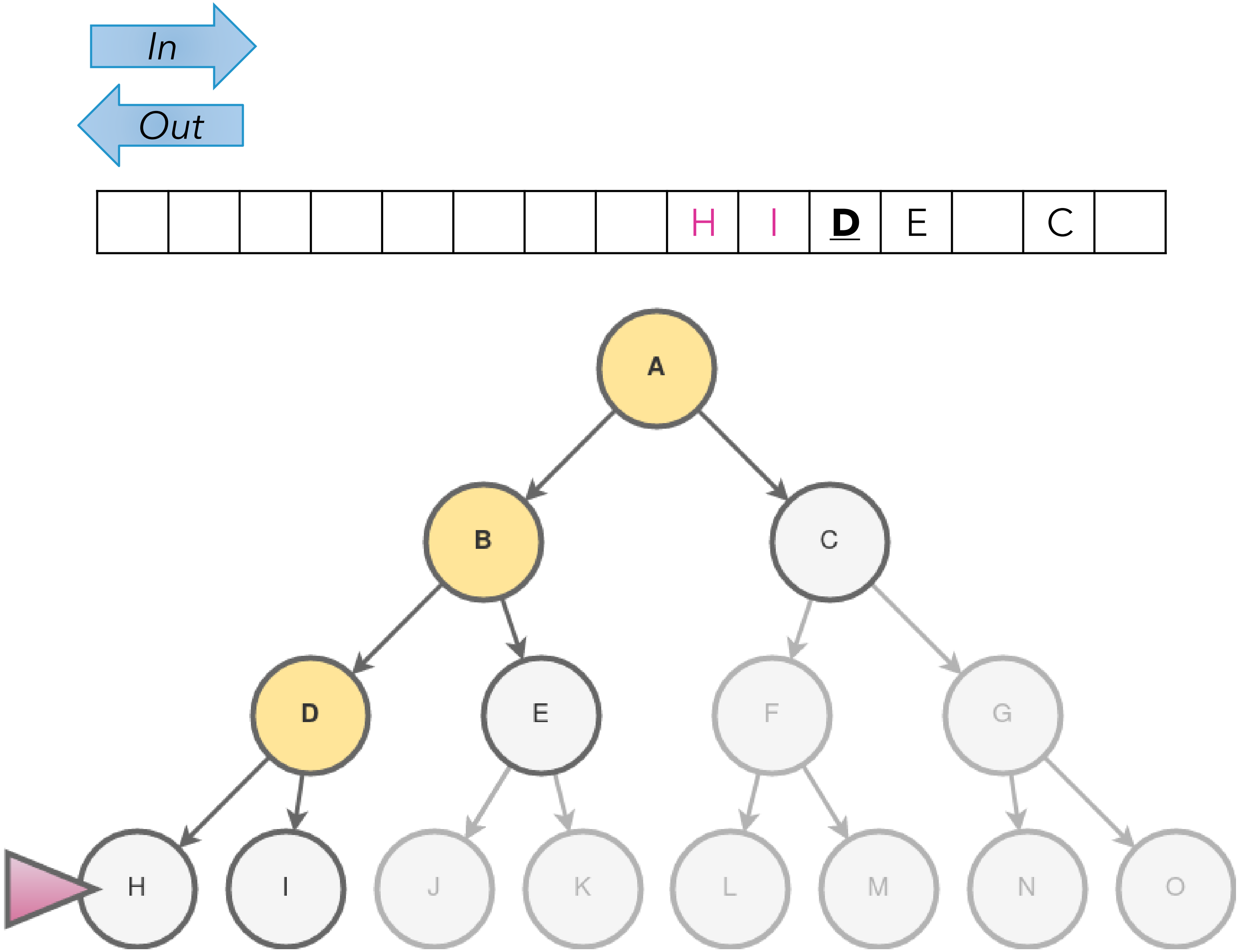


Depth-first search

Expand **deepest** unexpanded node

Implementation:

- *frontier* is a **LIFO** queue, i.e., new successors go at front

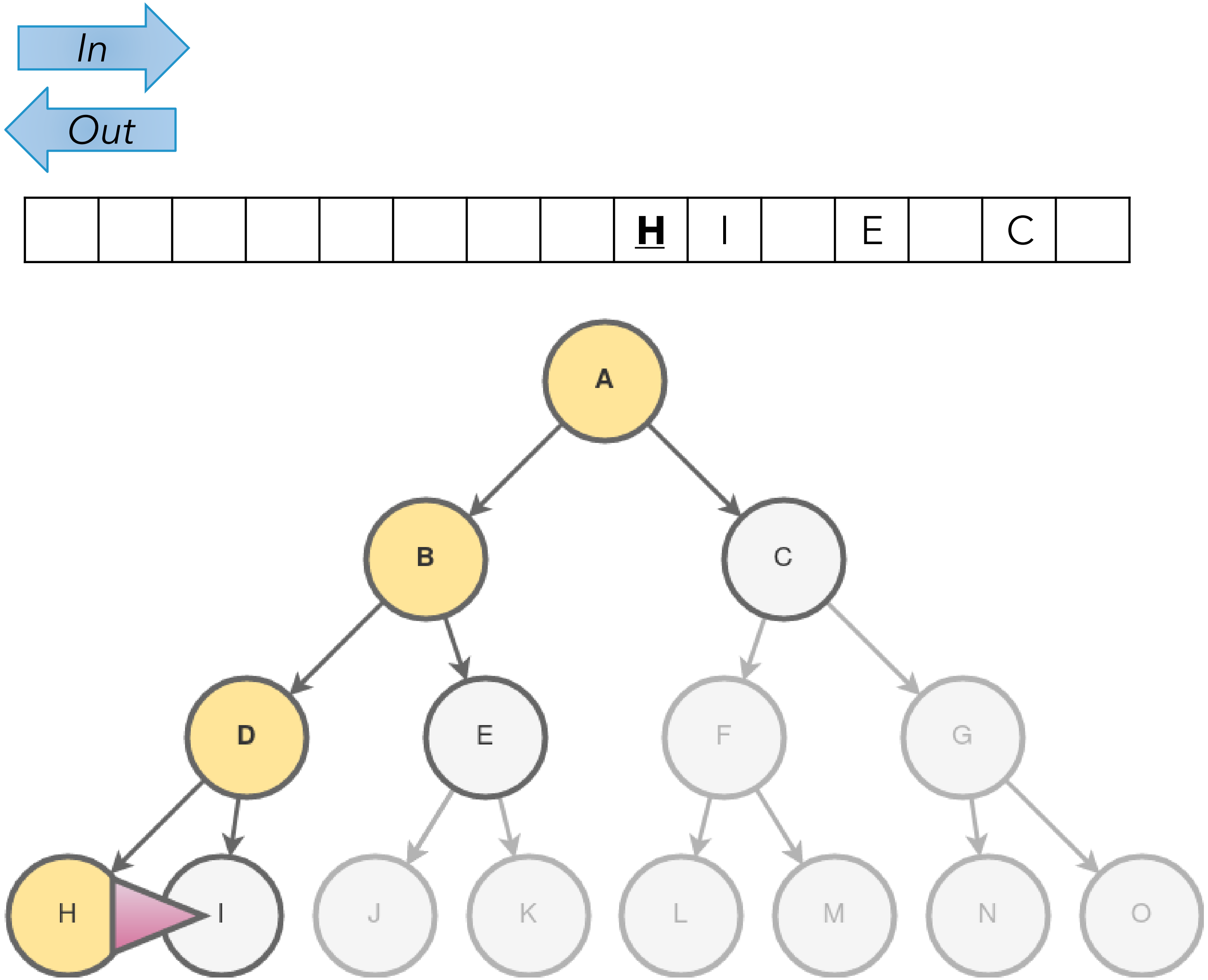


Depth-first search

Expand **deepest** unexpanded node

Implementation:

- *frontier* is a **LIFO** queue, i.e., new successors go at front

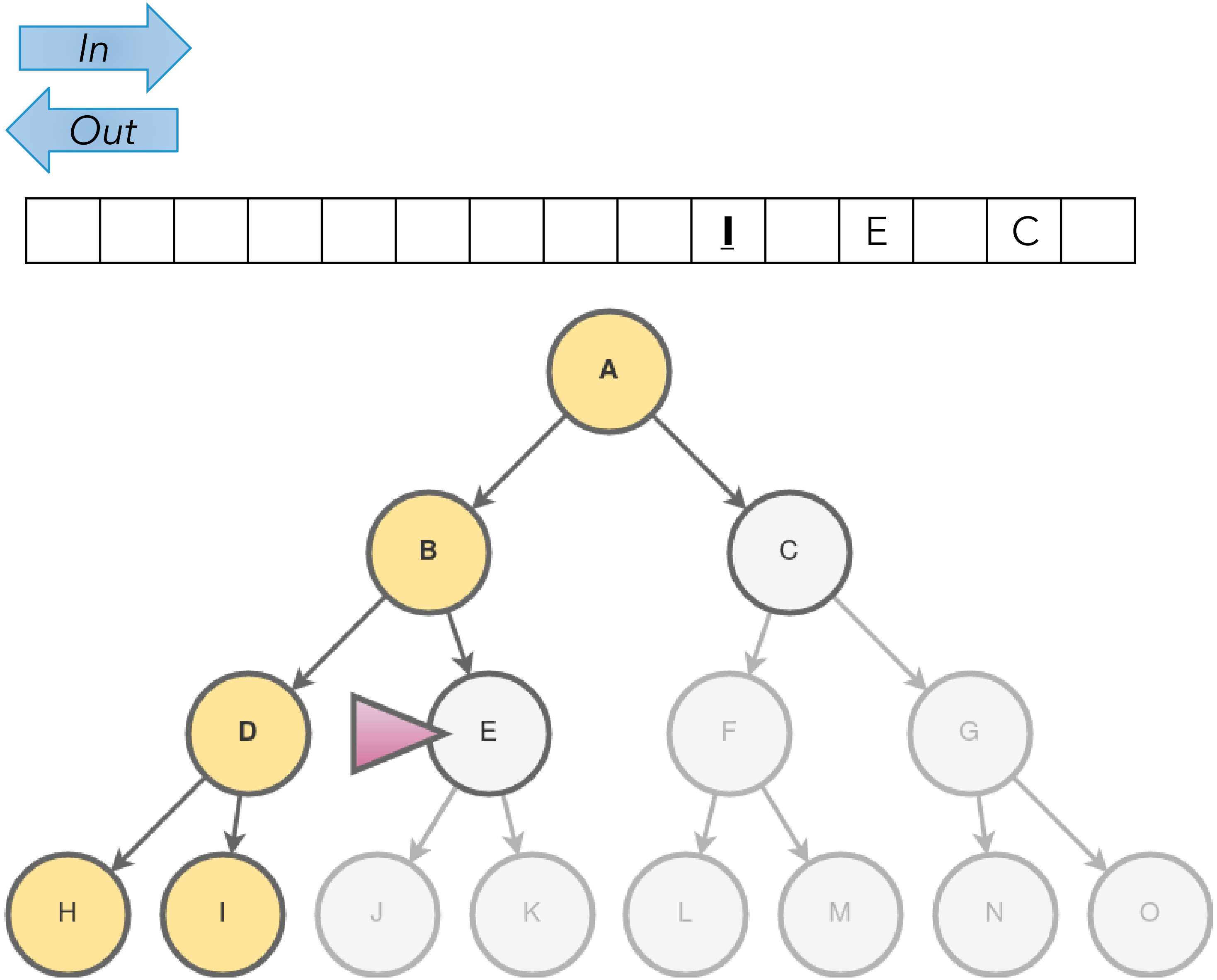


Depth-first search

Expand **deepest** unexpanded node

Implementation:

- *frontier* is a **LIFO** queue, i.e., new successors go at front

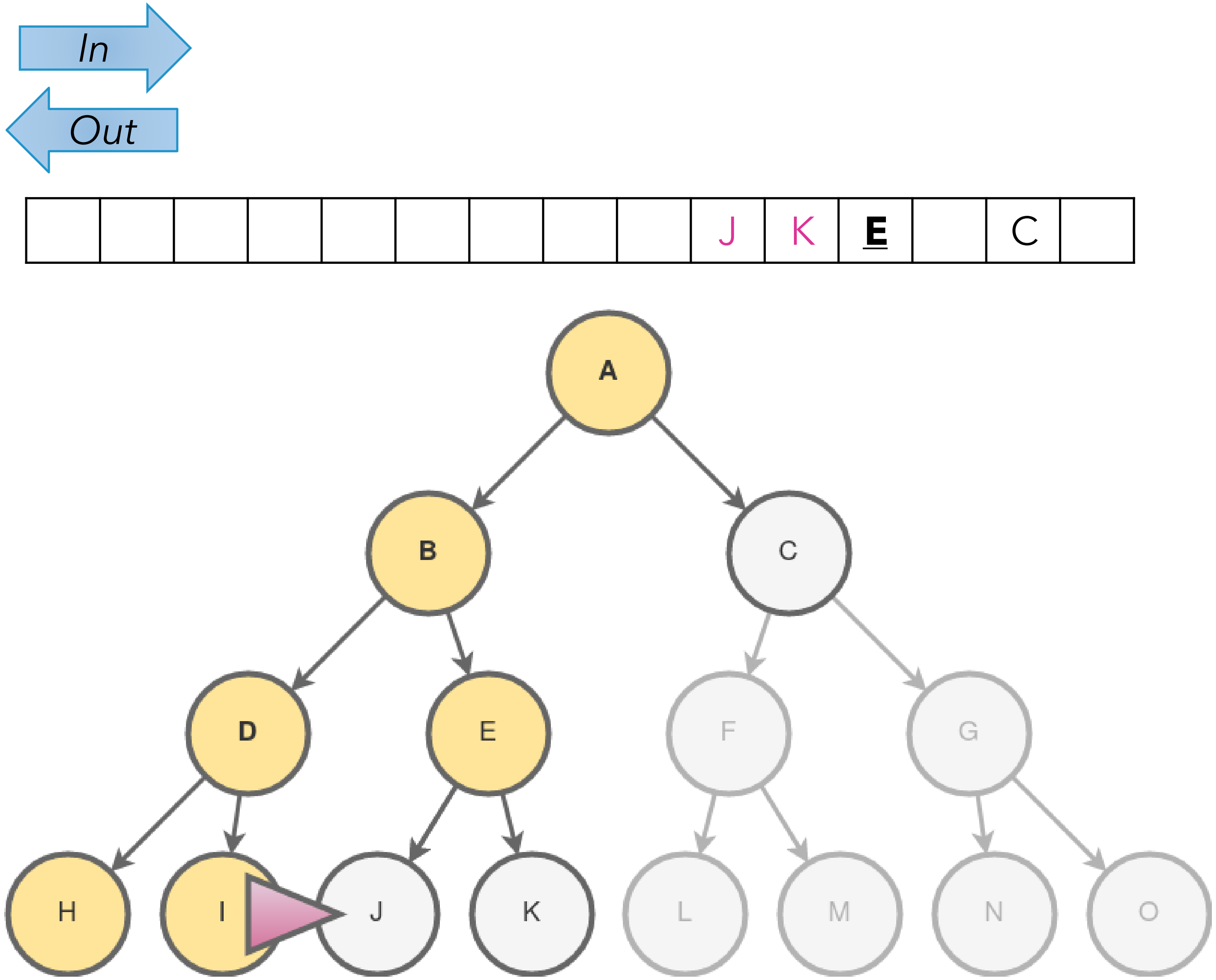


Depth-first search

Expand **deepest** unexpanded node

Implementation:

- *frontier* is a **LIFO** queue, i.e., new successors go at front

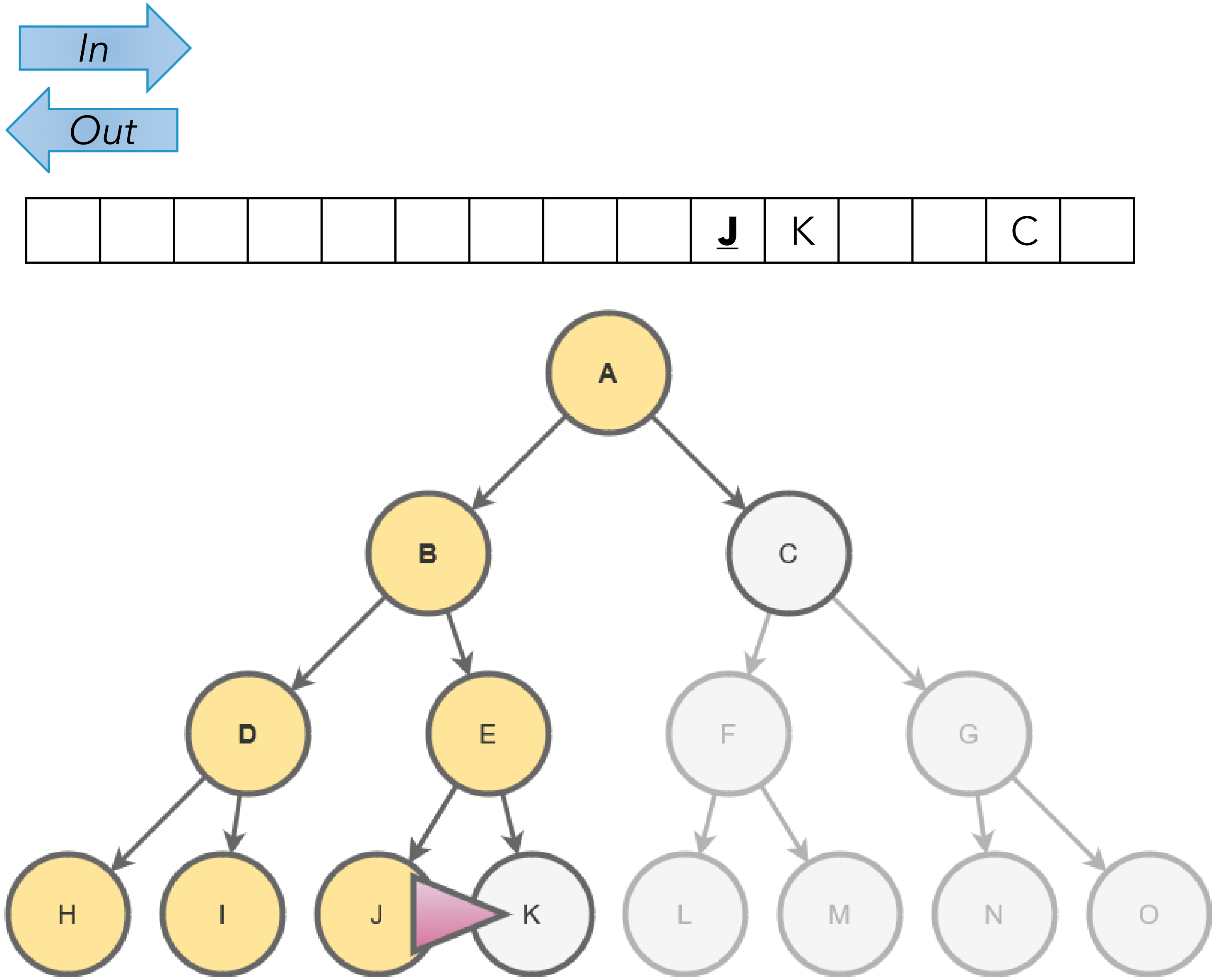


Depth-first search

Expand **deepest** unexpanded node

Implementation:

- *frontier* is a **LIFO** queue, i.e., new successors go at front

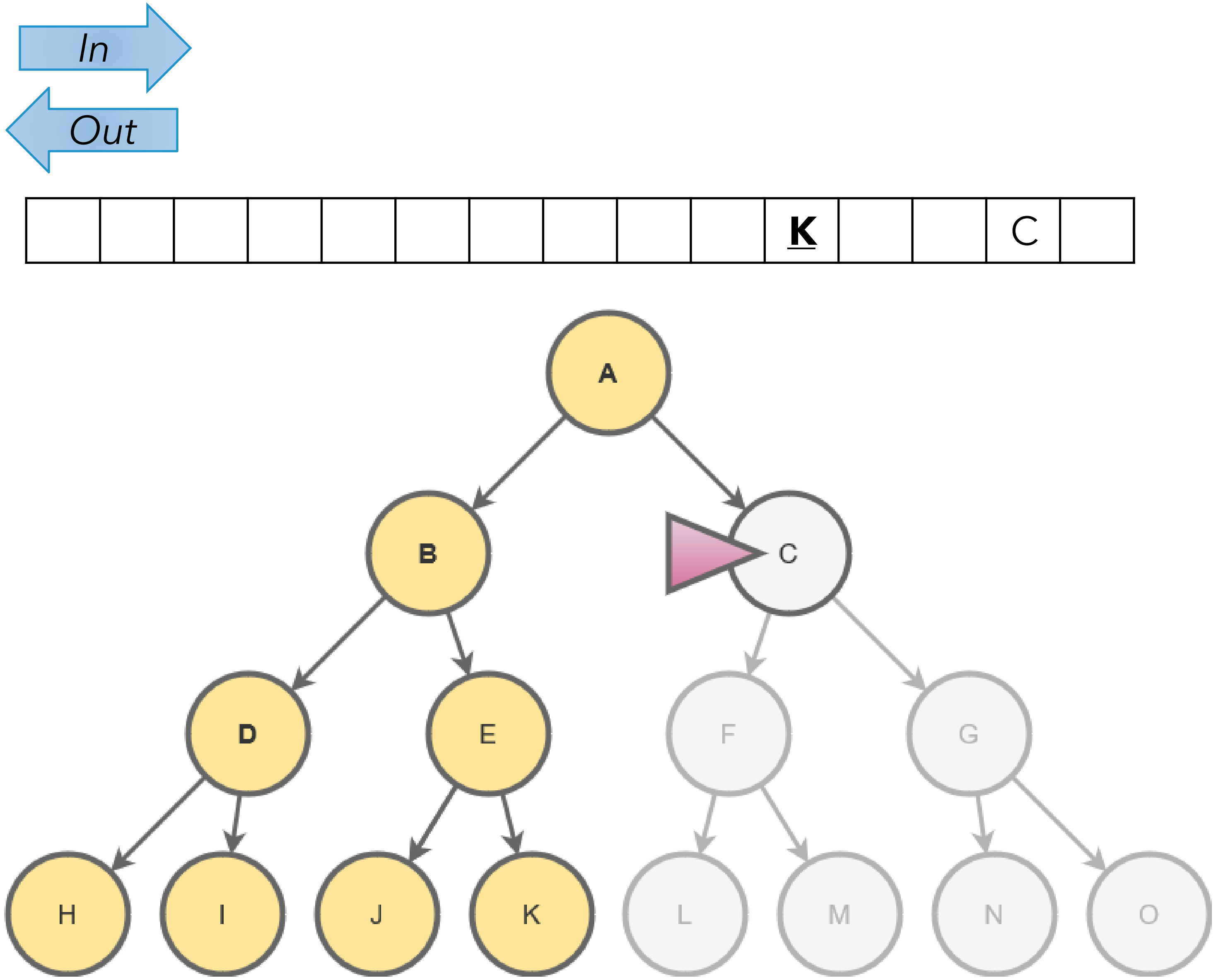


Depth-first search

Expand **deepest** unexpanded node

Implementation:

- *frontier* is a **LIFO** queue, i.e., new successors go at front

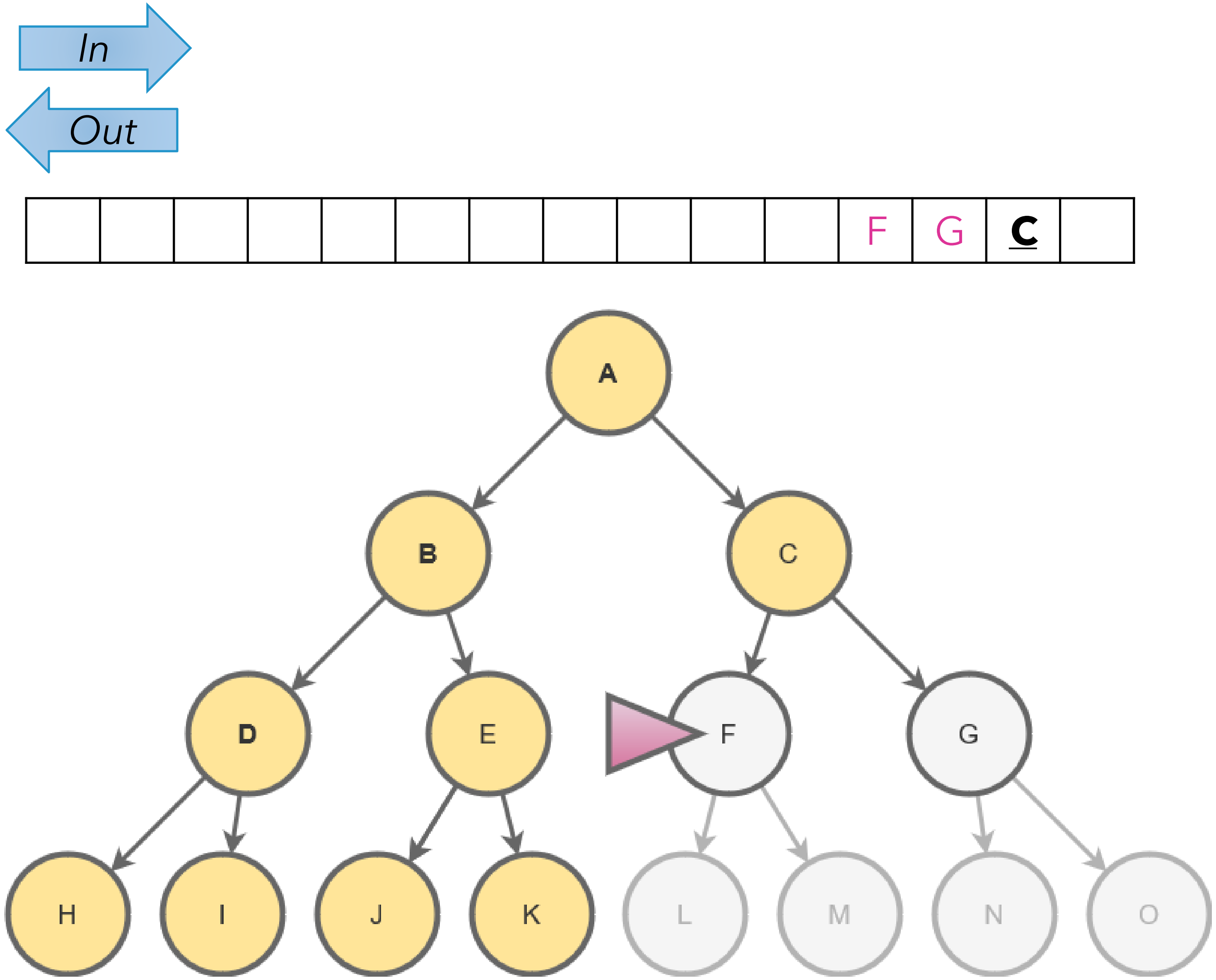


Depth-first search

Expand **deepest** unexpanded node

Implementation:

- *frontier* is a **LIFO** queue, i.e., new successors go at front



Properties of depth-first search

Complete?

Time complexity?

Space complexity?

Optimal?

Properties of depth-first search

Complete?
No: fails in infinite-depth spaces, spaces with loops

Time complexity?

Space complexity?

Optimal?

Properties of depth-first search

Complete?
No: fails in infinite-depth spaces, spaces with loops

avoid repeated states along path; complete in finite spaces

Time complexity?

Space complexity?

Optimal?

Properties of depth-first search

Complete?
No: fails in infinite-depth spaces, spaces with loops

Time complexity?
$O(b^m)$: terrible if m is much larger than d

Space complexity?

Optimal?

Properties of depth-first search

Complete?
No: fails in infinite-depth spaces, spaces with loops

Time complexity?
$O(b^m)$: terrible if m is much larger than d

If solutions are dense, depth-first may be much faster than breadth-first!

Space complexity?

Optimal?

Properties of depth-first search

Complete?
No: fails in infinite-depth spaces, spaces with loops

Time complexity?
$O(b^m)$: terrible if m is much larger than d

Space complexity?
$O(bm)$, i.e., linear space!

Optimal?

Properties of depth-first search

Complete?
No: fails in infinite-depth spaces, spaces with loops

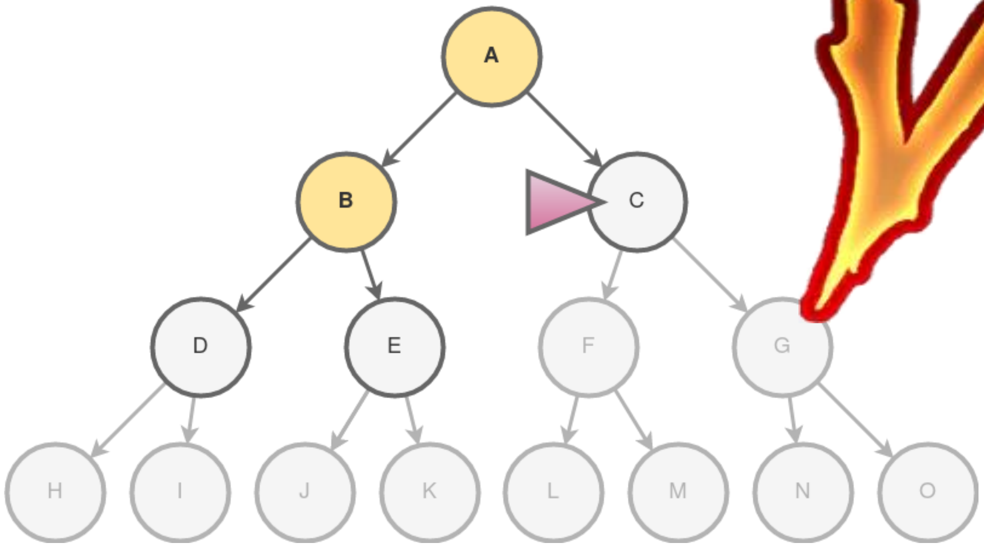
Time complexity?
$O(b^m)$: terrible if m is much larger than d

Space complexity?
$O(bm)$, i.e., linear space!

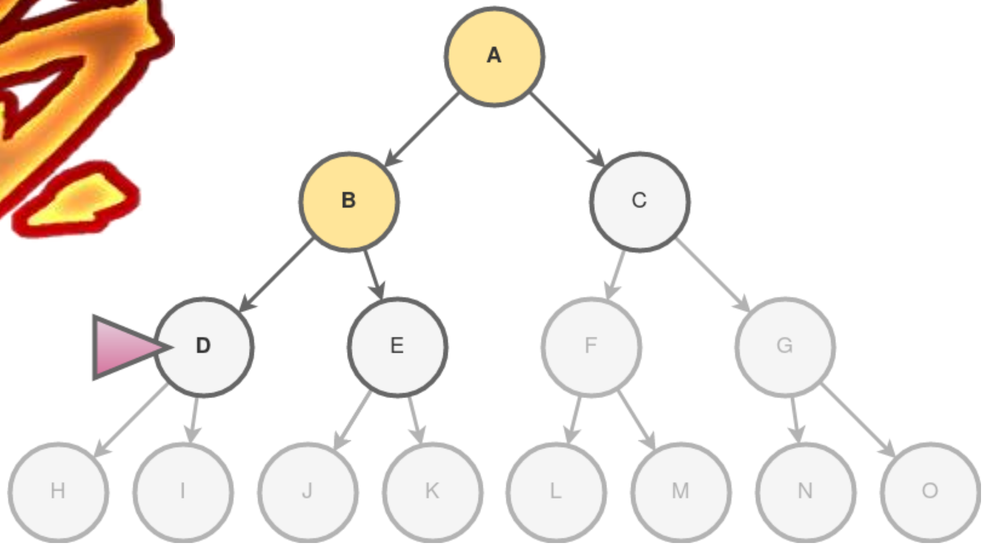
Optimal?
No

Mid-Lecture Exercise

BREADTH-FIRST

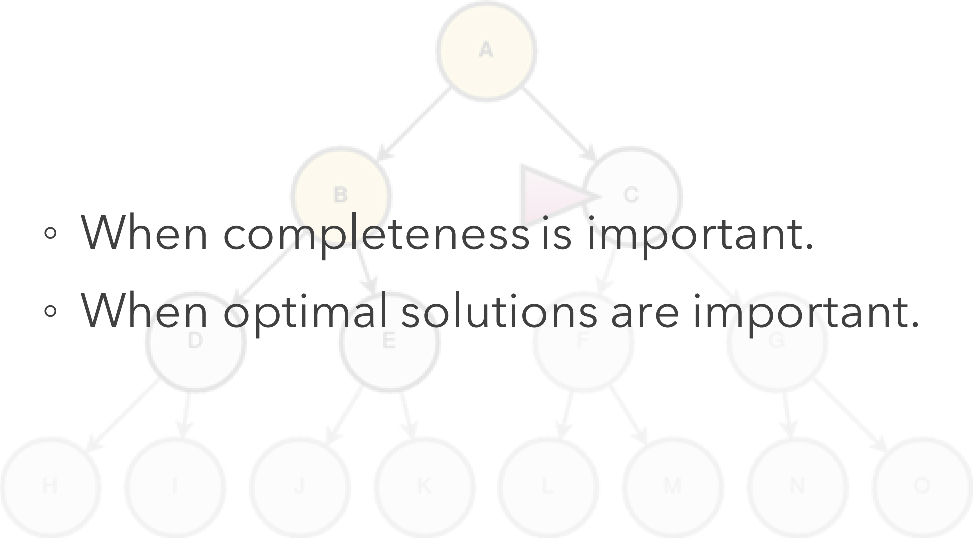


DEPTH-FIRST



Mid-Lecture Exercise

BREADTH-FIRST



- When completeness is important.
- When optimal solutions are important.

DEPTH-FIRST

- When solutions are dense and low-cost is important, especially space costs.

Iterative deepening search

… or how to improve depth-first search

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_occurred? ← false
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          result ← RECURSIVE-DLS(child, problem, limit − 1)
          if result = cutoff then cutoff_occurred? ← true
          else if result ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure
```

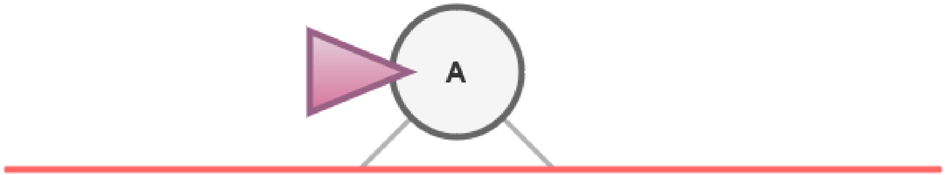
Depth-limited search

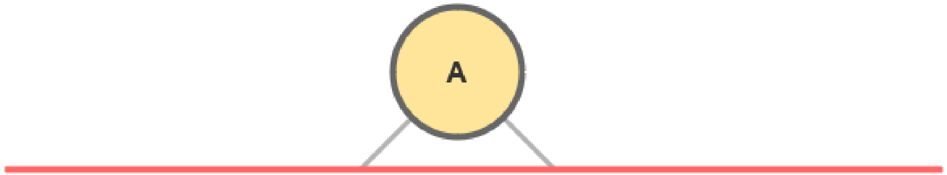
This is depth-first search with depth limit l, i.e., nodes at depth l have no successors

Iterative deepening search

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
   for depth = 0 to ∞ do
      result ← DEPTH-LIMITED-SEARCH(problem, depth)
      if result ≠ cutoff then return result
```

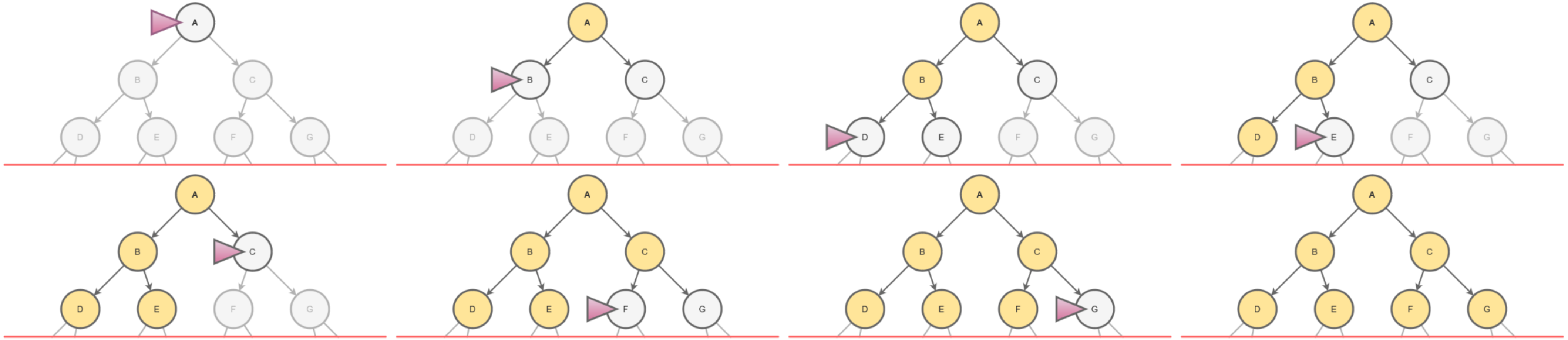
Iterative deepening search l =0



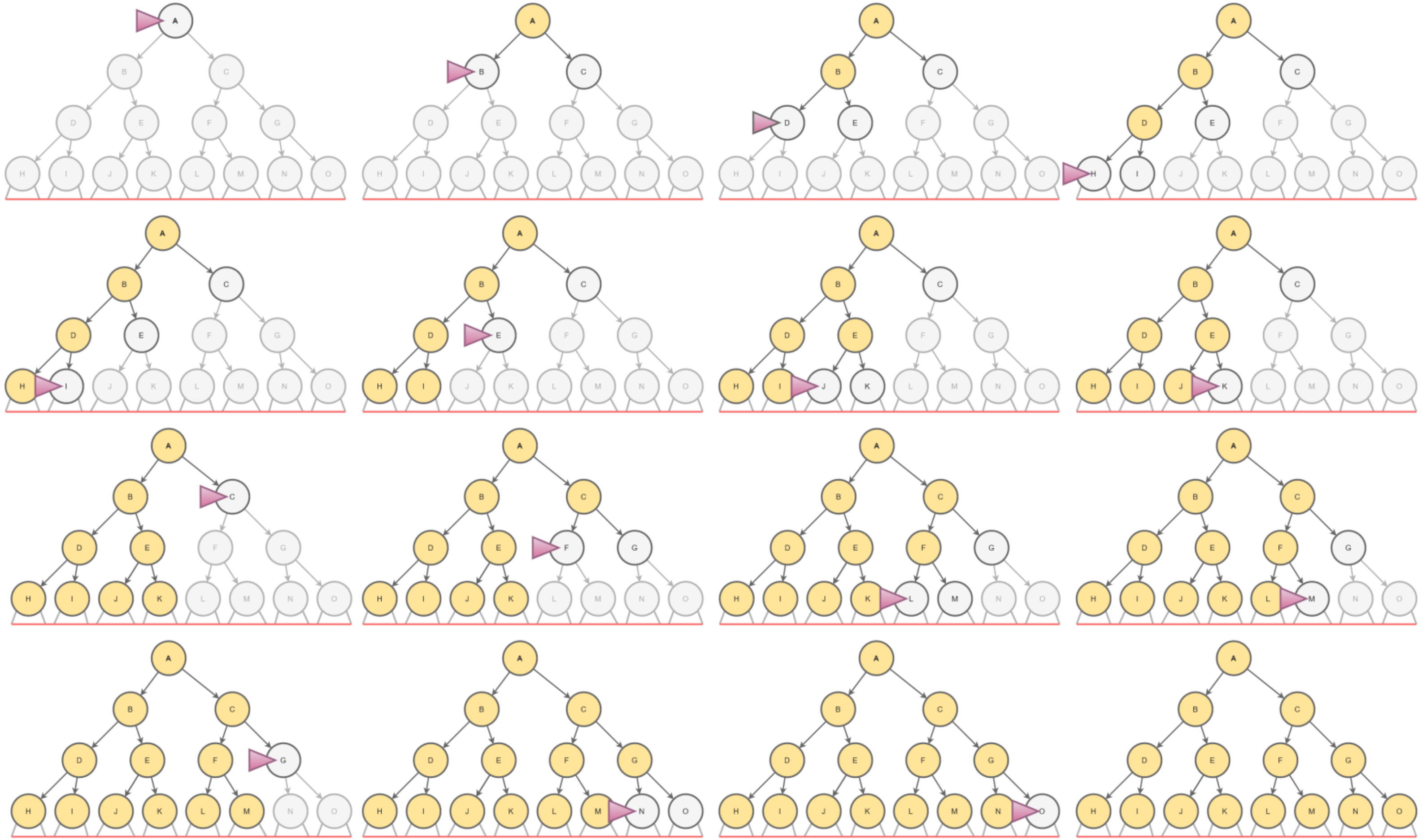


Iterative deepening search l =1

Iterative deepening search l =2



Iterative deepening search *l* =3



Iterative deepening search

Number of nodes generated in an iterative deepening search to depth *d* with branching factor *b*:

$$N_{IDS} = (d)b + (d-1)\, b^2 + \ldots + (2)b^{d-1} + (1)b^d$$

Some cost associated with generating upper levels multiple times

Example: For $b = 10$, $d = 5$,

- $N_{BFS} = 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}110$
- $N_{IDS} = 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}450$

Overhead = (123,450 - 111,110)/111,110 = 11%

Properties of iterative deepening search

Complete?

Time complexity?

Space complexity?

Optimal?

Properties of iterative deepening search

Complete?
Yes

Time complexity?

Space complexity?

Optimal?

Properties of iterative deepening search

Complete?
Yes

Time complexity?
$(d)b + (d\text{-}1)b^2 + \ldots + (1)b^d = O(b^d)$

Space complexity?

Optimal?

Properties of iterative deepening search

Complete?
Yes

Time complexity?
$(d)b + (d\text{-}1)b^2 + \ldots + (1)b^d = O(b^d)$

Space complexity?
$O(bd)$

Optimal?

Properties of iterative deepening search

Complete?
Yes

Time complexity?
$(d)b + (d\text{-}1)b^2 + \ldots + (1)b^d = O(b^d)$

Space complexity?
$O(bd)$

Optimal?
Yes, if step cost = 1

Summary of algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes	No	No	Yes
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes	Yes	No	No	Yes

Summary

Variety of uninformed search strategies:

- breadth-first, depth-first, iterative deepening

Iterative deepening search uses only linear space and not much more time than other uninformed algorithms

Why?

- Very common algorithms.
- Used whenever we are looking for a path between 2 points in a tree or graph.
  - Anywhere from games to programming languages.
- Properties matter!
- e.g. time or space complexity may be important depending on application.
- Understanding which algorithm to use in what circumstances.