Previously…

- Introduction and history of AI
- Agents: rational agent, agent types
- Task environment (i.e., PEAS) and its properties

Today…

- Uninformed Search

SOLVING PROBLEMS BY SEARCHING

AIMA Chapter 3.1 – 3.4

Outline

- Problem-solving agent: A type of goal-based agent using atomic representation
- Problem definition/formulation
- Example problems
- Uninformed search algorithms

Problem-Solving Agent

- Assume that task environment is fully observable, deterministic, and discrete → Solution is fixed sequence of actions

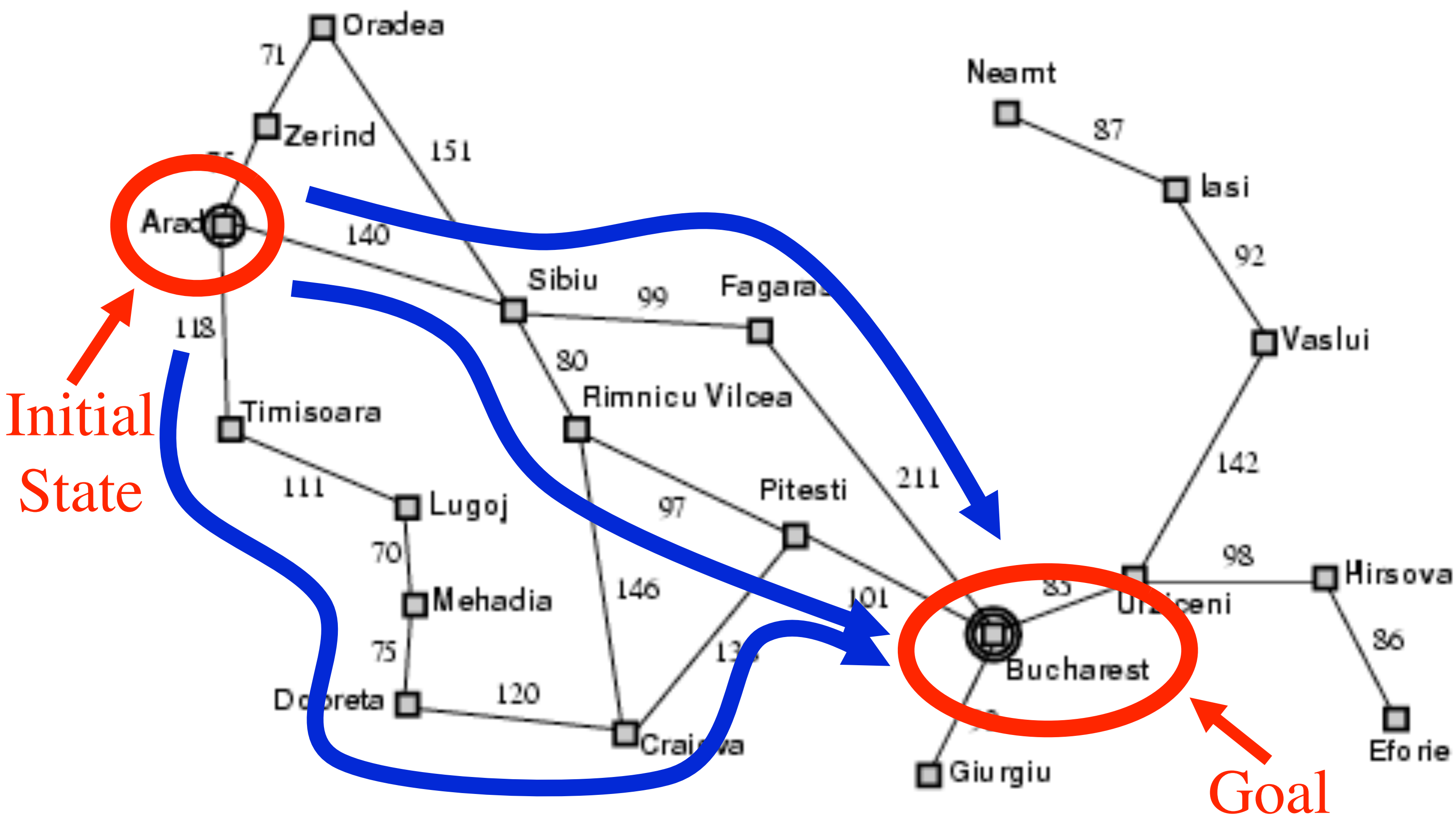
```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: seq, an action sequence, initially empty
              state, some description of the current world state
              goal, a goal, initially null
              problem, a problem formulation

  state ← UPDATE-STATE(state, percept)
  if seq is empty then
     goal ← FORMULATE-GOAL(state)
     problem ← FORMULATE-PROBLEM(state, goal)
     seq ← SEARCH(problem)
     if seq = failure then return a null action
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

- Are percepts needed to execute rest of sequence of actions?
  No! → Open-loop system

Example: Romania

- On tour in Romania; currently in Arad
- Flight leaves tomorrow from Bucharest
- Formulate goal:
  - be in Bucharest
- Formulate problem:
  - states: various cities
  - actions: drive between cities
- Solve search problem:
  - sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Example: Romania



Problem Formulation

A problem is defined by 6 components:

1. Initial state : e.g., $In(Arad)$
2. States : e.g., $\{In(Arad), In(Sibiu), In(Timisoara), In(Zerind), \ldots\}$
3. Actions : ACTIONS(s) returns set of actions that can executed in state s. e.g., ACTIONS($In(Arad)$) = $\{Go(Sibiu), Go(Timisoara), Go(Zerind)\}$
4. Transition model : RESULT(s, a) returns state that results from doing action a in state s. e.g., RESULT($In(Arad)$, $Go(Zerind)$) = $In(Zerind)$

Problem Formulation

A problem is defined by 6 components:

5. Goal test determines whether a given state is a goal state
   - Explicit set of goal states, e.g., $\{In(Bucharest)\}$
   - Implicit function, e.g., $Checkmate(s)$
6. Path cost is a cost function assigning a numeric cost to each path
   - Additive. e.g., sum of distances, number of actions executed
   - $c(s, a, s')$ is the step cost of taking action a in state s to reach state s'. Assumed to be ≥ 0

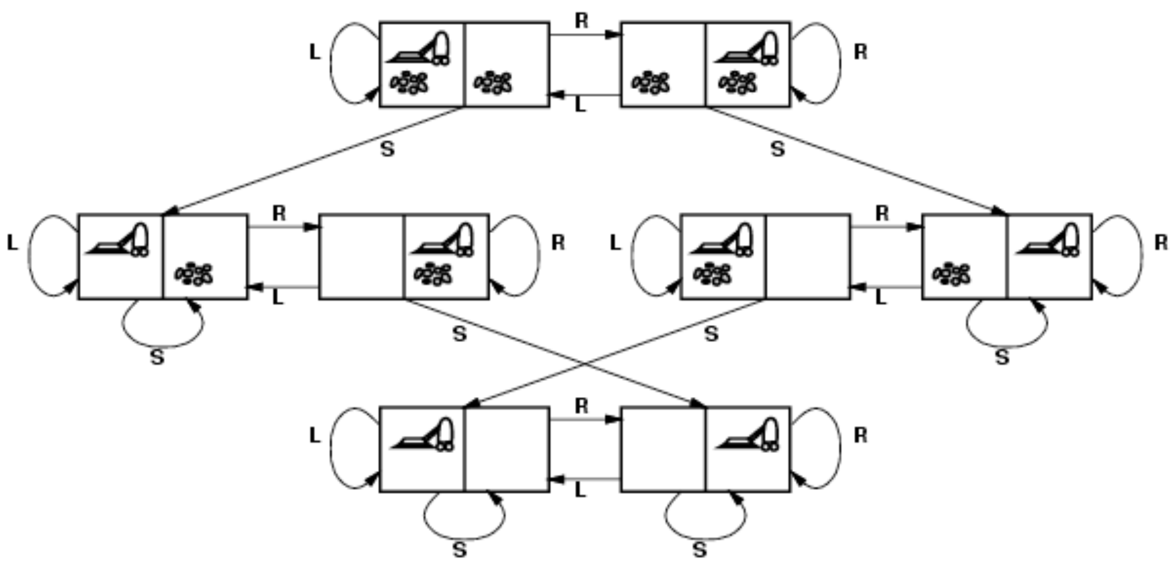
A solution is a sequence of actions leading from the initial state to a goal state

Defining State Space

- Real world is absurdly complex → state space must be abstracted for problem solving
- (Abstract) state = set of real states
- (Abstract) action = complex combination of real actions

  e.g., When in state *In*(*Arad*), action *Go*(*Zerind*) represents a complex set of possible routes, detours, rest stops, etc.
- (Abstract) solution = set of real paths that are solutions in the real world
- Valid abstraction: e.g., for any real state "in Arad", there is a real path to some real state "in Zerind"
- Useful abstraction: executing each abstract action is "easier" than the original problem

Example: Vacuum World



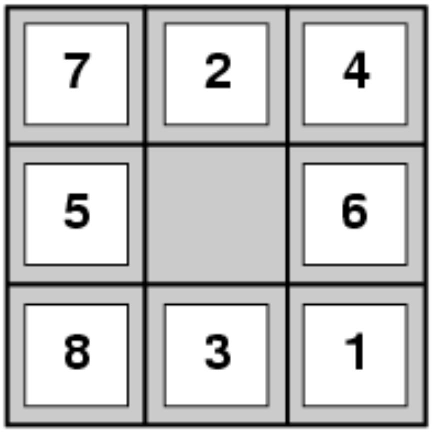
(LEFT) State-space graph

1. Initial state : Any state
2. States : Unique integer to represent agent and dirt locations
3. Actions : $\{Left, Right, Suck\}$

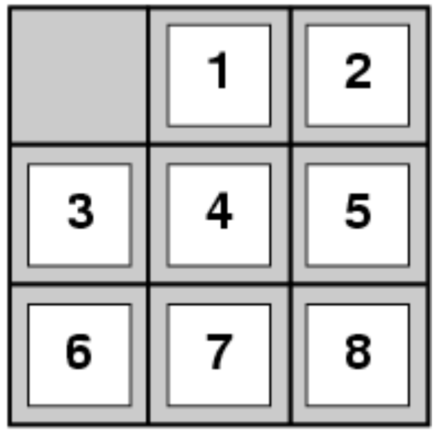
4. Transition model :

4. Goal test : no dirt at all locations
5. Path cost : number of steps in path (step cost of 1)

Example: 8-Puzzle



Start State



Goal State

1. Initial state : Any state
2. States : Locations of tiles
3. Actions : Move blank *Left*, *Right*, *Up*, or *Down*
4. Transition model : e.g., If we apply *Left* to start state, the resulting state has 5 and blank switched
5. Goal test : goal state is given
6. Path cost : number of steps in path (step cost of 1)

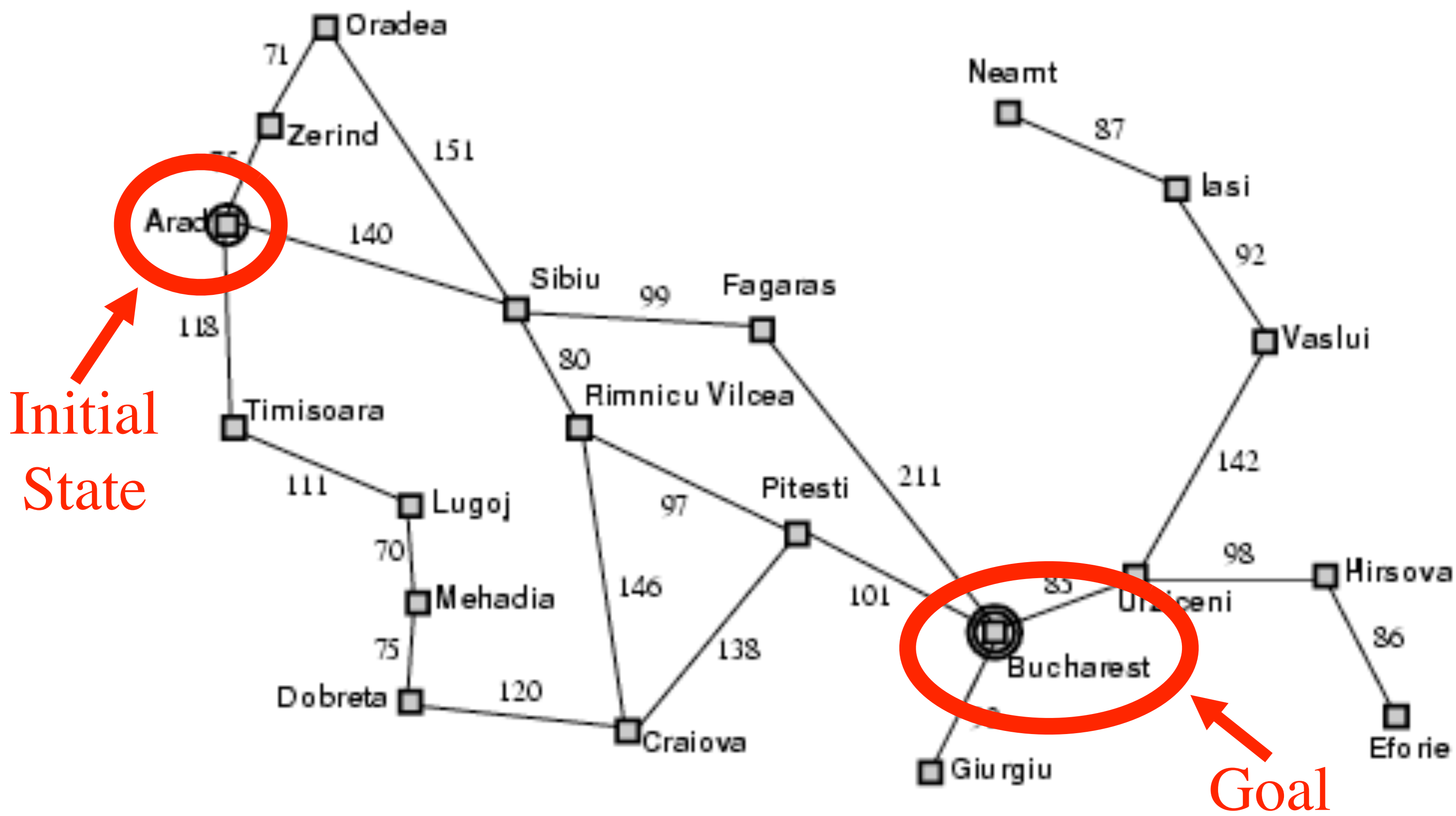
Note: Optimal solution of n-puzzle family is NP-hard.

Tree Search Algorithms

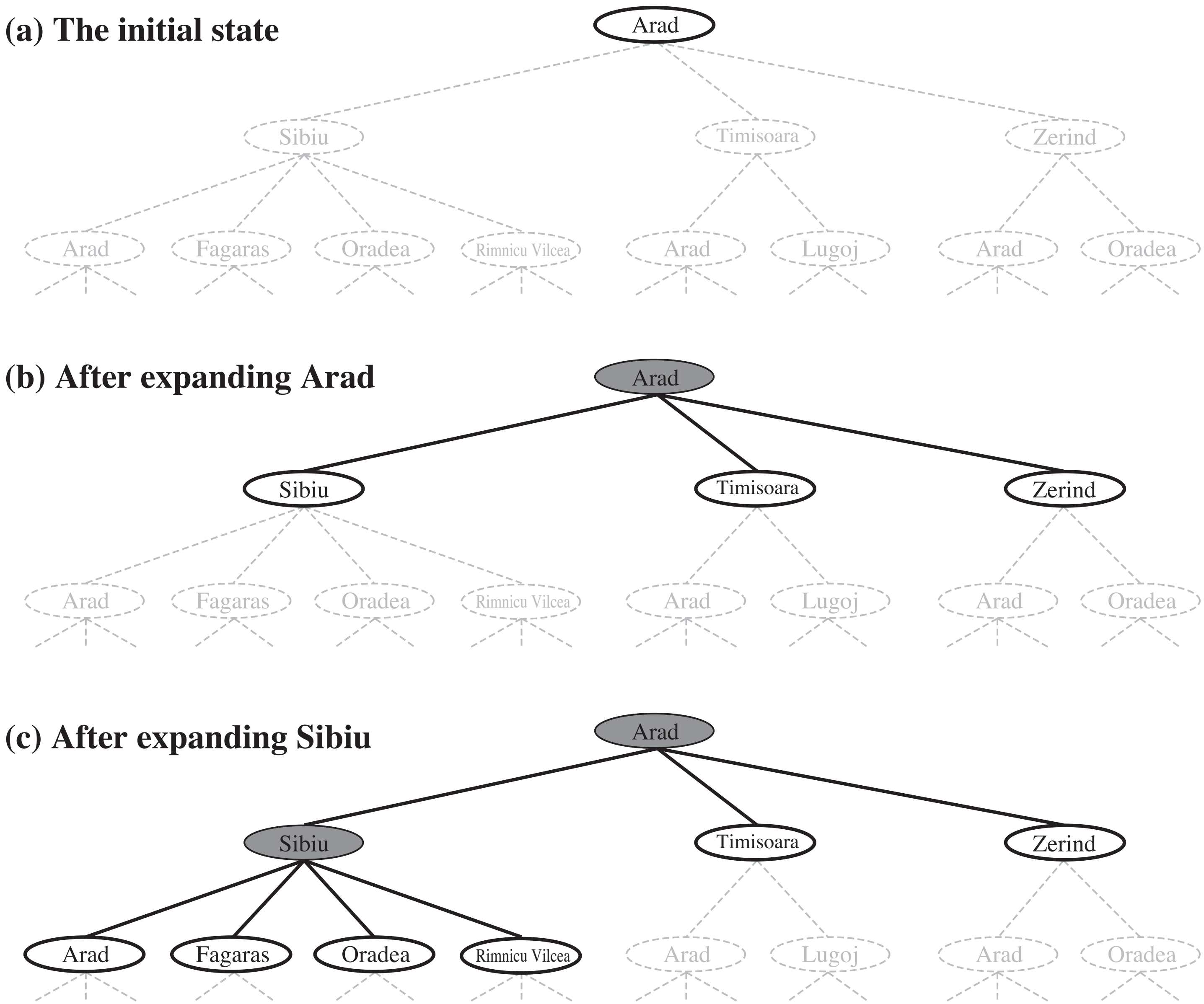
Basic idea: offline, simulated exploration of state space by generating successors of already-explored states (a.k.a. expanding states)

```
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
```

Example: Romania

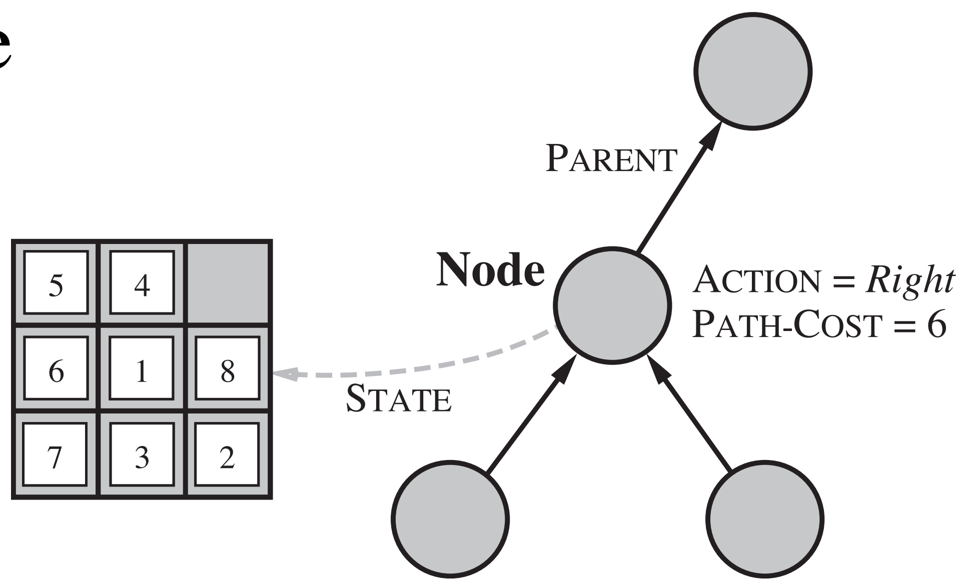


Tree Search Example



Implementation: States vs. Nodes

- A state represents a physical configuration
- A node is a data structure constituting part of search tree. It includes state, parent node, action, and path cost $g(n)$.
- Two different nodes are allowed to contain same world state



Search Strategies

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

- Strategies are evaluated with the following criteria:
  - **completeness**: does it always find a solution if one exists?
  - **optimality**: does it always find a least-cost solution?
  - **time complexity**: number of nodes generated during search
  - **space complexity**: maximum number of nodes in memory

- Time and space complexity are measured in terms of
  - b: maximum branching factor of search tree
  - d: depth of shallowest goal node
  - m: maximum depth of search tree (may be ∞)

Uninformed Search Strategies

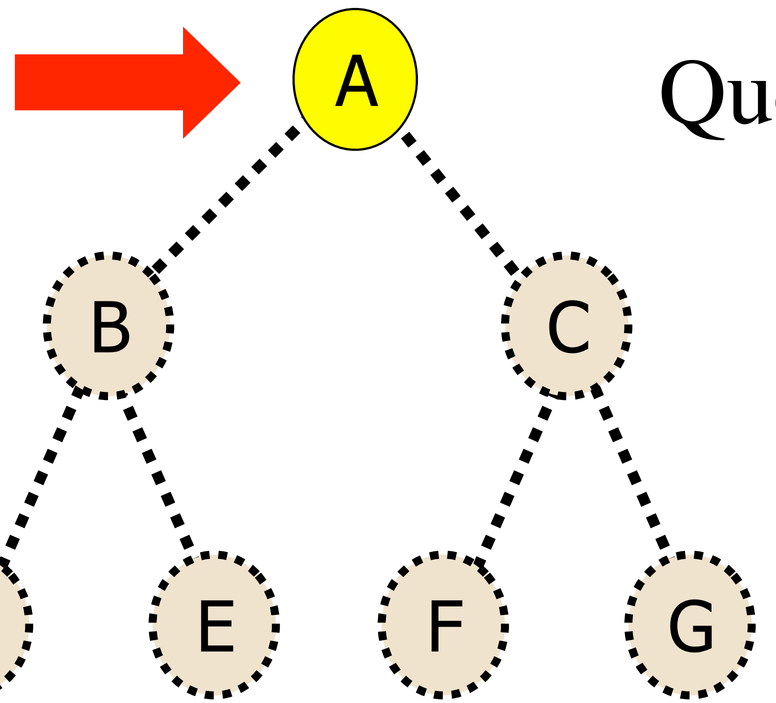
- Uninformed search strategies use only the information available in the problem definition
- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search

Breadth-First Search (BFS)

- Idea: Expand shallowest unexpanded node
- Implementation: *Frontier* is a FIFO queue, i.e., insert new successors at the end

Breadth-First Search (BFS)

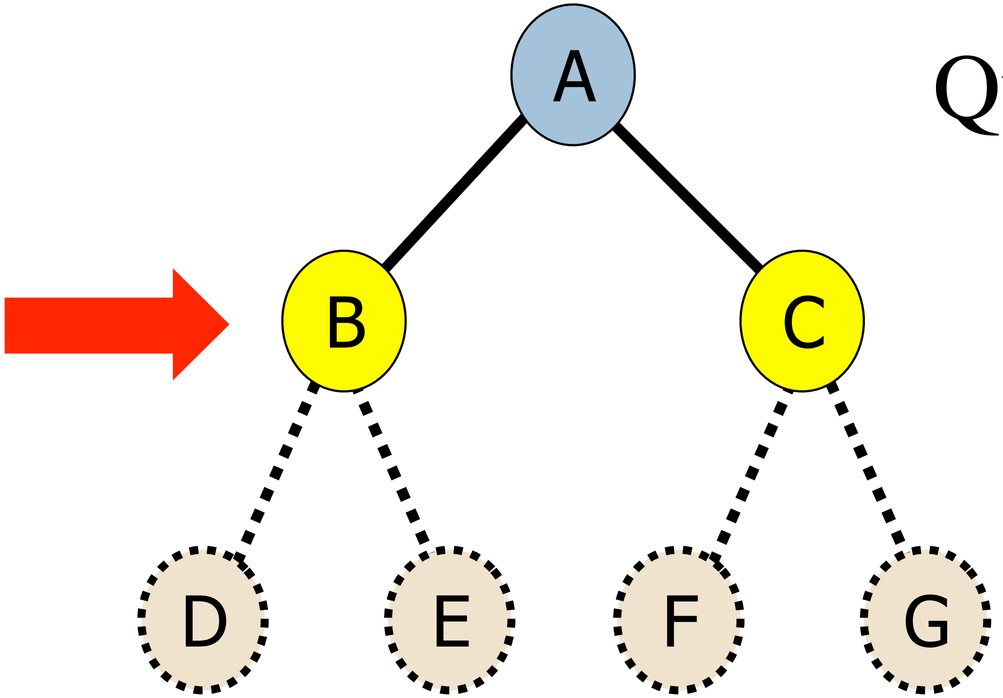
- Idea: Expand shallowest unexpanded node



Queue: {A}

Breadth-First Search (BFS)

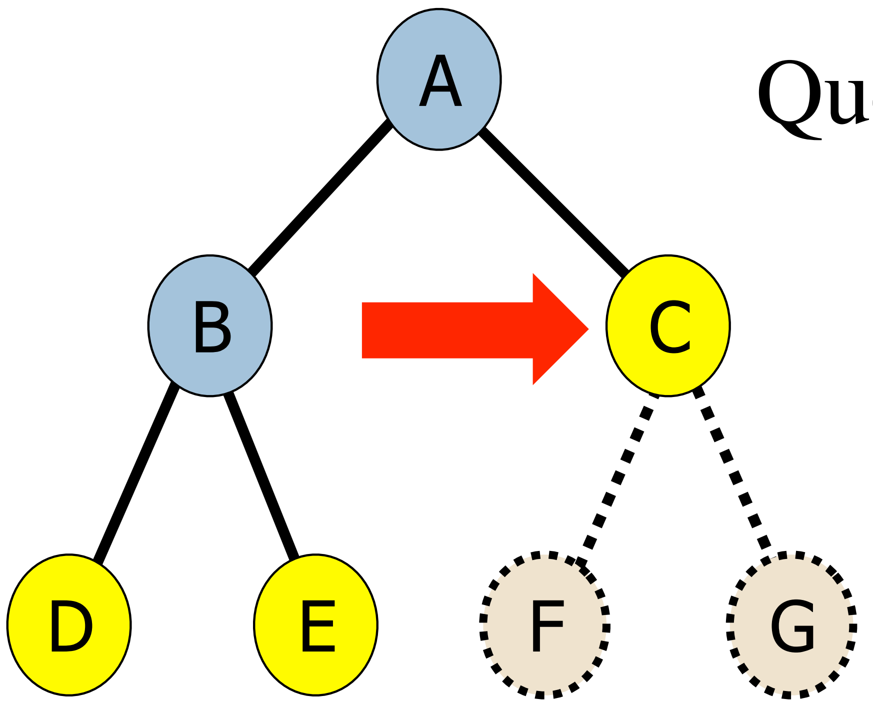
- Idea: Expand shallowest unexpanded node



Queue: {B,C}

Breadth-First Search (BFS)

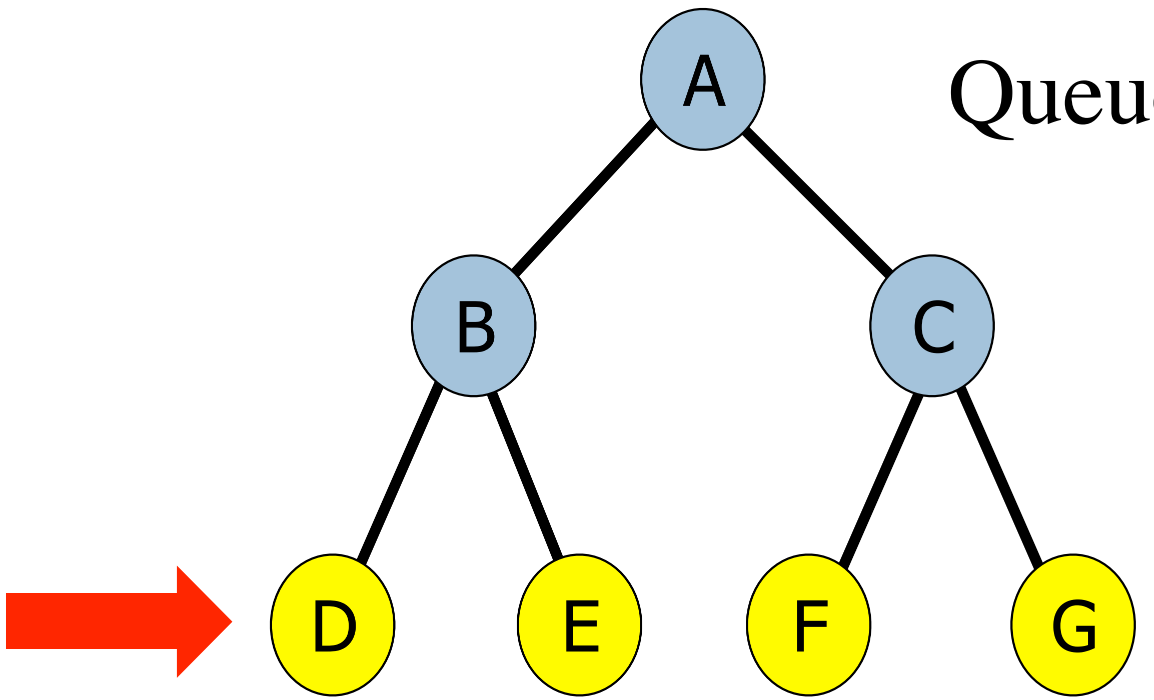
- Idea: Expand shallowest unexpanded node



Queue: {C,D,E}

Breadth-First Search (BFS)

- Idea: Expand shallowest unexpanded node



Queue: {D,E,F,G}

Properties of Breadth-First Search

- Complete? Yes (if b is finite)
- Optimal? Yes (if step cost = 1)
- Time? $b+b^2+b^3+\ldots+b^d = O(b^d)$
- Space? $O(b^d)$ (keeps frontier nodes in memory)

Space is the bigger problem (more than time)

Uniform-Cost Search (UCS)

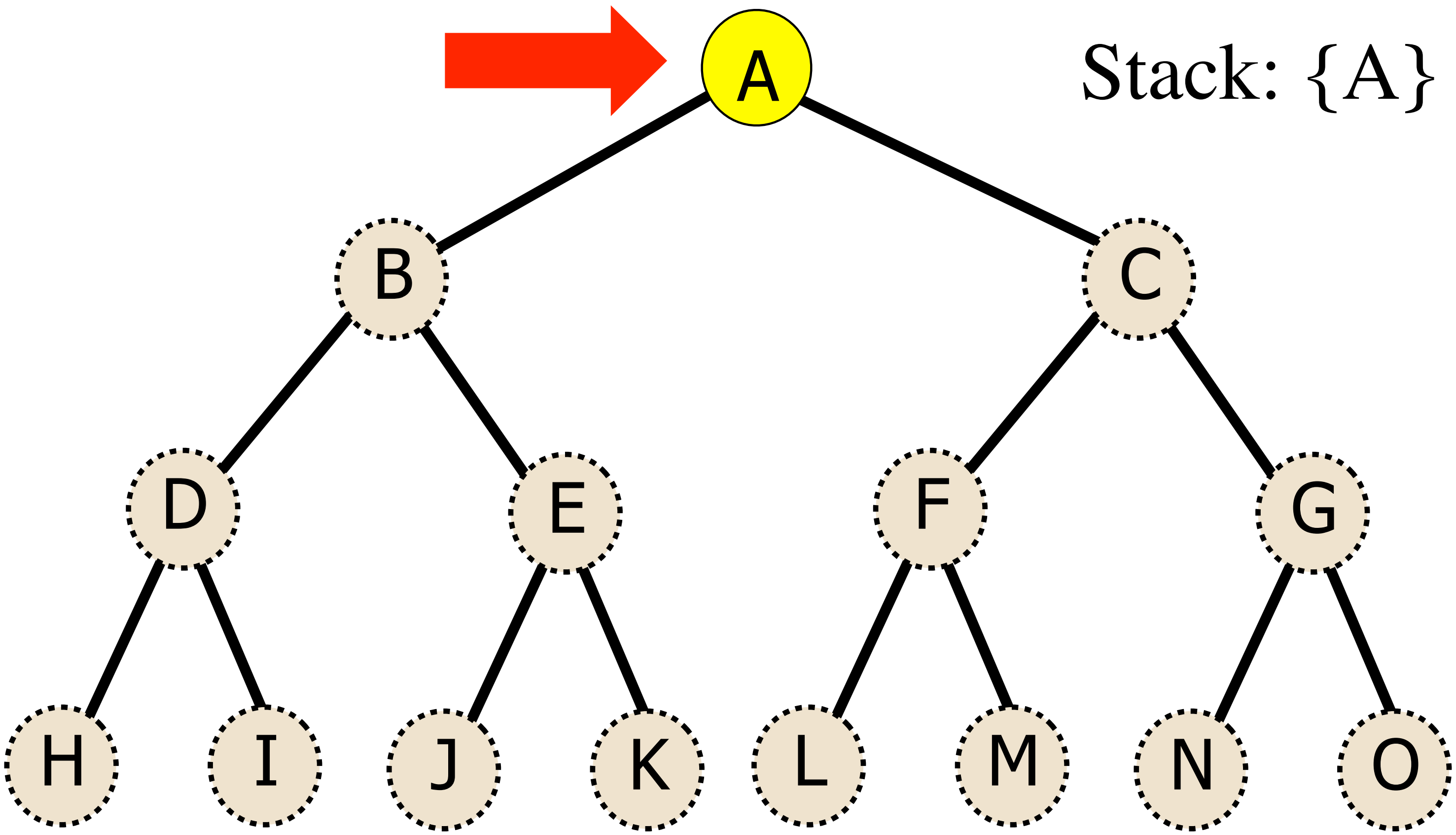
- Idea: Expand least-path-cost unexpanded node
- Implementation:
  Frontier = priority queue ordered by path cost g
- Similar to BFS if all step costs are equal
- Complete? Yes, if step cost ≥ ε
- Optimal? Yes – nodes expanded in increasing order of path cost $g(n)$
- Time? # of nodes with (path cost $g \leq$ cost C^* of optimal solution): $O(b^{1+floor(C^*/\varepsilon)})$
- Space? # of nodes with (path cost $g \leq$ cost C^* of optimal solution): $O(b^{1+floor(C^*/\varepsilon)})$

Depth-First Search

- Idea: Expand deepest unexpanded node
- Implementation: *Frontier* = LIFO stack, i.e., insert successors at the front

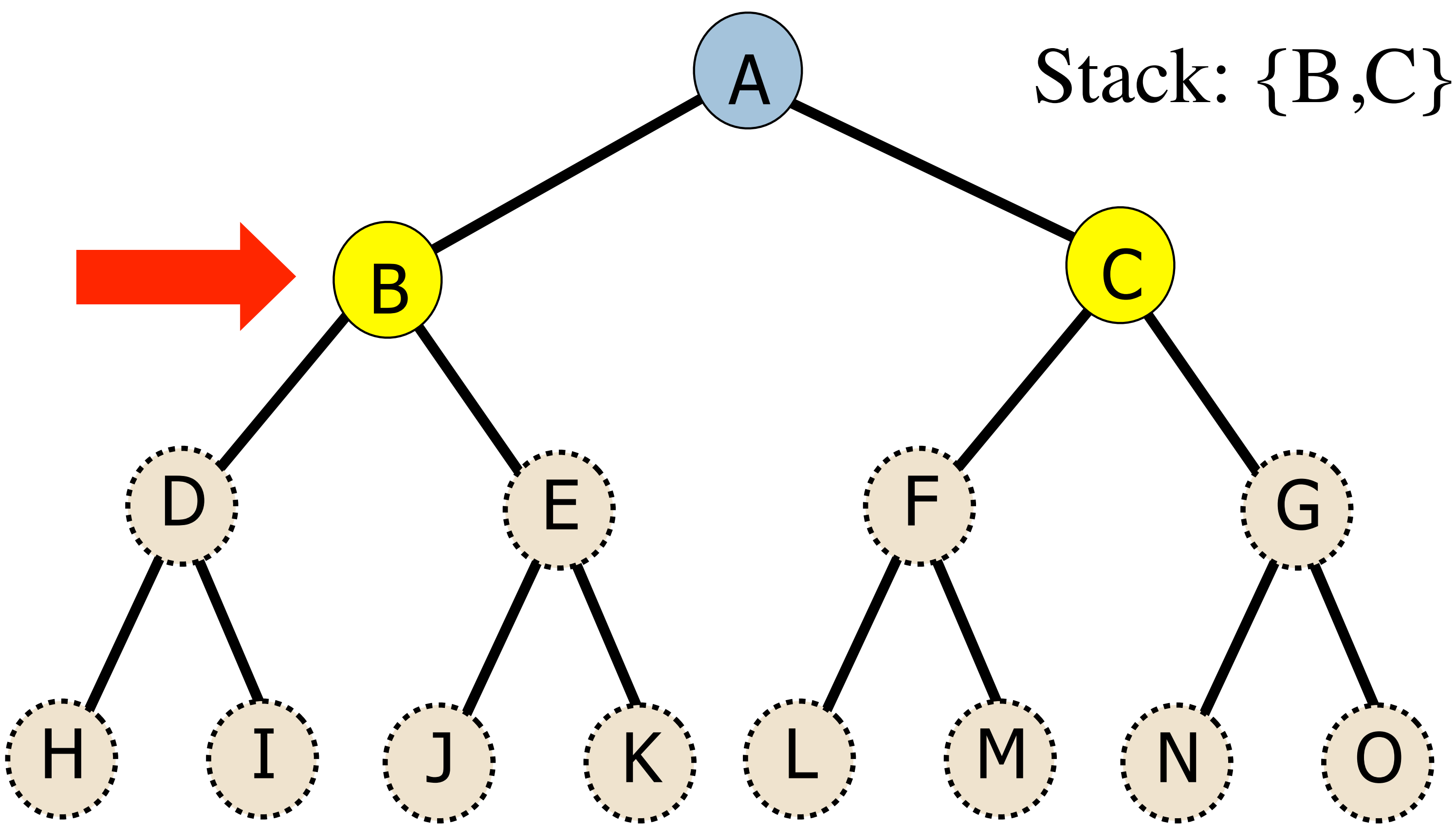
Depth-First Search

- Idea: Expand deepest unexpanded node



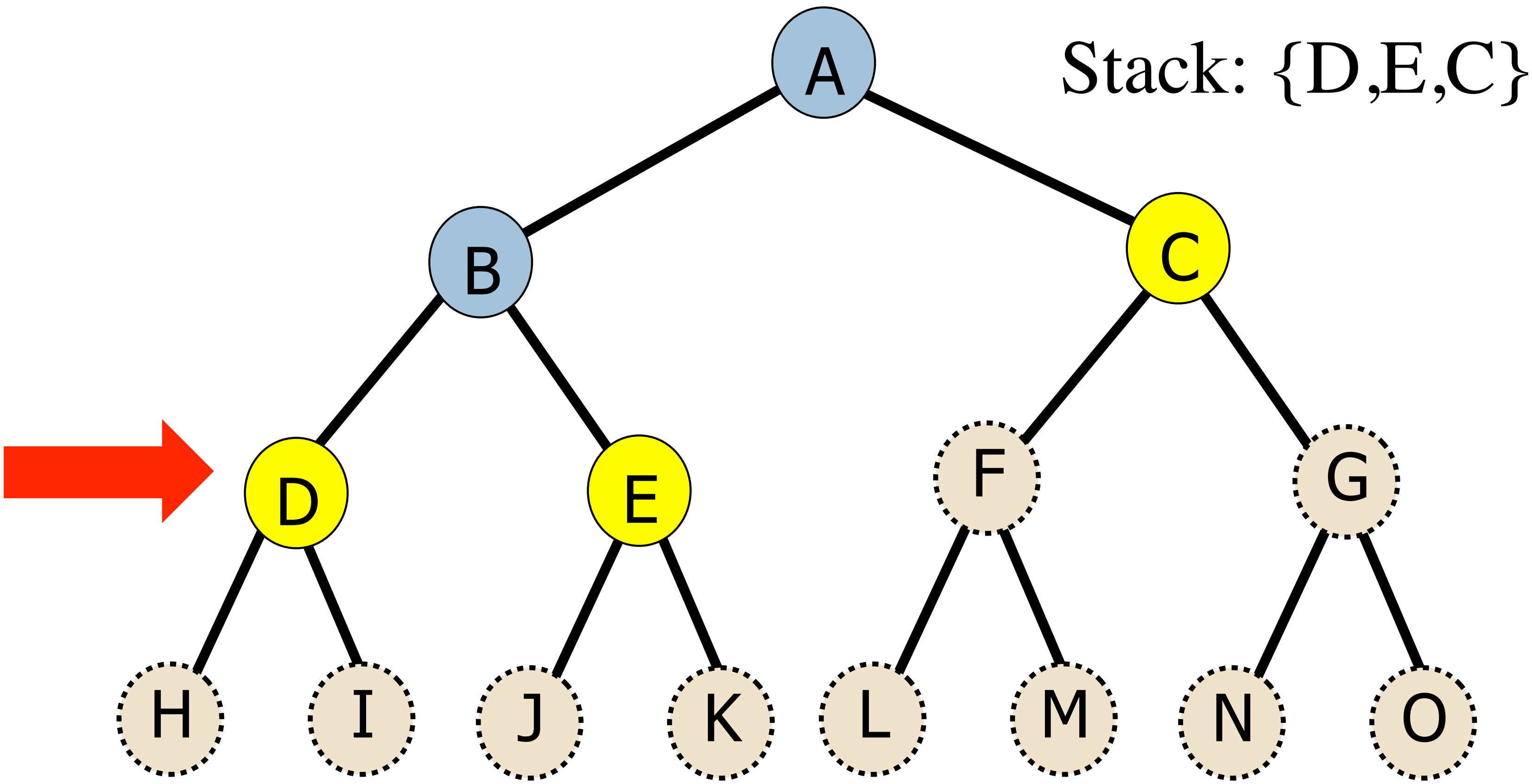
Depth-First Search

- Idea: Expand deepest unexpanded node



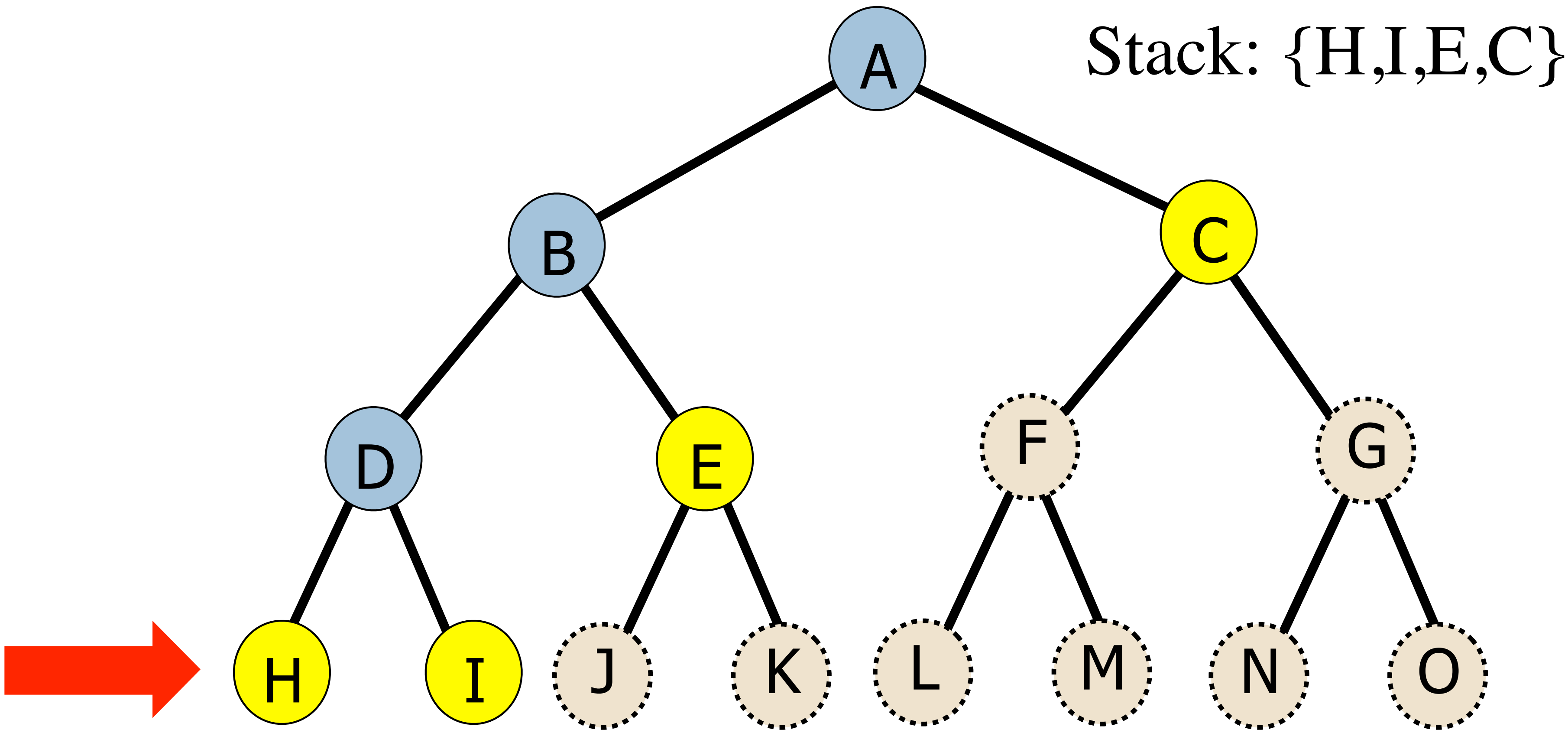
Depth-First Search

- Idea: Expand deepest unexpanded node



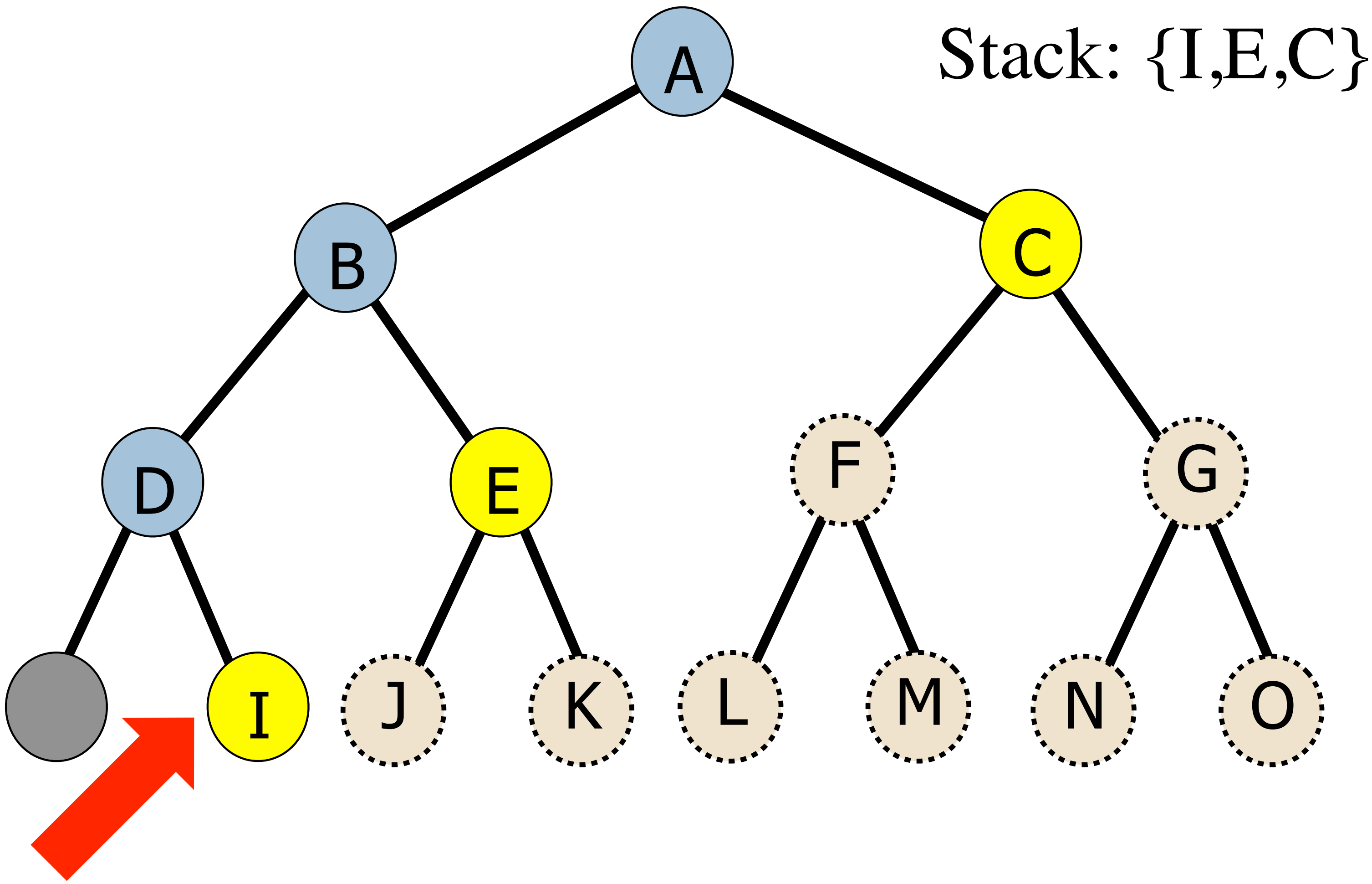
Depth-First Search

- Idea: Expand deepest unexpanded node



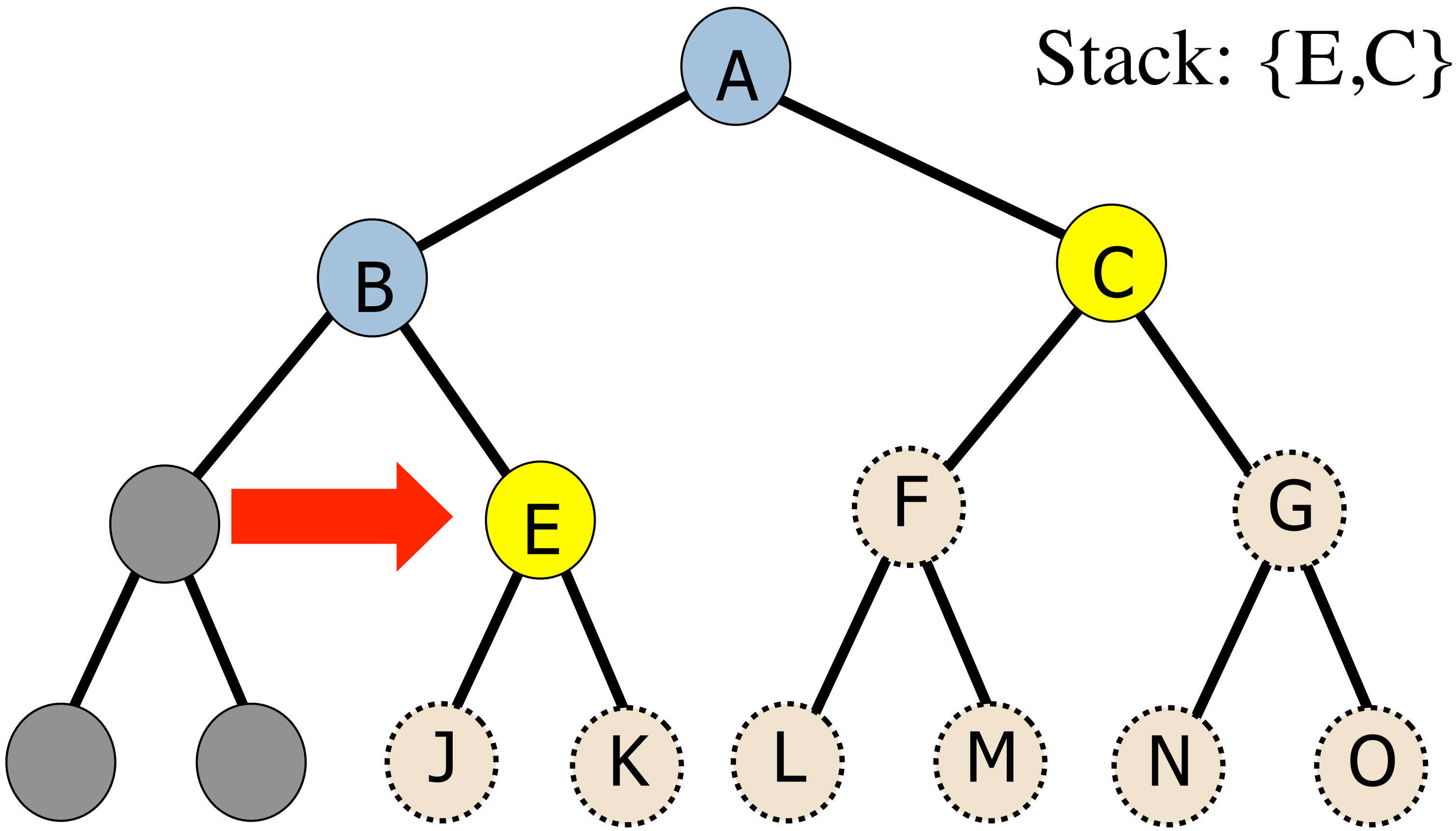
Depth-First Search

- Idea: Expand deepest unexpanded node



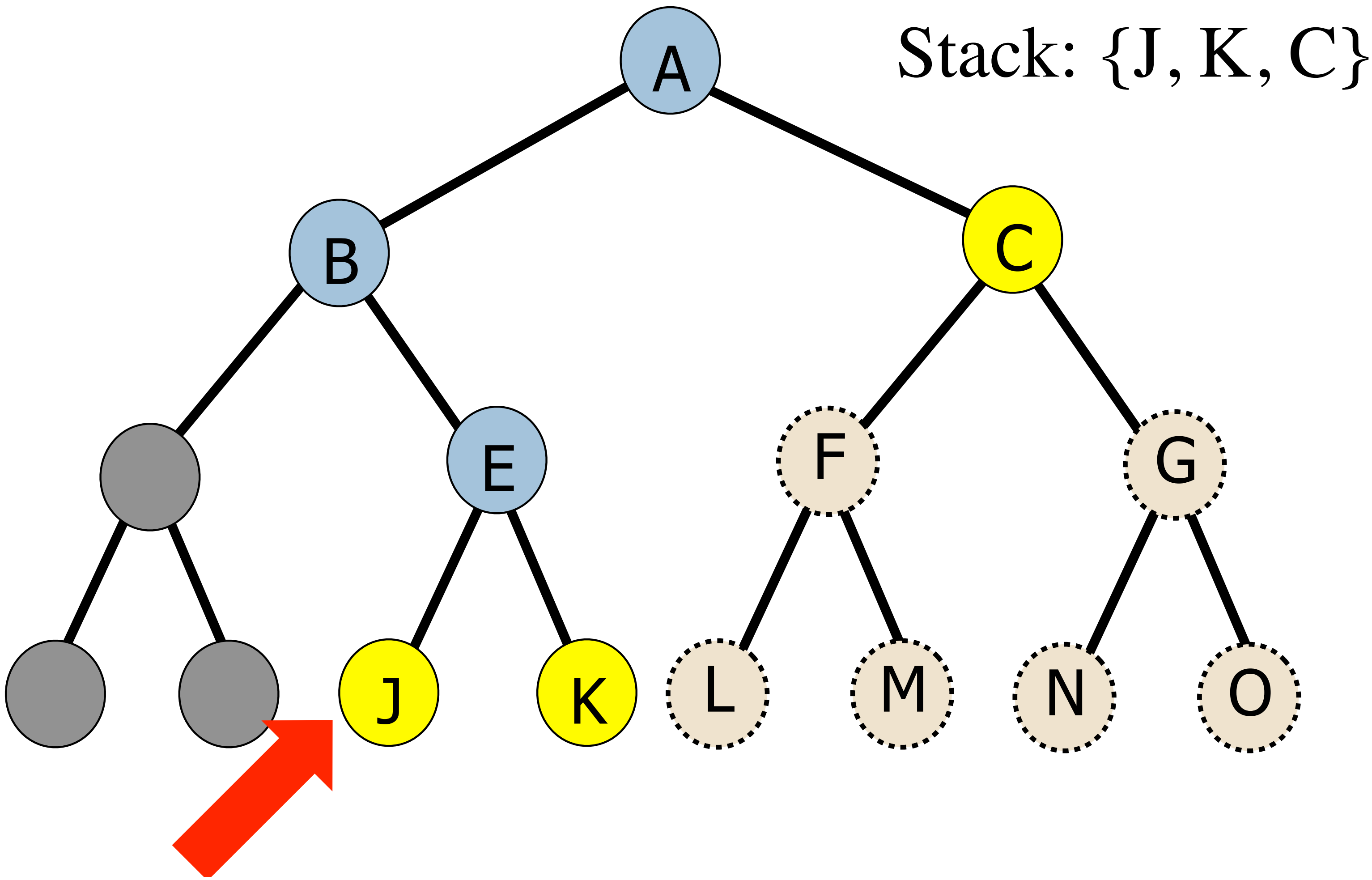
Depth-First Search

- Idea: Expand deepest unexpanded node



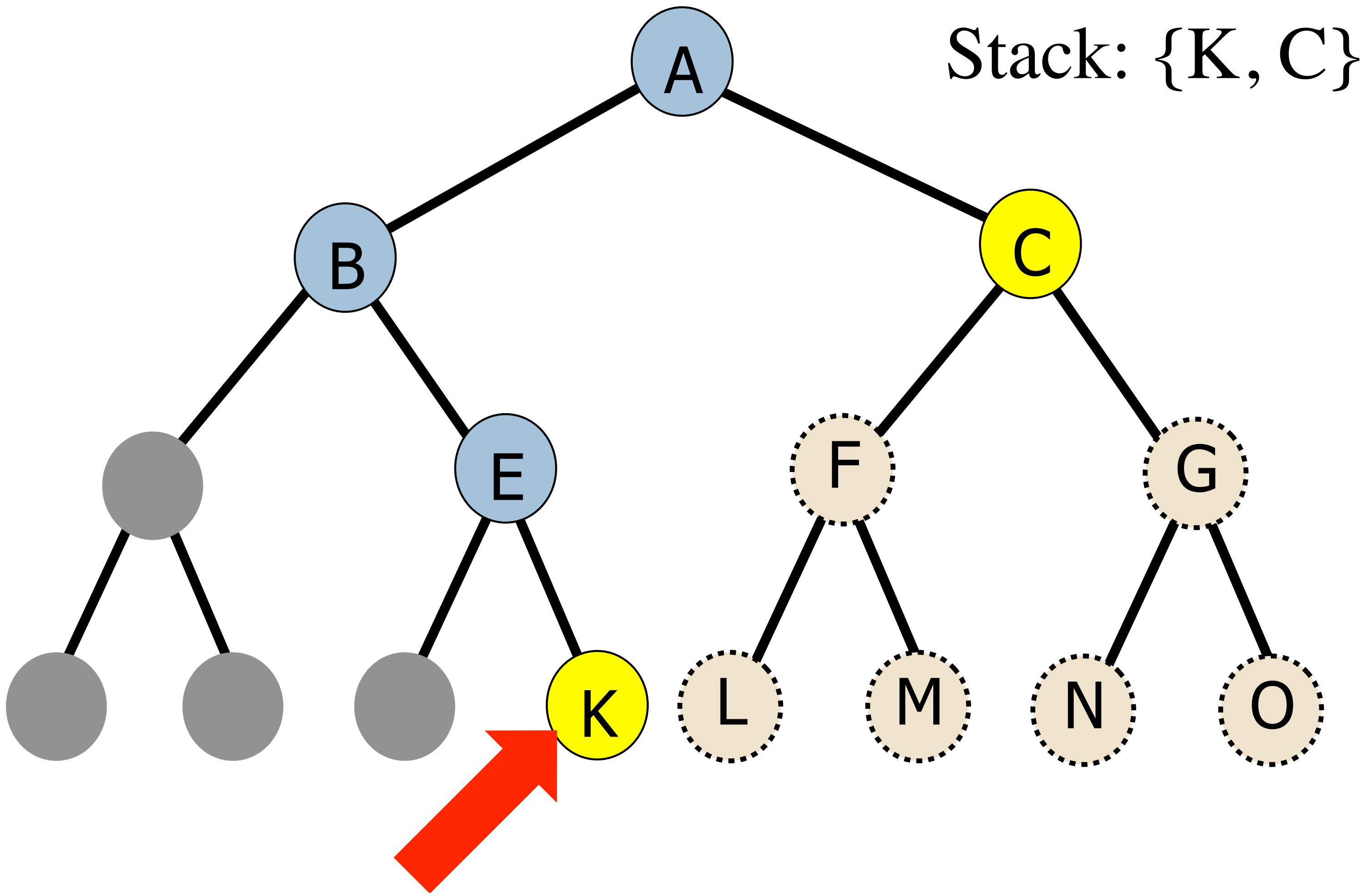
Depth-First Search

- Idea: Expand deepest unexpanded node



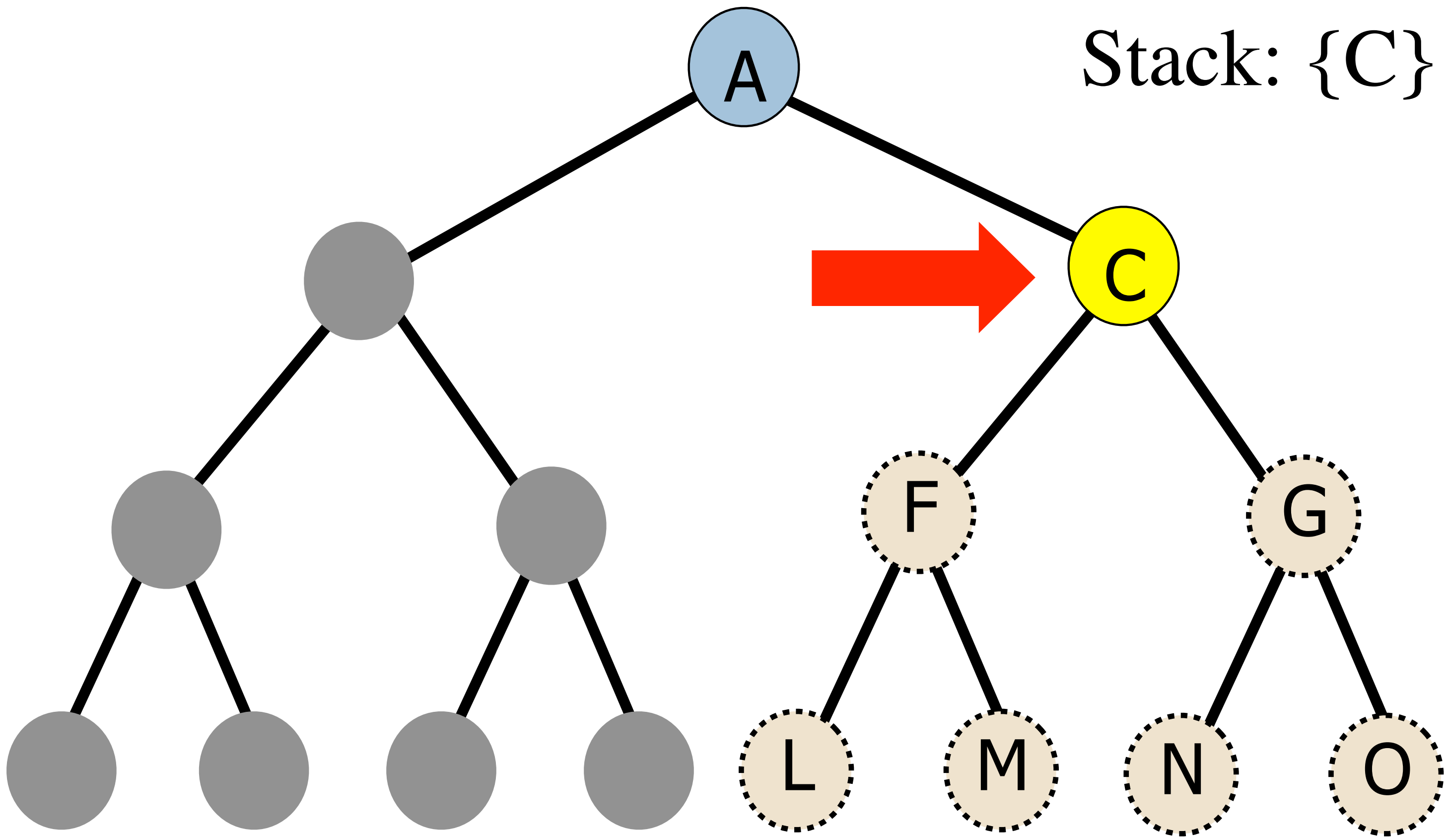
Depth-First Search

- Idea: Expand deepest unexpanded node



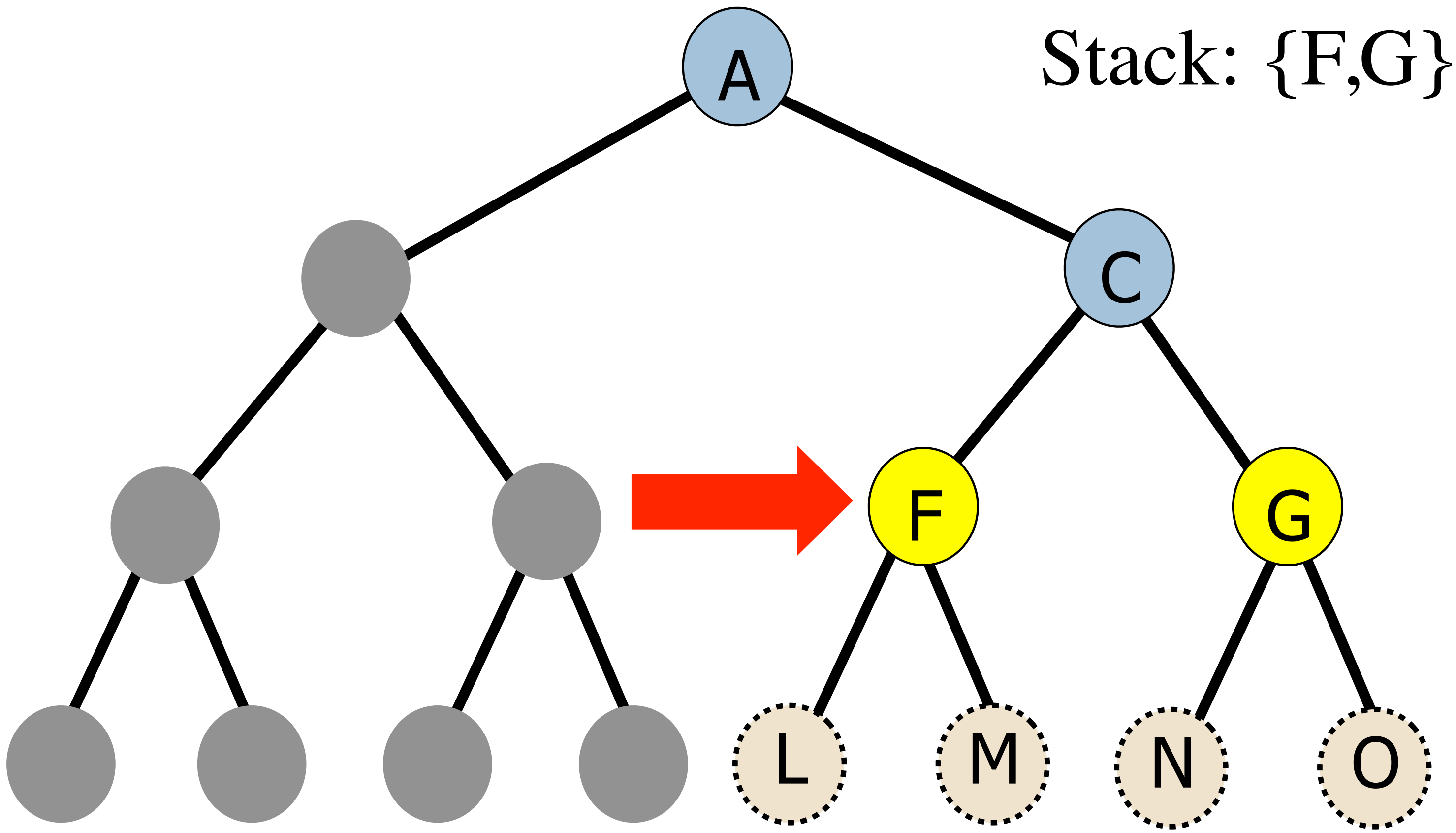
Depth-First Search

- Idea: Expand deepest unexpanded node



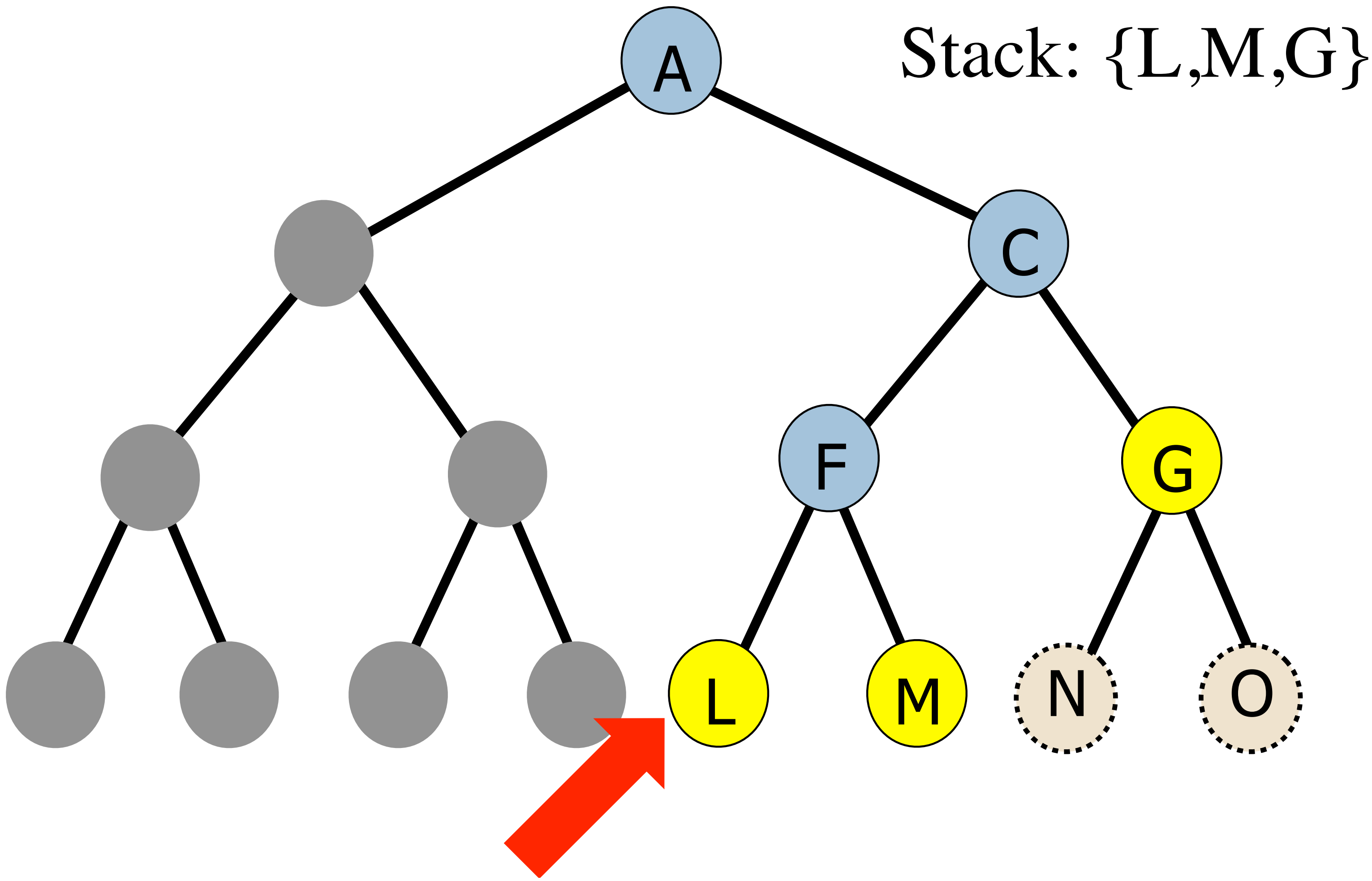
Depth-First Search

- Idea: Expand deepest unexpanded node



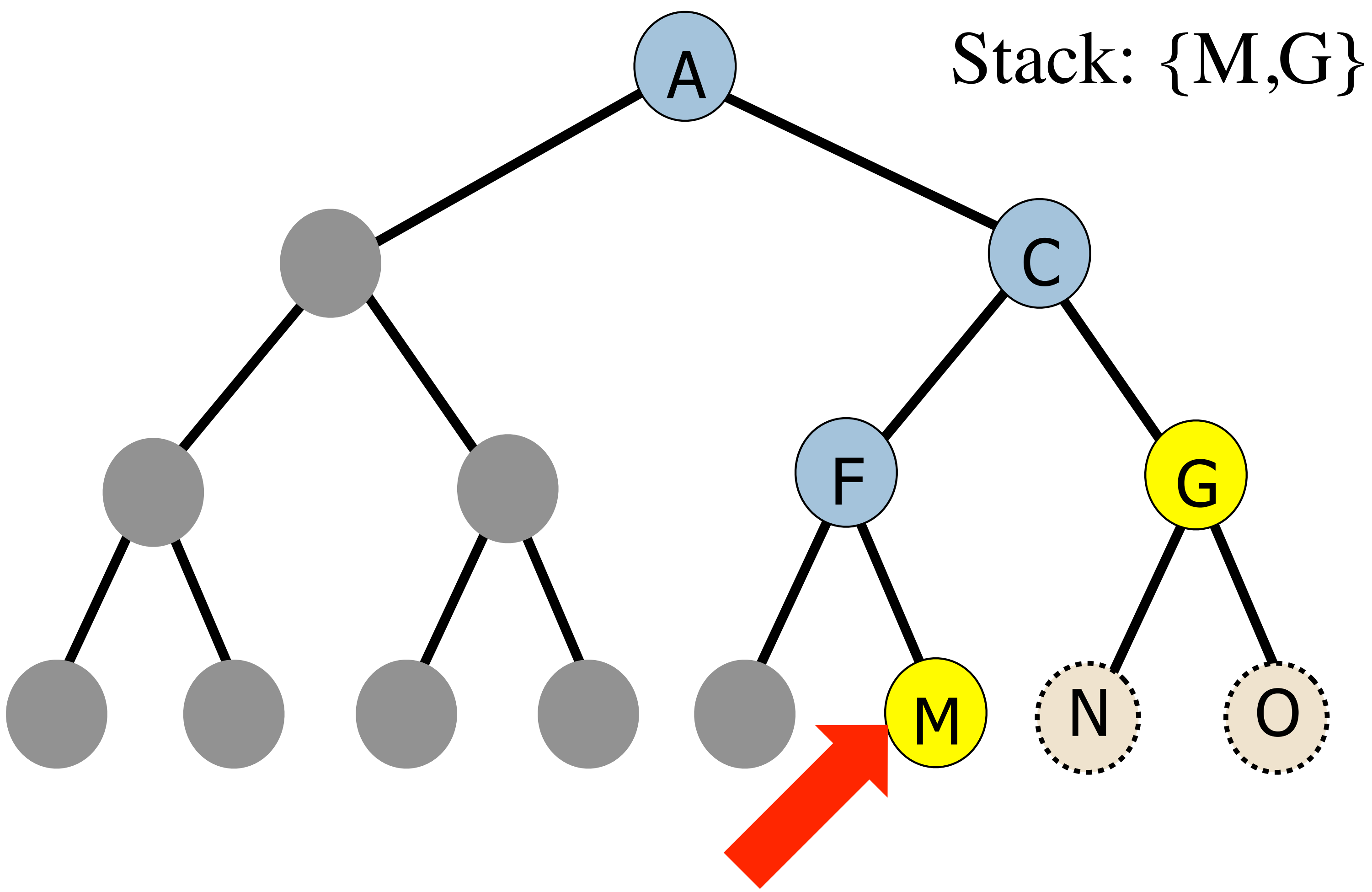
Depth-First Search

- Idea: Expand deepest unexpanded node



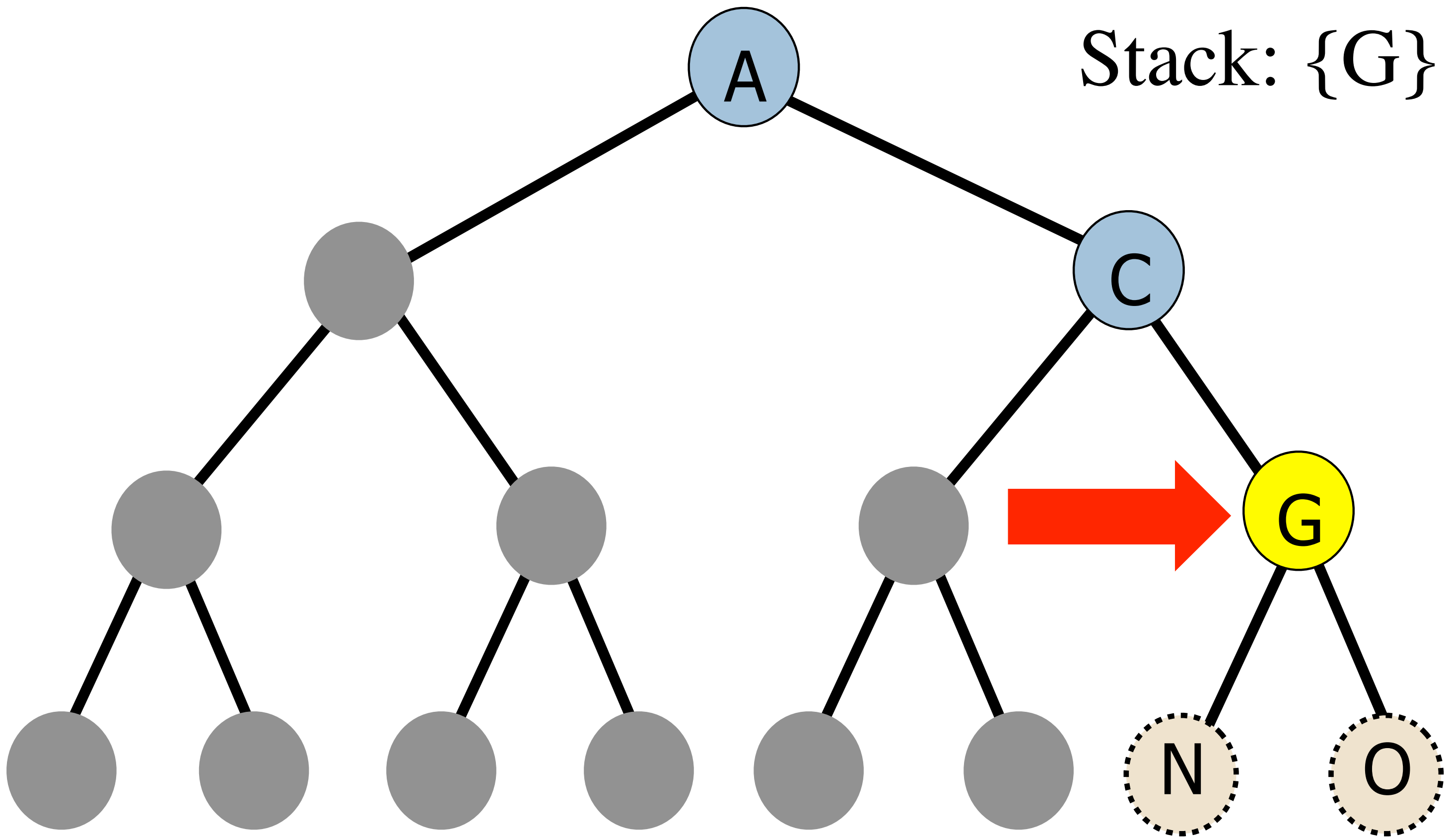
Depth-First Search

- Idea: Expand deepest unexpanded node



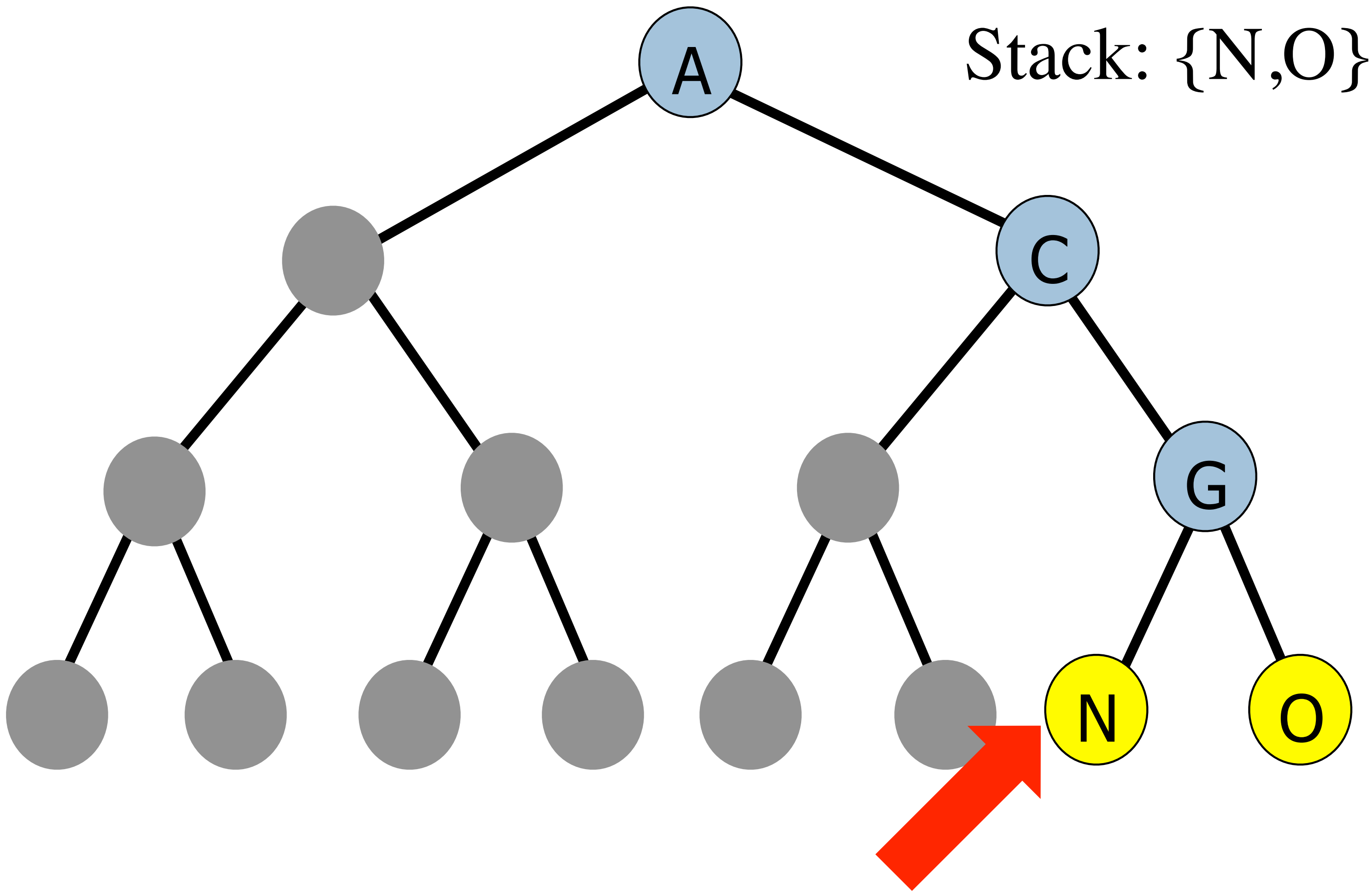
Depth-First Search

- Idea: Expand deepest unexpanded node



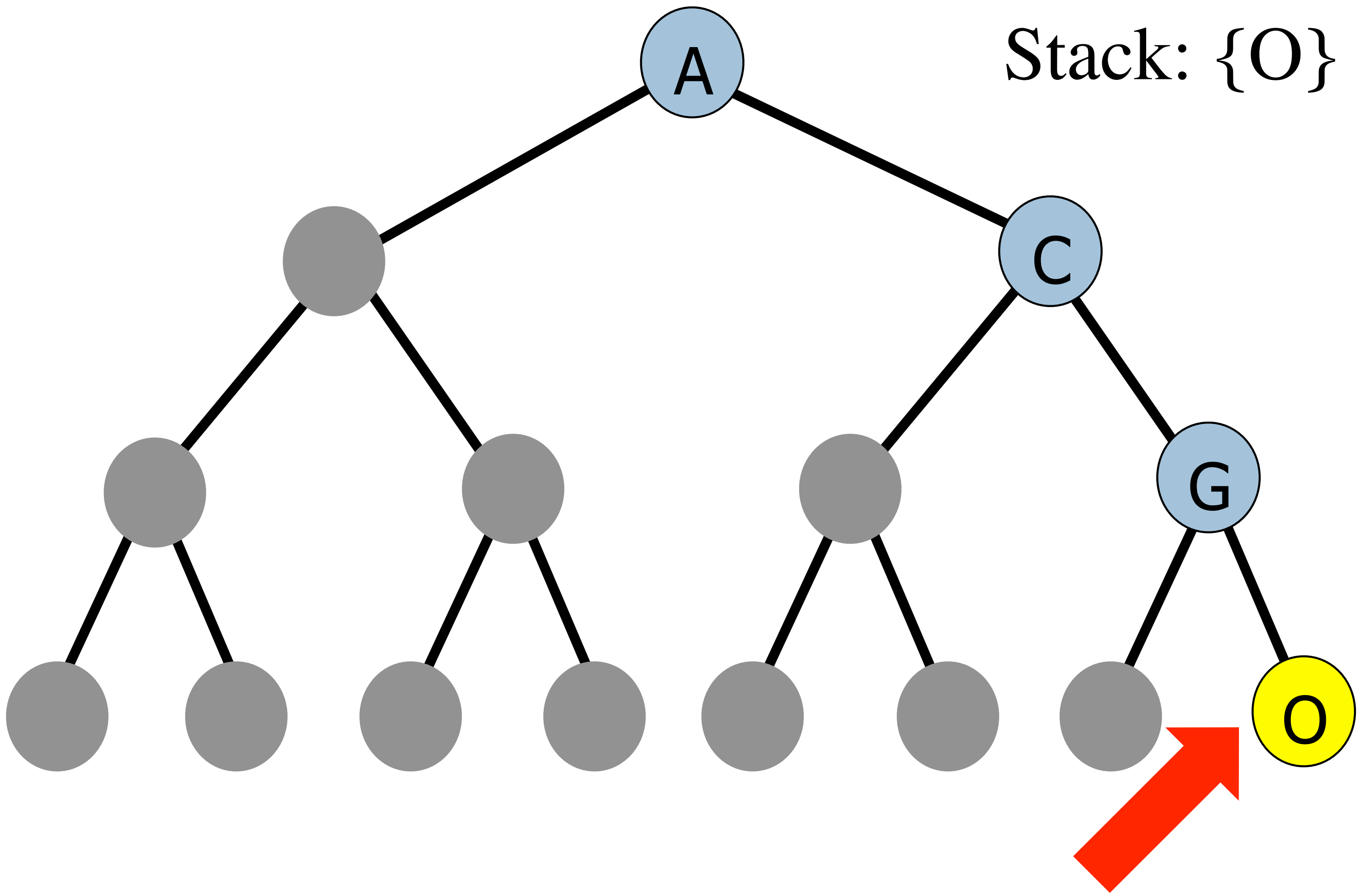
Depth-First Search

- Idea: Expand deepest unexpanded node



Depth-First Search

- Idea: Expand deepest unexpanded node



Properties of Depth-First Search

- Complete?

  No: fails in infinite-depth search tree.
  Can be modified to avoid repeated states along path → complete in finite state space

- Optimal?

  No.

- Time?

  $O(b^m)$: terrible if m is much larger than d.
  But, if solutions are dense, it may be much faster than BFS.

- Space?

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

Depth-Limited Search (DLS)

- Idea: Depth-first search with depth limit *l*, i.e., nodes at depth *l* have no successors
- Recursive implementation:

```
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
```

Iterative Deepening Search (IDS)

- Idea: Performs DLSs with increasing depth limit until goal node is found
- IDS is preferred if state space is large and depth of solution is unknown
- Implementation:

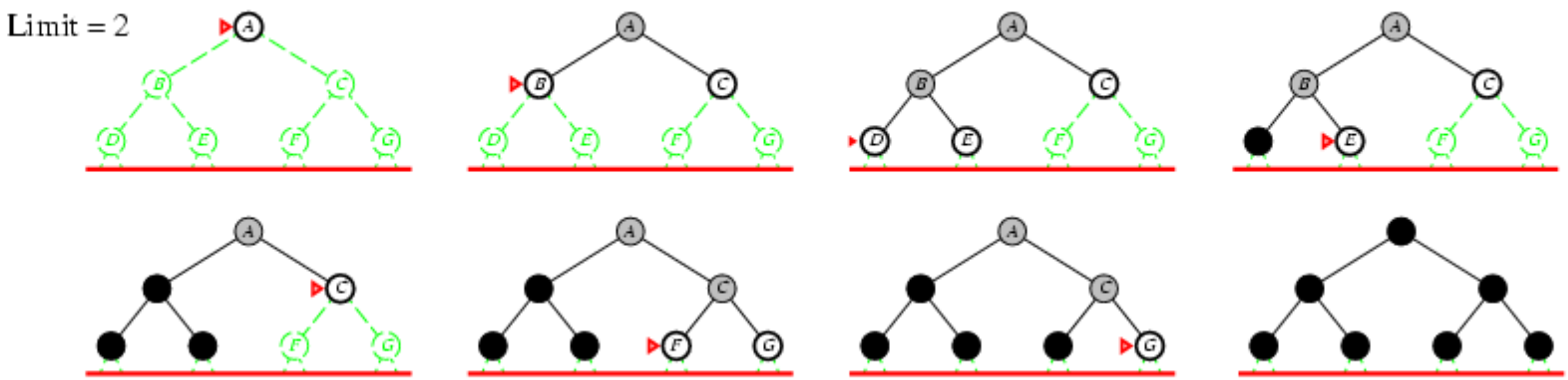
function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution, or failure
  for $depth = 0$ **to** ∞ **do**
    $result \leftarrow$ DEPTH-LIMITED-SEARCH(*problem*, *depth*)
    if $result \neq$ cutoff **then return** *result*

Iterative Deepening Search ($l = 0$)

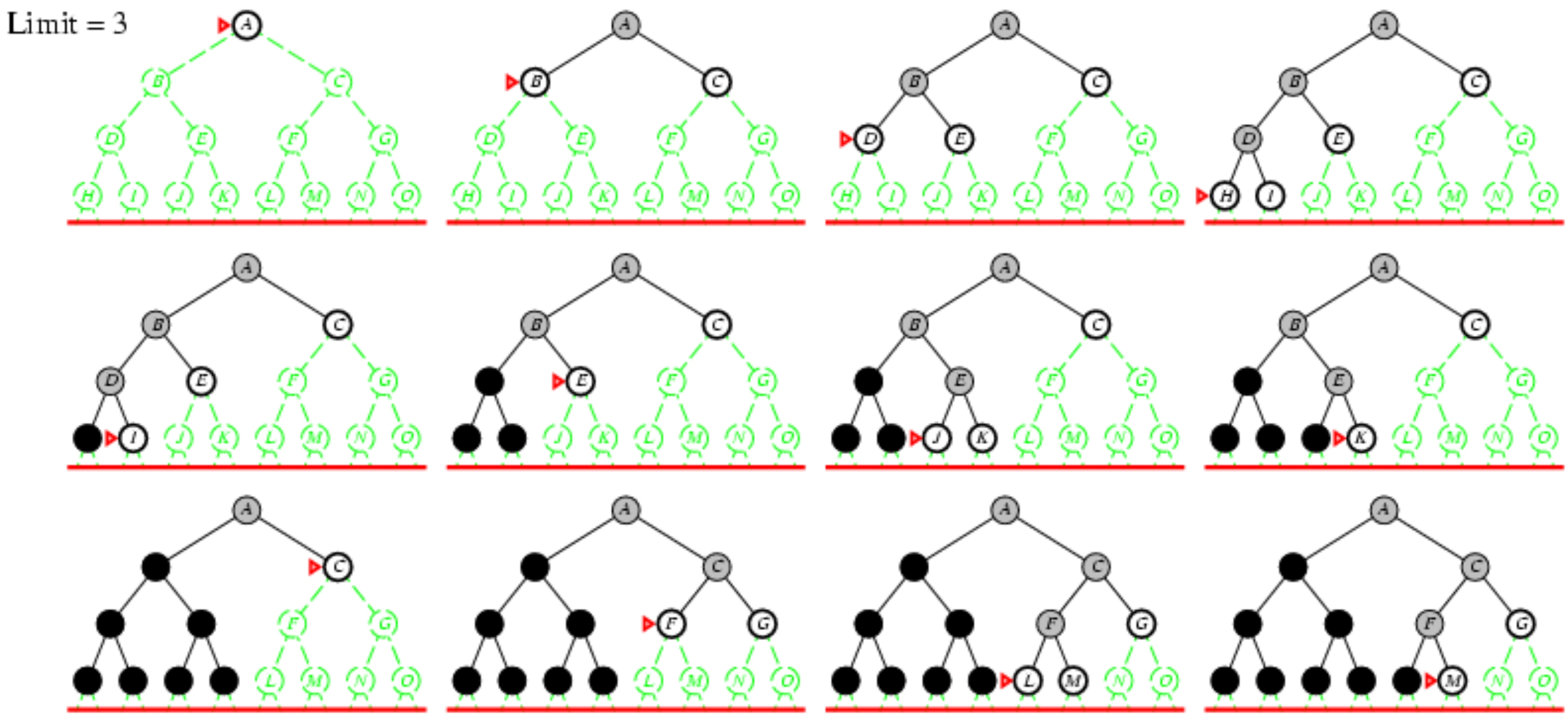
Limit = 0

Iterative Deepening Search ($l = 1$)

Iterative Deepening Search ($l = 2$)



Iterative Deepening Search ($l = 3$)



Iterative Deepening Search (IDS)

- Number of nodes generated in DLS (or BFS) to depth d with branching factor b:

$$N_{DLS} = b^0 + b^1 + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d$$

- Number of nodes generated in IDS to depth d with branching factor b:

$$N_{IDS} = (d+1)b^0 + d\, b^1 + (d-1)b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d$$

- For $b = 10$, $d = 5$,
  - $N_{DLS} = 1 + 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}111$
  - $N_{IDS} = 6 + 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}456$
- Overhead = (123,456 - 111,111)/111,111 = 11%

Properties of Iterative Deepening Search

- Complete? Yes (if b is finite)
- Optimal? Yes (if step cost = 1)
- Time? $(d+1)b^0+db^1+(d-1)b^2+\ldots+b^d = O(b^d)$
- Space? $O(bd)$

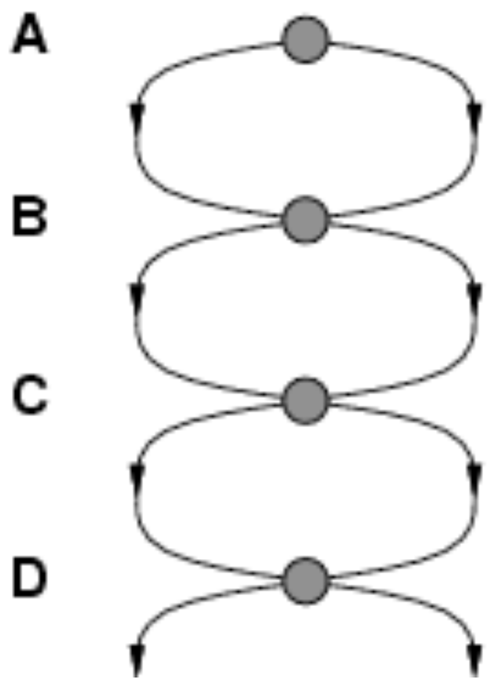
Comparing Tree Search Strategies

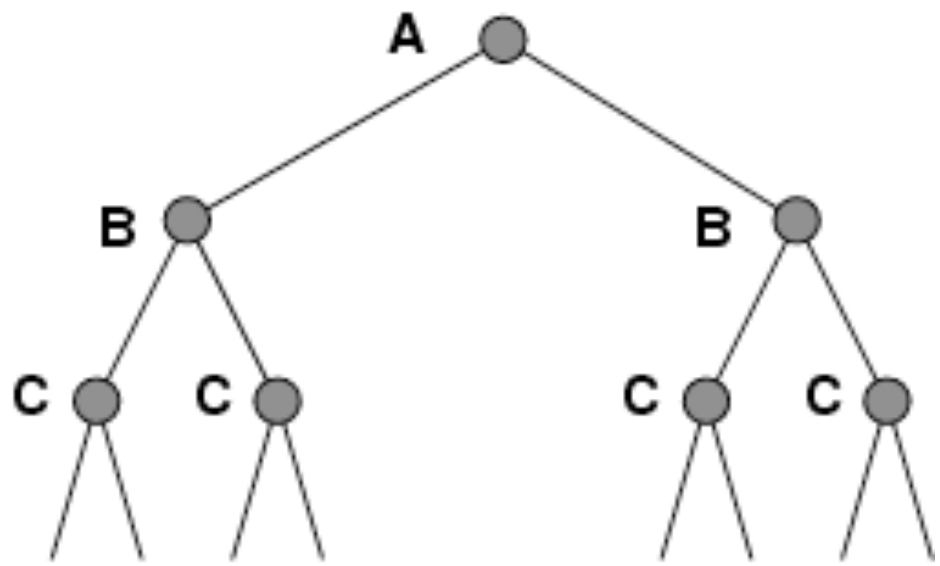
Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes	No	No	Yes
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes	Yes	No	No	Yes

1. BFS and IDS are complete if b is finite.
2. UCS is complete if b is finite and step cost $\geq \varepsilon$
3. BFS and IDS are optimal if step costs are identical.

Repeated States

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





Dealing with Repeated States

- How do we avoid repeats/loops?

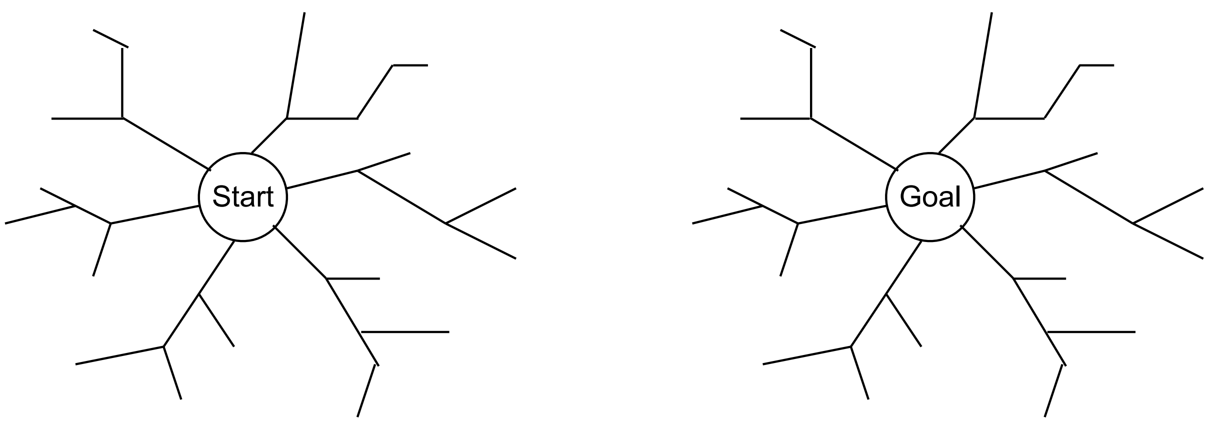
  Remember the nodes that are already visited!

Graph Search Algorithms

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

Bidirectional Search

- Simultaneously search both forward (from the initial state) and backward (from the goal state)
- Goal test: frontier of the two searches meet
- Intuition: 2 x O($b^{d/2}$) is smaller than O(b^d)



Bidirectional Search

- Numerical Example ($b = 10$, $d = 6$)
  - Bi-directional search finds solution at $d = 3$ for both forward and backward search. Assuming BFS in each half, 2222 nodes are generated (as compared to 1,111,110 for full BFS).
- Implementation issues:
  - Actions are reversible: predecessor = successor
  - There may be many possible goal states
    - Construct a dummy goal state whose immediate predecessors are all the actual goal states
  - Check if a node appears in the "other" search tree
  - Using different search strategies for each half.