T2: Uninformed Search

Objectives

- To explain various informed and uninformed search strategies

Outcomes

- Solve problem using search strategies to go next state

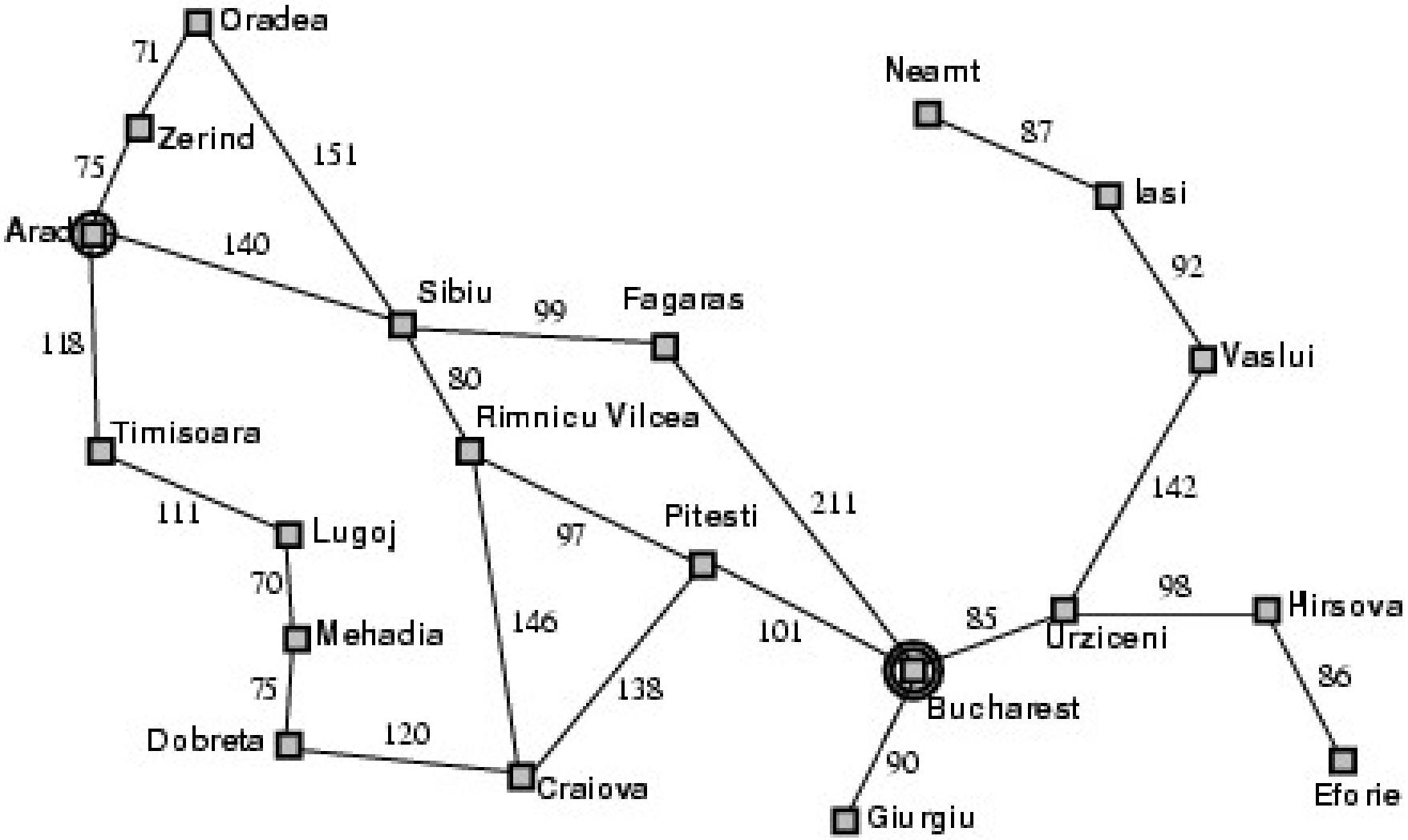
Tree search algorithms

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

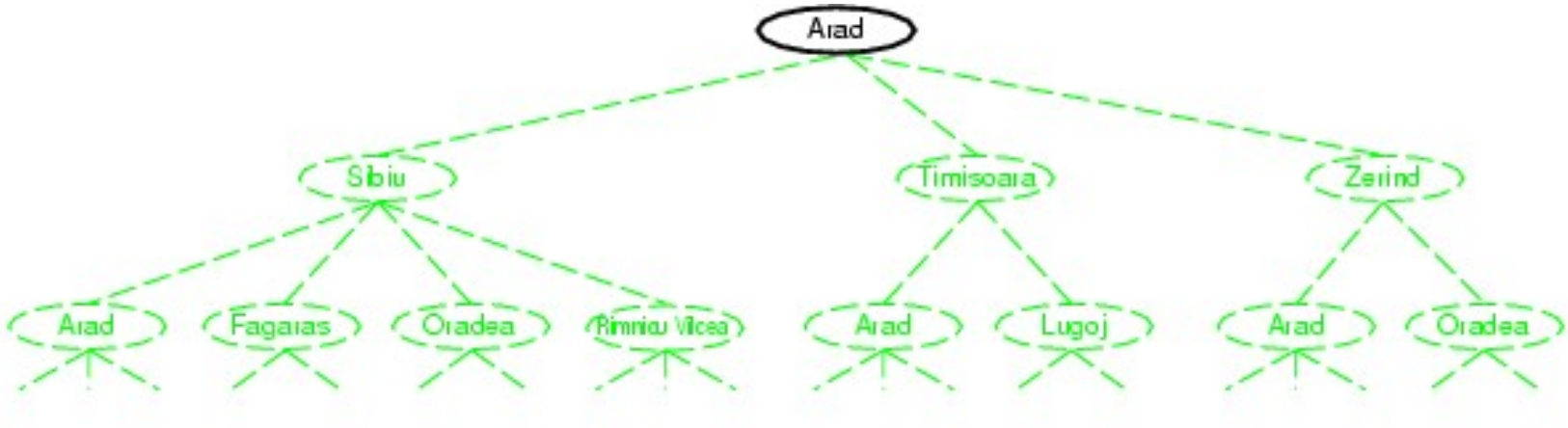
```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
```

Ack.: https://www.britannica.com/place/Romania

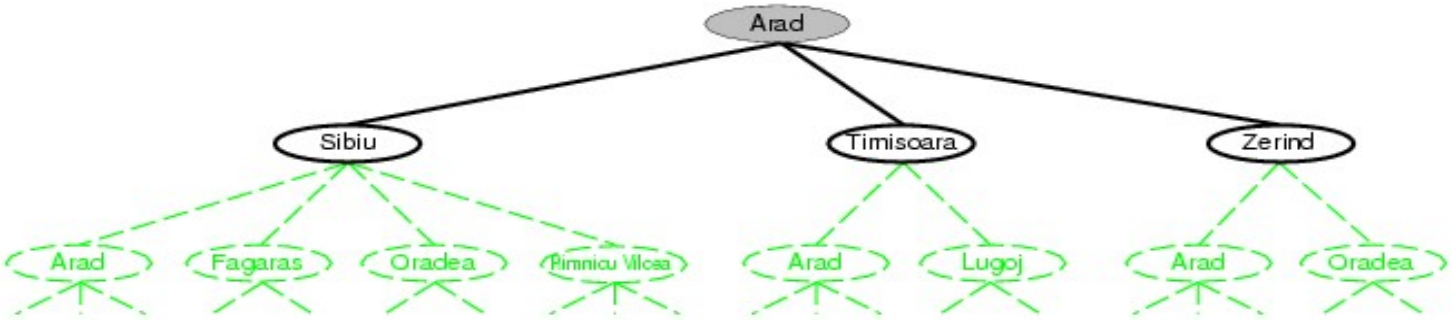
Example: Romania



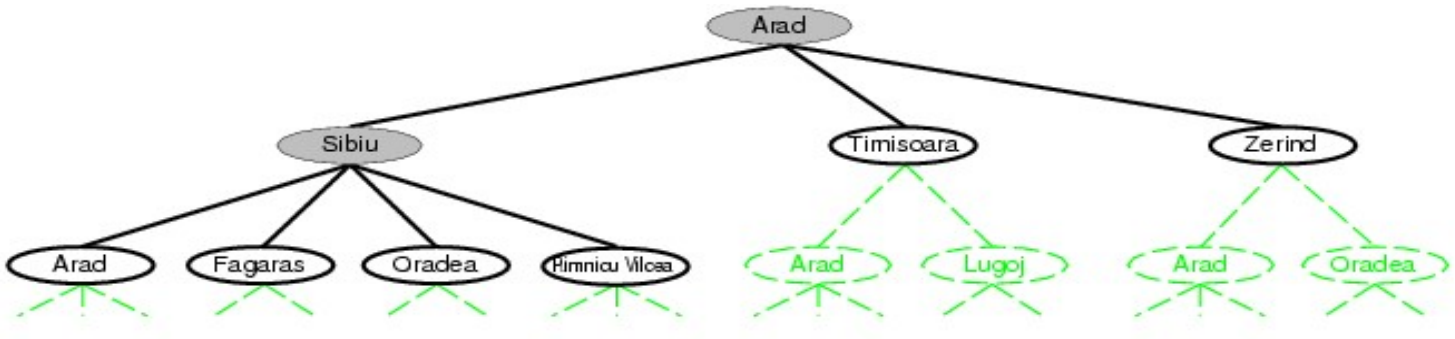
Tree search example



Tree search example



Tree search example



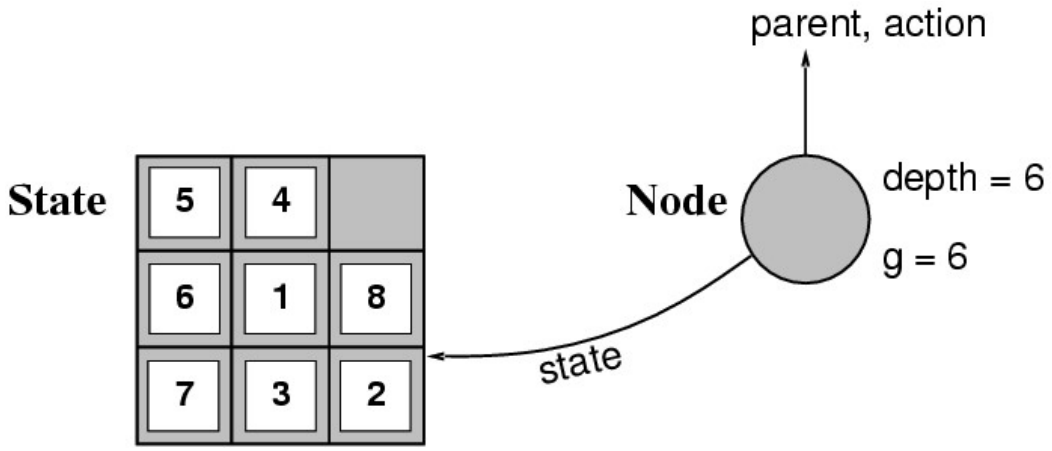
Implementation: General tree search

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

```
function EXPAND( node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
        s ← a new NODE
        PARENT-NODE[s] ← node;  ACTION[s] ← action;  STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
```

Implementation: states vs. nodes

- A state is a (representation of) a physical configuration
- A node is a data structure constituting part of a search tree includes state, parent node, action, path cost *g(x)*, depth



- The `Expand` function creates new nodes, filling in the various fields and using the **`SuccessorFn`** of the problem to create the corresponding states.

Search strategies

- A search strategy is defined by picking the order of node expansion
- Strategies are evaluated along the following dimensions:
  - **completeness**: does it always find a solution if one exists?
  - **time complexity**: number of nodes generated
  - **space complexity:** maximum number of nodes in memory
  - **optimality**: does it always find a least-cost solution?

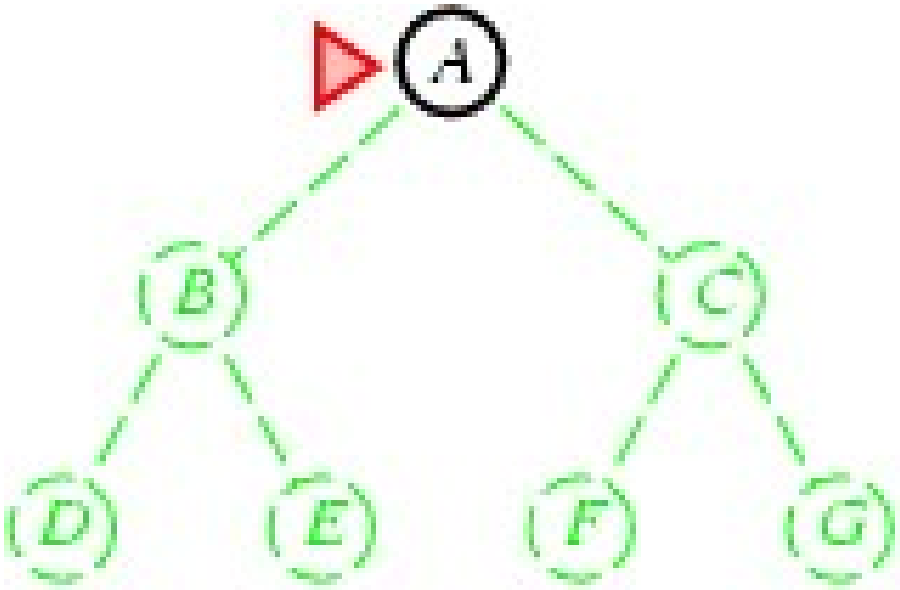
- Time and space complexity are measured in terms of
  - *b:* maximum branching factor of the search tree
  - *d:* depth of the least-cost solution
  - *m*: maximum depth of the state space (may be ∞)

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

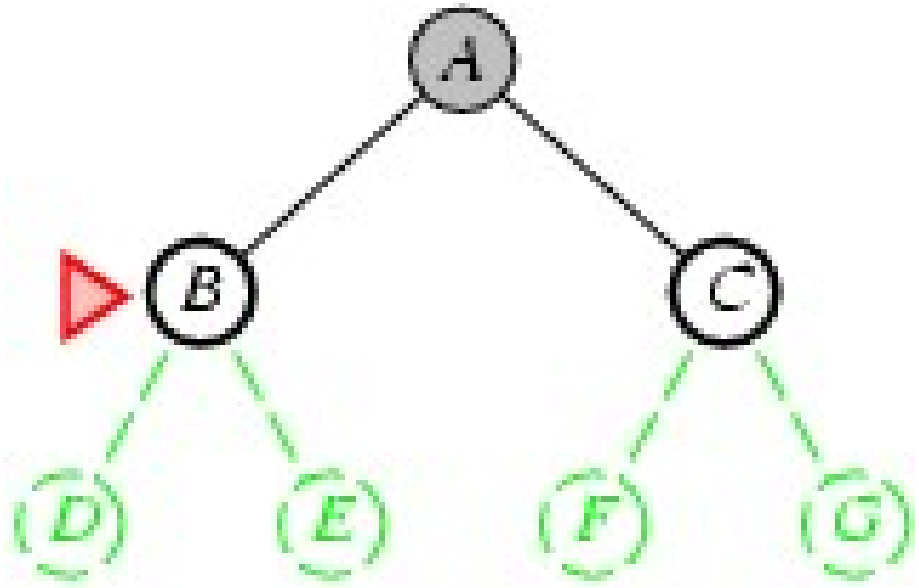
Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end



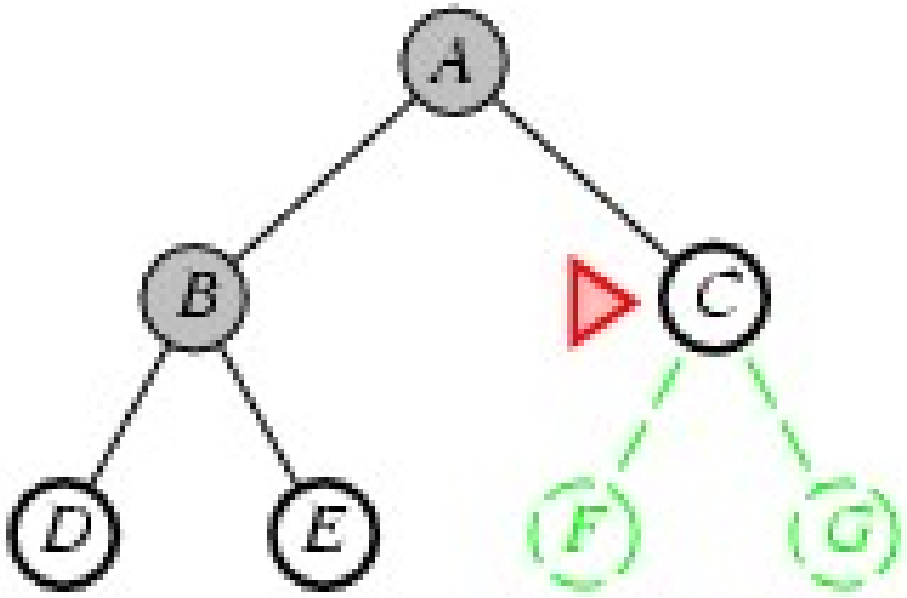
Breadth-first search

- Expand shallowest unexpanded node
- **Implementation**:
  - *fringe* is a FIFO queue, i.e., new successors go at end



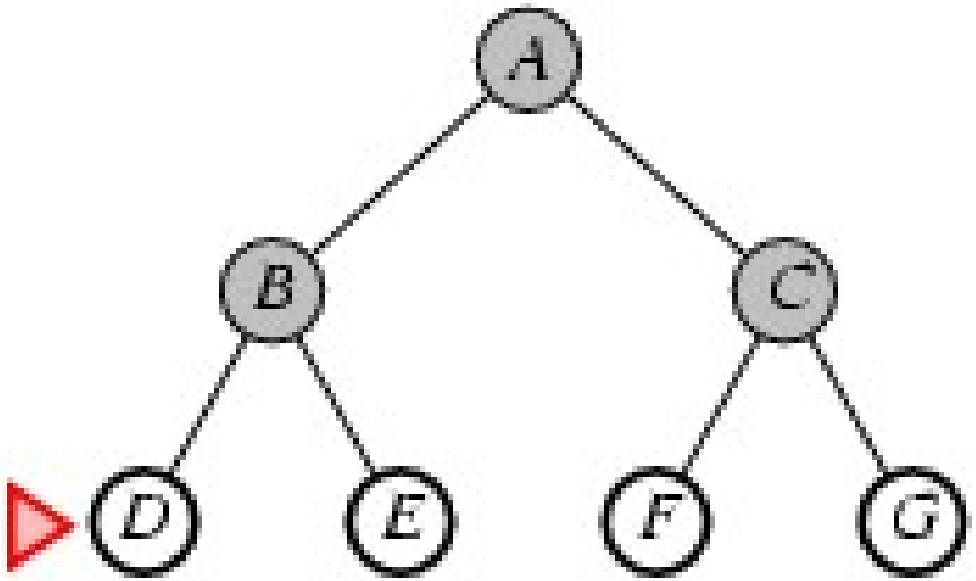
Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end



Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end



Properties of breadth-first search

- Complete? Yes (if *b* is finite) b is the number of successors
- Time? $1+b+b^2+b^3+\ldots+b^d + b(b^d-1) = O(b^{d+1})$
- Space? $O(b^{d+1})$ (keeps every node in memory)
- Optimal? Yes (if cost = 1 per step)

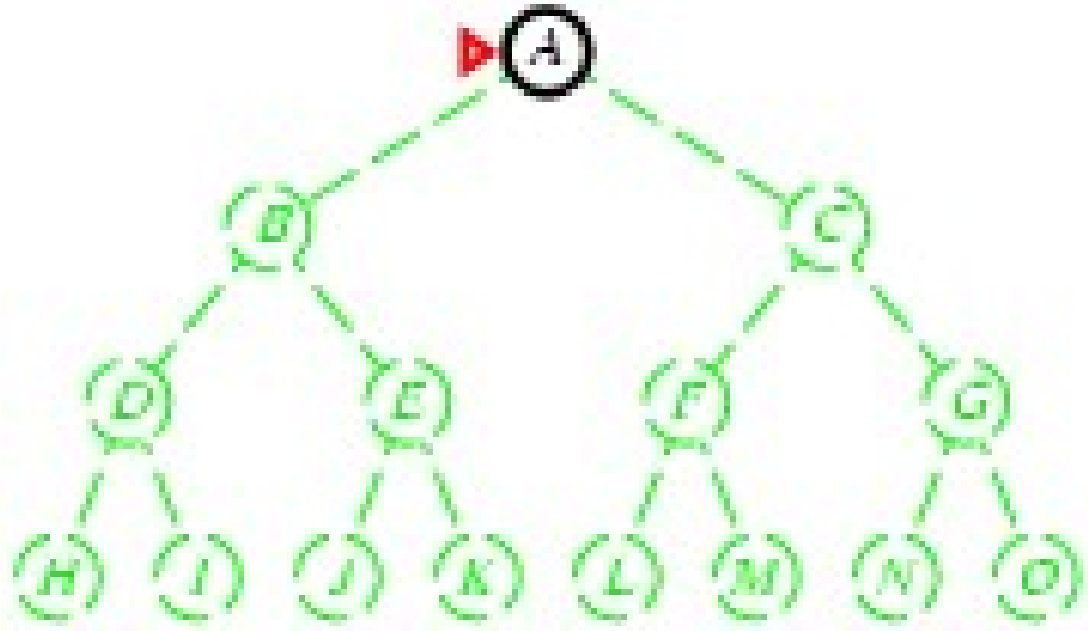
Space is the bigger problem (more than time)

Uniform-cost search

- The primary goal of the uniform-cost search is to find a path to the goal node which has the lowest cumulative cost. This algorithm comes into play when a different cost is available for each edge.
- Expand least-cost unexpanded node
- **Implementation**:
  - ***fringe* = queue ordered by path cost**
- Equivalent to breadth-first if step costs all equal
- Complete? Yes, if step cost ≥ ε
- Time? # of nodes with $g \leq$ cost of optimal solution, $O(b^{ceiling(C^*/\varepsilon)})$ where C^* is the cost of the optimal solution
- Space? # of nodes with $g \leq$ cost of optimal solution, $O(b^{ceiling(C^*/\varepsilon)})$
- Optimal? Yes – nodes expanded in increasing order of $g(n)$
- A uniform-cost search algorithm is implemented by the priority queue. It gives maximum priority to the lowest cumulative cost

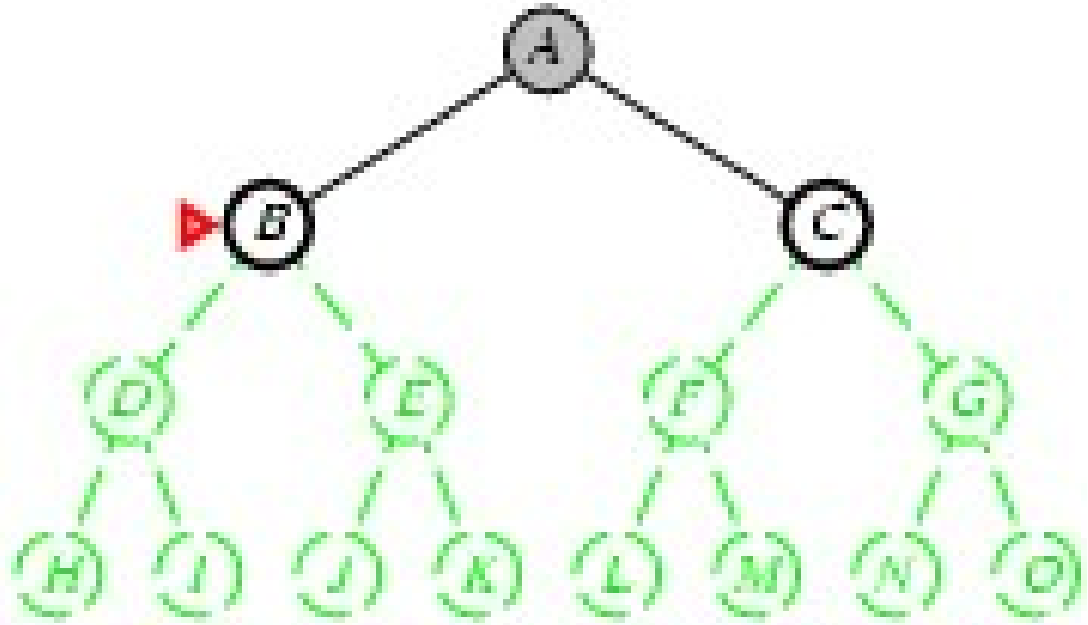
Depth-first search

- Expand deepest unexpanded node
- **Implementation**:
  - *fringe* = LIFO data, i.e., put successors at front



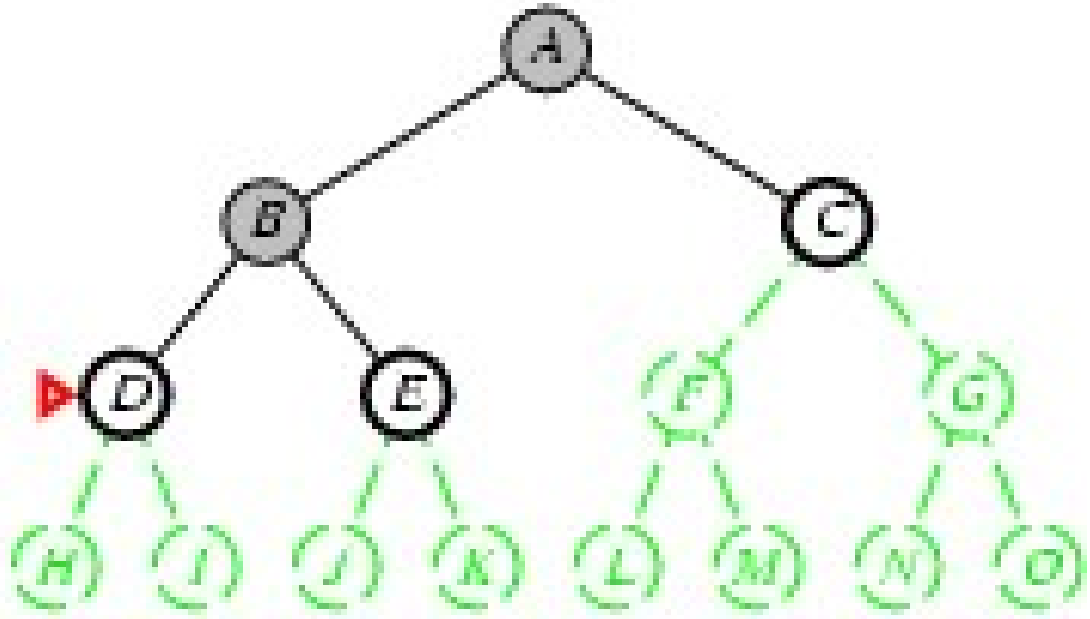
Depth-first search

- Expand deepest unexpanded node
- **Implementation**:
  - *fringe* = LIFO queue, i.e., put successors at front



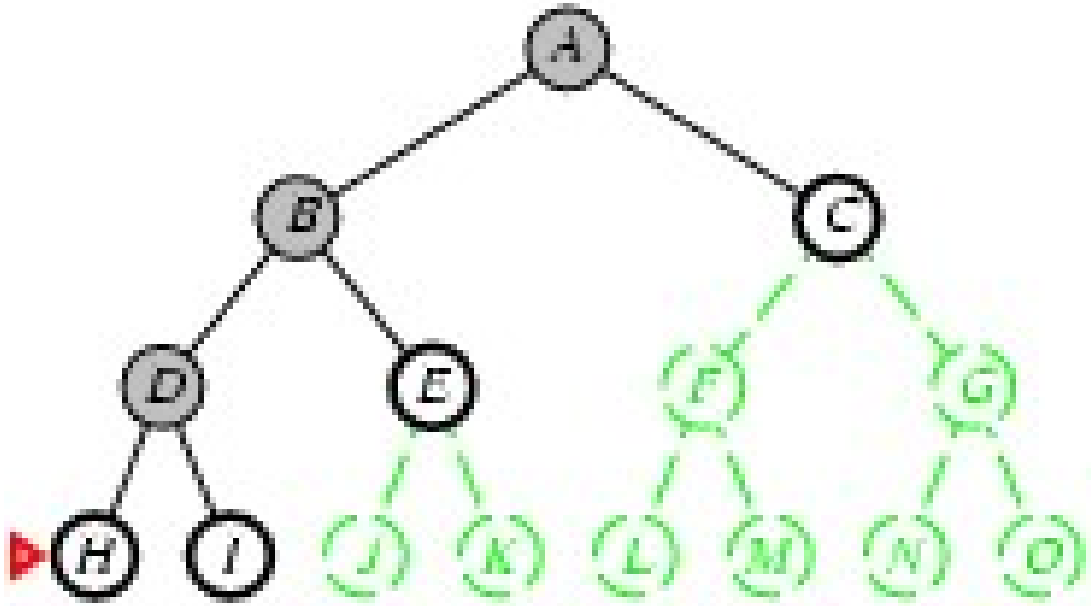
Depth-first search

- Expand deepest unexpanded node
- **Implementation**:
  - *fringe* = LIFO queue, i.e., put successors at front
  -



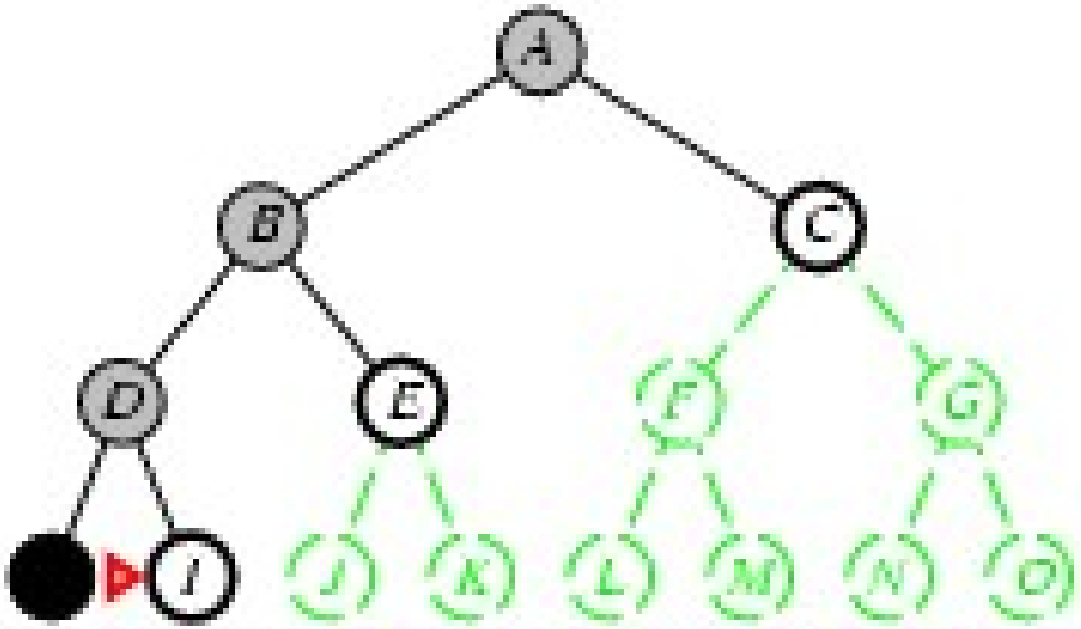
Depth-first search

- Expand deepest unexpanded node
- **Implementation**:
  - *fringe* = LIFO queue, i.e., put successors at front



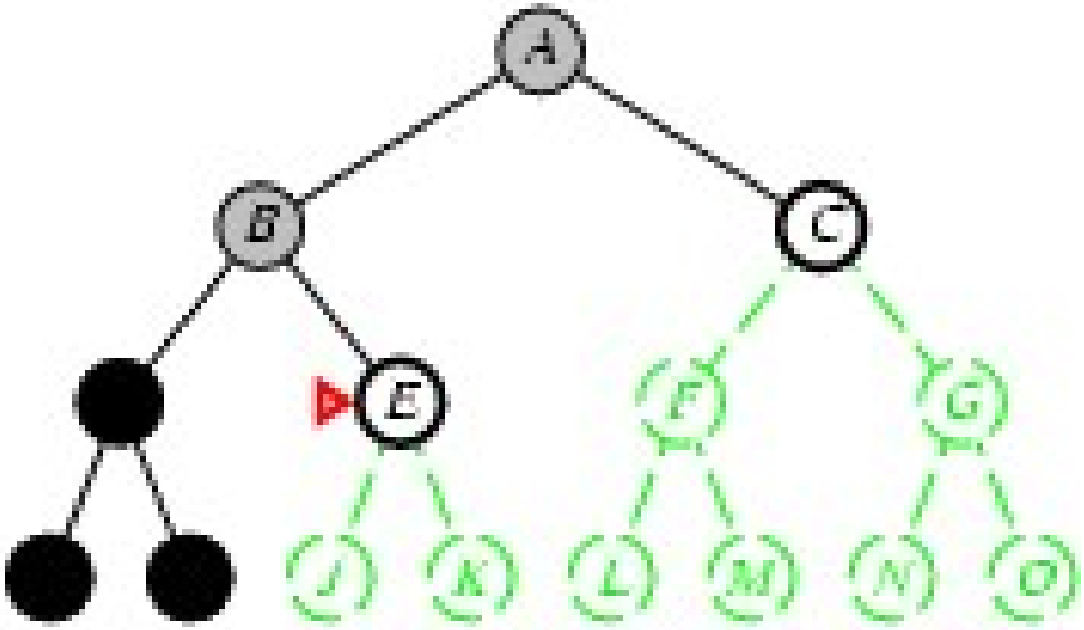
Depth-first search

- Expand deepest unexpanded node
- **Implementation**:
  - *fringe* = LIFO queue, i.e., put successors at front



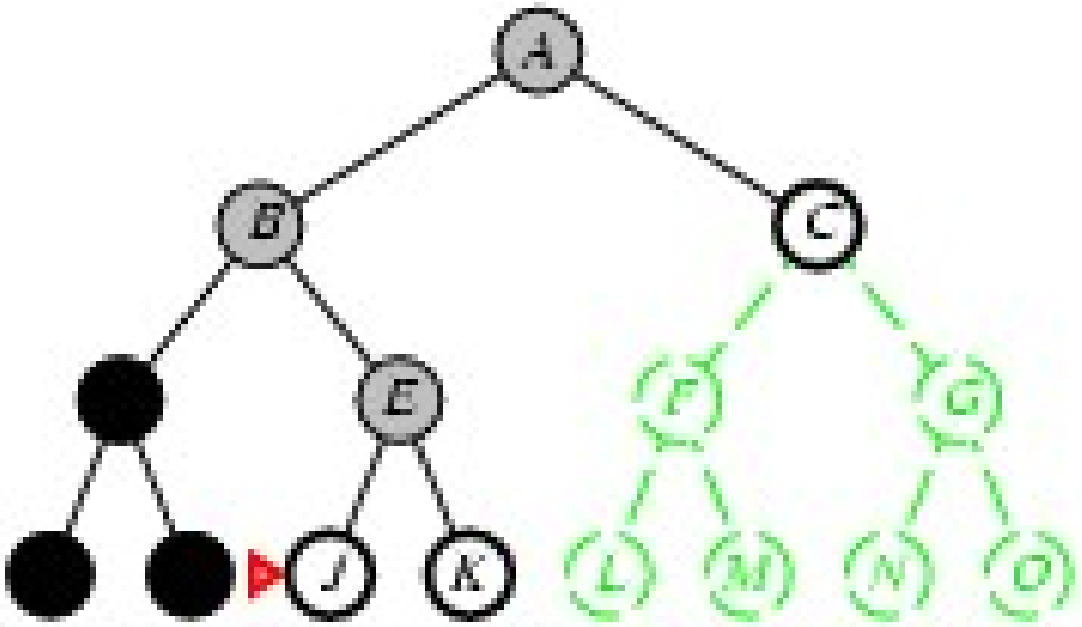
Depth-first search

- Expand deepest unexpanded node
- **Implementation**:
  - *fringe* = LIFO queue, i.e., put successors at front



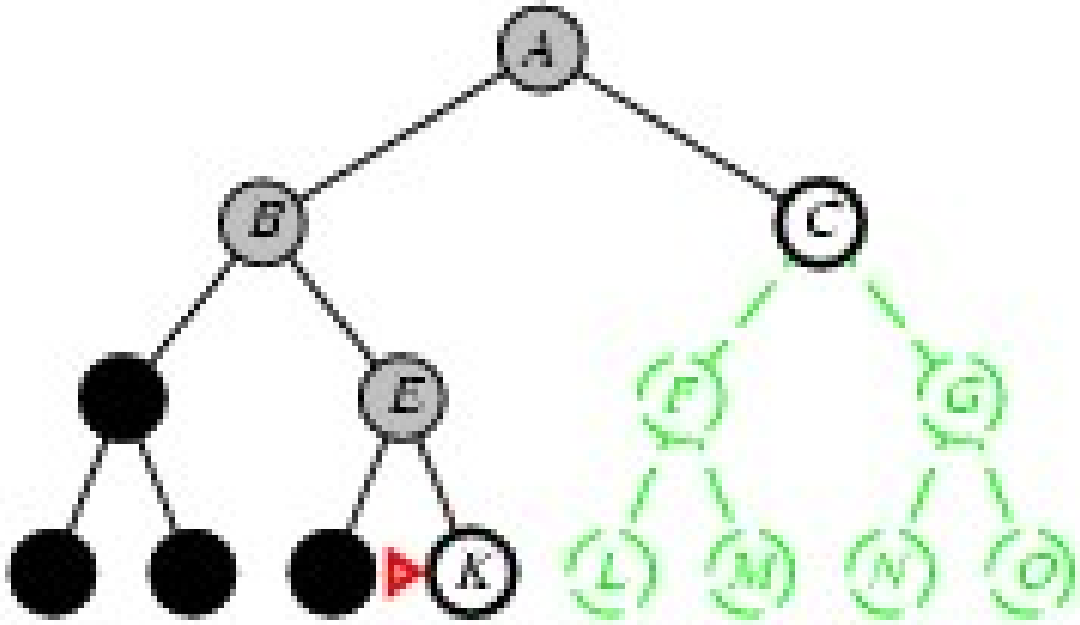
Depth-first search

- Expand deepest unexpanded node
- **Implementation**:
  - *fringe* = LIFO queue, i.e., put successors at front



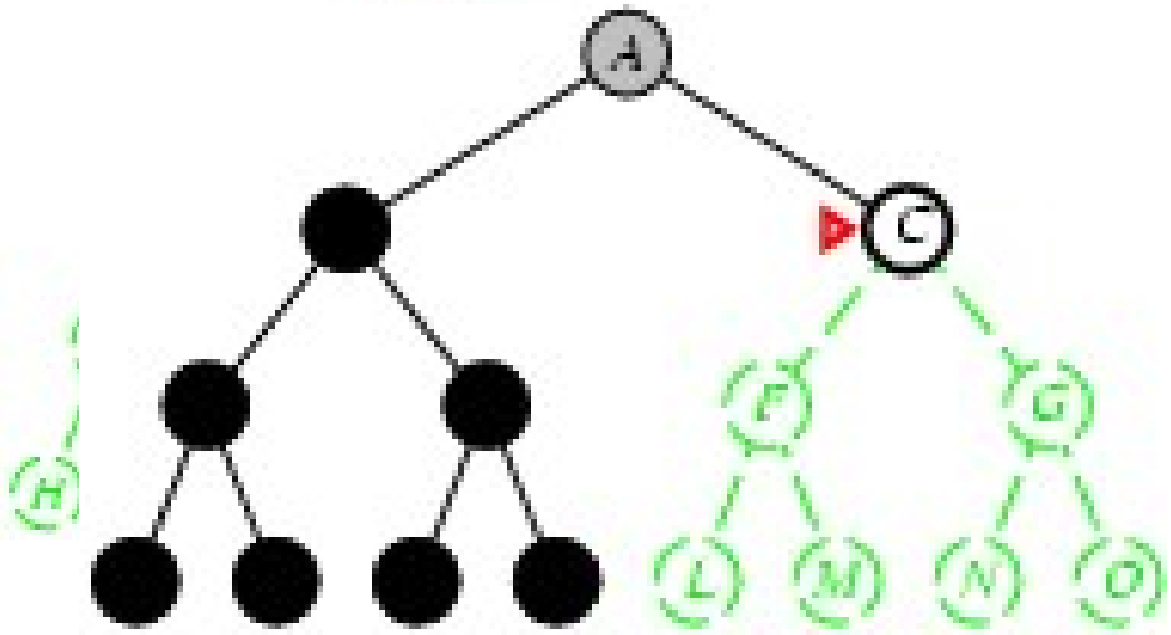
Depth-first search

- Expand deepest unexpanded node
- **Implementation**:
  - *fringe* = LIFO queue, i.e., put successors at front



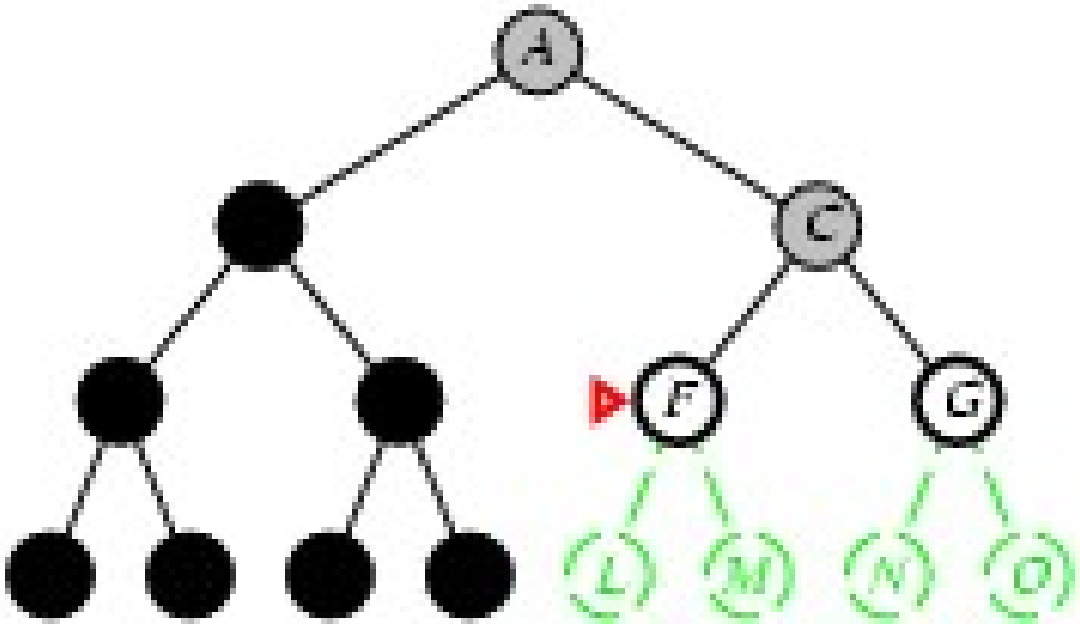
Depth-first search

- Expand deepest unexpanded node
- **Implementation**:
  - *fringe* = LIFO queue, i.e., put successors at front



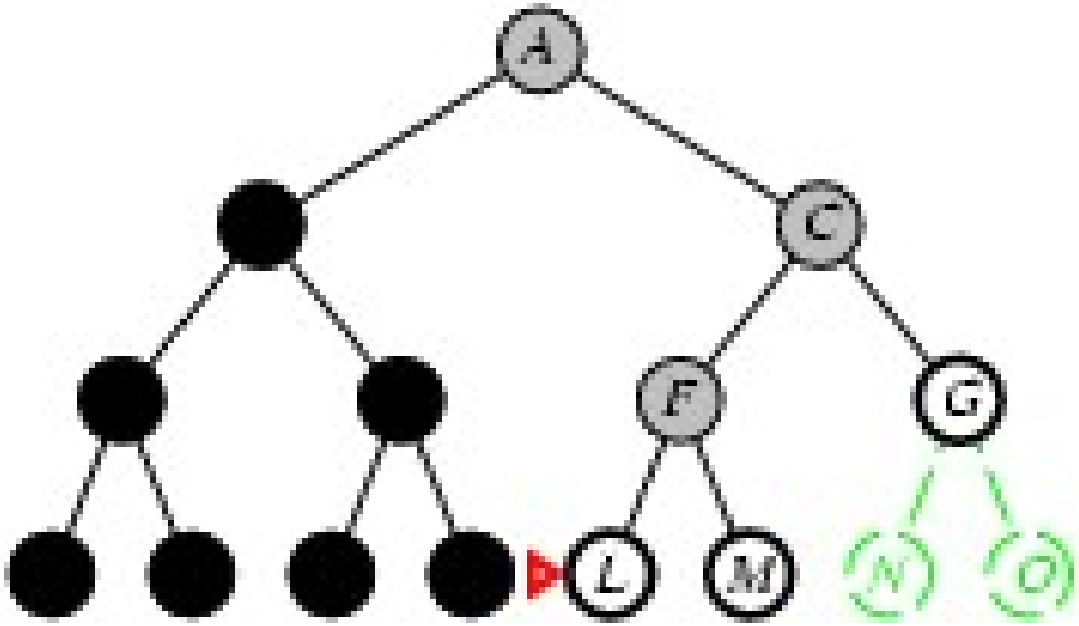
Depth-first search

- Expand deepest unexpanded node
- **Implementation**:
  - *fringe* = LIFO queue, i.e., put successors at front



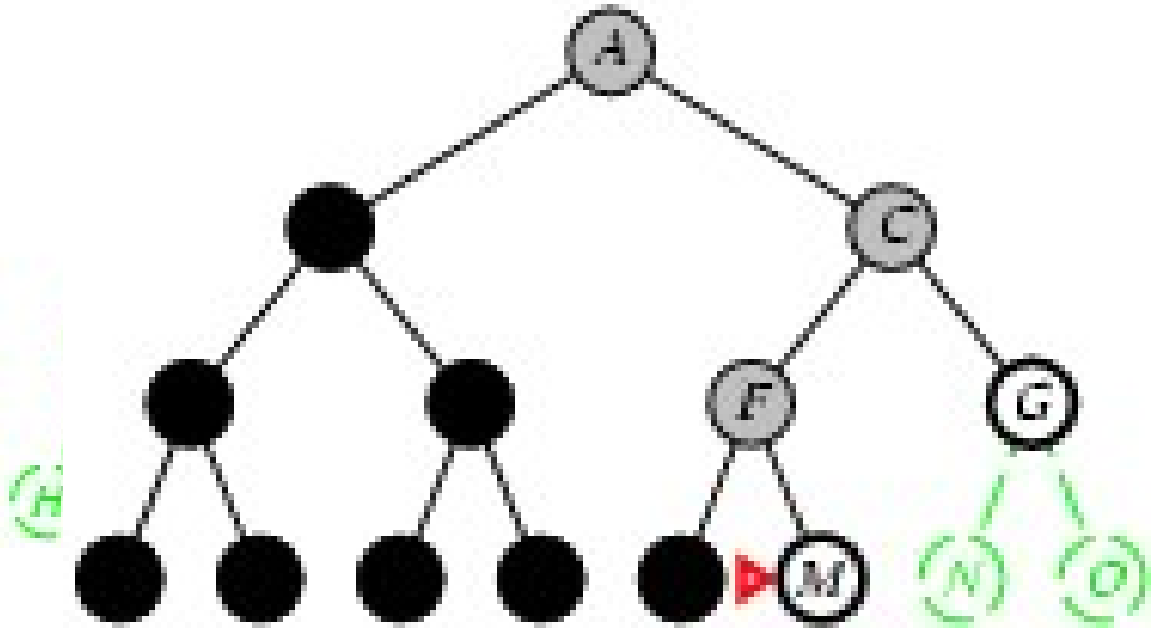
Depth-first search

- Expand deepest unexpanded node
- **Implementation**:
  - *fringe* = LIFO queue, i.e., put successors at front



Depth-first search

- Expand deepest unexpanded node
- **Implementation**:
  - *fringe* = LIFO queue, i.e., put successors at fron



Properties of depth-first search

- Complete? No: fails in infinite-depth spaces, spaces with loops
  - Modify to avoid repeated states along path
    → complete in finite spaces
- Time? $O(b^m)$: terrible if m is much larger than d
  - but if solutions are dense, may be much faster than breadth-first
- Space? $O(bm)$, i.e., linear space!
- Optimal? No

Depth-limited search

depth-first search with depth limit *l*,

i.e., nodes at depth *l* have no successors

Recursive implementation:

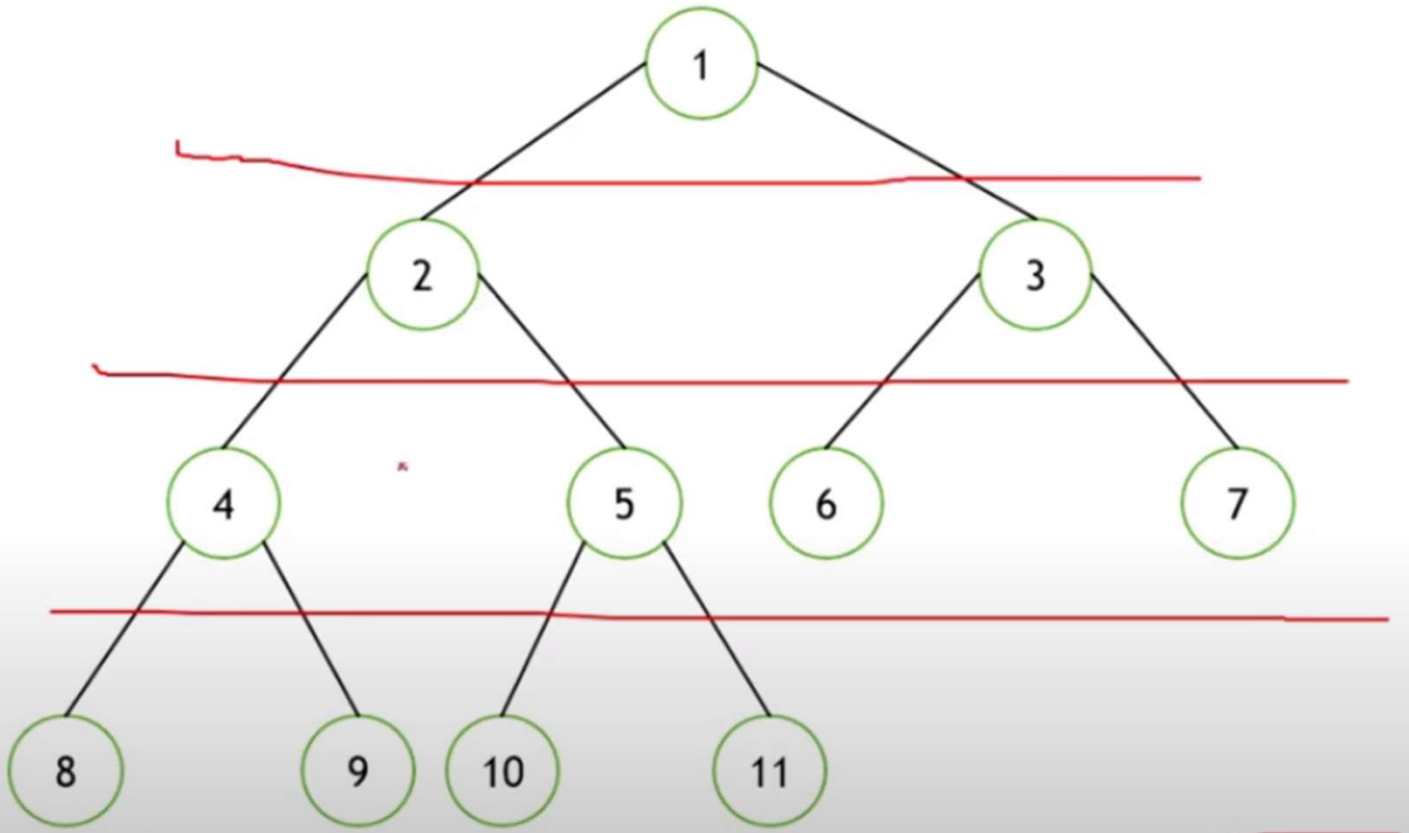
```
function DEPTH-LIMITED-SEARCH(problem, limit) returns soln/fail/cutoff
  RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
  cutoff-occurred? ← false
  if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
  else if DEPTH[node] = limit then return cutoff
  else for each successor in EXPAND(node, problem) do
      result ← RECURSIVE-DLS(successor, problem, limit)
      if result = cutoff then cutoff-occurred? ← true
      else if result ≠ failure then return result
  if cutoff-occurred? then return cutoff else return failure
```

DEFINITION

- Depth Limited Search is the combination of DFS & limits for level
- Depth limited search is also similar to the DFS as it also implements “Last In First Out (LIFO) stack data structure” but in addition it has a level limit

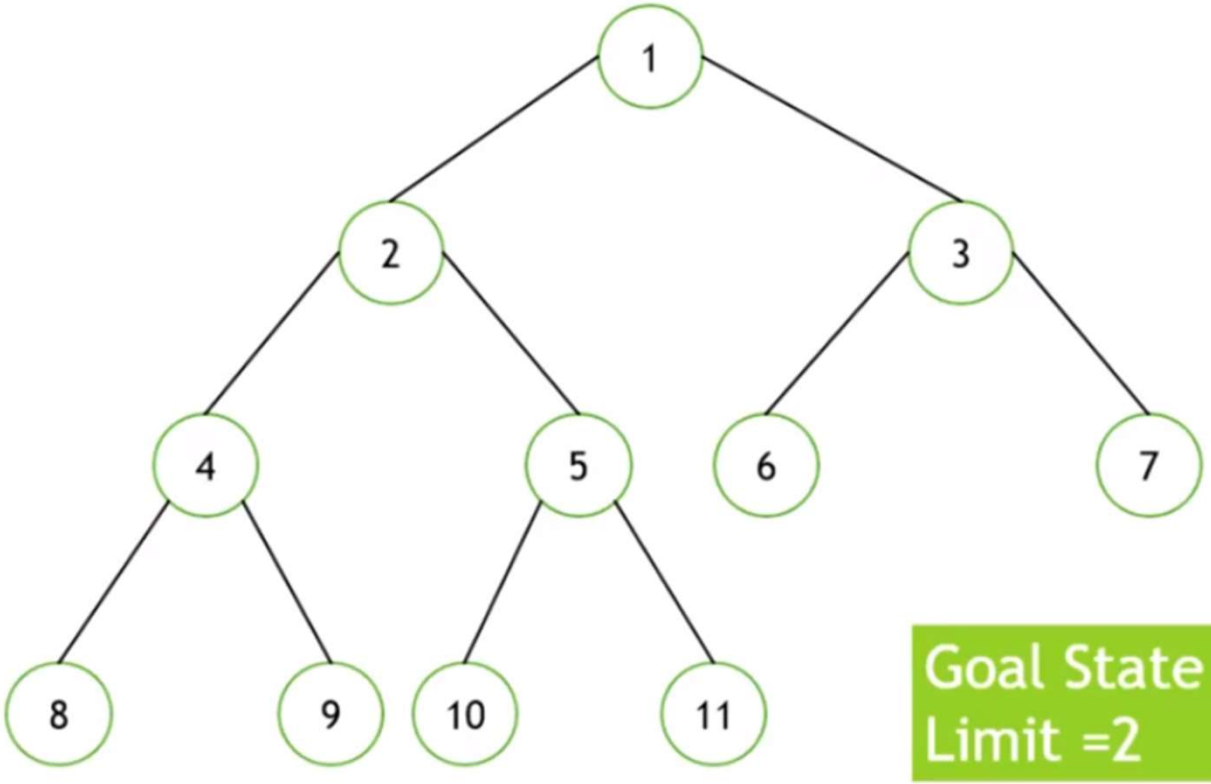
Let us consider



DEFINITION

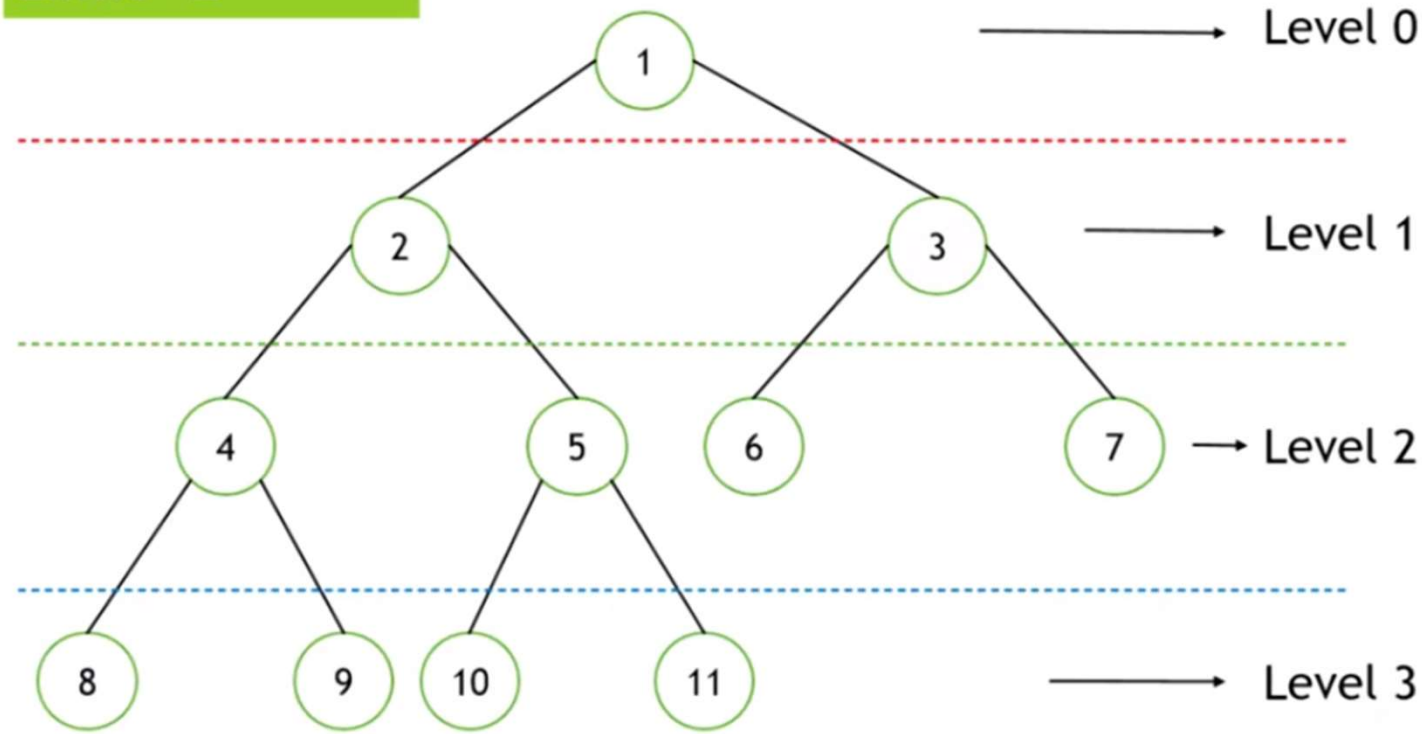
- Depth Limited Search is the combination of DFS & limits for level
- Depth limited search is also similar to the DFS as it also implements “Last In First Out (LIFO) stack data structure” but in addition it has a level limit
- This limit shows that upto which level the algorithm can perform and the levels which are next to the limit are not get executed
- It can be used to solve infinite path problem

Let us consider



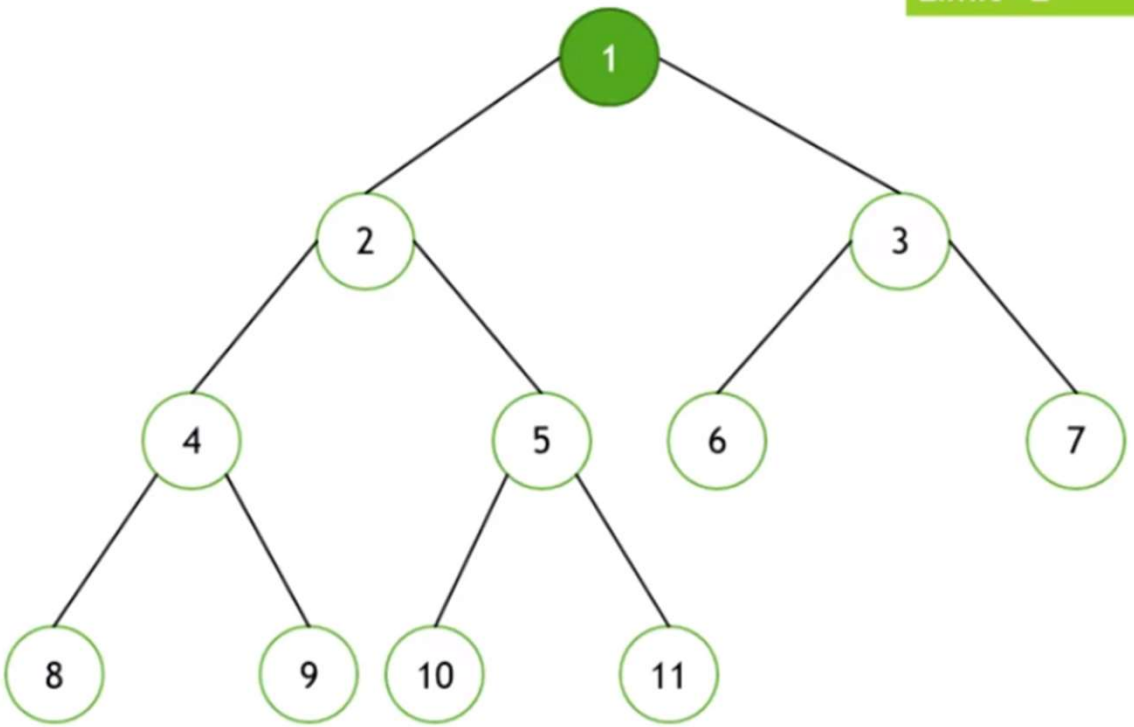
Goal State = 11
Limit =2

Goal State = 11
Limit =2

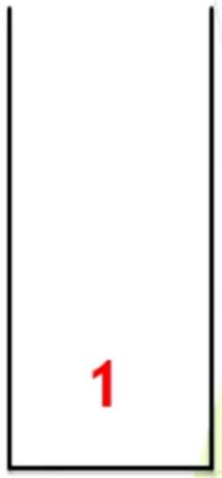


Step 1

Goal State = 11
Limit =2



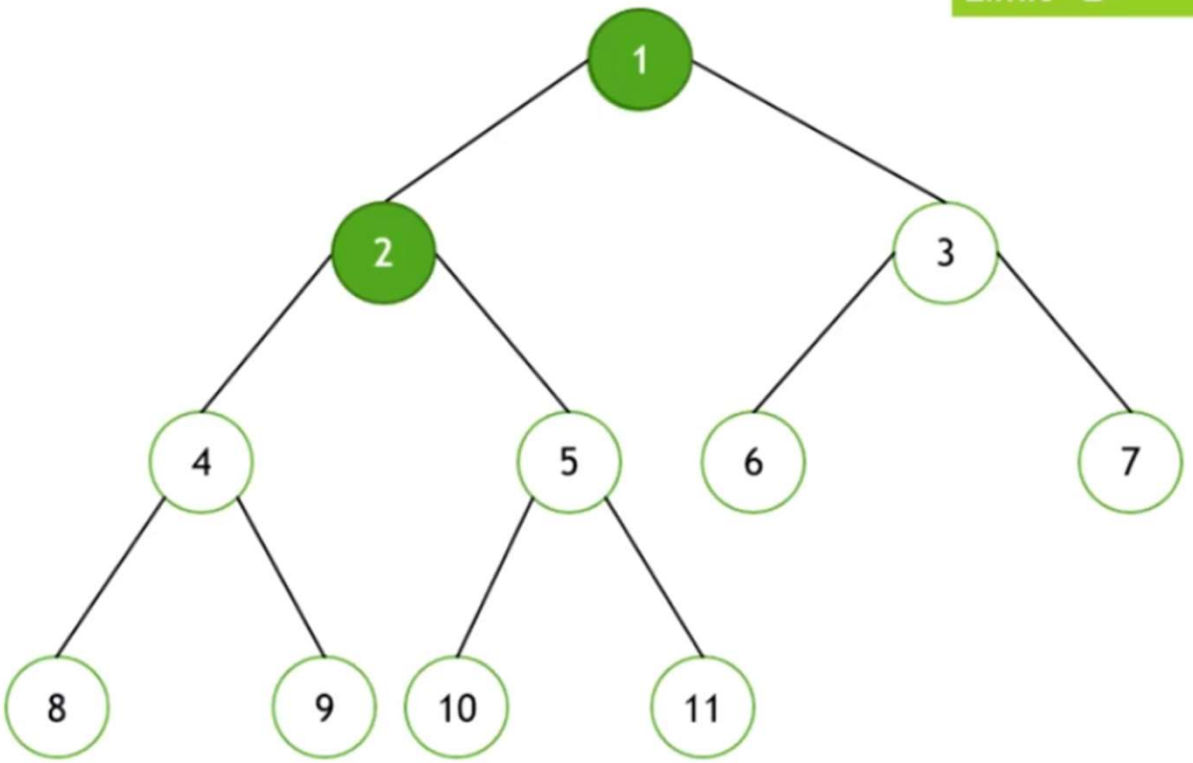
Stack Status:



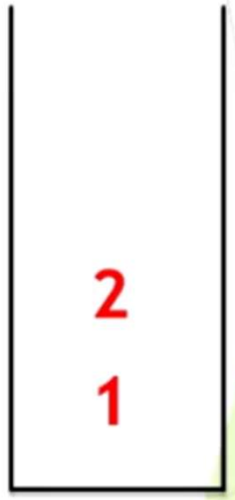
Output sequence: 1

Step 2

Goal State = 11
Limit =2



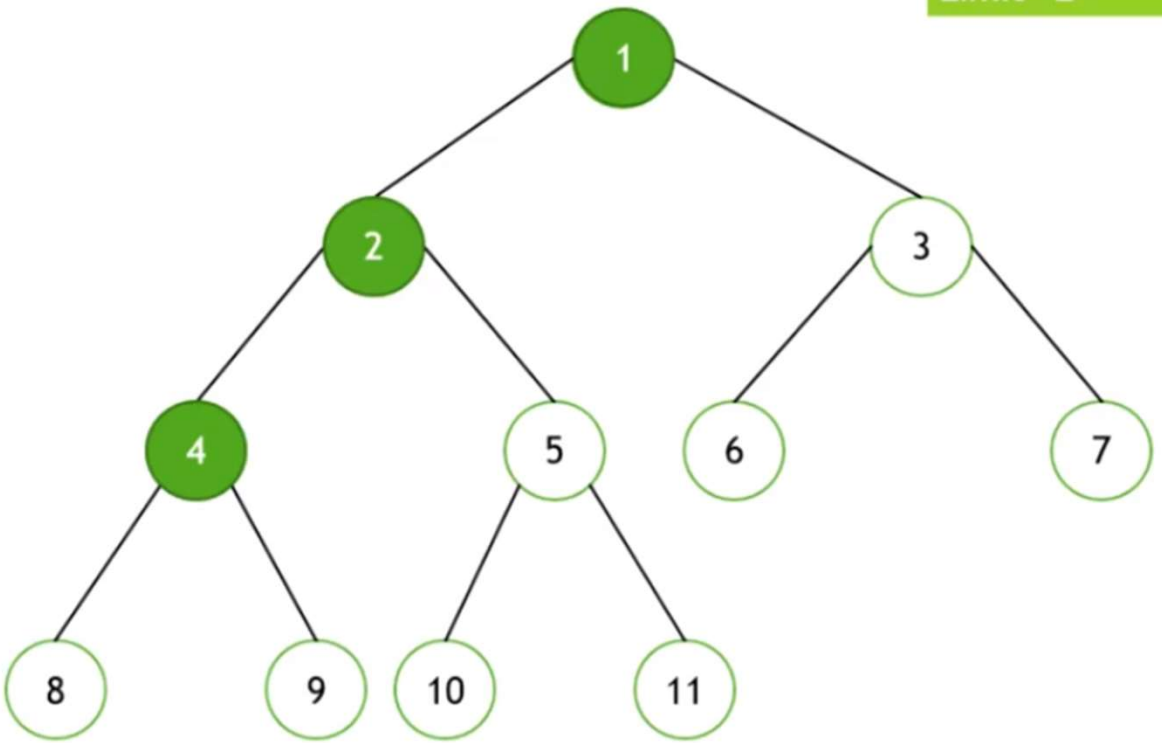
Stack Status:



Output sequence: 1 2

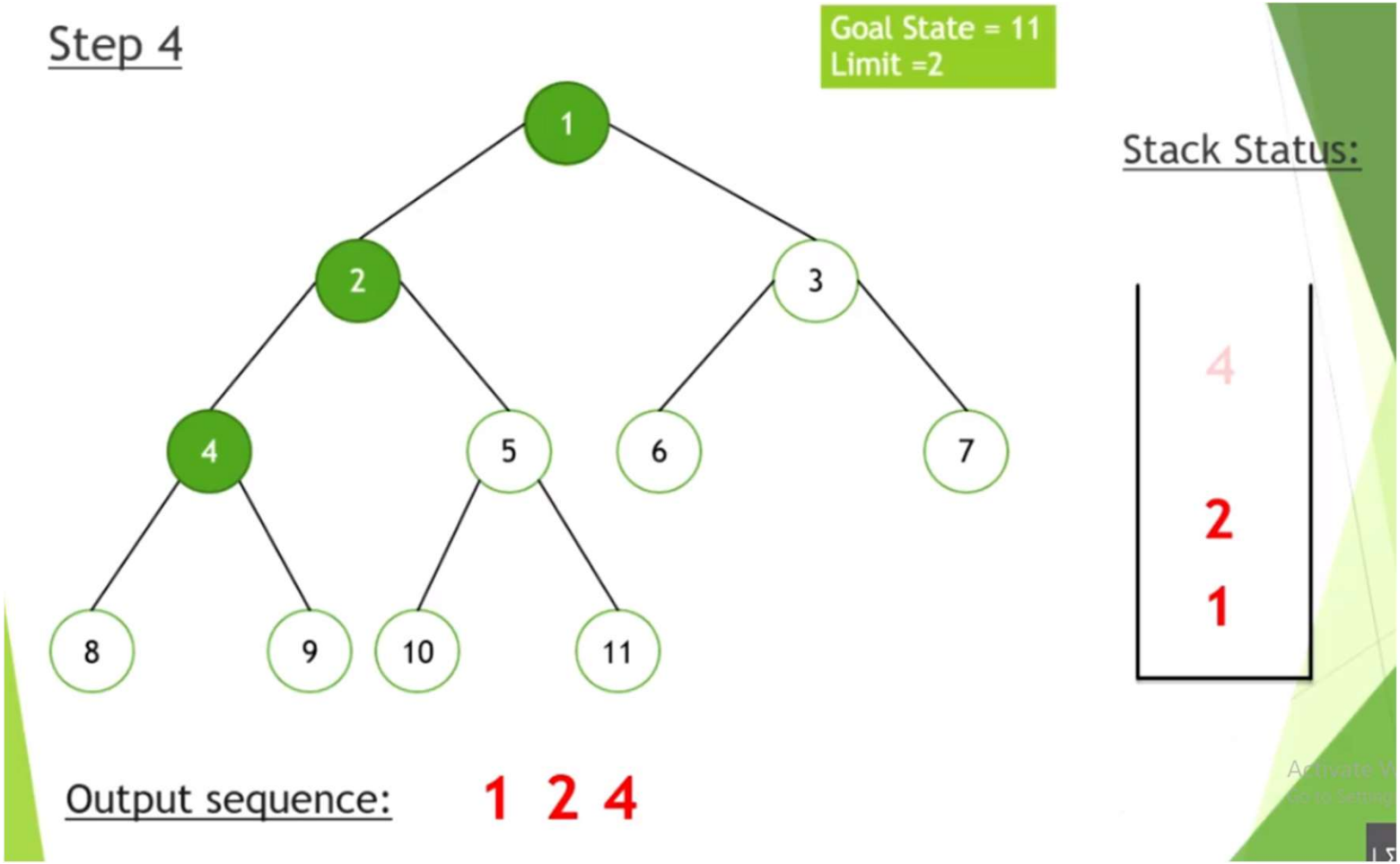
Step 3

Goal State = 11
Limit =2



Stack Status:

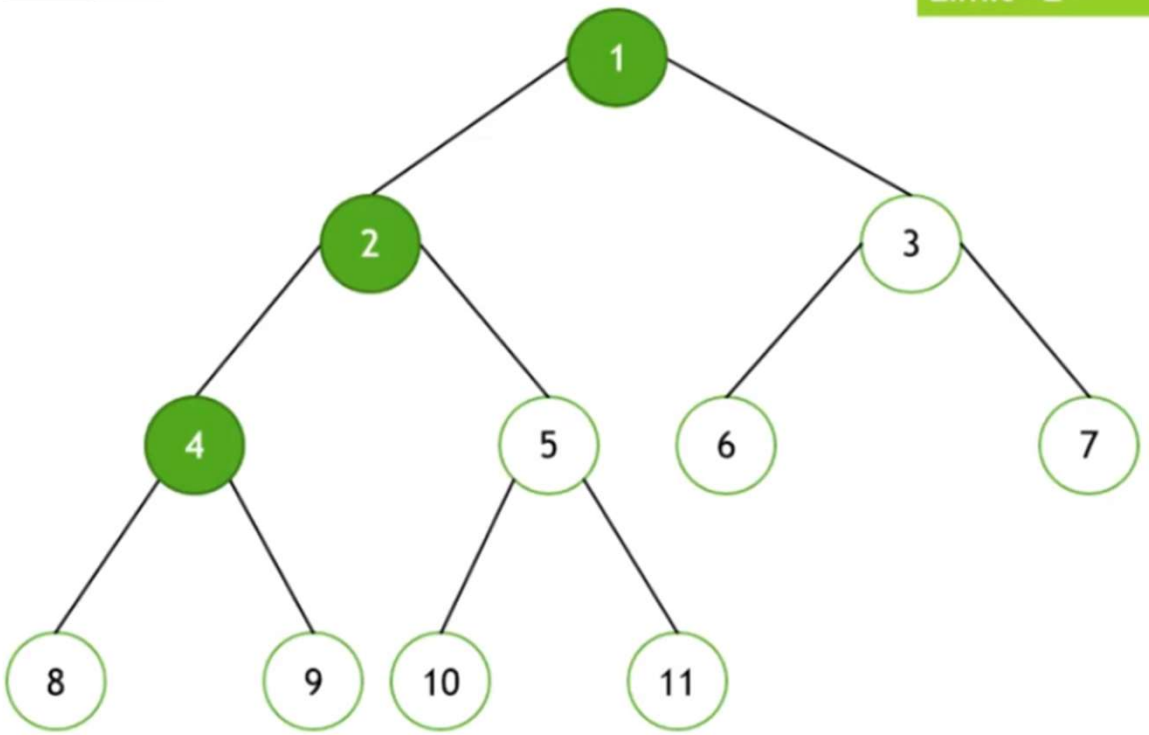
Output sequence: 1 2 4



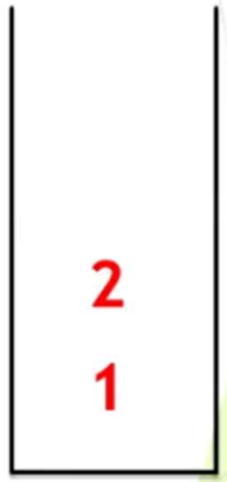
As *no unvisited nodes* are for 4, so **pop 4** from the stack

Step 5

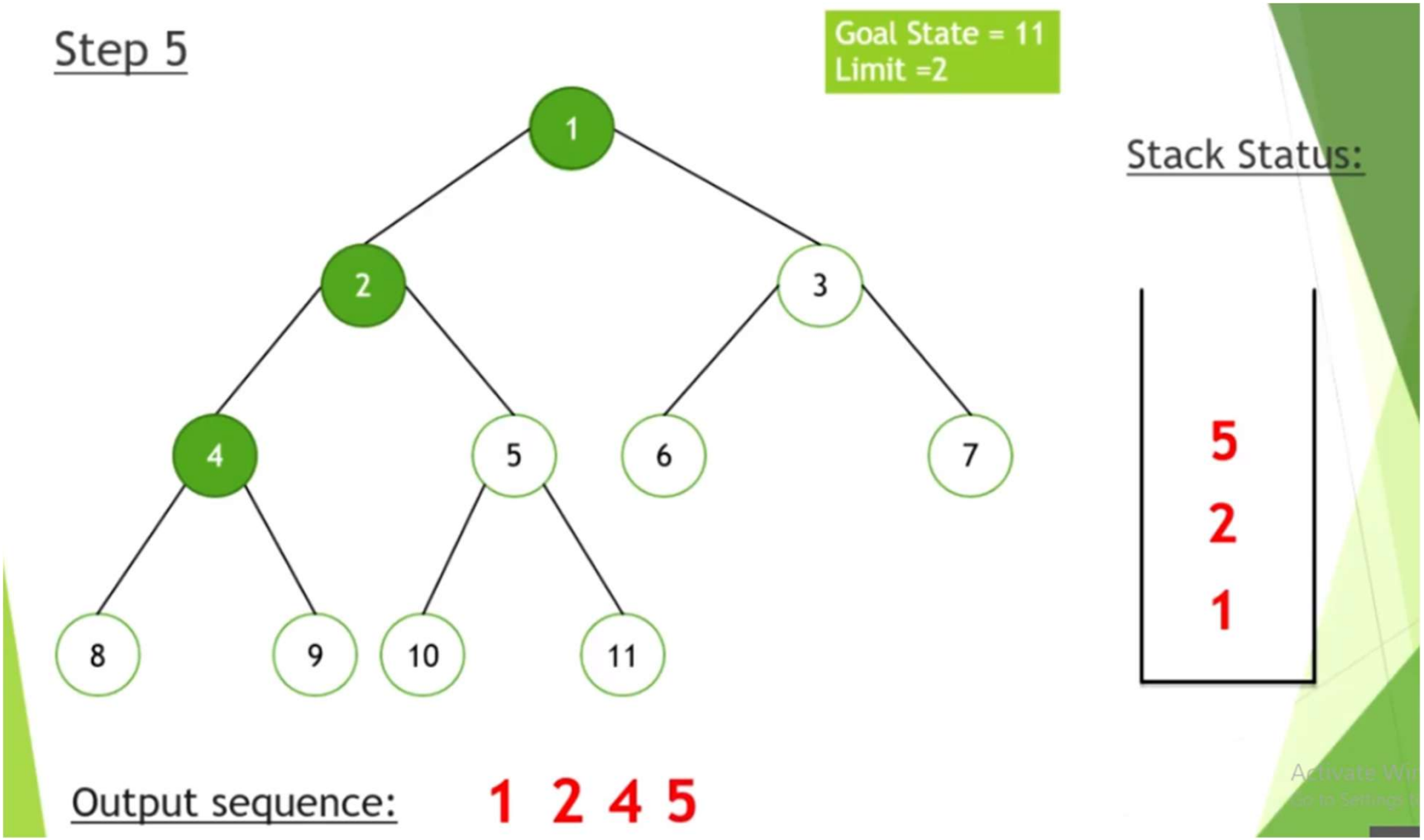
Goal State = 11
Limit =2



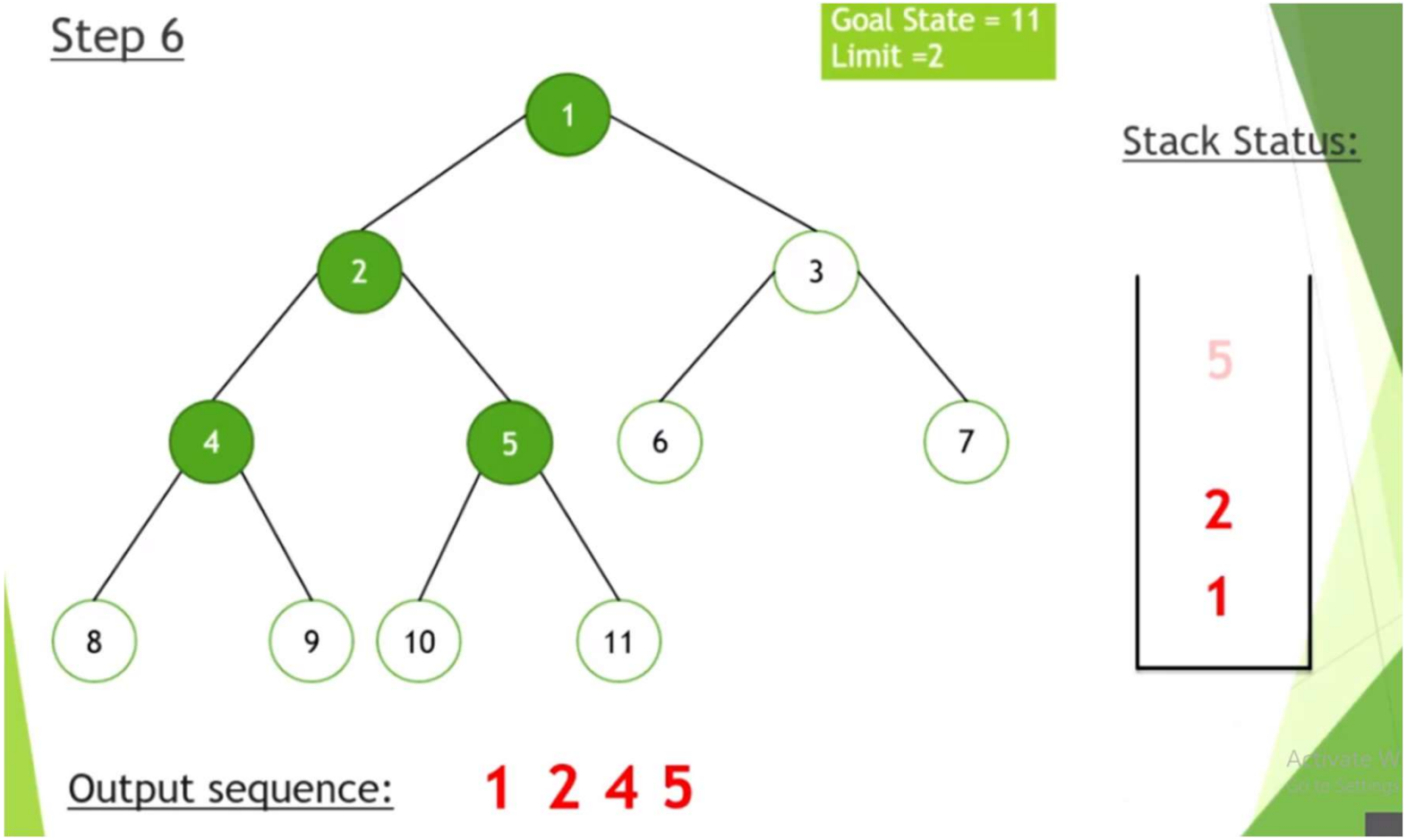
Stack Status:



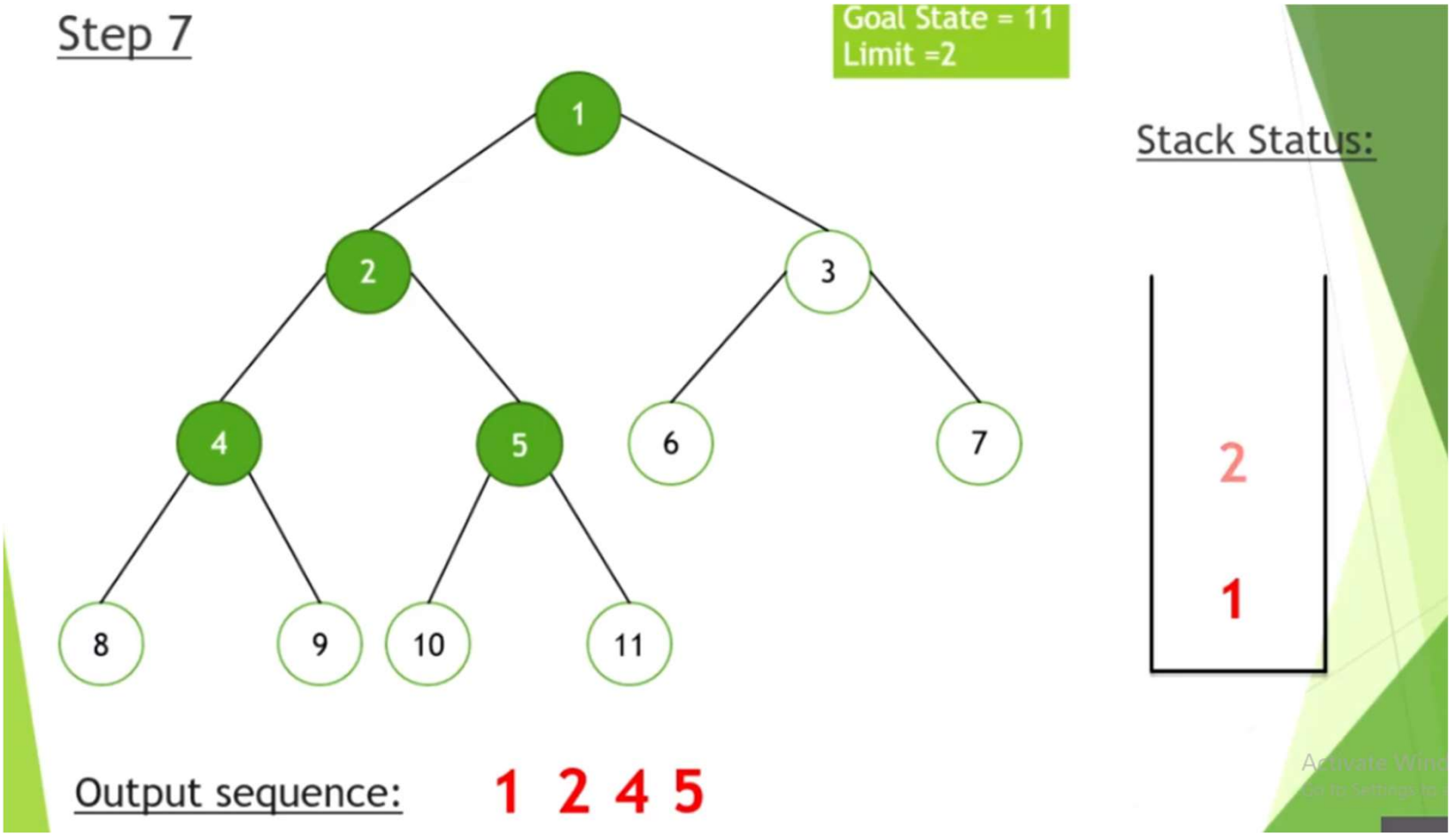
Output sequence: 1 2 4



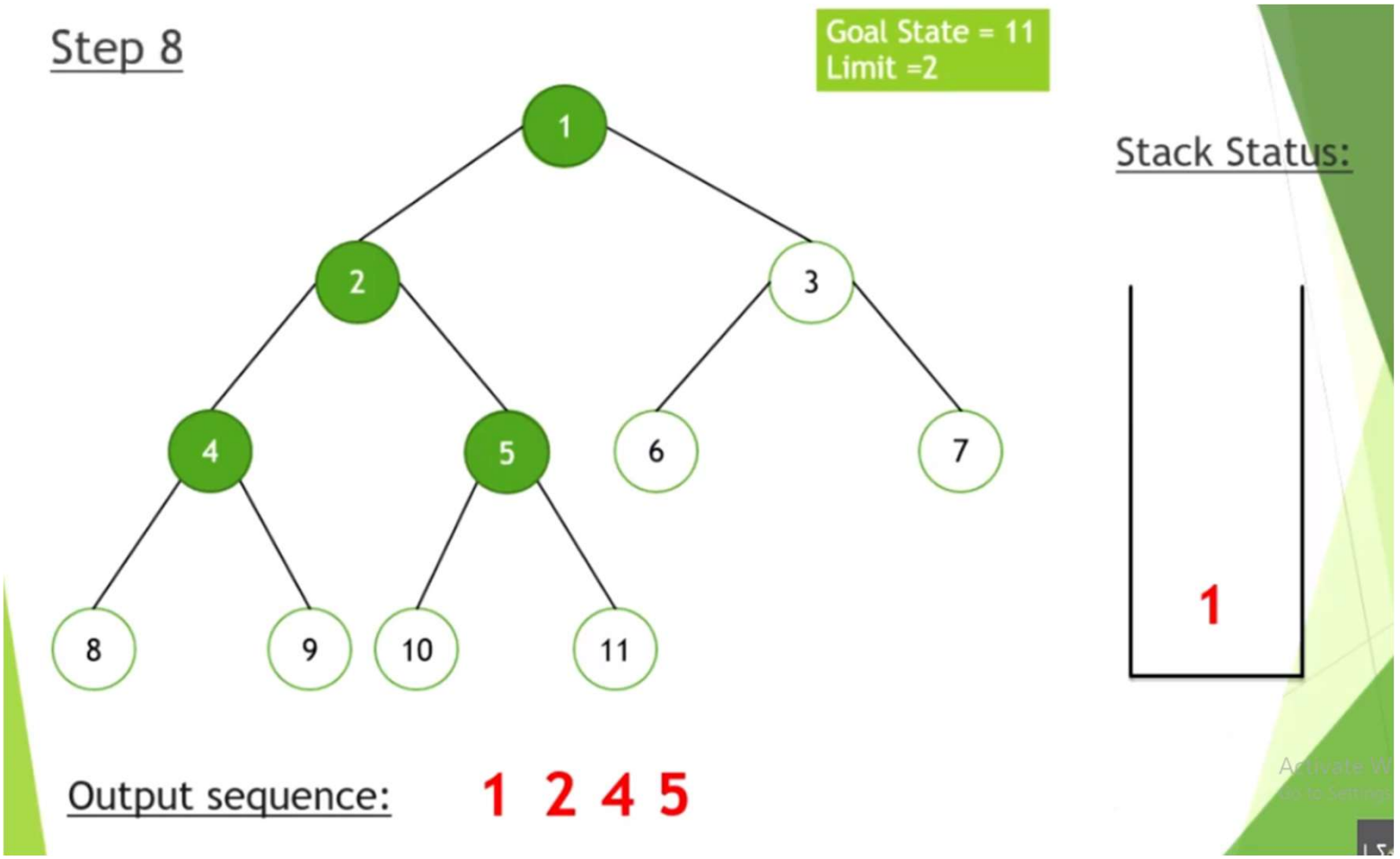
Though **11 is adjacent to 5**, we could not consider it. B'coz it is in **level 3**. Our **fixed level limit is 3**



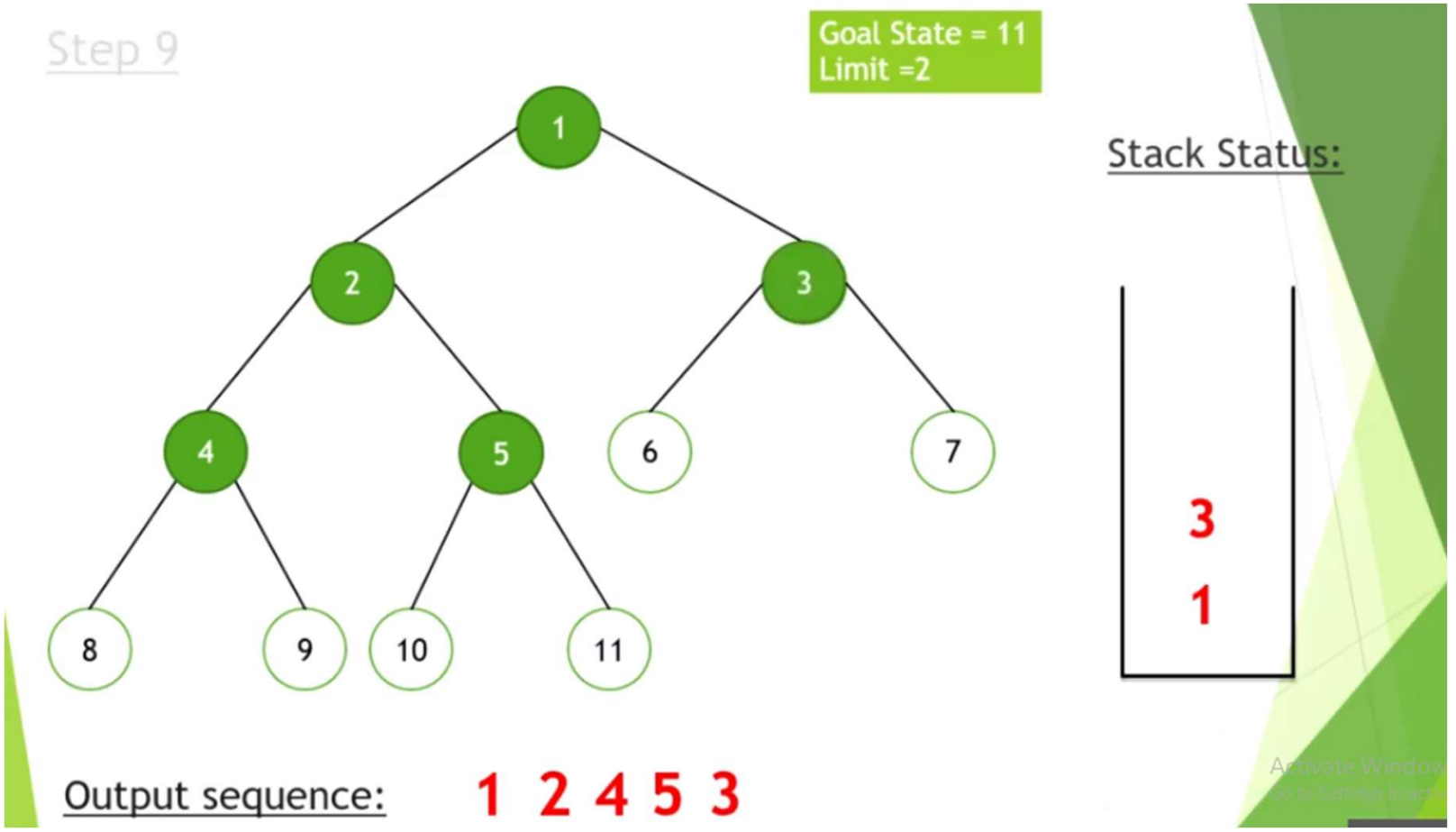
As we need to consider till limit level = 2, *no adjacent nodes* for **5**. so pop **5**



No new adjacent and unvisited nodes for **2**. So pop **2**



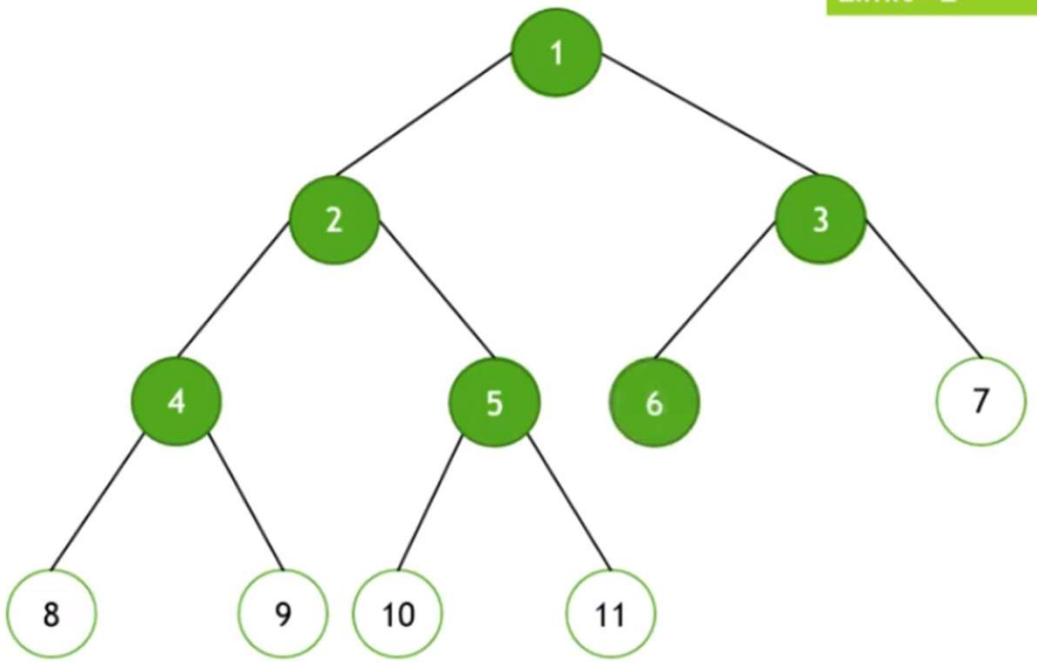
So Top of stack is **1**. Adjacent to 1 is **3**



So Top of stack is **1**. Adjacent to 1 is **3.** Now top of stack is **3**. Its adjacent is **?**

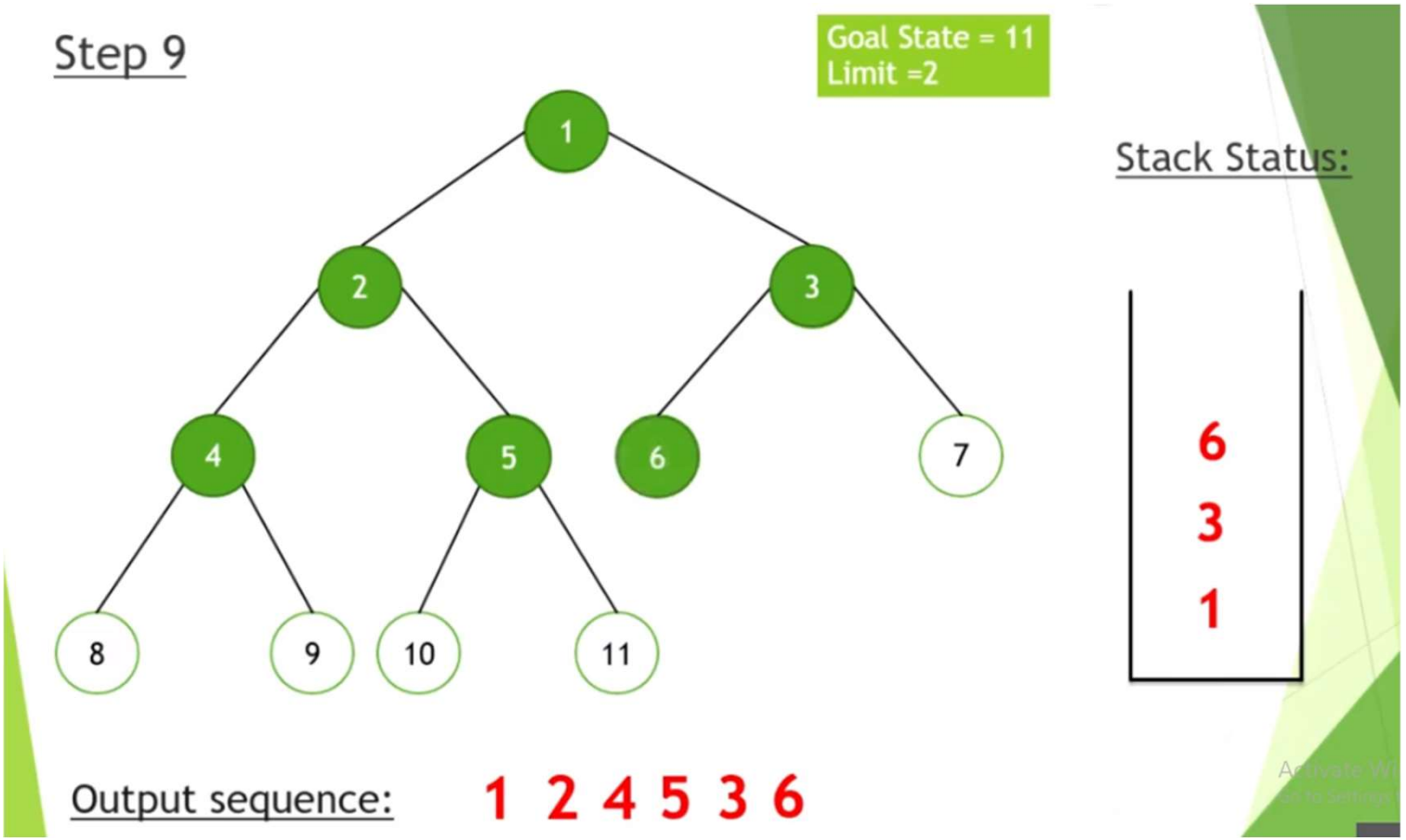
Step 9

Goal State = 11
Limit =2

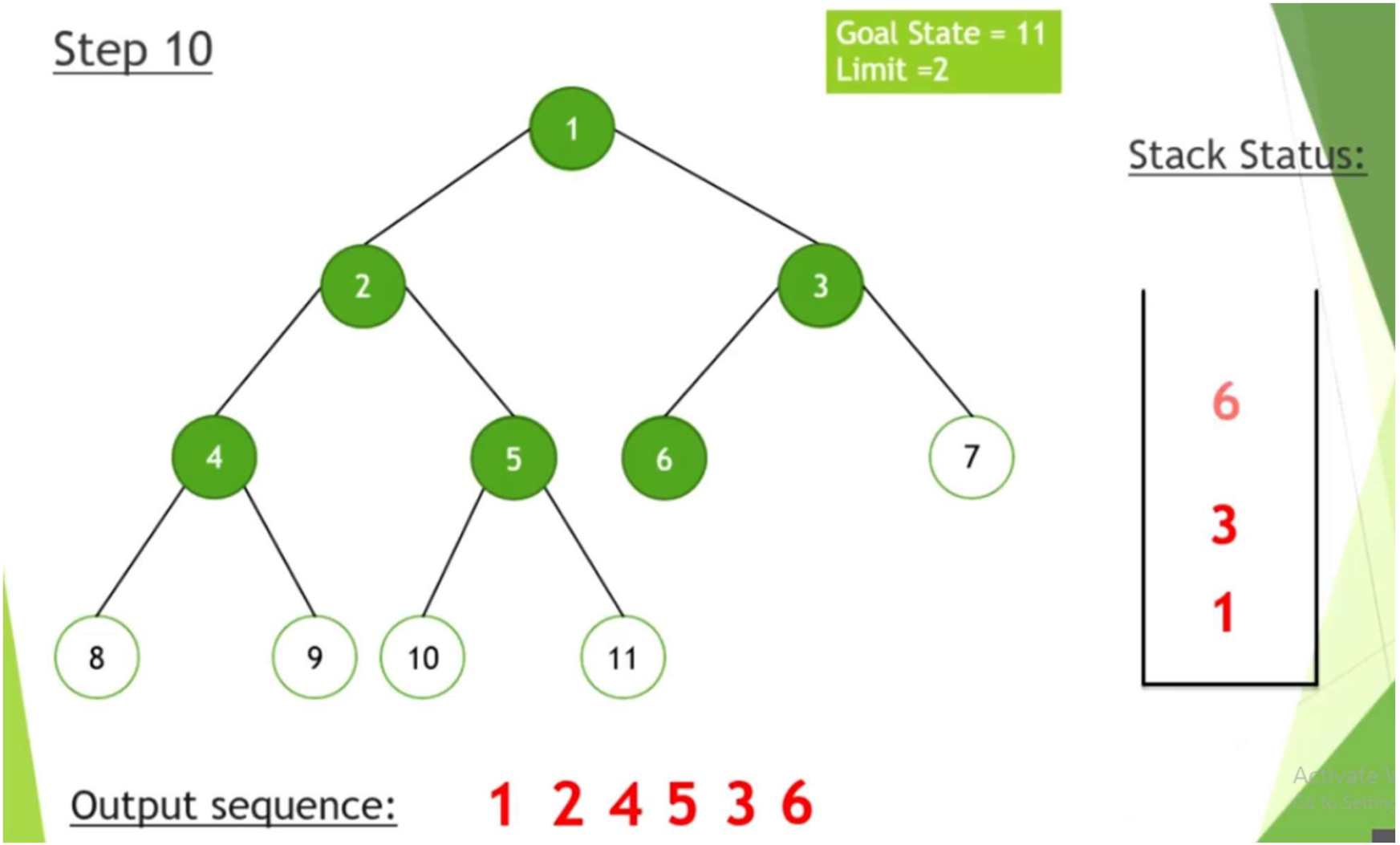


Stack Status:

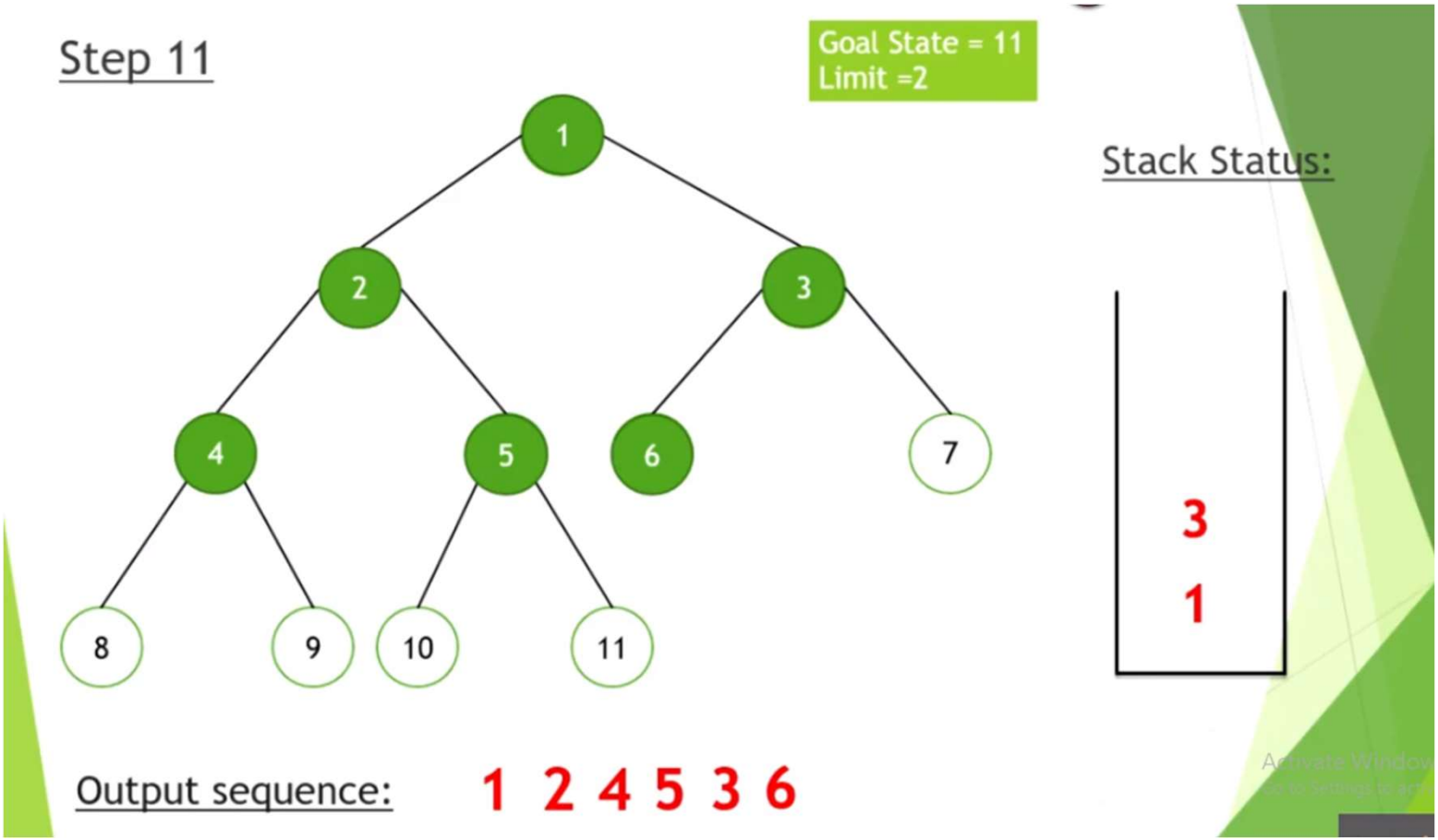
Output sequence: 1 2 4 5 3 6



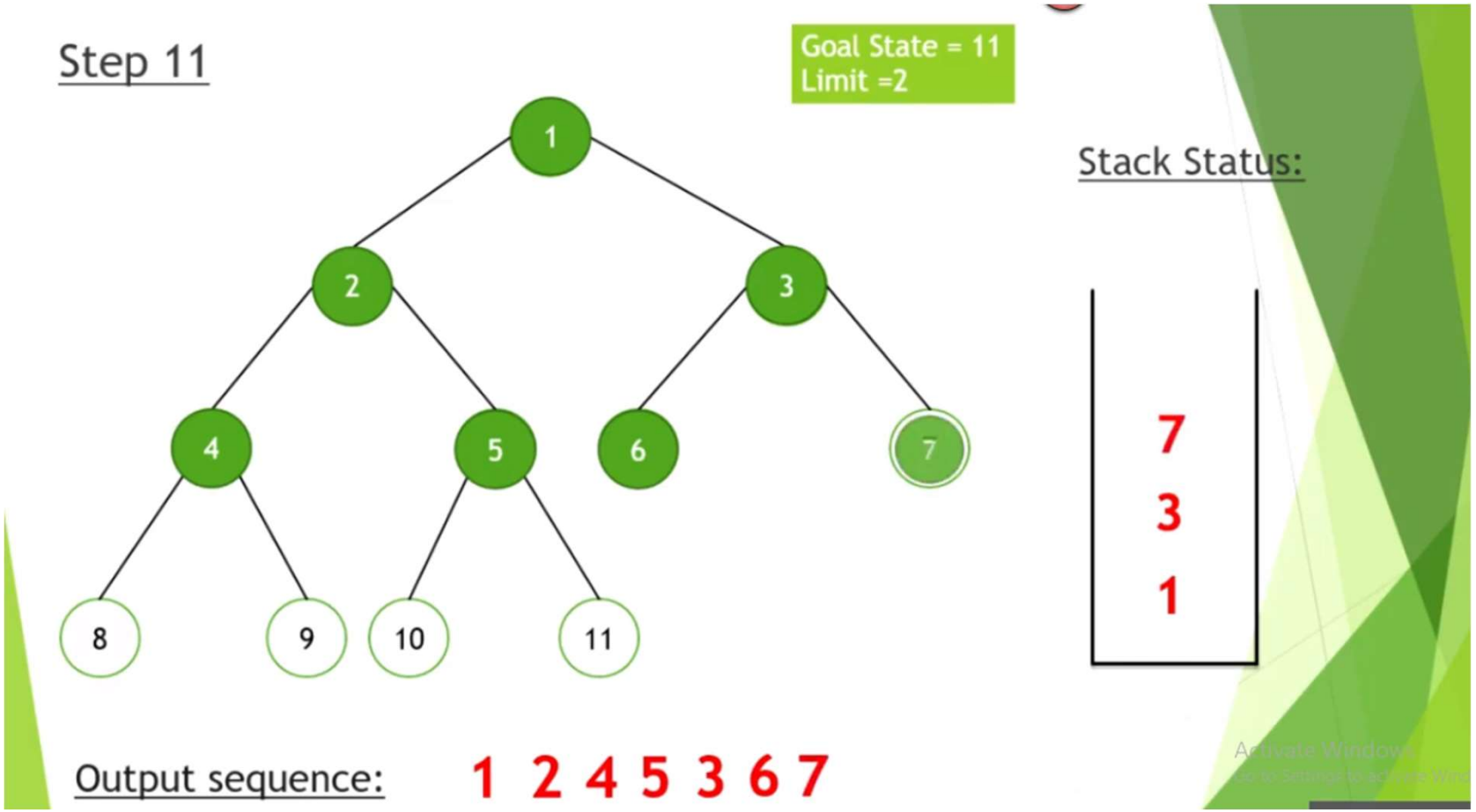
So Top of stack is **6**.



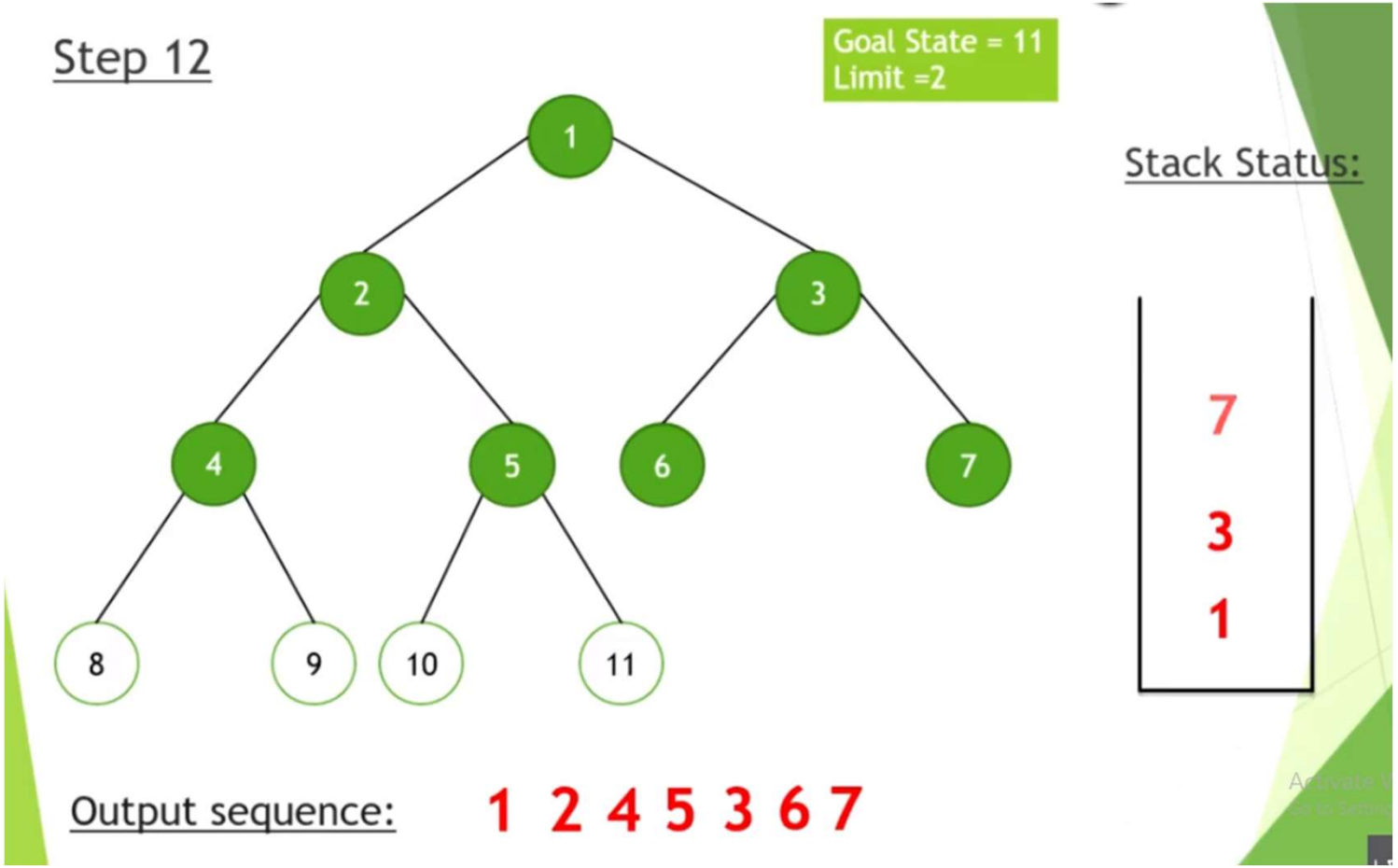
So Top of stack is **6**. No adjacent nodes for **6**. so pop **6**



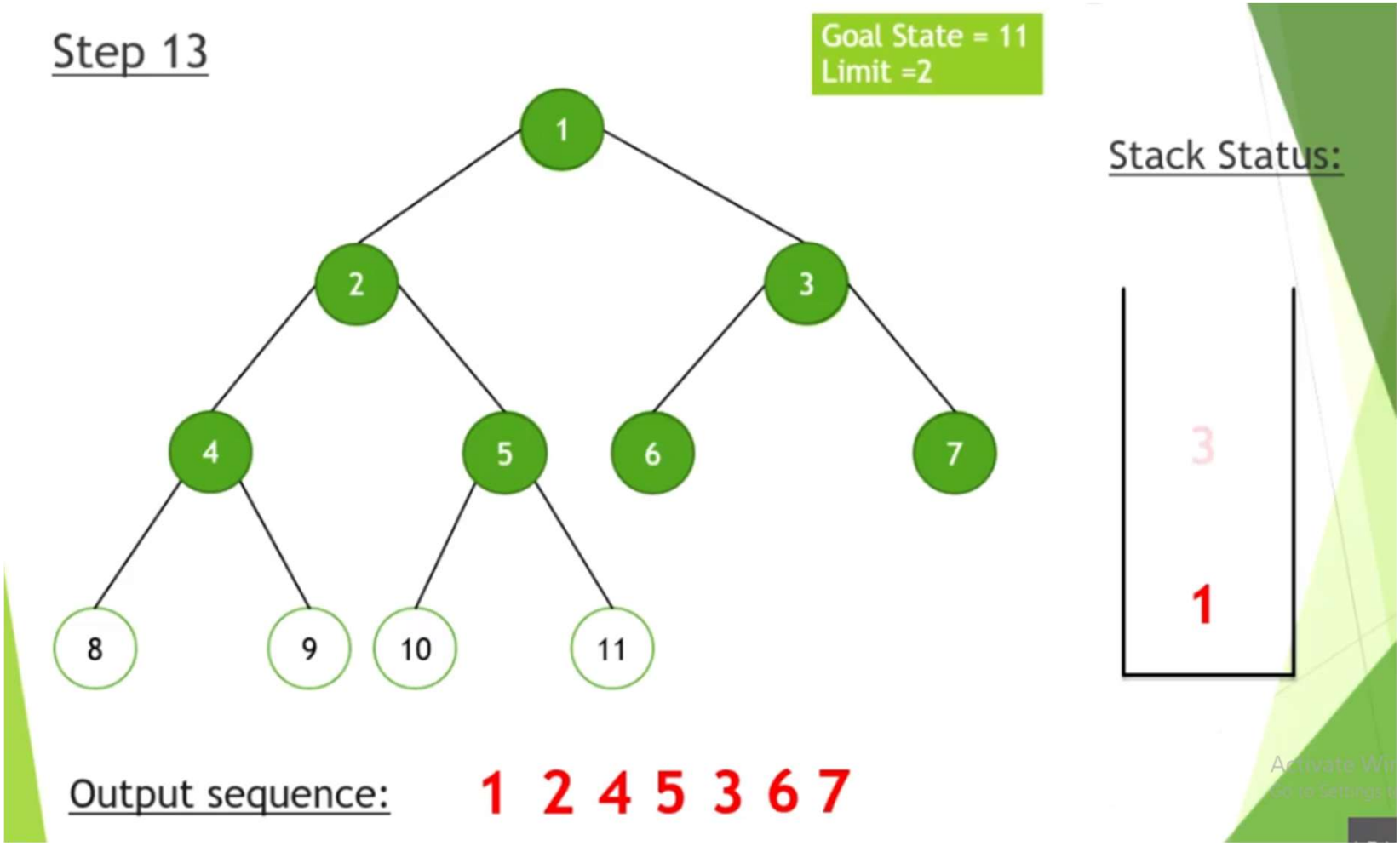
So Top of stack is **3**. Adjacent and Unvisited node is **7**



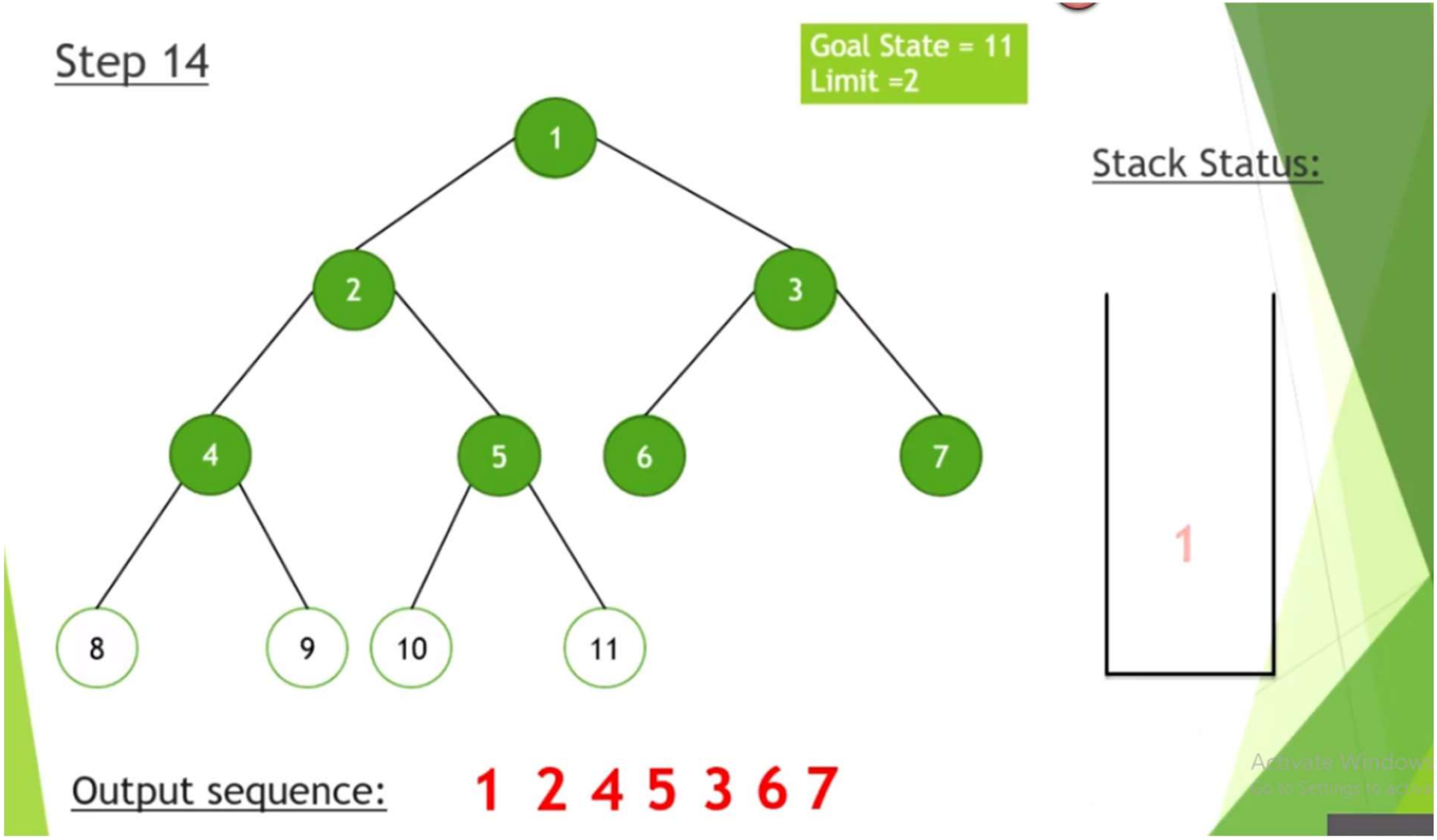
Insert **7** into output sequence. Push **7** onto stack



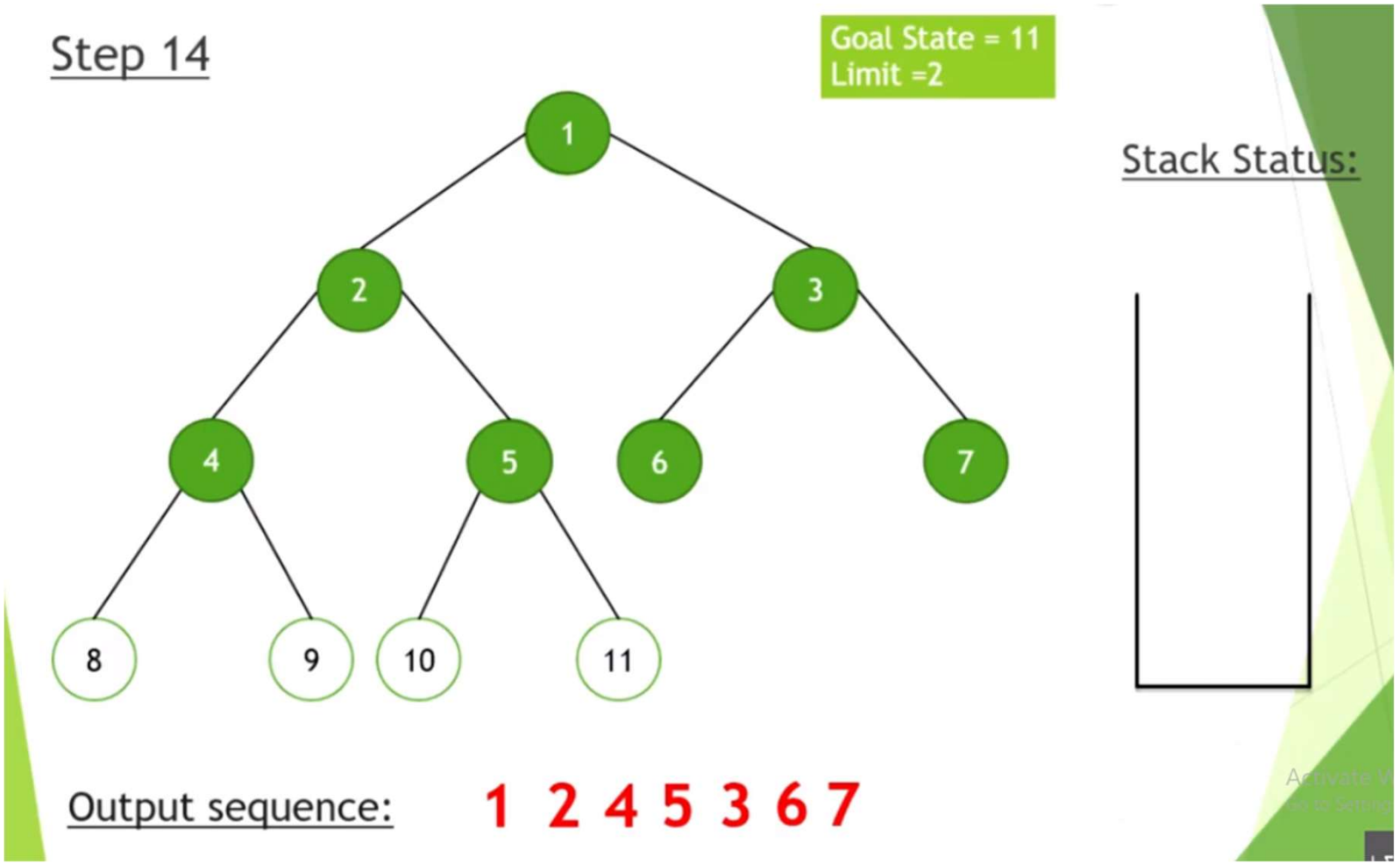
Top = 7 , No adjacent nodes for 7. So pop 7



Top = 3, No adjacent nodes for 3. So pop 3



Top = 1, No adjacent nodes for 1. So pop 1



Goal : Find the GOAL node and return its LEVEL.

Here: Did not reach the goal. So the level is not found
Hence: This is not a complete solution. Bcoz it gave some solution rather than a correct solution

Algorithm

- In DLS the root node or the starting node is pushed into the stack and the node is marked as visited
- Check the level limit value. If the limit is reached stop the process
- The top of the stack is need to check and its adjacent unvisited node is pushed into the stack based on alphabetical order
- Only one node is pushed into stack at a time
- If there is no unvisited nodes for a top of stack node then it is popped out from the stack
- Repeat from step 2
- Once the stack gets empty stop the process
- If the goal node is found. Return the level at which is found

Performance Evaluation

- Completeness -> DLS is not a completed search as it does not guarantees a solution i.e. the solution may or may not be obtain
- Optimality -> It is not optimal
- Time Complexity -> $O(b^l)$ (Where b is the branching factors or number of nodes and l is depth of the search tree or number of levels in search tree)
- Space Complexity = O(bl)

Iterative deepening search

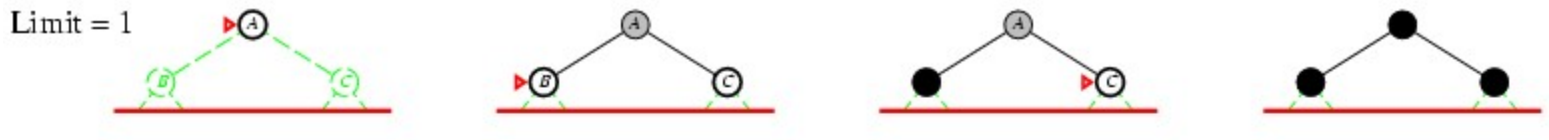
- The iterative deepening algorithm is a combination of DFS and BFS algorithms.
- This search algorithm finds out the best depth limit and does it by gradually increasing the limit until a goal is found
- This Search algorithm combines the benefits of Breadth-first search's fast search and depth-first search's memory efficiency
- The iterative search algorithm is useful uninformed search when search space is large, and depth of goal node is unknown

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

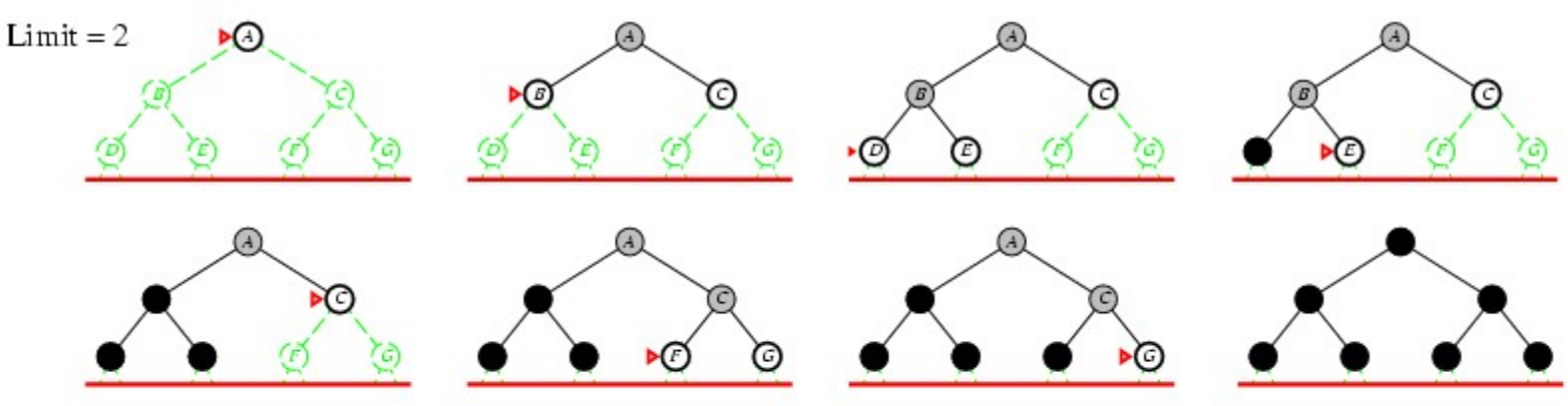
Iterative deepening search l =0

Limit = 0

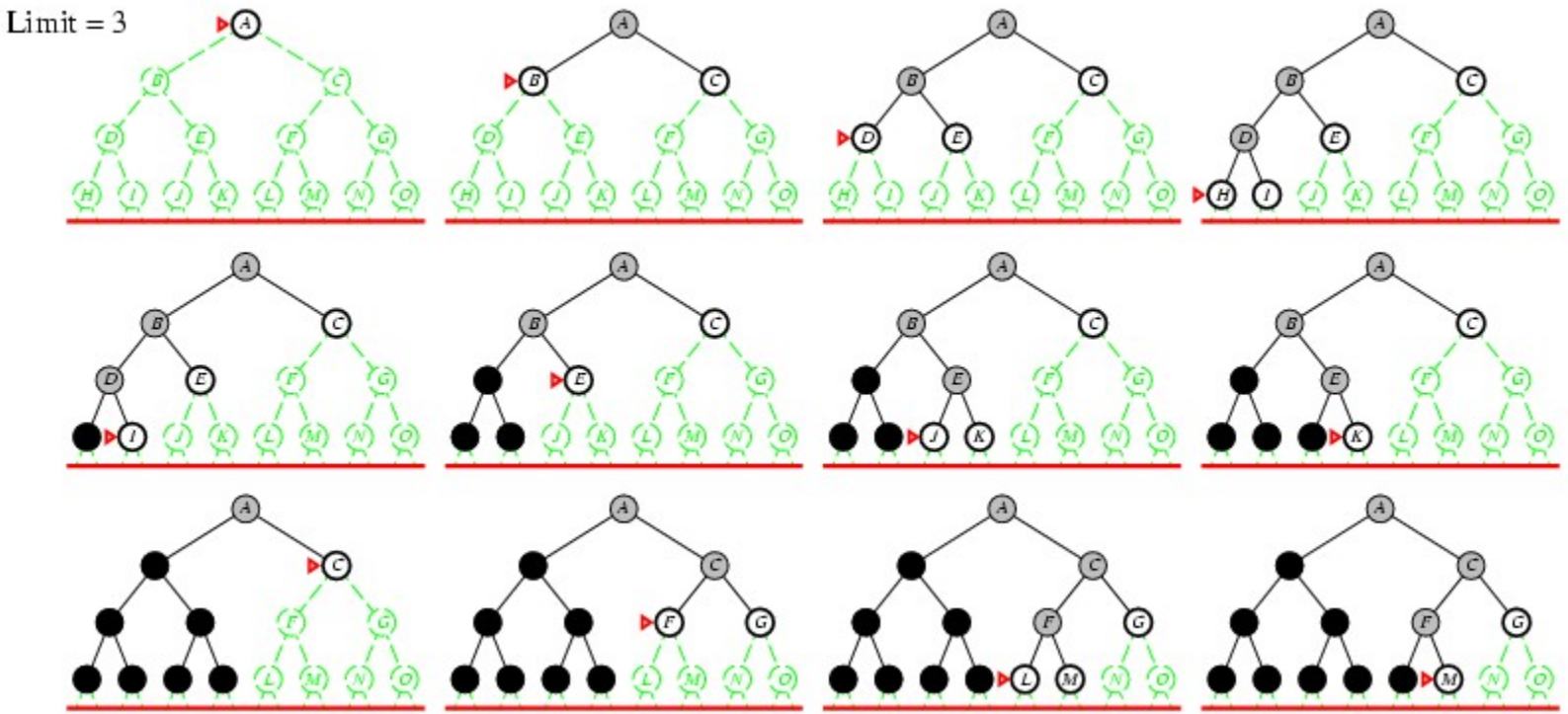
Iterative deepening search l =1



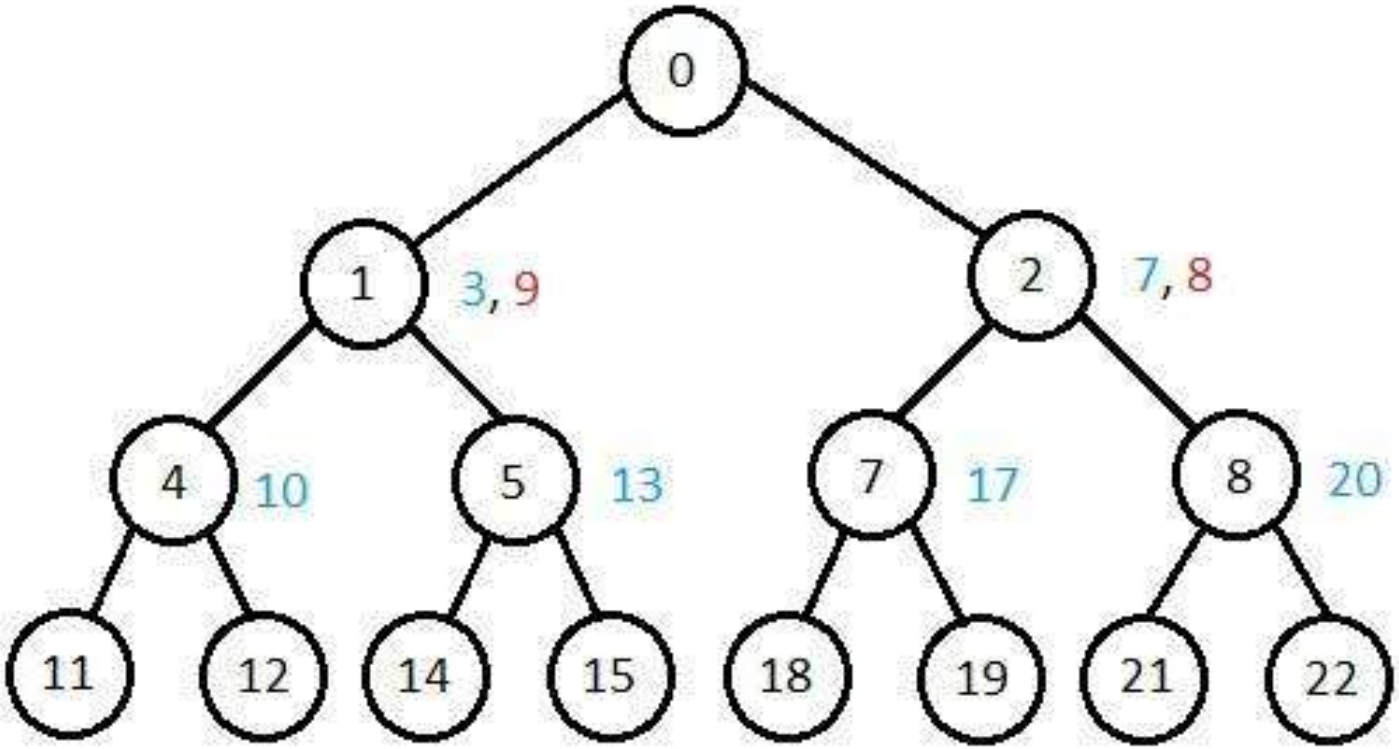
Iterative deepening search *l* =2



Iterative deepening search *l* =3



Another Example



Iterative deepening search

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

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

- 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

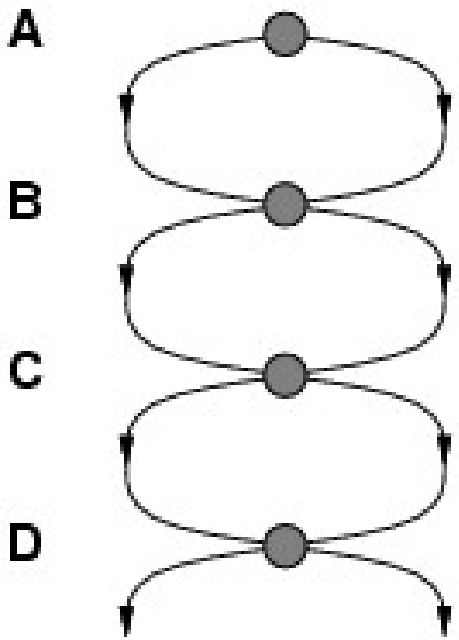
- Complete? Yes
- Time? $(d+1)b^0 + d\,b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space? $O(bd)$
- Optimal? Yes, if step cost = 1

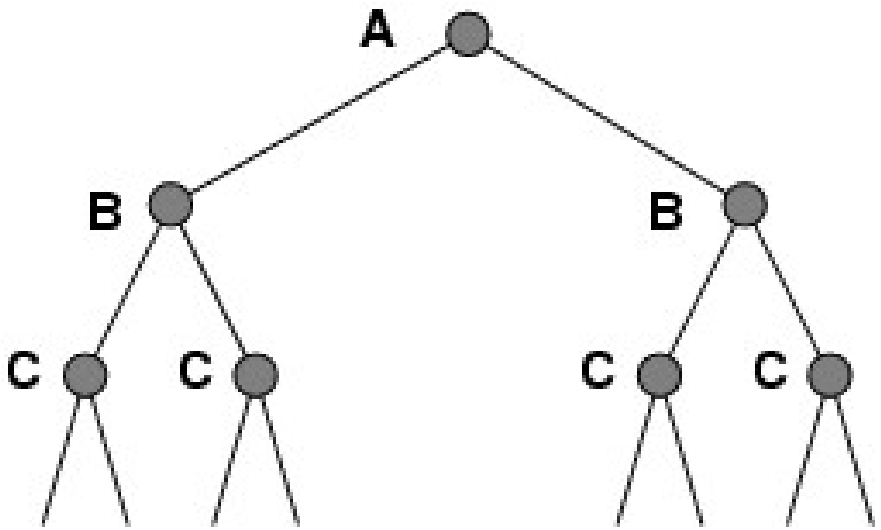
Summary of algorithms

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

Repeated states

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





Graph search

```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

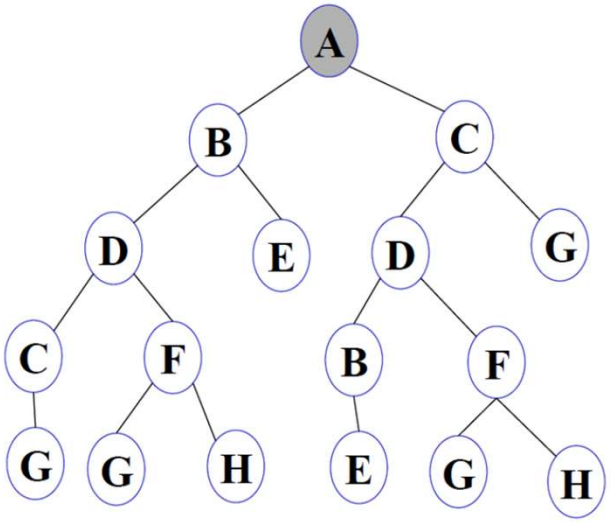
Summary

- Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored
- Variety of uninformed search strategies
- Iterative deepening search uses only linear space and not much more time than other uninformed algorithms

Assessment Questions

Write and explain the BFS search strategy with eg.
Write and explain the DFS search strategy with eg.
Write and explain the depth limited search strategy
Write and explain the bidirectional search strategy

Work out the order in which nodes will be visited for a. BFS b. IDS



Assessment Questions
…contd

The missionaries and cannibals problem is usually stated as follows. Three missionaries and three cannibals are on one side of a river, along with a boat that can hold one or two people. Find a way to get everyone to the other side, without ever leaving a group of missionaries in one place outnumbered by the cannibals in that place. The problem is famous in AI because it was the subject of the first paper that approached problem formulation from an analytical view point. a. Formulate the problem precisely, making only those distinctions necessary to ensure a valid solution. Draw a diagram of the complete state space. b. Implement and solve the problem optimally using an appropriate search algorithm. Is it a good idea to check for repeated states? c. Why do you think people have a hard time solving this puzzle, given that the state space is so simple?
a. Solve 8-puzzle using iterative deepening depth first search.