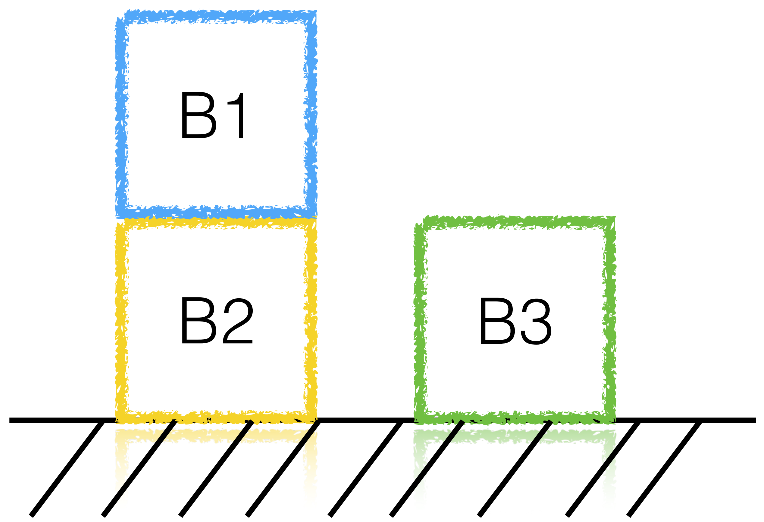
Algorithms for Data Structures: Uninformed Search

Phillip Smith
19/11/2013

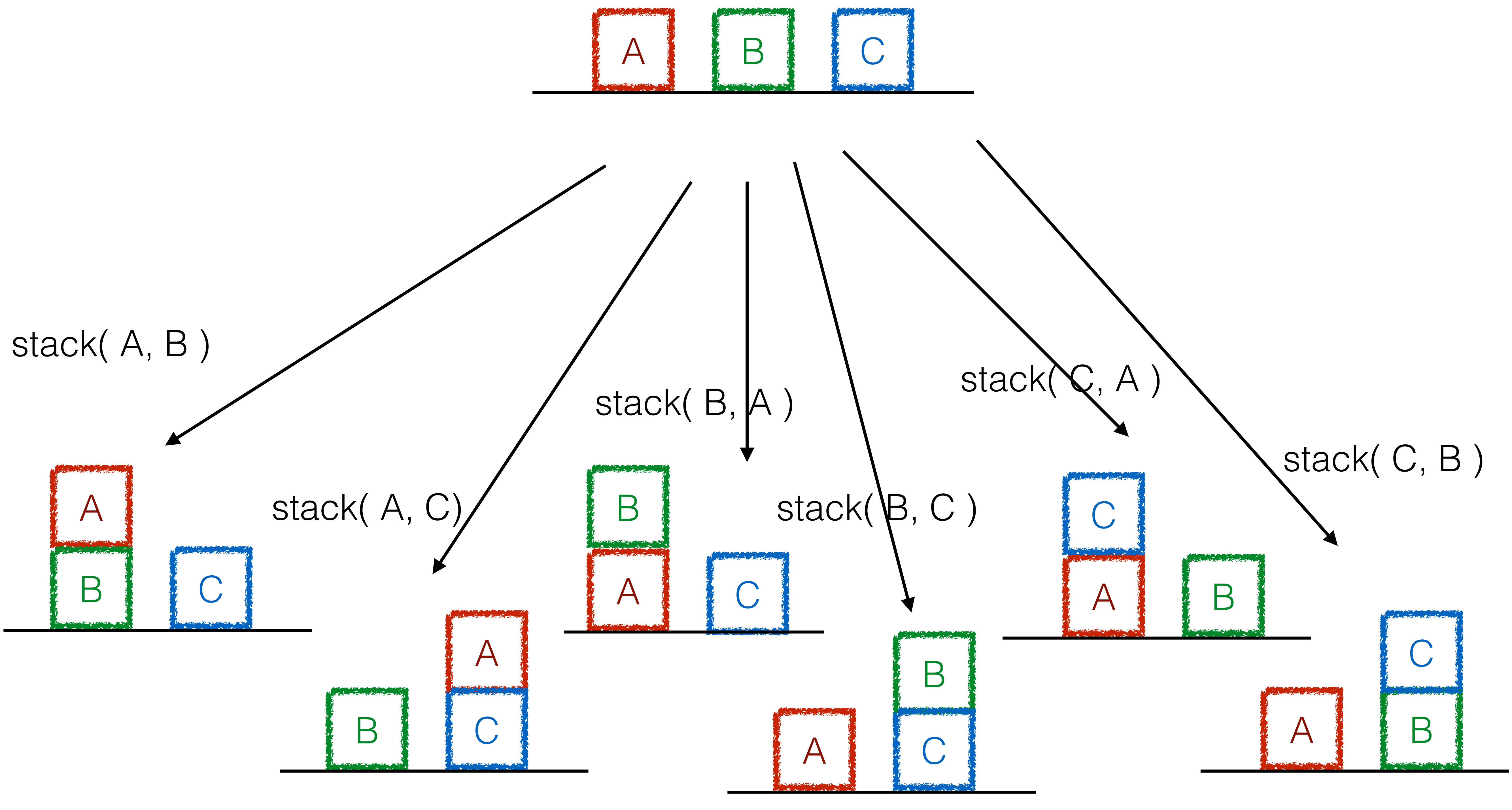
Representations for Reasoning

- We know (at least) two models of a world:
  - A model of the <u>static</u> states of the world
  - A model of the effects of actions on the first model

A Static World



Effects of Actions



Decision Making

- "*It is easy to choose among options when one appears better than all of the rest. But when you find things hard to compare, then you may have to deliberate.*" - Minsky (2006)

- We need to know about <u>goals</u> and <u>sub-goals</u>

Search Applications

- Simple social networking
- Searching the internet
- Playing games against an AI opponent: Chess
- Route finding: Robot navigation

Using a State-space Graph to Find Plans

1. Select a goal state

2. Identify the current state

- Finding a solution is simply a case of finding a path between these two in the state-space graph

Using a State-space Graph to Find Plans

- The <u>solution</u> or <u>plan</u> is the sequence of labels on the arcs
- Usually graphs are so large that we can't hold all of them explicitly in memory
- There may be many possible paths to a goal state
  - We may wish to find the path of least cost or optimal path

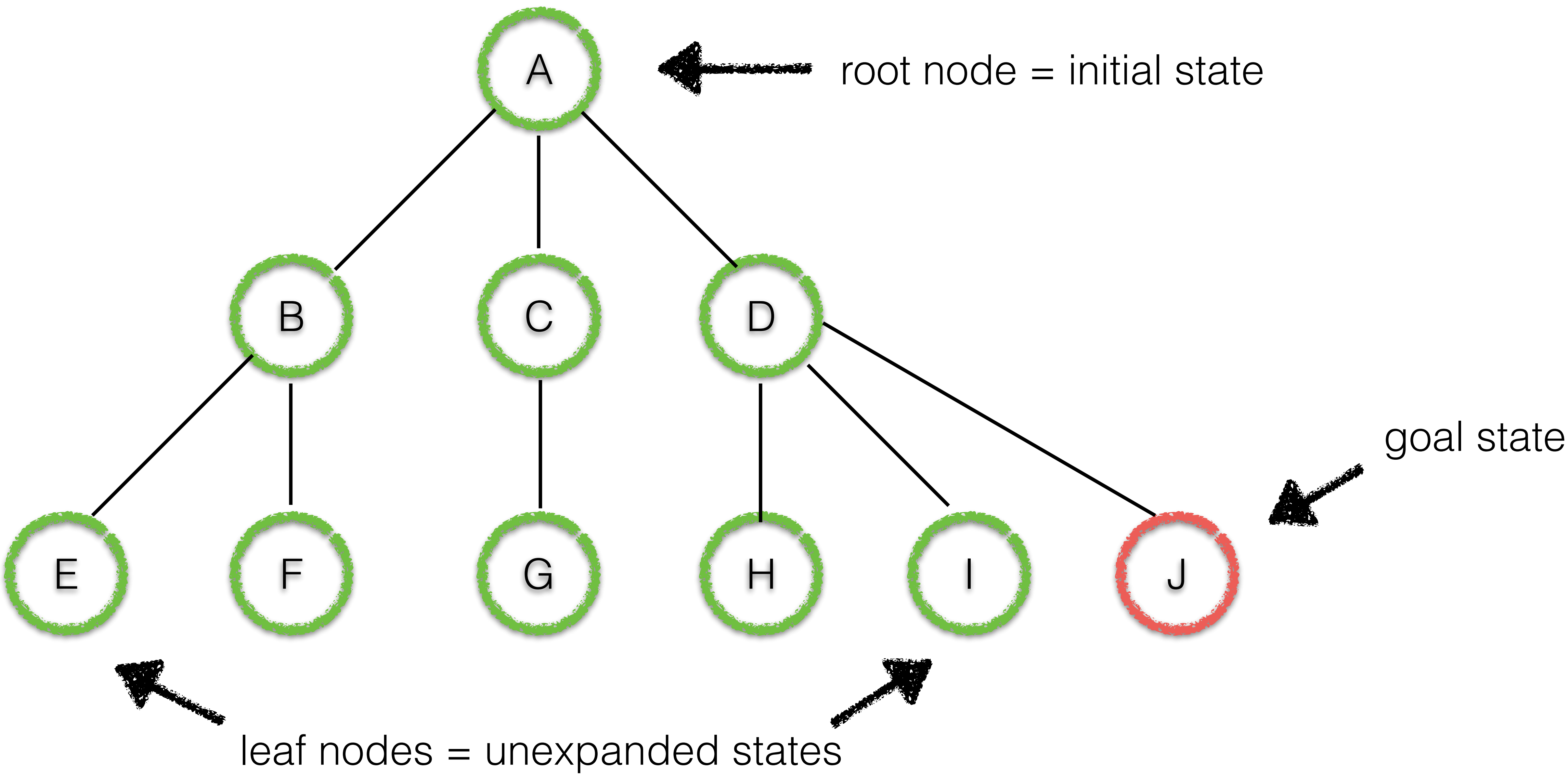
State-space Graphs

- Typically we need to predict the effects of <u>sequences</u> of actions

- If the number of states of the world is small enough we can draw a <u>complete</u> state-space graph

Search Trees

- We represent only the explored portion (or less) of the graph as a search tree:



Search Trees

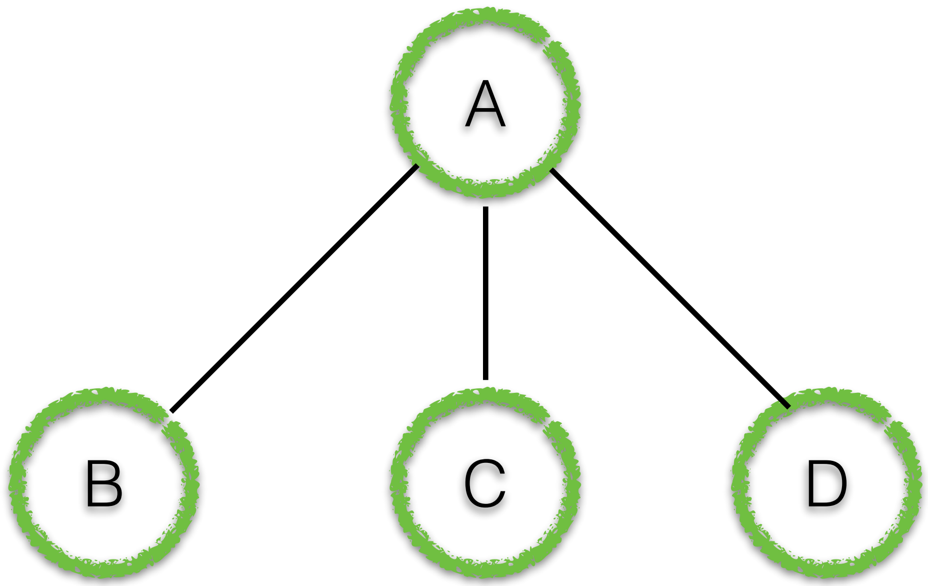
- Blocks world example:
  - B = node
    - state representation: ((B1 B2) (B3))
    - parent node: ((B1) (B2) (B3))
    - operator (action): stack(B1, B2)
    - depth: 1
    - path cost: 1

Search Trees: General Rules

- Each node has only one parent
- If a node can be reached by two paths, we only remember the parent on the path with the lowest cost

Generating Search Trees

- We generate the search tree by expanding nodes
- Expanding a node ≡ generating its children

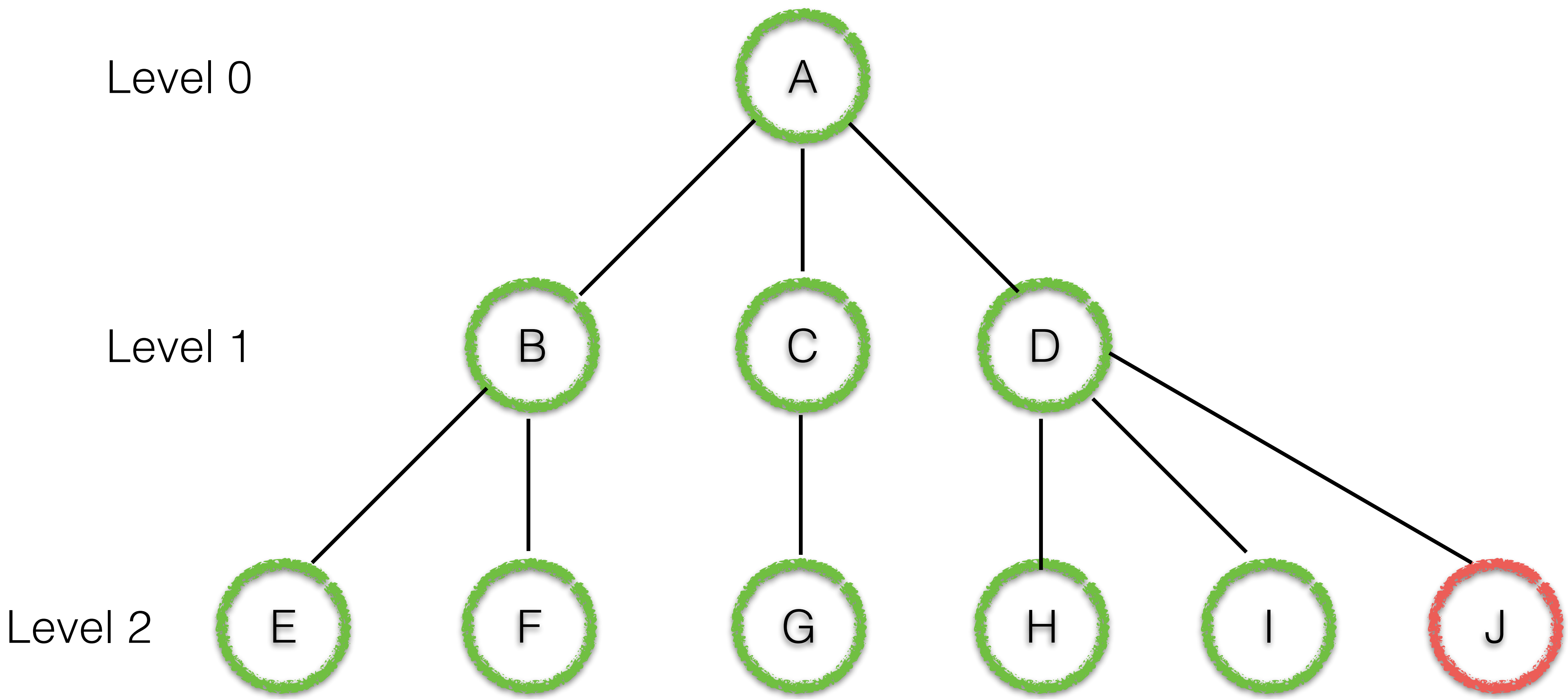


- Different search techniques essentially correspond to different ways of selecting the next node

Breadth-First Search

- Expand the leaf node with the lowest cost path so far
- Add 1 to the path cost for a node to obtain the path cost of each of its children

Breadth-First Search



- Sequence of nodes we expand is: A B C D E F G H I J
- Stop when you expand a node which is a goal node

Breadth-First Search: Algorithm

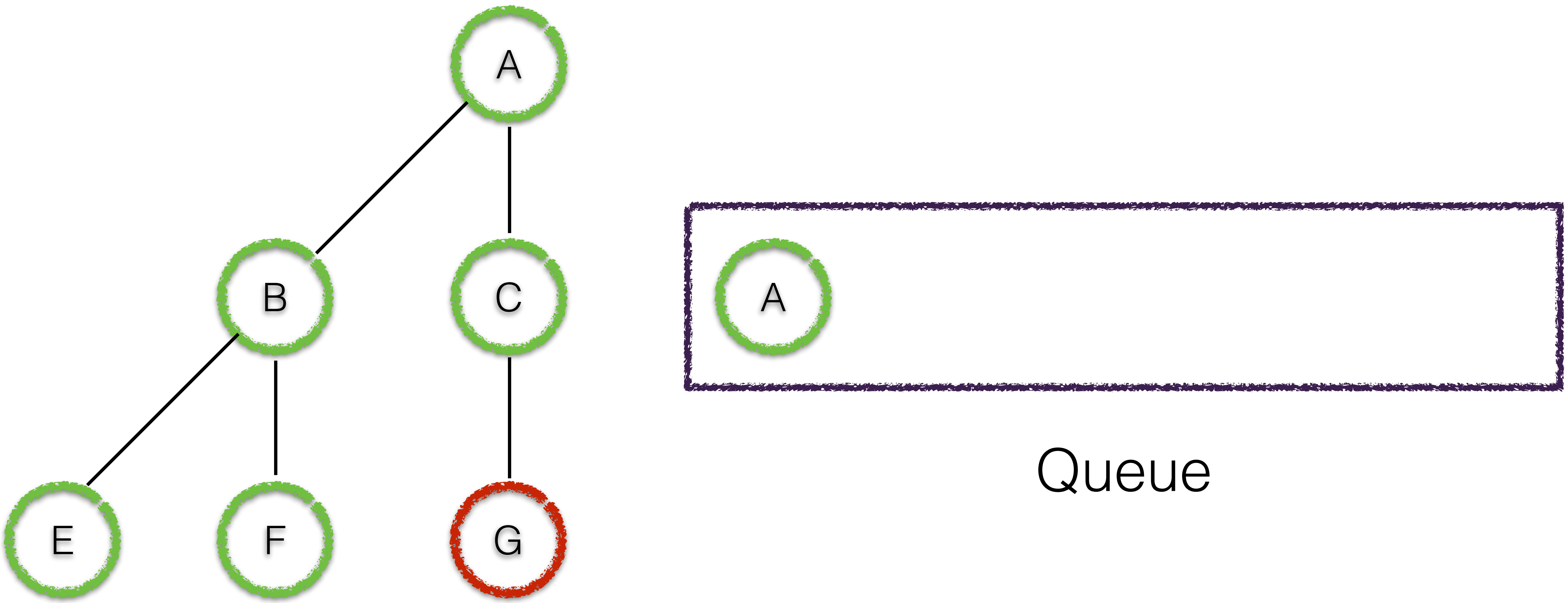
- When doing breadth-first search, a queue is an ideal data structure:
    - Add root node to the queue
    - Dequeue (remove and inspect) first element from queue
    - If it is the goal state: finish!
    - If it isn’t expand node to show it’s children, and add to queue
    - Dequeue first element in queue

Repeat

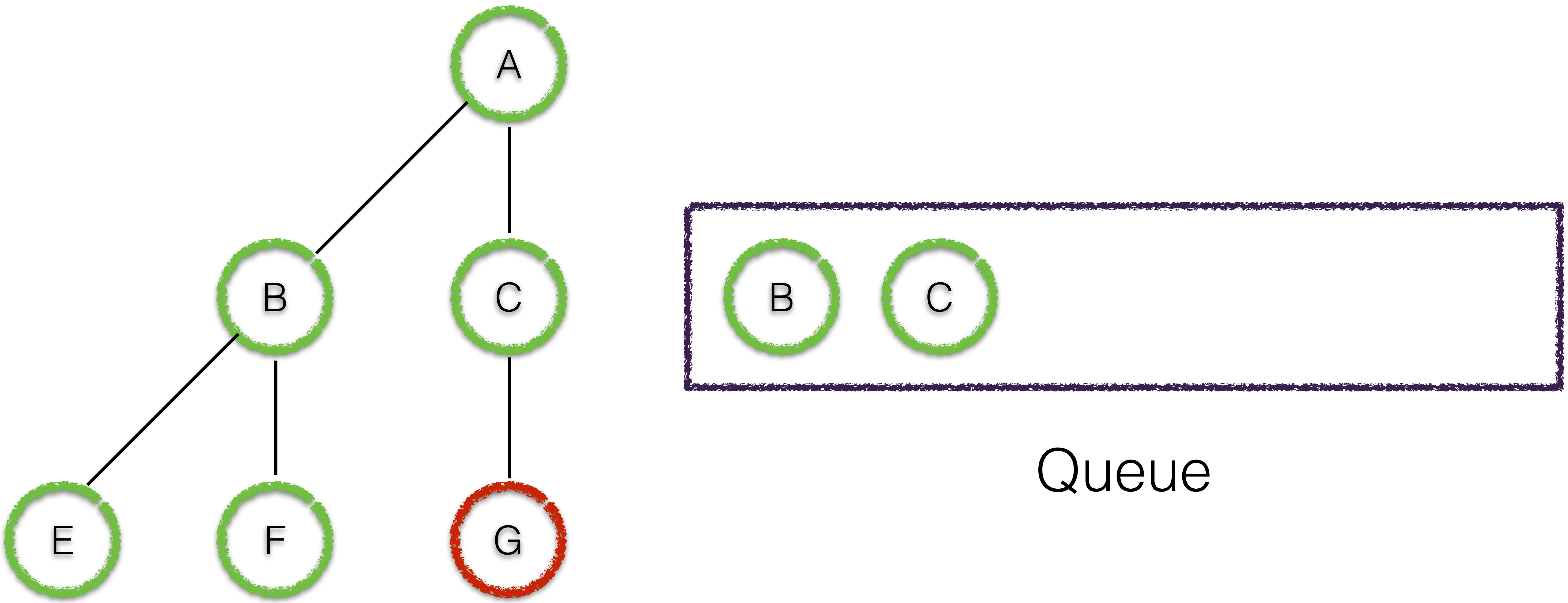
Breadth-First Search: Pseudocode

```
breadth-first-search(Tree):
    get root node r
    create a queue Q
    add r to Q
    while Q is not empty:
        t = Q.dequeue()
        if t is goal:
            return t // goal has been reached
        else:
            for all edges e Tree.adjacentEdges(t)
                V = Tree.adjacentVertices(t, e) //list of child nodes from t
                enqueue V onto Q
```

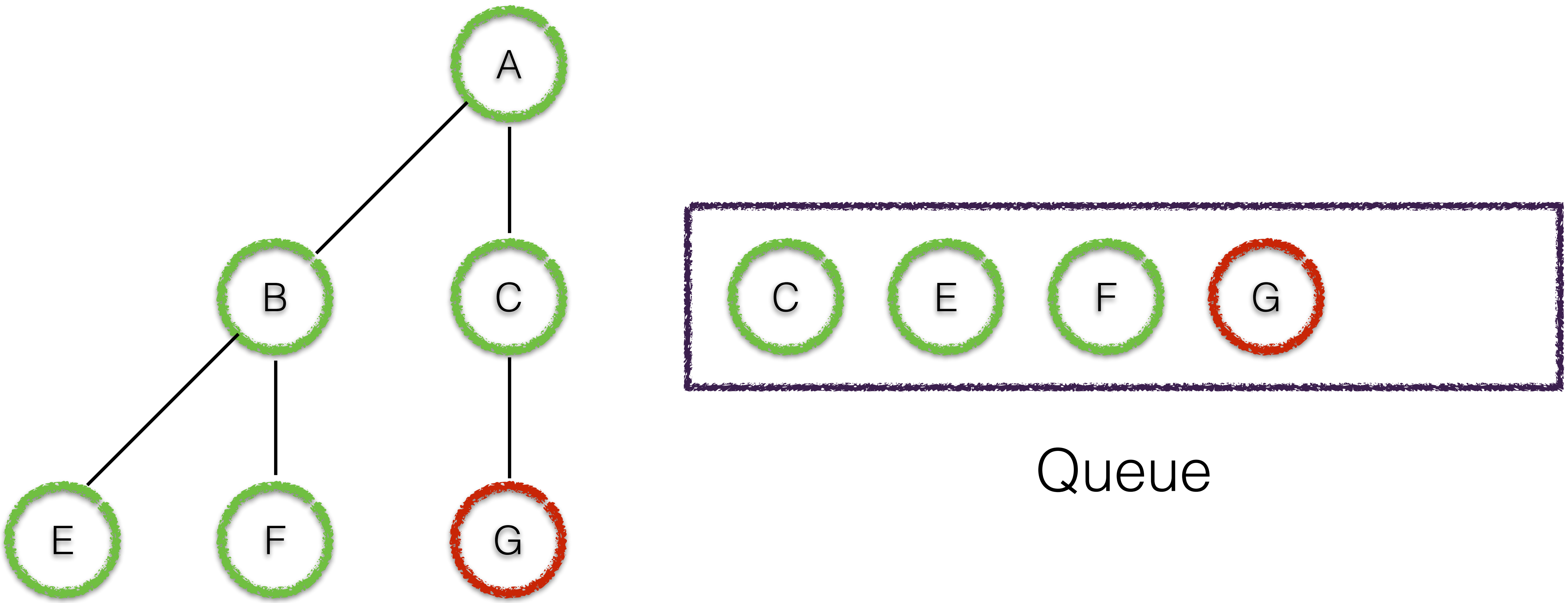
Breadth-First Search: Example with Queue



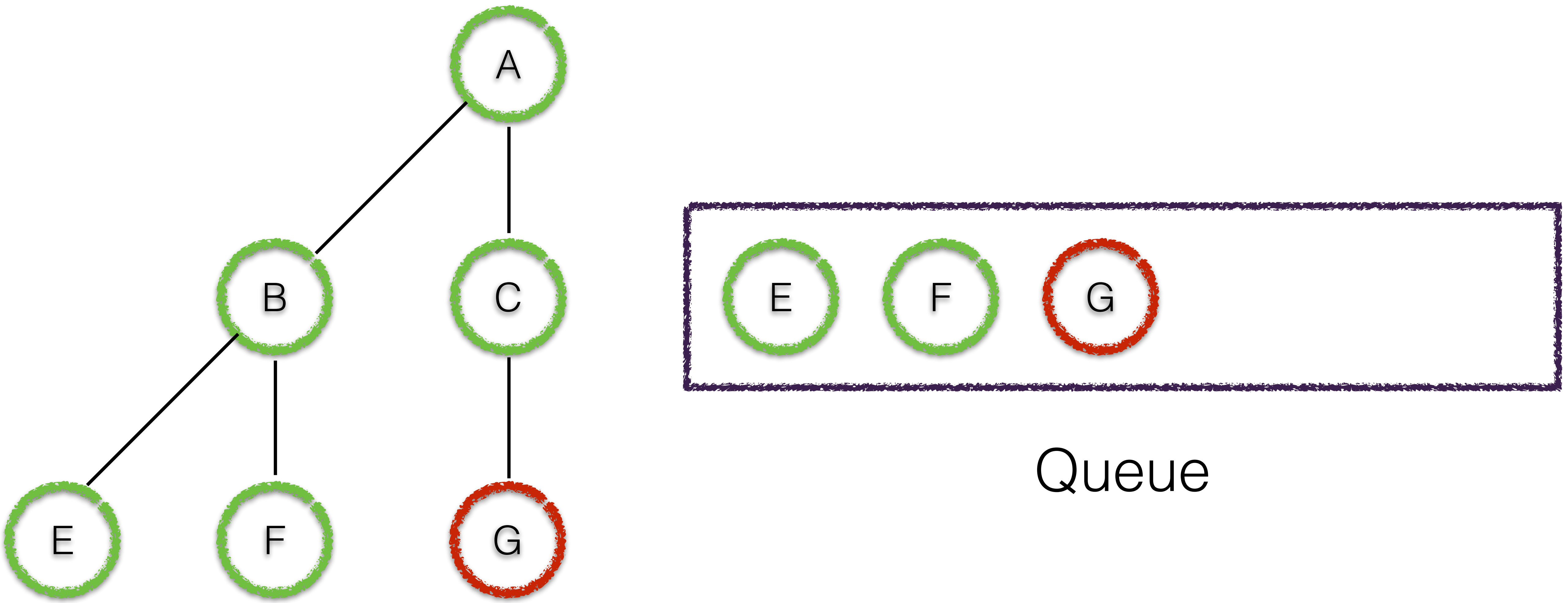
Breadth-First Search: Example with Queue



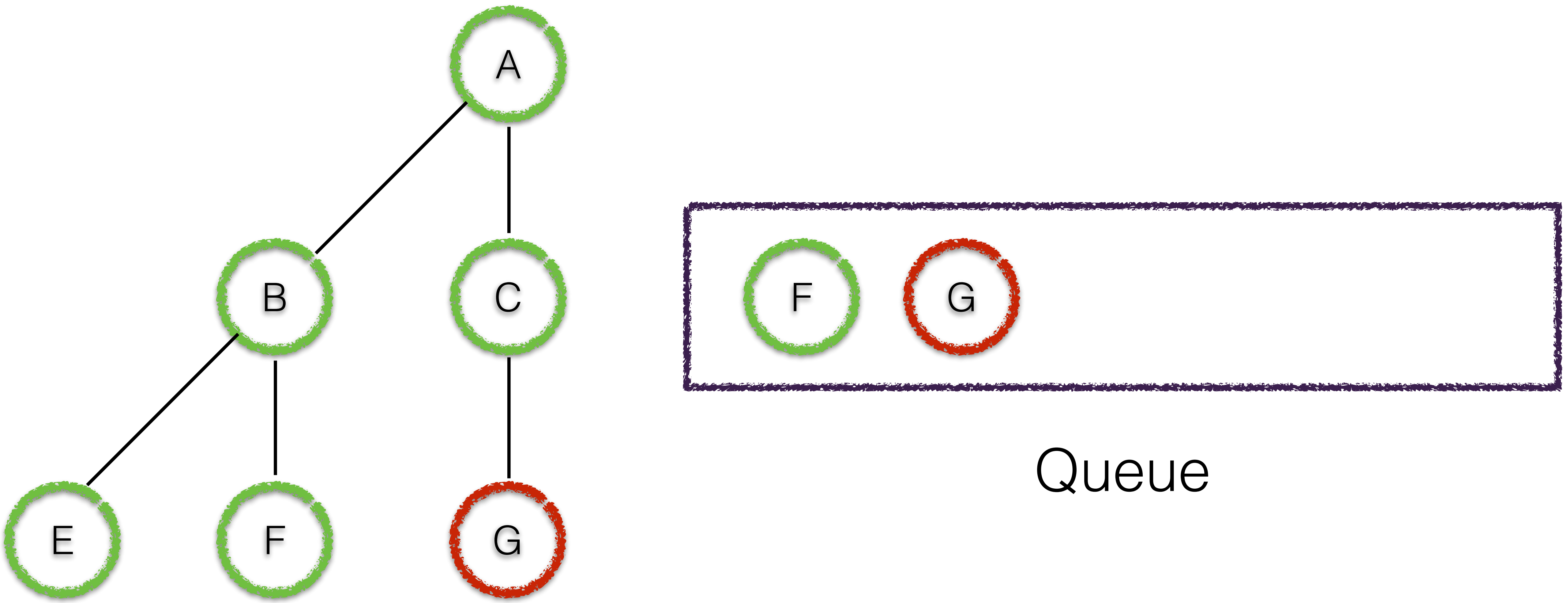
Breadth-First Search: Example with Queue



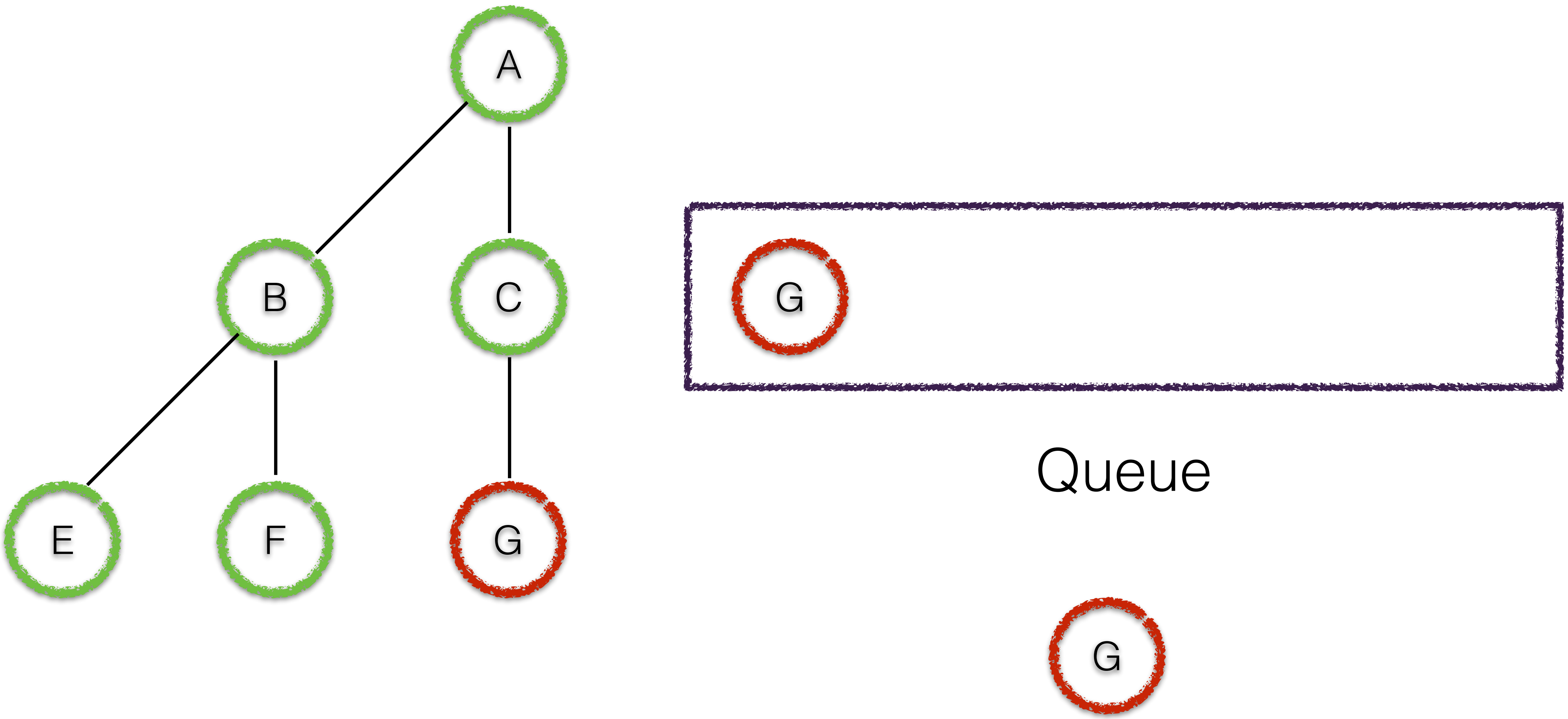
Breadth-First Search: Example with Queue



Breadth-First Search: Example with Queue



Breadth-First Search: Example with Queue



Breadth-First Search: Properties

- Guaranteed to find the shortest path
- Memory intensive if the space is large
  - Space complexity $O(b^d)$
  - Time complexity $O(b^d)$
    - b = branching factor
      - The number of children at each node
      - When not uniform, this can be averaged
    - d = depth of shallowest goal state

Depth	Nodes	Time	Memory
0	1	1 millisecond	100 bytes
2	111	.1 seconds	11 kilobytes
4	11,111	11 seconds	1 megabyte
6	10^6	18 minutes	111 megabytes
8	10^8	31 hours	11 gigabytes
10	10^{10}	128 days	1 terabyte
12	10^{12}	35 years	111 terabytes
14	10^{14}	3500 years	11,111 terabytes

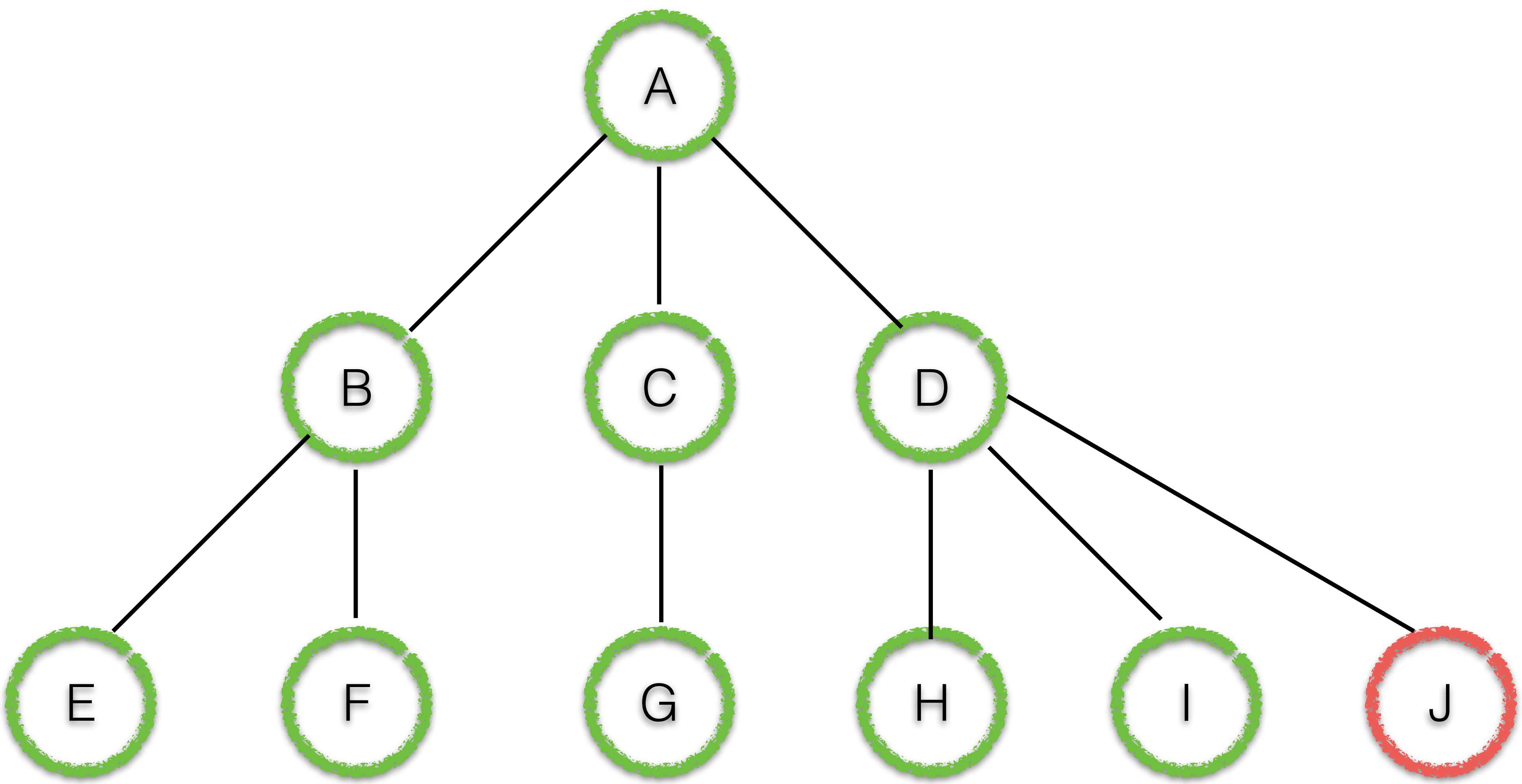
Figure 3.12 Time and memory requirements for breadth-first search. The figures shown assume branching factor b = 10; 1000 nodes/second; 100 bytes/node.

Table from Russell & Norvig (1995), Artificial Intelligence: A Modern Approach

Depth-First Search

- Generate the successors of the leaf-node with the highest cost path so far
- Add 1 to a node’s path cost to obtain the path cost of its children

Depth-First Search



- Sequence of nodes we expand is: A B E F C G D H I J
- Stop when you <u>expand</u> a node which is a goal node

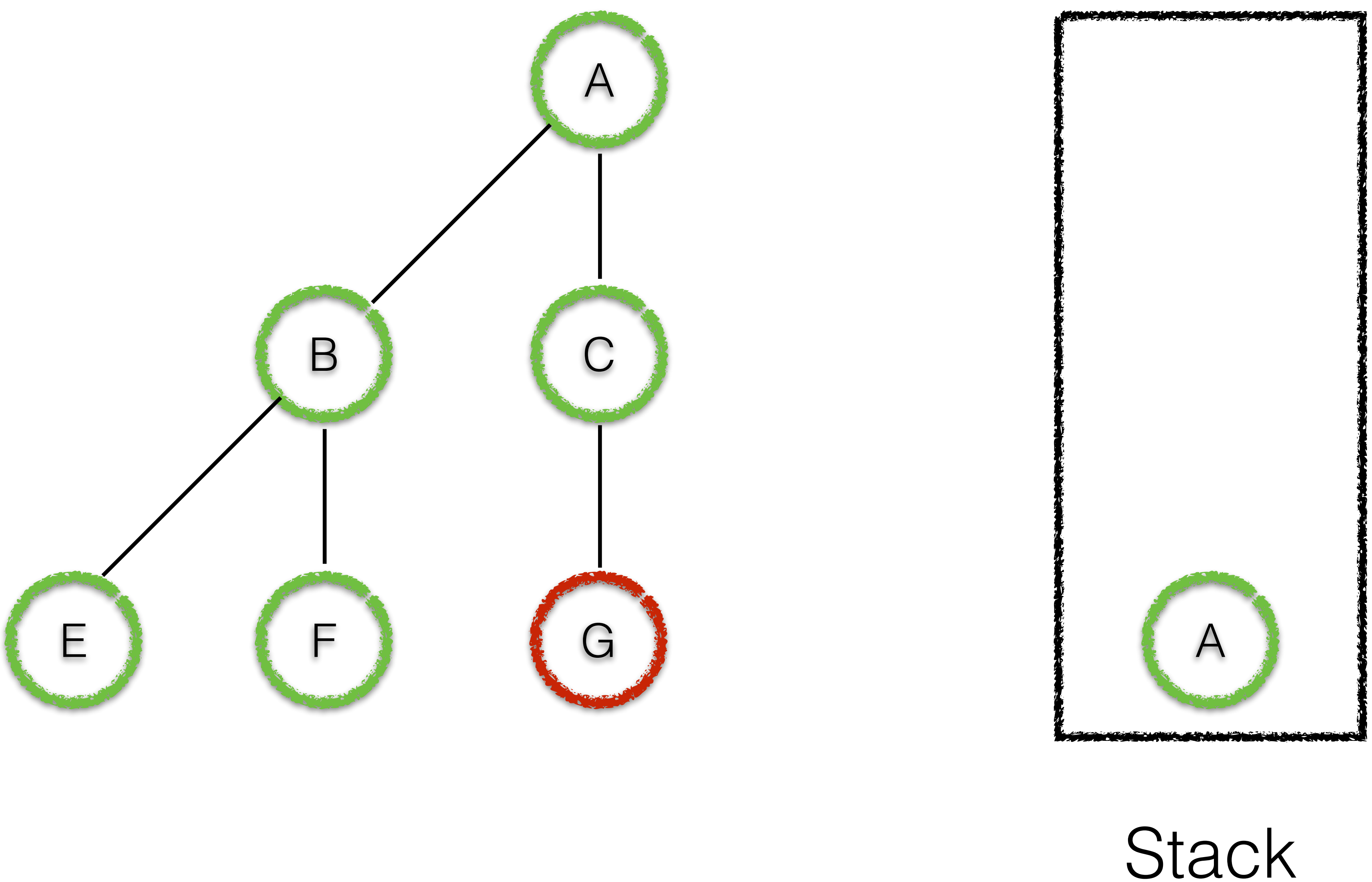
Depth-First Search: Algorithm

- An ideal data structure for depth-first search is a stack
- This adapts the breadth-first search algorithm we saw previously
- The nature of a stack changes the behaviour of the search
- Instead of adding items to examine to the end of a queue we add them to the top of a stack
- We pop the top item on the stack for each iteration of the algorithm

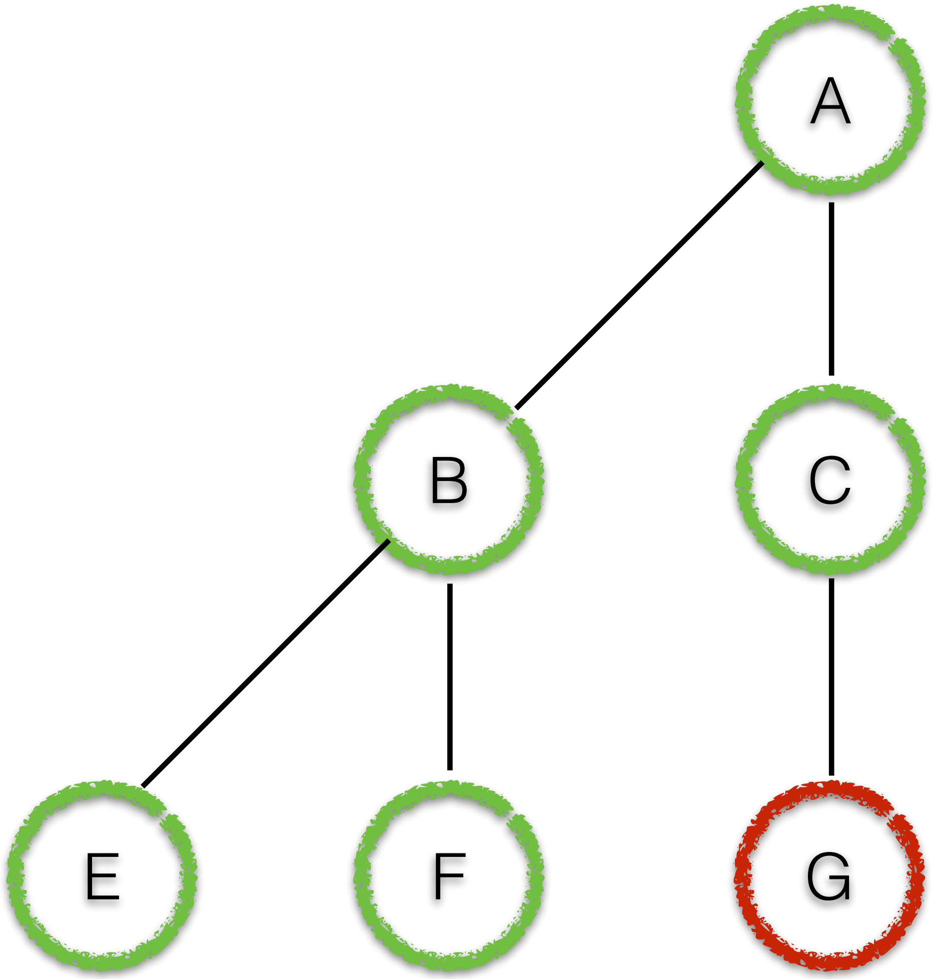
Depth-First Search: Pseudocode

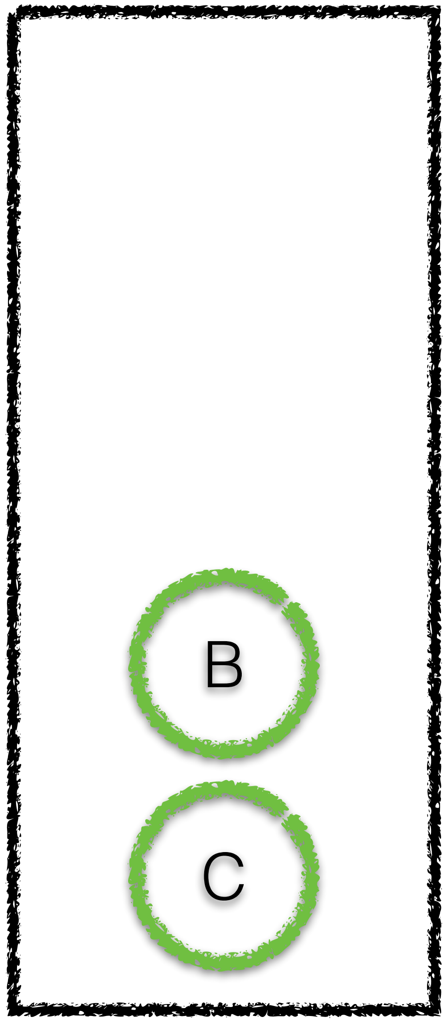
```
depth-first-search(Tree):
    get root node r
    create a stack S
    push r to S
    while S is not empty:
        t = S.pop()
        if t is goal:
            return t // goal has been reached
        else:
            for all edges e Tree.adjacentEdges(t)
                V = Tree.adjacentVertices(t, e) //list of child nodes from t
                push V onto S
```

Depth-First Search: Example with Stack



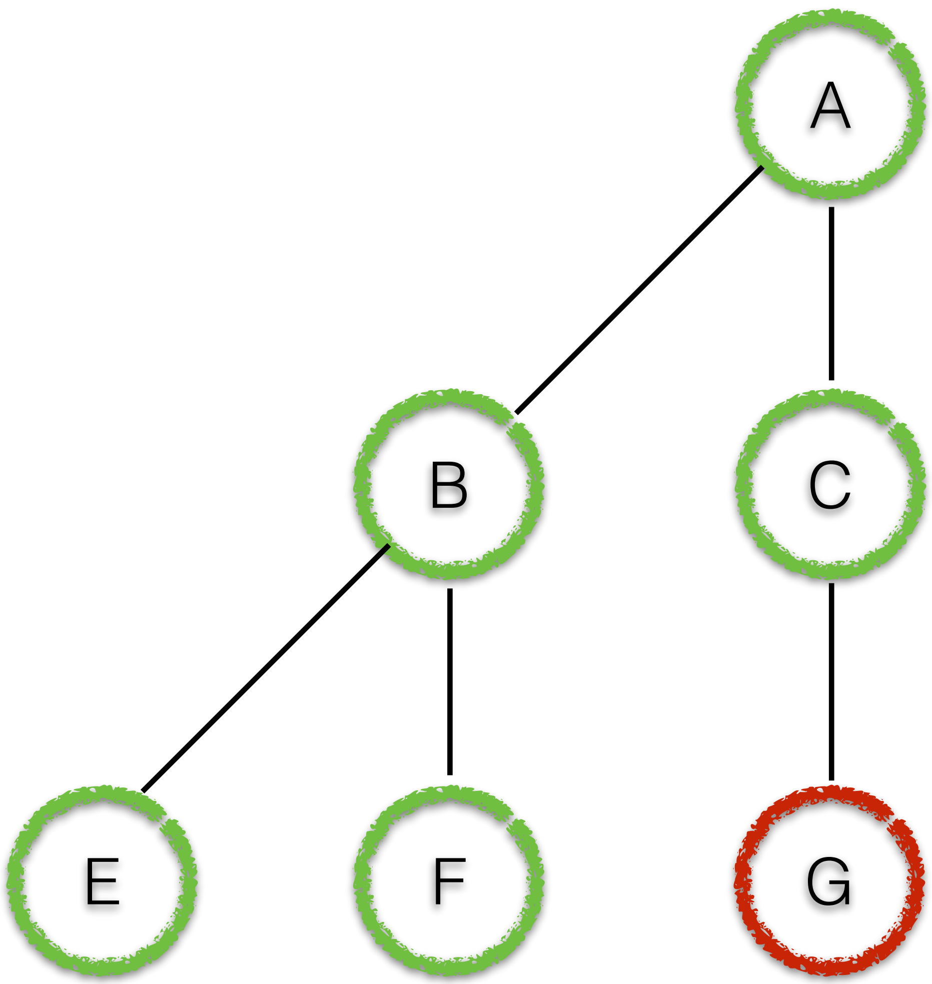
Depth-First Search: Example with Stack

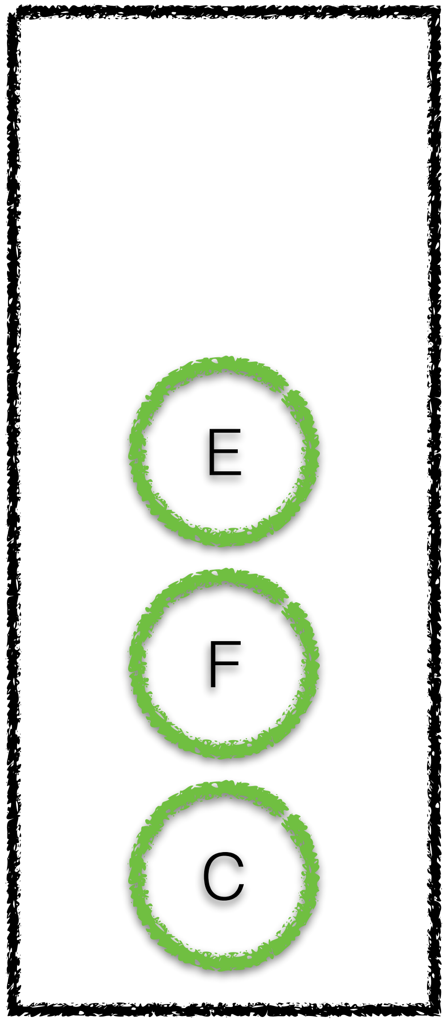




Stack

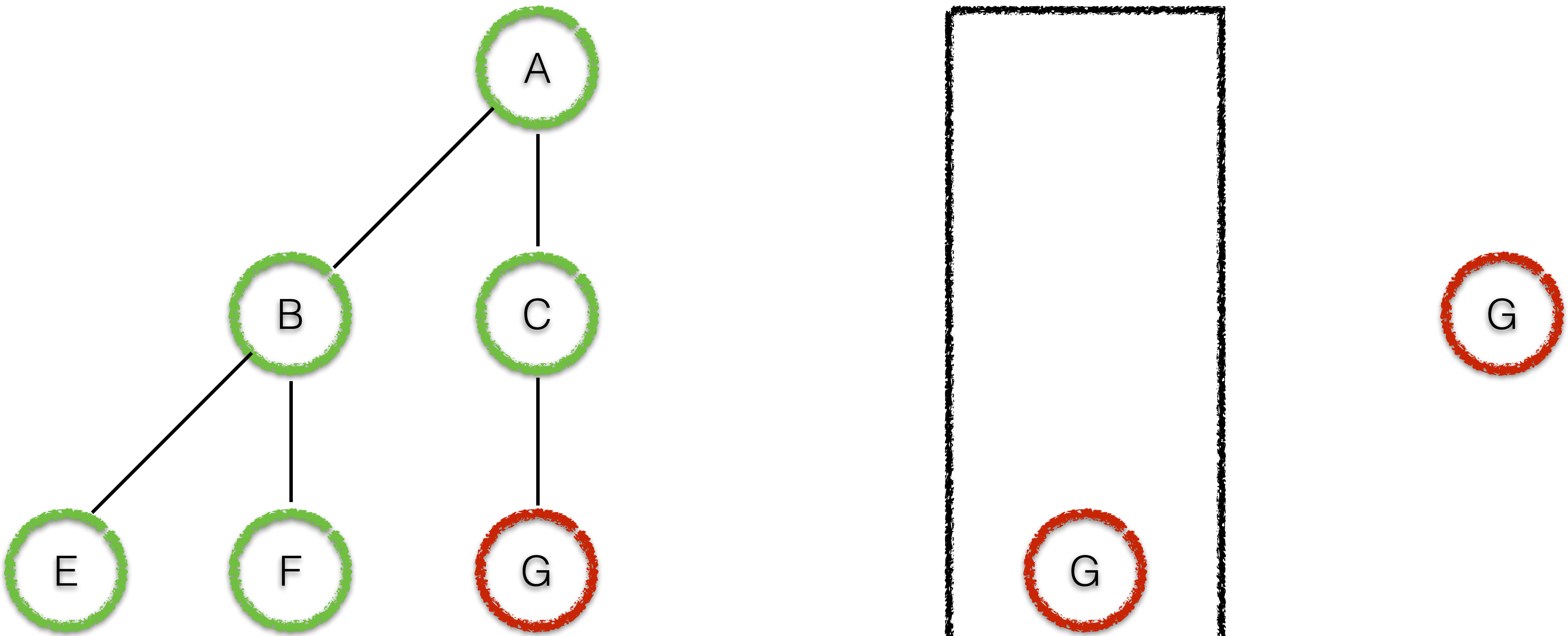
Depth-First Search: Example with Stack





Stack

Depth-First Search: Example with Stack



Stack

Depth-First Search: Properties

- Not guaranteed to find <u>any</u> path to a goal state
- Memory efficient
  - Space complexity ≈ O(bm)
  - Time complexity ≈ O(b^m)
    - m = maximum depth of search tree (can't be ∞)

Depth-Limited Search

- To guarantee that search will terminate (either in failure or success) we can put a limit on how deep DFS searches
- Depth-limited search does DFS to a depth limit *h*
- If goal's depth $\leq$ h then DLS is complete (guaranteed to find the solution
- Still not guaranteed to find the shortest path
  - Space complexity O(bd)
  - Time complexity O(b^d)

Depth-First Iterative Deepening

- Extends the idea of depth-limited search
- Start by doing DLS with h = 1
- Then we reset:
  - OPEN = [initial-state]
  - CLOSED = []
  - Increase h by 1
  - Repeat DLS with new limit
  - Iterate, increasing h by 1 each time

Depth-First Iterative Search

- Looks wasteful
  - However, is better than either BFS or DFS
- Although it always expands many nodes more than once, it still spends most of its time at the bottom level

Depth-First Iterative Deepening: Explanation

- At depth d there are b^d nodes
- Total nodes to depth d in DLS is:
  - $1 + b + b^2 + b^3 + \ldots + b^{d-1} + b^d$
- The total number of expansions after d iterations will be:
  - $(d+1)\,1 + (d)\,b + (d-1)\,b^2 + \ldots (2)\,b^{d-1} + (1)\,b^d$
- The sum of the first d expansions will be insignificant compared to b^d
  - e.g $b = 10$ $d = 5$
  - $6 + 50 + 400 + 3000 + 20000 + 100000 = 123{,}456$
  - So time complexity is $O(b^d)$
    - Same as BFS, better than DFS

Depth-First Iterative Deepening: Explanation

- As it's doing depth-first search only one path is maintained. Therefore the space complexity is the same as for DFS: O(bd)

- Finally, because all the nodes are expanded at each level DFID is complete (like DLS)

- As the limit is increased by 1 each iteration the algorithm is guaranteed to find the shortest path to the GOAL first, so it is optimal.

- Curiously, DFID is the best uninformed search algorithm in all respects

Analysing Search Algorithms

- Clearly the performance of any algorithm on a particular problem depends on properties of the problem domain, and of the representation you choose
- But, we can place some general bounds on the performance of algorithms too

Analysing Search Algorithms

- **Completeness -** A search algorithm is complete if it is guaranteed to find a solution when at least one solution exists
- **Optimality -** A search algorithm is optimal if it is <u>guaranteed</u> to find the <u>best</u> solution when there is more than one
- **Space Complexity -** The order of storage space required at any point during the search process, in order to find a solution in the worst case (number of nodes we must store)
- **Time Complexity -** The order of computation required during the search process to find a solution in the worst case (number of expansions)

Comparing Uninformed Search Algorithms

Strategy	Complete?	Optimal?	Time Complexity	Space Complexity
BFS	Yes	Yes	$O(b^d)$	$O(b^d)$
DFS	No	No	$O(b^m)$	$O(bm)$
DLS	Yes if $h \le d$	No	$O(b^h)$	$O(bh)$
DFID	Yes	Yes	$O(b^d)$	$O(bd)$

d = depth of shallowest goal state
m = maximum depth of search tree (could be ∞)
h = user defined limit on search

Summary of Uninformed Search Algorithms

- Uninformed Search - sometimes called blind search
- <u>Systematic</u> search with no information about the current distance (cost) to the goal
- So far we have seen
- Breadth-first: guaranteed to find the shallowest goal state in the search tree, but very expensive w.r.t space and time
- Depth-first: Less storage space required than BFS, but no guarantees, and worst case time complexity is poorer
- Depth-limited: Weak guarantee of completeness. Known bound on time complexity and good space complexity
- DFID: The best of a bad bunch. Low storage space, complete and optimal, however exponential time complexity