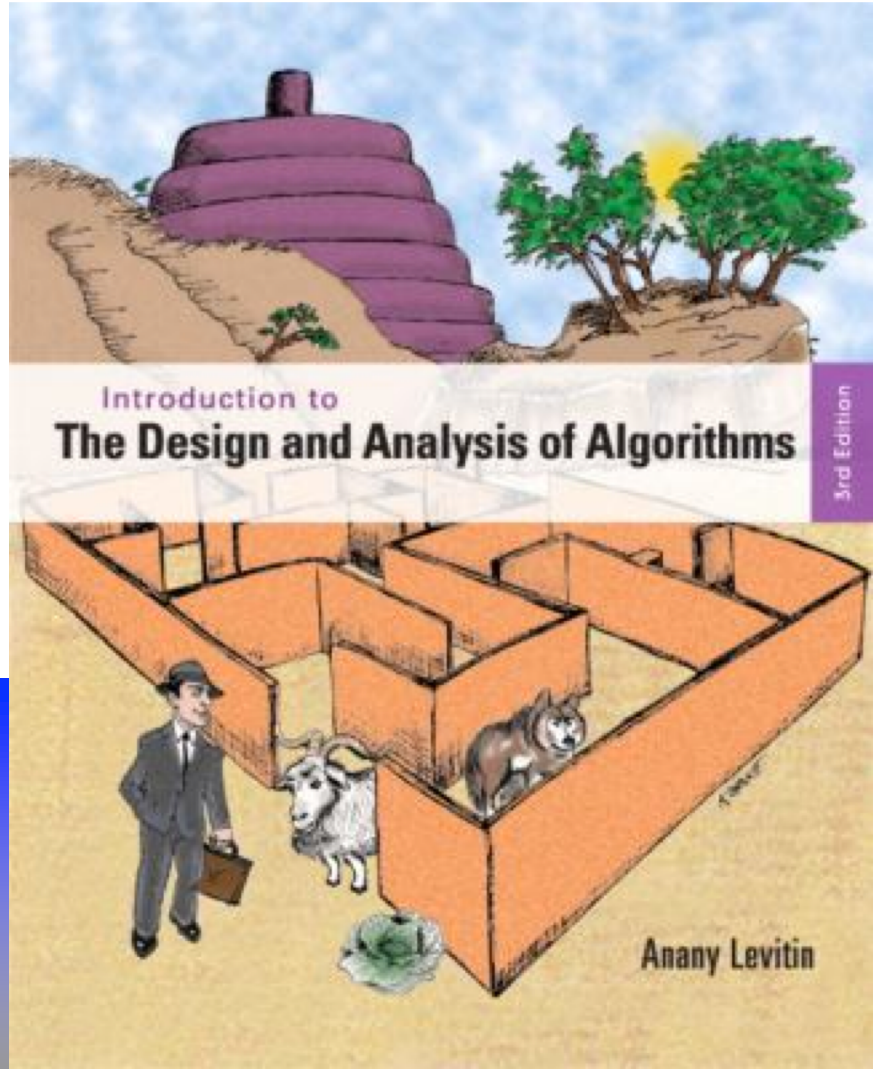
Chapter 4

Decrease-and-Conquer



Decrease-and-Conquer

1. Reduce problem instance to smaller instance of the same problem

2. Solve smaller instance

3. Extend solution of smaller instance to obtain solution to original instance

- Can be implemented either top-down or bottom-up
- Also referred to as *inductive* or *incremental* approach

3 Types of Decrease and Conquer

- ***Decrease by a constant*** **(usually by 1):**
  - graph traversal algorithms (BFS and DFS)
  - insertion sort
  - topological sorting
  - algorithms for generating permutations, subsets

- *Decrease by a constant factor* (usually by half)
  - binary search and bisection method
  - exponentiation by squaring

- *Variable-size decrease*
  - Euclid's algorithm
  - selection by partition
  - Nim-like games

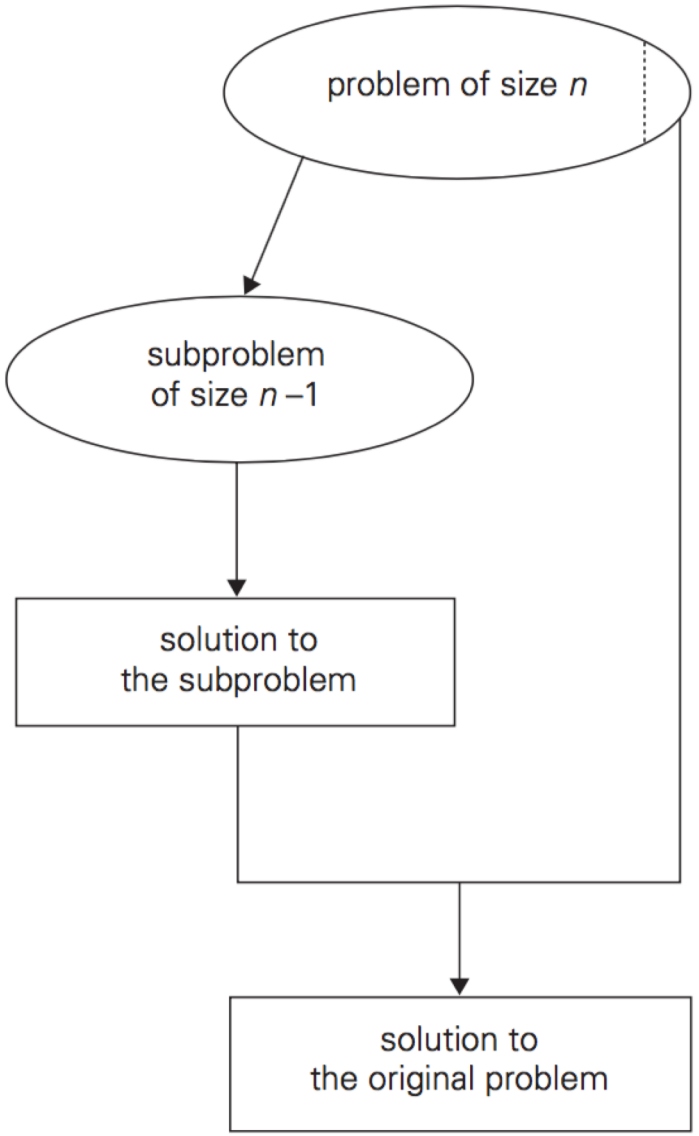


FIGURE 4.1 Decrease-(by one)-and-conquer technique.

3 Types of Decrease and Conquer

- *Decrease by a constant* (usually by 1):
  - graph traversal algorithms (BFS and DFS)
  - insertion sort
  - topological sorting
  - algorithms for generating permutations, subsets

- ***Decrease by a constant factor*** **(usually by half)**
  - binary search and bisection method
  - exponentiation by squaring

- *Variable-size decrease*
  - Euclid's algorithm
  - selection by partition
  - Nim-like games

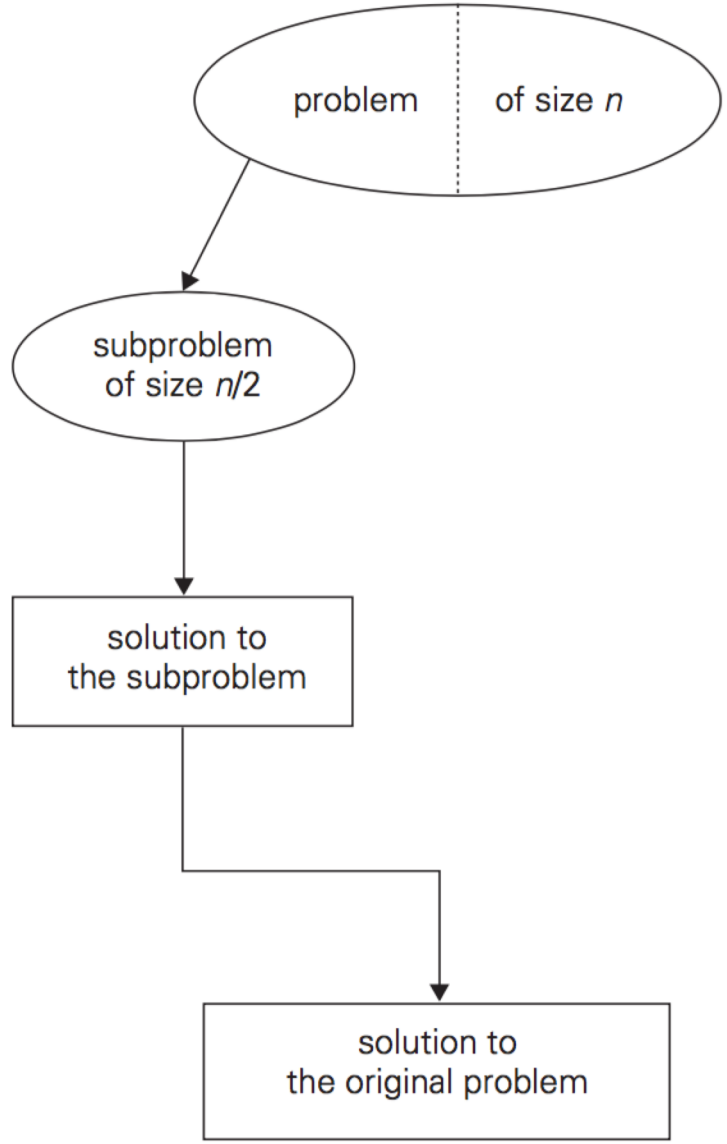


FIGURE 4.2 Decrease-(by half)-and-conquer technique.

Decrease by a Constant

- graph traversal algorithms (BFS and DFS)
- insertion sort
- topological sorting
- algorithms for generating permutations, subsets

Some Definitions

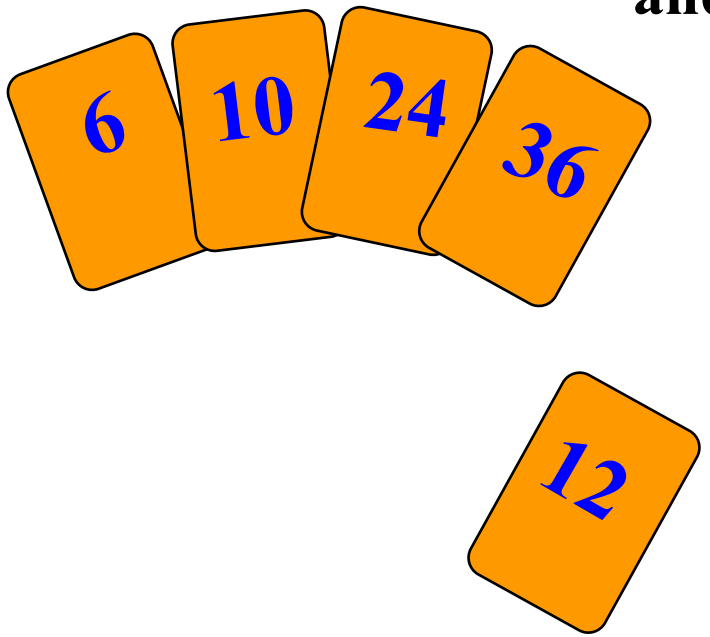
- In Place Sort
  - The amount of extra space required to sort the data is constant with the input size.
- A Stable Sort preserves relative order of records with equal keys
- Internal Sort
  - The data to be sorted is all stored in the computer's main memory.
- External Sort
  - Some of the data to be sorted might be stored in some external, slower, device.

Insertion Sort

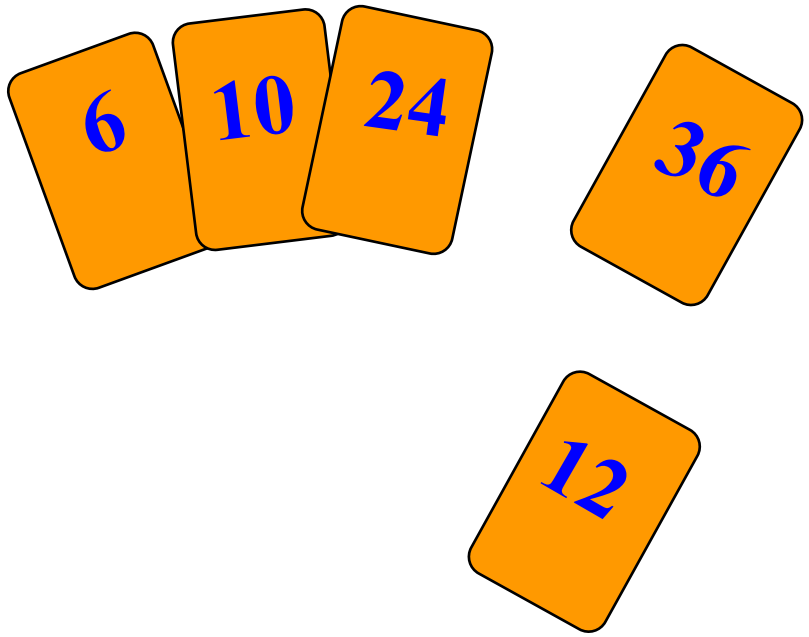
- Idea: like sorting a hand of playing cards
  - Start with an empty left hand and the cards facing down on the table.
  - Remove one card at a time from the table, and insert it into the correct position in the left hand
    - compare it with each of the cards already in the hand, from right to left
  - The cards held in the left hand are sorted
    - these cards were originally the top cards of the pile on the table

Insertion Sort

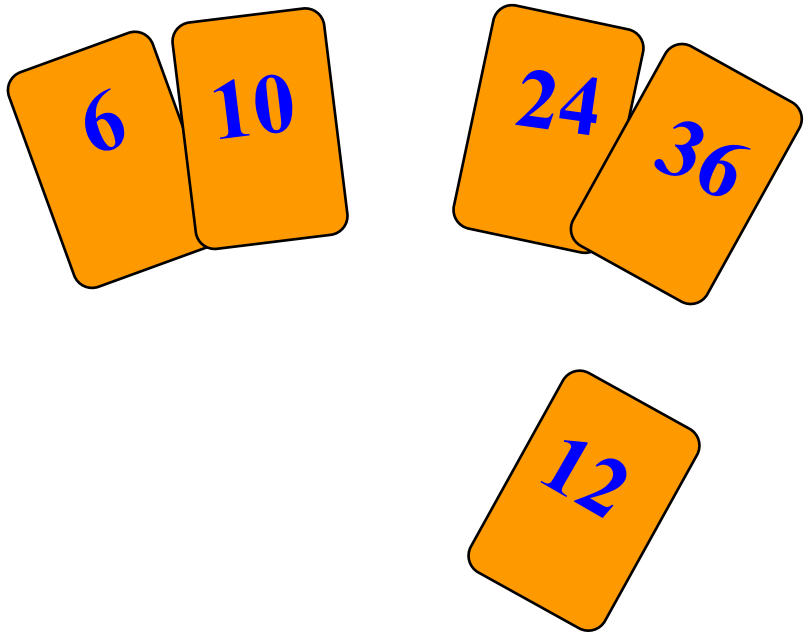
To insert 12, we need to make room for it by moving first 36 and then 24.



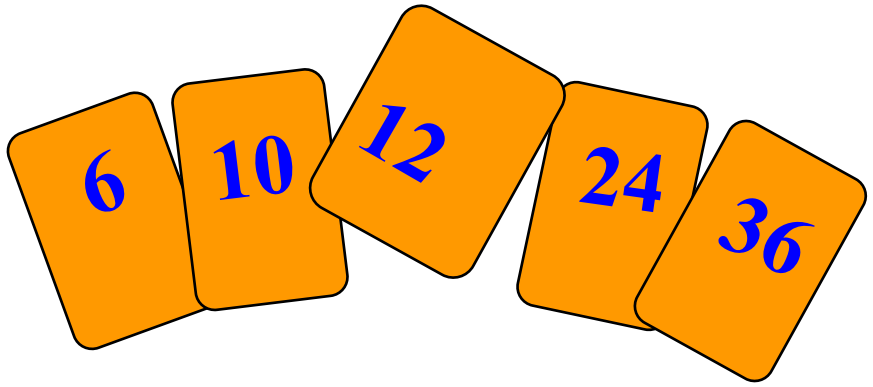
Insertion Sort



Insertion Sort



Insertion Sort



Insertion Sort

- We want to sort array $A[0..n\text{-}1]$

- $A[i]$ is inserted in its appropriate place among the first i elements of A that have been already sorted

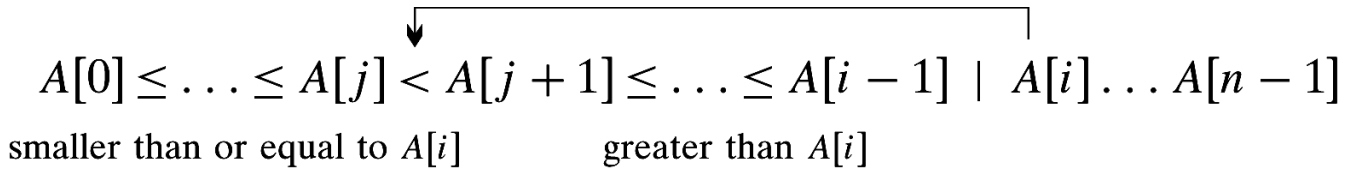


FIGURE 5.3 Iteration of insertion sort: $A[i]$ is inserted in its proper position among the preceding elements previously sorted.

89 |	**45**	68	90	29	34	17
45	89 |	**68**	90	29	34	17
45	68	89 |	**90**	29	34	17
45	68	89	90 |	**29**	34	17
29	45	68	89	90 |	**34**	17
29	34	45	68	89	90 |	**17**
17	29	34	45	68	89	90

FIGURE 5.4 Example of sorting with insertion sort. A vertical bar separates the sorted part of the array from the remaining elements; the element being inserted is in bold.

Insertion Sort Algorithm

ALGORITHM *InsertionSort*$(A[0..n-1])$

```
//Sorts a given array by insertion sort
//Input: An array A[0..n − 1] of n orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
for i ← 1 to n − 1 do
    v ← A[i]
    j ← i − 1
    while j ≥ 0 and A[j] > v do
        A[j + 1] ← A[j]
        j ← j − 1
    A[j + 1] ← v
```

Analysis of Insertion Sort

What is the Worst Case Scenario?

- $O(n)$ comparisons per pass
- $O(n)$ passes required (n-1 to be exact)
- Complexity: $O(n)$ passes * $O(n)$ comparisons = $O(n^2)$

What is the Best Case Scenario?

- $O(?)$ comparisons per pass
- $O(?)$ passes required
- Complexity: $O(?)$ passes * $O(?)$ comparisons = $O(?)$

Space efficiency:

Stability:

Analysis of Insertion Sort

What is the Worst Case Scenario?

- $O(n)$ comparisons per pass
- $O(n)$ passes required (n-1 to be exact)
- Complexity: $O(n)$ passes * $O(n)$ comparisons = $O(n^2)$

What is the Best Case Scenario?

- $O(?)$ comparisons per pass
- $O(?)$ passes required
- Complexity: $O(?)$ passes * $O(?)$ comparisons = $O(?)$

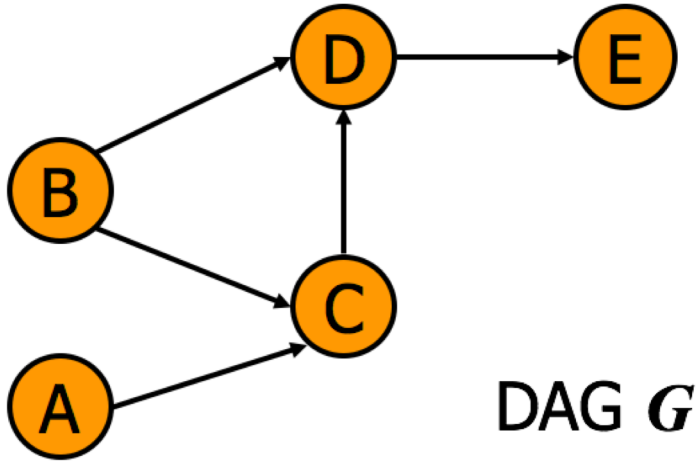
Space efficiency: ***in-place***

Stability: ***yes***

Best ***elementary*** **sorting algorithm overall**

Property of DAGs

- ***Degree*** of a vertex is the number of edges adjacent to the vertex, with loops counted twice.
- ***Outdegree*** of a vertex, the number of tail ends adjacent to the vertex.
- ***Indegree*** of a vertex, the number of head ends adjacent to the vertex



- A directed acyclic graph (DAG) is a digraph that has no directed cycles
- Prove: A DAG has at least one vertex with in-degree 0 and one vertex with out-degree 0.

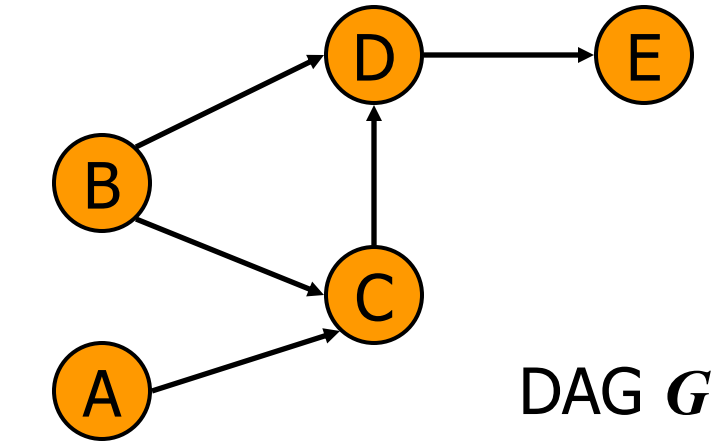
DAGs and Topological Ordering

- A topological ordering of a digraph is a numbering

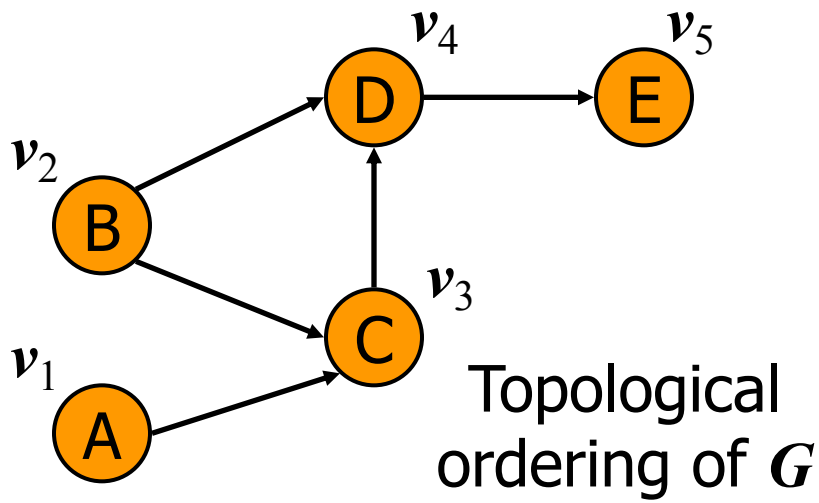
  $v_1, \ldots, v_n$

  of the vertices such that for every edge (v_i, v_j), we have $i < j$

- Example: in a task scheduling digraph, a topological ordering is a task sequence that satisfies the precedence constraints



DAG G



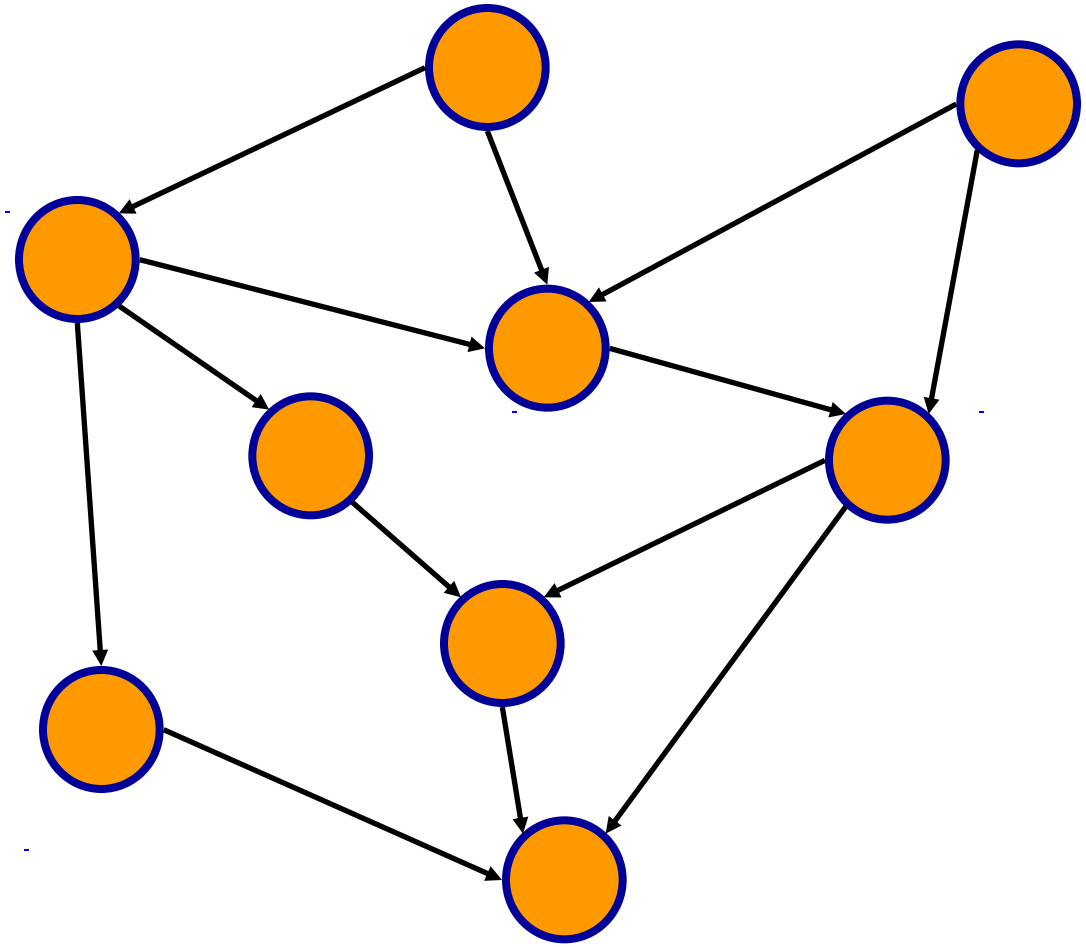
Topological ordering of G

Naïve Algorithm for Topological Sorting

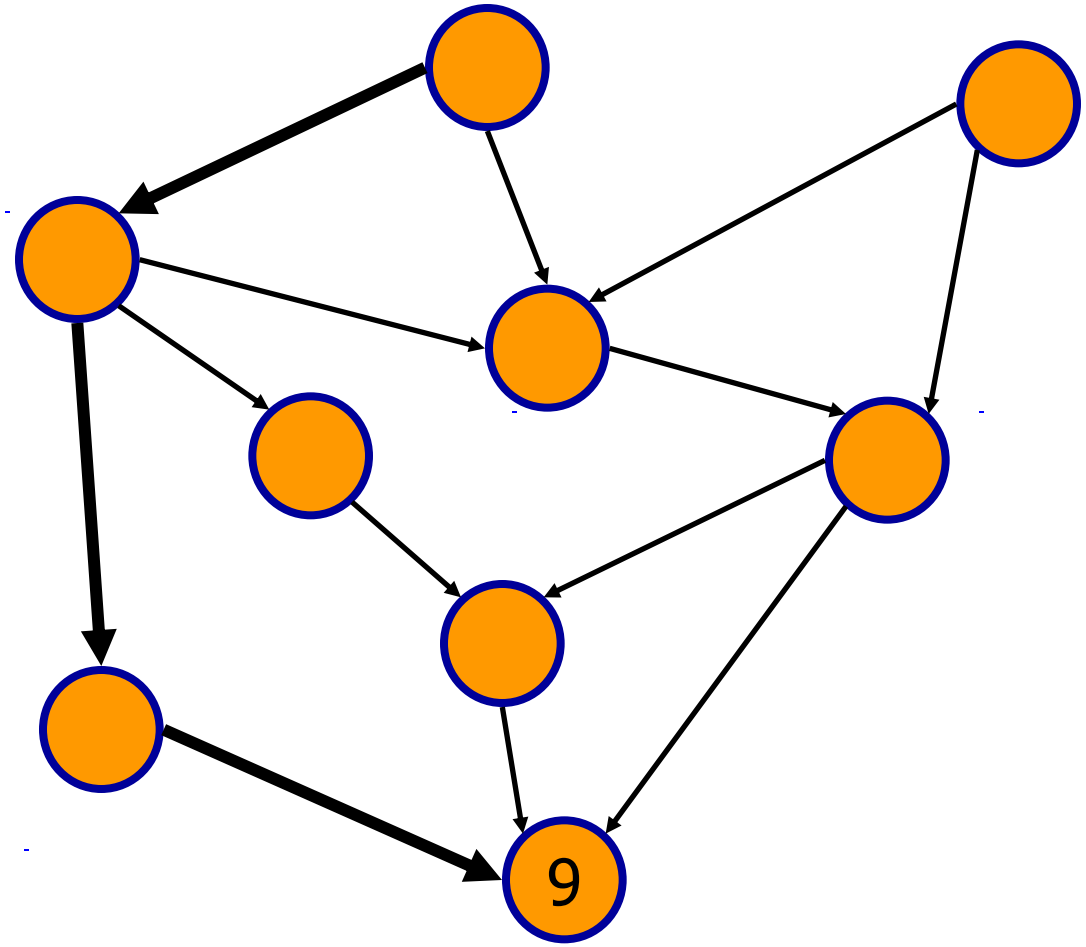
```
Algorithm TopologicalSort(G)
    H ← G          // Temporary copy of G
    n ← G.numVertices()
    while H is not empty do
        v ← find vertex with no outgoing edges
        Label v ← n
        n ← n - 1
        Remove v from H
```

- Whenever there is a choice between multiple vertices with no outgoing edges, the output is not unique.

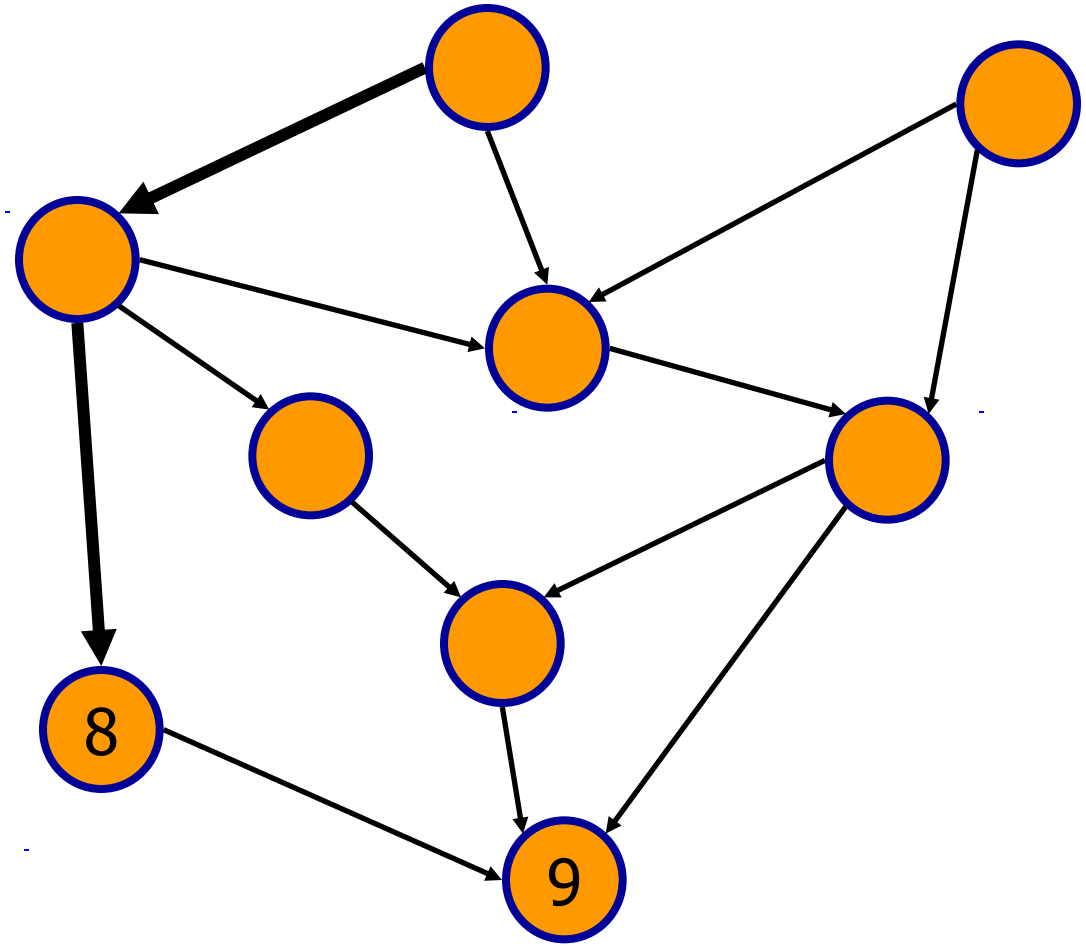
Topological Sorting Example



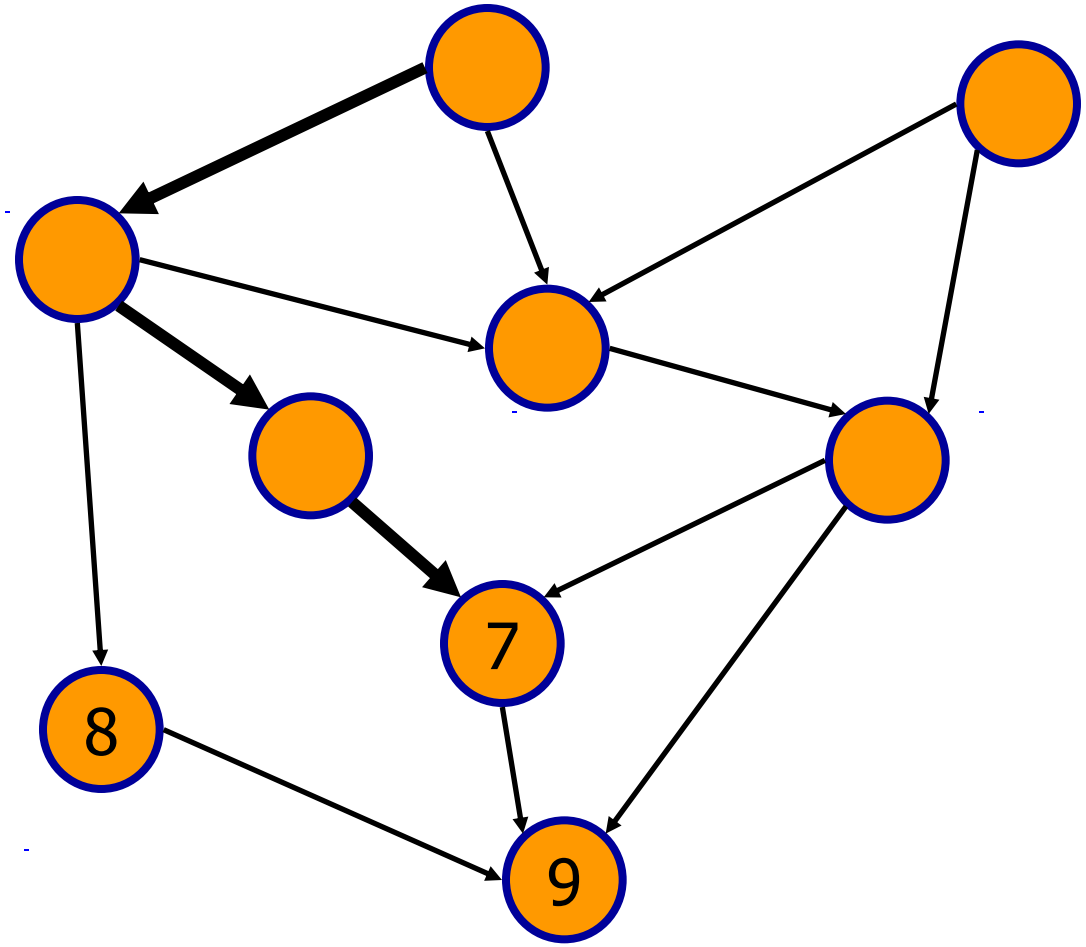
Topological Sorting Example



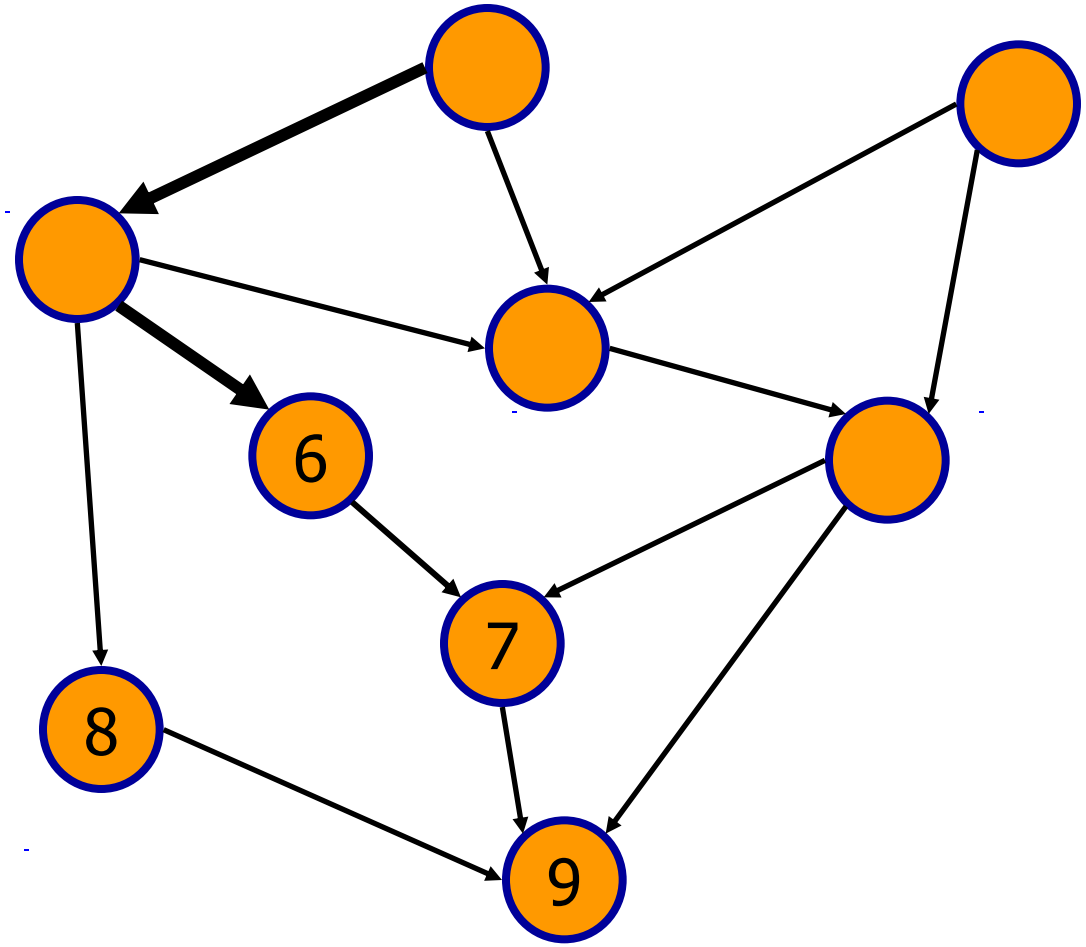
Topological Sorting Example



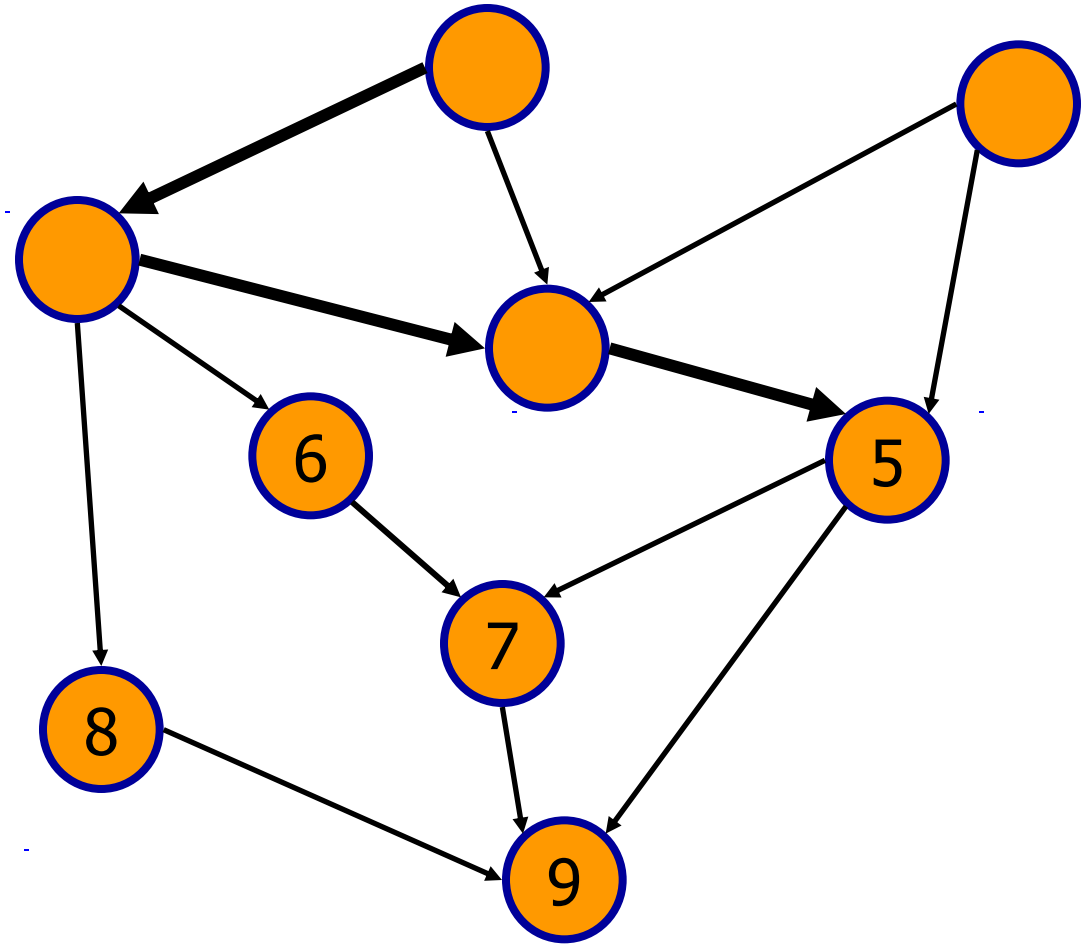
Topological Sorting Example



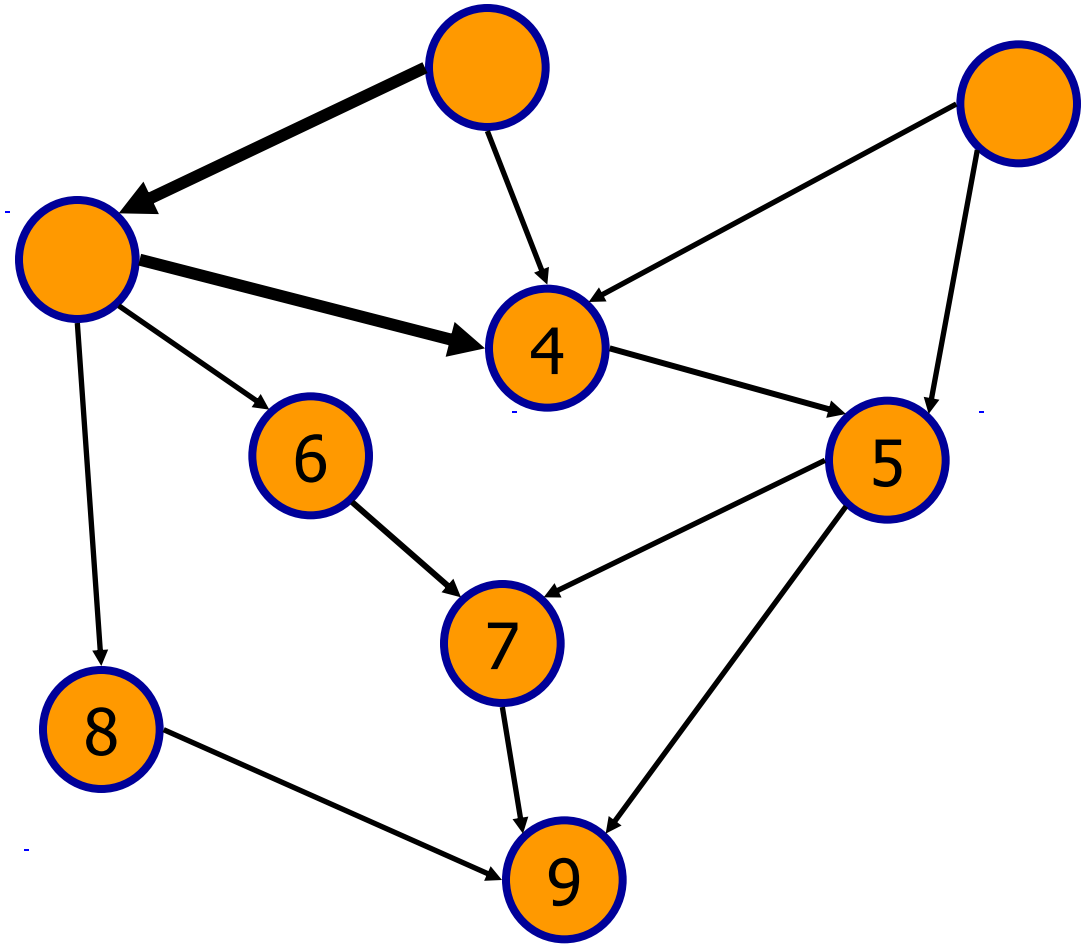
Topological Sorting Example



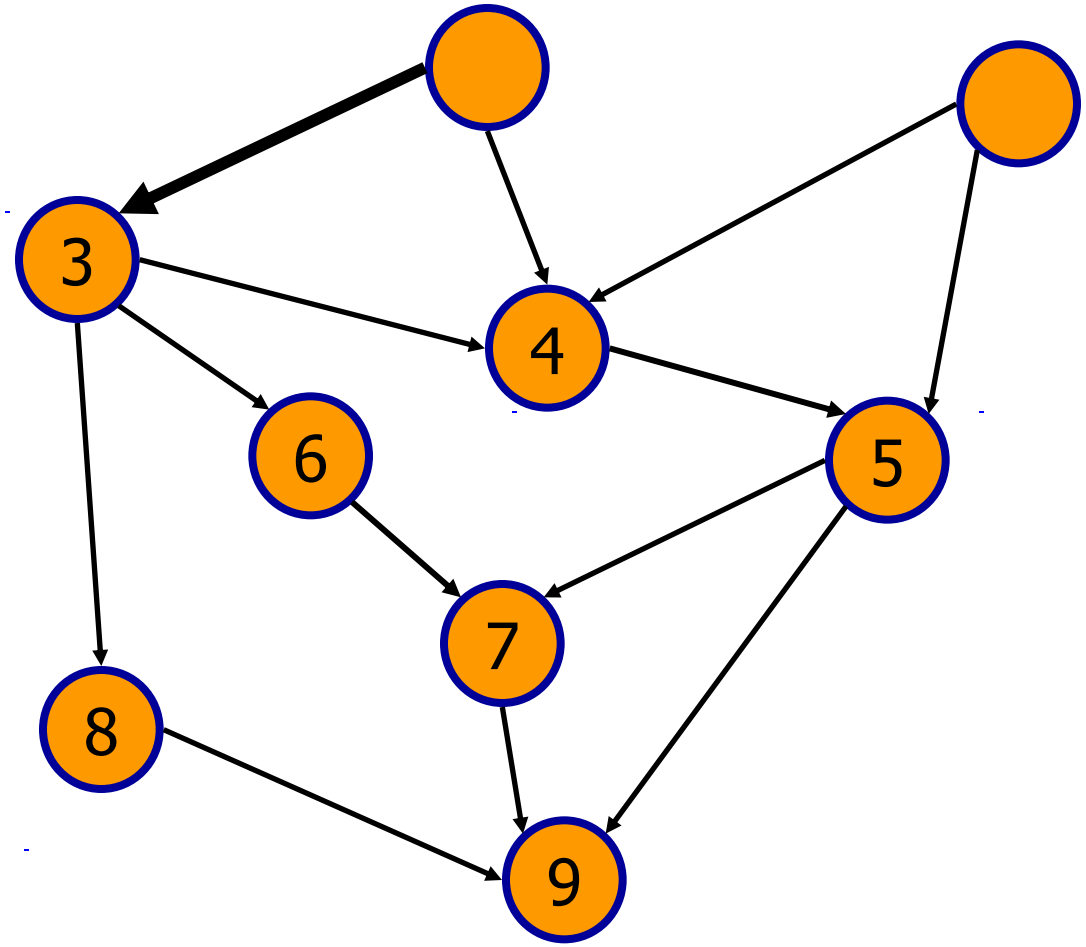
Topological Sorting Example



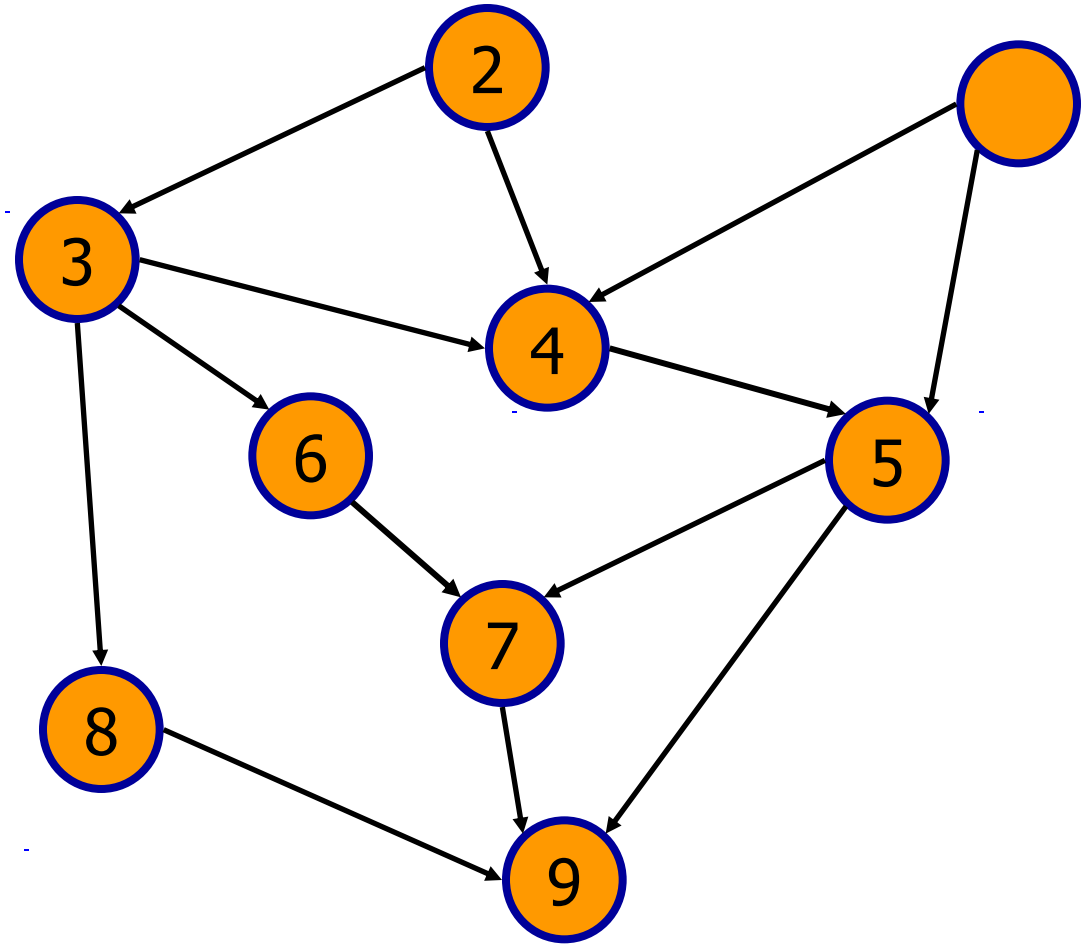
Topological Sorting Example



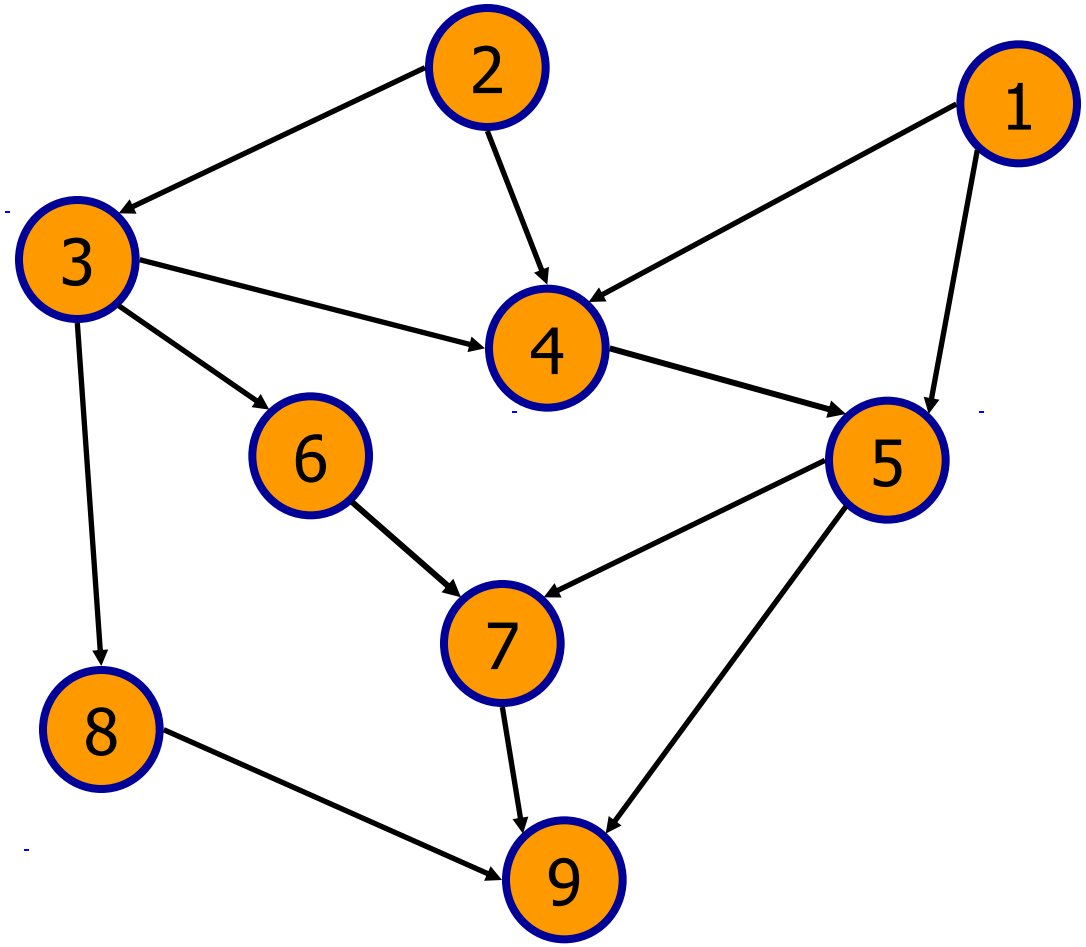
Topological Sorting Example



Topological Sorting Example

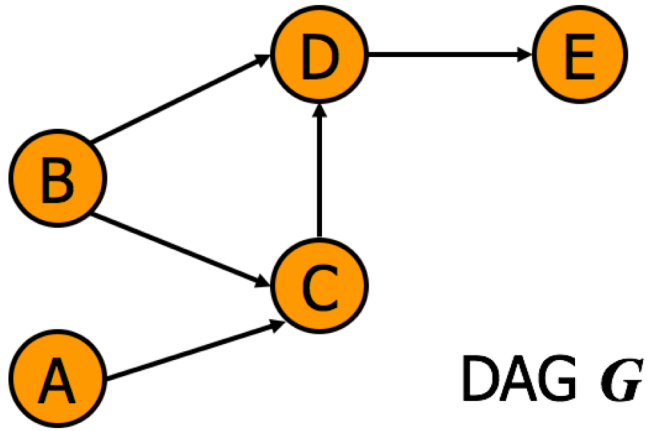


Topological Sorting Example



Naïve Algorithm for Topological Sorting

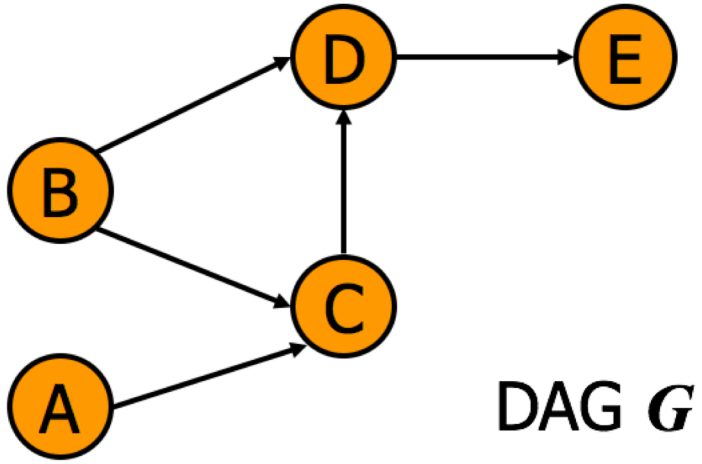
```
Algorithm TopologicalSort(G)
    H ← G          // Temporary copy of G
    n ← G.numVertices()
    while H is not empty do
        v ← find vertex with no outgoing edges
        Label v ← n
        n ← n - 1
        Remove v from H
```



DAG *G*

Recaps: Property of DAGs

- ***Degree*** of a vertex is the number of edges adjacent to the vertex, with loops counted twice.
- ***Outdegree*** of a vertex, the number of tail ends adjacent to the vertex.
- ***Indegree*** of a vertex, the number of head ends adjacent to the vertex

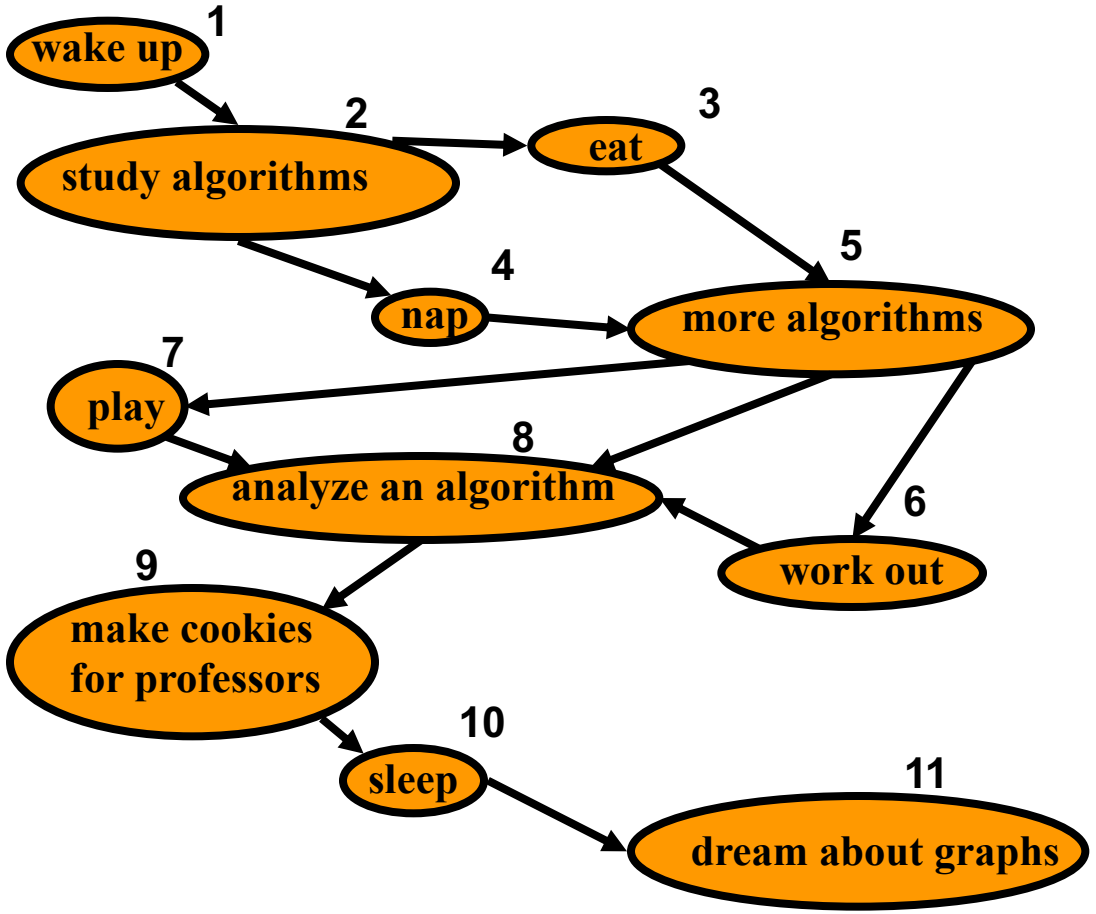


- A directed acyclic graph (DAG) is a digraph that has no directed cycles
- Prove: A DAG has at least one vertex with in-degree 0 and one vertex with out-degree 0.

An Example

- Number vertices, so that (u,v) in E implies $u < v$

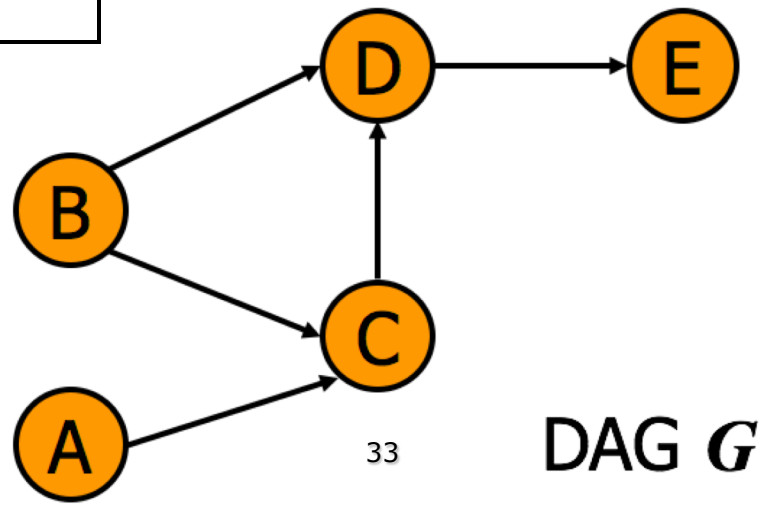
A typical CS (Algorithms) student day!



Naïve Algorithm for Topological Sorting

```
Algorithm TopologicalSort(G)
    H ← G          // Temporary copy of G
    n ← G.numVertices()
    while H is not empty do
        v ← find vertex with no outgoing edges
        Label v ← n
        n ← n - 1
        Remove v from H
```

- Running time???
  - O(|V| |E|)



DAG *G*

How to Improve it?

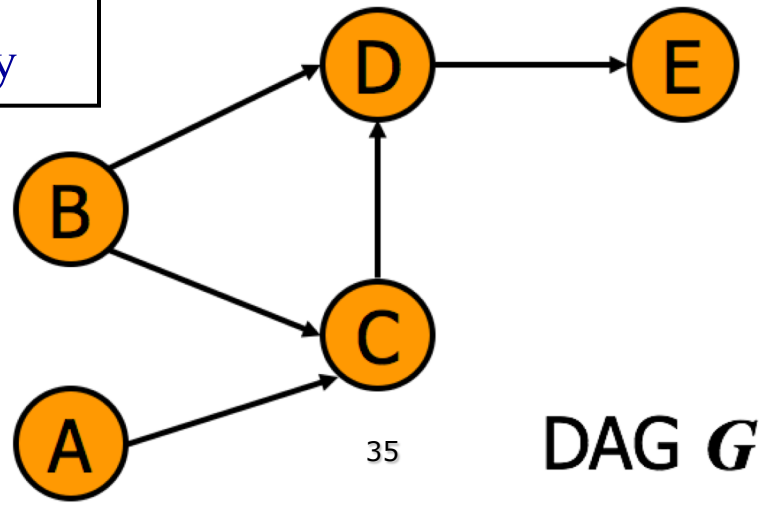
- We can repeatedly find vertex with no incoming edges, label and remove it

- How well does it work

Kahns Algorithm for Topological Sorting

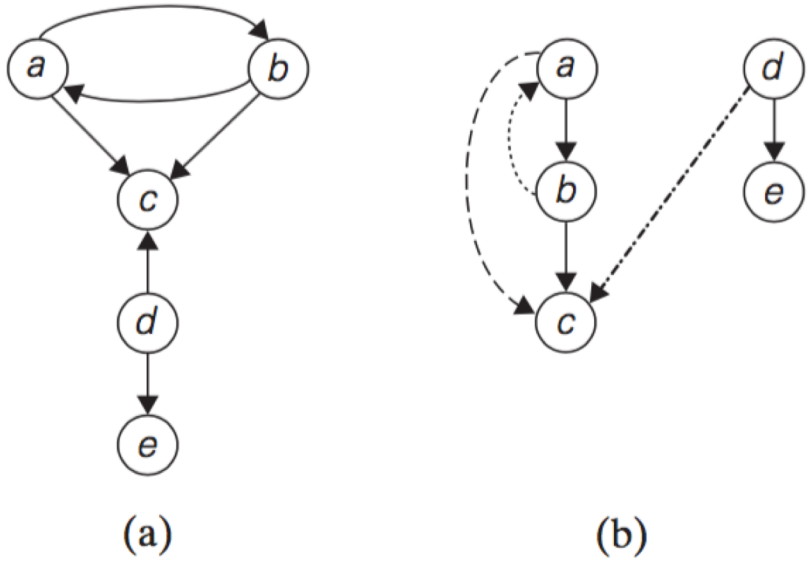
Algorithm TopologicalSort(***G***)
    H ← ***G*** // Temporary copy of ***G***
    L← List of vertices with no incoming edges
    n ← ***1***
    while ***L*** is not empty **do**
        v ← L
        Label ***v*** ← **n**
        n ← ***n*** + **1**
        Remove ***v*** *from* ***H***, update ***L*** accordingly

- Running time???



DAG ***G***

Types of Edges in Directed Graph (Digraph)




```
DFS(v)
{
   Label vertex v as reached.
   for (each unreached vertex u adjacenct from v)
            DFS(u);
}
```

FIGURE 4.5 (a) Digraph. (b) DFS forest of the digraph for the DFS traversal started at *a*.

Four types of edges possible in a DFS forest of a directed graph:

- ***tree edges*** (ab, bc, de),
- ***back edges*** (ba) from vertices to their ancestors,
- ***forward edges*** (ac) from vertices to their descendants in the tree other than their children, and
- ***cross edges*** (dc) leading to previously visited vertices other than their immediate predecessors and successors.

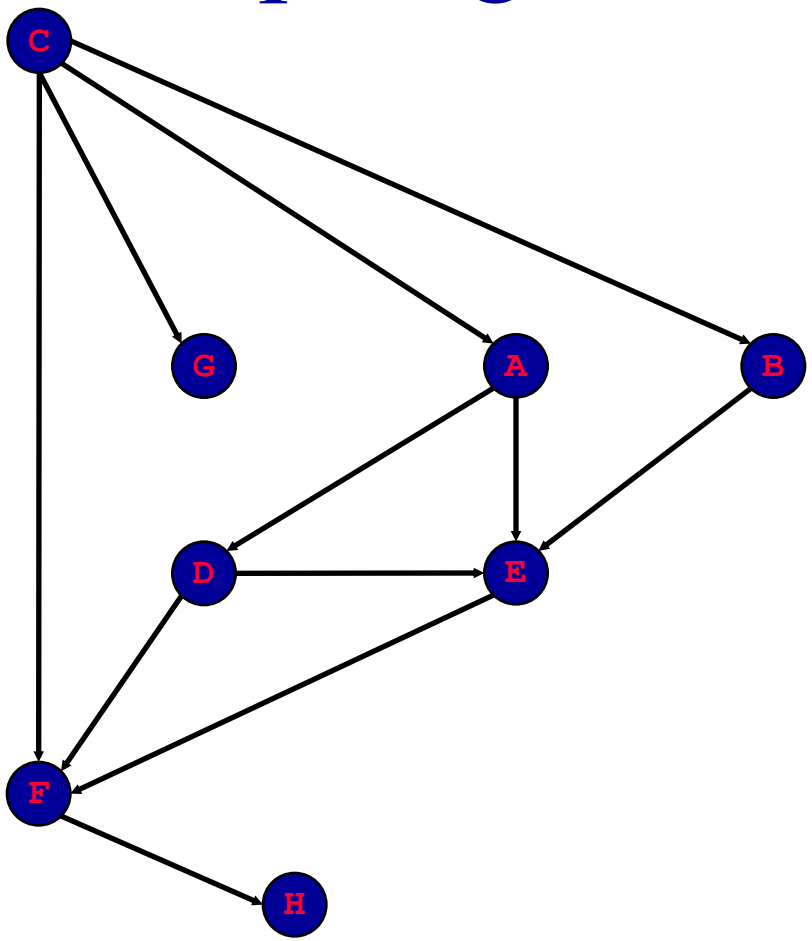
DFS and DAG

- Prove: If a DFS forest of a digraph has no back edges, the digraph is a ***dag.***

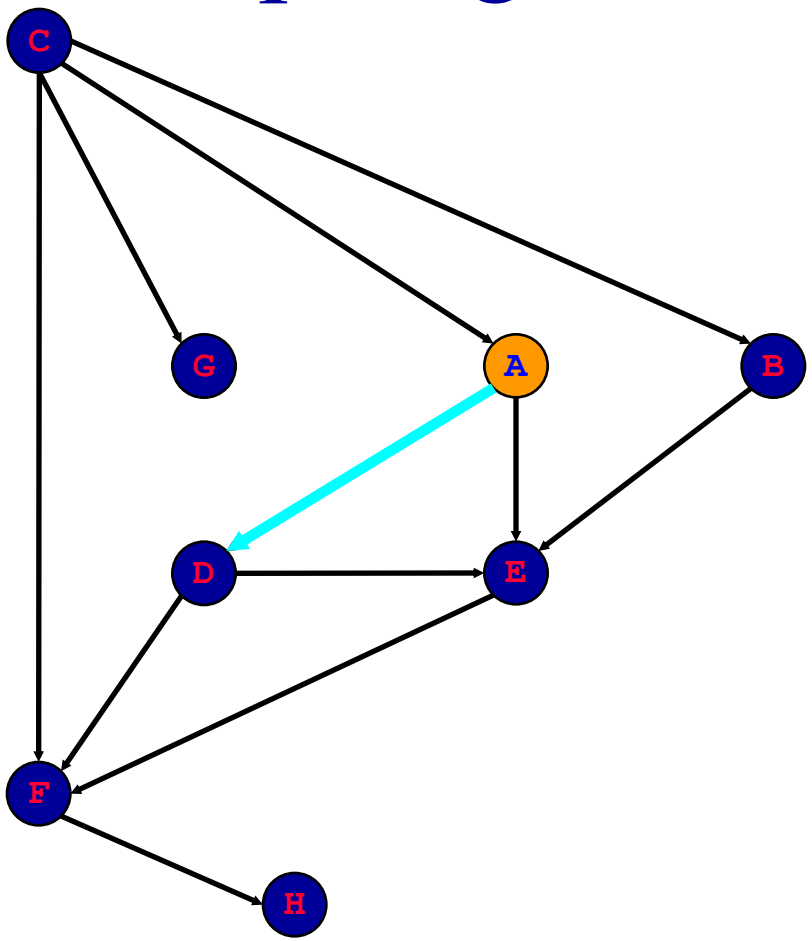
DFS to generate a topological ordering

- Perform a DFS traversal and note the order in which vertices become dead-ends (i.e., popped off the traversal stack).
- Reverse this order yields a solution to the topological sorting problem.
- If a back edge has been encountered, the digraph is not a dag, and topological sorting of its vertices is impossible.

Topological Sort: DFS

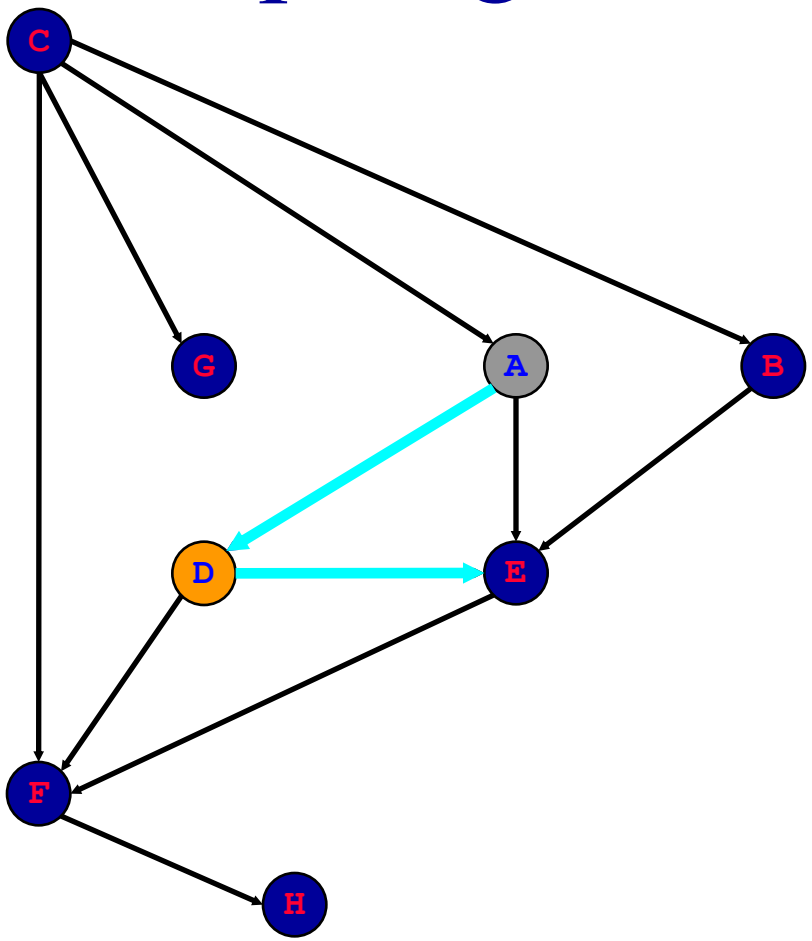


Topological Sort: DFS



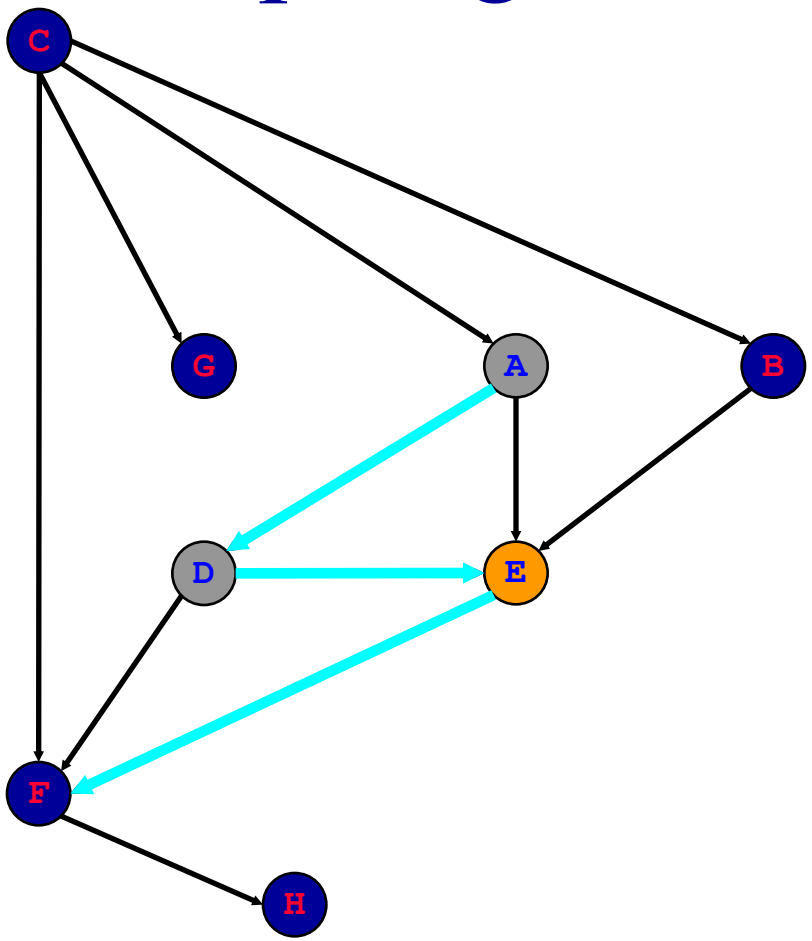

```
dfs(A)
```

Topological Sort: DFS



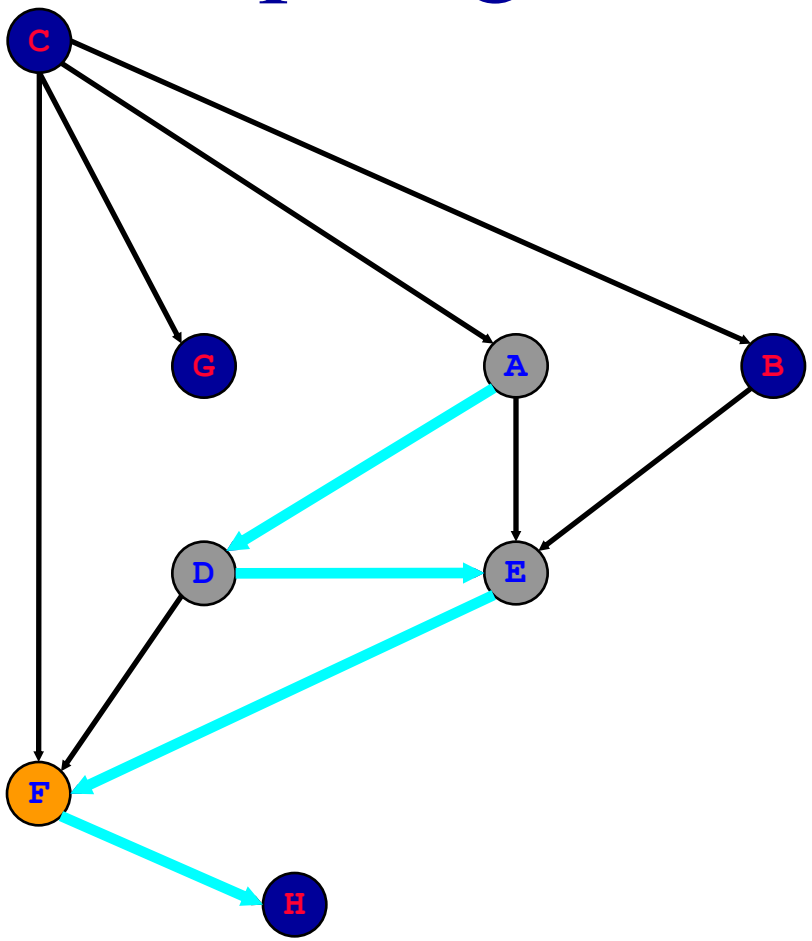

```
dfs(A)
  dfs(D)
```

Topological Sort: DFS



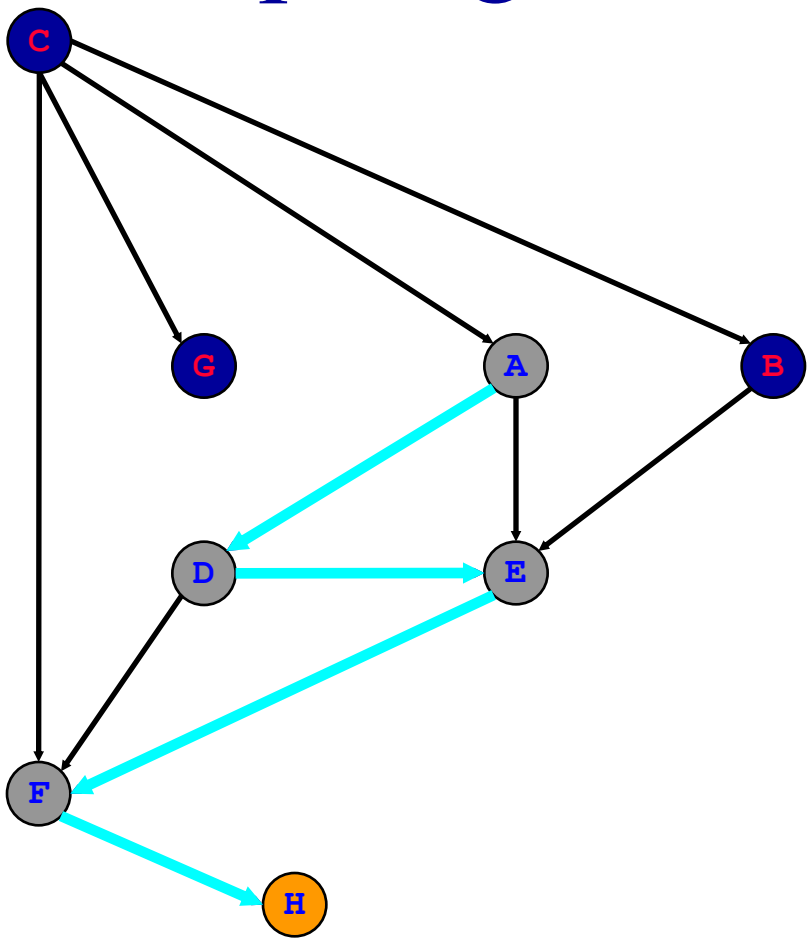

```
dfs(A)
  dfs(D)
    dfs(E)
```

Topological Sort: DFS



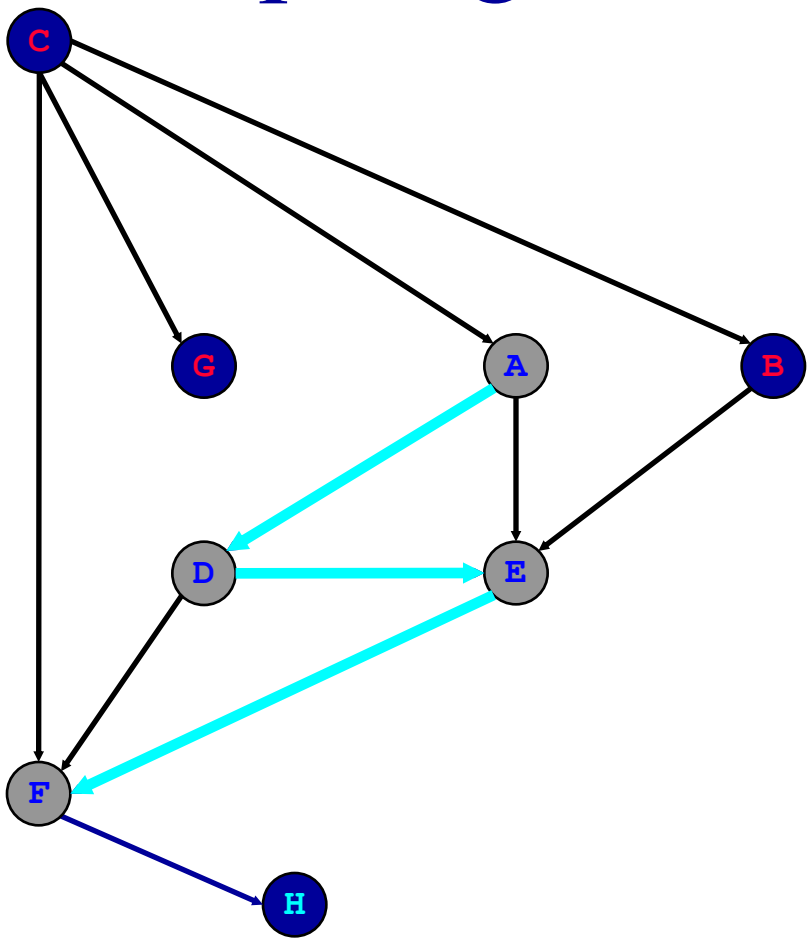

```
dfs(A)
  dfs(D)
    dfs(E)
      dfs(F)
```

Topological Sort: DFS



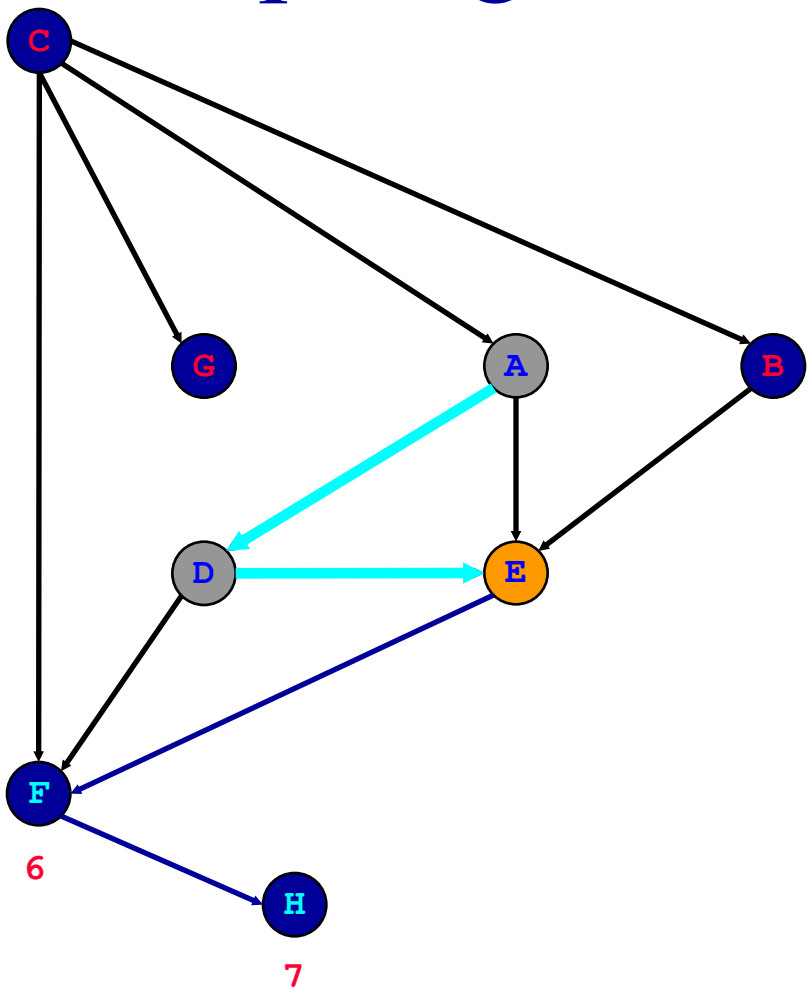

```
dfs(A)
  dfs(D)
    dfs(E)
      dfs(F)
        dfs(H)
```

Topological Sort: DFS



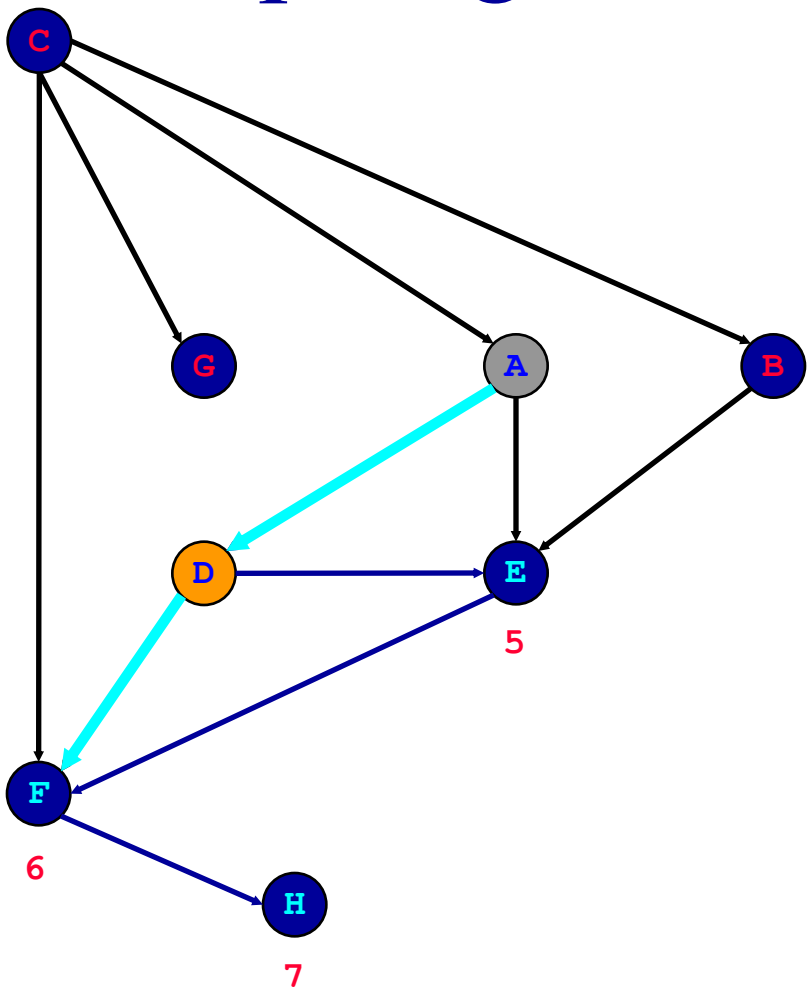

```
dfs(A)
  dfs(D)
    dfs(E)
      dfs(F)
```

Topological Sort: DFS



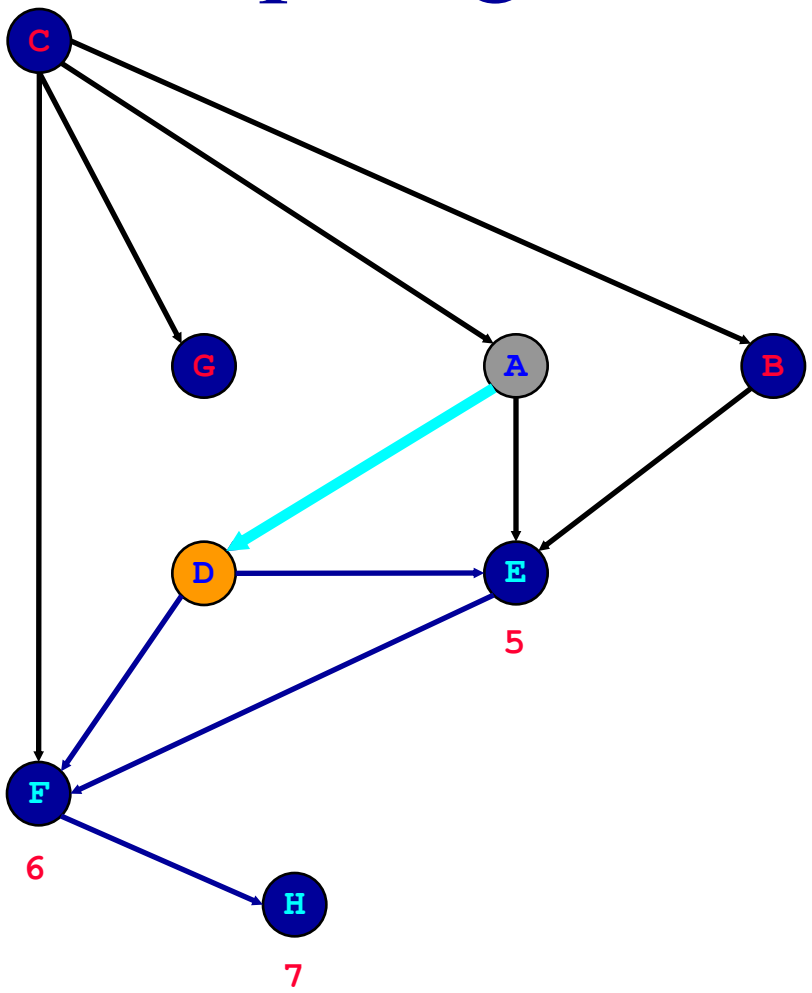

```
dfs(A)
  dfs(D)
    dfs(E)
```

Topological Sort: DFS



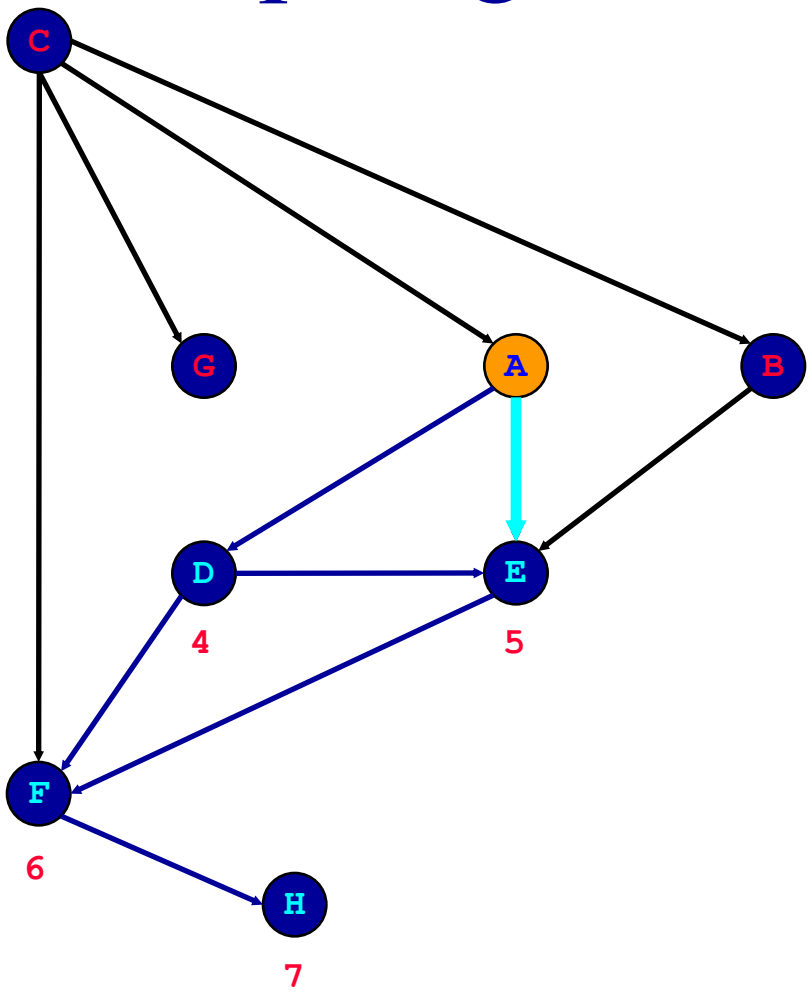

```
dfs(A)
  dfs(D)
```

Topological Sort: DFS



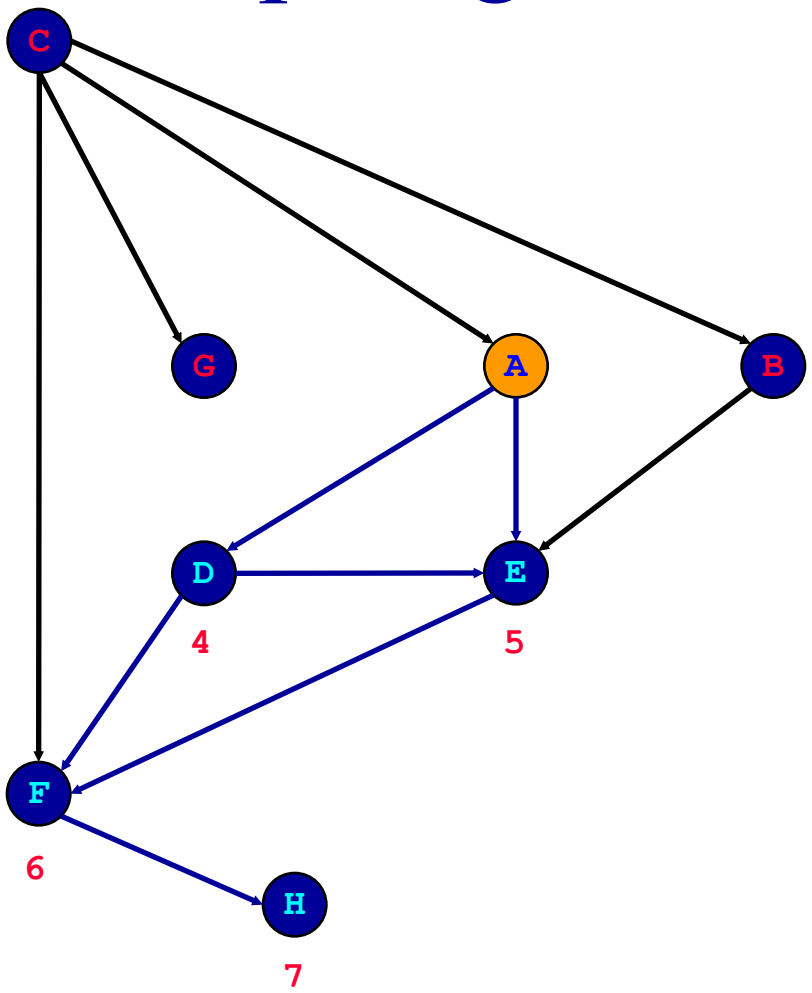

```
dfs(A)
  dfs(D)
```

Topological Sort: DFS



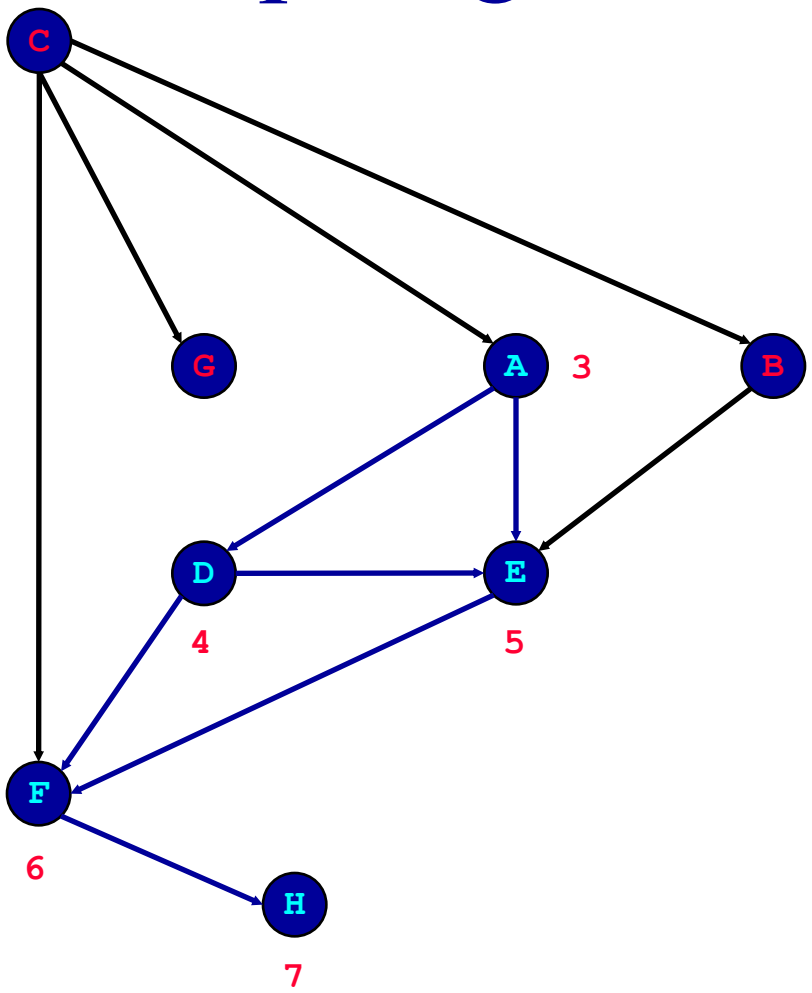
dfs(A)

Topological Sort: DFS

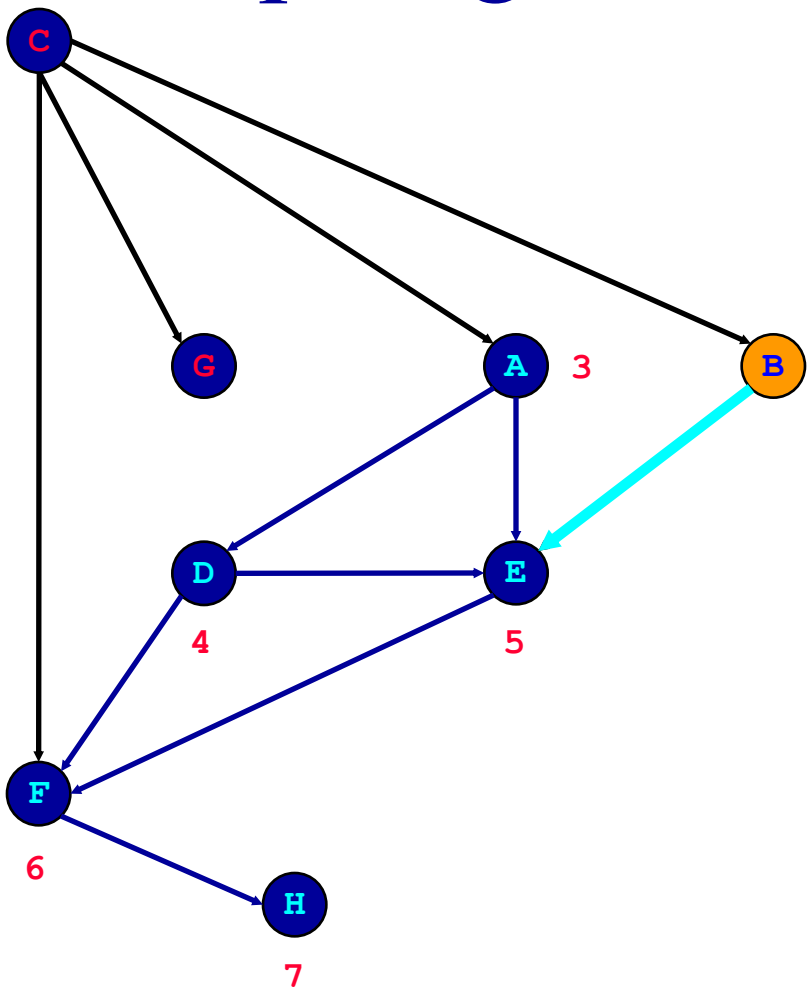


dfs(A)

Topological Sort: DFS

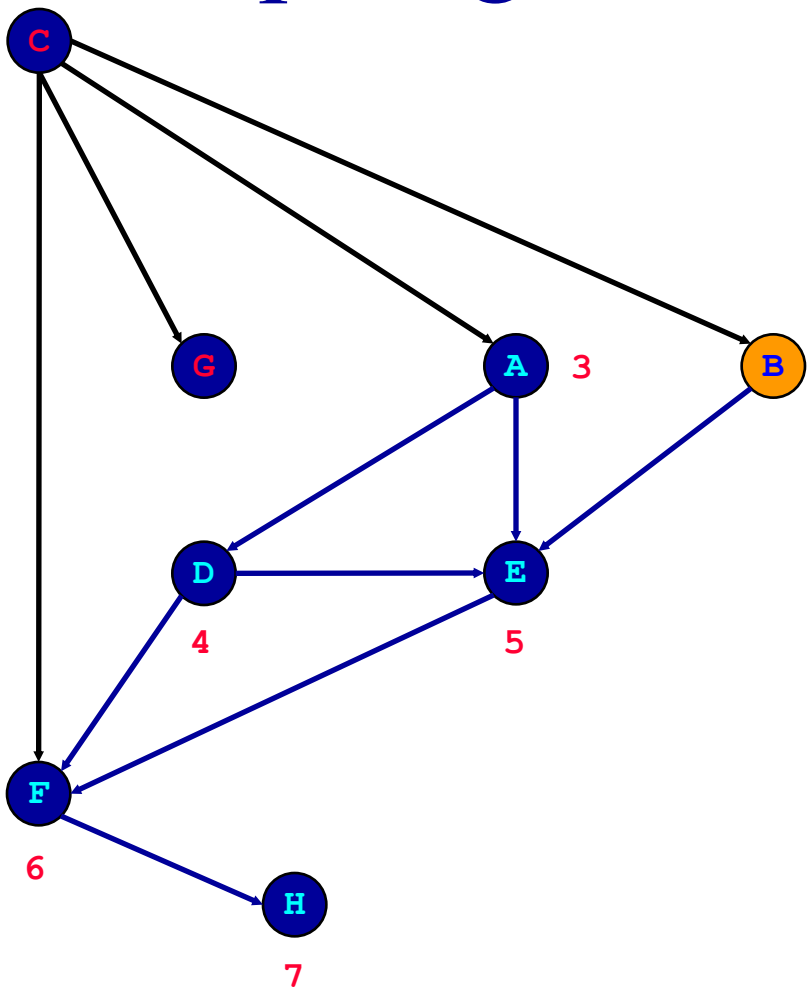


Topological Sort: DFS



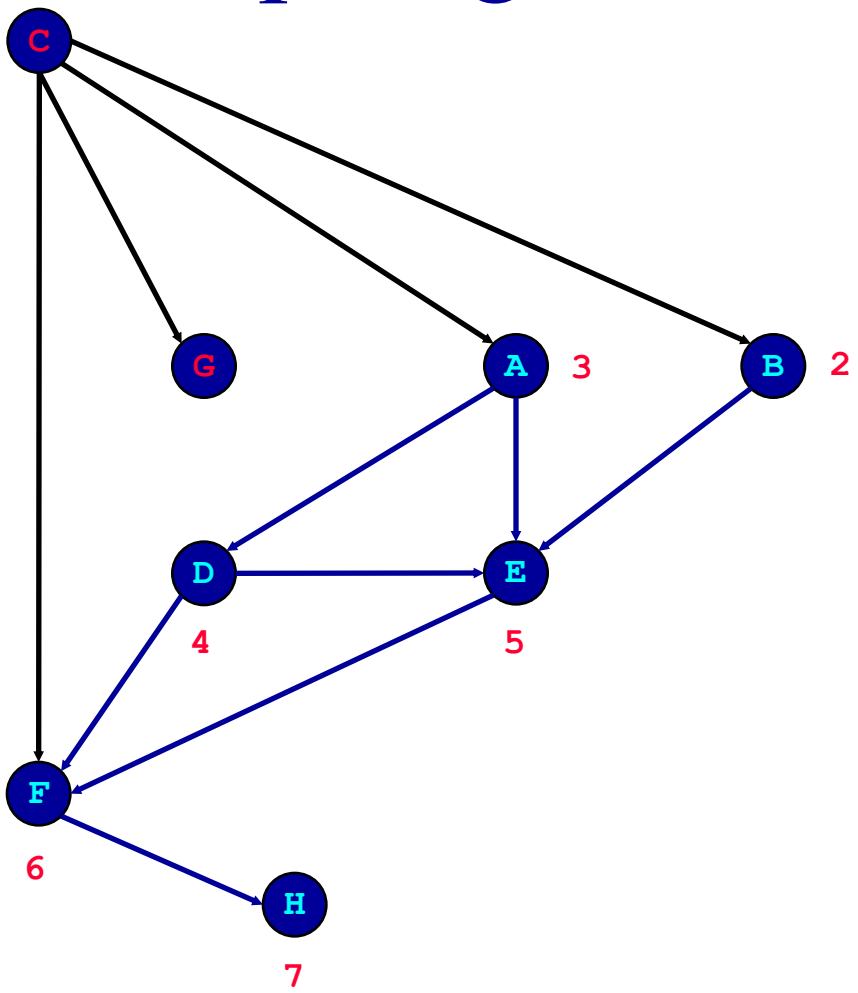
dfs(B)

Topological Sort: DFS

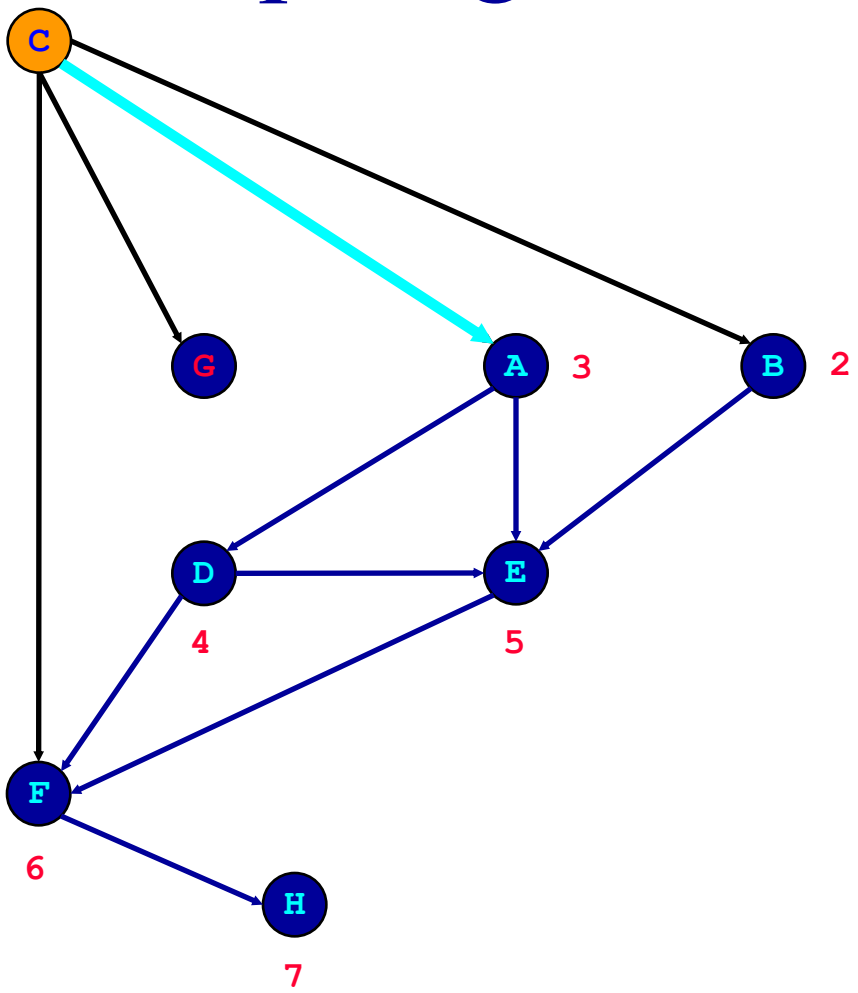


dfs(B)

Topological Sort: DFS

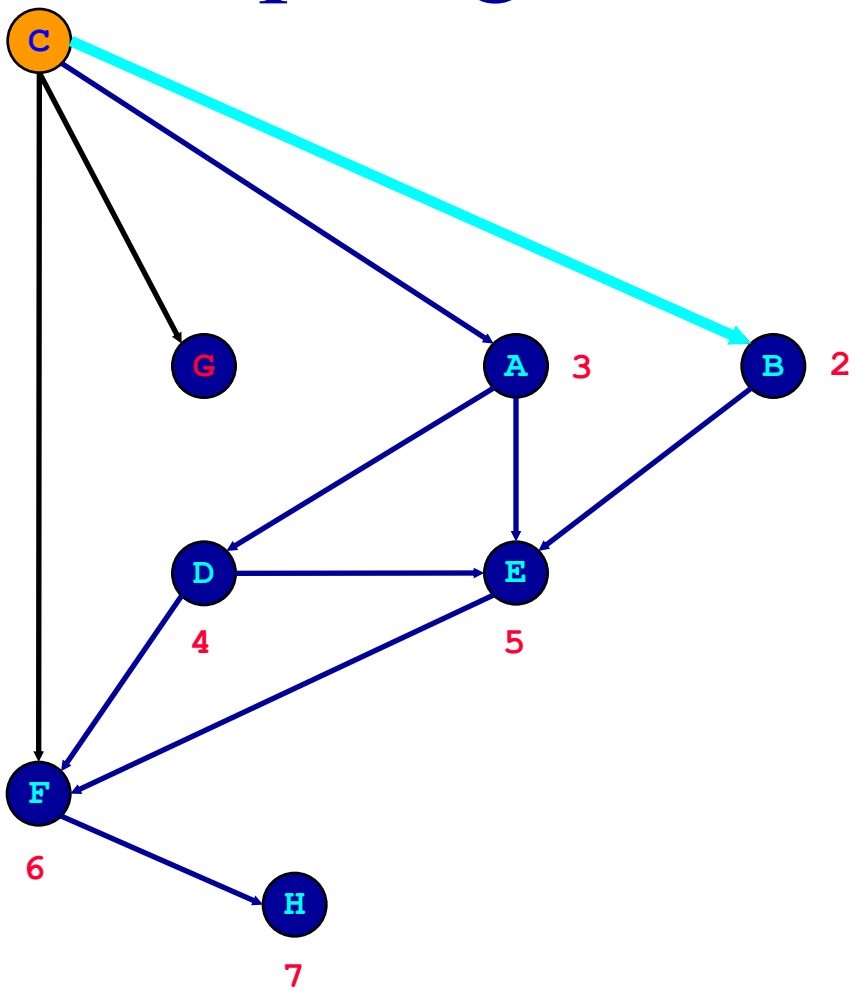


Topological Sort: DFS



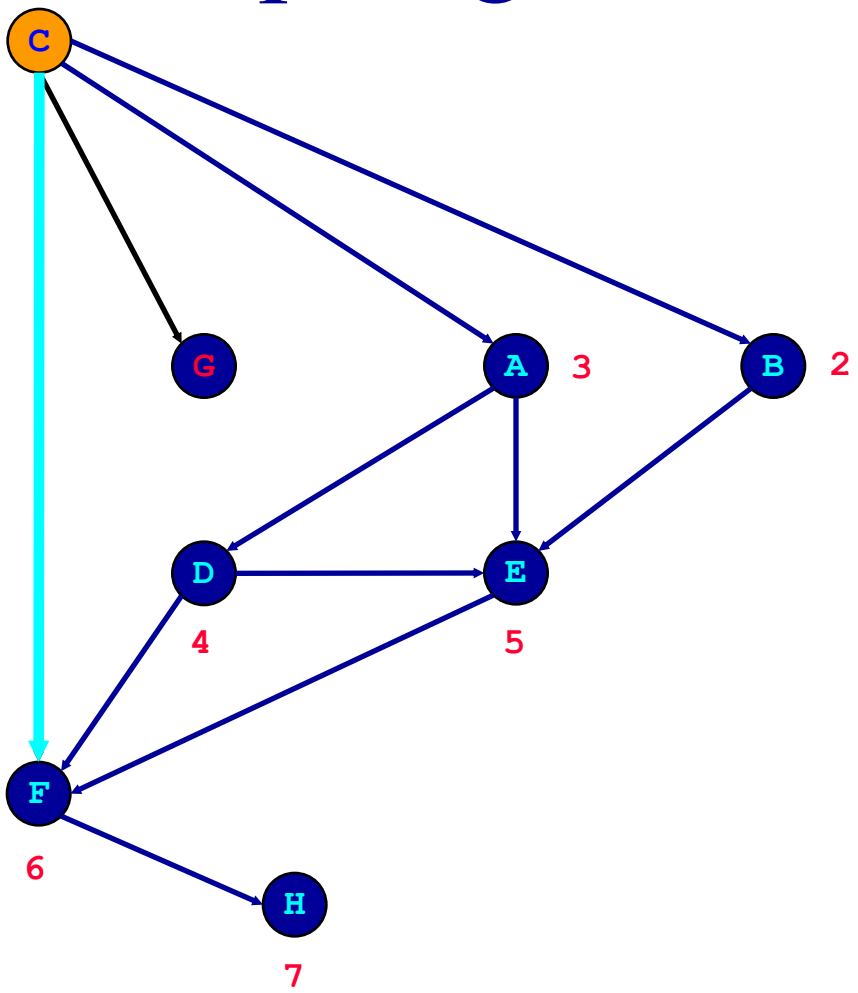
dfs(C)

Topological Sort: DFS



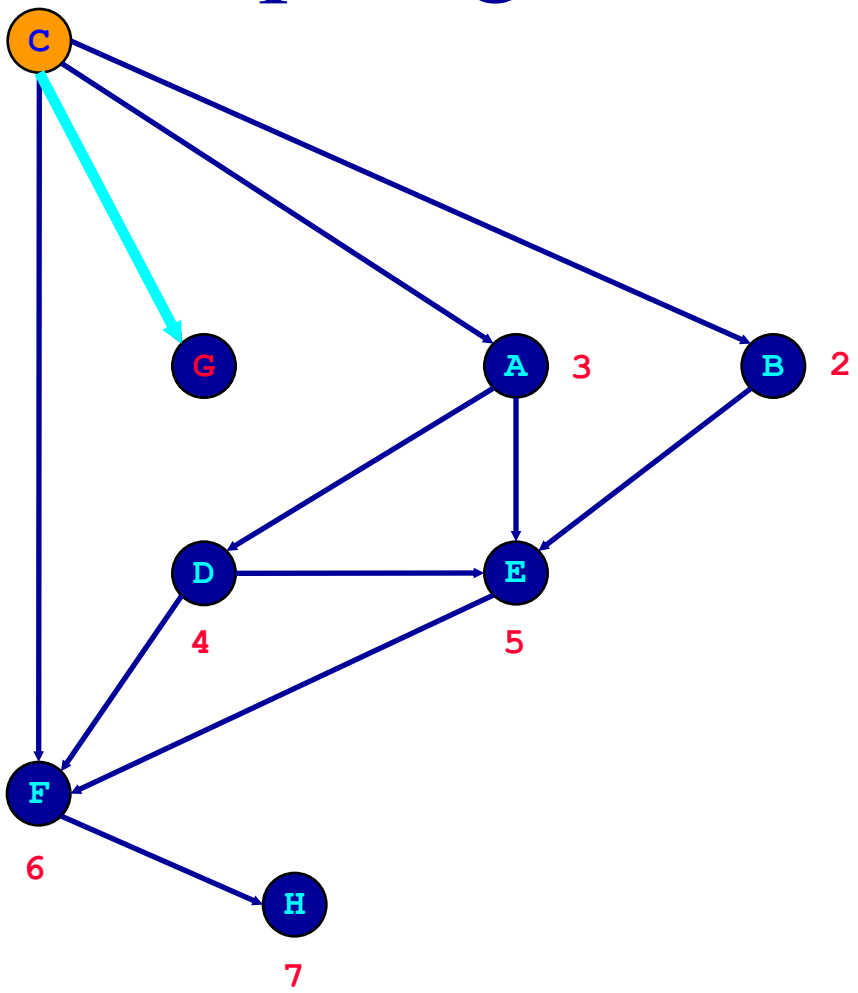
dfs(C)

Topological Sort: DFS



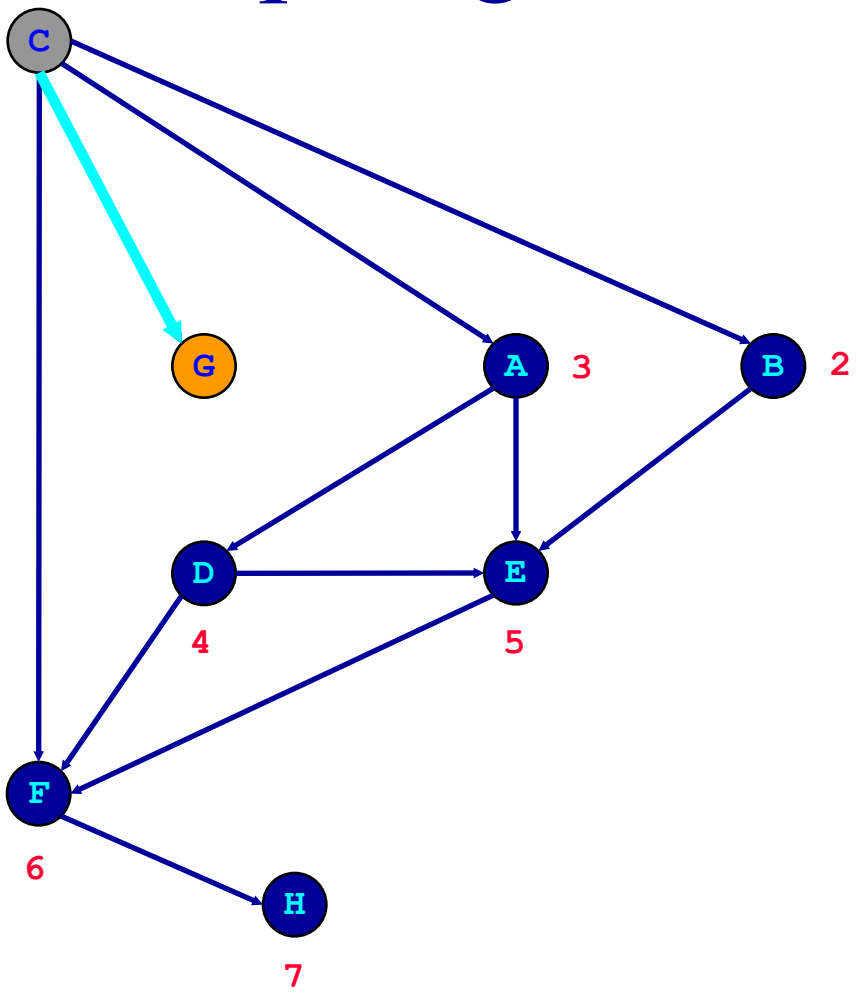
dfs(C)

Topological Sort: DFS



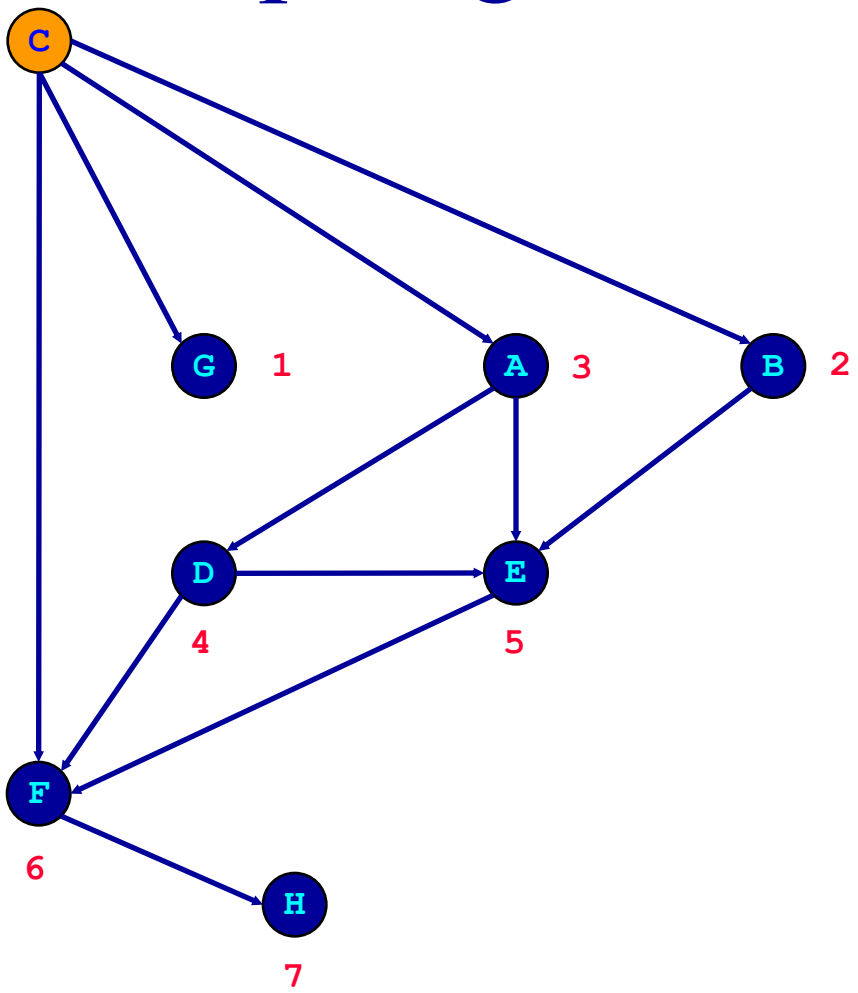
dfs(C)

Topological Sort: DFS



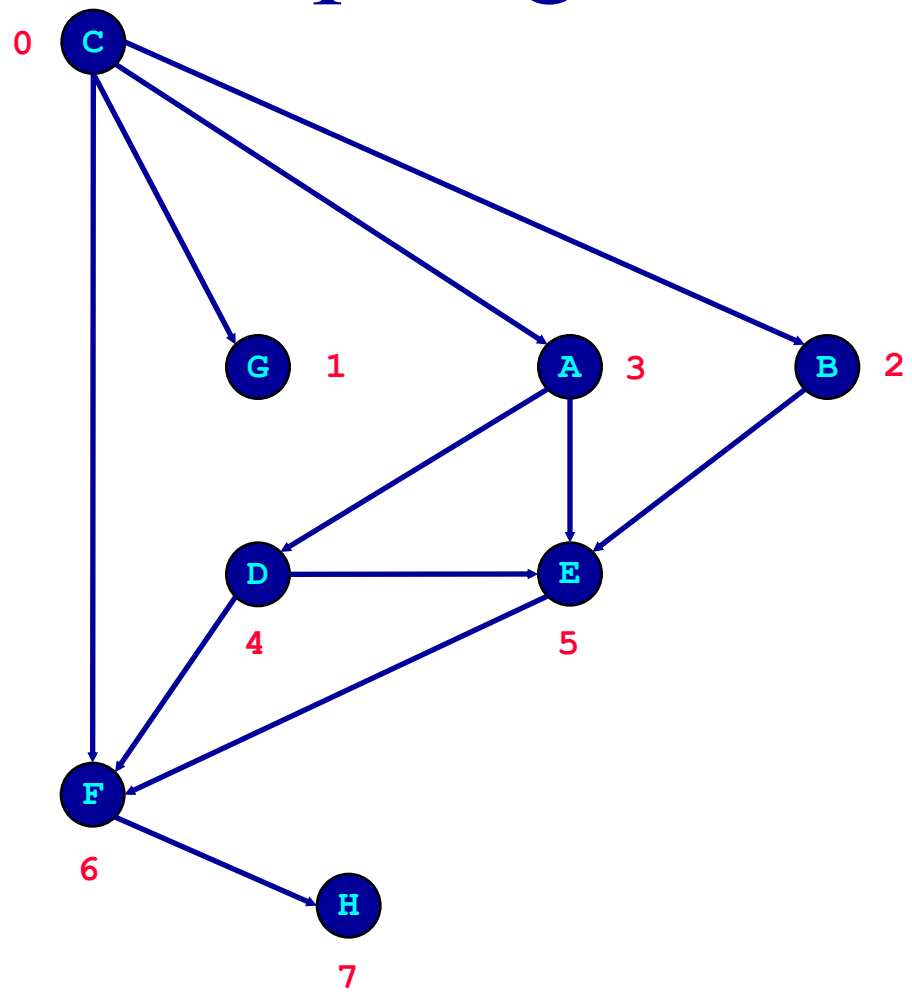

```
dfs(C)
  dfs(G)
```

Topological Sort: DFS

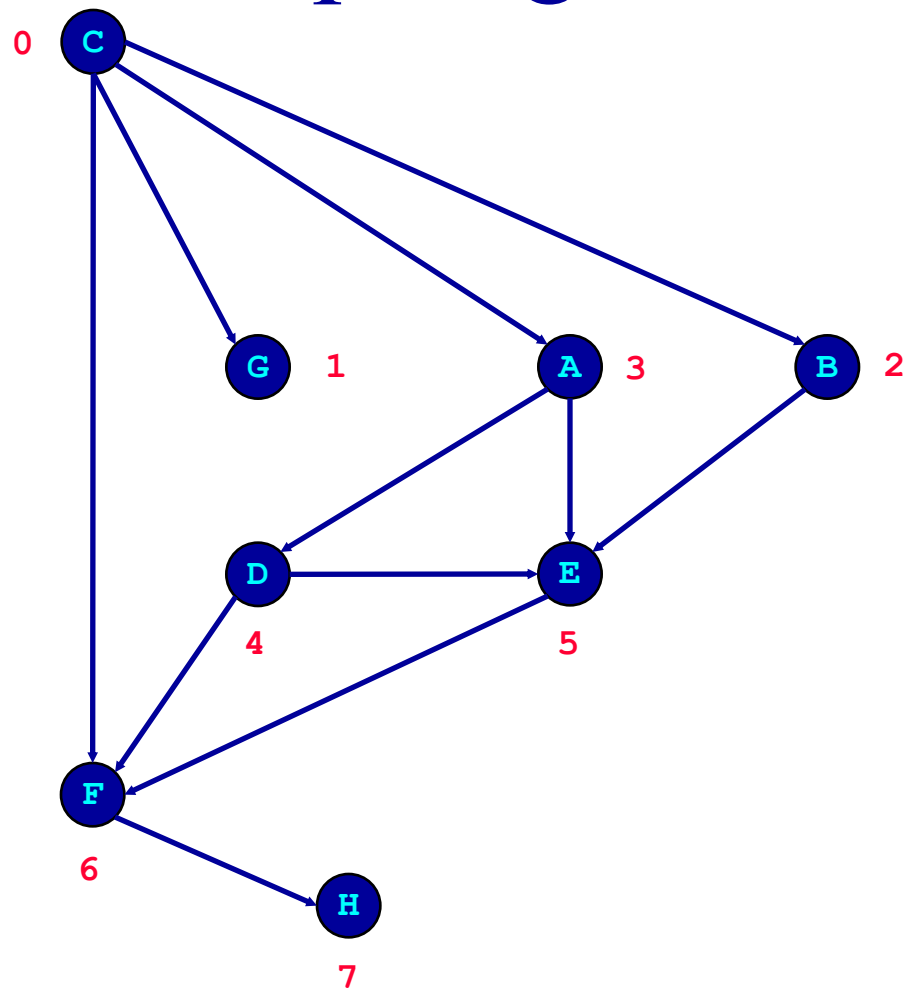


dfs(C)

Topological Sort: DFS



Topological Sort: DFS



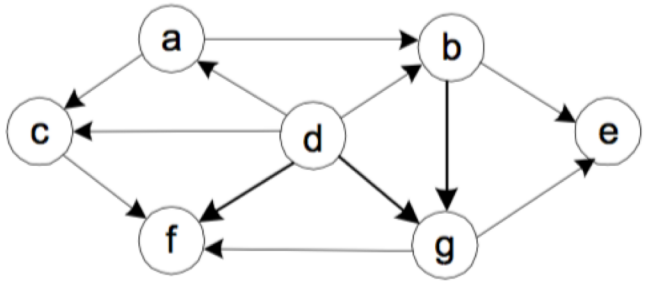
Topological order: **C G B A D E F H**

An Algorithm for Topological Sorting using DFS

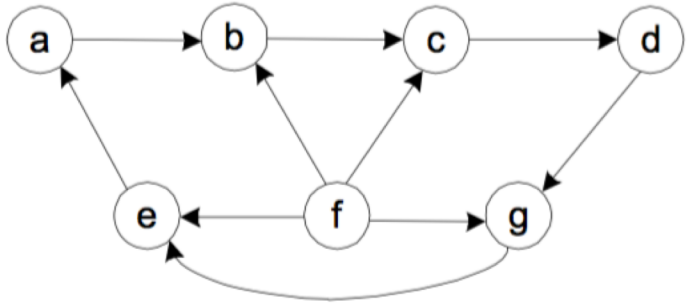
Algorithm TopologicalSort(***G***)
    L ← an empty list
    while ***G*** is not fully explored **do**
        pick an arbitrary node ***u***
        perform DFS starting from ***u***
        every time a node is finished,
            insert it in front of ***L***
        return ***L***

- Running time? O(|V| + |E|)

1. Apply the DFS-based algorithm to solve the topological sorting problem for the following digraphs:



(a)

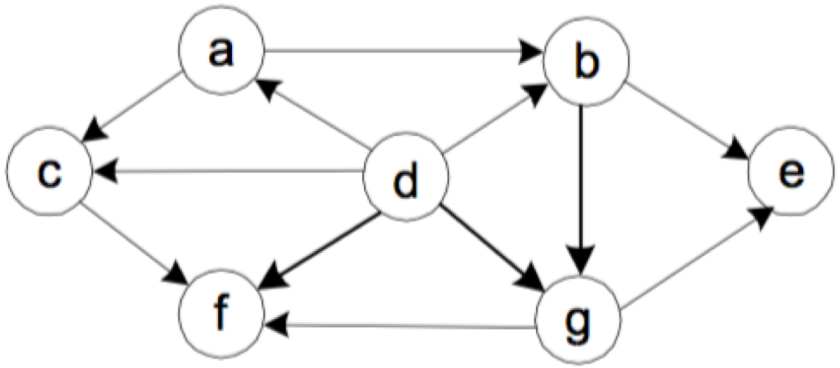


(b)

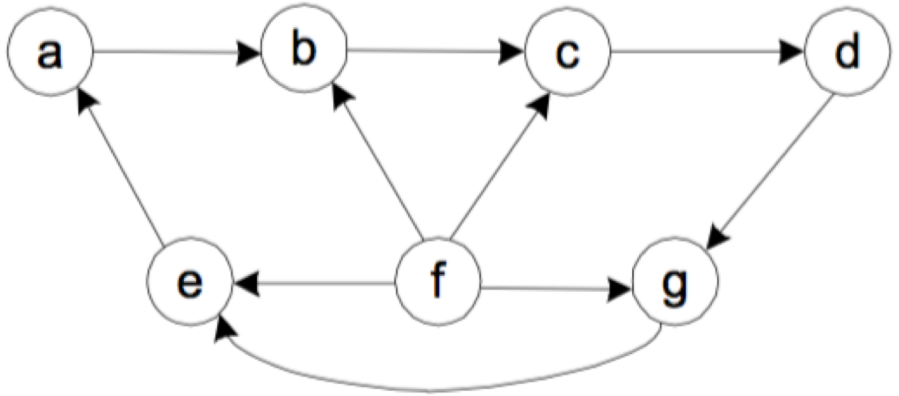
.

Exercise

1. Apply the DFS-based algorithm to solve the topological sorting problem for the following digraphs:



(a)



(b)

.

e f g b c a d

No solution!

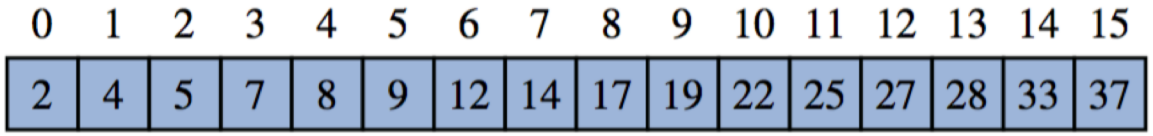
Decrease by Constant Factor

- Decrease by a constant factor (usually by half)
  - binary search and bisection method
  - exponentiation by squaring
  - multiplication à la russe

Binary Search

- When the sequence is ***sorted*** and ***indexable***, there is a more efficient algorithm
  - Initially, low = 0 and high = $n - 1$. We then compare the target value to the ***median candidate***

$$\text{mid} = \lfloor (\text{low} + \text{high})/2 \rfloor$$



Binary Search Algorithm

ALGORITHM *BinarySearch*$(A[0..n-1], K)$

```
//Implements nonrecursive binary search
//Input: An array A[0..n − 1] sorted in ascending order and
//       a search key K
//Output: An index of the array's element that is equal to K
//       or −1 if there is no such element
l ← 0;    r ← n − 1
while l ≤ r do
    m ← ⌊(l + r)/2⌋
    if K = A[m] return m
    else if K < A[m] r ← m − 1
    else l ← m + 1
return −1
```

Efficiency

- $C_{worst}(n)$: the number of key comparisons in the worst case.
- Since after one comparison the algorithm faces the same situation but for an array half the size, we get the following recurrence relation for $C_{worst}(n)$:

$$C_{worst}(n) = C_{worst}(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1, \quad C_{worst}(1) = 1. \qquad \textbf{(4.3)}$$

Efficiency Class

$$C_{worst}(n) = \lfloor \log_2 n \rfloor + 1 = \lceil \log_2 (n+1) \rceil.$$

$$C_{avg}(n) \approx \log_2 n.$$

Fake-Coin Problem

- We have *n* identically looking coins
- One is fake
- We can compare any two sets of coins with a balance scale
  - Fake coin is lighter than real coin
- Design an efficient algorithm for detecting the fake coin

Fake-Coin Problem

- Divide n coins into 2 piles of $n/2$ coins each, leaving 1 extra coin if n is odd

- If the two piles weigh the same, the coin aside is fake
  - Otherwise proceed in the same manner with the lighter pile

- $Weighing_{worst} = W(n) = W(n/2)+1$ for n>1, W(1) = 0

A More Efficient Algorithm

Divide the coins into 3 piles of $n/3$ coins each

If A = B, then C contains the counterfeit coin.

After weighing the 2 piles, we reduce the instance size by a factor of 3

Efficiency class: $\Theta(\log_3 n)$

Recaps: Decrease and Conquer

- ***Decrease by a constant factor*** **(usually by half)**
  - binary search and bisection method
  - exponentiation by squaring

- *Variable-size decrease*
  - Euclid's algorithm
  - selection by partition
  - Nim-like games

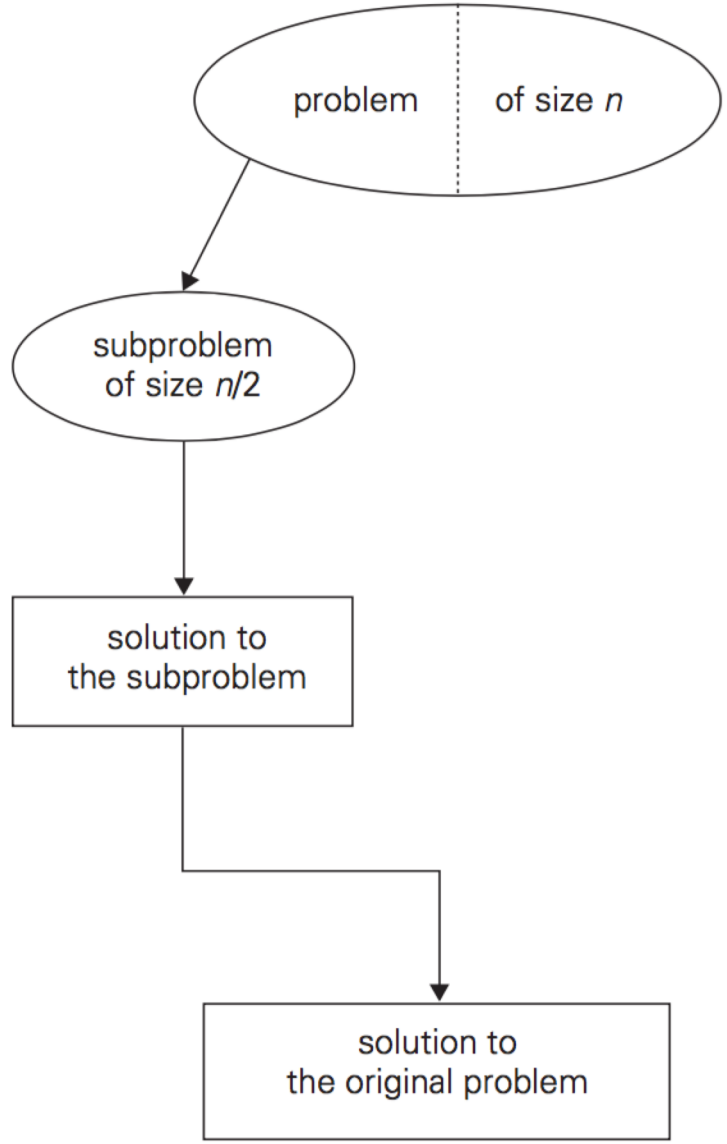


FIGURE 4.2 Decrease-(by half)-and-conquer technique.

Russian Peasant Multiplication

- A method for multiplying two numbers that does not require the multiplication table, only the ability to multiply and divide by 2, and to add.
- Known to Egyptian mathematicians as early as 1650 B.C. Widely used in Russia in the nineteenth century;
- It leads to easy hardware implementation.

Russian Peasant Multiplication

The problem: Compute the product of two positive integers
Can be solved by a decrease-by-half algorithm based on the following formulas.

For even values of *n*:

$$n * m = \frac{n}{2} * 2m$$

For odd values of *n*:

$$n * m = \frac{n-1}{2} * 2m + m \quad \text{if } n > 1 \quad \text{and} \quad m \text{ if } n = 1$$

Repetitively apply formula until n=1.

Example of Russian Peasant Multiplication

Compute 20 * 26

n	*m*	
20	26	
10	52	
5	104	104
2	208	+
1	416	416
		520

Note: Method reduces to adding *m*'s values corresponding to odd *n*'s.

Variable-Size-Decrease Algorithms

Here, instance size reduction varies from one iteration to another

Examples:

- Euclid's algorithm for greatest common divisor
- Partition-based algorithm for selection problem
- Interpolation search
- Some algorithms on binary search trees
  Nim and Nim-like games

Euclid's Algorithm

Euclid's algorithm is based on repeated application of equality

$$\gcd(m, n) = \gcd(n, m \bmod n)$$

Ex.: gcd(80,44)

= gcd(44,36)

= gcd(36, 8)

= gcd(8,4)

= gcd(4,0)

= 4

One can prove that the size, measured by the second number, decreases at least by half after two consecutive iterations.

Hence, $T(n) \in \Theta(\log n)$

Selection Problem

Find the *k*-th smallest element (k-th *order statistic*) in a list of *n* numbers

- For $k = 1$ or $k = n$, simply scan list to find the smallest and largest element, respectively
- Interesting case is to find the *median*: $k = \lceil n/2 \rceil$

  Example: 4, 1, 10, 9, 7, 12, 8, 2, 15 median = ?

The median is used in statistics as a measure of an average value of a sample. In fact, it is a better (more robust) indicator than the mean, which is used for the same purpose.

Two Partitioning Algorithms

The sorting-based algorithm: Sort and return the *k*-th element. Efficiency: ???

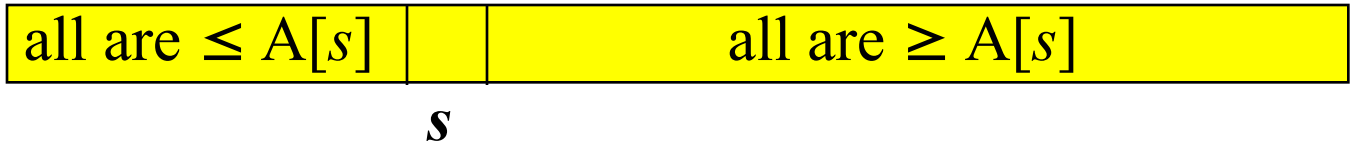
A faster algorithm is based on using the *quicksort-like* partition of the list.

There are two principal ways to partition an array:

- One-directional scan (Lomuto's partitioning algorithm)
- Two-directional scan (Hoare's partitioning algorithm)

Algorithms for the Selection Problem

Let s be a split position obtained by a partition:



Assuming that the list is indexed from 0 to $n-1$:

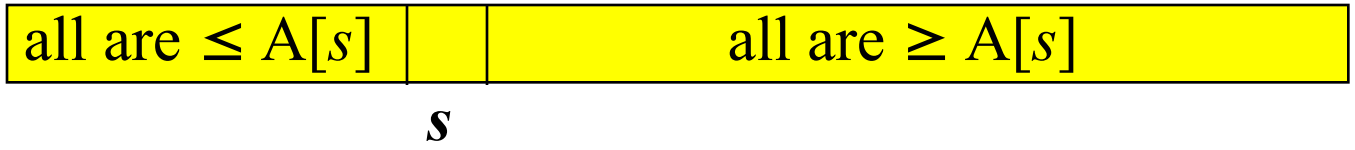
If $s = k-1$, what shall we do??

if $s > k-1$, ??

if $s < k-1$, ??

Algorithms for the Selection Problem

Let s be a split position obtained by a partition:



Assuming that the list is indexed from 0 to $n-1$:

If $s = k-1$, the problem is solved;

if $s > k-1$, look for the k-th smallest elememt in the ***left*** part;
if $s < k-1$, look for the $(k-s)$-th smallest element in the ***right*** part.

Note: The algorithm can simply continue until $s = k-1$.

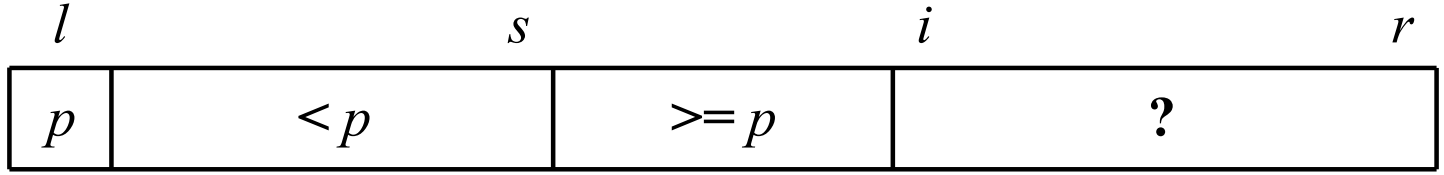
Algorithm QuickSelect

ALGORITHM *Quickselect*$(A[l..r], k)$

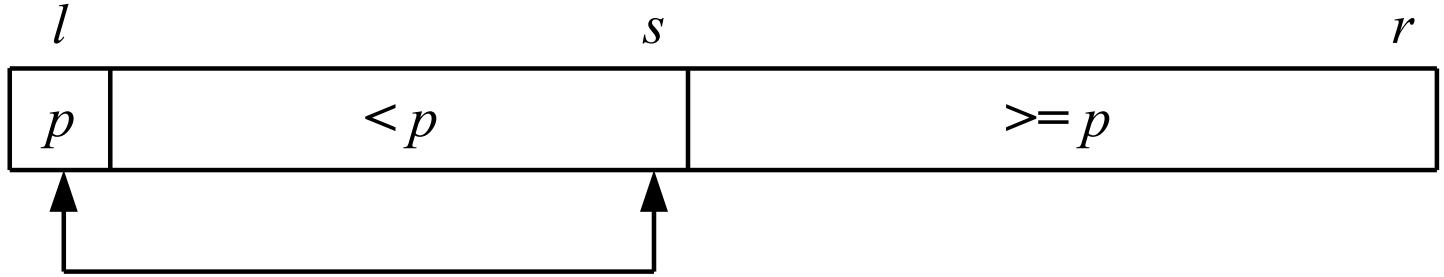
//Solves the selection problem by recursive partition-based algorithm
//Input: Subarray $A[l..r]$ of array $A[0..n-1]$ of orderable elements and
// integer k $(1 \le k \le r - l + 1)$
//Output: The value of the kth smallest element in $A[l..r]$
$s \leftarrow LomutoPartition(A[l..r])$ //or another partition algorithm
if $s = k - 1$ **return** $A[s]$
else if $s > l + k - 1$ *Quickselect*$(A[l..s-1], k)$
else *Quickselect*$(A[s+1..r], k - 1 - s)$

Lomuto's Partitioning Algorithm

Scans the array left to right maintaining the array's partition into three contiguous sections: $< p$, $\geq p$, and unknown, where p is the value of the first element (the partition's *pivot*).



On each iteration the unknown section is decreased by one element until it's empty and a partition is achieved by exchanging the pivot with the element in the split position s.



A. Levitin "Introduction to the Design & Analysis of Algorithms," 3rd ed., Ch. 4 ©2012 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

Algorithm LomutoPartition

ALGORITHM *LomutoPartition*$(A[l..r])$

```
//Partitions subarray by Lomuto's algorithm using first element as pivot
//Input: A subarray A[l..r] of array A[0..n − 1], defined by its left and right
//         indices l and r (l ≤ r)
//Output: Partition of A[l..r] and the new position of the pivot
p ← A[l]
s ← l
for i ← l + 1 to r do
    if A[i] < p
        s ← s + 1;   swap(A[s], A[i])
swap(A[l], A[s])
return s
```

Tracing the Median / Selection Algorithm

Example: 4 1 10 9 7 12 8 2 15 Here: $n = 9$, $k = \lceil 9/2 \rceil = 5$

array index	1	2	3	4	5	6	7	8	9	
	4	1	*10*	9	7	12	8	*2*	15	
	4	1	2	9	7	12	8	10	15	--- *swap 10 and 2*
	2	1	**4**	9	7	12	8	10	15	--- s=3 < k=5 (do Right)
				9	7	*12*	*8*	10	15	
				9	7	8	12	10	15	--- *swap 12 and 8*
				8	7	**9**	12	10	15	--- s=6 > k=5 (do Left)
				8	7					
				7	**8**					--- s=k=5

Solution: median is 8

Efficiency of the Partition-based Algorithm

ALGORITHM *Quickselect*$(A[l..r], k)$

//Solves the selection problem by recursive partition-based algorithm
//Input: Subarray $A[l..r]$ of array $A[0..n-1]$ of orderable elements and
// integer k $(1 \le k \le r - l + 1)$
//Output: The value of the kth smallest element in $A[l..r]$
$s \leftarrow LomutoPartition(A[l..r])$ //or another partition algorithm
if $s = k - 1$ **return** $A[s]$
else if $s > l + k - 1$ *Quickselect*$(A[l..s-1], k)$
else *Quickselect*$(A[s+1..r], k - 1 - s)$

Best case ???

Average case ???

Worst case ???

Efficiency of the Partition-based Algorithm

Average case (average split in the middle):

$C(n) = C(n/2) + \Theta(n)$ $\qquad\qquad$ $C(n) \in \Theta(n)$ [cf. Master Theorem]

Worst case (degenerate split): $C(n) \in \Theta(n^2)$

A more sophisticated choice of the pivot leads to a complicated algorithm with $\Theta(n)$ worst-case efficiency.

3.4.6

```
def split(a):
    for i from 1 to 2^n - 2:
        s1 = set()
        s2 = set()
        for j from 0 to n - 1:
            if bit j of i is set:
                s1.add(a[j])
            else:
                s2.add(a[j])
        if sum(s1) == sum(s2):
            return (s1, s2)
    return null
```

ALGORITHM *additivePartitions*(*A*[0...(*n* - 1)])
// Input: an array *A* containing *n* positive integers

// Output: Two lists *B* and *C*, which together contain all of *A*'s elements. The sum of *B*'s
// elements equals the sum of *C*'s elements. If this is impossible, **null** is returned instead.

B ← <empty list>
C ← <empty list>
if *n* = 0
    return (*B*, *C*)
total ← *sum*(A)
// only generating subsets represented by bitstrings with leading zero, not including the
// empty set
for *m* ← 1 **to** 2^{n-1} - 1 **do**
    psum ← 0
    for *i* ← 0 **to** *n* - 2 **do**
        if (2^i **bitwise and** *m*) ≠ 0
            psum ← *psum* + *A*[*i*]
            B.append(*A*[*i*])
    if *psum* = *total* - *psum*
        // Generate complement of current subset.
        for *j* ← 0 **to** *n* - 2 **do**
            if (2^j **bitwise and** *m*) = 0
                C.append(*A*[*j*])
        C.append(*A*[*n* - 1])
        return (*B*, *C*)
    B ← <empty list>
    C ← <empty list>
return null

Recap: Decrease and Conquer

- *Decrease by a constant* (usually by 1):
  - graph traversal algorithms (BFS and DFS)
  - insertion sort
  - topological sorting
  - algorithms for generating permutations, subsets

- ***Decrease by a constant factor*** **(usually by half)**
  - binary search and bisection method
  - exponentiation by squaring

- *Variable-size decrease*
  - Euclid's algorithm
  - selection by partition
  - Nim-like games

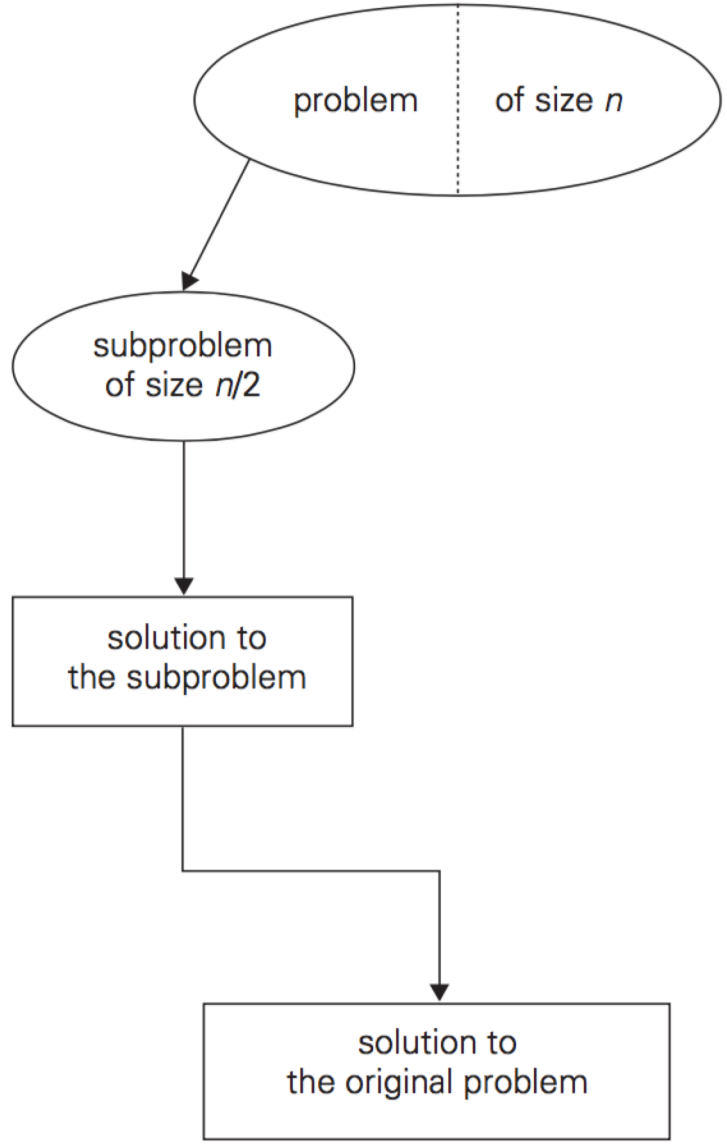


FIGURE 4.2 Decrease-(by half)-and-conquer technique.

Interpolation Search

- Assume the values of the array's elements to grow linearly from $A[l]$ to $A[r]$.
- Interpolation search: estimate location of the search key in $A[l..r]$ by using its value v.
- Binary search is probably better for smaller files but interpolation search is worth considering for large files and for applications where comparisons are particularly expensive or access costs are very high -- Robert Sedgewick

Interpolation Search

- The location of v is estimated as the x-coordinate of the point on the straight line through (l, A[l]) and (r, A[r]) whose y-coordinate is v:

$$x = l + \lfloor (v - A[l])(r - l)/(A[r] - A[l]) \rfloor$$

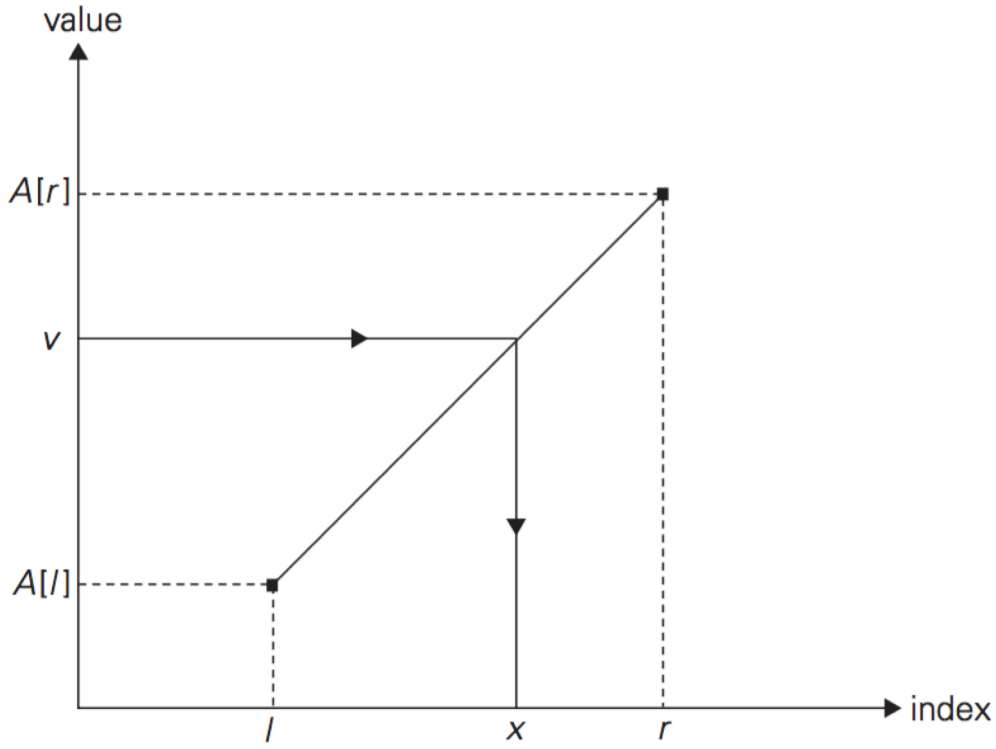


FIGURE 4.14 Index computation in interpolation search.

Interpolation Search

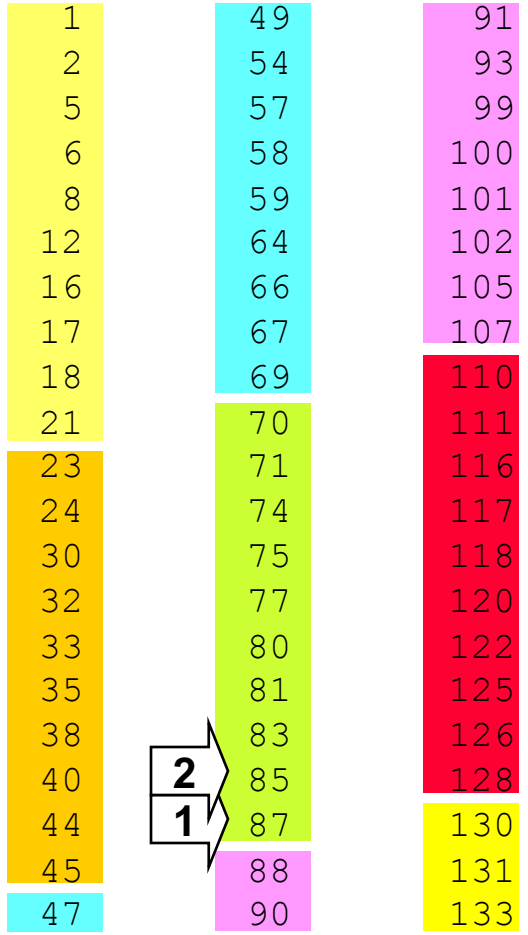
Is the number 85 in the 63-entry list to the right?

When looking for an entry x in a list, probe it at $l+(r-l)*(v-min)/(max-min)$

First probe is at $62(85 - 1)/(133 - 1) \approx 39$

Second probe is at $39(85 - 1)/(87 - 1) \approx 38$

First	Last	Probe	Entry	Outcome
0	62	39	87	<
0	39	38	85	=



Dictionary lookup:

When looking up a word in the dictionary, we instinctively use interpolation search

Interpolation Search

- Average Case: the search uses fewer than $\log_2 \log_2 n + 1$ comparisons. This function grows so slowly that the number of comparisons is a very small constant for all practically feasible inputs.

- Exercise: Find the smallest value of n for which $\log_2 \log_2 n + 1$ is greater than 6.

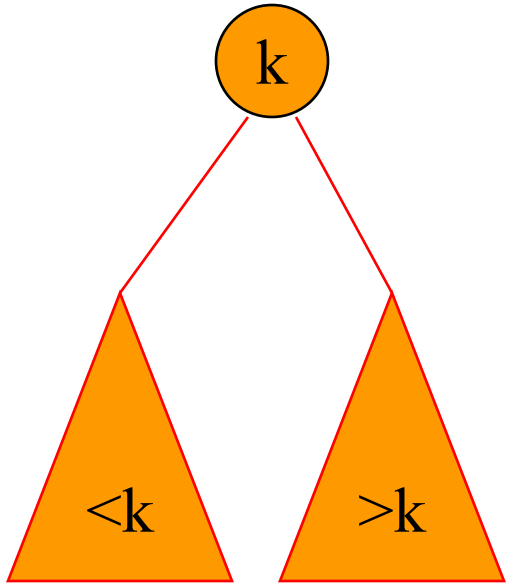
Interpolation Search

- Worse case: $O(n)$

Binary Search Tree Algorithms

Several algorithms on BST requires recursive processing of just one of its subtrees, e.g.,

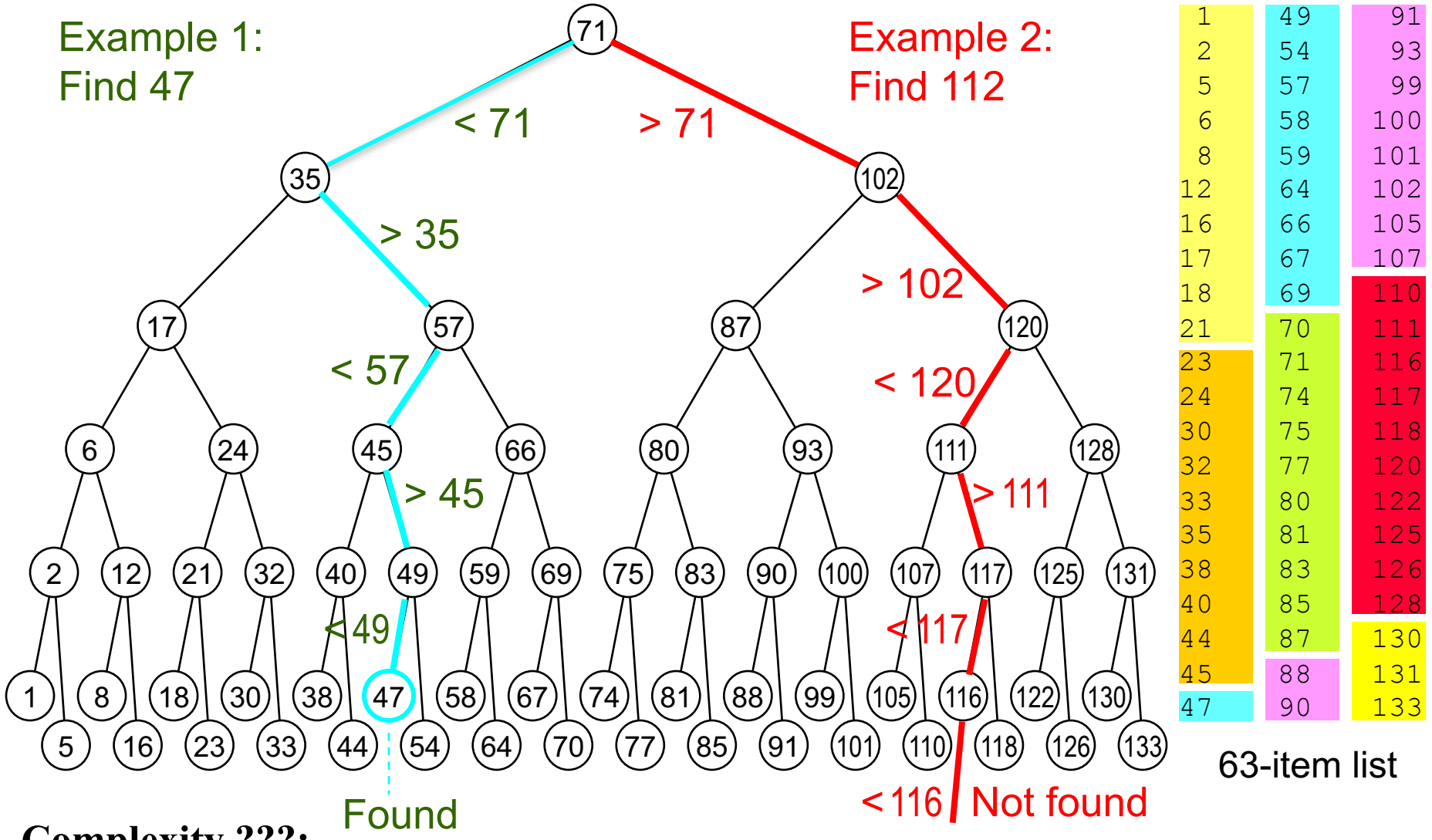
- Searching
- Insertion of a new key
- Finding the smallest (or the largest) key



Definition Of Binary Search Tree

- A binary tree.
- Each node has a (*key, value*) pair.
- For every node *x*, all keys in the left subtree of *x* are smaller than that in *x*.
- For every node *x*, all keys in the right subtree of *x* are greater than that in *x*.

Binary Search Trees



1 2 5 6 8 12 16 17 18 21 23 24 30 32 33 35 38 40 44 45 47
49 54 57 58 59 64 66 67 69 70 71 74 75 77 80 81 83 85 87 88 90
91 93 99 100 101 102 105 107 110 111 116 117 118 120 122 125 126 128 130 131 133

63-item list

Complexity ???:

Worst case: $\Theta(n)$, where n is number of nodes/elements

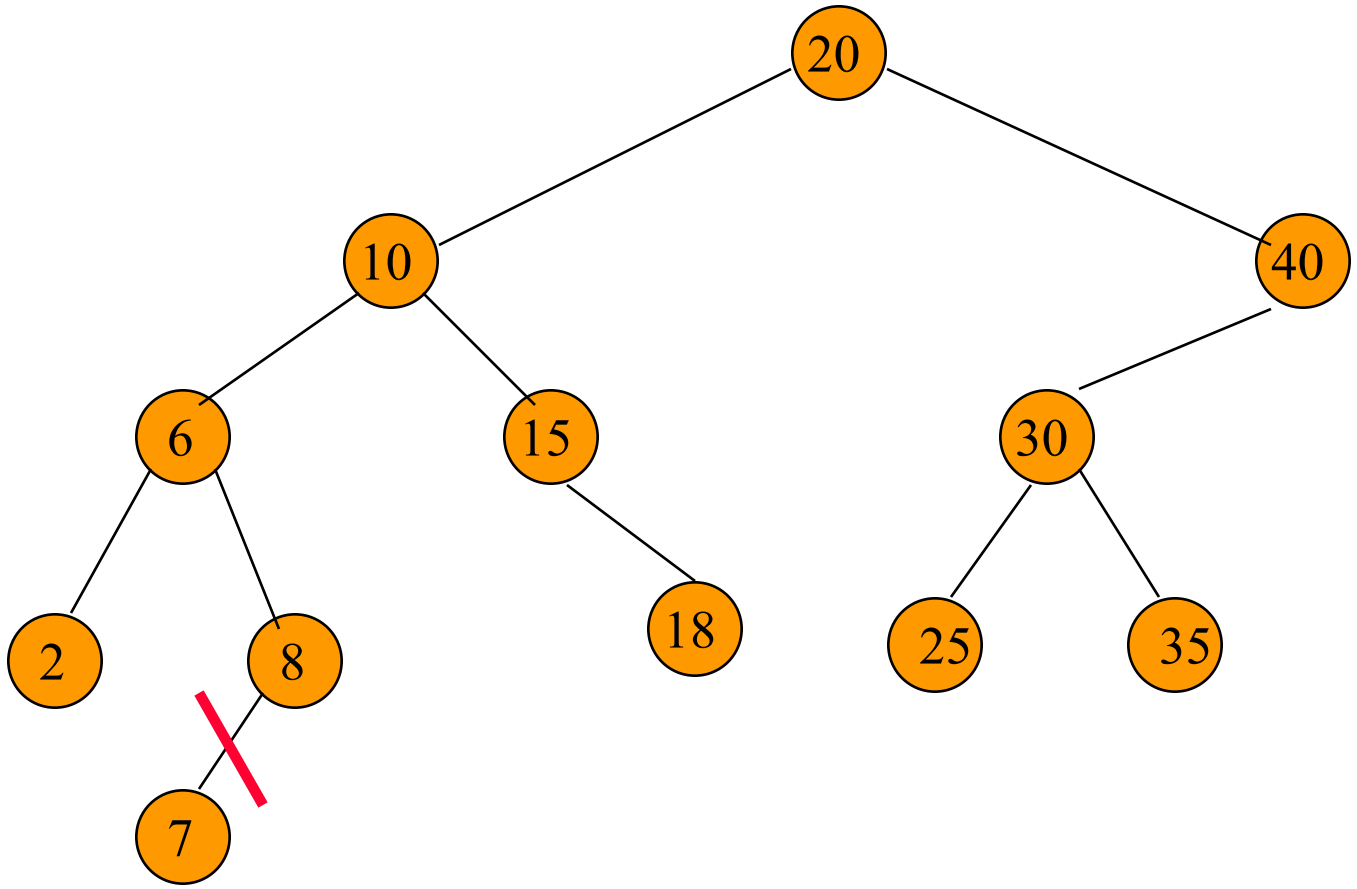
Average case: $\Theta(\log n)$

The Operation delete()

Three cases:

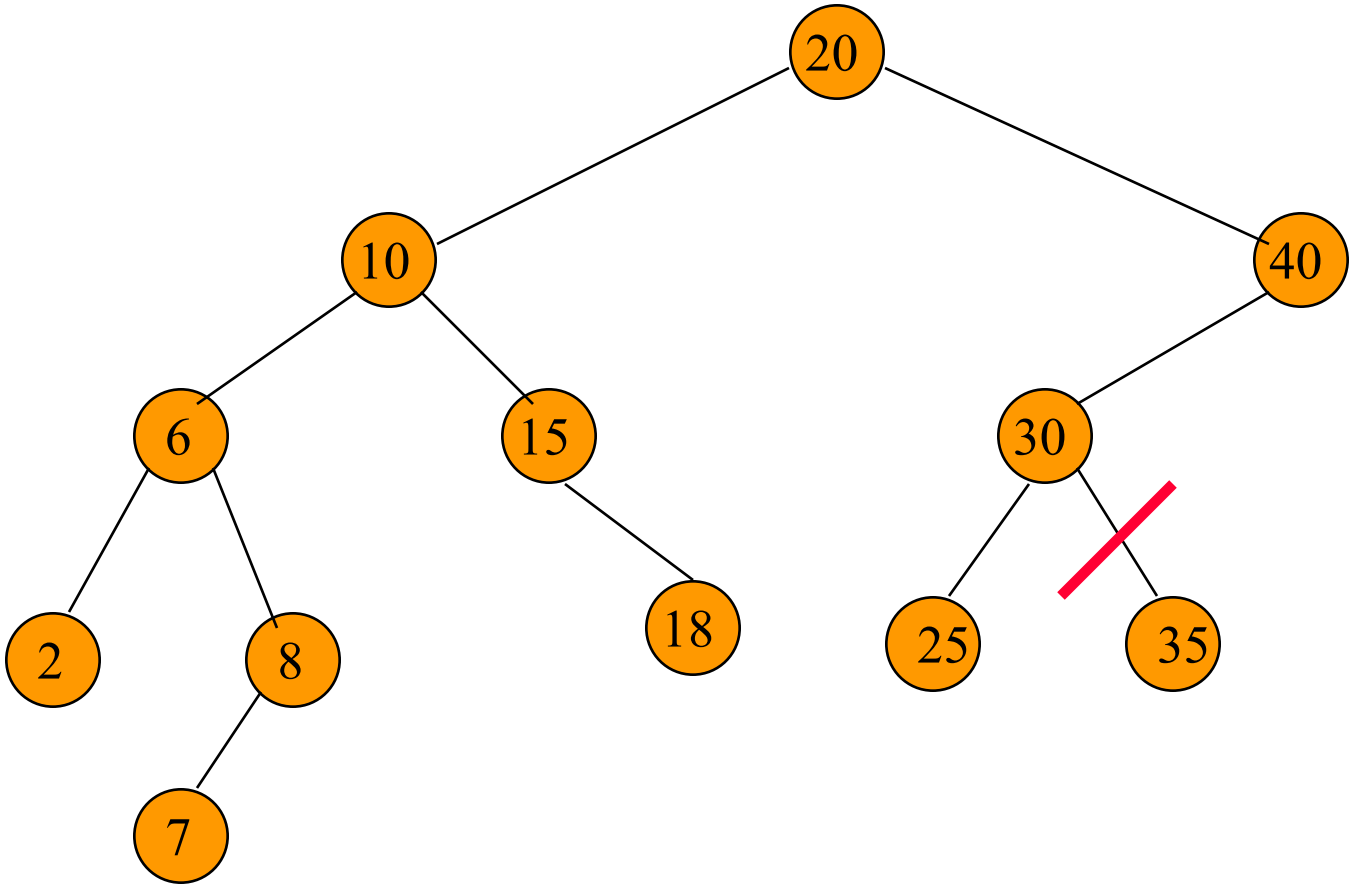
- Element is in a leaf.
- Element is in a degree 1 node.
- Element is in a degree 2 node.

Delete From A Leaf



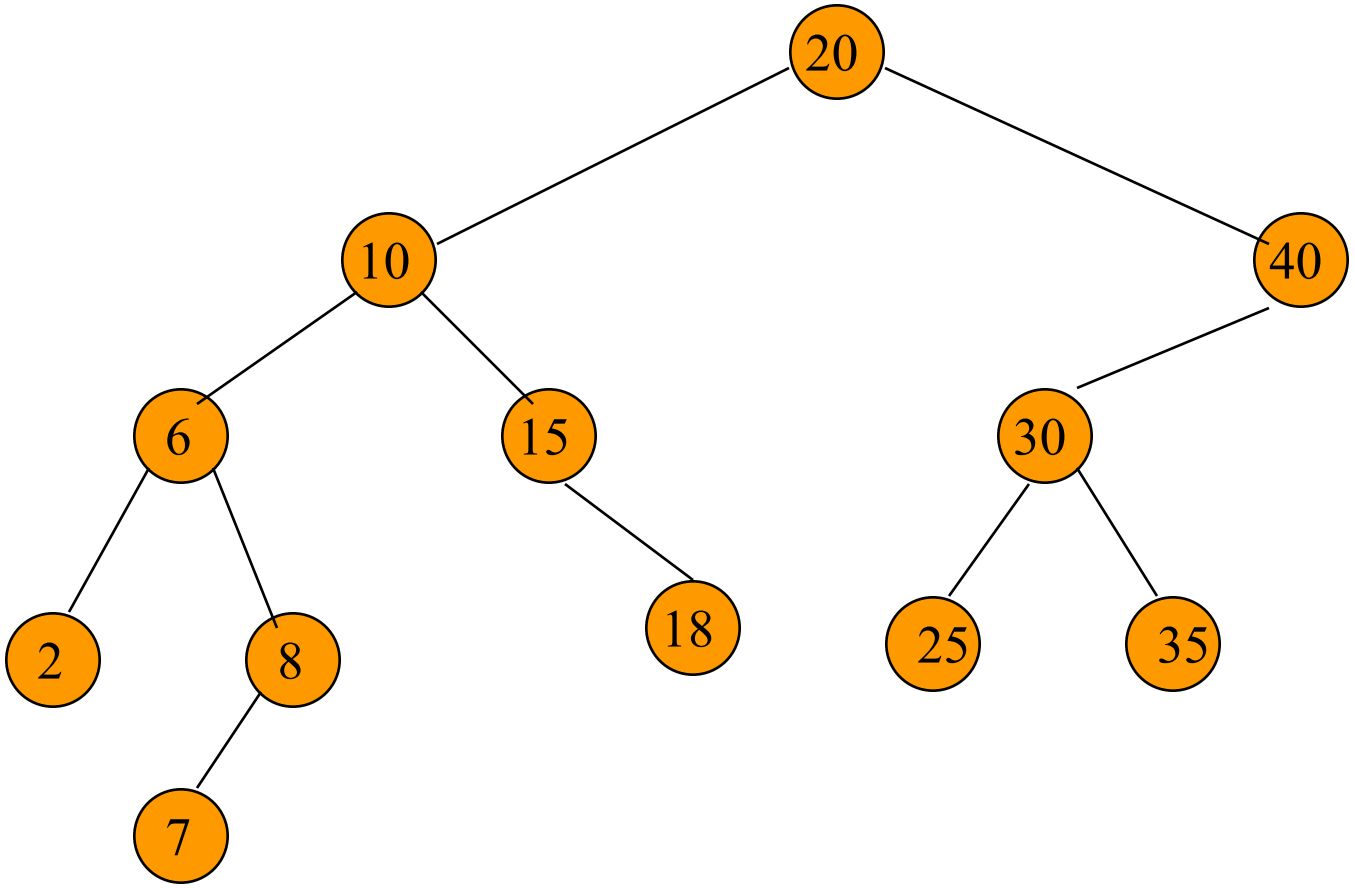
Delete a leaf element. key = 7

Delete From A Leaf (contd.)



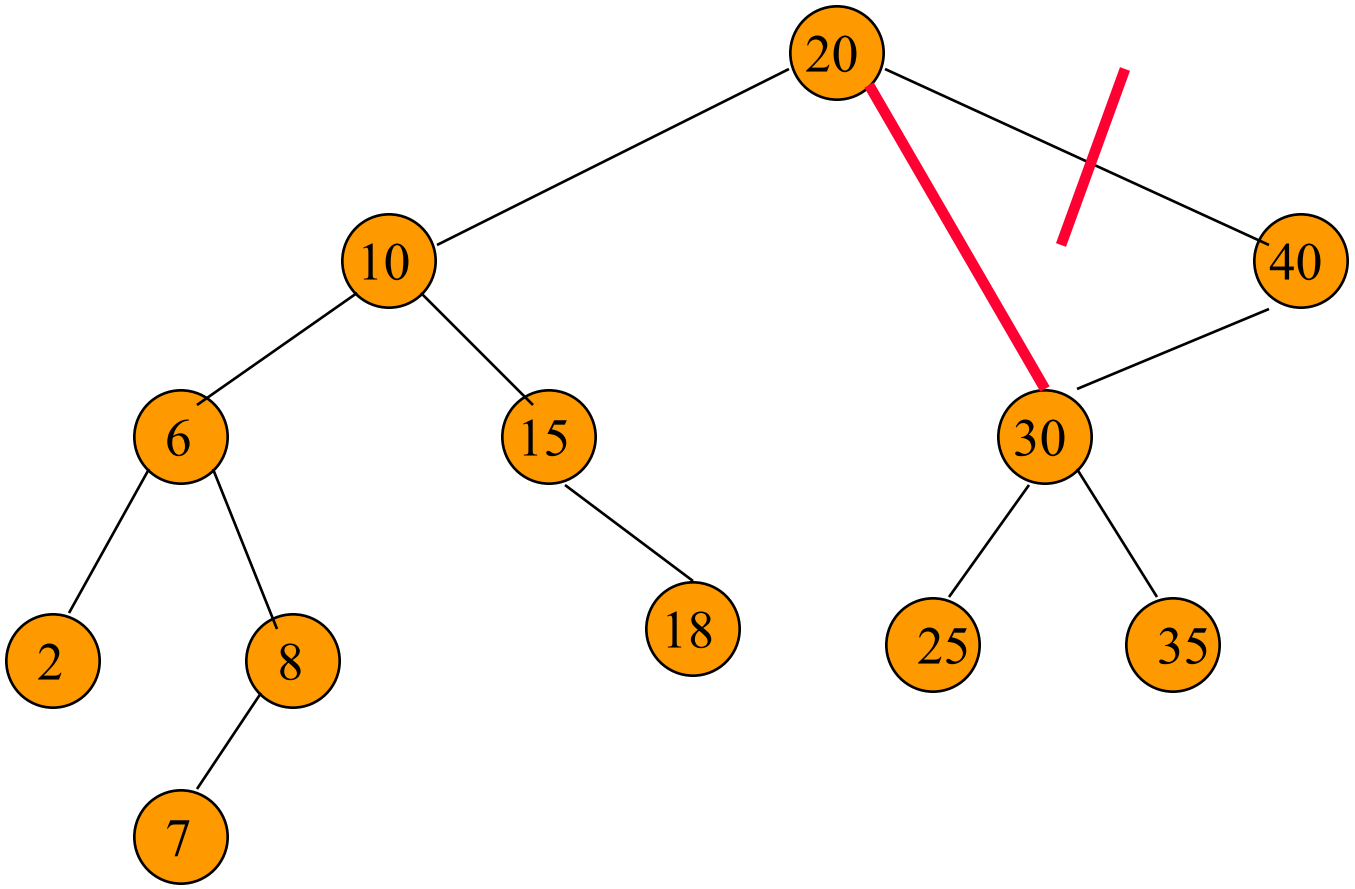
Delete a leaf element. key = 35

Delete From A Degree 1 Node



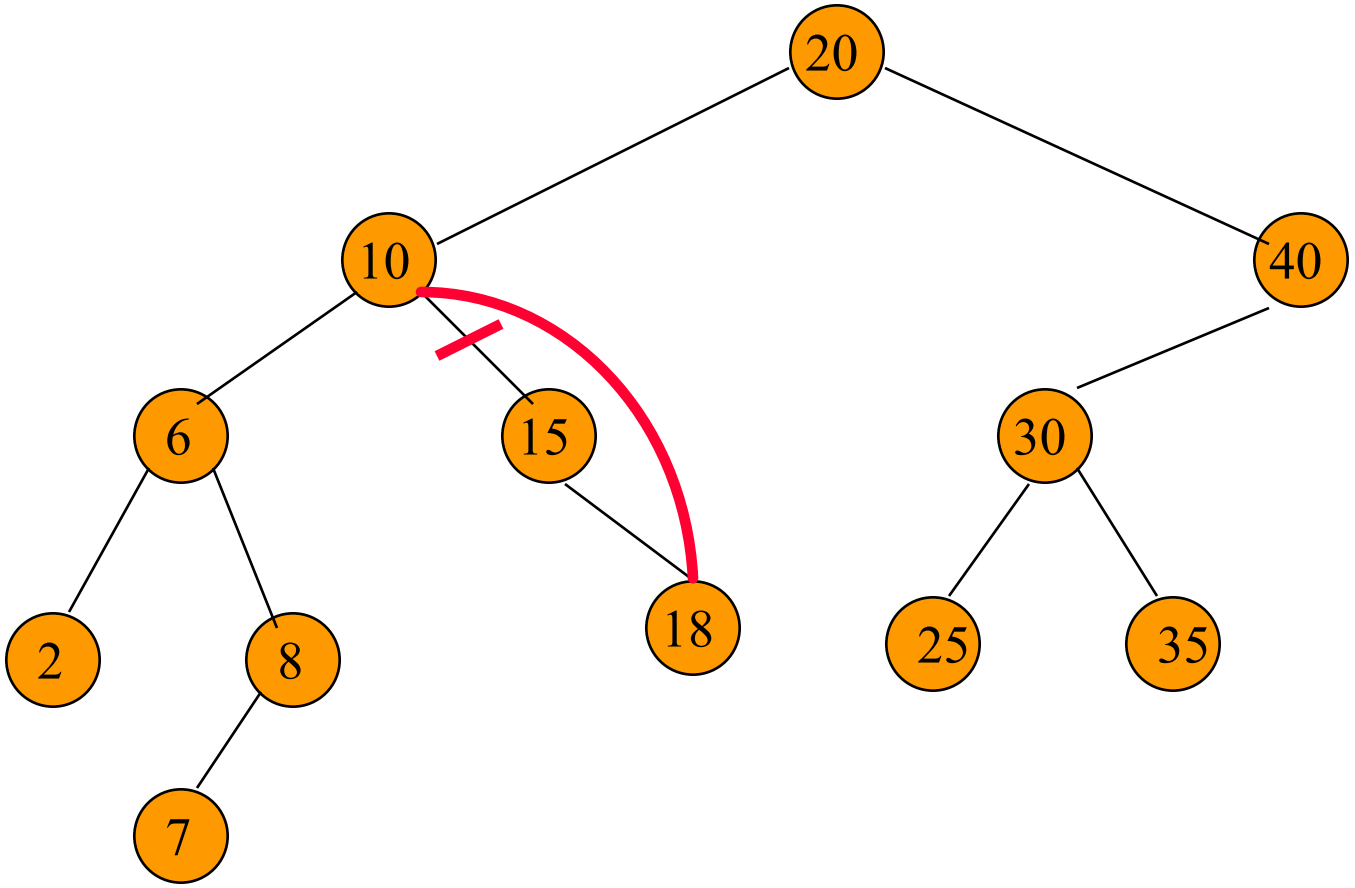
Delete from a degree 1 node. key = 40

Delete From A Degree 1 Node



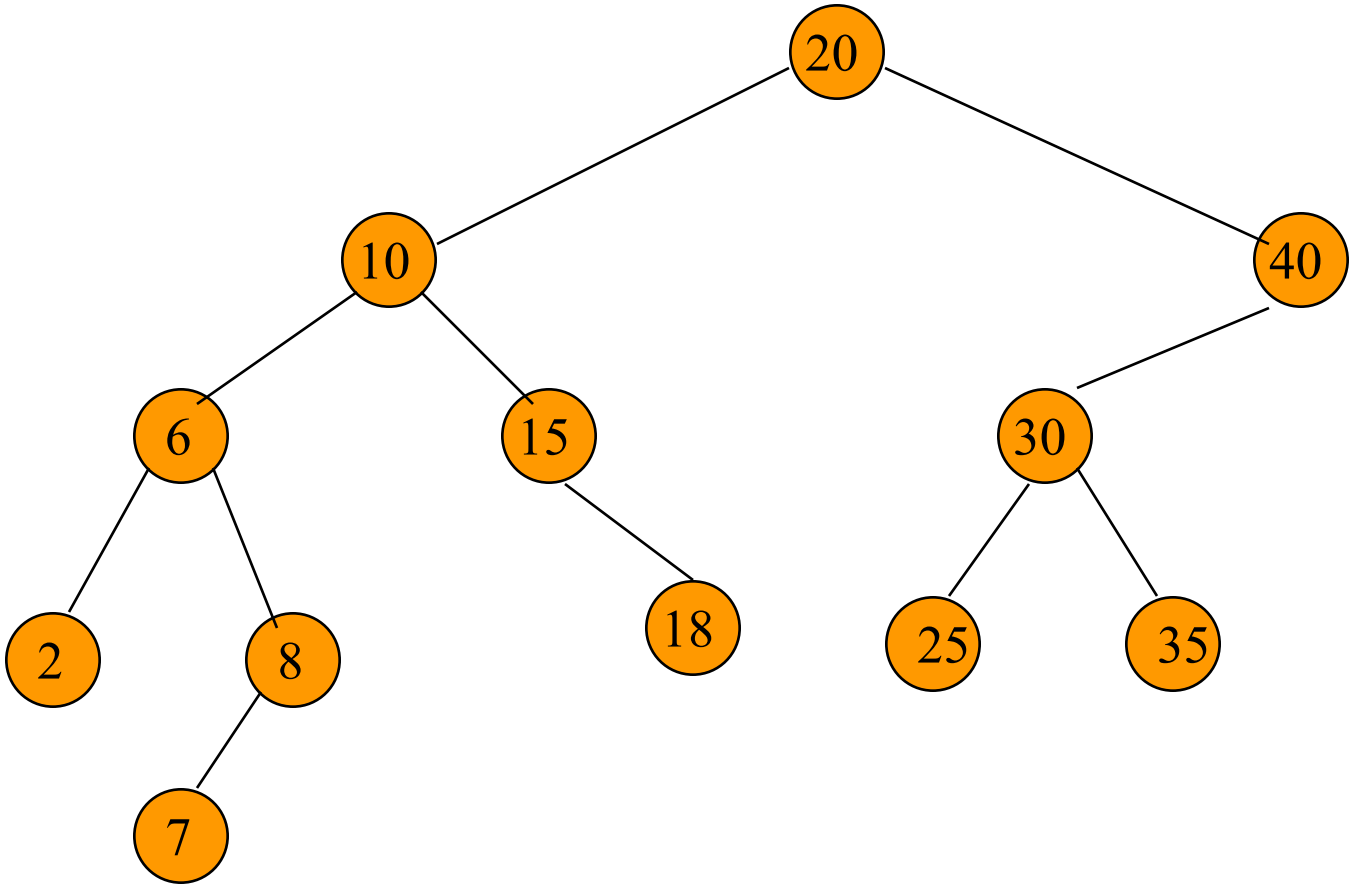
Delete from a degree 1 node. key = 40

Delete From A Degree 1 Node (contd.)



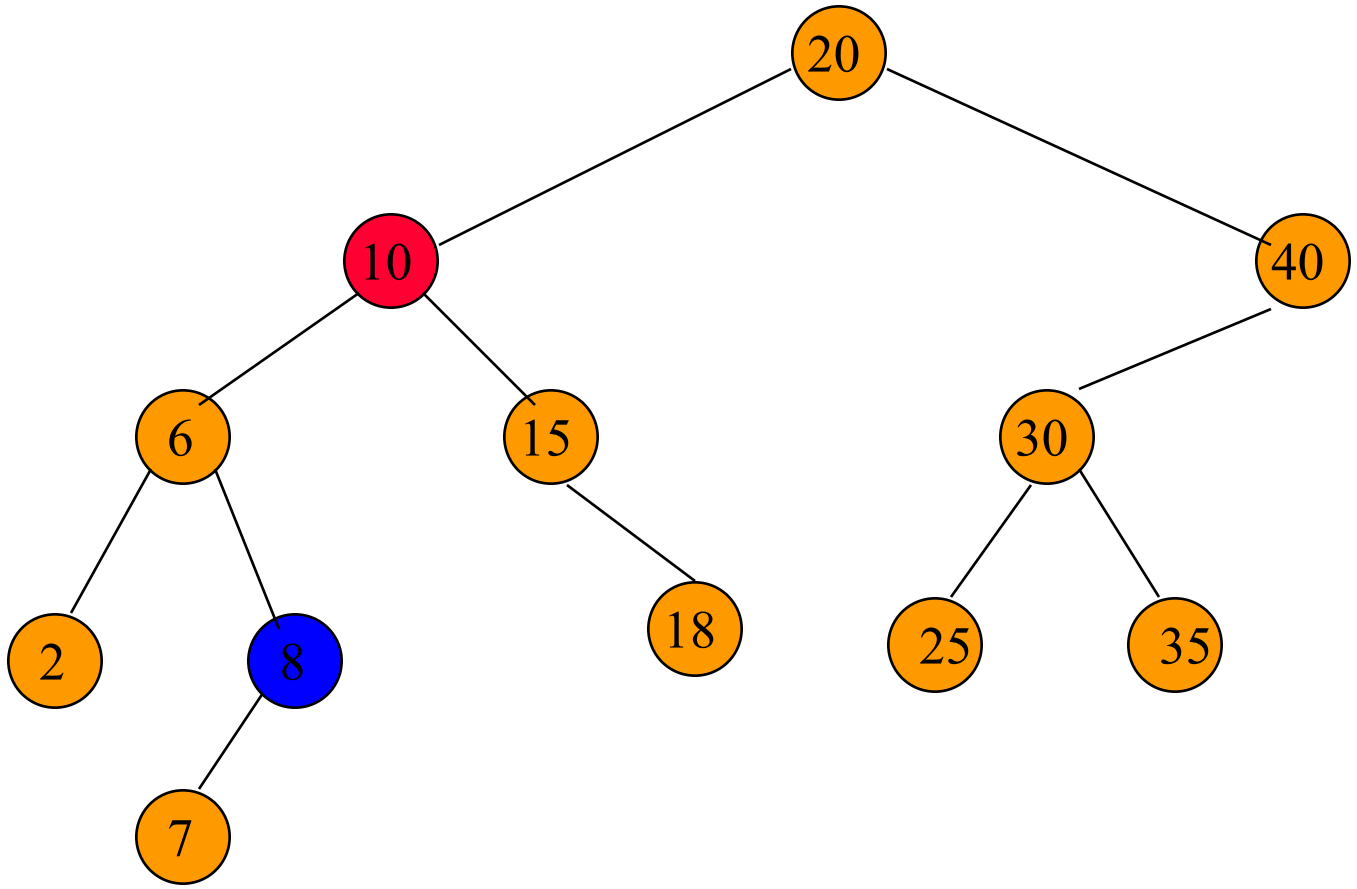
Delete from a degree 1 node. key = 15

Delete From A Degree 2 Node



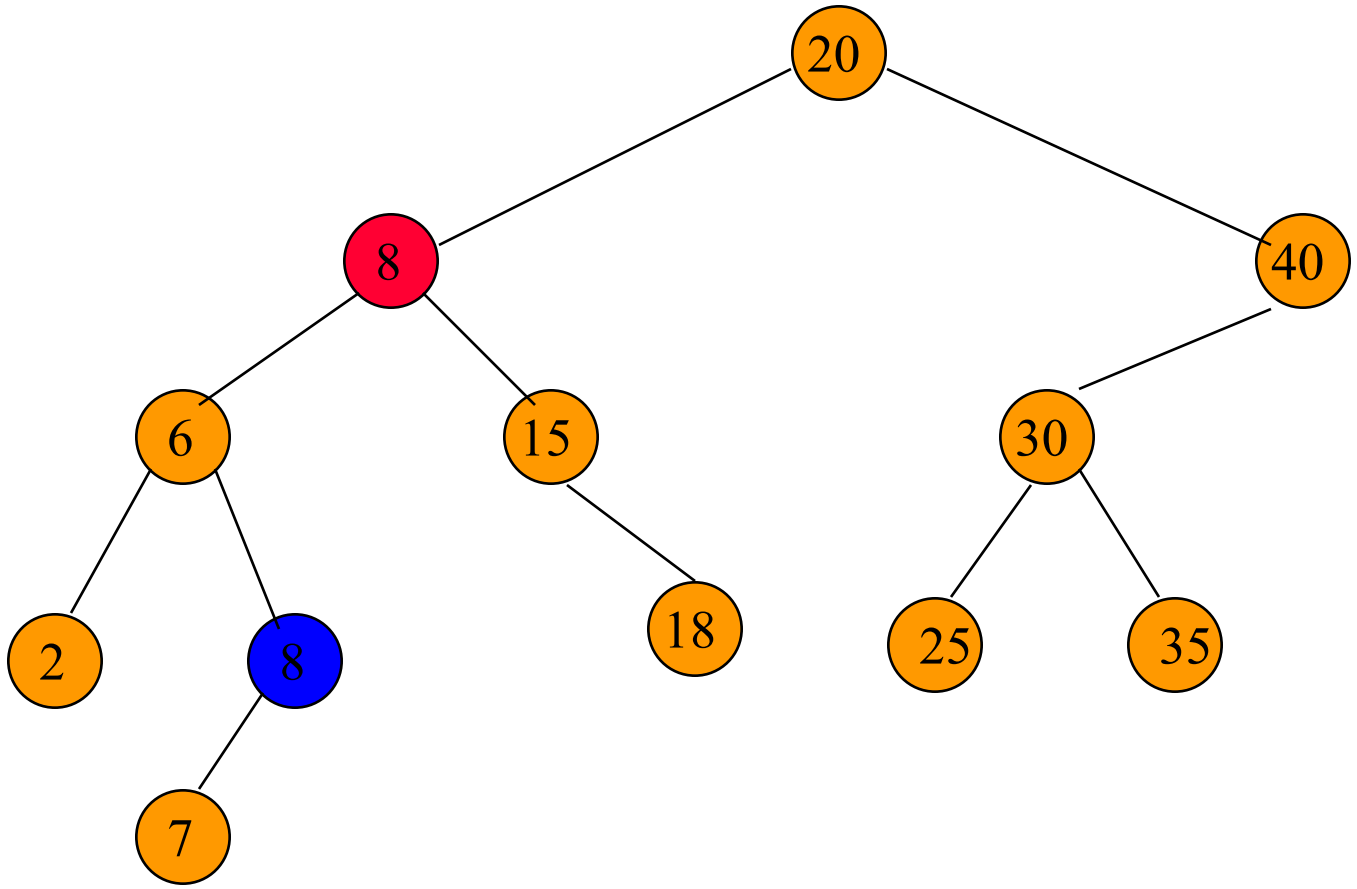
Delete from a degree 2 node. key = 10

Delete From A Degree 2 Node



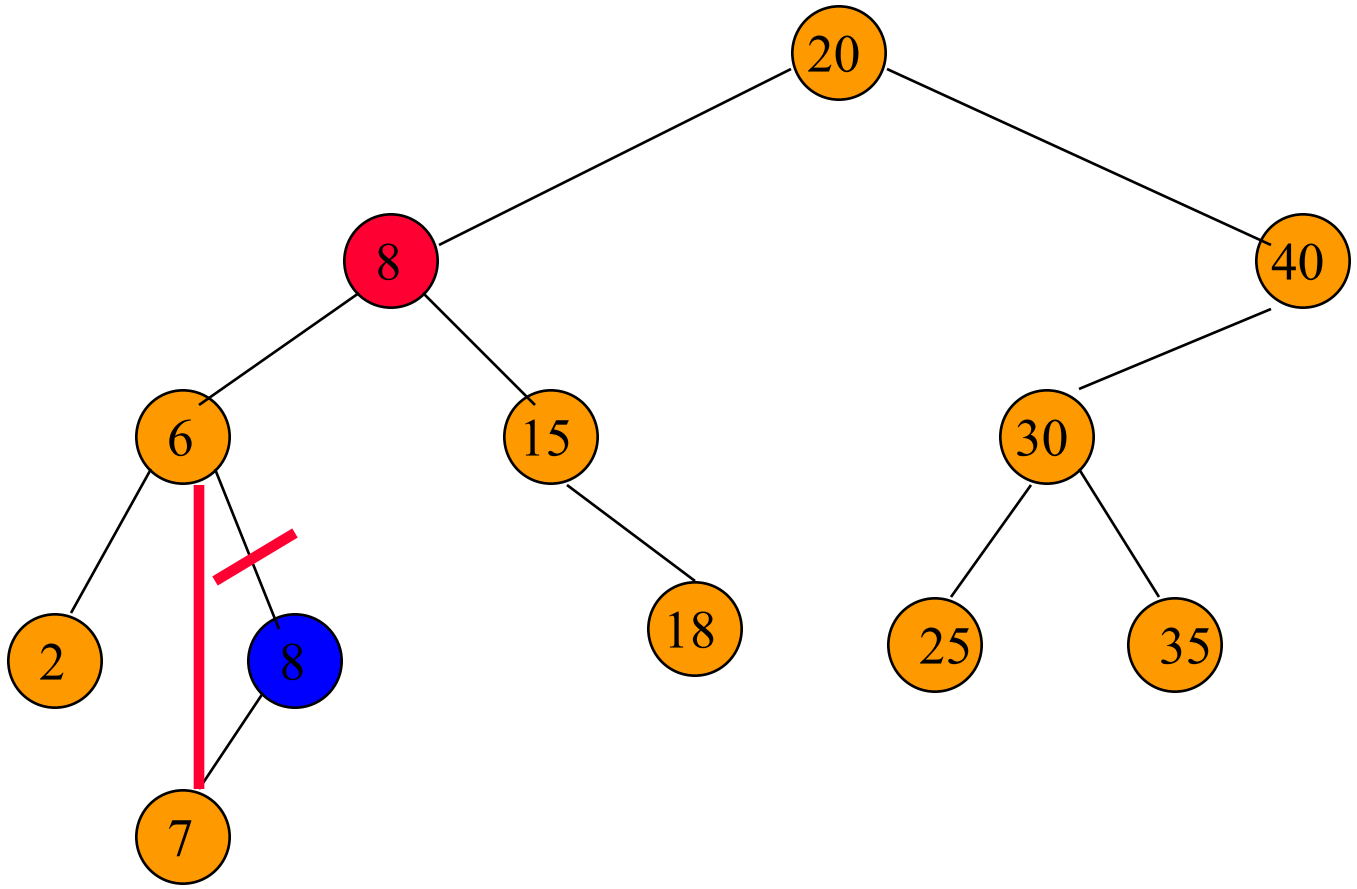
Replace with largest key in left subtree (or smallest in right subtree).

Delete From A Degree 2 Node



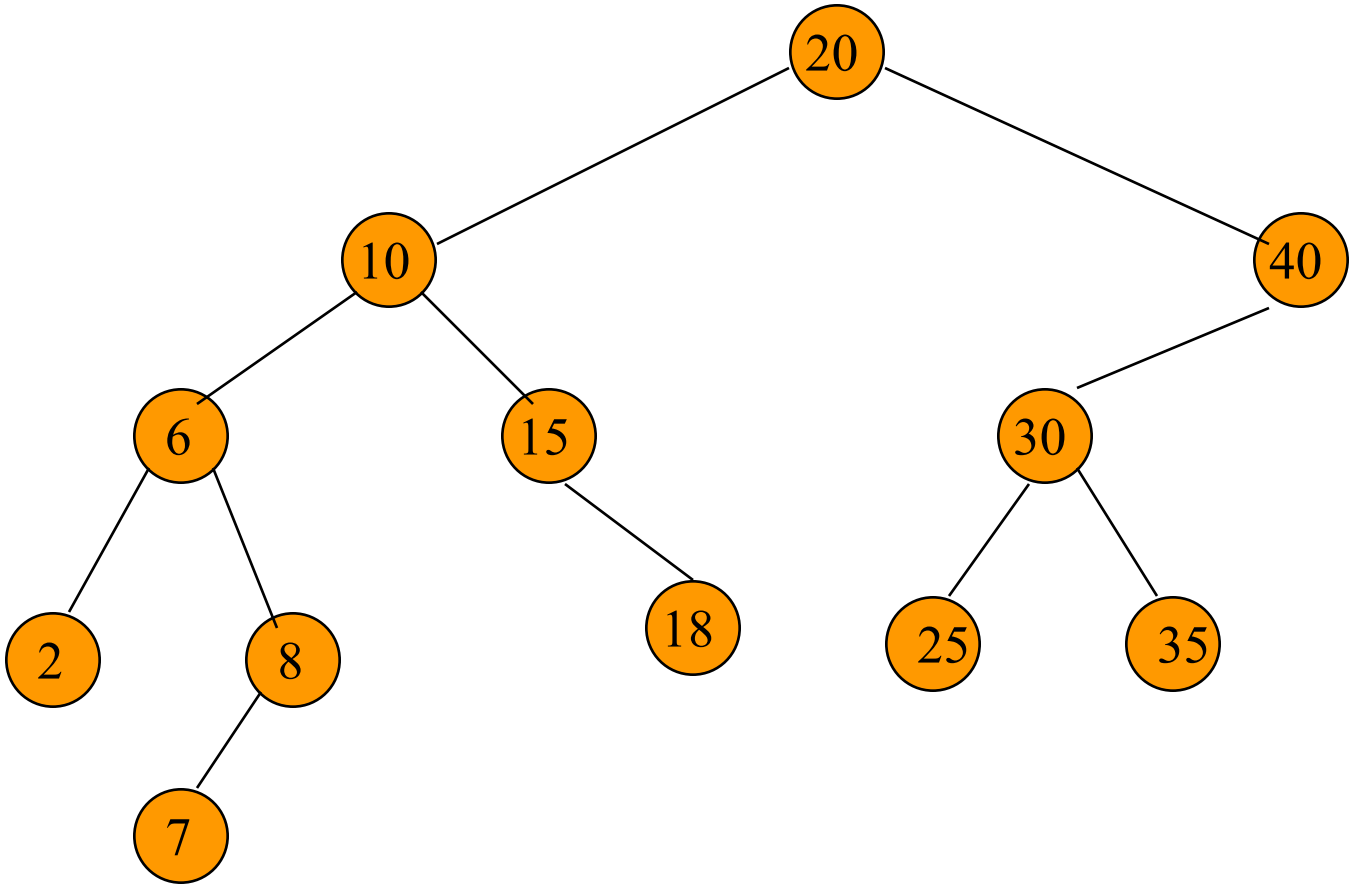
Replace with largest key in left subtree (or smallest in right subtree).

Delete From A Degree 2 Node



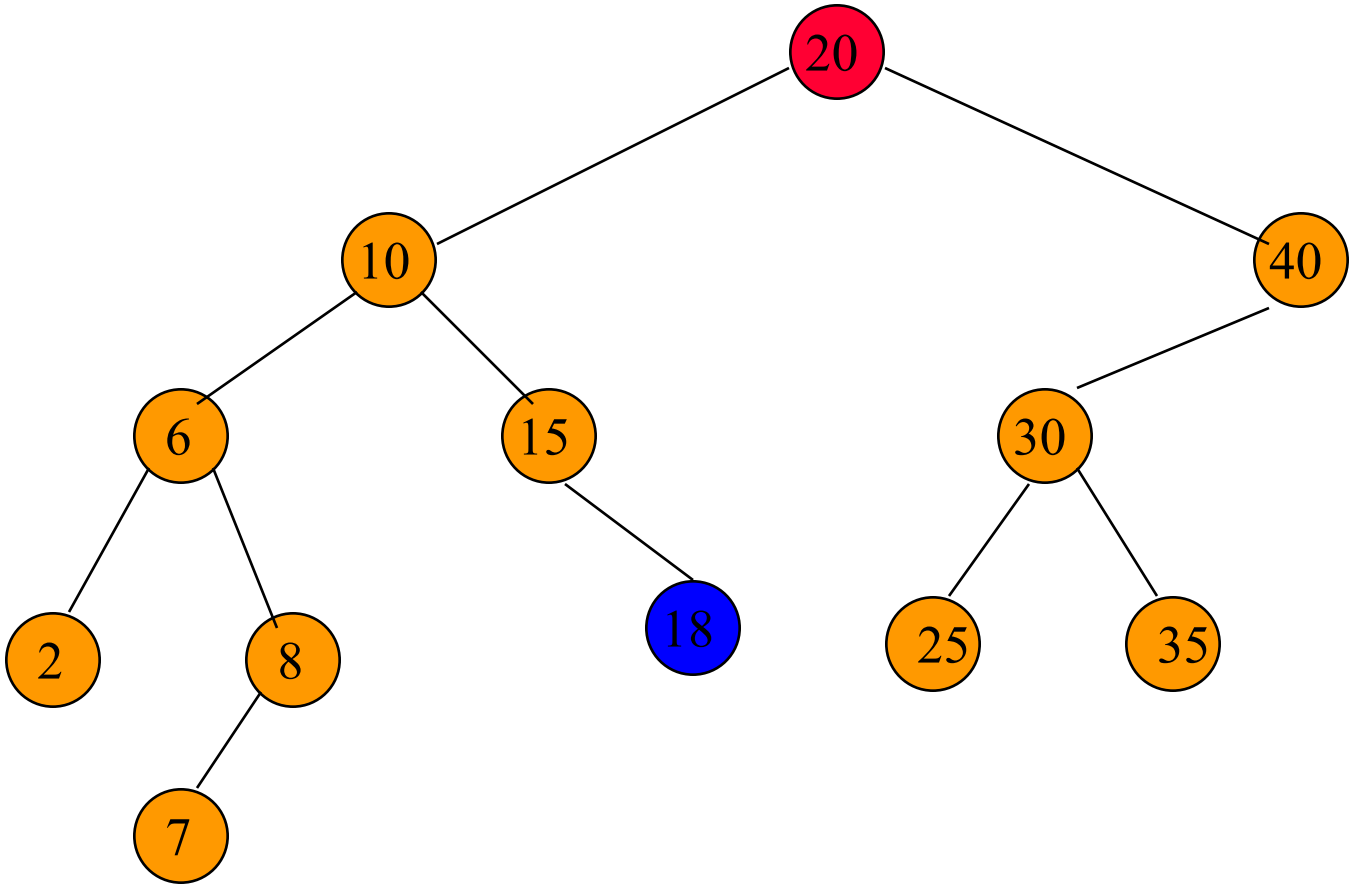
Largest key must be in a leaf or degree 1 node.

Another Delete From A Degree 2 Node



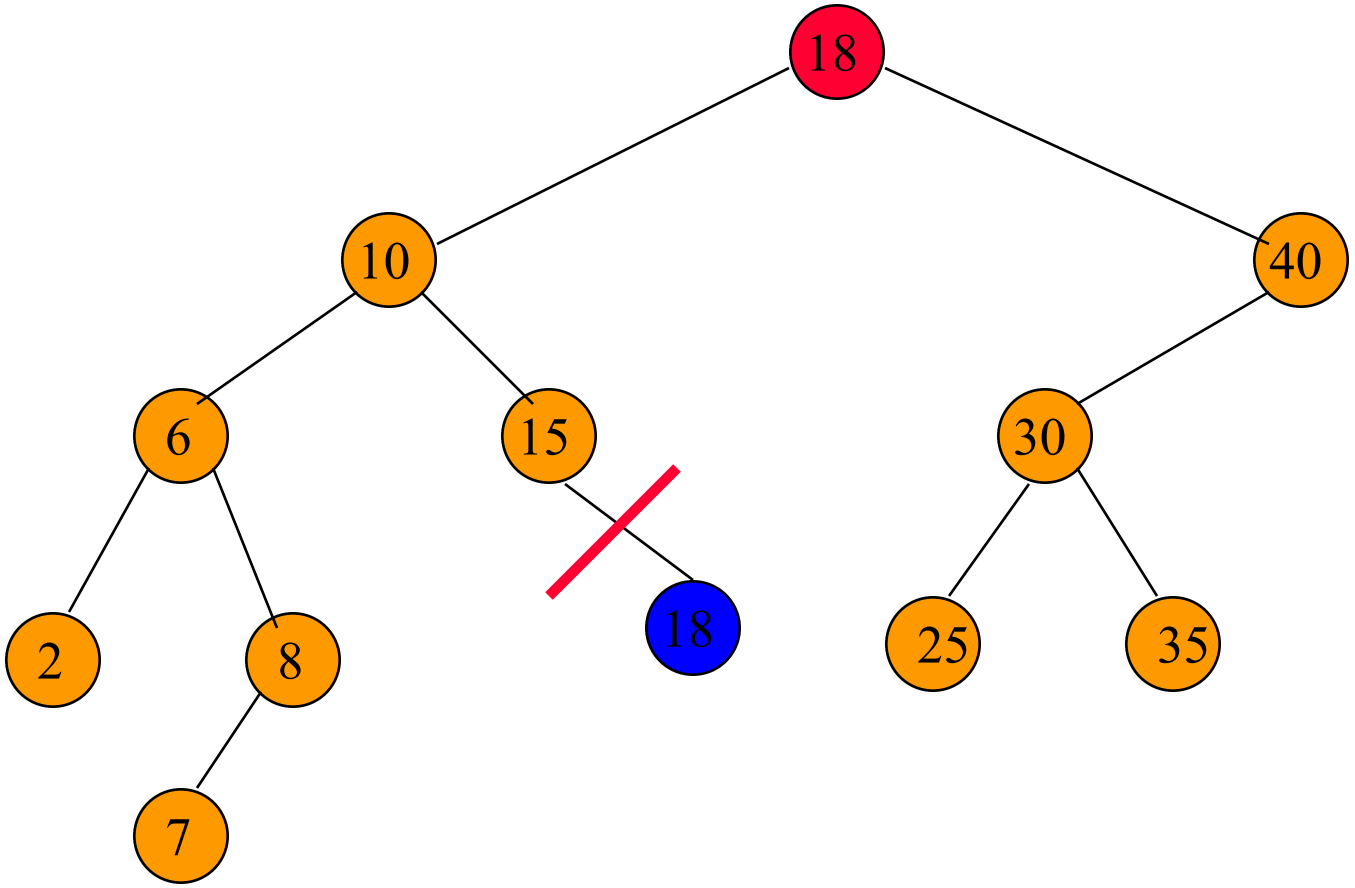
Delete from a degree 2 node. key = 20

Delete From A Degree 2 Node



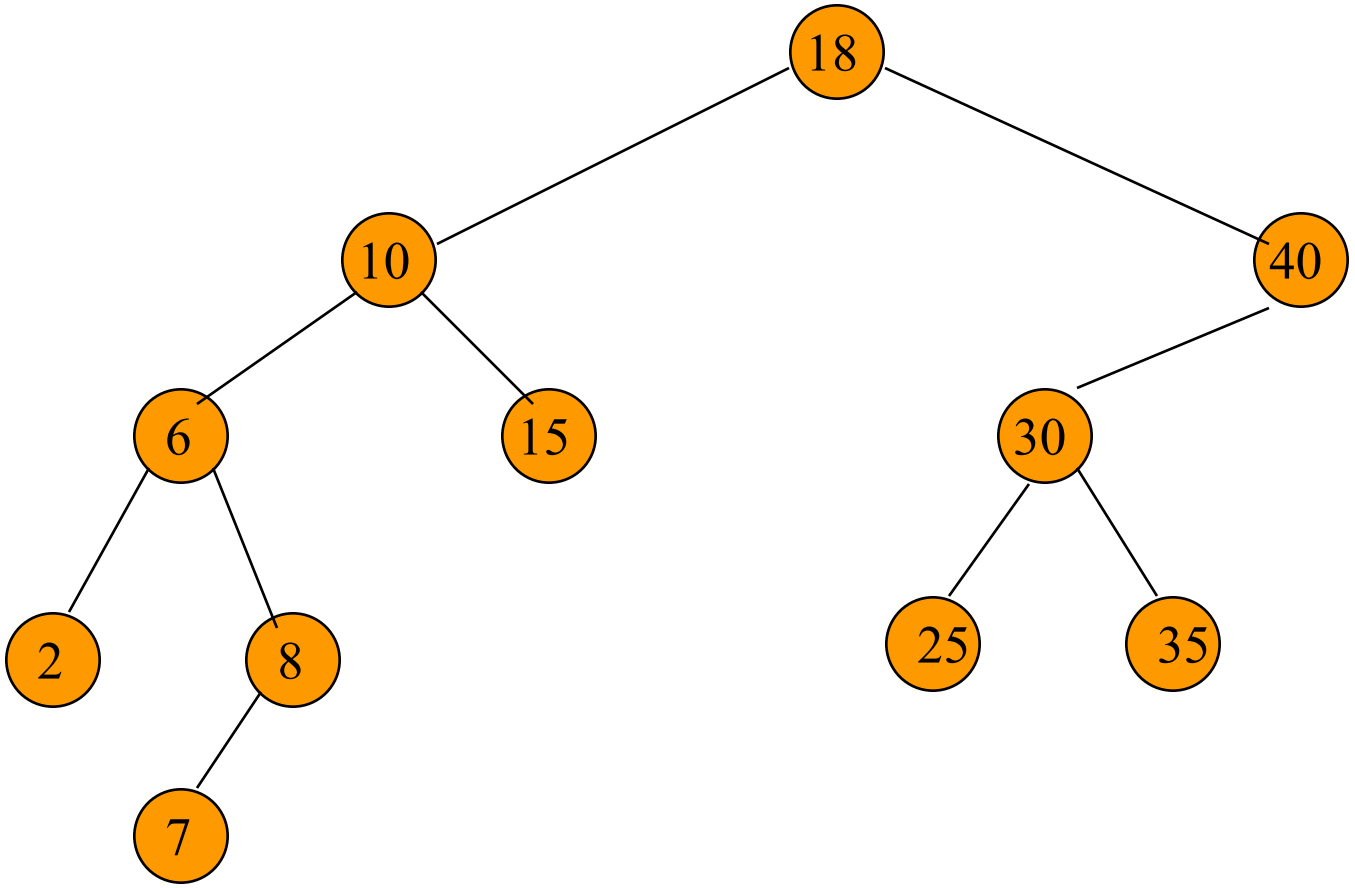
Replace with largest in left subtree.

Delete From A Degree 2 Node



Replace with largest in left subtree.

Delete From A Degree 2 Node



Complexity is $\Theta(height)$.

Exercise

- You need to search for a given number in an n × n matrix in which every row and every column is sorted in increasing order. Can you design an efficient algorithm for this problem?

Questions in the exam

- Give a Big-Oh analysis of the running time for program segments.
- Indicate whether big-Oh related questions are **True** or **False**.
- Given an algorithm, answer questions like:
  - What does it compute?
  - What is the basic operation?
  - How many times is the basic operation executed?
  - What is its efficiency class?
- Set up and solve recurrence relations.
- Graph-related problems and algorithms.
- Desing algorithms to solve problems
- Others.