CS 344

MIDTERM 1 REVIEW

TOPICS

- Big-oh notation
- Recurrence relations
  - Master theorem
  - Recursion trees
  - Substitution method
- Sorting:
  - insertion
  - selection
  - merge
  - counting
  - radix
  - quicksort

TOPICS

- Divide and conquer
  - Maximum subarray problem
  - Kadane's algorithm
  - Matrix multiplication
  - Multiplying n-bit numbers
  - Towers of Hanoi
  - Median finding
- Graphs:
  - DFS, BFS
  - Dijkstra's algorithm
  - Bellman-Ford algorithm
  - Minimum spanning trees
  - Kruskal & Prim

TOPICS

- Greedy algorithms:
  - Huffman encoding
  - Horn clauses, satisfiability
  - Set cover

ASYMPTOTIC NOTATION

- $f(n) = O(g(n)) : \exists c, \exists n_0, \forall n > n_0, 0 \leq f(n) \leq$
- $f(n) = \Omega(g(n)) : g(n) = O(f(n))$
- $f(n) = \Theta(g(n)) : f(n) = O(g(n)) \text{ and } g(n) = O$

RULES OF THUMB

- Constants can be omitted: $7n \to n$
- Higher exponents dominate: $n^3 + n^2 \to n^3$
- Exponentials dominate polynomials: $2^n + n^2 \to 2^n$
- Polynomials dominate logarithms: $n + \log n \to n$

DIVIDE AND CONQUER

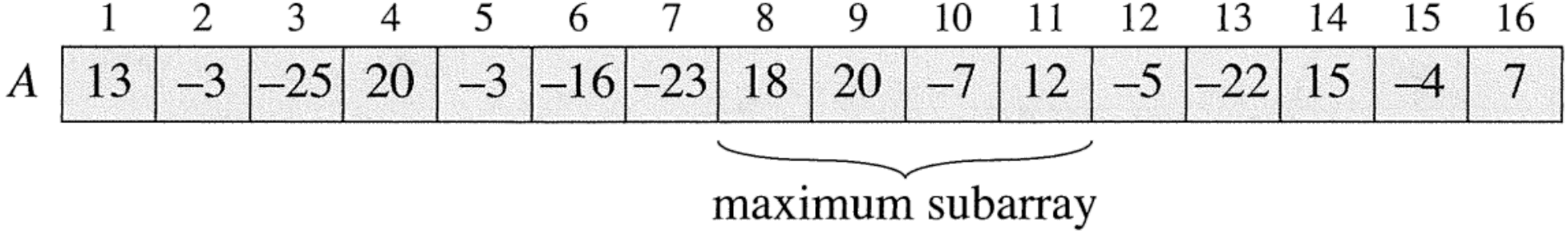
- divide: split into subproblems of the same type
- conquer: solve the subproblems recursively
- combine: combine the results into a solution for the original problem

MERGE SORT

$$T(n) = 2T(n/2) + O(n)$$

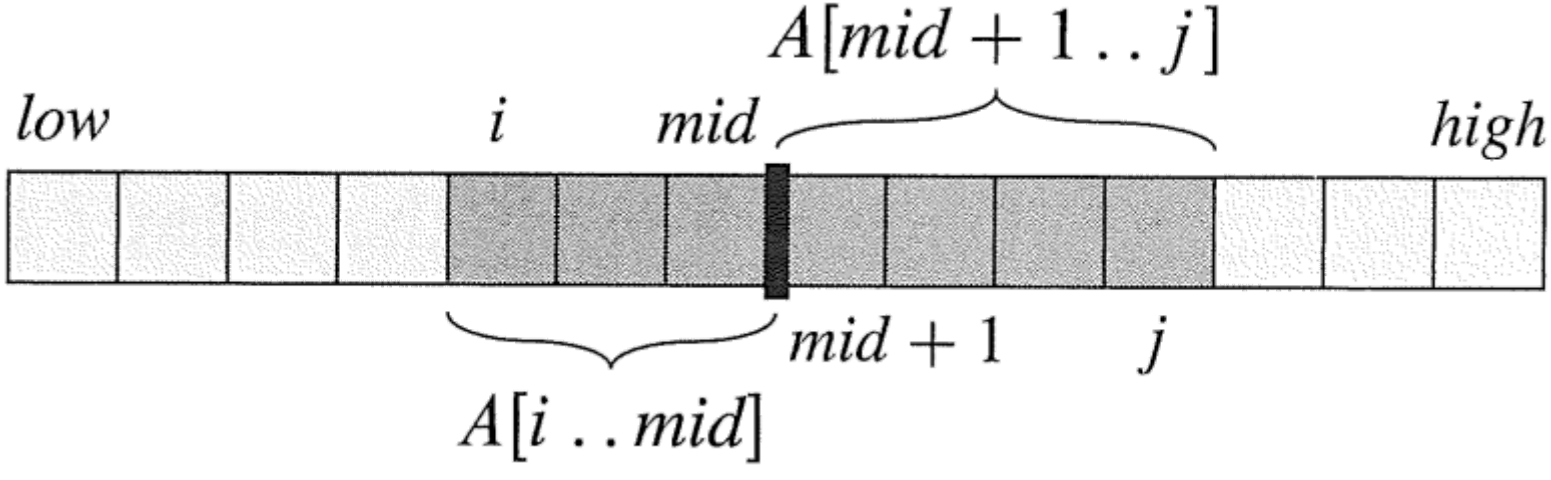
MAXIMUM SUBARRAY PROBLEM

We're looking for a subarray with the largest sum:



MAXIMUM SUBARRAY PROBLEM

If it crosses the midpoint, there must be a part in the left and a part in the right.



```
left-sum = −∞
sum = 0
for i = mid downto low
    sum = sum + A[i]
    if sum > left-sum
        left-sum = sum
        max-left = i
right-sum = −∞
sum = 0
for j = mid + 1 to high
    sum = sum + A[j]
    if sum > right-sum
        right-sum = sum
        max-right = j
return (max-left, max-right, left-sum + right-sum)
```

```
if high == low
    return (low, high, A[low])
else mid = ⌊(low + high)/2⌋
    (left-low, left-high, left-sum) =
        FIND-MAXIMUM-SUBARRAY(A, low, mid)
    (right-low, right-high, right-sum) =
        FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
    (cross-low, cross-high, cross-sum) =
        FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
    if left-sum ≥ right-sum and left-sum ≥ cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
        return (right-low, right-high, right-sum)
    else return (cross-low, cross-high, cross-sum)
```

How long does this take?

- base case: $T(1) = O(1)$
- recursion: 2 subproblems of size $n/2$
- finding the max crossing subarray: $O(n)$

How long does this take?

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ 2T(n/2) + O(n) & \text{if } n > 1 \end{cases}$$

Same as merge sort! $O(n \log n)$

MASTER THEOREM

$$T(n) = aT(n/b) + O(n^d), a > 0, b > 1, d \geq 0:$$

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

- binary search: $T(n) = T(n/2) + O(1)$
- merge sort: $T(n) = 2T(n/2) + O(n)$

MATRIX MULTIPLICATION

$n = A.rows$
let C be a new $n \times n$ matrix
if $n == 1$
    $c_{11} = a_{11} \cdot b_{11}$
else partition A, B, and C as in equations (4.9)
    $C_{11} =$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{11}, B_{11})
        $+$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{12}, B_{21})
    $C_{12} =$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{11}, B_{12})
        $+$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{12}, B_{22})
    $C_{21} =$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{21}, B_{11})
        $+$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{22}, B_{21})
    $C_{22} =$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{21}, B_{12})
        $+$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{22}, B_{22})
return C

- Divide into 8 blocks of size $n/2 \times n/2$
- Recursively multiply
- Add (some of) the resulting matrices

Then the time required has the form:

$$T(n) = 8T(n/2) + O(n^2)$$

STRASSEN

$$T(n) = 7T(n/2) + O(n^2)$$

$$\log_2 7 \approx 2.807 > d$$

MULTIPLY TWO n-BIT NUMBERS

$$(2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) =$$

$$2^n x_L y_L + 2^{n/2}x_R y_L + 2^{n/2}x_L y_R + x_R y_R =$$

$$2^n x_L y_L + 2^{n/2}(x_R y_L + x_L y_R) + x_R y_R$$

$$T(n) = 4T(n/2) + O(n)$$

Gauss:

$$(2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) =$$
$$2^n x_L y_L + 2^{n/2}(x_R y_L + x_L y_R) + x_R y_R$$

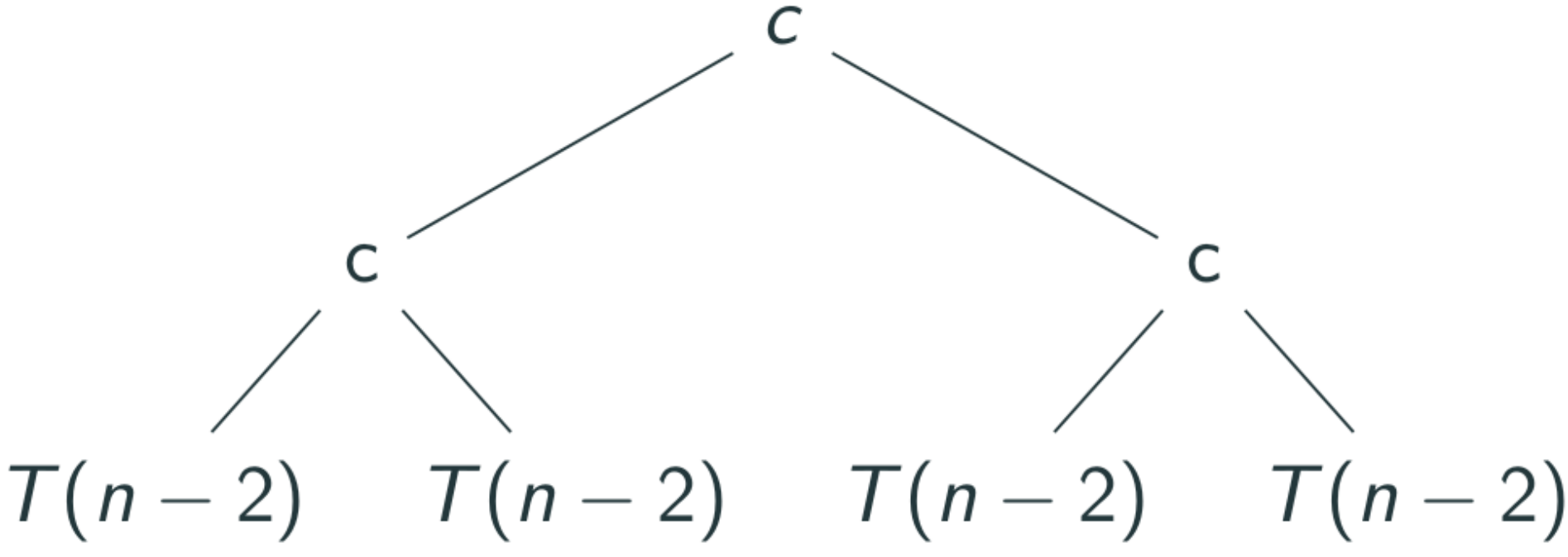
So

$$x_R y_L + x_L y_R = (x_L + x_R)(y_L + y_R) - x_L y_L - x_$$

TOWER OF HANOI

$$T(n) = 2T(n-1) + O(1)$$

RECURSION TREES



KADANE'S ALGORITHM

```
def maxSubarray(numbers):
    bestSum = 0
    currSum = 0
    for x in numbers:
        currSum = max(0, currSum + x)
        bestSum = max(bestSum, currSum)
    return bestSum
```

QUICKSORT

- Pick a pivet value $A[q]$
- Partition A into A_1 and A_2 such that the first is all $\leq A[q]$ and the second is all $\geq A[q]$
- Recursively sort the two subarrays

What is the running time (worst case)?

- Partitioning gives a maximally unbalanced set of regions
- Since we use one element as the pivot, $n - 1$ elements remain unsorted

$$T(n) = T(n-1) + O(n) = O(n^2)$$

MEDIANS

Let's pick some v and split the list into three new lists:

- Values less than v (call this S_L)
- Values equal to v (call this S_v)
- Values greater than v (call this S_R)

So we can recursively search one of these three new arrays based on k and the three sizes:

```
def selection(S, k):
    # partition S into SL, SV, SR
    if k < len(SL):
        return selection(SL, k)
    elif k < len(SL) + len(SV):
        return v
    else:
        return selection(SR, k - len(SL) - len(SV))
```

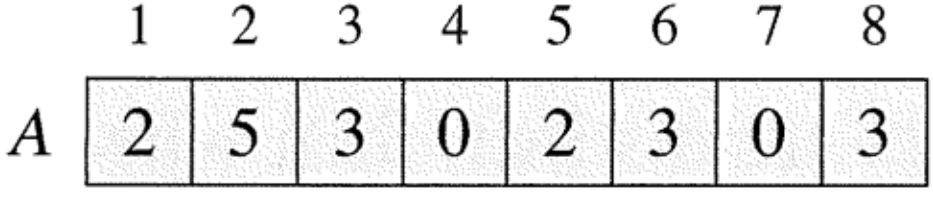
If we can pick a v that splits the list evenly:

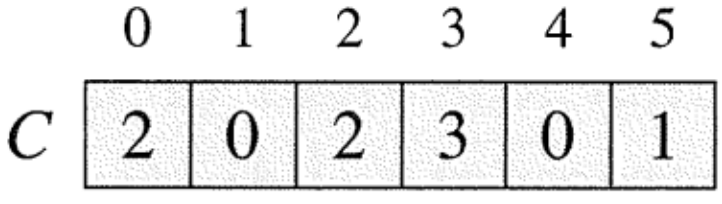
$$T(n) = T(n/2) + O(n)$$

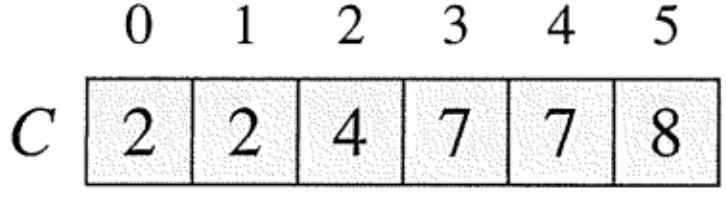
COUNTING SORT

The idea behind counting sort:

- For each element x of the input,
- figure out how many elements are less than x.
- That tells us where x belongs in the sorted array. For example, if there are 17 elements smaller than x, then x should be placed at $A[18]$.


1 2 3 4 5 6 7 8
A
2 5 3 0 2 3 0 3

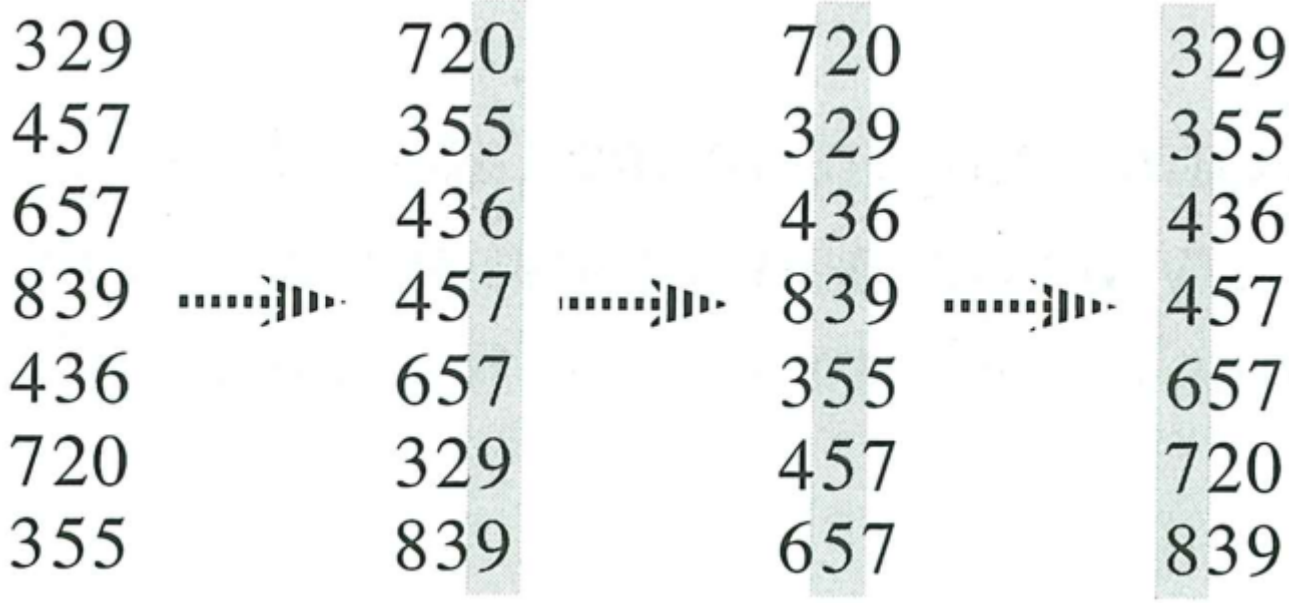

0 1 2 3 4 5
C
2 0 2 3 0 1


0 1 2 3 4 5
C
2 2 4 7 7 8

RADIX SORT

Suppose we have n numbers to sort, each d-digits long.

- Sort by one column of digits at a time


329
457
657
839
436
720
355
720
355
436
457
657
329
839
720
329
436
839
355
457
657
329
355
436
457
657
720
839

Worst-case running times:

- insertion: $O(n^2)$
- selection: $O(n^2)$
- merge: $O(n \log n)$
- counting: $O(k + n)$
- radix: $O(d(n + k))$
- quicksort: $O(n^2)$

GRAPHS

A graph is a pair $< V, E >$:

- V: vertices
- E: edges, where $E \subseteq V \times V$

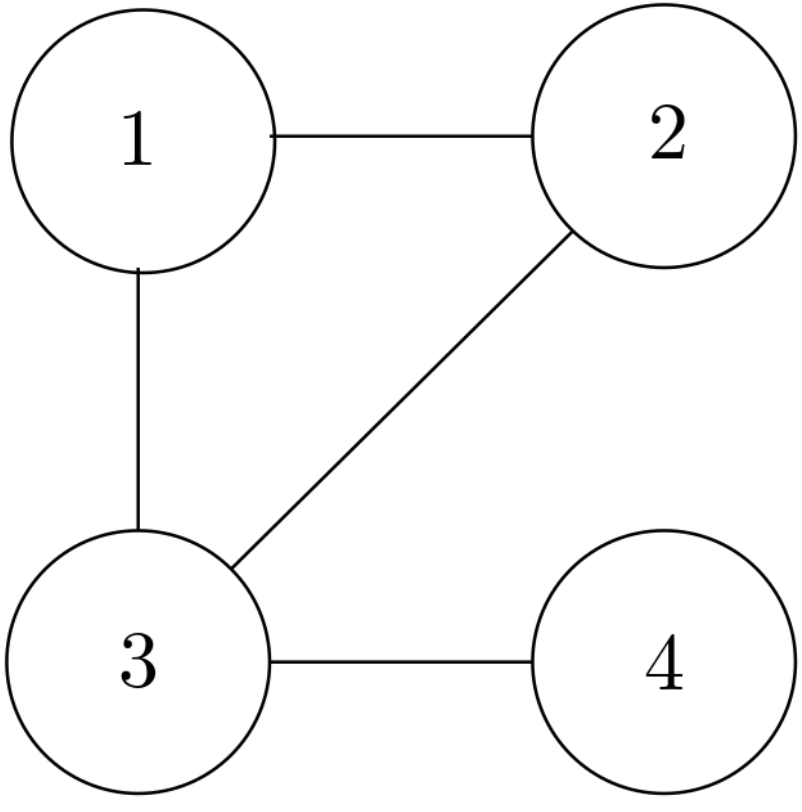
ADJACENCY MATRIX

If we have $|V| = n$ vertices,

the adjacency matrix is an $n \times n$ matrix:

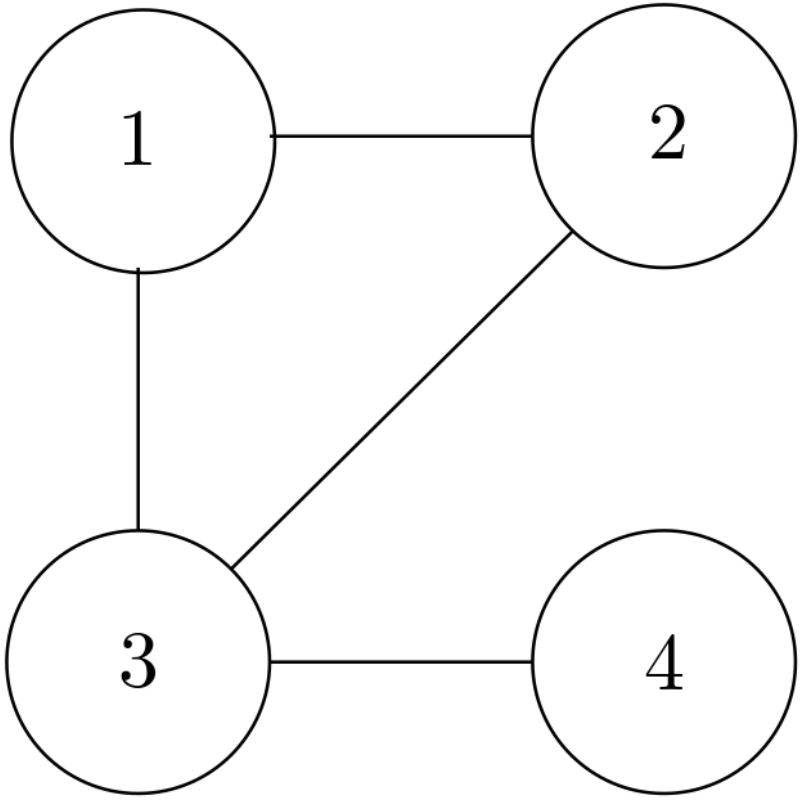
$$a_{ij} = \begin{cases} 1 & \text{if there is an edge from } v_i \text{ to } v_j \\ 0 & \text{otherwise} \end{cases}$$

ADJACENCY MATRIX



$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

ADJACENCY LIST



Vertex	List
1	2, 3
2	1, 3
3	1, 2, 4
4	3

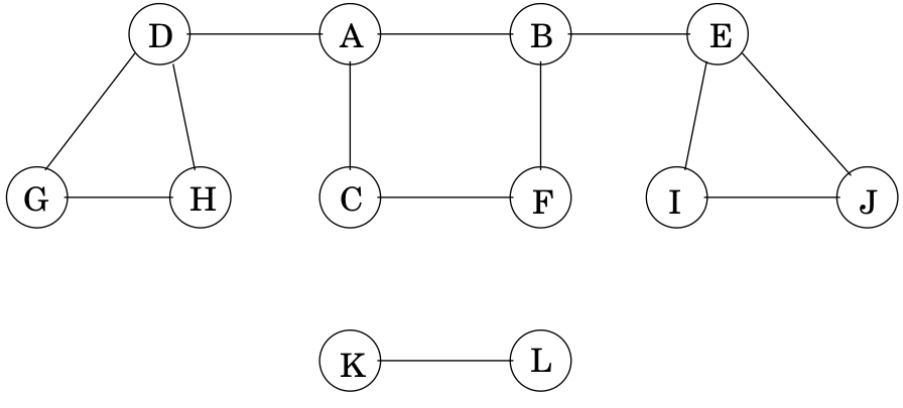
ADJACENCY LIST

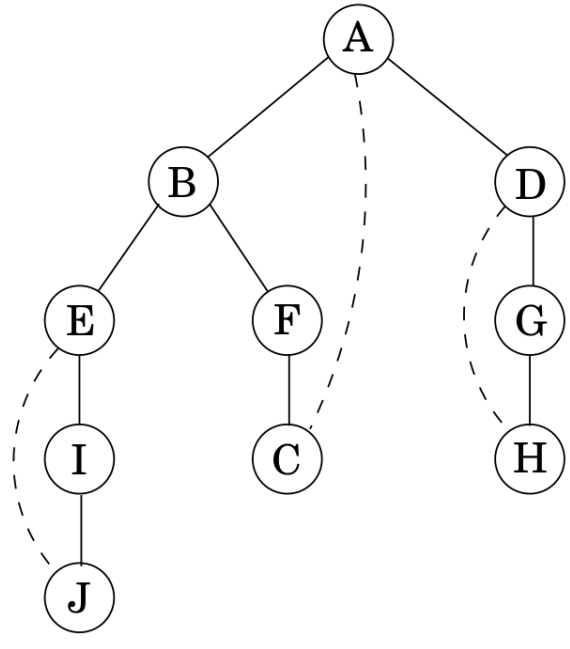
- Have to walk a list to check for an edge
- Takes $O(|E|)$ space

Let's find all nodes reachable from a particular node:

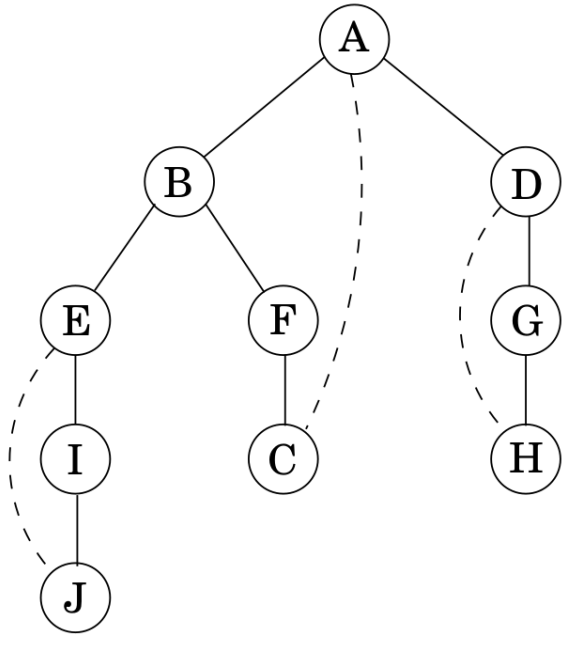
```
procedure explore(G, v)
Input:    G = (V, E) is a graph; v ∈ V
Output:   visited(u) is set to true for all nodes u reachable from v

visited(v) = true
previsit(v)
for each edge (v, u) ∈ E:
   if not visited(u):  explore(u)
postvisit(v)
```

Let's run explore(A) on the graph on the left:





Then we have two kinds of edges:



- tree edges: black lines
- back edges: dotted lines

DEPTH-FIRST SEARCH

The explore function only visits nodes reachable from the starting point

To examine the rest of the graph, we can repeatedly call explore

```
procedure dfs(G)

for all v ∈ V:
    visited(v) = false

for all v ∈ V:
    if not visited(v):  explore(v)
```

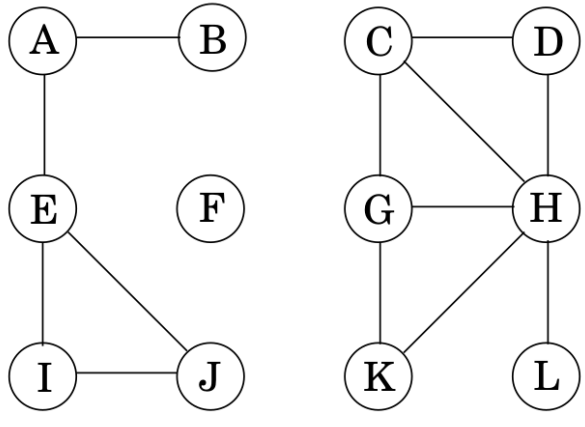
TIME TO RUN DFS

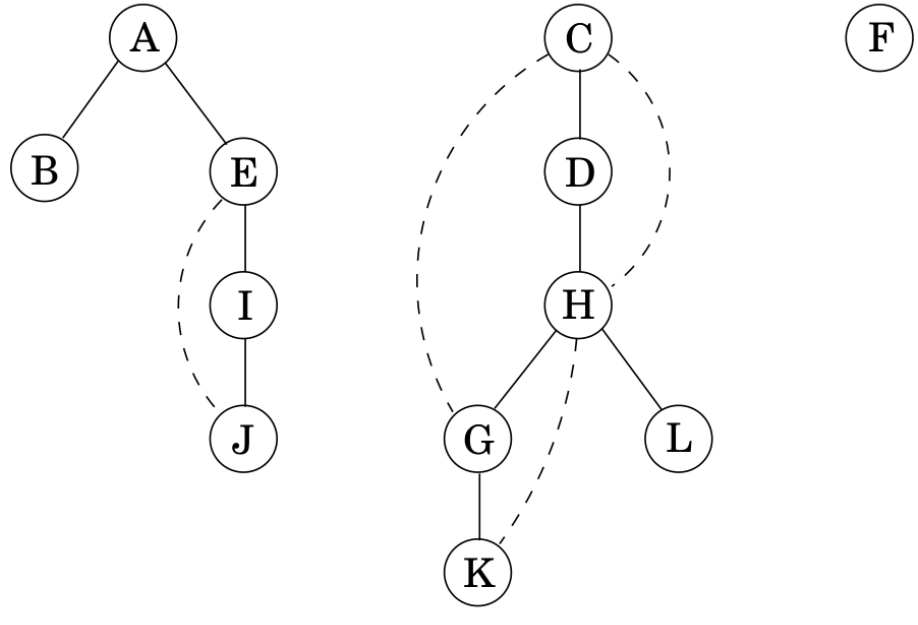
For all vertices together,

- $O(|V|)$ to mark nodes visited, call pre/postvisit
- Each edge (u, v) will be visited twice
  - once in explore(u)
  - once in explore(v)
- Therefore $O(|E|)$ work to scan edges

Total: $O(|V| + |E|)$

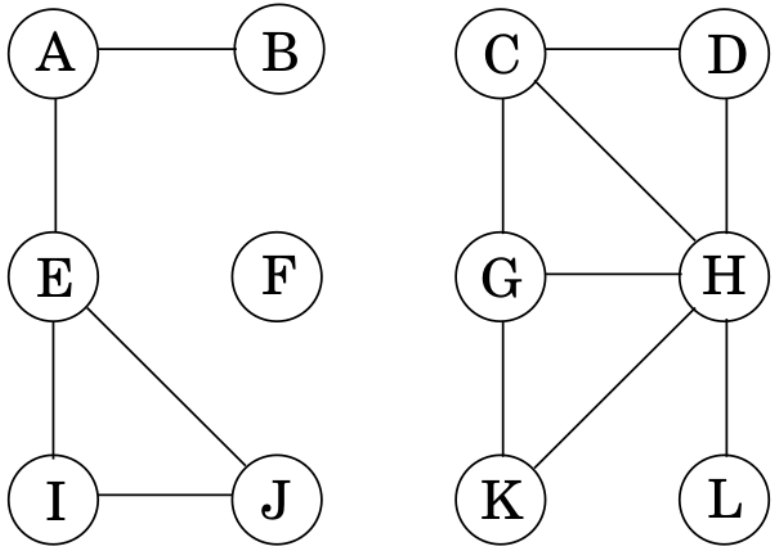
Running DFS on this graph (left) generates this forest (right):





An undirected graph is **connected** if there is a path from any vertex to any other.

In a disconnected graph, each connected subgraph is called a **connected component**.



Connected components:

- $\{A, B, E, I, J\}$
- $\{C, D, G, H, K, L\}$
- $\{F\}$

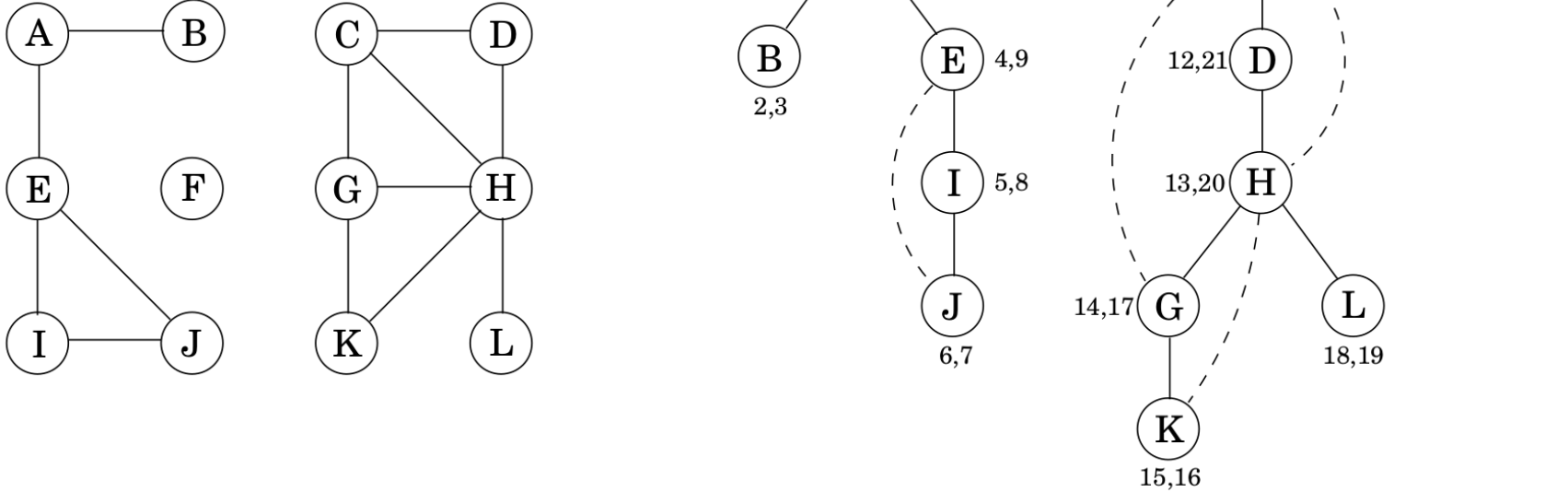
```
procedure previsit(v)
ccnum[v] = cc
```

- Initialize to 0
- Increment each time DFS calls explore

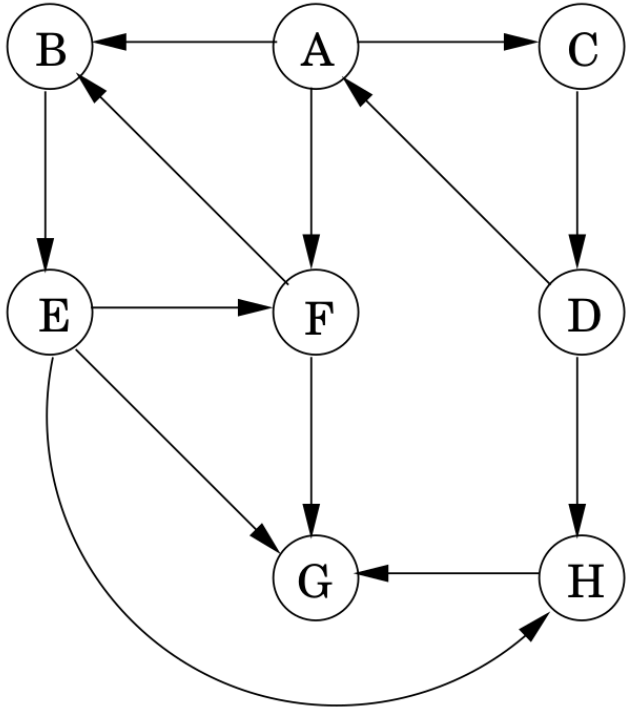
Let's add a counter to see when we enter/leave nodes:

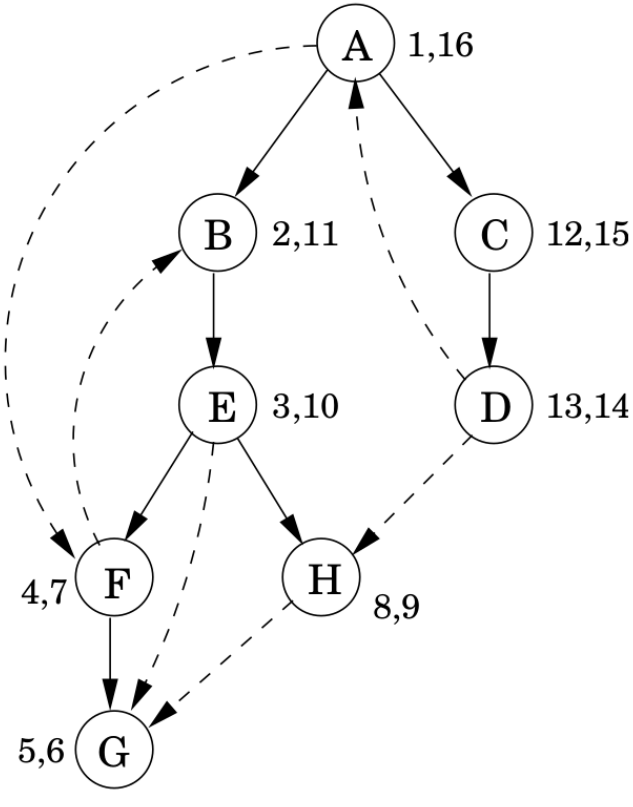
```
procedure previsit(v)
pre[v] = clock
clock = clock + 1

procedure postvisit(v)
post[v] = clock
clock = clock + 1
```

Then our forest now looks like this:

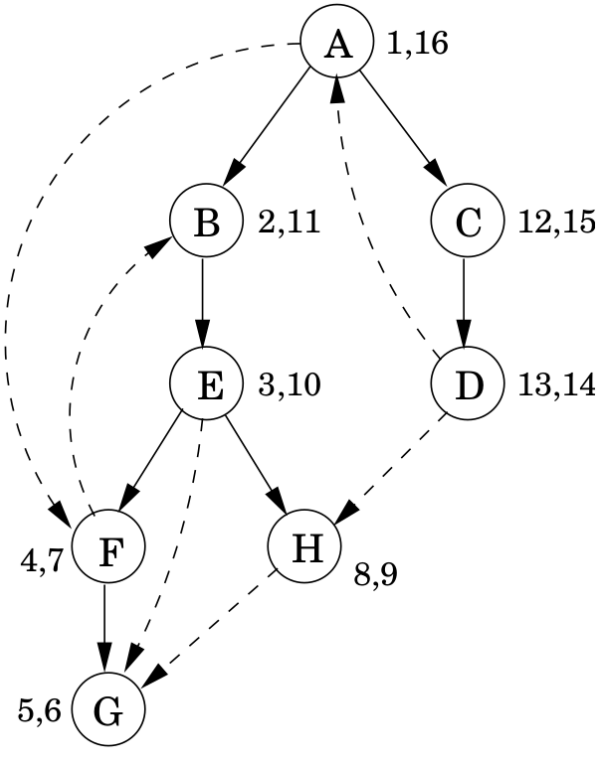


DFS ON DIRECTED GRAPHS



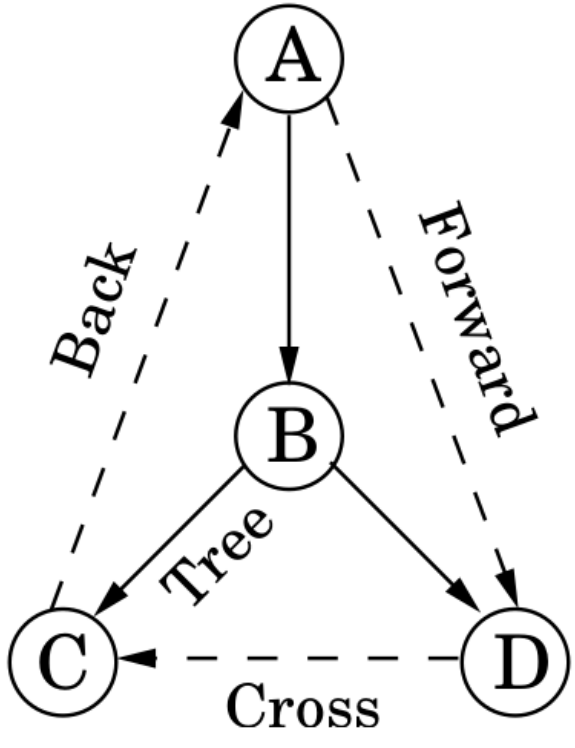


Terminology:

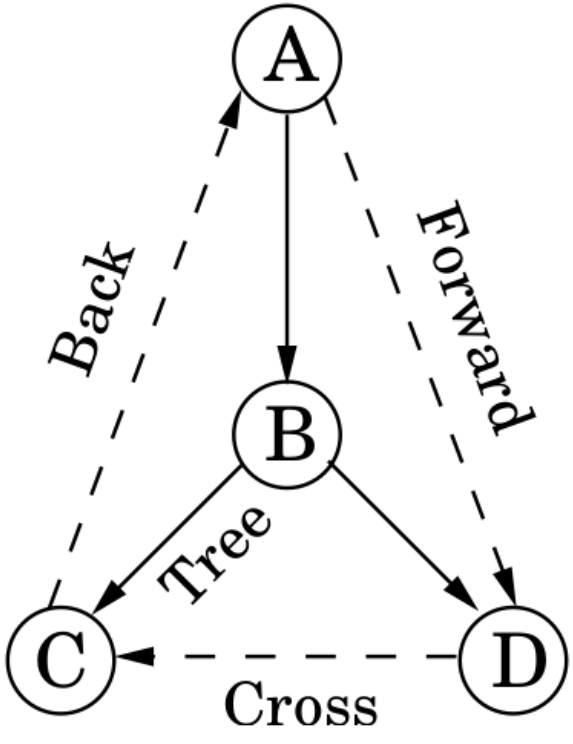


- A is the **root**
- E has **descendants** F, G, and H
- E is an **ancestor** of F, G, and H
- C is the **parent** of D
- D is the **child** of C

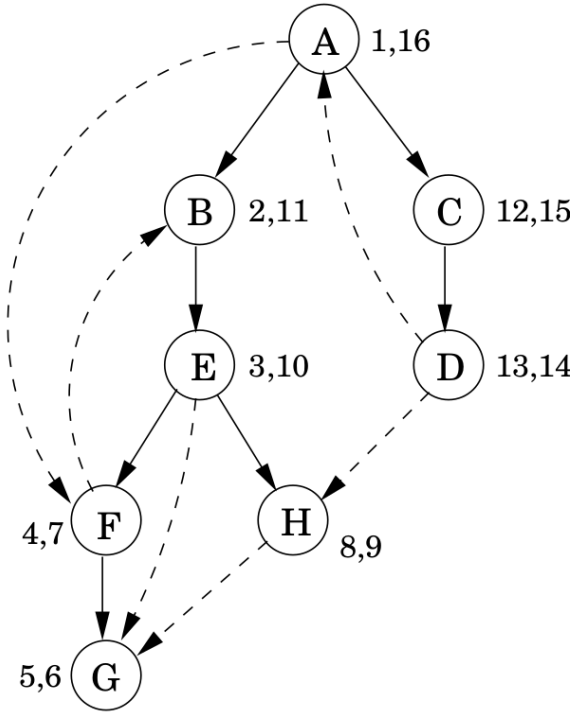
We can also have finer-grained distinctions on edges in the generated tree:



- tree edges
- forward edges
- back edges
- cross edges



- tree edges: part of the DFS forest
- forward edges: node to nonchild descendent
- back edges: node to ancestor
- cross edges: lead to neither descendent nor ancestor



Edge categories:

`pre/post` *ordering for* (u, v)	*Edge type*
$[_u \; [_v \;]_v \;]_u$	Tree/forward
$[_v \; [_u \;]_u \;]_v$	Back
$[_v \;]_v \; [_u \;]_u$	Cross

CYCLES

A cycle is a circular path $v_0 \to v_1 \to \ldots \to v_k \to v_0$.

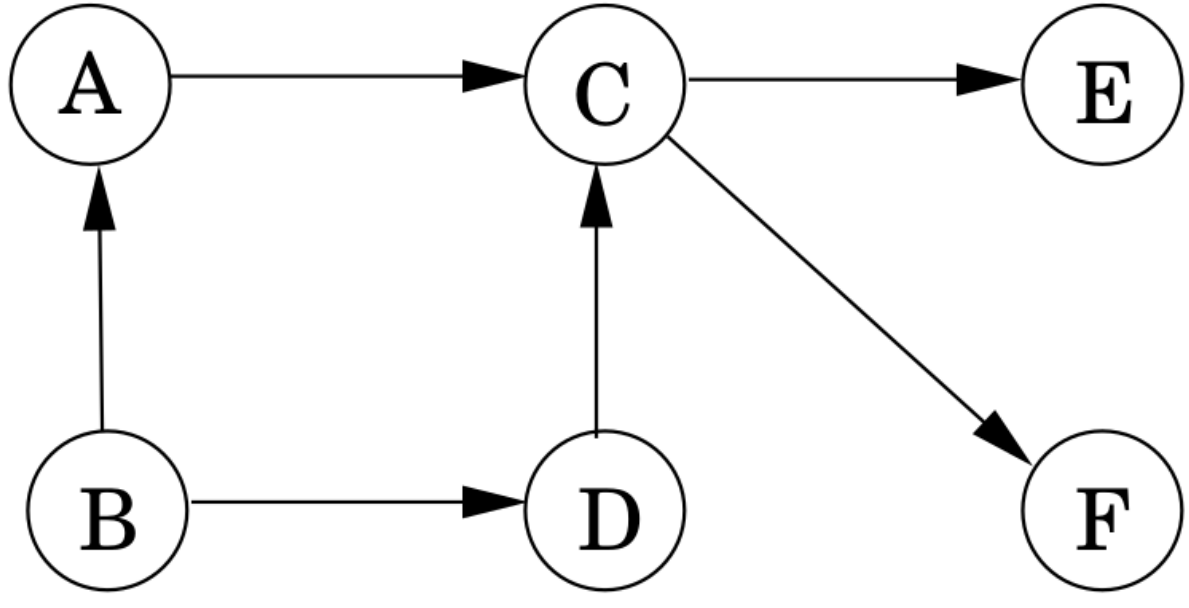
A graph without cycles is acyclic.

DFS AND CYCLES

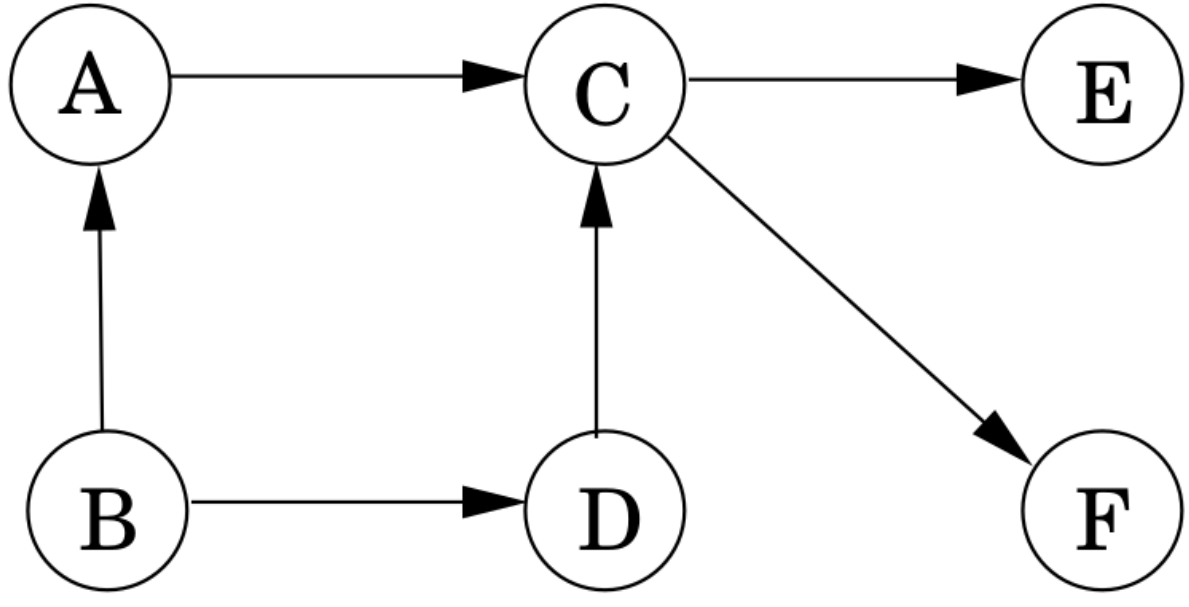
A directed graph has a cycle if and only if DFS reveals a back edge.

DIRECTED ACYCLIC GRAPH (DAG)

Dags are good for modeling hierarchies or dependencies (e.g., course prerequisites).

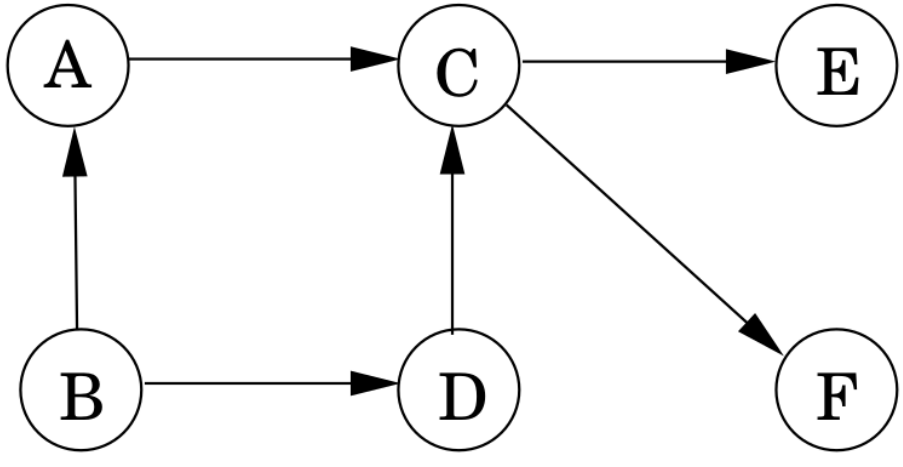


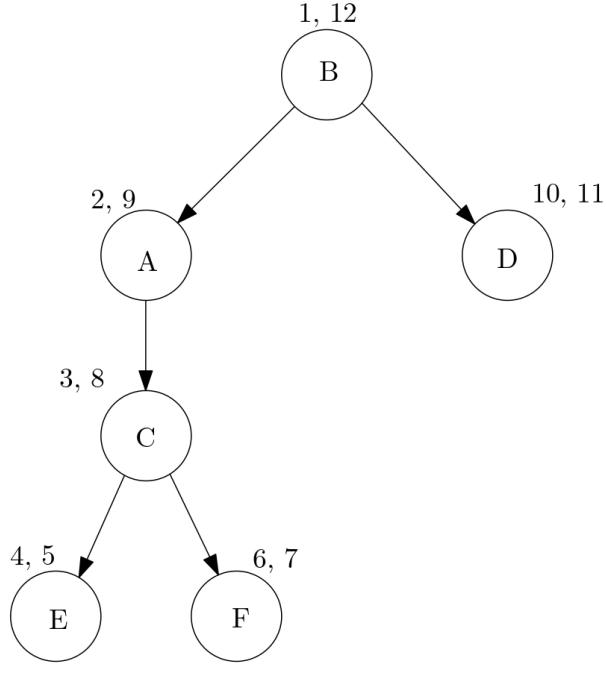
Given a dag, we may want to linearize (topologically sort) the nodes.



One possibility: B, A, D, C, E, F

Linearize a dag: list nodes in decreasing order of post numbers





$$B, D, A, C, F, E$$

Given a dag, we can define two kinds of special nodes:

- source: a node with no incoming edges
- sink: a node with no outgoing edges

  Then a linearization must start with a source and end with a sink.

Another algorithm for linearization:

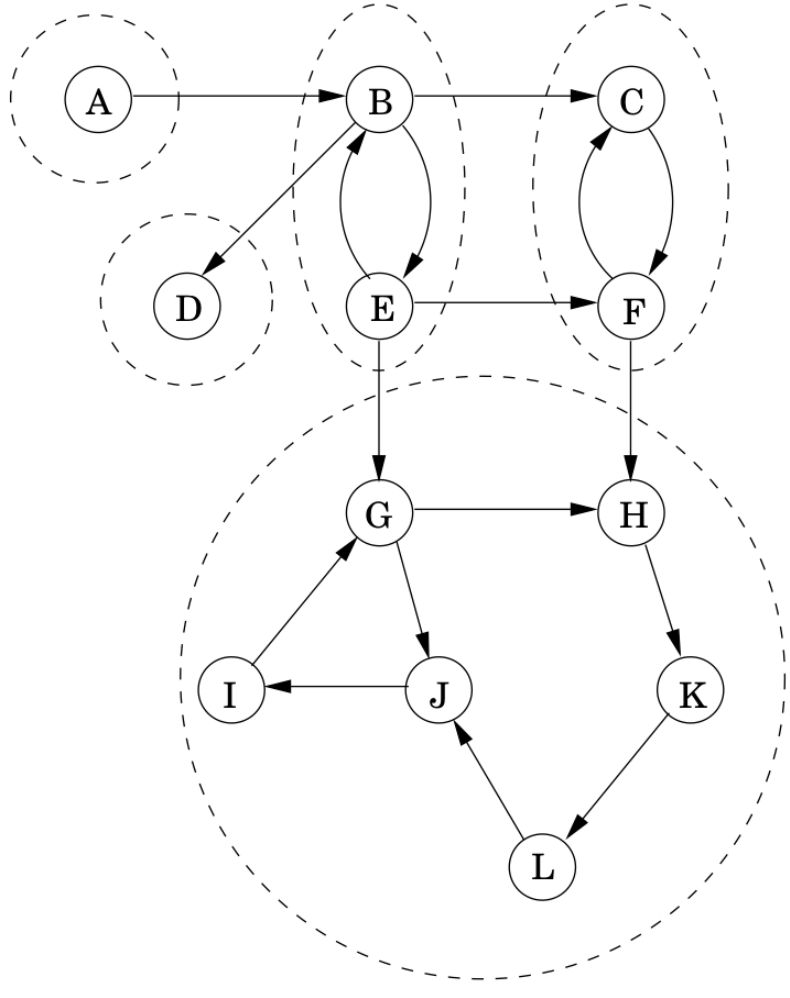
- find a source
- output it
- delete it
- repeat until graph is empty

STRONGLY-CONNECTED COMPONENTS

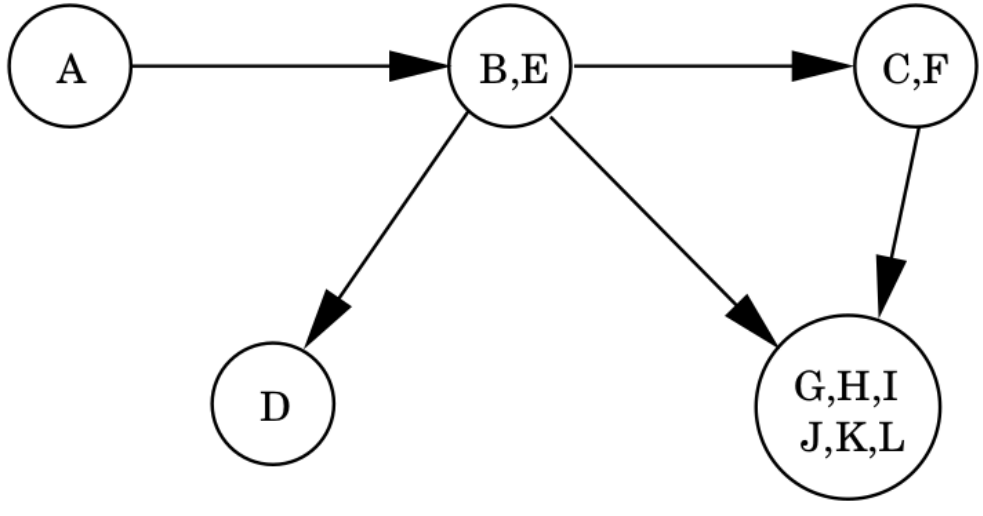
Two nodes u and v are connected if there is a u-v path as well as a v-u path.

We can partition a directed graph into a set **strongly-connected components** (SCCs), where all vertices are connected.

Dotted lines indicate the SCCs:



Then we can shrink each component to a single meta-vertex:



A directed graph is a dag of its strongly-connected components.

How can we decompose a graph to SCCs?

- If we found a node in a sink SCC
  - explore would find all nodes in its SCC
- Then we could remove it and repeat

How can we decompose a graph to SCCs?

But it's easier to find a node in a source SCC:

node with the highest post number in DFS

More generally,

- if C and C' are SCCs,
- and there's a $C \to C'$ edge,
- then the highest post number in $C >$ highest in C'

This means SCCs can be linearized by decreasing highest post numbers.

(a generalization of linearization of dags)

How can we decompose a graph to SCCs?

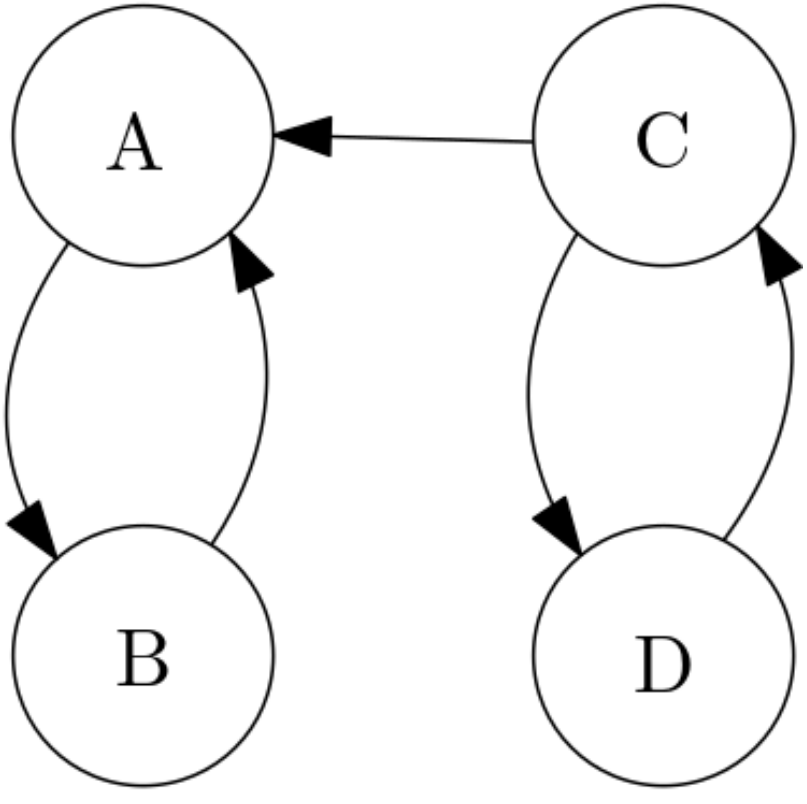
We can find a source SCC, but we need a sink SCC

Transform the graph G into the reverse graph G^R!

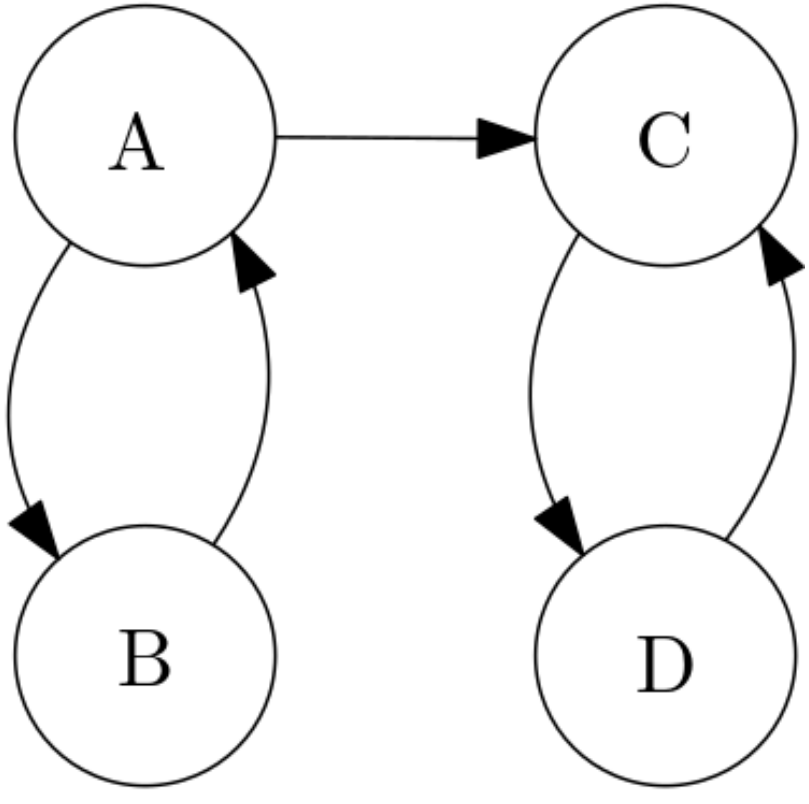
How can we decompose a graph to SCCs?

- Compute G^R
- Run DFS on G^R
- Find the connected components using explore, in decreasing order of post numbers

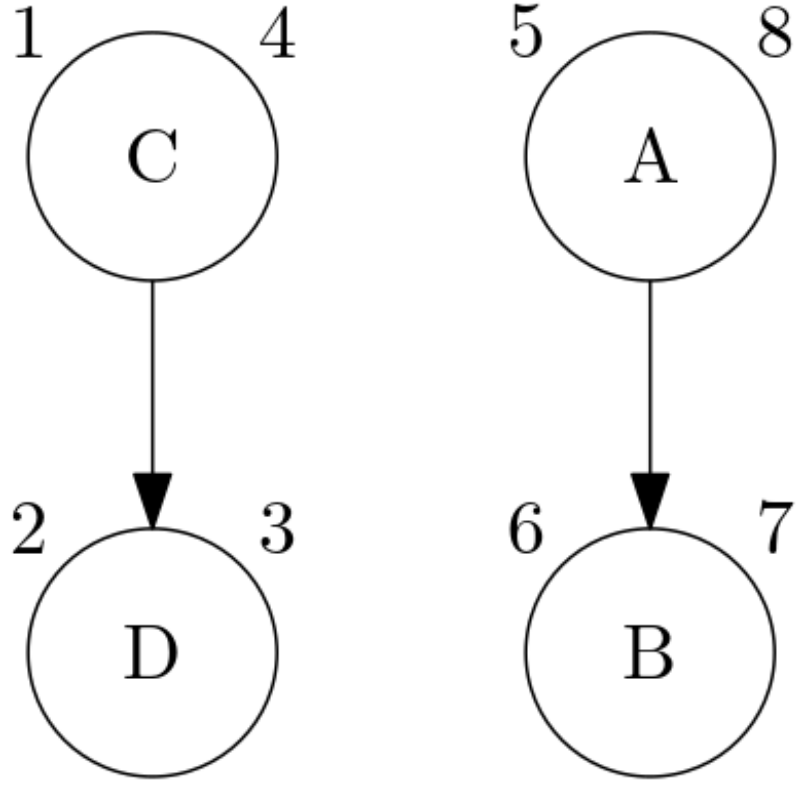
Suppose we have this graph:



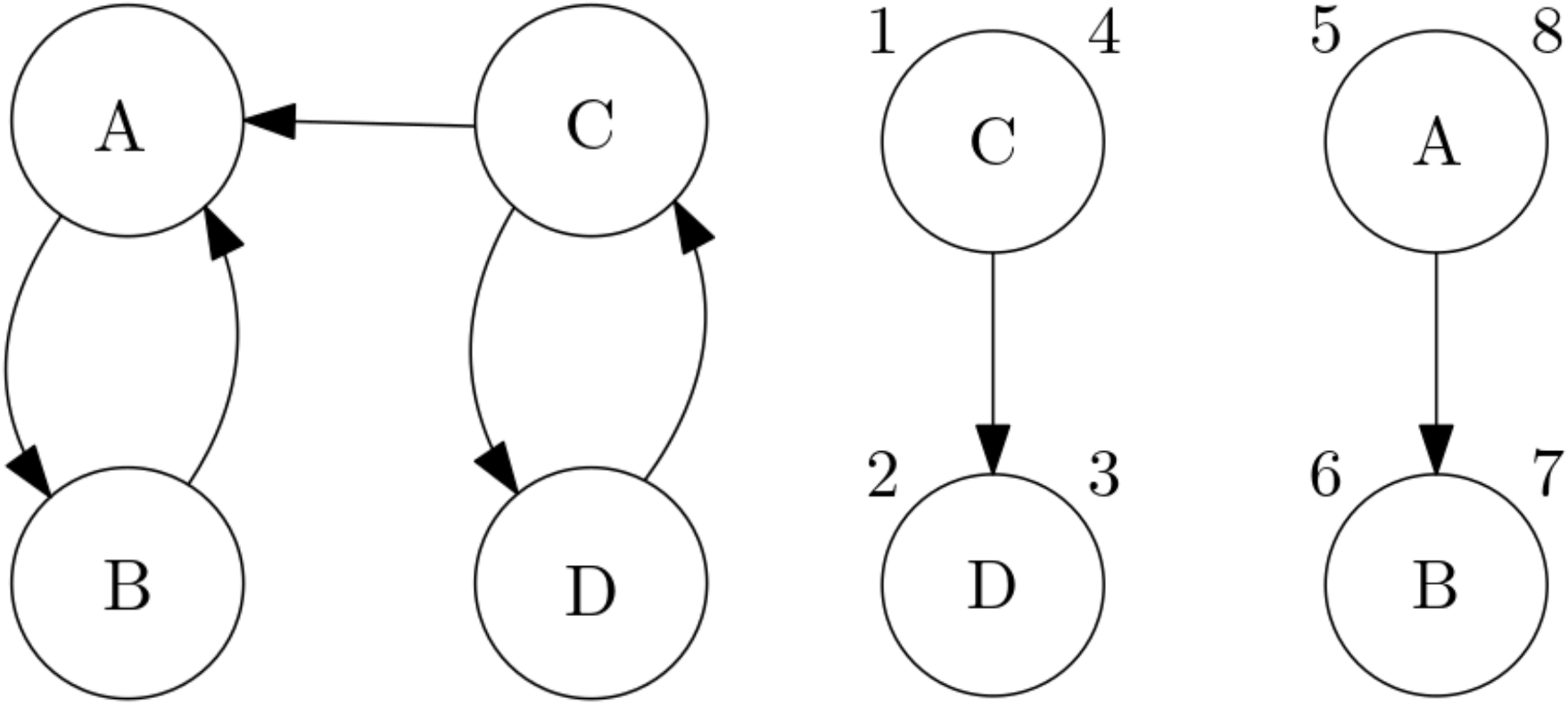
Then G^R is:



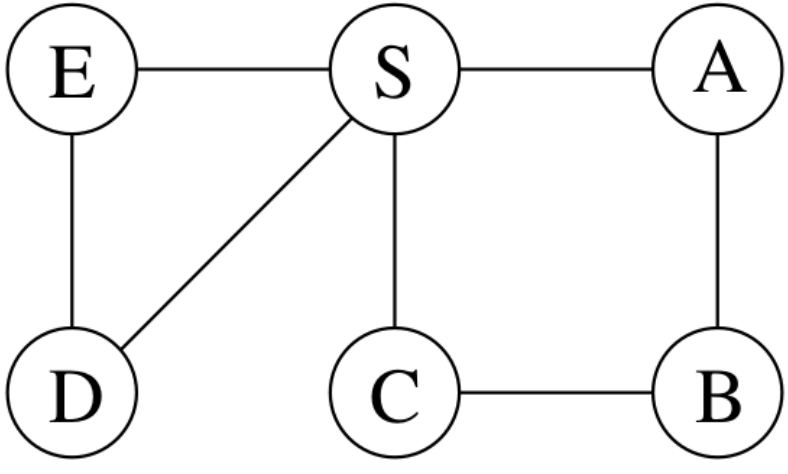
And a DFS gives these post numbers:

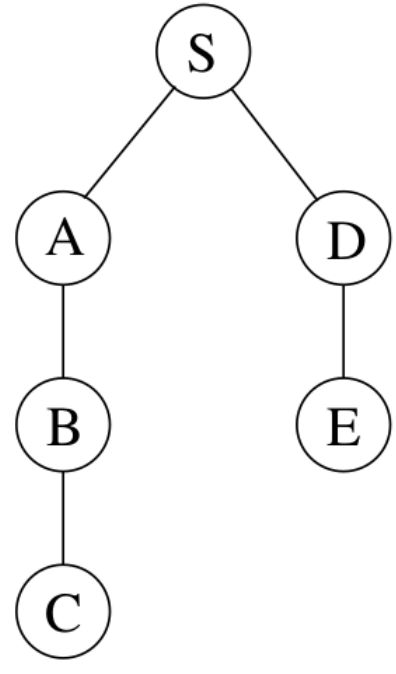


Then we can separate this into SCCs $\{C, D\}$ and $\{A, B\}$:



If we do DFS search from S, we should get the following tree:





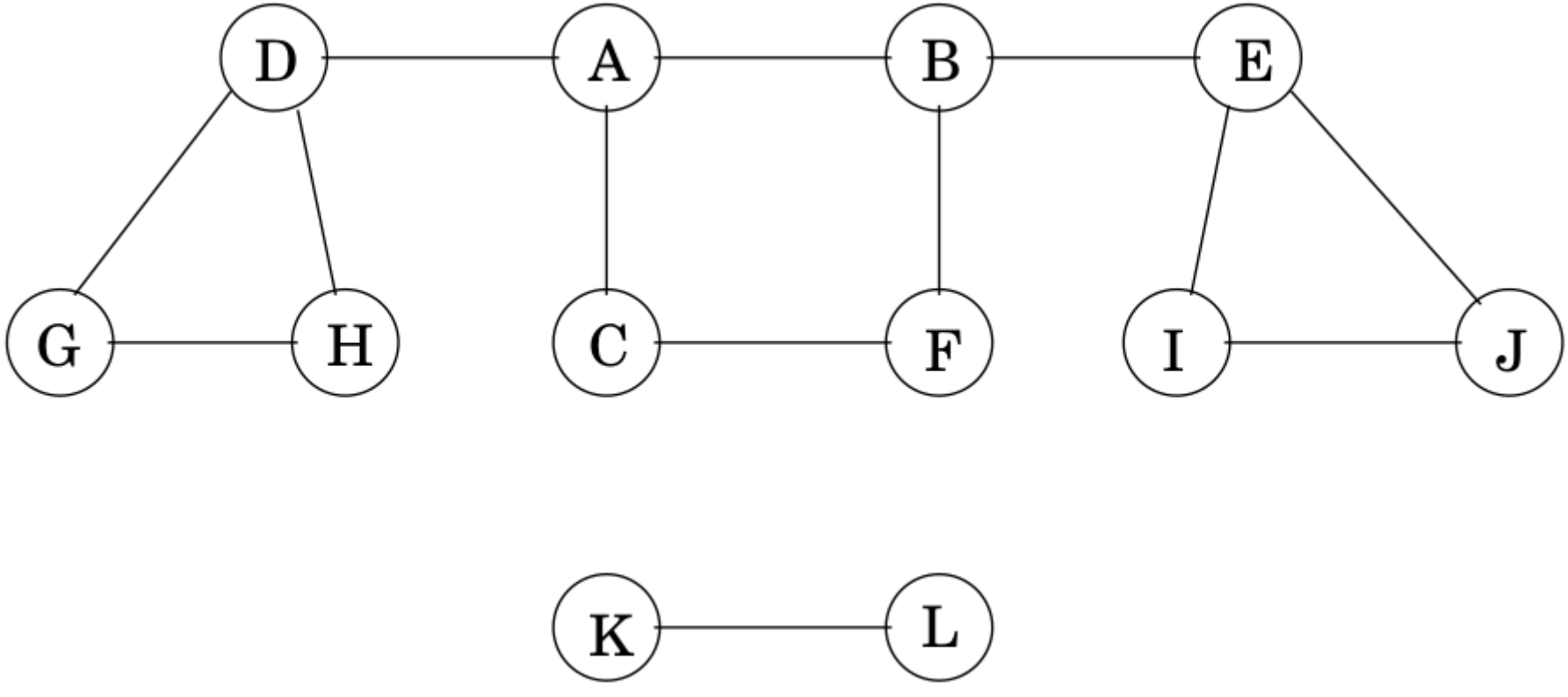
But this gives an inefficient route of $S \to A \to B \to C$ instead of $S \to C$

BREADTH-FIRST SEARCH (BFS)

- Proceed layer by layer
- Find layer $d + 1$ by scanning neighbors of layer d

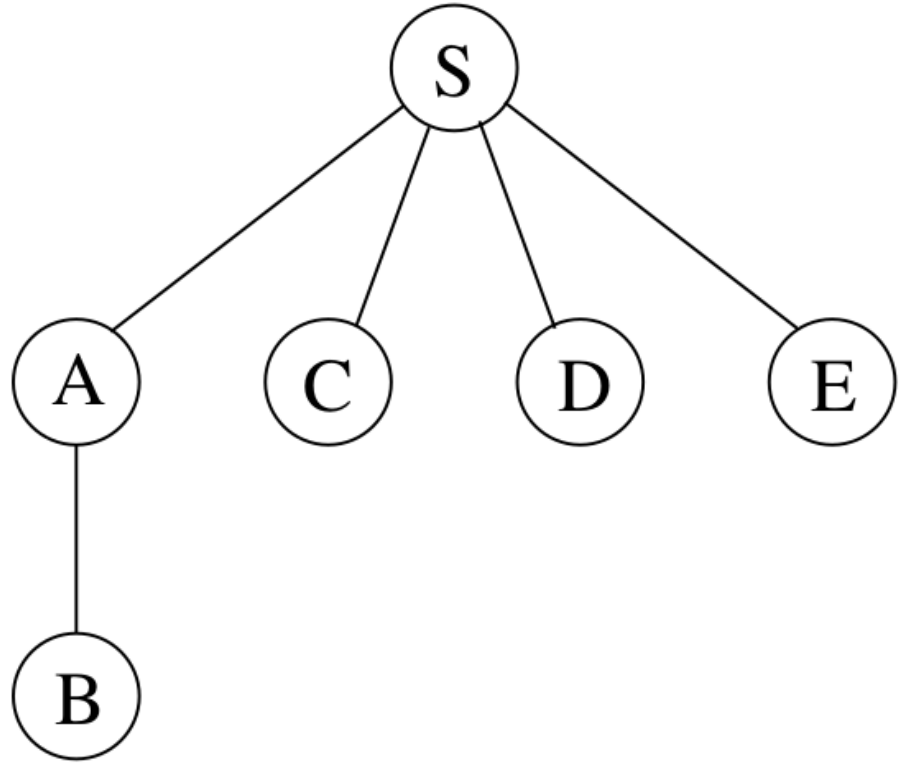
```
procedure bfs(G, s)
Input:    Graph G = (V, E), directed or undirected; vertex s ∈ V
Output:   For all vertices u reachable from s, dist(u) is set
          to the distance from s to u.

for all u ∈ V:
    dist(u) = ∞

dist(s) = 0
Q = [s] (queue containing just s)
while Q is not empty:
    u = eject(Q)
    for all edges (u, v) ∈ E:
        if dist(v) = ∞:
            inject(Q, v)
            dist(v) = dist(u) + 1
```

Let's run BFS on our graph, starting at S:



Let's run BFS on our graph, starting at S:

Order of visitation	Queue contents after processing node
	$[S]$
S	$[A\ C\ D\ E]$
A	$[C\ D\ E\ B]$
C	$[D\ E\ B]$
D	$[E\ B]$
E	$[B]$
B	$[\,]$



BFS RUNNING TIME

- Each vertex put onto queue once
- Examine each edge once (directed) or twice (undirected)

$$O(|V| + |E|)$$

DIJKSTRA'S ALGORITHM

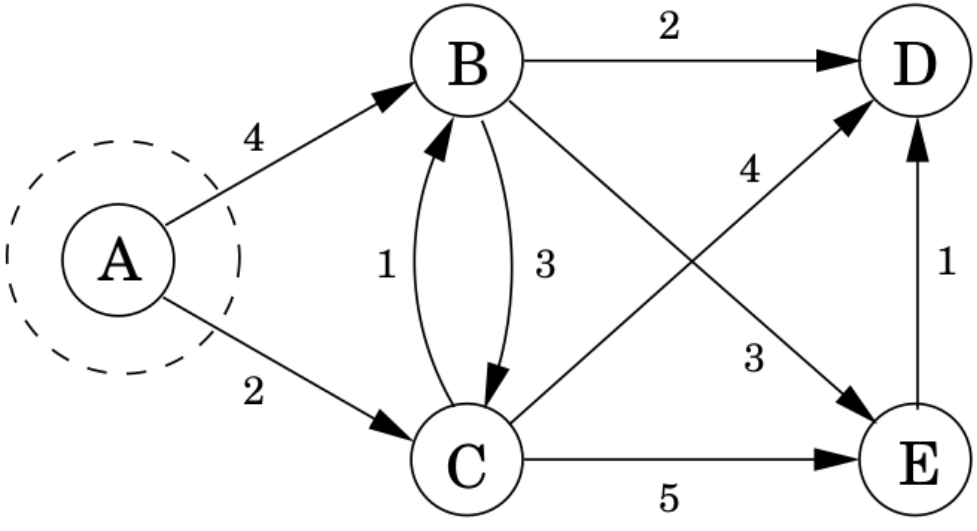
Given a graph G and a starting vertex s,
find shortest paths to all reachable vertices

DIJKSTRA'S ALGORITHM

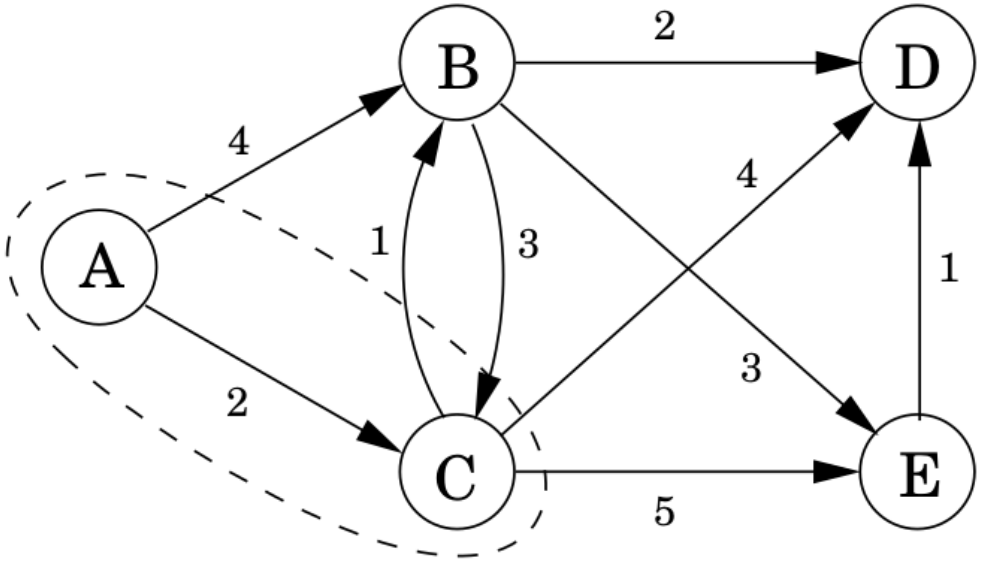
We need to use a **priority queue** with these operations:

- insert
- decrease-key
- delete-min
- make-queue

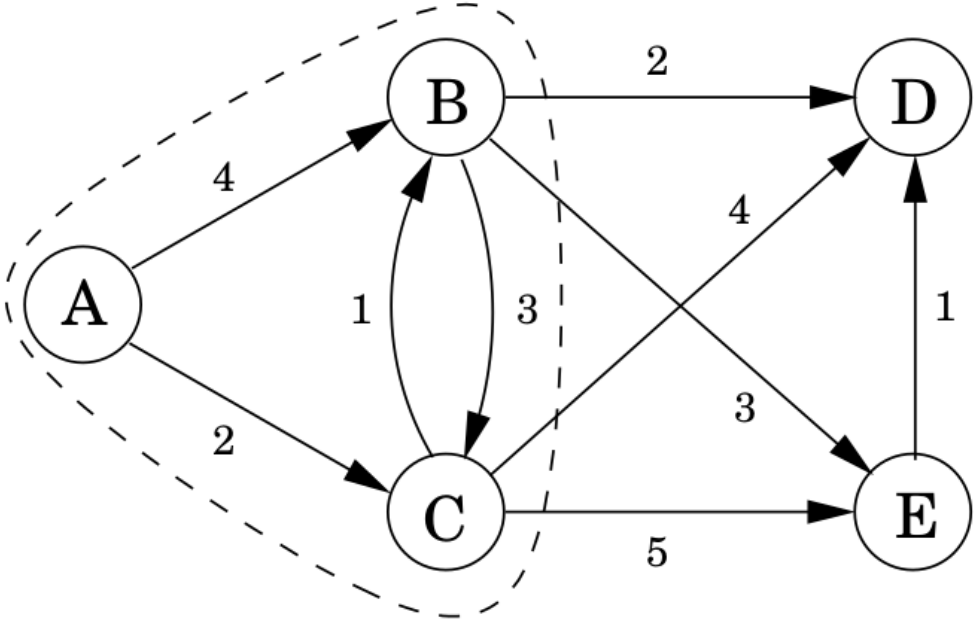
```
procedure dijkstra(G, l, s)
Input:    Graph G = (V, E), directed or undirected;
          positive edge lengths {l_e : e ∈ E}; vertex s ∈ V
Output:   For all vertices u reachable from s, dist(u) is set
          to the distance from s to u.

for all u ∈ V:
    dist(u) = ∞
    prev(u) = nil
dist(s) = 0

H = makequeue(V)   (using dist-values as keys)
while H is not empty:
    u = deletemin(H)
    for all edges (u, v) ∈ E:
        if dist(v) > dist(u) + l(u, v):
            dist(v) = dist(u) + l(u, v)
            prev(v) = u
            decreasekey(H, v)
```



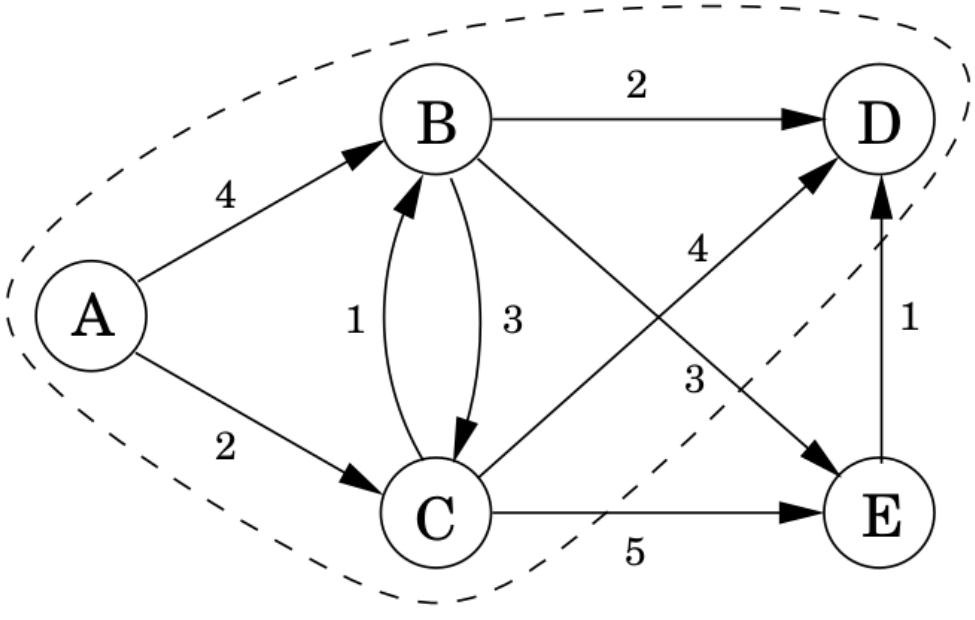
A: 0	D: ∞
B: 4	E: ∞
C: 2	



A: 0	D: 6
B: 3	E: 7
C: 2	

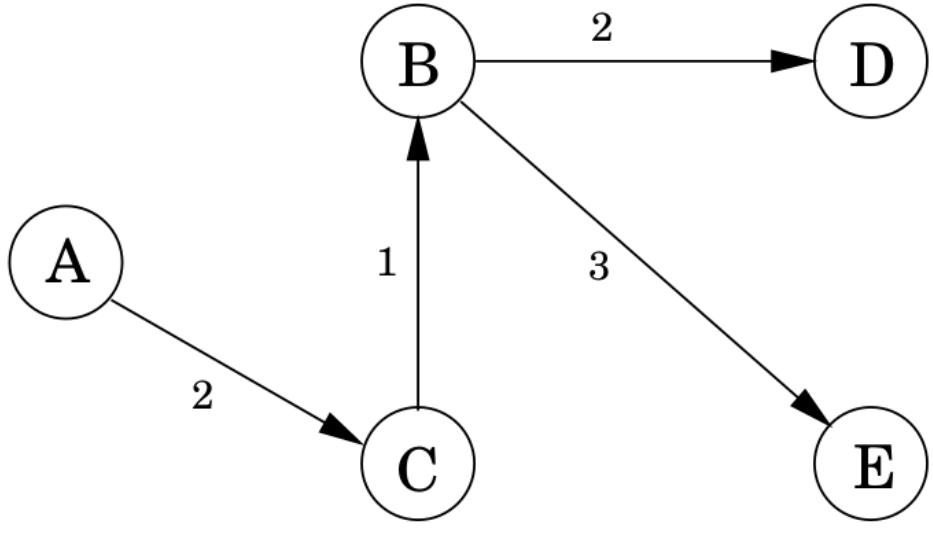


A: 0	D: 5
B: 3	E: 6
C: 2	



A: 0	D: 5
B: 3	E: 6
C: 2	

Finally, we get this tree for paths from A:



Dijkstra's algorithm is basically just BFS:

- Instead of a regular queue,
- use a priority queue to account for lengths

DIJKSTRA'S RUNNING TIME

- makequeue: at most $|V|$ insert operations
- $|V|$ deletemin operations
- $|V| + |E|$ insert/decreasekey operations

Time depends on implementation, but if we use a binary heap:

$$O((|V| + |E|) \log |V|)$$

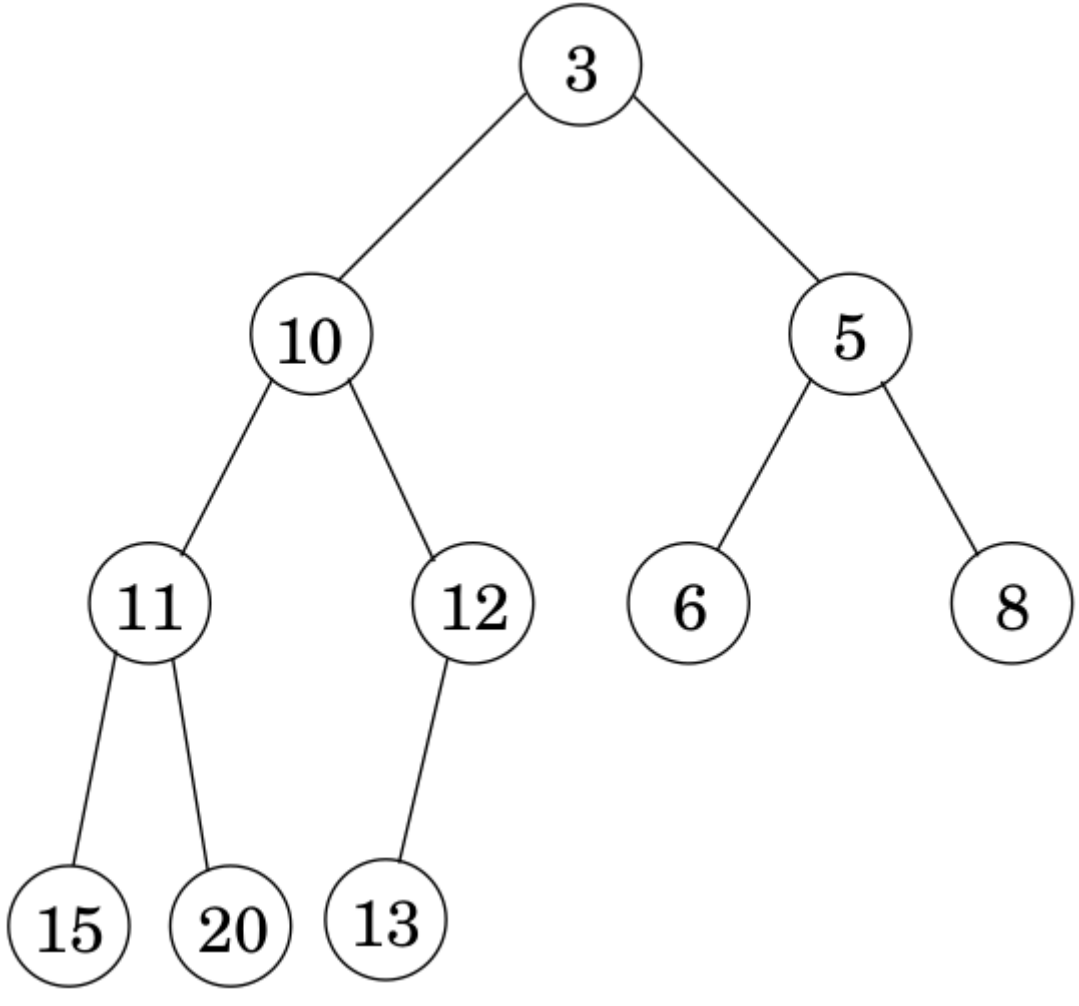
BINARY HEAP

- Complete binary tree (except last level)
- Key value $\leq$ that of its children

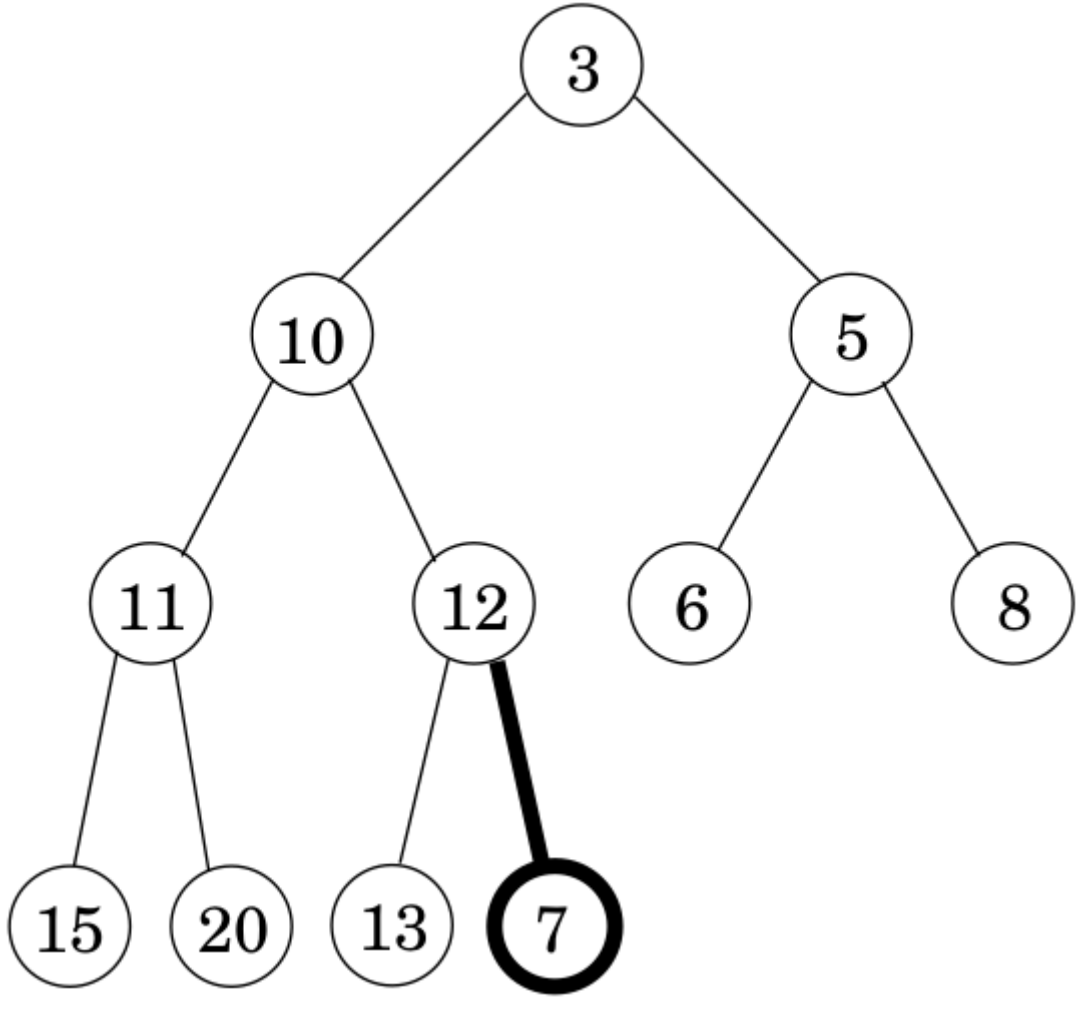
BINARY HEAP

- insert:
  - place node at bottom of tree and "bubble up"
- decreasekey:
  - "bubble up" (it's already in the tree)
- deletemin:
  - remove (and return) root
  - move last element to root
  - "sift down"

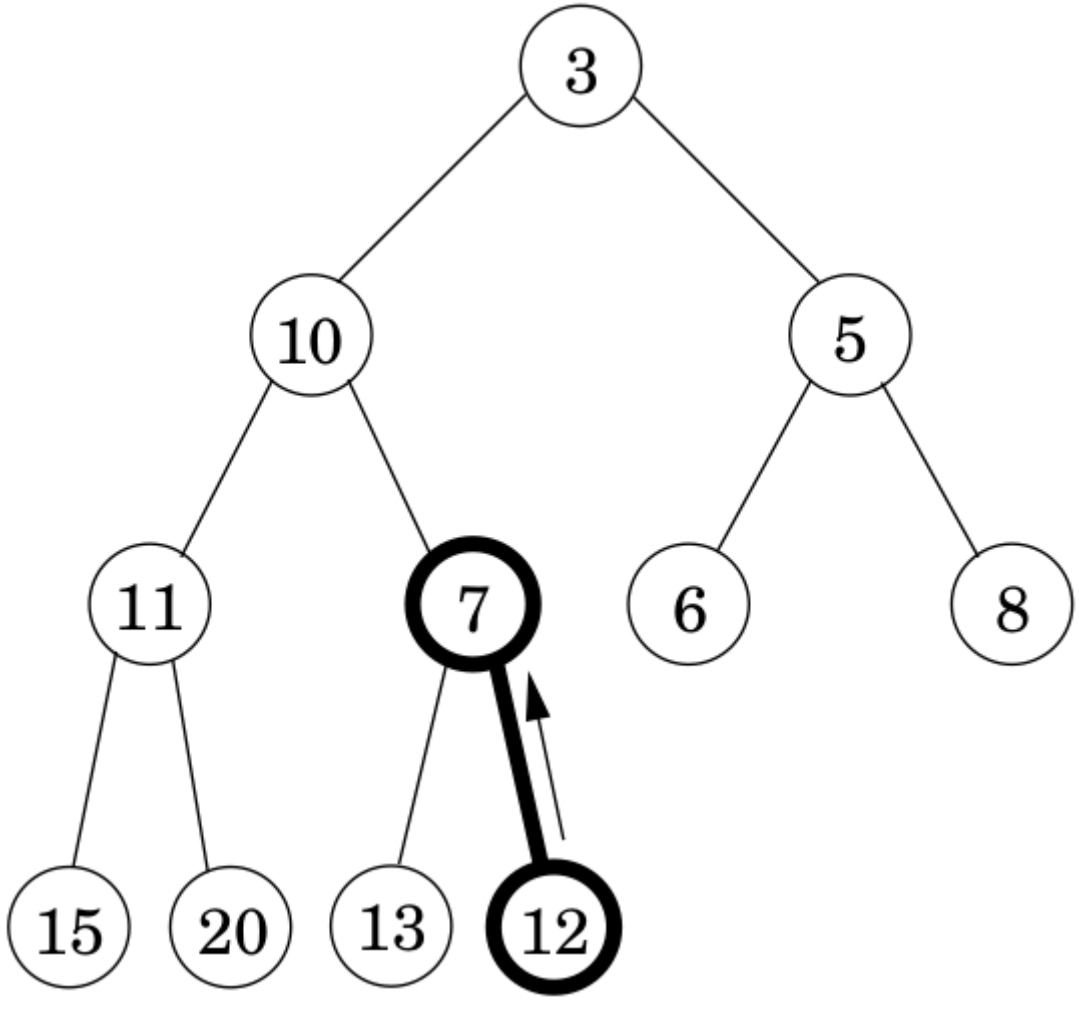
Let's say we have this initial heap:



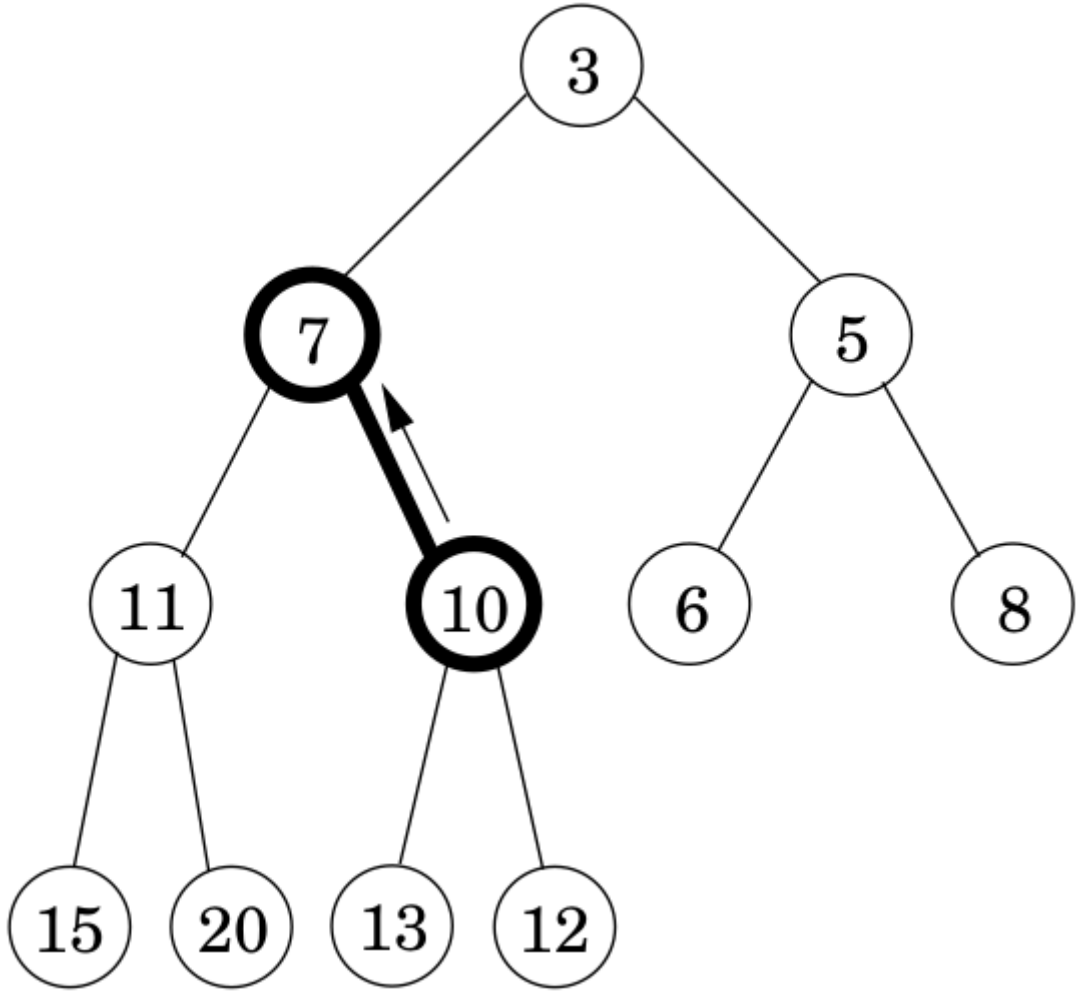
We insert 7:



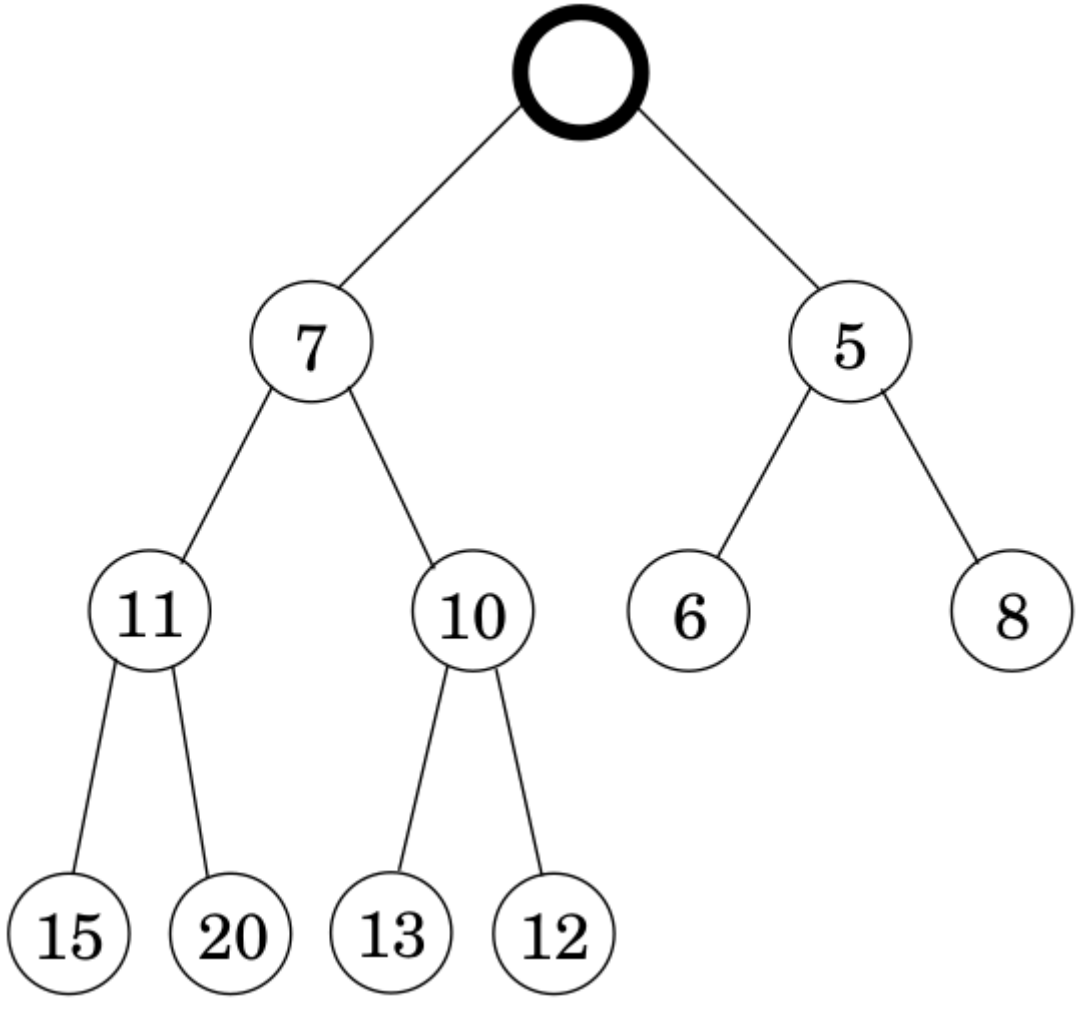
Bubble up:



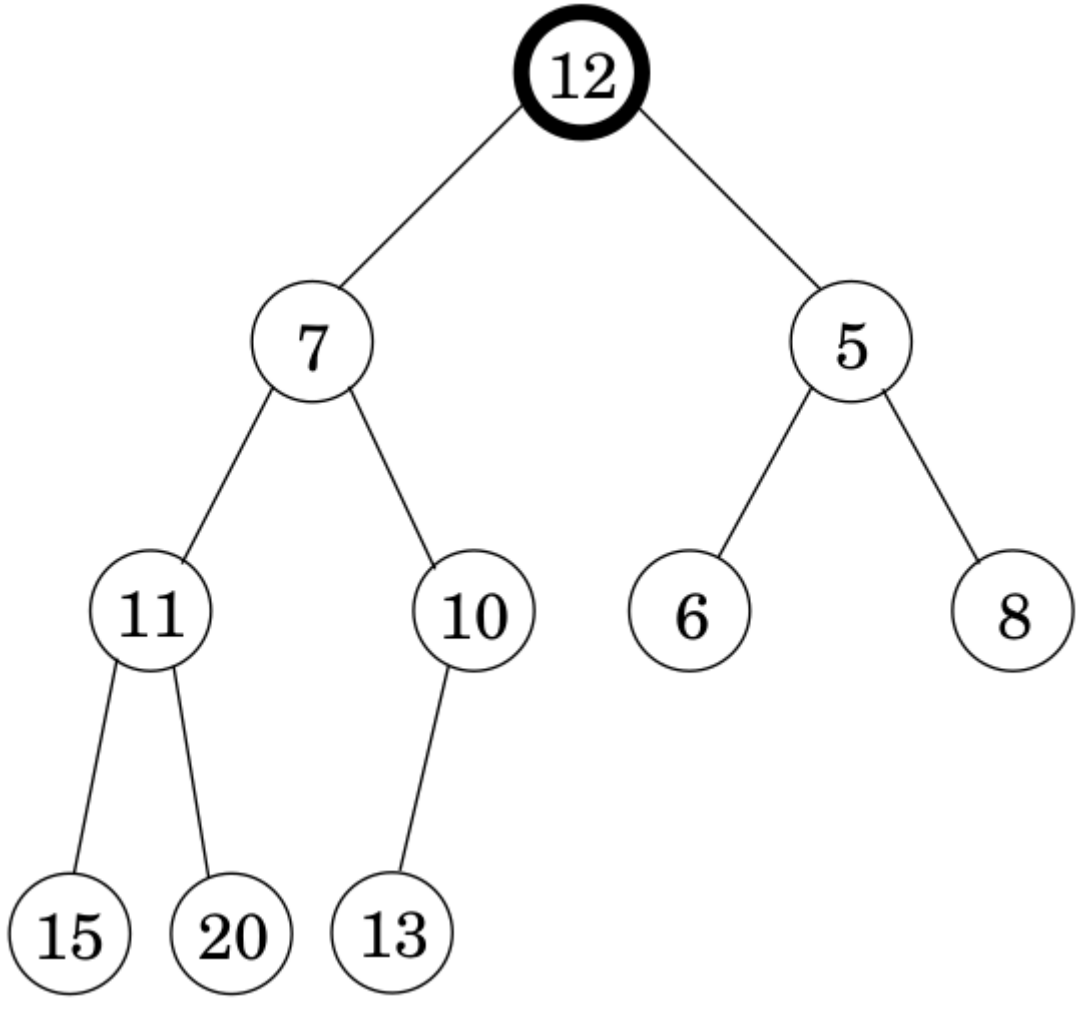
Bubble up:



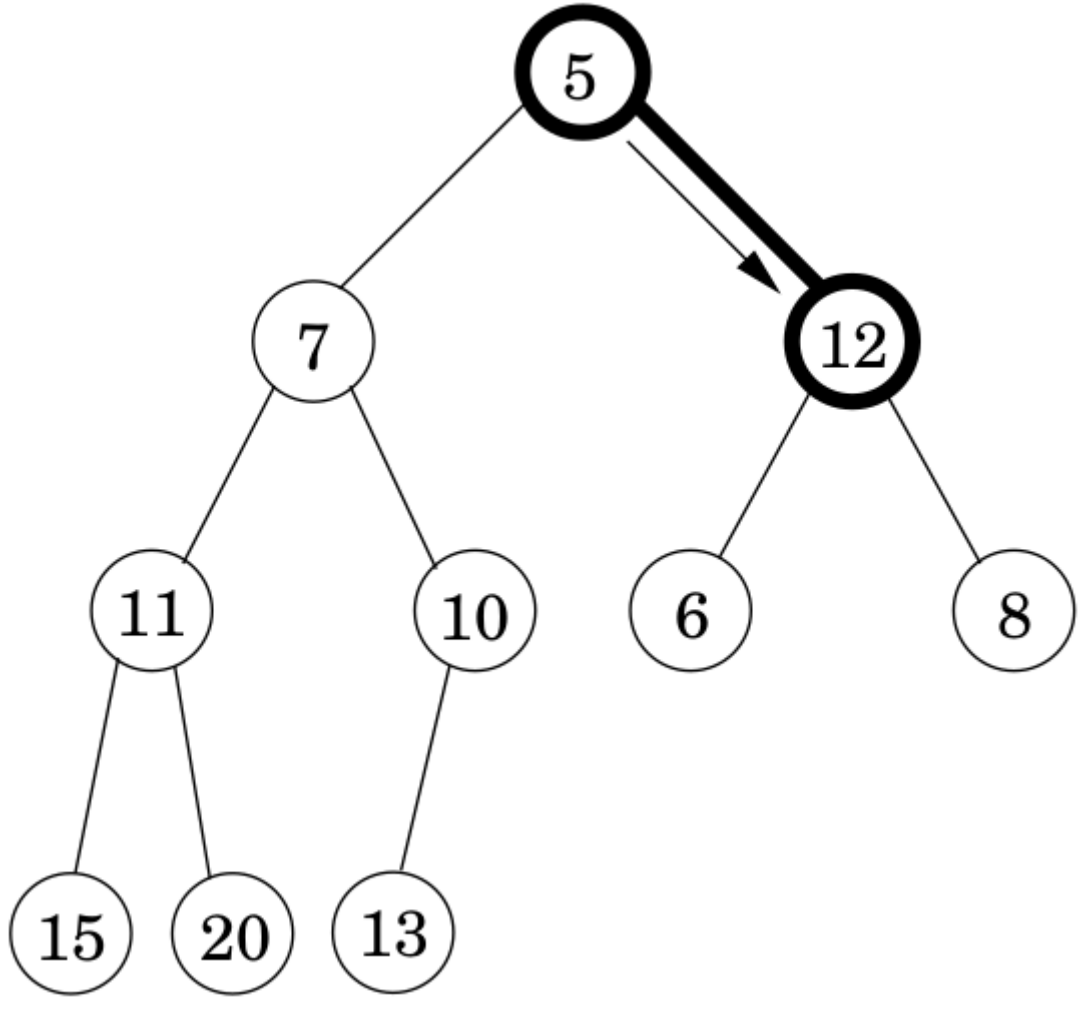
Now let's run delete-min:



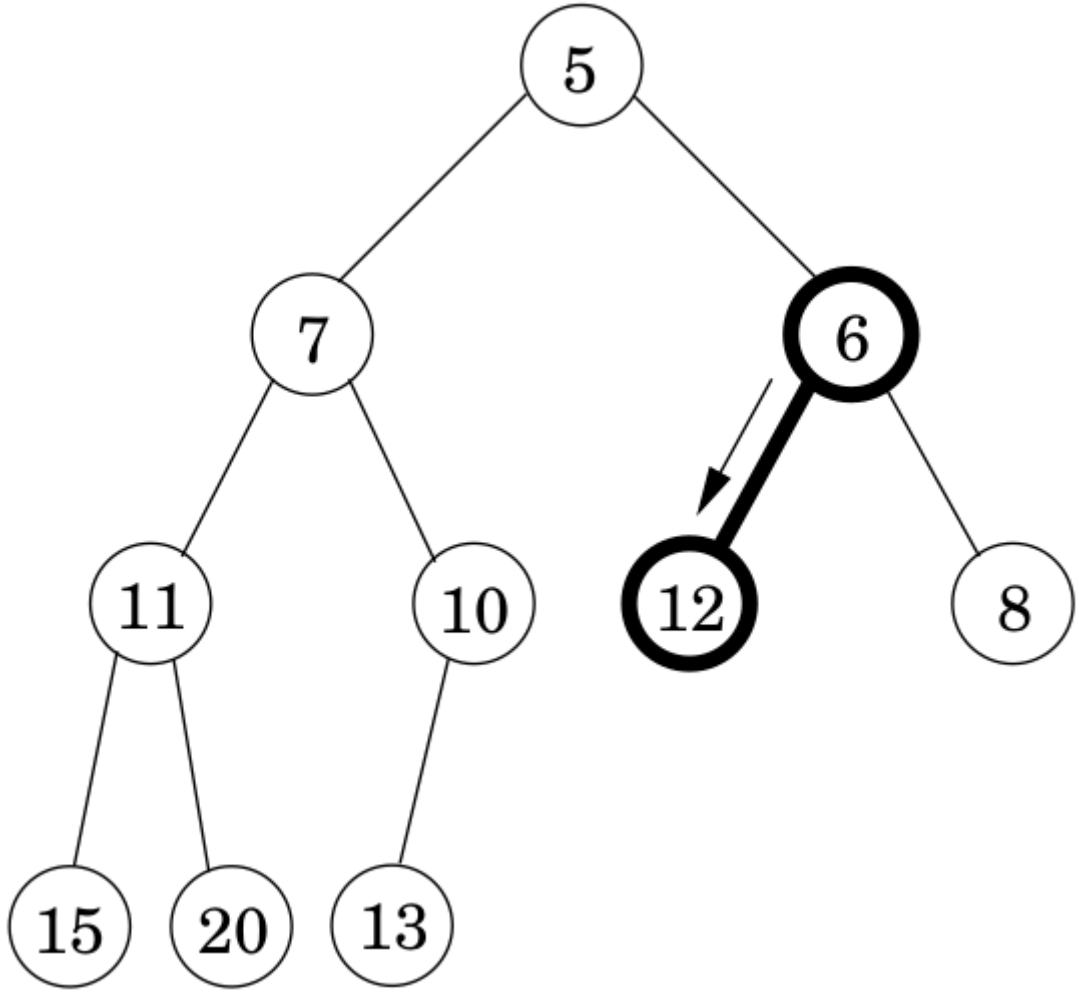
Move last node to root:



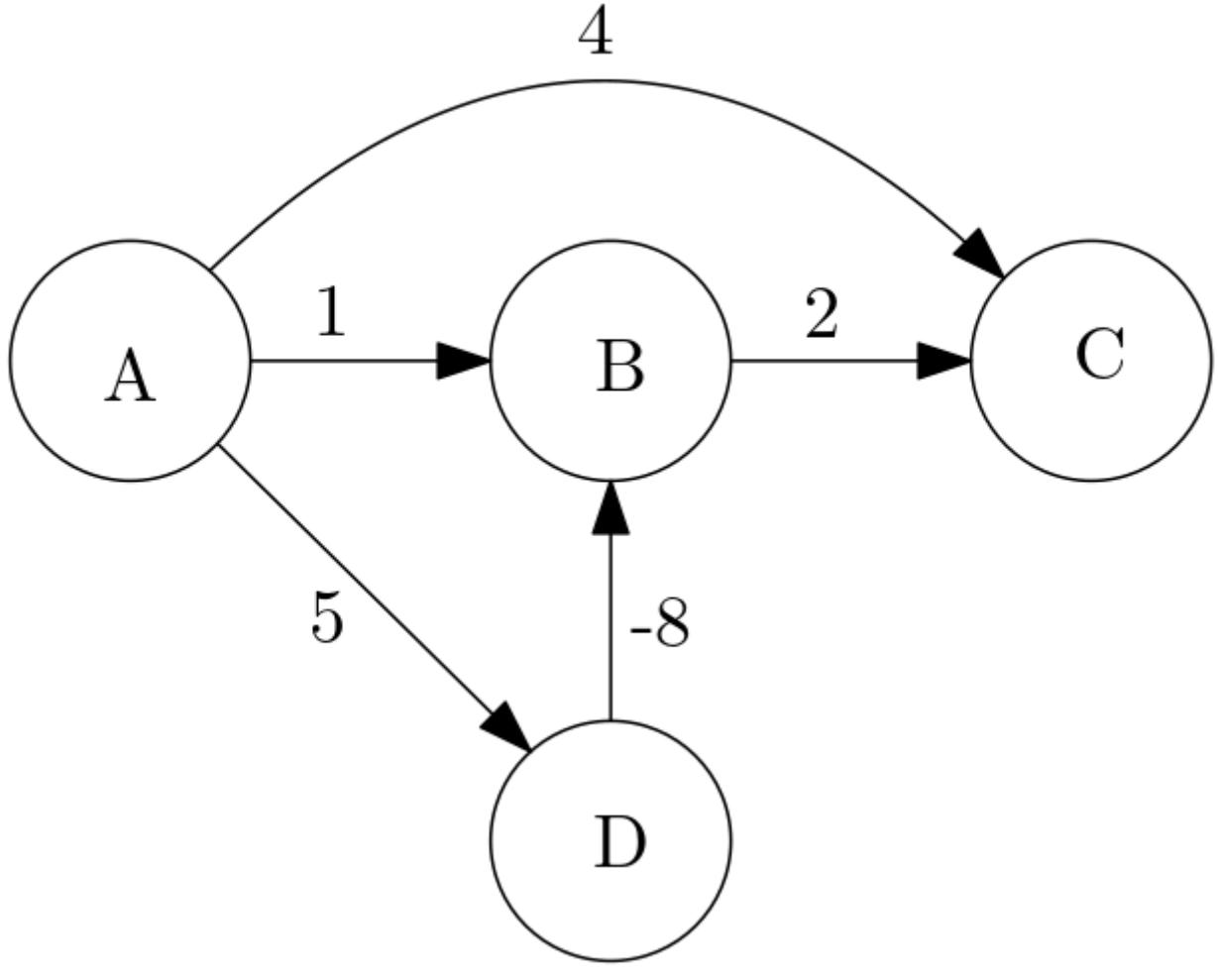
Sift down:



Sift down:



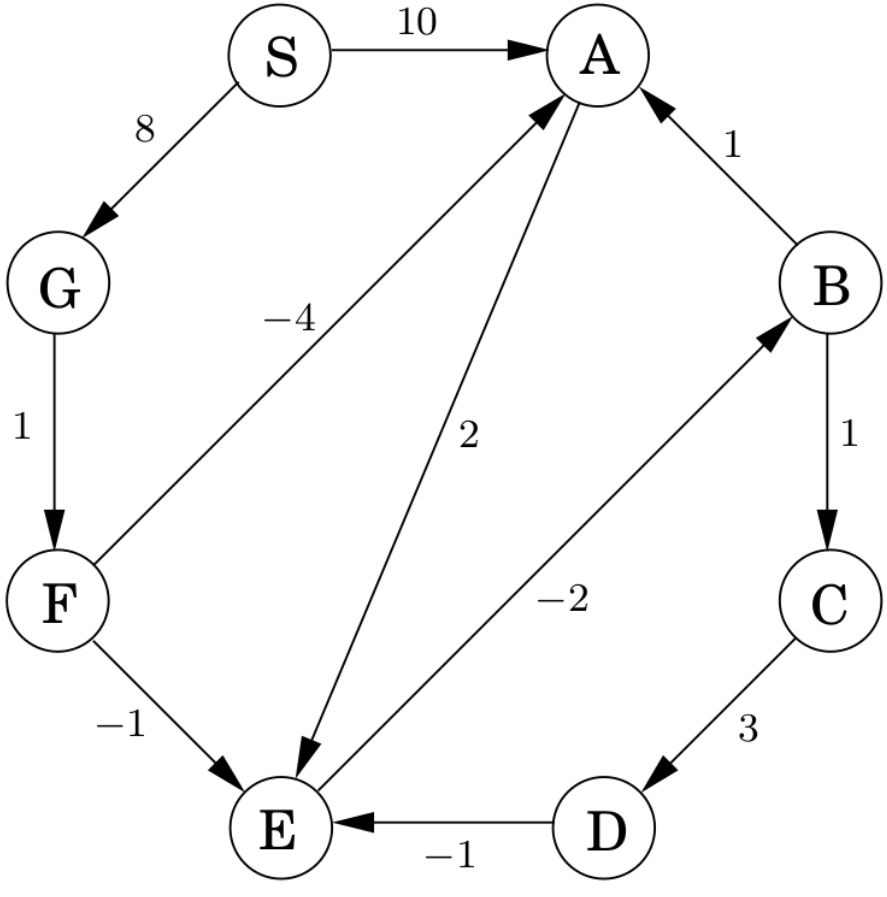
Dijkstra's algorithm with negative edges



Bellman-Ford algorithm:

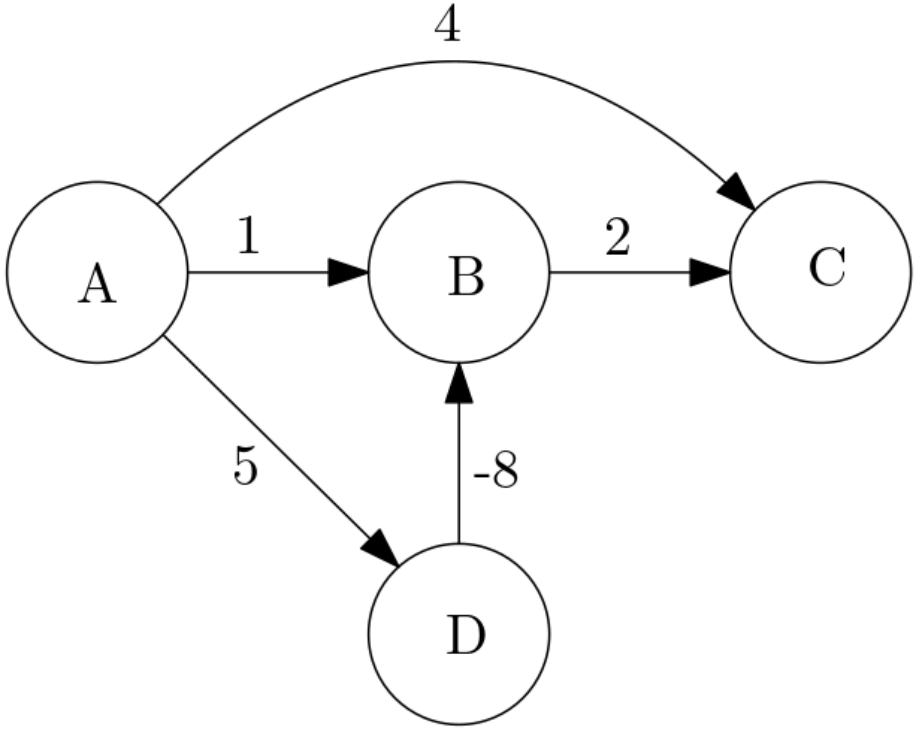
```
procedure shortest-paths(G, l, s)
Input:    Directed graph G = (V, E);
          edge lengths {l_e : e ∈ E} with no negative cycles;
          vertex s ∈ V
Output:   For all vertices u reachable from s, dist(u) is set
          to the distance from s to u.

for all u ∈ V:
   dist(u) = ∞
   prev(u) = nil

dist(s) = 0
repeat |V| − 1 times:
   for all e ∈ E:
      update(e)
```

Bellman-Ford algorithm:



	Iteration							
Node	0	1	2	3	4	5	6	7
S	0	0	0	0	0	0	0	0
A	∞	10	10	5	5	5	5	5
B	∞	∞	∞	10	6	5	5	5
C	∞	∞	∞	∞	11	7	6	6
D	∞	∞	∞	∞	∞	14	10	9
E	∞	∞	12	8	7	7	7	7
F	∞	∞	9	9	9	9	9	9
G	∞	8	8	8	8	8	8	8

Bellman-Ford algorithm:



A	0	0	0	0
B	∞	1	-3	-3
C	∞	4	3	-1
D	∞	5	5	5

SHORTEST PATHS IN DAGS

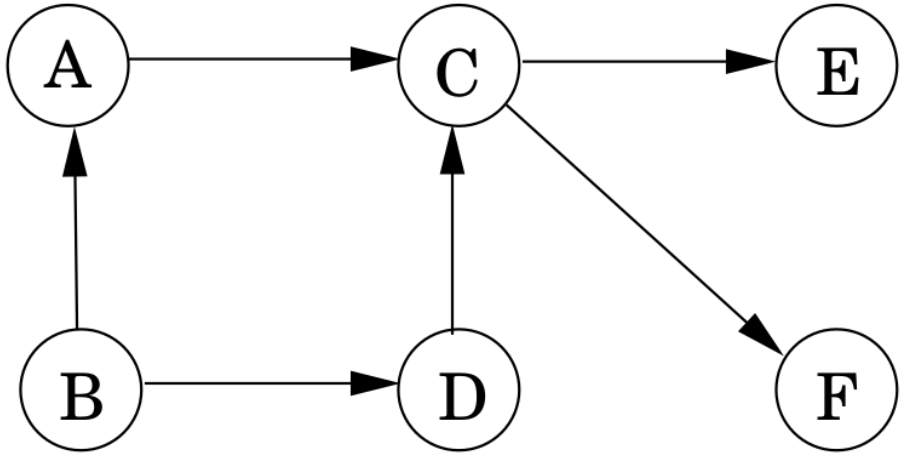
Key idea:

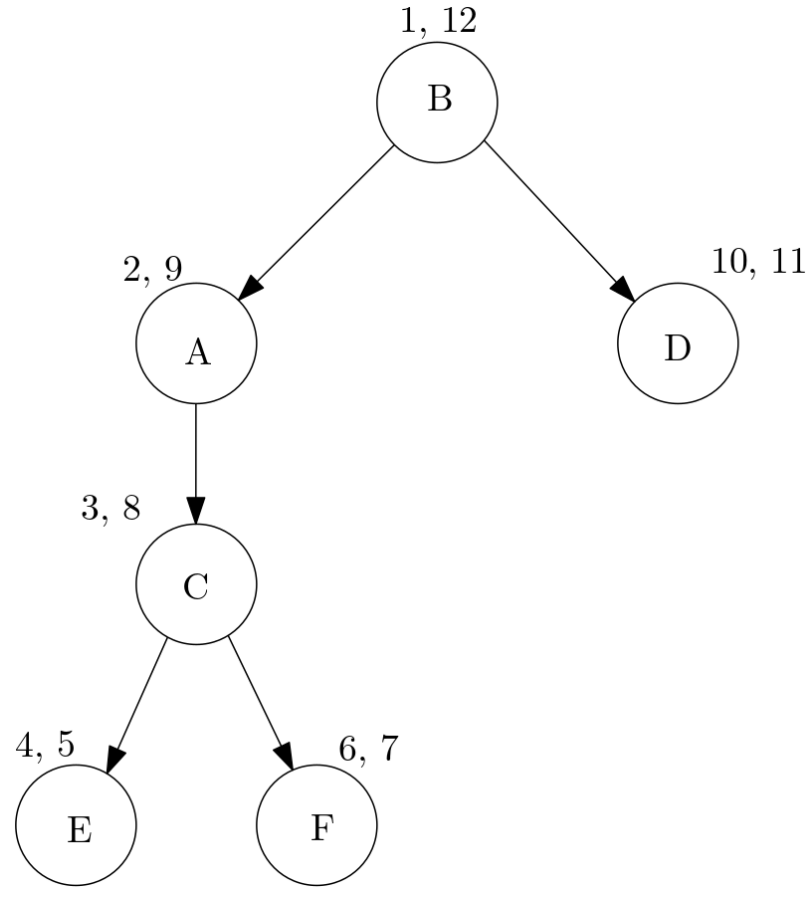
- In any path, the vertices appear in increasing linearized order

SHORTEST PATHS IN DAGS

- Linearize the dag using DFS (decreasing post numbers)
- Visit the vertices in sorted order
- Update all outgoing edges

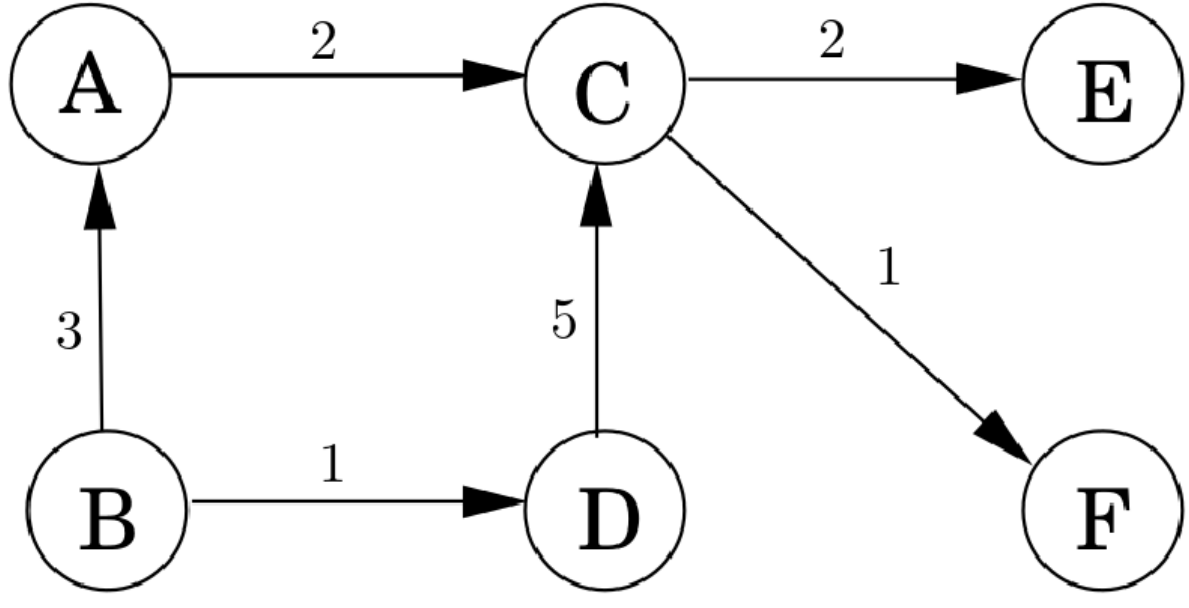
```
procedure dag-shortest-paths(G, l, s)
Input:    Dag G = (V, E);
          edge lengths {l_e : e ∈ E}; vertex s ∈ V
Output:   For all vertices u reachable from s, dist(u) is set
          to the distance from s to u.

for all u ∈ V:
    dist(u) = ∞
    prev(u) = nil

dist(s) = 0
Linearize G
for each u ∈ V, in linearized order:
    for all edges (u, v) ∈ E:
        update(u, v)
```



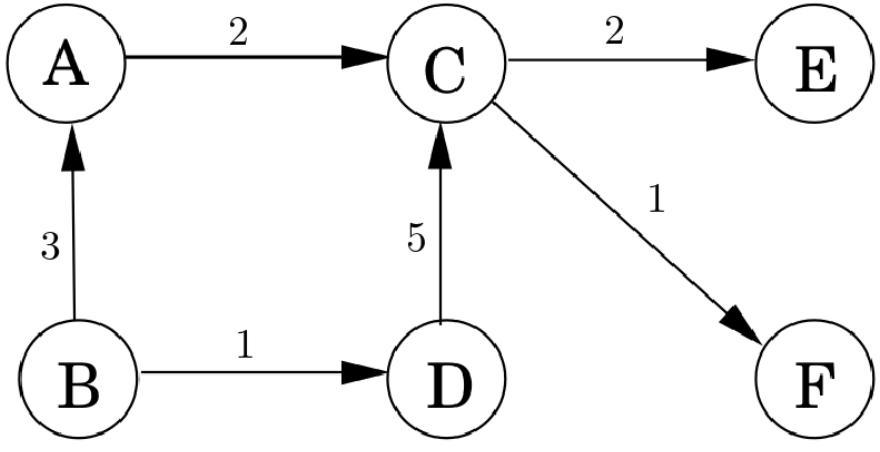


Order: B, D, A, C, F, E

Now let's add some lengths:



Order: B, D, A, C, F, E



$$B, D, A, C, F, E$$

	0	1	2	3	4
A	∞	3	3	3	3
B	0	0	0	0	0
C	∞	∞	6	5	5
D	∞	1	1	1	1
E	∞	∞	∞	∞	7
F	∞	∞	∞	∞	6

GREEDY ALGORITHM

- Builds up a solution piece by piece
- Picks the best option at each stage

MINIMUM SPANNING TREE

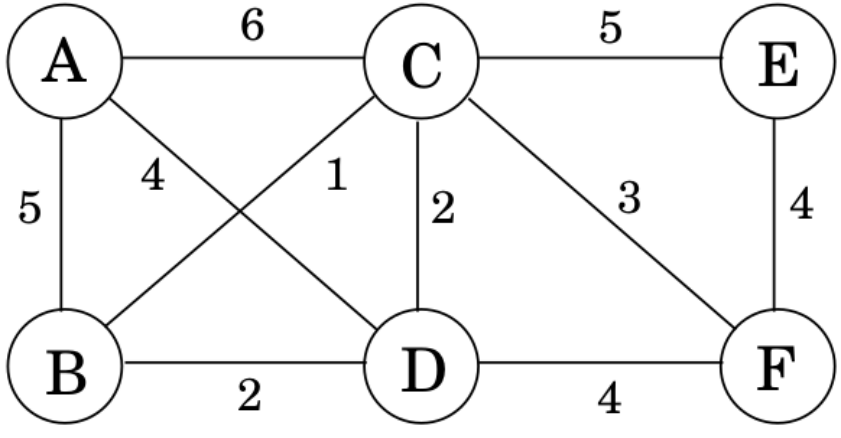
Given an undirected graph $G = (V, E)$ with edge weights w_e,
find a tree $T = (V, E')$, where $E' \subseteq E$,
that minimizes

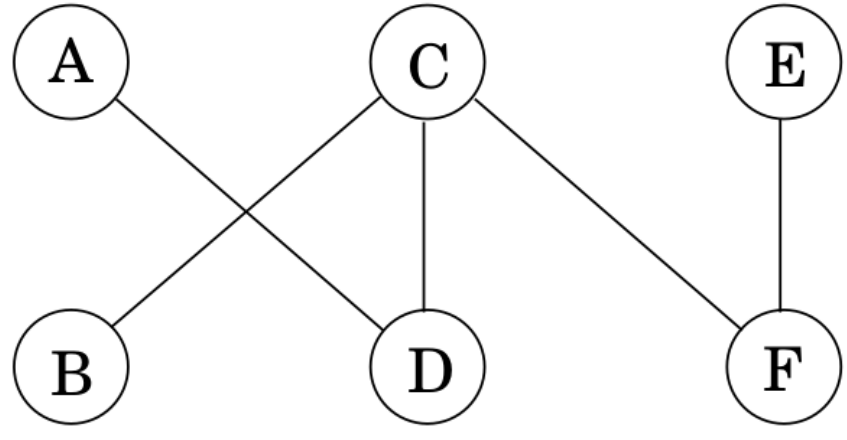
$$\text{weight}(T) = \sum_{e \in E'} w_e$$

KRUSKAL'S ALGORITHM

Start with the empty set

Repeatedly add the next lightest edge that doesn't create a cycle


A
C
E
B
D
F
6
5
5
4
1
2
3
4
2
4


A
C
E
B
D
F

ASIDE: TREES

- undirected graph
- connected
- acyclic

NICE TREE PROPERTIES

A tree on n nodes has $n - 1$ edges

Any connected, undirected graph with $|E| = |V| - 1$ is a tree

An undirected graph is a tree iff there is a unique path between any pair of nodes

So Kruskal's algorithm starts with n trees (single vertices)

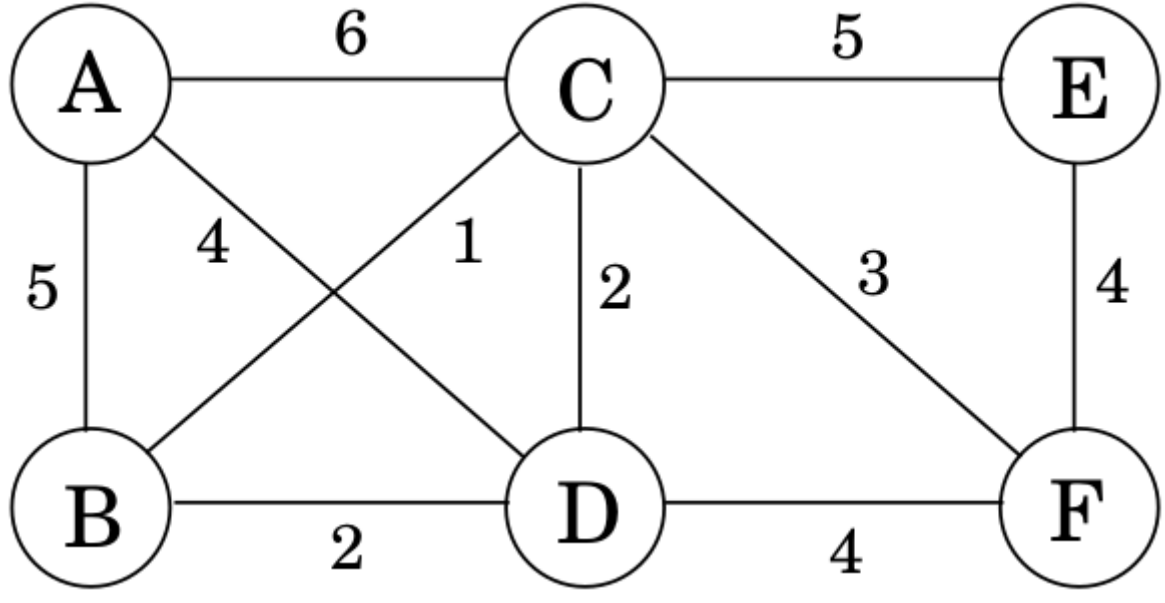
At each stage it adds an edge to connect two trees T_1 and T_2

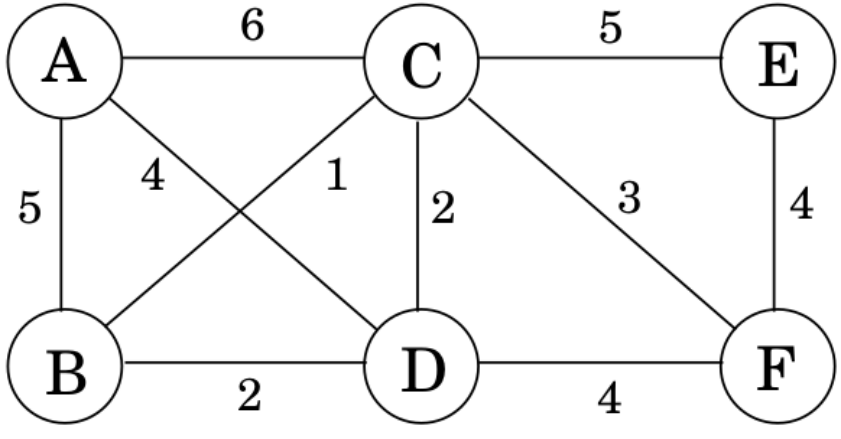
Let the cut be T_1 and $V - T_1$.

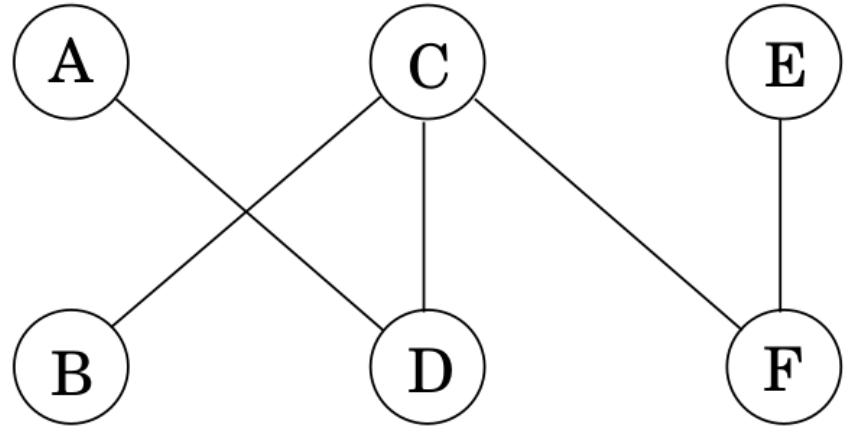
Since e is the lightest edge in the graph (that doesn't create a cycle), it must be the lightest edge in this cut.

- makeset(x) -- create a singleton set
- find(x) -- find which set x belongs to
- union(x, y) -- merge sets containing x and y

```
procedure kruskal(G, w)
Input:    A connected undirected graph G = (V, E) with edge weights w_e
Output:   A minimum spanning tree defined by the edges X

for all u ∈ V:
   makeset(u)

X = {}
Sort the edges E by weight
for all edges {u, v} ∈ E, in increasing order of weight:
   if find(u) ≠ find(v):
      add edge {u, v} to X
      union(u, v)
```


A
C
E
B
D
F
6
5
5
4
1
2
3
4
2
4

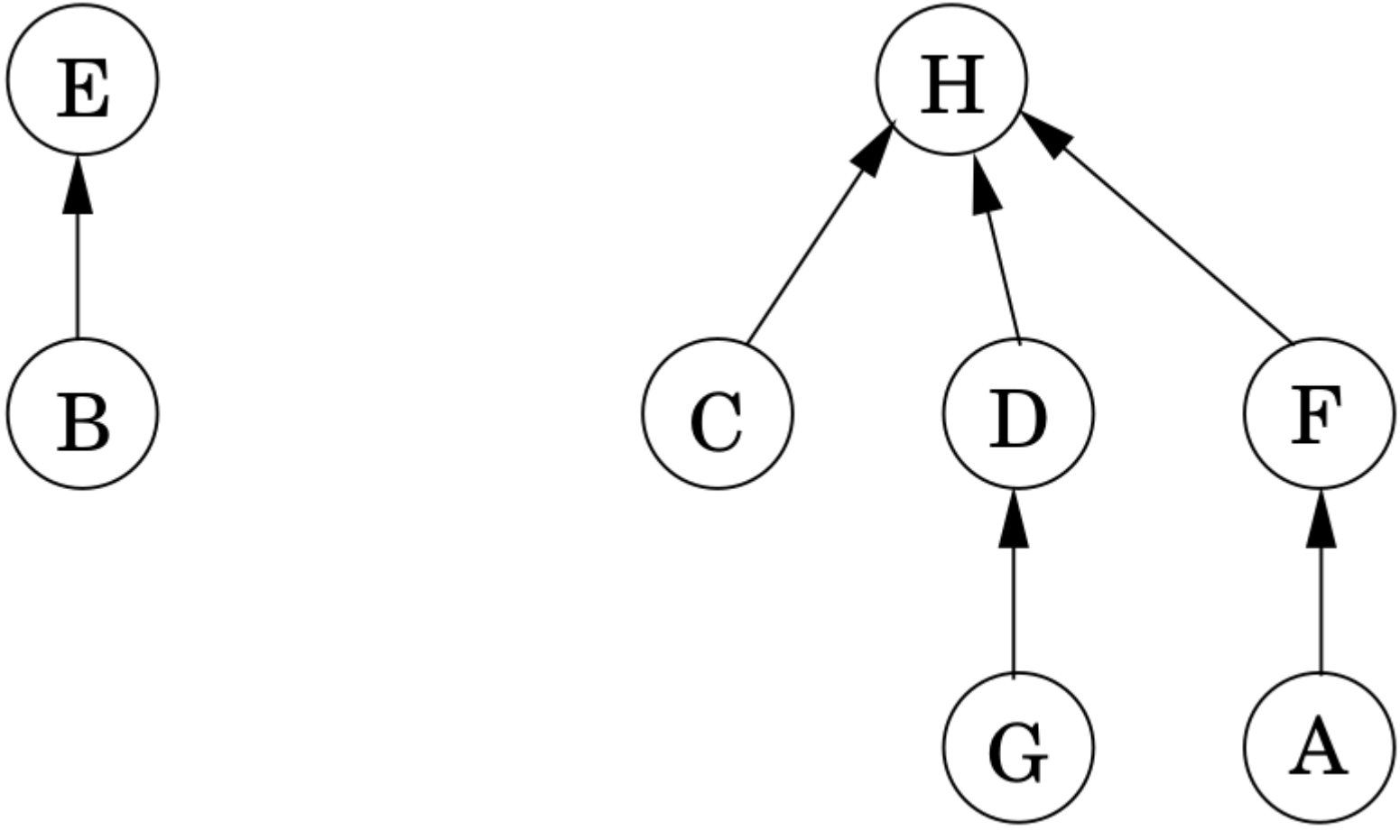

A
C
E
B
D
F
6
5
5
4
1
2
3
4
2
4


A
C
E
B
D
F

UNION BY RANK

- Use directed trees
- Each node has a parent pointer π
- Each node has a rank (height of its subtree)

Directed tree representation of sets $\{B, E\}$ and $\{A, C, D, F, G, H\}$:

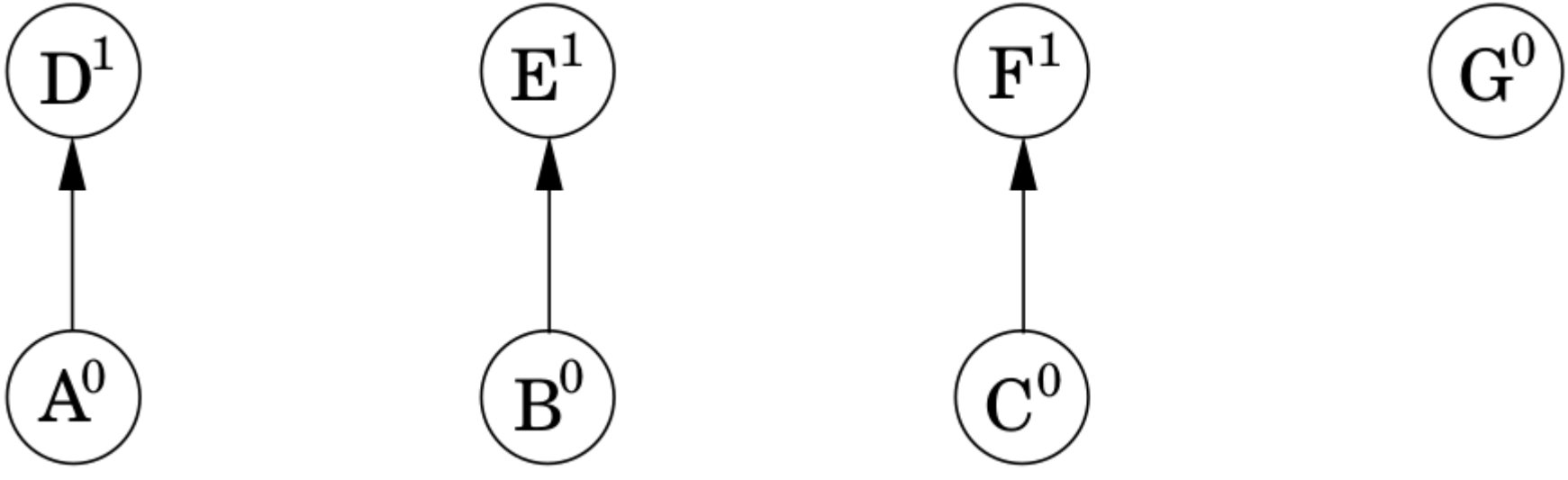


```
procedure makeset(x)
π(x) = x
rank(x) = 0

function find(x)
while x ≠ π(x): x = π(x)
return x
```

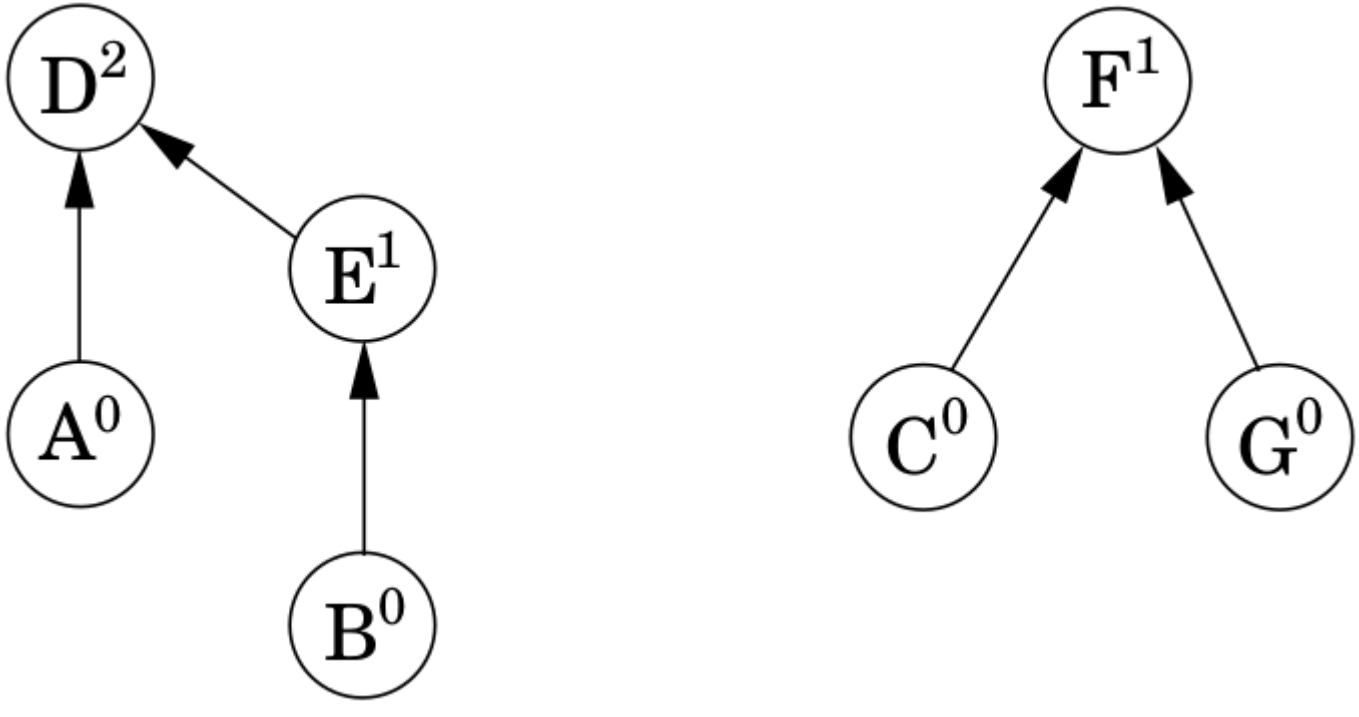
```
procedure union(x, y)
r_x = find(x)
r_y = find(y)
if r_x = r_y:   return
if rank(r_x) > rank(r_y):
    π(r_y) = r_x
else:
    π(r_x) = r_y
    if rank(r_x) = rank(r_y):   rank(r_y) = rank(r_y) + 1
```

After makeset on A through G:

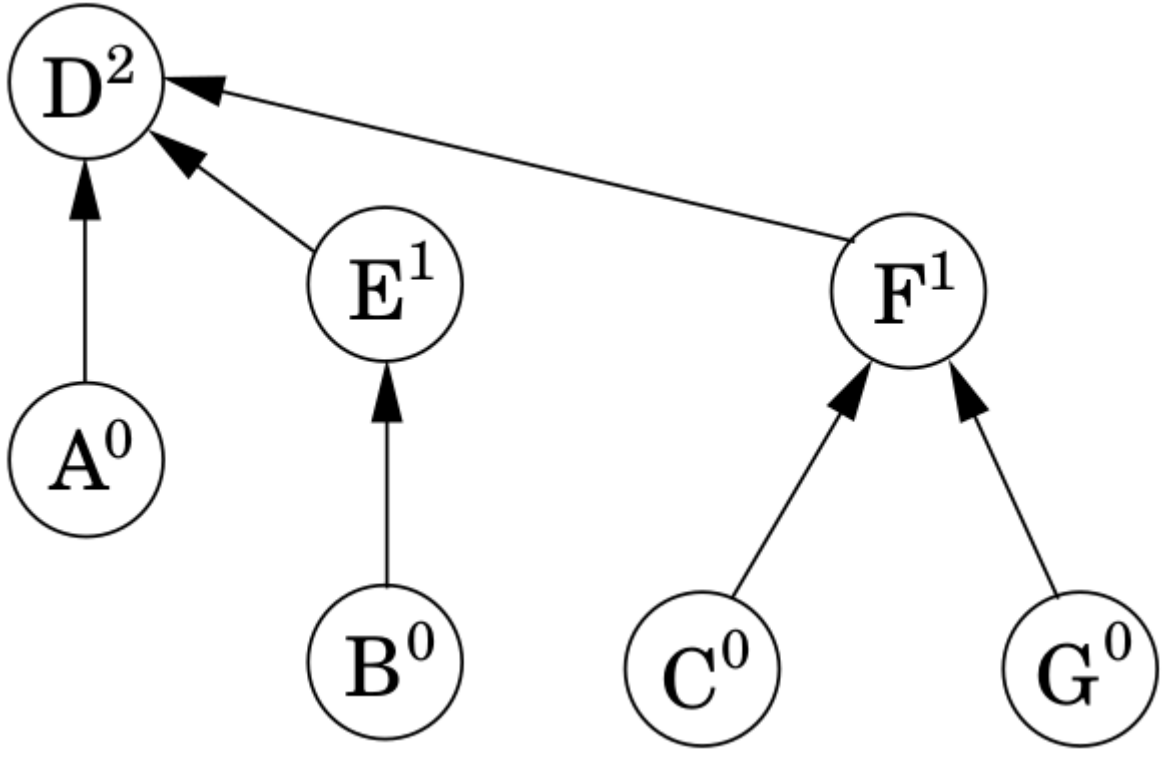
After union(A, D), union(B, E), union(C, F):



After union(C, G), union(E, A):



After union(B, G):



```
procedure kruskal(G, w)
Input:    A connected undirected graph G = (V, E) with edge weights w_e
Output:   A minimum spanning tree defined by the edges X

for all u ∈ V:
   makeset(u)

X = {}
Sort the edges E by weight
for all edges {u, v} ∈ E, in increasing order of weight:
   if find(u) ≠ find(v):
      add edge {u, v} to X
      union(u, v)
```

Time for Kruskal's

- $|V|$ makeset operations
- $2|E|$ find operations
- $|V| - 1$ union operations

Time for Kruskal's

- $O(|E| \log |V|)$ for sorting edges
- $O(|E| \log |V|)$ for union and find

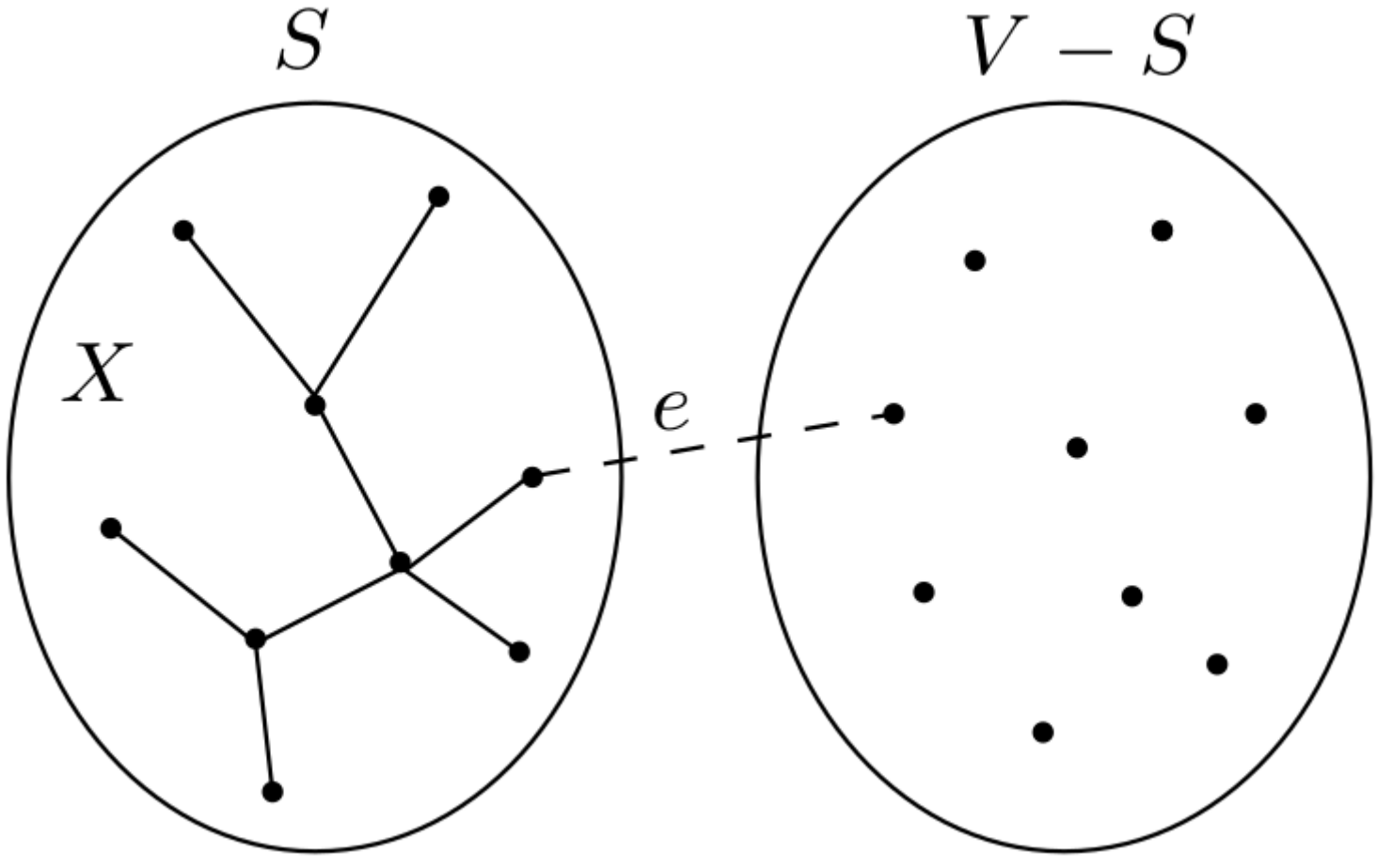
In general, this form of greedy scheme will work:

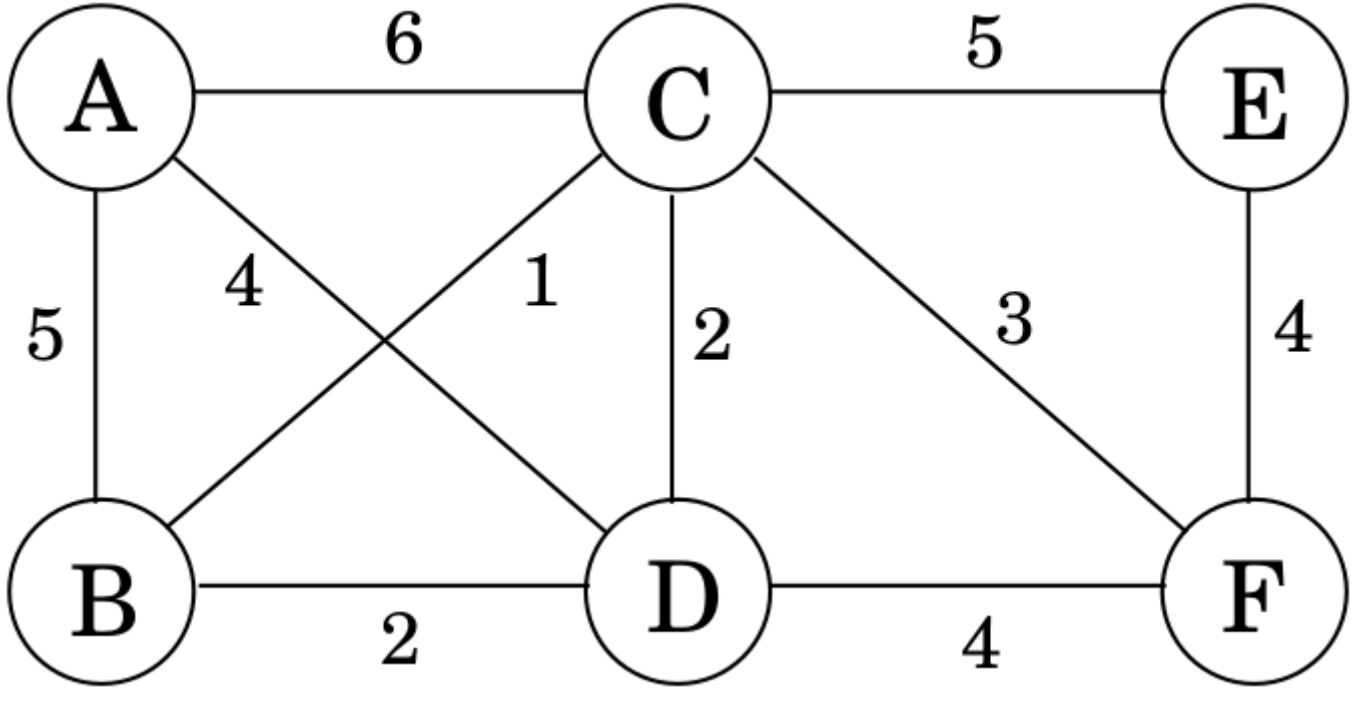
$X = \{\ \}$ (edges picked so far)
repeat until $|X| = |V| - 1$:
  pick a set $S \subset V$ for which X has no edges between S and $V - S$
  let $e \in E$ be the minimum-weight edge between S and $V - S$
  $X = X \cup \{e\}$

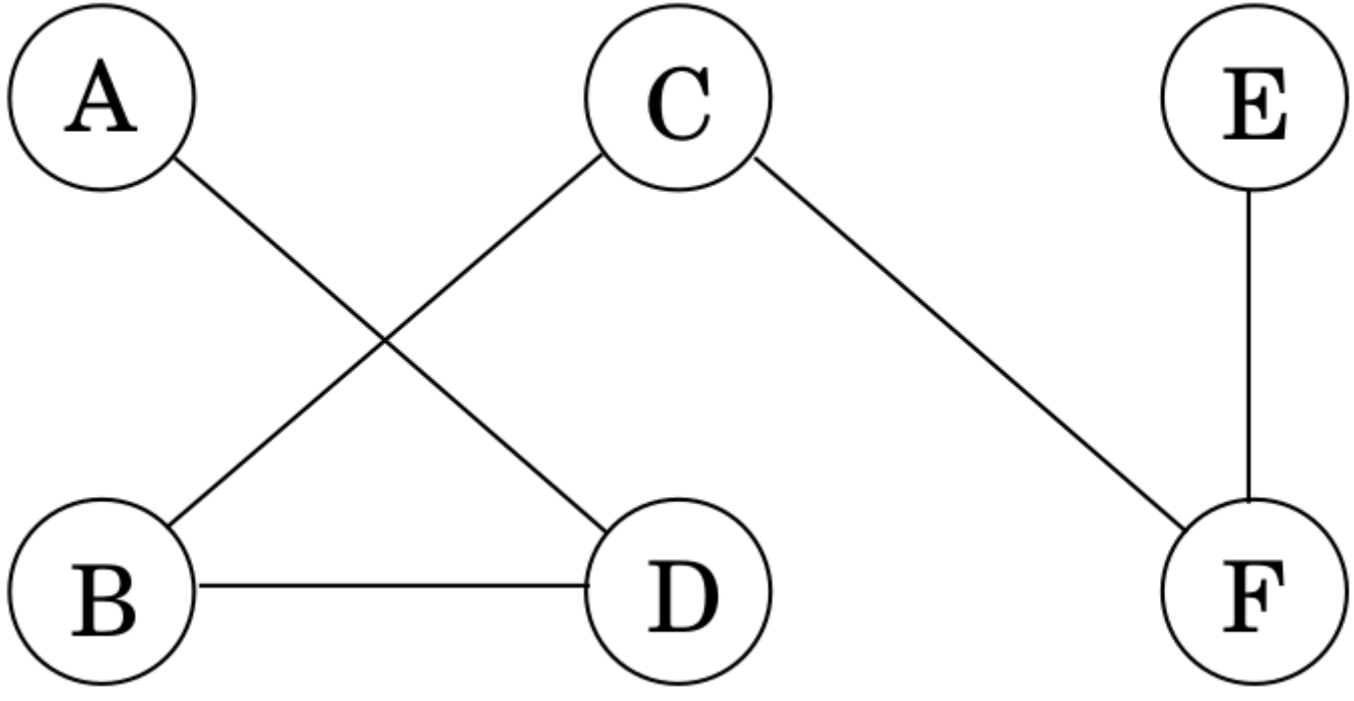
PRIM'S ALGORITHM

- Edge set X is always a subtree of G
- X grows by one (lightest) edge each time

(or think of S as growing to include the cheapest next vertex)


S
$V - S$
X
e


A
C
E
B
D
F
6
5
4
1
5
2
3
4
2
4


A
C
E
B
D
F

HUFFMAN ENCODING

Suppose not all symbols appear equally often:

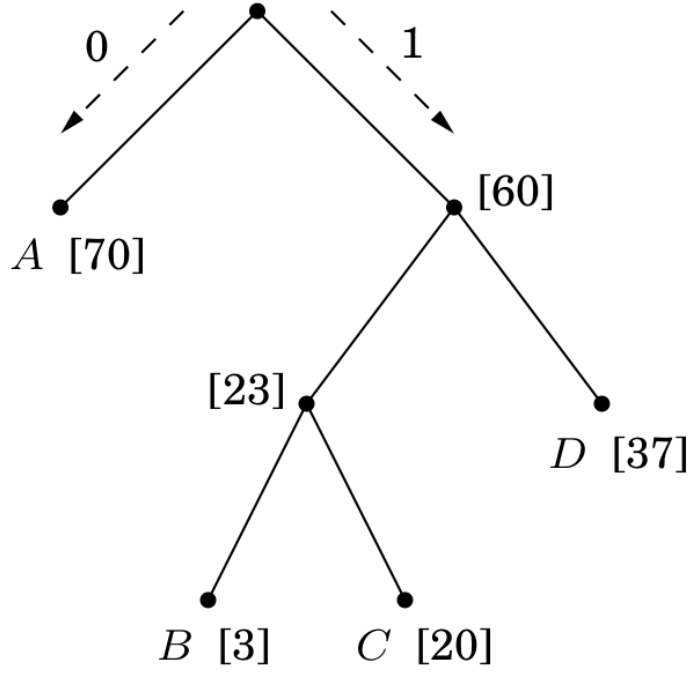
Symbol	Frequency
A	70 million
B	3 million
C	20 million
D	37 million

Could we use a variable-length encoding to make A shorter, at the expense of longer encodings for B, C, and D?

We require that the encoding be **prefix-free**: no codeword can be a prefix of another.

Symbol	**Codeword**
A	0
B	100
C	101
D	11

Any prefix-free encoding can be drawn as a **full** binary tree (every node has 0 or 2 children)



When every codeword had length 2, it took 260 Mb to encode 130 million symbols.

With the new encoding:

```
A: 70 million * 1 bit  = 70 Mb
B:  3 million * 3 bits =  9 Mb
C: 20 million * 3 bits = 60 Mb
D: 37 million * 2 bits = 74 Mb
```

Total: 213 Mb!

How do we find an optimal coding tree?

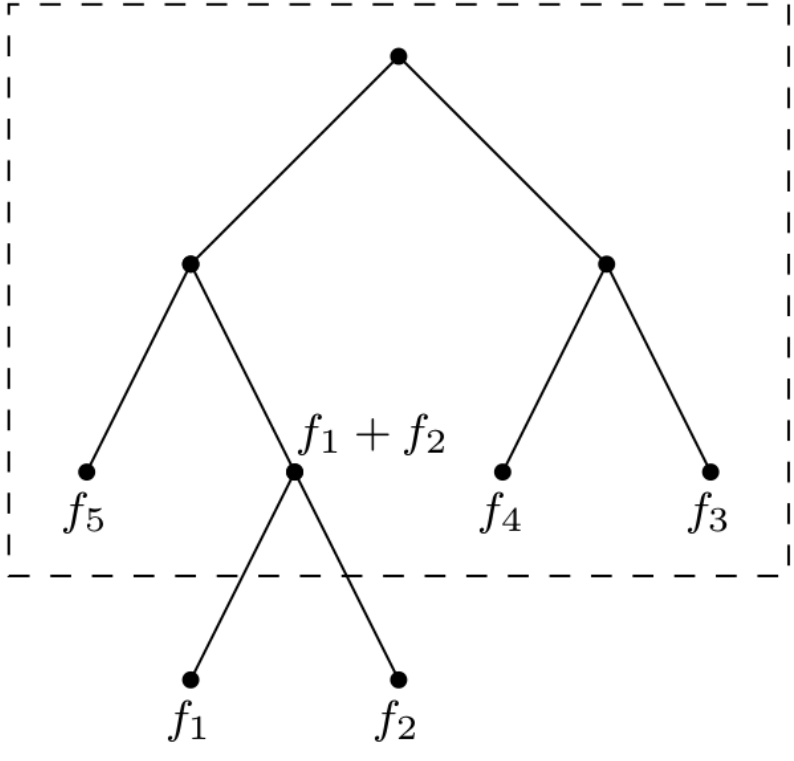
Minimize cost:

$$\sum_{i=1}^{n} f_i \cdot (\text{depth of } i\text{th symbol in tree})$$

Another way to look at this:

- the frequency of each internal node is the sum of its children's frequencies
- each time we traverse an edge in the tree, we output a bit
- cost of a tree is the sum of the frequency of all non-root nodes

This suggests a greedy algorithm:



- Take two smallest with frequencies f_i and f_j
- Make them children of a new node
- New node has frequency $f_i + f_j$
- Repeat

```
procedure Huffman(f)
Input:  An array f[1···n] of frequencies
Output:  An encoding tree with n leaves

let H be a priority queue of integers, ordered by f
for i = 1 to n:  insert(H, i)
for k = n + 1 to 2n − 1:
  i = deletemin(H),  j = deletemin(H)
  create a node numbered k with children i, j
  f[k] = f[i] + f[j]
  insert(H, k)
```

How long does it take to run Huffman?

If we use a binary heap,

- n inserts: $n \cdot O(\log n)$
- n iterations of k loop: $n(3 \cdot O(\log n))$

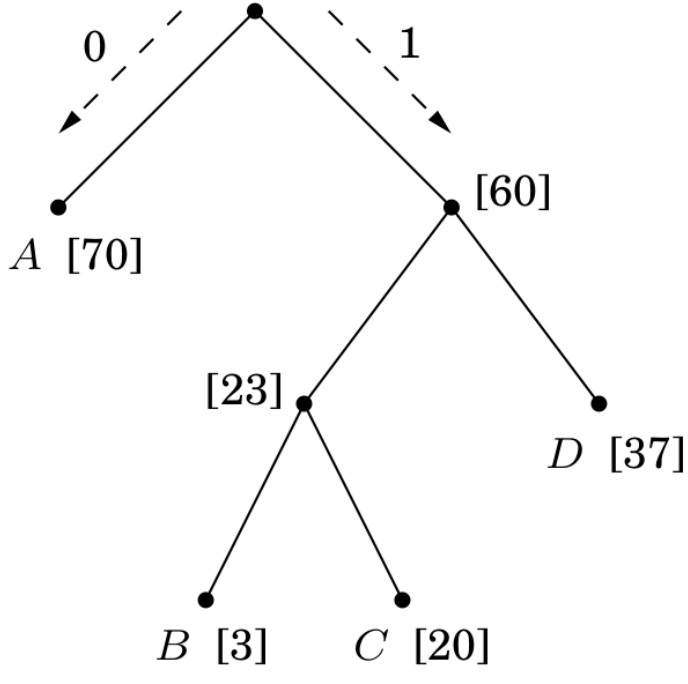
Total: $O(n \log n)$

Symbol	Frequency
A	70 million
B	3 million
C	20 million
D	37 million

B + C = 23 million

B/C + D = 60 million

B/C/D + A = 130 million



B + C = 23 million

B/C + D = 60 million

B/C/D + A = 130 million

HORN FORMULAE

Horn formulas are made up of

- Boolean variables
- Horn clauses
  - Implication
  - Negative clause

Implication:

$$(z \wedge w) \Rightarrow u$$

Facts are written with an empty LHS:

$$\Rightarrow u$$

Negative clauses:

$$(\bar{u} \vee \bar{v} \vee \bar{y})$$

Given a set of clauses, find an assignment of true/false values to variables that makes all clauses true.

(called a **satisfying** assignment)

Greedy algorithm for Horn formulae:

```
set all variables to false

while there is an implication that is not satisfied:
    set the right-hand variable of the implication to true

if all pure negative clauses are satisfied:
    return the assignment
else:
    return "formula is not satisfiable"
```

For example, suppose we have these clauses

$$(w \wedge y \wedge z) \Rightarrow x,$$
$$(x \wedge z) \Rightarrow w,$$
$$x \Rightarrow y,$$
$$\Rightarrow x,$$
$$(x \wedge y) \Rightarrow w,$$
$$(\bar{w} \vee \bar{x} \vee \bar{z}),$$
$$(\bar{z})$$

Initially, everything is false

- x must be true due to $\Rightarrow x$
- y must be true due to $x \Rightarrow y$
- w must be true due to $(x \wedge y) \Rightarrow w$

Result: $\{w : T, x : T, y : T, z : F\}$

$$(w \wedge y \wedge z) \Rightarrow x,$$
$$(x \wedge z) \Rightarrow w,$$
$$x \Rightarrow y,$$
$$\Rightarrow x,$$
$$(x \wedge y) \Rightarrow w,$$
$$(\bar{w} \vee \bar{x} \vee \bar{y}),$$
$$(\bar{z})$$

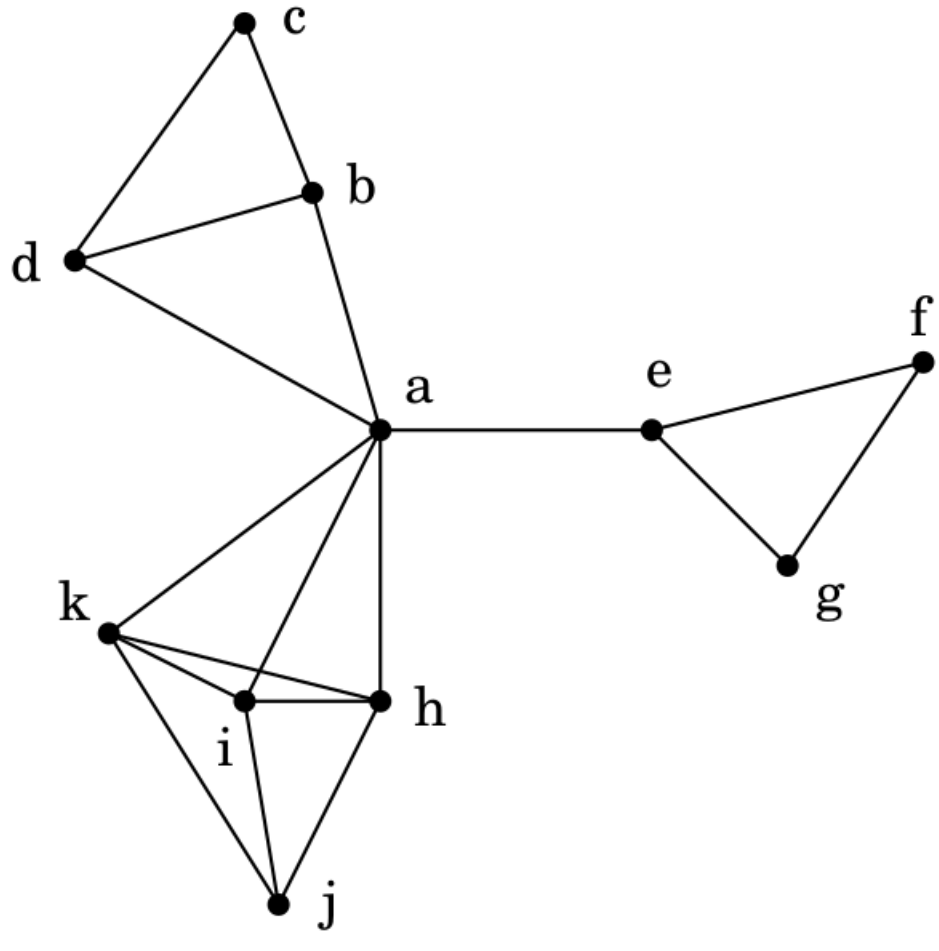
Initially, everything is false

- x must be true due to $\Rightarrow x$
- y must be true due to $x \Rightarrow y$
- w must be true due to $(x \wedge y) \Rightarrow w$

Result: cannot be satisfied

SET COVERS

Given a graph, find a set of vertices that **cover** all vertices in the graph.



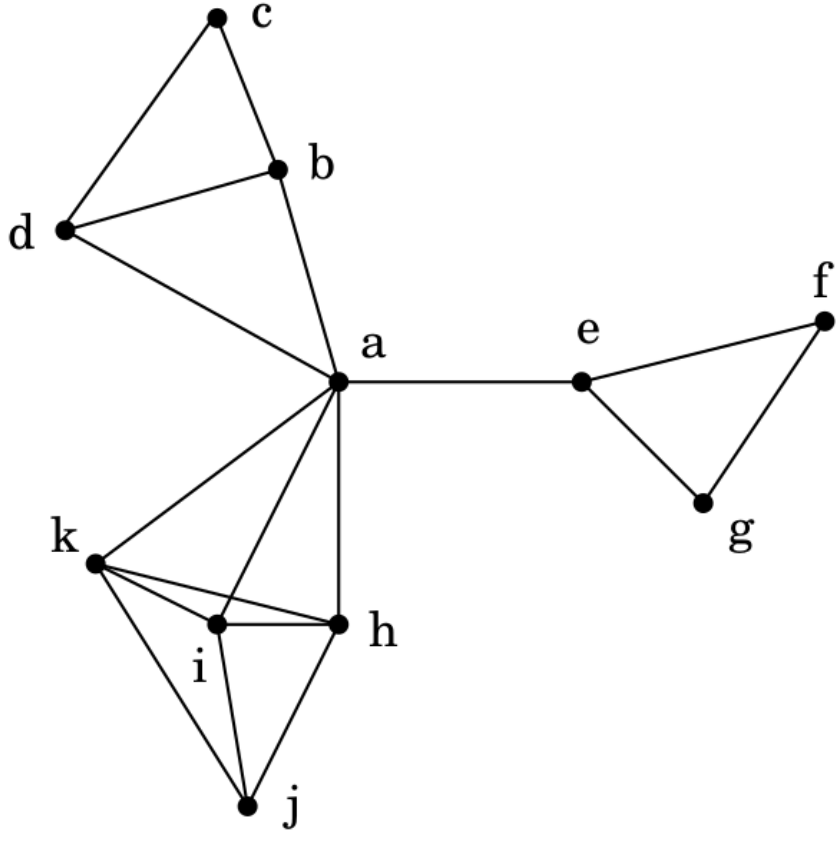
SET COVERING

Let S_v be the set of vertices adjacent to vertex v (including v itself).

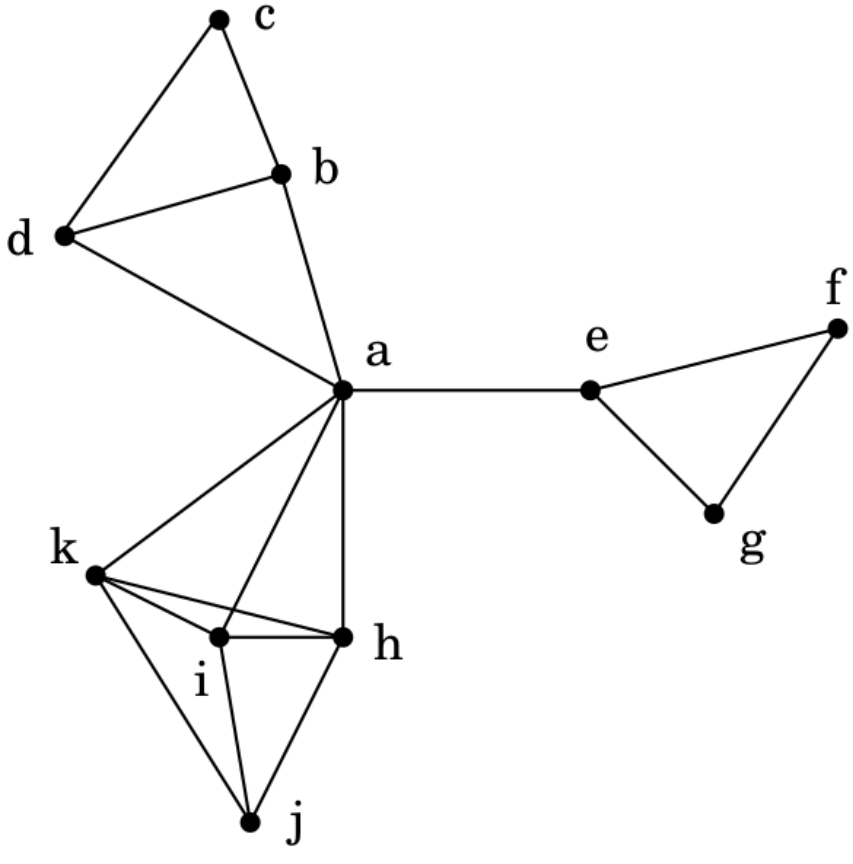
Given V and $S_1, S_2, \ldots, S_m$, find a minimal selection of S_i whose union is B

A greedy approach:

- Until everything in B is covered:
  - Pick the S_i with the largest number of uncovered elements

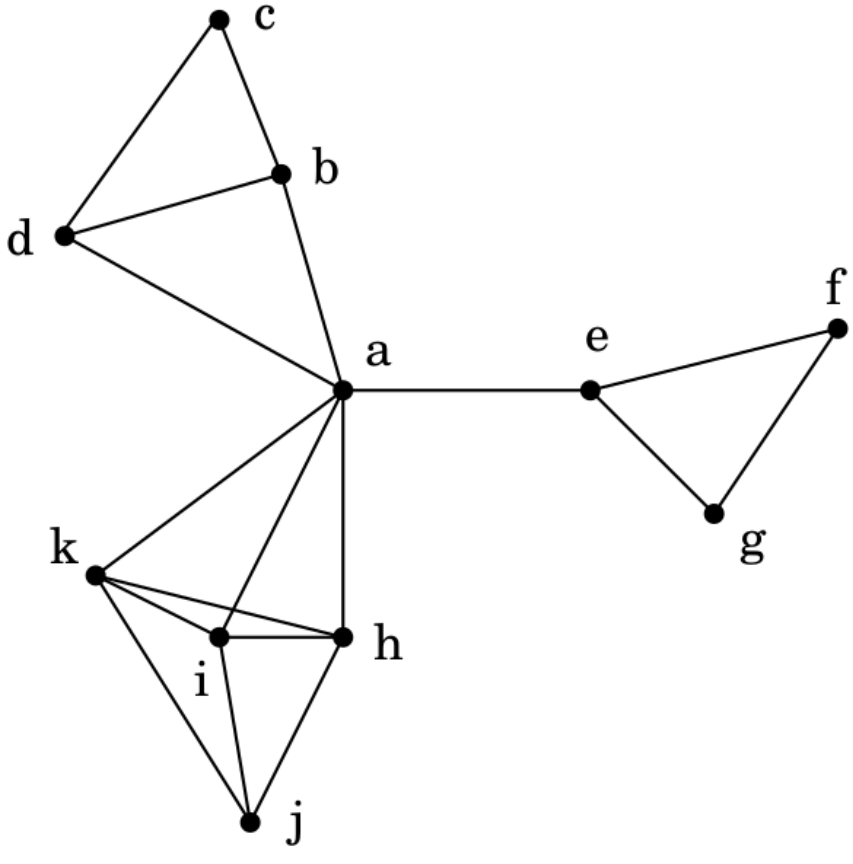


- $S_a = \{a, b, d, e, k, h, i\}$
- $S_b = \{a, b, c, d\}$
- $S_c = \{b, c, d\}$
- $S_d = \{a, b, c, d\}$
- $S_e = \{a, e, f, g\}$
- ...



We pick these towns:

a, c, j, and f (or g)



But consider these instead:

b, e, i

The greedy algorithm is not optimal!

But it's not too bad...

- if B has n elements,
- and the optimal cover has k sets,
- then the greedy algorithm uses at most $k \ln n$ sets.