Graphs

Data Structures and Parallelism

Announcements

P3 checkpoint 1 today

Checkpoint 2 on Wednesday

Parallelism exercises are due Monday.

We’ll announce details of using tokens to redo exercises over the weekend.

ADTs so far

We’ve seen:

Queues and Stacks

- Our data points have some order we’re maintaining

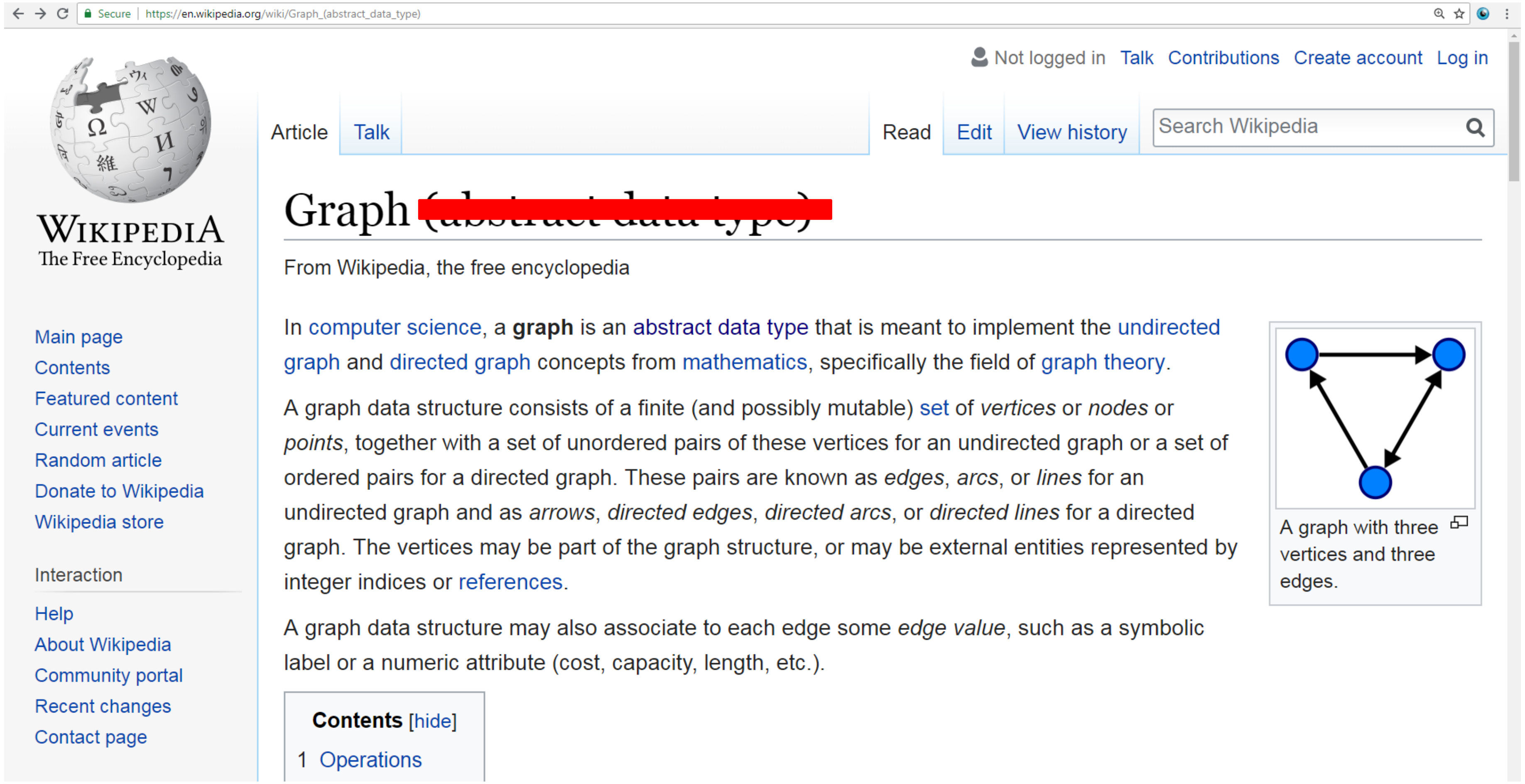
Priority Queues

- Our data had some priority we needed to keep track of.

Dictionaries

- Our data points came as (key, value) pairs.

Graphs


Secure | https://en.wikipedia.org/wiki/Graph_(abstract_data_type)

Wikipedia
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction

Help
About Wikipedia
Community portal
Recent changes
Contact page

Not logged in Talk Contributions Create account Log in

Article Talk Read Edit View history Search Wikipedia

Graph ~~(abstract data type)~~

From Wikipedia, the free encyclopedia

In computer science, a **graph** is an abstract data type that is meant to implement the undirected graph and directed graph concepts from mathematics, specifically the field of graph theory.

A graph data structure consists of a finite (and possibly mutable) set of *vertices* or *nodes* or *points*, together with a set of unordered pairs of these vertices for an undirected graph or a set of ordered pairs for a directed graph. These pairs are known as *edges*, *arcs*, or *lines* for an undirected graph and as *arrows*, *directed edges*, *directed arcs*, or *directed lines* for a directed graph. The vertices may be part of the graph structure, or may be external entities represented by integer indices or references.

A graph data structure may also associate to each edge some *edge value*, such as a symbolic label or a numeric attribute (cost, capacity, length, etc.).

A graph with three vertices and three edges.

Contents [hide]

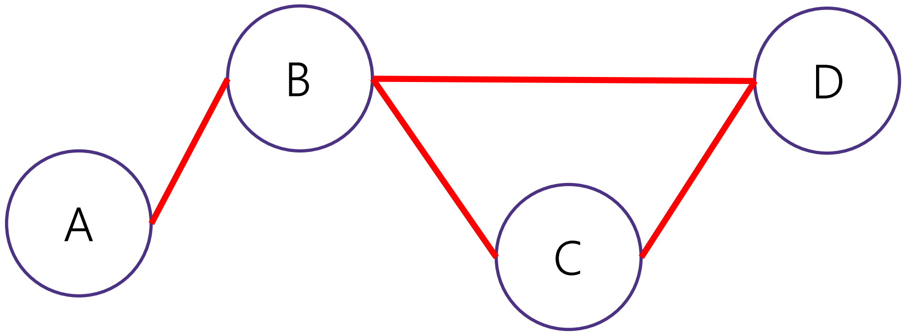
1 Operations

Graphs are too versatile to think of them as only an ADT!

Graphs

Represent data points and the relationships between them.

That's vague.



Formally:

A graph is a pair: G = (V,E)

V: set of **vertices** (aka **nodes**) $\{A, B, C, D\}$

E: set of **edges** $\{(A, B), (B, C), (B, D), (C, D)\}$

- Each edge is a pair of vertices.

Making Graphs

If your problem has **data** and **relationships**, you might want to represent it as a graph

How do you choose a representation?

Usually:

Think about what your “fundamental” objects are

- Those become your vertices.

Then think about how they’re related

- Those become your edges.

Some examples

For each of the following think about what you should choose for vertices and edges.

The internet.

Facebook friendships

Input data for the “6 degrees of Kevin Bacon” game

Course Prerequisites

Some examples

For each of the following think about what you should choose for vertices and edges.

The internet.

- Vertices: webpages. Edges from a to b if a has a hyperlink to b.

Facebook friendships

- Vertices: people. Edges: if two people are friends

Input data for the "6 Degrees of Kevin Bacon" game

- Vertices: actors. Edges: if two people appeared in the same movie
- Or: Vertices for actors and movies, edge from actors to movies they appeared in.

Course Prerequisites

- Vertices: courses. Edge: from a to b if a is a prereq for b.

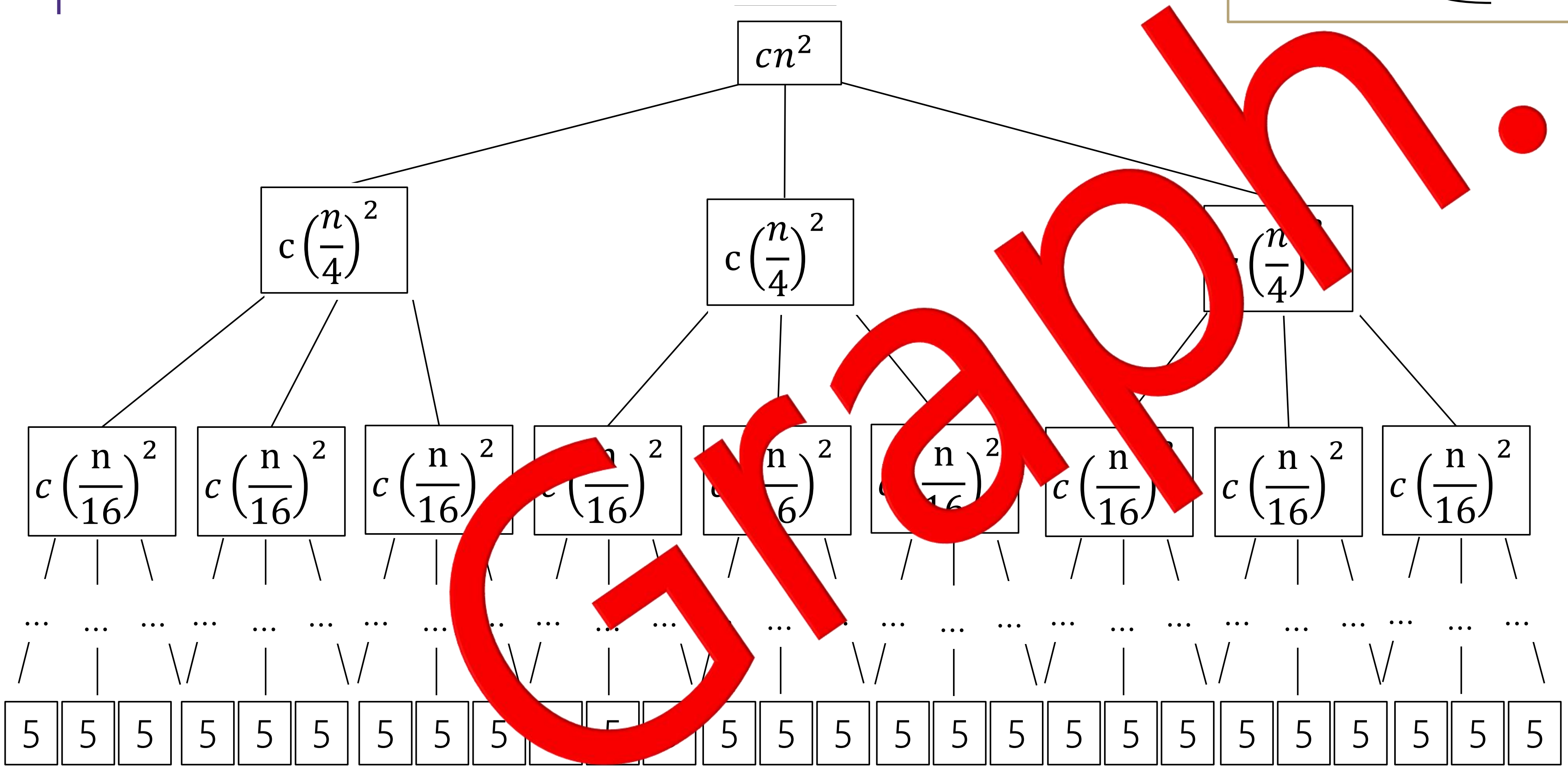
More Graphs

We've already used graphs to represent things in this course:

A LOT

Solving Recurrences III

$$T(n) = \begin{cases} 5 \ when \ n \leq 4 \\ 3T\left(\frac{n}{4}\right) + cn^2 \ otherwise \end{cases}$$



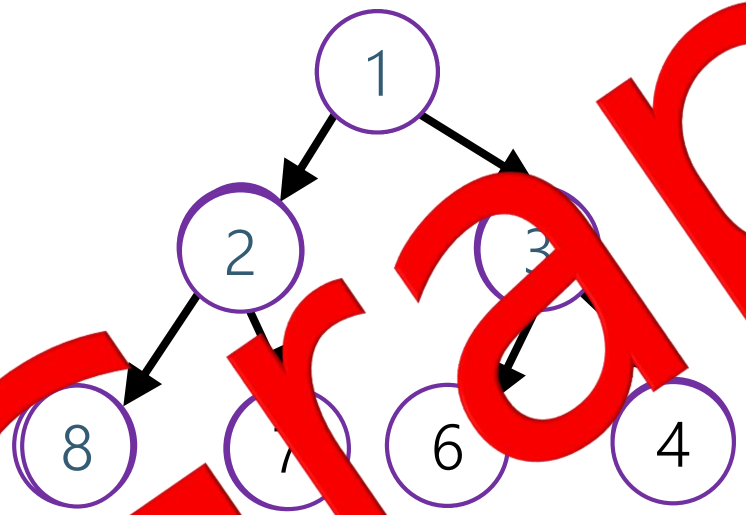
Answer the following questions:

1. What is input size on level i?
2. Number of nodes at level i?
3. Work done at recursive level i?
4. Last level of tree?
5. Work done at base case?
6. What is sum over all levels?

BuildHeap: Only One Possibility

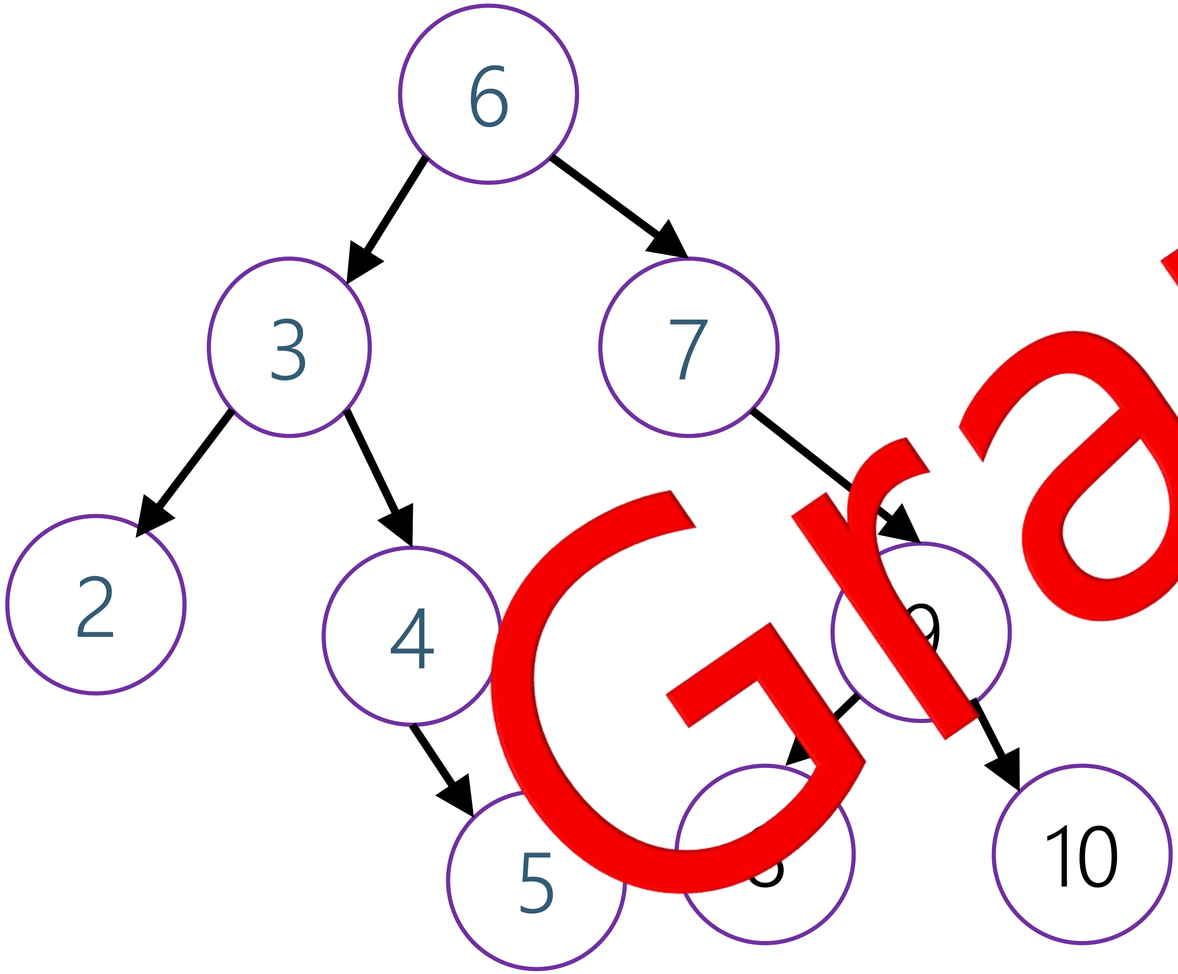
But `StartBottom()` seems to work.

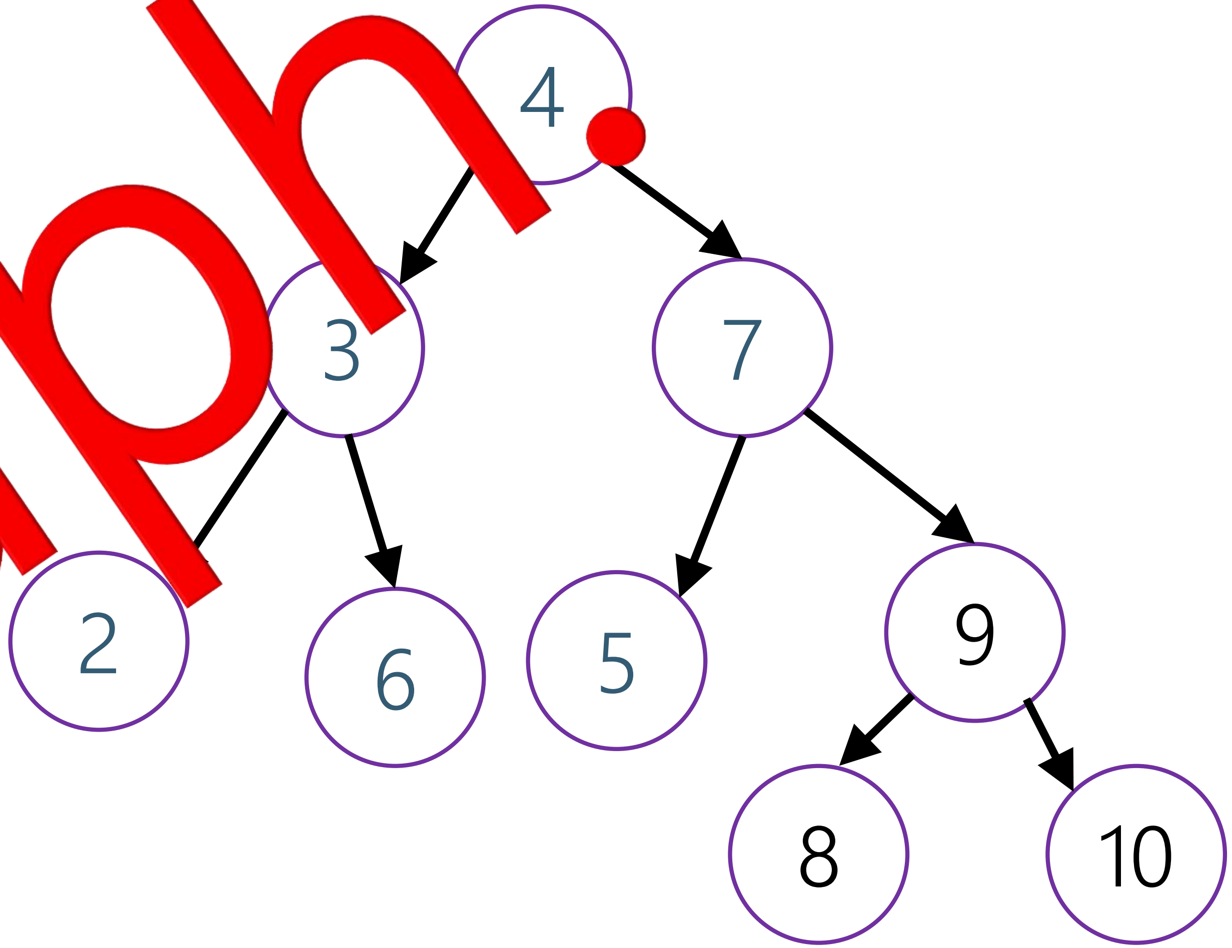
Graph.



Does it always work?

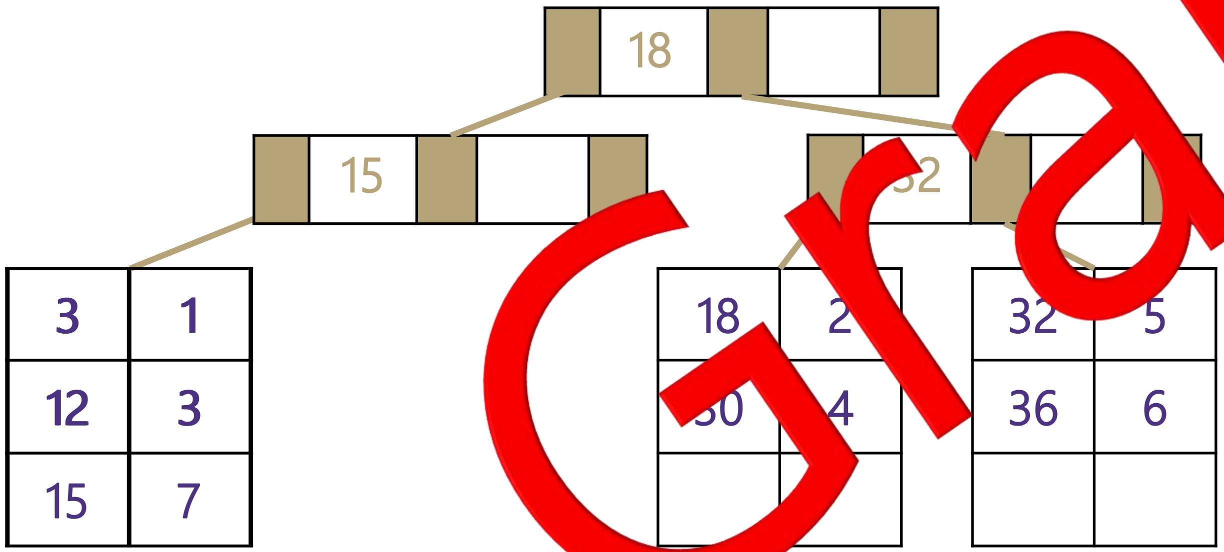
Are These AVL Trees?

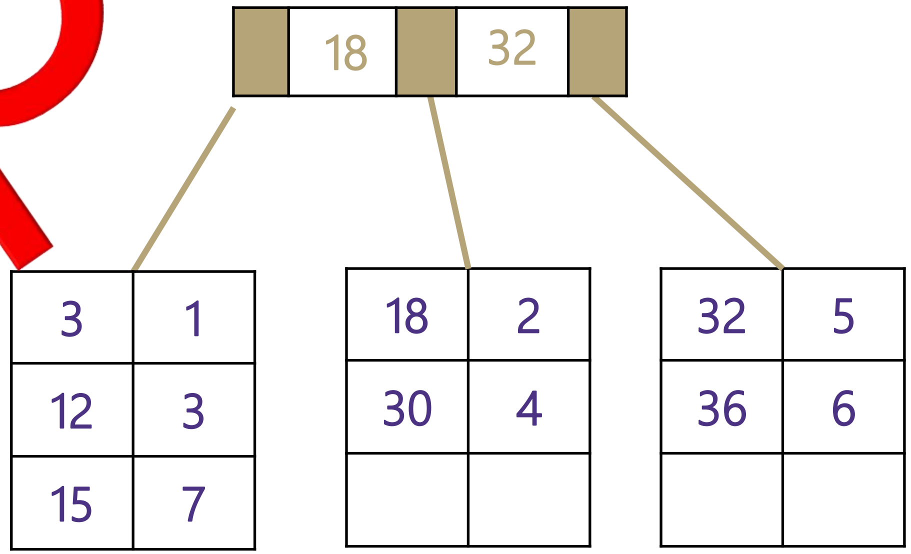


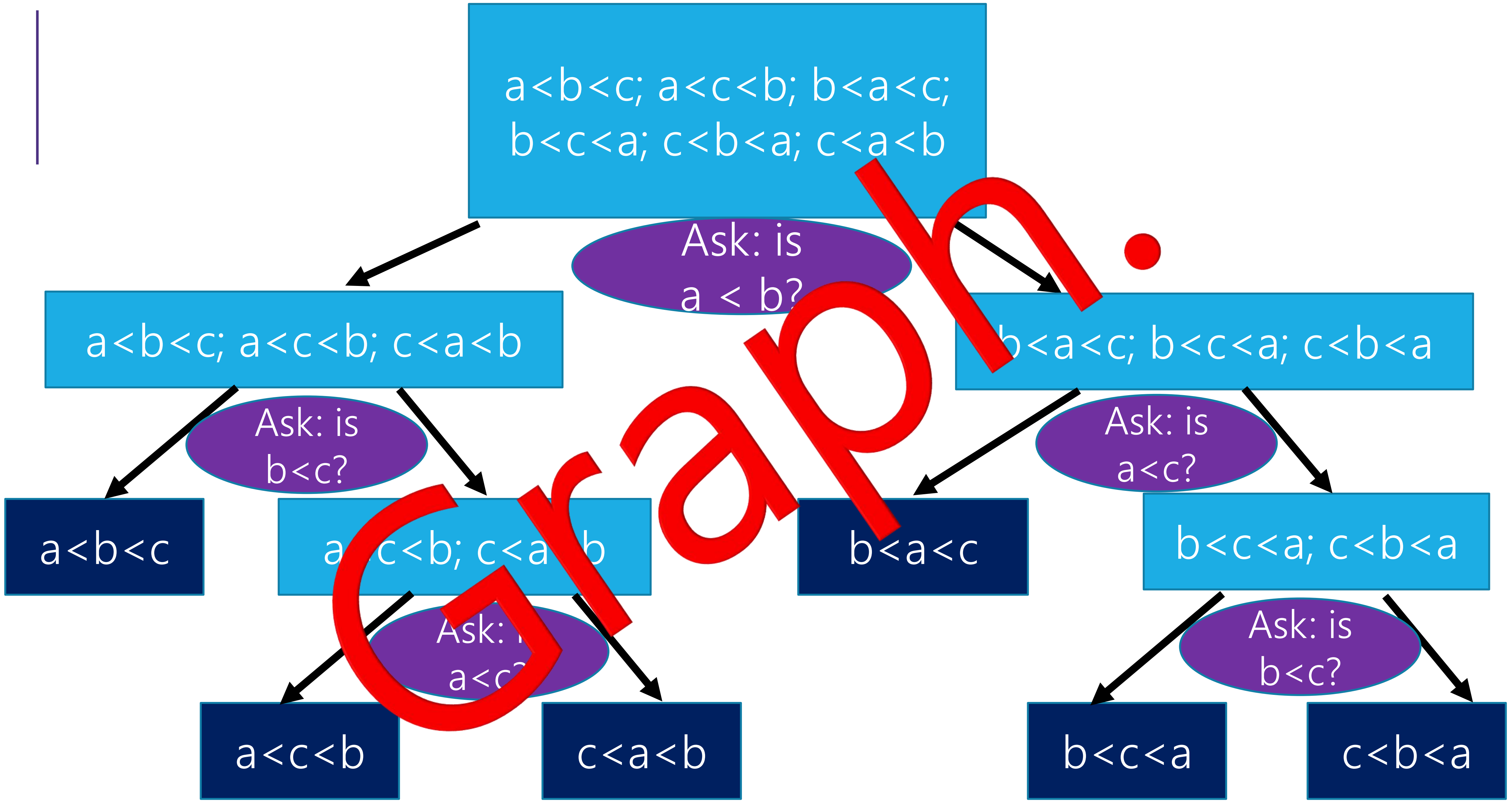


Deletion – merging nodes

Delete 14. Merge up the tree.
Update "signpost" to be smallest key in its right subtree.

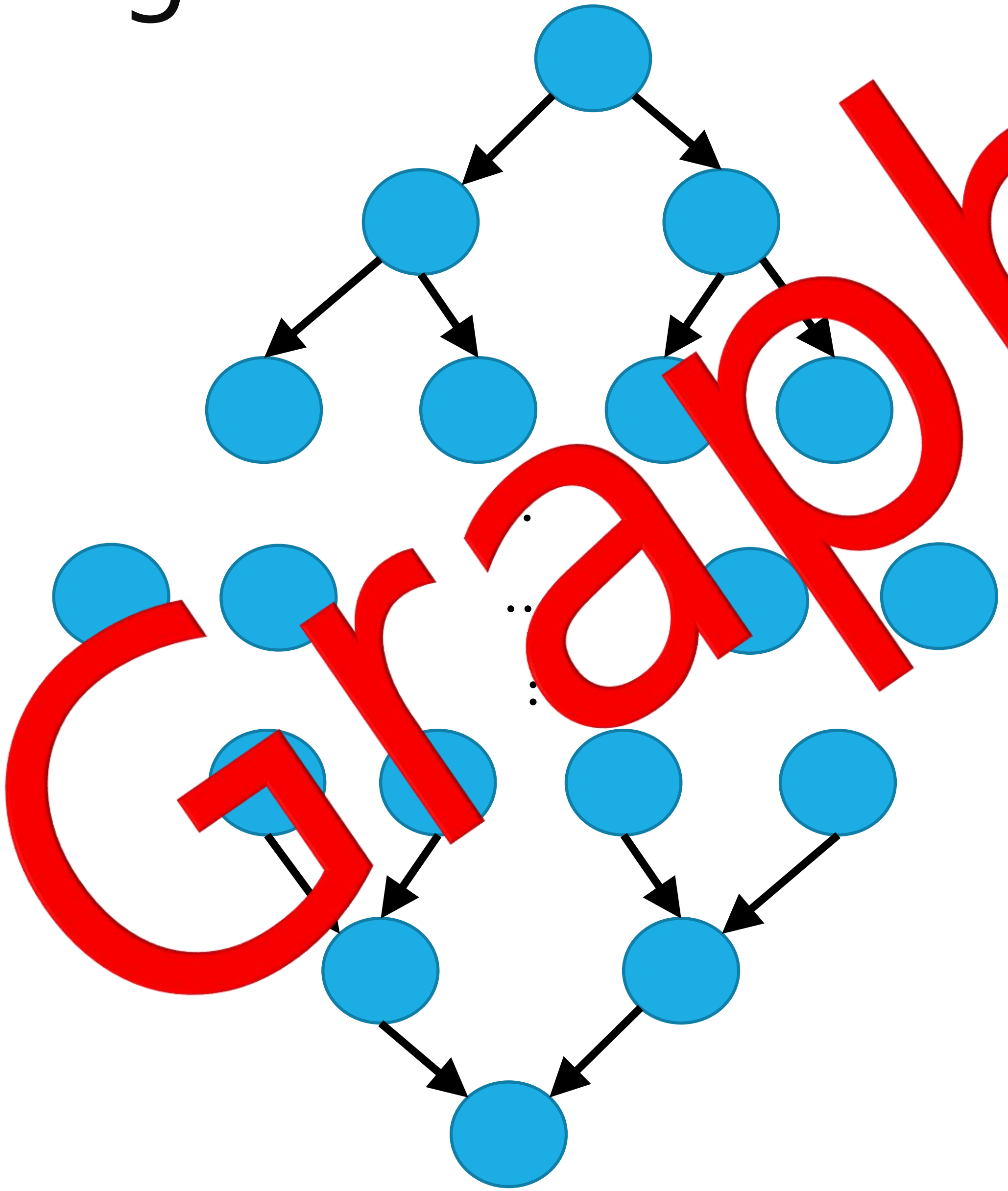





a<b<c; a<c<b; b<a<c;
b<c<a; c<b<a; c<a<b
Ask: is
a < b?
a<b<c; a<c<b; c<a<b
b<a<c; b<c<a; c<b<a
Ask: is
b<c?
Ask: is
a<c?
a<b<c
a<c<b; c<a<b
b<a<c
b<c<a; c<b<a
Ask: is
a<c?
Ask: is
b<c?
a<c<b
c<a<b
b<c<a
c<b<a
Graph.

Useful Diagram

One node per
O(1) operation

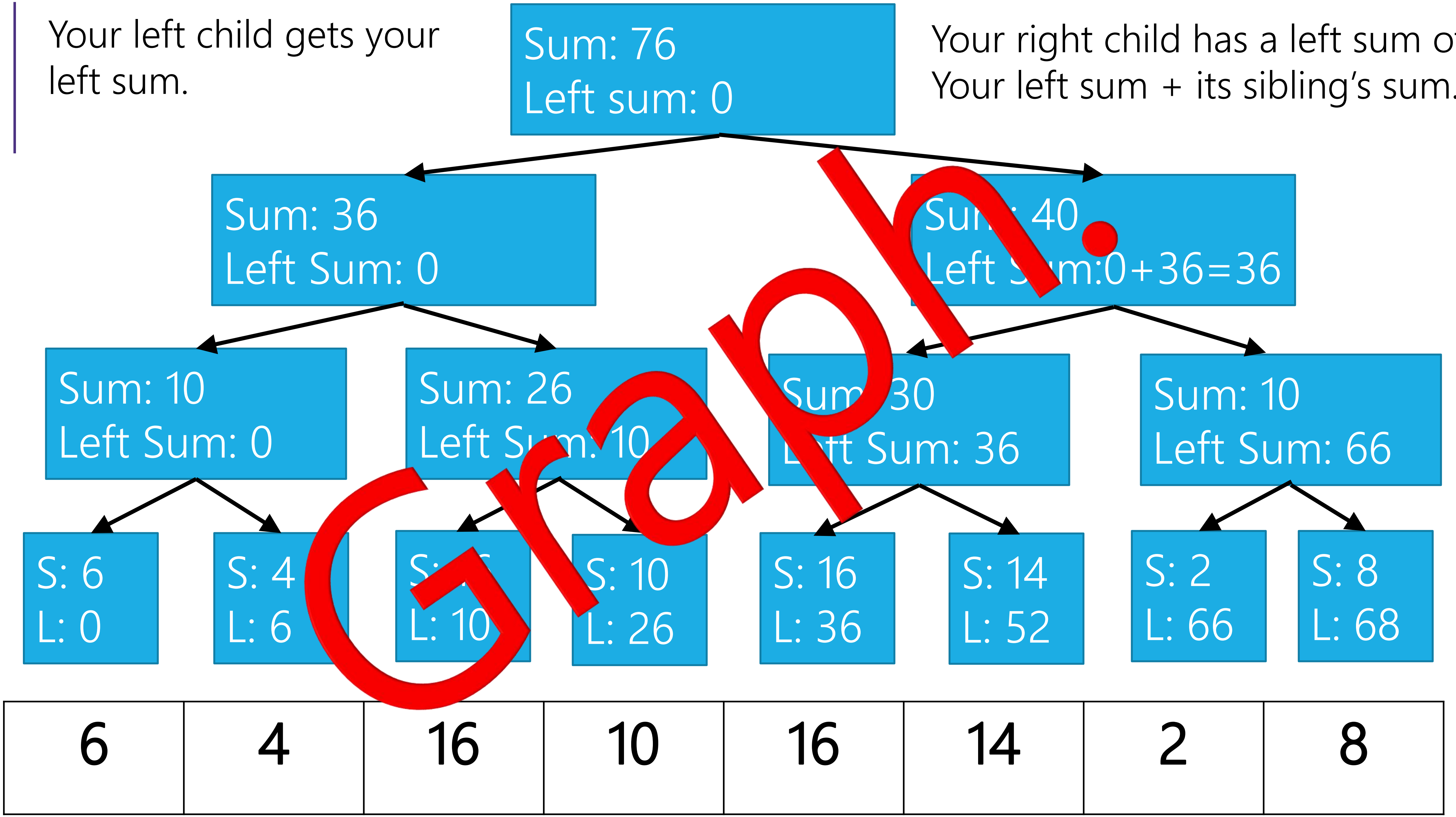


Edge from u to v
I.e. v waits for u.
I.e. v can't start
until u finishes.

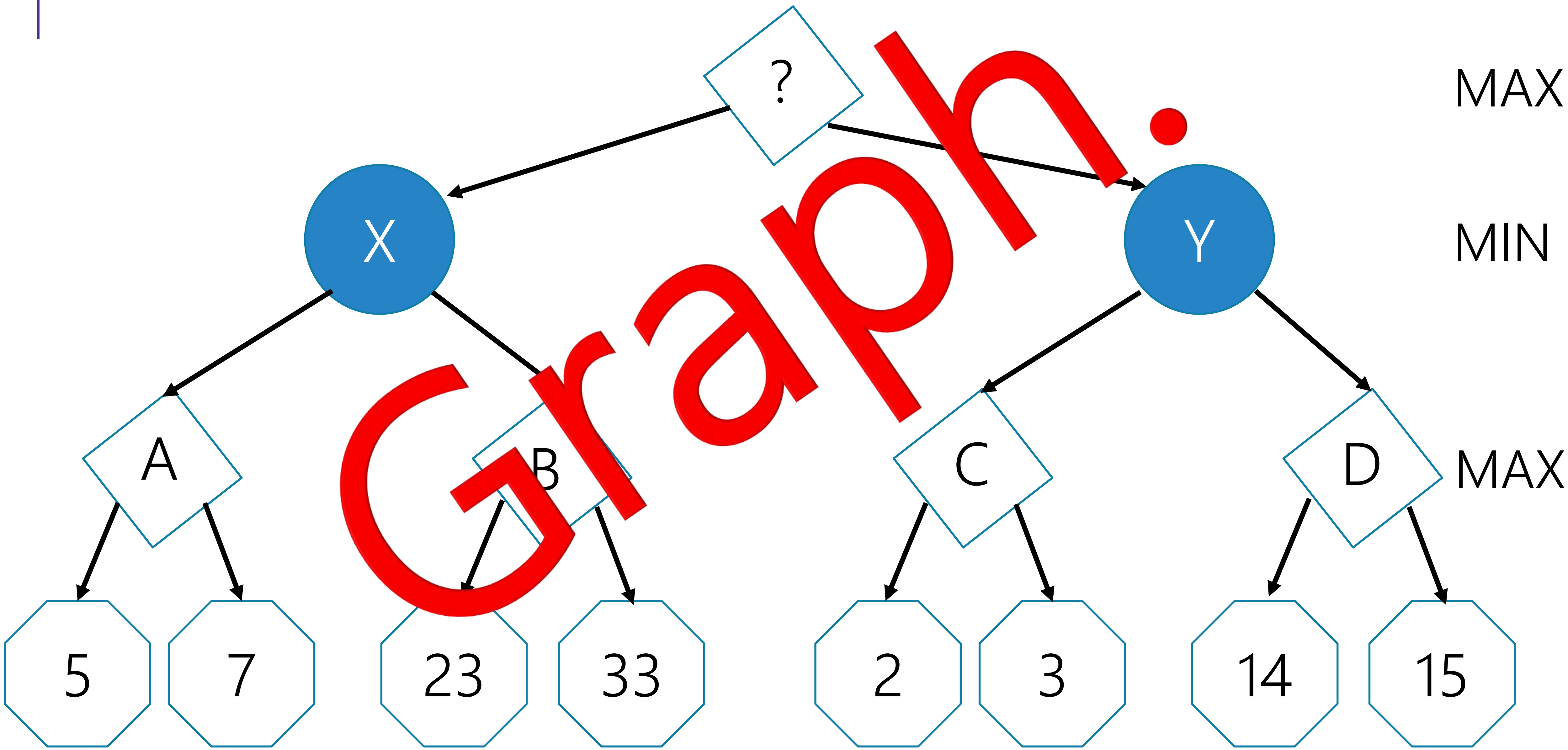
Question: why are
there no cycles in
this graph?

Your left child gets your left sum.

Your right child has a left sum of: Your left sum + its sibling's sum.



Try by hand



More Graphs

EVERYTHING was graphs.

The whole time.

They don’t just show up in data structures.

311: NFAs/DFAs and relations

Compilers: Use graphs to figure out valid compilation orders.

Networking: Building a graph

- To the point that some CS people call graphs “networks”

Circuits: represented as graphs

Some examples

For each of the following think about what you should choose for vertices and edges.

The internet.

- Vertices: webpages. Edges from a to b if a has a hyperlink to b.

Edges have direction

Facebook friendships

- Vertices: people. Edges: if two people are friends

Edges don't have direction

Input data for the "6 Degrees of Kevin Bacon" game

- Vertices: actors. Edges: if two people appeared in the same movie
- Or: Vertices for actors and movies, edge from actors to movies they appeared in.

Course Prerequisites

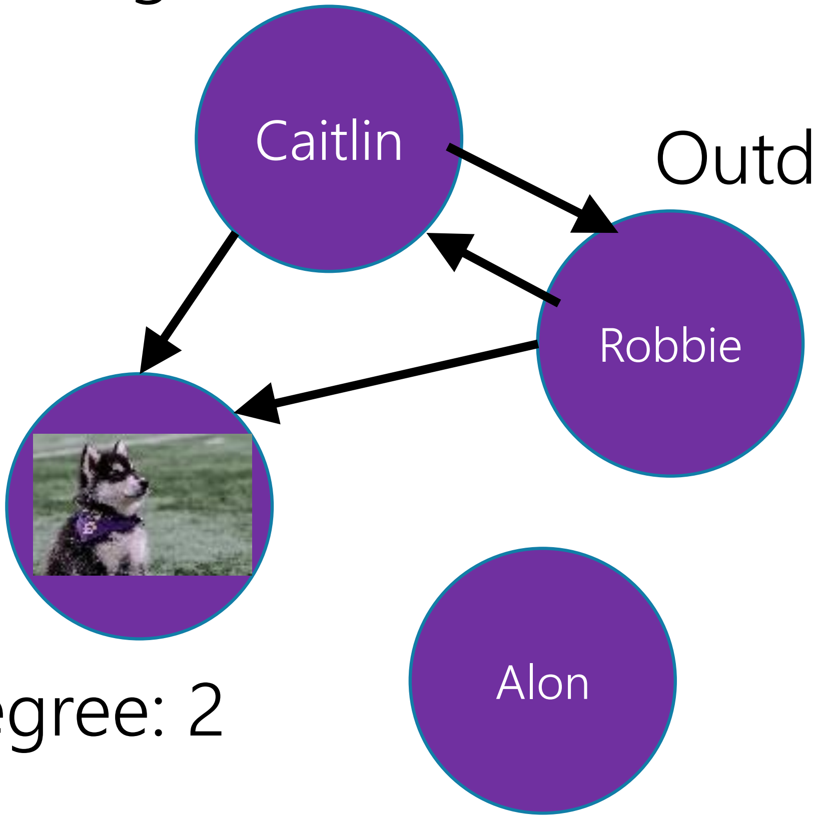
- Vertices: courses. Edge: from a to b if a is a prereq for b.

Edges have direction

Graph Terms

Graphs can be directed or undirected.

Following on twitter.

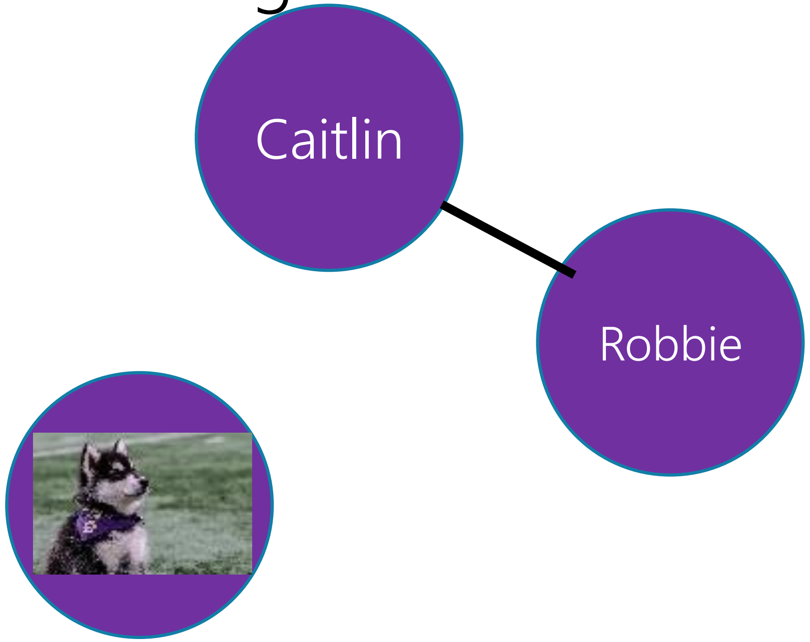


Outdegree: 2

Indegree: 2

This graph is **disconnected**.

Degree: 1

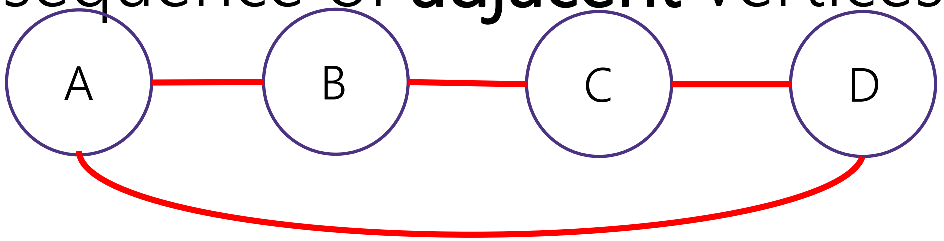


Degree: 0

Friendships on Facebook.

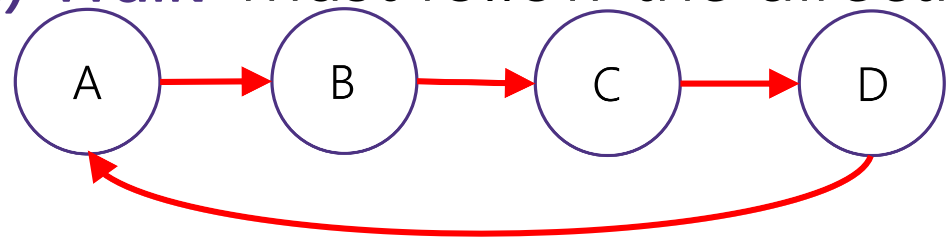
Graph Terms

Walk – A sequence of **adjacent** vertices. Each connected to next by an edge.



A,B,C,D is a walk.
So is A,B,A

(Directed) Walk–must follow the direction of the edges



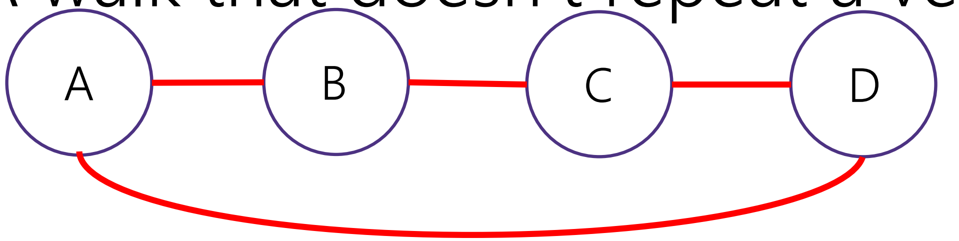
A,B,C,D,B is a directed walk.
A,B,A is not.

Length – The number of edges in a walk

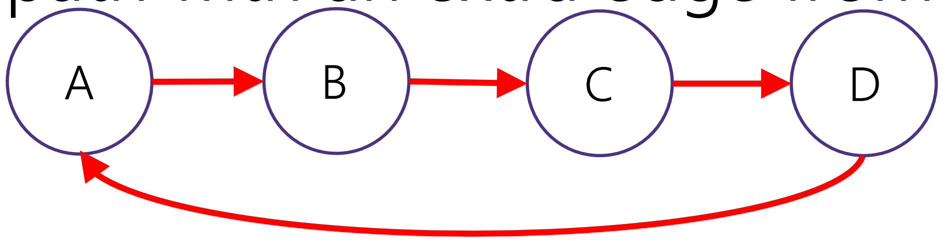
- (A,B,C,D) has length 3.

Graph Terms

Path – A walk that doesn't repeat a vertex. A,B,C,D is a path. A,B,A is not.



Cycle – path with an extra edge from last vertex back to first.



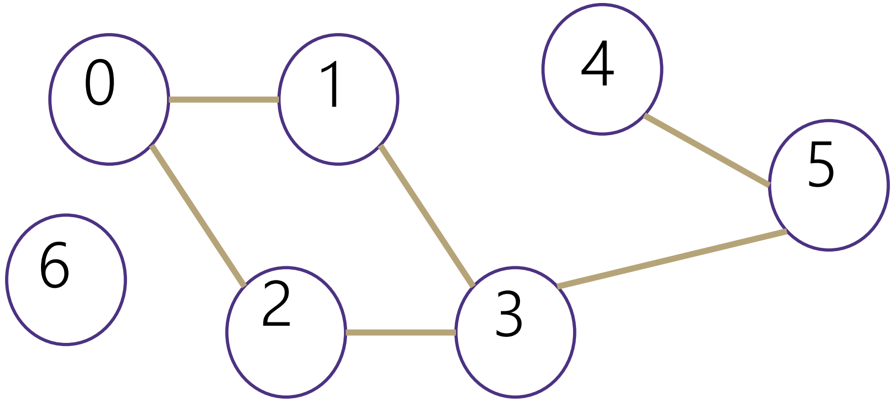
Be careful looking at other sources.

Some people call our "walks" "paths" and our "paths" "simple paths"

Use the definitions on these slides.

Representing and Using Graphs

Adjacency Matrix



	0	1	2	3	4	5	6
0	0	1	1	0	0	0	0
1	1	0	0	1	0	0	0
2	1	0	0	1	0	0	0
3	0	1	1	0	0	1	0
4	0	0	0	0	0	1	0
5	0	0	0	1	1	0	0
6	0	0	0	0	0	0	0

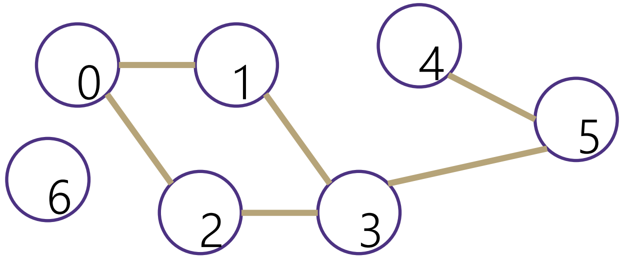
In an adjacency matrix a[u][v] is 1 if there is an edge (u,v), and 0 otherwise.

Time Complexity (|V| = n, |E| = m):

- Add Edge: O(1)
- Remove Edge: O(1)
- Check edge exists from (u,v): O(1)
- Get neighbors of u (out): O(n)
- Get neighbors of u (in): O(n)

Space Complexity: $O(n^2)$

Adjacency List



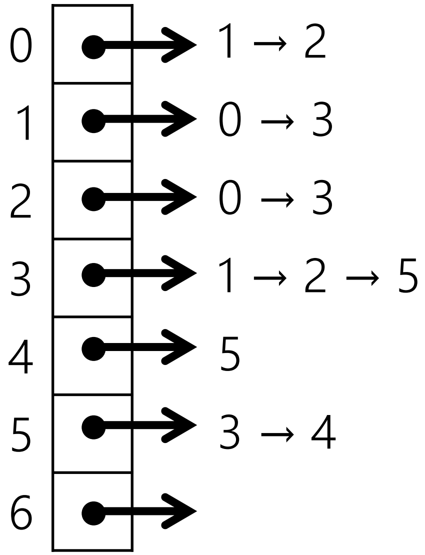
An array where the u'th element contains a list of neighbors of u.

Directed graphs: put the out neighbors (a[u] has v for all (u,v) in E)

Time Complexity (|V| = n, |E| = m):

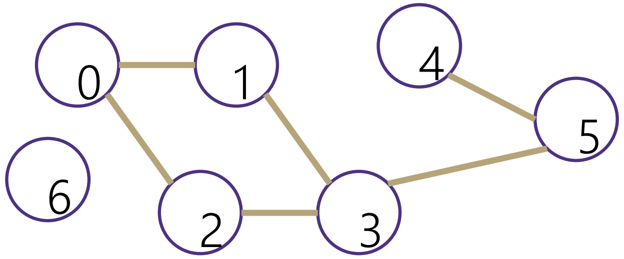
- Add Edge: $O(1)$
- Remove Edge: $O(\min(n, m))$
- Check edge exists from (u,v): $O(\min(n, m))$
- Get neighbors of u (out): $O(n)$
- Get neighbors of u (in): $O(n + m)$

Space Complexity: $O(n + m)$



Suppose we use a linked list for each node.

Adjacency List



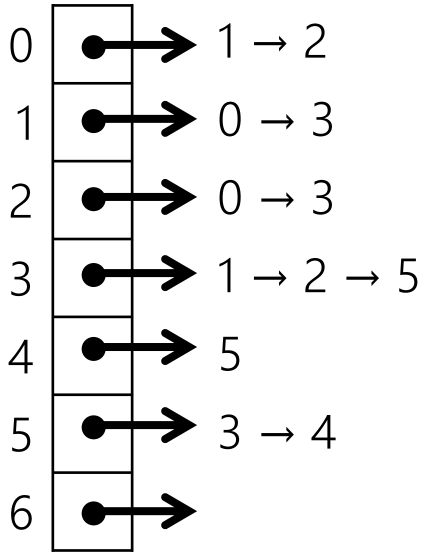
An array where the u'th element contains a list of neighbors of u.

Directed graphs: put the out neighbors (a[u] has v for all (u,v) in E)

Time Complexity (|V| = n, |E| = m):

- Add Edge: O(1)
- Remove Edge: O(1)
- Check edge exists from (u,v): O(1)
- Get neighbors of u (out): O(n)
- Get neighbors of u (in): O(n)

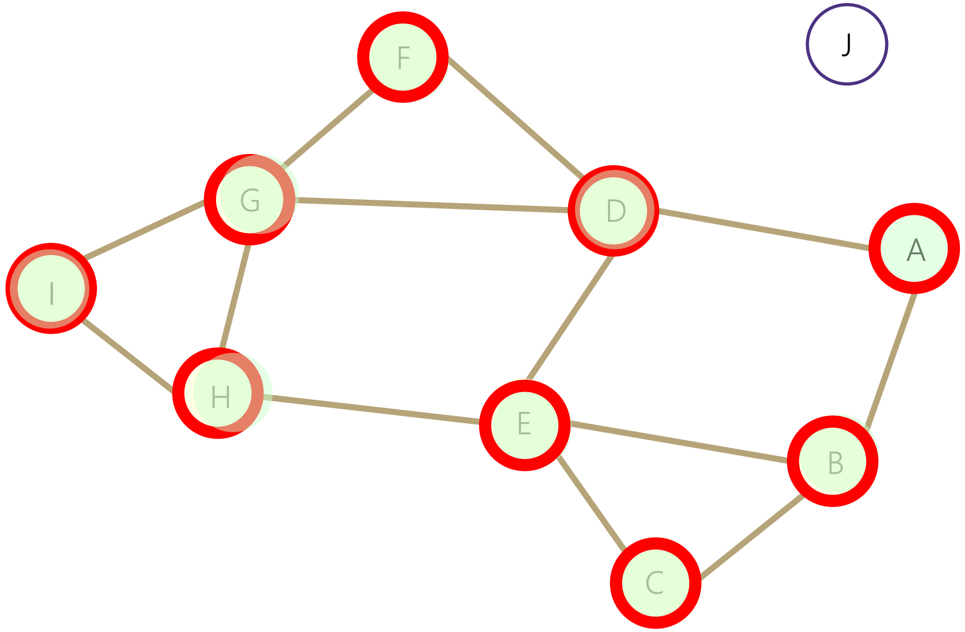
Space Complexity: O(n + m)



Switch the linked lists to hash tables, and do average case analysis.

Breadth First Search

```
search(graph)
    toVisit.enqueue(first vertex)
      mark first vertex as visited
    while(toVisit is not empty)
       current = toVisit.dequeue()
       for (V : current.neighbors())
          if (v is not visited)
             toVisit.enqueue(v)
                  mark v as visited
       finished.add(current)
```



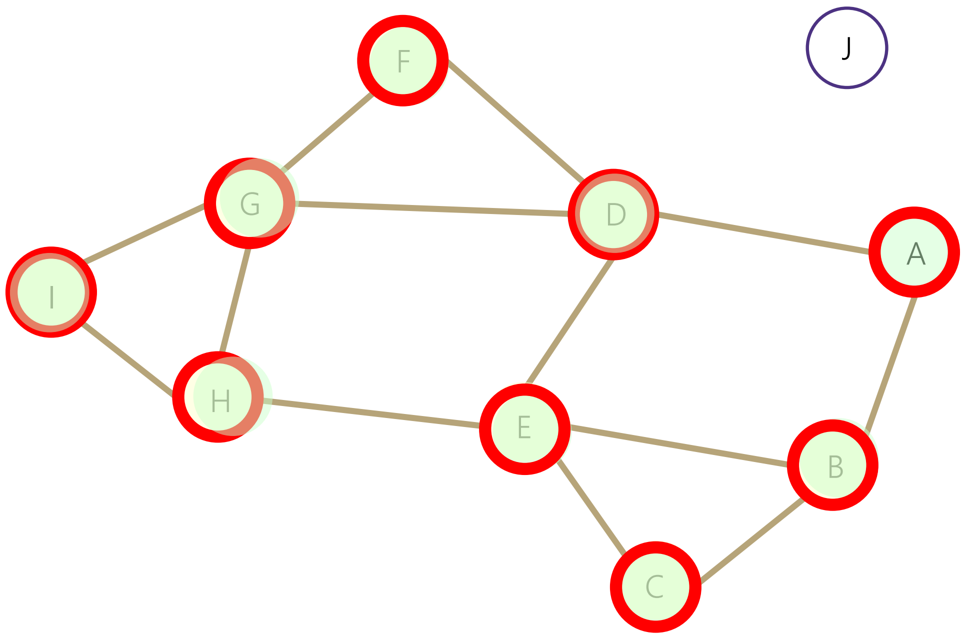
Current node: I

Queue: B D E C F G H I

Finished: A B D E C F G H I

Breadth First Search

```
search(graph)
   toVisit.enqueue(first vertex)
     mark first vertex as visited
   while(toVisit is not empty)
      current = toVisit.dequeue()
      for (V : current.neighbors())
         if (v is not visited)
            toVisit.enqueue(v)
                 mark v as visited
      finished.add(current)
```



What's the running time of this algorithm?

We visit each vertex at most twice, and each edge at most once: $O(|V| + |E|)$

Depth First Search (DFS)

BFS uses a queue to order which vertex we move to next

Gives us a growing "frontier" movement across graph

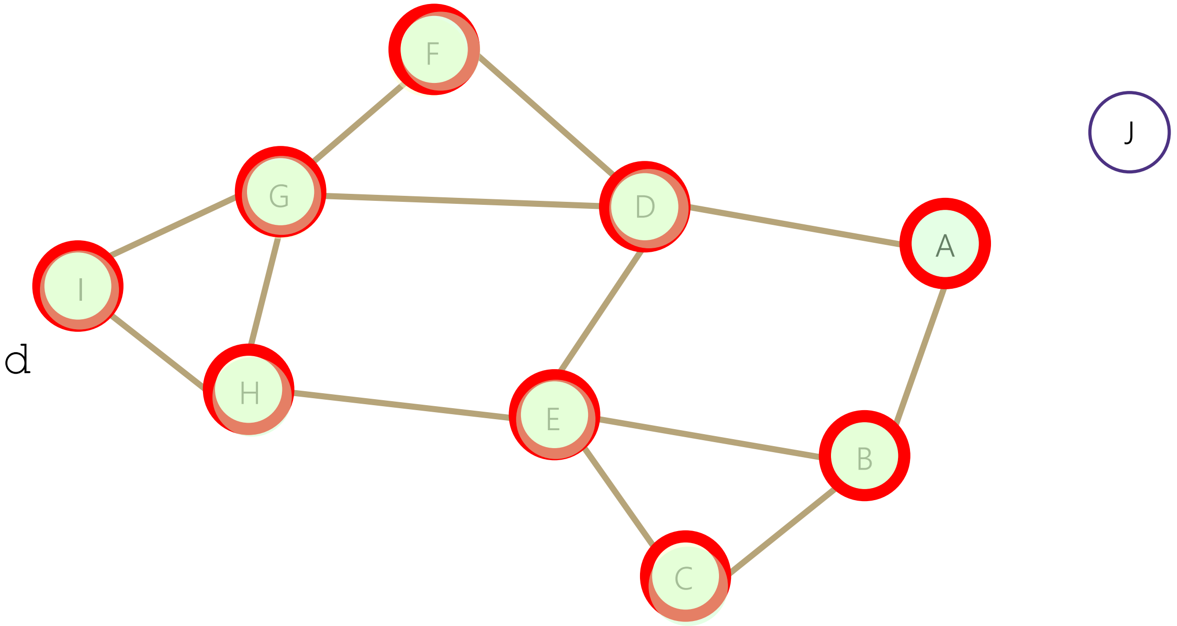
Can you move in a different pattern? What if you used a stack instead?

```
bfs(graph)
    toVisit.enqueue(first vertex)
       mark first vertex as visited
    while(toVisit is not empty)
        current = toVisit.dequeue()
        for (V : current.neighbors())
            if (v is not visited)
                toVisit.enqueue(v)
                        mark v as visited
        finished.add(current)
```

```
dfs(graph)
    toVisit.push(first vertex)
       mark first vertex as visited
    while(toVisit is not empty)
        current = toVisit.pop()
        for (V : current.neighbors())
            if (V is not visited)
                toVisit.push(v)
                        mark v as visited
        finished.add(current)
```

Depth First Search

```
dfs(graph)
    toVisit.push(first vertex)
        mark first vertex as visited
    while(toVisit is not empty)
        current = toVisit.pop()
        for (V : current.neighbors())
            if (V is not visited)
                toVisit.push(v)
                            mark v as visited
        finished.add(current)
```



Current node: D

Stack: D B E H G

Finished: A B E H G F I C D

DFS

Running time?

- Same as BFS: $O(|V| + |E|)$

You can rewrite DFS to be a recursive method.

Use the call stack as your stack.

No easy trick to do the same with BFS.

Next week: Using BFS, DFS and other algorithms to solve problems!