Review of Priority Queues and Graphs

Priority Queue Abstract Datatype

What does a priority queue look like?

- An ordered sequence of elements

$$(a_1, \ldots, a_n)$$

- A linear data structure
- a_1 is the first element in the queue

- a_n is the last element in the queue

Priorities

What role do priorities play?

- Ordering of elements is determined by the priority associated with each element

- a_1 has as high a priority as any element in $a_1, \ldots, a_n$

- Priorities do not need to be distinct

  — provide a partial ordering of elements in the queue

- Priorities can change as a computation proceeds

Issues

- How might priorities be determined?
- What operations are typically performed on priority queues?

- How can priority queues be efficiently implemented?

Priorities

- A queue of jobs to be processed by some resource

  — priority determined by importance/urgency of job

- A queue of items on an agenda

  — priority measures whether items are ready to be considered

Priority Queue Operations

- Find/remove element with highest priority on queue
- Add new element to queue
- Update priority of element on queue
- Determine if queue is empty (or more generally length of queue)
- Return an empty priority queue

Implementations of Priority Queues

We will consider two alternatives:

- Unsorted list implementation

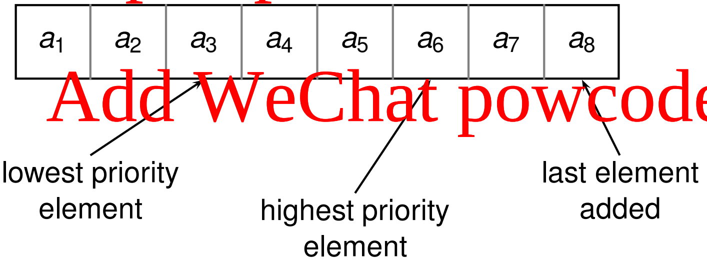
- Heap implementation

Unsorted List Implementation

- Elements arranged in arbitrary order

  — e.g. the order in which they were added

Example:



Efficiency of Unsorted List Implementation

Suppose we have a queue containing n elements: $(a_1, \ldots, a_n)$

- Adding new element:

  — add to end of list
  — takes $\Theta(1)$ time to execute

- Removing front of queue:

  — requires linear search of queue
  — takes $\Theta(n)$ time in worst-case

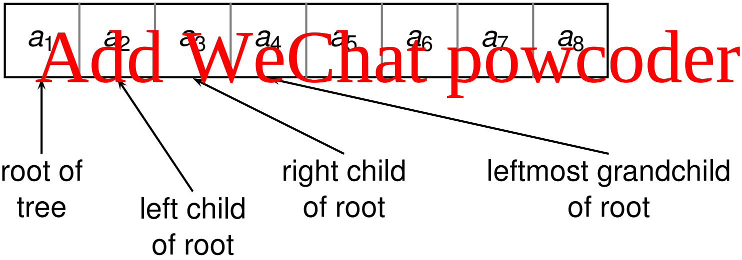
- Update priority of an element:

  — no reordering of elements required
  — takes $\Theta(1)$ assuming constant time access to priority values

Heap Implementation

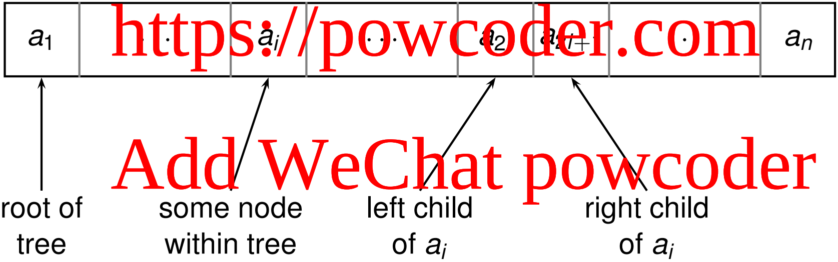
What is a heap?

- An array of values where ordering is constrained in a particular way
- A full binary tree

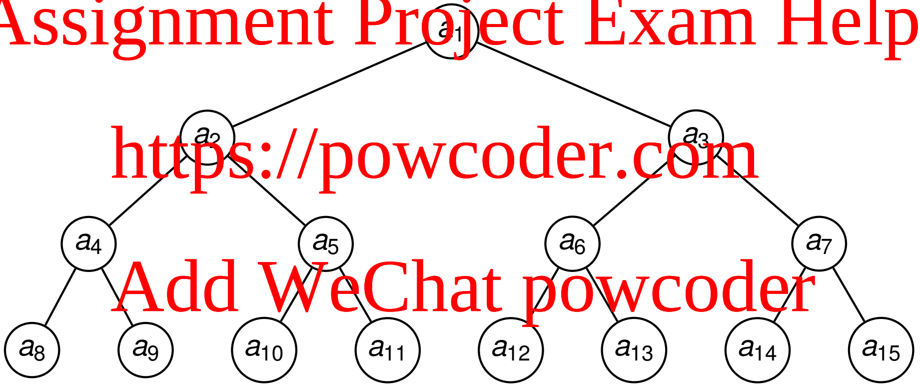


Heaps

In general:



Heap as Tree



Heap Ordering

The clever idea behind heaps:

- Exploits a "light-touch" ordering scheme

- Goal: just ordered enough to be useful

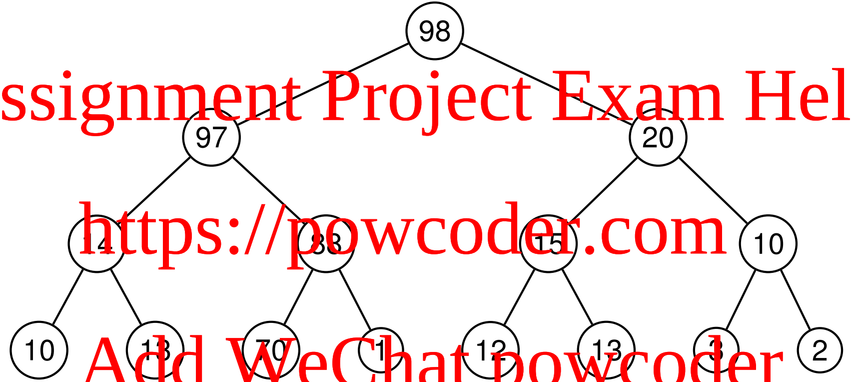
— the highest priority element can be found quickly
— avoid unnecessarily maintaining complete orderliness

A heap is well-formed when for each i the priority of a_i is as high as the priority of its children

For all i:

$$priority(a_i) \geq priority(a_{2i})$$
$$priority(a_i) \geq priority(a_{2i+1})$$

Example Heap



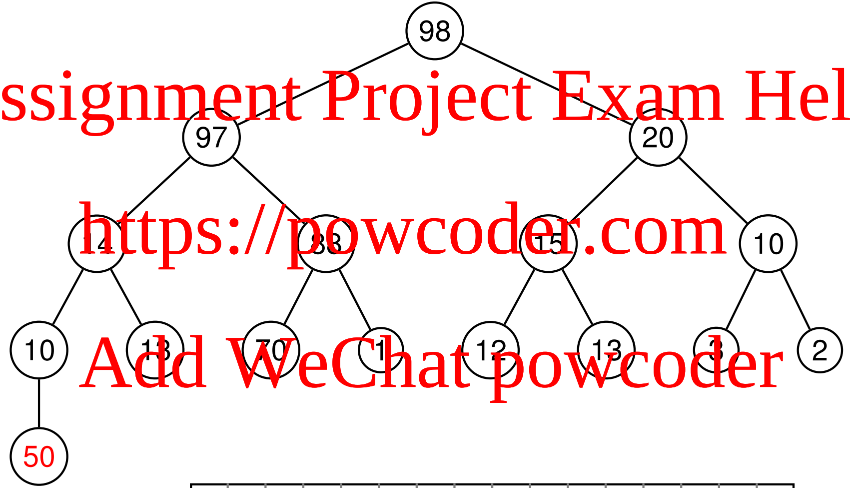
98	97	20	14	88	15	10	10	13	70	1	12	13	3	2
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Working with Heaps

How do we add an element to a heap?

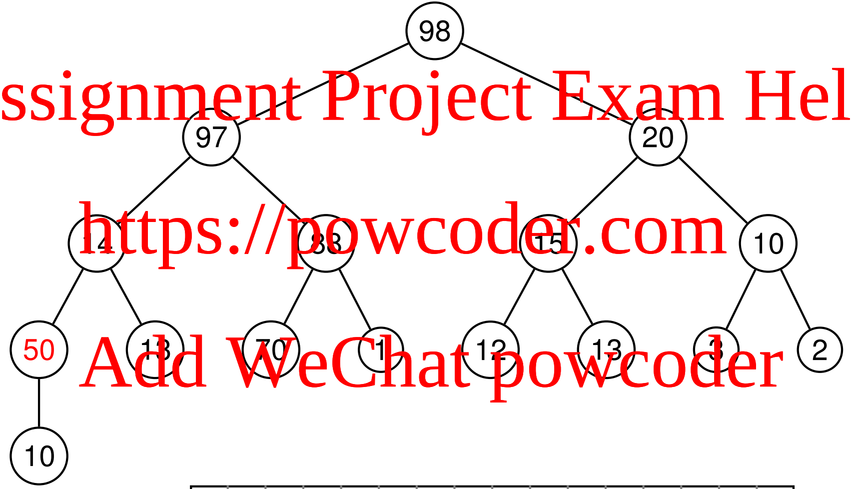
- Insert new element to end of sequence
- Repeatedly swap with parent if higher priority
- This is operation is called *heapify*

Adding a New Element



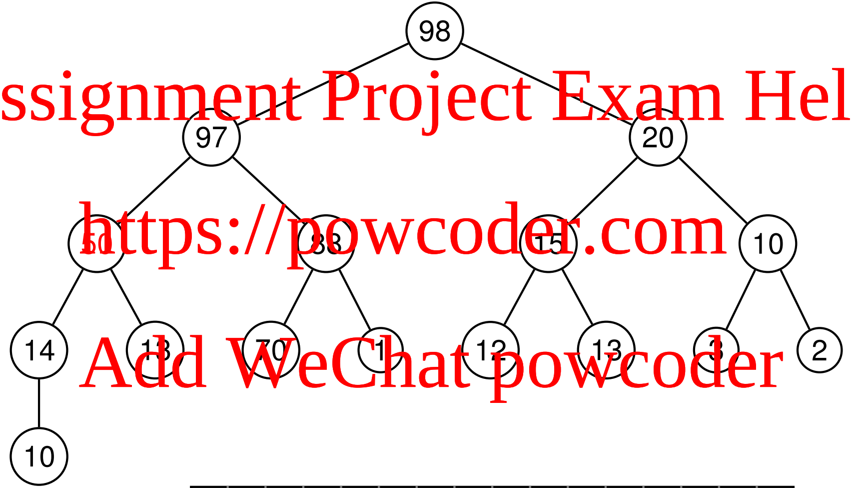
98	97	20	14	88	15	10	10	13	70	1	12	13	3	2	50
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Exchange with Parent



98	97	20	14	88	15	10	50	13	70	1	12	13	3	2	10
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Exchange with Parent



98	97	20	50	88	15	10	14	13	70	1	12	13	3	2	10
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Running Time

How long does it take to insert an element into a heap?

- The number of exchanges is bounded by the depth of the tree

- The tree is a balanced binary tree so has depth $\log n$
- The running time is $\Theta(\log n)$ in worst-case

- Best-case running time is $\Theta(1)$

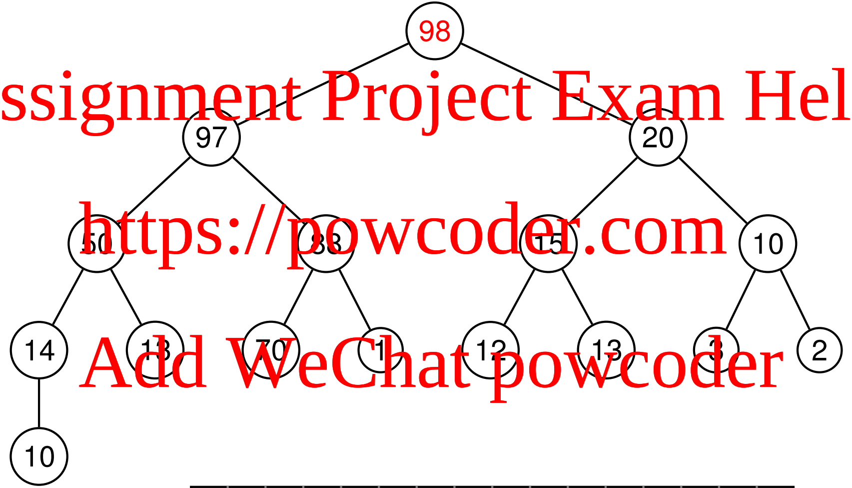
Removing Element from Heap

Not quite as easy as it sounds

- The highest priority element is at the front of the sequence

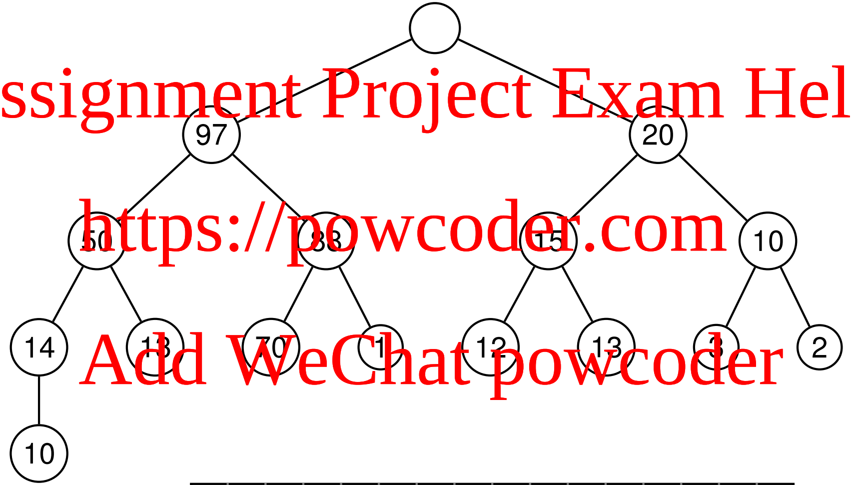
- Need to restore heap property
- Insert last element in sequence at front and put down tree as required

Illustrative Example



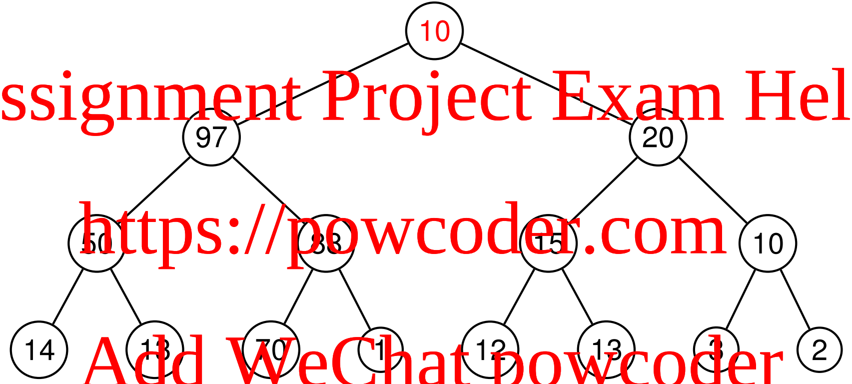
98	97	20	50	88	15	10	14	13	70	1	12	13	3	2	10
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Remove Root



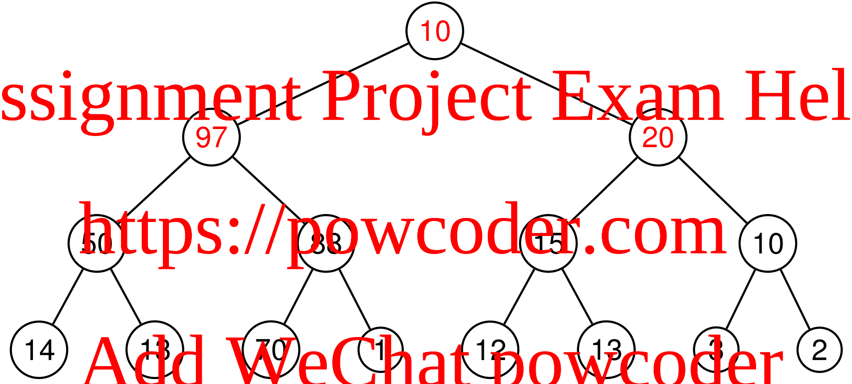
	97	20	50	88	15	10	14	13	70	1	12	13	3	2	10
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Insert Last Element at Root



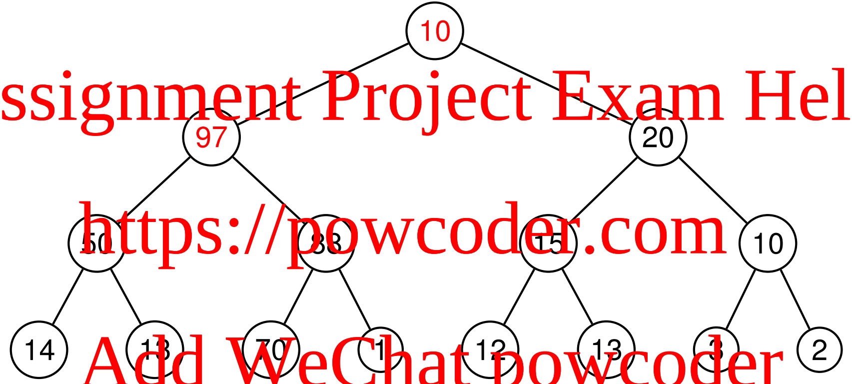
10	97	20	50	88	15	10	14	13	70	1	12	13	3	2	
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Compare Root with Children



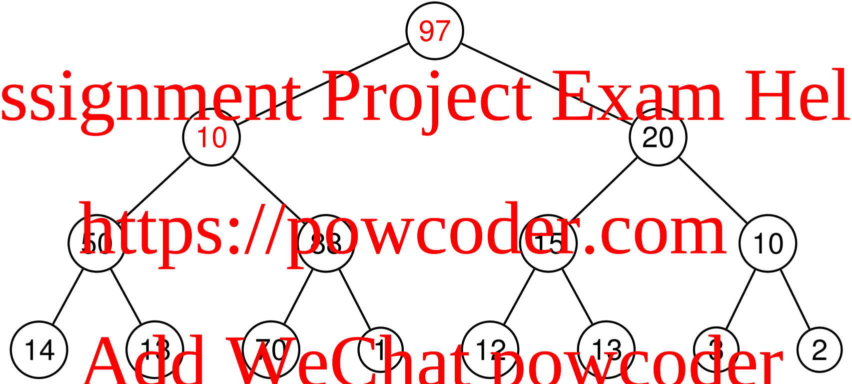
10	97	20	50	88	15	10	14	13	70	1	12	13	3	2	
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Left Child Should be Root



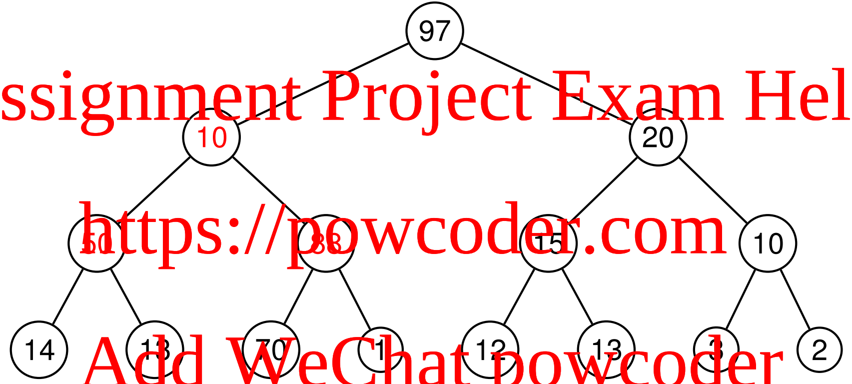
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
10	97	20	50	88	15	10	14	13	70	1	12	13	3	2	

Swop Root and Left Child



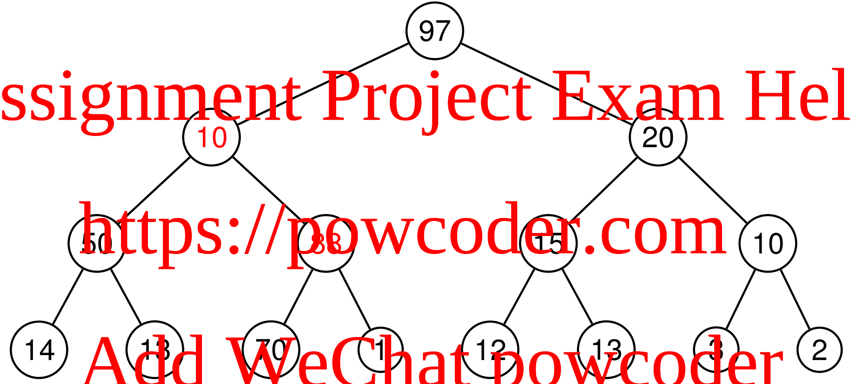
97	10	20	50	88	15	10	14	13	70	1	12	13	3	2	
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Compare with Children



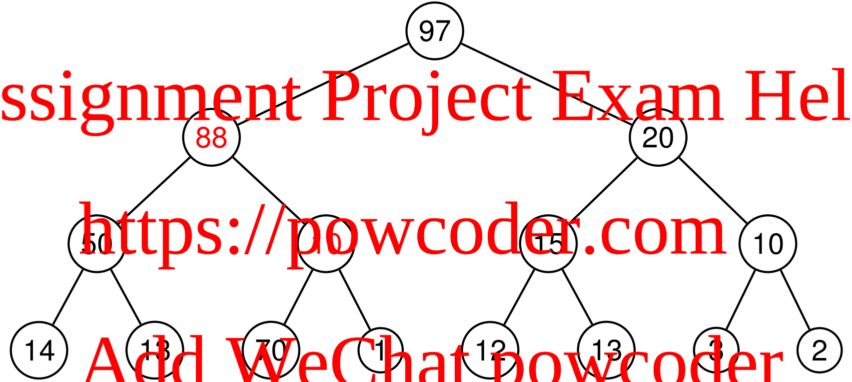
97	10	20	50	88	15	10	14	13	70	1	12	13	3	2	
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Right Child Should Move Up



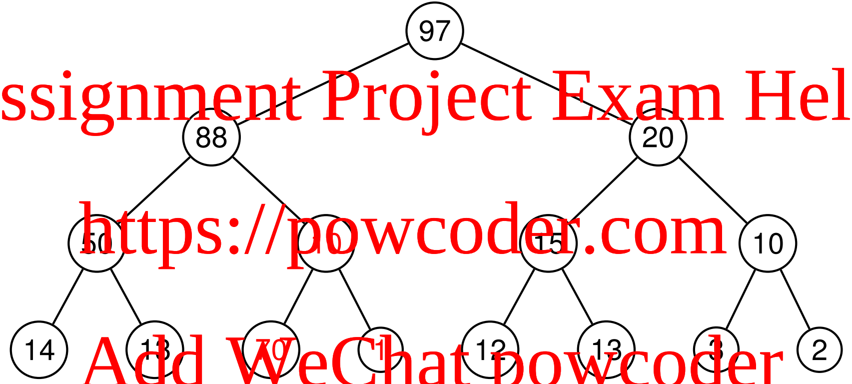
97	10	20	50	88	15	10	14	13	70	1	12	13	3	2	
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Swop Nodes



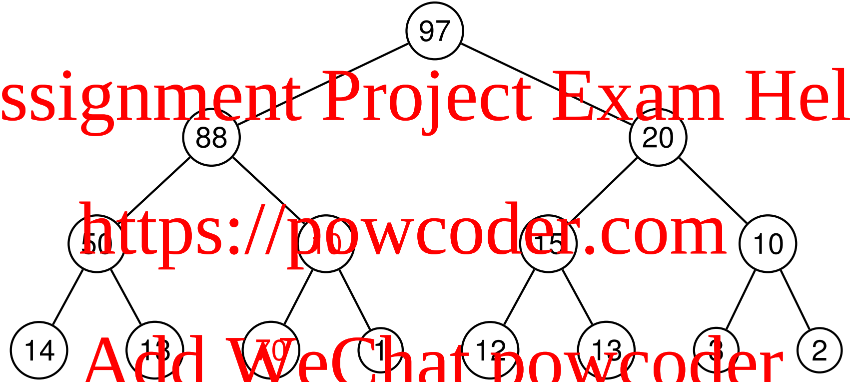
97	88	20	50	10	15	10	14	13	70	1	12	13	3	2	
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Compare with Children



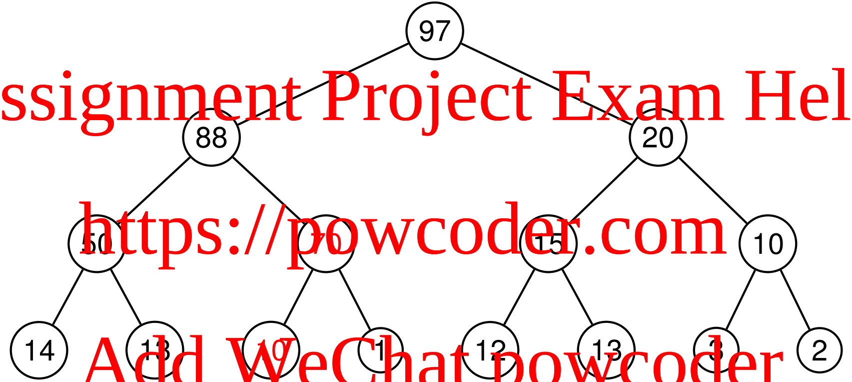
97	88	20	50	10	15	10	14	13	70	1	12	13	3	2	
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Left Child Should Move Up



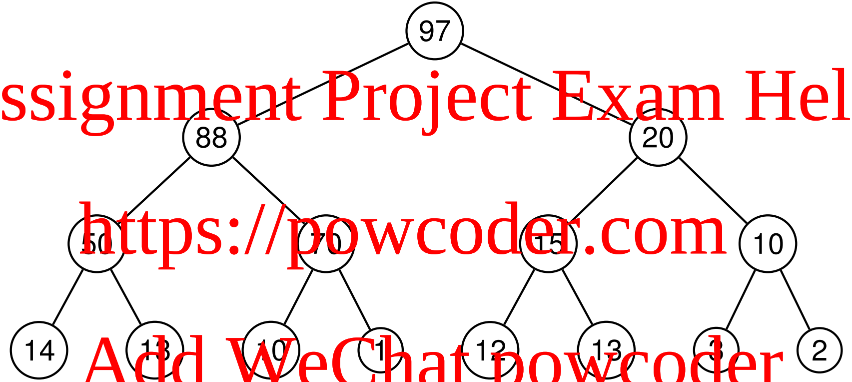
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
97	88	20	50	10	15	10	14	13	70	1	12	13	3	2	

Swop Nodes



97	88	20	50	70	15	10	14	13	10	1	12	13	3	2	
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Restored the Heap



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
97	88	20	50	70	15	10	14	13	10	1	12	13	3	2	

A Heap for you to restore

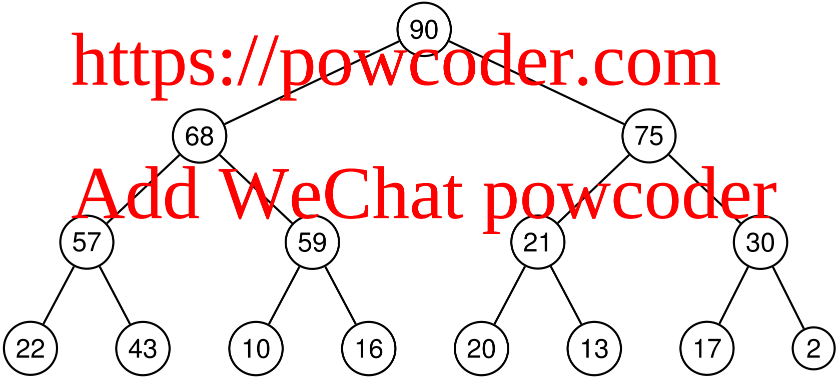
	68	90	57	59	75	30	22	43	10	16	20	13	17	2	21
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Show the heap as a tree

A Heap for you to restore

	68	90	57	59	75	30	22	43	10	16	20	13	17	2	21
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Show the heap as a tree



Running Time

How Long Does This Take?

- Worst-case bounded by depth of tree, so $\Theta(\log n)$
- Best-case $\Theta(1)$

Building a Heap from Scratch

Two approaches:

- Straightforward approach:

  — repeatedly insert new elements
  — each insertion takes $O(\log n)$ time
  — total running time is $O(n \log n)$

- More efficient alternative:

  — create heap bottom-up

Bottom-up Heap Construction

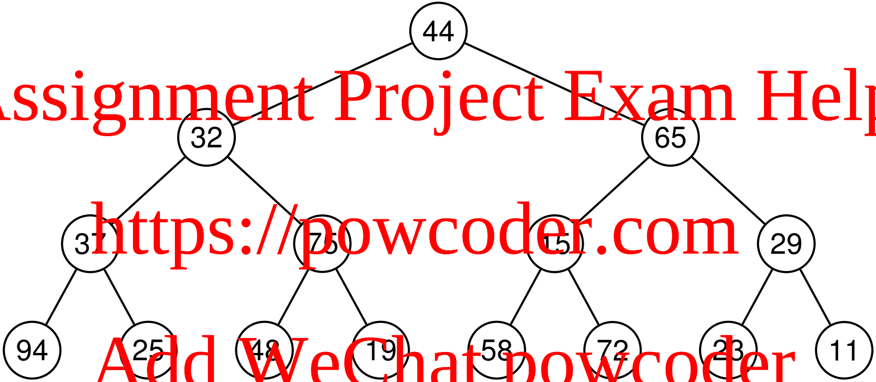
- Consider nodes in order of increasing height in tree

- Height of a node is length of longest path to leaf node

- Restore heap for subtree rooted at node being considered

  — through required exchanges with highest priority child

Heapification Order



	44	32	65	37	75	15	29	94	25	48	19	58	72	23	11
height	3	2	2	1	1	1	1	0	0	0	0	0	0	0	0

Analysis of Running Time

- For each node, processing time for node of height

$$O(h)$$

- Number of nodes of height h

$$\leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$$

Total Running Time

$$\sum_{h=0}^{\lceil \log n \rceil} \frac{n}{2^{h+1}} O(h) = \frac{1}{2} n \sum_{h=0}^{\lceil \log n \rceil} \frac{1}{2^h} O(h)$$

$$= O\left(n \sum_{h=0}^{\lceil \log n \rceil} \frac{h}{2^h} \right)$$

$$= O(n)$$

note that $\sum_{h=0}^{\infty} \frac{h}{2^h}$ converges to 2

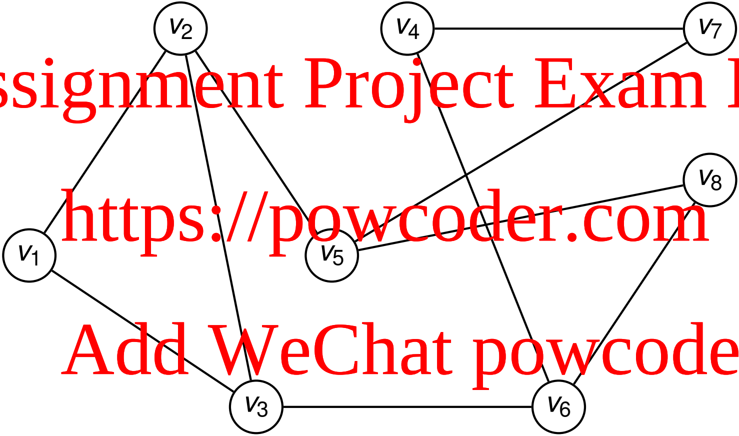
Review of Graphs

Graphs

A general purpose data structure

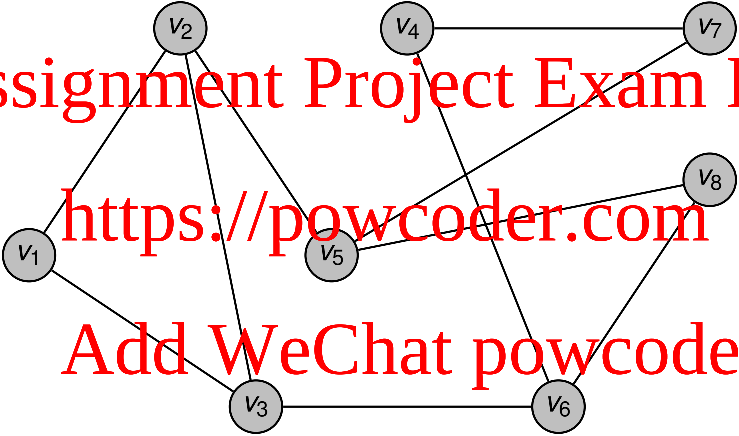
- Need not be linear like a list, but could be
- Need not be hierarchical like a tree, but could be
- Can express arbitrary binary relationship between a collection of elements
- Strength of relationship can be encoded using weights

Graphs: Examining an Example



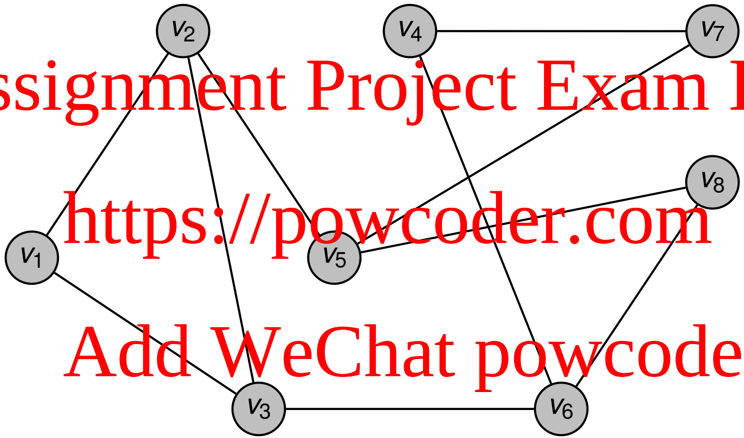
Let's look at an example graph

Graphs: Examining an Example



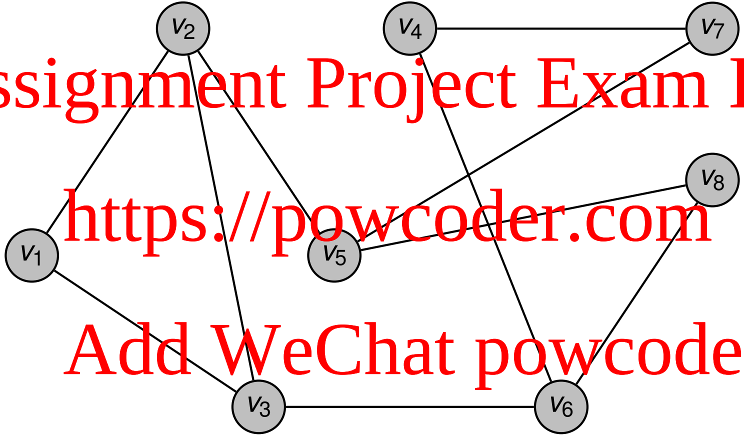
There are 8 vertices or nodes

Graphs: Examining an Example



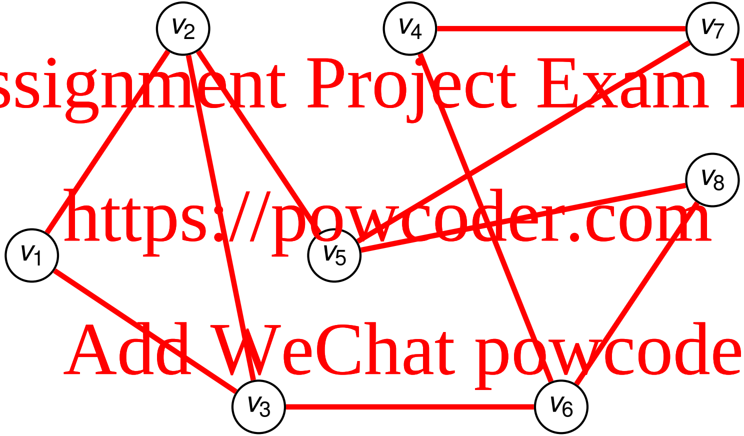
Use n to refer to the number of nodes — so $n = 8$

Graphs: Examining an Example



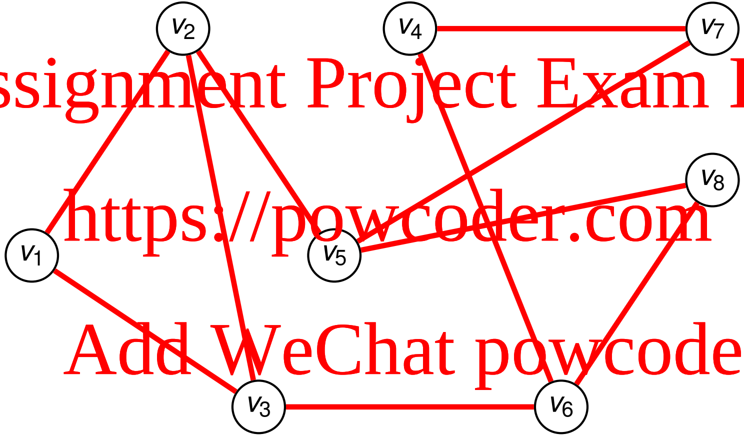
Use V to refer to the set of all nodes — so $V = \{v_1, \ldots, v_8\}$

Graphs: Examining an Example



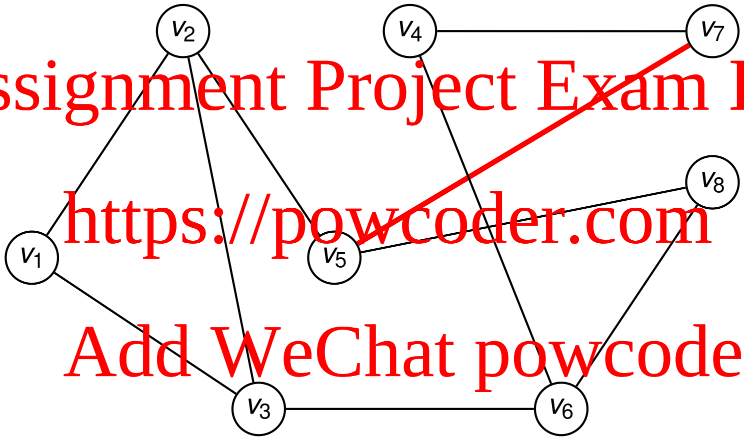
There are 10 edges

Graphs: Examining an Example



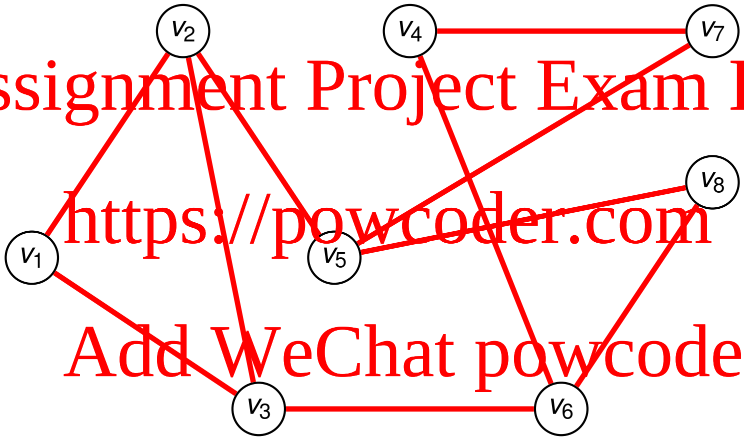
Each edge denoted by a set of two vertices

Graphs: Examining an Example



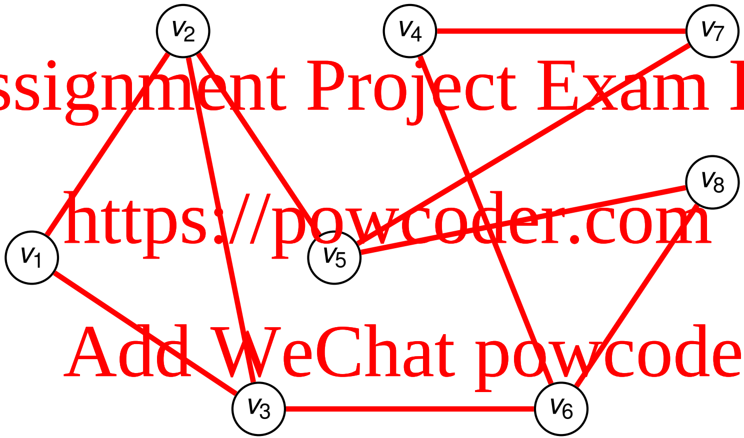
For example: $\{v_5, v_7\}$

Graphs: Examining an Example



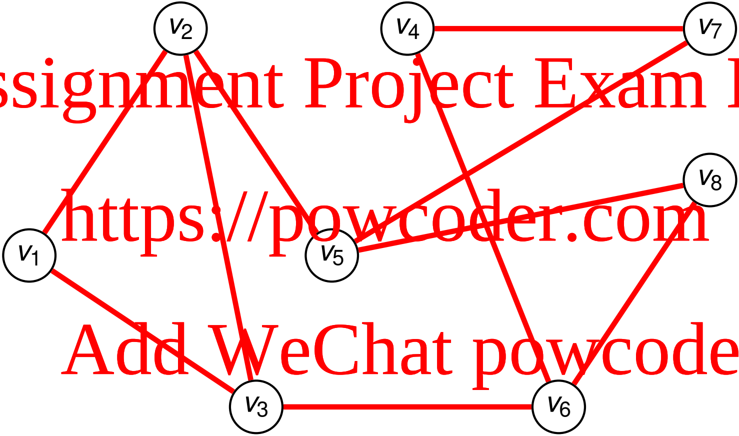
Use m to refer to the number of edges — so $m = 10$

Graphs: Examining an Example



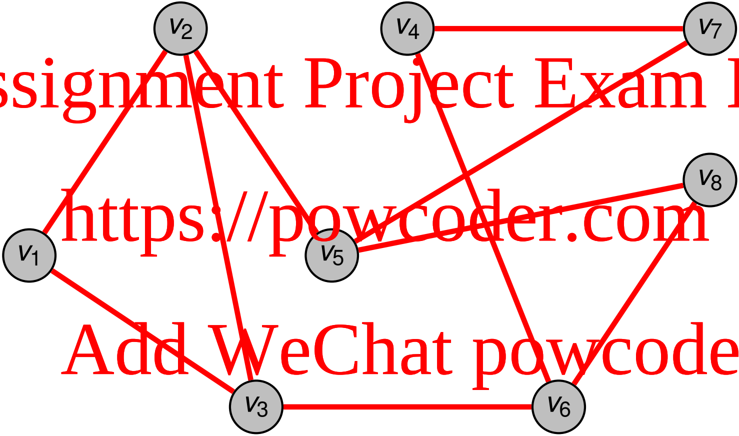
Use E to refer to the set of all edges

Graphs: Examining an Example



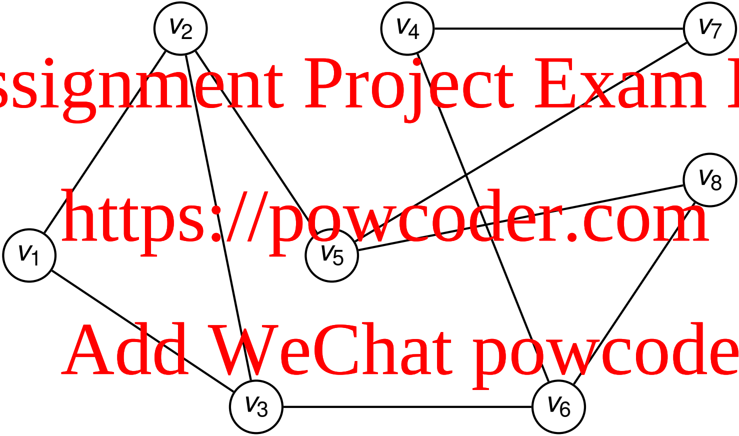
$$E = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_2, v_3\}, \{v_2, v_5\}, \{v_3, v_6\}, \{v_4, v_6\}, \{v_4, v_7\}, \{v_5, v_7\}, \{v_5, v_8\}, \{v_6, v_8\}\}$$

Graphs: Examining an Example



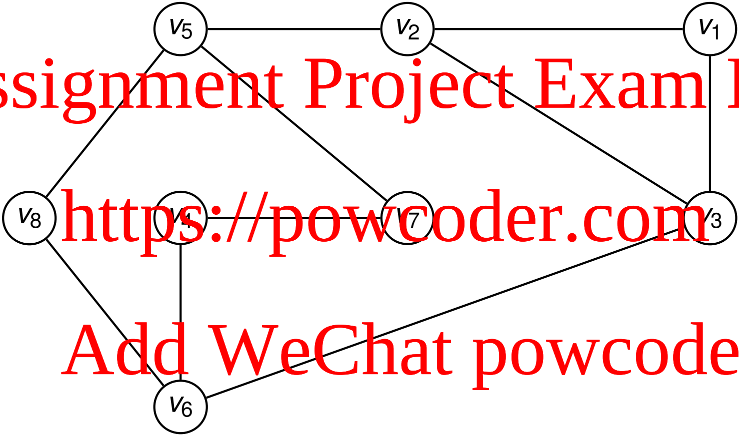
Use G to refer to the graph — so $G = (V, E)$

Graphs: Examining an Example



The layout of the graph is unimportant

Graphs: Examining an Example



This is the same graph

What Are Graphs Good For?

Expressing all sorts of binary relationships

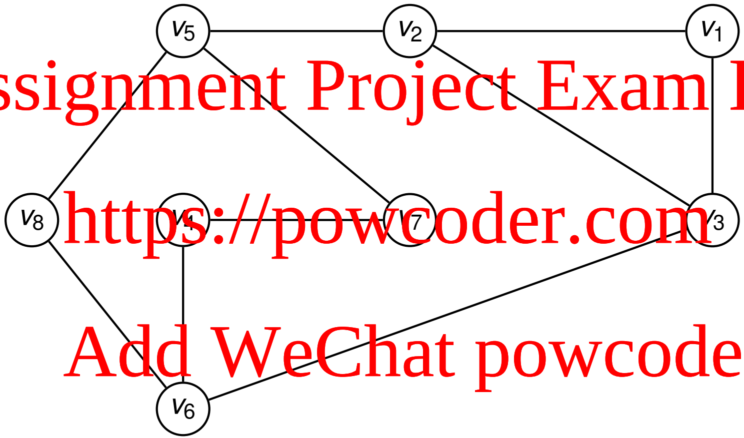
- Transport networks
  - node: a place
  - edge: route connecting two places
- Communication networks
  - node: computer cluster
  - edge: communication link between two clusters

What Are Graphs Good For?

- Information networks
  - node: web page
  - edge: hyperlink from one page to another
- Social networks
  - node: person
  - edge: some sort of relationship between two people
- Dependency networks
  - node: task to be performed
  - edge: dependency between two tasks

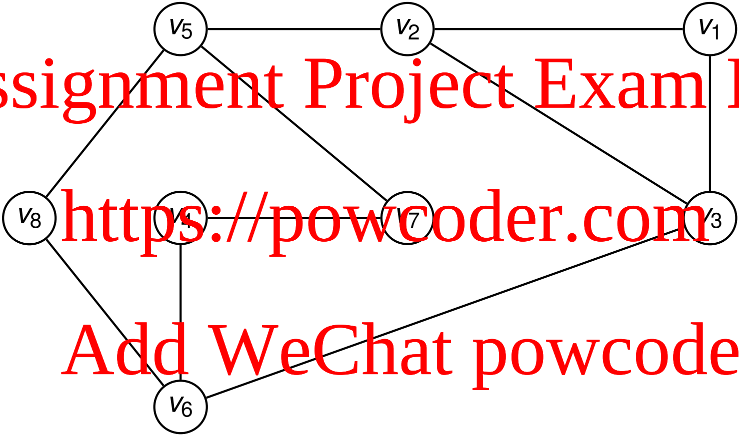
And much much more

Directed Graphs



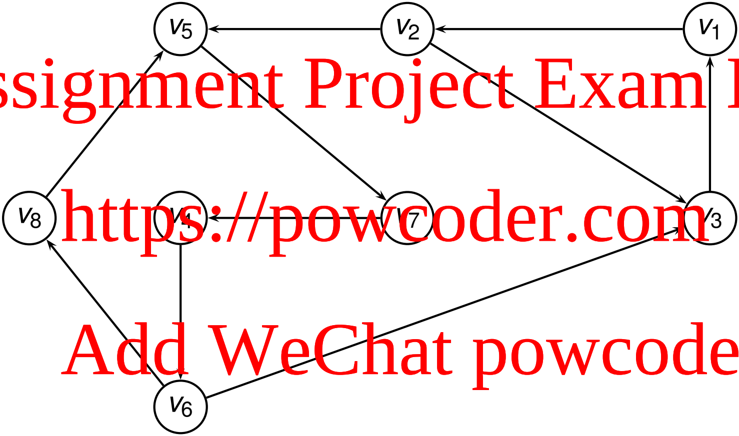
Sometimes relationships are directional

Directed Graphs



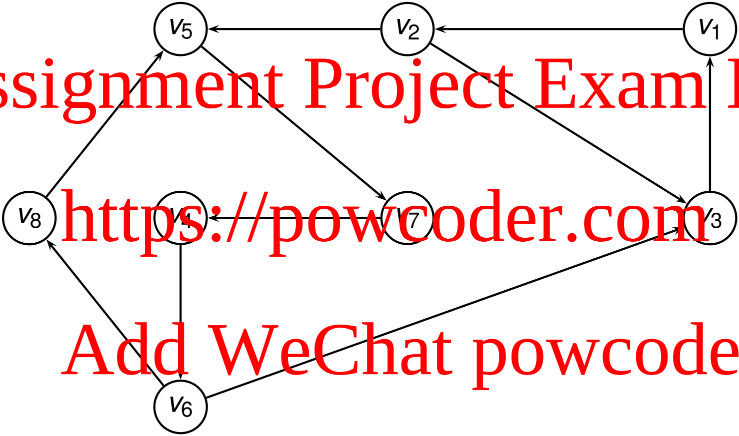
Capture this with directional edges

Directed Graphs



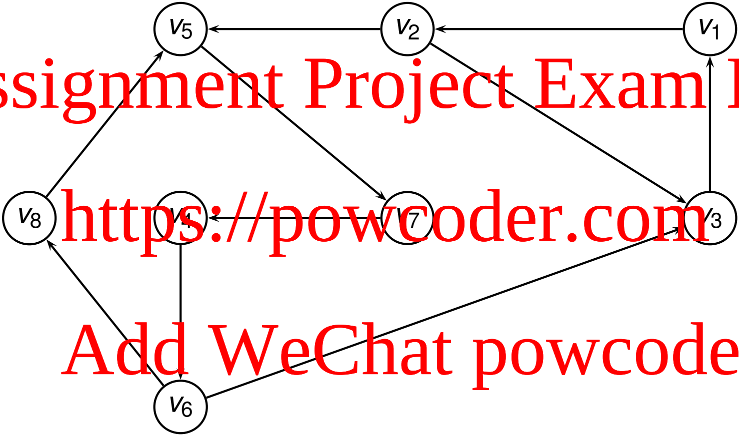
This is called a *directed* graph

Directed Graphs



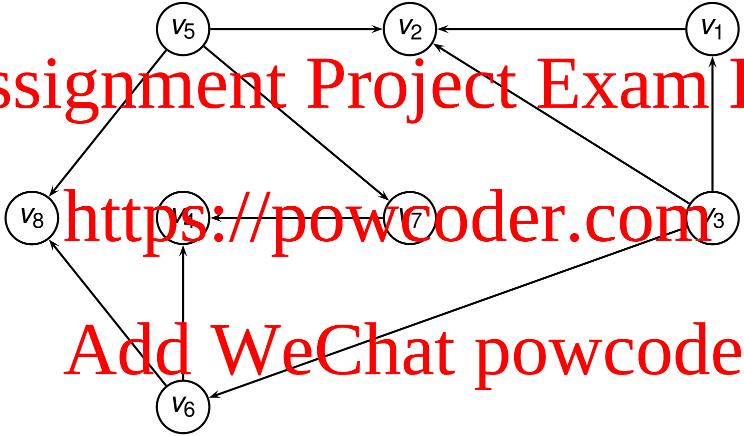
E is now a set of directed *pairs*

Directed Graphs



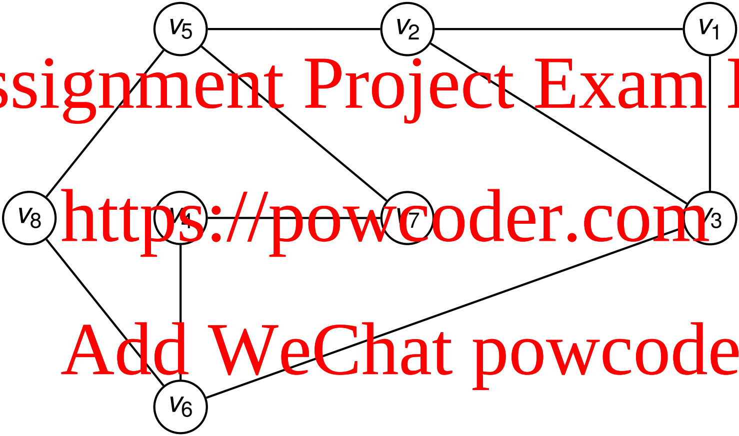
$$E = \{(v_1, v_2), (v_3, v_1), (v_2, v_3), (v_2, v_5), (v_6, v_3), (v_4, v_6), (v_7, v_4), (v_5, v_7), (v_8, v_5), (v_6, v_8)\}$$

Directed Acyclic Graphs



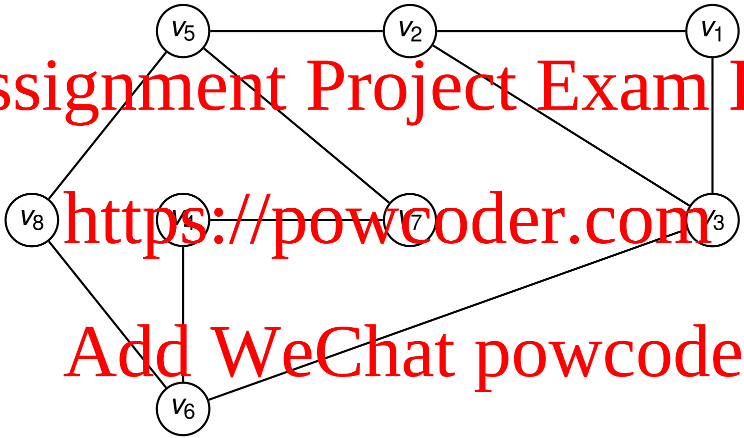
Important special case: directed *acyclic* graphs or DAGs

Weighted Graphs



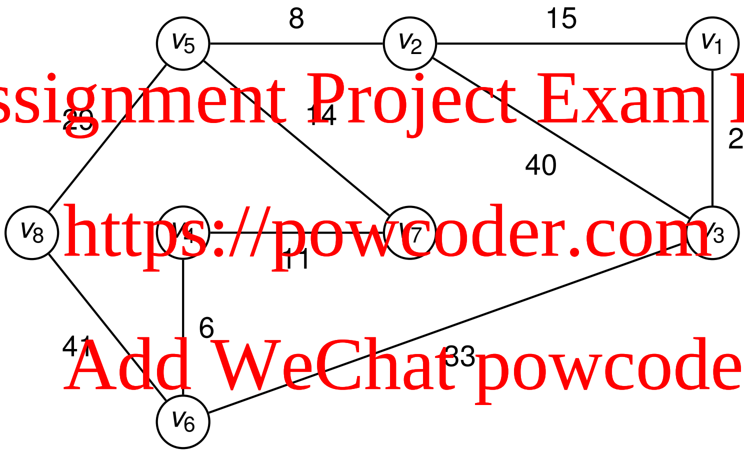
Sometimes want to associate weights with edges - e.g. distance

Weighted Graphs



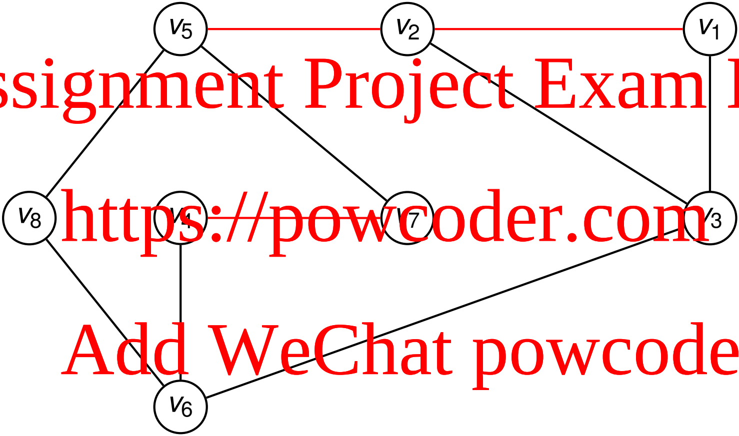
Capture this with weighted edges

Weighted Graphs



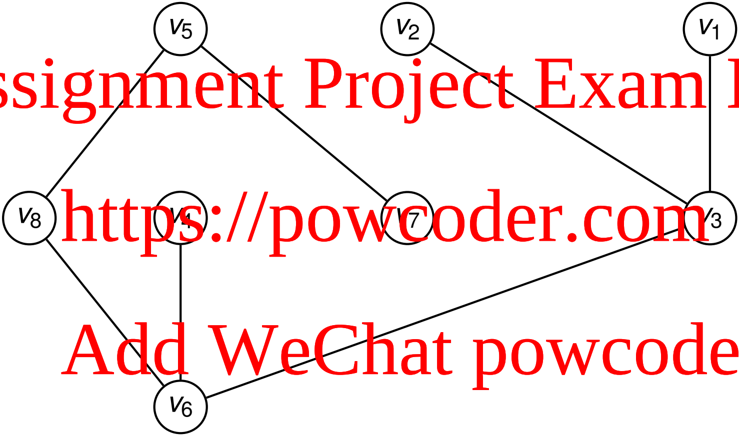
This is called a *weighted* graph

Trees: A Special Kind of Graph



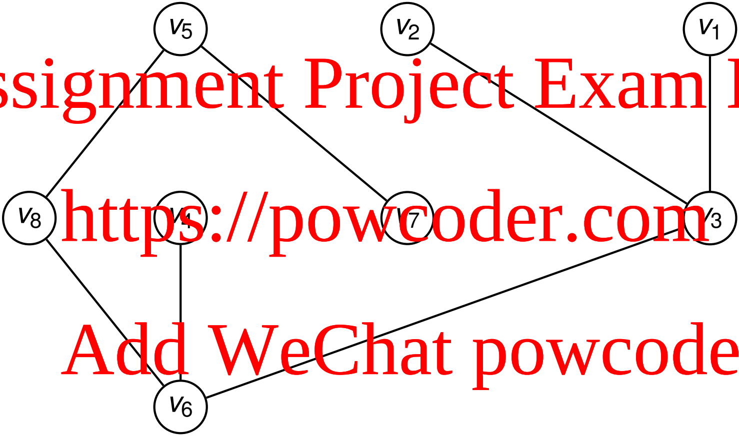
Let's remove these edges to give a tree

Trees: A Special Kind of Graph



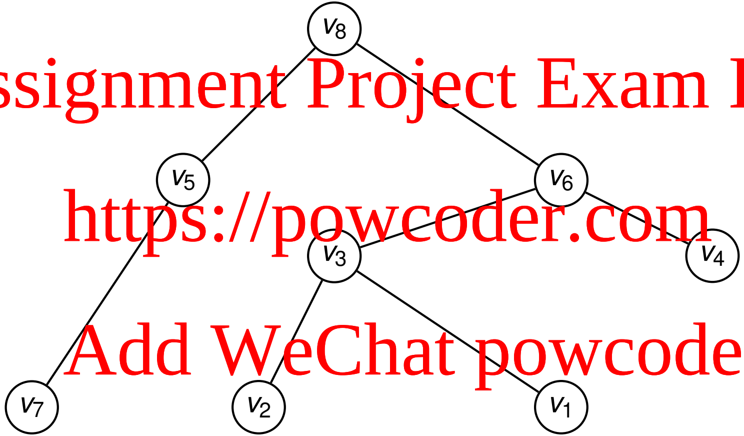
How can we tell that its a tree?

Trees: A Special Kind of Graph



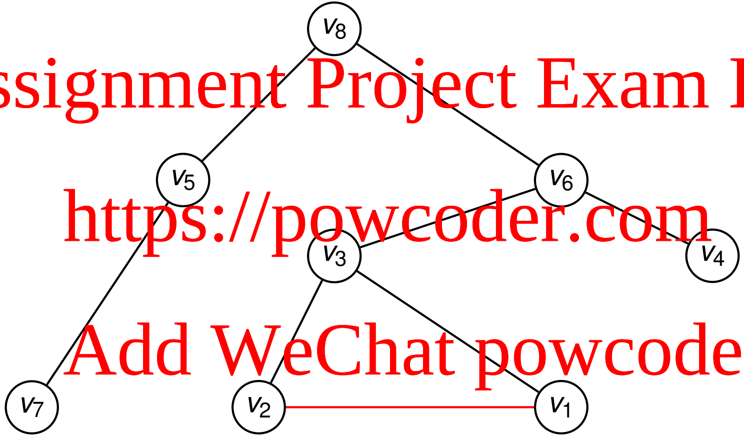
Let's pick it up by v_8

Trees: A Special Kind of Graph



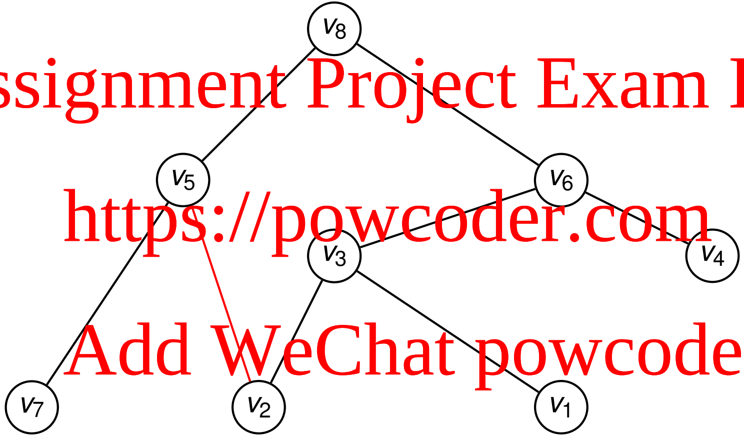
What happens if we put back the edges we removed

Trees: A Special Kind of Graph



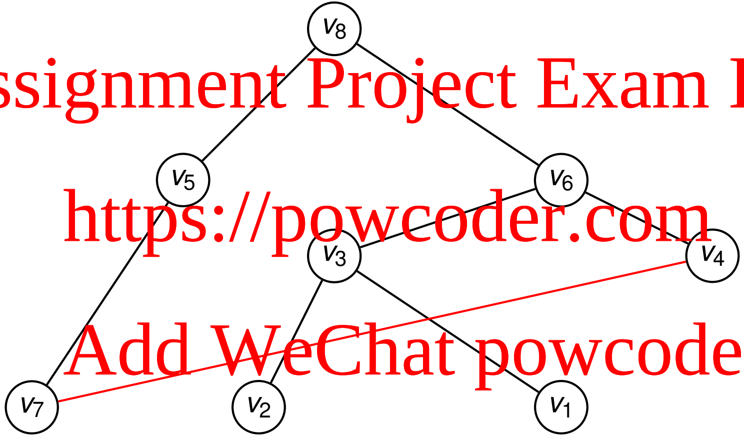
Adding that one creates a cycle

Trees: A Special Kind of Graph



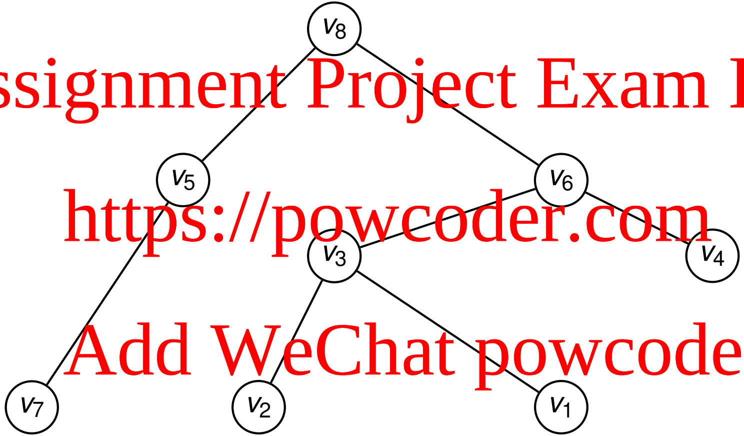
Adding that one also creates a cycle

Trees: A Special Kind of Graph



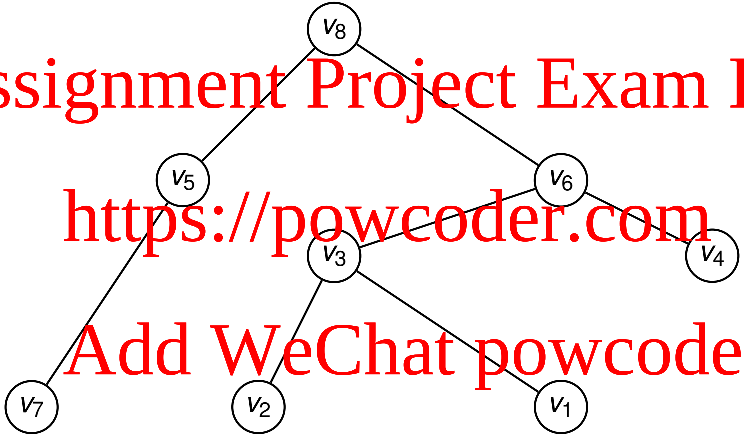
Guess what!

Trees: A Special Kind of Graph



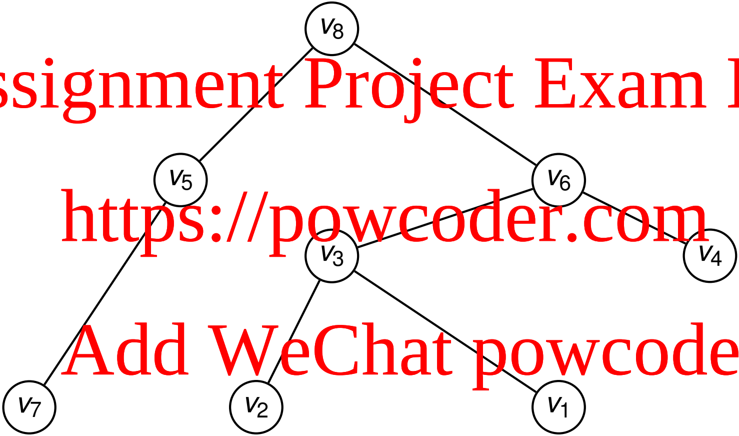
Fact 1: there are no cycles

Trees: A Special Kind of Graph



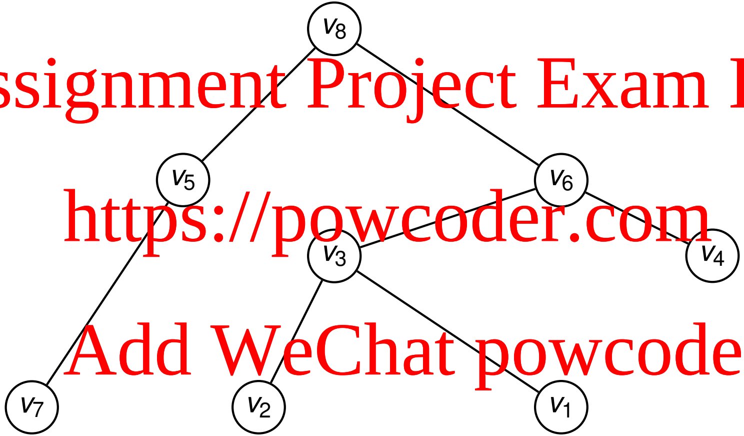
Fact 2: its connected, i.e. path between any pair of nodes

Trees: A Special Kind of Graph



Fact 3: n nodes and $n - 1$ edges

Trees: A Special Kind of Graph



. . . and that's what makes it a tree

Graphs: Implementation

Question: What data structure can we use to store a graph?

Answer: There are two alternatives:

- Adjacency matrix
- Adjacency list

Adjacency Matrix Representation

Adjacency matrix for undirected graph shown earlier

	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
v_1	0	1	1	0	0	0	0	0
v_2	1	0	1	0	1	0	0	0
v_3	1	1	0	0	0	1	0	0
v_4	0	0	0	0	0	1	1	0
v_5	0	1	0	0	0	0	1	1
v_6	0	0	1	1	0	0	0	1
v_7	0	0	0	1	1	0	0	0
v_8	0	0	0	0	1	1	0	0

Entry value in (i, j) same as value in (j, i) for undirected graph

Adjacency Matrix Representation

Adjacency matrix for directed graph shown earlier

	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
v_1	0	1	0	0	0	0	0	0
v_2	0	0	1	0	1	0	0	0
v_3	1	0	0	0	0	0	0	0
v_4	0	0	0	0	0	0	0	0
v_5	0	0	0	0	0	0	1	0
v_6	0	0	1	1	0	0	0	1
v_7	0	0	0	1	0	0	0	0
v_8	0	0	0	0	1	0	0	0

Adjacency Matrix Representation (cont.)

Two issues worth considering:

- Space efficiency of adjacency matrix representation
- Running time of basic operations of adjacency matrix representation

Space Efficiency of Adjacency Matrix Representation

Given a graph $G = (V, E)$ where $|V| = n$ and $|E| = m$

Question: How much space used to store G?

Answer: $\Theta(n^2)$

Doesn't matter how many edges there are
i.e. its not a function of m

Inefficient for graphs without many edges

Time Efficienty of Matrix Representation

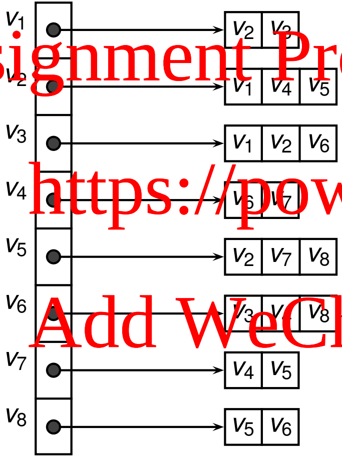
First, the good news:

- It takes $\Theta(1)$ to determine if there's an edge between two nodes

The not so good news:

- It takes $\Theta(n)$ to find all nodes adjacent to some node
- Even if there aren't any adjacent nodes it still takes $\Theta(n)$ to discover this
- Common for an algorithm to need to enumerate adjacent nodes
- Desirable that finding next node in enumeration take constant time

Adjacency List Representation



Adjacency List Representation Features

Same two issues need to be considered:

- Space efficiency of adjacency list representation
- Running time of basic operations of adjacency list representation

Space Efficiency of Adjacency List Representation

Given a graph $G = (V, E)$ where $|V| = n$ and $|E| = m$

Question: How much space used to store G?

Answer: $\Theta(m)$

More efficient than adjacency matrix for graphs without many edges

Time Efficiency of Adjacency List Representation

First the good news:

- Enumeration of adjacent nodes $\Theta(1)$ per adjacent node

Not so good news:

- Takes $O(n)$ to establish if a particular edge is in graph
- Linear search of adjacency list
- Adjacency list length is $O(n)$

For most of the algorithms we consider here, the adjacency list implementation is preferable.

Vertices and Edges

Let $G = (V, E)$ be a graph where $|V| = n$ and $|E| = m$

Question: What can we say about $|E|$ in terms of $|V|$?

How few edges could there be?

- Possible that E could be empty, i.e. $m = 0$
- If G is connected then $m \geq n - 1$

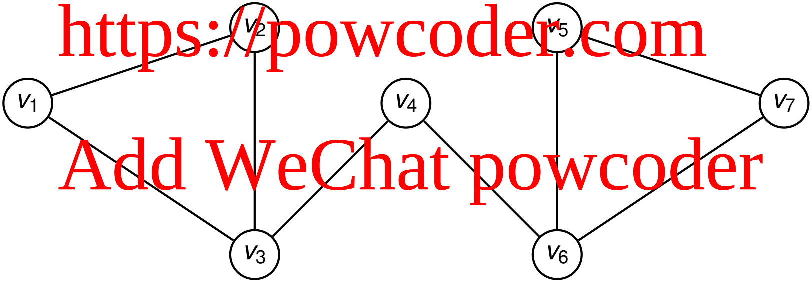
How many edges could there be?

- If G is a **complete** graph then $m = n(n-1)/2$ which is $\Theta(n^2)$

So, in general, m is $O(n^2)$

Questions for you

- Give adjacency list and adjacency matrix encodings of this graph
- Which uses the least space?
- If this was a complete graph, how many edges would it have?



Questions for you

- Give adjacency list and adjacency matrix encodings of this graph
- Which uses the least space?
  Adjacency list uses less memory
- If this was a complete graph, how many edges would it have?
  21