Heap Sort & Graphs

15-121 Fall 2020

Margaret Reid-Miller

Today

- Heap implementation
- Heap Sort
- Introduction to Graphs
  - Representations
  - A few algorithms

Priority Queue ADT

```
public interface PriorityQueue {
    boolean isEmpty();
    void add (Comparable obj);
    Comparable removeMin();
    Comparable peekMin();
}
```

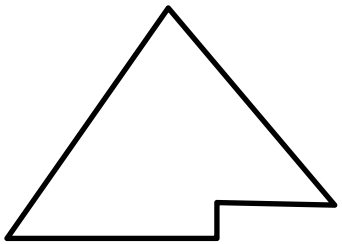
What is one way we can implement a priority queue?

Binary heap (one of many heap data structures)

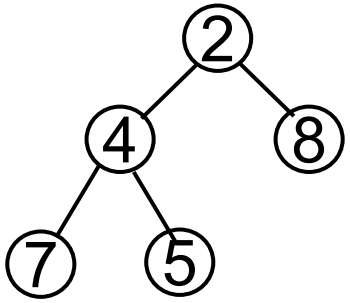
Binary Heaps

Binary heaps are a data structure with two properties:

1. Shape (structure) – complete binary tree



2. Order (heap) – For all nodes,
   - parent >= children (max-heap)
   - parent <= children (min-heap)



Add an element to heap

What property do you maintain first?

Shape (structure)

What do you need to do that?

Put the new element at the end of the heap

What next?

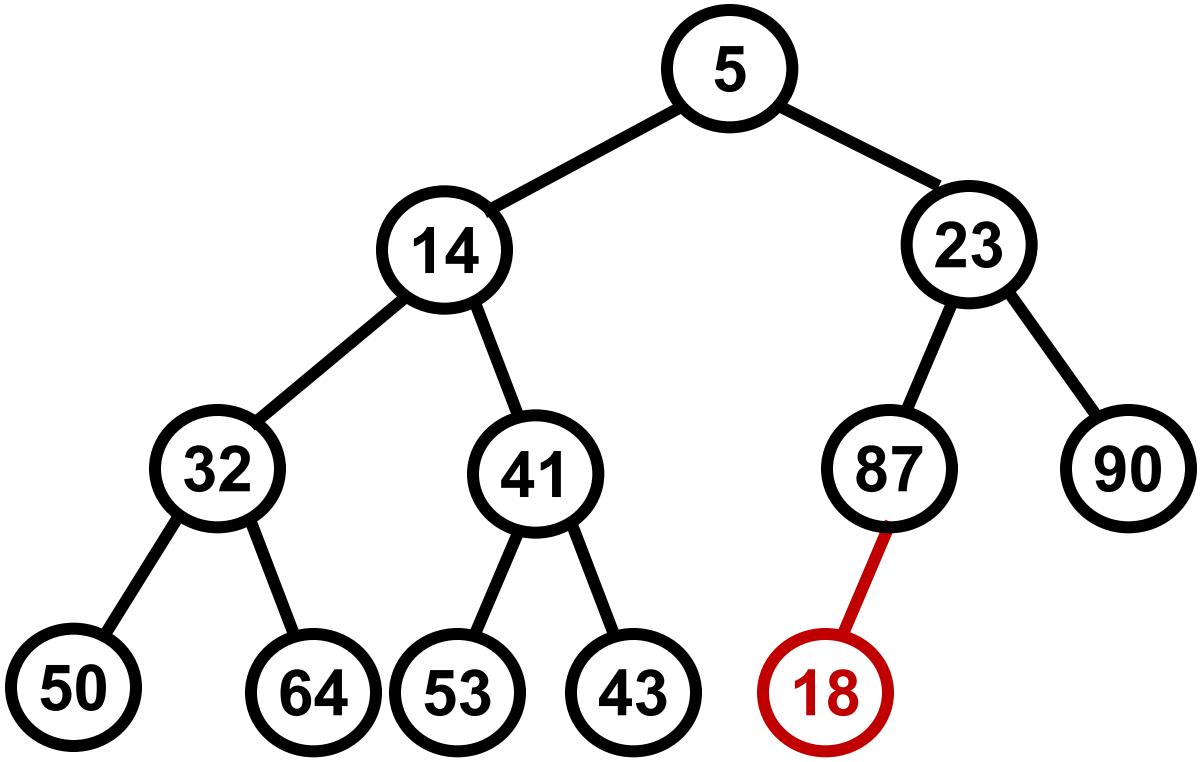
Restore the order (heap) property

How do you do that?

"Heapify up": Repeatedly swap with its parent

Add to a min-heap

add 18

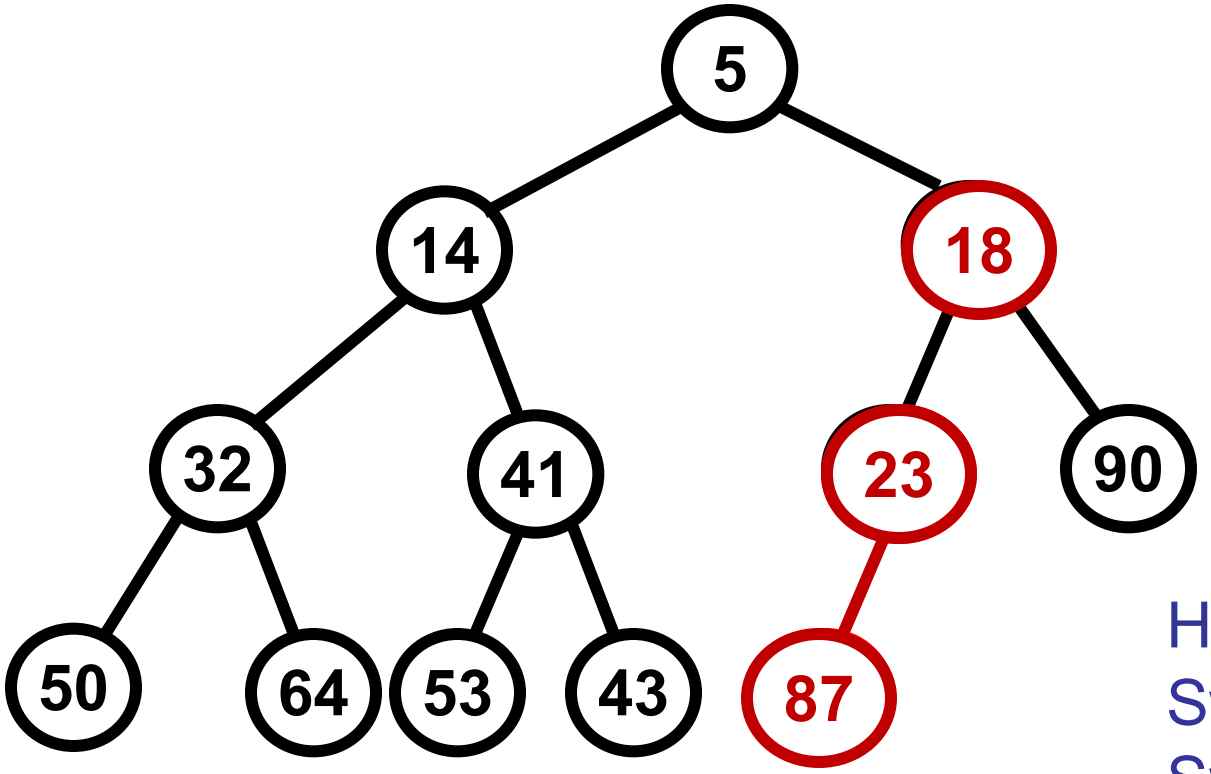


1. Add leaf

2. Heapify up:
   (see next slide)

Add to a min-heap

18 added



Heapified up:
Swapped 18 & 87
Swapped 18 & 23

Remove the minimum

Which element do we remove from a heap?

The root (element at index 1)

What property do you fix first?

Shape (structure)

What do you need to do that?

Remove the last element and put it at the root

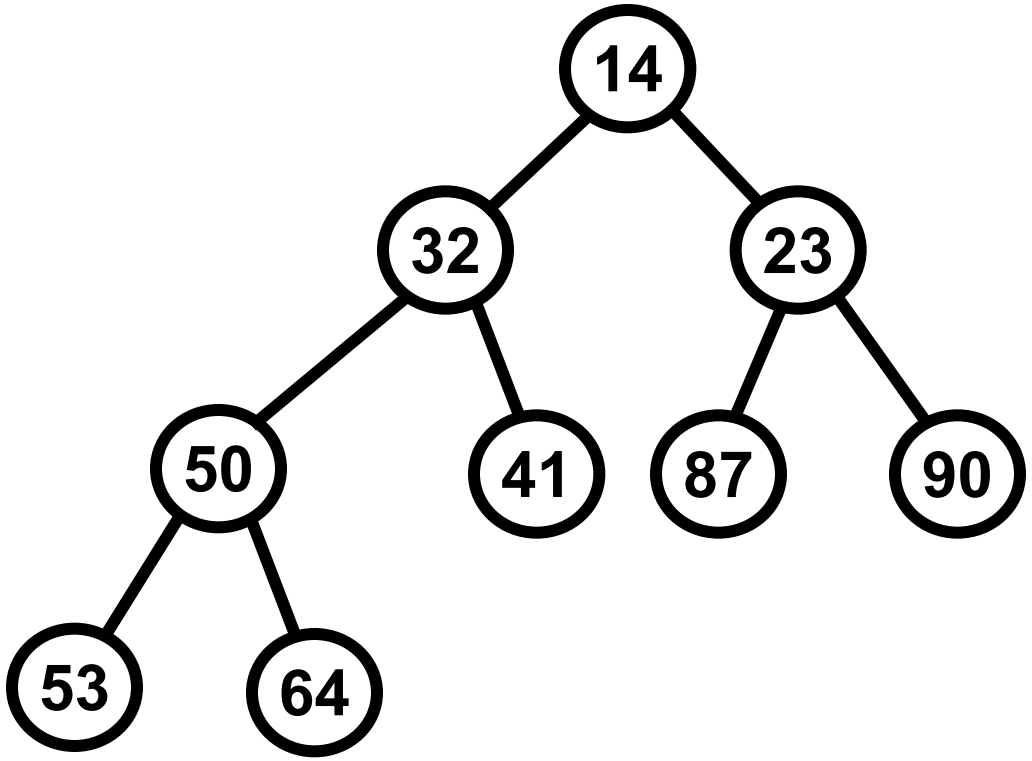
What next?

"Heapify down" to restore the heap property:

Repeatedly swap with smaller child (min-heap) or larger child (max-heap)

Removing from a min-heap

Remove min (14)



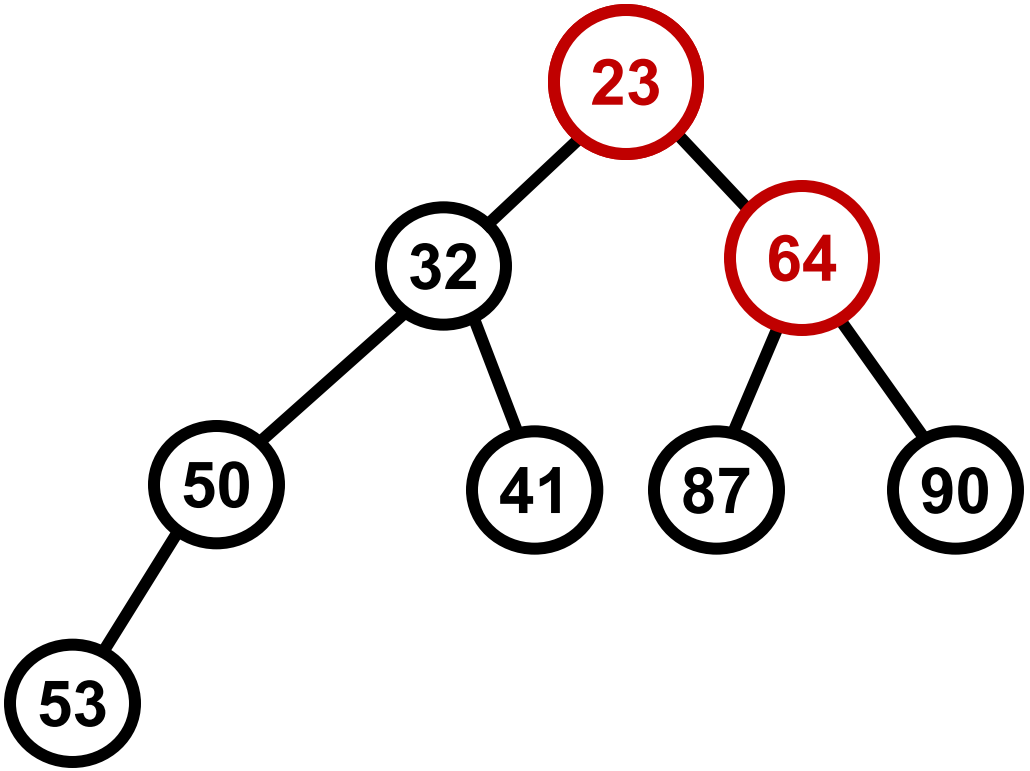
returnValue 14

1. Put 64 at the root

(continued)

Removing from a min-heap

Remove min (14)



returnValue 14

1. Put 64 at the root
2. heapify down:
Swap 64 & 23 (not 32)

If the data structure is a binary tree

Add Problem:

How can I find where to put the new element?

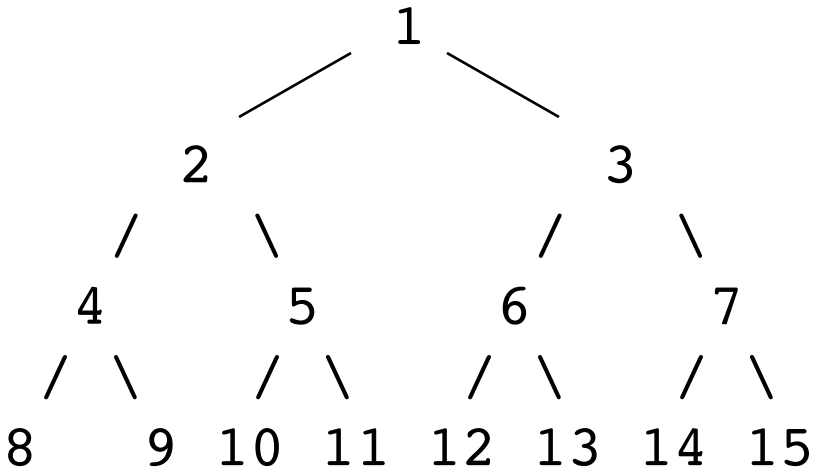
How do we find the parent of a child?

Remove Problem:

How can I find the element to put at the root?

Towards a data structure

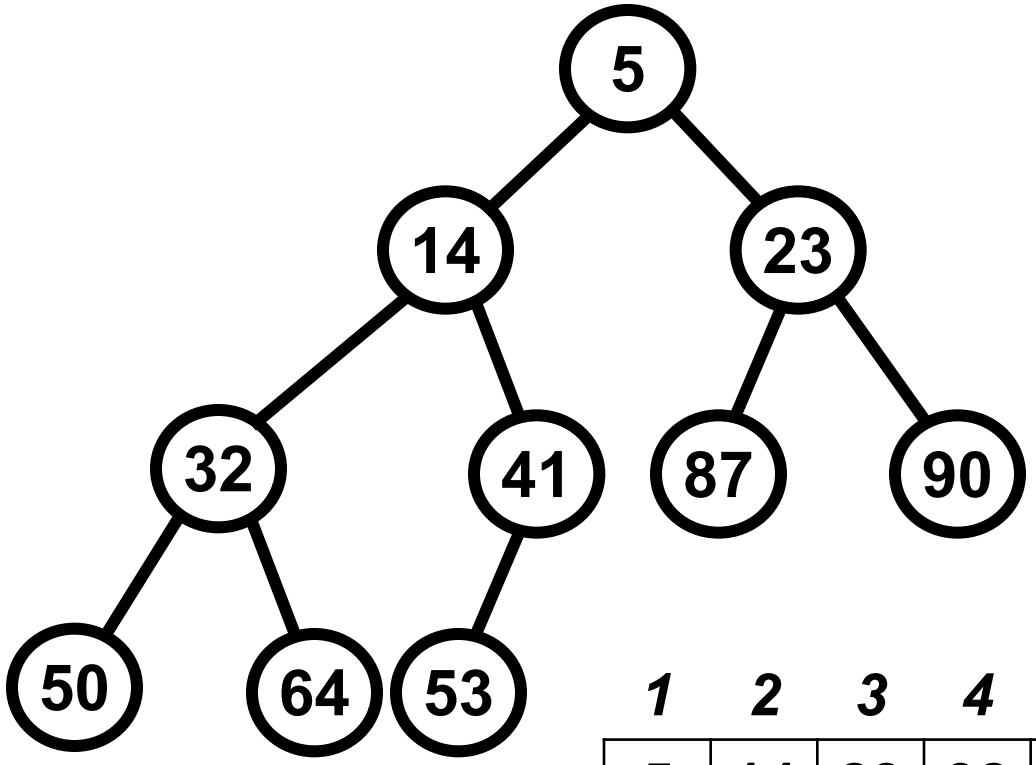
Suppose we number the nodes of the binary heap as follows. Do you see a relationship between a node and its children? A node and its parent?



For a node numbered i
- left child is 2*i
- right child is 2*i + 1
- parent is i / 2 (integer division)

Using this indexing we can store a binary tree in an array (starting at index 1).

Array implementation: put the nodes of the tree into the array level by level



For a node i
- left child is 2*i
- right child is 2*i + 1
- parent is i / 2

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
5	14	23	32	41	87	90	50	64	53

Binary heaps runtime complexity

What is the height of the binary heap? O(log n) – ALWAYS

Runtime (min-heap)**:**

isEmpty: O(1)

peekMin: O(1)

add: best: O(1) – sometimes add a large element

expected: O(1) – most nodes are at bottom 2 layers

worst: O(log n) – sometimes move up to root

removeMin: O(log n) – always move a large element from the root down (usually to bottom 2 layers)

Heap Sort

If we add n values to an empty min-heap and then we remove all the values from a heap, in what order will they be removed?

Smallest to largest. We just invented Heap Sort!

Heap Sort Runtime:

1. Build the heap: n * O(log n)
2. Repeatedly remove the min: n * O(log n)

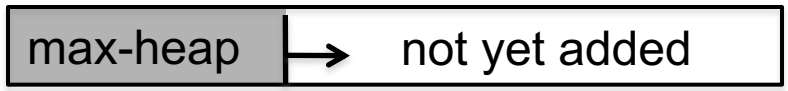
Total: O(n log n): best, expected, and worst case

What other sort has the same worst-case runtime? Merge sort

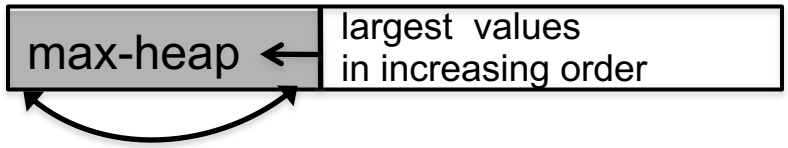
What is the disadvantage of merge sort? Not in place

Heap Sort (in place)

1. Build a **max**-heap by adding each successive element in the array to the heap.



2. Remove the maximum and put it at the last index, remove the next maximum and put it at 2nd to last index, and so on. In particular, repeatedly swap the root with last element in the heap and heapify down the new root to restore the heap one size smaller.



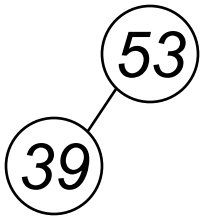
parent of j = (j-1)/2

1. Building the max-heap

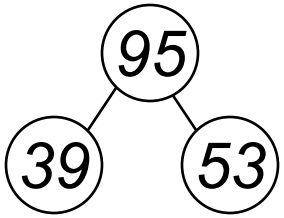
ADD NEXT VALUE TO HEAP AND FIX HEAP

0	1	2	3	4	5	6
39	**53**	95	72	61	48	83

0	1	2	3	4	5	6
53	**39**	**95**	72	61	48	83



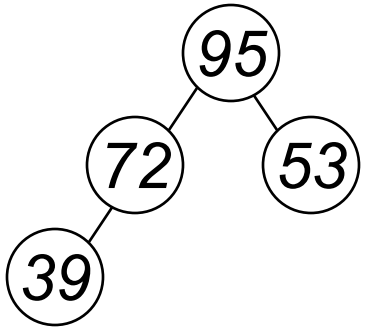
0	1	2	3	4	5	6
95	**39**	**53**	**72**	61	48	83



parent of j = (j-1)/2

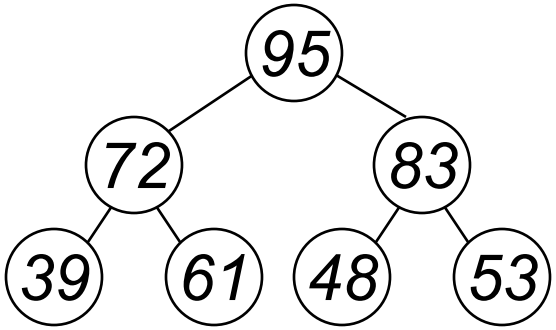
1. Building the max-heap (cont'd)

0	1	2	3	4	5	6
95	72	53	39	61	48	83



CONTINUE UNTIL THE HEAP IS COMPLETED...

0	1	2	3	4	5	6
95	72	83	39	61	48	53



children of j = (j+1)*2-1, (j+1)*2

2. Sorting from the heap

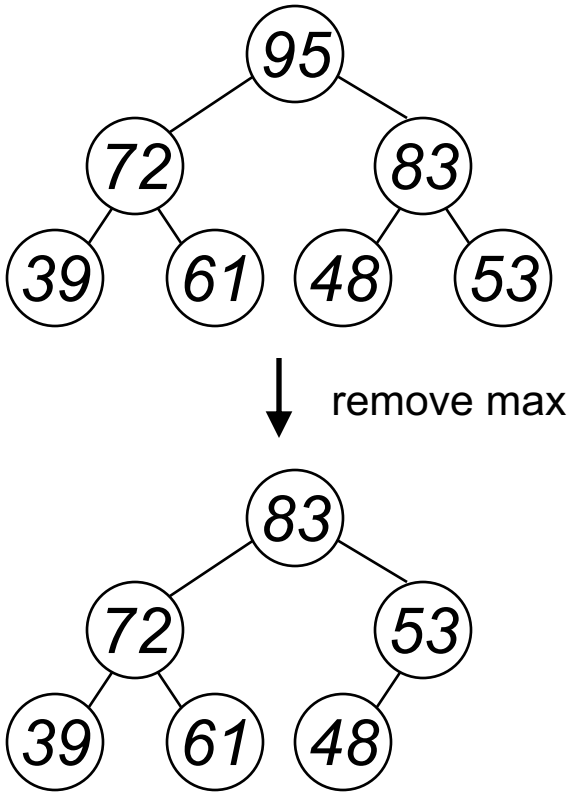
0	1	2	3	4	5	6
95	72	83	39	61	48	53

SWAP THE MAX OF THE HEAP
WITH THE LAST VALUE OF THE HEAP:

0	1	2	3	4	5	6
53	72	83	39	61	48	95

FIX THE HEAP (NOT INCLUDING MAX):

0	1	2	3	4	5	6
83	72	53	39	61	48	95



children of j = (j+1)*2-1, (j+1)*2

2. Sorting from the heap (cont'd)

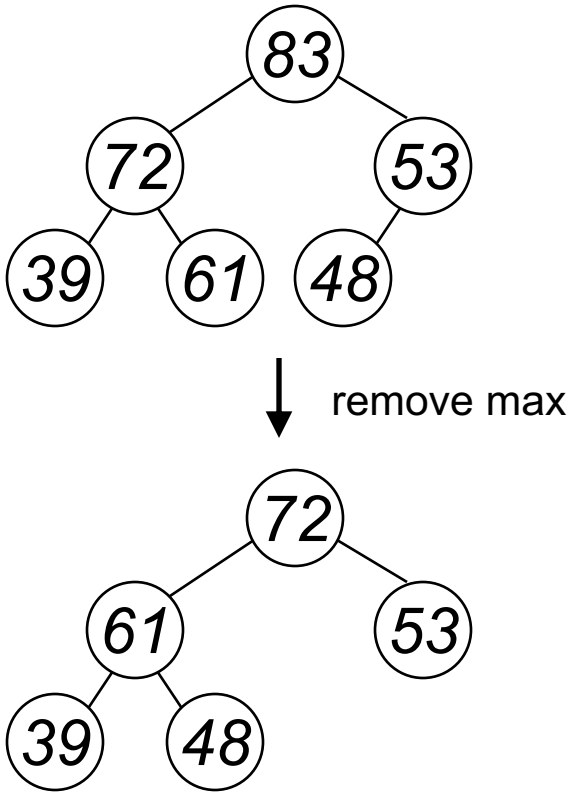
0	1	2	3	4	5	6
83	72	53	39	61	48	95

SWAP THE MAX OF THE HEAP
WITH THE LAST VALUE OF THE HEAP:

0	1	2	3	4	5	6
48	72	53	39	61	83	95

FIX THE HEAP (NOT INCLUDING MAX):

0	1	2	3	4	5	6
72	61	53	39	48	83	95



children of j = (j+1)*2-1, (j+1)*2

2. Sorting from the heap (cont'd)

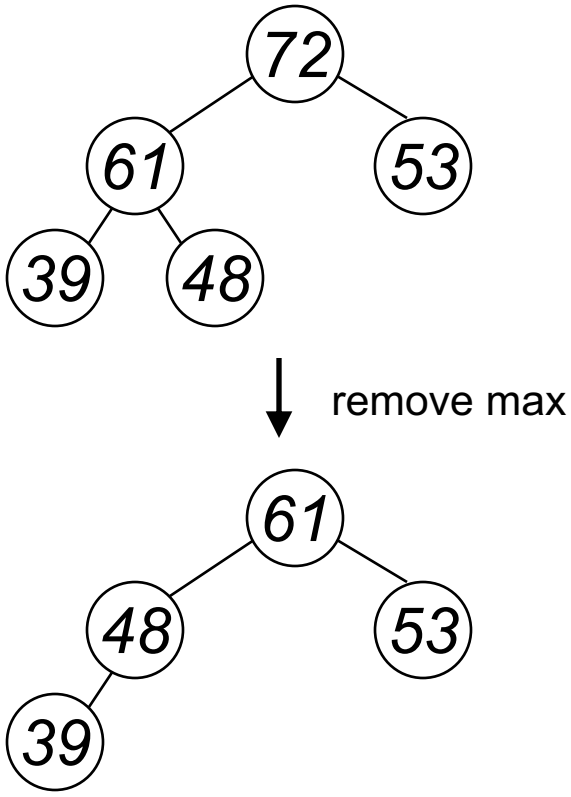
0	1	2	3	4	5	6
72	61	53	39	48	83	95

SWAP THE MAX OF THE HEAP
WITH THE LAST VALUE OF THE HEAP:

0	1	2	3	4	5	6
48	61	53	39	72	83	95

FIX THE HEAP (NOT INCLUDING THAT MAX):

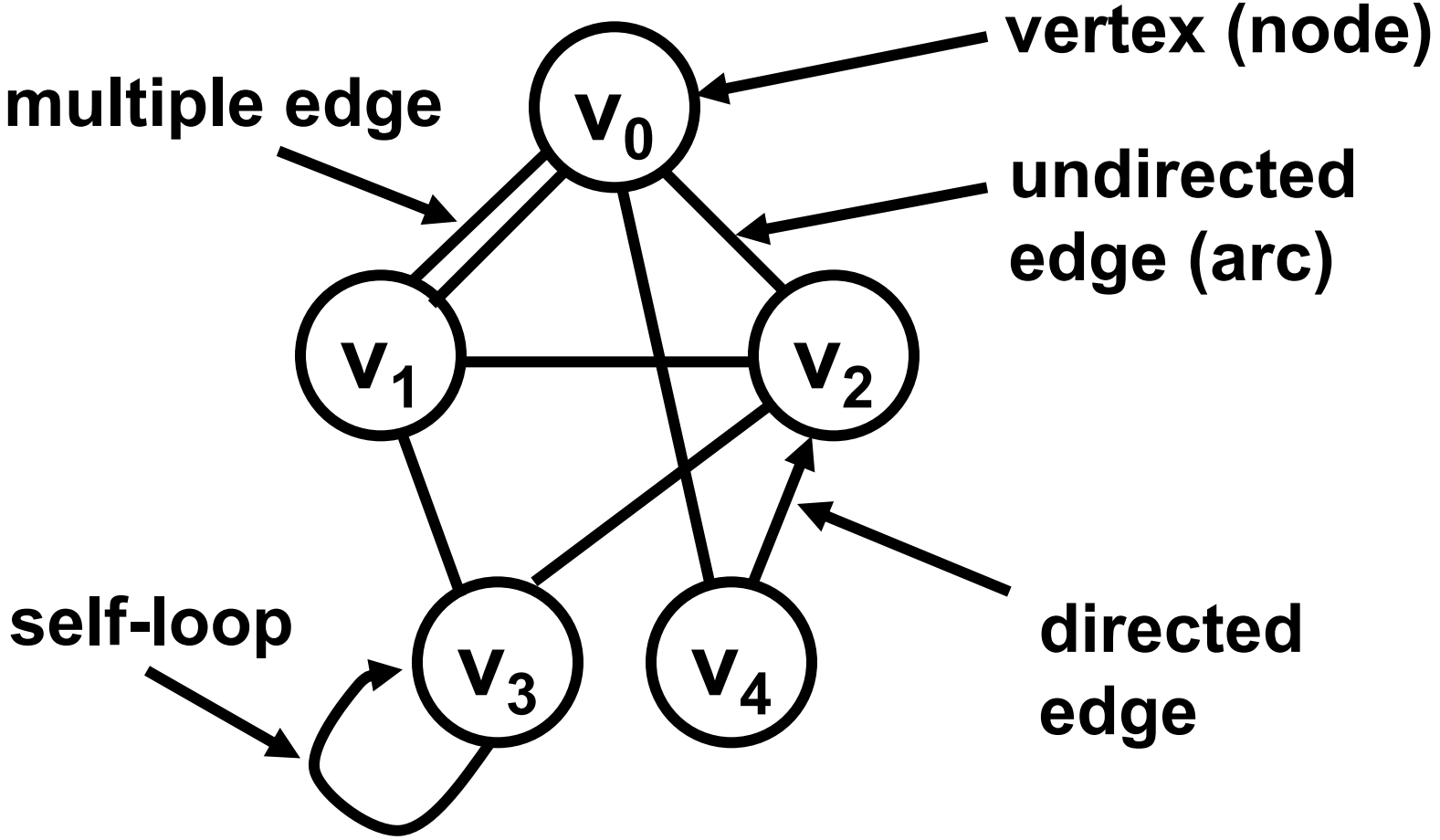
0	1	2	3	4	5	6
61	48	53	39	72	83	95



REPEAT UNTIL THE HEAP HAS 1 NODE LEFT

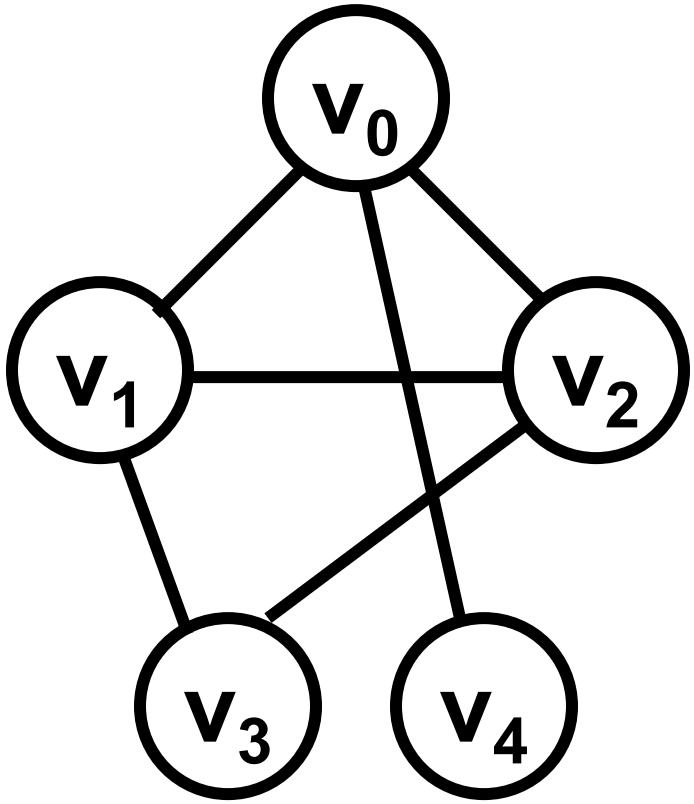
Graphs

A graph G = (V,E) is a set of vertices V and a collection of edges E



In an undirected graph, an edge E = (x,y) connects vertex x to vertex y (and vice-versa)

Thus, the edges (x,y) and (y,x) are the same edge.



$V = \{v_0, v_1, v_2, v_3, v_4\}$

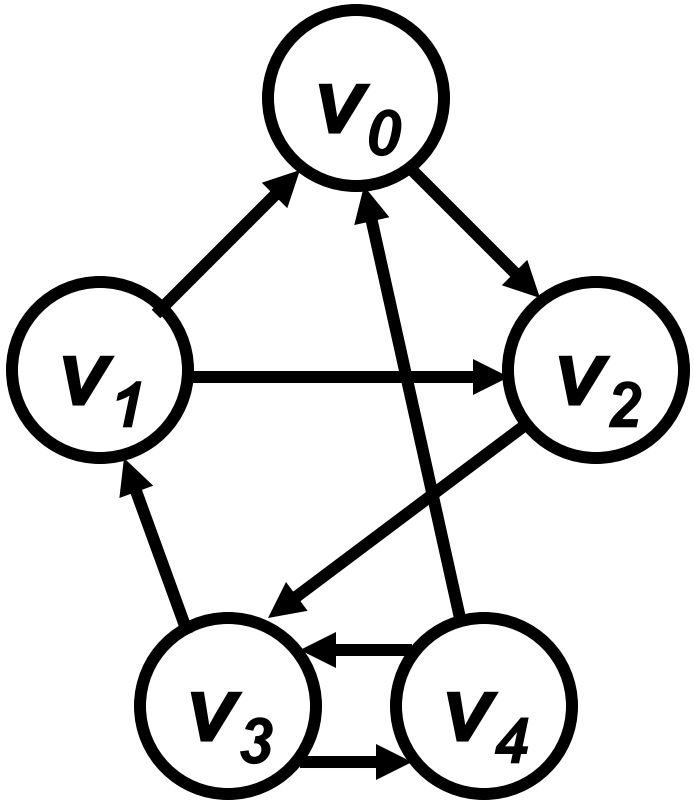
$E = \{ \{v_0, v_1\}, \{v_0, v_2\}, \{v_0, v_4\}, \{v_1, v_2\}, \{v_1, v_3\}, \{v_2, v_3\} \}$

An undirected graph with n vertices can have at most how many edges?

(n choose 2) = n * (n-1) / 2

In a directed graph, an edge E = (x,y) connect vertex x to vertex y (but not vice-versa).

Thus, (x,y) and (y,x) are not the same edges.



$V = \{v_0, v_1, v_2, v_3, v_4\}$

$E = \{ \{v_1, v_0\}, \{v_0, v_2\}, \{v_4, v_0\}, \{v_1, v_2\}, \{v_3, v_1\}, \{v_2, v_3\}, \{v_3, v_4\}, \{v_4, v_3\} \}$

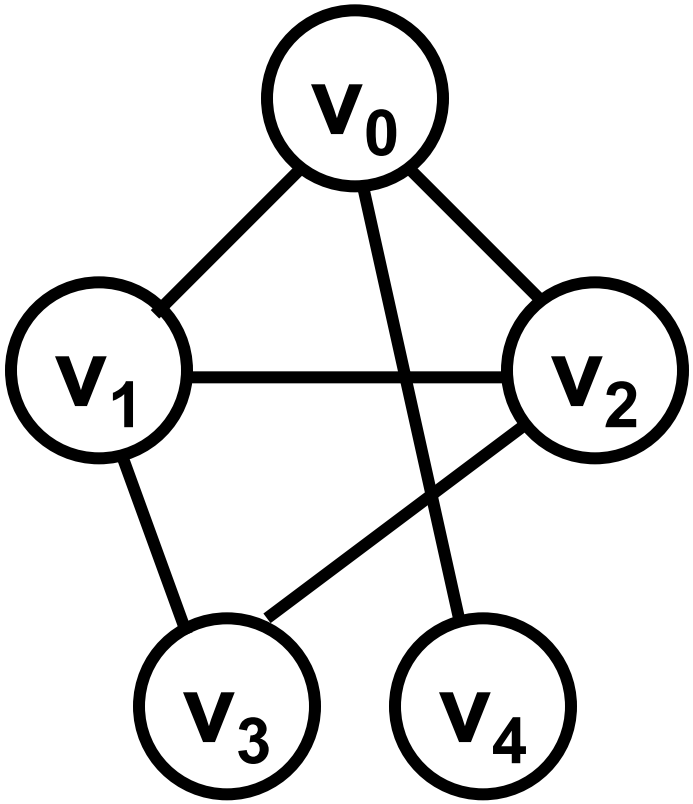
A directed graph with n vertices can have at most how many edges?

n * (n-1)

Graph Terminology

- A graph G = (V,E) is a set of vertices V and a collection of edges E.
- In an undirected graph, an edge E = (x,y) connects vertex x to vertex y (and vice-versa). Thus, the edges (x,y) and (y,x) are the same edge.
- In a directed graph, an edge E = (x,y) connect vertex x to vertex y (but not vice-versa). Thus, (x,y) and (y,x) are not the same edges.
- A simple graph has no multiple edges between vertices or loops from a vertex to itself.

Graph Terminology



v_0 and v_4 are adjacent

paths from v_0 to v_2:

$v_0 \rightarrow v_2$

$v_0 \rightarrow v_1 \rightarrow v_2$

$v_0 \rightarrow v_1 \rightarrow v_3 \rightarrow v_2$

cycles starting at v_3:

$v_3 \rightarrow v_1 \rightarrow v_2 \rightarrow v_3$

$v_3 \rightarrow v_2 \rightarrow v_0 \rightarrow v_1 \rightarrow v_3$

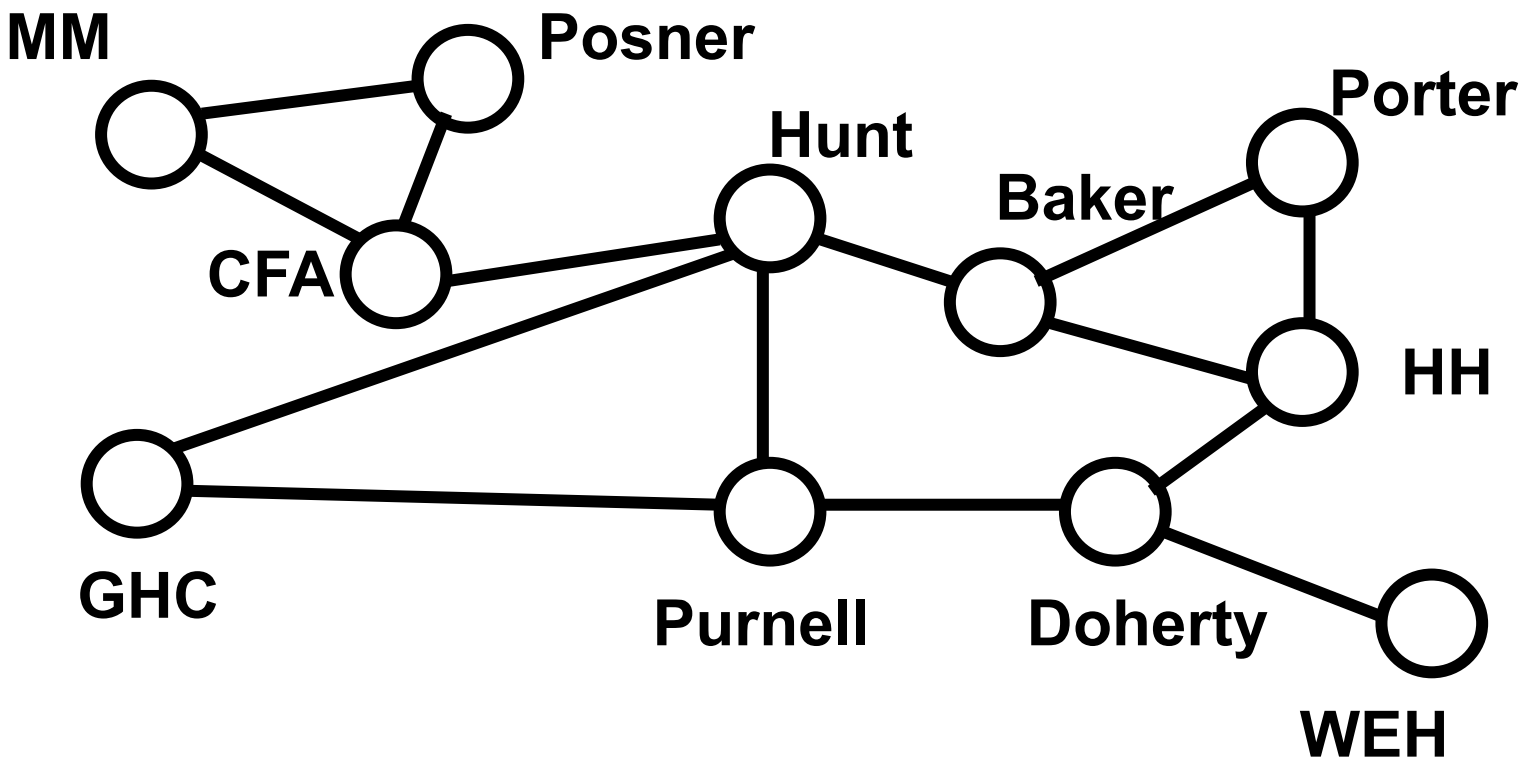
degree of v_2 is 3

More Terminology

- Node v_b is adjacent to node v_a in a graph if there is an edge from v_a to v_b.
- A path in a graph is a sequence of vertices p_0, ..., p_n such that each adjacent pair of vertices p_k and p_{k+1} are connected by an edge from p_k to p_{k+1}.
- A cycle is a path that starts and ends at the same vertex (i.e. $p_0 = p_n$).
- The degree of a vertex in an undirected graph is the number of edges that connect to the vertex.

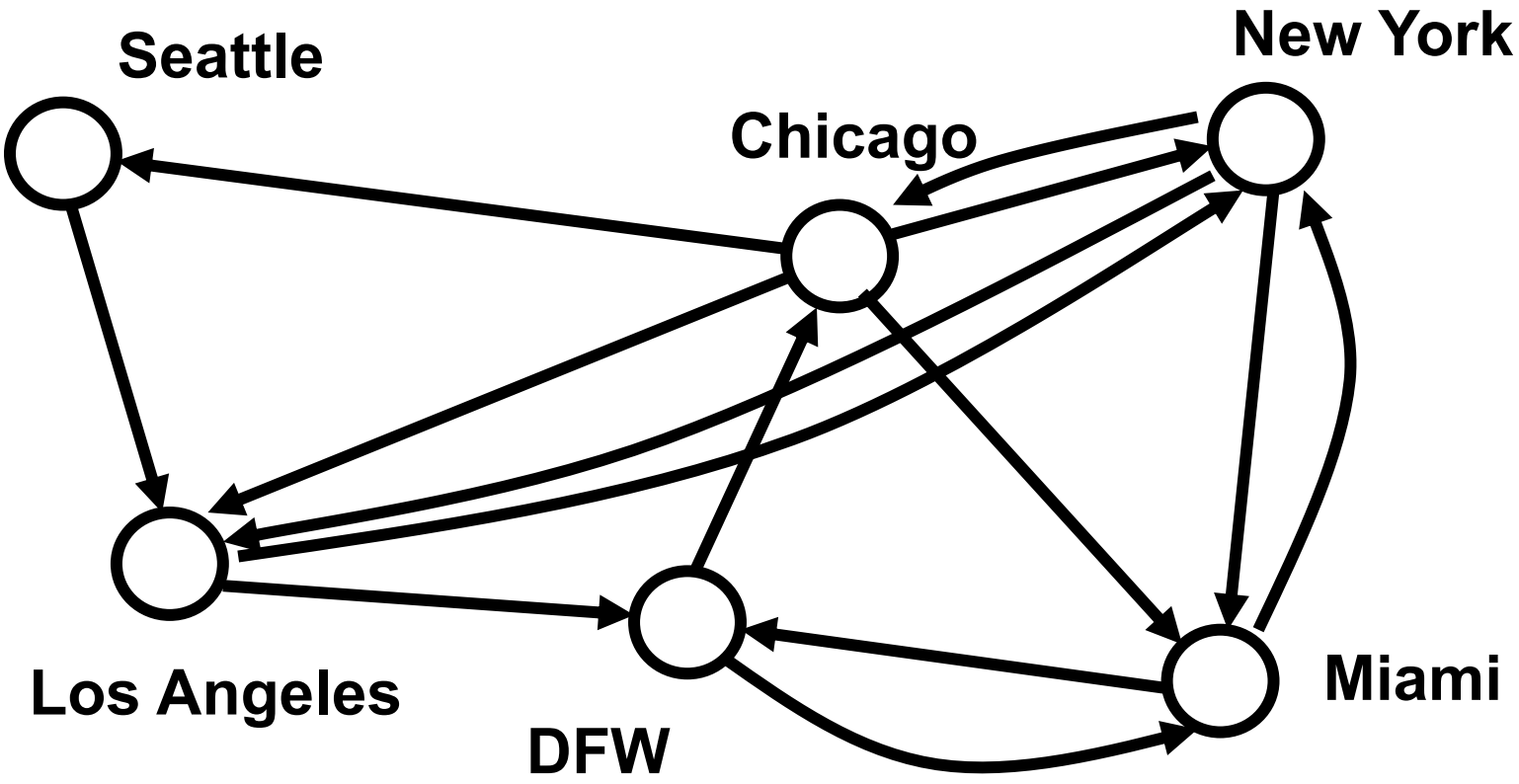
Graph Examples

- Communication Networks



Graph Examples

- Transportation Routes



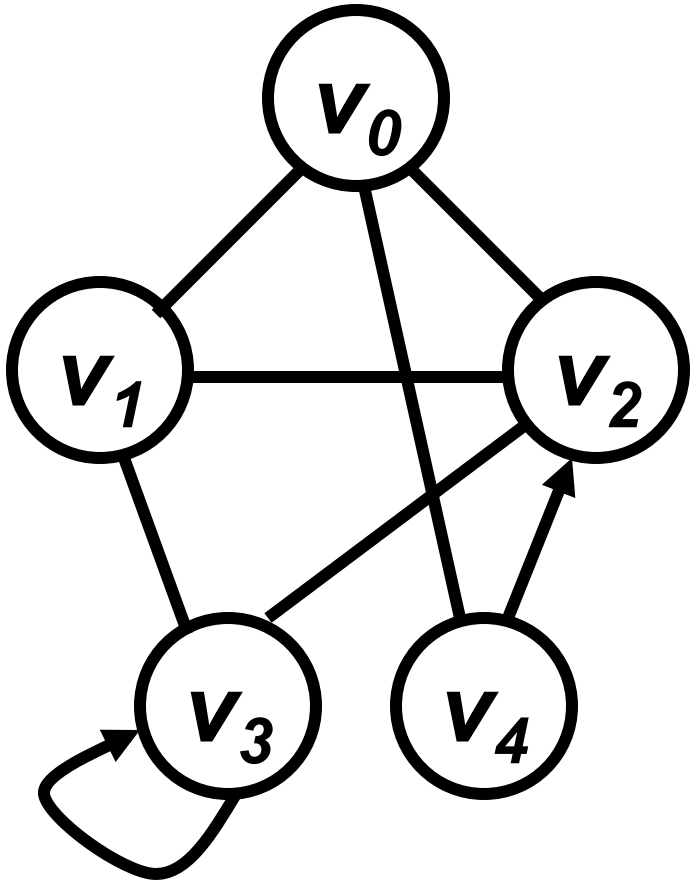
Storing a graph

Adjacency Matrix

An adjacency matrix G for an n-vertex graph is an n x n array of 0/1 values such that $G_{jk} = 1$ if vertex k is adjacent to vertex j; otherwise $G_{jk} = 0$.

In other words, $G_{jk} = 1$ if there is an edge from vertex j to vertex k; otherwise it is 0.

Example (Adjacency Matrix)



destination

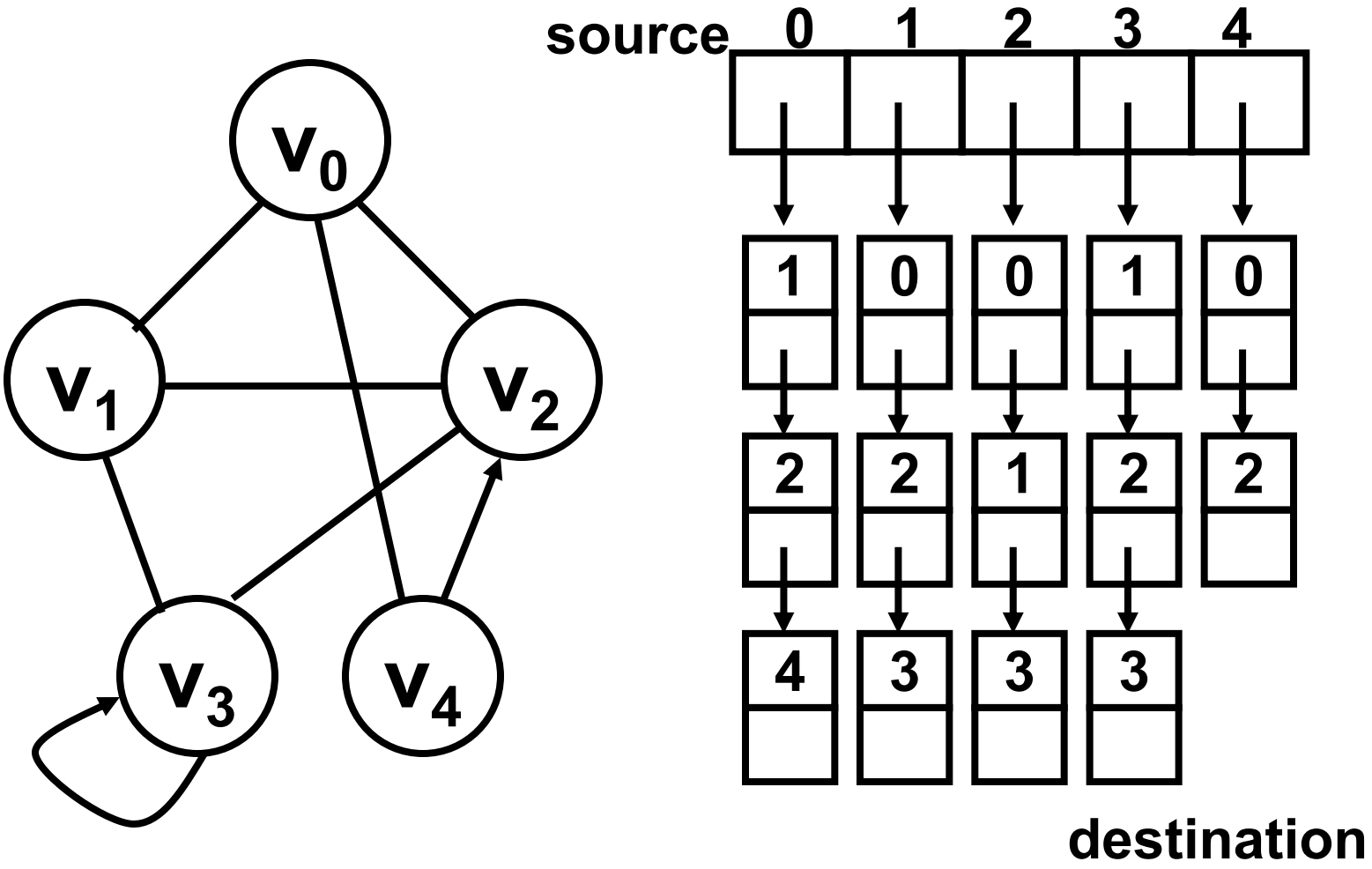
source	0	1	2	3	4
0	0	1	1	0	1
1	1	0	1	1	0
2	1	1	0		
3	1	1	1	1	
4	1	0			0

Storing a graph: Another way

Adjacency List: an array of edge lists.

- An **edge list** for vertex k is a linked list that stores all nodes that are adjacent to vertex k.
- There is a linked list for every vertex of the graph.

Example (Adjacency List)

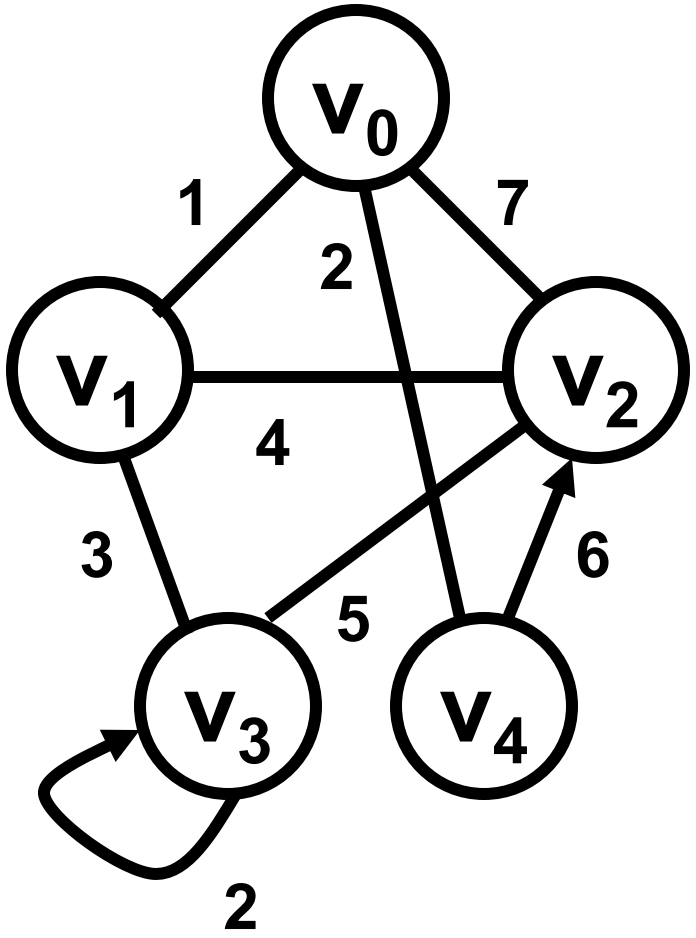


Weighted Graphs

Some graphs have an associated “weight” assigned to each edge.

- Weights: cost, distance, capacity, etc.
- Costs are typical non-negative integer values.
- Possible problems to solve using weighted graphs: shortest path between nodes, minimal spanning tree, etc.

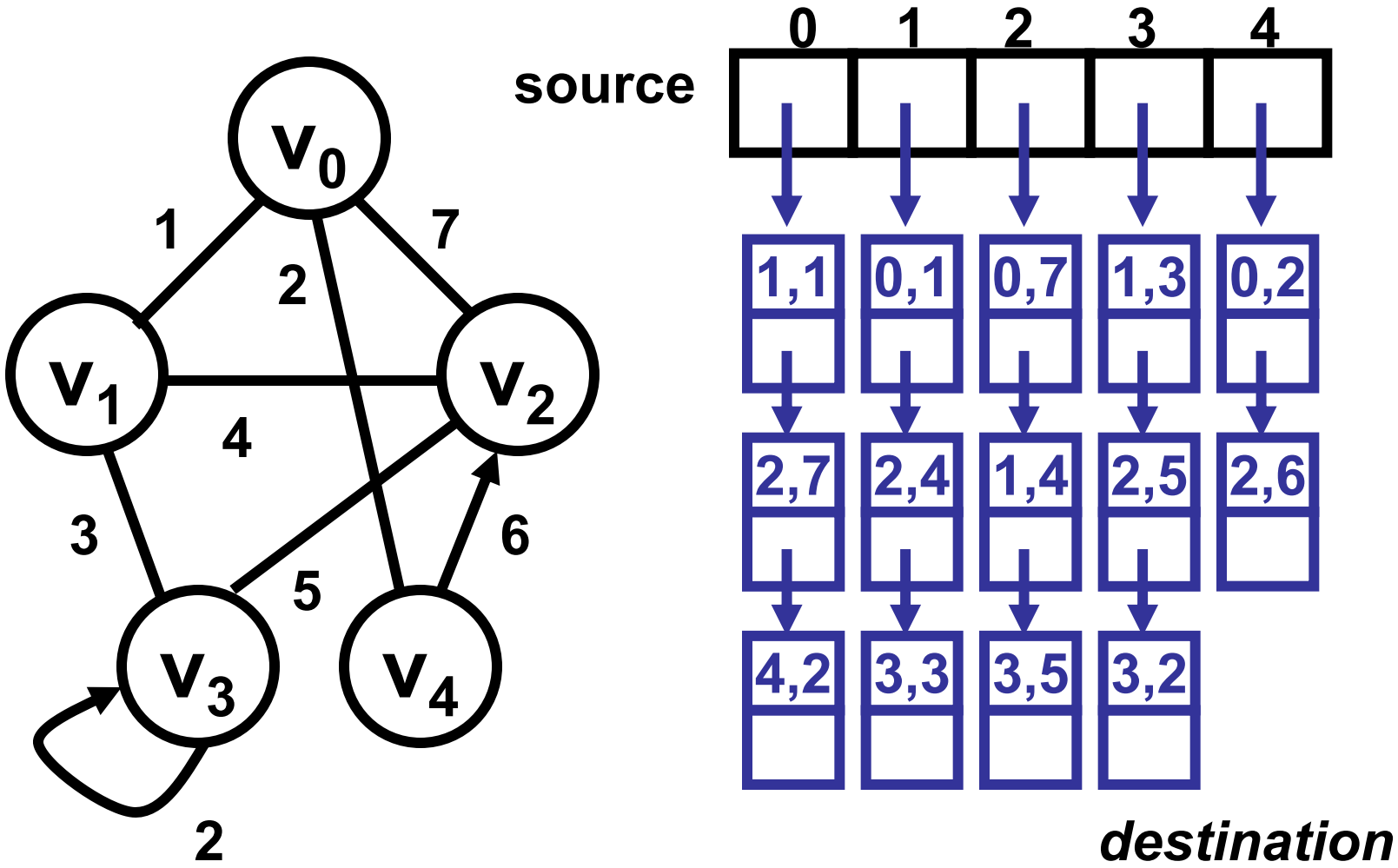
Weighted graph represented as an Adjacency Matrix



destination

source	0	1	2	3	4
0	∞	1	7	∞	2
1	1	∞	4	3	∞
2					
3					
4					

Example (weighted)



Storage methods advantages (+) and disadvantages(-)

- **Adjacency Matrix**
  - \+ O(1) to add or remove an edge
  - \+ O(1) to determine if an edge exists in a graph.
  - \+ Generally better for **dense** graphs where $|E| = O(|V|^2)$
  - \- Storage: $O(|V|^2)$
- **Adjacency Lists**
  - \+ Faster to perform an operation on all nodes adjacent to a node in a sparse graph.
  - \+ Generally better for **sparse** graphs where $|E| = O(|V|)$
  - \+ Storage: $O(|V| + |E|)$
  - \- Access edge (u, v) O(degree of v)