

Greedy Algorithms

Lecture 5

Timothy Kim
GWU CSCI 6212

Quiz

- You are given n activity schedule $[s_i, f_i]$ for $1 \leq i \leq n$ for one room, where s_i and f_i denote the start and the finishing time of activity i . You are to select the maximum number of activities that can be schedule such that no two activities have an overlapping period. Design a greedy algorithm.

Answer

1. **Schedule(s, f):**
2. Sort (s, f) by f
3. $S = \{1\}$
4. curr = f[1]
5. for i = 2 to n:
6. if $s[i] \geq \text{curr}$:
7. $S = S \cup i$
8. curr = f[i]
- 9.
10. *Proof of optimality for homework*

Agenda

- Data Structure
 - Heap
 - Graphs
- Greedy Algorithms
 - MST - Minimum Spanning Tree problems
 - Dijkstra's Shortest Path

Heap

- **max-heap:** A complete binary tree, where all nodes have key values that are greater than or equal to each of its children.
- **min-heap:** A complete binary tree, where all nodes have key values that are less than or equal to each of its children.

Heap Operations

- **Insert:** add a new object to a heap - $O(\log n)$
- **Extract Min:** remove a node in with a minimum key value - $O(\log n)$
- **Heapify:** n batched inserts - $O(n)$
- **Delete:** remove a node - $O(\log n)$

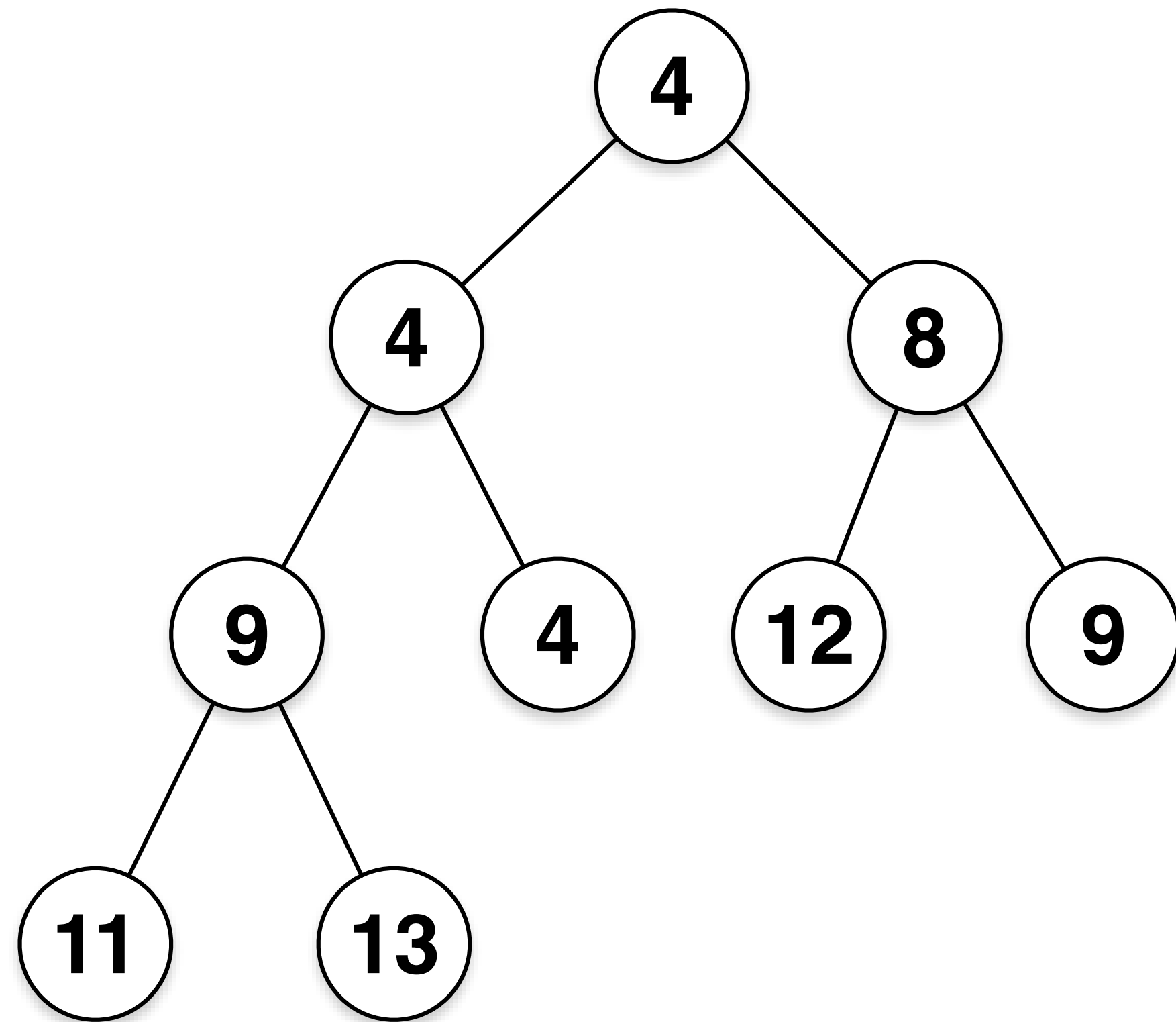
Application

- If your application requires minimum calculation repeatedly
 - **Selection Sort:** find min $O(n^2)$
 - **Heap Sort:** same as selection sort but $O(n \log n)$
 - Heapify input array - $O(n)$
 - Extract-min n times - $O(n \log n)$
- Event manager - "priority queue"
- Median maintenance - extract median

Implementation

- Usually trees are implemented using pointers and references. However because Heaps are complete binary tree, we can use an array.
- root node: 1st element
- $\text{child}(i) = (2i, 2i + 1)$
- $\text{parent}(i) = \begin{cases} i/2 & \text{if } n \text{ is even} \\ \lfloor i/2 \rfloor & \text{if } n \text{ is odd} \end{cases}$

Implementation



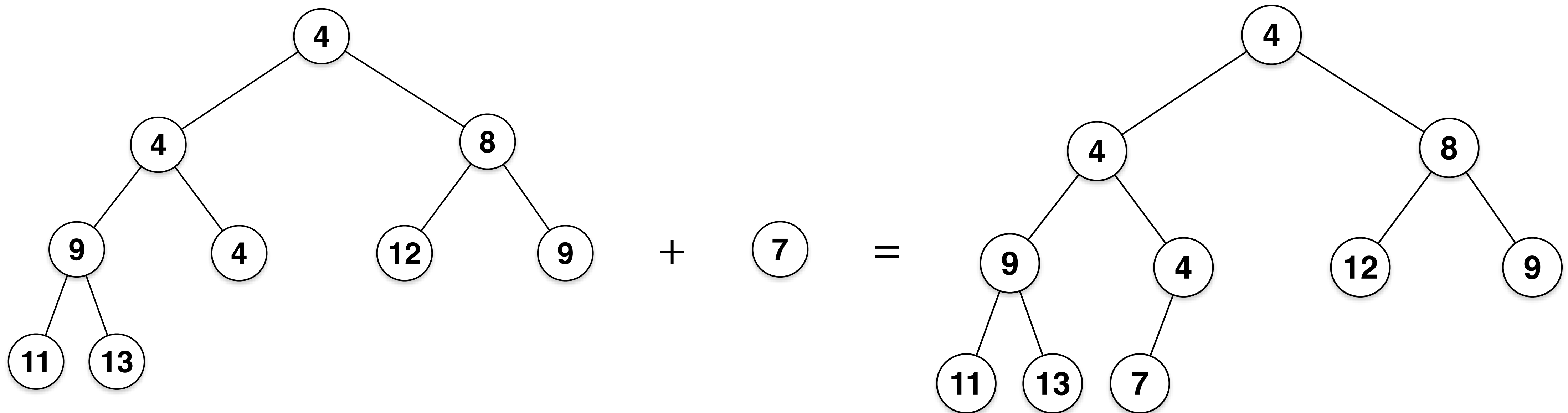
1	2	3	4	5	6	7	8	9
4	4	8	9	4	12	9	11	13

Insert

1. **Insert(k):**
2. Add k to the next place (append)
3. if (parent(k).key < k.key):
4. done
5. else:
6. heapify-up (bubble up)

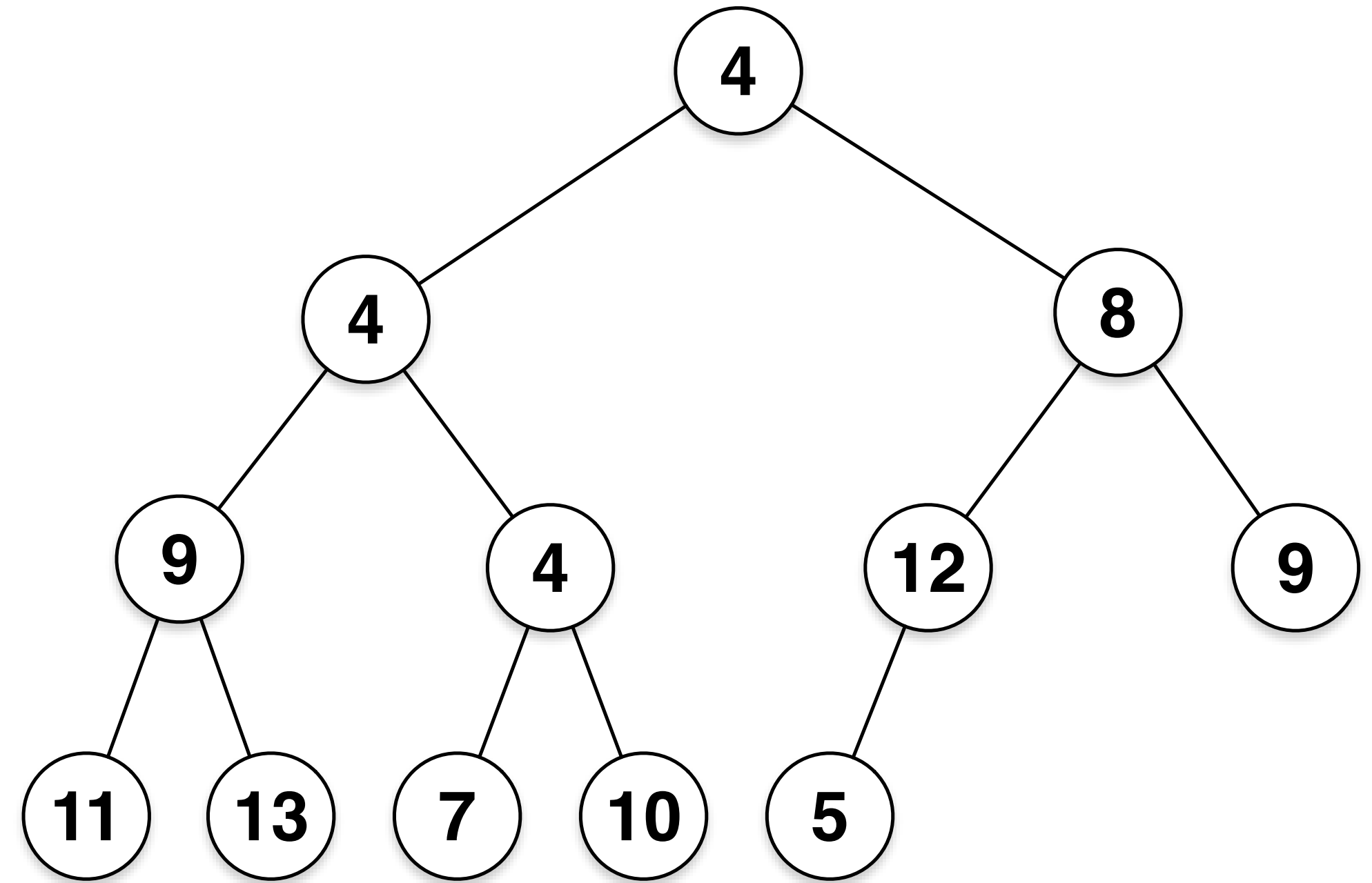
- **Invariant:** all nodes has key value that are less than or equal to each of its children

Example

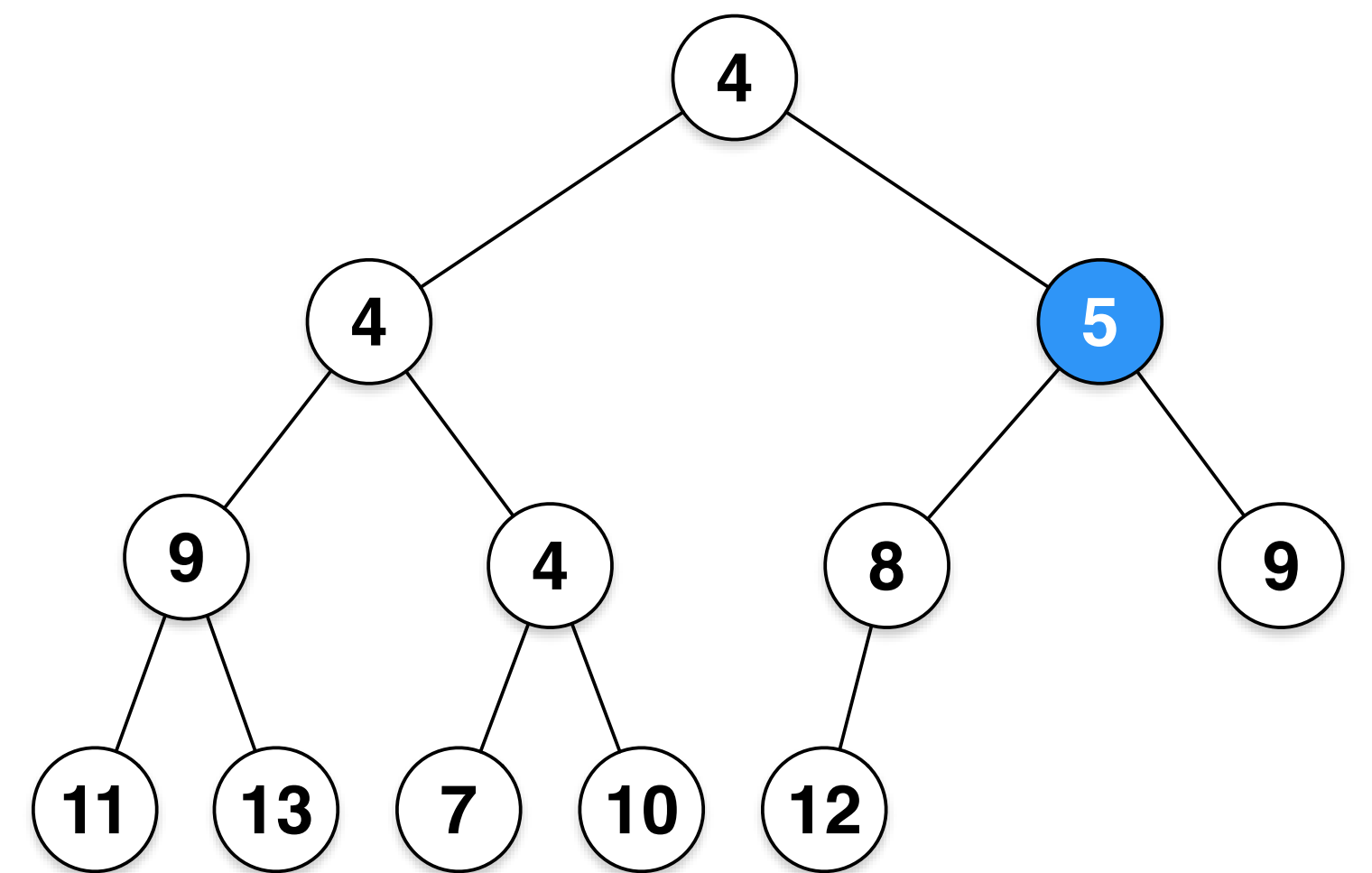
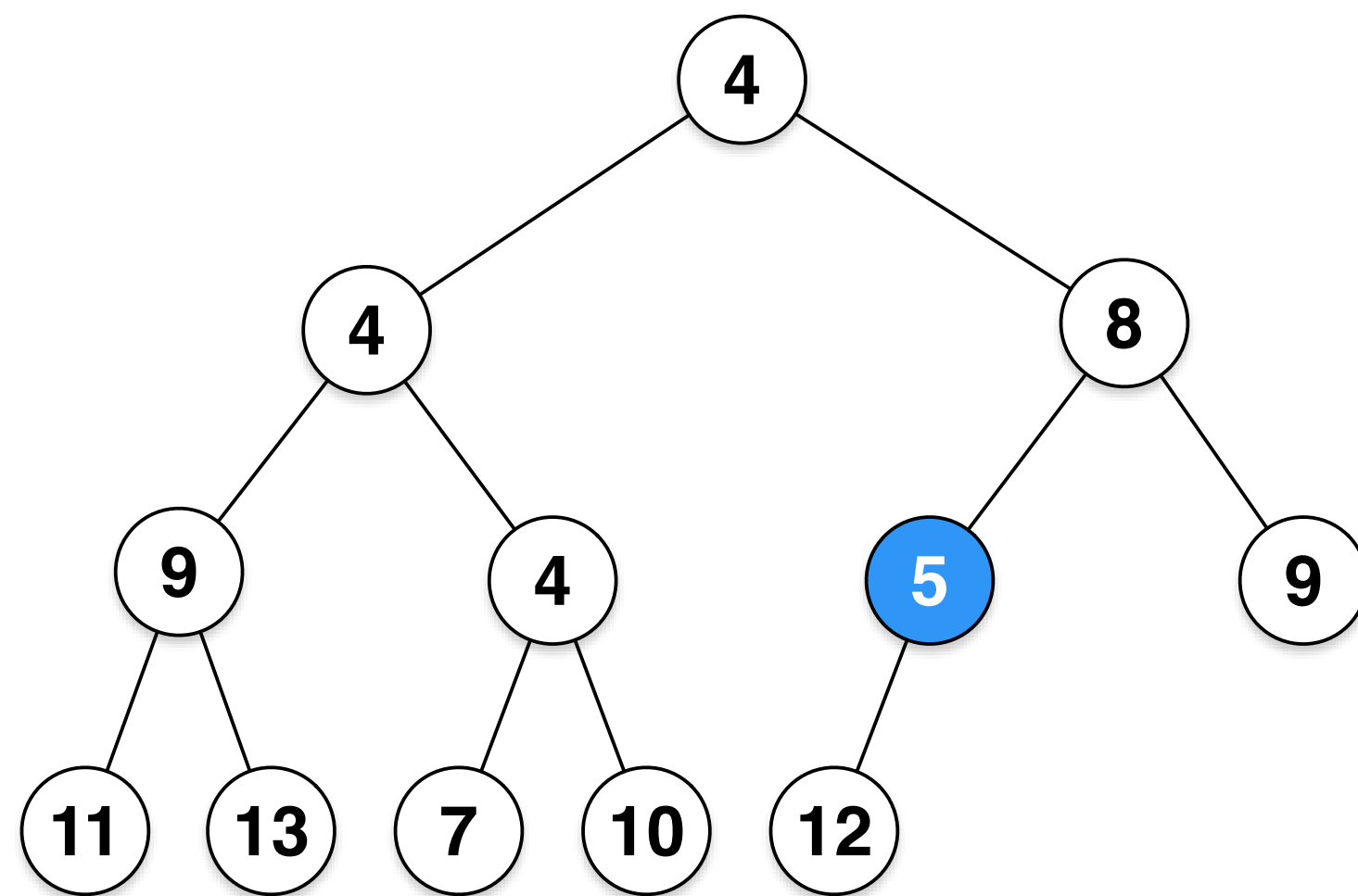
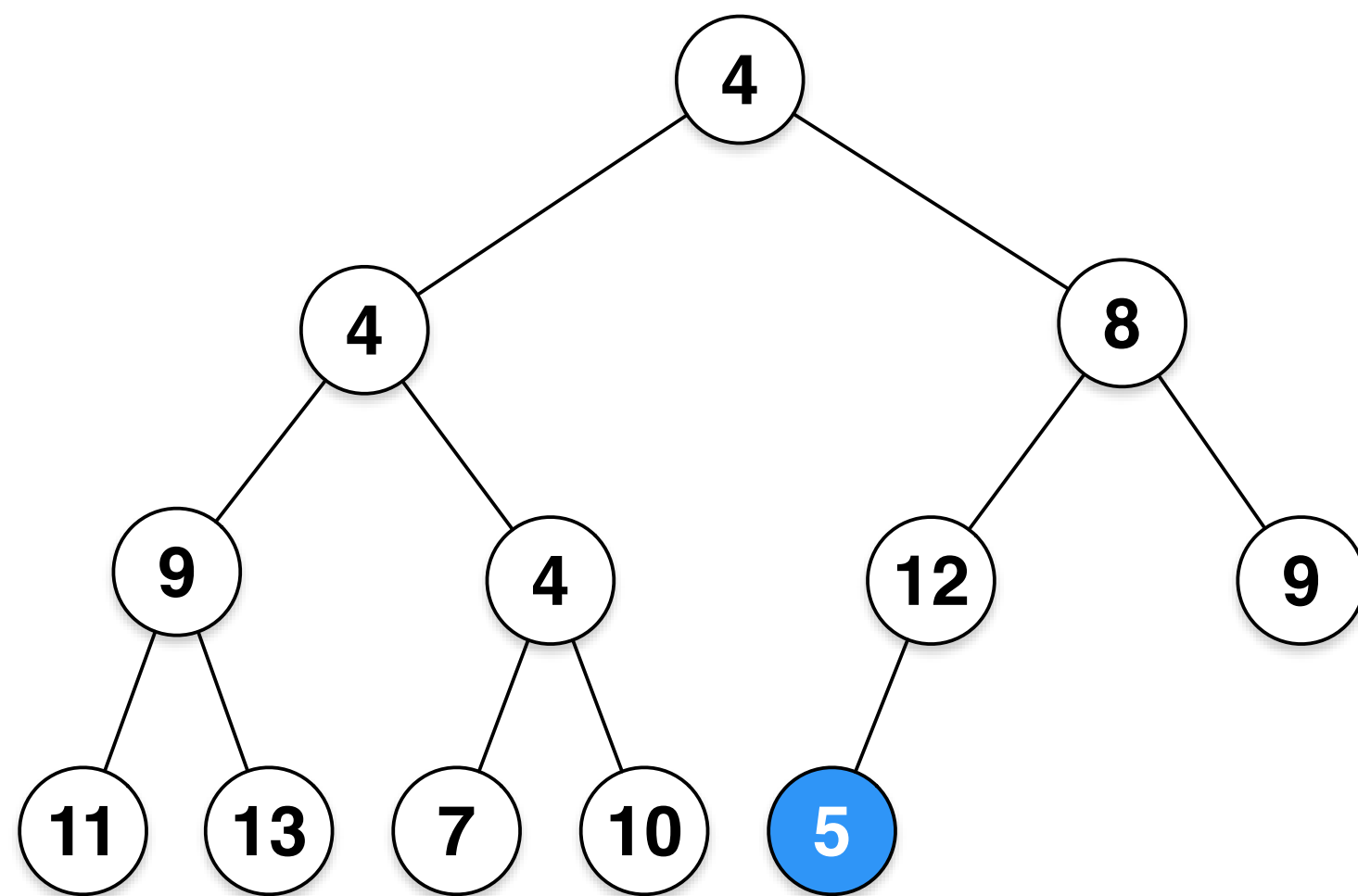


Heapify Up

1. **HeapifyUp**(A, k):
2. while(parent(k).key > k.key):
3. swap(parent(k), k)



Heapify Up



Insert Running Time

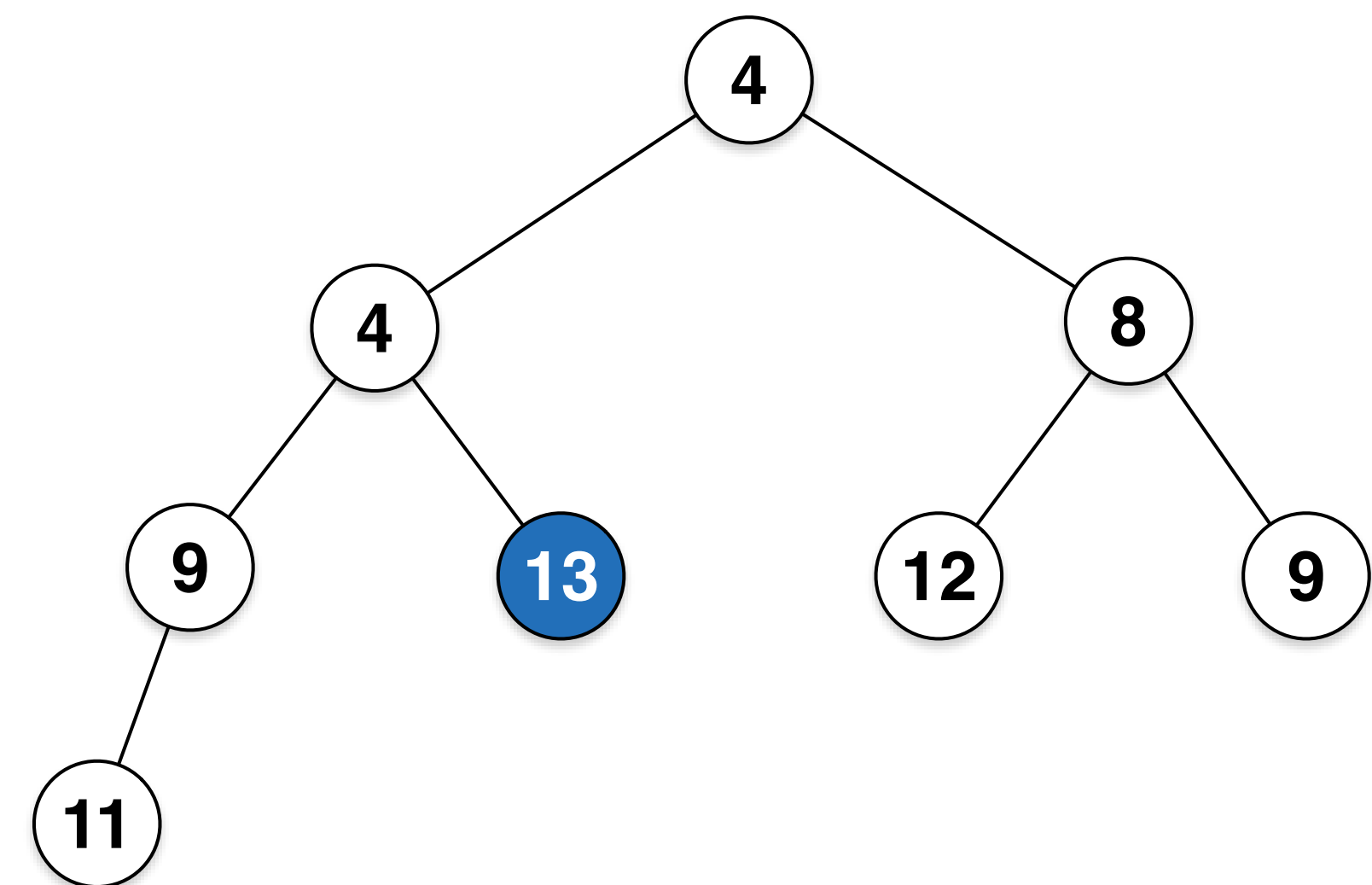
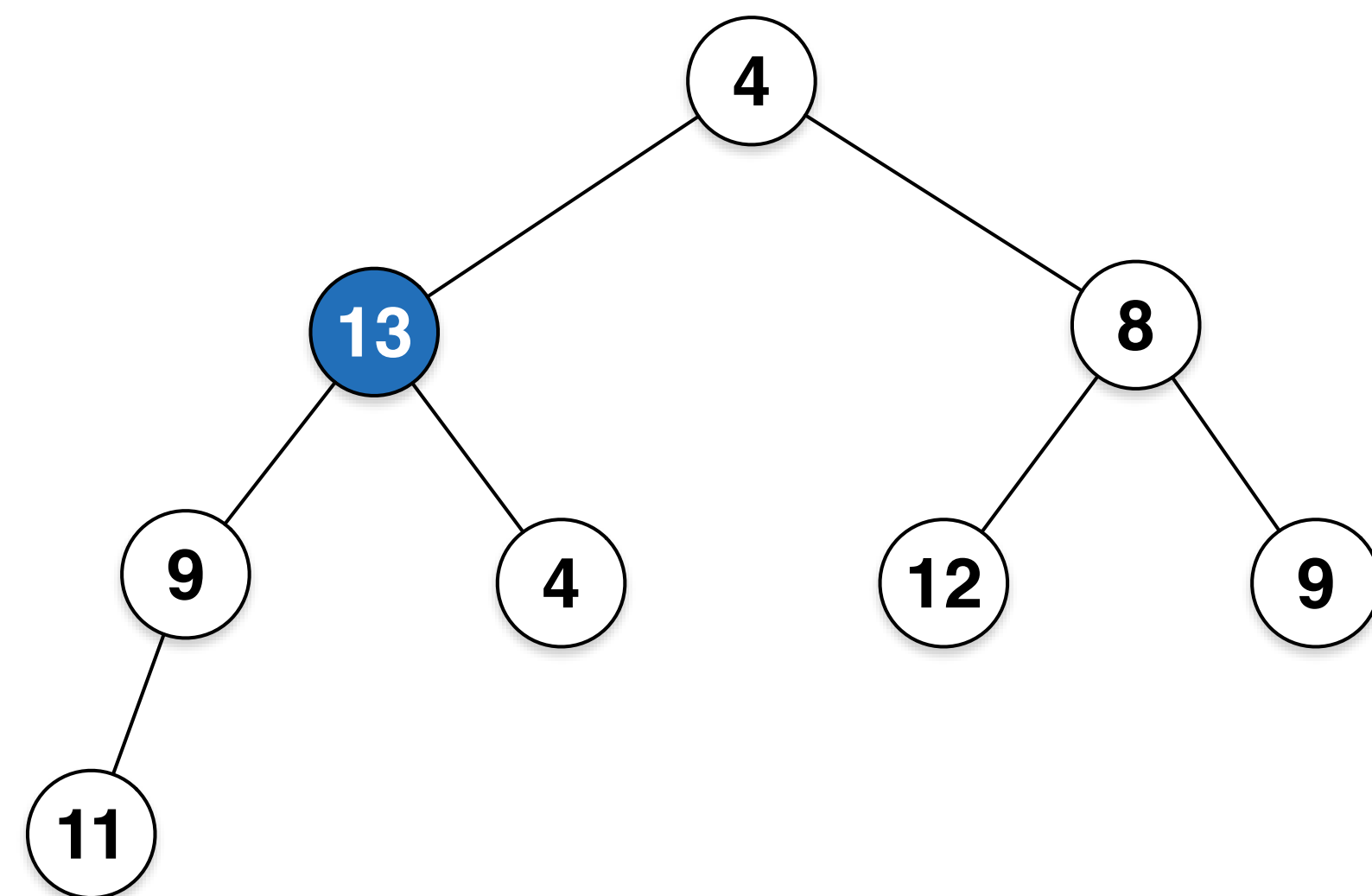
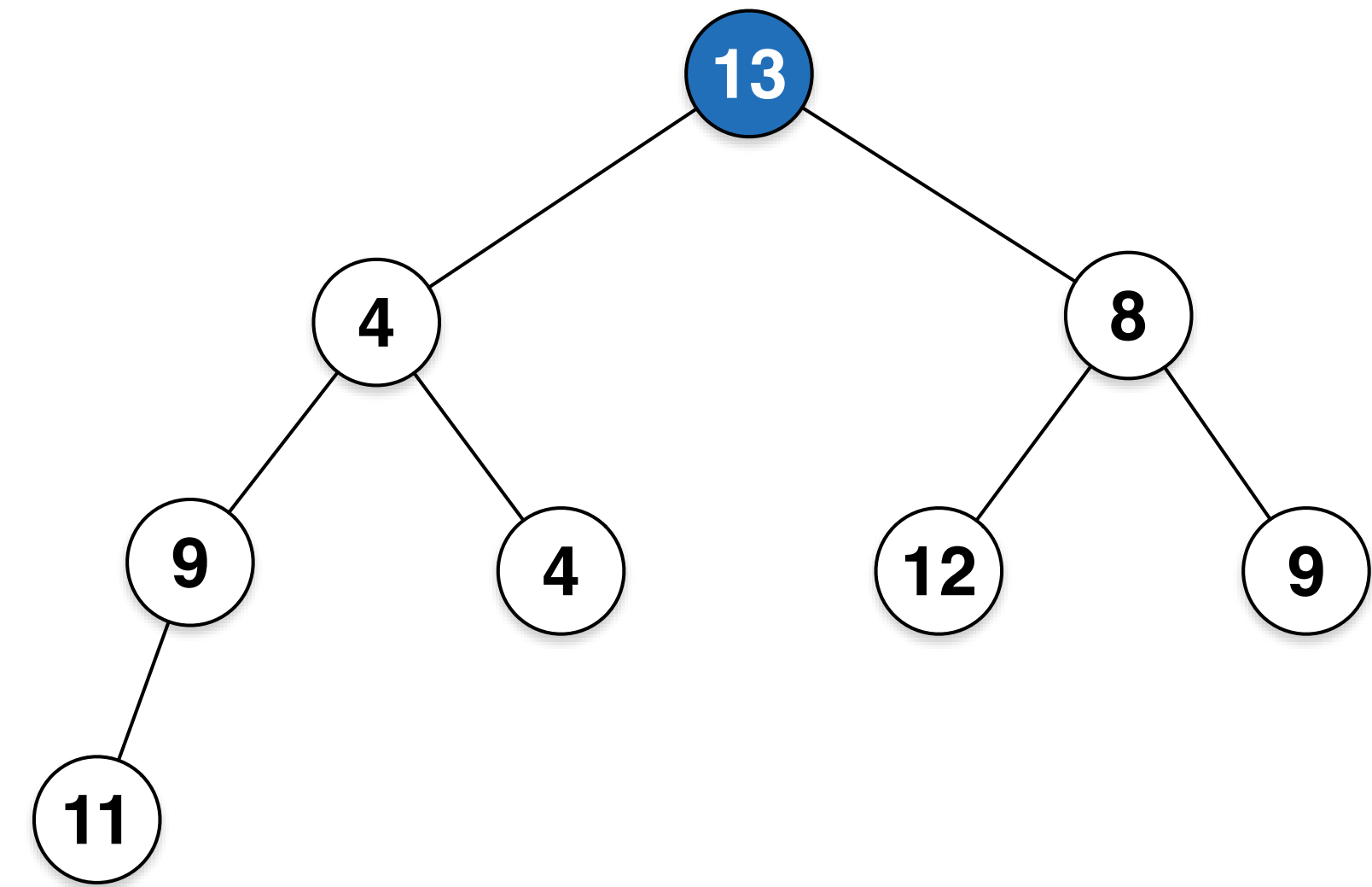
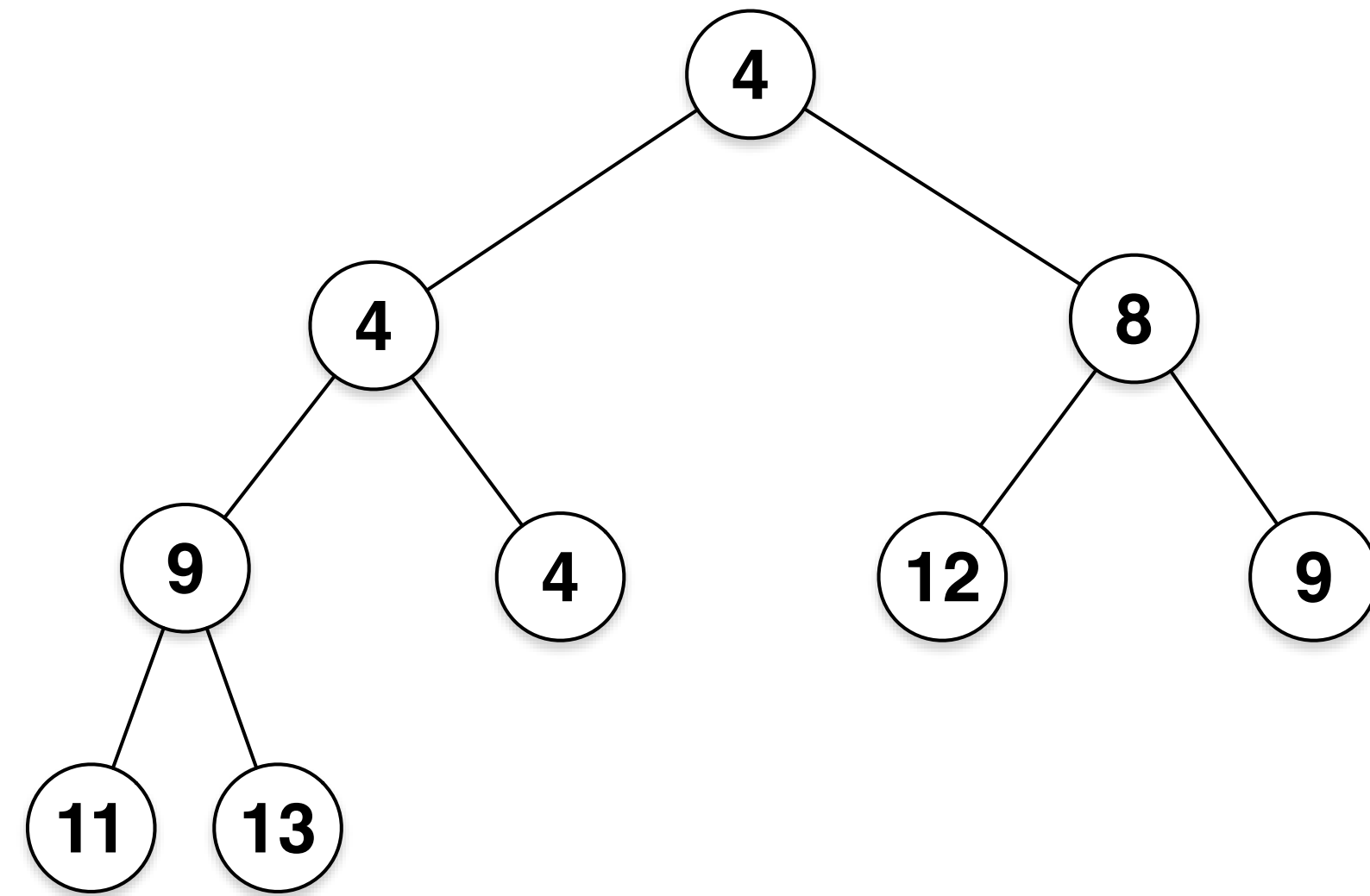
Extract Min

1. **ExtractMin(A):**
2. swap root with the last element
3. remove root
4. **MinHeapify(A, 1)**
5. return root

MinHeapify

```
1. MinHeapify(A, i):
2.     left = 2i
3.     right = 2i + 1
4.     min = i
5.     if (A[min] < A[left] && A[min] < A[right])
6.         return
7.     if (A[left] < A[right]):
8.         min = left
9.     else:
10.        min = right
11.    swap(A[i], A[min])
12.    MinHeapify(A, min)
```


MinHeapify



Heapify

1. **Heapify(A):**
2. Assume A is already a heap
3. start = floor($n/2$) // first root with children
4. for i = start to 1:
5. **MinHeapify(A, i)**

Heapify Running Time

- $O(\log n)$??
- If all the subtree at height h has been heapified, then heapifying the sub tree at $h+1$ level will only require bubbling down the root nodes.
 - $O(h)$ operations (swap) per node
 - Height is measured from bottom up starting at 0
- Notice how most of the heapifying happens at the bottom.

Heapify Running Time

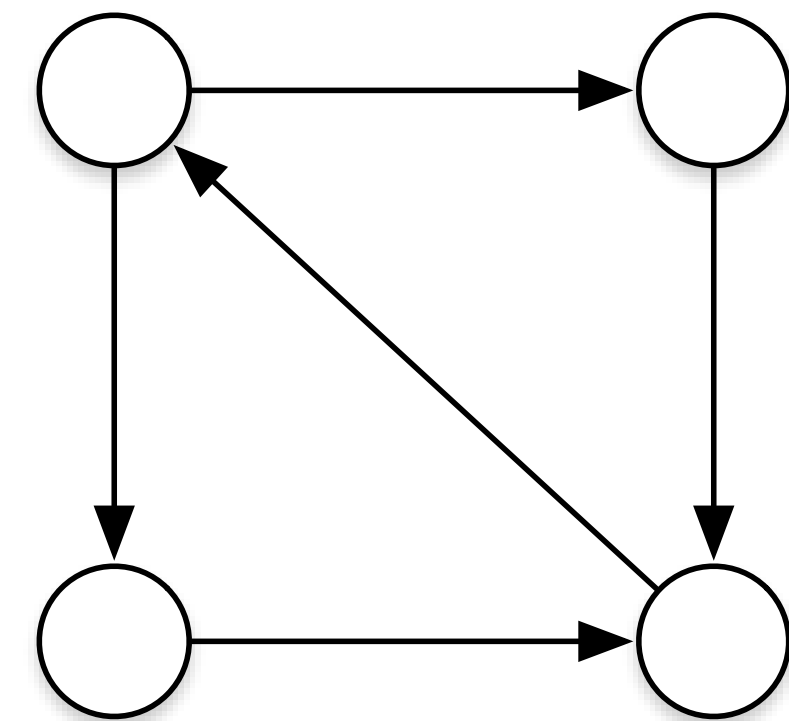
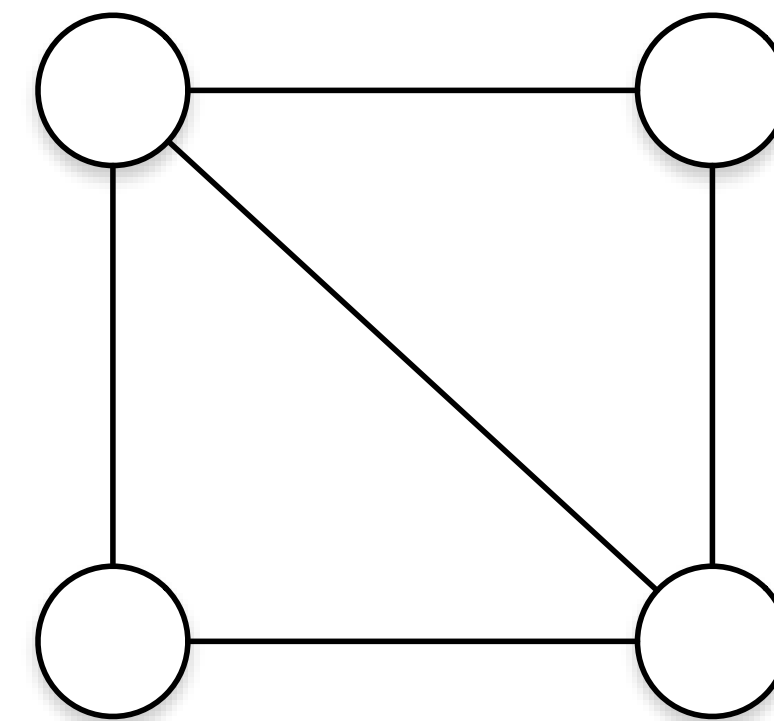
- $O(h)$ operations (swap) per node
- $\text{NodeCount}(h) = \lceil 2^{(\log_2 n - h) - 1} \rceil = \left\lceil \frac{2^{\log n}}{2^{h+1}} \right\rceil = \left\lceil \frac{n}{2^{h+1}} \right\rceil$
- Cost of heapifying the entire tree:

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

- http://www.symbolab.com/solver/series-calculator/%5Csum_%7Bn%3D0%7D%5E%7B%5Cinfty%7D%20%5Cfrac%7Bn%7D%7B2%5E%7Bn%7D%7D

Graphs

- **Vertices** (nodes) = V
- **Edges** (pairs of vertices) = E
 - Edges = directed or undirected (pair is ordered or not)
 - Parallel edges = edges that connect the same vertices
- **Degree of a vertex** = $\#$ of incident edges
- **Application:**
 - Map Application
 - Web
 - Social Networks
 - Any many more

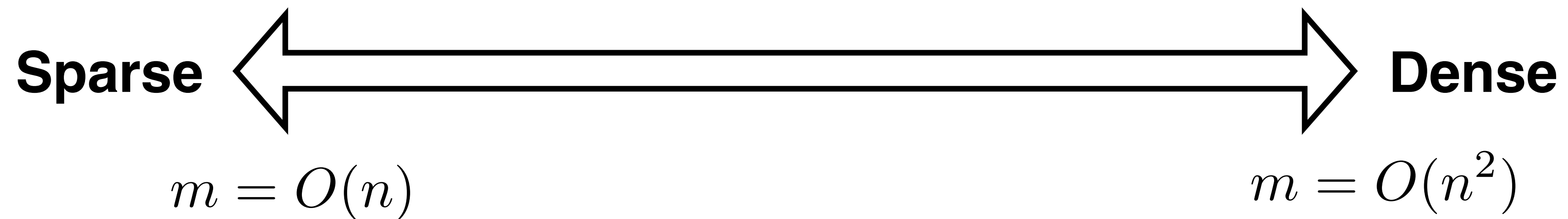


Graph Size

- Given a graph where $|V| = n$ where no parallel edges allowed
- Minimum number of edges $= n - 1$
- Maximum number of edges $= \binom{n}{2} = \frac{n^2 - n}{2}$

Sparse vs Dense Graphs

- Let $|V| = n, |E| = m$
- In most applications $m = \Omega(n)$ and $O(n^2)$



Graph Representation

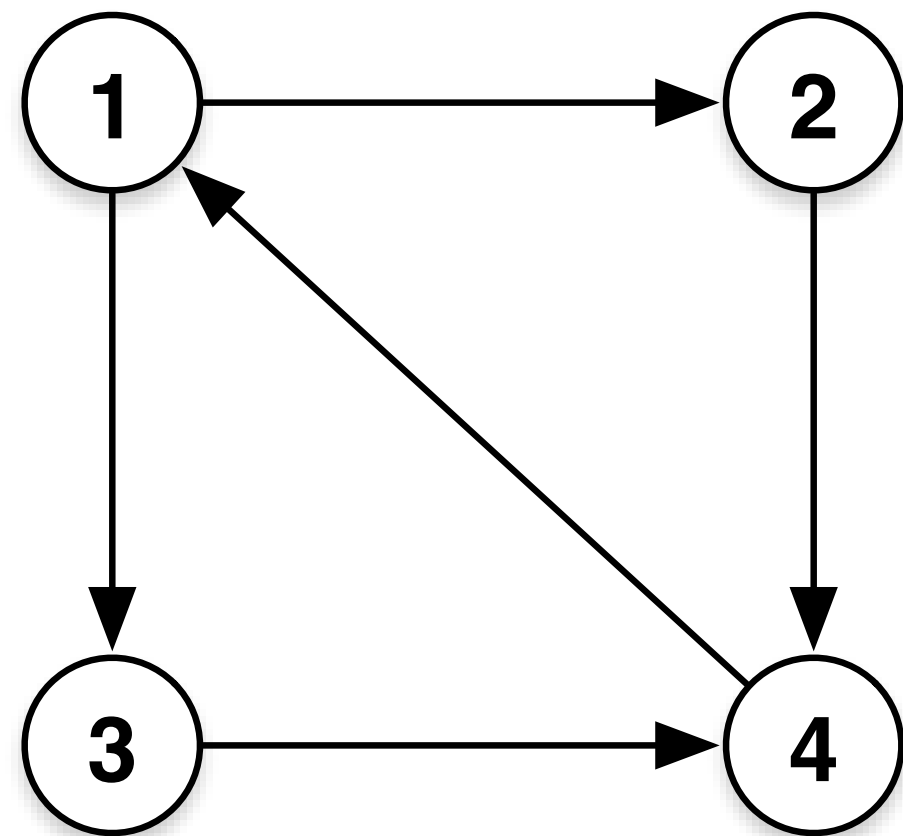
- Adjacency Matrix for undirected graph - $\theta(n^2)$

Given $G = (V, E)$, use $n \times n$ matrix, A , where

$$A_{ij} = \begin{cases} 1 & \text{if } E(v_i, v_j) \text{ exists} \\ 0 & \text{if } E(v_i, v_j) \text{ doesn't exist} \end{cases}$$

- Graph with parallel edges: $A_{ij} = \#$ of edges between v_i and v_j
- Weighted Edges: $A_{ij} = \text{weight } E(v_i, v_j)$
- Directed Graph: $A_{ij} = \begin{cases} +1 & E(v_i, v_j) \\ -1 & E(v_j, v_i) \end{cases}$

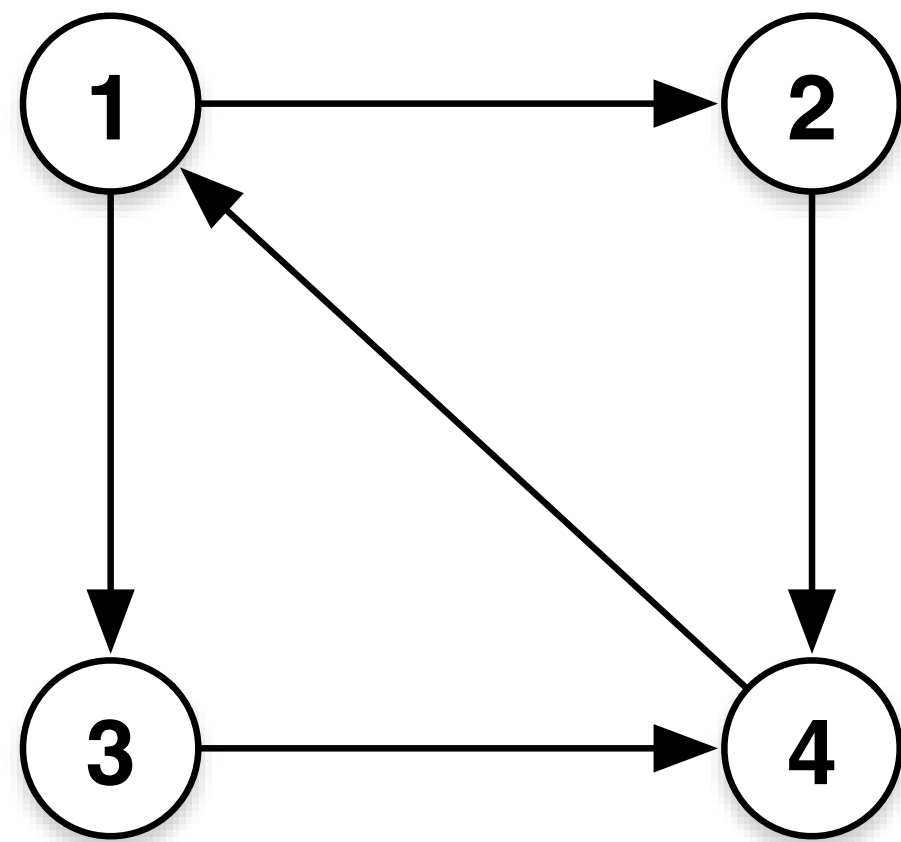
Graph Representation



i\j	1	2	3	4
1	0	+1	+1	-1
2	-1	0	0	+1
3	-1	0	0	+1
4	+1	-1	-1	0

More Graph Representation

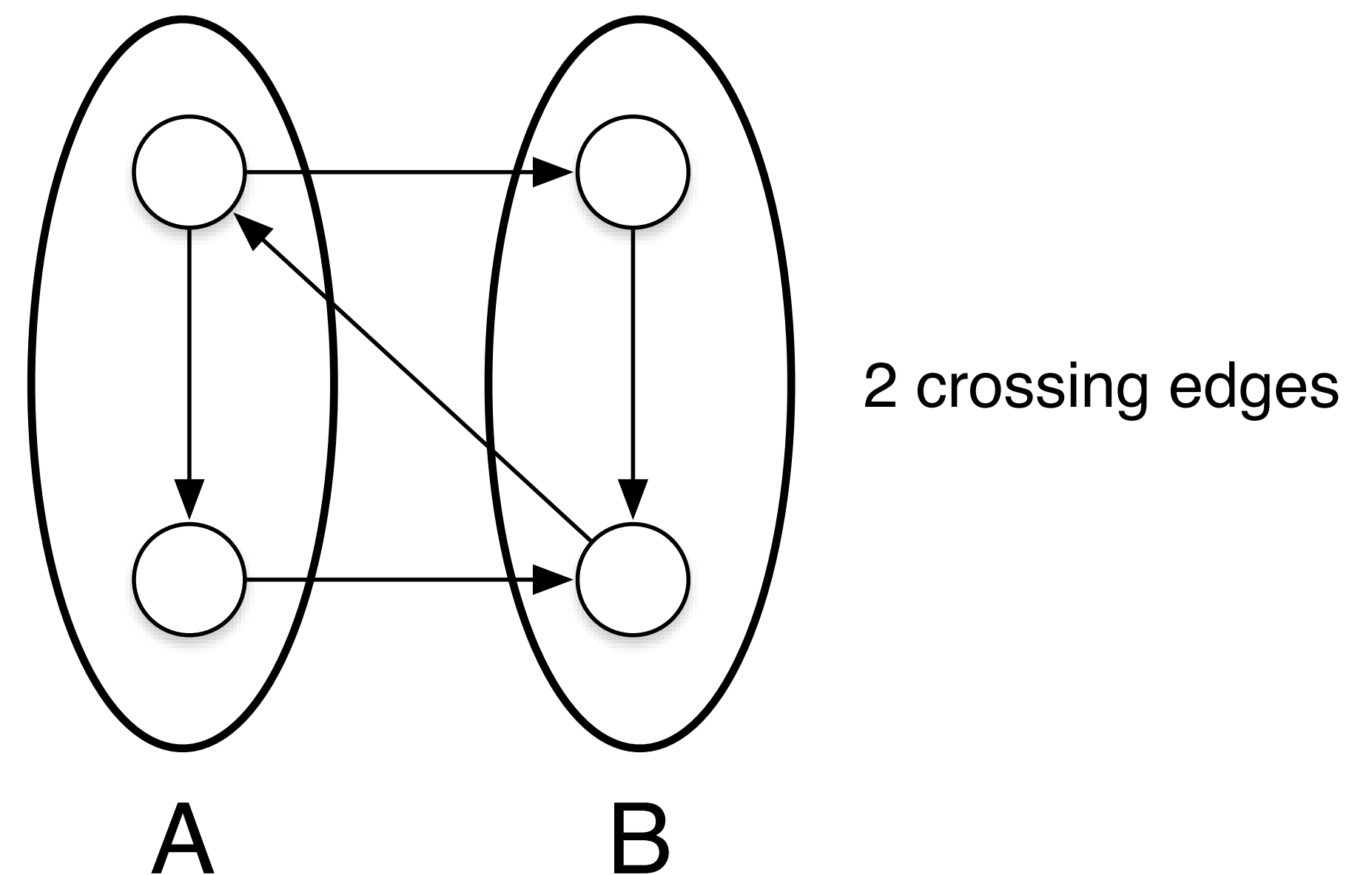
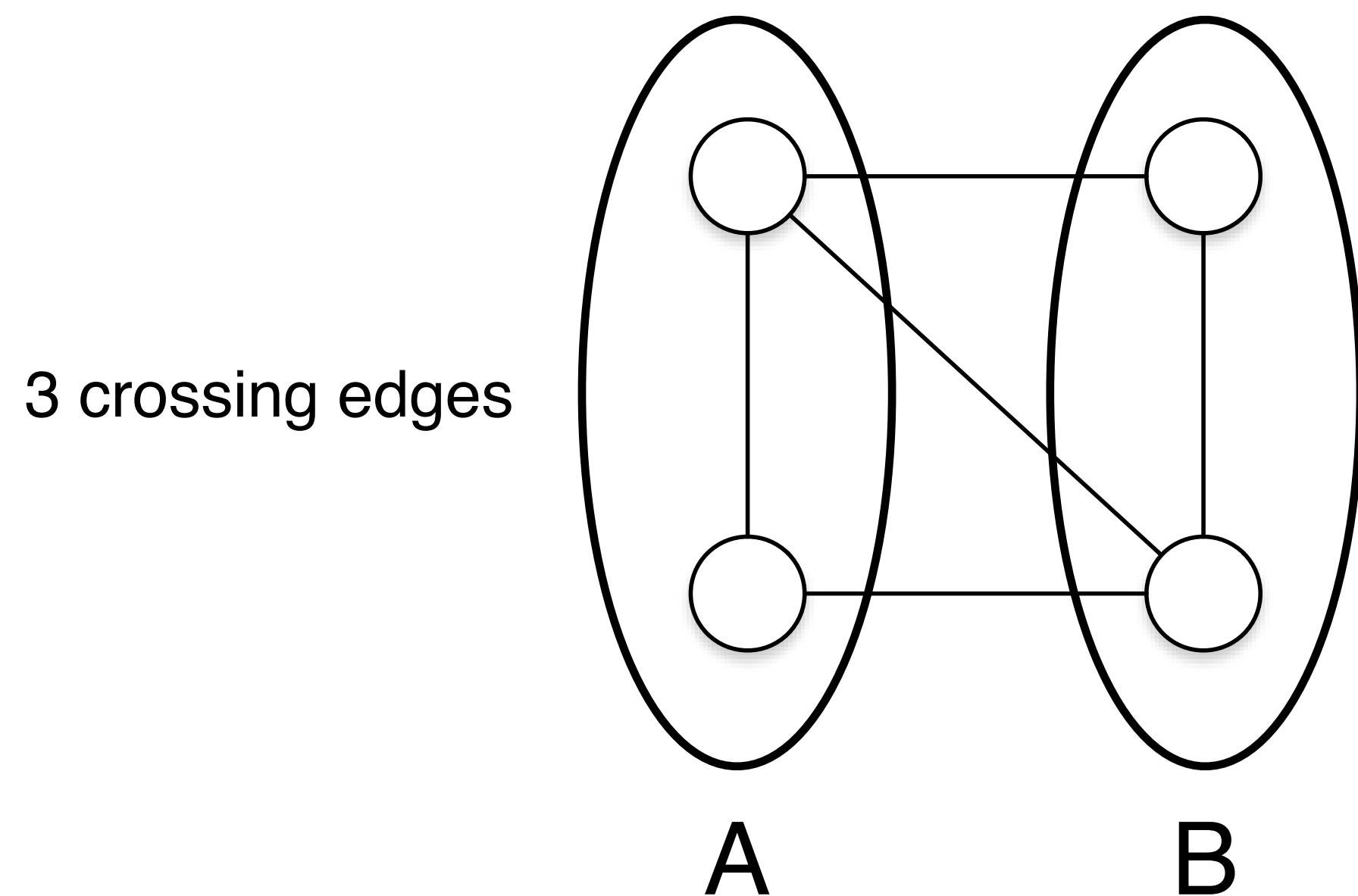
- Adjacency List
- For each of the vertices, have a list of vertices its connected to.
- Better for sparse array.



1	2	3
2	4	
3	4	
4	1	

Cut of graphs

- Given a graph $G(V,E)$, a cut of G is a partition of V into two non-empty set (A,B) .
- Crossing edges = Set of edges where endpoints are in each of (A,B)
- For directed graphs, count the edges where tail in A and head in B



Cut of a Graph

- Given a $G(V,E)$, where $|V| = n$, how many possible cuts does G have?
- $|\text{Set of all possible cuts}| = 2^n - 2$

MinCut Problem

- Given $G(V, E)$,
 - find a cut with fewest number of crossing edges. (a mincut)
- Brute force algorithm?

MinCut Application

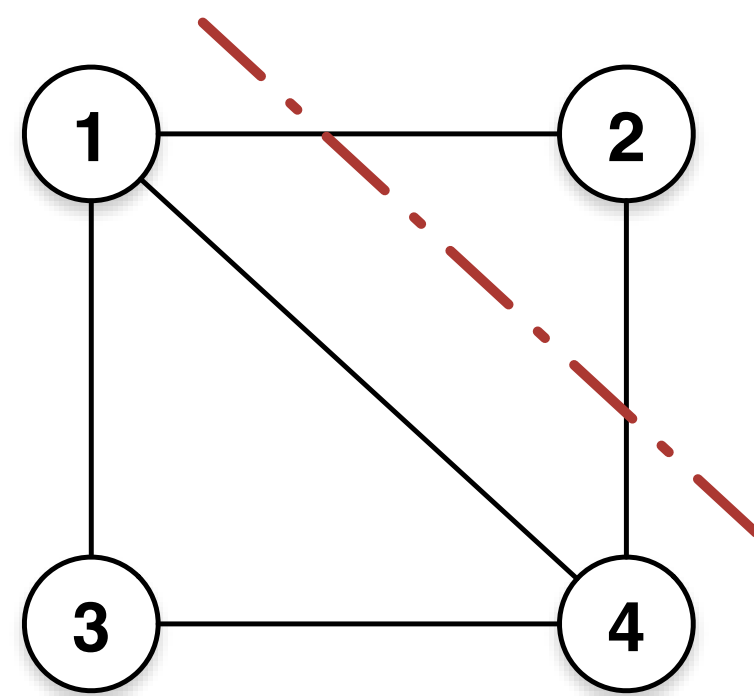
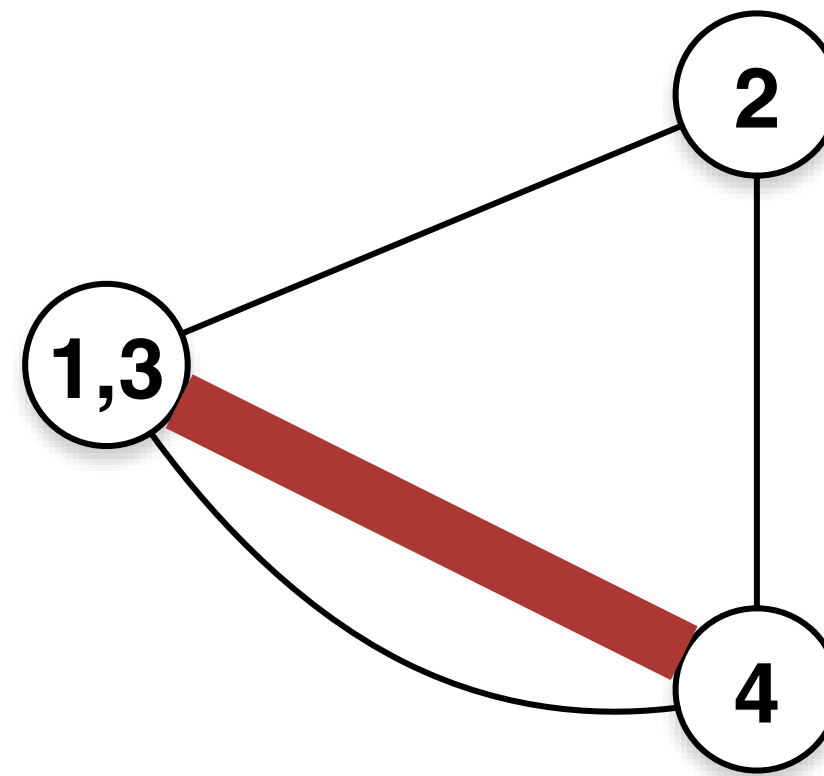
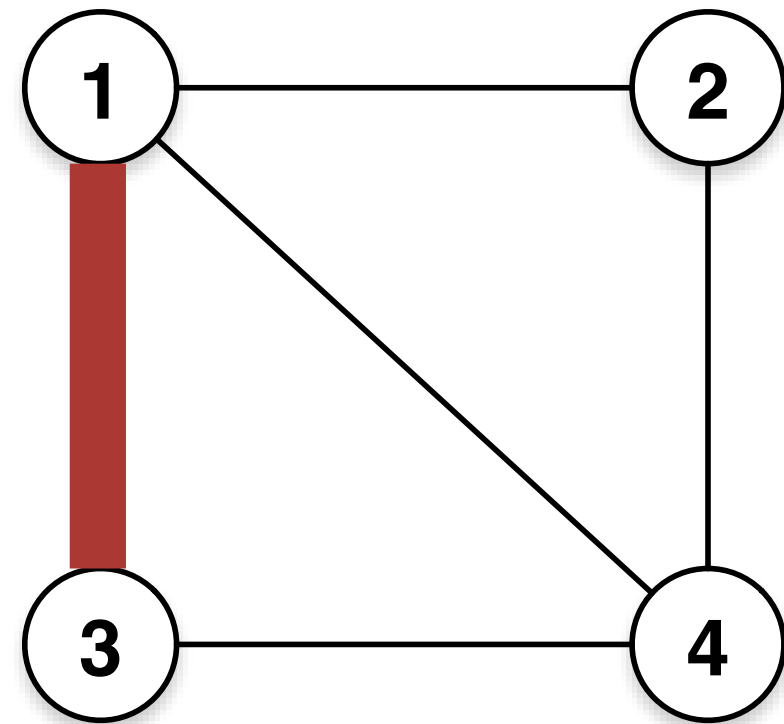
- Finding the weak spot in network (bottleneck)
- Detecting community in a social network
- Image segmentation
 - Graph of pixels
 - Neighboring pixels have weighted edges if two pixels come from a "same" object
- And many many more

Random Contraction Algorithm

1. $\text{RCA}(G)$:
2. $V = G(V)$
3. while($|V| > 2$) {
4. $e(v_i, v_j) = \text{random}(E)$
5. remove e
6. fuse (v_i, v_j)
7. }
8. return the cut represented by final 2 vertices

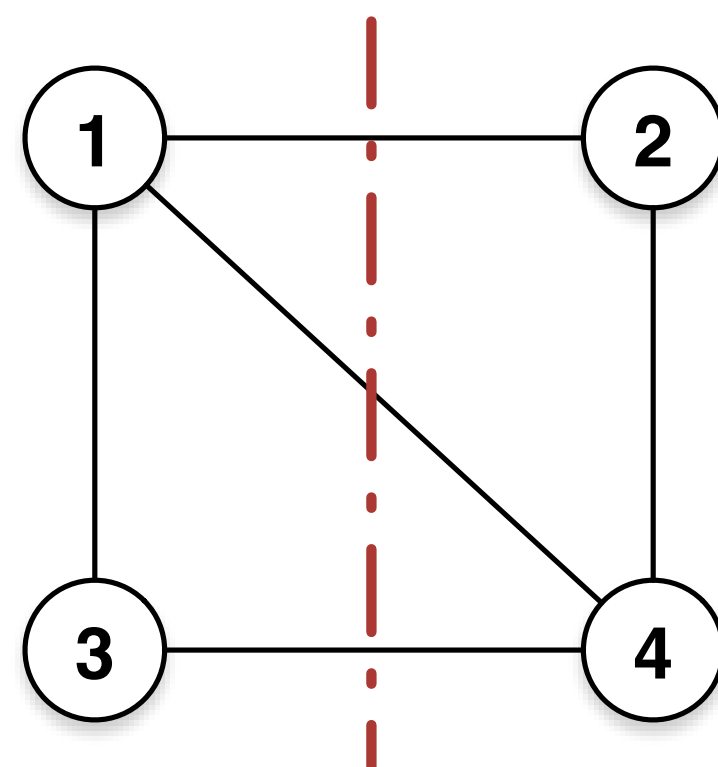
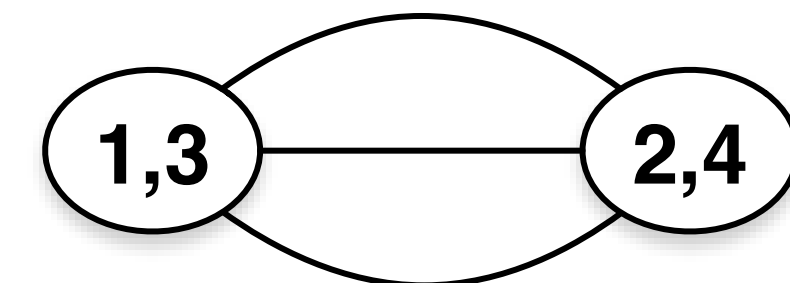
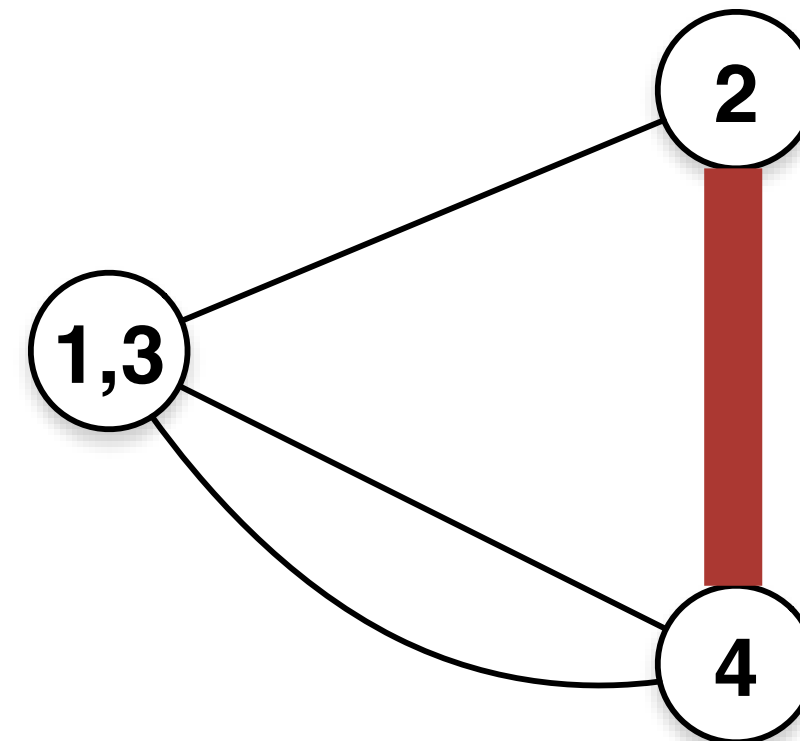
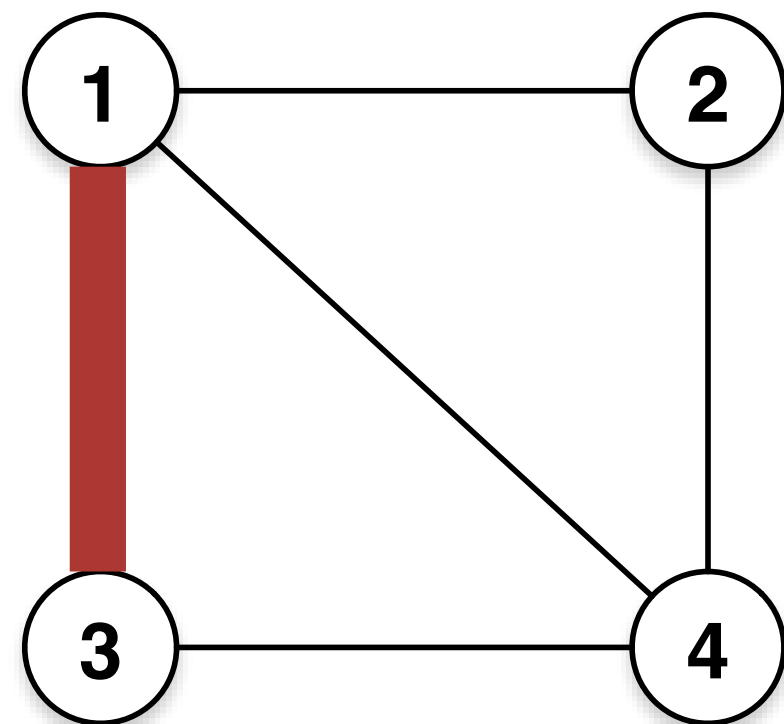
10. $\text{Fuse}(v_i, v_j)$:
11. create a brand new edge, v_{new}
12. take all the edges connecting v_i, v_j then connect them to v_{new}
13. remove any self-loops created

Example



= Cut of $(\{1,3,4\}, \{2\})$ = 2 crossing edges

Example 2



= Cut of $(\{1,3\}, \{2,4\})$ = 3 crossing edges!

Random Contraction Algorithm

- Doesn't work!
- Is it useful?
 - What is the probability that it'll create MinCut?

RCA Analysis

- Fix a graph, $G(V,E)$ with n vertices and m edges
- Fix a minimum cut (A,B)
- Let $F = \{\text{crossing edges}\}$, $|F| = k$
- If the algorithm chooses an edge from F to contract, RCA will fail.
- Converse: If RCA never fails, then none of the edges in F gets chosen.
- $P(\text{Output of RCA is } (A,B)) = P(\text{RCA never contracts an edge in } F)$

RCA Analysis Cont.

- Computing $P(\text{RCA never contracts an edge in } F) = PF$
- Let $S_i = \text{event that an edge in } F \text{ gets contracted at } i^{\text{th}} \text{ iteration}$
- $PF = P(\neg S_1 \wedge \neg S_2 \wedge \dots \wedge \neg S_{n-2})$
- $P(S_1) = k/m$
- $P(\neg S_1) = 1 - k/m$
- $P(\neg S_2 | \neg S_1) = ??$ how does m change??
- It'd be nice if we can instead track the probability in terms of n

RCA Analysis Cont.

- Note, degree of each vertex in G is at least k
 - Proof idea: Note each vertex v defines a cut $(\{v\}, V - \{v\})$
- This implies that

$$\sum_v \text{degree}(v) = 2m \geq kn$$
$$m \geq \frac{kn}{2}$$

- Finally, $P(S_i) = \frac{k}{m} \leq \frac{2}{n}$

RCA Analysis Cont.

- $P(\neg S_1 \wedge \neg S_2) = P(\neg S_2 | \neg S_1) \cdot P(\neg S_1)$
 $\geq \left(1 - \frac{k}{\# \text{ of remaining edges}}\right) \cdot \left(1 - \frac{2}{n}\right)$
- Note, contracted vertex creates a cut with a degree of at least k

$$\# \text{ of remaining edges} \geq \frac{k(n-1)}{2}$$

$$P(\neg S_1 \wedge \neg S_2) \geq \left(1 - \frac{2}{n-1}\right) \cdot \left(1 - \frac{2}{n}\right)$$

RCA Analysis Cont.

$$\begin{aligned} P(\neg S_1 \wedge \neg S_2 \wedge \neg S_3 \wedge \dots \wedge S_{n-2}) &= \\ P(\neg S_1) \cdot P(\neg S_2 | \neg S_1) \cdot P(\neg S_3 | \neg S_1 \wedge \neg S_2) \cdot \dots \cdot P(\neg S_{n-2} | \neg S_1 \wedge \dots \wedge \neg S_{n-3}) \\ &\geq \left(1 - \frac{2}{n}\right) \left(1 - \frac{2}{n-1}\right) \left(1 - \frac{2}{n-2}\right) \dots \left(1 - \frac{2}{n-(n-4)}\right) \left(1 - \frac{2}{n-(n-3)}\right) \\ &= \left(\frac{n-2}{n}\right) \cdot \left(\frac{n-3}{n-1}\right) \cdot \left(\frac{n-4}{n-2}\right) \cdot \dots \cdot \left(\frac{2}{4}\right) \cdot \left(\frac{1}{3}\right) \cdot = \frac{2}{n(n-1)} \geq \frac{1}{n^2} \end{aligned}$$

- Low success probability but it's not trivial!
- Blind random cut picking has a success rate of $\frac{1}{2^n}$

RCA Solution

- **Solution:** run RCA many times to increase the probability!
- How many times? Assume N trials
- Let T_i = event that the MinCut is found on i^{th} try.
- All T_i s are independent
- $$P(\text{All } N \text{ trials fail}) = P(\neg T_1 \wedge \neg T_2 \wedge \dots \wedge \neg T_N)$$
$$= P(\neg T_1)P(\neg T_2) \dots P(\neg T_N)$$
- Since the success probability is bounded above $1/n^2$
- Therefore, the failure probability is bounded below $1 - (1/n^2)$

RCA Solution Cont.

- $$\begin{aligned} P(\text{All } N \text{ trials fail}) &= P(\neg T_1 \wedge \neg T_2 \wedge \dots \wedge \neg T_N) \\ &= P(\neg T_1)P(\neg T_2) \dots P(\neg T_N) \leq (1 - 1/n^2)^N \end{aligned}$$

- Note: $1 + x \leq e^x$

- $$P(\text{All } N \text{ trials fail}) \leq \left(e^{-1/n^2}\right)^N$$

- If we do n^2 trials,

$$P(\text{All } n^2 \text{ trials fail}) \leq \left(e^{-1/n^2}\right)^{n^2} = \frac{1}{e}$$

- If we do $n^2 \ln n$ trials,

$$P(\text{All } n^2 \ln n \text{ trials fail}) \leq \left(\frac{1}{e}\right)^{\ln n} = \frac{1}{n}$$

Multiple RCA Running time

- Running Time:
 - $O(\text{number of trials} \times \text{running single trial})$
 $= \Omega(n^2 \times m)$
 - Slow but much better than doing brute force
 - There are LOT of clever tricks to shave off time that gives us $O(n^2)$

Graph Search

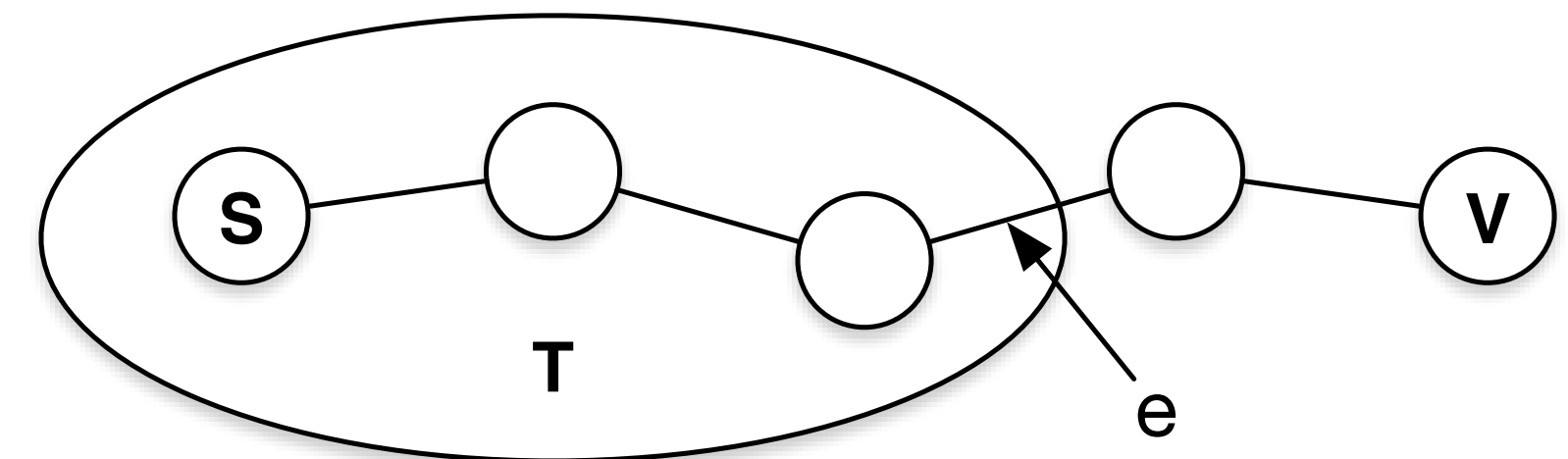
- **Goal:** Find all reachable vertices given a starting vertex
- Some definition:
 - **Connected Graph:** a graph there is a path between every pair of vertices
 - **Disconnected Graph:** a graph that is not connected

Algorithm Blueprint

```
1. Search( $G, v_s$ ):  
2.    $X = V - v_s$  // unexplored  
3.    $T = \{v_s\}$  // explored  
4.   while true:  
5.     pick an edge  $(u,v)$  such that  $u \in T$  and  $v \in X$   
6.      $T = T \cup v$   
7.      $X = X - v$   
8.     if no such edge exist halt
```

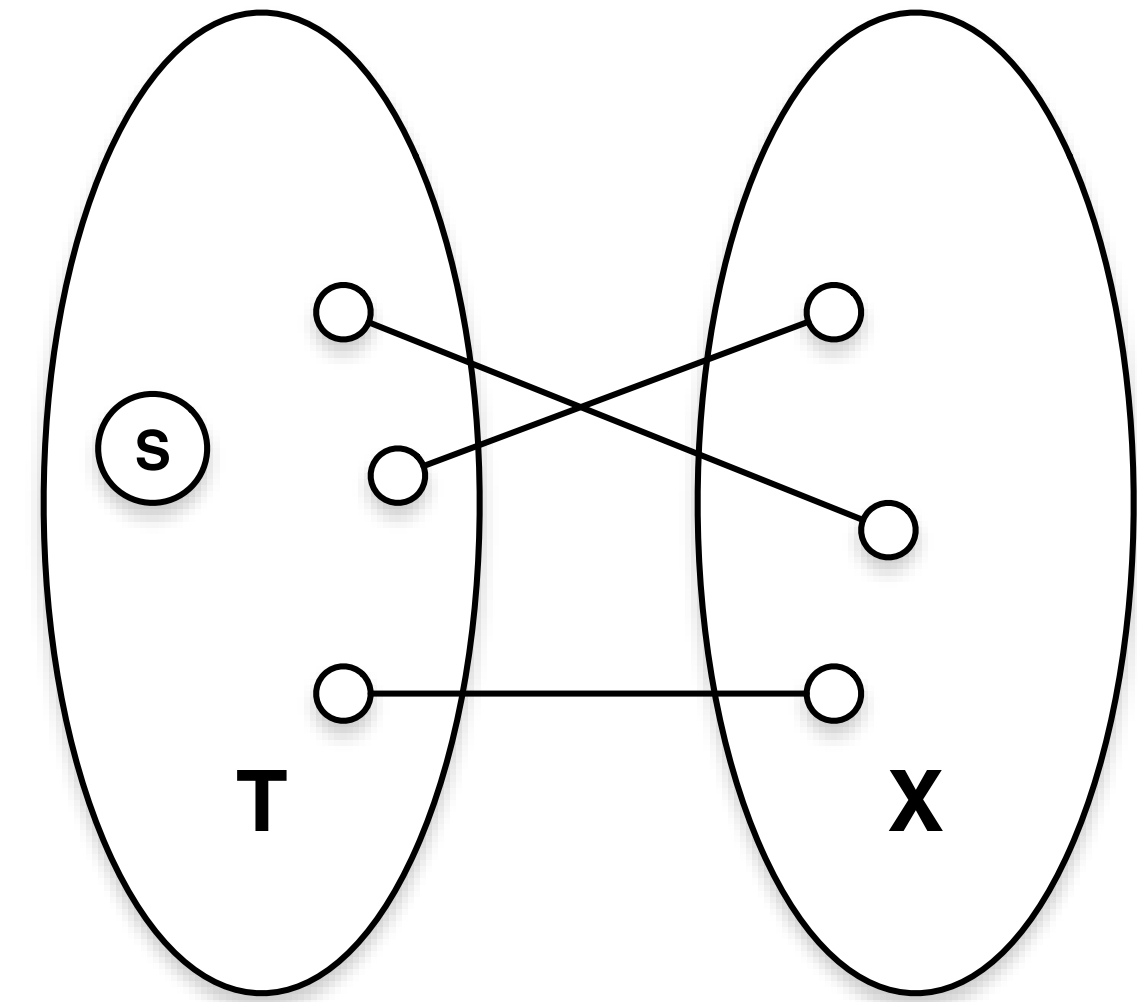
Proof of the algorithm

- **Claim:** Given the output of Search algorithm, T :
 - for all $v \in T \iff G$ has a path from s to v
- **Proof:**
 - \implies use induction (summary: in order for v to be included in T , we traversed to it)
 - \Leftarrow By contradiction. Suppose G has a path from s to v but $v \notin T$
 - But the algorithm then would have terminate without traversing into e
 - Contradiction. QED



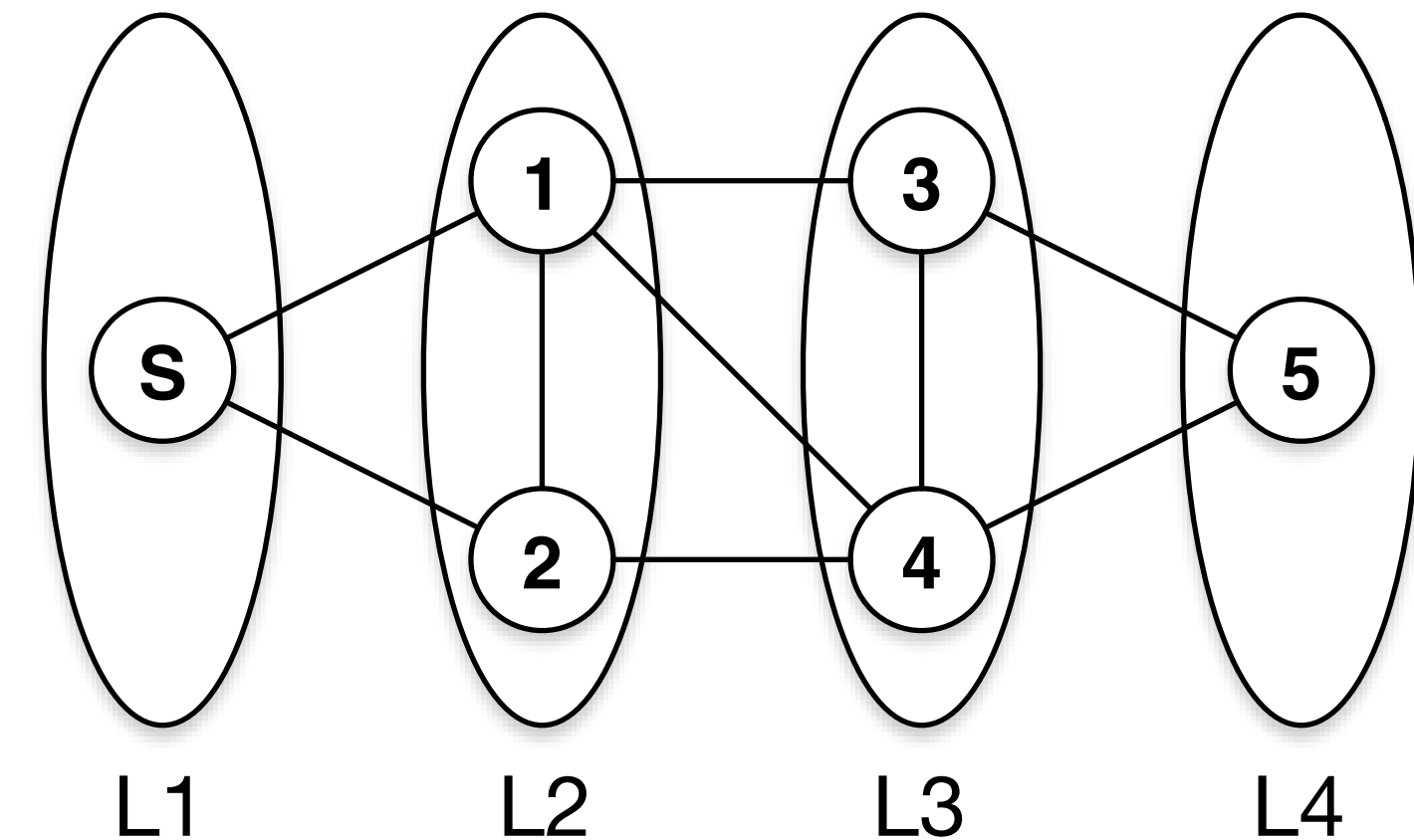
BFT vs DFT

- How do we pick the edge?
- **Breadth First Search (BFT)**
 - Search by "layer"
 - Can compute shortest path
 - Compute connected components of undirected graph
 - $O(m+n)$ using a queue
- **Depth First Search (DFT)**
 - Go deep as possible then backtrack
 - Gives you a topological ordering
 - Compute connected components of directed graph
 - $O(m+n)$ using recursion



BFT

```
1. BFS( $G, v_s$ ):  
2.    $T = \{v_s\}$  // explored  
3.    $Q = \{v_s\}$   
4.    $X = V - v_s$  // unexplored  
5.   while  $Q \neq \emptyset$ :  
6.      $v = \text{dequeue}(Q)$   
7.     for each edge  $(v, u)$ :  
8.       if  $u \in X$ :  
9.          $T = T \cup u$   
10.        enqueue( $Q, u$ )  
11.   return  $T$ 
```



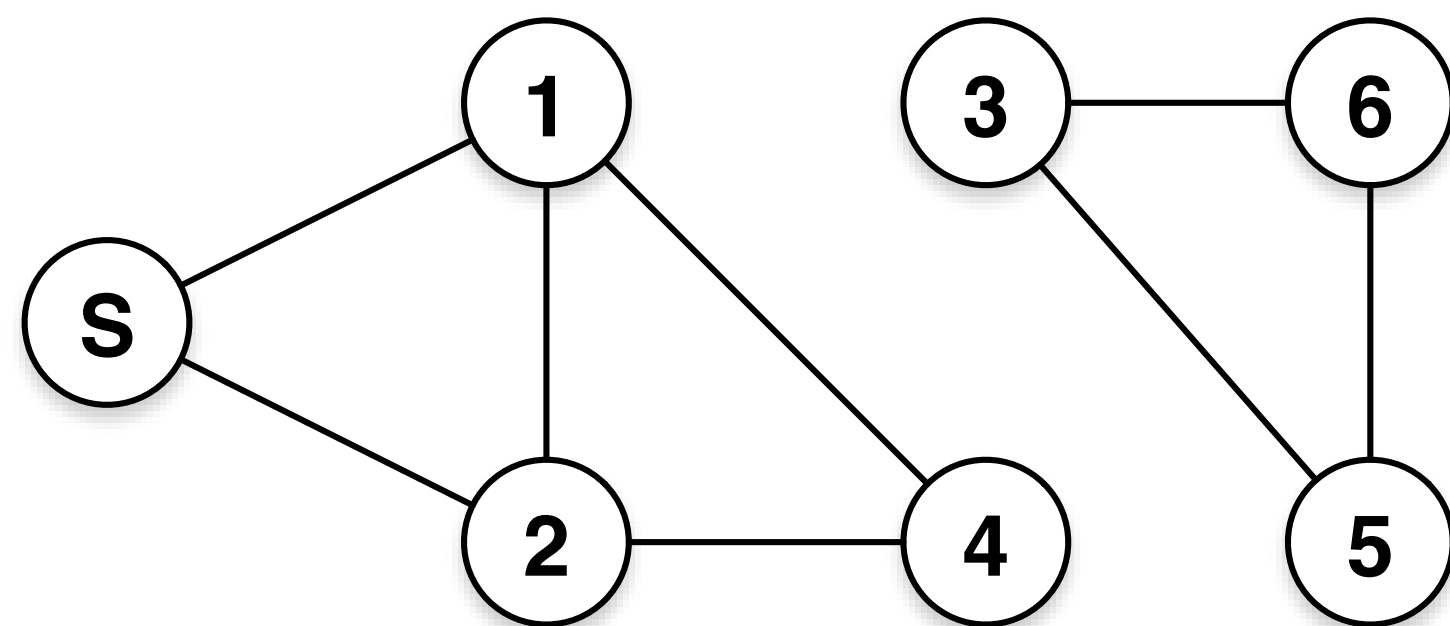
BFS Application - Shortest Path

```
1. ShortestPaths( $G, v_s$ ):
2.    $T = \{v_s\}$ 
3.    $Q = \{v_s\}$ 
4.    $X = V - v_s$ 
5.    $D[v_s] = 0$ 
6.    $D[\text{all other } v] = \text{infinity}$ 
7.   while  $Q \neq \emptyset$ :
8.      $v = \text{dequeue}(Q)$ 
9.     for each edge  $(v, u)$ :
10.      if  $u \in X$ :
11.         $T = T \cup u$ 
12.        enqueue( $Q, u$ )
13.         $D[u] = D[v] + 1$ 
14.   return  $T, D$ 
```

- Compute $\text{Distance}(v) = \text{fewest } \# \text{ of edges on a path from } s \text{ to } v$
- Add a Distance data structure to keep track of the hops
- $\text{Distance}(v) \Leftrightarrow i^{\text{th}} \text{ layer}$
- You can modify the data structure of Distance to keep track of the path

BFS Application - Connectivity

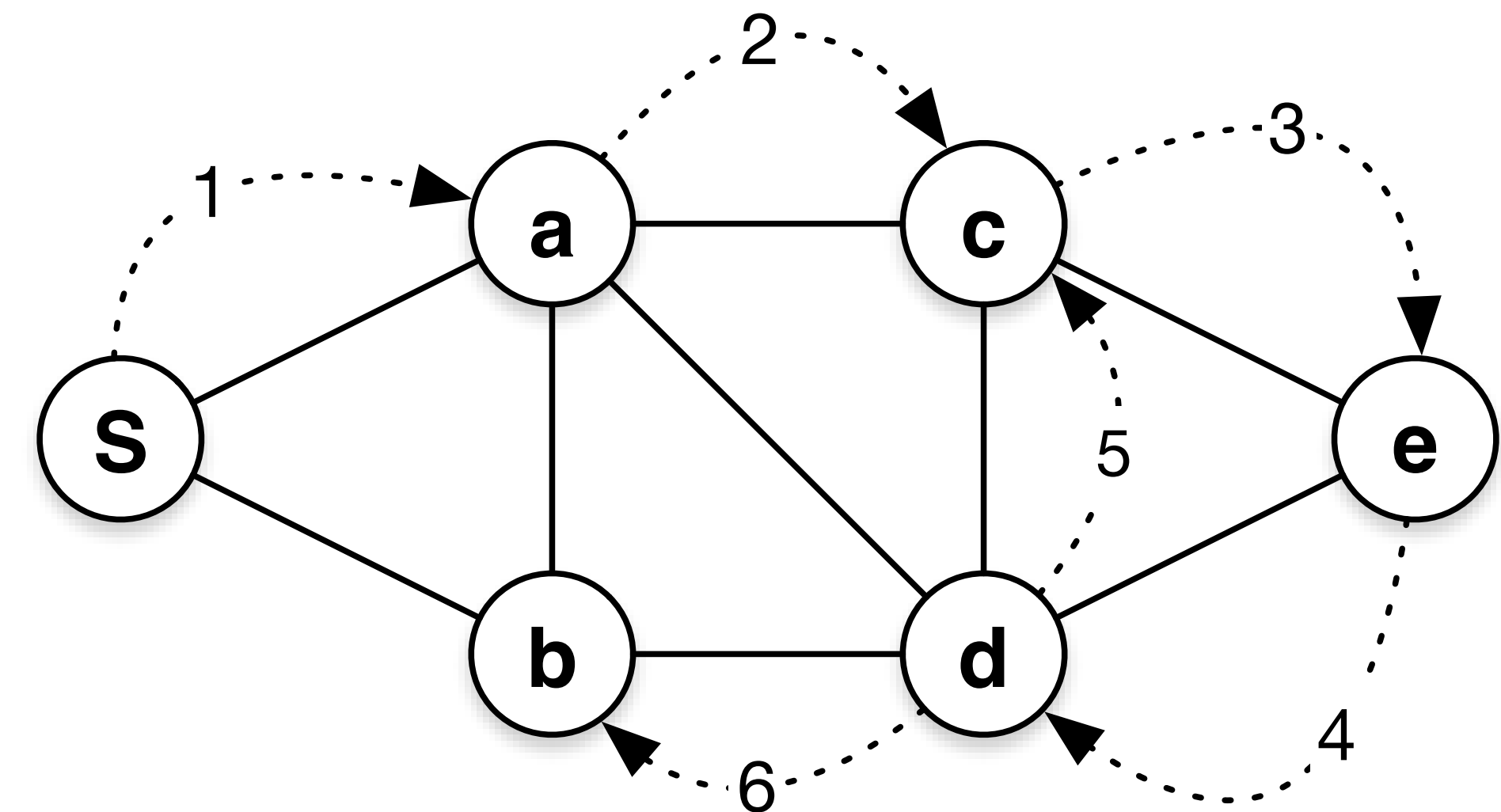
- **Problem:** Compute all connected components of a graph



```
1. FindConnected(G):
2.   X = V  // unexplored
3.   j = 1
4.   for i = 1 to n:
5.     if i ∈ V:
6.       Tj = Tj + BFS(G, i)
7.       X = X - Tj
8.   return all Tj
```

DFS

1. $\text{DFS}(G, X, v_s):$
2. $X = X + v_s$
3. for each edge $(v_s, u):$
4. if $u \notin X:$
5. $\text{DFS}(G, X, u)$
7. $\text{DFS}(G, [], v_s)$

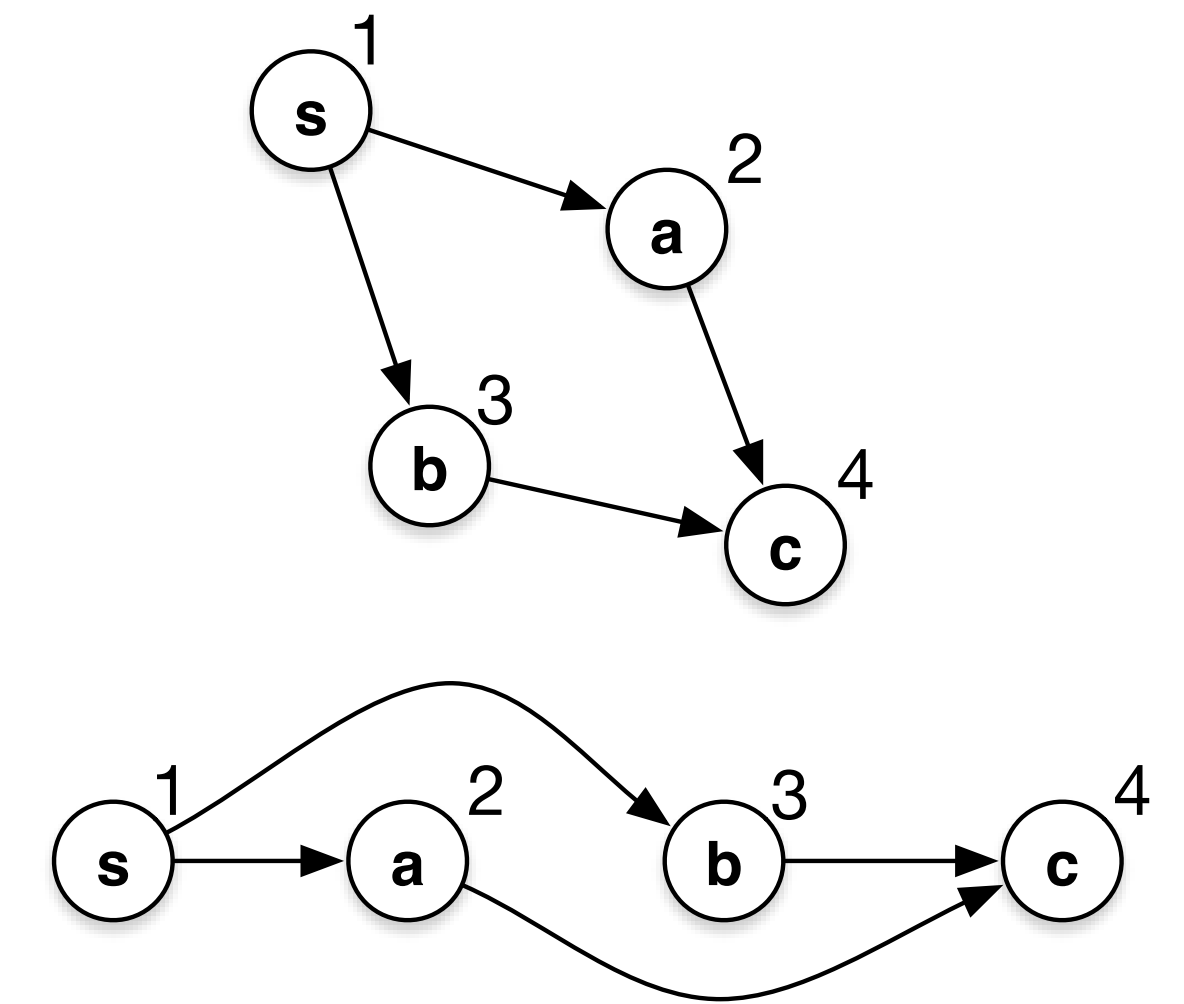


DFS Application - Topological Ordering

- **Problem:** Given a directed graph of $G(V,E)$,

$$\forall v \in G, \text{ find } f(v) \text{ where } \forall (u, v) \in G \implies f(u) < f(v)$$

- **Application:** Find all the prerequisite for a given goal/destination
- **Note:** If g has directed cycle, then no topological ordering can be found
- **Theorem:**
no directed cycle \implies topological ordering can be found



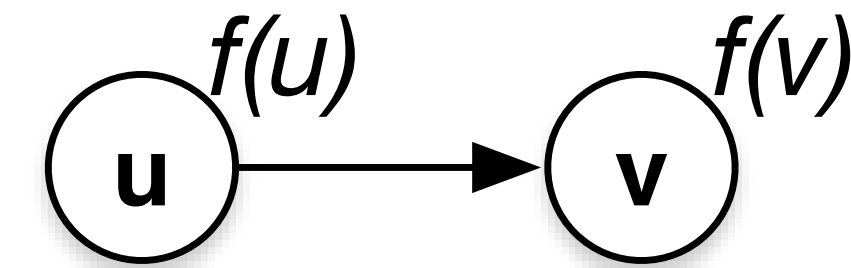
DFS Application - Topological Ordering

```
1. TopoOrder(G):  
2.   X = {}  
3.   global count = |V|  
4.   f = {}  
5.   for each v ∈ G:  
6.     if v ∉ X:  
7.       DFS'(G, X, v)
```

```
1. DFS'(G, X, vs):  
2.   X = X + vs  
3.   for each edge (vs, u):  
4.     if u ∉ X:  
5.       DFS'(G, X, u)  
6.   f(vs) = count  
7.   count = count - 1
```

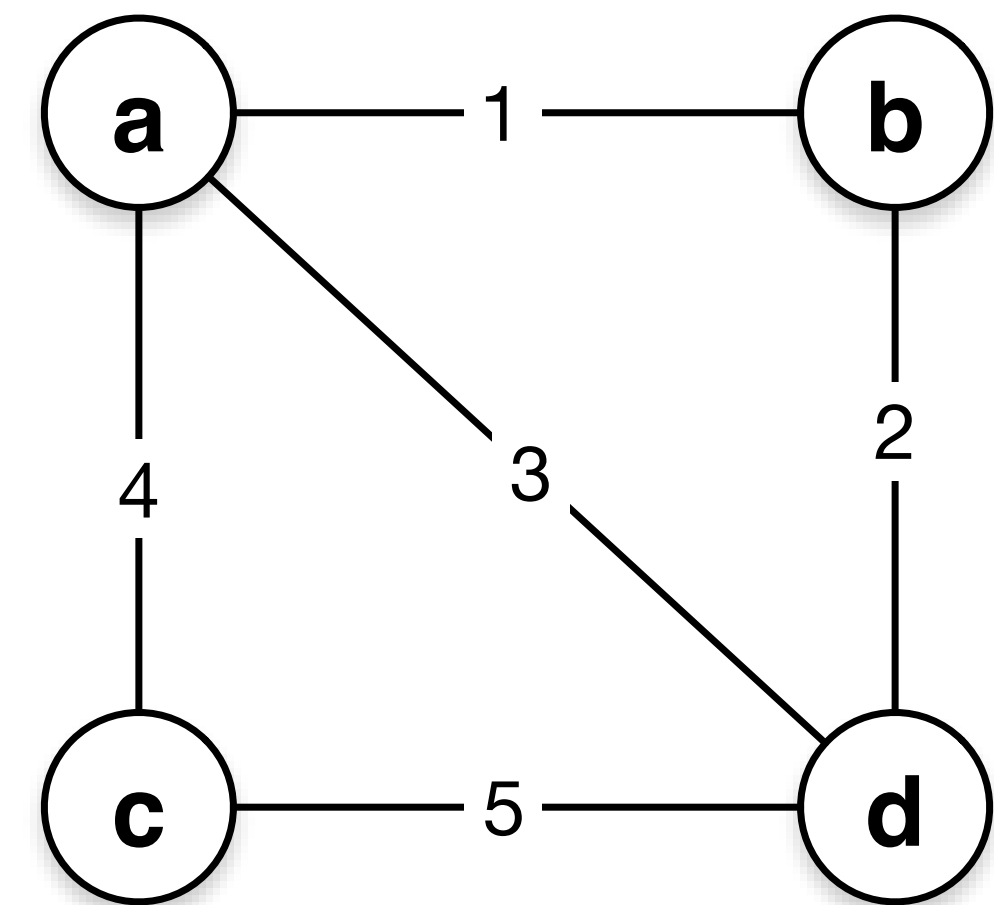
Topological Ordering Proof of Correctness

- **Claim:** The algorithm produces f values such that if (u, v) is an edge, then $f(u) < f(v)$
- **Case 1:** if u is visited by DFS before v , then
DFS(G, X, v) call finishes before DFS(G, X, u)
due to the recursive nature. Thus $f(u) < f(v)$
- **Case 2:** if v is visited by DFS before u , since there is no cycle
the call stack that caused by DFS(G, X, v) will complete before
calling DFS(G, X, u). Thus $f(u) < f(v)$



Minimum Spanning Tree

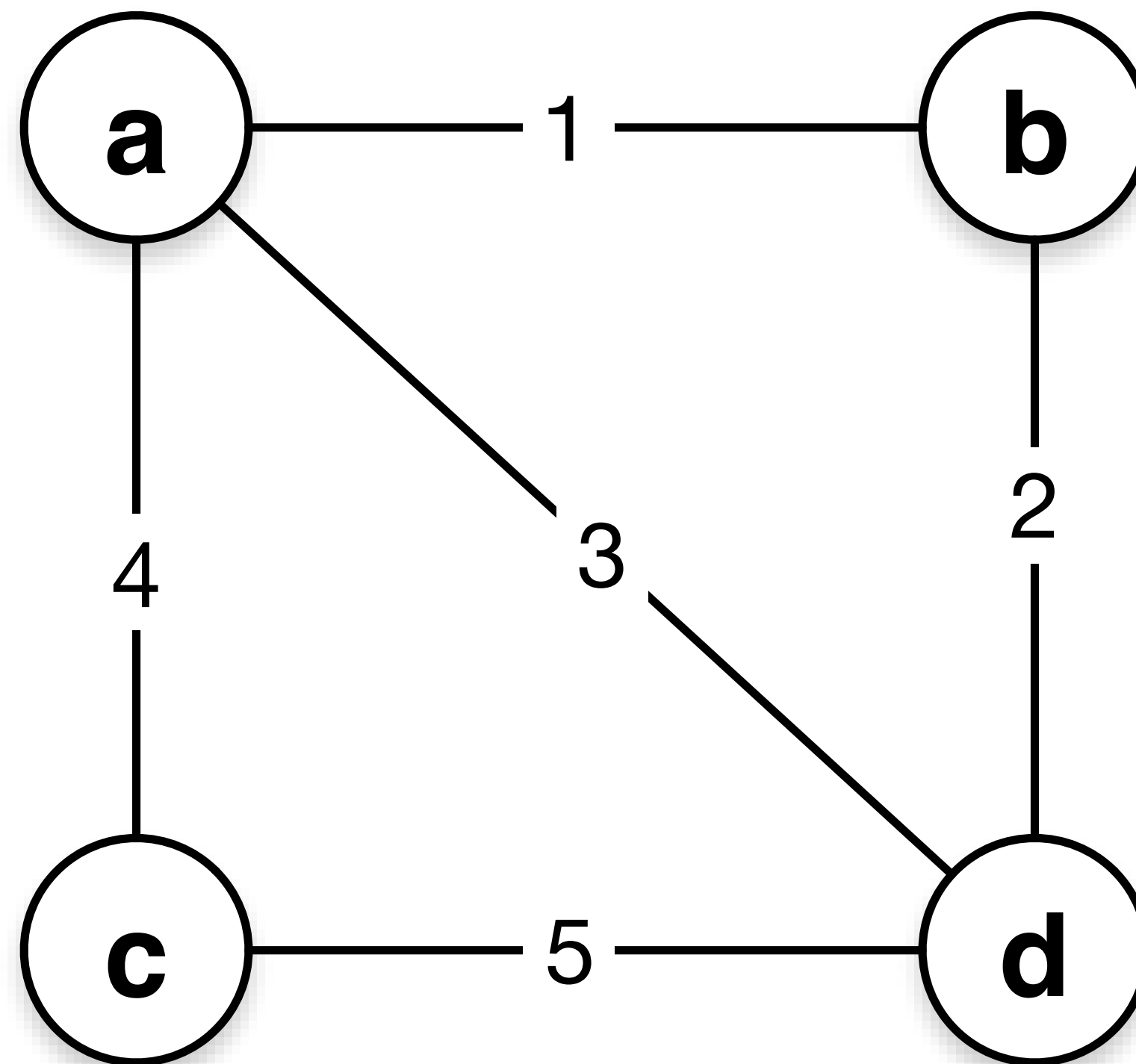
- **Problem:** Connect all vertices together deeply as possible
- **Input:** undirected graph $G=(V,E)$ and c_e for $e \in E$
- **Output:**
 - Minimum Cost Tree $T \subseteq E$ that spans V
 - No cycle
 - Connected
- **Assumptions:**
 - G is connected to begin with
 - C is unique



Prim's Algorithm

```
1. Prim(G):  
2.   X = {s}  // s is chosen arbitrarily  
3.   T = {}  
4.   while X ≠ V:  
5.       let e = (u,v)  
6.       where e is the cheapest crossing edge of cut (X, V-X)  
7.       T = T + e  
8.       X = X + v  
9.   return T
```


Prim's Algorithm



Proof of Correctness

- **Claim:** Prim's algorithm correctly computes an MST
- **Part I:** Prim's algorithm produces a spanning tree T^*
 - Spanning = all vertices are included
 - Tree = no cycles
- **Part II:** T^* is a MST
 - Minimal cost

Definitions to recall

- **Connected Graph:** a graph there is a path between every pair of vertices

Part I

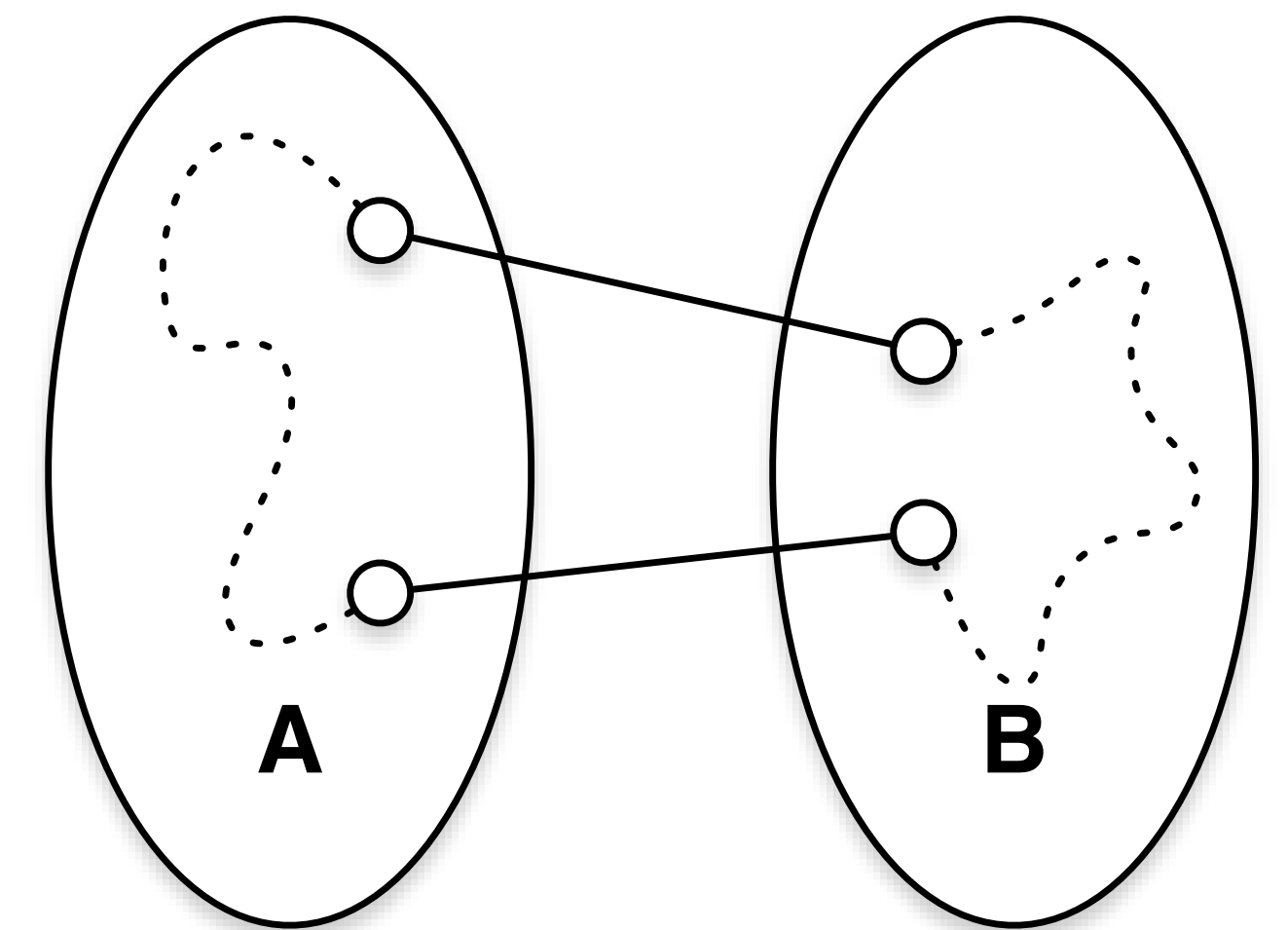
- **Empty cut lemma:**
 - Graph, G , is not connected $\iff \exists$ cut (A,B) of G with no crossing edges
- **Proof of \Leftarrow :**
 - Assume \exists cut (A,B) with no crossing edges
 - Pick any $u \in A$ and $v \in B$
 - Since there are no crossing edges given cut (A,B) , there is no path (u,v)
 - \therefore by the definition of connected graph, G is not connected

Part I cont.

- **Empty cut lemma:**
 - Graph, G , is not connected $\Leftrightarrow \exists$ cut (A,B) of G with no crossing edges
- **Proof of \Rightarrow :**
 - Assume Graph, G , is not connected
 - Pick any $u \in G$
 - Create a cut of (A, B) such that
 - $A = \{ \text{all vertices reachable from } u \}$
 - $B = \{ \text{all other vertices} \}$
 - Then cut (A, B) has no crossing edges

Part 1 cont.

- **Double Crossing Lemma:**
 - Suppose cycle, $C \subseteq E$, has an edge crossing a cut (A,B) then there must exist another edge that crosses that cut.
 - Proof by contradiction
- **Lemma 3:**
 - If e is the only edge crossing a cut (A,B) then e is not part of any cycle.
 - Proof using Double Crossing Lemma



Part 1 cont.

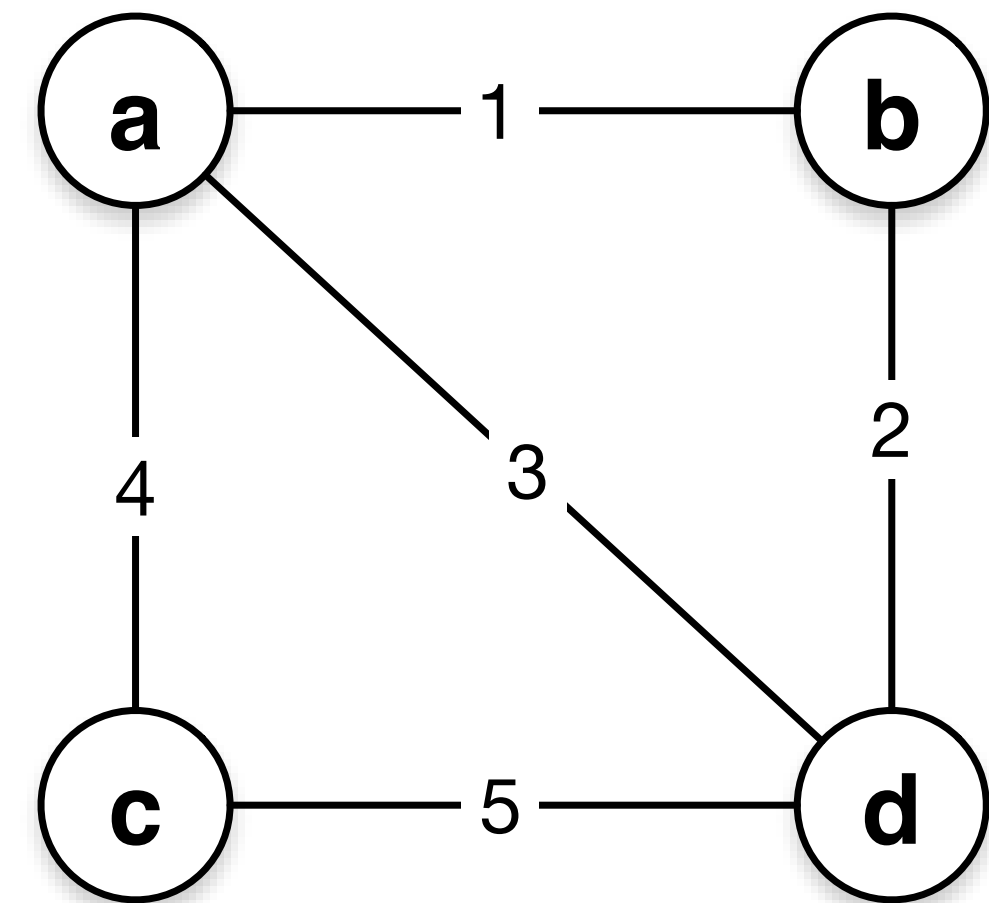
- **Part I:** Prim's algorithm produces a **spanning** tree
 - The algorithm chooses only edges stemming from X . Therefore the algorithm maintains the invariant of T spans X (meaning T includes all the vertices in X).
 - Algorithm must halt (X eventually $= V$), otherwise the cut of $(X, V - X)$ will have no crossing edges. If the cut has no crossing edges, by Empty Cut Lemma, G must be disconnected. This is a contradiction, thus the algorithm halts with $X = V$.
- \therefore Prim's algorithm produces a T that spans V

Part I cont.

- **Part I:** Prim's algorithm produces a spanning **tree** (no cycle)
 - Whenever an edge, e , gets added to T , e is the first edge to cross the cut $(X, V - X)$. By Lemma 3, e does not create a cycle
 - \therefore Prim's algorithm produces a tree

Part II

- **Part II:** T^* is a MST
 - Minimal cost
- **The Cut Property**
 - Given $e \in G$, suppose \exists cut (A,B) | e is the cheapest crossing edge, then $e \in \text{MST}(G)$



Part II

- **Claim:** Cut Property \Rightarrow Prim's Algorithm produces MST(G)
- Every edge $e \in T^*$ is chosen as the cheapest crossing edge of cut $(X, V - X)$.
- By the cut property, $T^* \subseteq \text{MST}(G)$.
- From Part I, since T^* is a spanning tree of G , $T^* = \text{MST}(G)$. QED

Proof of cut property

- **The Cut Property**
 - Given $e \in G$, suppose \exists cut (A,B) | e is the cheapest crossing edge, then $e \in \text{MST}(G)$
- By contradiction.
 - Suppose e is the cheapest crossing edge of a cut (A,B) of G , yet $e \notin \text{MST}(G)$

Proof of cut property

- Suppose e is the cheapest crossing edge of a cut (A,B) of G , yet $e \notin \text{MST}(G)$, T^*
- Since T^* doesn't include e ,
then it must include another edge, $f \mid c_f > c_e$, $f \in T^*$ and cuts (A,B)
and is part of T^* . Otherwise, T^* is not connected.
- (We want to use a swap method here, but if e and f are part of a cycle, we can't just swap.)
- Since $f \in T^*$, and T^* is a spanning tree, $T^* + e$ would create a cycle.

Proof of cut property

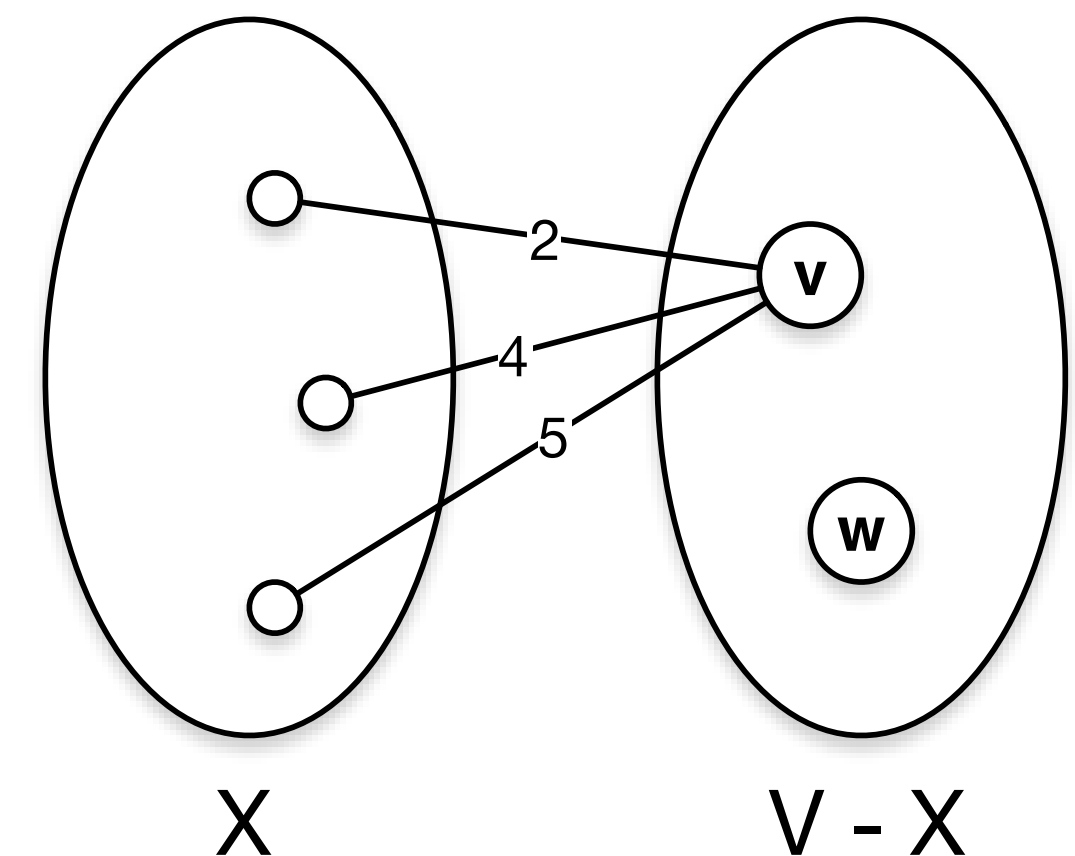
- By Double Cut Lemma, there must exist another edge, $e' \mid c_{e'} > c_e$ and creates the cycle with e
- (Now we can swap e with e')
- Note, $T = T^* + e - e'$ is a spanning tree.
- But cost of $T < T^*$ which is a contradiction. QED.

Prim's Algorithm Running Time

```
1. Prim(G):
2.   X = {s}  // s is chosen arbitrarily
3.   T = {}
4.   while X ≠ V:
5.     let e = (u,v)
6.     where e is the cheapest crossing edge of cut (X, V-X)
7.     T = T + e
8.     X = X + v
9.   return T
```

Running time of Prim's

- $O(n \times m)$ - Literal implementation
- We can use MinHeap where most of its operations are in $O(\log n)$
- Use Heap to store edges $\Rightarrow O(m \log n)$
- But faster to store vertices in Heap with following invariant
 - Invariant #1: Elements in heap = $v \in V - X$
 - Invariant #2: for $v \in V - X$:
 $\text{key}_v = \text{cheapest edge } (u, v)$

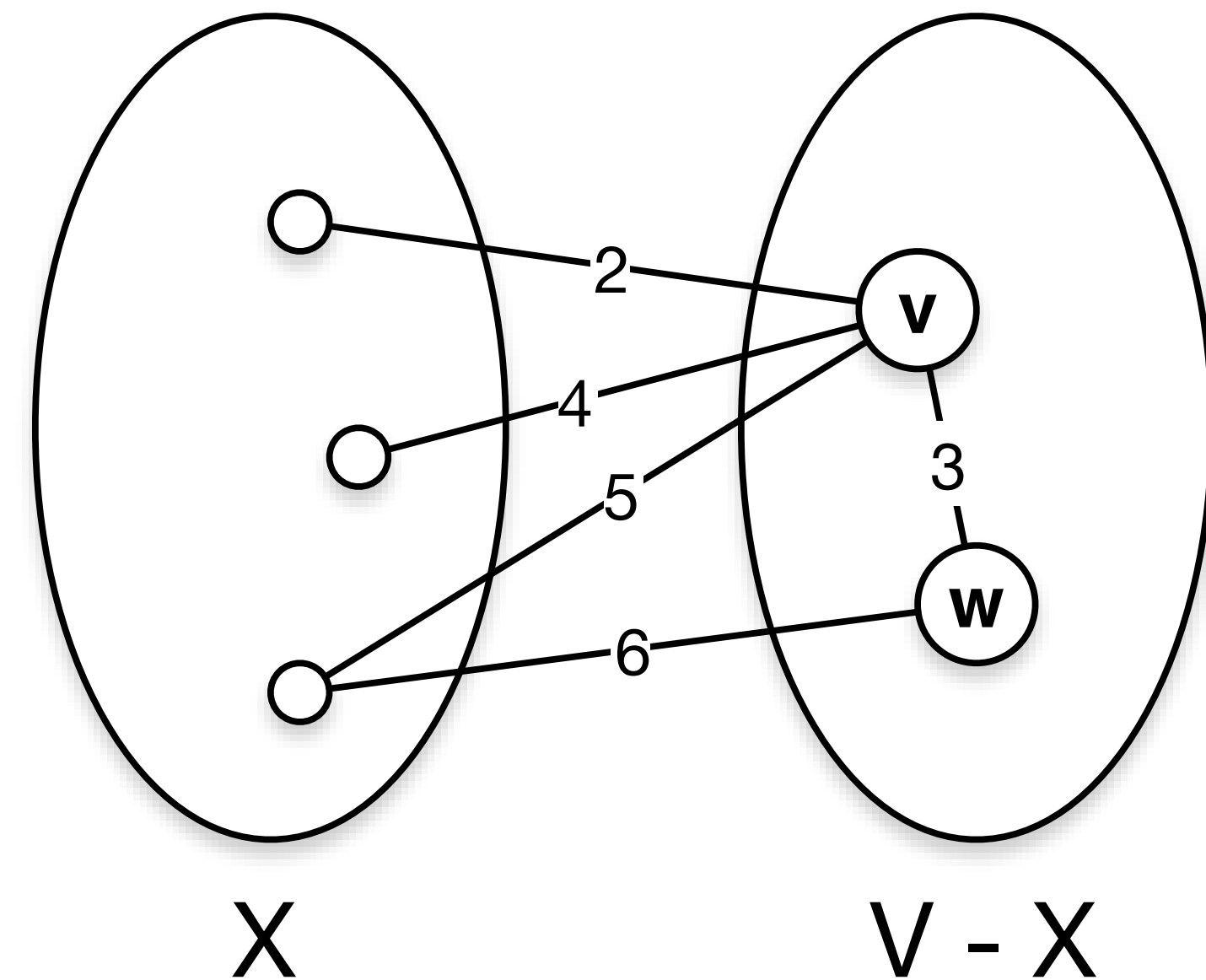


Prim's with MinHeap of Vertices

- **Preprocessing** (Initialization) of heap:
 - Initial cut = $(\{s\}, V - \{s\})$
 - Find all the edges that cross that cut and create heap:
 - $O(m + n \log n)$ or $O(m + n)$ if you use heapify
 - Since $m \geq n - 1$, $O(m + n) = O(n)$

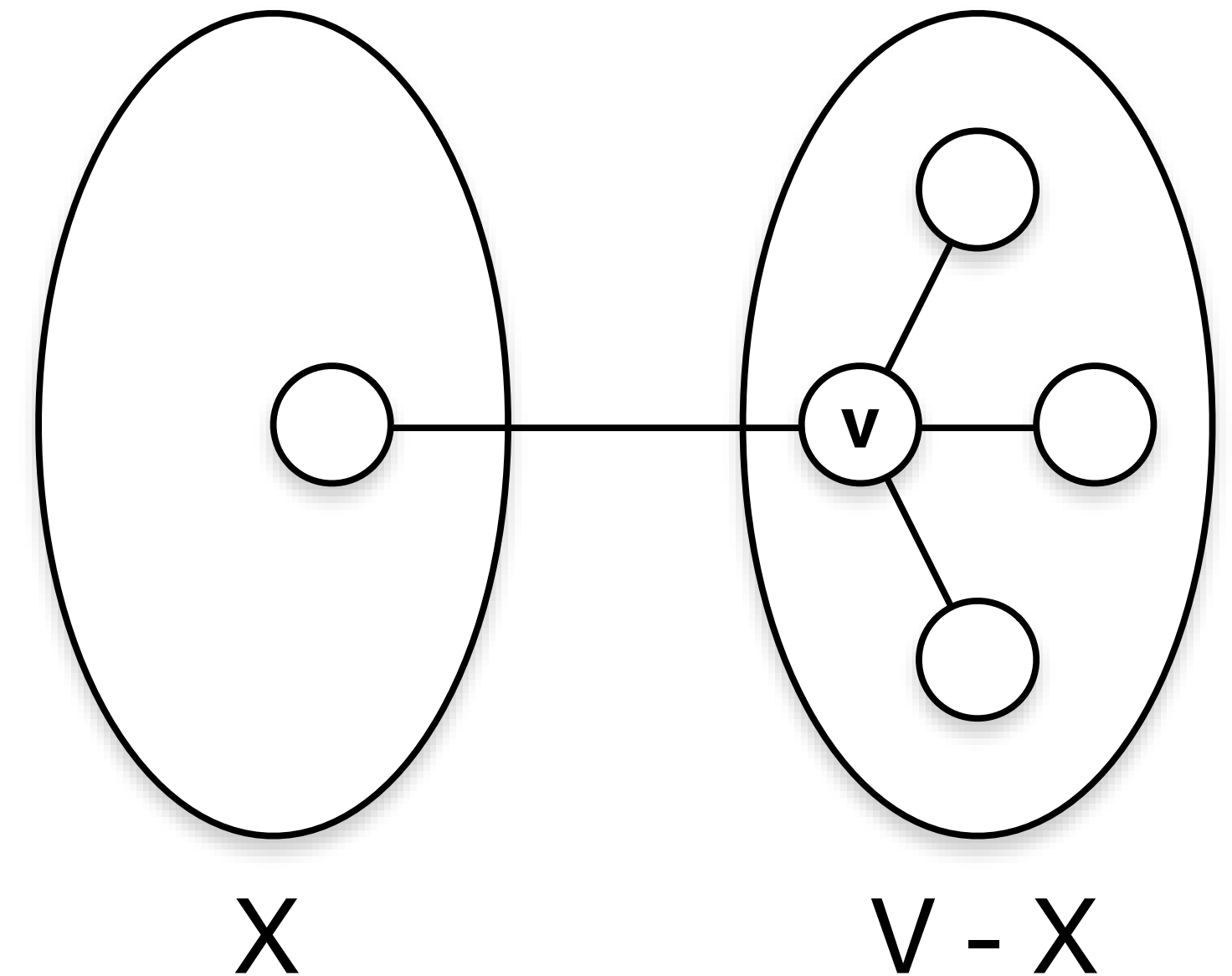
Prim's with MinHeap of Vertices

- **During execution:**
 - To pick the cheapest edge, single ExtractMin call to heap will give you the right edge.
 - $O(\log n)$
- **Keeping the invariant**
after ExtractMin has been called



Prim's with MinHeap of Vertices

- **Keeping the invariant** after ExtractMin has been called
 - Extra metadata needed to delete
1. When v is added to X :
 2. for each edge $(v, w) \in E$:
 3. if $w \in V - X$:
 4. delete w from heap
 5. $\text{key}[w] = \min\{\text{key}[w], c_{v,w}\}$
 6. insert w into heap



Final Running Time of Prim

- Preprocessing - $O(n)$
- One ExtractMin called per each vertex - $O(n \log n)$
- Each edge triggers at most one delete/insert - $O(m \log n)$
- Entire running time = $O(m \log n)$