

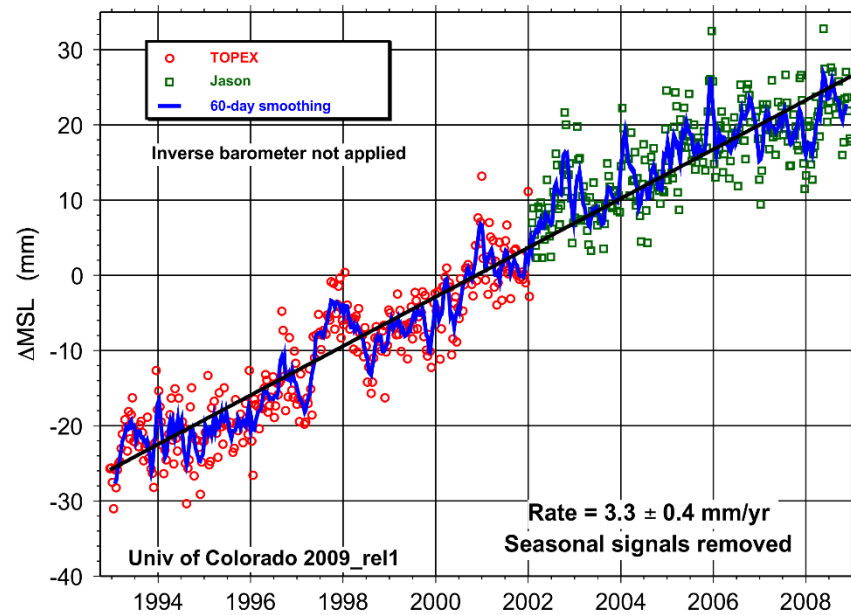
CSIT113

Problem Solving

Week 9

Graphs and Trees

- Not this...

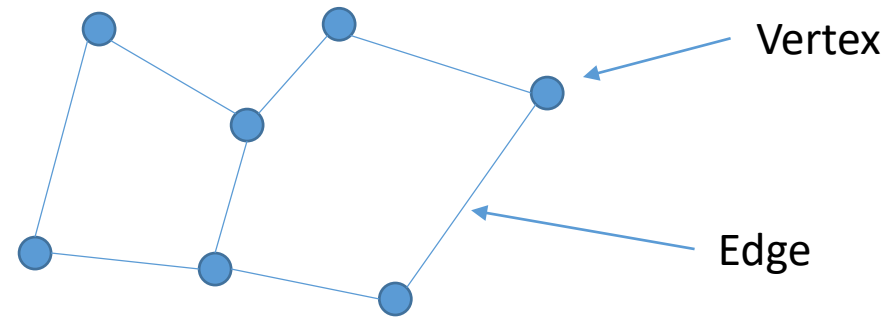


- Or this...



What then?

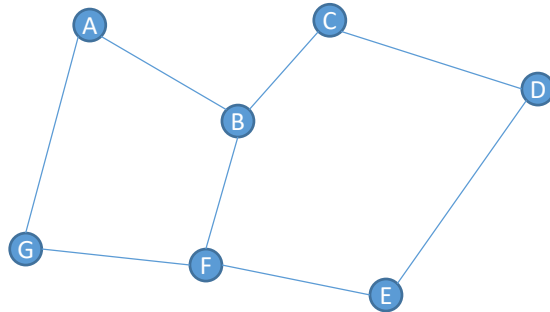
- A graph is defined as the combination of two sets, V and E .
- V is the set of vertices.
 - Points in space.
- E is the set of edges.
 - Lines connecting vertices.



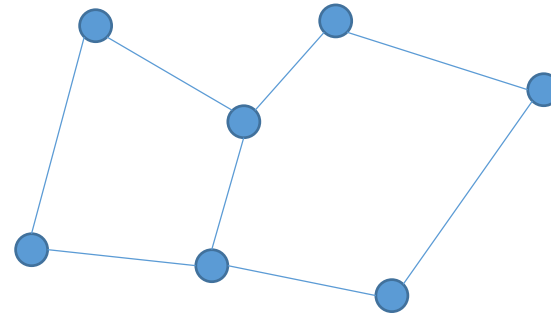
Some terminology: Labelled

- If there is a sequence label associated with each vertex we say the graph is *labelled*.

Labelled Graph



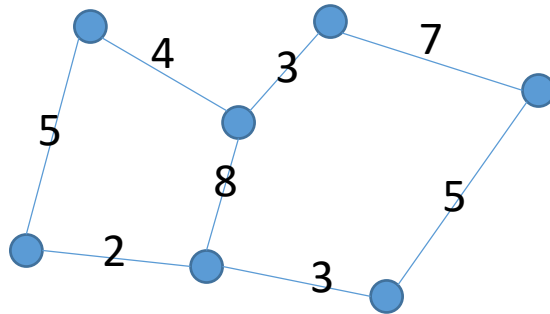
Unlabelled Graph



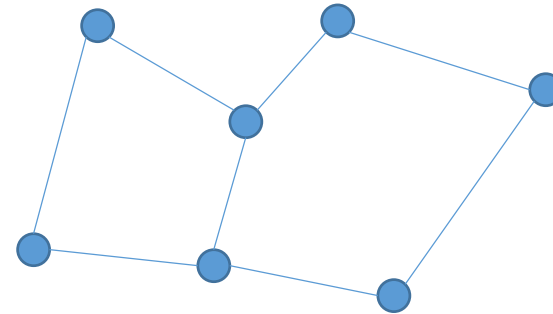
Some terminology: Weighted

- If there is a value associated with each edge we say the graph is *weighted*.

Weighted Graph



Unweighted Graph

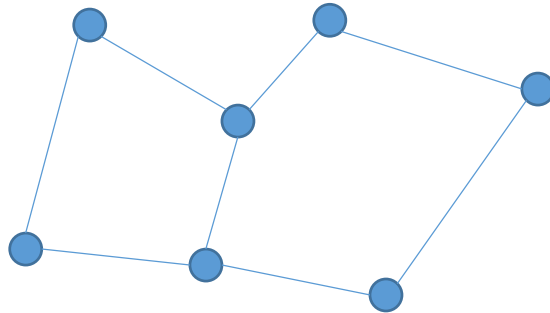


- The weight values may indicate a distance, a cost or some other property of the edge.

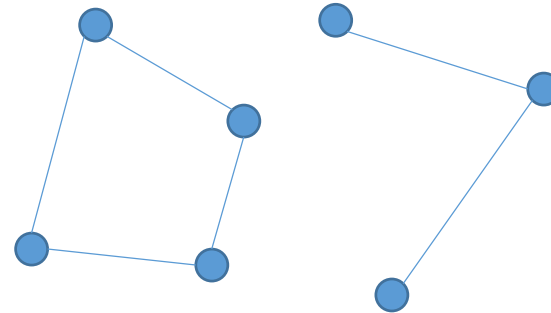
Some terminology: Connected

- If there is a sequence of edges from any vertex to any other vertex we say the graph is *connected*.

Connected Graph



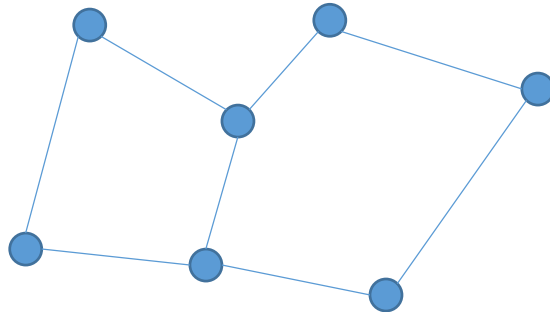
Disconnected Graph



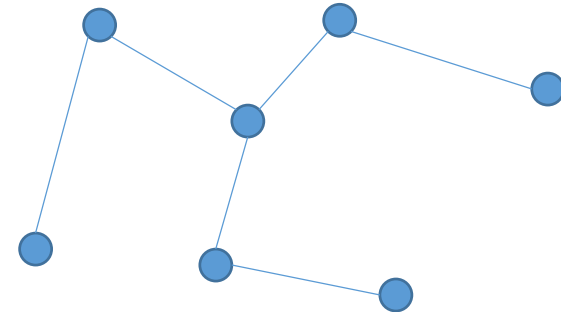
Some terminology: Cyclic

- If there is more than one path between some pair of vertices we say the graph is *cyclic*.

Cyclic Graph



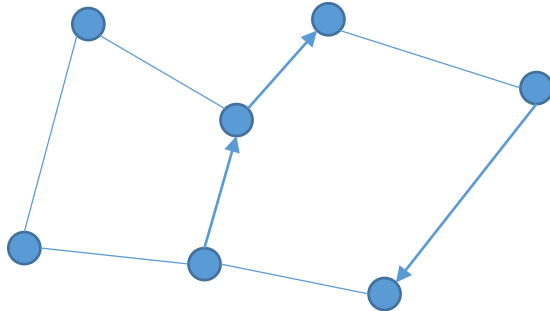
Acyclic Graph



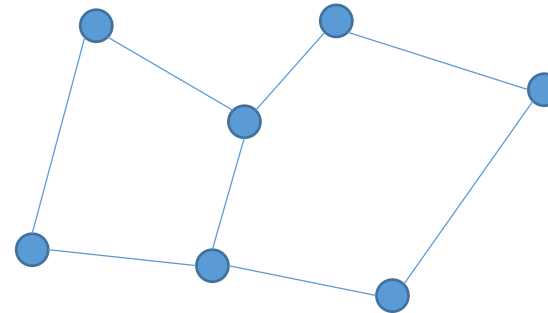
Some terminology: Directed

- If one or more edges may only be traversed in a specified direction we say the graph is *directed*.

Directed Graph



Undirected Graph

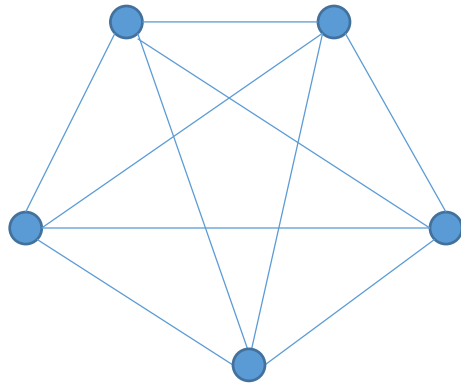


- Arrows indicate direction
- An undirected edge is the same as two directed edges.

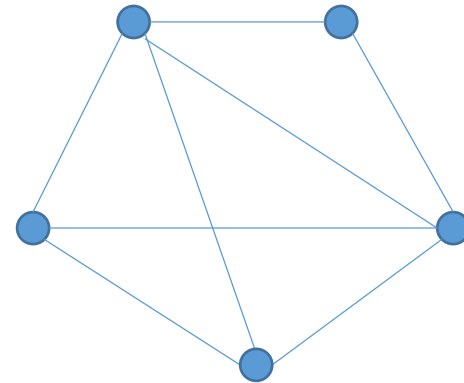
Some terminology: Complete

- If every pair of vertices is connected with an edge we say the graph is *complete*.

Complete Graph



Incomplete Graph

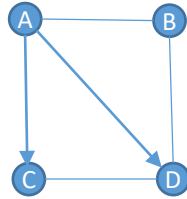


Representing a Graph

- We can represent a graph in a number of ways, apart from drawing it.
 1. List all vertices directly connected to a vertex in turn. This is also known as an *adjacency list*.
 2. List each pair of vertices connected by an edge. This is also known as an *edge list*.
 3. Construct a table showing all possible vertex pairs and fill in the locations where edges exist. This is also known as an *adjacency matrix*.
- Let us look at the different representations for a couple of sample graphs; one undirected and one directed.
- To make the process clearer we will use labelled graphs.

Adjacency list

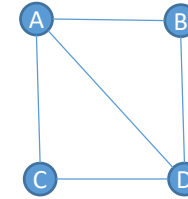
- Graph 1



- Adjacency List

- A: B, C, D
- B: A, D
- C: D
- D: B, C

- Graph 2

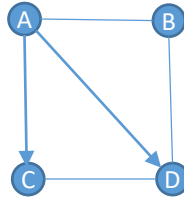


- Adjacency List

- A: B, C, D
- B: A, D
- C: A, D
- D: A, B, C

Edge list

- Graph 1

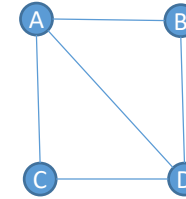


- Edge List

- AB, AC, AD, BA, BD, CD, DB, DC

- Note that undirected edges appear twice. E.g. AB and BA.

- Graph 2



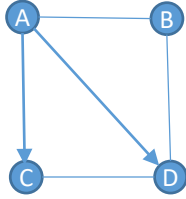
- Edge List

- AB, AC, AD, BD, CD

- Note that we only need to list each edge once if the graph is not directed.

Adjacency matrix

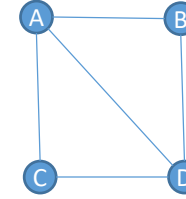
- Graph 1



- Adjacency Matrix

	to				
f r o m		A	B	C	D
	A		X	X	X
	B	X			X
	C				X
	D		X	X	

- Graph 2

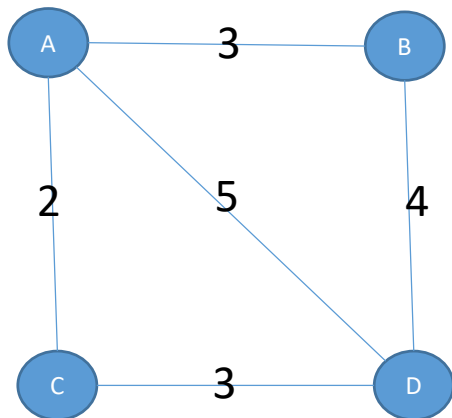


- Adjacency Matrix

	to				
f r o m		A	B	C	D
	A		X	X	X
	B	X			X
	C	X			X
	D	X	X	X	

Best representation

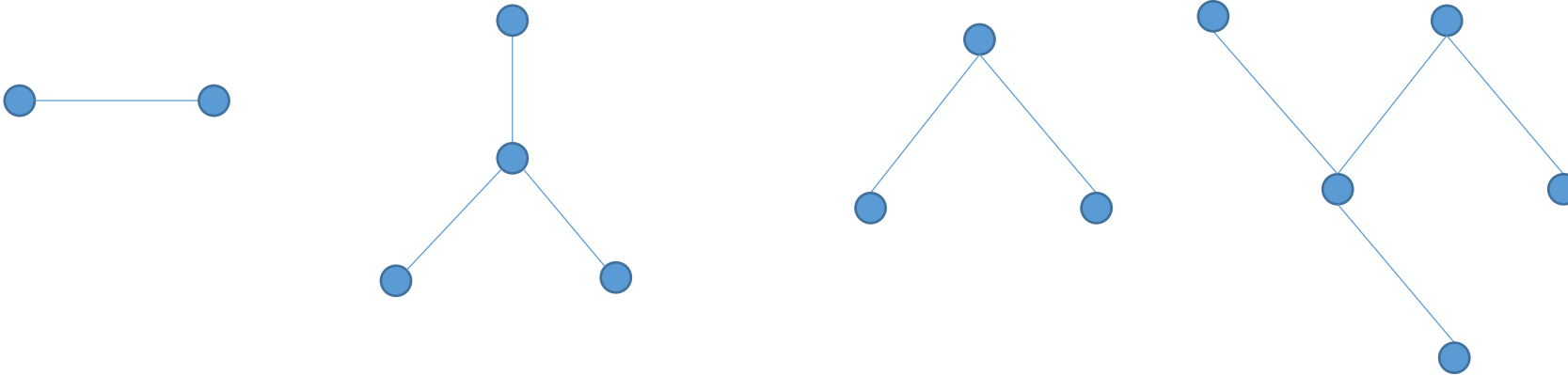
- There is no best representation for a graph. Each is useful in different circumstances.
- The adjacency matrix is often the preferred form, especially for weighted graphs.
- This is because we can add the weights to the table directly.



	to				
		A	B	C	D
from	A		3	2	5
	B	3			4
	C	2			3
	D	5	4	3	

Trees

- A tree is a special type of graph.
- It is a connected, acyclic graph.
- These are all trees:

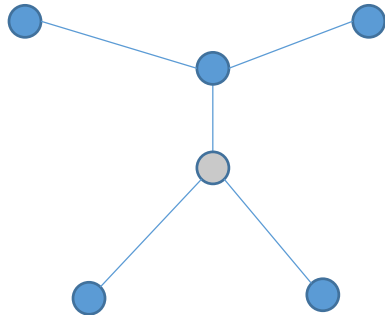


Some properties of trees

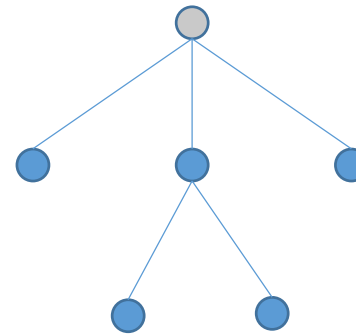
- There is a unique path between any two vertices.
- A tree with n vertices has $n - 1$ edges.
- We call the number of edges that are connected to a vertex the *degree* of the vertex.
- A vertex with degree > 1 is called an *internal* vertex.
- A vertex with degree $= 1$ is called an *external* vertex.
- Some trees have a special vertex, called the *root*. These trees are of particular interest.

Rooted trees

- A tree with a root vertex can be drawn with the root at the top and the other vertices below it in rows.
- Each row contains all the vertices that are the same number of edges away from the root.
- E.g. this rooted tree

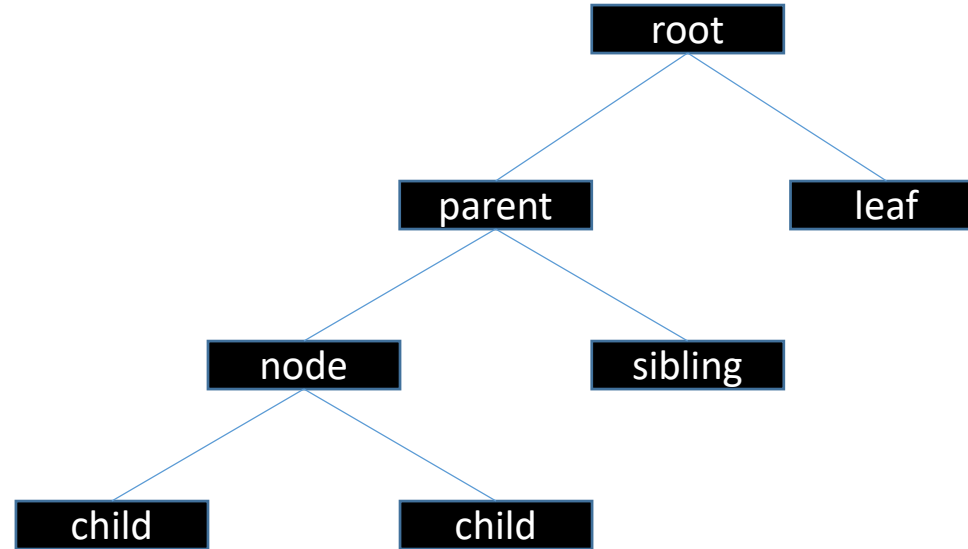


Can be drawn like this



Naming nodes.

- For a rooted tree we have the following naming conventions:



K -ary trees

- If each node can have no more than k children we say it is a k -ary tree.
- Where $k = 2$ we call it a *binary* tree.
- We will confine ourselves to binary trees for the time being.
- Specifically we will consider *ordered* binary trees.
- These are binary trees in which the left and right children are distinct.

Ordered binary trees

- These are different trees:



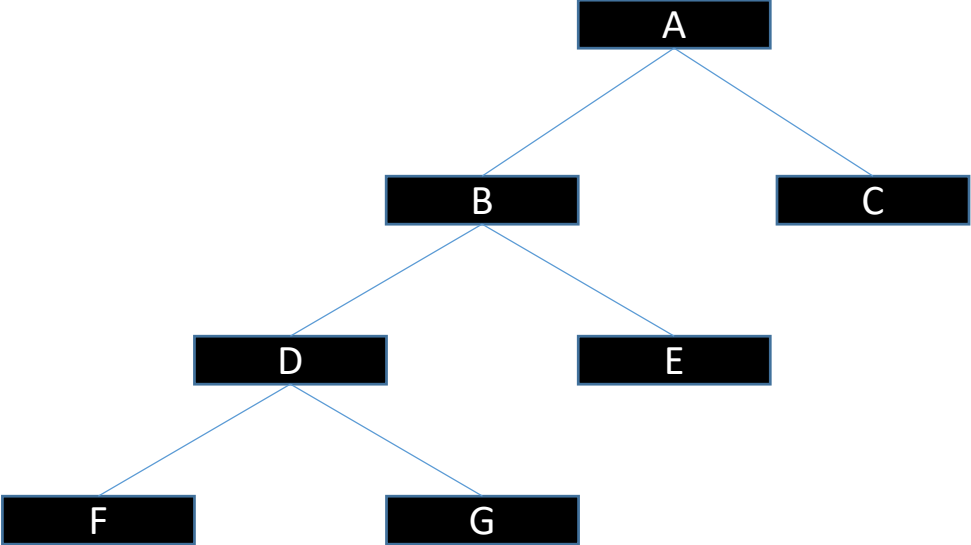
- So are these



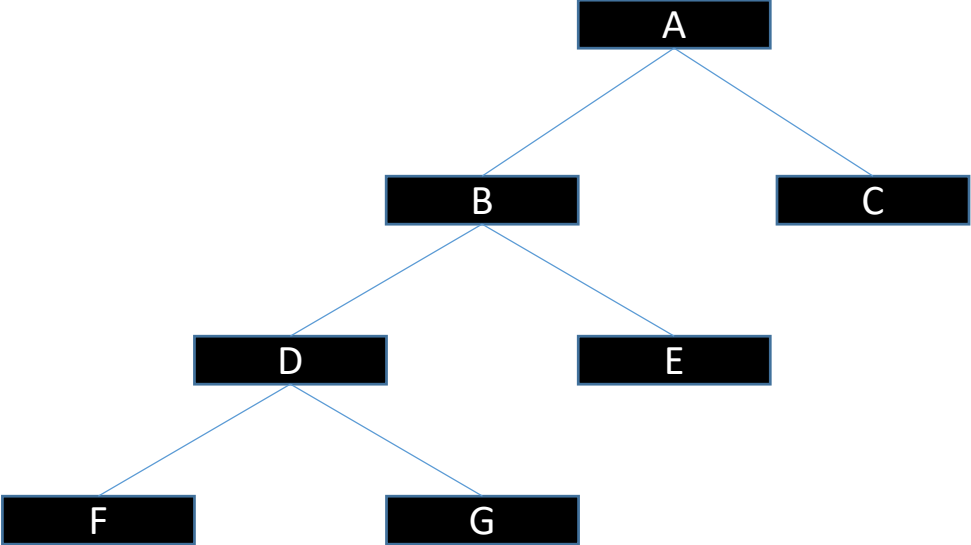
More tree terminology

- The *height* of a node is the number of edges in the longest path from that node to a leaf
- The *depth* of a node is the number of edges in the path from that node to the root
- The *level* of a node is equal to the height of the root of the tree minus the depth of the node
- For example:

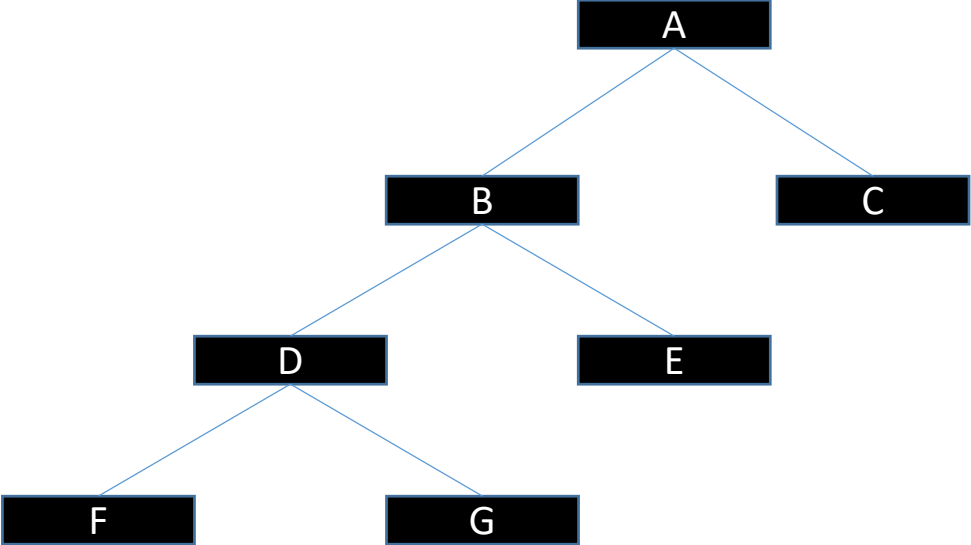
Node	Height	Depth	Level
A			
B			
C			
D			
E			
F			
G			



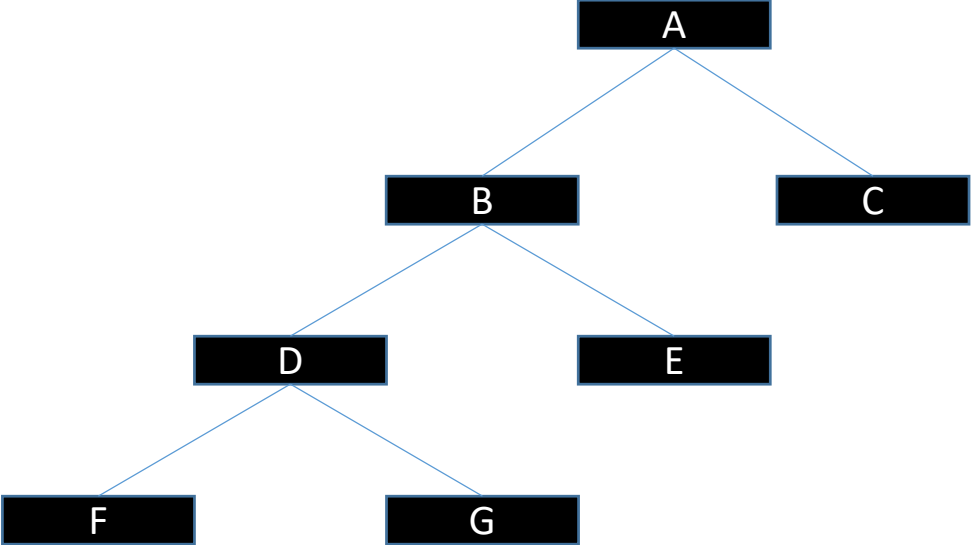
Node	Height	Depth	Level
A	3	0	3
B			
C			
D			
E			
F			
G			



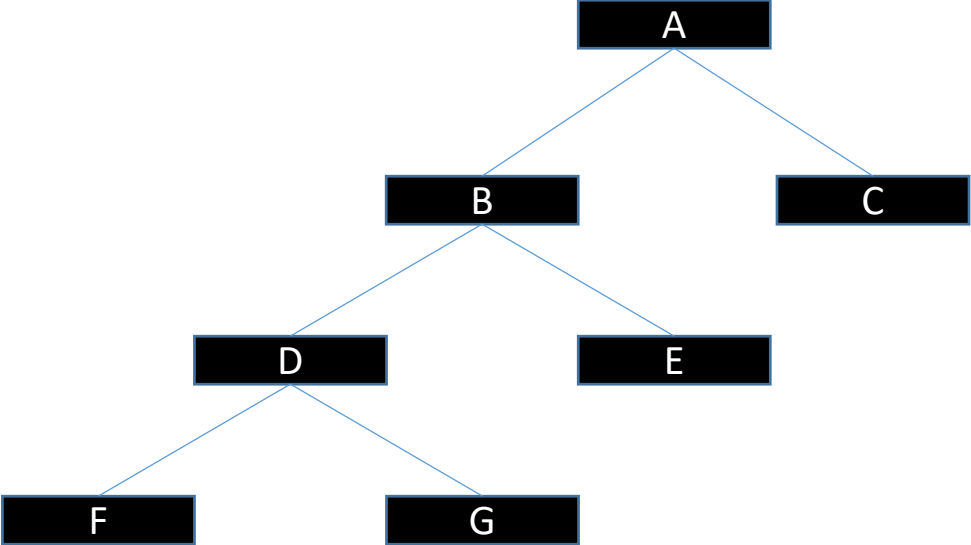
Node	Height	Depth	Level
A	3	0	3
B	2	1	2
C			
D			
E			
F			
G			



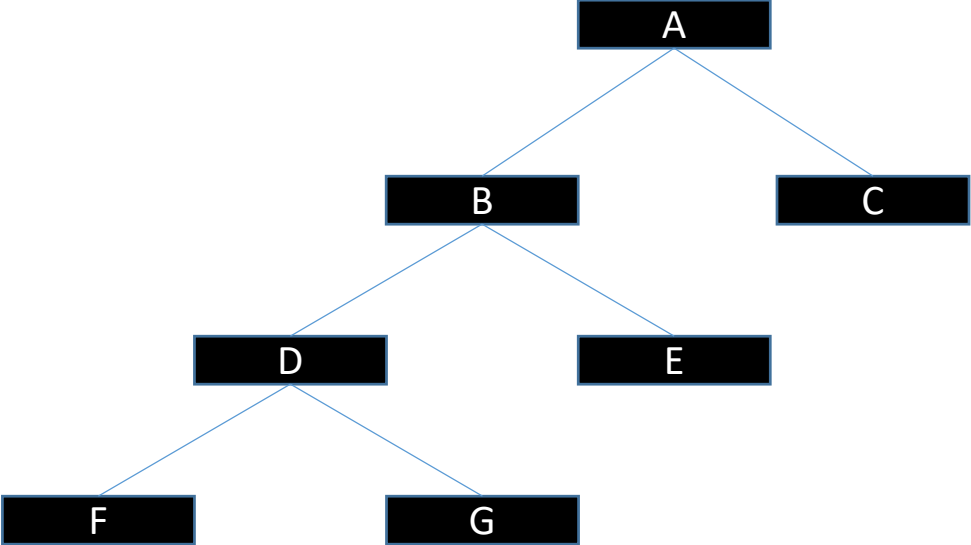
Node	Height	Depth	Level
A	3	0	3
B	2	1	2
C	0	1	2
D			
E			
F			
G			



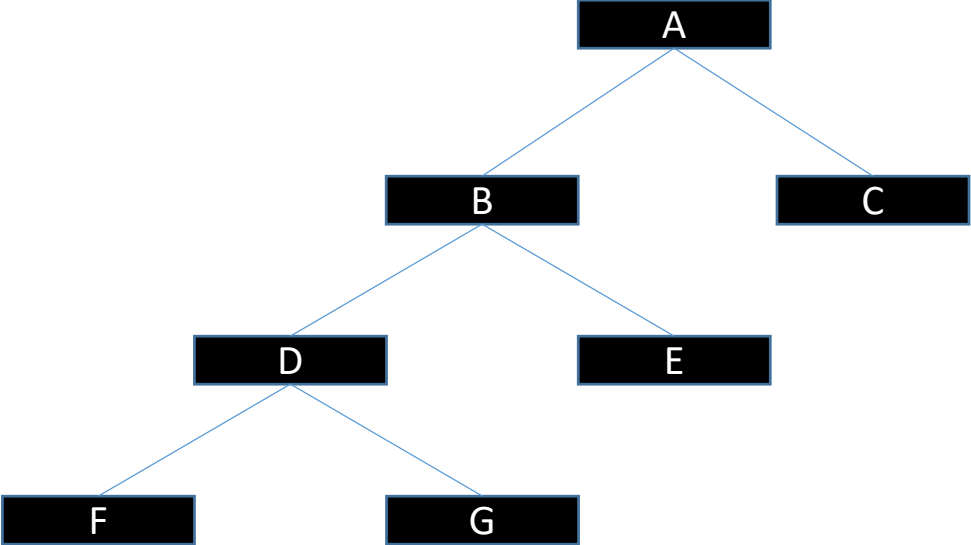
Node	Height	Depth	Level
A	3	0	3
B	2	1	2
C	0	1	2
D	1	2	1
E			
F			
G			



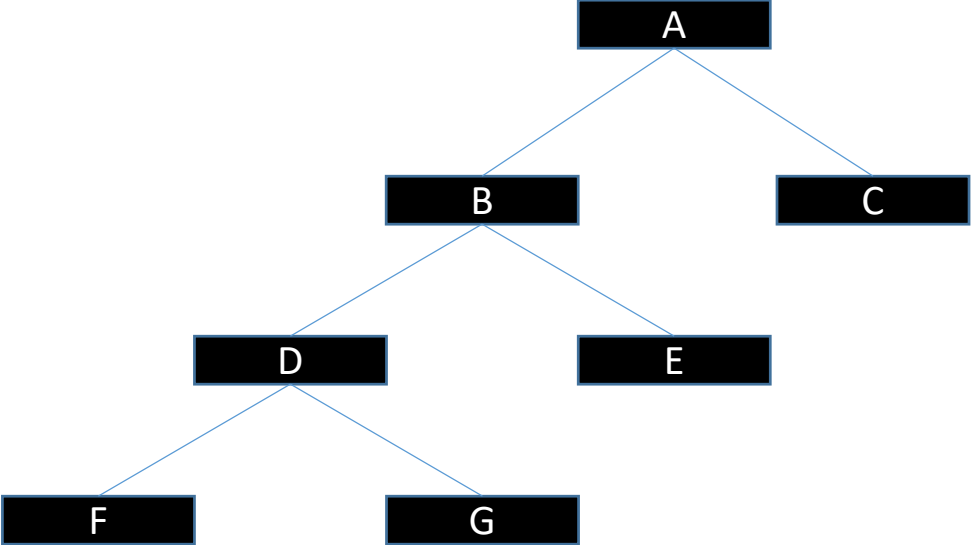
Node	Height	Depth	Level
A	3	0	3
B	2	1	2
C	0	1	2
D	1	2	1
E	0	2	1
F			
G			



Node	Height	Depth	Level
A	3	0	3
B	2	1	2
C	0	1	2
D	1	2	1
E	0	2	1
F	0	3	0
G			

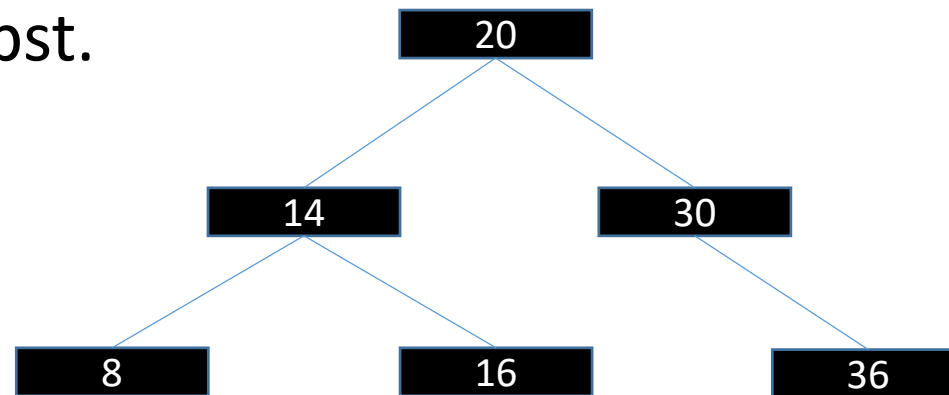


Node	Height	Depth	Level
A	3	0	3
B	2	1	2
C	0	1	2
D	1	2	1
E	0	2	1
F	0	3	0
G	0	3	0



Binary Search Trees

- A *binary search tree* (bst) has the following properties:
 1. It is a binary tree.
 2. The value in each node is greater than or equal to all the values in its left child or any of that child's descendants.
 3. The value in each node is less than or equal to all the values in its right child or any of that child's descendants.
- The following is a bst.



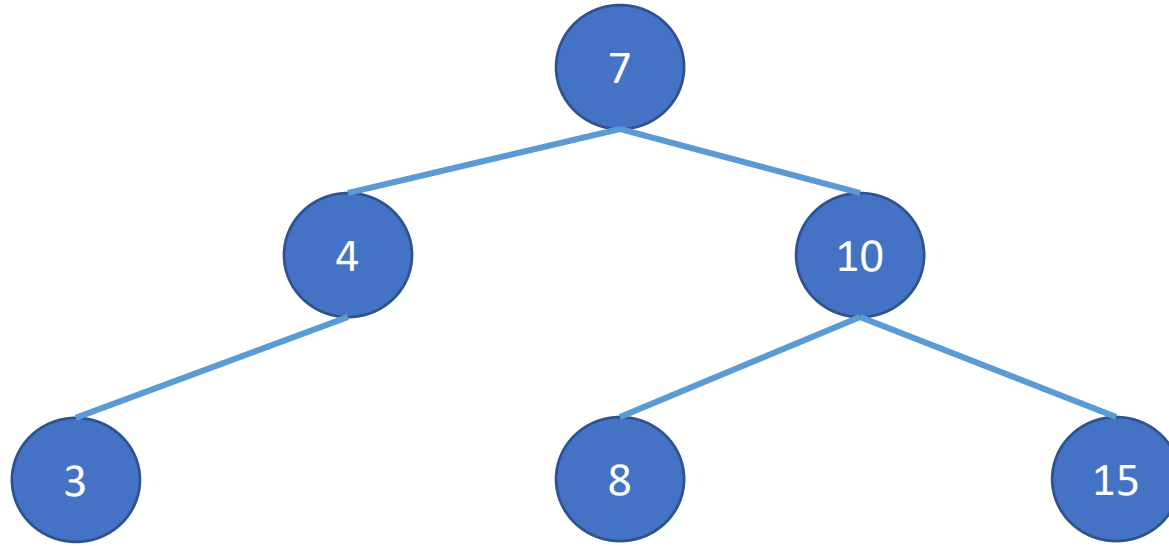
Searching a bst.

- We can search a bst for a given value using the following recursive strategy.
- Start searching at the root node.
 - If the value we are looking for equals the value of the node:
 - FOUND!
 - Stop.
 - If the value we are looking for is less than the value of the node:
 - If there is a left child search again starting at the left child.
 - Else
 - NOT FOUND
 - Stop
 - If the value we are looking for is greater than the value of the node:
 - If there is a right child search again starting at the right child.
 - Else
 - NOT FOUND
 - Stop

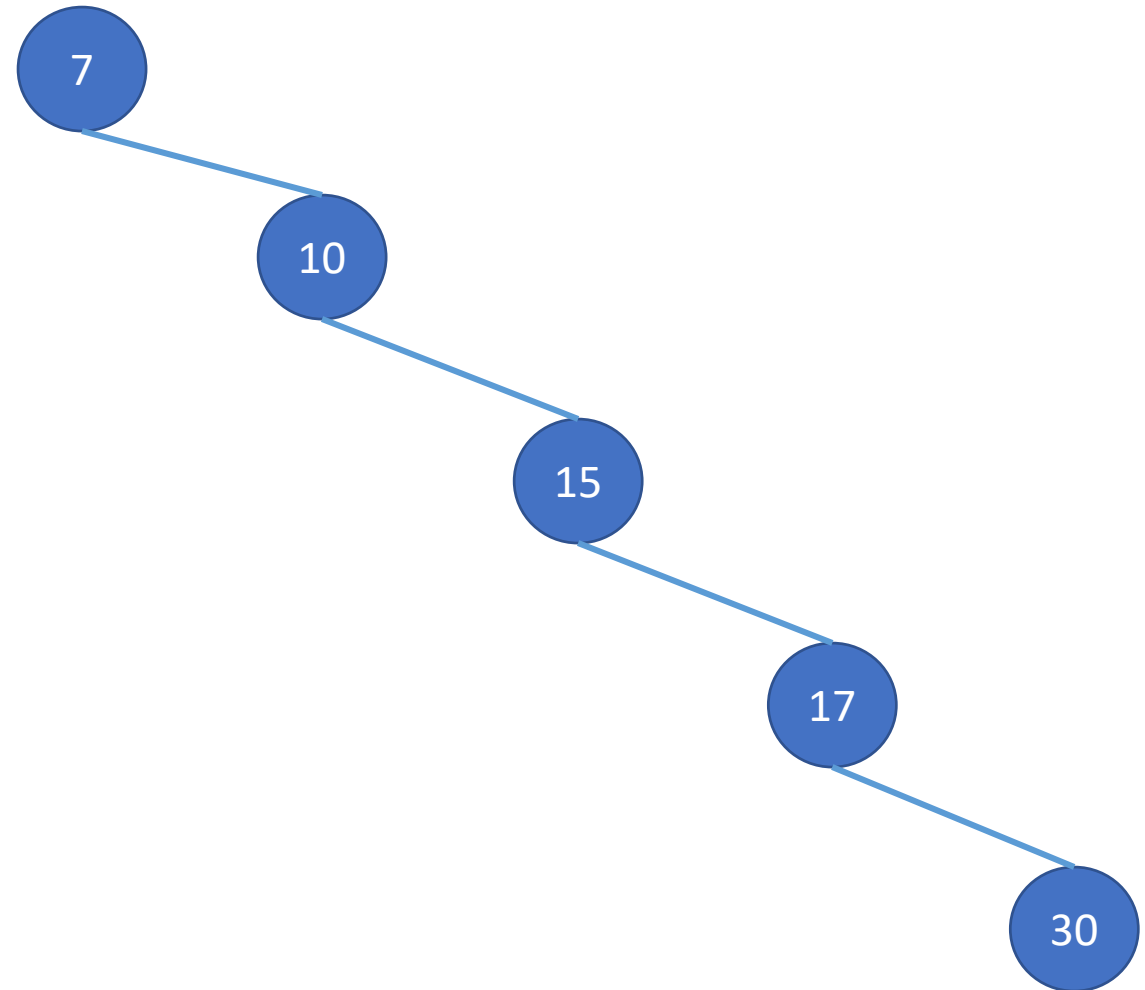
Building a bst.

- We can use the same approach to build a bst from a list of values.
 - For each value in the list:
 - Search for the value.
 - If we find it ERROR duplicate value.
 - Else keep searching
 - we eventually get to a point where the next node is missing.
 - Insert a new node, containing the value, at this location .
- Note: if we start with an ordered list we get a bst with only one child at each level.
 - This is really bad to search!

Example: Build a bst from {7,10,8,15,4,3}



Example: Build a bst from {7,10,15,17,30}



Deleting from a bst.

- To delete from a binary search tree we must do two things.
 1. Find the node to be deleted.
 - This is easy – do the search procedure from the root downwards.
 2. Delete the node.
 - This may be harder.
 - If the node to be deleted is a leaf we can simply remove it.
 - If it is not a leaf removing it would break the tree.
- Consider the following example.

- Delete the node containing 14.

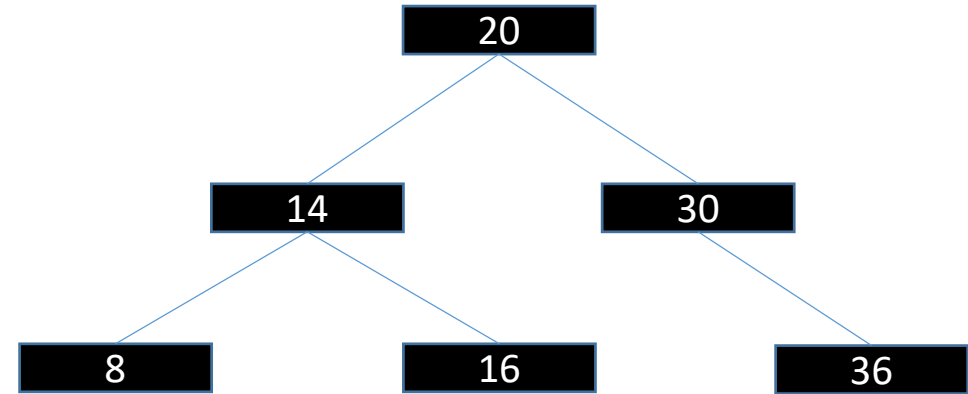
1. Find the node to delete.

1. Start at the root

1. $20 > 14$ so go left

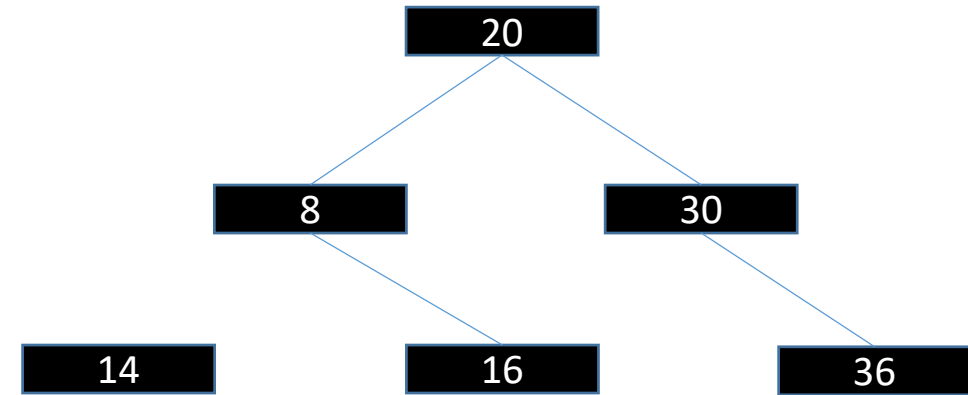
2. At node 14

1. $14 = 14$ this is the node to delete

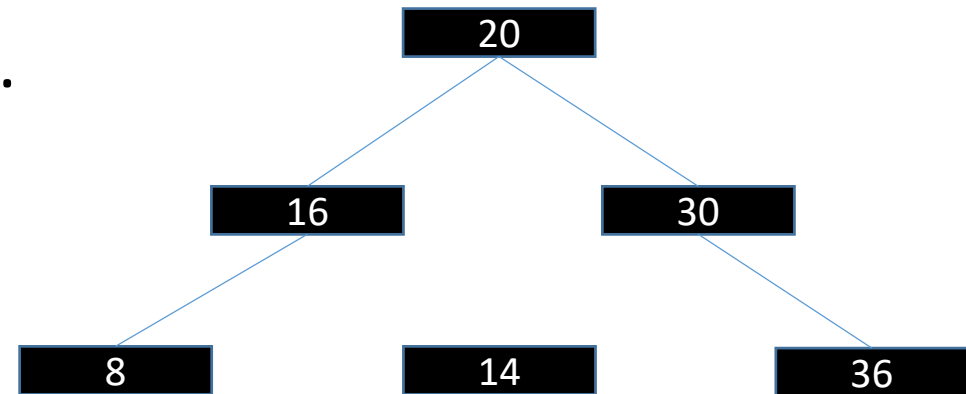


- If we simply delete the node with value 14 we lose the nodes 8 and 16 from the tree.
- What we have to do is swap the value of the node we wish to delete so that:
 - We keep the tree connected
 - When we delete node 14 from its new position we still have a bst.
- In this case we could swap 14 with either 8 or 16 and all will be well.

- If we swap the left child we get...



- If we swap the right child we get...



- What if the children also have children?

- Delete the node containing 30.

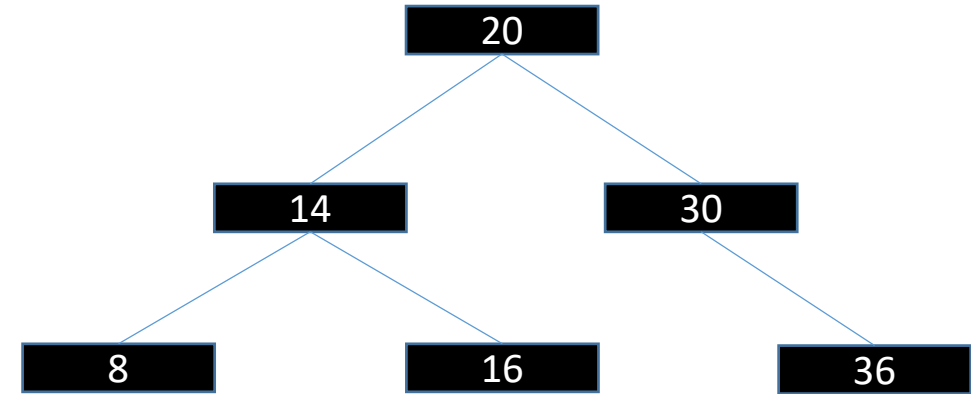
1. Find the node to delete.

1. Start at the root

1. $20 < 30$ so go right

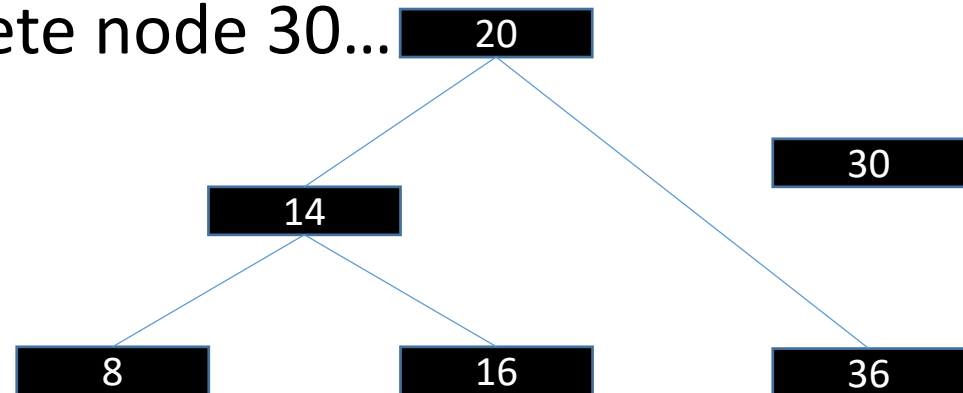
2. At node 30

1. $30 = 30$ this is the node to delete

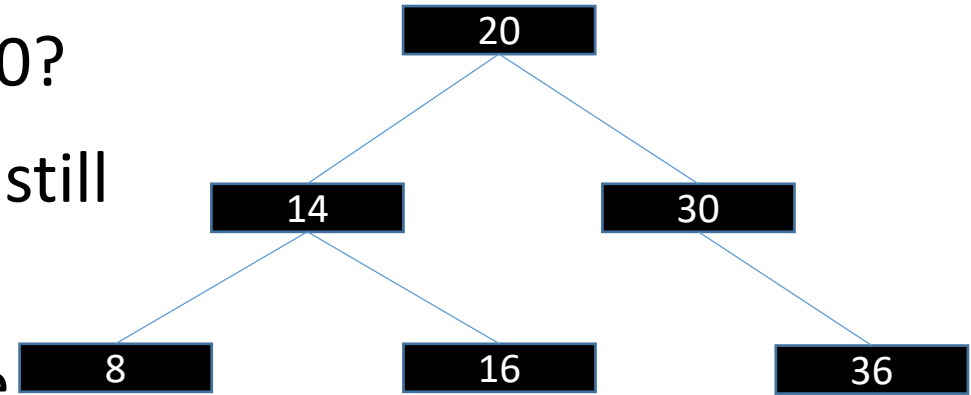


- If we simply delete the node with value 30 we lose nodes 36 from the tree.

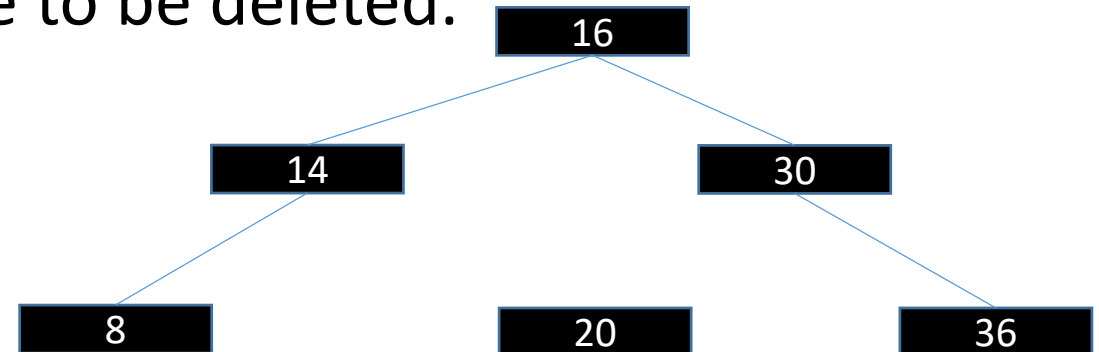
- In this case node 30 has a single child so we can simply link the parent of 30 to this child and delete node 30...



- What if we want to delete node 20?
- If we simply swap with a child we still have a problem.
- We need to swap with a leaf node.



- But which one.
- The strategy in this case is to find the rightmost node in the left subtree and swap it with the node to be deleted.
- In this case the result is...

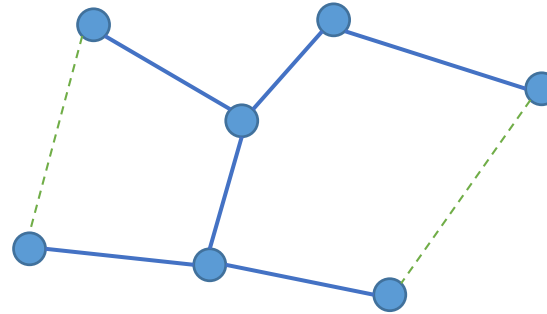
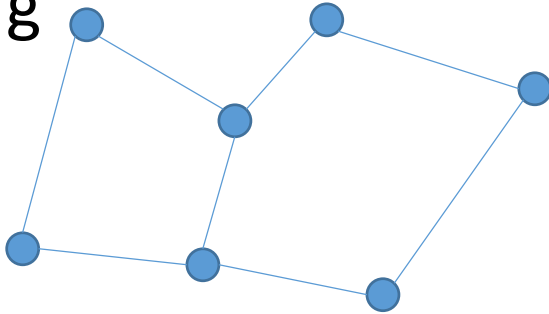


- We could also swap the leftmost node of the right subtree.

Trees and Graphs

- Given any connected graph G , we can always find at least one tree which contains all of the vertices of G with a subset of its edges.

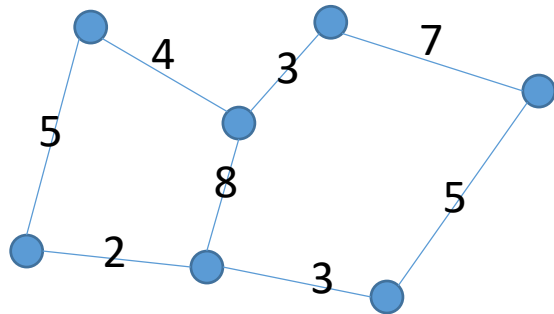
- E.g



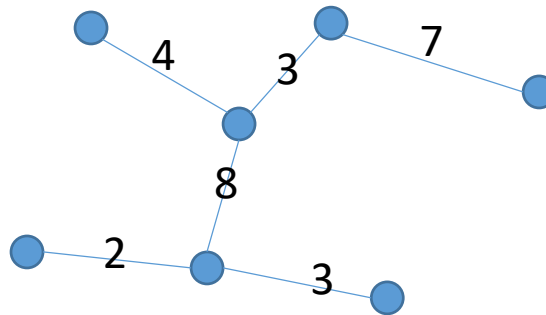
- This is called a *spanning tree*.
- Note: This is not usually a unique tree.

Minimal spanning tree

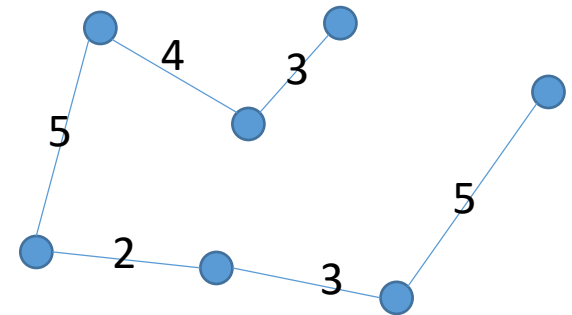
- If G is a weighted graph we can define the *weight* of a spanning tree as the sum of the weights of all the edges in the tree.
- E.g.



Weight = 27



Weight = 22



- We call the spanning tree with the smallest weight the *minimal spanning tree*.

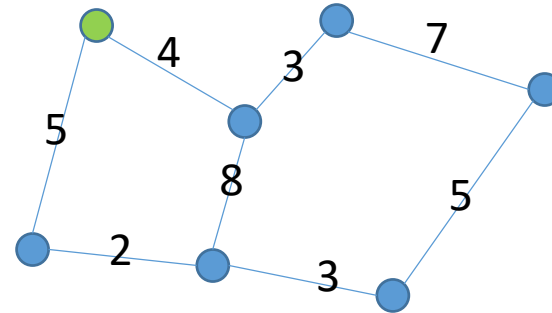
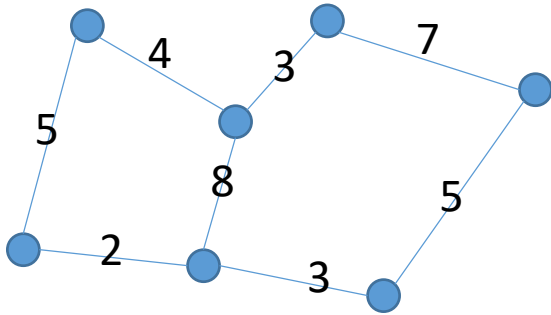
Finding the minimal spanning tree.

- If we need to find the minimal spanning tree of a graph we can take two approaches.
 - Add a vertex to the spanning tree at each step.
 - Add an edge at each step.
- Each approach clearly requires a bit more detail to make it work.
- Each approach can be used to find the minimal spanning tree.
- Let us look at the two approaches in more detail.

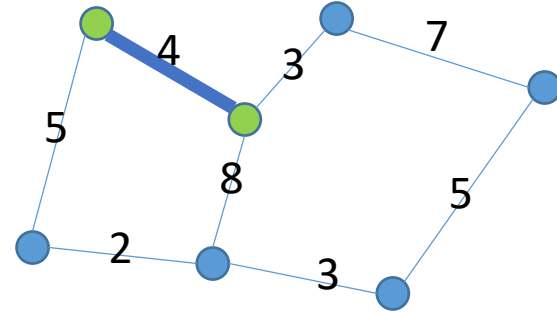
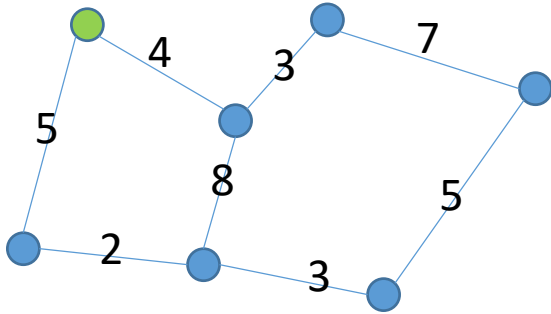
Vertex at a time (Prim's Algorithm)

- Start with any vertex. This is our starting tree.
- Find the vertex connected to the tree so far by the shortest edge.
- Add this edge and vertex to the tree.
- Repeat this process, always choosing a vertex which is not already part of the tree.
- Stop when all vertices are selected.
- Let us try this with a sample graph:

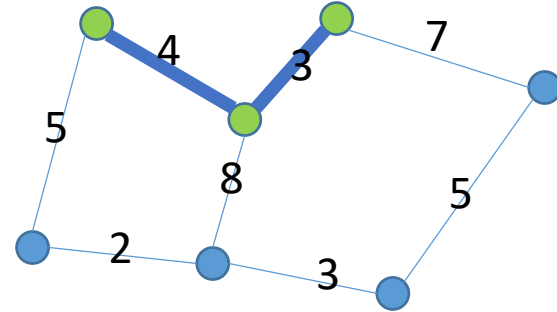
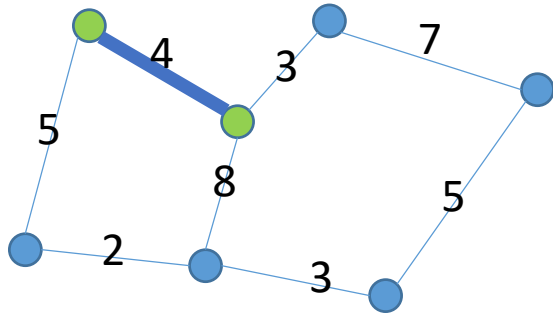
- Pick a vertex



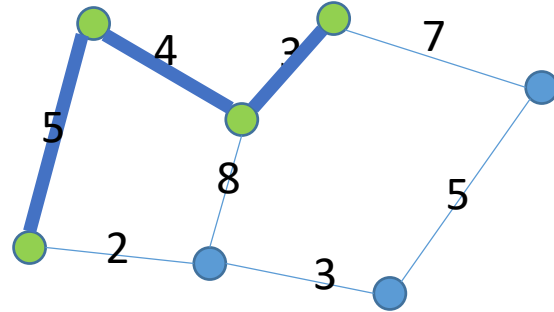
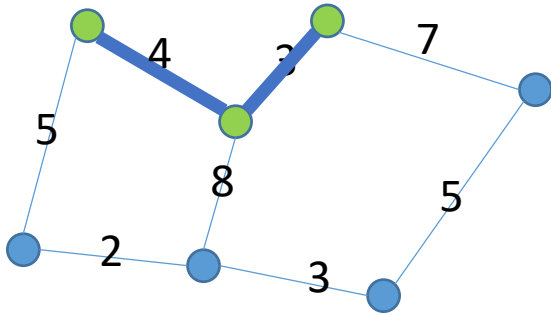
- Pick the nearest connected vertex



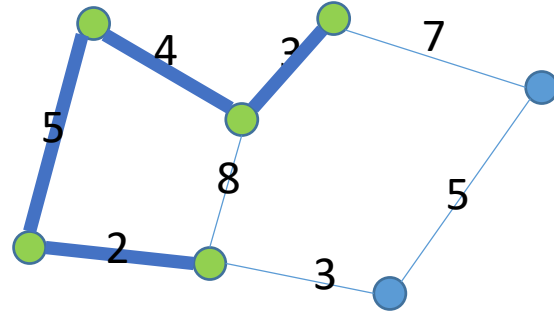
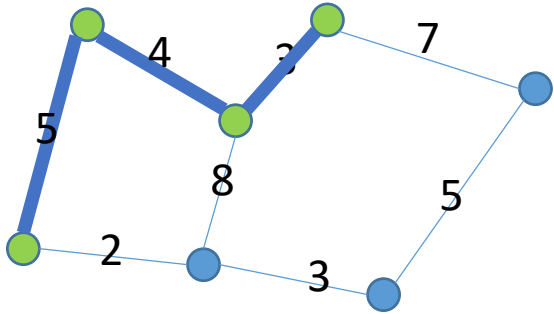
- Pick the nearest connected vertex



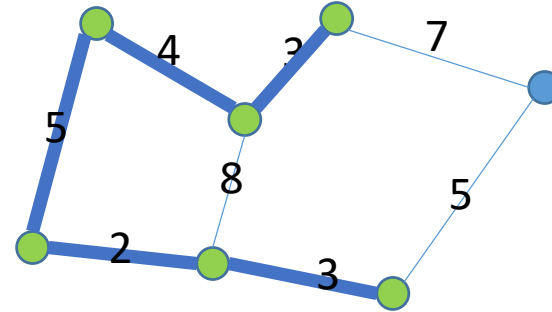
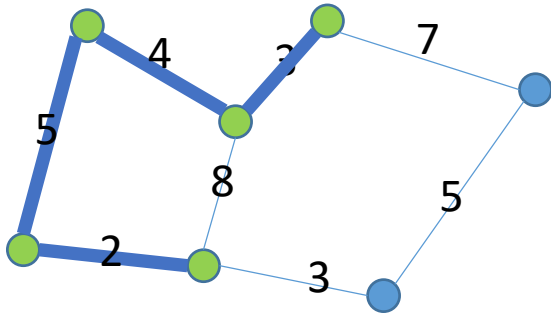
- Pick the nearest connected vertex



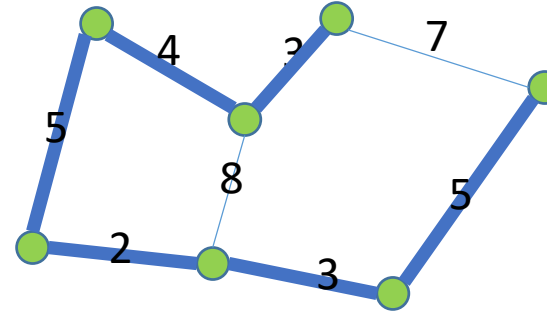
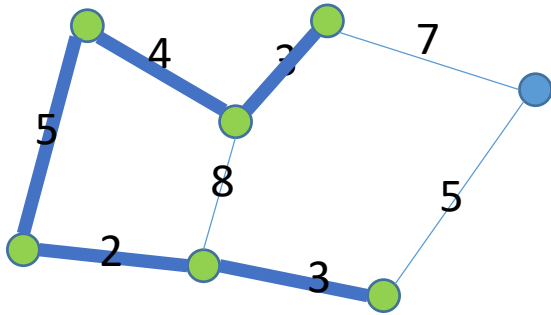
- Pick the nearest connected vertex



- Pick the nearest connected vertex



- Pick the nearest connected vertex

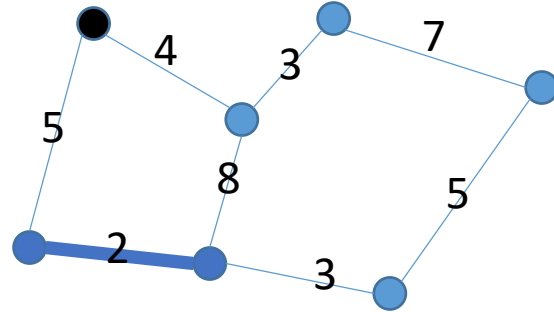
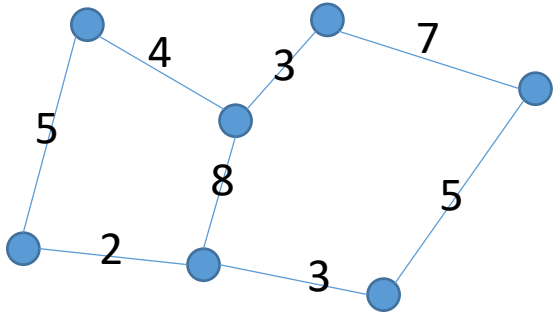


- And we are done.
 - It doesn't matter which vertex we start with.

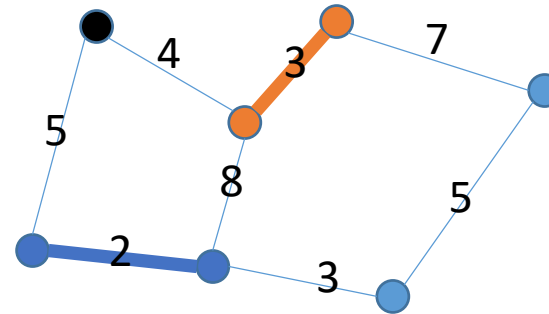
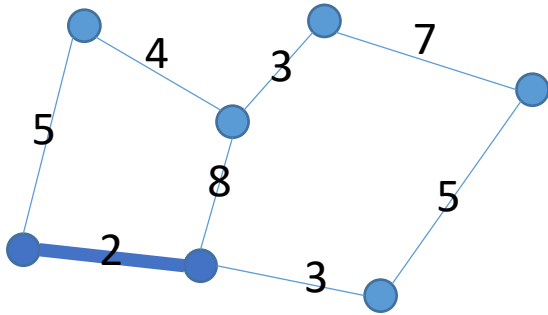
Edge at a time. (Kruskal's Algorithm)

- Pick the shortest edge. This is our starting tree.
- Now pick the next shortest edge.
 - If it connects previously unconnected vertices add it to our tree.
 - Otherwise reject it.
- Repeat until all vertices are connected.
- Let us try this with a sample graph:

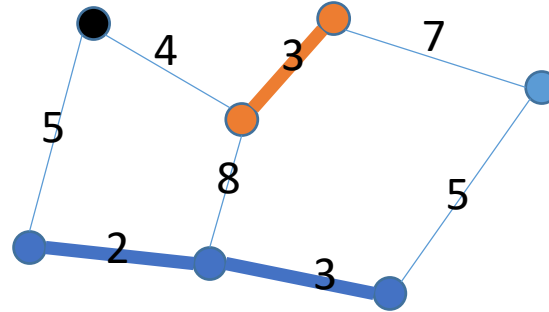
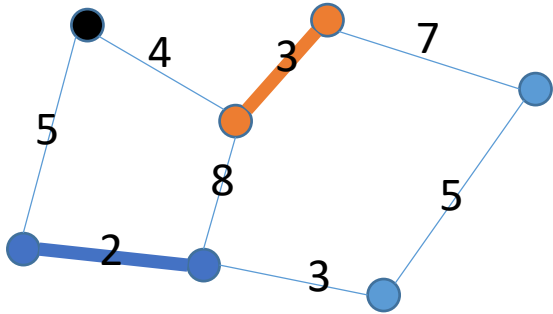
- Pick the shortest edge



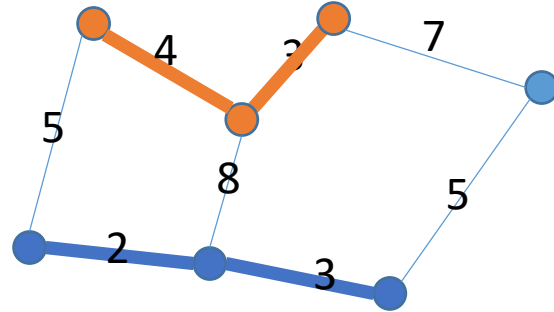
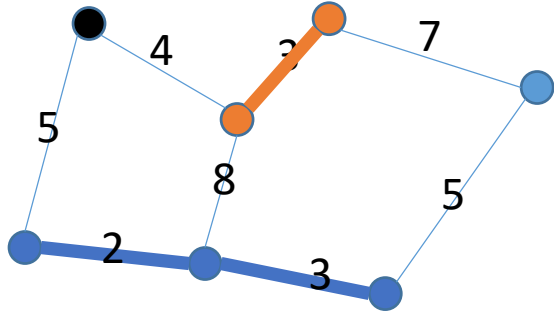
- Pick the next shortest edge



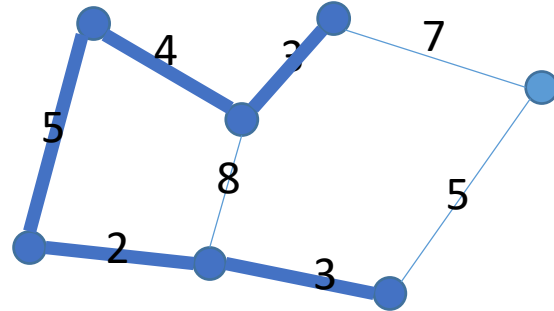
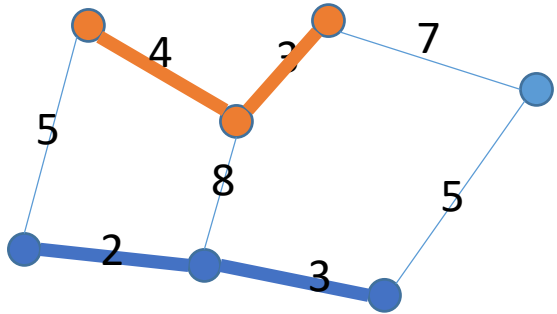
- Pick the next shortest edge



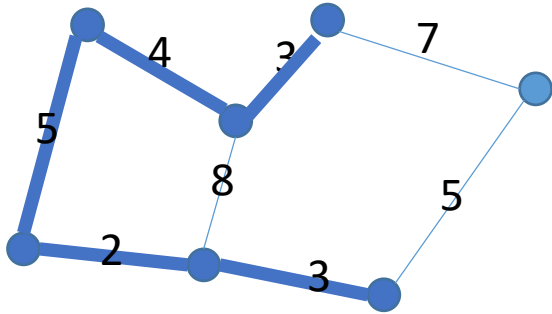
- Pick the next shortest edge



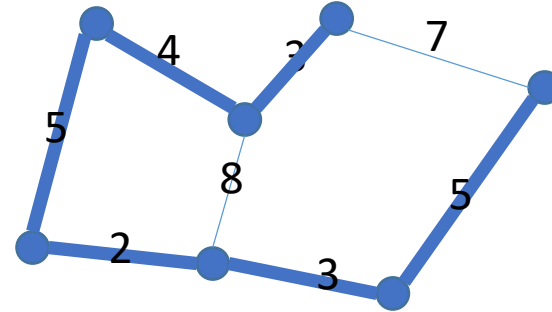
- Pick the next shortest edge



- Pick the next shortest edge



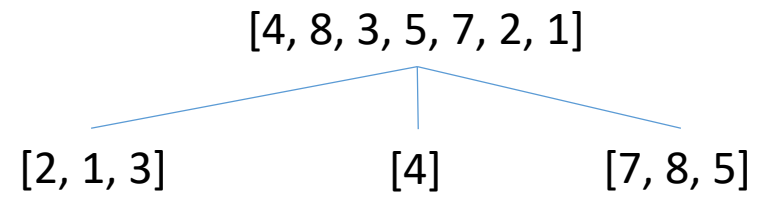
- And we are done.



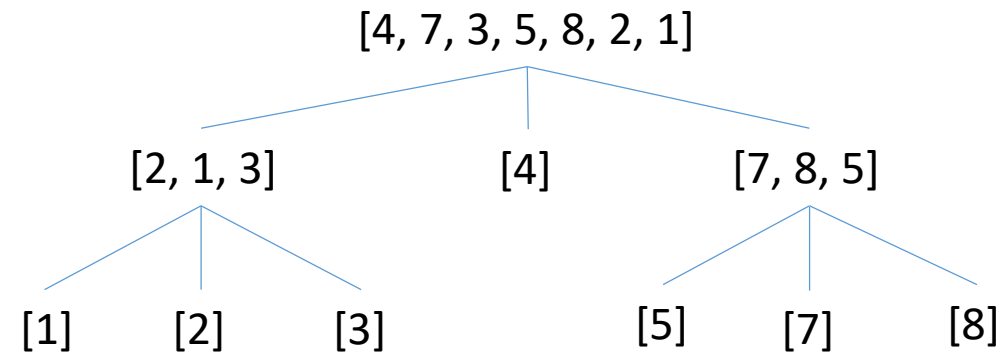
Quicksort and Trees

- Quicksort can be looked at in terms of trees, as follows:
- The root node is the unsorted list.
- When we partition the list to be sorted we can view the partitions as its children.
- Repeated partitioning grows the tree.
- E.g. sort the list [4, 8, 3, 5, 7, 2, 1]

- First Partition



- Second Partition



How many operations?

- If the list to be sorted contains n elements.
- At each level of the tree we carry out roughly n operations in all of the partitions counted together.
- This means that the total number of operations is roughly given by n times the depth of the tree.
- We will estimate this depth in the following slides.
- What is the depth of a tree with n leaves?
 - It depends...
 - What is the order of the tree?
 - How full is it?

Refining the question

- Ok, what is the depth of a complete binary tree with n leaves?

n	tree	depth
2		
4		
8		

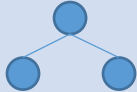
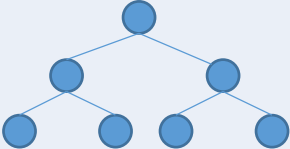
Refining the question

- Ok, what is the depth of a complete binary tree with n leaves?

n	tree	depth
2		1
4		
8		

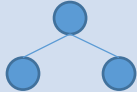
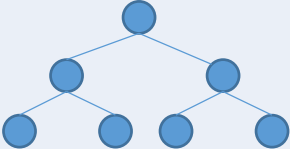
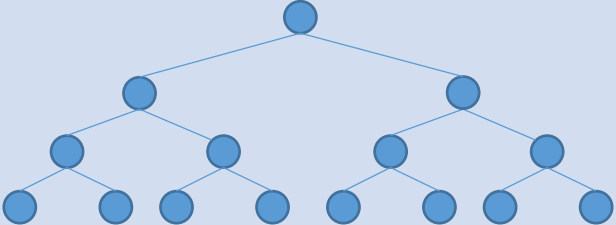
Refining the question

- Ok, what is the depth of a complete binary tree with n leaves?

n	tree	depth
2		1
4		2
8		

Refining the question

- Ok, what is the depth of a complete binary tree with n leaves?

n	tree	depth
2		1
4		2
8		3

Quicksort efficiency.

- So, if we have $n = 2^k$ leaves the complete tree has a depth of k .
- Another way of stating this is that if a complete tree has n leaves it has a depth of $\log_2 n$.
- Thus, provided the partition always splits the lists into equal halves, we can expect quicksort to take around $n \times \log_2 n$ operations.
- This is the *best case* behaviour for quicksort.
- In the worst case quicksort can take up to n^2 operations!
 - Those of you who do CSCI203 will see sorts that always use $n \times \log_2 n$ operations.
- The factor that controls how well quicksort works is how well the partitioning scheme works.

Quicksort partitioning.

- Let us examine in more detail how the partitioning process of quicksort works.
- Take the list [4, 8, 3, 5, 7, 2, 1] as an example.
- Our partition (or *pivot*) value is 4.
- Let us also mark the two ends of the remainder of the list.
 - Let us call these values *head* and *tail*.
- We now proceed as follows:

Quicksort partitioning.

1. Compare the pivot to the head.
 - If it is larger than head move head to the right and repeat step 1.
 - Otherwise go to step 2.
2. Compare the pivot to the tail.
 - If it is smaller than tail move tail to the left and repeat step 2.
 - Otherwise go to step 3.
3. If head and tail have met or crossed over, swap the pivot with tail and stop.
 - Otherwise go to step 4.
4. Swap the values at head and tail
 - Move head to the right
 - Move tail to the left

Quicksort partitioning

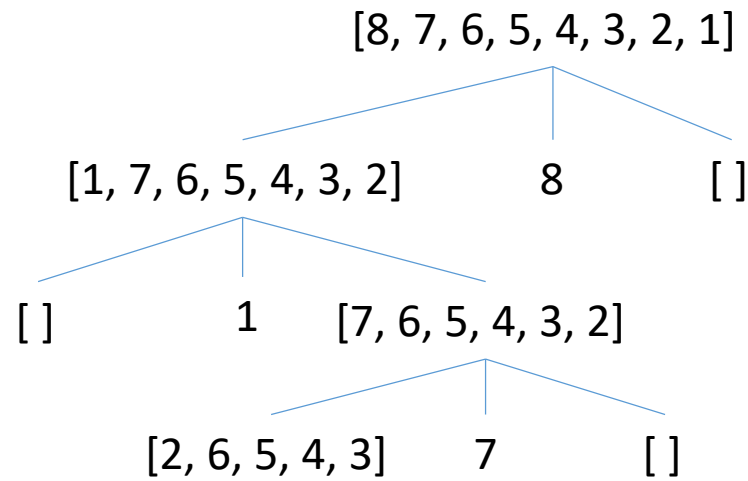
- Start: [4, 8, 3, 5, 7, 2, 1]
 - Step 1. compare 4 and 8
 - $8 > 4$ so go to step 2.
 - Step 2. compare 4 and 1
 - $1 < 4$ so go to step 3.
 - Step 4. swap head and tail and move them.
 - [4, 1, 3, 5, 7, 2, 8]
 - Step 1. compare 4 and 3
 - $3 < 4$ so move head and repeat step 1
 - [4, 1, 3, 5, 7, 2, 8]
 - Step 1. compare 4 and 5
 - $5 > 4$ so go to step 2

1. Compare the pivot to the head.
 - If it is larger than head move head to the right and repeat step 1.
 - Otherwise go to step 2.
2. Compare the pivot to the tail.
 - If it is smaller than tail move tail to the left and repeat step 2.
 - Otherwise go to step 3.
3. If head and tail have met or crossed over, swap the pivot with tail and stop.
 - Otherwise go to step 4.
4. Swap the values at head and tail
 - Move head to the right
 - Move tail to the left

- Step 2. compare 4 and 2
 - $2 < 4$ so go to step 3
- Step 4. swap head and tail and move them.
 - [4, 1, 3, 2, 7, 5, 8]
- Step 1. compare 4 and 7
 - $7 > 4$ so go to step 2
- Step 2. compare 4 and 7
 - $7 > 4$ so move tail and repeat step 2
 - [4, 1, 3, 2, 7, 5, 8]
- Step 2. compare 4 and 2
 - $2 < 4$ so go to step 3
- Step 3. swap 4 and 2 and stop.
 - [2, 1, 3, 4, 7, 5, 8]

When partitioning goes wrong

- If our list is nearly sorted (or reverse ordered) the partitioning process goes badly wrong.
- Consider the list [8, 7, 6, 5, 4, 3, 2, 1].
- The start of our partition tree looks like this:



A better way to partition

- We can improve this process in a very simple way.
- Instead of choosing the first element as the pivot do the following:
 - Compare the first middle and last elements of the partition
 - Swap the middle-sized value of these into the start position.
- Now continue partitioning as usual.
- Let us see what happens if we do this with our last example.
 - [8, 7, 6, 5, 4, 3, 2, 1]
 - Compare 8, 5 and 1; swap 8 and 5. [5, 7, 6, 8, 4, 3, 2, 1]
 - Now our first partition results in [4, 1, 2, 3] 5 [8, 6, 7]
 - This turns into [3, 1, 2, 4] 5 [7, 6, 8] ready for the next set of partitions