CSIT113
Problem Solving

UNIT 8
GRAPH AND TREE FOR MODELLING

UNIVERSITY OF WOLLONGONG AUSTRALIA

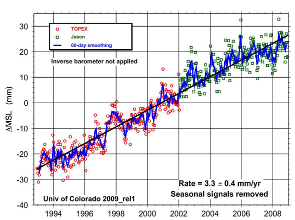
SIM GLOBAL EDUCATION

Overview

- Terminologies
- Binary Tree
- Binary Search Tree
- Graph
- Three well-known and important algorithms:
  - Finding Minimal Spanning Trees: Kruskal's Algorithm and Prim's Algorithm
  - Finding Shortest Paths: Dijkstra's Algorithm
- Analysing Quicksort using Binary Tree

Graphs and Trees

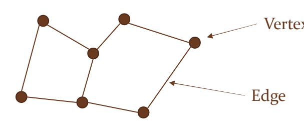
- Not this...



- Or this...

What then?

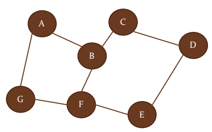
- A graph is defined as the combination of two sets, *V* and *E*.
- *V* is the set of vertices.
  - Points in space.
- *E* is the set of edges.
  - Lines connecting vertices.



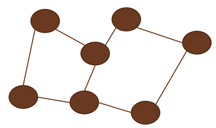
Terminologies: Labelled

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

Labelled Graph



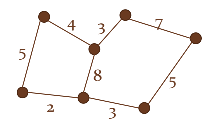
Unlabelled Graph



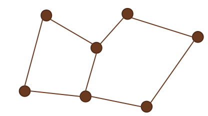
Terminologies: Weighted

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

Weighted Graph



Unweighted Graph

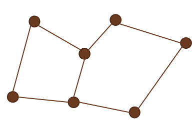


- The weight values may indicate a distance, a cost or some other property.

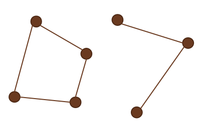
Terminologies: Connected

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

Connected Graph



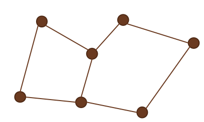
Disconnected Graph



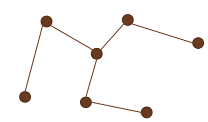
Terminologies: Cyclic

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

Cyclic Graph



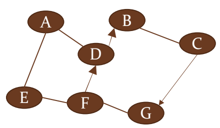
Acyclic Graph



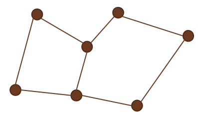
Terminologies: Directed, Undirected and Adjacent

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

Directed Graph



Undirected Graph

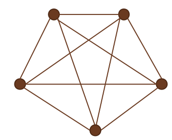


- Arrows indicate direction
- An undirected edge is the same as two directed edges.
- A vertex w is said to be adjacent to another vertex v if the graph contains an edge (v, w). For example, in the above directed graph, D is adjacent to F and D is also adjacent to A.

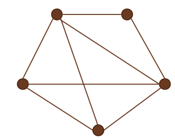
Terminologies: Complete

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

Complete Graph



Incomplete Graph

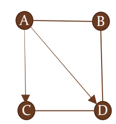


Representing a Graph

- We can represent a graph in a number of ways, apart from drawing it.
  1. For each vertex v, list all vertices that are adjacent to v. 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.

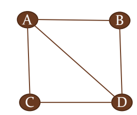
Adjacency list

- Graph 1



- Adjacency List
  - A: B, C, D
  - B: A, D
  - C: D
  - D: B, C

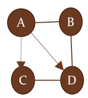
- Graph 2



- Adjacency List
  - A: B, C, D
  - B: A, D
  - C: A, D
  - D: A, B, C

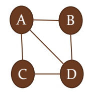
Edge list

- Graph 1



- Edge List
  - AB, AC, AD, BA, BD, CD, DB, DC
- Note that undirected edges appear twice. E.g. AB and BA.

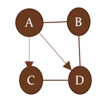
- Graph 2



- 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

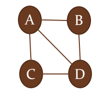
- Graph 1



- Adjacency Matrix

		to			
		A	B	C	D
from	A		X	X	X
	B	X			X
	C				X
	D		X	X	

- Graph 2

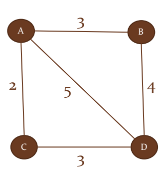


- Adjacency Matrix

		to			
		A	B	C	D
from	A		X	X	X
	B	X			X
	C	X			X
	D	X	X	X	

Best representation

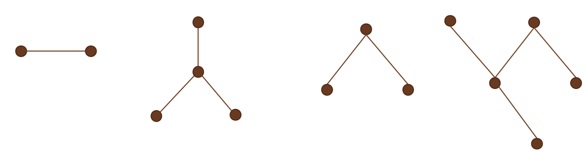
- There is no best representation for a graph. Each is useful in different circumstances.
- The adjacency matrix is often the preferred form, especially for weighted graphs.
- This is because we can add the weights to the table directly.



		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:

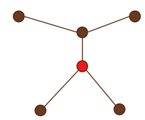


Some properties of trees

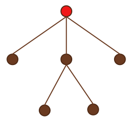
- There is a unique path between any two vertices.
- A tree with n vertices has $n - 1$ edges.
- We call the number of edges that are connected to a vertex the *degree* of the vertex.
- A vertex with degree > 1 is called an *internal* vertex.
- A vertex with degree = 1 is called an *external* vertex (does not have children).
- Some trees have a special vertex, designated as 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

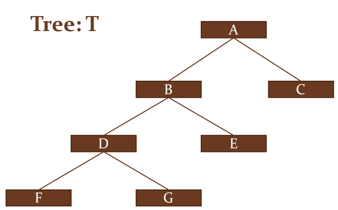


Can be drawn like this



Naming Conventions for Rooted Tree

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



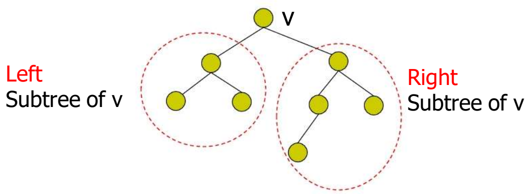
- A is the root
- A, B, C, D, E, F, G are all nodes.
- C, E, F and G are all leafs.
- D is the parent of F and G, B is the parent of D and E, etc.
- F is a child of D, D is a child of B, etc.
- D, B and A are all ancestor of F. A and B are both ancestors of E, etc.
- F, G, D and E are all descendants of B, etc.
- B itself, all B's descendants and the edges in T connecting all these nodes form the subtree of T rooted at B, etc.
- The subtree rooted at the left child of B is called the left subtree of B.
- The subtree rooted at the right child of B is called the right subtree of B.

K-ary trees

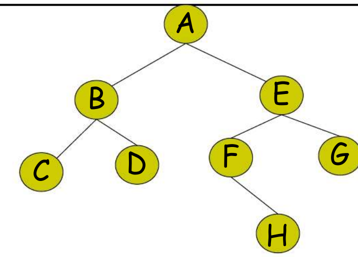
- If each node can have no more than k children we say it is a k-ary tree.
- Where $k = 2$ we call it a *binary* tree.
- We will confine ourselves to binary trees for the time being.
- Specifically we will consider *ordered* binary trees.

Binary Trees

- An empty tree (or null tree) is denoted as NIL.
- A tree in which no node has more than 2 subtrees.
- These subtrees are called the left and right subtrees



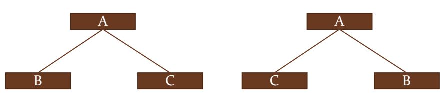
Binary trees



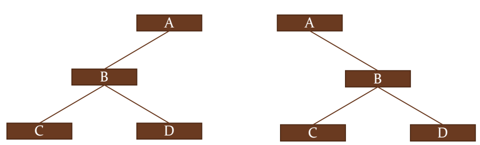
C is the ***LEFT child*** of B
D is the ***RIGHTchild*** of B.

Ordered binary trees

- These are different trees:



- So are these

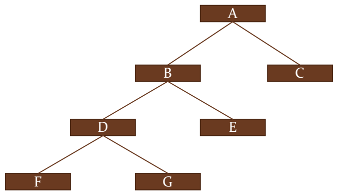


More Tree terminologies

- The *height* of a node is the number of edges in the longest path from the node to a leaf.
- The *height* of a binary tree is the height of the root.
- The *depth* of a node v is the number of edges in the path from the root to v.
- Th *level* of a node v is equal to the depth of v.

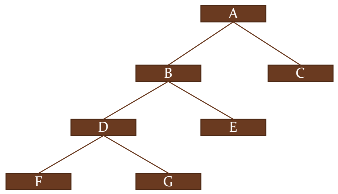
Example

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



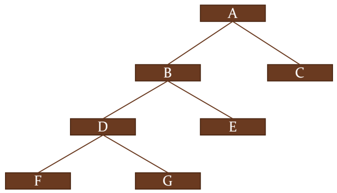
Example

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



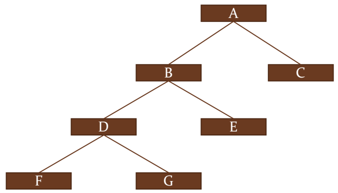
Example

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



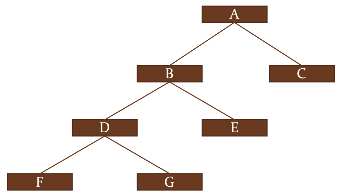
Example

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



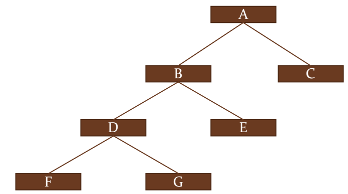
Example

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



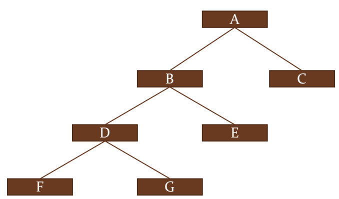
Example

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



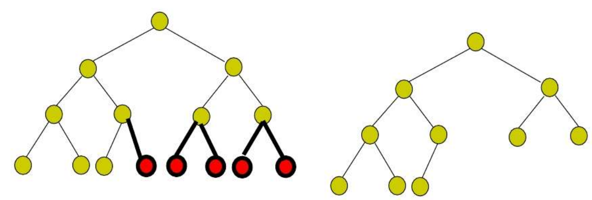
Example

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



The height of the binary tree = the height of the root (A) = 3

Complete and Nearly Complete Binary Trees



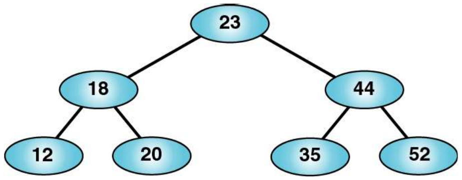
"complete trees":
If it has the maximum number of nodes for its height.

"Nearly complete trees":
If all nodes in the last level are found on the left, and all the other levels are fully filled.

Binary Tree Traversal

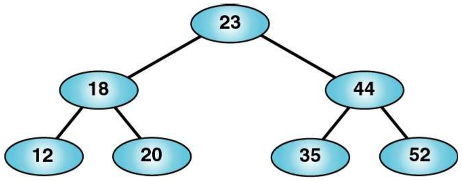
- Preorder
- Inorder
- Postorder

Binary Tree Traversal



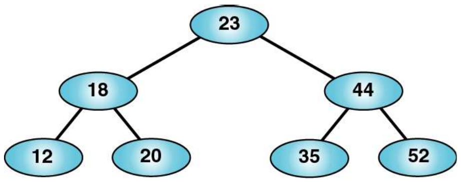
Preorder traversal:
23
18
12
20
44
35
52

Binary Tree Traversal



Inorder traversal:
12
18
20
23
35
44
52

Binary Tree Traversal



Postorder traversal:
12
20
18
35
52
44
23

Efficient structure (representation) for searching and maintenance

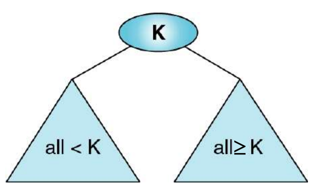
- Many areas in IT require structure (representation) that is efficient for searching and maintenance.
  - Efficient search
  - Efficient deletion
  - Efficient insertion.
- The binary search tree provides that structure.

The Binary Search Tree

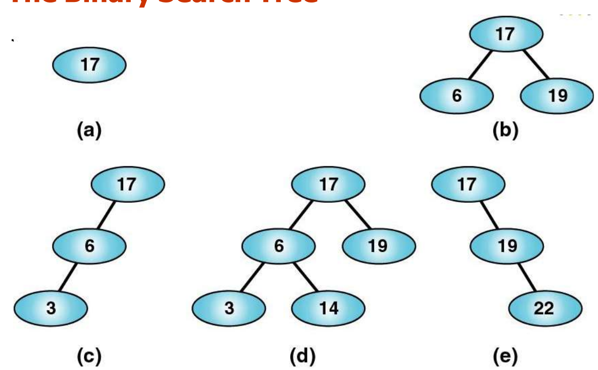
- A binary tree
- All items on the left subtree < the root
- All items in the right subtree >= the root
- Each subtree is itself a binary search tree.

The Binary Search Tree

- Each node of the tree
  - Usually a record
  - The key of the record is used to arrange the nodes in the required order

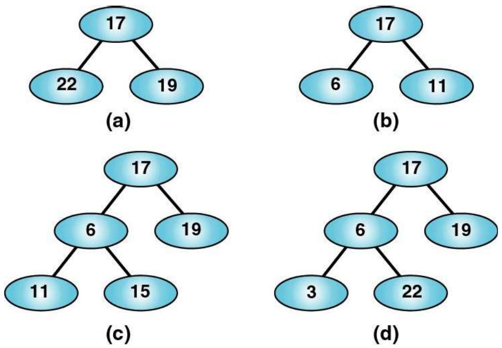


The Binary Search Tree



Binary search trees

The Binary Search Tree

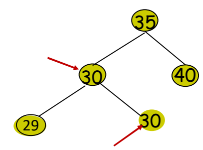


Are they binary search trees? (none of them are)

Searching a key in BST

- To search for **all the nodes** with a given key (K) in Binary Search Tree:
  - ➢We compare K with the root:
    - if K is less than (<) the root's key, we recur for the left subtree of the root node.
    - if K is greater than or equal to (≥) the root's key, then:
      - if K is present at the root, the root is concluded as one node found, and
      - we recur for right subtree of the root node.

Searching for the key 30

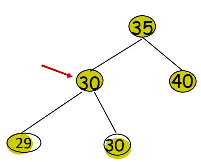


Two nodes found

Searching a key in BST

- To search for **one node** with a given key (K) in Binary Search Tree:
  - ➢We compare it with the root, if K is present at the root, the root is concluded as the node found. Otherwise:
    - if the key is less than (<) the root's key, we recur for the left subtree of the root node.
    - if the key is greater than (>) the root's key, we recur for right subtree of the root node.

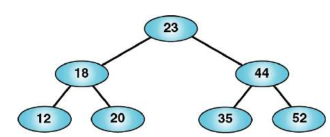
Searching for the key 30



One node found

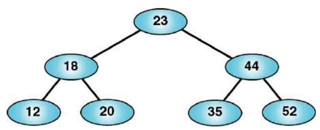
BST Insertion

- Insertion
  - Starting from the root, traverse the BST node by node in the following way until an empty subtree is located:
    - ✓ if the key to be inserted is less than (<) the node's key, we traverse the left subtree of the node.
    - ✓ if the key is greater than or equal to (≥) the node's key, we traverse the right subtree of the node
  - Insert the new node as the empty subtree encountered upon the traversal.

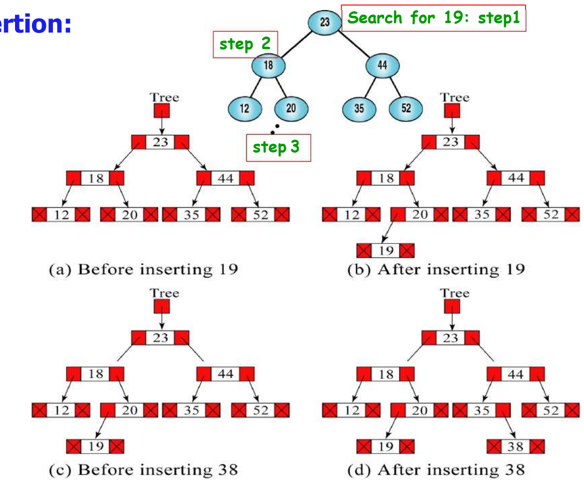


BST Insertion

- Insertion
  - All inserts take place at
    - a leaf node, or
    - a leaflike node--- a node having only one null branch

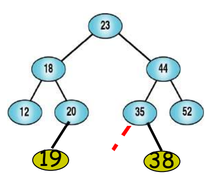


BST Insertion:



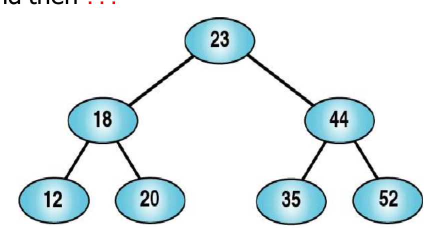
BST Insertion

- BST Insert
  - **How about inserting a duplicate 23?**



BST Deletion

- We need to locate the node to be deleted first
- And then ???



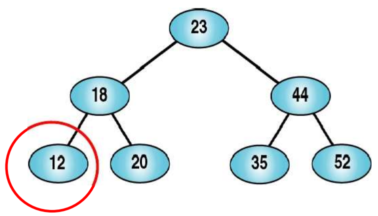
BST Deletion

FOUR possible cases:

① Node to be deleted has no children
② Node to be deleted has only a right subtree
③ Node to be deleted has only a left subtree
④ Node to be deleted has both left and right subtrees.

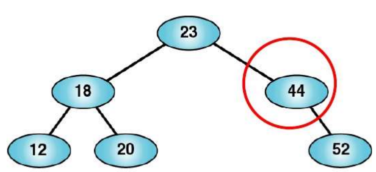
BST Deletion (case 1)

① Node to be deleted has no children
- Simply just delete it



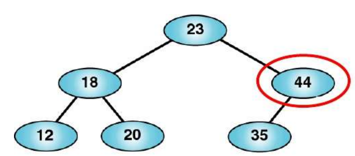
BST Deletion (case 2)

② Node to be deleted has only a right subtree
- Simply attach the node's only subtree to the parent of the node directly by replacing the node with the root of the subtree



BST Deletion (case 3)

③ Node to be deleted has only a left subtree
- Simply attach the node's only subtree to the parent of the node directly by replacing the node with the root of the subtree



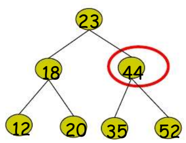
BST Deletion (case 4)

④ Node to be deleted d has both left and right subtrees

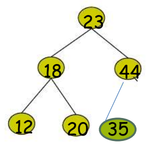
- Find the smallest node v in d's right subtree
- Recur to delete v
- Replace d by v

Example 1:

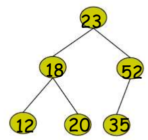
delete 44



Find the smallest node in 44's (d) right subtree: 52 (v)



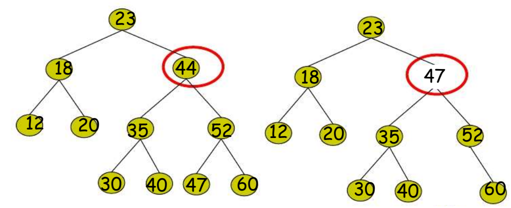
Recur to delete 52: case 1



Replace 44 by 52

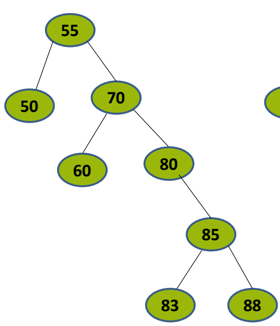
BST Deletion (case 4)

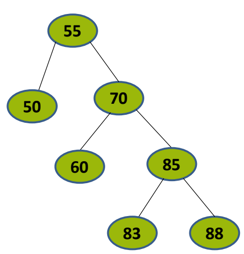
- Second example: delete 44



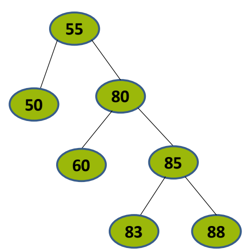
BST Deletion (case 4)

- Third example: delete 70





Recur to delete 80: case 3



Replace 70 by 80

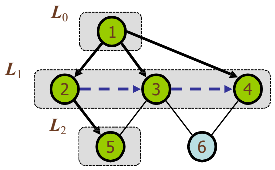
80 is the smallest node in 70s 'right subtree

Graph

- Graph Traversal
  - Breath-First Search
  - Depth-First Search

  These two graph traversal approaches form the basis for problem solving. Many methods for solving problems can be classified into these approaches.
- Finding Minimal Spanning Trees
  - Kruskal's Algorithm
  - Prim's Algorithm
- Finding Shortest Paths: Dijkstra's Algorithm

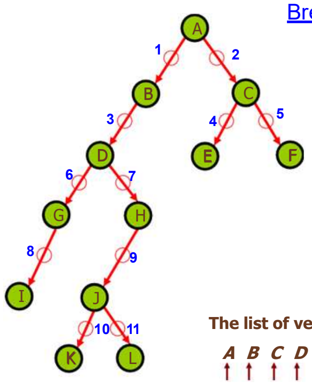
Breadth-First Search



Breadth-First Search

- Given a source vertex s, explores the edges to "discover" every vertex that is reachable from s.
- Expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier.
- Order that vertices are discovered is a "breadth-first tree" that contains all reachable vertices from s.

Example



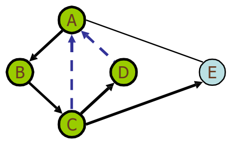
Breadth-First-Search (s)

1) visit s, label s as visited.
2) add s to a queue (a queue is a first-in-first out data structure) q.
3) while q is not empty:
   i. get the front value of q and store it as v
   ii. visit each unvisited vertex u, such that is v is adjacent to u, and add u to the queue q.
   iii. remove the front value of q

The list of vertices visited in order is:

A B C D E F G H I J K L

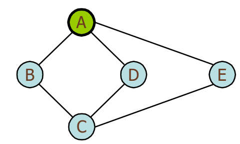
Depth-First Search



Depth-First Search

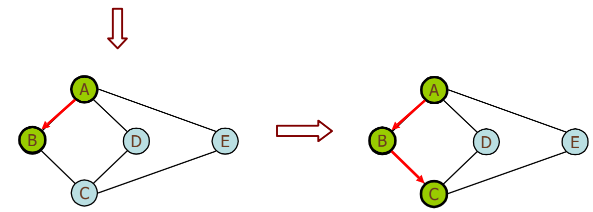
- Search “deeper” in the graph whenever possible
- Explores edges out of the most recently visited vertex *v* that still has unvisited neighbors.
- If all of *v*’s neighbors have been visited, “backtracks” to vertex from which *v* was visited.
- Continue process from there until we have visited all vertices reachable from original first vertex.

Example

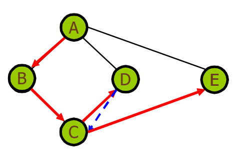


Depth-First-Search (v)

1) visit v, label v as visited.
2) For each unvisited vertex u, such that v is adjacent to u, visit it using Depth-First-Search on u.

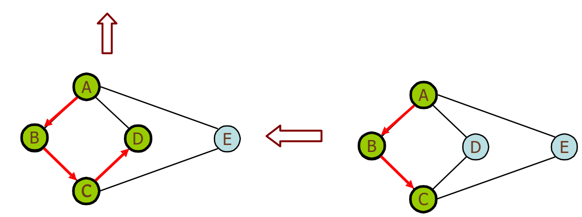


Example

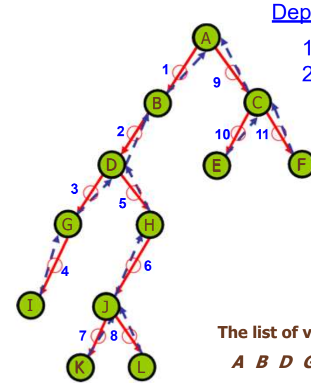


Depth-First-Search (v)

1) visit v, label v as visited.
2) For each unvisited vertex u, such that v is adjacent to u, visit it using Depth-First-Search on u.



Example



Depth-First-Search (v)

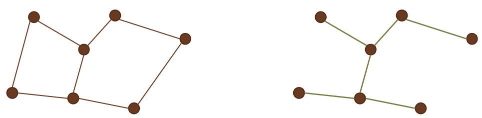
1) visit v, label v as visited.
2) For each unvisited vertex u, such that v is adjacent to u, visit it using Depth-First-Search on u.

The list of vertices visited in order is:

A B D G I H J K L C E F

Trees and Graphs

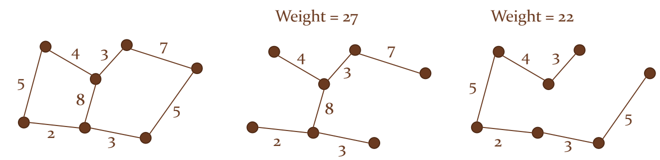
- Given any connected graph *G*, we can always find at least one tree which contains all of the vertices of *G* with a subset of its edges.
- E.g



- This is called a ***spanning tree***.
- Note: Usually, there are more than one spanning tree.

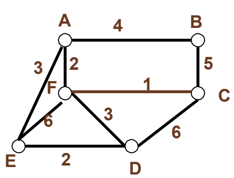
Minimal Spanning Tree

- 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.



- We call the spanning tree with the smallest weight the *minimal spanning tree*.
- We shall introduce two algorithms both designed using Greedy approach for finding minimal spanning tree:
  - Kruskal's Algorithm
  - Prim's Algorithm

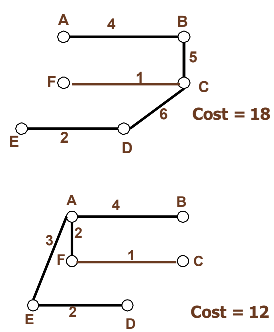
An Application of MST



Each node represents a city

Weight of each edge: cost of building a road connecting two cities

Problem: to build enough roads so that each pair of cities will be connected and to use the lowest cost possible

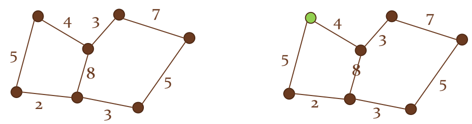


Prim's Algorithm -- One Vertex at a time

- Start with any vertex in the graph that has n vertices. This is our starting minimal spanning tree (MST).
- If the current MST does not have n-1 edges yet, then:
  - ✓add an edge of minimum weight that has one vertex in the MST and another vertex not in the MST.

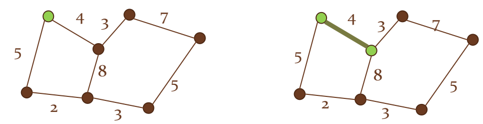
Finding MST using Prim's Algorithm

- Pick a vertex



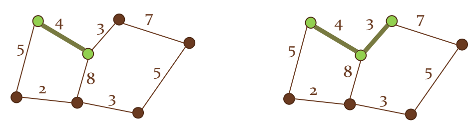
Finding MST using Prim's Algorithm

- Add an edge with min weight that introduces a new vertex



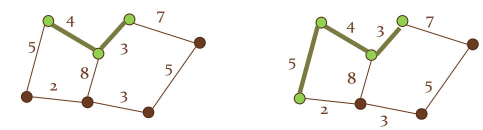
Finding MST using Prim's Algorithm

- Add an edge with min weight that introduces a new vertex



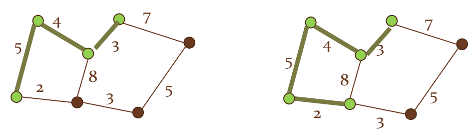
Finding MST using Prim's Algorithm

- Add an edge with min weight that introduces a new vertex



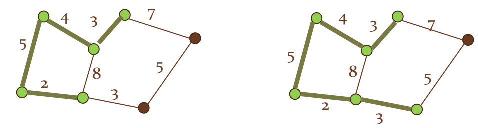
Finding MST using Prim's Algorithm

- Add an edge with min weight that introduces a new vertex



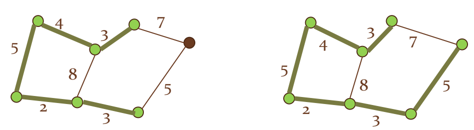
Finding MST using Prim's Algorithm

- Add an edge with min weight that introduces a new vertex



Finding MST using Prim's Algorithm

- Add an edge with min weight that introduces a new vertex



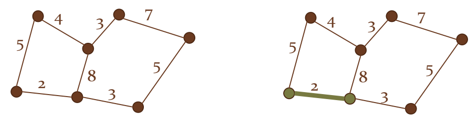
- And we are done.
  - It doesn't matter which vertex we start with.

Kruskal's Algorithm: One Edge at a time

- Start with the shortest edge in the graph that has n vertices. This is our starting minimal spanning tree (MST).
- If the current MST does not have n-1 edges yet, then:
  ✓add an edge of minimum weight that will not form cycle.

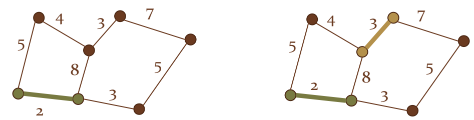
Finding MST using Kruskal's Algorithm

• Pick the shortest edge



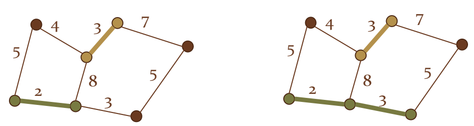
Finding MST using Kruskal's Algorithm

• Add a new edge with lowest weight that will not introduces cycle



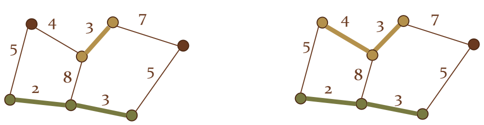
Finding MST using Kruskal's Algorithm

• Add an new edge with lowest weight that will not introduces cycle



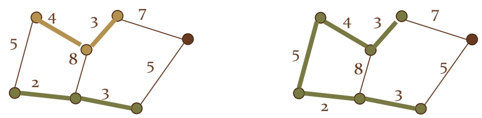
Finding MST using Kruskal's Algorithm

• Add a new edge with lowest weight that will not introduces cycle



Finding MST using Kruskal's Algorithm

- Add a new edge with lowest weight that will not introduces cycle

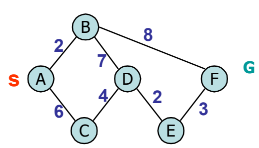


Finding MST using Kruskal's Algorithm

- Add a new edge with lowest weight that will not introduces cycle

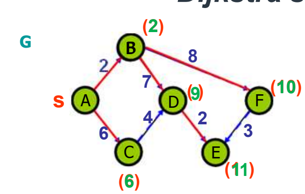
- And, we are done.

Shortest Path Problem

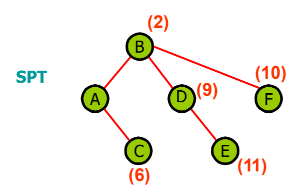


- **Find shortest paths from a given vertex s to all the other vertices in a given connected graph**
- **Dijkstra's Algorithm can be used to find the shortest paths from a connected graph**
- **Dijkstra's Algorithm is designed using Greedy approach**
- **Applications of shortest paths include finding shortest routes for driving, etc.**

Dijkstra's Algorithm



Input: weighted connected graph (G) with n vertices and non-negative weights; and a vertex (s) in G

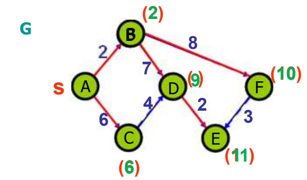


Outline of Dijkstra's Algorithm (G, s)

1) **Add s to an empty SPT to form a path from s to s of length 0.**
2) **If the number of the edges of the SPT is less than n-1, keep growing the SPT by repeatedly adding an edge connecting to a vertex not in the SPT yet, that can extend a path from s in the SPT as short as possible.**

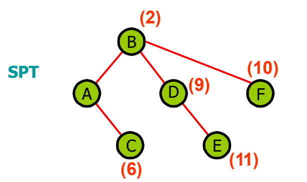
SPT always remains as a tree when Dijkstra's Algorithm runs

Dijkstra's Algorithm



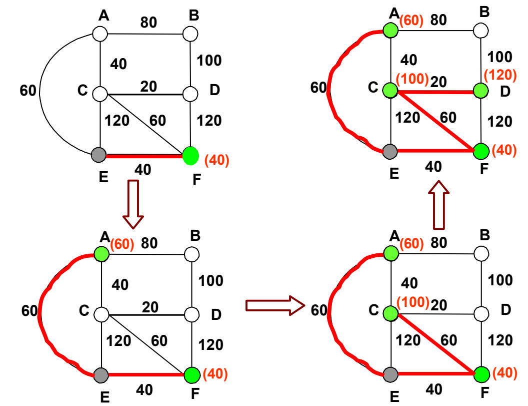
Important: Note that the paths in the table must be ordered according to order added by Dijkstra's Algorithm

Shortest Path	Length
A	0
A, B	2
A, C	6
A, B, D	9
A, B, F	10
A, B, D, E	11



SPT always remains as a tree when Dijkstra's Algorithm runs

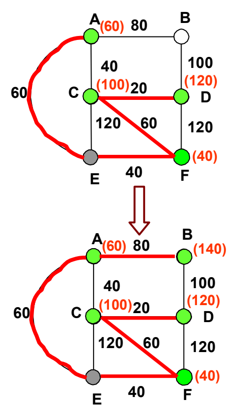
Example



Shortest Path	Length
E	0
E, F	40
E, A	60
E, F, C	100
E, F, C, D	120

Example

Shortest Path	Length
E	0
E, F	40
E, A	60
E, F, C	100
E, F, C, D	120
E, A, B	140

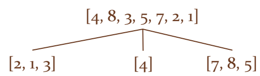


Quicksort and Trees

- 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]

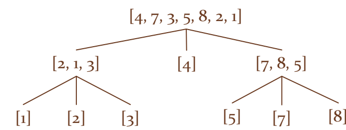
Quicksort and Trees

- First Partition



Quicksort and Trees

- 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		

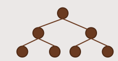
Refining the question

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

n	tree	depth
2		1
4		
8		

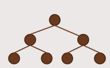
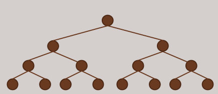
Refining the question

- 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. Hence, $k = \log_2(N)$.
- 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 the head has not reached the end and 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 the tail has not reached the beginning and 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
   - Go to step 1

Partitioning in action

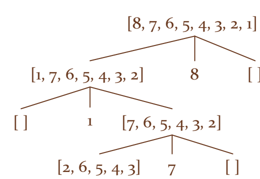
- Start:[4, 8, 3, 5, 7, 2, 1] (head = 8, tail =1)

- 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] (head = 3, tail =2)
- Step 1. compare 4 and 3
  - 3 < 4 so move head and repeat step 1
  - [4, 1, 3, 5, 7, 2, 8] (head = 5)
- Step 1. compare 4 and 5
  - 5>4 so go to step 2
- 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] (head = 7, tail =7)
- 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] (tail = 2)
- 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]

Q

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:



Q

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, the middle and the 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