## TREE PROPERTIES:

**Theorem:** Every tree with at least two vertices has a leaf.

**Proof:** Suppose for a contradiction that there exists a tree T with at least two vertices that does not contain a leaf. Since T must be connected, we know no vertex has degree 0, and therefore every vertex of T must have a degree at least 2. But then by Theorem 2.5 we know T must have a cycle, which contradicts that T is acyclic. Thus T must contain a leaf.

**Lemma:** Given a tree T with a leaf v, the graph T − v is still a tree.

**Proof:** Removing a leaf from a tree will always remove exactly one vertex and one edge, creating a tree with a smaller size. This technique naturally lends itself to induction arguments.

**Theorem:** A tree with n vertices has n − 1 edges for all n ≥ 1.

**Proof:** Argue by induction on $n$, the number of vertices in $T$. If $n = 1$ then $T$ cannot have any edges since any edge in a graph with one vertex must be a loop, which would create a cycle.

Suppose for some $k \geq 1$ that any tree with $k$ vertices has $k - 1$ edges and let $T$ be a tree with $k + 1$ vertices. Since $k + 1 \geq 2$, by Theorem 3.4 we know $T$ must have at least one leaf, call it $v$. Then by Lemma 3.5 we know $T - v$ is a tree with $k$ vertices. Thus by the induction hypothesis $T - v$ must have $k - 1$ edges. Since $v$ is a leaf, its removal from $T$ deleted exactly one edge and so $T$ must have $k$ edges.

Therefore, for all $n \geq 1$ any tree with $n$ vertices must have $n - 1$ edges.

- The corollary below on the total degree of a tree is an immediate consequence of the above Theorem.

**Corollary**: The total degree of a tree on n vertices is 2n-2.

**Proposition:** Let T be a tree. Then for every pair of distinct vertices x and y there exists a unique x − y path.

**Proposition:** Every tree is minimally connected, that is the removal of any edge disconnects the graph.

**Proposition:** If any edge e is added to a tree T then T +e contains exactly one cycle.

**Theorem 3.11** Let $T$ be a graph with $n$ vertices. The following conditions are equivalent:

(a) $T$ is a tree.

(b) $T$ is acyclic and contains $n-1$ edges.

(c) $T$ is connected and contains $n-1$ edges.

(d) There is a unique path between every pair of distinct vertices in $T$.

(e) Every edge of $T$ is a bridge.

(f) $T$ is acyclic and for any edge $e$ from $T$, $T+e$ contains exactly one cycle.

- Using these results about the overall properties of trees, we can now prove why Kruskal's Algorithm is guaranteed to find a minimum spanning tree.

**Theorem:** Kruskal's Algorithm produces a minimum spanning tree.

**TREE ENUMERATION:**

- We discussed how a degree sequence can help determine the structure of a graph, though it may not be unique.
- As it turns out, there is a similar result for trees, but one in which the tree can be uniquely determined from the sequence given.
- Instead of focusing on degrees, this sequence, called a Prüfer sequence, uses the location of leaves within a tree.
- These sequences were introduced in 1918 by the German mathematician Ernst Paul Heinz Prüfer as a means for proving Cayley's Theorem (see Theorem 3.14)[70].

**Prüfer Sequence:**

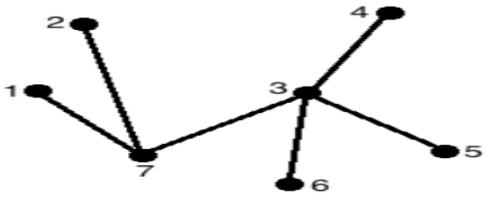
**Definition 3.13** Given a tree $T$ on $n > 2$ vertices (labeled $1, 2, \ldots, n$), the ***Prüfer sequence*** of $T$ is a sequence $(s_1, s_2, \ldots, s_{n-2})$ of length $n-2$ defined as follows:

- Let $l_1$ be the leaf of $T$ with the smallest label.
- Define $T_1$ to be $T - l_1$.
- For each $i \geq 1$, define $T_{i+1} = T_i - l_{i+1}$, where $l_{i+1}$ is the leaf with the smallest label of $T_i$.
- Define $s_i$ to be the neighbor of $l_i$.

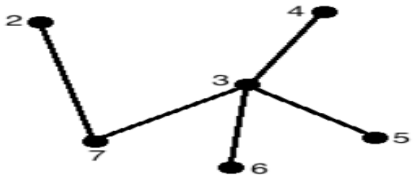
**Step to Construct Prüfer Sequence:**

- For a tree graph G on n labeled vertices, the Prüfer Sequence of G is a sequence containing n−2 entries of {1,2,...,n} labeled vertices that is unique to G.
- It is constructed by deleting the leaf in the graph with the smallest label and making the first entry in the sequence the vertex label that is adjacent to that vertex.
- The second entry of the Prüfer sequence comes from deleting the leaf with the second smallest label and making the second entry in the sequence the vertex label adjacent to that vertex, and so forth.

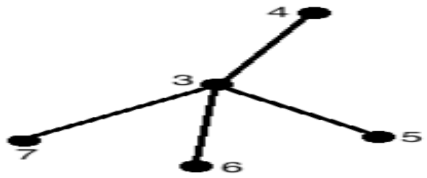
**Example:** For example, let's look at the following labelled graph:



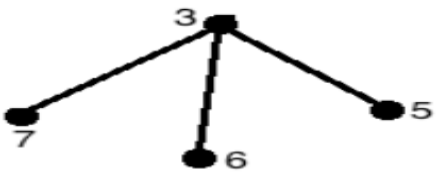

Step 1. The leaf with the smallest label is 1, so let's delete that vertex and put 7 as the first entry in our sequence since 7 was adjacent to 1. So far our sequence is (7) and our graph is as follows:




Step 2. The next leaf with the smallest label is 2. It's neighbour is 7 as well, so our sequence so far is (7,7) and when we delete leaf 2, we obtain the following graph:

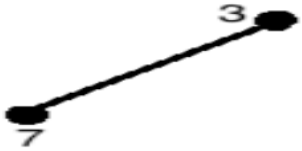


Step 3. The next leaf with the smallest label is 4, which is adjacent to 3, so our sequence is now (7,7,3). Deleting the leaf we obtain the following graph:




Step 4. The next leaf with the smallest label is 5 which is adjacent to 3, so our sequence is (7,7,3,3). Deleting the leaf we obtain the following graph:


Step 5. And lastly, the leaf with the smallest label is 6, which is also adjacent to 3, so our sequence is (7,7,3,3,3). We are now done with the following graph:



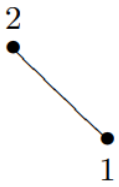

So our Prüfer Sequence for the original graph is (7,7,3,3,3).

- In the next example, we draw the graph by using s Prüfer equence.

1) Draw the tree whose Prüfer code is $(1, 1, 1, 1, 6, 5)$.

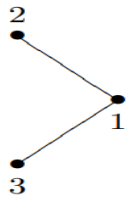
**Solution:** The given Prüfer code has six entries, therefore the corresponding tree will have $6 + 2 = 8$ entries.

The first number in the Prüfer code is 1 and the lowest number not included in the Prüfer code is 2, so we connect 1 to 2.



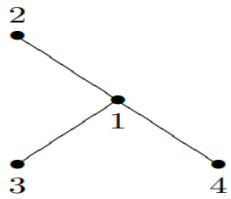

We then drop the leading 1 in the code and put 2 at the back of the code: $(1, 1, 1, 6, 5, 2)$.

The first number in the code is still 1 and the lowest number not included is now 3, so we connect 1 to 3.



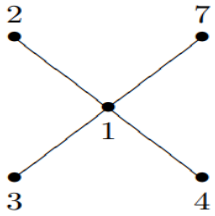

We, again, drop the leading 1 and then put 3 at the back of the code: $(1, 1, 6, 5, 2, 3)$.

The first number in the code is still 1 and the lowest number not included is now 4, so we connect 1 to 4.



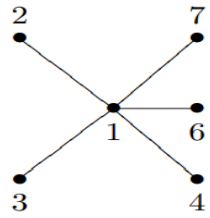

We drop the leading 1 and put 4 at the back of the code: $(1, 6, 5, 2, 3, 4)$.

The first number in the code is, yet again, 1 and the lowest number not included is now 7. So we connect 1 to 7.



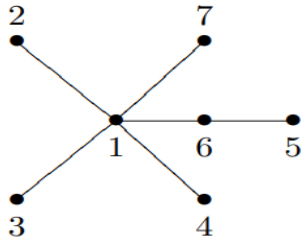

We drop the leading 1 an put 7 at the back of the code: $(6, 5, 2, 3, 4, 7)$.

The first number in the code is now 6 and the lowest number not included is 1, so we connect 6 to 1.



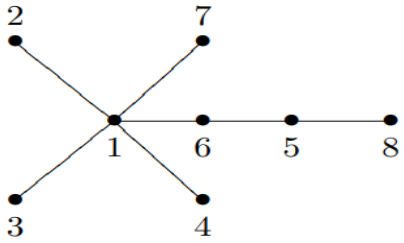

We drop the 6 from the code and put 1 at the back: $(5, 2, 3, 4, 7, 1)$.

Now the first number in the code is 5 and the lowest number not included is 6, so we connect 5 to 6.




We drop the 5 from the code and put 6 at the back: $(2, 3, 4, 7, 1, 6)$.

We have iterated all the way through the code and the two numbers missing are 5 and 8. So we connect them.




And we are done.

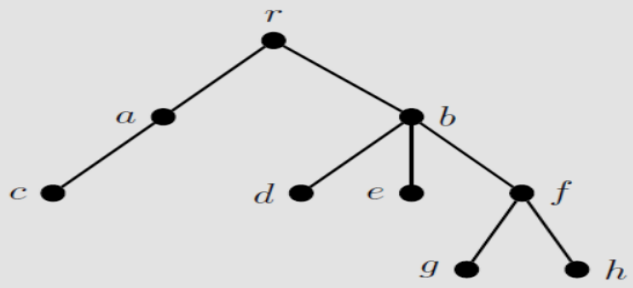
**Cayley's Theorem:** There are $n^{n-2}$ different labeled trees on $n$ vertices.

## ROOTED TREES:

- What would we mean by a root of a tree? We can think of the root as the place from which a tree grows.

**Definition 3.15** A ***rooted tree*** is a tree $T$ with a special designated vertex $r$, called the ***root***. The ***level*** of any vertex in $T$ is defined as the length of its shortest path to $r$. The ***height*** of a rooted tree is the largest level for any vertex in $T$.

**Example 3.6** Find the level of each vertex and the height of the rooted tree shown below.




*Solution:* Vertices $a$ and $b$ are of level 1, $c, d, e$, and $f$ of level 2, and $g$ and $h$ of level 3. The root $r$ has level 0. The height of the tree is 3.

Most people have encountered a specific type of rooted tree: a family tree. In fact, much of the terminology for rooted trees comes not from a plant version of a tree but rather from genealogy and family trees.

- The root of a family tree would be the person for whom the descendants are being mapped and the level of a vertex would represent a generation; see the tree below.

**Definition 3.16** Let $T$ be a tree with root $r$. Then for any vertices $x$ and $y$

- $x$ is a ***descendant*** of $y$ if $y$ is on the unique path from $x$ to $r$;
- $x$ is a ***child*** of $y$ if $x$ is a descendant of $y$ and exactly one level below $y$;
- $x$ is an ***ancestor*** of $y$ if $x$ is on the unique path from $y$ to $r$;
- $x$ is a ***parent*** of $y$ if $x$ is an ancestor of $y$ and exactly one level above $y$;
- $x$ is a ***sibling*** of $y$ if $x$ and $y$ have the same parent.

- A common type of tree, especially in computer science, adds the restriction that no vertex can have more than two children. You can think of this as a decision tree, where at every stage you can answer yes or no (or for the more technically inclined this would correspond to 1's and 0's). These trees are called binary trees.

**Definition 3.17** A tree in which every vertex has at most two children is called a ***binary tree***. If every parent has exactly two children we have a ***full binary tree***. Similarly, if every vertex has at most $k$ children then the tree is called a ***k-nary tree***.

## Depth-First Search Tree:

- The main idea behind a depth-first tree is to travel along a path as far as possible from the root of a given graph. If this path does not encompass the entire graph, then branches are built off this central path to create a tree.
- The formal description of this algorithm relies on an ordered listing of the neighbors of each vertex and uses this order when adding new vertices to the tree.

**Depth-First Search Tree**

Input: Simple graph $G = (V, E)$ and a designated root vertex $r$.

Steps:

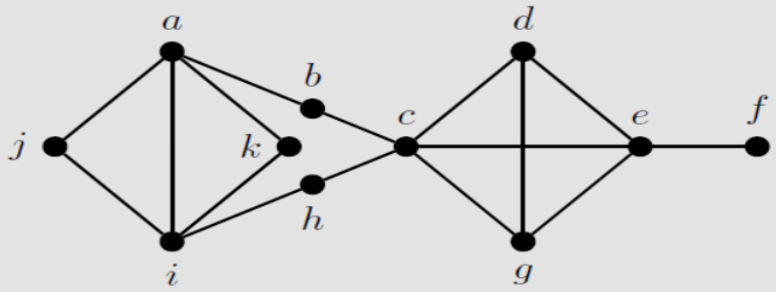
1. Choose the first neighbor $x$ of $r$ in $G$ and add it to $T = (V, E')$.
2. Choose the first neighbor of $x$ and add it to $T$. Continue in this fashion—picking the first neighbor of the previous vertex to create a path $P$. If $P$ contains all the vertices of $G$, then $P$ is the depth-first search tree. Otherwise continue to Step (3).
3. Backtrack along $P$ until the first vertex is found that has neighbors not in $T$. Use this as the root and return to Step (1).

Output: Depth-first search tree $T$.

**Remarks:**

- In creating a depth-first search tree, we begin by building a central spine from which all branches originate.
- These branches are as far down on this path as possible.
- The resulting rooted tree is often of large height and is more likely to have more vertices at the lower levels.

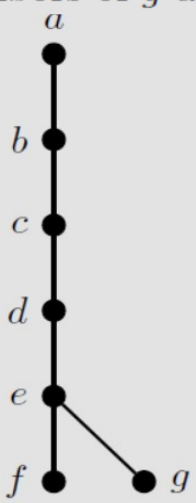
**Example 3.8** Find the depth-first search tree for the graph below with the root $a$.



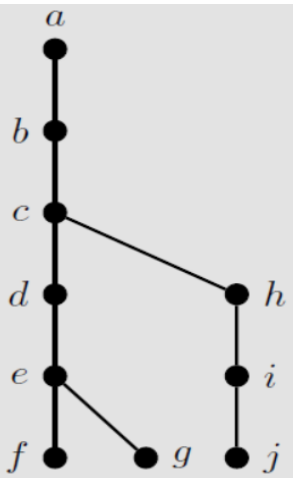
*Solution:*

Step 1: Since $a$ is the root, we add $b$ as it is the first neighbor of $a$. Continuing in this manner produces the path shown below. Note this path stops with $f$ since $f$ has no further neighbors in $G$.

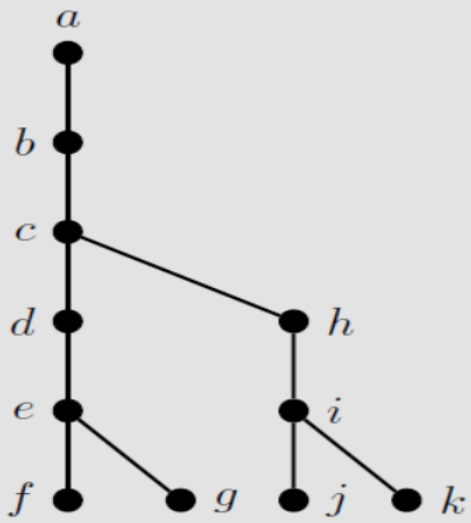

Step 2: Backtracking along the path above, the first vertex with an unchosen neighbor is $e$. This adds the edge $eg$ to $T$. No other edges from $g$ are added since the other neighbors of $g$ are already part of the tree.




Step 3: Backtracking again along the path from Step 1, the next vertex with an unchosen neighbor is $c$. This adds the path $c\,h\,i\,j$ to $T$.




Step 4: Backtracking again along the path from Step 3, the next vertex with an unchosen neighbor is $i$. This adds the edge $ik$ to $T$ and completes the depth-first search tree as all the vertices of $G$ are now included in $T$.




Output: The tree above is the depth-first search tree.

- ✓ Note that the tree created above has height 5 and with one vertex each at level 1 and 2, two vertices each at level 3 and 4, and four vertices at level 5.

## Breadth-First Search Tree:

- The main objective for a breadth-first search tree is to add as many neighbors of the root as possible in the first step.
- At each additional step, we are adding all available neighbors of the most recently added vertices.

**Breadth-First Search Tree**

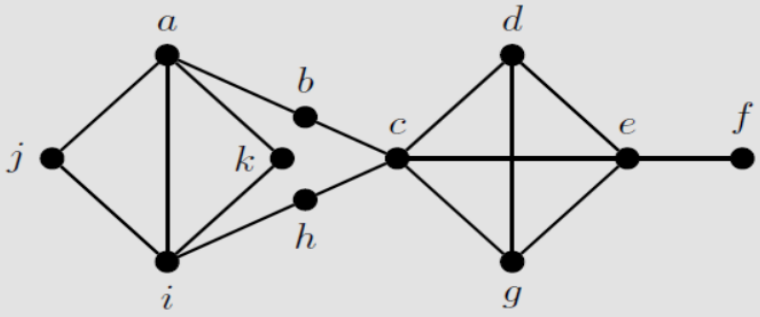
Input: Simple graph $G = (V, E)$ and a designated root vertex $r$.

Steps:

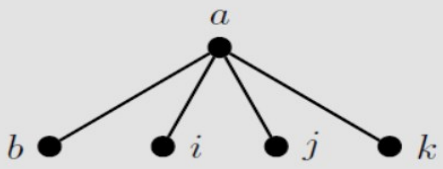
1. Add all the neighbors of $r$ in $G$ to $T = (V, E')$.
2. If $T$ contains all the vertices of $G$, then we are done. Otherwise continue to Step (3).
3. Beginning with $x$, the first neighbor of $r$ that has neighbors not in $T$, add all the neighbors of $x$ to $T$. Repeat this for all the neighbors of $r$.
4. If $T$ contains all the vertices of $G$, then we are done. Otherwise repeat Step (3) with the vertices just previously added to $T$.

Output: Breadth-first tree $T$.

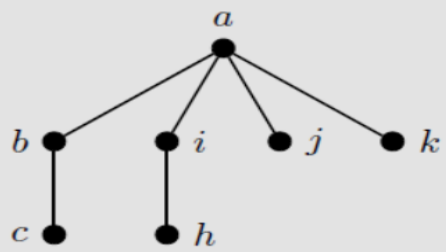
**Example 3.9** Find the breadth-first search tree for the graph below with the root $a$.



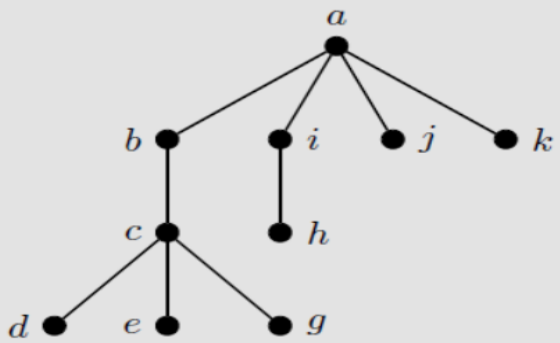

Step 1: Since $a$ is the root, we add all of the neighbors of $a$ to $T$.



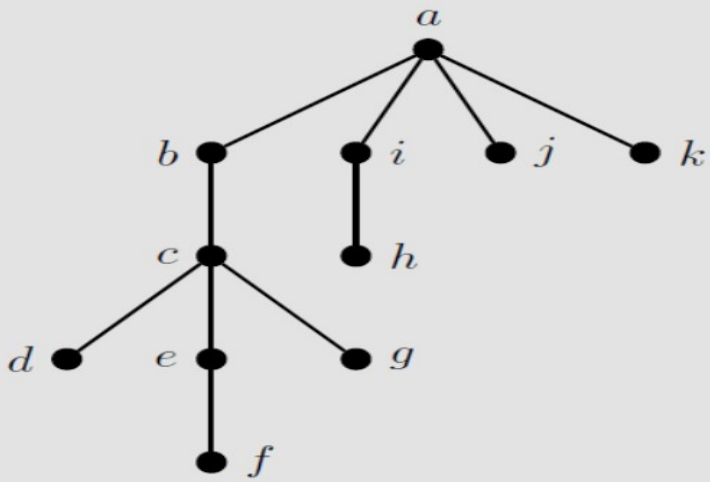

Step 2: We next add all the neighbors of $b, i, j$, and $k$, that are not already in $T$, beginning with $b$ as it is the first neighbor of $a$ that was added in Step 1. This adds the edge $bc$. Moving to $i$ we add the edge $ih$. No other edges are added since $j$ and $k$ do not have any unchosen neighbors.




Step 3: We next add all the neighbors of $c$ not in $T$, namely $d, e$, and $g$. No other vertices are added since all the neighbors of $h$ are already part of the tree $T$.




Step 4: Since $d$ has no unchosen neighbors, we move ahead to adding the unchosen neighbors of $e$. This completes the breadth-first search tree as all the vertices of $G$ are now included in $T$.




Output: The tree above is the breadth-first search tree.

- ✓ It should come as no surprise that breadth-first search trees are likely to be of shorter height than their depth-first search counterpart.
- ✓ The breadth- first search tree created above has height 4 with four vertices on level 1, two vertices on level 2, three vertices on level 3, and one on level 4.

**Conclusion:**

The main difference between these two algorithms is that depth-first focuses on traveling as far into the graph in the beginning, whereas the breadth-first focuses on building outward using neighborhoods.