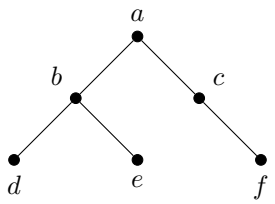
2.3 Representing Trees

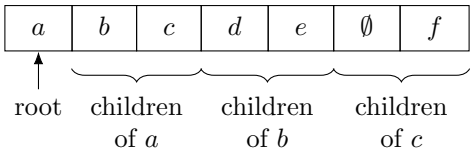
Just like with graphs, there are a variety of different methods to represent trees. Since trees are a fundamental computer data structure, choosing the most appropriate way to represent a tree in memory or on a disk is vital if we want to optimize measures like access time or data retrieval.

The first class of tree representation methods use **linear** storage. Since rooted trees have a natural hierarchical property, we can represent the vertices of a tree quite easily in an array or a list. The array representation of a tree uses the parent/child relationship to establish at which index a vertex is to be located and allows for direct access to vertices. The list-of-lists method, on the other hand, relies heavily on recursion. The list-of-lists method is more similar to the adjacency list representation of a graph, but with additional nesting.

Example 14. Consider the following binary tree:



This tree can be stored in an array representation as follows:

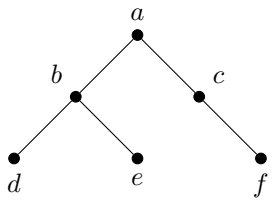


Alternatively, we may store the same tree in a list-of-lists representation as follows:

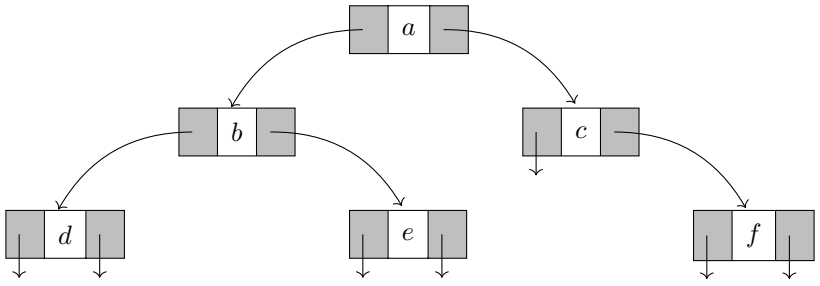
$$[a, [b, [d, [\emptyset], [\emptyset]], [e, [\emptyset], [\emptyset]]], [c, [\emptyset], [f, [\emptyset], [\emptyset]]]]$$

The second class of tree representation methods use **nonlinear** storage. Nonlinear storage not only gives us the advantage of not having to allocate a large block of contiguous memory, but it also has a similar structure to a tree itself: blocks of memory (vertices) connected by pointers (edges).

Example 15. Consider again the tree from Example 14:



This tree can be stored in a doubly-linked list (a form of nonlinear storage) as follows:



3 Tree Traversals

Now that we are familiar with trees and tree representations, we can consider methods of retrieving data stored in a tree. We call such methods **tree traversals**. Unlike arrays and lists, where we have one straightforward way to traverse the data within the structure, we can traverse data within a tree in more than one way. However, to maximize efficiency, we should strive to avoid visiting vertices more than once in a traversal; that is, we should only visit a vertex in a given tree exactly once during a traversal.

Tree traversal methods can be split into two categories: **depth-first** methods and **breadth-first** methods. Depth-first methods focus on traversing as deep within the tree as possible before moving to another part of the tree, while breadth-first methods visit all vertices on a given level before moving to a deeper level within the tree.

In what follows, we will only consider binary trees. However, each of the methods presented in this section can be generalized to m-ary trees.

We will also use a mnemonic system to remember the order of vertices visited for each tree traversal method. The letter V denotes the current vertex, the letter L denotes the left subtree of the current vertex, and the letter R denotes the right subtree of the current vertex.

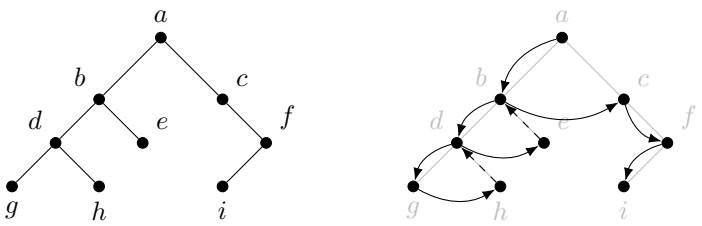
3.1 Pre-order Traversal

In a pre-order traversal of a tree T, we visit the root of T first before descending into the left subtree and right subtree of the root, respectively. A pre-order traversal is a depth-first tree traversal, since we descend as far down into the left subtrees as possible before visiting the right subtrees. In our mnemonic system, the traversal order is VLR.

We can represent a pre-order traversal algorithm in pseudocode as follows:

```
preorder(T):
  if the root of T is empty:
    print nothing
  else:
    print the root of T
    preorder(left subtree of T)
    preorder(right subtree of T)
```

Example 16. Consider the following binary tree T:



A pre-order traversal of T gives the sequence of vertices $[a, b, d, g, h, e, c, f, i]$.

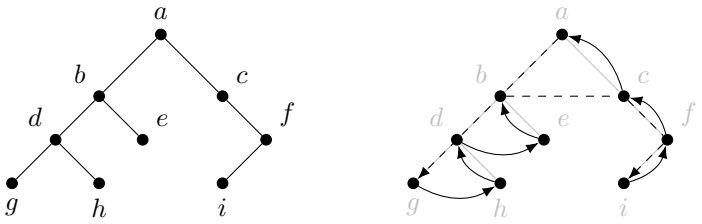
3.2 Post-order Traversal

A post-order traversal is, in a sense, the "opposite" of a pre-order traversal. In a post-order traversal of a tree T, we begin by visiting the left and right subtrees of the root of T before visiting the root itself. A post-order traversal is a depth-first traversal, by a similar line of reasoning as the pre-order traversal. In our mnemonic system, the traversal order is LRV.

We can represent a post-order traversal algorithm in pseudocode as follows:

```
postorder(T):
  if the root of T is empty:
    print nothing
  else:
    postorder(left subtree of T)
    postorder(right subtree of T)
    print the root of T
```

Example 17. Consider the same binary tree T from Example 16:



A post-order traversal of T gives the sequence of vertices $[g, h, d, e, b, i, f, c, a]$.

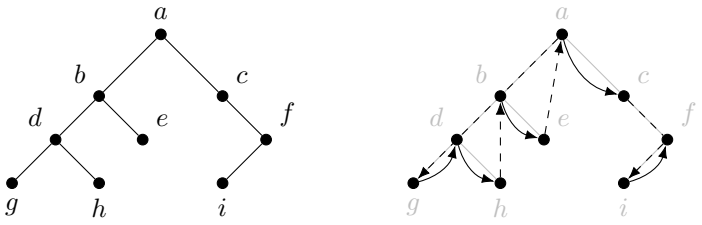
3.3 In-order Traversal

If we imagine the vertices of a binary tree as being ordered from left to right, where the leftmost vertex of the tree is the "first vertex" and the rightmost vertex of the tree is the "last vertex", then we can develop a tree traversal method that preserves the left-to-right order of visiting vertices. The in-order traversal of a tree T goes as far left in the tree as possible before visiting a vertex. It then visits the parent of the left subtree before descending into the right subtree. Thus, an in-order traversal is a depth-first traversal. In our mnemonic system, the traversal order is LVR.

We can represent an in-order traversal algorithm in pseudocode as follows:

```
inorder(T):
  if the root of T is empty:
    print nothing
  else:
    inorder(left subtree of T)
    print the root of T
    inorder(right subtree of T)
```

Example 18. Consider again the binary tree T from Examples 16 and 17:



An in-order traversal of T gives the sequence of vertices $[g, d, h, b, e, a, c, i, f]$.

3.4 Level-order Traversal

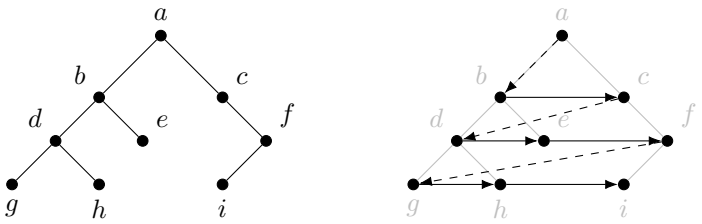
The level-order traversal is the only example of a breadth-first traversal we will see in this section. This method is a breadth-first traversal because it visits each vertex on a given level before descending further into the tree.

In order to keep track of vertices that we have yet to visit, we must use an auxiliary data structure. The natural choice in this scenario is a queue data structure, since we can add new vertices to the queue as we encounter them and remove vertices from the queue in the order in which they were added.

We can represent a level-order traversal algorithm in pseudocode as follows:

```
levelorder(T):
  add the root of T to queue Q
  while queue Q is not empty:
    dequeue vertex V from queue Q
    print V
    enqueue left child of V to queue Q
    enqueue right child of V to queue Q
```

Example 19. Consider once more the binary tree T from Examples 16, 17, and 18:



A level-order traversal of T gives the sequence of vertices $[a, b, c, d, e, f, g, h, i]$.

3.5 Applications of Traversals

Depending on what we wish to do with a tree, we may prefer to use one tree traversal method over another.

A pre-order traversal of a tree allows us to perform actions where we need to read the data in a parent vertex before the children vertices. For instance, given both a pre-order traversal and an in-order traversal, we may duplicate a tree. We can do so by comparing the positions of vertices in both traversal sequences.

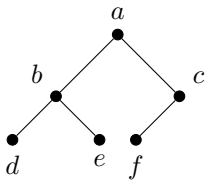
Example 20. Suppose some binary tree has a pre-order traversal sequence $[a, b, d, e, c, f]$ and an in-order traversal sequence $[d, b, e, a, f, c]$. What is the structure of the binary tree?

First, observe that the initial element of the pre-order traversal sequence is the root of the tree. Thus, all other elements in the pre-order traversal sequence are children of the vertex a.

The in-order traversal sequence tells us in which subtree of the vertex a a given vertex belongs. Since vertices d, b, and e come before a in the in-order traversal sequence, each of these vertices are in the left subtree of a. Likewise, vertices f and c are in the right subtree of a.

From here, we can perform the same analysis on the "left subtree" subsequences $[b, d, e]$ and $[d, b, e]$, which reveal that b is the root of the left subtree, d is the left child of b, and e is the right child of b. Likewise, given the "right subtree" subsequences $[c, f]$ and $[f, c]$, we see that c is the root of the right subtree and f is the left child of c.

Altogether, we conclude that the structure of the binary tree is as follows:



A post-order traversal of a tree allows us to perform actions where we must operate on children vertices before the parent vertex. For instance, using a post-order traversal, we may delete a tree from the memory of a computer. It is vital that we delete vertices from the bottom-up, because if we delete a parent vertex before its children, then we lose references to the children vertices (and, therefore, leak memory that cannot be freed).

Finally, a **binary search tree** is a binary tree where each element is stored according to some order; for example, given some vertex u storing a value d, all values less than d are stored in vertices in the left subtree of u and all values greater than d are stored in vertices in the right subtree of u.

Unlike with general binary trees, it is possible to reconstruct a binary search tree using *only* the pre-order traversal sequence of the tree's vertices. We can do so by comparing the vertex to be inserted to the parent vertex. If the vertex contains a value smaller than that of the parent vertex, then we place it into the left subtree. If the vertex contains a value that is larger than that of the parent vertex, but smaller than that of the grandparent vertex, then we place the vertex to be inserted into the right subtree. Otherwise, we move up one level in the tree and check again.

We can also "collapse" the data stored in a binary search tree to one dimension by performing an in-order traversal. Doing so allows us to store the data (still in sorted order) in an array or list.

4 Spanning Trees

At this point, we know that trees are special kinds of graphs. But what can we say about the reverse relationship between these two structures; namely, how can we convert a graph into a tree? Of course, the answer seems clear: just delete enough edges from a graph to make it acyclic, but don't delete so many edges as to make the graph disconnected.

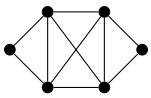
Naturally, this "clear" answer may leave you wanting. What steps must we take exactly to obtain a tree from a graph? And how do we know when we have done enough to obtain a tree? In this section, we investigate these questions and more, but first we must define the object of our study. The tree-from-a-graph that we will study here is known as a **spanning tree**.

Definition 21 (Spanning tree)**.** Given a connected graph G, a spanning tree of G is a tree T where T is a subgraph of G and T includes every vertex in G.

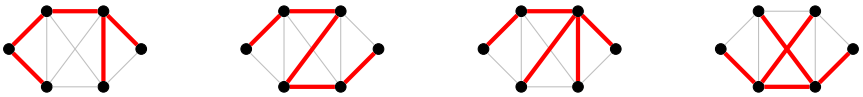
A spanning tree gets its name from the fact that it spans the vertices of a graph. Similarly, we may define **spanning forests** for a disconnected set of trees, but we will not discuss that notion in as much depth here.

A spanning tree can be defined in terms of the vertices or the edges of a graph. In our definition, we focused on vertices. Alternatively, if we wish to focus on edges, then we say that a spanning tree is a maximal subset of edges of a graph that contains no circuit.

Example 22. Consider the following graph:



Some spanning trees contained in this graph are as follows. (This is not a complete list!)



Observe that we don't refer to a spanning tree of a graph as *the* spanning tree. This is because spanning trees need not be unique; a graph may contain many different spanning trees, and there are methods of counting the number of spanning trees a given graph contains. For general graphs, Kirchhoff's matrix-tree theorem, named for the German physicist Gustav Kirchhoff, allows us to calculate the number of spanning trees in a graph from the determinant of a matrix derived from the graph. (This theorem is a little outside the scope of this course, so we won't present it here.)

In addition to counting spanning trees themselves, we can count different measures on a spanning tree. For instance, as a consequence of Theorem 11, we know that any spanning tree in a graph with n vertices must contain exactly $n-1$ edges. Moreover, from Theorem 9, we know that adding any edge to a spanning tree will create a circuit. We call each of these circuits a **fundamental cycle**, and the study of such circuits leads to many more interesting graph theory results.

Before we continue, it is worth mentioning a small result that relates spanning trees to a certain property of their associated graph.

Theorem 23. *A graph G is connected if and only if G contains a spanning tree.*

Proof. ($\Rightarrow$): Suppose G is connected. If G is acyclic, then G is its own spanning tree and we are done. Otherwise, G contains at least one circuit. Select an arbitrary circuit in G and remove one edge e from the circuit. The resultant subgraph $G-e$ contains one fewer edge, but it remains connected. Repeat this process until no circuits remain. The resultant subgraph is a spanning tree of G.

($\Leftarrow$): Suppose G contains a spanning tree T. By definition, T includes every vertex of G. Moreover, since T is a tree, we know by Theorem 9 that there exists exactly one path between every pair of vertices in T. Hence, there also exists a path between every pair of vertices in G, so G is connected. □

4.1 Constructing Spanning Trees

Now comes the big question: how do we construct a spanning tree? We could use the technique given in the proof of Theorem 23, but the "taking away edges" approach is inefficient due to the fact that we must repeatedly identify circuits in the given graph. Instead, we can devise a "building up edges" approach. Just like with tree traversals, we have two different techniques we can use for this approach: we may construct a spanning tree of a graph in either a depth-first or a breadth-first manner.

In either case, we begin the construction process by selecting an arbitrary vertex v from our given graph G.

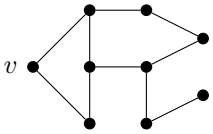
Using **depth-first search**, starting at vertex v, we build a path by visiting some vertex adjacent to v. We then continue this process of visiting new vertices and adding new edges to our path. If we reach some vertex and we cannot visit any other vertices (either because there are no more edges left to follow or because we have already visited all of the adjacent vertices), then we backtrack to the previous vertex and continue the process by branching off to a new path. We halt after we have visited every vertex of G.

We can use a stack data structure to keep track of the vertices we have visited as we traverse a path. To backtrack, we simply remove the top vertex from the stack.

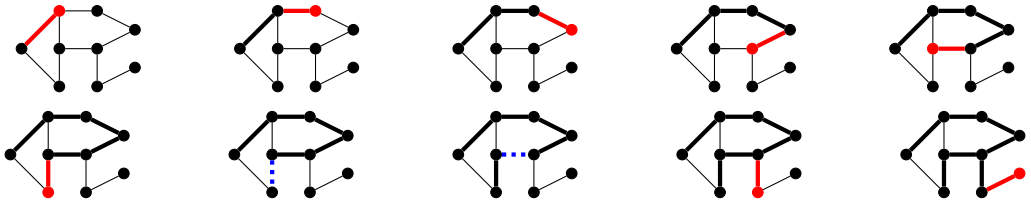
We can represent a depth-first spanning tree construction method in pseudocode as follows:

```
DFSspanningtree(G):
  T = a tree containing an arbitrary vertex V of G
  push V to stack S
  while S is not empty:
    for each vertex U adjacent to vertex at top of S:
      push U to stack S
      if U is already in T:
        pop U from stack S
      else:
        add U and incident edge to T
    if no more vertices are adjacent to vertex at top of S:
      pop vertex at top of S and backtrack
```

Example 24. Consider the following graph G:



Starting from vertex v in G, a depth-first search may construct a spanning tree in the following way:



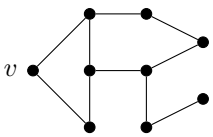
Using **breadth-first search**, starting at vertex v, we add all vertices adjacent to v and edges incident to v to our tree T. We do not add a vertex if it is already in T. Then, we visit each adjacent vertex in the order in which it was added and repeat the process. Again, we halt after we have visited every vertex of G.

We can use a queue data structure to keep track of the vertices we have yet to visit as we build our tree. The actions of enqueueing and dequeueing ensure that we visit vertices in the order in which they were added, as desired.

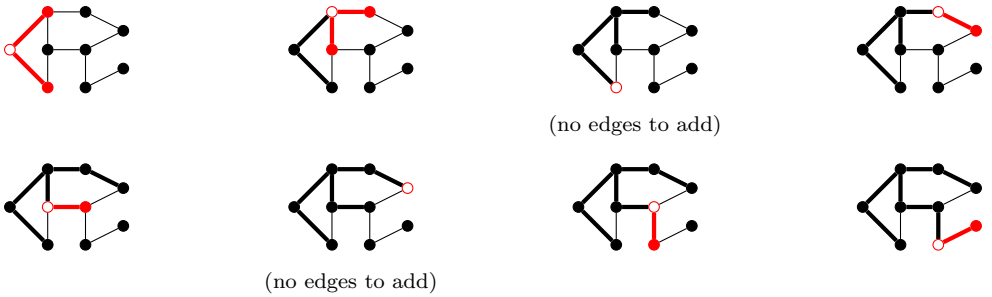
We can represent a breadth-first search spanning tree construction method in pseudocode as follows:

```
BFSspanningtree(G):
  T = a tree containing an arbitrary vertex V of G
  enqueue V into queue Q
  while queue Q is not empty:
    dequeue vertex from Q
    for each vertex U adjacent to current vertex:
      if U is not in T or Q:
        add U and incident edge to T
        enqueue U into queue Q
```

Example 25. Consider again the graph G from Example 24:



Starting from vertex v in G, a breadth-first search may construct a spanning tree in the following way:

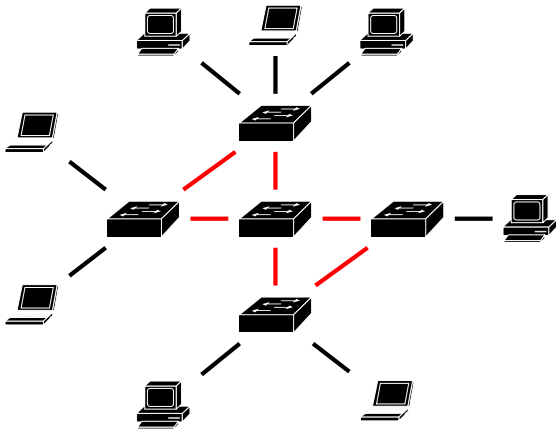


4.2 Applications of Spanning Trees

Spanning trees find uses in a number of applications, both within and outside of computing. They are especially handy for applications where circuits within a graph structure are undesirable, since a spanning tree gives a subset of edges of a graph that retains every vertex while avoiding circuits. Thus, spanning trees can be used to create search engine crawlers, plan plumbing systems, lay out power grids, and optimize communication networks. For now, let's focus on the last application.

In a computer network, redundancy is useful, but too much redundancy can harm performance. We say that a computer network contains a "switching loop" when there exists more than one path between two endpoints in the network. Switching loops can provide resilience if, say, one path in the network becomes disconnected. However, since switches blindly send frames across all of their network connections, switching loops can impact data transmission if a frame gets caught in an infinite loop between switches. In the worst case, a large number of infinitely-looping frames can create "broadcast radiation" and grind normal network traffic to a halt.

As an example, the following computer network contains switching loops:



The Spanning Tree Protocol, invented by the American programmer Radia Perlman, uses spanning trees to avoid switching loops in a network. The protocol constructs a spanning tree across the network in much the same way as we constructed spanning trees across graphs, where network connections not included in the spanning tree are disabled. In this way, switching loops are avoided and broadcast radiation is minimized. The protocol was later standardized by the IEEE and is still used to this day.

5 Minimum Spanning Trees

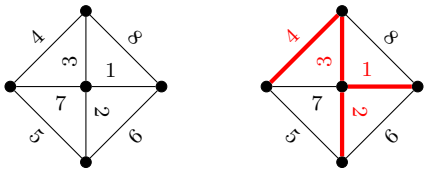
Previously in our study of graph theory, we briefly discussed graphs that labelled each of their edges with values. These so-called **weighted graphs** are graphs in which each edge is assigned a numerical weight. These weights could represent distances, costs, capacities, or any other measure that might be applied to a graph. These weights are usually positive integer values, but they could also be zero, negative, or non-integer values.

Remark. One thing we didn't mention previously is that the weights of edges do not necessarily need to abide by geometric rules like the triangle inequality. Therefore, it is best to think of weights abstractly, rather than as representing some concrete measure based on the drawing of the graph.

Just like in unweighted graphs, we can find spanning trees in weighted graphs. Moreover, we can use exactly the same techniques to find spanning trees in weighted graphs as we did for unweighted graphs. However, since edge weights give us valuable information about the graph, we might wish to refine our methods of finding spanning trees to find bounds on the weights of edges in a spanning tree. If we focus on adding only the lowest-weighted edges to a spanning tree, then we get what is known as a **minimum spanning tree**.

Definition 26 (Minimum spanning tree)**.** Given a connected, weighted graph G, a minimum spanning tree of G is a spanning tree where the sum of weights of the edges in the tree is as small as possible.

Example 27. The weighted graph at left has a minimum spanning tree at right.



It is possible to count the number of minimum spanning trees in a graph in a similar manner to how we counted general spanning trees. However, if we know one particular property of a graph in advance, then we get an interesting uniqueness result that does not apply to general spanning trees. This result tells us that if no pair of edges in the graph share the same weight, then the graph contains only one minimum spanning tree.

Theorem 28. *If a graph G is such that each edge in G has a distinct weight, then G contains exactly one minimum spanning tree.*

Proof. Assume that a graph G with distinct-weight edges contains two minimum spanning trees T_1 and T_2. Since these minimum spanning trees are different, there must exist at least one edge in one minimum spanning tree that does not exist in the other. Without loss of generality, let e_1 denote the edge of least weight that is in T_1 but not T_2.

Since T_2 is a tree, adding e_1 to T_2 will create a circuit. Since both T_1 and T_2 were constructed from the same graph G, there must exist an edge e_2 in the circuit of T_2 that is not in T_1. Moreover, e_2 must have a larger weight than e_1. However, replacing e_2 with e_1 in T_2 would create a minimum spanning tree of smaller total weight, which contradicts our assumption that T_2 was a minimum spanning tree.

Therefore, there cannot exist more than one minimum spanning tree in G. □

If we know other facts about certain aspects of a graph, then we can determine whether or not a given edge in the graph belongs to any minimum spanning tree of the graph. Our first lemma focuses on cut edges of a graph. (Recall that a cut edge is an edge that, when removed, increases the number of connected components of the graph.)

Lemma 29 (Cut property)**.** *Given a graph G, if the weight of some cut edge e in G is smaller than the weight of any other cut edge in G, then e belongs to every minimum spanning tree of G.*

Proof. Assume that there exists a minimum spanning tree T of G that does not contain a cut edge e of smallest weight. Adding e to T will create a circuit that traverses the cut of G once via e and again via some other cut edge e'. If we remove the cut edge e', then we would create a minimum spanning tree of smaller total weight, which contradicts our assumption that T was a minimum spanning tree.

Therefore, the cut edge e must belong to any minimum spanning tree T of G. □

Our second lemma focuses on edges within a circuit of a graph. If we consider the weights of each edge in some circuit of a graph, then we can find at least one edge that cannot belong to any minimum spanning tree of that graph.

Lemma 30 (Cycle property)**.** *For any circuit C in a graph G, if the weight of some edge e in C is larger than the weight of any other edge in C, then e does not belong to any minimum spanning tree of G.*

Proof. Assume that there exists a minimum spanning tree T of G that contains such an edge e. By removing e from T, we disconnect the minimum spanning tree into two connected components. By adding any other unused edge in the circuit C, we can reconnect both connected components into one tree. Since all other edges in C are of smaller weight than e, adding any such edge would create a minimum spanning tree of smaller total weight, which contradicts our assumption that T was a minimum spanning tree.

Therefore, such an edge e cannot belong to any minimum spanning tree T of G. □

5.1 Constructing Minimum Spanning Trees

Now, once again comes the big question: how do we construct a minimum spanning tree? Presumably, we don't want to use the same methods that we used for constructing general spanning trees, since those methods don't take edge weights into account. However, we can use the same basic ideas of selecting vertices and edges from a graph until we form a minimum spanning tree.

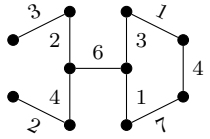
Our first method for constructing a minimum spanning tree, known as Kruskal's algorithm, works by sorting the edge set of a given graph G by increasing weight and then continually selecting the edge of least weight to add to the spanning tree. This method was invented by the American mathematician Joseph Kruskal in 1956.

Note that, although the final product is a connected minimum spanning tree, the intermediate trees need not be connected. Since Kruskal's algorithm selects edges only by weight, it may construct multiple connected components before linking them all together into one tree.

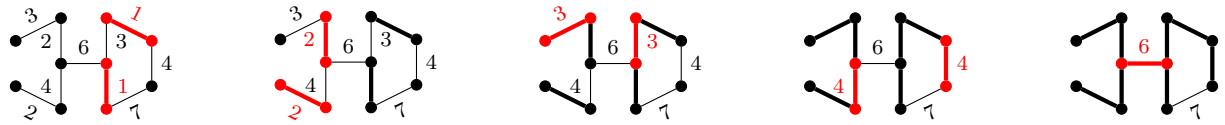
In pseudocode, Kruskal's algorithm is expressed as follows.

```
Kruskal(G):
  T = an empty tree
  sort all edges of G by increasing weight
  for each edge e = {u,v}:
    if e does not create a circuit in T:
      add edge e to T
```

Example 31. Consider the following graph G:



Using Kruskal's algorithm, we get the following minimum spanning tree:



Although Kruskal's algorithm is intuitive and easy to understand, it comes with some downsides; for instance, before we even begin to build a spanning tree, we must first sort our edge set by increasing weight. For graphs with many edges, this process could take some time.

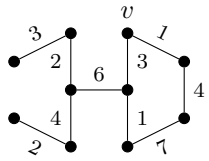
Our next method for constructing a minimum spanning tree takes a more vertex-centric approach, removing the requirement that we first sort the edge set. The Prim-Jarník algorithm works by selecting an arbitrary vertex v in a graph G, then constructing a spanning tree T by adding adjacent vertices by smallest edge weight. This algorithm uses the cut property of Lemma 29 to ensure that every edge added to T must belong to some minimum spanning tree of G.

The Prim-Jarník algorithm was originally invented by the Czech mathematician Vojtěch Jarník in 1930, and then rediscovered by the American mathematician Robert C. Prim in 1957—one year after Kruskal invented his method.

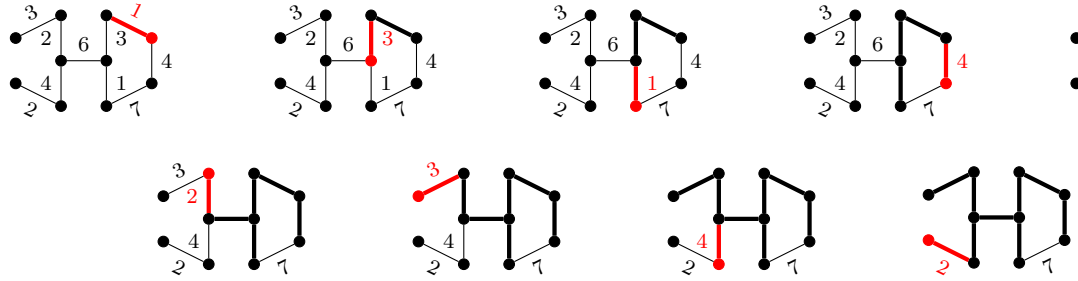
In pseudocode, the Prim-Jarník algorithm is expressed as follows.

```
PrimJarnik(G):
  T = a tree containing an arbitrary vertex of G
  while T contains fewer vertices than G:
    choose an edge e = {u,v} in G of smallest weight that is incident to a vertex v in T
    if e does not create a circuit in T:
      add edge e and vertex v to T
```

Example 32. Consider again the graph G from Example 31:



Using the Prim-Jarník algorithm starting from vertex v in G, we get the following minimum spanning tree:



Finally, as with so many other topics we have studied in this course, what we have seen here is not the complete picture. Other methods exist to construct minimum spanning trees. As an example, Borůvka's algorithm (invented by the Czech scientist Otakar Borůvka in 1926) was the first method to construct a minimum spanning tree. The algorithm works similarly to the Prim-Jarník algorithm, but in parallel over the entire graph. Thus, it is especially useful if you are constructing minimum spanning trees on a computer that is capable of parallelization.