Binary search trees: an implementation of dictionary ADT.
A binary tree in which each node stores one of the elements, satisfying the following ordering condition at *every* node. For a node storing element x, all elements stored in its left subtree are smaller than x, and all elements stored in its right subtree are greater than x.

Let h be the height of a given tree. The following implementation takes O(h) time for each operation.

Example:

```
   12
  /  \
 8    27
  \     \
   10    33
```

```
// Find x in tree. Returns node where search ends.
Entry<T> find( x ):
    // class object stack for stack of ancestors
    stack ← new Stack<Entry<T>>( )
    stack.push( null )
    return find( root, x )

Entry<T> find( t, x ): // LI: stack.peek() is parent of node t
    if t = null or t.element = x then return t
    while true do
        if x < t.element then
            if t.left = null then break
            else { stack.push(t);  t ← t.left }
        else if x = t.element then break
        else // x > t.element
            if t.right = null then break
            else { stack.push(t);  t ← t.right }
    return t

boolean contains( x ):
    t ← find( x )
    return t ≠ null and t.element = x

T min( ):
    if root = null then return null
    t ← root
    while t.left ≠ null do
        t ← t.left
    return t.element

T max( ):
    if root = null then return null
    t ← root
    while t.right ≠ null do
        t ← t.right
    return t.element
```

```
// Element is replaced if it already exists.
boolean add( x ):
    if root = null then
        root ← new Entry<>(x)
        size ← 1
        return true
    t ← find( x )
    if x = t.element then
        t.element ← x      // replace
        return false
    else if x < t.element then
        t.left ← new Entry<>(x)
    else
        t.right ← new Entry<>(x)
    size++;  return true

T remove( x ):
    if root = null then return null      空树
    t ← find( x )
    if t.element ≠ x then return null    没找着
    result ← t.element
    if t.left = null or t.right = null then
        bypass( t )
    else // t has 2 children
        stack.push( t )
        minRight ← find( t.right, t.element )
        t.element ← minRight.element
        bypass( minRight )
    size--;  return result

bypass( t ):  // called when t has at most one child
    pt ← stack.peek( )    t节点的parent
    c ← t.left == null ? t.right : t.left
    if pt = null then  // t is root
        root ← c
    else if pt.left = t then pt.left ← c      跳过t就相当于删掉t
    else pt.right ← c
```

BST

```
T get(x):
    t ← find(x)
    if t ≠ null and t.element = x then
        return t.element
    else return null
```

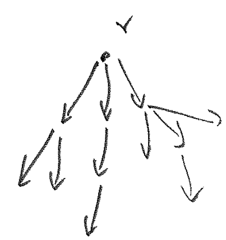
MST in directed graphs — Optimal Branching problem.

Input: ~~Directed~~ Graph $G=(V,E)$, root vertex $r \in V$

Edge weights $w: E \to \mathbb{Z}^+$ (nonnegative integer weights)

Output: A spanning tree (outgoing tree), rooted at r of minimum weight.

Directed ~~spanning~~ tree or Arborescence or Branching



Facts about rooted spanning trees:

In ~~outgoing~~ tree:

r has no incoming edges

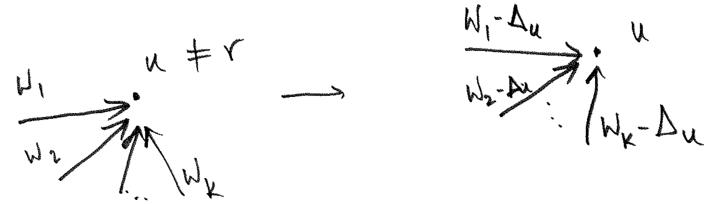
$u \in V - \{r\}$ has exactly one incoming edge.

r ~~has~~ can reach all nodes.

Greedy algorithms like Prim / Boruvka / Kruskal do not work.

not a tree.

Ideas:

1. 

If we decrease all edges into $u \neq r$ by Δ_u, then every spanning tree rooted at r decreases by Δ_u.

(because there is only one edge in tree into u)

$\Rightarrow$ MST's are invariant under this transformation.

Weight of MST (new weights) = Weight of MST (old weights) $- \Delta_u$

2. If we do this at all nodes (except r):

Weight of MST (new weights) = Weight of MST (old weights) $- \sum_{u \in V - \{r\}} \Delta_u$

3. If we choose

$$\Delta_u = \min_{x \in V} \{ w(x,u) \}$$

then weights of edges will stay nonnegative.

4. Let $G = (V, E)$ be a graph, root $r \in V$,
$w: E \to \mathbb{Z}^+$.
If G has a spanning tree T, rooted at r, whose total weight is 0, then T is an MST.

5. strategy:
(i) For each node u: subtract Δ_u from edges into u, where $\Delta_u = \min_{x \in V} \{ w(x,u) \}$
(ii) Check if there is a 0-weight spanning tree rooted at r.

If yes, output tree as MST.

? Otherwise !?
All nodes except r has one or more 0-edges coming into it.
If there is no 0-spanning tree then there is a 0-cycle not reachable from r.
(start walking back from a node u not reachable from r — infinite walk $\Rightarrow$ it has a cycle within).

6. Prelude: G, r, w, every node except r has 0-edge in, no s.t. rooted at r of 0-weight!

H = shrink a 0-cycle into a single node c.

G and H have same weight MST.

Proof:
($\Rightarrow$) Take an MST of G $\to$ shrink the nodes of cycle $\to$ discard parallel edges (all but one), edges within cycle $\to$ sp. Tree of H

$$MST(G) \geq MST(H).$$

($\Leftarrow$) Take $MST(H)$
Add all edges of cycle
Total weight = $MST(H)$

Graph: r can reach all nodes.

Tree $\subseteq$ contained in $MST(H) \cup$ cycle
$W(\text{Tree}) \leq MST(H)$.
$MST(G) \leq MST(H)$

Kruskal's algorithm: MST algorithm, using the disjoint-set data structure with Union/Find operations:

kruskal(g):	makeSet(u):
for u ∈ V do makeSet(u)	u.p ← u; u.rank ← 0
Create an empty list of edges, mst	find(u):
Sort edges by weight	if u ≠ u.p then u.p ← find(u.p)
for each edge e=(u,v) in sorted order do	return u.p
ru ← find(u)	union(x, y):
rv ← find(v)	if x.rank > y.rank then y.p ← x
if ru ≠ rv then	else if y.rank > x.rank then x.p ← y
mst.add(e)	else
union(ru, rv)	x.rank++;
return mst	y.p ← x

Boruvka's algorithm: MST algorithm suitable for parallel or distributed computing:

```
boruvka( g ):
   F ← Spanning forest of g, with no edges.  Each vertex is in a separate component.
   while F has more than one connected component do
       Let the connected components of F be C.
       For c ∈ C, find emin(c), a minimum weight edge of G connecting c to another component c' ∈ C
       Proposal step: Each component c proposes to add emin(c) to F
       Merge step: Add as many proposed edges to F as possible, without creating cycles
   return F
```

MST in directed graphs (Optimal branching algorithms of Chu and Liu | Edmonds):
Greedy algorithm does not work in directed graphs. Two kinds of rooted trees: incoming, outgoing.
Outgoing tree: acyclic subgraph in which (a) root node has no incoming edges, (b) there is a path from the root to every vertex, (c) all non-root nodes have exactly one incoming edge.

Input: Directed graph G=(V,E), edges have weights, root node r ∈ V.
Output: Outgoing spanning tree, rooted at r, of minimum weight.

Theorem 1. Consider u ∈ V, u ≠ r. Suppose we decrease the weight of every edge into u by Δ. Then the weight of the MST decreases by Δ.
Proof: Consider any spanning tree T, rooted at r. T has exactly one edge into u. Therefore the above transformation decreases weight of T by Δ. Since the weight of all spanning trees rooted at r decrease by the same amount, MST of G is unchanged.

Remark: Suppose the weights of every edge into u is decreased by Δ_u, for all u ∈ V − { r }. Then the net reduction in the weight of each tree rooted at r is equal to the sum of Δ_u, u ∈ V − { r }.

Theorem 2. Let G=(V,E) be a graph with nonnegative edge weights. If G has a spanning tree T rooted at r, and w(T) = 0, then T is an MST of G, rooted at r.
Proof: Since all edges have nonnegative weights, the weight of any tree is nonnegative. Therefore a tree of weight 0 has minimum weight.

Chu and Liu | Edmonds Algorithm for finding optimal branchings (MST in directed graphs):
Input: Directed graph G=(V,E), nonnegative weight function w on its edges, root r $\in$ V.
Output: Directed tree rooted at r (outgoing tree), of minimum weight. Assume that G has no edges into r.

1. Transform weights so that every node except r has an incoming edge of weight 0:

   for u $\in$ V$-$\{ r \} do
       Let Δ_u be the weight of a minimum weight edge into u
       for all edges e = (p, u) into u do
           e.weight ← e.weight - Δ_u // called "reduced weight" of e

2. Let G_0 = (V, Z) be the subgraph of G containing all edges of 0-weight: Z = {e $\in$ E: e.weight = 0}. Run DFS/BFS in G_0, from r. Note that we are using only edges of G with 0-weight. If all nodes of V are reached from r, then return this DFS/BFS tree as MST.

3. If there is no spanning tree rooted at r in G_0, then there is a 0-weight cycle. Find a 0-weight cycle as follows:
   a. Find a node z that is not reachable from r in G_0, in the above search.
   b. Walk backward from z in G_0, using incoming edges of 0-weight at each node visited. Every node except r has a 0-edge coming into it, and so this walk can keep going forever. Since r has no path to z using 0-edges, this walk will never get to r. There are only a finite number of nodes. So, some node x will be repeated on this walk. The path from x to itself on this walk is composed of 0-weight edges, and this gives a 0-weight cycle C.

4. Shrink cycle C into a single node c. There may be many edges from the nodes of C to a node u outside the cycle. These are replaced by a single edge. For each edge e=(a,u) in G, with a $\in$ C and u $\notin$ C, introduce the edge (c,u) of weight w(a,u).

   Similarly, for edges of G that are going into C, do the following. For each edge (u,a) in G, with u $\notin$ C and a $\in$ C, introduce the edge (u,c) of weight w(u,a).

   For each vertex u $\notin$ C, if the above process creates multiple edges (c,u), keep just one edge with minimum weight, and record the corresponding edge of G. Similarly, process multiple edges (u,c) by replacing each multi-edge by a single edge of minimum weight.

   The new graph has fewer nodes than the original graph, and the MSTs of the two graphs have equal weight.

5. Recursively find an MST of the smaller graph. This MST has exactly one edge into c, and this edge corresponds to some actual edge (u,a) in the graph before shrinking, where a $\in$ C. Now, expand node c, and include the edges of the 0-weight cycle C. Since the total weight of the cycle is 0, adding it to the MST does not increase its weight. But node a will have 2 incoming edges: edge (u,a) from the MST, and one edge from the cycle. Delete the edge coming into node a in the cycle, to get an MST of the original graph. Return this MST.

Tarjan's improved algorithm for optimal branchings:

Modify the shrinking step as follows:
In the zero-graph G_0 = (V, Z), find its strongly connected components. If it has only one scc, then DFS or BFS can find a 0-weight spanning tree, rooted at r. This is an MST.

Shrinking step: Otherwise, let G_0 have k strongly connected components. Let r be in scc number 1. Since r has no incoming edges in G, that scc will not have other nodes in it. Shrink each scc into a single node. The new graph has k nodes, $C_1 \cdots C_k$. The weight of edge (C_i, C_j) is equal to the minimum weight of an edge connecting some $u \in C_i$ to some $v \in C_j$:

$$(C_i,\ C_j).weight\ =\ \min_{u \in C_i,\, v \in C_j} (u,\ v).weight$$

In the new graph H, C_1 (the node containing r) is the root node. For each edge of H, record its image, which is the edge of G to which it corresponds (i.e., a minimum-weight edge that is argmin in the above equation).

Theorem: Weight of MST of G rooted at r = Weight of MST of H rooted at C_1.

Expansion step: After finding an MST of H, rooted at C_1, we can expand each scc and find an MST of G, rooted at r as follows. C_1 contains only the root vertex, r. MST(H) rooted at C_1 has exactly one edge into each C_i, $i = 2 \cdots k$. Let the edge into C_i correspond to edge (u,v) of G, where $v \in C_i$. Find a spanning tree within C_i, rooted at v, using only 0-weight edges. The MST of G is the union of the k-1 spanning trees within C_i, $i = 2 \cdots k$, and the edges of G that are images of the edges of MST(H).

Implementation notes

To be able to solve large instances, it is not feasible to create a new graph in each phase of the algorithm. In many iterations, most strongly connected components may have just one node each, and it is possible that only one scc actually shrinks. Therefore, it is necessary to be able to add new vertices and edges as the algorithm progresses. Existing edges and vertices that are entirely contained within the same component have to be disabled. We can extend the graph, vertex and edge classes to facilitate these changes. Design the classes and their iterators carefully. If designed properly, it should be possible to call standard implementations of SCC, BFS, or DFS on this extended graph, and when it iterates over the outgoing edges of a node, the algorithm uses only edges of zero weight. When iterating over the vertices of a graph, it should automatically skip the disabled vertices.