Minimum Spanning Trees using indexed priority queues

- Modify BinaryHeap<T>

  Normally "T extends Comparable<? super T>"

  T also extends Index (interface).

<u>Index</u>: public int getIndex()
public void putIndex(int index).

index = index of element in the binary heap.

<u>BinaryHeap</u>:

move (i, x) : pq[i] = x ;

<u>Indexed heap</u>: override move.

```
move (i, x) :
        super.move(i, x);
        x.putIndex(i);
```

<u>Decrease Key operation</u>:

When u.d becomes smaller because of an update,
then we call q.decreaseKey(u) (q = indexed heap)

```
decreaseKey(Vertex u):
        ~~q.~~ percolateUp(u.getIndex());
```

```
Prim3( G=(V,E), src ):  // Implementation #3 using indexed priority queue of vertices
   // Node v ∈ V−S stores in v.d, the weight of a smallest edge that connects v to some u∈S
   for u ∈ V do  { u.seen ← false;   u.parent ← null;   u.d ← ∞ }
   src.d ← 0
   wmst ← 0
   q ← new indexed priority queue of vertices of G, with u.d as priority of u  // actually, PrimVertex
   for u in g do q.add( u )  // q.add( get(u) )
   while q is not empty do
         u ← q.remove( )
         u.seen ← true
         wmst ← wmst + u.d
         for all edges e incident on u do
            v ← e.otherEnd( u )
            if not v.seen and e.weight < v.d then
                  v.d ← e.weight
                  v.parent ← u
                  q.decreaseKey( v )  // Need to call percolateUp(index of v in q).  How do we find it?
   return wmst
```

```
class PrimVertex implements Comparable<PrimVertex>, Factory, Index {
   int index;  // To store index of this node in the priority queue
...
   public void putIndex( int index ) { this.index = index; }  // called by move() in IndexedHeap
   public int getIndex( ) { return index; }   // called by Prim3 to get index in pq for calling percolateUp
}
```

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

```
kruskal( g ):
   for u∈V do makeSet( u )
   // Above step is automatic with GraphAlgorithm
   mst ←  new list of edges
   edgeArray ← g.getEdgeArray()
   Arrays.sort( edgeArray ) // sort edges by weight
   for each edge e=(u,v) in edgeArray do
         ru ← u.find( )
         rv ← v.find( )
         if ru ≠ rv then
            mst.add( e )
            ru.union( rv )
   return mst
```

```
// Following methods are in KruskalVertex class:
make(Vertex u):  // makeSet( )
   parent ← this;   rank ← 0

find( ):
   if this ≠ parent then
        parent ← parent.find( )
   return parent

union( rv ): // Pre:  this.parent = this,  rv.parent = rv
   if this.rank > rv.rank then
         rv.parent ← this
   else if  this.rank < rv.rank then
        this.parent ← rv
   else
        this.rank++;
        rv.parent ← this
```

Summary of MST algorithms:

1. Prim1 : PriorityQueue <Edge>
   RT is dominated by Priority Queue operations.
   $= O(|E| \log |V|)$.
   Prim2

2. Prim3 : IndexedQueue <~~Vertex~~ PrimVertex>.

   $RT = O(|E| \log |V|)$.

   Prim3 will outperform Prim1 on dense graphs.
   Advanced data structure called Fibonacci heaps that can be used to get $RT: O(|E| + |V| \log |V|)$.

3. Kruskal's algorithm: using Union/Find Disjoint set data structure:

   $RT = O(|E| \log |E| + |E| \alpha(|V|))$.

   RT is dominated by the time to sort edges by weight.

---

shortest path problems

Input: Directed graph $G=(V,E)$, edge weights $w: E \to \mathbb{Z}$ (or $\mathbb{R}$)

weight of a path P from u to v:

$$u \xrightarrow{5} \xrightarrow{3} \xrightarrow{2} \xrightarrow{-1} \xrightarrow{8} \xrightarrow{4} \xrightarrow{-5} v \quad : 16$$

$$w(P) = \sum_{e \in P} w(e)$$

Define $\delta(u,v)$ = Weight of a shortest path (minimum weight) from u to v (path = simple path)

$$= \min \left\{ w(P_{uv}) : P_{uv} \text{ is a simple path from } u \text{ to } v \right\}.$$

Shortest path problem:

1. Single source problem:
   Given a source vertex s,
   find $\delta(s,u)$ for all $u \in V$.

2. All-pairs version:
   Find $\delta(u,v)$ for all $u,v \in V$.

Basis of shortest path algorithms:

Let P be a shortest path from s to v.
Let u be predecessor of v on this path.
Then subpath from s to u is a shortest path from s to u.

$$\delta(s,v) = \delta(s,u) + w(u,v).$$

"Subpath of a shortest path is a shortest path".
This can be false when the graph has a cycle C such that $\sum_{e \in C} w(e) < 0$ (negative cycle).

<u>Shortest paths</u>:

Input: Graph G = (V, E) (usually, directed), source vertex $s \in V$, edge weights $w : E \rightarrow \mathbb{Z}$ (more generally, $\mathbb{R}$).

Weight (or length) of a path P, $w(P) = \sum_{e \in P} w(e)$. A cycle C is called a negative cycle if $w(C) < 0$.

Output: For each $u \in V$, find a <u>simple</u> path from s to u, of minimum weight.

<u>Overview of shortest path algorithms</u>

Algorithm	**Condition**	**Class of graph**	**Running time**
Breadth-First Search (BFS)	No weights on edges	Directed or undirected	$E + V$
DAG-shortest-path	No cycles	DAG	$E + V$
Dijkstra's algorithm	No negative edges	Directed or undirected	$E \log V$
Bellman-Ford algorithm	No cycles of negative length	Directed	$E\ V$

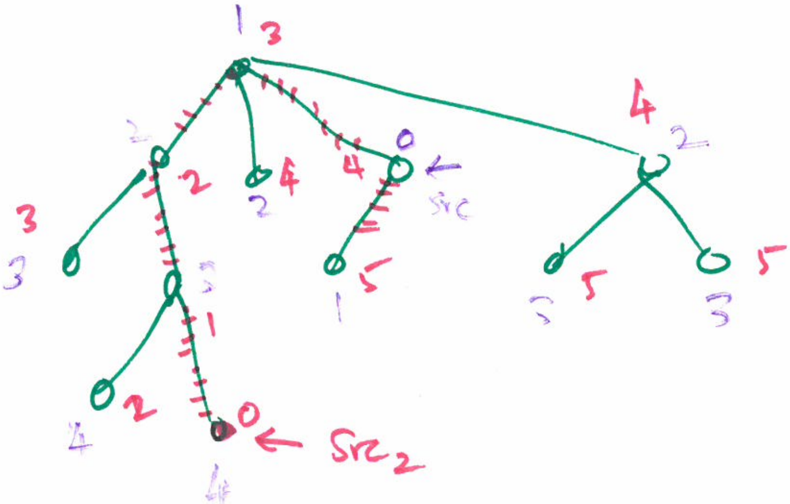
<u>Breadth-First Search (BFS)</u>: Find shortest number of hops from a source s to all nodes of G.

```
bfs(g, s):
    for u ∈ g do
        u.d ← ∞
        u.π ← null
        u.seen ← false
    Create a queue q of vertices
    s.d ← 0
    s.seen ← true
    q.add( s )
    while q is not empty do
        u ← q.remove( )
        for edge e=(u, v) incident on u  do
            if not v.seen then
                v.d ← u.d + 1
                v.π ← u
                v.seen ← true
                q.add( v )
```

<u>Applications of BFS</u>:

(1) Broadcast trees,
(2) Test if an undirected graph is bipartite,
(3) Find diameter of an unrooted tree,
(4) Find shortest paths in graphs whose edges have small integer weights,
(5) Find an odd-length cycle in a non-bipartite undirected graph,
(6) Find a shortest odd-length cycle of an undirected graph,
(7) Used as a subroutine in maximum flow algorithms of Edmonds and Karp, and, Dinitz.

Diameter of a tree using BFS - edges have no weights.



Diameter of a tree

$$= \max_{u,v \in V} \{ \delta(u,v) \}$$

1. Run BFS using any vertex as src.
2. Run BFS again, with a vertex at max distance from first source, as source.

Diameter of tree = max distance of any node from the second source.

Finding odd-length cycles:

1. If G is bipartite $\longrightarrow$ no odd cycles.

Run BFS on G.
If G has any edge (u,v) with $u.d = v.d$ then G is not bipartite.

cycle of length $2k+1$

k = length of path from u, v to their least common ancestor (LCA) in BFS tree.