Important topics to study for Quiz 2

SkipList: Entry class, constructor, find, add, contains, floor, ceiling, iterator.

Applications of BST and hashing from class, assignments (and similar problems).

BST: TreeSet operations: add, contains, remove, floor, ceiling, size, iterator.
TreeMap operations: put, containsKey, get, remove, entrySet.

Hash tables: HashSet: add, contains, remove, iterator. HashMap: put, containsKey, get, remove, entrySet.

Priority queues: Implementation: as arrays, order property, structure property, add, percolateUp.
Applications: kth largest of array/stream, heap sort, Huffman coding, Worst fit (Bin packing).

DFS-based algorithm for topological order

Fields of class: topNum, time, finishList, acyclic. Field g is inherited from GraphAlgorithm.

Attributes of Vertex (stored in PERTVertex, say): color, dis, fin, top, parent. Colors: {white, gray, black}.

```
toplogicalOrder():
   if g is not directed then
      throw exception “Graph is not directed”
   acyclic ← true
   topNum ← g.size()
   dfs( )
   if acyclic then return finishList
   else throw exception “Graph is not acyclic”

dfs():
   time ← 0
   finishList ← new Linked List of vertices
   for u in g do
      u.color ← white;     u.parent ← null
   for u in g do
      if u.color = white then dfsVisit(u)
```

```
dfsVisit(u):
   u.color ← gray
   u.dis ← ++time
   for each edge (u,v) going out of u do
      if v.color = white then
         v.parent ← u
         dfsVisit(v)
      else if v.color = gray then // back edge
         acyclic ← false
   u.fin ← ++time
   u.color ← black
   u.top ← topNum
   topNum ← topNum − 1
   finishList.addFirst(u)
```

Another algorithm for topological ordering of the vertices of a DAG:

```
topologicalOrder():  // g is inherited from GraphAlgorithm.
topNum ← 0
q ← new Queue of vertices
topList ← new List of vertices
for u in g do
   u.degree ← u.inDegree()
   if u.degree = 0 then q.add(u)
while q is not empty do
   u ← q.remove()
   u.top ← ++topNum
   topList.add(u)
   for each edge (u,v) going out of u do
         v.degree ← v.degree − 1
         if v.degree = 0 then q.add(v)
if topNum = |V| then return topList
else return null  // or throw exception “Graph is not acyclic”
```

Minimum Spanning trees :

Input: Undirected, connected graph $G = (V, E)$
with edge weights $w : E \to \cancel{\mathbb{R}}\ \mathbb{Z}$
(edge e has weight $w(e)$.
In Graph class, $w(e)$ is stored in e.weight).

Problem: Given a spanning tree T, weight of T,

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

Find a spanning tree of G, whose weight is minimum.

Running time of Prim's algorithm:
An edge $e = (u, v)$ is added to q only when one end, say u, is added to the tree, and its edges are checked, and v.seen = false.
Since each node is added to the tree only once, each edge is added to q at most once. In the worst case, every edge is added to q and later deleted. RT of operations on $q = O(|E| \log(|E|))$.
Other operations take $O(|V| + |E|)$ time. So total running time is

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

For simple graphs, $|E| \le \binom{|V|}{2} = \frac{|V|(|V|-1)}{2} < |V|^2$

Therefore $\log(|E|) < \log(|V|^2) = 2 \log(|V|)$.

So, RT of Prim's algorithm $= O(|E| \log |V|)$.

Minimum spanning trees (MST)

Input: Undirected, connected graph $G = (V, E)$, weights on edges $w: E \rightarrow \mathbb{Z}$ (can be $\mathbb{R}$, the set of reals).

Output: Spanning tree $T \subseteq E$, such that $w(T) = \sum_{e \in T} w(e)$ is a minimum among all spanning trees of G.

Prim's algorithm for finding MST:

```
   Grow a tree starting at some node src as source.
   S = Set of nodes connected by the tree.  Initially, S = {src}.
   while S ≠ V do
         Find a edge, e = (u, v) of minimum weight, connecting some u ∈ S with some v ∈ V − S.
         Extend tree by adding edge e to tree.  S ← S ⋃ {v}.
```

```
Prim1( G=(V,E), src ):  // Implementation #1 using a priority queue of edges
   for u ∈ V do  { u.seen ← false;   u.parent ← null }
   src.seen ← true
   wmst ← 0;    mst ← new list of edges
   Create a priority queue q of edges
   for each edge e incident to src do q.add(e)

   while q is not empty do
      e ← q.remove( ).  Let e = (u, v), with u.seen = true.
      if v.seen then continue  // skip this edge
      v.seen ← true
      v.parent ← u
      wmst ← wmst + e.weight;     mst.add(e)
      for each edge e2 incident to v do
         if not e2.otherEnd( v ).seen then q.add( e2 )
   return wmst  // MST is implicitly stored by parent pointers
```

Kruskal's algorithm: 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
```