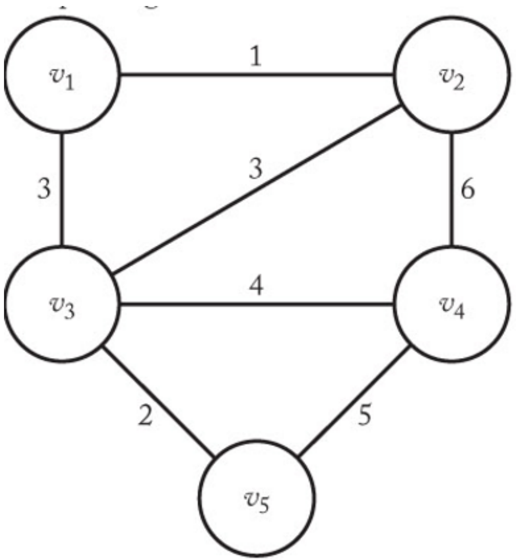
Lecture 10: Chapter 4 Part 2

The Greedy Approach

CS3310

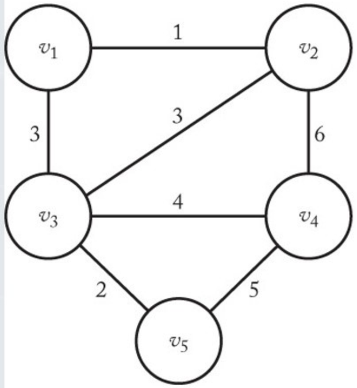
Graph Theory

What is a minimum spanning tree of this graph?

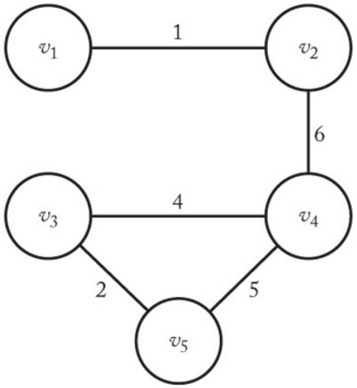


Minimum Spanning Trees

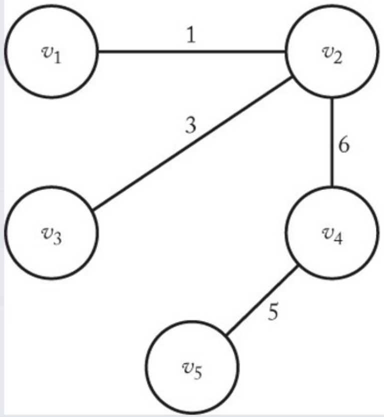
(a) A connected, weighted, undirected graph G.



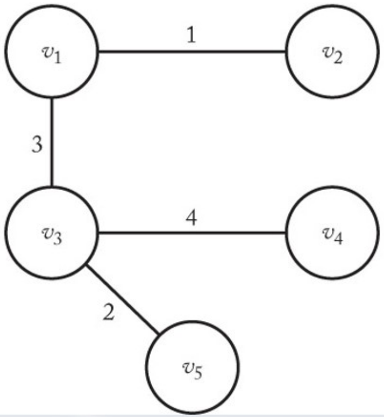
(b) If (v_4, v_5) were removed from this subgraph, the graph would remain connected.



(c) A spanning tree for G.



(d) A minimum spanning tree for G.



Kruskal's Algorithm

Like Prim's Algorithm, Kruskal's Algorithm calculates a minimum spanning tree given a weighted, connected graph G = (V, E). The edges of the spanning tree are placed in F.

- Kruskal's algorithm starts by placing each vertex in V in its own disjoint set.
- For example: V = {1, 2, 3, 4, 5} → {1}, {2}, {3}, {4}, {5}
- Every edge in the graph is then sorted in *nondecreasing* order of weight.
- We select the smallest remaining edge *e* and make sure the vertices it connects are not in the same set. If they are not, we add *e* to F and merge their two subsets.

Kruskal's Algorithm High-Level

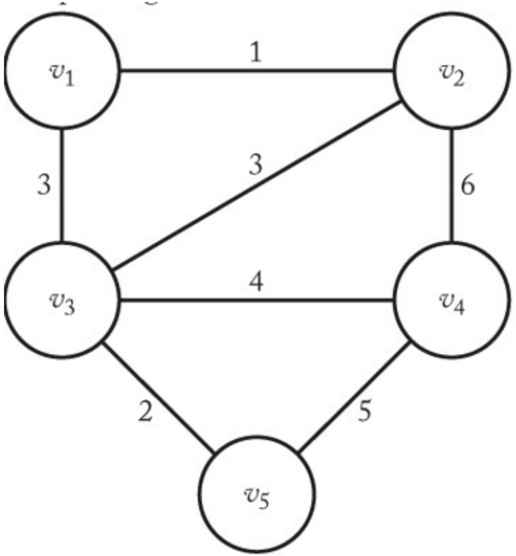
```
F = {}                    // initialize set of edges in spanning tree to empty

create disjoint subsets of V, one for each vertex;

sort the edges in E in nondecreasing order;

while (the instance is not solved)
          select next edge;                                                  //
selection procedure
          if ( it connects 2 vertices in disjoint subsets)  // feasibility check
                    merge the subsets;
                    add the edge to F;
          if (all the subsets are merged)                               // solution
check
                    the instance is solved;
```

Kruskal's Algorithm

V: $\{v_1, v_2, v_3, v_4, v_5\}$

E: $(v_1, v_2), (v_1, v_3), (v_2, v_3), (v_2, v_4), (v_3, v_4), (v_3, v_5), (v_4, v_5)$

How do we initialize this problem?



Kruskal's Algorithm

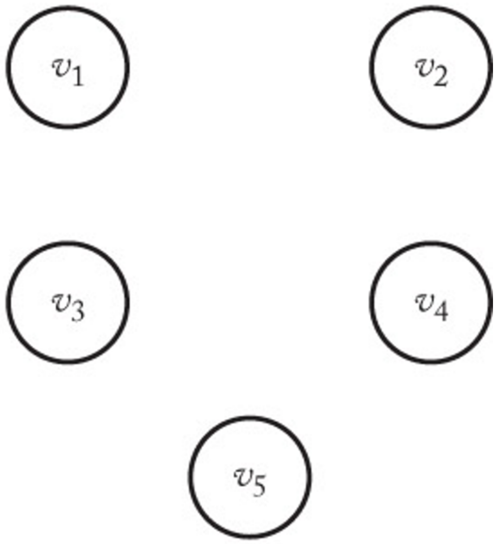
Disjoint Sets: $\{v_1\}, \{v_2\}, \{v_3\}, \{v_4\}, \{v_5\}$

F: $\{\}$

Edges (sorted): $(v_1, v_2), (v_3, v_5), (v_1, v_3), (v_2, v_3), (v_3, v_4), (v_4, v_5), (v_2, v_4)$

1 2 3 3 4 5 6

- Each edge is sorted by weight and each vertex is placed in its own disjoint set.
- Initialize F to the empty set.



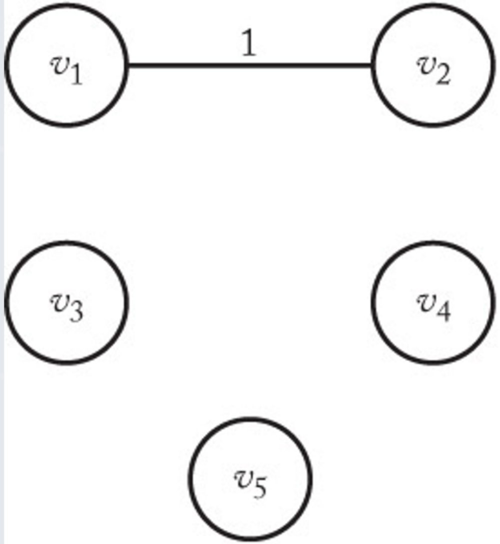
Kruskal's Algorithm

Disjoint Sets: $\{v_1, v_2\}$, $\{v_3\}$, $\{v_4\}$, $\{v_5\}$

F: $\{(v_1, v_2)\}$

Edges (sorted):	$\boldsymbol{(v_1, v_2)}$,	(v_3, v_5),	(v_1, v_3),	(v_2, v_3),	(v_3, v_4),	(v_4, v_5),	(v_2, v_4)
	1	2	3	3	4	5	6

- The smallest edge, (v_1, v_2), is chosen first.
- v_1 and v_2 are in disjoint sets, so we add (v_1, v_2) to F and merge the sets v_1 and v_2 are in.



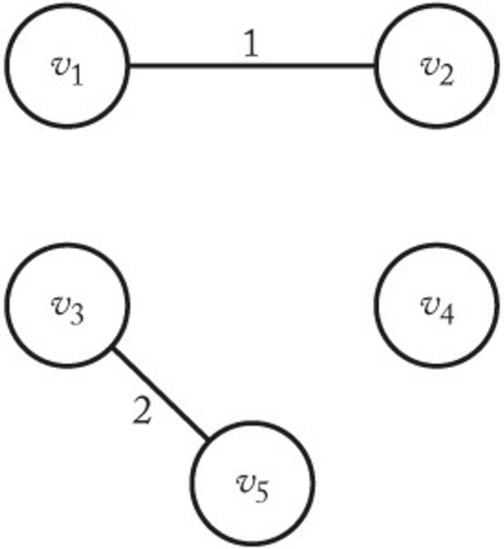
Kruskal's Algorithm

Disjoint Sets: $\{v_1, v_2\}$, $\{v_3, v_5\}$, $\{v_4\}$

F: $\{(v_1, v_2), (v_3, v_5)\}$

Edges (sorted): (v_1, v_2), $\mathbf{(v_3, v_5)}$, (v_1, v_3), (v_2, v_3), (v_3, v_4), (v_4, v_5), (v_2, v_4)

1 2 3 3 4 5 6



- The next smallest edge, (v_3, v_5), is chosen.
- v_3 and v_5 are in disjoint sets, so we add (v_3, v_5) to F and merge the sets v_3 and v_5 are in.

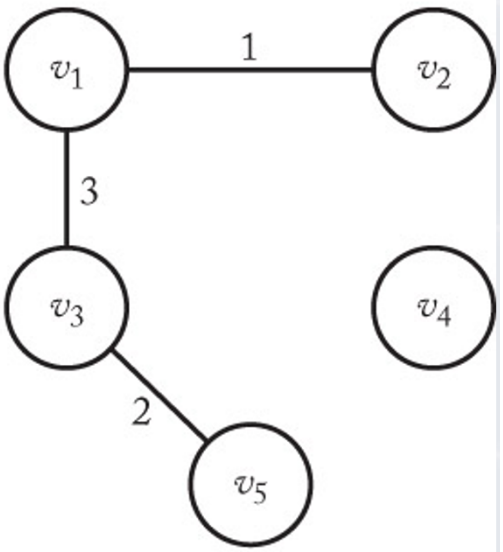
Kruskal's Algorithm

Disjoint Sets: $\{v_1, v_2, v_{3,} v_5\}$, $\{v_4\}$

F: $\{(v_1, v_2), (v_3, v_5), (v_1, v_3)\}$

Edges (sorted):	(v_1, v_2),	(v_3, v_5),	$\mathbf{(v_1, v_3)}$,	(v_2, v_3),	(v_3, v_4),	(v_4, v_5),	(v_2, v_4)
	1	2	3	3	4	5	6

- The next smallest edge, (v_1, v_3), is chosen.
- v_1 and v_3 are in disjoint sets, so we add (v_1, v_3) to F and merge the sets v_1 and v_3 are in.



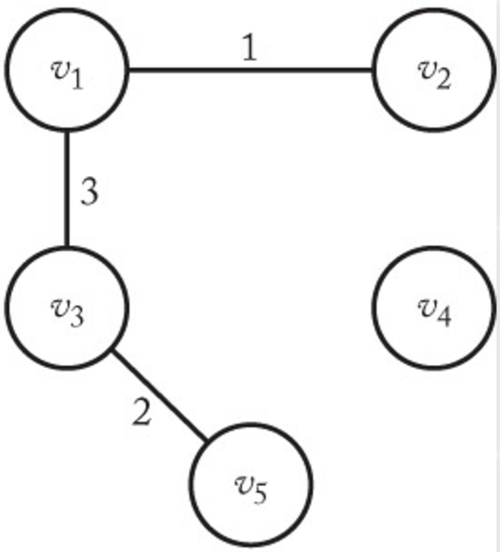
Kruskal's Algorithm

Disjoint Sets: $\{v_1, v_2, v_{3,} v_5\}$, $\{v_4\}$

F: $\{(v_1, v_2), (v_3, v_5), (v_1, v_3)\}$

Edges (sorted): (v_1, v_2), (v_3, v_5), (v_1, v_3), $\mathbf{(v_2, v_3)}$, (v_3, v_4), (v_4, v_5), (v_2, v_4)

1 2 3 3 4 5 6



- The next smallest edge, (v_2, v_3), is chosen.
- v_2 and v_3 are <u>not</u> in disjoint sets, so we reject (v_2, v_3)

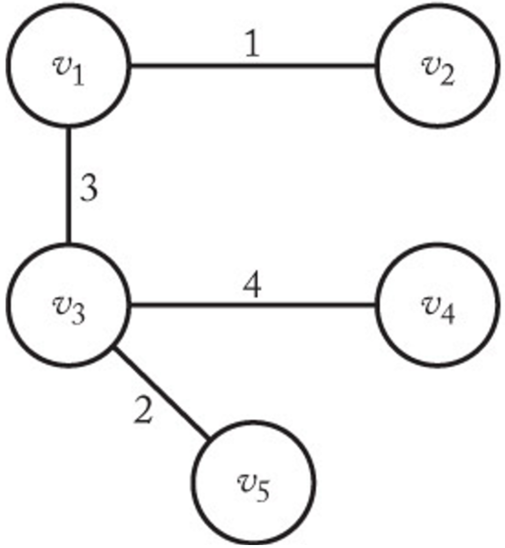
Kruskal's Algorithm

Disjoint Sets: $\{v_1, v_2, v_{3,} v_{4,} v_5\}$

F: $\{(v_1, v_2), (v_3, v_5), (v_1, v_3), (v_3, v_4)\}$

Edges (sorted): $(v_1, v_2), (v_3, v_5), (v_1, v_3), (v_2, v_3), \boldsymbol{(v_3, v_4)}, (v_4, v_5), (v_2, v_4)$

1 2 3 3 4 5 6



- The next smallest edge, (v_3, v_4), is chosen.
- v_3 and v_4 are in disjoint sets, so we add (v_3, v_4) and merge the two sets

All disjoint sets are now merged, so we stop here!

Kruskal’s Algorithm

- As with Prim’s, Kruskal’s Algorithm is easier to discuss at a high level than to implement in code.
- To make some algorithms efficient, we need a cleverly designed data structure.
- Kruskal’s algorithm makes use of a **disjoint set** data structure.

Disjoint Sets for Kruskal's Algorithm

Let's say we start with a universe U of elements:

U = {A, B, C, D, E}

We can write a function `makeset` that creates a disjoint set for its argument:

```
for (each x ∈ U)
        makeset(x)                    // make a unique set for each element of U
```

We define a data type `set_pointer` and a function `find`. If `p` and `q` are of type `set_pointer`:

```
p = find('B');
q = find('C');
```

- `p` now points to the set B is in and `q` points to the set C is in.

Disjoint Sets for Kruskal's Algorithm

We will also define a function `merge`.

Calling `merge(p, q)` performs step b, merging two sets into one:

(a) There are five disjoint sets. We have executed $p = \mathit{find}(\mathrm{B})$ and $q = \mathit{find}(\mathrm{C})$.

$$\begin{array}{ccccc} \{\mathrm{A}\} & \{\mathrm{B}\} & \{\mathrm{C}\} & \{\mathrm{D}\} & \{\mathrm{E}\} \\ & \uparrow & \uparrow & & \\ & p & q & & \end{array}$$

(b) There are four disjoint sets after $\{\mathrm{B}\}$ and $\{\mathrm{C}\}$ are merged.

$$\begin{array}{cccc} \{\mathrm{A}\} & \{\mathrm{B},\mathrm{C}\} & \{\mathrm{D}\} & \{\mathrm{E}\} \end{array}$$

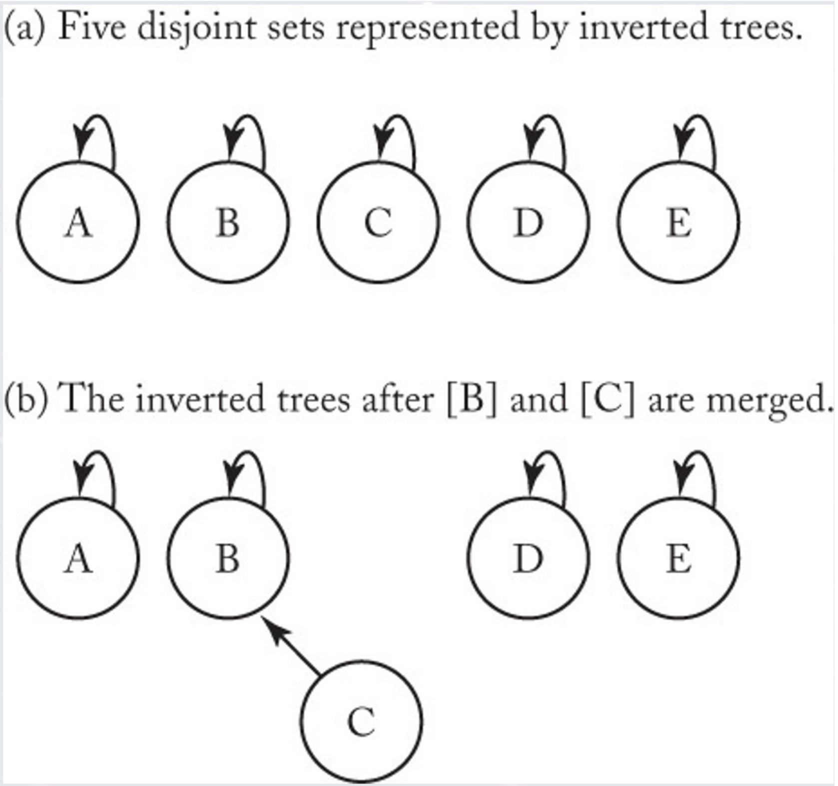
(c) We have executed $p = \mathit{find}(\mathrm{B})$.

$$\begin{array}{cccc} \{\mathrm{A}\} & \{\mathrm{B},\mathrm{C}\} & \{\mathrm{D}\} & \{\mathrm{E}\} \\ & \uparrow & & \\ & p & & \end{array}$$

Disjoint Set Data Structure

- We will represent disjoint sets by using **inverted trees**.
- With an *inverted tree*, each nonroot points to its parent, and each root points to itself.
- Each disjoint set is represented by one tree:

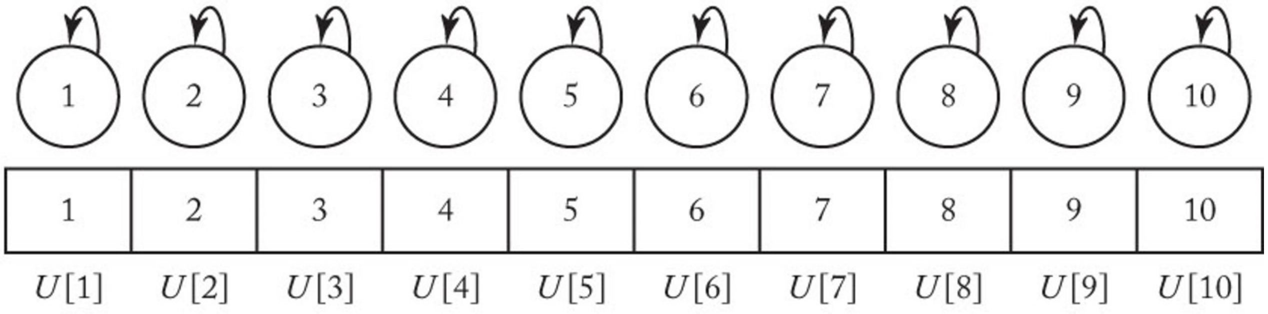
(a) represents {A}, {B}, {C}, {D}, {E}
(b) represents {A}, {B, C}, {D}, {E}



Disjoint Set Data Structure

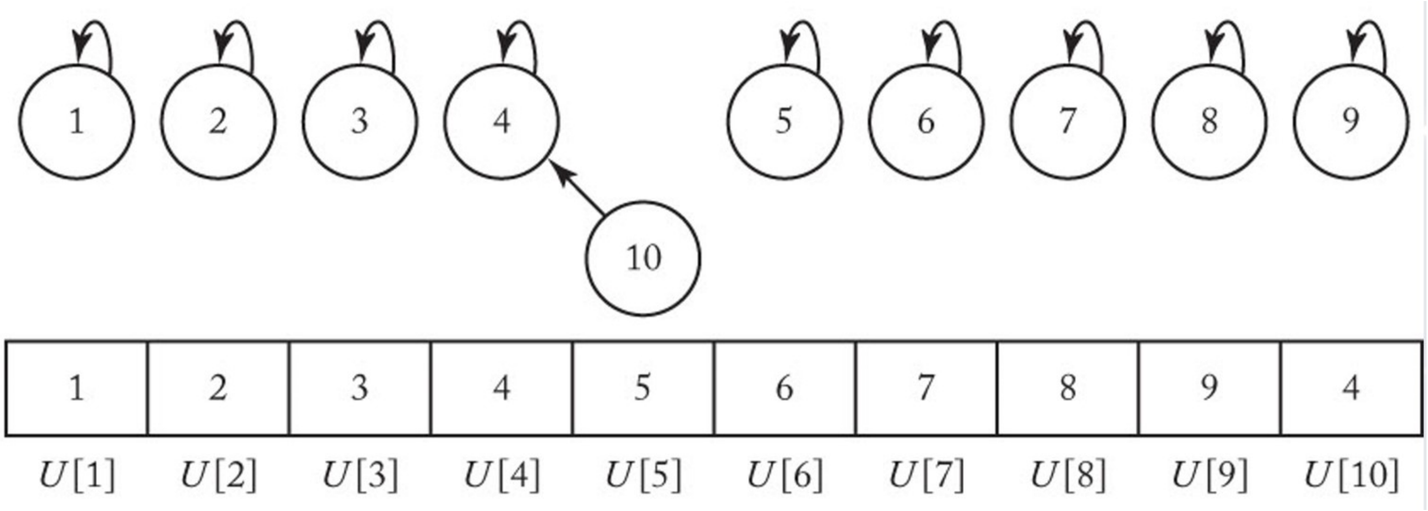
One way to represent inverted trees is with an array.

- Below is array U with 10 disjoint sets.
- U[*i*] = the parent of *i*
  - i.e. U[4] = 4 since 4 points to itself; it is the root of its tree.



Disjoint Set Data Structure

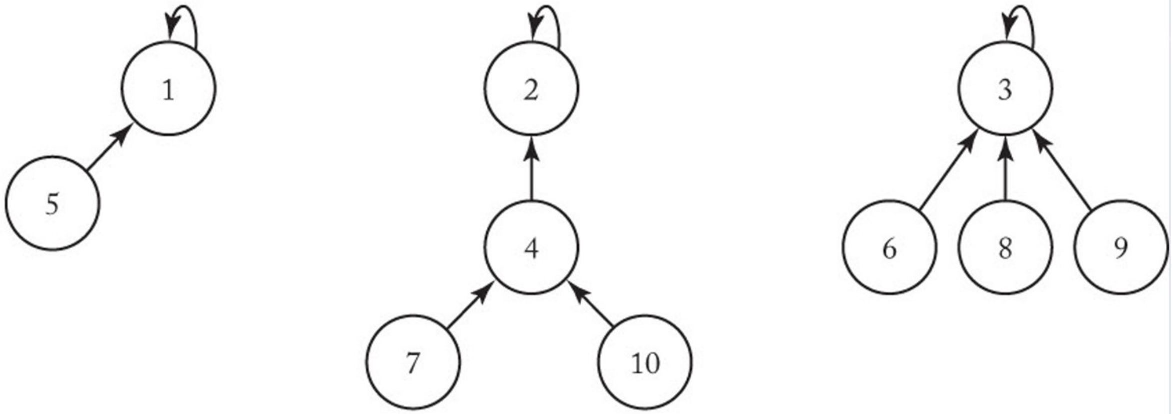
- `merge(4, 10)` places 10 in the same set as 4
    - U[10] is updated to 4, thus pointing node 10 to node 4.



Disjoint Set Data Structure

The following image represents:

{1, 5}, {2, 4, 7, 10}, {3, 6, 8, 9}



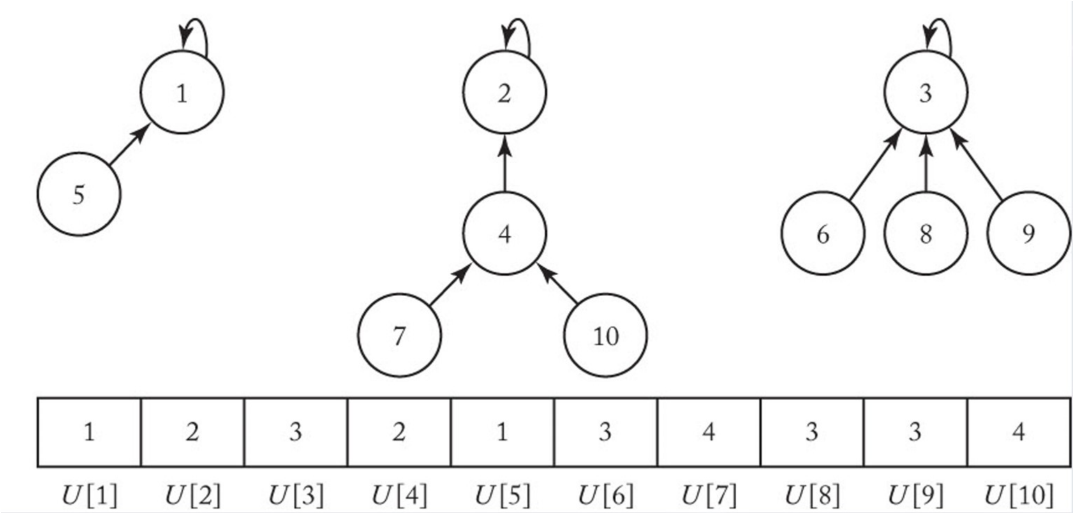
1	2	3	2	1	3	4	3	3	4
$U[1]$	$U[2]$	$U[3]$	$U[4]$	$U[5]$	$U[6]$	$U[7]$	$U[8]$	$U[9]$	$U[10]$

Disjoint Set Data Structure

Suppose we have:

```
p = find(10)
q = find(4)
```

What should `find` return?

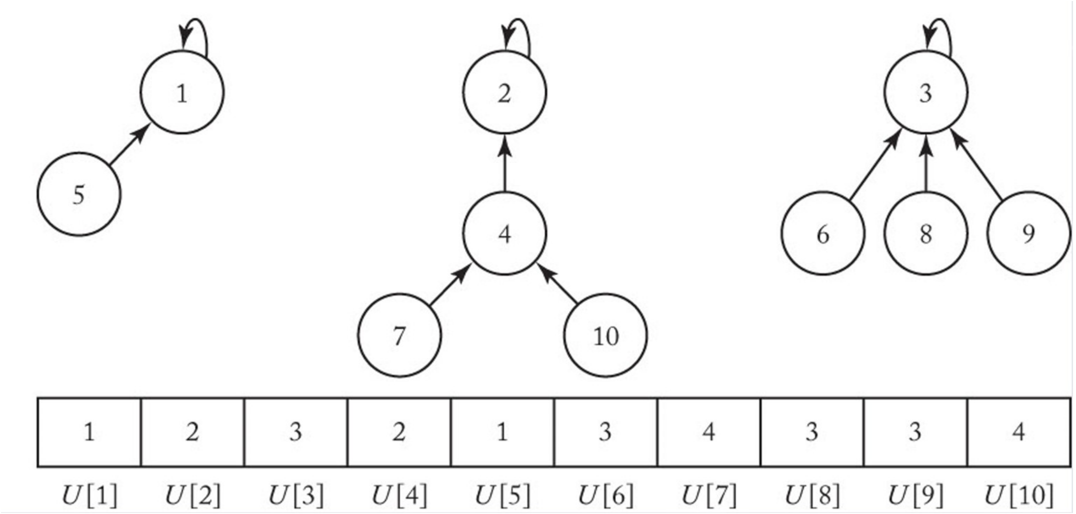


Disjoint Set Data Structure

Suppose we have:

```
p = find(10)
q = find(4)
```

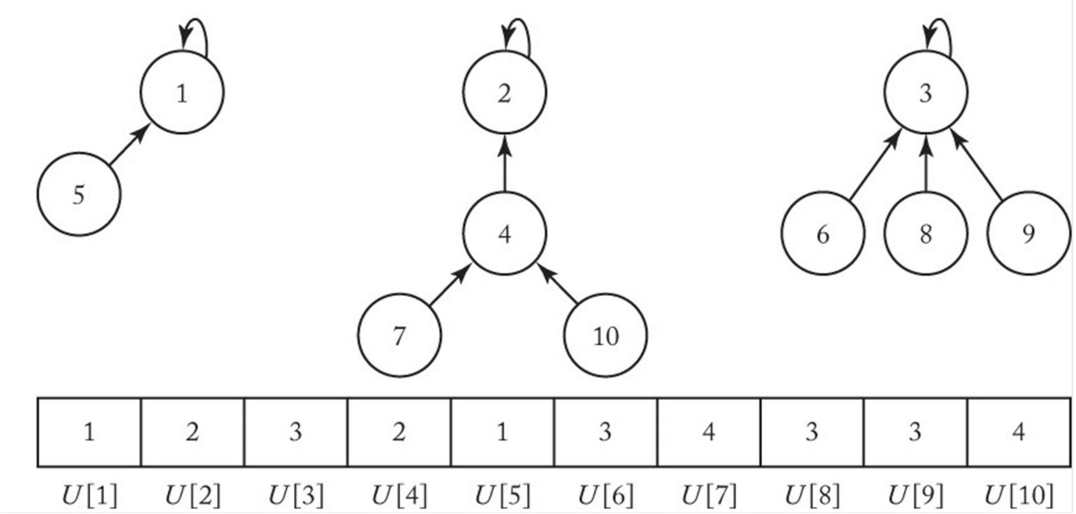
What should `find` return? The *root* of the tree the specified value is in.



Disjoint Set Data Structure

```
set_pointer find (index i)
     while (U[i] != i)
          i = U[i];
     return i;
```

This `find` procedure will return 2 for both `find(10)` and `find(4)`.



Disjoint Set Data Structure Definition

```
void makeset (index i)
          U[i] = i;

set_pointer find (index i)
     while (U[i] != i)
          i = U[i];
     return i;

void merge (set_pointer p, set_pointer q)
          if (p < q)
                    U[q] = p;                                       // p is now the parent of q
          else
                    U[p] = q;                                       // q is now the parent of p

bool equal (set_pointer p, set_pointer q)
          return p == q;
```

Kruskal's Algorithm

```
void kruskal(int n, int m, set_of_edges E, set_of_edges &F)
          edge e;
          Sort the m edges in E by weight in nondecreasing order;
          F = {}
          initial(n)                                              // initialize n disjoint
subsets
          while (# of edges in F is less than n - 1)
                    e = edge with least weight not yet considered;
                    index i, j = indices of vertices connected by e;
                    set_pointer p = find(i);
                    set_pointer q = find(j);
                    if (! equal(p, q))
                              merge (p, q);
                              add e to F;
```

Kruskal's Vs. Prim's

Prim's Algorithm: $\mathrm{T}(n) \in \Theta\ (n^2)$

Kruskal's Algorithm: $\mathrm{B}(m, n) \in \Theta\ (m\lg n)$ and $\mathrm{W}(m, n) \in \Theta\ (n^2\lg n)$

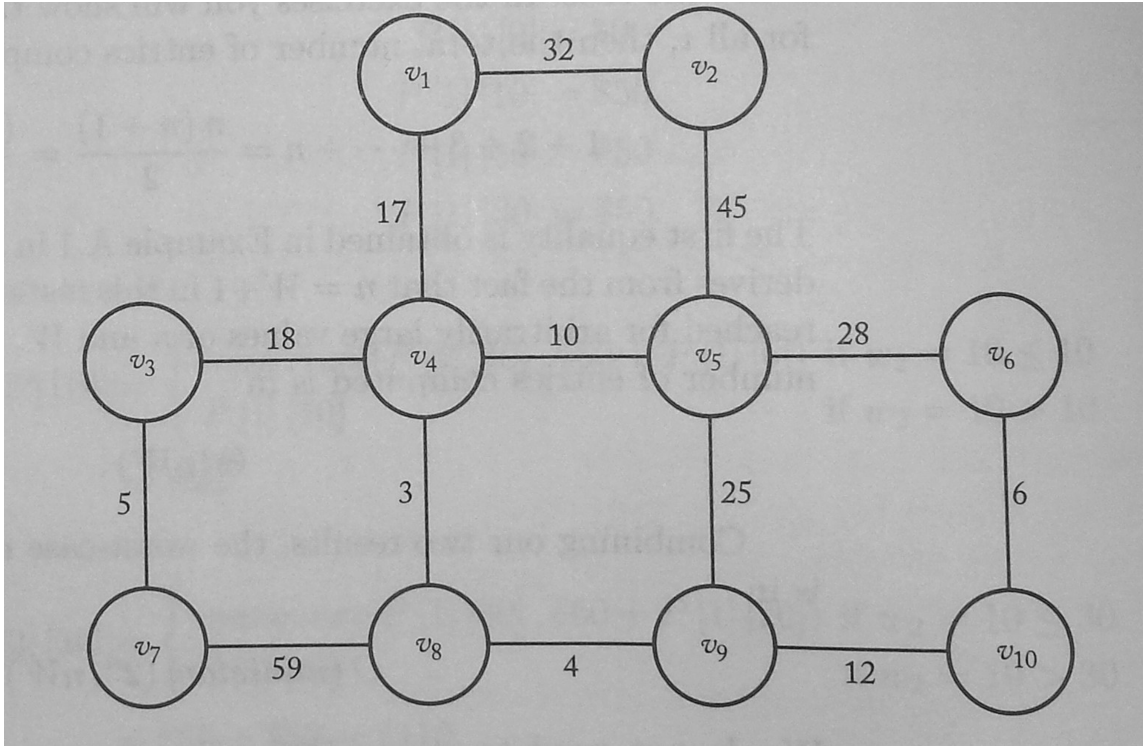
- m is the # of edges and n is the # of vertices

In a sparse graph with less edges, Kruskal's is $(n\lg n)$. In a dense graph with a large amount of edges, Kruskal's is $(n^2\lg n)$

Therefore, Prim should be used with graphs with lots of edges and Kruskal's should be used with graphs that have fewer edges.

In-Class Exercise

1. Use Kruskal's Algorithm to find a minimum spanning tree for the following graph



Scheduling With Deadlines

Imagine that we have a list of jobs to be done. Each job:

- Takes 1 unit of time.
- Provides a profit if completed.
- Must be finished by a specified deadline.

If a job starts before or at its deadline, the profit is obtained. How can we schedule the jobs so that maximum profit is obtained?

Note: Not all jobs have to be scheduled. We don't consider any schedule that has a job starting after its deadline since it has the same profit as one that doesn't have that job.

- ➢ Such a schedule is called **impossible**.

Scheduling With Deadlines

A **Deadline** of 2 means that the job can start at time 1 *or* time 2 (there is no time 0).

- What schedules are **possible** given the following list of jobs?

Job	Deadline	Profit
1	2	30
2	1	35
3	2	25
4	1	40

Scheduling With Deadlines

A **Deadline** of 2 means that the job can start at time 1 *or* time 2 (there is no time 0).

- What schedules are **possible** given the following list of jobs?

Impossible schedules have not been listed i.e. [2, 4], [2, 1, 3] etc.

Job	Deadline	Profit
1	2	30
2	1	35
3	2	25
4	1	40

Schedule	Total Profit
[1, 3]	30 + 25 = 55
[2, 1]	35 + 30 = 65
[2, 3]	35 + 25 = 60
[3, 1]	25 + 30 = 55
[4, 1]	40 + 30 = 70
[4, 3]	40 + 25 = 65

Scheduling With Deadlines

- Schedule [4, 1] is optimal. It provides a profit of 70.
    - However, considering every potential schedules takes *factorial* time!

Before we move forward, let's define a few terms:

- A sequence of jobs is a **feasible sequence** if all its jobs start by their deadlines.
    - [4, 1] is feasible but [1, 4] isn't; 4 has a deadline of 1 and can't start at time 2.
- A set of jobs is called a **feasible set** if there exists at least one feasible sequence for the jobs in the set.
    - {1, 4} is a *feasible set* since [4, 1] is a sequence that can be scheduled from the jobs in the set. However, {2, 4} is not feasible since both have a deadline of 1.
- A feasible sequence with maximum total profit is called an **optimal sequence**.
- The set of jobs in that sequence is called an **optimal set of jobs**.

Scheduling With Deadlines

High-Level pseudocode:

```
sort the jobs in nonincreasing order by profit;

S = {};

while (the instance is not solved)
        select next job;
        if (S is feasible with this job added)
                add this job to S;
        if (there are no more jobs)
                the instance is solved;
```

Scheduling With Deadlines

Job	Deadline	Profit
1	3	40
2	1	35
3	1	30
4	3	25
5	1	20
6	3	15
7	2	10

1. S is set to {}
2. S is set to {1}, because [1] is feasible
3. S is set to {1, 2} because [2, 1] is feasible
4. {1, 2, 3} is rejected because there is no feasible sequence for this set
5. S is set to {1, 2, 4} because [2, 1, 4] is feasible
6. {1, 2, 4, 5} is rejected.
7. {1, 2, 4, 6} is rejected
8. {1, 2, 4, 7} is rejected

The final value of S is {1, 2, 4}, and a feasible sequence for S is [2, 1, 4] (we could also use [2, 4, 1])

Total Profit: 100

Scheduling With Deadlines

When we add a job to the set S, we need to determine if it is still feasible. This is easy for a human on small sets, but takes factorial time for a computer.

What is an efficient way to determine this?

Scheduling With Deadlines

When we add a job to the set S, we need to determine if it is still feasible. This is easy for a human on small sets, but takes factorial time for a computer.

What is an efficient way to determine this?

- ➢ Sort the jobs in S in nondecreasing order by deadline.

Suppose we have the following set: {1, 2, 4, 7}

- Job 1 has a deadline of 3, job 2 a deadline of 1, job 4 a deadline of 3, and job 7 a deadline of 2.
- We sort the jobs in nondecreasing order by deadline: [2, 7, 1, 4]
  - Job 4 starts at time 4, but its deadline is 3...therefore, the set is <u>not</u> feasible.

Scheduling With Deadlines

Problem: determine the schedule with maximum total profit give that each job has a profit that will be obtained only if the job is scheduled by its deadline.

Inputs: *n*, the number of jobs. Array of integers `deadline`, indexed from 1 to *n*. `deadline[i]` is the deadline for the *i*th job. The jobs should be sorted in nonincreasing order by their profits.

Outputs: an optimal sequence `finalSequence` for the jobs.

Scheduling With Deadlines

```
void schedule (int n, const int deadline[], sequence_of_integers &finalSequence)
          index i;
          sequence_of_integers temp;

          finalSequence = [1];          // initialize finalSequence to contain the first 
job
          for (i = 2; i <= n; i++)
                    temp = finalSequence with job i added;
          sort temp in nondecreasing order by deadline[i];
          if (temp is feasible)
                    finalSequence = temp;
```

Scheduling With Deadlines

Job	Deadline	Profit
1	3	40
2	1	35
3	1	30
4	3	25
5	1	20
6	3	15
7	2	10

1. `finalSequence` is set to [1]
2. `temp` is set to [1, 2] and sorted to [2, 1]. It is feasible. `finalSequence` is set to [2, 1] since `temp` is feasible.
3. `temp` is set to [2, 1, 3] and sorted to [2, 3, 1]. It is not feasible and is rejected.
4. `temp` is set to [2, 1, 4] and is already sorted. `finalSequence` is set to [2, 1, 4] because `temp` is feasible.
5. `temp` is set to [2, 1, 4, 5], sorted to [2, 5, 1, 4]. Rejected.
6. `temp` is set to [2, 1, 4, 6], sorted to [2, 1, 6, 4]. Rejected.
7. `temp` is set to [2, 1, 4, 7], sorted to [2, 7, 1, 4]. Rejected.

The final value of `finalSequence` is [2, 1, 4].

In-Class Exercise

1. Consider the following jobs, deadlines, and profits. Use the Scheduling with Deadlines algorithm to maximize the total profit

Job	Deadline	Profit
1	2	40
2	4	15
3	3	60
4	2	20
5	3	10
6	1	45
7	1	55