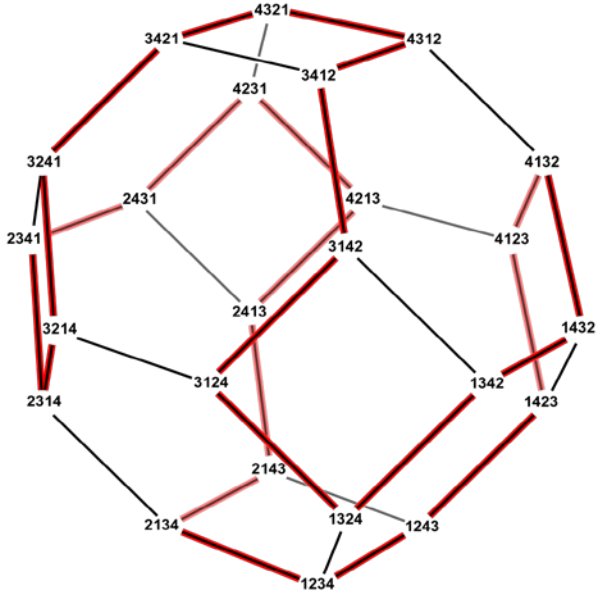
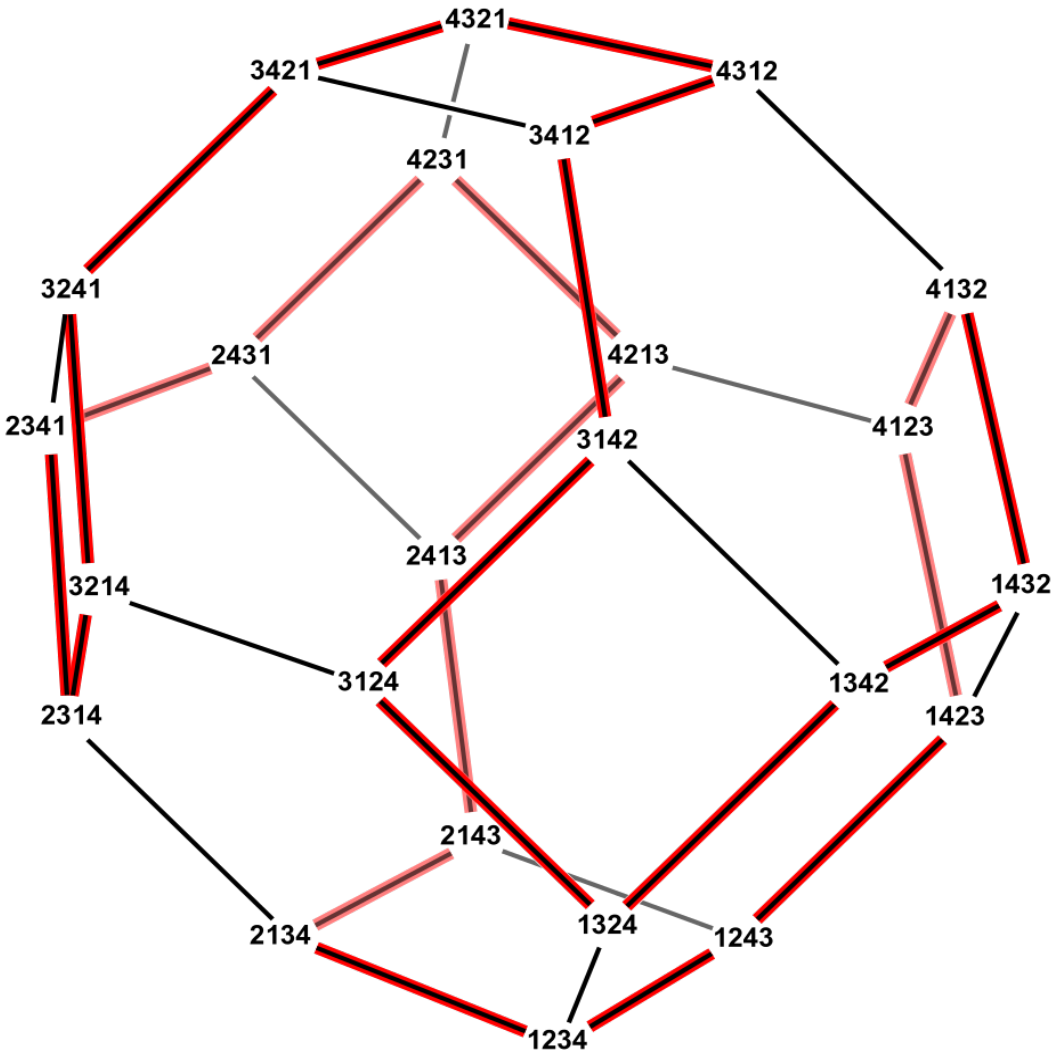
CS3334 Data Structures

Lecture 11: Permutation Generation, Disjoint Sets and Kruskal's Algorithm



Chee Wei Tan

Let's Play the Icosian Game



Another Challenge:
Selecting two vertices and ask them to trace out a path that passes through all other vertices exactly once while starting and returning to the selected vertices.
The Icosian Game gamifies a well-known NP-hard problem "Traveling Salesman Problem"
https://simple.wikipedia.org/wiki/Travelling_salesman_problem

Permutation Generation

Permutation Generation Methods*

ROBERT SEDGEWICK

Program in Computer Science and Division of Applied Mathematics
Brown University, Providence, Rhode Island 02912

This paper surveys the numerous methods that have been proposed for permutation enumeration by computer. The various algorithms which have been developed over the years are described in detail, and implemented in a modern ALGOL-like language. All of the algorithms are derived from one simple control structure.

The problems involved with implementing the best of the algorithms on real computers are treated in detail. Assembly-language programs are derived and analyzed fully.

The paper is intended not only as a survey of permutation generation methods, but also as a tutorial on how to compare a number of different algorithms for the same task

Key Words and Phrases: permutations, combinatorial algorithms, code optimization, analysis of algorithms, lexicographic ordering, random permutations, recursion, cyclic rotation.

CR Categories: 3.15, 4.6, 5 25, 5.30.

INTRODUCTION

Over thirty algorithms have been published during the past twenty years for generating by computer all $N!$ permutations of N elements. This problem is a nontrivial example of the use of computers in combinatorial mathematics, and it is interesting to study because a number of different approaches can be compared. Surveys of the field have been published previously in 1960 by D. H. Lehmer [26] and in 1970-71 by R. J. Ord-Smith [29, 30]. A new look at the problem is appropriate at this time because several new algorithms have been proposed in the intervening years.

Permutation generation has a long and distinguished history. It was actually one of the first nontrivial nonnumeric problems to be attacked by computer. In 1956, C. Tompkins wrote a paper [44] describing a number of practical areas where permutation generation was being used to solve problems. Most of the problems that he described are now handled with more sophisticated techniques, but the paper stimulated interest in permutation generation by computer *per se*. The problem is simply stated, but not easily solved, and is often used as an example in programming and correctness. (See, for example, [6]).

The study of the various methods that have been proposed for permutation generation is still very instructive today because together they illustrate nicely the relationship between counting, recursion, and iteration. These are fundamental concepts in computer science, and it is useful to have a rather simple example which illustrates so well the relationships between them. We shall see that algorithms which seem to differ markedly have essentially the same structure when expressed in a modern language and subjected to simple program transformations. Many readers may find it surprising to discover that "top-down" (recursive) and "bottom-up"

* This work was supported by the National Science Foundation Grant No. MCS75-23738

Bell ringing, in which a band of ringers plays long sequences of permutations on a set of peal bells, is a little-known but surprisingly rich and beautiful acoustical application of mathematics.

Mathematical Impressions, Simons Foundation: https://www.youtube.com/watch?v=3lyDCUKsWZs&t=1s

Kruskal's Algorithm

ON THE SHORTEST SPANNING SUBTREE OF A GRAPH AND THE TRAVELING SALESMAN PROBLEM

JOSEPH B. KRUSKAL, JR.

Several years ago a typewritten translation (of obscure origin) of [1] raised some interest. This paper is devoted to the following theorem: If a (finite) connected graph has a positive real number attached to each edge (the *length* of the edge), and if these lengths are all distinct, then among the spanning[1] trees (German: Gerüst) of the graph there is only one, the sum of whose edges is a minimum; that is, the shortest spanning tree of the graph is unique. (Actually in [1] this theorem is stated and proved in terms of the "matrix of lengths" of the graph, that is, the matrix $\|a_{ij}\|$ where a_{ij} is the length of the edge connecting vertices i and j. Of course, it is assumed that $a_{ij}=a_{ji}$ and that $a_{ii}=0$ for all i and j.)

The proof in [1] is based on a not unreasonable method of constructing a spanning subtree of minimum length. It is in this construction that the interest largely lies, for it is a solution to a problem (Problem 1 below) which on the surface is closely related to one version (Problem 2 below) of the well-known traveling salesman problem.

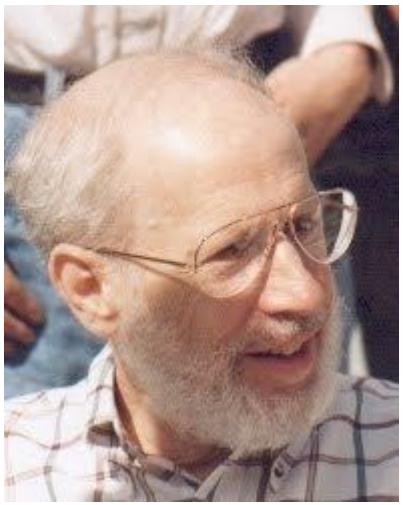
PROBLEM 1. Give a practical method for constructing a spanning subtree of minimum length.

PROBLEM 2. Give a practical method for constructing an unbranched spanning subtree of minimum length.

The construction given in [1] is unnecessarily elaborate. In the present paper I give several simpler constructions which solve Problem 1, and I show how one of these constructions may be used to prove the theorem of [1]. Probably it is true that any construction

Received by the editors April 11, 1955.

[1] A subgraph spans a graph if it contains all the vertices of the graph.



Joseph Bernard Kruskal, Jr. (1928 –2010) was an American mathematician, statistician, computer scientist and psychometrician. https://en.wikipedia.org/wiki/Joseph_Kruskal

Kruskal's Algorithm

ON THE SHORTEST SPANNING SUBTREE OF A GRAPH AND THE TRAVELING SALESMAN PROBLEM

JOSEPH B. KRUSKAL, JR.

Several years ago a typewritten translation (of obscure origin) of [1] raised some interest. This paper is devoted to the following theorem: If a (finite) connected graph has a positive real number attached to each edge (the *length* of the edge), and if these lengths are all distinct, then among the spanning[1] trees (German: Gerüst) of the graph there is only one, the sum of whose edges is a minimum; that is, the shortest spanning tree of the graph is unique. (Actually in [1] this theorem is stated and proved in terms of the "matrix of lengths" of the graph, that is, the matrix $\|a_{ij}\|$ where a_{ij} is the length of the edge connecting vertices i and j. Of course, it is assumed that $a_{ij}=a_{ji}$ and that $a_{ii}=0$ for all i and j.)

The proof in [1] is based on a not unreasonable method of constructing a spanning subtree of minimum length. It is in this construction that the interest largely lies, for it is a solution to a problem (Problem 1 below) which on the surface is closely related to one version (Problem 2 below) of the well-known traveling salesman problem.

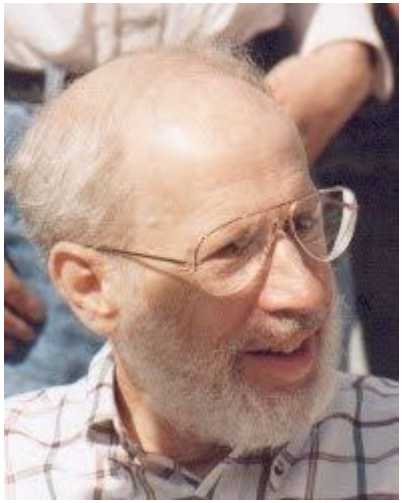
PROBLEM 1. Give a practical method for constructing a spanning subtree of minimum length.

PROBLEM 2. Give a practical method for constructing an unbranched spanning subtree of minimum length.

The construction given in [1] is unnecessarily elaborate. In the present paper I give several simpler constructions which solve Problem 1, and I show how one of these constructions may be used to prove the theorem of [1]. Probably it is true that any construction

Received by the editors April 11, 1955.

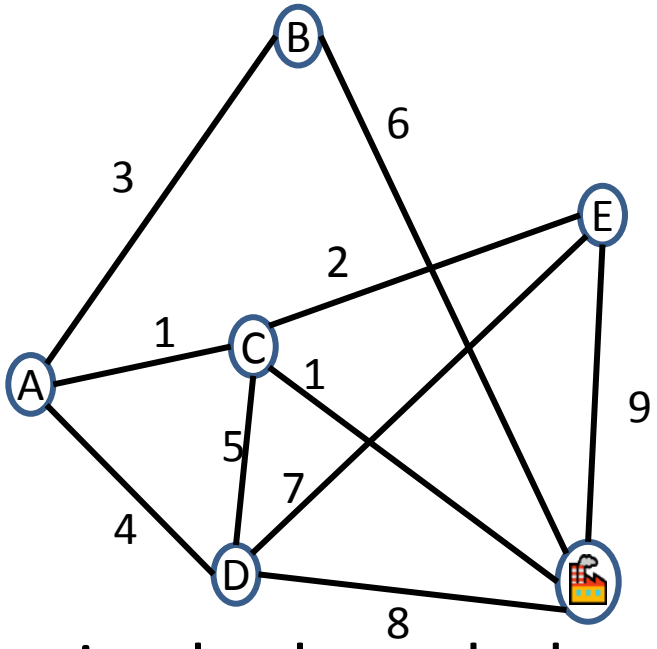
[1] A subgraph spans a graph if it contains all the vertices of the graph.



Joseph Bernard Kruskal, Jr. (1928 –2010) was an American mathematician, statistician, computer scientist and psychometrician. https://en.wikipedia.org/wiki/Joseph_Kruskal

Minimum Spanning Trees

- Build wires to connect all these cities to the plant.



- Find the maximal subgraph that spans the entire given graph with least costs.

Kruskal's Algorithm

which solves Problem 1 may be used to prove this theorem.

First I would like to point out that there is no loss of generality in assuming that the given connected graph G is complete, that is, that every pair of vertices is connected by an edge. For if any edge of G is "missing," an edge of great length may be inserted, and this does not alter the graph in any way which is relevant to the present purposes. Also, it is possible and intuitively appealing to think of missing edges as edges of infinite length.

CONSTRUCTION A. Perform the following step as many times as possible: Among the edges of G not yet chosen, choose the shortest edge which does not form any loops with those edges already chosen. Clearly the set of edges eventually chosen must form a spanning tree of G, and in fact it forms a shortest spanning tree.

https://en.wikipedia.org/wiki/Kruskal%27s_algorithm#/media/File:KruskalDemo.gif

Kruskal's Algorithm

- Models the algorithm as a collection of disjoint sets, each of which contains the nodes of a particular component
- Initially, each node is in a component by itself: makeset(x)
- Repeatedly test pairs of nodes to see if they belong to the same set.
  - Find (x): to which set does x belong?
  - And whenever we add an edge, we merge two components
  - Union(x,y): merge the sets containing elements x and y
- This data structure is also called a union-find data structure or merge-find set.

Disjoint-set Data Structure (1/2)

- A disjoint-set data structure organizes a set of elements partitioned into a number of disjoint (non-overlapping) subsets.
- Operation makeset(x) creates a singleton consisting of the element x only
- Two operations
  - Find (x): return the name of the set containing a given element x
  - Union(x,y): merge two elements' subsets into a single subset

Disjoint-set Data Structure (2/2)

- If two elements x and y are in the same subset, then Find(x) = Find(y); otherwise, Find(x) != Find(y) .
- If two elements x and y are not in the same subset (i.e., x and y are in s_i and s_j, respectively, where $s_i \neq s_j$), connect them by a union operation, i.e., Union(s_i, s_j).

Example

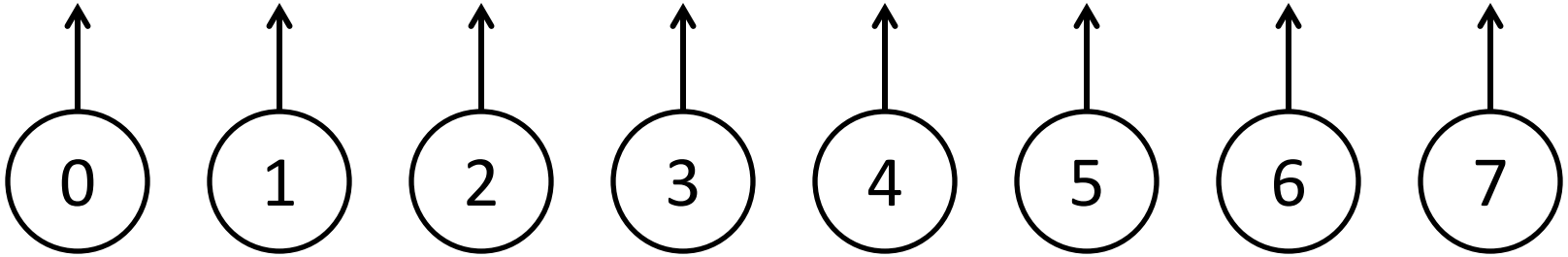
- Organize a set of pairwise disjoint sets
  - {3}, {2, 4, 8}, {1, 6}, {9}
- Each set has a unique name, i.e., underlined number
  - {3}, {2, 4, 8}, {1, 6}, {9}
  - The name of {3} is 3.
- Union(6, 9)
  - {3}, {2, 4, 8}, {1, 6, 9}
- Find(6) = 6 and Find(2) = 4

Implementation

- Assign a unique identity to each element (i.e., 0, 1, 2, 3, etc.)
- Initially, each subset contains a unique element.
- Use a tree structure to organize a subset of elements; disjoint sets form a forest (i.e., a collection of trees)
- The root of a tree = the name of a subset
- Use an array s[i] to keep track the parent of each element i
  - If element j is a root, s[j] = -1 (i.e., a special value)

Examples (1/4)

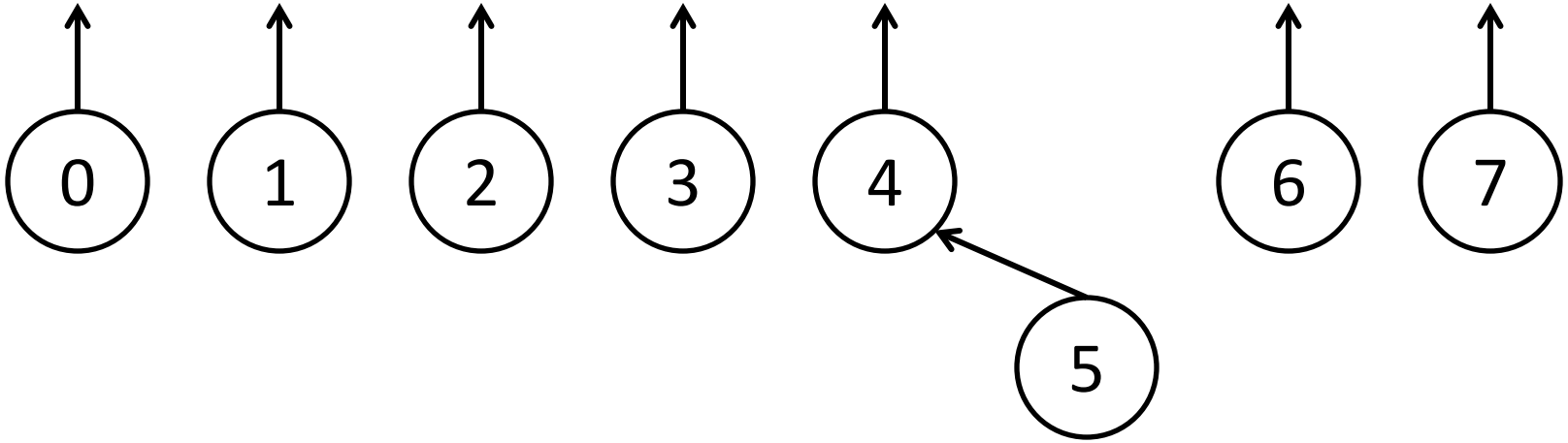
- Initially, 8 elements and 8 subsets



-1	-1	-1	-1	-1	-1	-1	-1
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**

Examples (2/4)

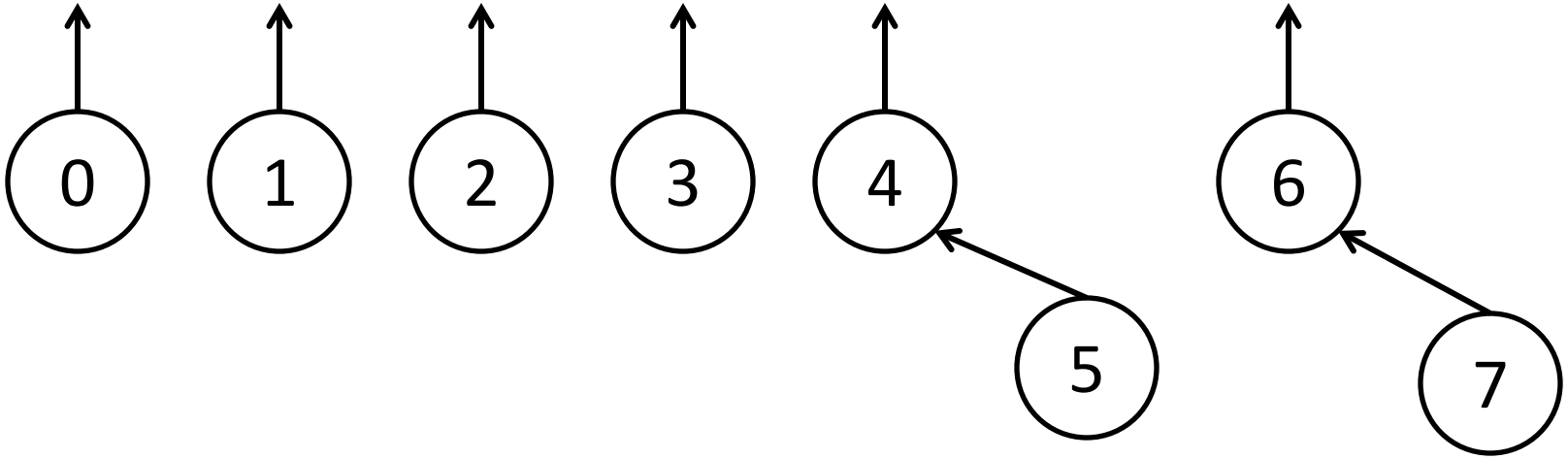
- Union (4,5)



-1	-1	-1	-1	-1	4	-1	-1
0	1	2	3	4	5	6	7

Examples (3/4)

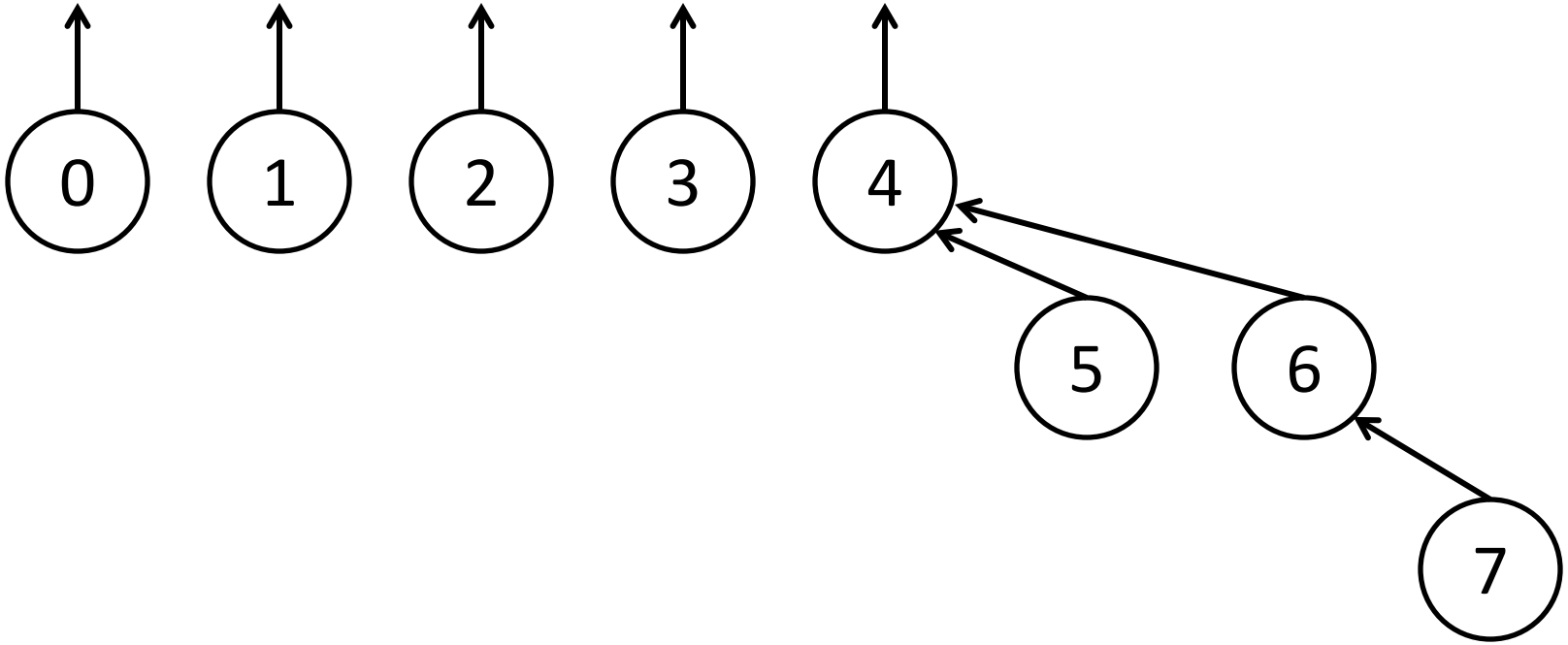
- Union (6,7)



-1	-1	-1	-1	-1	4	-1	6
0	1	2	3	4	5	6	7

Examples (4/4)

- Union (4,6)



-1	-1	-1	-1	-1	4	4	6
0	1	2	3	4	5	6	7

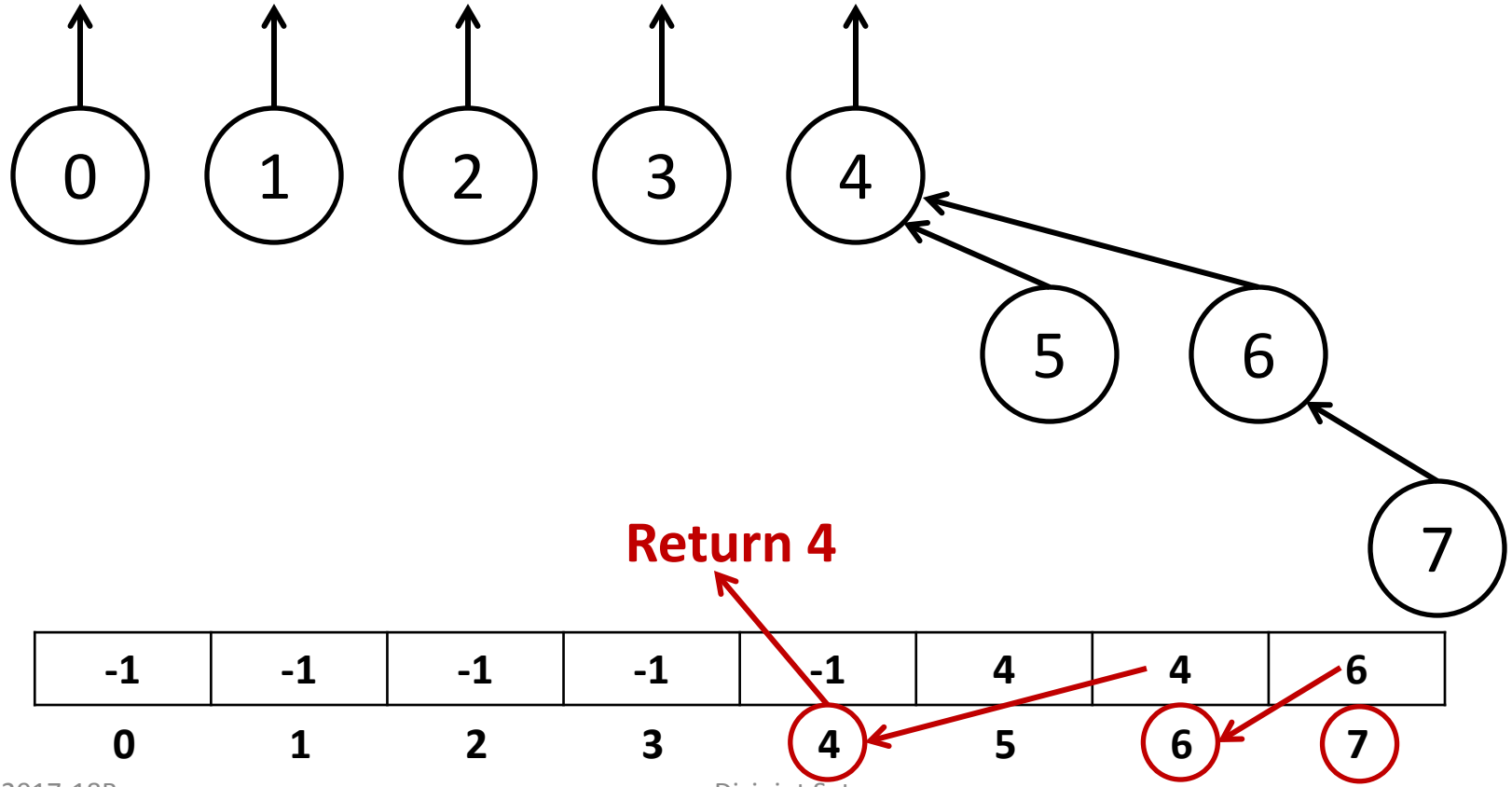
Find(x)

- Find (x) returns the name of the set containing x
- Basic ideas
  - If s[x] is a root, return x (base case)
  - Otherwise, call Find(s[x]) for x's parent recursively until the base case takes place
- Code

```
int DisjSets::Find(int x) const {
   if (s[x] < 0) return x;
   else
        return Find(s[x]);
}
```

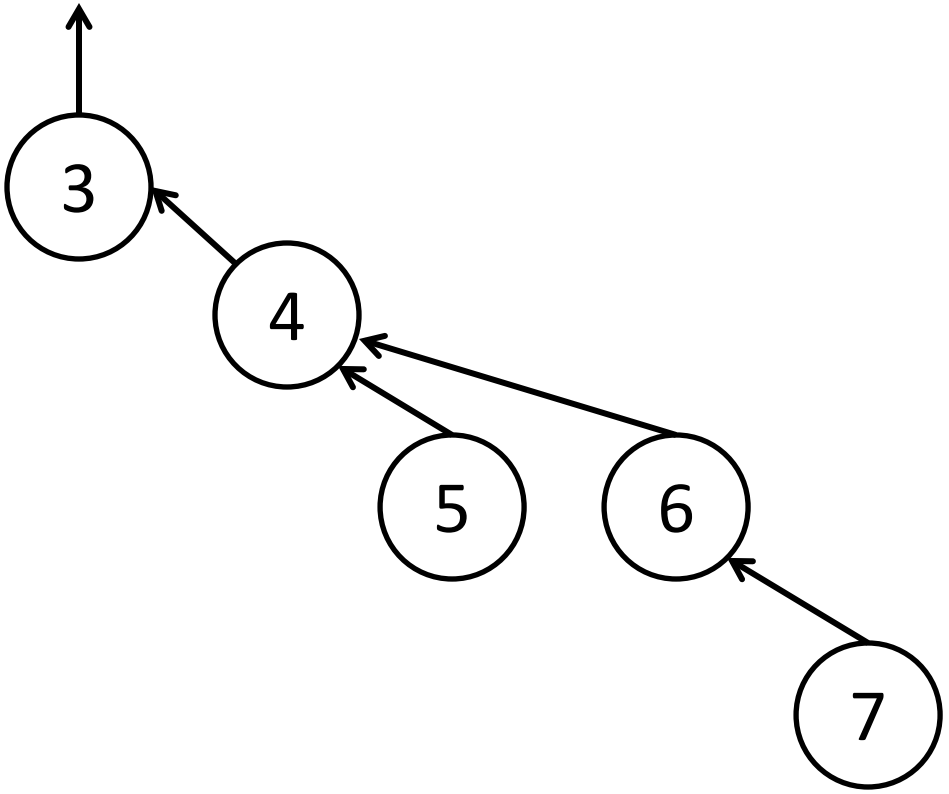
Example

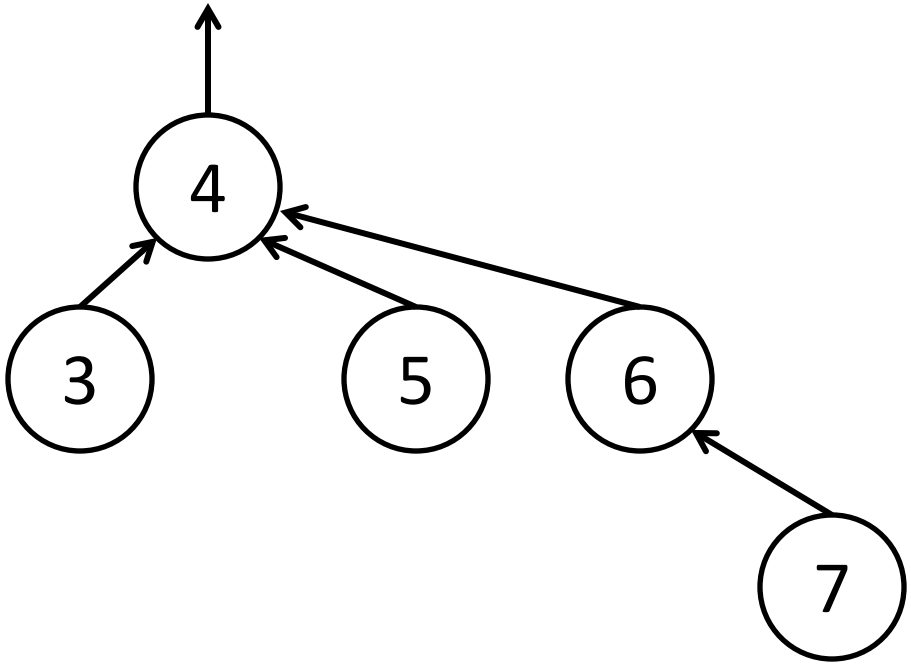
- Find (7)



Smart Union

- Which result is better for Find(7) after Union(3,4)?





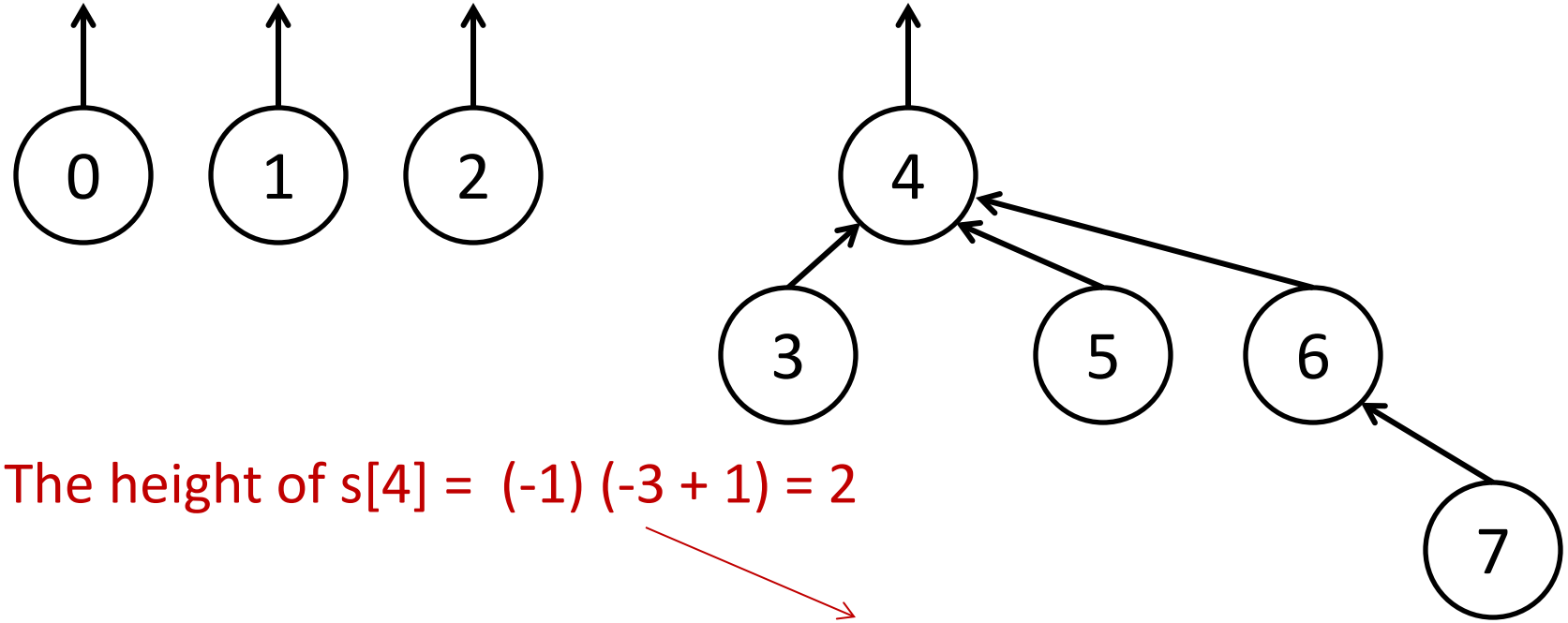
Smaller height => better

Union-by-Height (1/2)

- Guarantee all the trees have depth at most O(log N), where N is the total number of elements
- Running time for a find operation is O(log N)
- Keep track of the height of each tree
  - Store the negative of height, minus an additional one
  - The height of a tree increases (minus one) only when two equally deep trees are joined.

Union-by-Height (2/2)

- After Union(3,4)



The height of s[4] = (-1) (-3 + 1) = 2

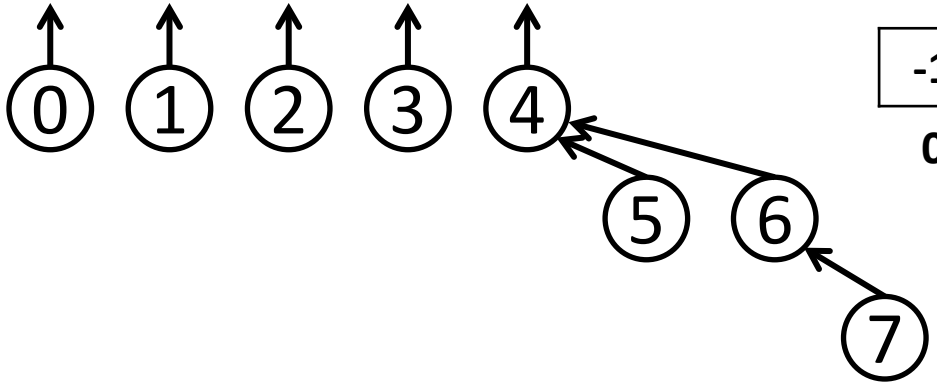
-1	-1	-1	4	-3	4	4	6
0	1	2	3	4	5	6	7

Union-by-Height Code

```
void DisjSets::Union(int root1, int root2) {
   if (s[root2] < s[root1])//root2 is deeper
      s[root1] = root2;//make root2 new root
   else
   {
      if(s[root1] == s[root2])
        s[root1]--;//increase its height
      s[root2] = root1;//make root1 new root
   }
}
```

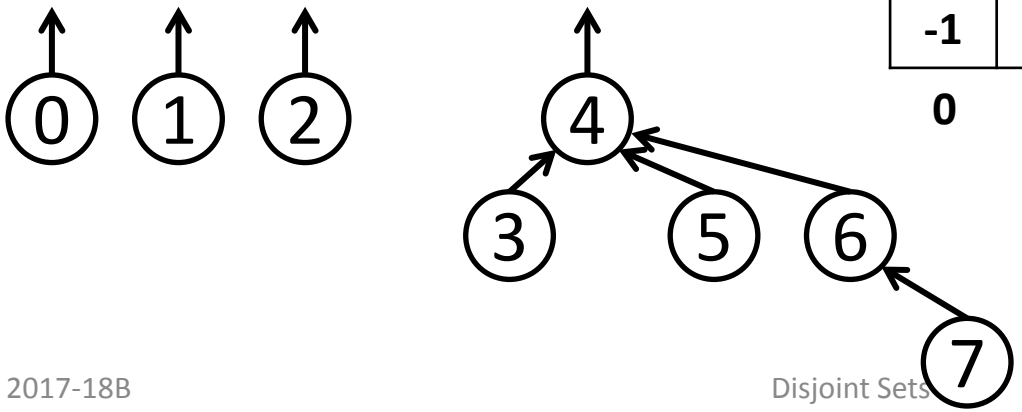
Union-by-Height Example

- Before Union(3,4): s[4] < s[3] (i.e. case 1)



-1	-1	-1	-1	-3	4	4	6
0	1	2	3	4	5	6	7

- After Union(3,4): s[3] = 4



-1	-1	-1	4	-3	4	4	6
0	1	2	3	4	5	6	7

Code

```
class DisjSets {
   public:
      DisjSets(int N){
         s = new int[N];
         for (int i=0; i<N; i++)
            s[i]=-1;
      }
      int Find(int x) const;
      void Union(int root1, int root2);
   private:
      int *s;
};
```

Disjoint-Set Theorem

- **Theorem:** Union-by-height guarantees that all the trees have depth at most O(log N), where N is the total number of elements.
- **Lemma 1.** For any root x, size(x) ≥ $2^{\text{height(x)}}$, where size(x) is the number of nodes of x's tree, including x.

Proof of Lemma 1 (1/3)

- Proof (by induction)
  - **Base case:** At beginning, all heights are 0 and each tree has size 1.
  - **Inductive step:** Assume true just before Union(x,y). Let size'(x) and height'(x) (or size'(y) and height'(y)) be the size and length of the merged tree after union, respectively.

Proof of Lemma 1 (2/3)

- **Case 1. height(x) < height(y)**
  Then, size'(y) = size(x) + size(y)
  $\geq 2^{height(x)} + 2^{height(y)}$
  $\geq 2^{height(y)}$
  $= 2^{height'(y)}$
- **Case 2. height(x) = height(y)**
  Then, size'(y) = size(x) + size(y)
  $\geq 2^{height(x)} + 2^{height(y)}$
  $\geq 2(2^{height(y)})$
  $\geq 2^{height(y)+1}$
  $= 2^{height'(y)}$

Proof of Lemma 1 (3/3)

- **Case 3. height(x) > height(y)**
  Then, size'(x) = size(x) + size(y)

$$\geq 2^{\text{height}(x)} + 2^{\text{height}(y)}$$

$$\geq 2^{\text{height}(x)}$$

$$= 2^{\text{height}'(x)}$$

Proof of Disjoint-Set Theorem

- By Lemma 1, size(x) ≥ $2^{height(x)}$.
- Let n=size(x) and h=height(x).
- Every node has size n ≥ 2^h
  => log n ≥ h
  => h = O(log n)

Kruskal's Algorithm and Beyond

For Kruskal's algorithm, how many times do you call

- makeset operations
- find operations
- union operations

This is the complexity of finding a minimum spanning tree in a given graph

Can you describe an algorithm to solve the Traveling Salesman Problem?

Hint: Use your favorite sorting algorithm (Lecture 4-7) to sort edges in the graph