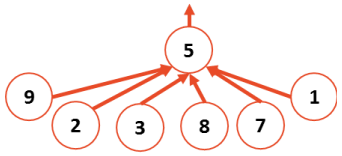
CS 225 Spring 2019 :: TA Lecture Notes
4/5 Disjoint Sets Finale and Graphs Intro

By Wenjie

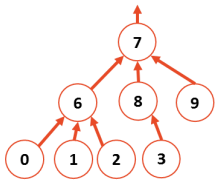
Disjoint Set Find

```
1  int DisjointSets::find() {
2    if ( s[i] < 0 ) { return i; }
3    else { return _find( s[i] ); }
4  }
```

- Algorithm
  - If we have some negative number, we have the root
  - If not, we recursively call find on the parent node
- Running time : $O(h) <= O(n)$.
- The ideal UpTree: every element is the direct child of the root!



Smart Unions



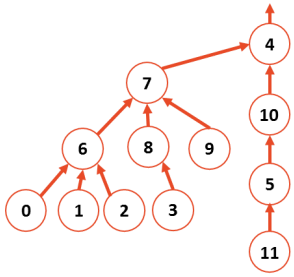
0	1	2	3	4	5	6	7	8	9	10	11
6	6	6	8	-1	10	7	-1	7	7	4	5

- **Union by height** (*minimize height*):
  - root := -h-1

0	1	2	3	4	5	6	7	8	9	10	11
6	**6**	**6**	**8**	**-4**	**10**	**7**	**-3**	**7**	**7**	**4**	**5**

CS 225 Spring 2019 :: TA Lecture Notes
4/5 Disjoint Sets Finale and Graphs Intro

By Wenjie



- After union by height we have

0	1	2	3	4	5	6	7	8	9	10	11
6	**6**	**6**	**8**	**-4**	**10**	**7**	**4**	**7**	**7**	**4**	**5**

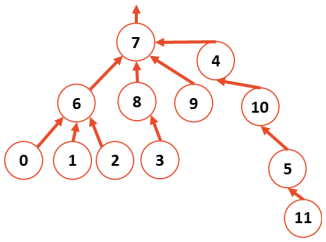
- Union by height worsen the average case (the majority of the nodes, namely the tree with root 7, has height increased by 1)

- **Union by Size** (*Minimize the number of nodes that increase in height*):

0	1	2	3	4	5	6	7	8	9	10	11
6	**6**	**6**	**8**	**-4**	**10**	**7**	**-8**	**7**	**7**	**4**	**5**

- **root := -n :** where n is the size of the tree

0	1	2	3	4	5	6	7	8	9	10	11
6	**6**	**6**	**8**	**7**	**10**	**7**	**-12**	**7**	**7**	**4**	**5**



CS 225 Spring 2019 :: TA Lecture Notes
4/5 Disjoint Sets Finale and Graphs Intro

By Wenjie

    - Union by size keeps the average case average, but worsen the worst case (11 gets height increased by one)
- (Also there's **union by rank**, which keeps track of the number of unions happened to a certain tree)
- Both guarantee the height of the tree to be **O(log n)**

The code

- **Union by size**

```
1  int DisjointSets::find(int i) {
2    if ( s[i] < 0 ) { return i; }
3    else { return _find( s[i] ); }
4  }
```

```
1  void DisjointSets::unionBySize(int root1, int root2) {
2    int newSize = arr_[root1] + arr_[root2];
3
4    // If arr_[root1] is less than (more negative), it is the larger set;
5    // we union the smaller set, root2, with root1.
6    if ( arr_[root1] < arr_[root2] ) {
7      arr_[root2] = root1;
8      arr_[root1] = newSize;
9    }
10
11   // Otherwise, do the opposite:
12   else {
13     arr_[root1] = root2;
14     arr_[root2] = newSize;
15   }
16 }
```

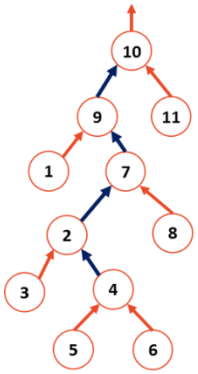
    - Running time of find is $O(h) = O(log\ n)$ since we are using the smart union; Running time of union is $O(1)$ given that **root1** and **root2** are actual root nodes.
    - **root1** and **root2** must be root nodes! So first we have to run **find**; therefore, the running time will be $O(h) = O(log\ n)$

Path Compression

- Look at the following UpTree:

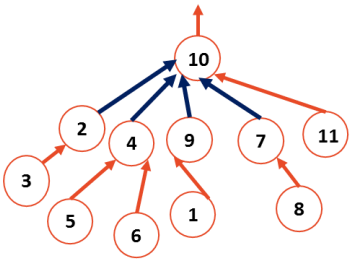
CS 225 Spring 2019 :: TA Lecture Notes
4/5 Disjoint Sets Finale and Graphs Intro

By Wenjie



0	1	2	3	4	5	6	7	8	9	10	11
		7		2			9		10	- n	

- Find is a recursive function, we have the following recursive calls:
  - Find(4) → Find(2) → Find(7) → Find(9) → Find(10)
- The last call to find gives us the base case, -10, which is the same for all the elements in the tree. If we make all the elements point to 10, running time of the find will decrease for each element (after this, it will take O(1) time to find them). To do so, when we unwind the recursion, we can update the value for each element. After running Find(4), we will have:



0	1	2	3	4	5	6	7	8	9	10	11
		10		10			10		10	-n	

- Since we are updating values as we unwind the recursion calls, no 'extra' time is spent on path compression. Note: other elements (that are not called during find(4)) are still pointing to the same old nodes.
- Path compression decrease the height of the tree. After running find infinite number of times, running time of find will become constant.
- Implementation

	disjointSet.cpp
1 2	`int DisjointSets::_find(int i) {` `  if (arr_[i] < 0)`

CS 225 Spring 2019 :: TA Lecture Notes
4/5 Disjoint Sets Finale and Graphs Intro

By Wenjie

```
3       return i;
4     else {
5       int root = _find(arr[i]);
6       arr[i] = root;
7       return root;
8     }
```

Running time with Path Compression

- Running time of find is getting closer and closer to constant time when we are using path compression. However, the first call (or some calls from time to time) will still take $O(\log n)$ and that will dominate overall runtime, so the algorithm is not O(1) time.
- To define running time for this algorithm, we are going to use a new mathematical notation:
- The iterated log function **log**:
    - The number of times you can take a log of a number:

```
log*(n) = 0                  , n ≤ 1
          1 + log*(log(n))   , n > 1
```

    - Example:
        - $lg^* 2^{65536} = 1 + lg^* 65536 = 2 + lg^* 16 = 3 + lg^* 4 = 4 + lg^* 2 = 1 + lg\ 1 = 5$
    - This is a ***very*** slowly growing function
    - We can't say that the algorithm runs in constant time, but we can say that the algorithm runs *really* close to the constant time ($lg^* 2^{65536}$ only takes 5 operations).
    - Iterated log is one of the smallest growing mathematical function. It is still in terms of n, but every time we do path compression, we are decreasing the size of the tree by $\log n$.
- The result:
    - Any sequence of m union and find operations results in the worst case running time of $O(m * lg^* n)$ (average cost per operation is $O(lg^* n)$), where n is the number of items in the Disjoint Sets.

CS 225 Spring 2019 :: TA Lecture Notes
4/5 Disjoint Sets Finale and Graphs Intro

By Wenjie

- Since $log^* n$ is really close to constant, disjoint sets' algorithms are considered **effectively constant** when used with other algorithms (when it is used by itself, running time is still $O(log^* n)$.
- So we say it's O(1)!

Panorama of Data Structures:

- We can divide all major data structures studied so far into two major groups:
  - **Array Based (Cache-optimized)**
    - Sorted Array
      - good Binary Search **O(lg n)**
      - terrible insert **O(n)**
    - Unsorted Array
      - good insert **O(1),** bad search **O(n)**
      - Stacks, Queues, providing limited ordering
      - Hashing, Heaps, Priority Queues, UpTrees (Disjoint Set).
  - **List/Pointer Based**
    - Singly Linked Lists, Doubly Linked List, Skip Lists (didn't appear)
    - Trees
      - BTree
      - Binary Tree
        - Huffman Encoding
        - kd-Tree
        - AVL Tree.
- When we are deciding on which data structure to use, we need to think about what we want to do with our data. For example:
  - If we want fast search, we should maintain sorted array on which we can perform a gold standard search algorithm binary search (in O(lg(n)) time).
  - On the other hand if we want fast insert and remove, we should use unsorted array. In on of the MPs we used unsorted array in the form of stack to perform traversals.

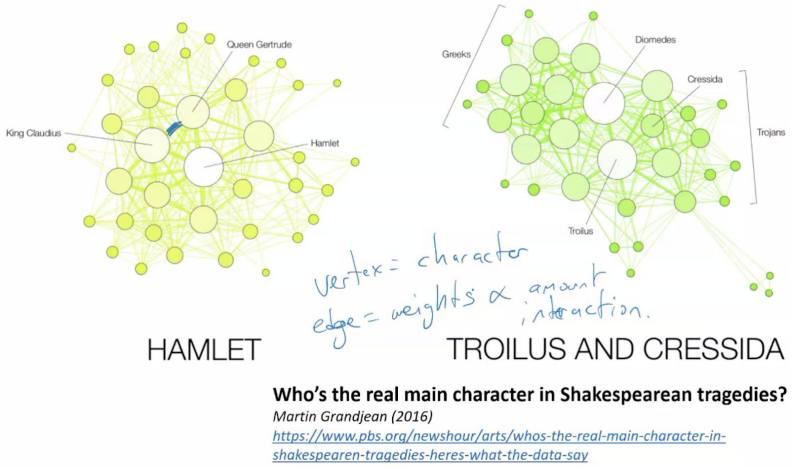
Graphs!

- Representing interconnected data

CS 225 Spring 2019 :: TA Lecture Notes
4/5 Disjoint Sets Finale and Graphs Intro

By Wenjie

- Various applications
    - Natural Language Processing (NLP)
    - Rush Hour Strategy
        - Nodes represent game states.
        - Edges represent moves (every move generates a different game state).
        - We need to traverse the graph to get to the solution (green nodes).
        - Then a solution corresponds to a valid path in the graph!
    - Data visualisation
        - Character connections in Shakespeare plays
        - Wade's facebook friend network

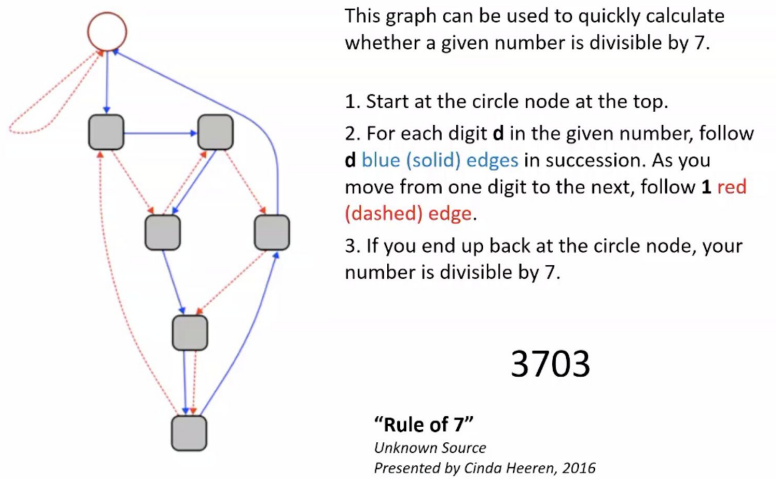


Who's the real main character in Shakespearean tragedies?
Martin Grandjean (2016)
https://www.pbs.org/newshour/arts/whos-the-real-main-character-in-shakespearen-tragedies-heres-what-the-data-say

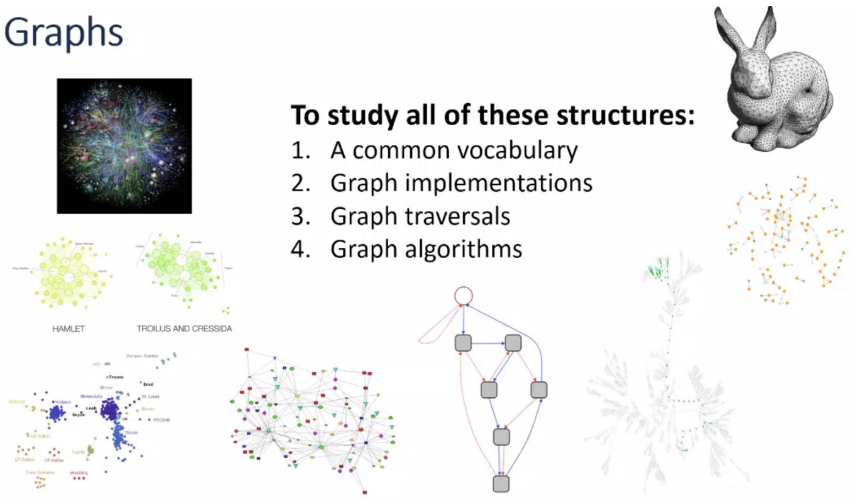
    - Divisibility problem :: "Rule of 7"
        - We know the rules to decide if a value is divisible by certain numbers.

CS 225 Spring 2019 :: TA Lecture Notes
4/5 Disjoint Sets Finale and Graphs Intro

By Wenjie



- Scheduling a conflict free exam
  - Turns into graph coloring problem → how to color a graph such that no two nodes of the same color are connected?
- Class Hierarchy
  - Directed graph showing what classes are prerequisites for other classes.



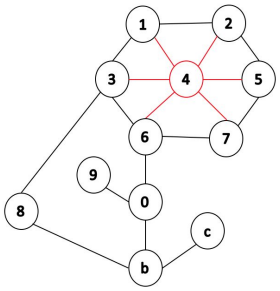
Graph Vocabulary

- size of the **vertices** $|V| = n$
- size of the **edges** $|E| = m$

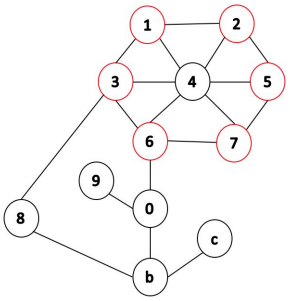
CS 225 Spring 2019 :: TA Lecture Notes
4/5 Disjoint Sets Finale and Graphs Intro

By Wenjie

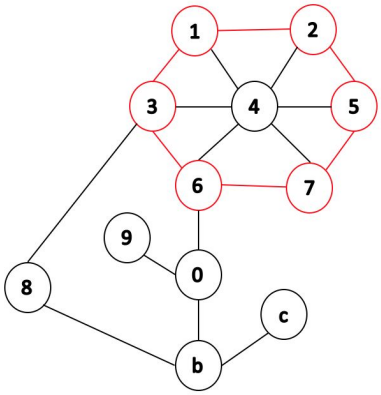
- **Incident edges**: all edges that connected to that node.
  Example: incident edges to 4 are (4,1), (4,2), (4,3), (4,5), (4,6), and (4,7).



- **Degree**: the number of incident edges.
  Example: the degree of vertex 4 is 6 because it has 6 incident edges.
- **Adjacent vertex**: a vertex at the other end of the incident edge.
  Example: adjacent vertices to 4 are 1, 2, 3, 5, 6, and 7.



- **Path**: a sequence of vertices connected by edges.
  Example: a path from 1 to b includes visiting nodes 4, 6, and 0.
- **Cycle**: a path with common beginning and end.



CS 225 Spring 2019 :: TA Lecture Notes
4/5 Disjoint Sets Finale and Graphs Intro

By Wenjie

- **Simple graph**: a graph with no self-loop edges and no multi-edges.