### **Disjoint Sets ADT**

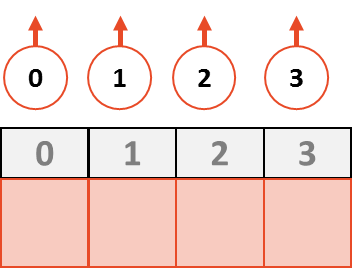
* Maintain a collection *(a set of disjoint sets).*
* Each set has an element as its representatives
* API:
  + void makeSet(const T & t); *(make set with one element)*
  + void union(const T & k1, const T & k2); *(set1 + set2)*
  + T & find(const T & k); *(find the representative element)*

### **Implementation 1**

* The array indices are the keys of the elements. Any type of elements could be converted to int through a hash function
* Find(k): O(1)
* Union(k1, k2):
  + Naive implementation: going through entire array to update representations
  + O(n)

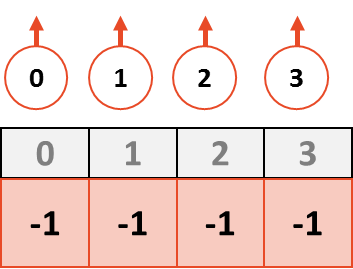
### **Implementation 2**

* The array indices are the keys of elements
* The value of the array at index **i** would be
  + **-1**: if **i** is a representative element
  + **The** **index of the parent of i**: if we haven’t found the rep. element.
* We call these **UpTrees**

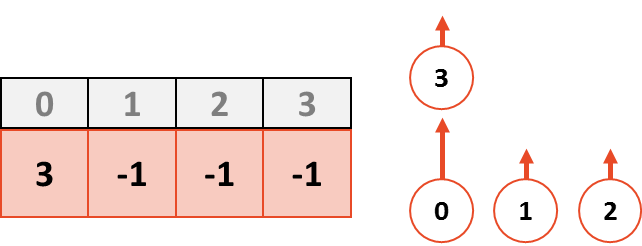
****

#### **Example of Implementation 2**

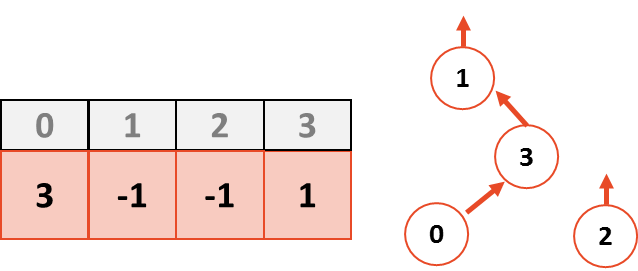
* Initial state:



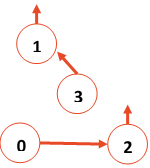
* **Union (3,0)** - we are going to point 0 to 3 and update the value for index 0



* **Union(1, 3)** – 3 will point to 1 and we will update the value for index 3:

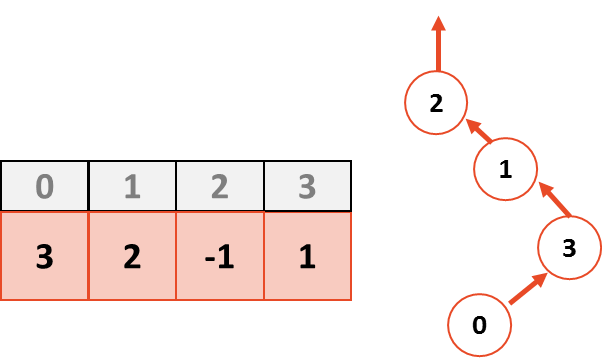


* **Union(2,0):**
  + **BAD PRACTICE:** if we just follow the previous step, point 0 to 2, we get

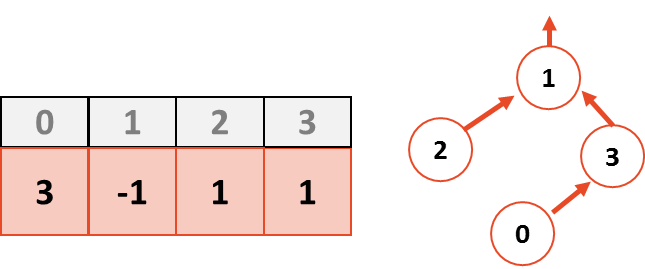


which is **is not good**

* + Instead, we need to union those roots
  + Union(find(2), f ) = Union(2, 1)



* Notice that, this is **NOT** the unique UpTree created by this set
  + We can also do Union(1, 2) and we would get:

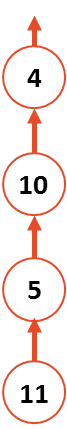


* + This gives us a better tree since the height is smaller
  + We resolve this issue later - if we want the shortest UpTree

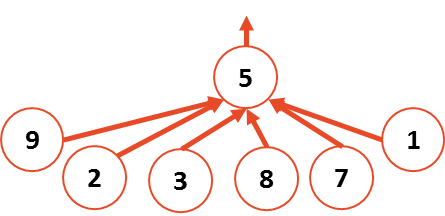
### **Disjoint Set Find**

****

* Algorithm
  + If we have **-1**: we have the root
  + If not, we recursively call find() on the parent node
* Running time
  + worst case could be

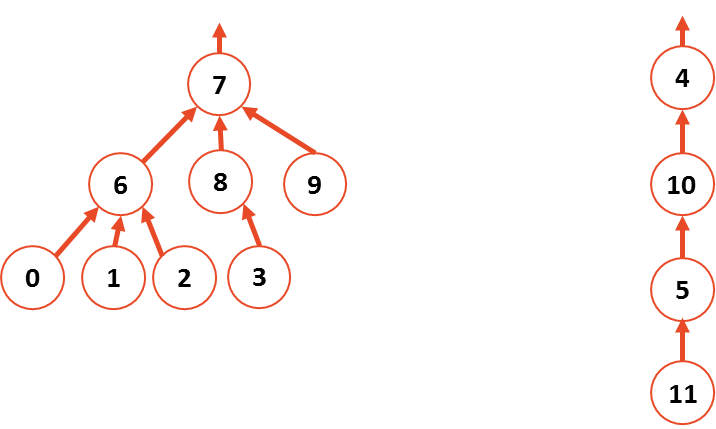


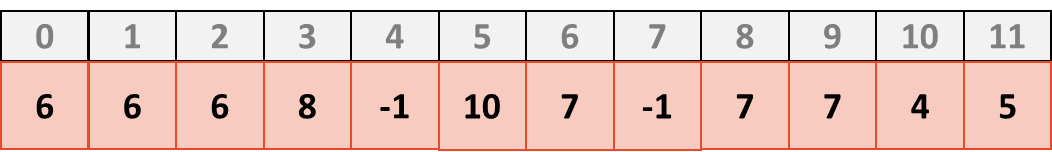
* The ideal UpTree: every element is the direct child of the root!

****

* + O(1) time!

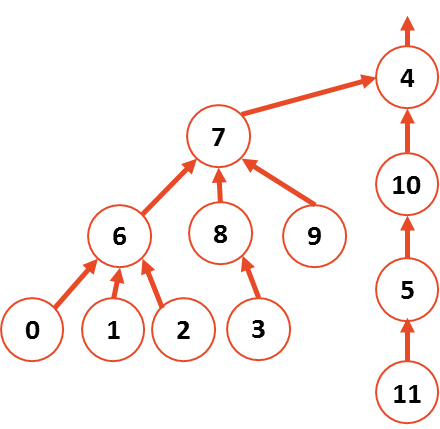
### **Disjoint Set Union**

****

****

* We want 7 a child of 4 (or vice versa), since that makes the total height smaller
* **Union by height** (*Keep the height of the tree as small as possible*):
  + Make root of a taller tree the parent of the root of the shorter tree (add the shorter tree to the taller tree);
  + For this approach, we need to keep track of heights:
  + In every **root** node, we store the **negative value of the height of the tree - 1** (to make sure -0 doesn’t happen)
  + **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** |



* + 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** |

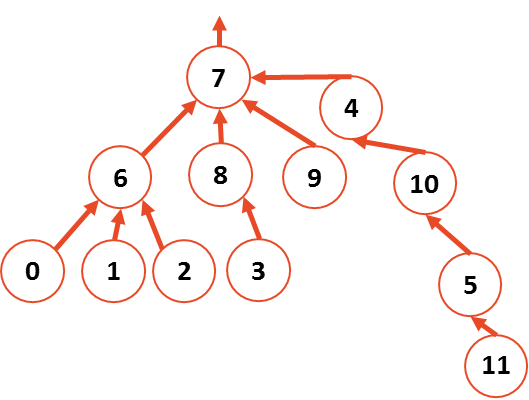
#### 

* **Union by Size** (*Minimize the number of nodes that increase in height*):
  + **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** | **-4** | **10** | **7** | **-8** | **7** | **7** | **4** | **5** |

* + The results after union:

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



* + Union by size keeps the average case average, but worsen the worst case (node 11 gets height increased by one)
* Both guarantee the height of the tree to be **O(log n)**