Disjoint Sets (Chapter 21.4)

Tarjan (1975)

Background

- Disjoint sets is a collection of sets where each set is initialized with a single unique item across all sets.

- Because the items are unique, the sets have no intersection among themselves and are therefore disjoint.

Data structure

```
public class DisjointSets : IDisjointSets
{
    private int[] set;      // If set[i] = j < 0 then set i has size |j|
                            // (i.e. set i exists with the given size)
                            // If set[i] = j > 0 then set i points to set j
                            // (i.e. set i no longer exists)
    private int numItems;   // Number of items
)
```

Primary methods

- public bool Union (int S1, int S2)
  - takes the union of sets S1 and S2
  - uses union-by-size which determines the resultant set
- public int Find (int x)
  - returns the set that x belongs to
  - uses path compression

Constructor

- Basic strategy
  - Set each array position to -1
  - The negative sign indicates that the set exists, and the value indicates the size of the set
  - Item i is assumed to be in set i

Initially

Set i contains item i
e.g. Item 4 is in set 4

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

All sets 0 to 7 exist and have a size of 1

```java
public DisjointSets(int numItems)
{
     int i;

     this.numItems = numItems;
     set = new int[numItems];
     for (i = 0; i < numItems; i++)
          set[i] = -1;         // Each set has an initial size of 1
                               // Assumption: item i belongs to set i
}
```

Time complexity

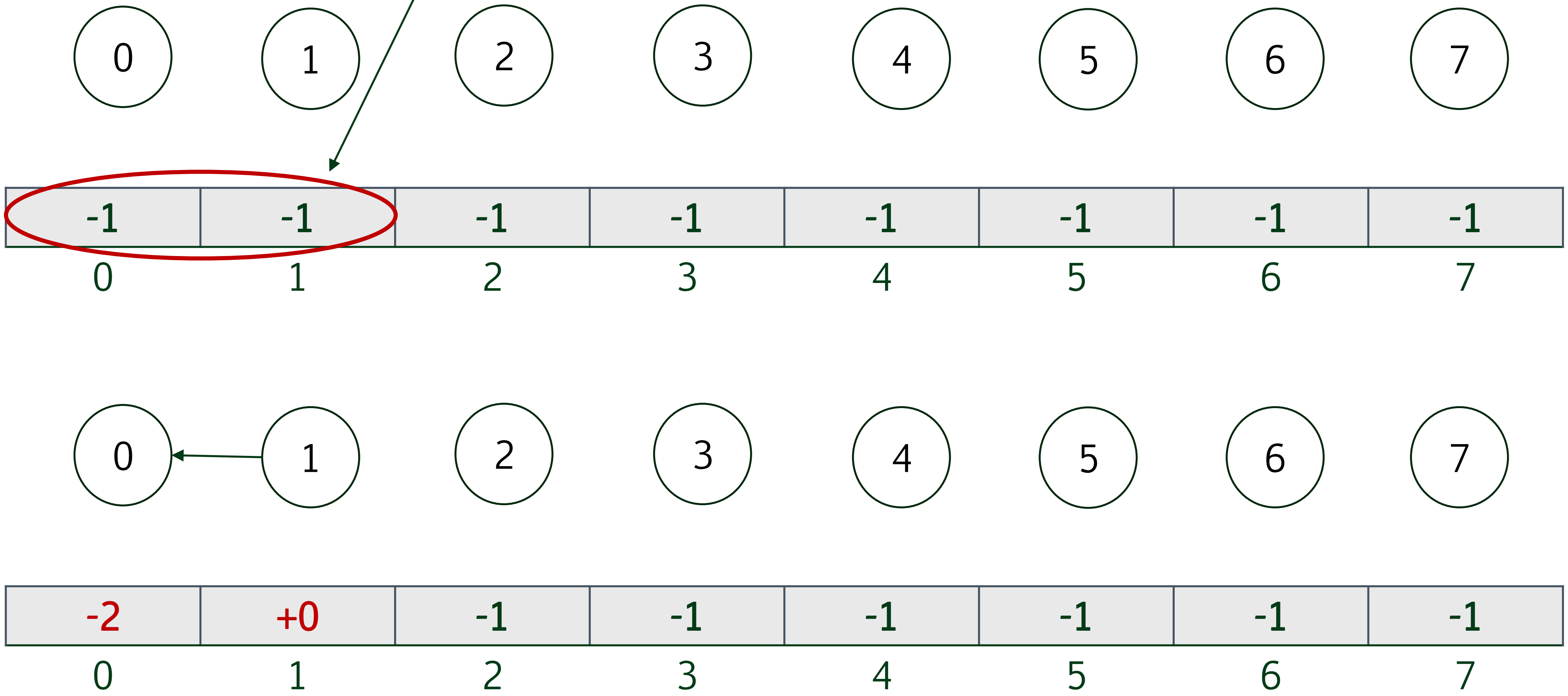
- To initialize the array to -1 takes O(n) time

Union

- Basic strategy
  - Check that both sets i and j exist i.e., set[i] < 0 and set[j] < 0
  - Merge the set with the lesser size into the set with the greater size i.e., union-by-size
  - If the two sets have the same size, merge either one into the other

Union(0,1)

Both sets exist and have a size of 1



Union(2,3)

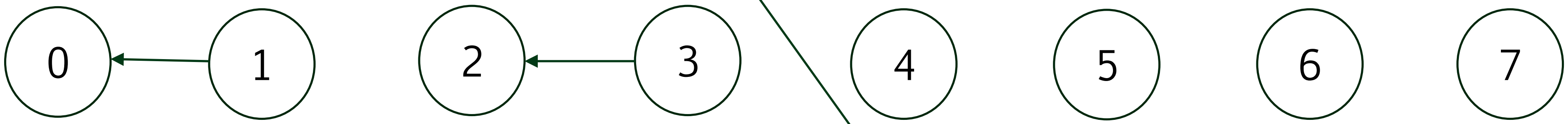
Both sets exist and have a size of 1

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

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

Union(5,4)

Both sets exist and have a size of 1



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

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

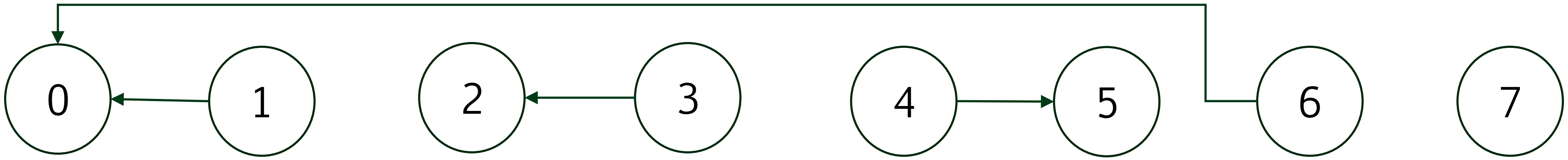
Union(2,1)

Set 1 does NOT exist; return false

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

Union(0,6) Both sets exist. Set 0 has greater size => Merge set 6 into set 0

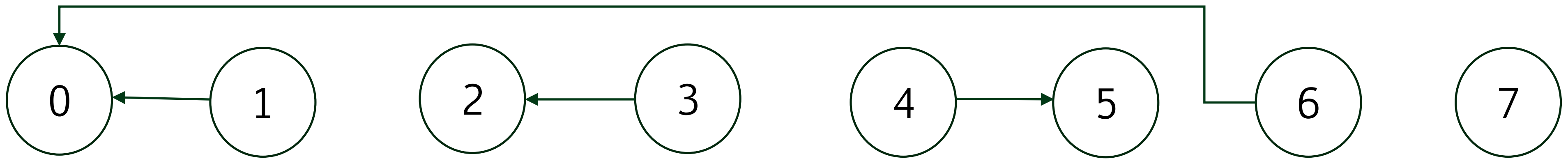
-2	+0	-2	+2	+5	-2	-1	-1
0	1	2	3	4	5	6	7



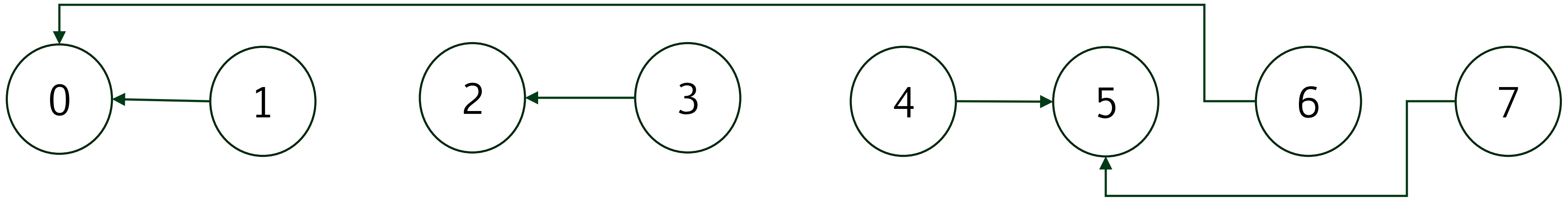
-3	+0	-2	+2	+5	-2	+0	-1
0	1	2	3	4	5	6	7

Union(5,7)

Both sets exist. Set 5 has greater size => Merge set 7 into set 5



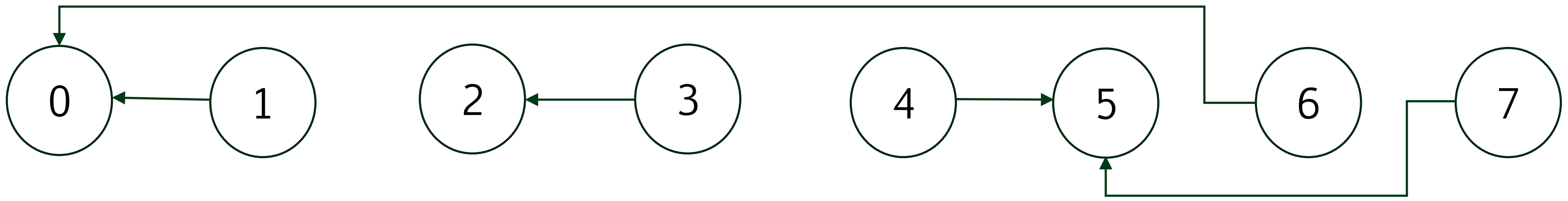
-3	+0	-2	+2	+5	-2	+0	-1
0	1	2	3	4	5	6	7



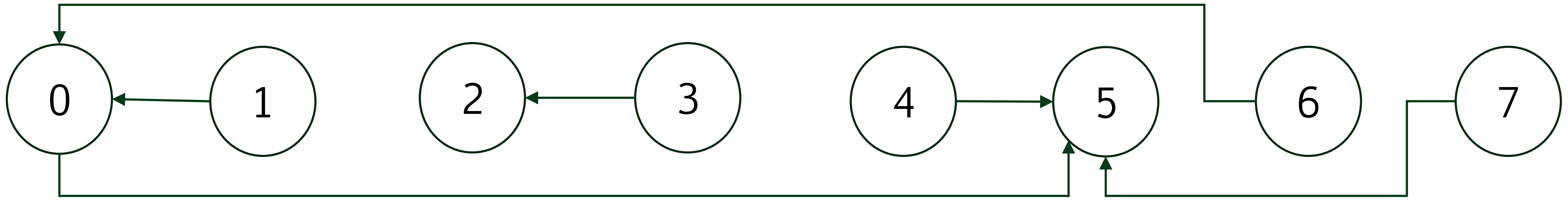
-3	+0	-2	+2	+5	-3	+0	+5
0	1	2	3	4	5	6	7

Union(5,0)

Both sets exist and have a size of 3



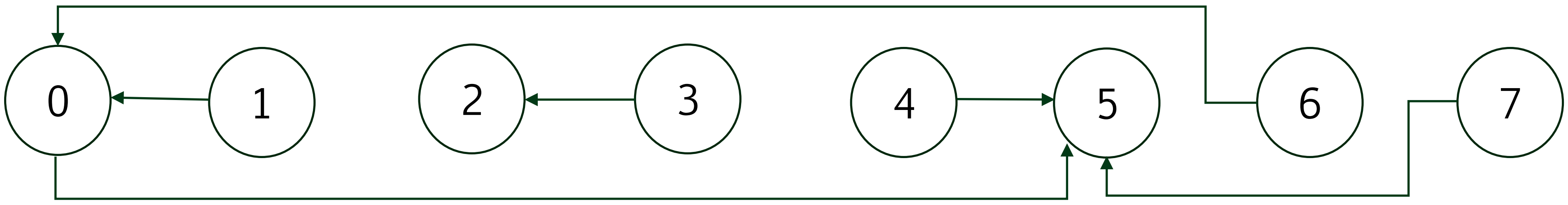
-3	+0	-2	+2	+5	-3	+0	+5
0	1	2	3	4	5	6	7



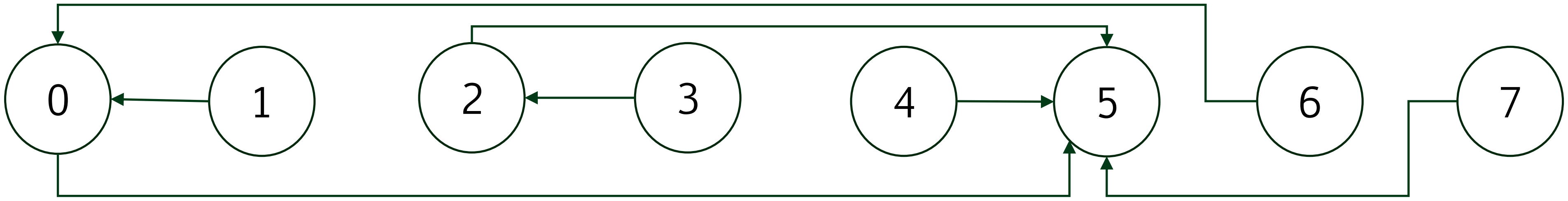
+5	+0	-2	+2	+5	-6	+0	+5
0	1	2	3	4	5	6	7

Union(2,5)

Both sets exist. Set 5 has greater size => Merge set 2 into set 5

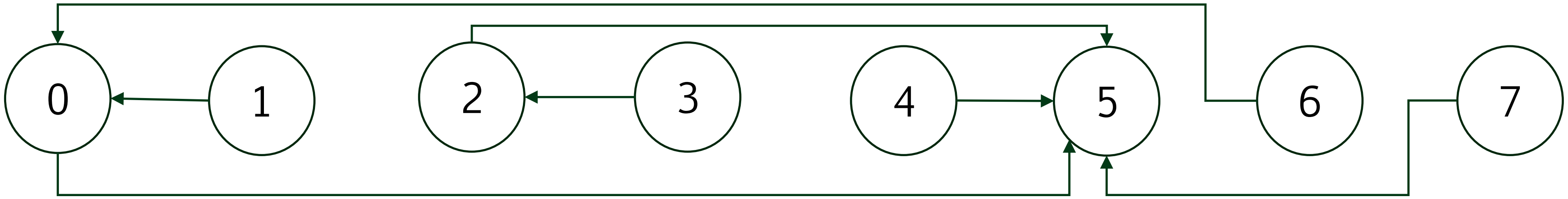


0	1	2	3	4	5	6	7
+5	+0	-2	+2	+5	-6	+0	+5



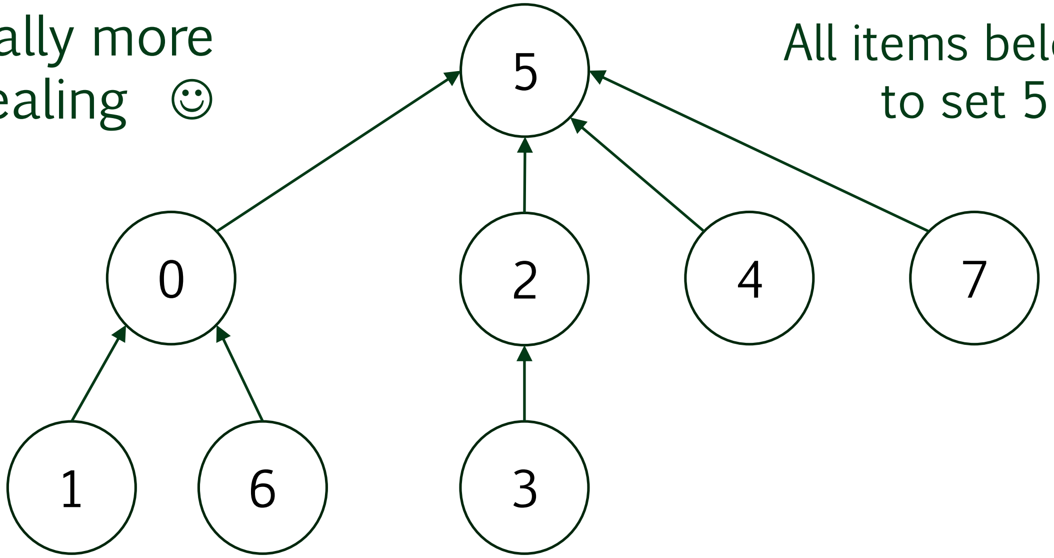
0	1	2	3	4	5	6	7
+5	+0	+5	+2	+5	-8	+0	+5

Final Set



+5	+0	+5	+2	+5	-8	+0	+5
0	1	2	3	4	5	6	7

Visually more appealing ☺



All items belong to set 5

```
public bool Union(int s1, int s2)
{
    // Union by size
    if (set[s1] < 0 && set[s2] < 0)  // Both sets exist
    {
        if (set[s2] < set[s1])       // If size of set s2 > size of set s1
        {                            // Then
            set[s2] += set[s1];      // Increase the size of s2 by the size of s1
            set[s1] = s2;            // Have set s1 point to set s2
        }                            // (i.e. s2 = s1 U s2)

        else                         // If the size of set s1 >= size of set s2
        {                            // Then
            set[s1] += set[s2];      // Increase the size of s1 by the size of s2
            set[s2] = s1;            // Have set s2 point to set s1
        }                            // (i.e. s1 = s1 U s2)
        return true;
    }
    else
        return false;
}
```

Time complexity

- The maximum number of Unions for n disjoint sets is n-1. Why?
- Each Union takes O(1) or constant time
- Therefore, n-1 Unions take O(n) time

Theorem 1

Using union-by-size, for every set i, $n_i \geq 2^{h_i}$

Notation:

Let n_i be the size of set i

Let h_i be the height of set i

Let n_i^* be the size of the resultant set i after a union-by-size

Let h_i^* be the height of the resultant set i after a union-by size

Proof by induction on the number of links k

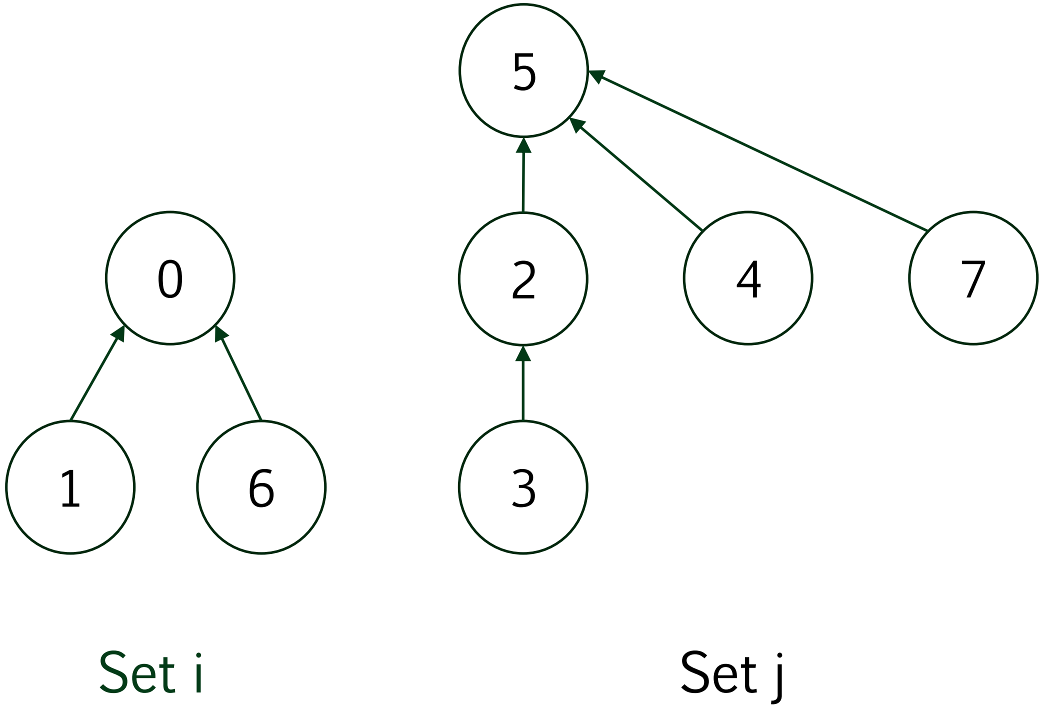
Basis of induction (k=0): With no links, every set contains one item and has a height of 0. The theorem holds for i=0 since $1 \geq 2^0$

Induction hypothesis: Assume the theorem is true after k links

Inductive step (k+1): Prove the theorem holds for an additional link

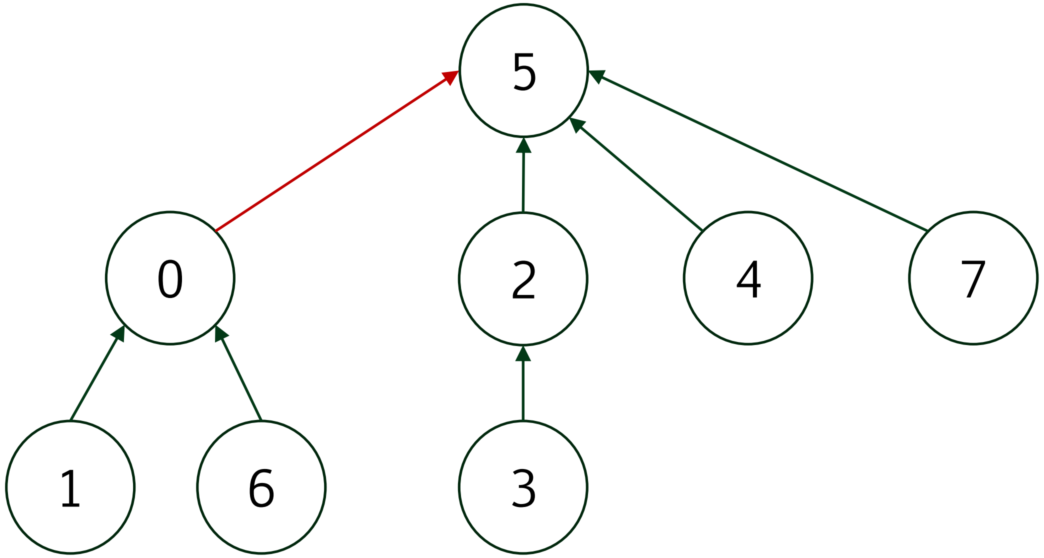
Case 1 ($h_i < h_j$)

Let set i be the smaller set and set j be the larger set.



Case 1 ($h_i < h_j$)

Let set i be the smaller set and set j be the larger set.



Set j

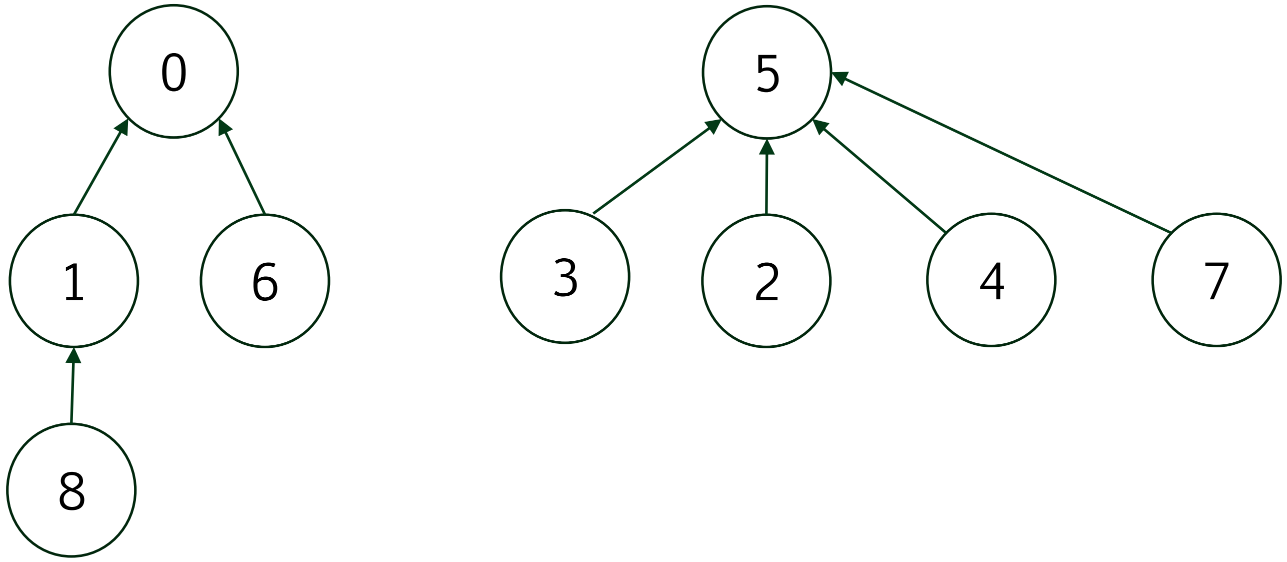
Case 1 ($h_i < h_j$)

Since $h_i < h_j$ the height of the resultant set j does not change.

$$n_j^* = n_i + n_j \geq \underbrace{n_j \geq 2^{h_j}}_{\text{Induction hypothesis}} = 2^{h_j^*}$$

Case 2 ($h_i \geq h_j$)

Let set i be the smaller set and set j be the larger set.

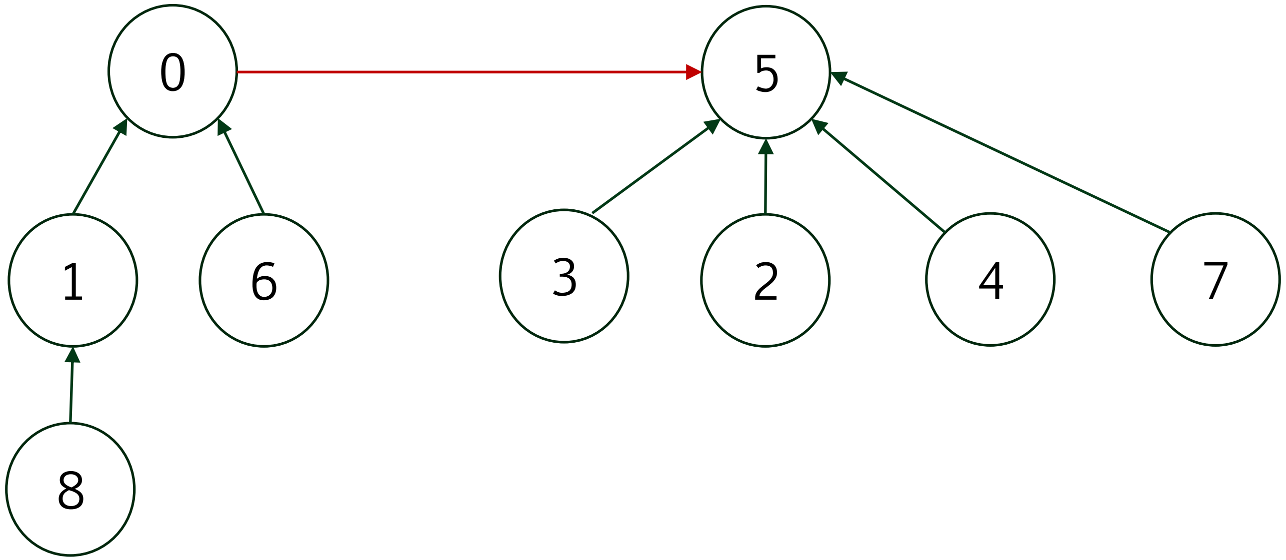


Set i

Set j

Case 2 ($h_i \geq h_j$)

Let set i be the smaller set and set j be the larger set.



Set j

Case 2 ($h_i \geq h_j$)

Since $h_i \geq h_j$ the height of the resultant set j is $h_i + 1$

$$n_j^* = n_i + n_j \geq \underbrace{2n_i \geq 2x2^{h_i}}_{\text{Induction hypothesis}} = 2^{h_i+1} = 2^{h_j^*}$$

Theorem 2

Using union-by-size, the height of the final resultant set is $O(\log n)$.

Proof:

The resultant set has n items. From Theorem 1, $n \geq 2^h$ which implies:

$$h \leq \log_2 n$$

Therefore, h is $O(\log n)$.

Exercises

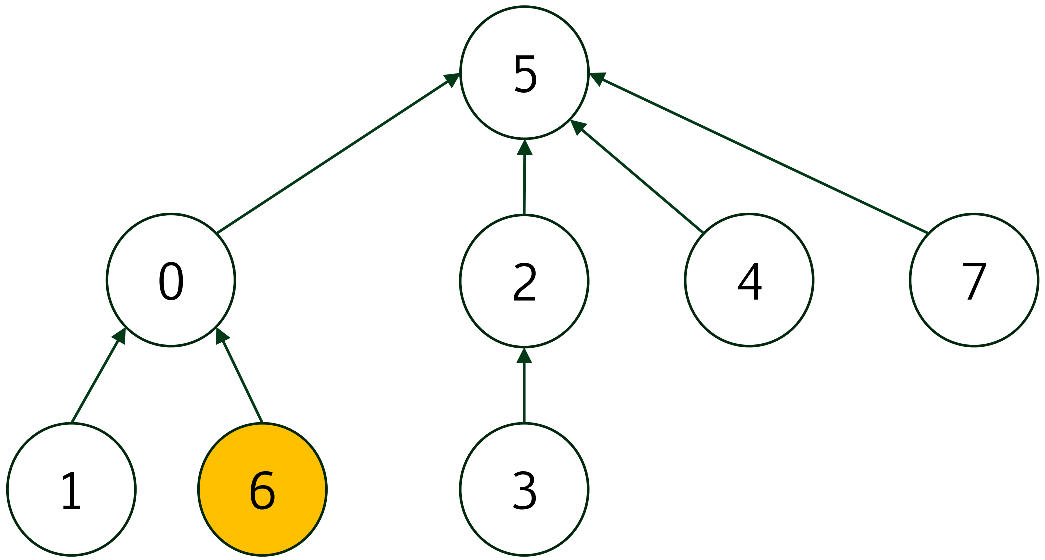
- Show that if union-by-rank is not used for n-1 unions, the final resultant set could have a height of n-1
- Carry out a series of seven Unions on 8 disjoint sets that yields a single set with a height of 3
- Redraw using the 2-D tree representation the resultant sets after each of the Union operations

Find

- Basic strategy
  - Recurse to the root of the set in which item belongs and return the root set
  - On the way back from the recursive calls, update each item along the path to point directly to the root set i.e., path compression

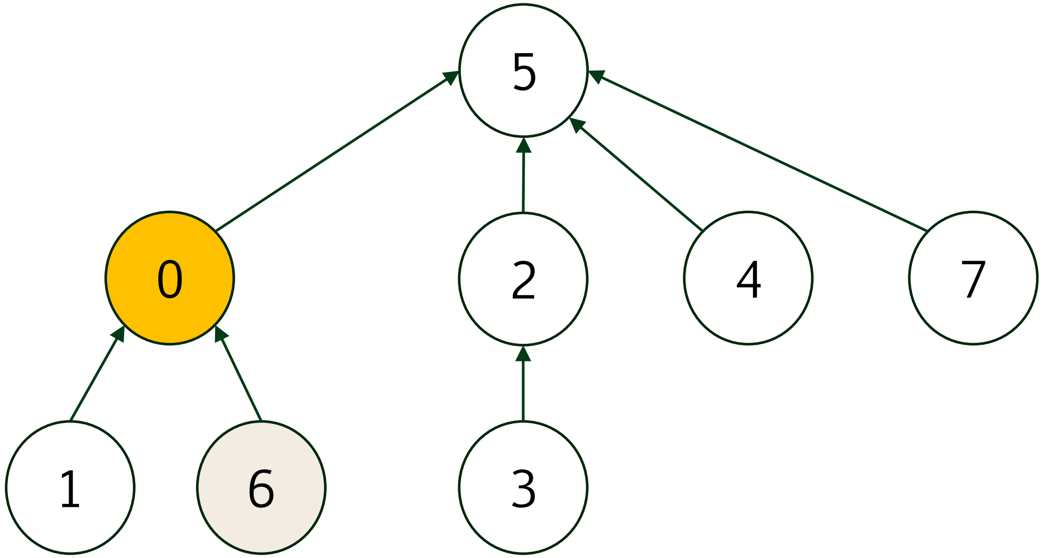
Find(6)

+5	+0	+5	+2	+5	-8	+0	+5
0	1	2	3	4	5	6	7

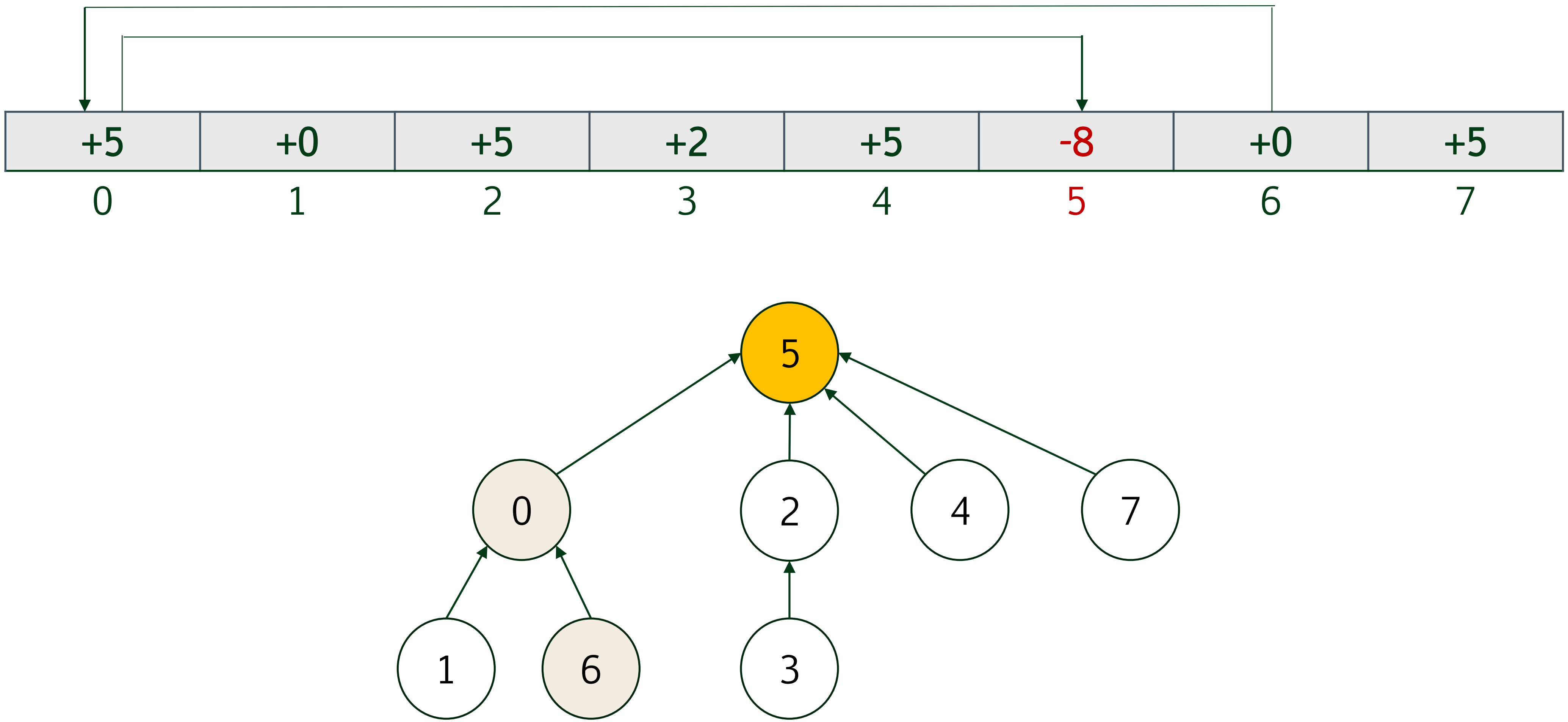


Find(6)

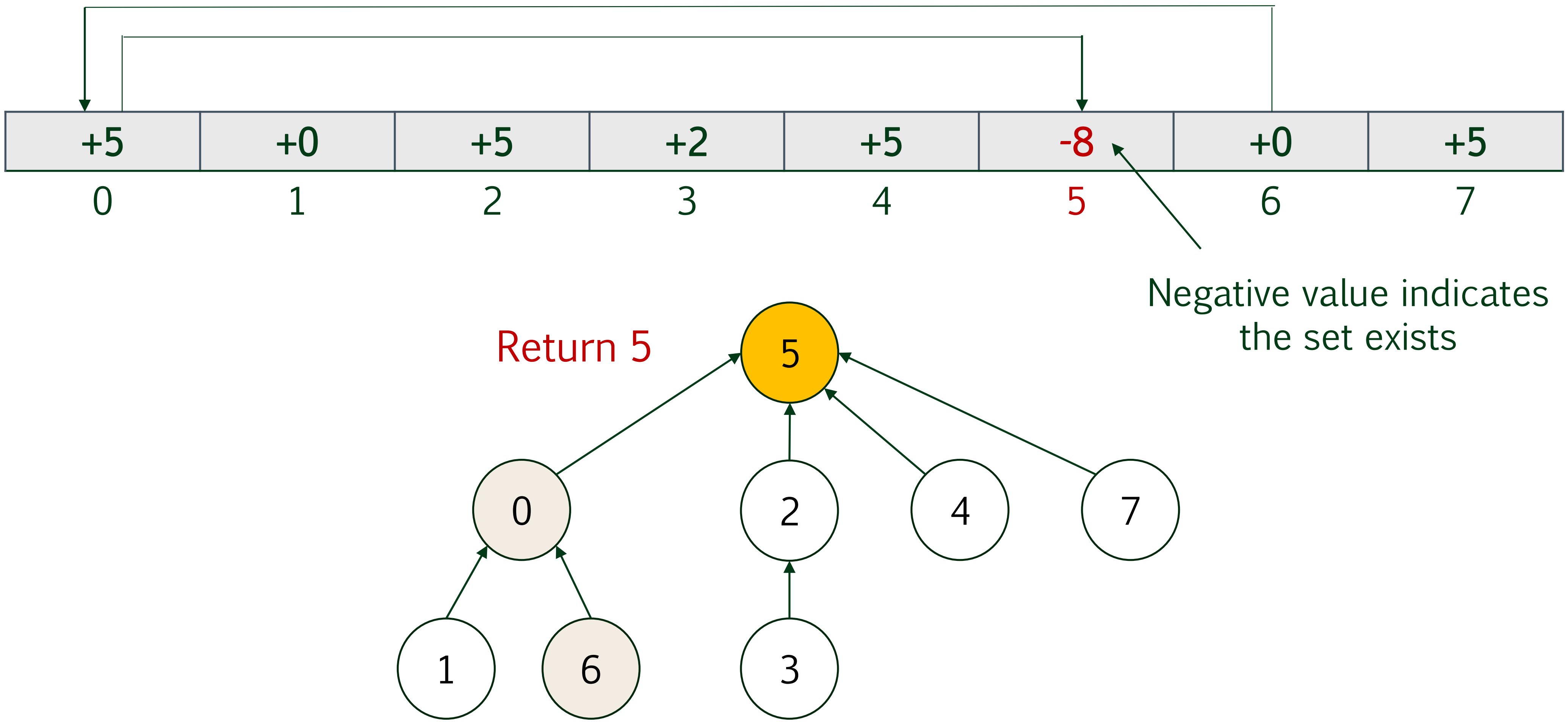
+5	+0	+5	+2	+5	-8	+0	+5
0	1	2	3	4	5	6	7



Find(6)



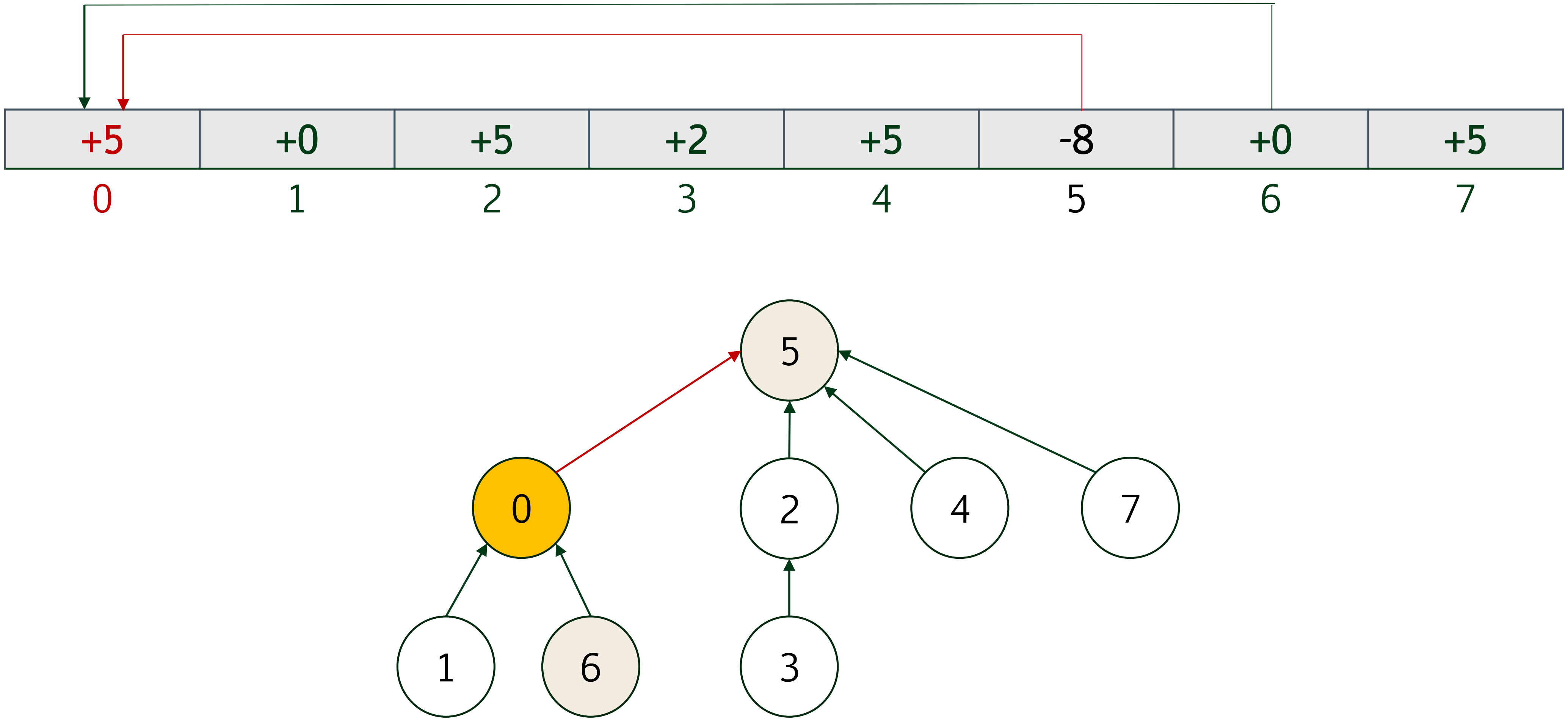
Find(6)



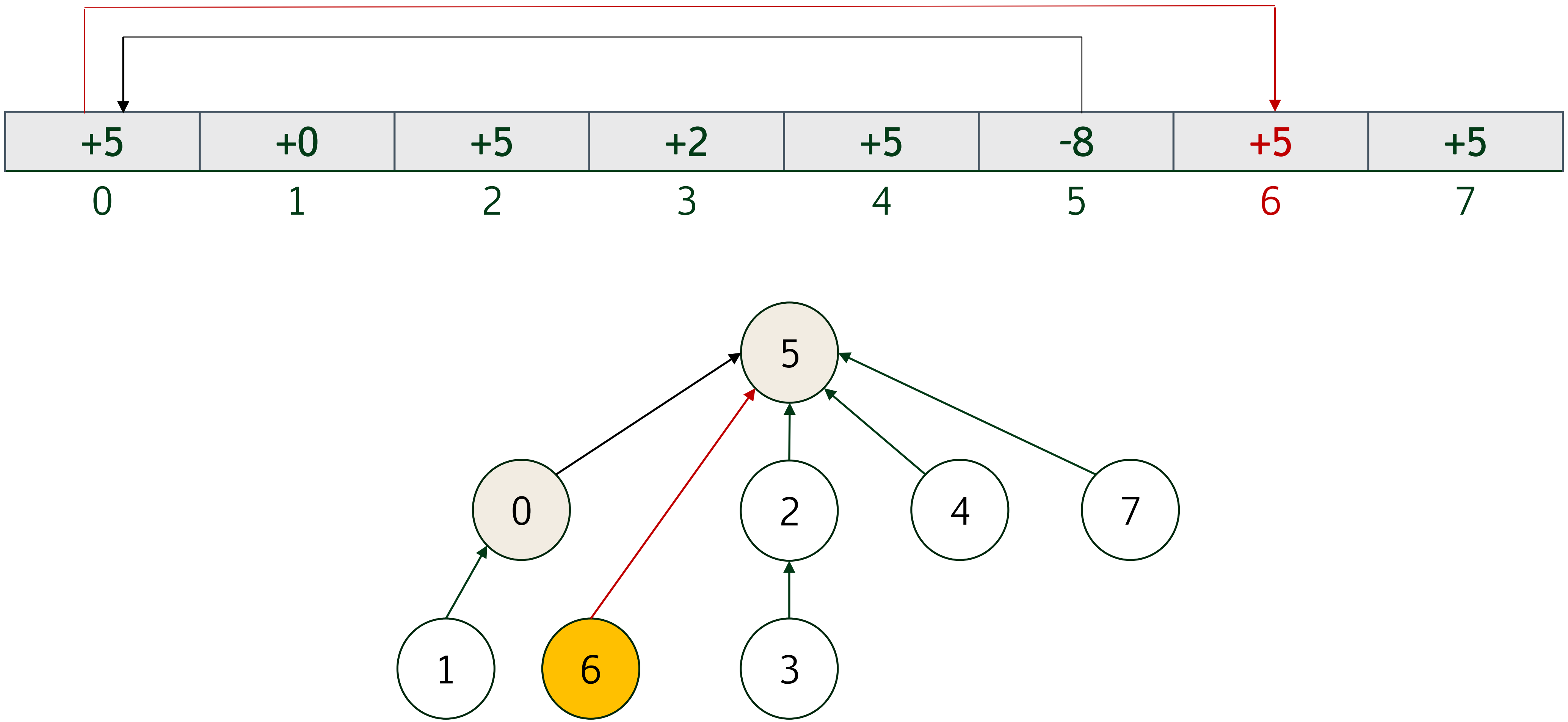
+5 +0 +5 +2 +5 -8 +0 +5
0 1 2 3 4 5 6 7
Negative value indicates the set exists
Return 5
5
0 2 4 7
1 6 3

- If the tree structure doesn’t change, each Find operation takes O(log n) time i.e., the height of the tree.
- But we also know that every node (item) along the path to the root belongs to set 5! Therefore, set each node along the path (and only nodes along the path) to point directly to 5.
- Next time, Find(6) will only take one step.

Find(6)



Find(6)



- But how do we backtrack along the same path from the root?
- The beauty of recursion

Luc Devroye
McGill University

```
public int Find(int item)
{
    if (item < 0 || item >= numItems) // Item is not in the range 0..n-1
        return -1;
    else
        if (set[item] < 0)                // set[item] exists
            return item;
        else
            // Path Compression
            // Recurse to the root set
            // And then update all sets along the path to point to the root
            return set[item] = Find(set[item]);
  }
```

```
public int Find(int item)
{
    if (item < 0 || item >= numItems) // Item is not in the range 0..n-1
        return -1;
    else
        if (set[item] < 0)                // set[item] exists
            return item;
        else
            // Path Compression
            // Recurse to the root set
            // And then update all sets along the path to point to the root
            return set[item] = Find(set[item]);
      }
```

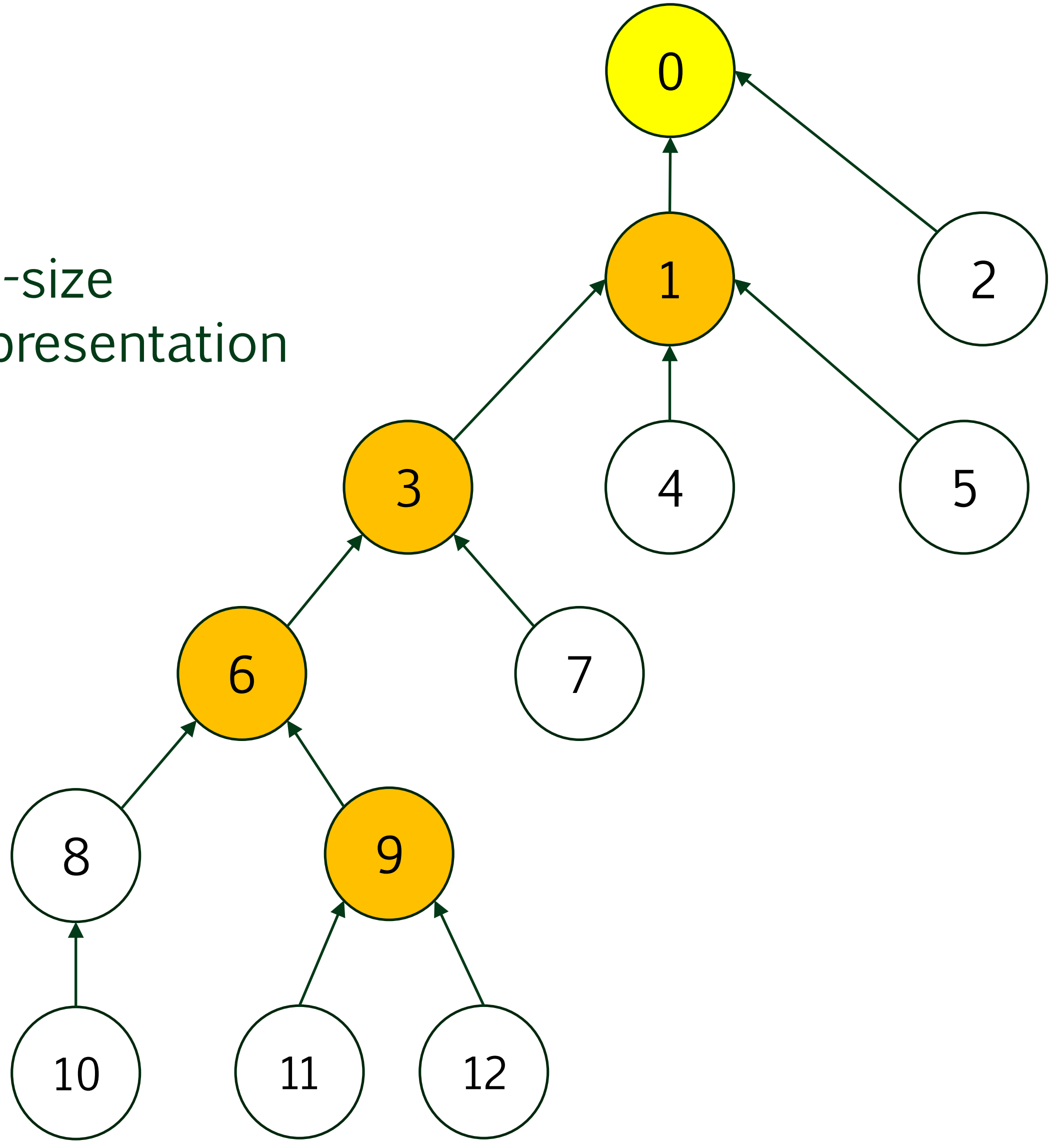
Terminating condition

```
public int Find(int item)
{
    if (set[item] < 0)
        return item;
    else
        return set[item] = Find(set[item]);
}
```

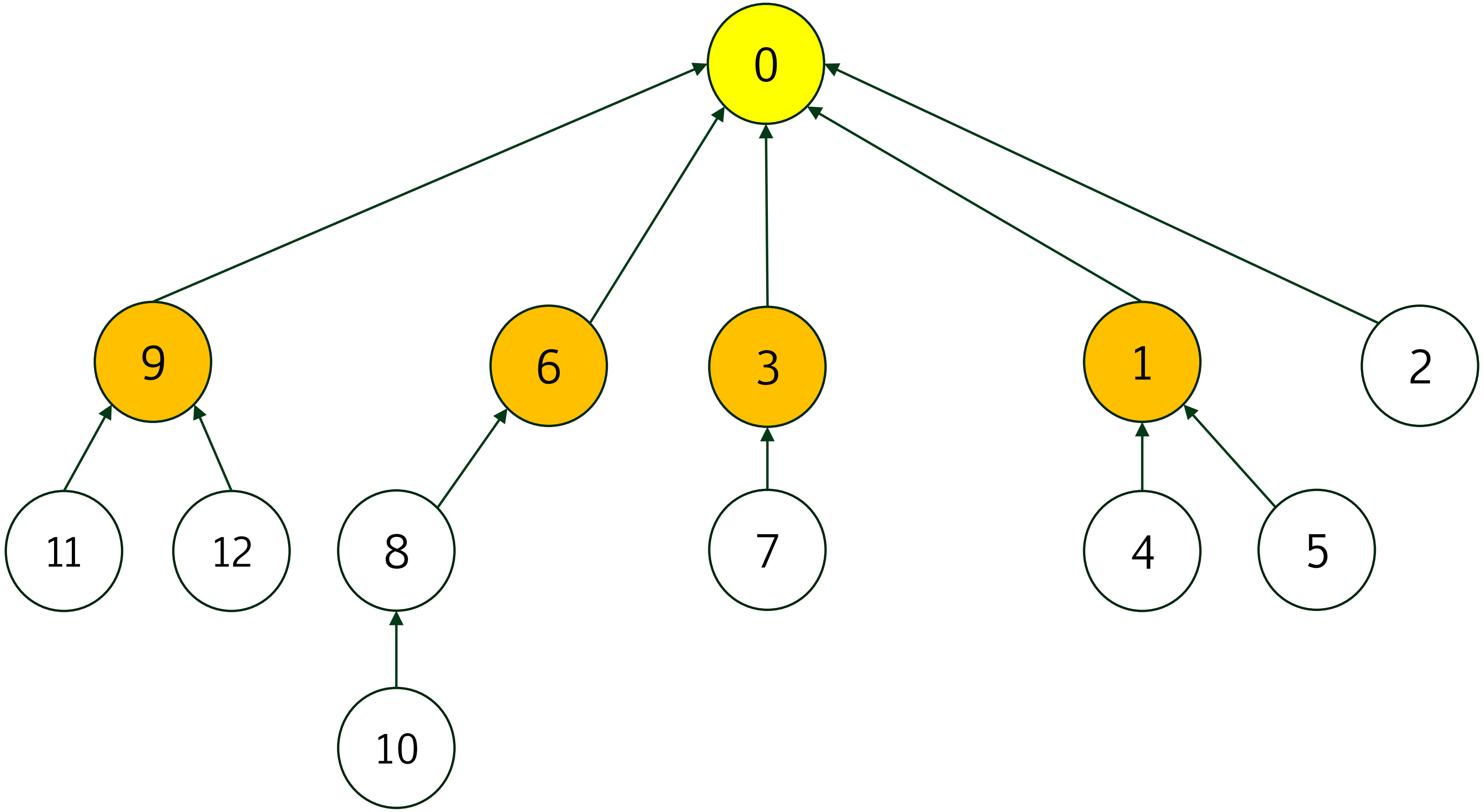
Terminating condition

Find(9)

Question: Was union-by-size used to construct this representation of disjoints sets?



Find(9)



Time complexity

Using union-by-size plus path compression, any sequence of $m \geq n$ Find and $n-1$ Union operations is:

$$O(m\ \alpha(m,n))$$

where $\alpha(m,n)$ is the functional inverse of Ackermann's function. But $\alpha(m,n)$ grows sooooo slowly that the average time to complete any sequence of m operations is virtually $O(m)$ which averages out to $O(1)$ per operation.

Exercises

- Review the reference on the next page and learn about union-by-rank.
  - The rank is a proxy for what measurement?
  - When does the rank of a set increase?
- Describe a set representation where one Find operation reduces the height of the set from n-1 to 1. Note that union-by-size would not have been used.

Useful reference

Kevin Wayne, Princeton University

https://www.cs.princeton.edu/courses/archive/spring13/cos423/lectures/UnionFind.pdf