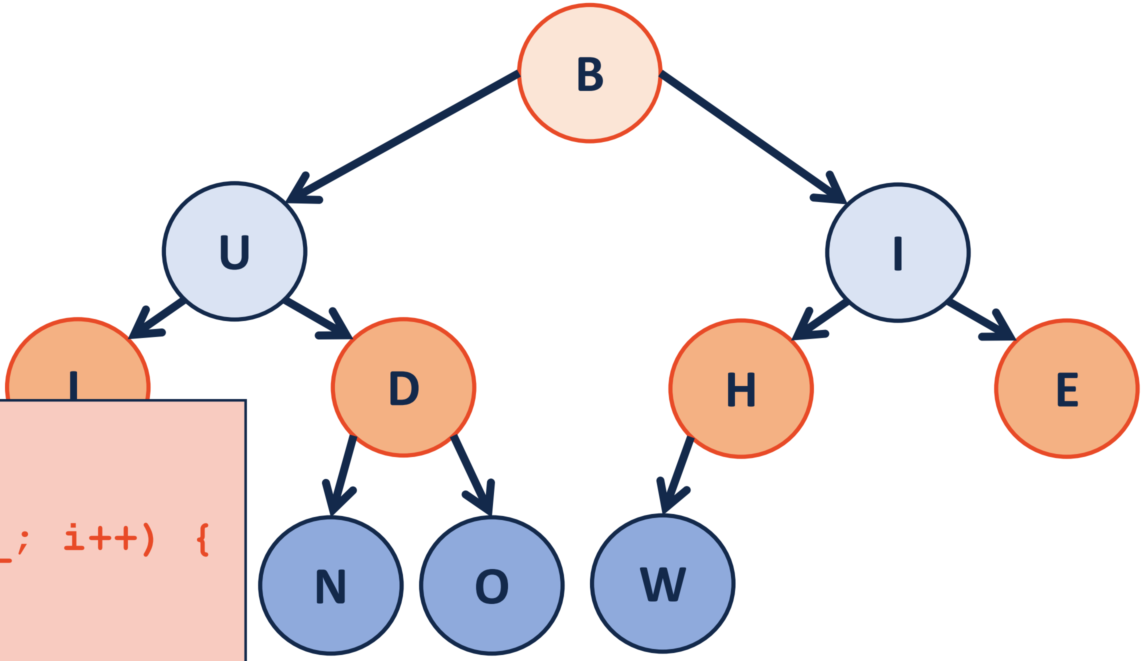
CS 225

Data Structures

April 2 – Disjoint Sets Intro

Wade Fagen-Ulmschneider

buildHeap



1. Sort the array – it's a heap!

2.

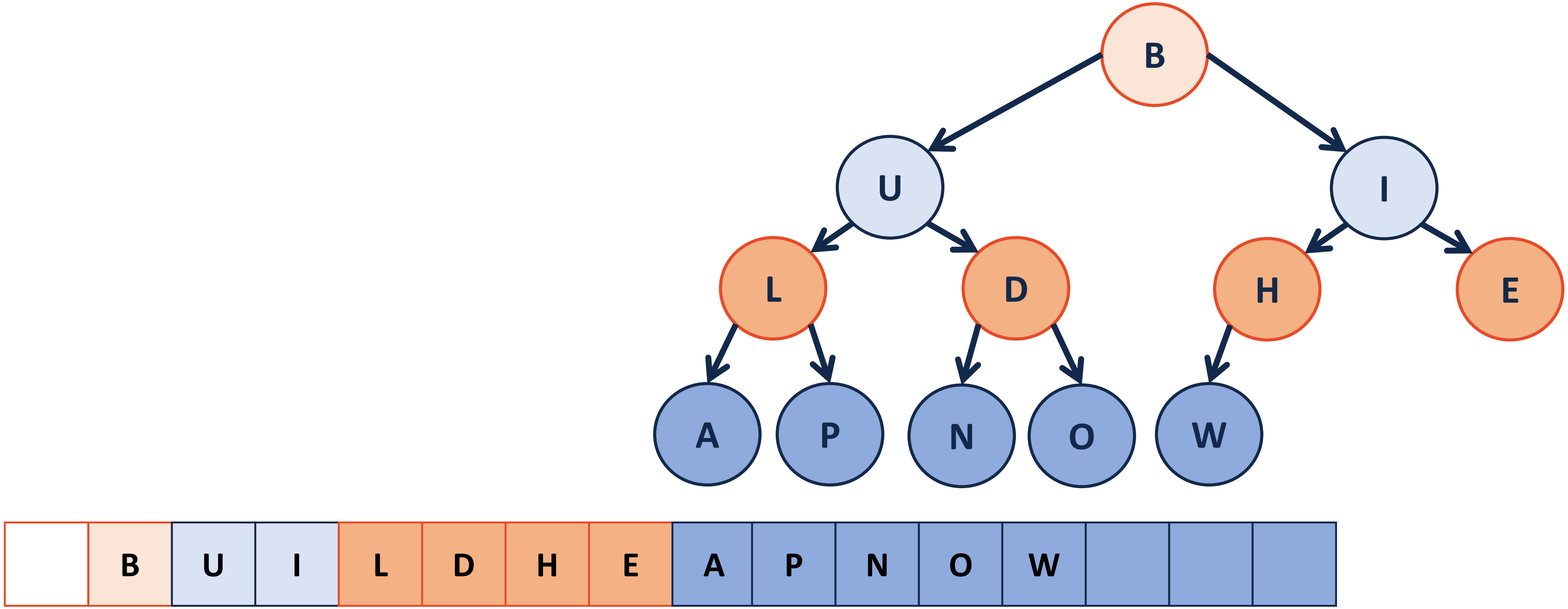
```
template <class T>
void Heap<T>::buildHeap() {
  for (unsigned i = 2; i <= size_; i++) {
    heapifyUp(i);
  }
}
```

3.

```
template <class T>
void Heap<T>::buildHeap() {
  for (unsigned i = parent(size); i > 0; i--) {
    heapifyDown(i);
  }
}
```

	B	U	I	L	D	H	E	A	P	N	O	W			

buildHeap - heapifyDown



Proving buildHeap Running Time

Theorem: The running time of buildHeap on array of size **n** is: ___________.

Strategy:

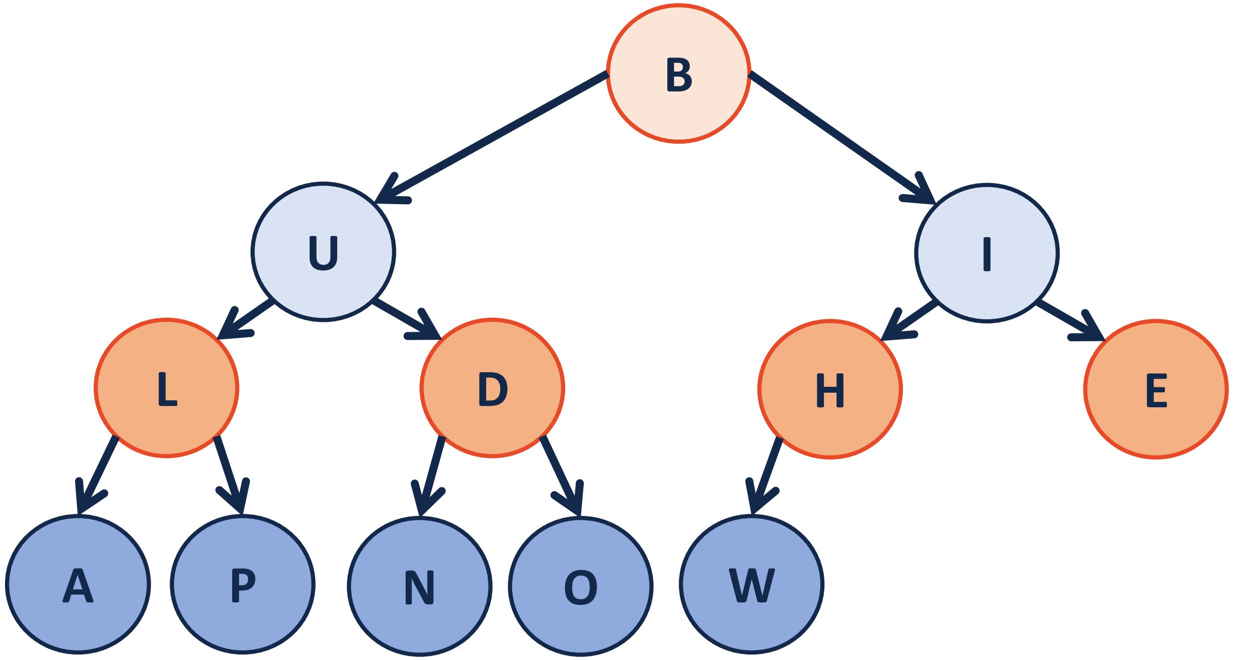
-

-

-

Proving buildHeap Running Time

S(h): Sum of the heights of all nodes in a complete tree of height **h**.



S(0) =

S(1) =

S(2) =

S(h) =

Proving buildHeap Running Time

Proof the recurrence:

Base Case:

General Case:

Proving buildHeap Running Time

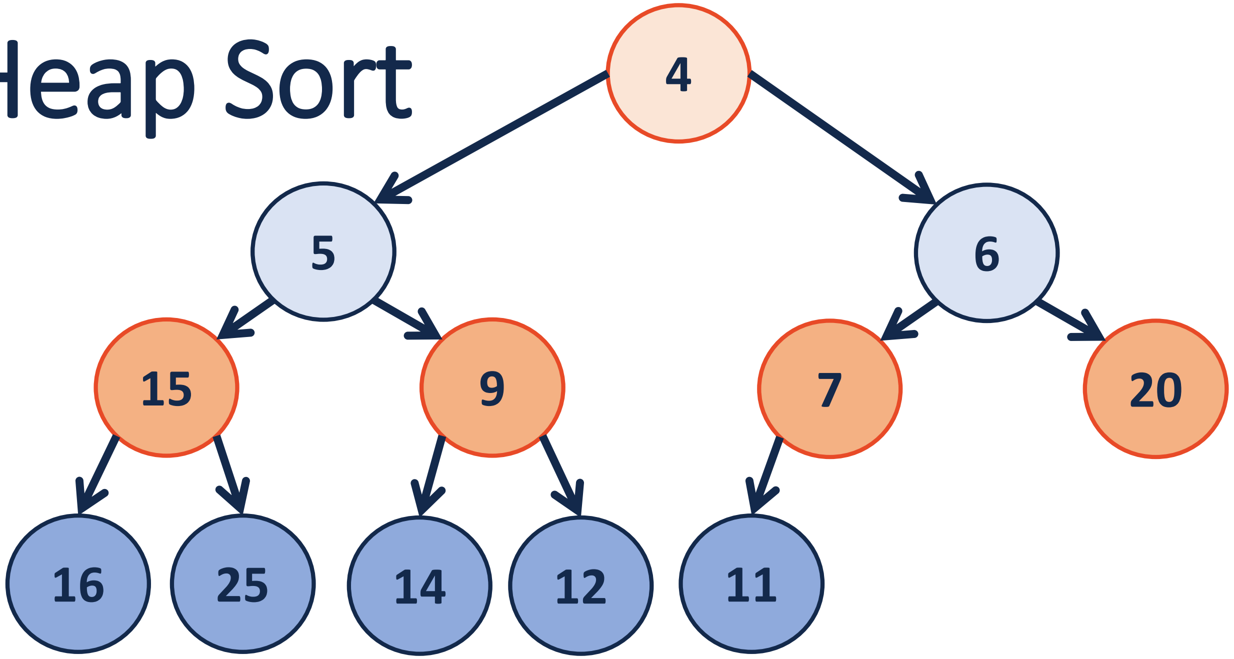
From S(h) to RunningTime(n):

S(h):

Since h ≤ lg(n):

RunningTime(n) ≤

Heap Sort



1.

2.

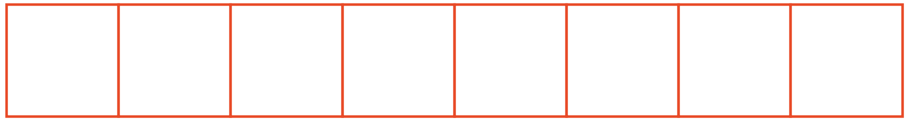
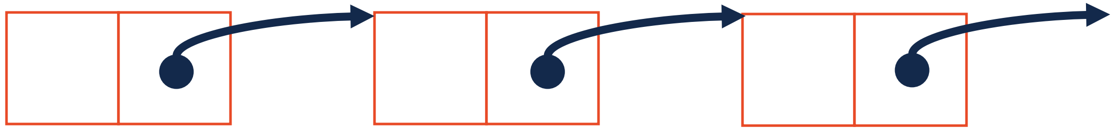
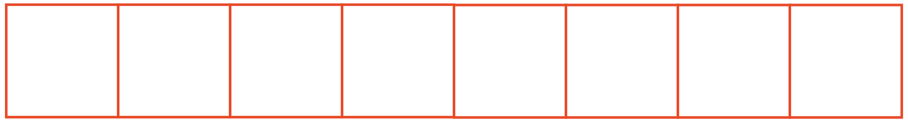
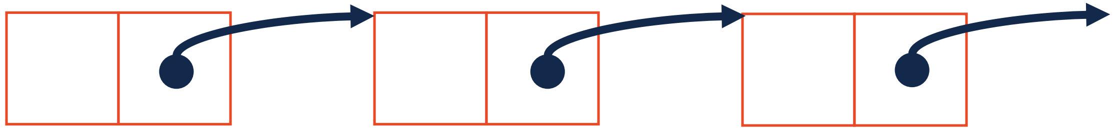
3.

	4	5	6	15	9	7	20	16	25	14	12	11			

Running Time?

Why do we care about another sort?

Priority Queue Implementation

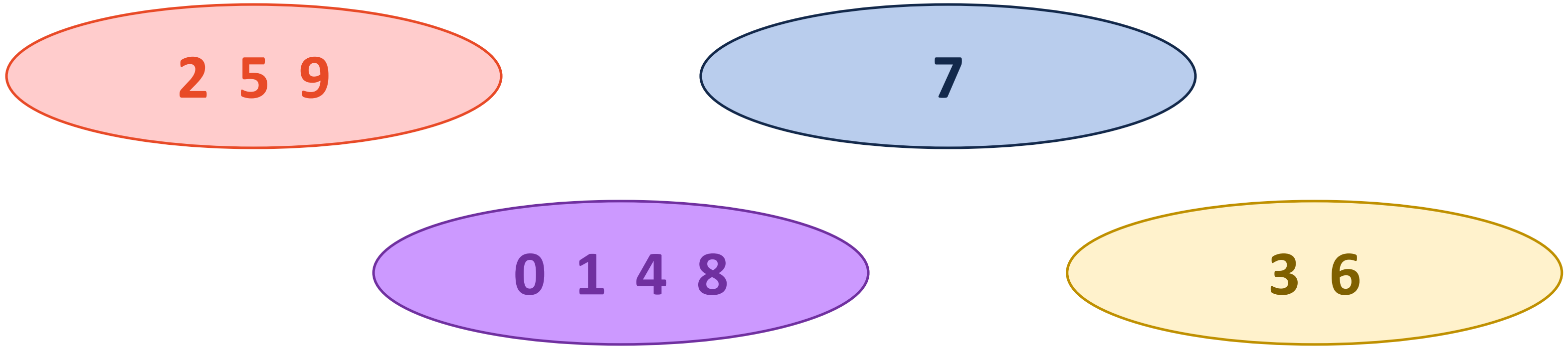
insert	removeMin	buildHeap	
O(1)[A]	O(n)	O(n lg(n))	
O(1)	O(n)	O(n lg(n))	unsorted 
O(n)	O(1)	O(n lg(n))	
O(n)	O(1)	O(n lg(n))	sorted 
O(lg(n))	O(lg(n))	O(n lg(n))	**AVL Tree**
O(lg(n))	O(lg(n))	O(n)	**Heap**

A(nother) throwback to CS 173…

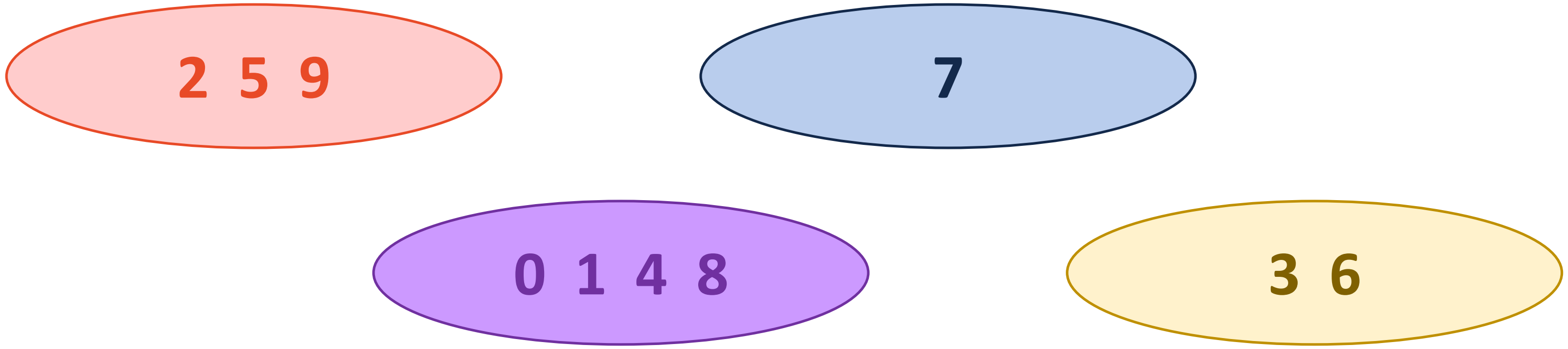
Let **R** be an equivalence relation on *us* where **(s, t)** $\in$ **R** if **s** and **t** have the same favorite among:

{ ____, ____, _____, ____, _____, }

Disjoint Sets

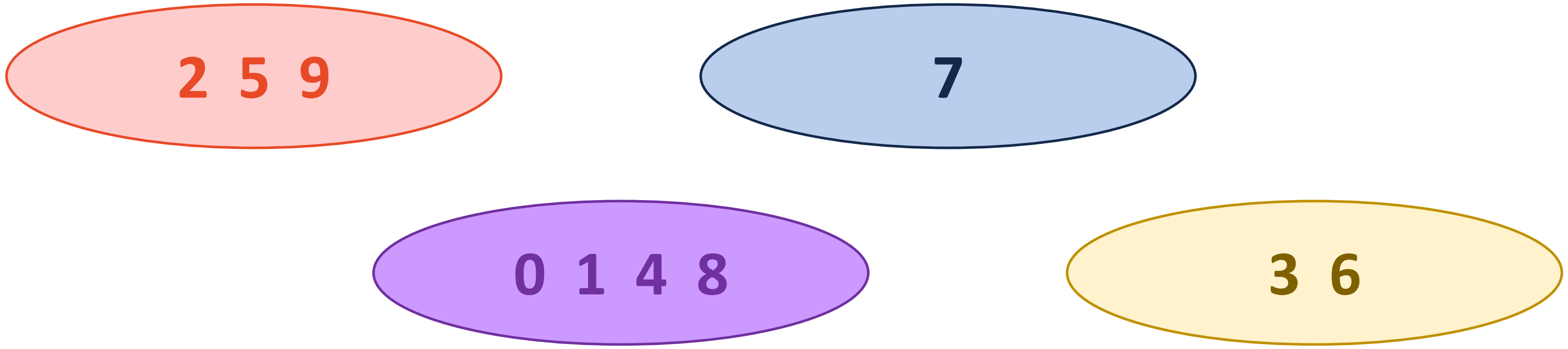


Disjoint Sets



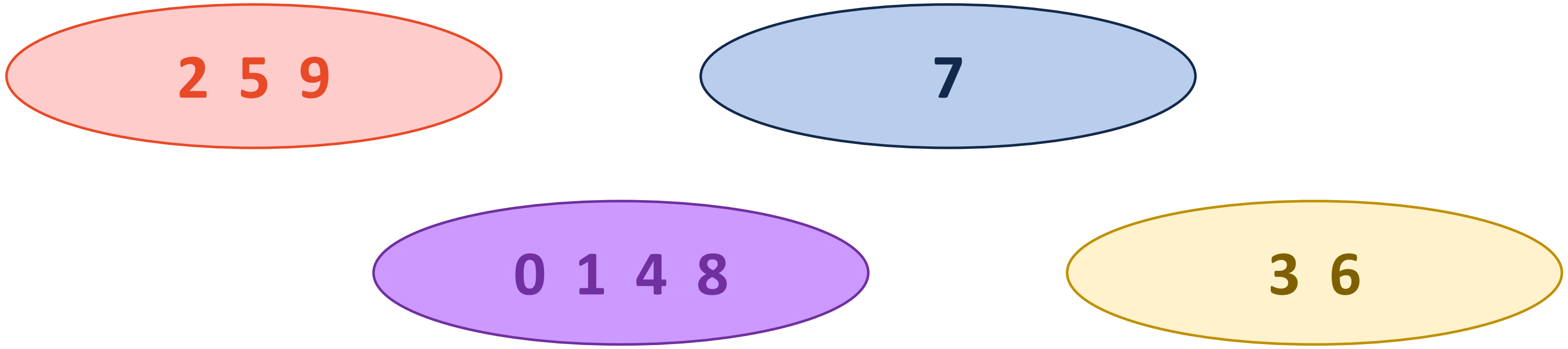
Operation: find(4)

Disjoint Sets



Operation: find(4) == find(8)

Disjoint Sets



Operation:

```
if ( find(2) != find(7) ) {
    union(   find(2), find(7)    );
}
```

Disjoint Sets ADT

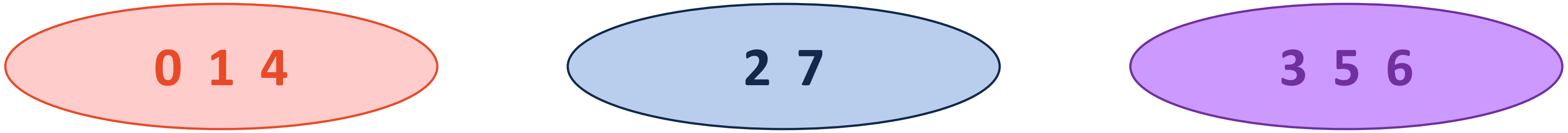
- Maintain a collection $S = \{s_0, s_1, \ldots s_k\}$

- Each set has a representative member.

- API:

```
void makeSet(const T & t);
void union(const T & k1, const T & k2);
T & find(const T & k);
```

Implementation #1



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

Find(k):

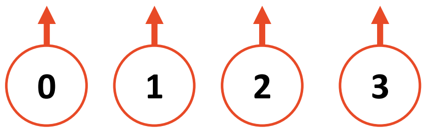
Union(k1, k2):

Implementation #2

- We will continue to use an array where the index is the key

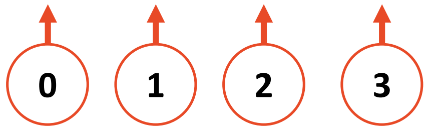
- The value of the array is:
  - **-1,** if we have found the representative element
  - **The index of the parent**, if we haven't found the rep. element

- We will call theses **UpTrees**:



0	1	2	3
-1	-1	-1	-1

UpTrees



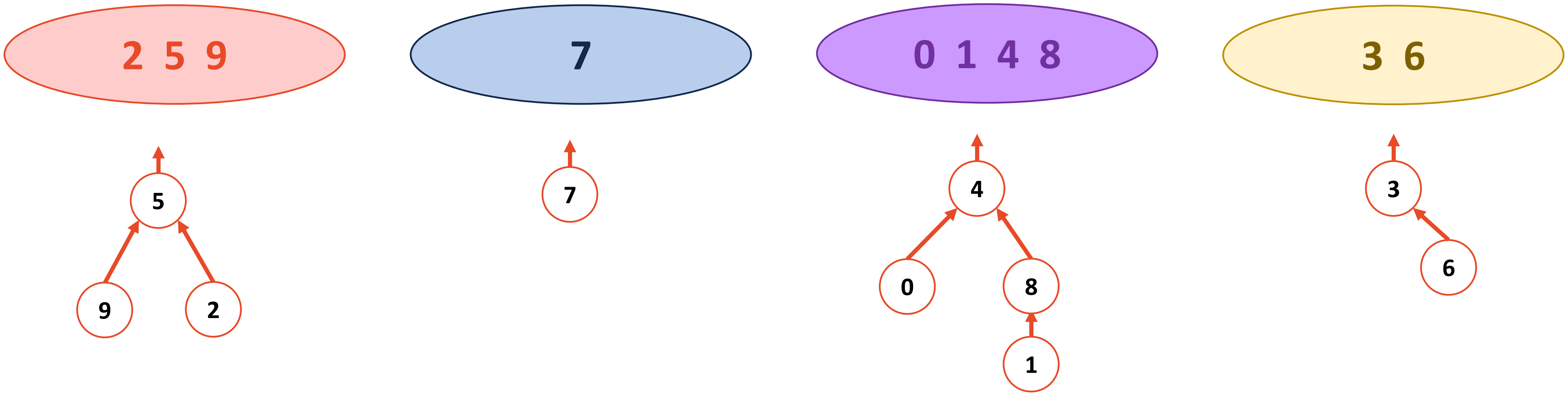
0	1	2	3
-1	-1	-1	-1

0	1	2	3

0	1	2	3

0	1	2	3

Disjoint Sets



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