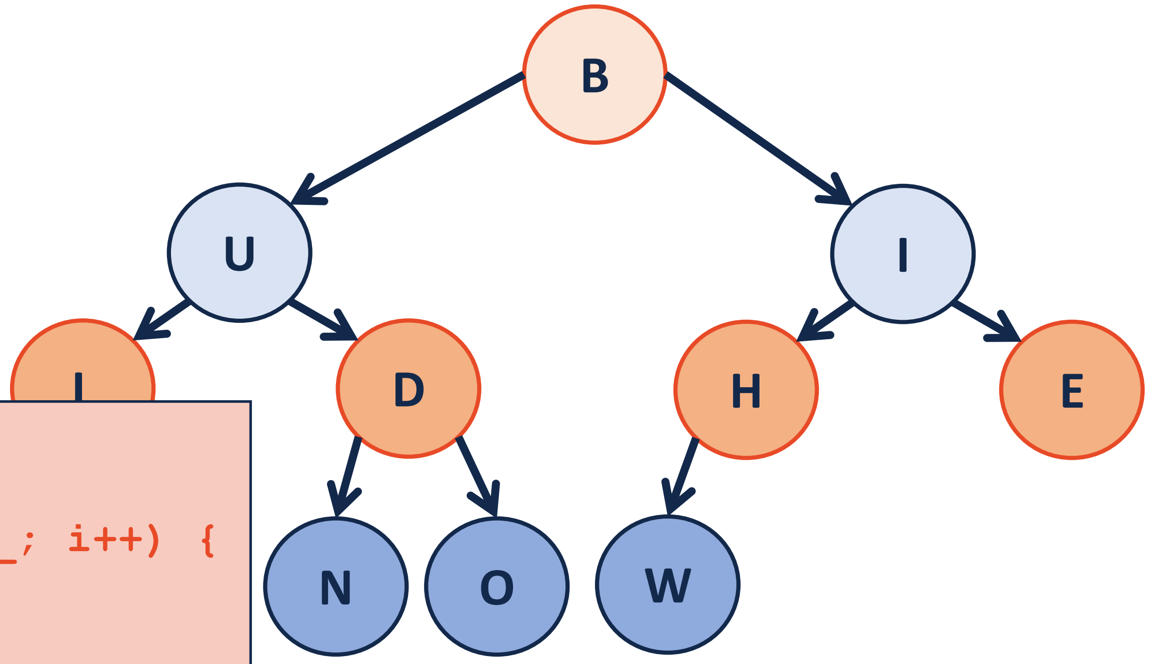
CS 225

Data Structures

November 5 – Heap Analysis and Disjoint Sets

Wade Fagen-Ulmschneider

buildHeap



1. Sort the array – it's a heap!

2.

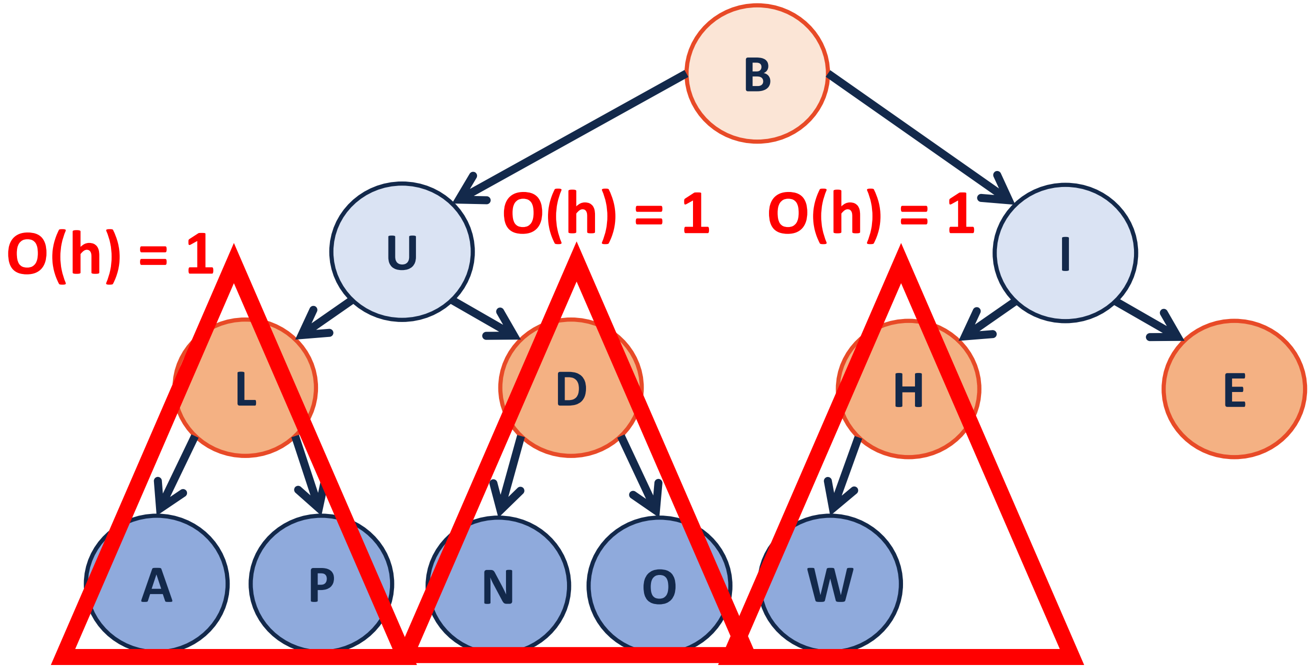
```
template <class T>
void Heap<T>::buildHeap() {
  for (unsigned i = 2; i <= size_; i++) {
    heapifyUp(i);
  }
}
```

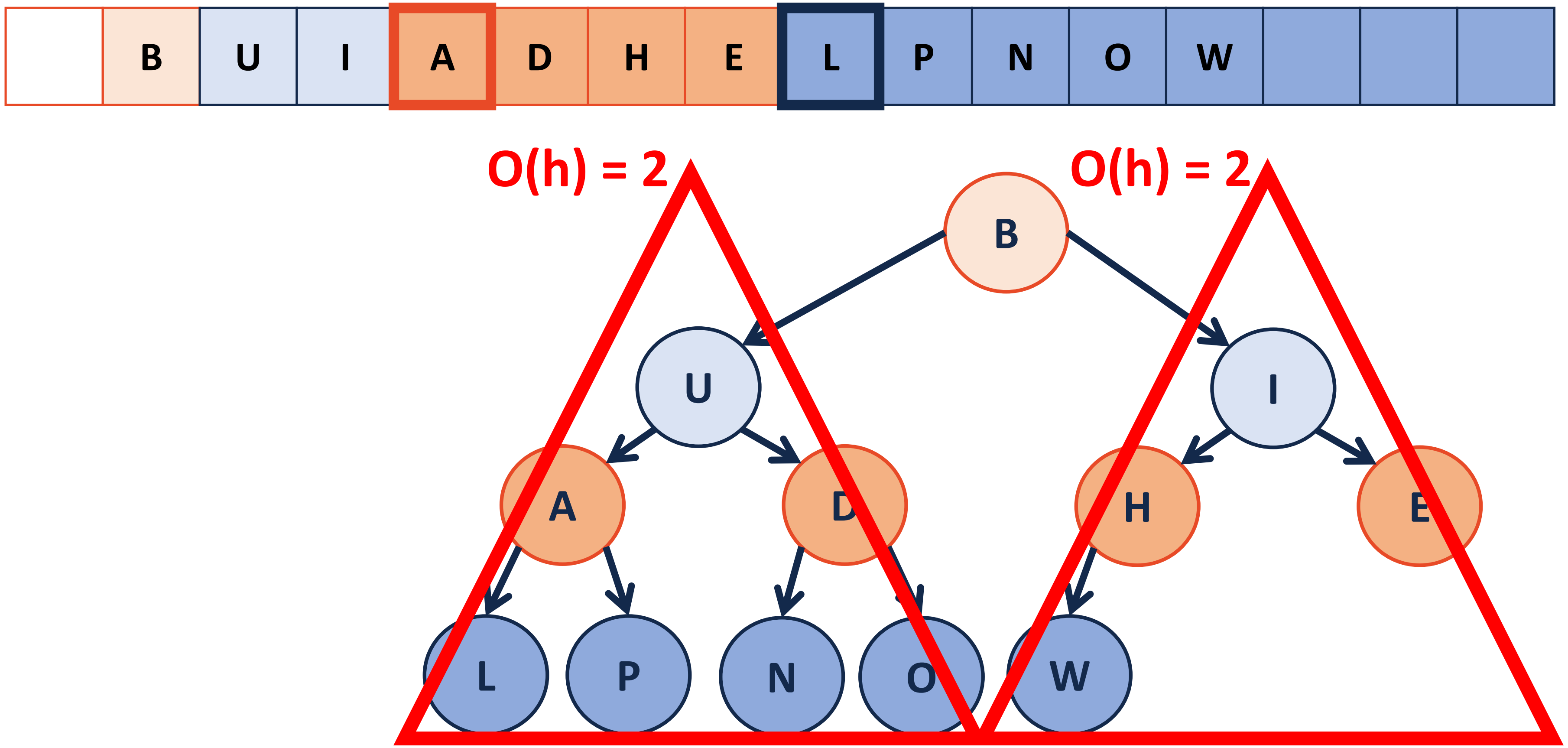
3.

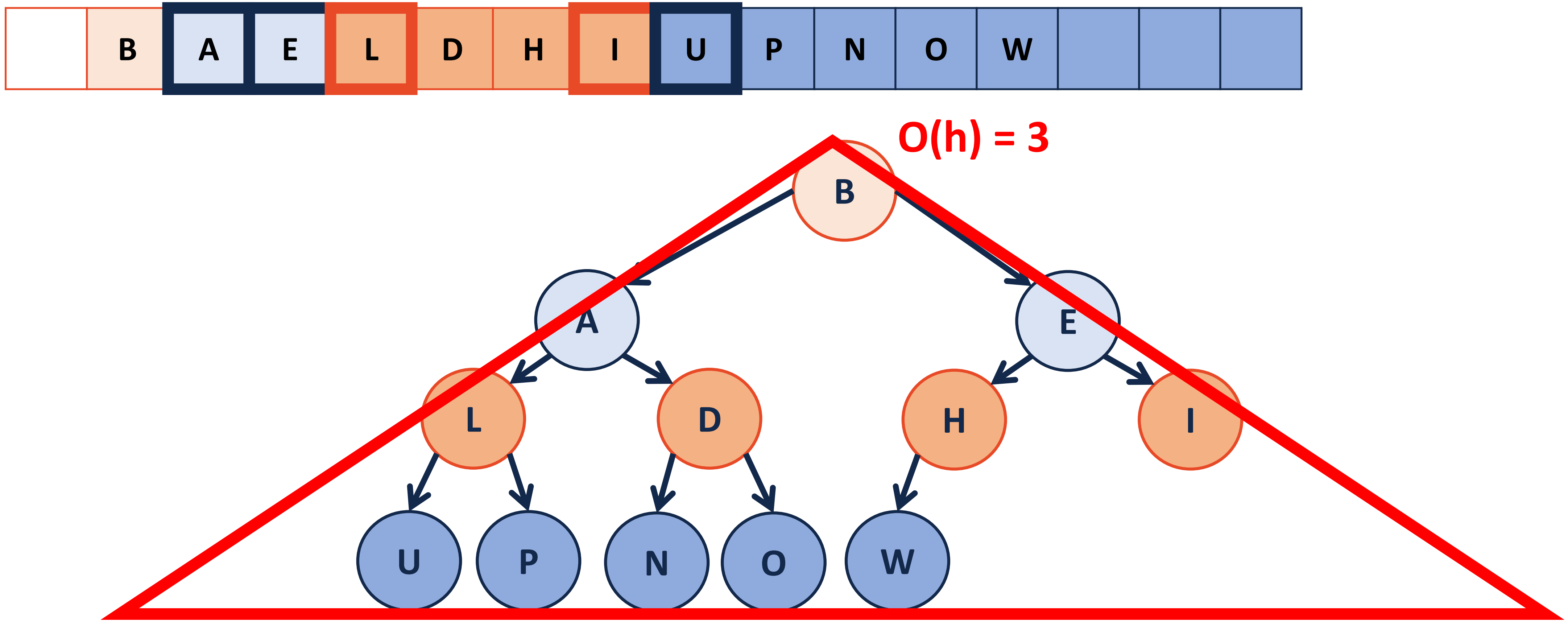
```
template <class T>
void Heap<T>::buildHeap() {
  for (unsigned i = parent(size); i > 0; i--) {
    heapifyDown(i);
  }
}
```

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

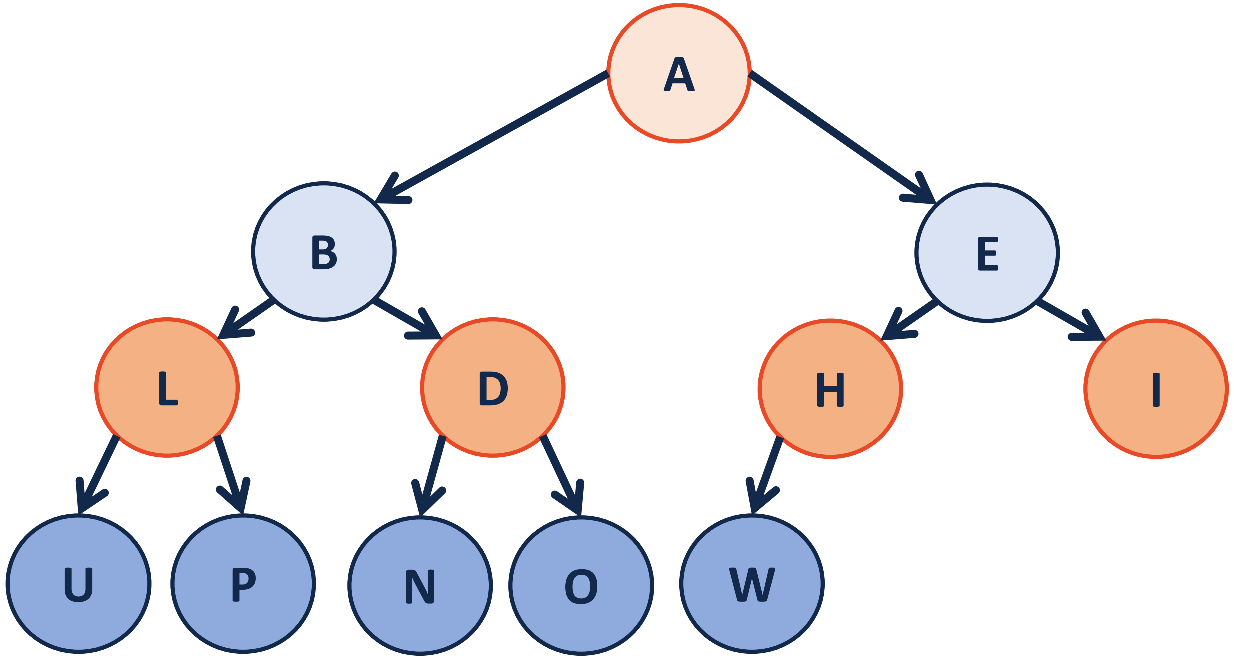
	B	U	I	L	D	H	E	A	P	N	O	W			




B
U
I
A
D
H
E
L
P
N
O
W
O(h) = 2
O(h) = 2


B A E L D H I U P N O W
O(h) = 3
B
A
E
L
D
H
I
U
P
N
O
W

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



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

Proving buildHeap Running Time

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

Strategy:

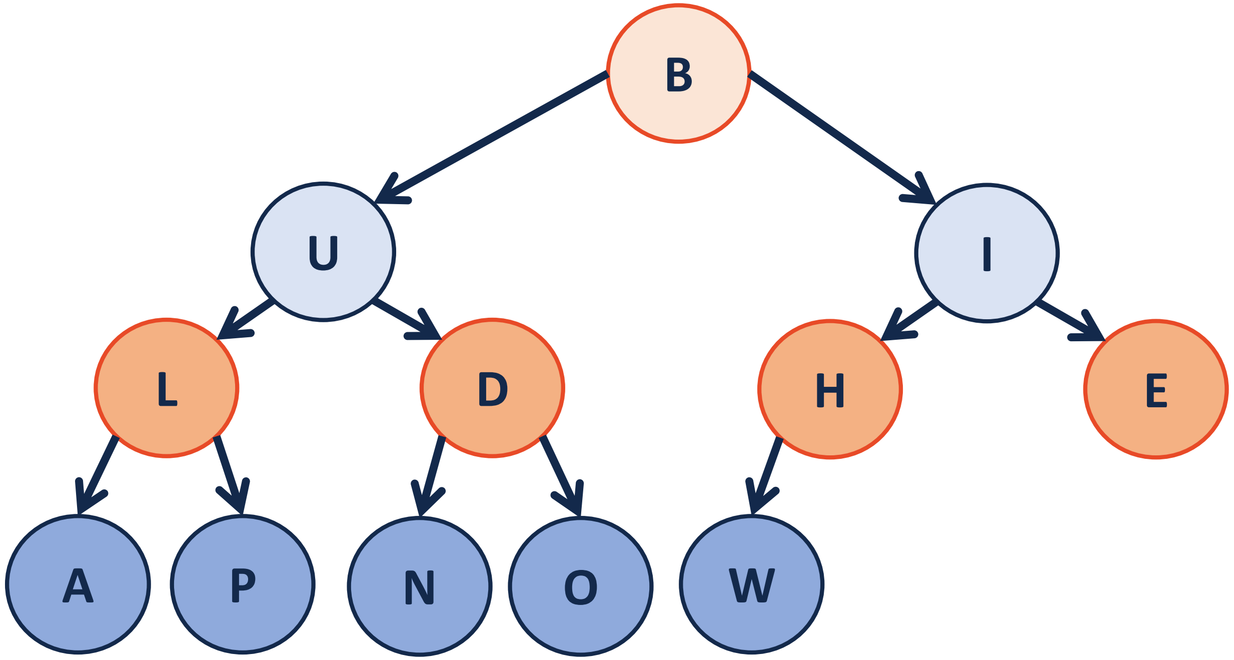
-

-

-

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(h) =

Proving buildHeap Running Time

Proof the recurrence:

Base Case:

General Case:

Proving buildHeap Running Time

From S(h) to RunningTime(n):

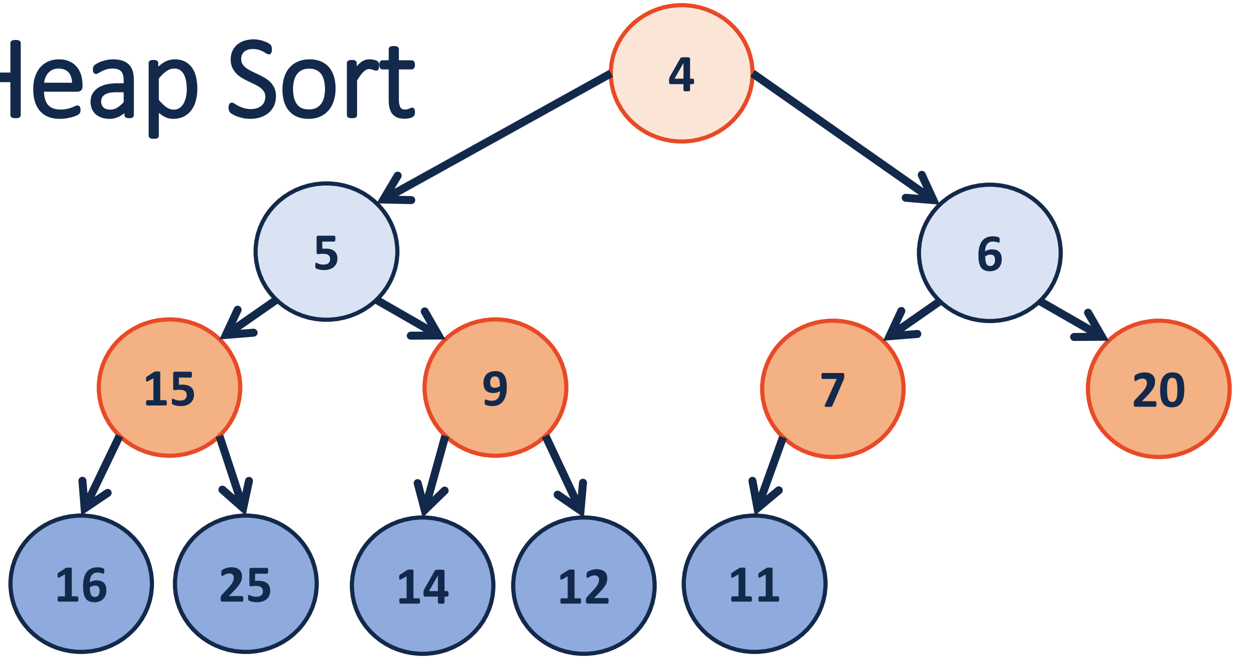
S(h):

Since h ≤ lg(n):

RunningTime(n) ≤

Mattox Monday

Heap Sort



1.

2.

3.

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

Running Time?

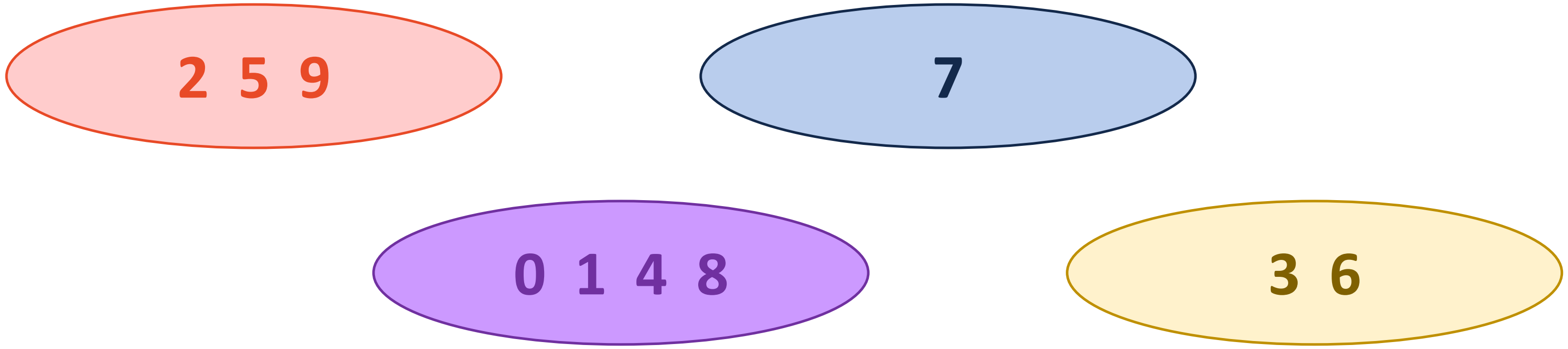
Why do we care about another sort?

A(nother) throwback to CS 173…

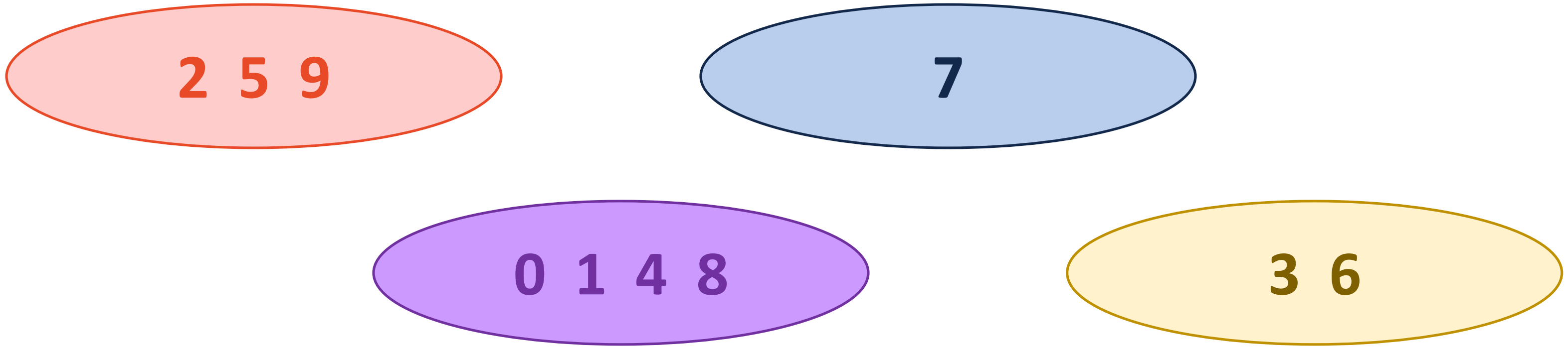
Let **R** be an equivalence relation on *us* where **(s, t) ∈ R** if **s** and **t** have the same favorite among:

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

Disjoint Sets

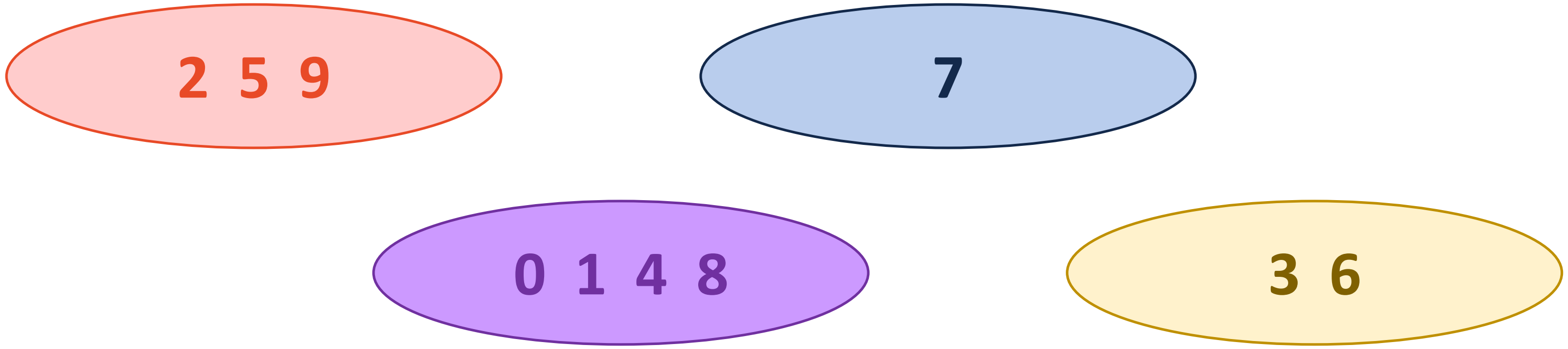


Disjoint Sets



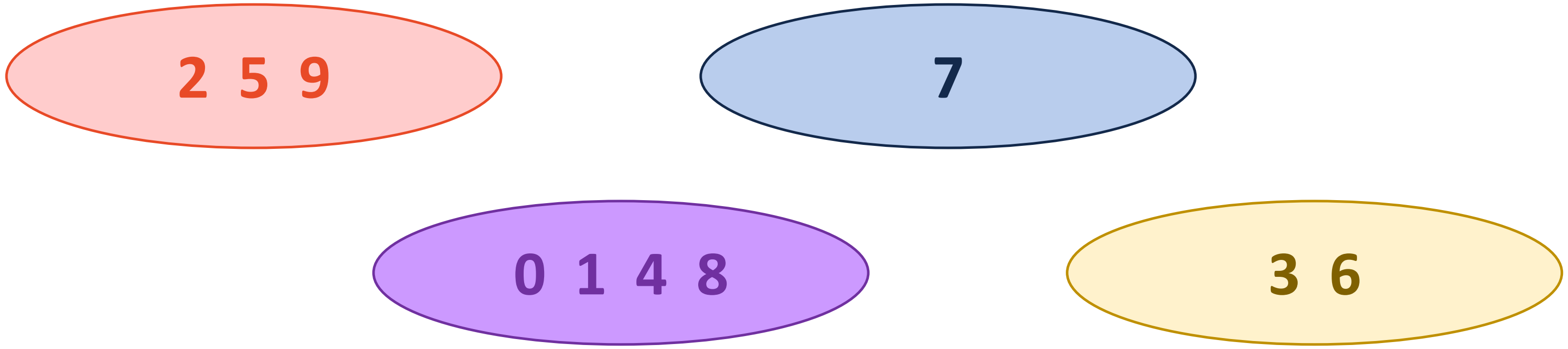
Operation: find(4)

Disjoint Sets



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

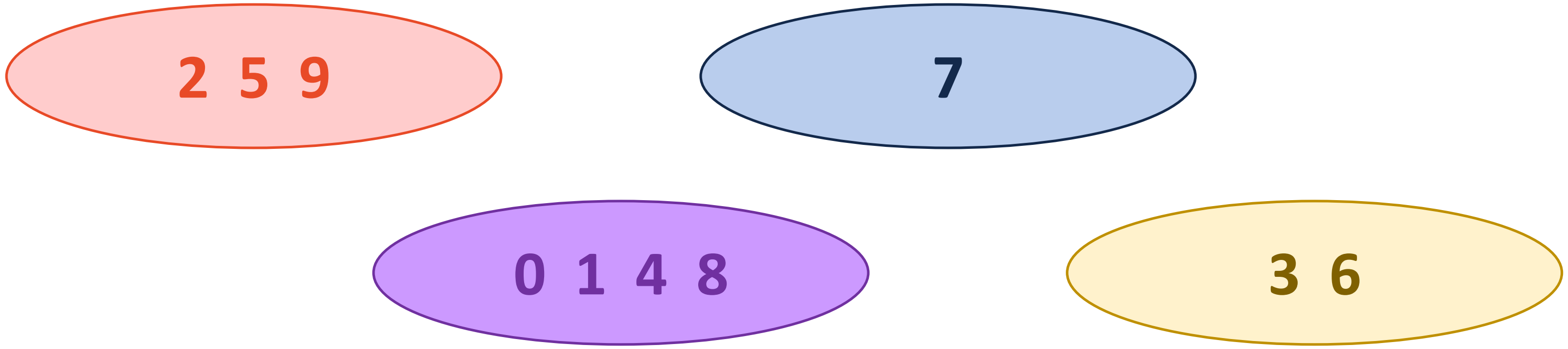
Disjoint Sets



Operation:

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

Disjoint Sets



Key Ideas:

- Each element exists in exactly one set.
- Every set is an equitant representation.
  - Mathematically: $4 \in [0]_R \rightarrow 8 \in [0]_R$
  - Programmatically: find(4) == find(8)

Disjoint Sets ADT

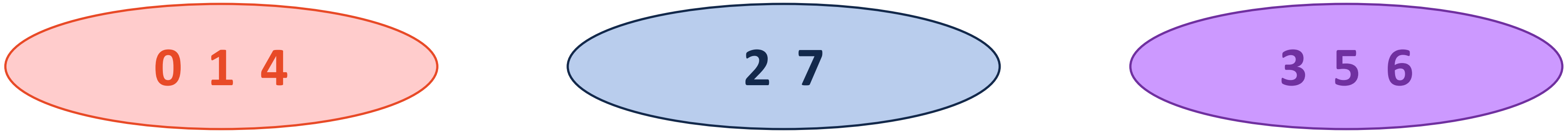
- Maintain a collection $S = \{s_0, s_1, \ldots s_k\}$

- 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):

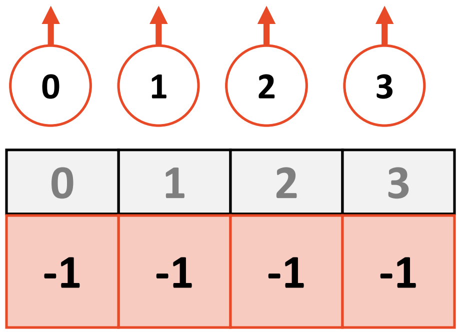
Union(k1, k2):

Implementation #2

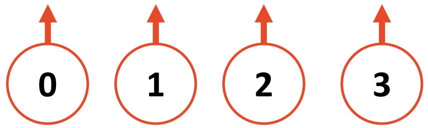
- We will continue to use an array where the index is the key

- 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**:



UpTrees



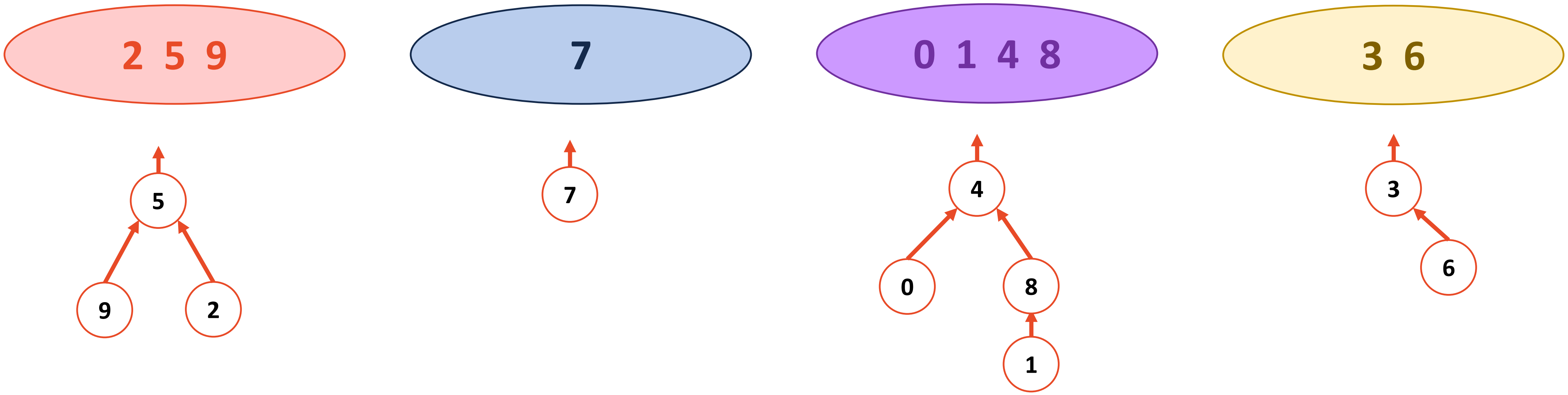
0	1	2	3
-1	-1	-1	-1

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

Disjoint Sets Find

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

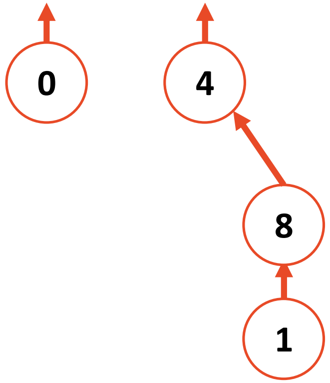
Running time?

What is the ideal UpTree?

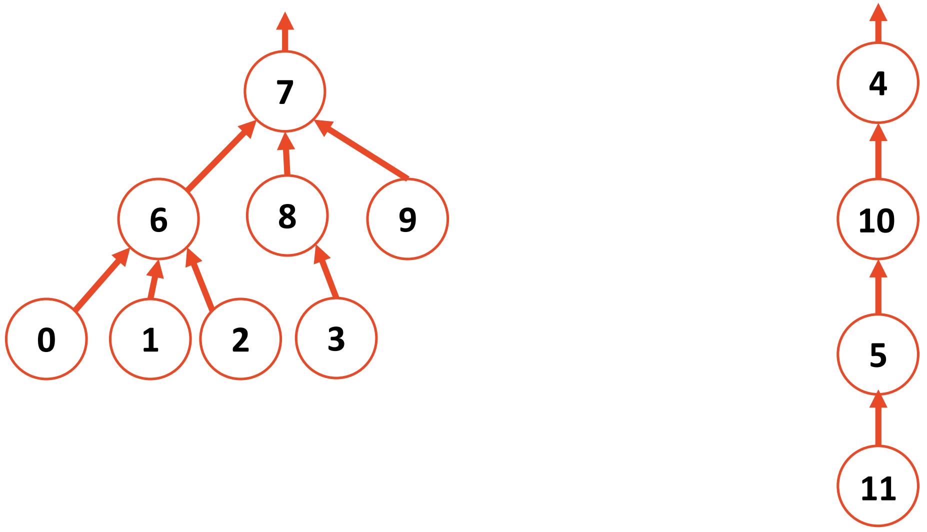
Disjoint Sets Union

```
void DisjointSets::union(int r1, int r2) {


}
```

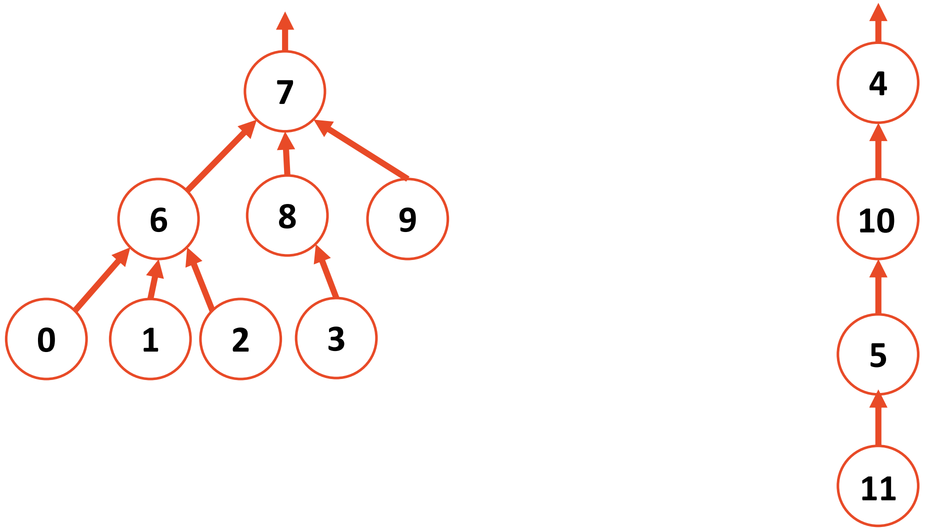


Disjoint Sets – Union



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

Disjoint Sets – Smart Union

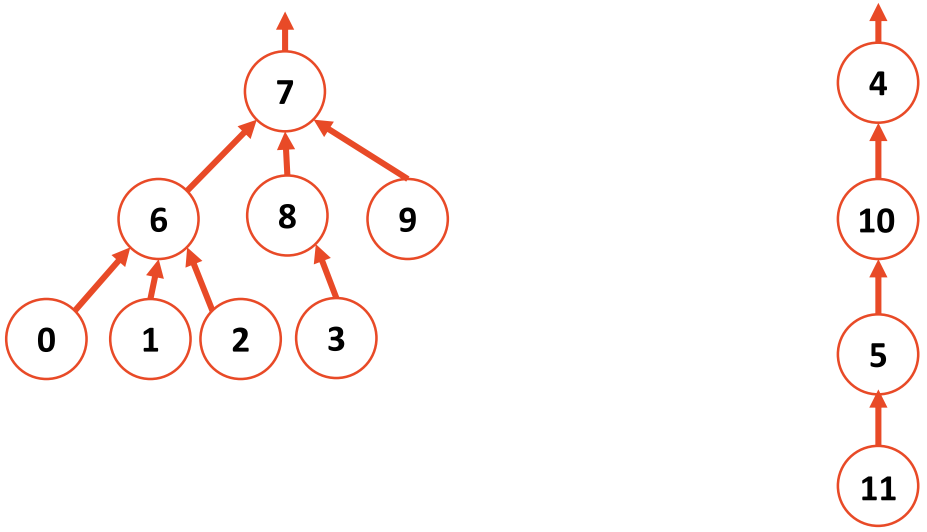


Union by height

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

***Idea**: Keep the height of the tree as small as possible.*

Disjoint Sets – Smart Union



Union by height

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

***Idea**: Keep the height of the tree as small as possible.*

Union by size

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

***Idea**: Minimize the number of nodes that increase in height*

Both guarantee the height of the tree is: _______________.

Disjoint Sets Find

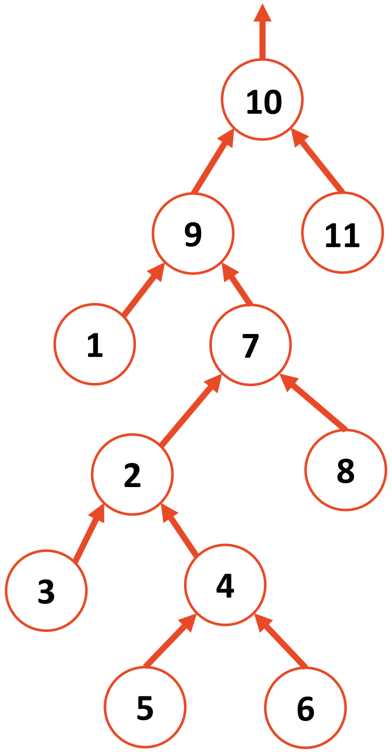
```
int DisjointSets::find(int i) {
  if ( s[i] < 0 ) { return i; }
  else { return _find( s[i] ); }
}
```

```
void DisjointSets::unionBySize(int root1, int root2) {
  int newSize = arr_[root1] + arr_[root2];

  // If arr_[root1] is less than (more negative), it is the larger set;
  // we union the smaller set, root2, with root1.
  if ( arr_[root1] < arr_[root2] ) {
    arr_[root2] = root1;
    arr_[root1] = newSize;
  }

  // Otherwise, do the opposite:
  else {
    arr_[root1] = root2;
    arr_[root2] = newSize;
  }
}
```

Path Compression



Disjoint Sets Analysis

The **iterated log** function:

The number of times you can take a log of a number.

log*(n) =

0 , $n \leq 1$

1 + log*(log(n)) , $n > 1$

What is **lg*(2^{65536})**?

Disjoint Sets Analysis

In an Disjoint Sets implemented with smart **unions** and path compression on **find**:

Any sequence of **m union** and **find** operations result in the worse case running time of O(____________),
where **n** is the number of items in the Disjoint Sets.