CS 225

Data Structures

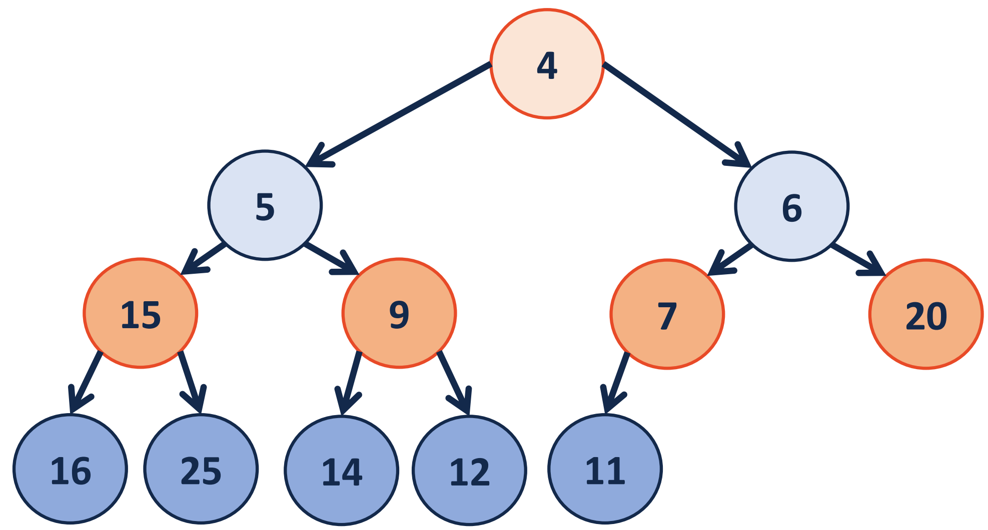
November 2 – Heaps Take 2

G Carl Evans

(min)Heap

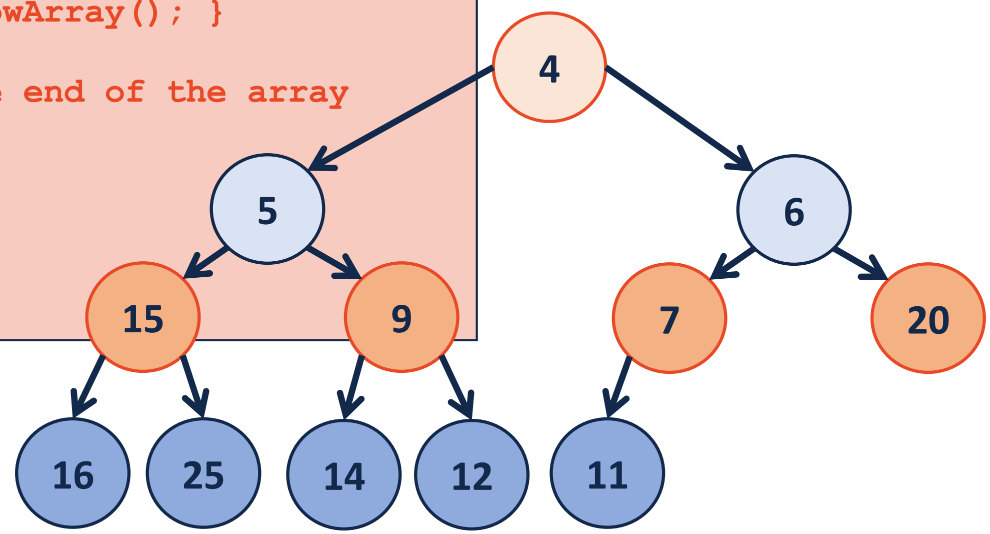
A complete binary tree T is a min-heap if:

- **T = {}** or
- **T = {r, T_L, T_R}**, where **r** is less than the roots of {**T_L, T_R**} and {**T_L, T_R**} are min-heaps.



insert

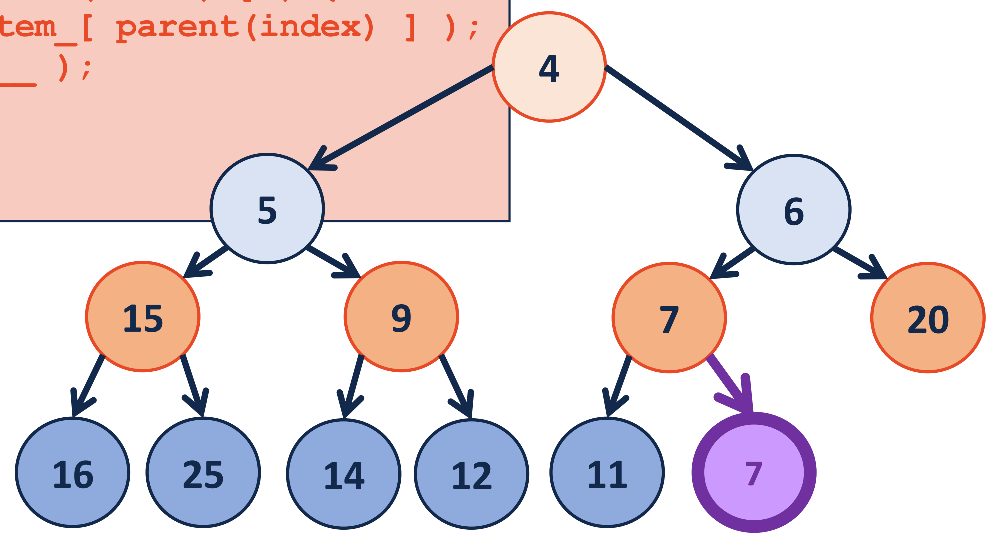
```
template <class T>
void Heap<T>::_insert(const T & key) {
  // Check to ensure there’s space to insert an element
  // ...if not, grow the array
  if ( size_ == capacity_ ) { _growArray(); }

  // Insert the new element at the end of the array
  item_[++size] = key;

  // Restore the heap property
  _heapifyUp(size);
}
```



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

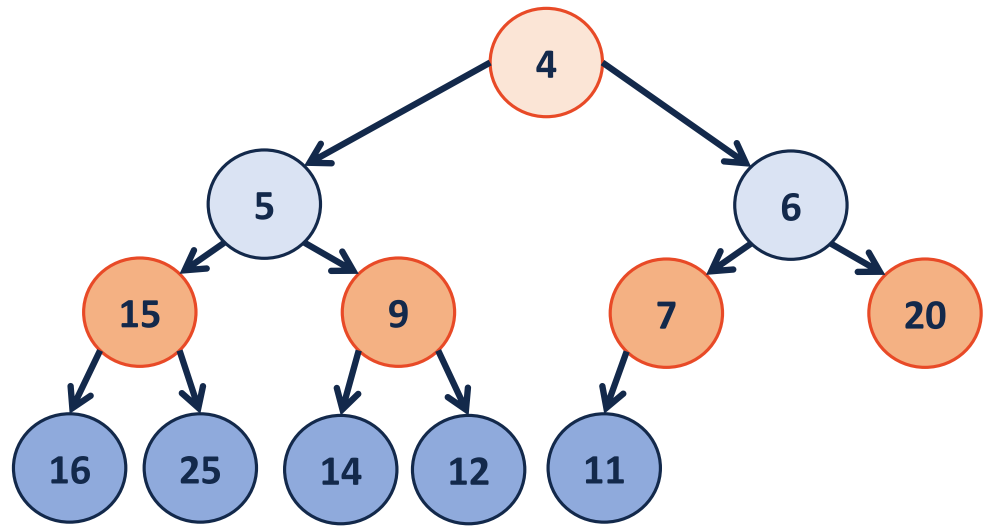
heapifyUp

```
template <class T>
void Heap<T>::_heapifyUp( __________________ ) {
  if ( index > _________ ) {
    if ( item_[index] < item_[ parent(index) ] ) {
       std::swap( item_[index], item_[ parent(index) ] );
       _heapifyUp( ________________ );
    }
  }
}
```



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

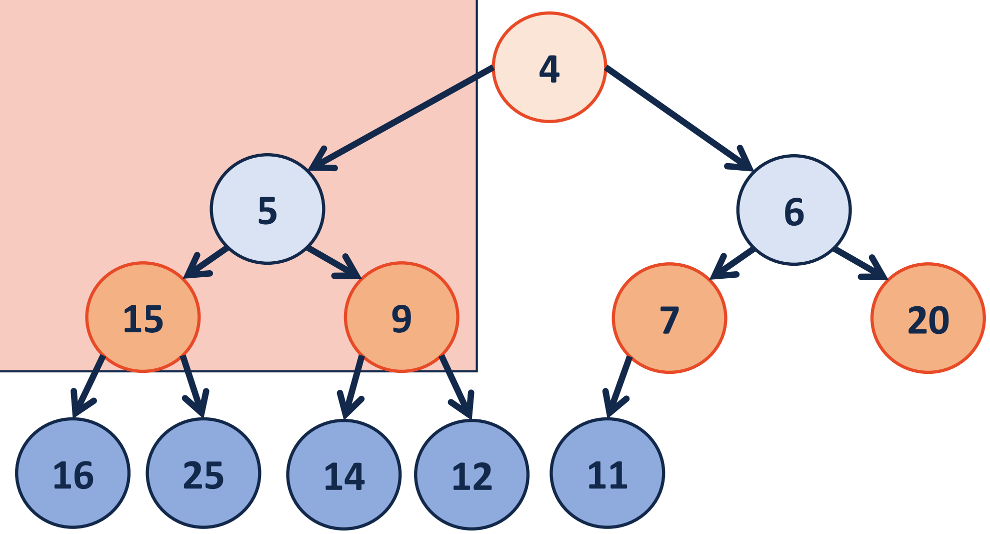
removeMin



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

removeMin

```
template <class T>
void Heap<T>::_removeMin() {
  // Swap with the last value
  T minValue = item_[1];
  item_[1] = item_[size_];
  size--;

  // Restore the heap property
  heapifyDown(1);

  // Return the minimum value
  return minValue;
}
```



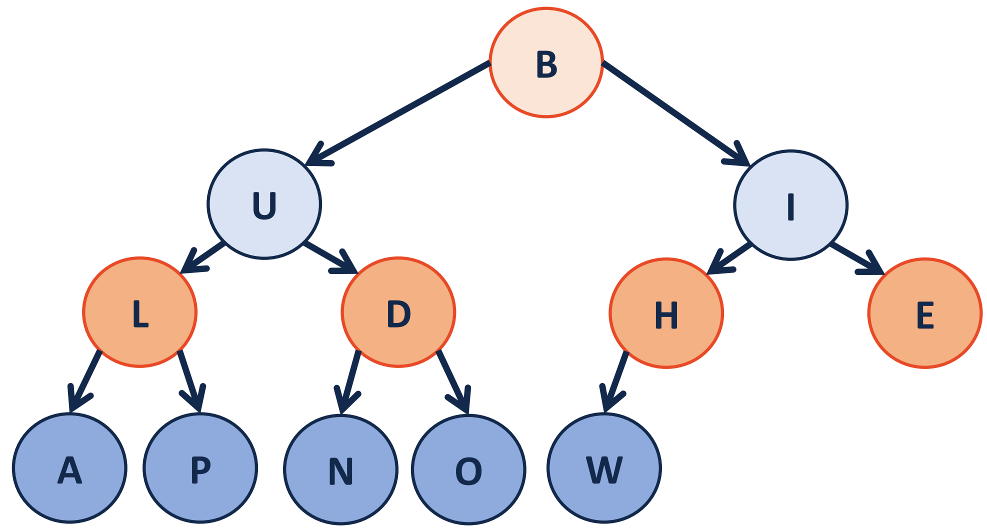
	4	5	6	15	9	7	20	16	25	14	12	11			

removeMin - heapifyDown

```
template <class T>
void Heap<T>::_removeMin() {
  // Swap with the last value
  T minValue = item_[1];
  item_[1] = item_[size_];
  size--;

  // Restore the heap property
  _heapifyDown(1);

  // Return the minimum value
  return minValue;
}
```

```
template <class T>
void Heap<T>::_heapifyDown(size_t index = 1) {
  if ( !_isLeaf(index) ) {
    size_t minChildIndex = _minChild(index);
    if ( item_[index] ___ item_[minChildIndex] ) {
       std::swap( item_[index], item_[minChildIndex] );
       _heapifyDown( ________________ );
    }
  }
}
```

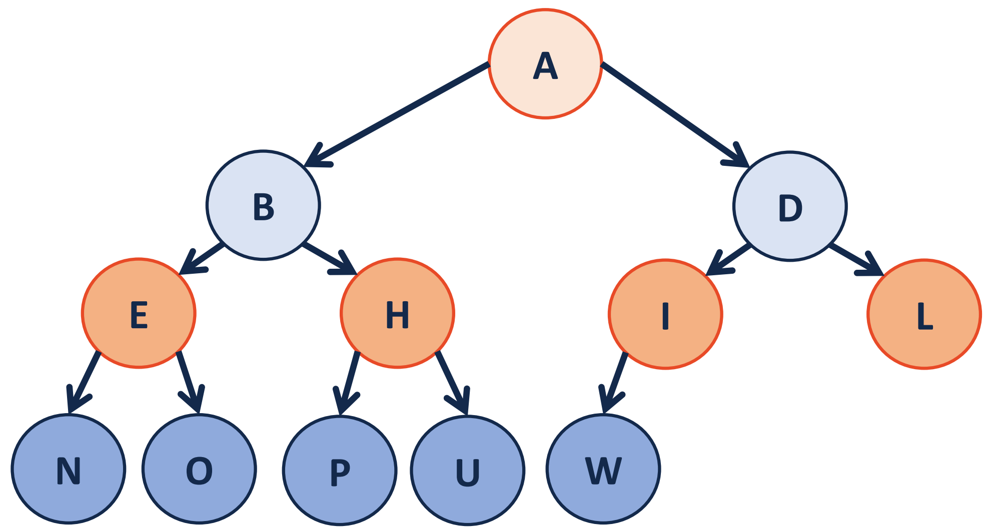
buildHeap



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

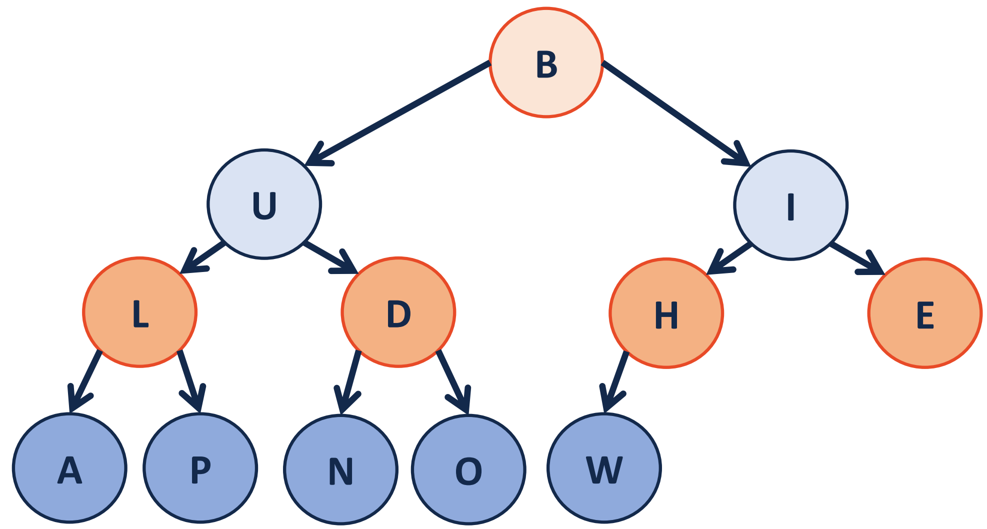
buildHeap – sorted array

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



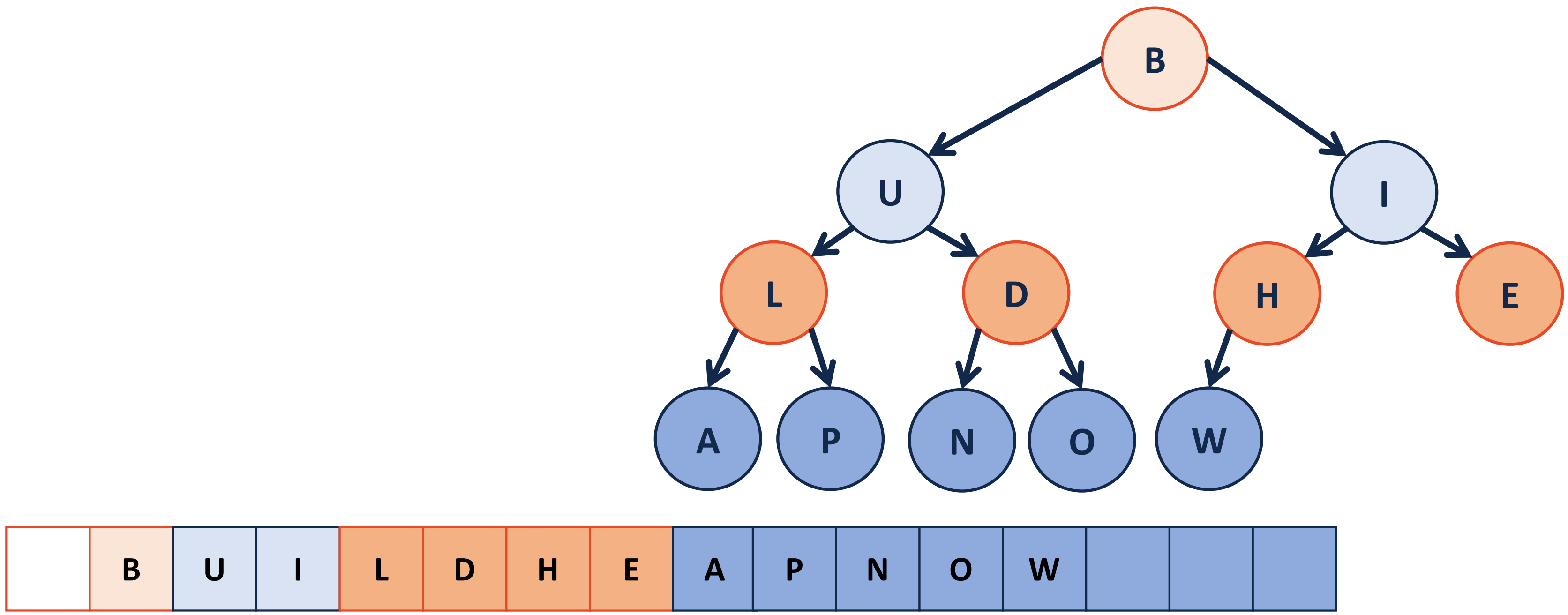
	A	B	D	E	H	I	L	N	O	P	U	W			

buildHeap - heapifyUp



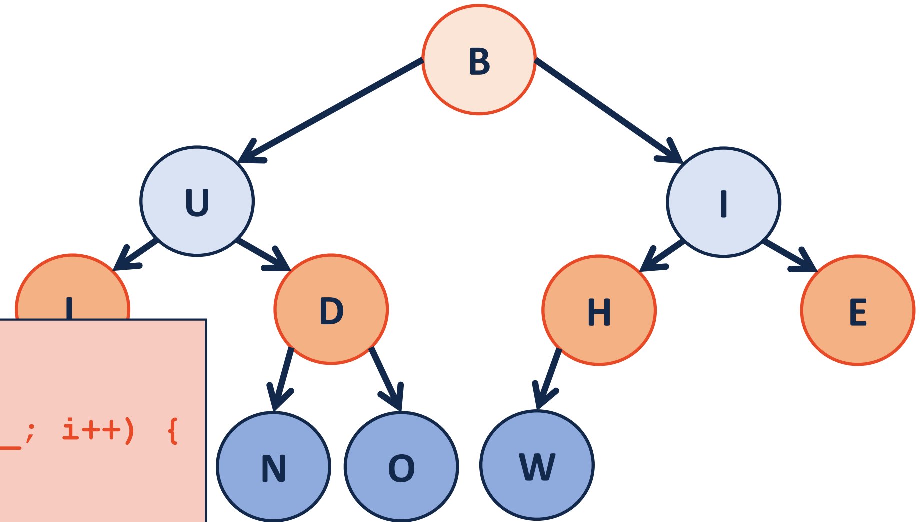
	B	U	I	L	D	H	E	A	P	N	O	W			

buildHeap - heapifyDown



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);
  }
}
```

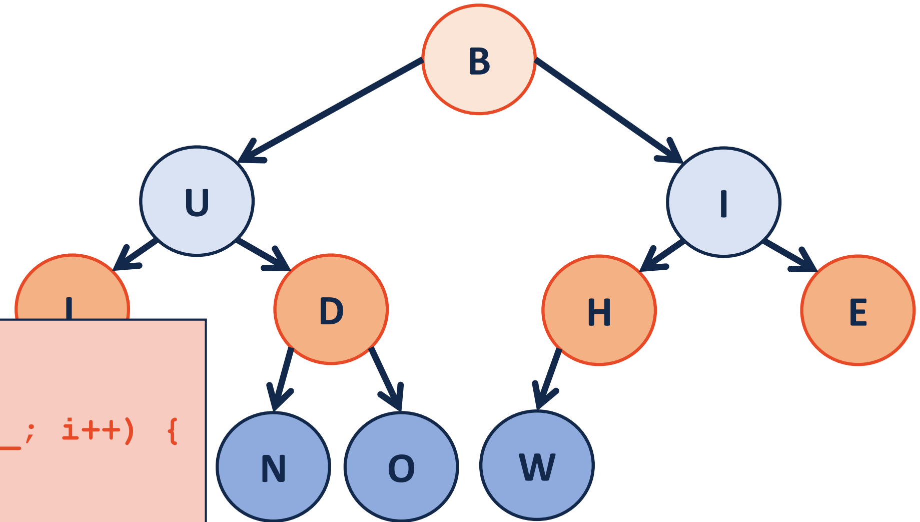
3.

```
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

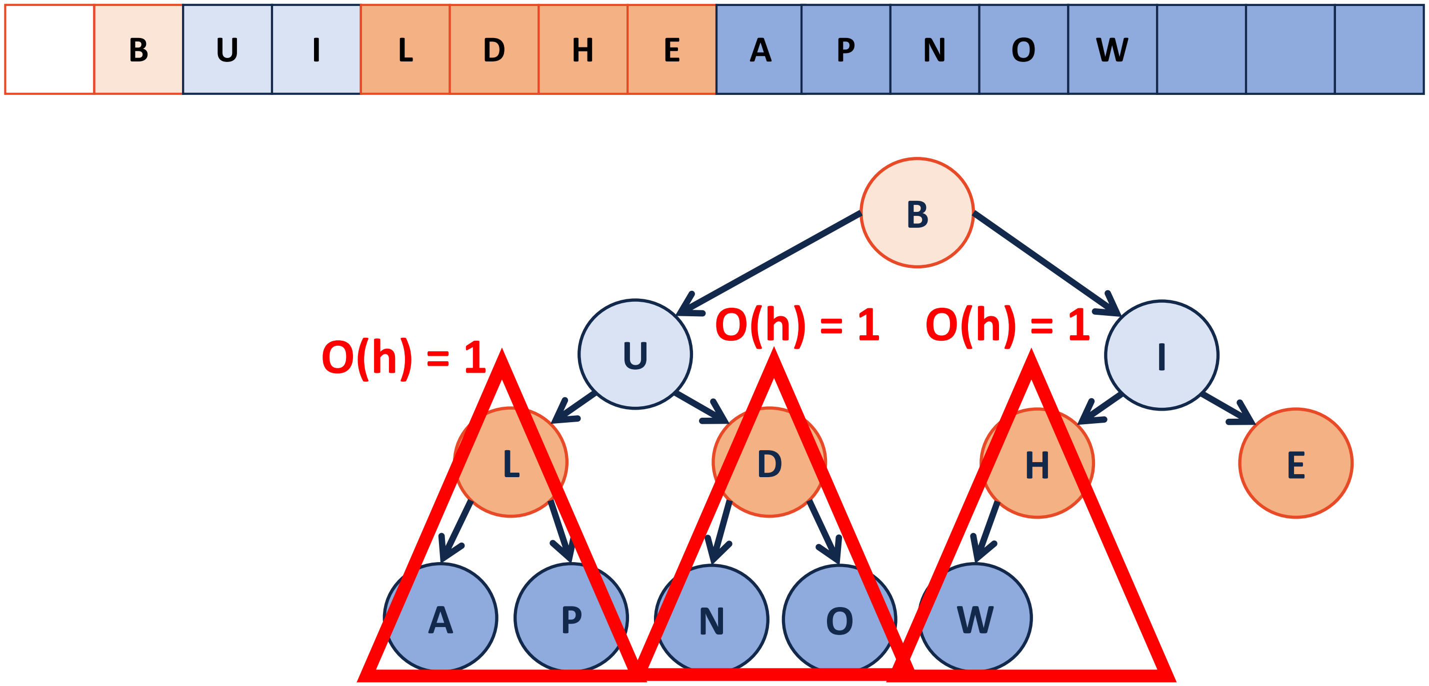
1. Sort the array – it's a heap!



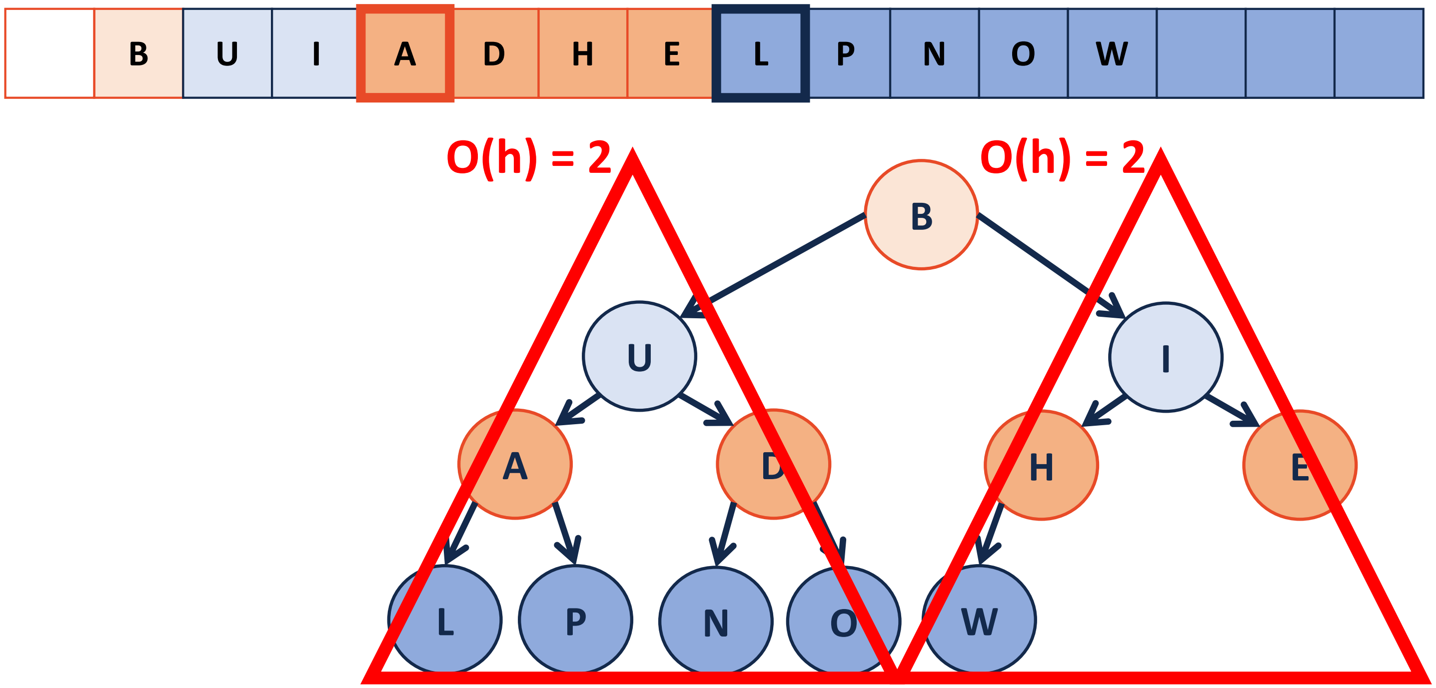
2.
```
template <class T>
void Heap<T>::buildHeap() {
  for (unsigned i = 2; i <= size_; i++) {
    heapifyUp(i);
  }
}
```

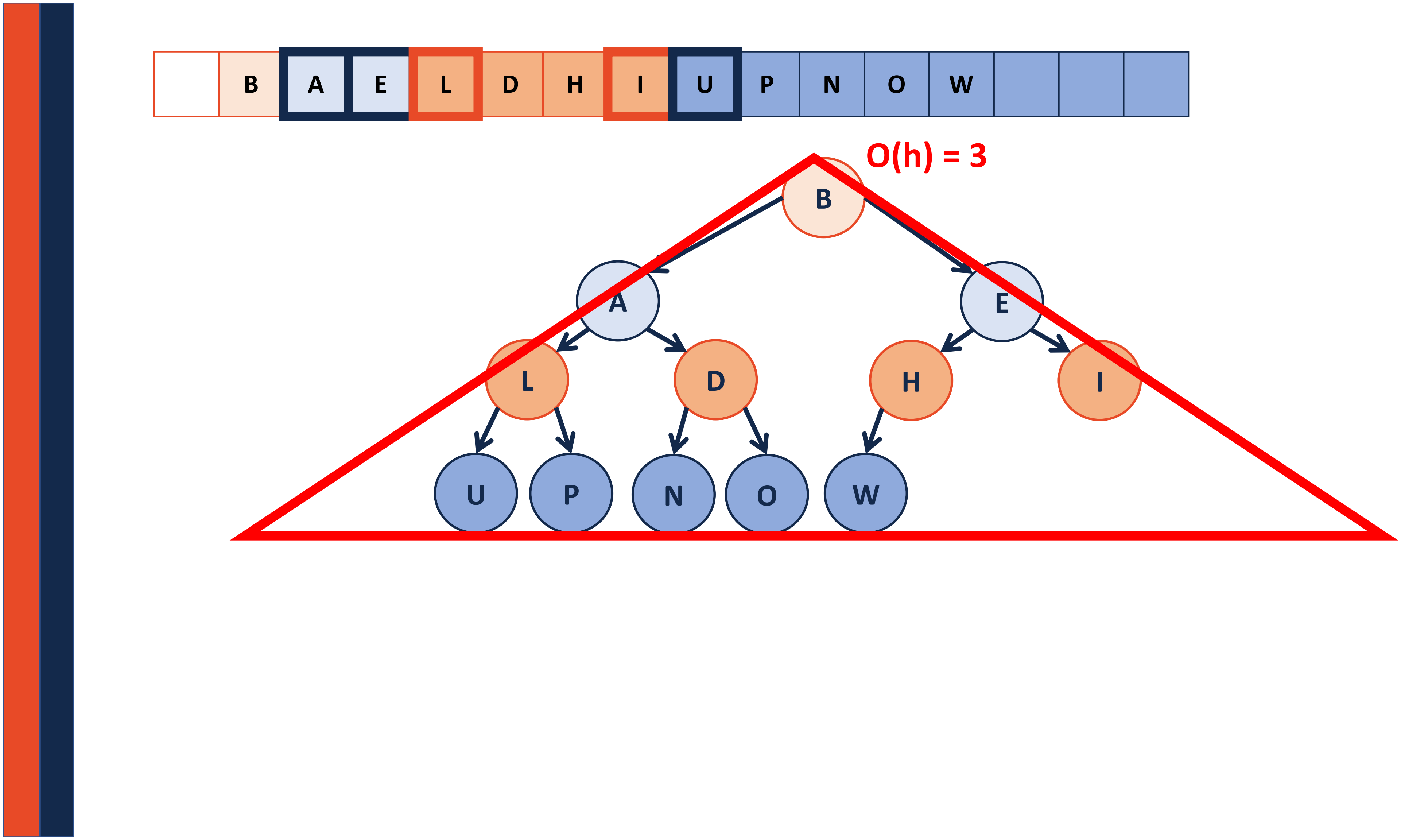
3.
```
template <class T>
void Heap<T>::buildHeap() {
  for (unsigned i = parent(size); i > 0; i--) {
    heapifyDown(i);
  }
}
```

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



B
U
I
L
D
H
E
A
P
N
O
W
O(h) = 1
O(h) = 1
O(h) = 1


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



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

Proving buildHeap Running Time

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

Strategy:

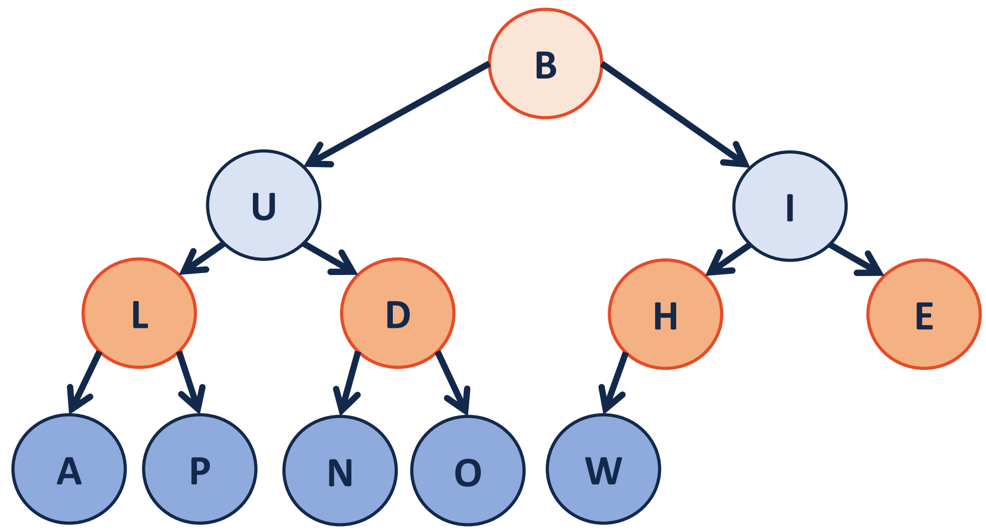
-

-

-

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:

IH:

General Case:

Proving buildHeap Running Time

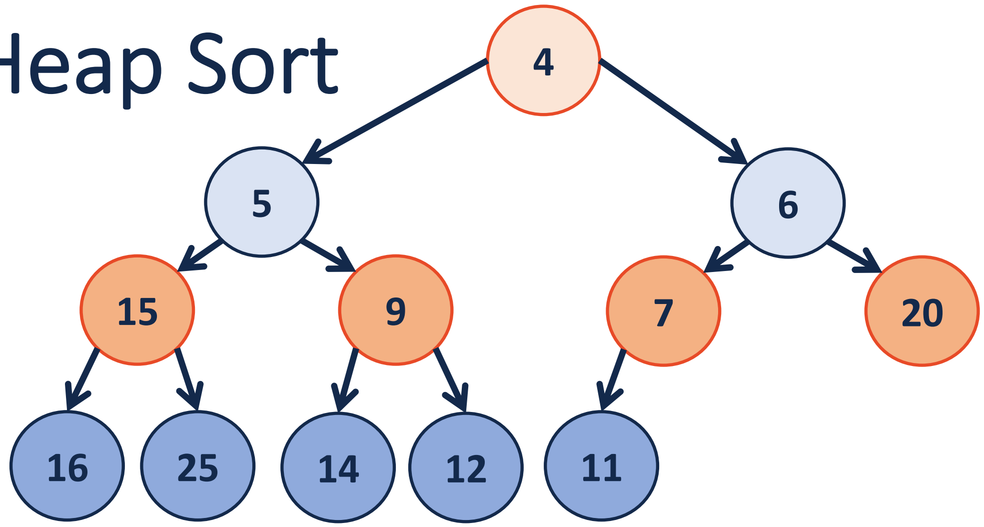
From S(h) to RunningTime(n):

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?

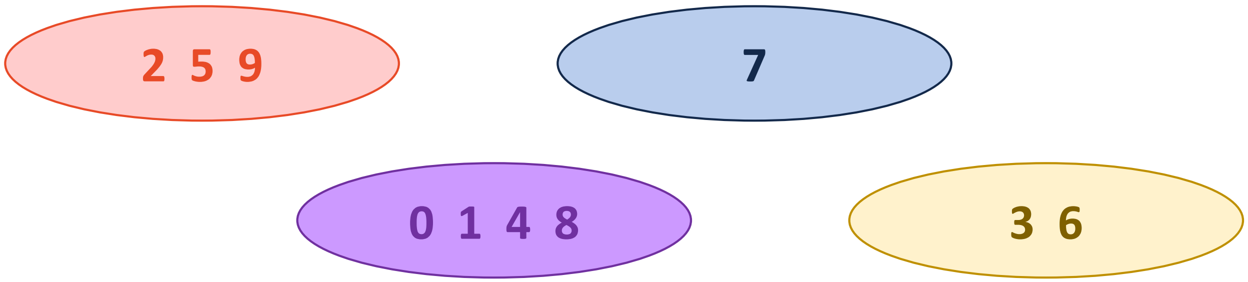
Why do we care about another sort?

A(nother) throwback to CS 173…

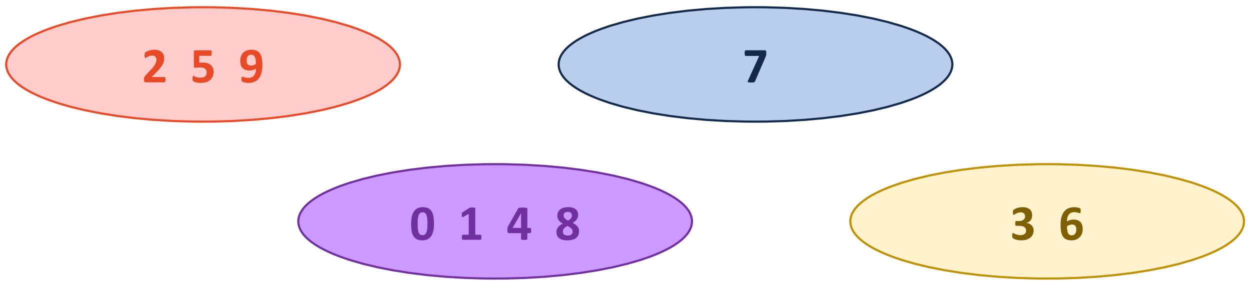
Let **R** be an equivalence relation on *us* where **(s, t)** ∈ **R** if **s** and **t** have the same favorite among:

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

Disjoint Sets

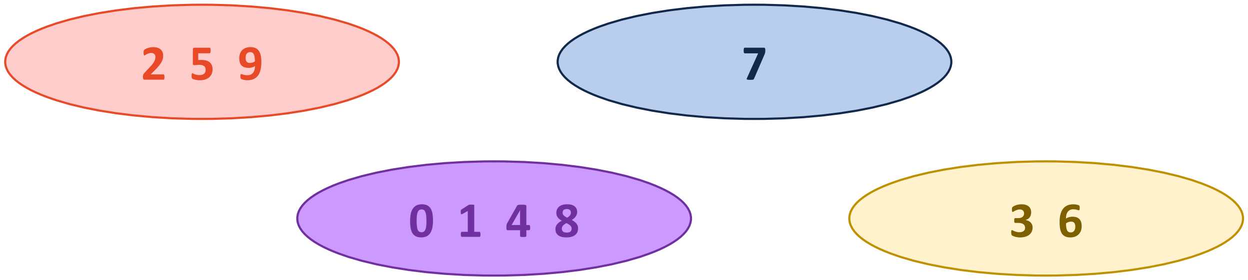


Disjoint Sets



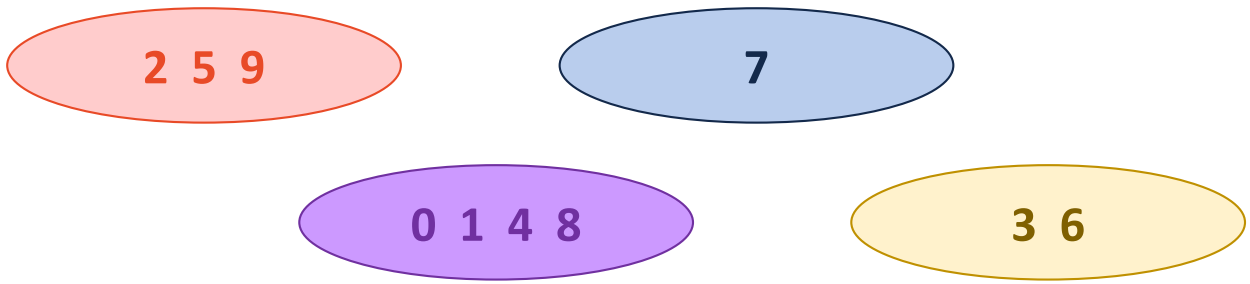
Operation: find(4)

Disjoint Sets



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

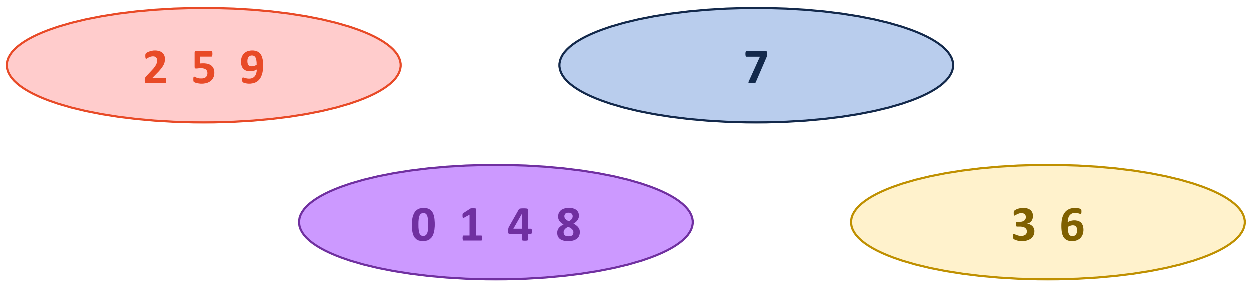
Disjoint Sets



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);
```