



# CS 225

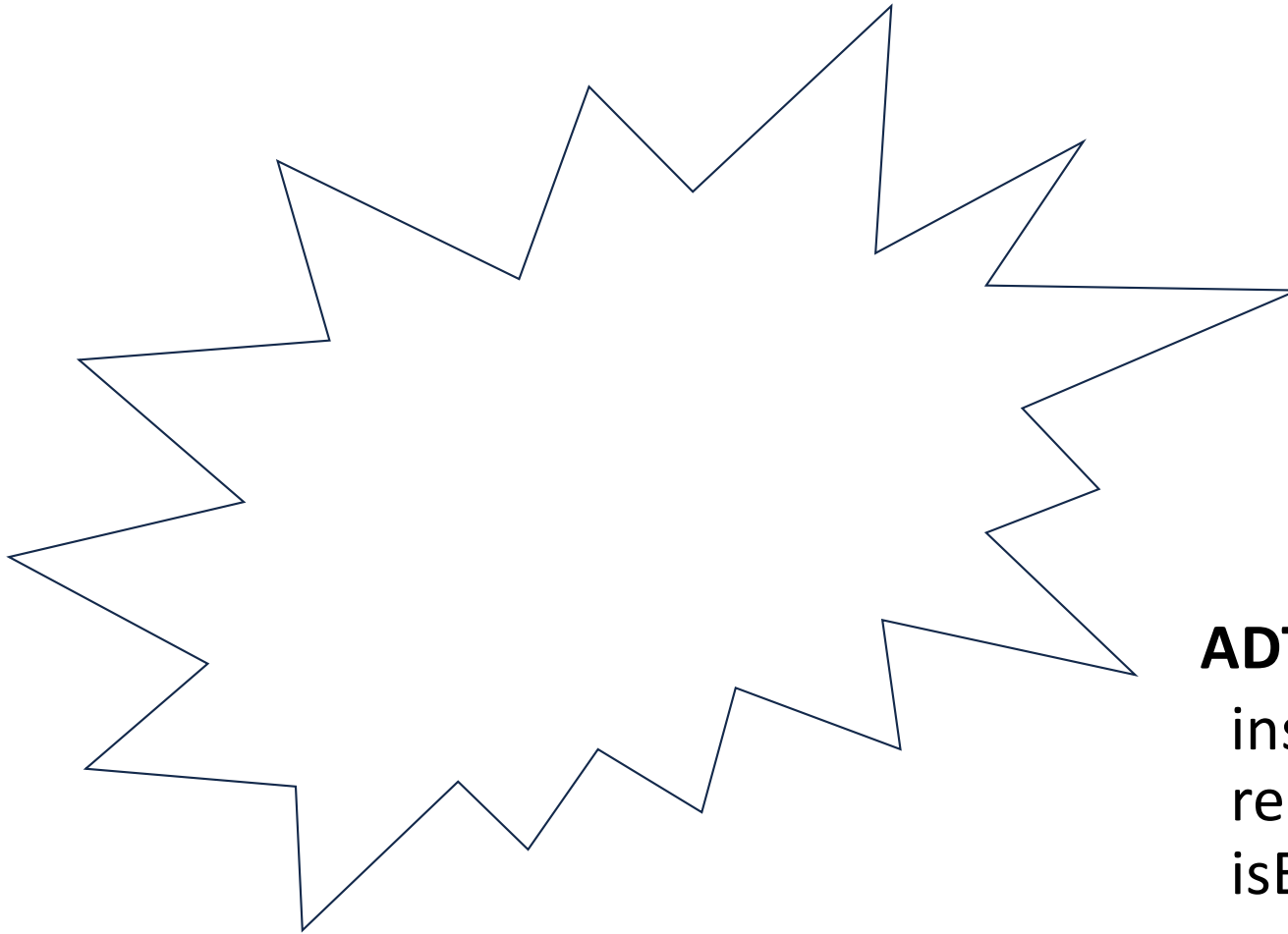
## Data Structures

*October 31 – Heaps and Priority Queues*

*G Carl Evans*



# Secret, Mystery Data Structure



**ADT:**

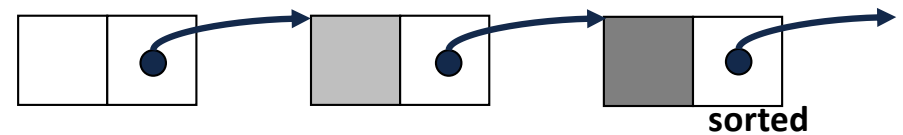
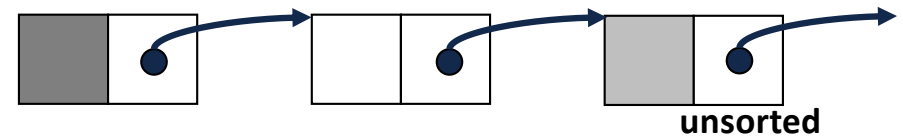
insert

remove

isEmpty

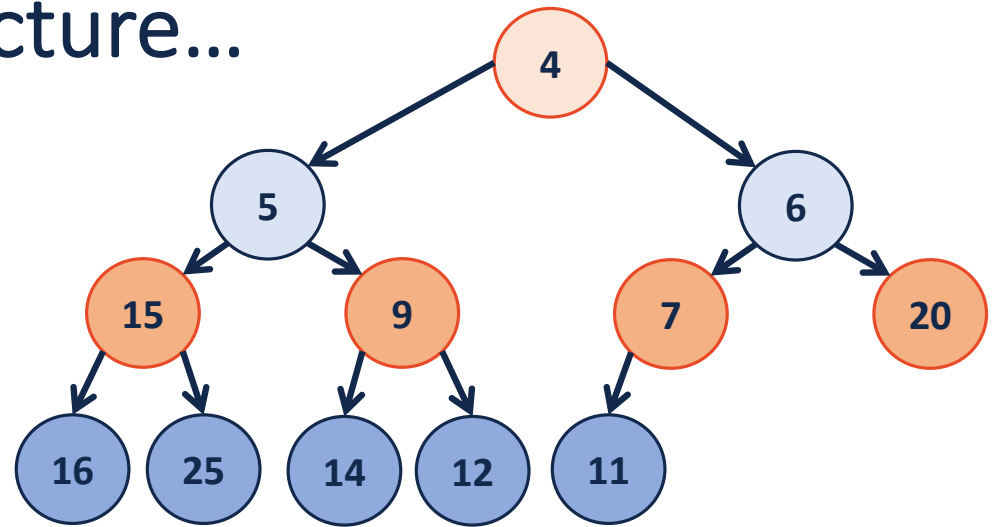
# Priority Queue Implementation

insert	removeMin
$O(n)$	$O(n)$
$O(1)$	$O(n)$
$O(\lg(n))$	$O(1)$
$O(\lg(n))$	$O(1)$



**What all is wrong in this table?**

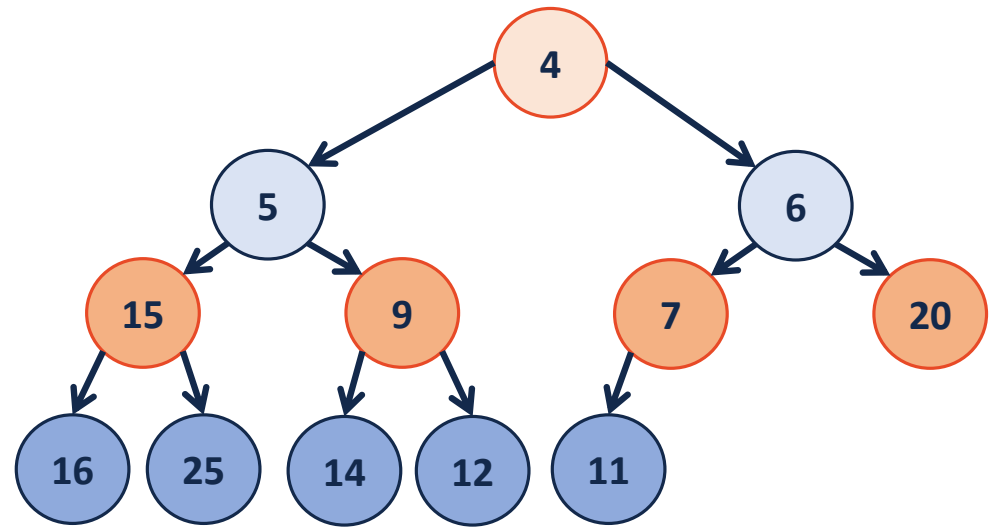
Another possibly structure...



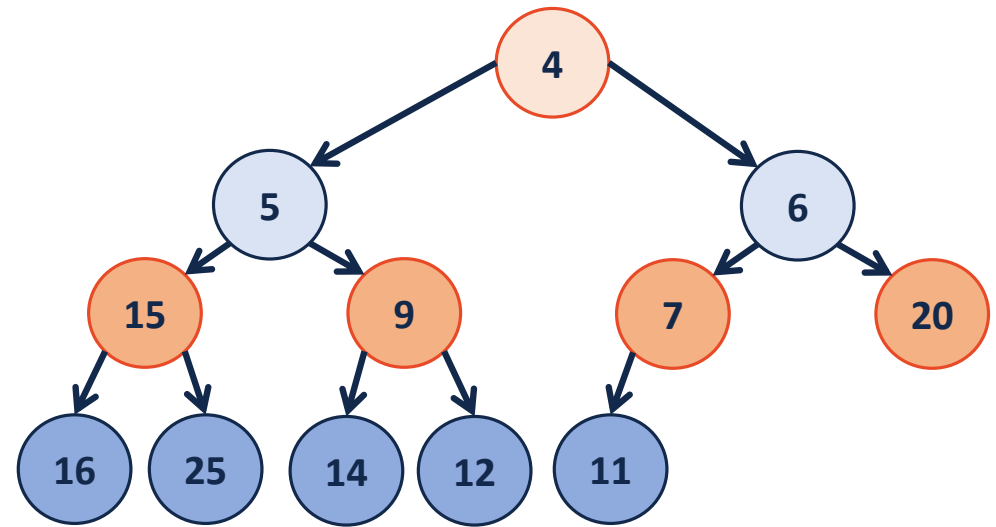
# (min)Heap

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.

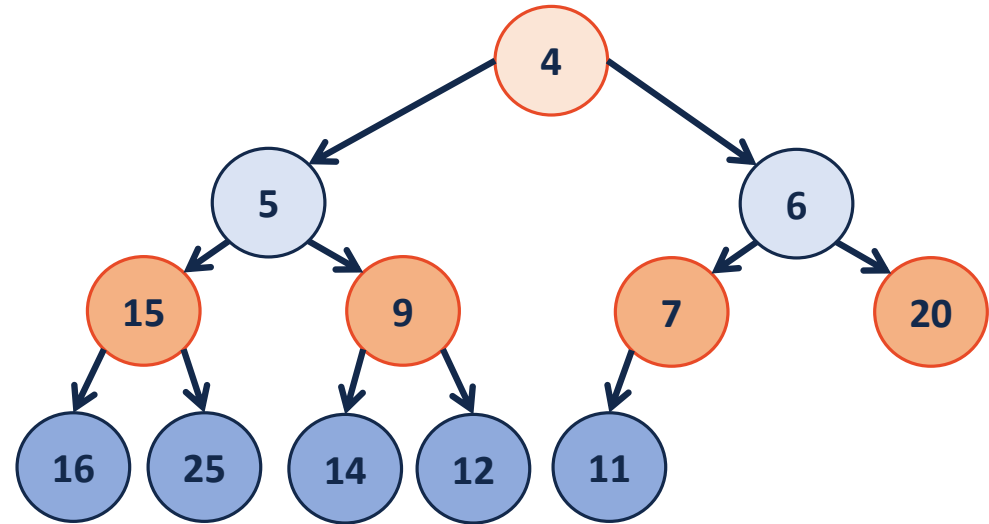


# (min)Heap



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

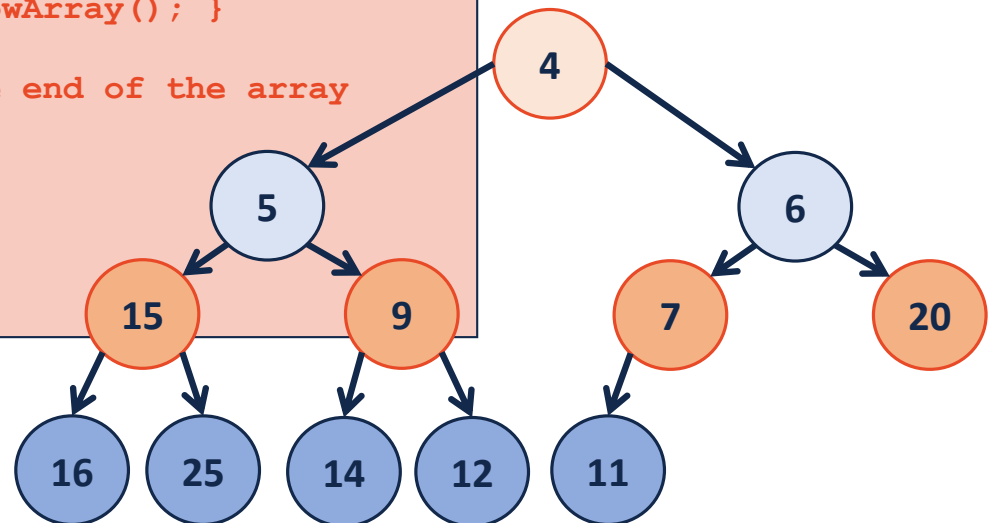
insert



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

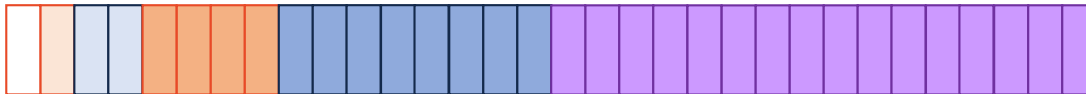
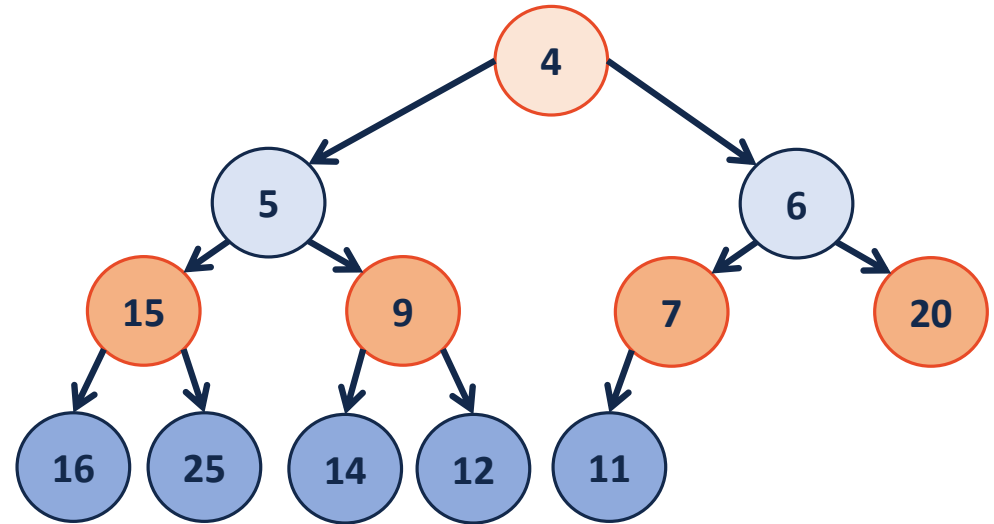
# insert

```
1 template <class T>
2 void Heap<T>::_insert(const T & key) {
3     // Check to ensure there's space to insert an element
4     // ...if not, grow the array
5     if ( size_ == capacity_ ) { _growArray(); }
6
7     // Insert the new element at the end of the array
8     item_[++size] = key;
9
10    // Restore the heap property
11    _heapifyUp(size);
12 }
```





# growArray



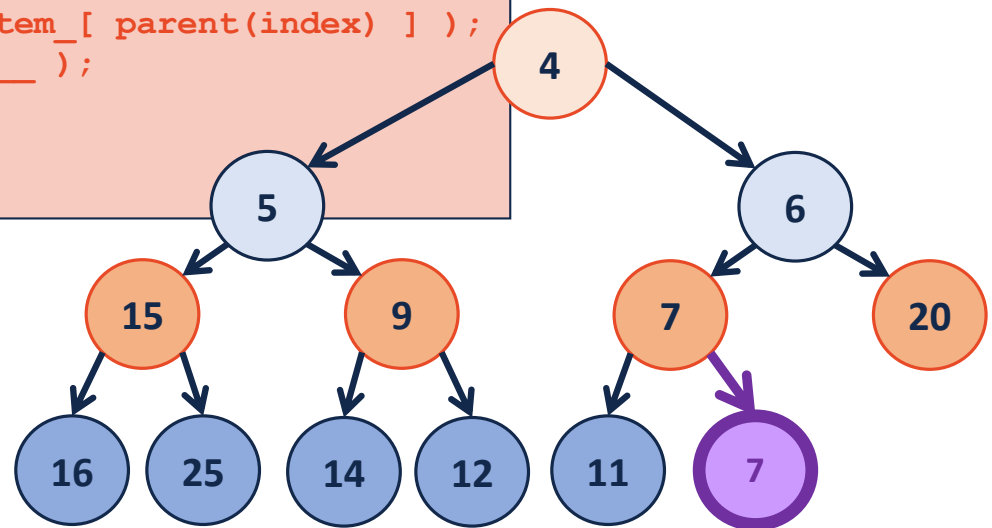
# insert - heapifyUp

```
1 template <class T>
2 void Heap<T>::_insert(const T & key) {
3     // Check to ensure there's space to insert an element
4     // ...if not, grow the array
5     if ( size_ == capacity_ ) { _growArray(); }
6
7     // Insert the new element at the end of the array
8     item_[++size] = key;
9
10    // Restore the heap property
11    _heapifyUp(size);
12 }
```

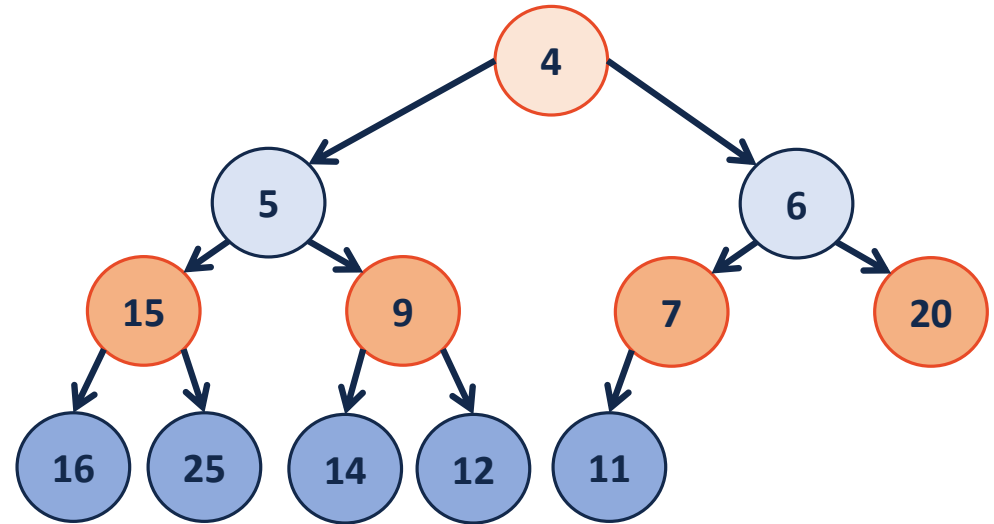
```
1 template <class T>
2 void Heap<T>::_heapifyUp( _____ ) {
3     if ( index > _____ ) {
4         if ( item_[index] < item_[ parent(index) ] ) {
5             std::swap( item_[index], item_[ parent(index) ] );
6             _heapifyUp( _____ );
7         }
8     }
9 }
```

# heapifyUp

```
1 template <class T>
2 void Heap<T>::_heapifyUp( _____ ) {
3     if ( index > _____ ) {
4         if ( item_[index] < item_[ parent(index) ] ) {
5             std::swap( item_[index], item_[ parent(index) ] );
6             _heapifyUp( _____ );
7         }
8     }
9 }
```



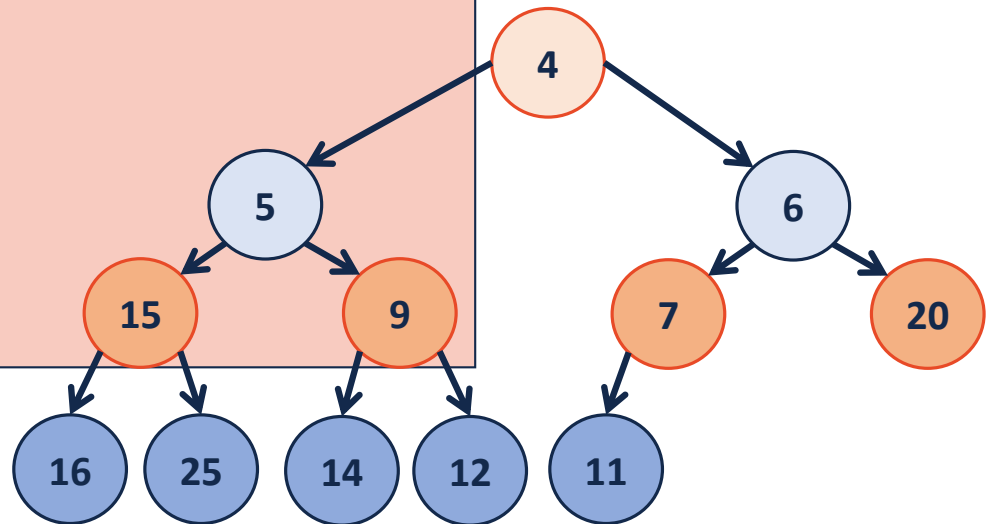
# removeMin



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

# removeMin

```
1 template <class T>
2 void Heap<T>::_removeMin() {
3     // Swap with the last value
4     T minValue = item_[1];
5     item_[1] = item_[size_];
6     size--;
7
8     // Restore the heap property
9     heapifyDown();
10
11     // Return the minimum value
12     return minValue;
13 }
```



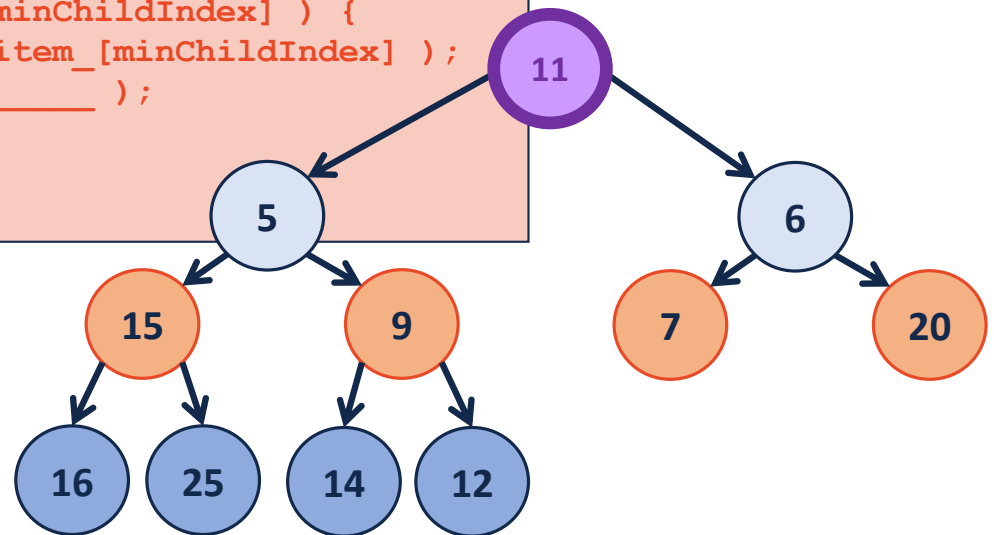
# removeMin - heapifyDown

```
1  template <class T>
2  void Heap<T>::_removeMin() {
3      // Swap with the last value
4      T minValue = item_[1];
5      item_[1] = item_[size_];
6      size--;
7
8      // Restore the heap property
9      _heapifyDown();
10
11     // Return the minimum value
12     return minValue;
13 }
```

```
1  template <class T>
2  void Heap<T>::_heapifyDown(int index) {
3      if ( !_isLeaf(index) ) {
4          T minChildIndex = _minChild(index);
5          if ( item_[index] > item_[minChildIndex] ) {
6              std::swap( item_[index], item_[minChildIndex] );
7              _heapifyDown( minChildIndex );
8          }
9      }
10 }
```

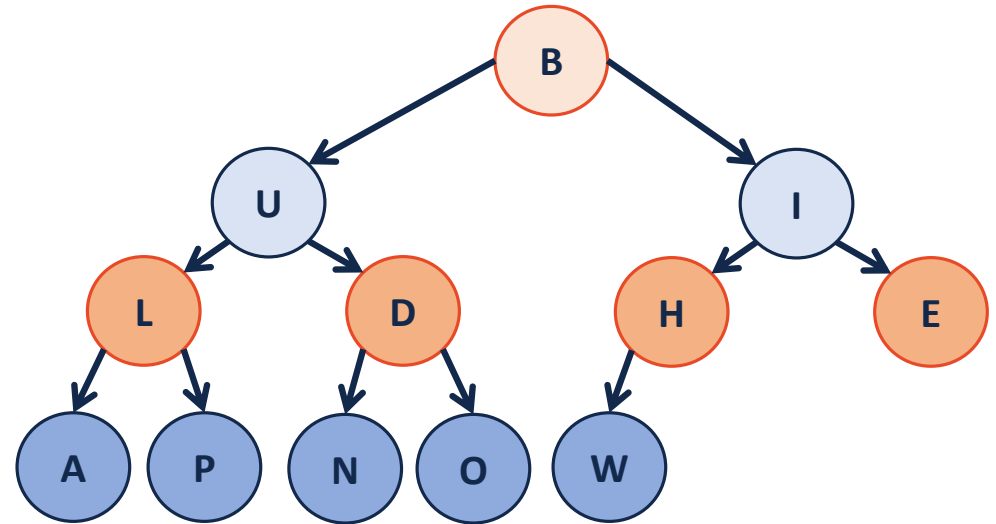
# removeMin

```
1 template <class T>
2 void Heap<T>::_heapifyDown(int index) {
3     if ( !_isLeaf(index) ) {
4         T minChildIndex = _minChild(index);
5         if ( item_[index] > item_[minChildIndex] ) {
6             std::swap( item_[index], item_[minChildIndex] );
7             _heapifyDown( minChildIndex );
8         }
9     }
10 }
```



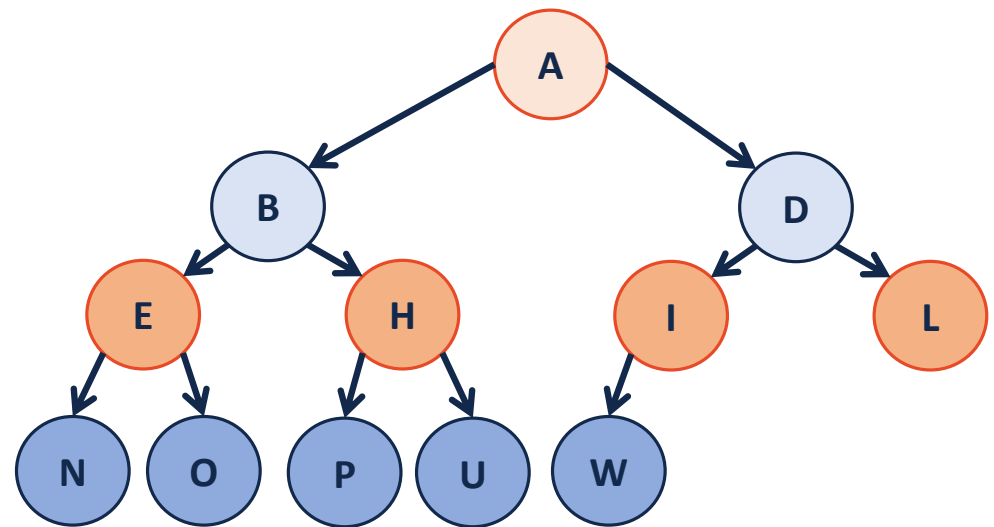
	4	5	6	15	9	7	20	16	25	14	12	11			
--	---	---	---	----	---	---	----	----	----	----	----	----	--	--	--

# buildHeap

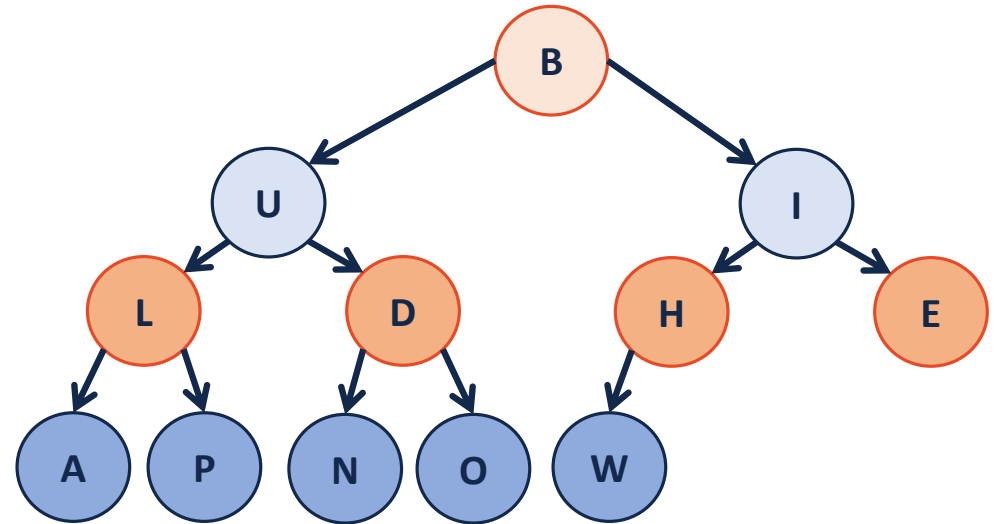




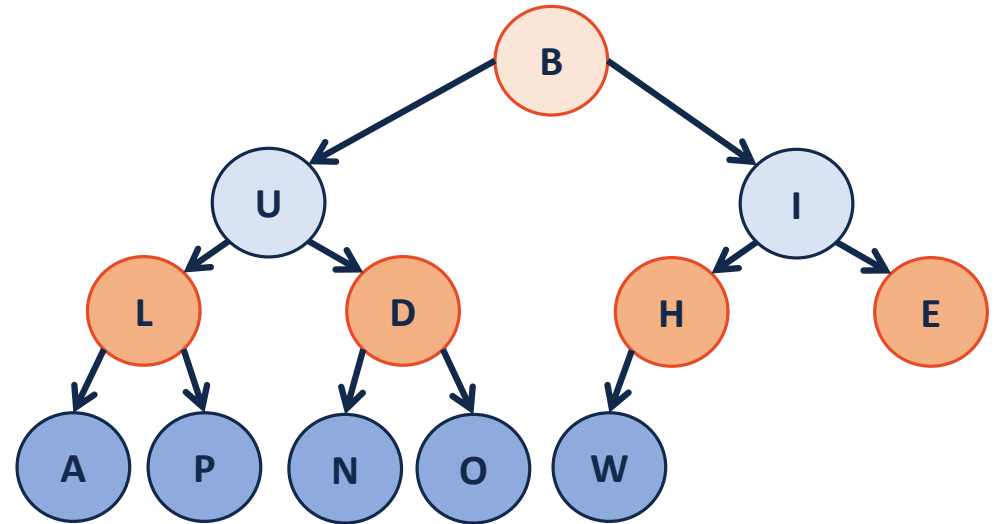
# buildHeap – sorted array



# buildHeap - heapifyUp



# buildHeap - heapifyDown



# buildHeap

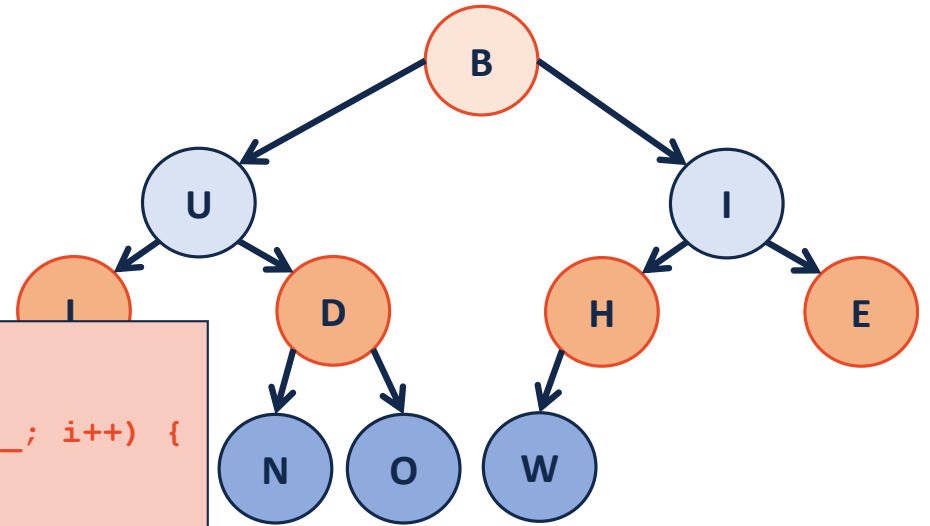
1. Sort the array – it's a heap!

2.

```
1 template <class T>
2 void Heap<T>::buildHeap() {
3     for (unsigned i = 2; i <= size_; i++) {
4         heapifyUp(i);
5     }
6 }
```

3.

```
1 template <class T>
2 void Heap<T>::buildHeap() {
3     for (unsigned i = parent(size); i > 0; i--) {
4         heapifyDown(i);
5     }
6 }
```





# Proving buildHeap Running Time

**Theorem:** The running time of buildHeap on array of size  $n$  is: \_\_\_\_\_.

**Strategy:**

-

-

-

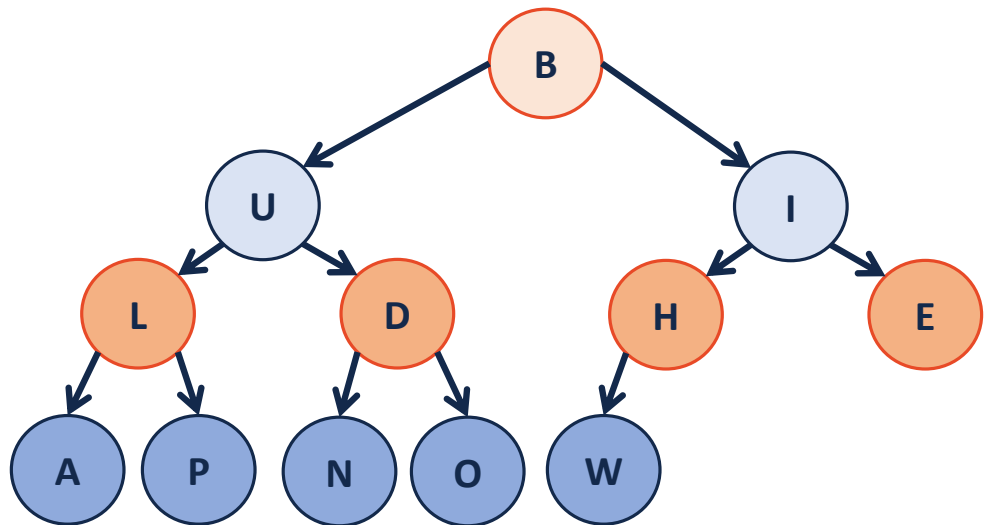
# Proving buildHeap Running Time

$S(h)$ : 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):**

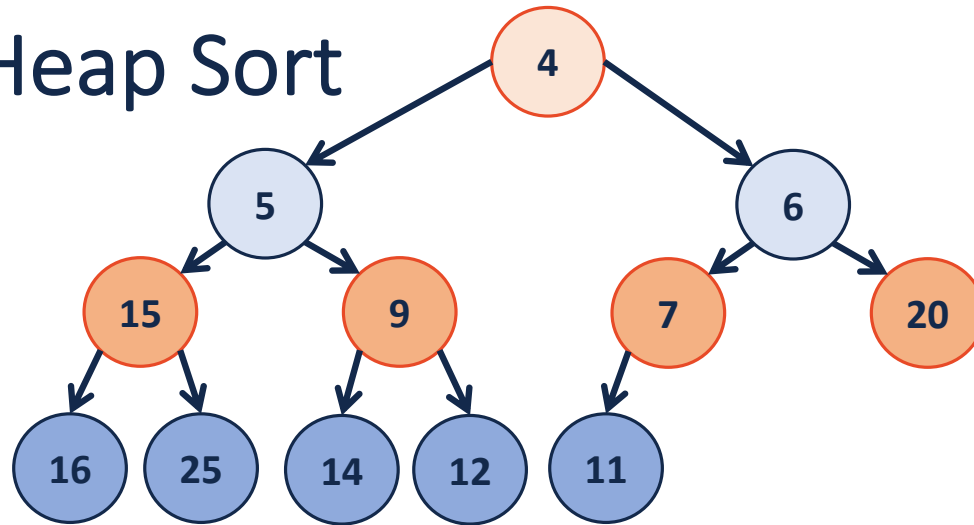
$S(h)$ :

Since  $h \leq \lg(n)$ :

RunningTime(n)  $\leq$



# Heap Sort



1.

2.

3.



Running Time?

Why do we care about another sort?



## A(nother) throwback to CS 173...

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

{ \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, }