CS 400

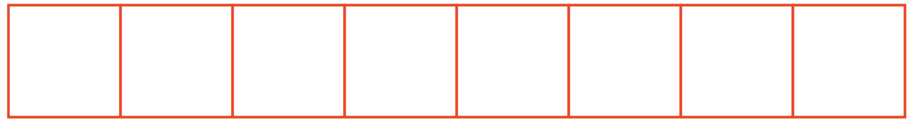
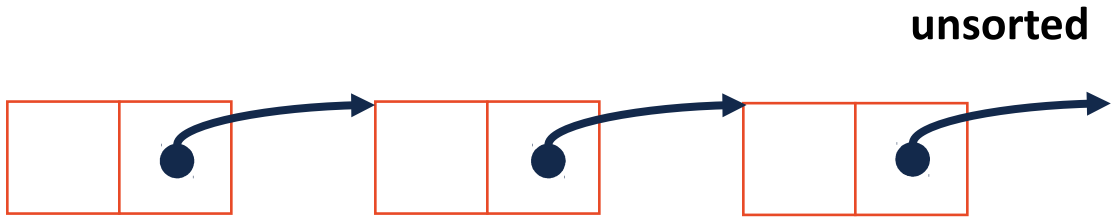
Heap - Introduction

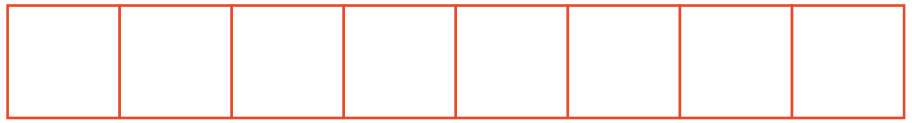
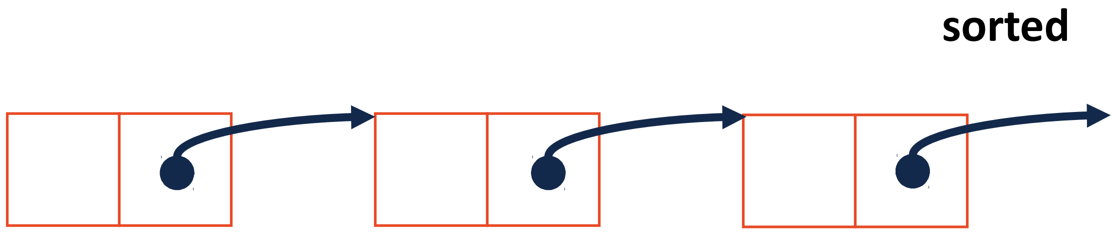
ID: 10-01

Priority Queue

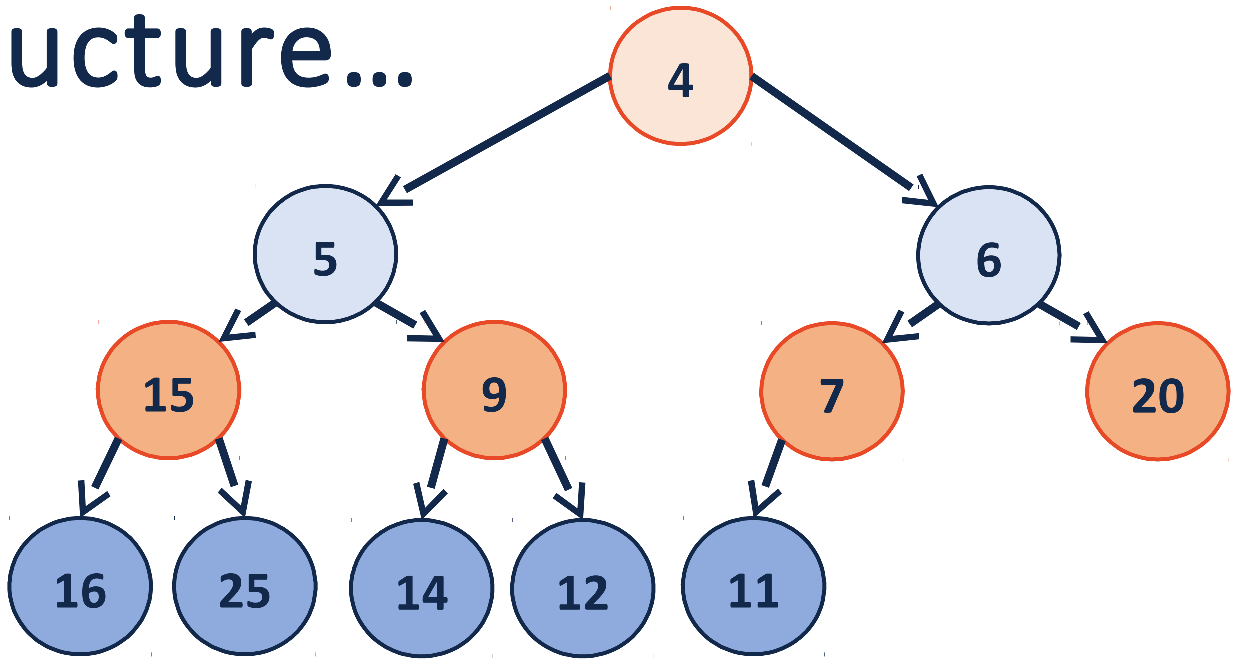
Priority Queue Implementation

insert	removeMin
O(1)*	O(n)
O(1)	O(n)
O(n)	O(1)
O(n)	O(1)

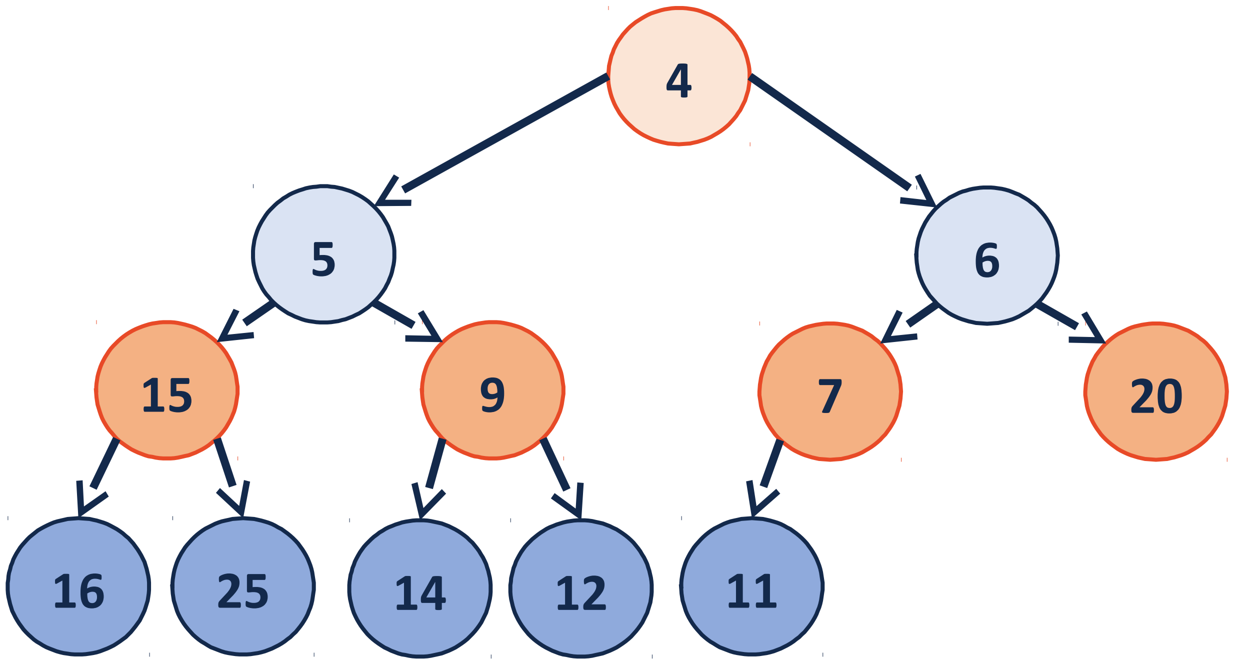
Another possibly structure…



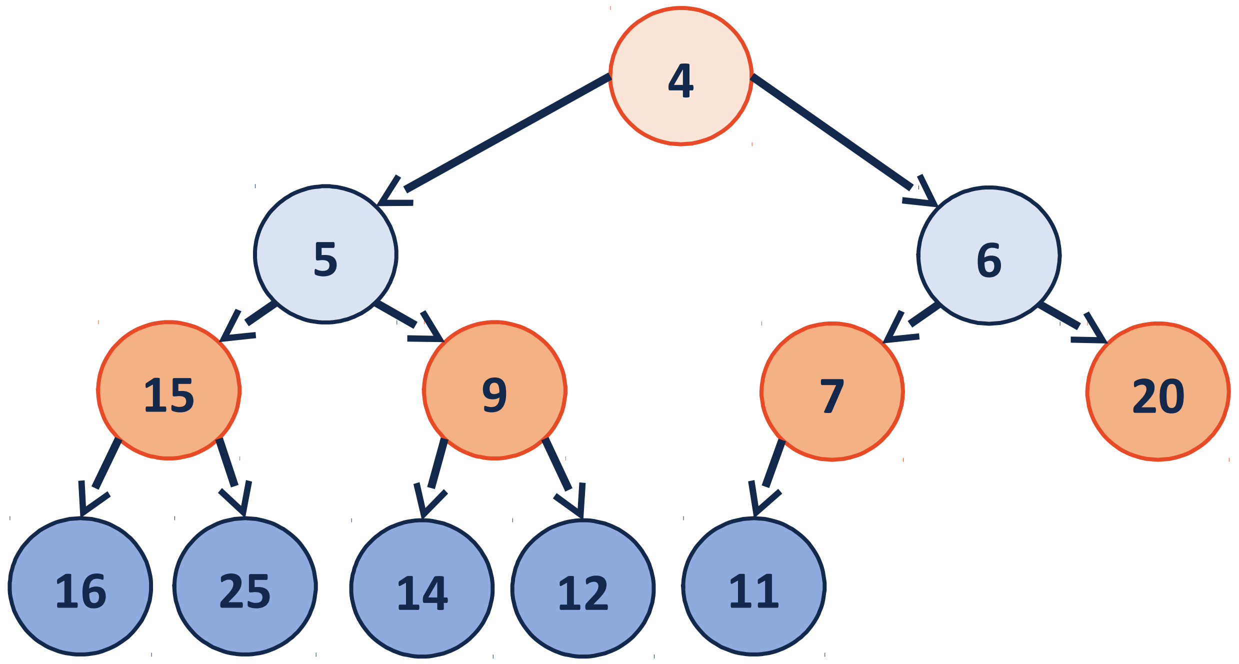
(min)Heap

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



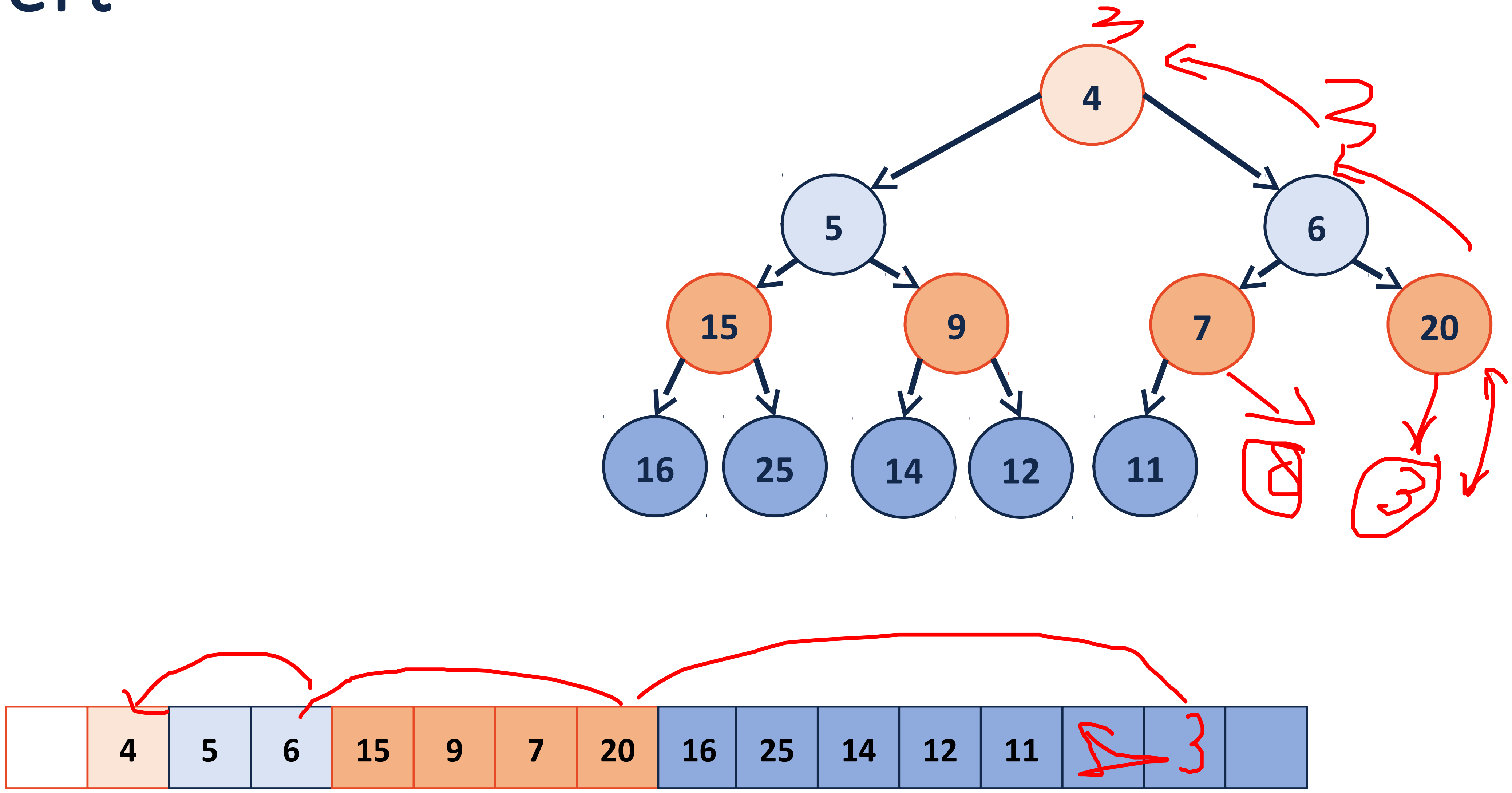
4	5	6	15	9	7	20	16	25	14	12	11			

CS 400

Heap – Insert and removeMin

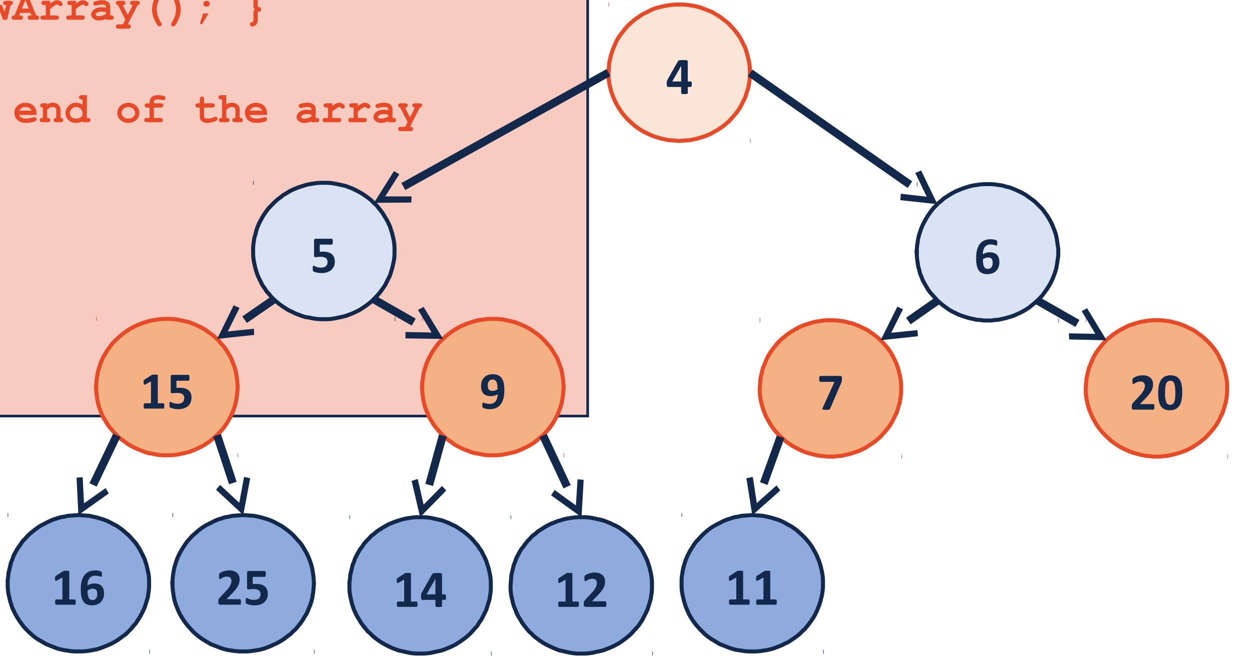
ID: 10-02

insert



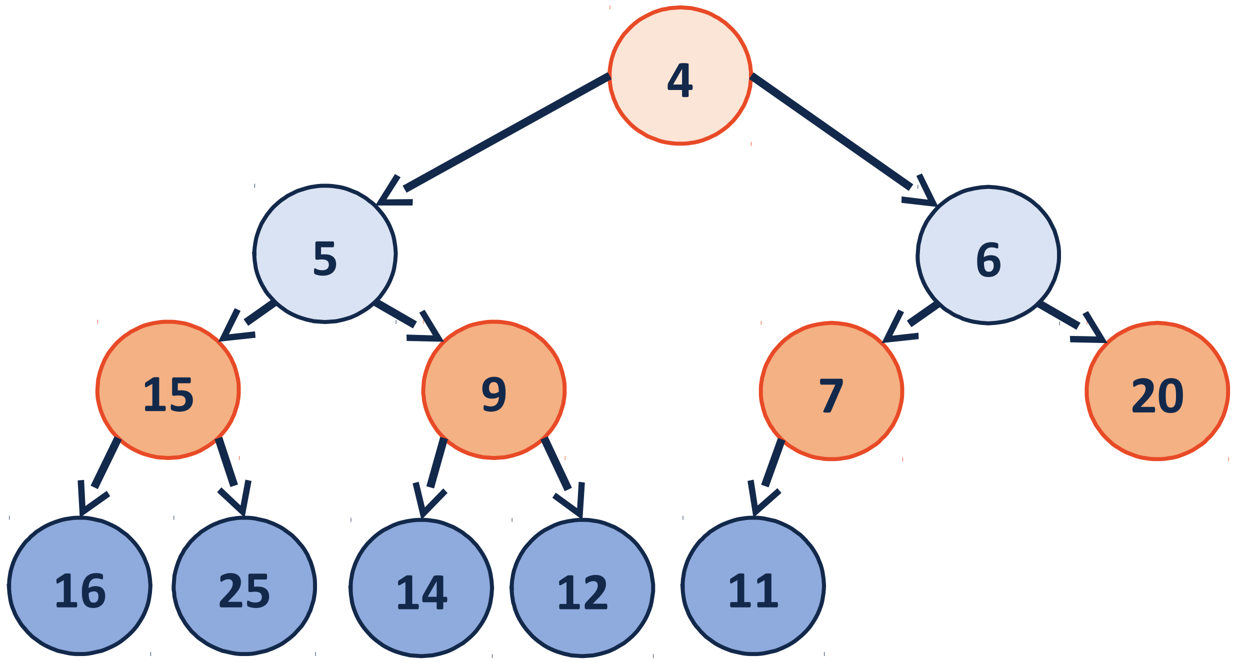
insert

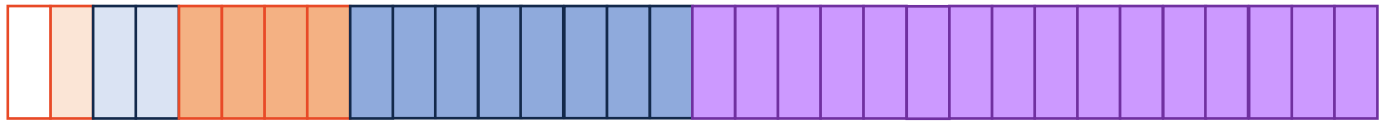
```
template <class T>
void Heap<T>::_insert(const T & key) {
  // Check to ensure there's space to insert an element
  // ...if not, grow the array
  if ( size_ == capacity_ ) { _growArray(); }

  // 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			

growArray



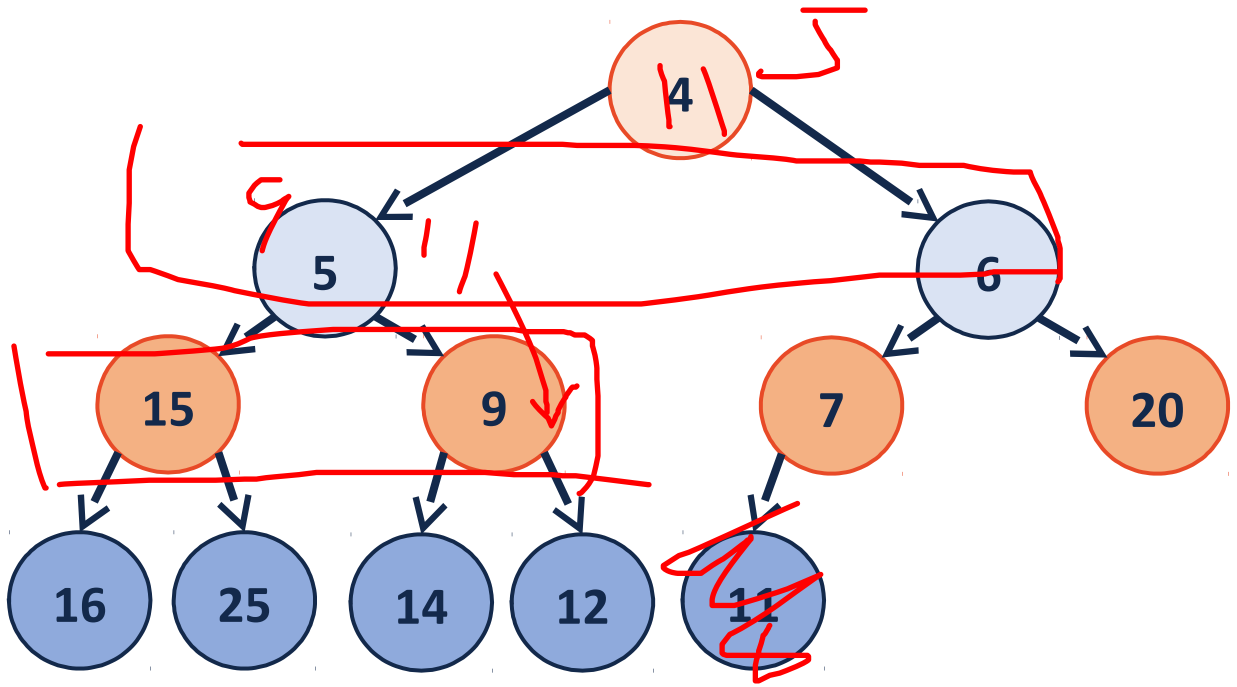


insert- heapifyUp

```cpp
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);
}
```

```cpp
template <class T>
void Heap<T>::_heapifyUp( int index ) {
  if ( index > 1 ) {
    if ( item_[index] < item_[ parent(index) ] ) {
      std::swap( item_[index], item_[ parent(index) ] );
      _heapifyUp( parent (index) );
    }
  }
}
```

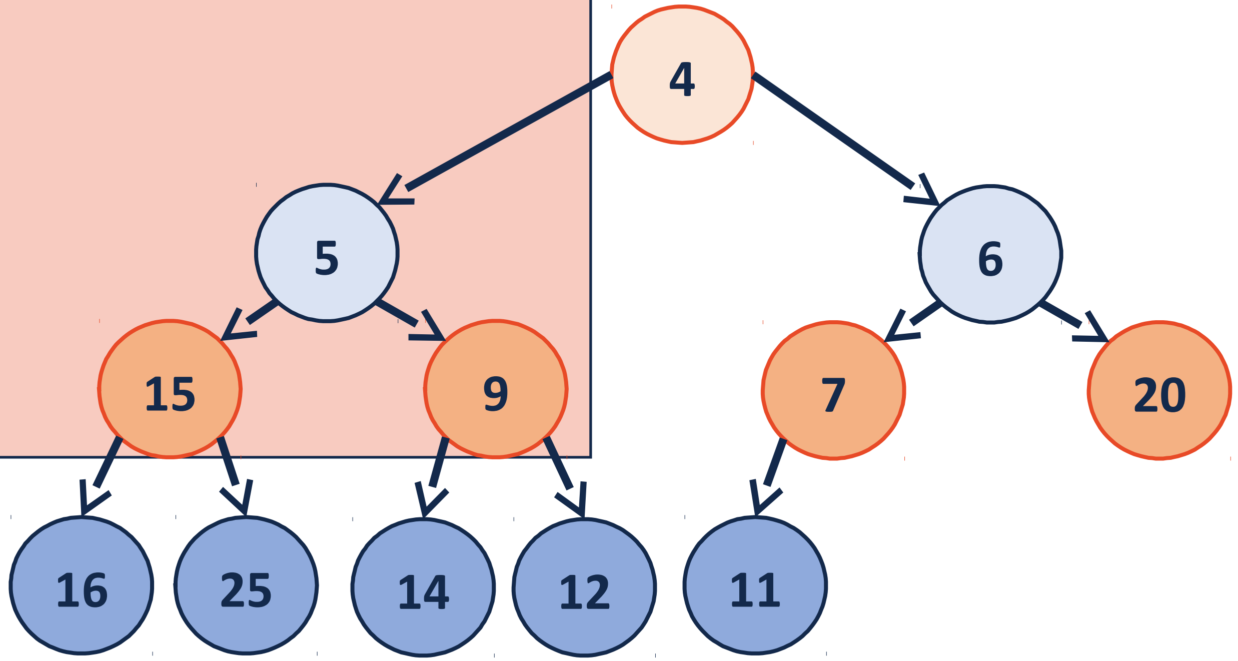
removeMin



	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();

  // 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();

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

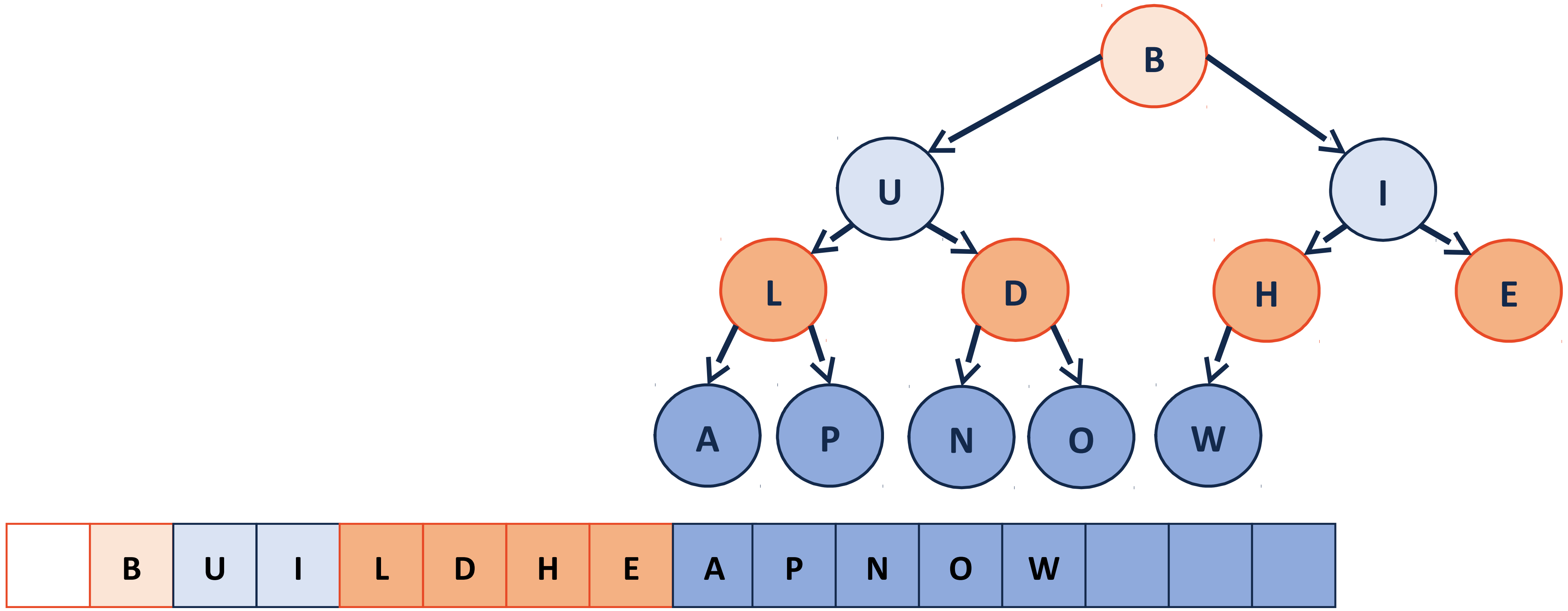
```
template <class T>
void Heap<T>::_heapifyDown(int index) {
  if ( !_isLeaf(index) ) {
    T minChildIndex = _minChild(index);
    if ( item_[index] ___ item_[minChildIndex] ) {
       std::swap( item_[index], item_[minChildIndex] );
       _heapifyDown( ________________ );
    }
  }
}
```

CS 400

Heap – buildHeap

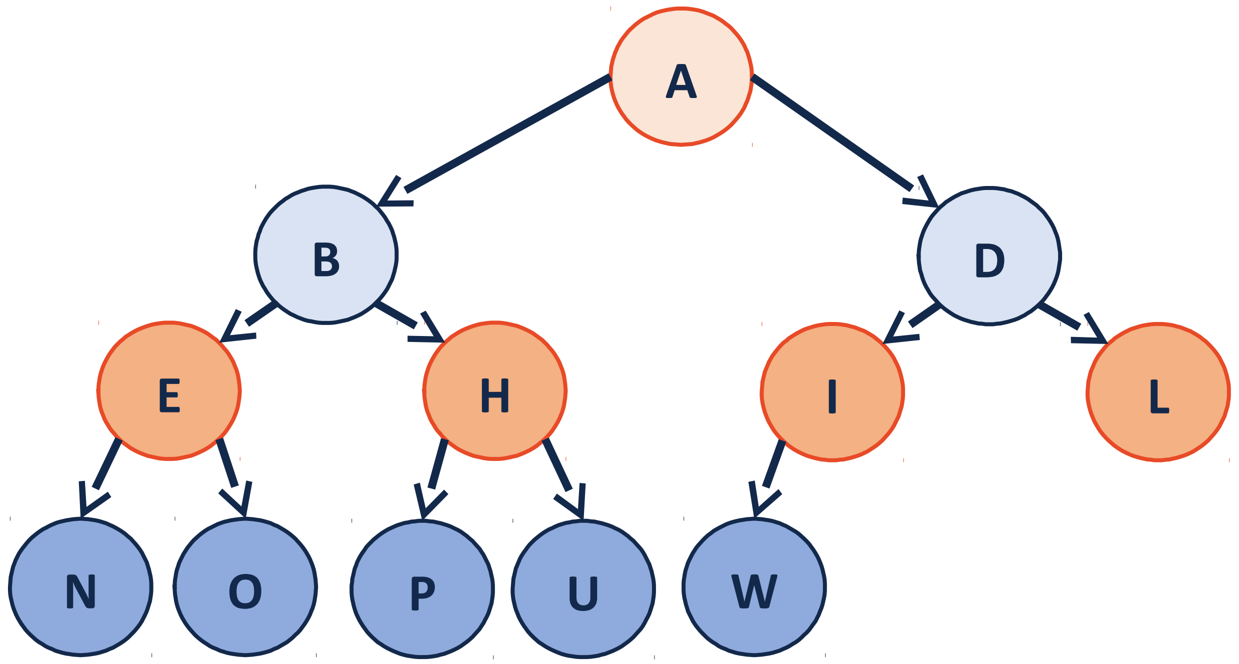
ID: 10-03

buildHeap



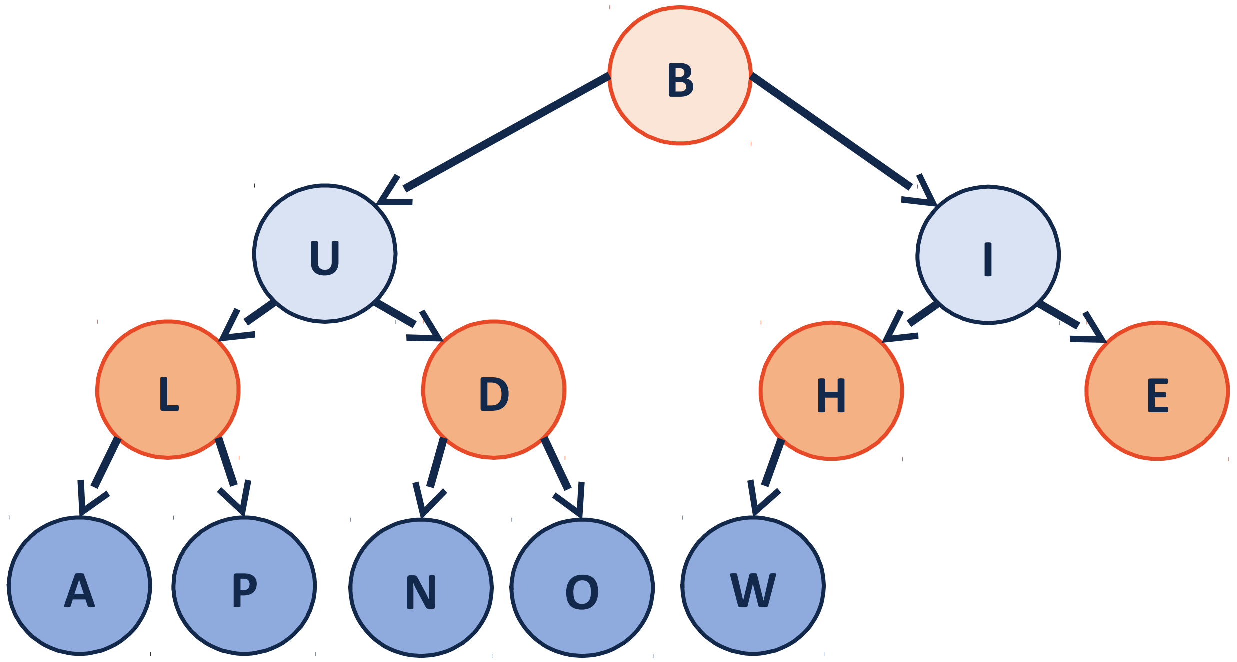
buildHeap – sorted array

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



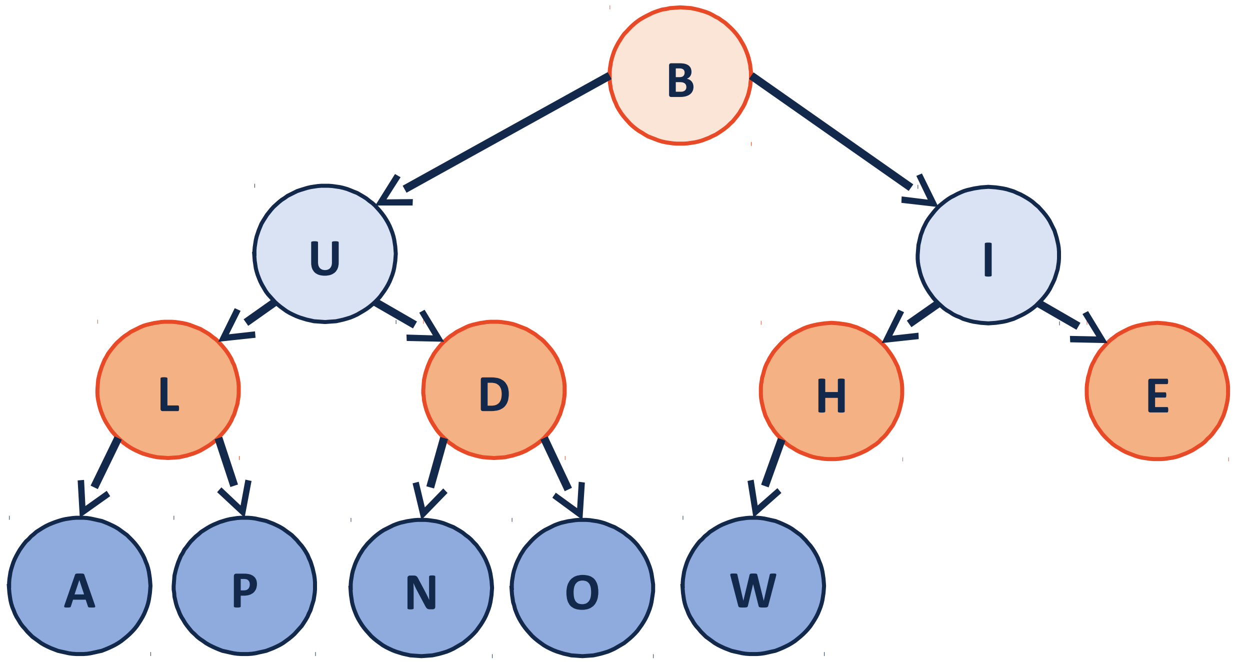
	A	B	D	E	H	I	L	N	O	P	U	W			

buildHeap - heapifyUp



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

buildHeap - heapifyDown



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

buildHeap

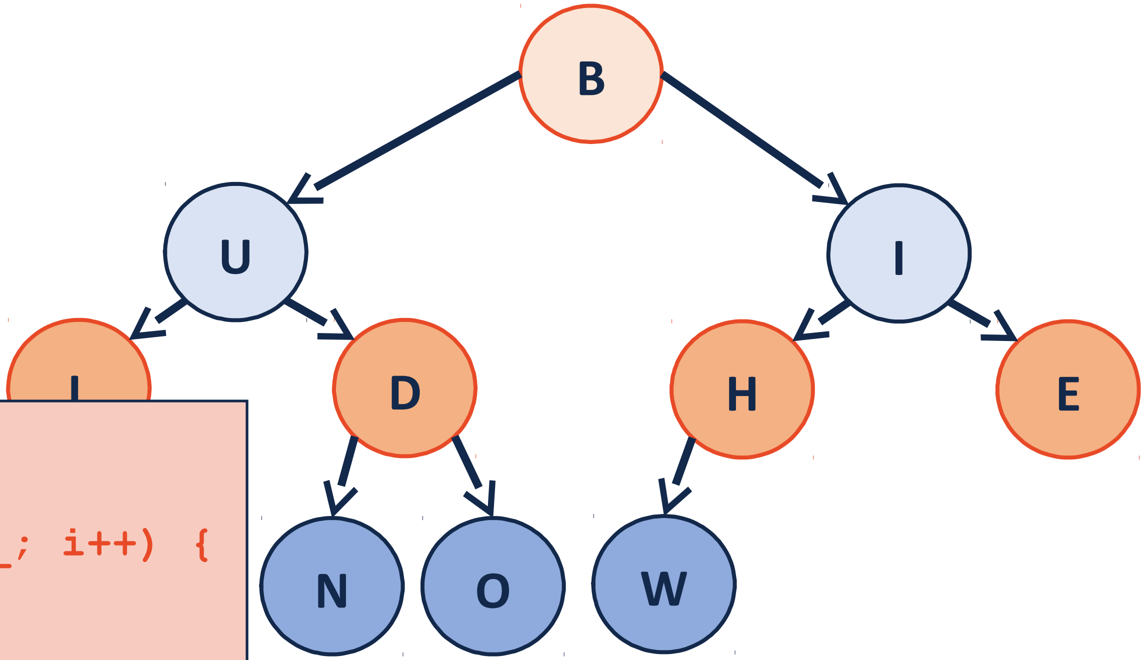
1. Sort the array – it's a heap!

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);
  }
}
```



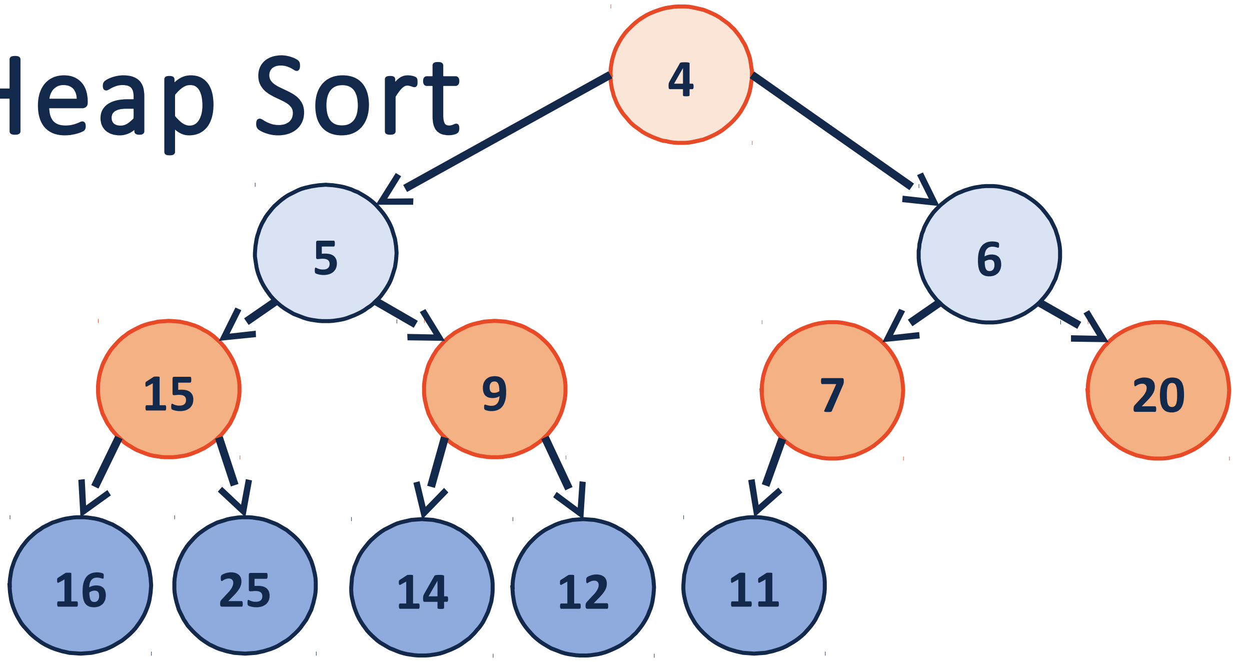
	B	U	I	L	D	H	E	A	P	N	O	W			

CS 400

Heap – Runtime Analysis

ID: 10-04

Heap Sort



1. built heap

2. n * removeMin O(log(n))

3. Swap elements to order list (ASC/DESC)

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

Running Time?

n * log(n)

Why do we care about another sort?

it's often going to be very convenient to do this sort entirely in memory