CS 225

Data Structures

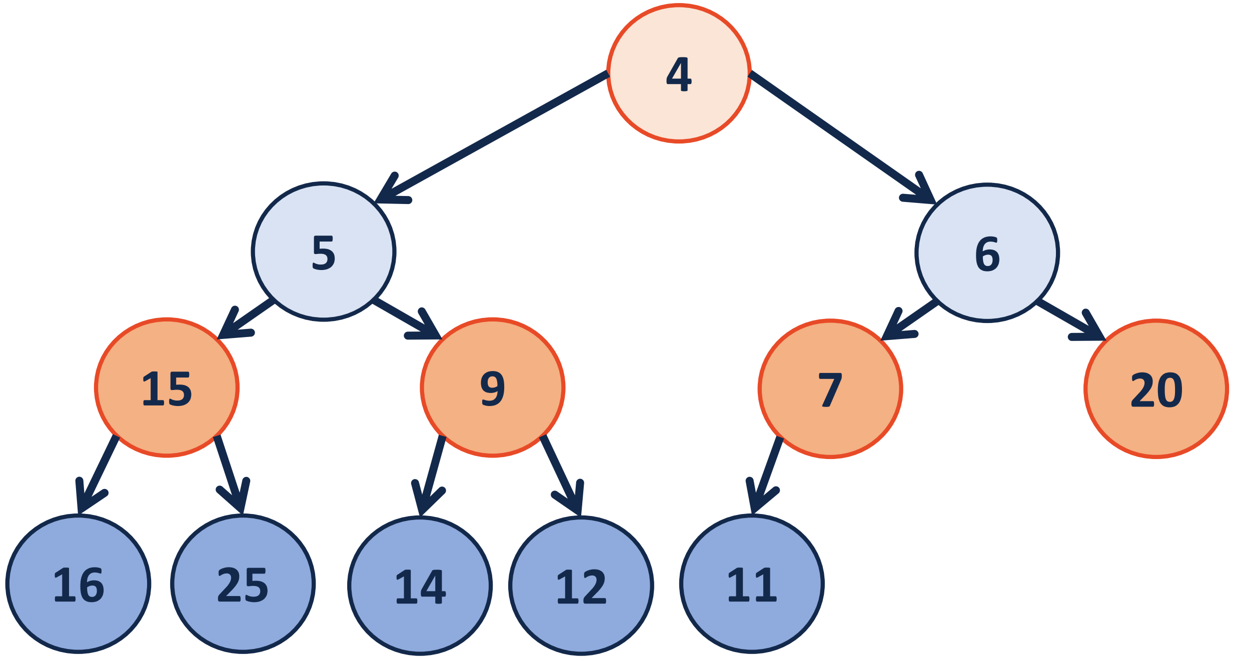
November 2 – Heap Operations

Wade Fagen-Ulmschneider

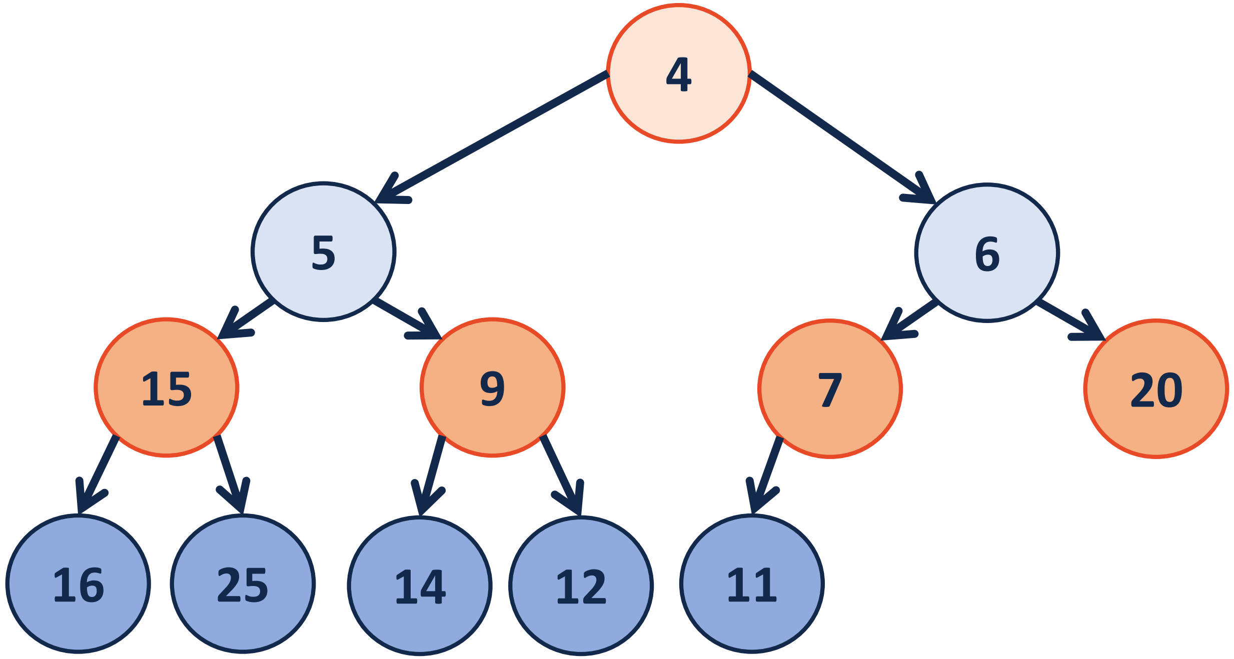
(min)Heap

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.



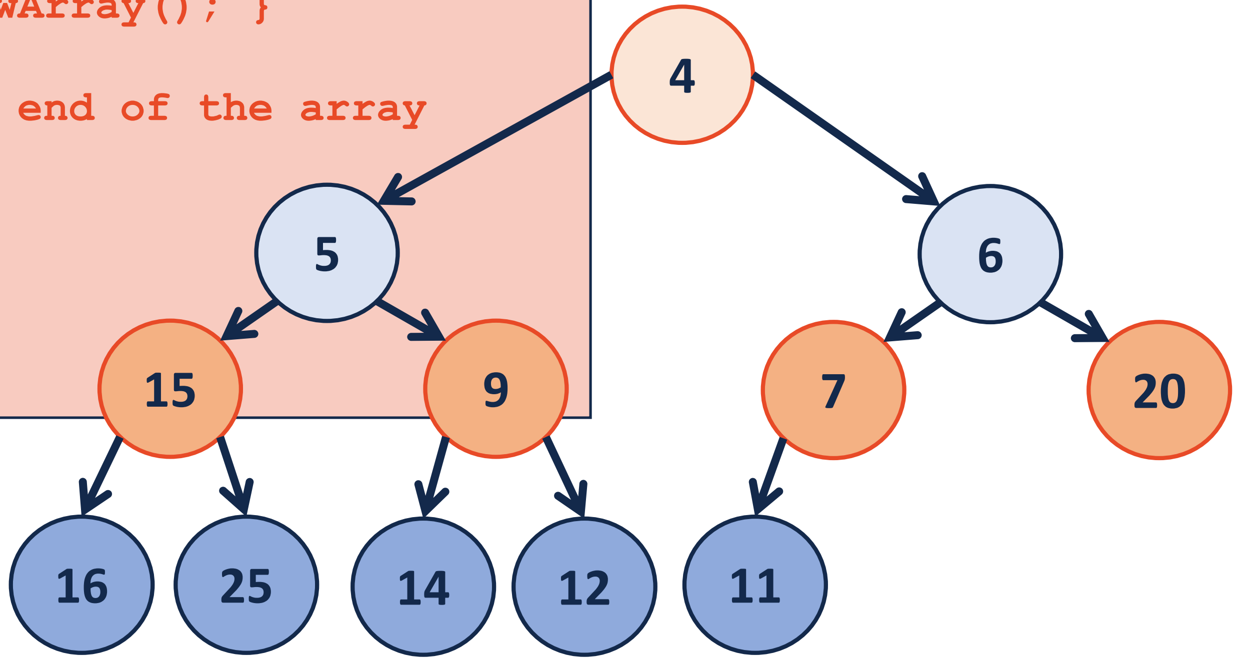
insert



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

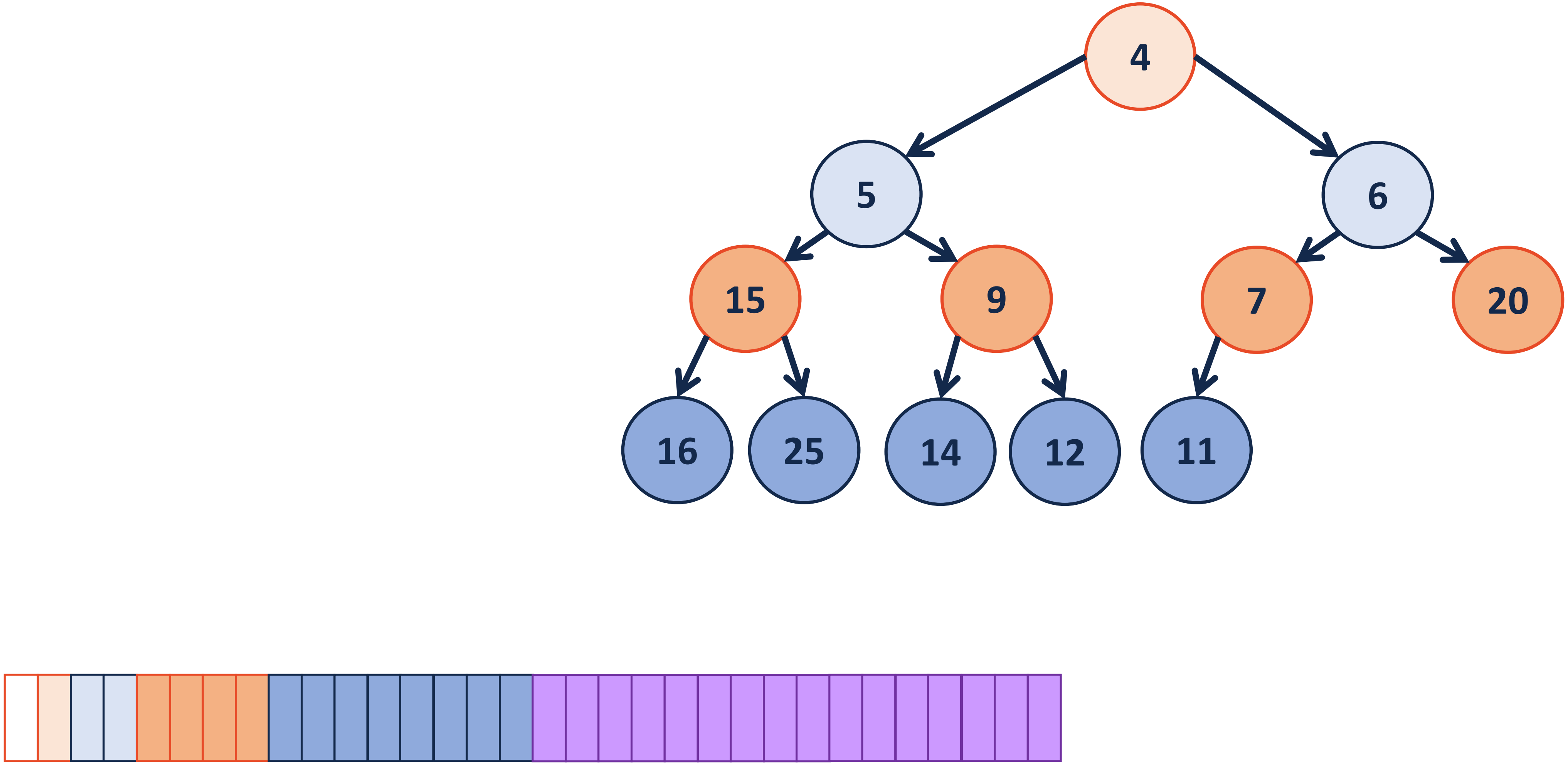
insert

```
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



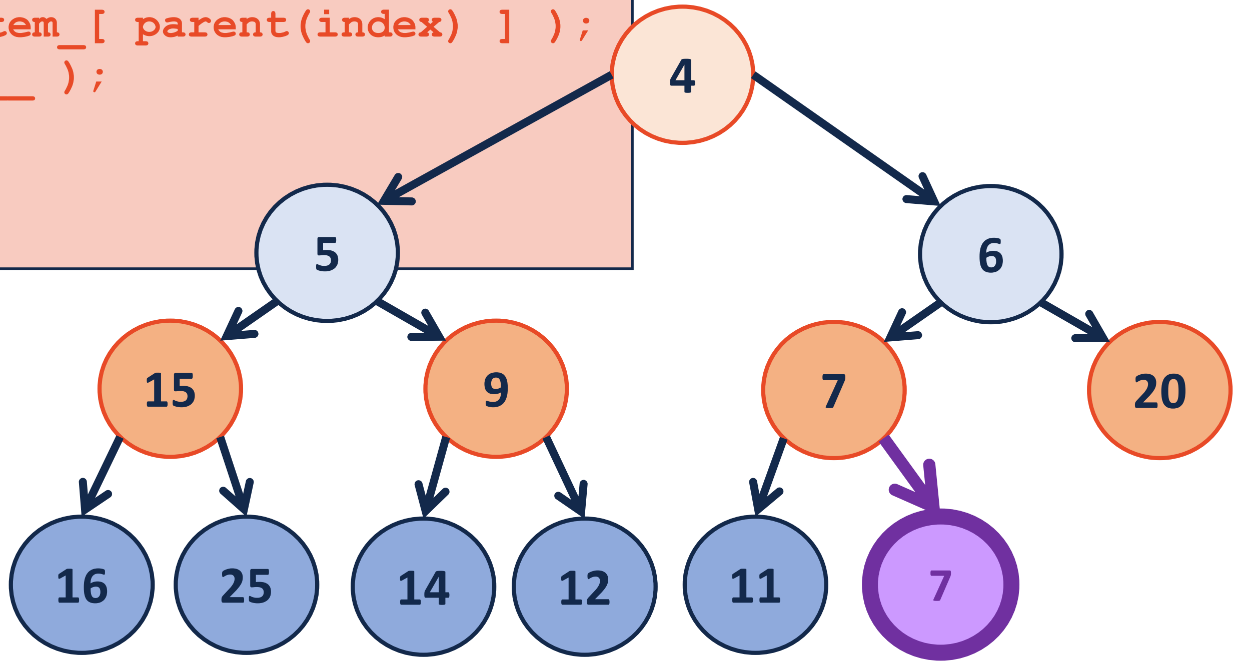
insert - heapifyUp

```
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);
}
```

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

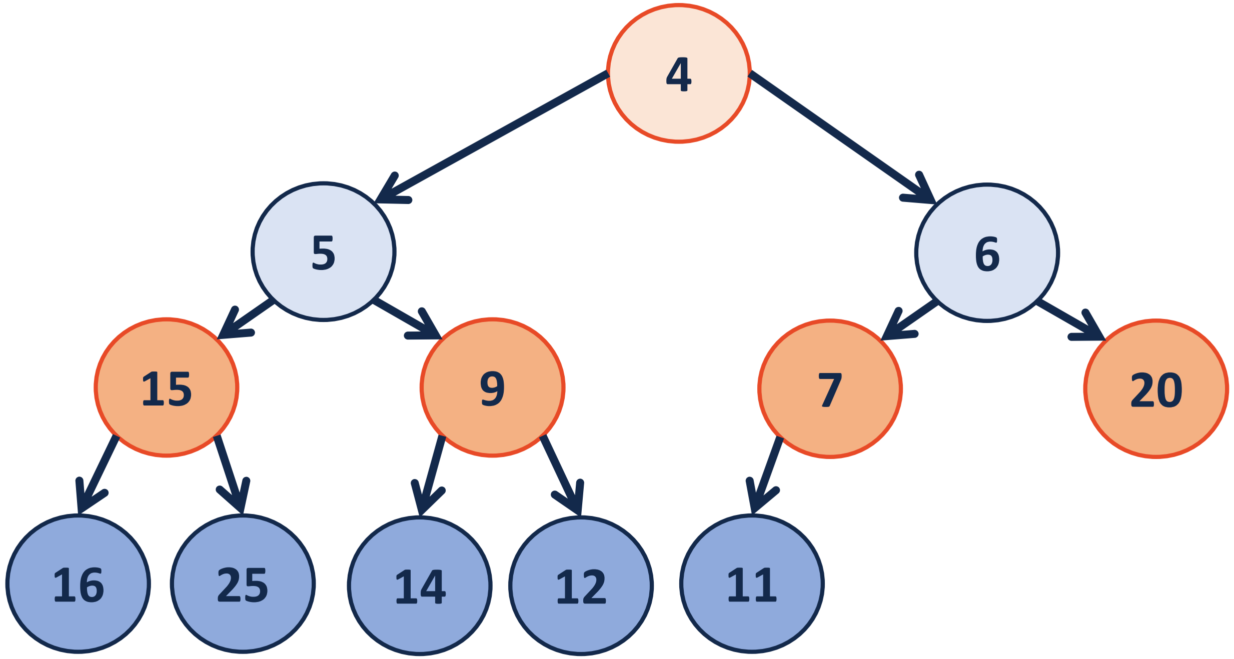
heapifyUp

```
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			

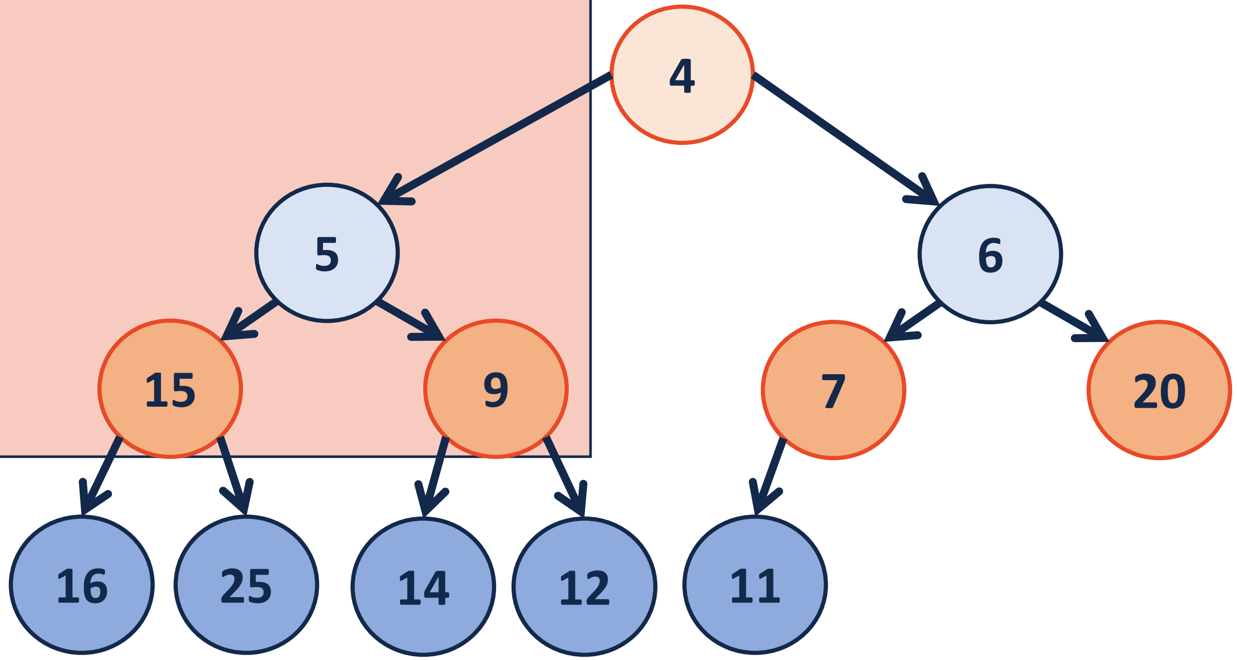
removeMin



	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();

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



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

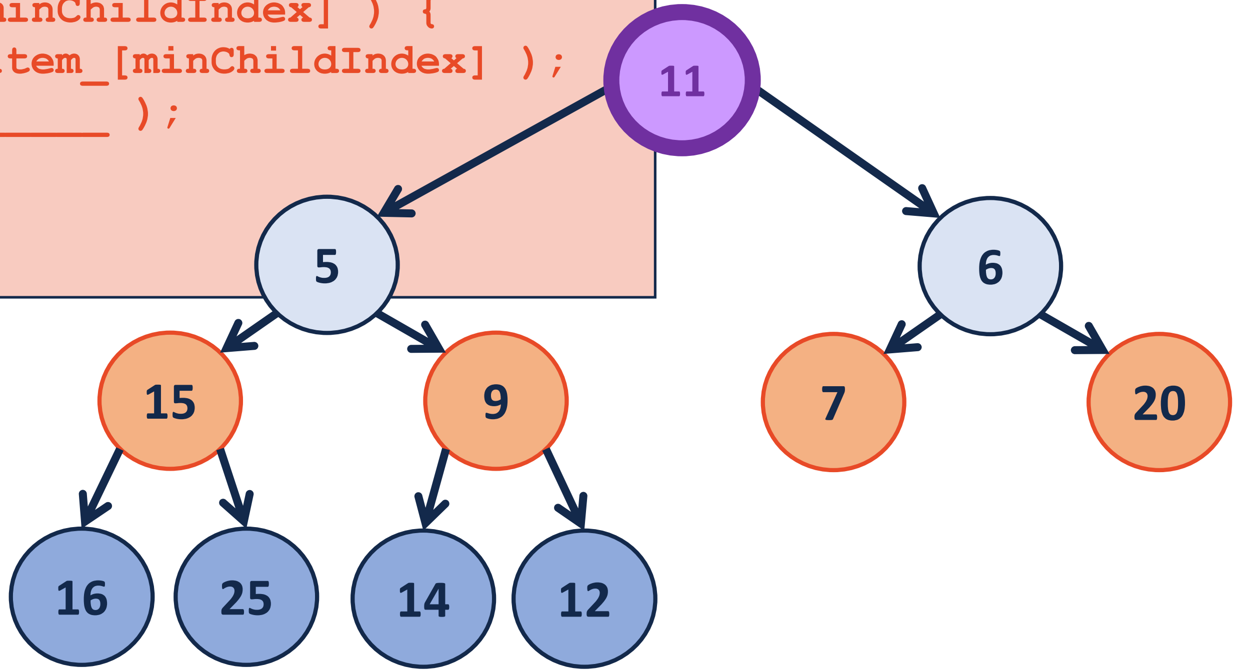
removeMin - heapifyDown

```
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( ________________ );
    }
  }
}
```

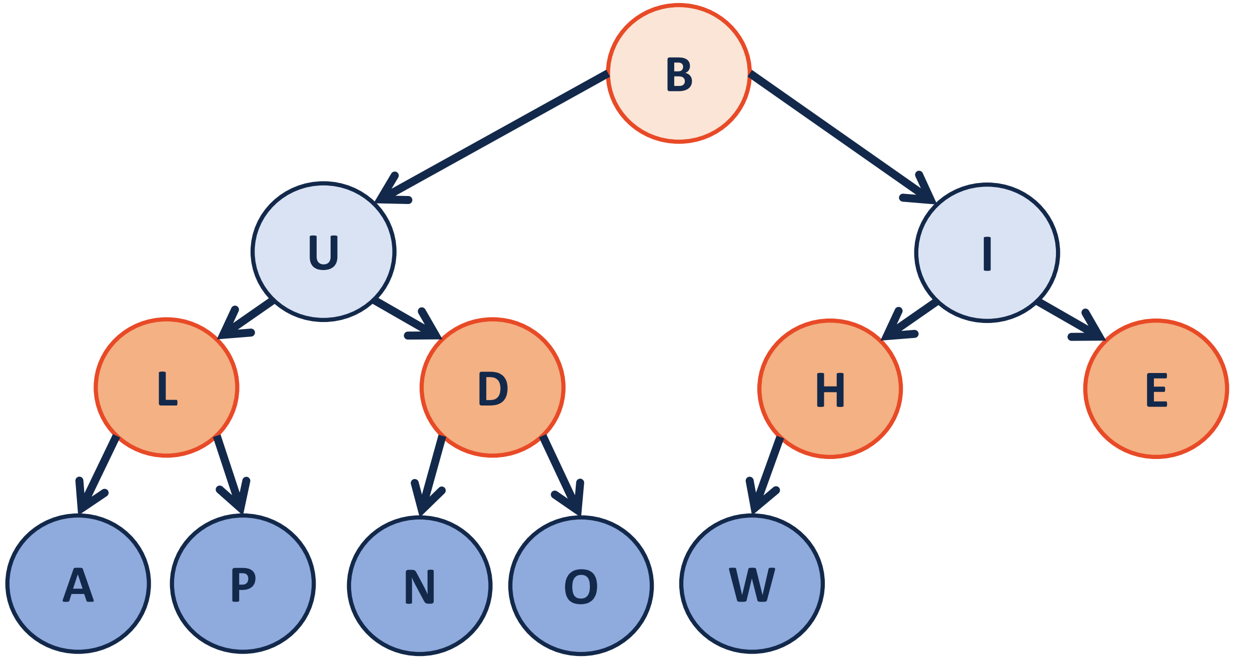
removeMin

```
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( _________________ );
    }
  }
}
```



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

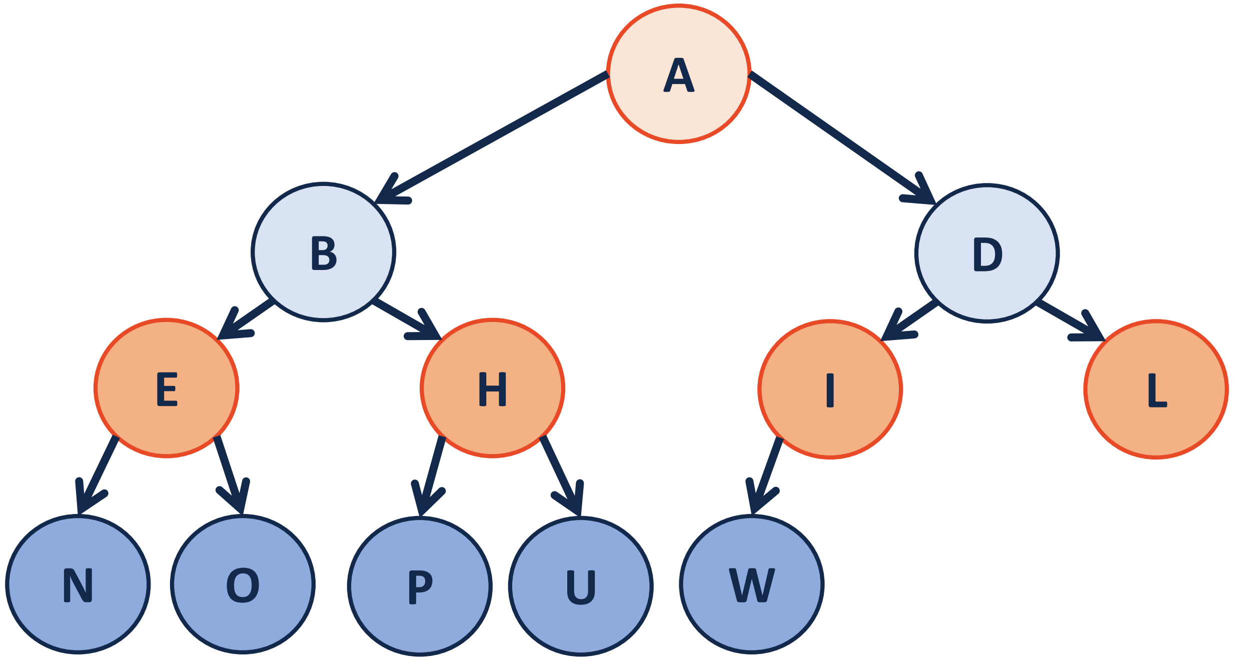
buildHeap



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

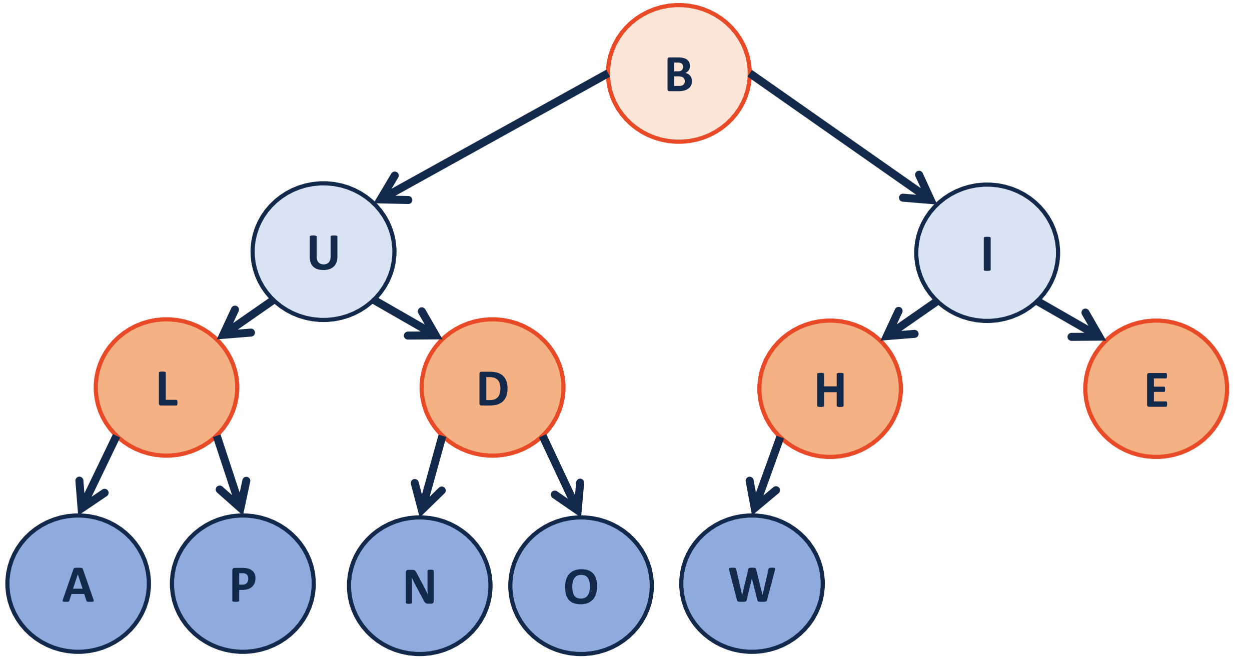
buildHeap – sorted array

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



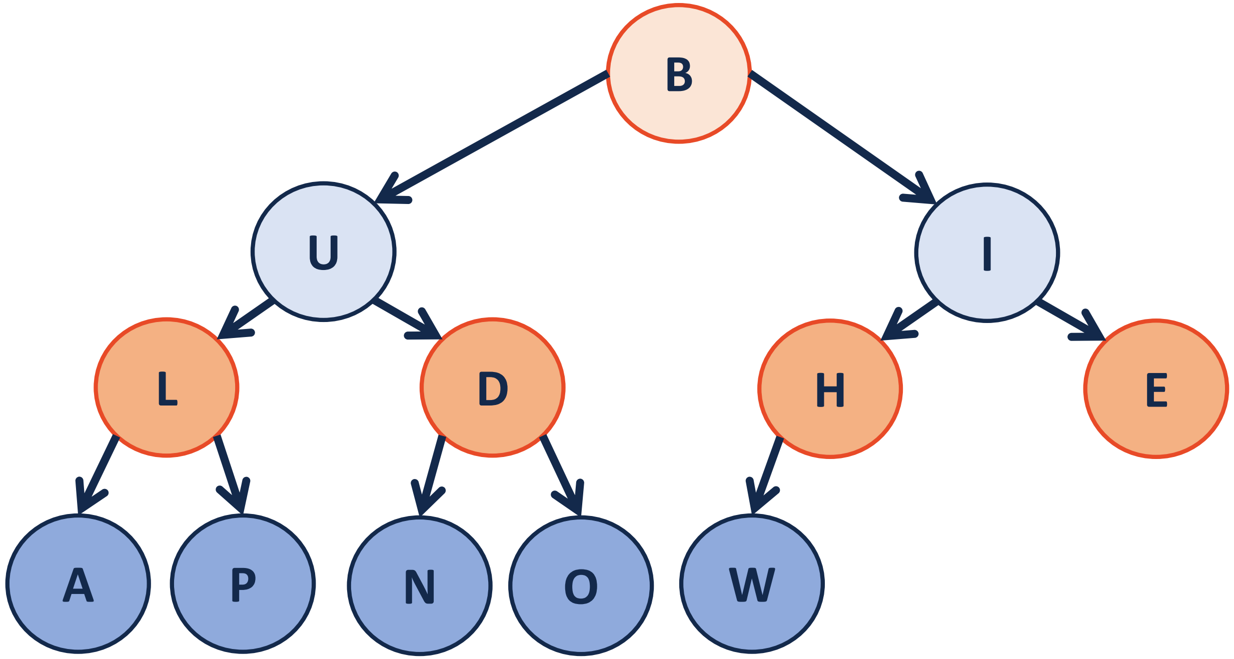
	A	B	D	E	H	I	L	N	O	P	U	W			

buildHeap - heapifyUp



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

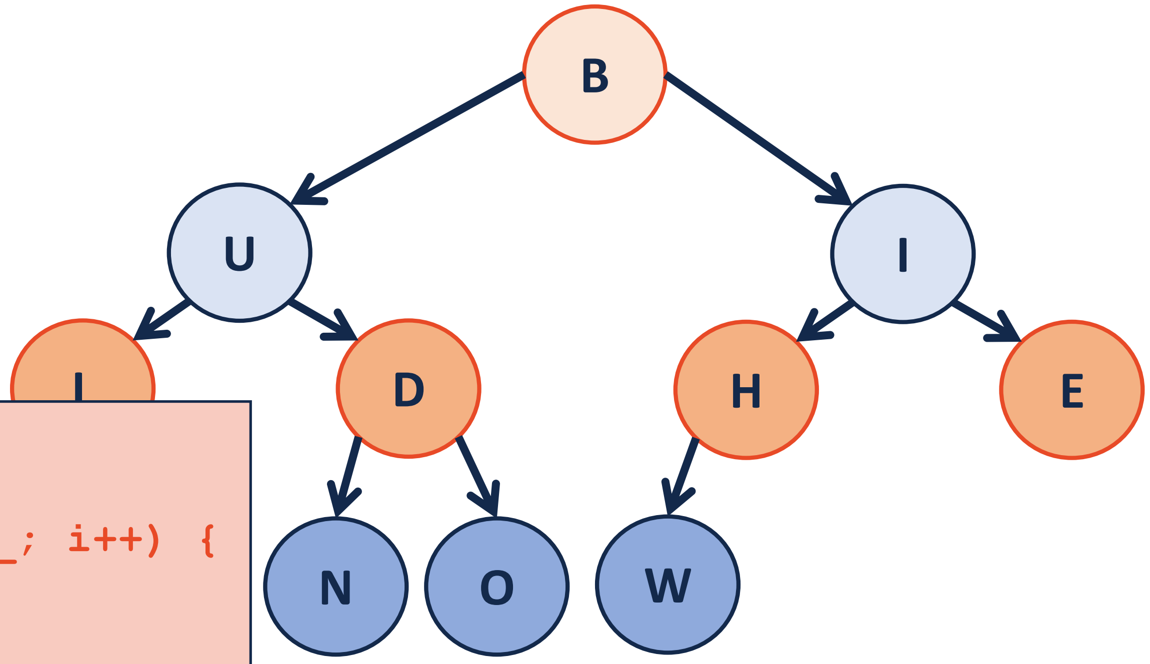
buildHeap - heapifyDown



	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			

Proving buildHeap Running Time

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

Strategy:

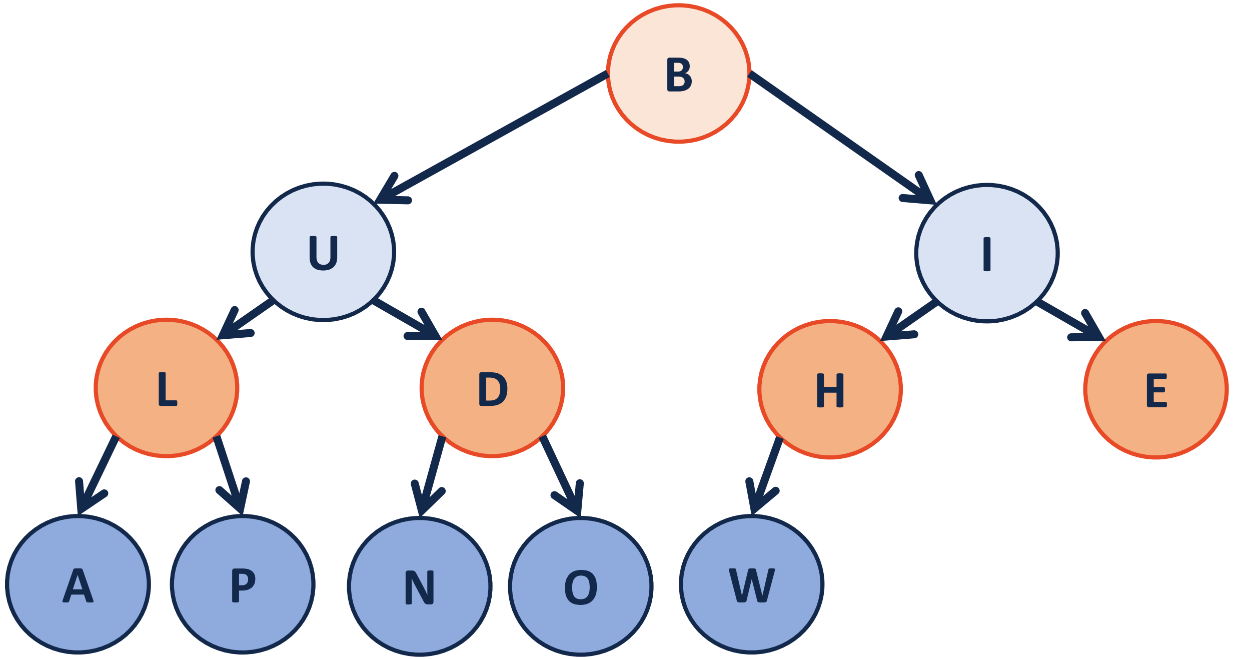
-

-

-

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

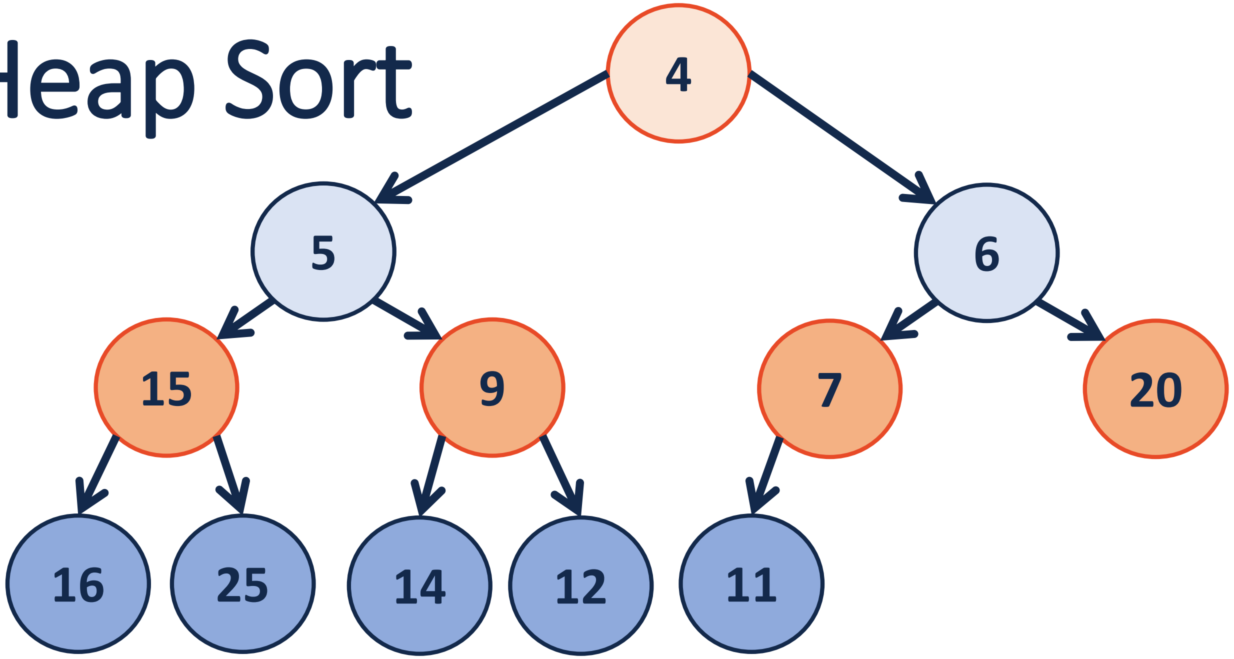
From S(h) to RunningTime(n):

S(h):

Since h ≤ lg(n):

RunningTime(n) ≤

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:

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