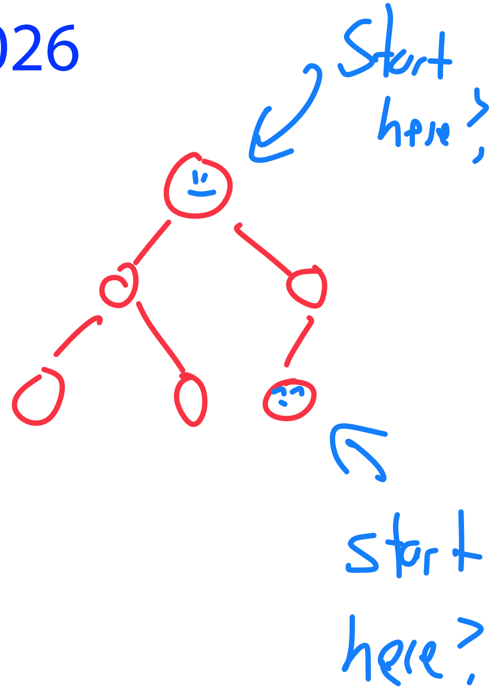
Data Structures

Heaps Analysis

CS 225
Brad Solomon

March 11, 2026



UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN
Department of Computer Science

HACK TO THE FUTURE

PRESENTED BY PRODUCT SPACE UIUC

STARRING YOU

REGISTER NOW*

April 03, 2026 -- April 05 2026

Siebel Center for Design

Questions,comments or concerns? Email productspaceui@gmail.com

Spring Break Logistics

Nothing is due over spring break

Spring break doesn’t count as a ‘week’ for assignments

No office hours over spring break

(Still a lab tomorrow — its due the following Sunday)

Exam 3 (3/23 — 3/25) → Friday will be review session

Autograded MC and one coding question

Manually graded short answer prompt

Practice exam on PL → Bonus video out soon!

Topics covered can be found on website

Registration started March 5

https://courses.engr.illinois.edu/cs225/exams/

Learning Objectives

Review the heap data structure

Discuss heap ADT implementations

Prove the runtime of the heap

(min)Heap

in an array

By storing as a complete tree, can avoid using pointers at all!

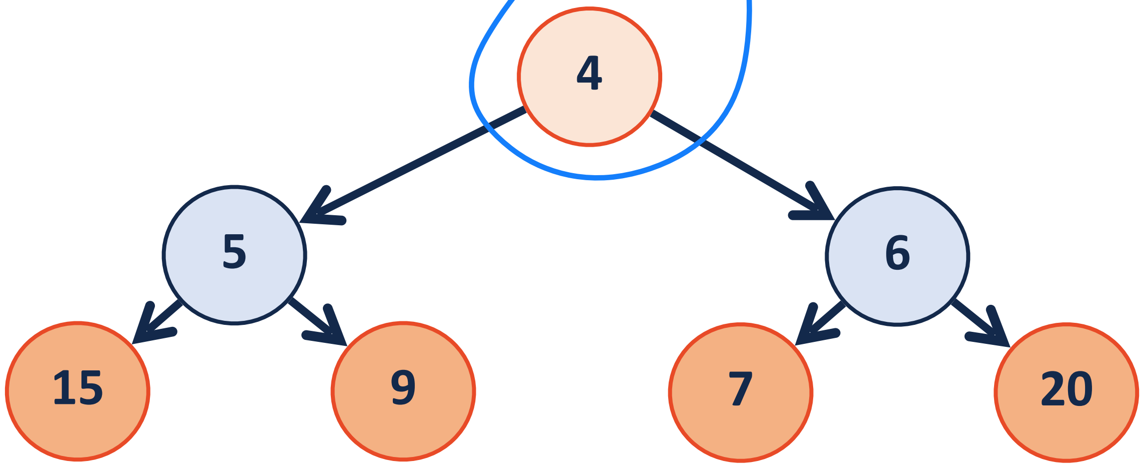
If index starts at 1:

`leftChild(i): 2i`

`rightChild(i): 2i+1`

`parent(i): floor(i/2)` * !! ☺

access min item



	4	5	6	15	9	7	20
0	1	2	3	4	5	6	7

(min)Heap

By storing as a complete tree, can avoid using pointers at all!

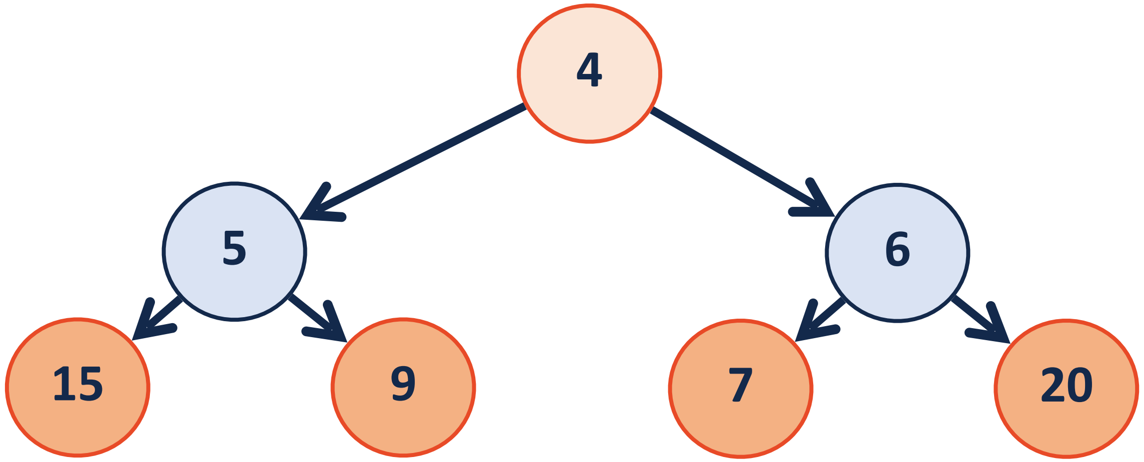
$2^{h+1} - 1$ (max) total items

If Index starts at 0:

`leftChild(i): 2i+1`

`rightChild(i): 2(i+1)`

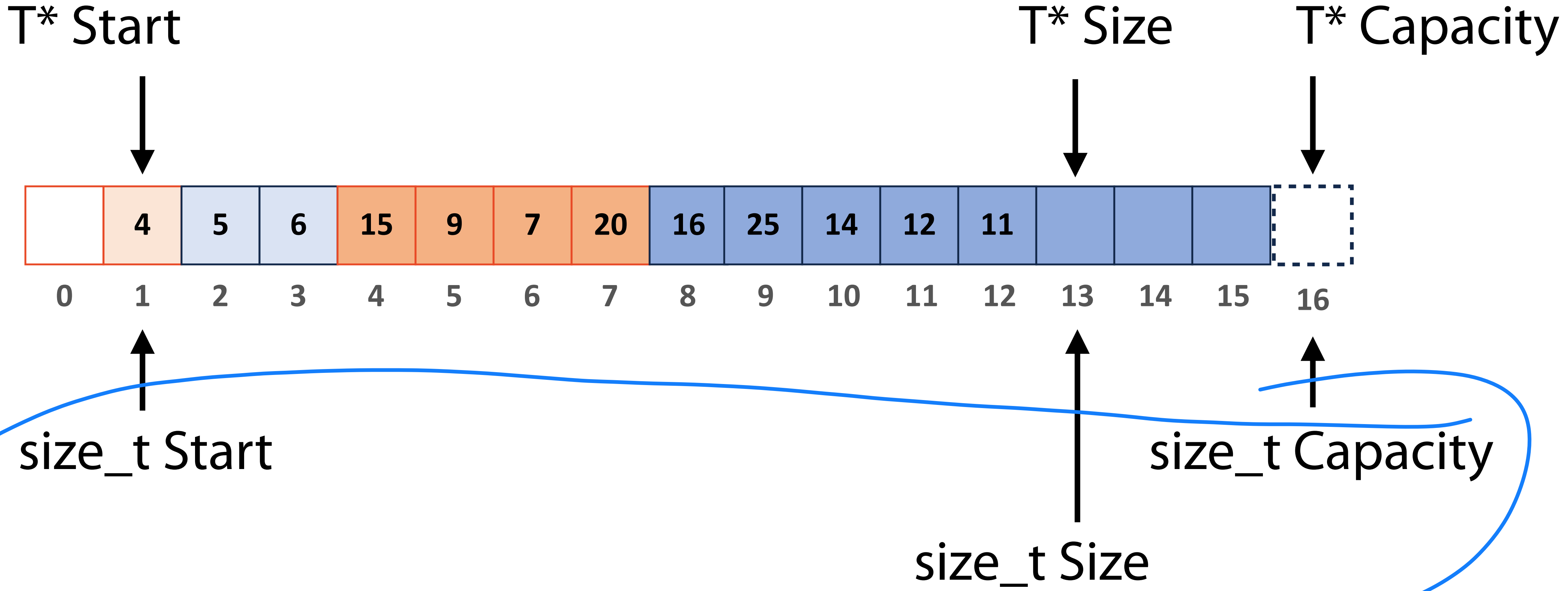
`parent(i): floor((i-1)/2)`



0	1	2	3	4	5	6	7
4	5	6	15	9	7	20	

Implementation of heap array

Array List (Pointer implementation)



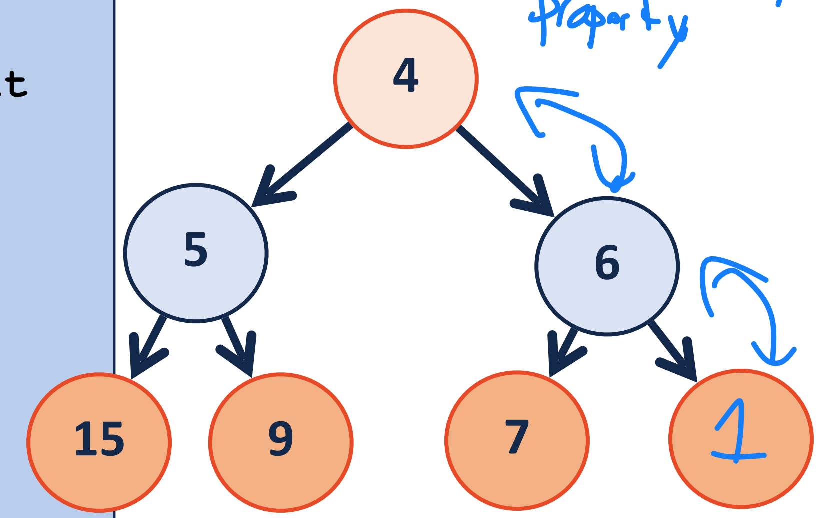
Array List (Index implementation)

insert - heapifyUp

Great example of tradeoffs

1) Insert
2) Restore heap property

```
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_ - 1);
}
```




```
template <class T>
void Heap<T>::_heapifyUp( size_t index ) {

  if ( index > 1 ) {
    if ( item_[index] < item_[ parent(index) ] ) {
       std::swap( item_[index], item_[ parent(index) ] );

       _heapifyUp( parent(index) ); // index / 2;
    }
  }
}
```

	4	5	6	15	9	7	1
0	1	2	3	4	5	6	7

Add at array back

$O(1)$

!!

removeMin

What is the Big O of array remove?

↳ O(n) to remove front item from an array

What else can we do?

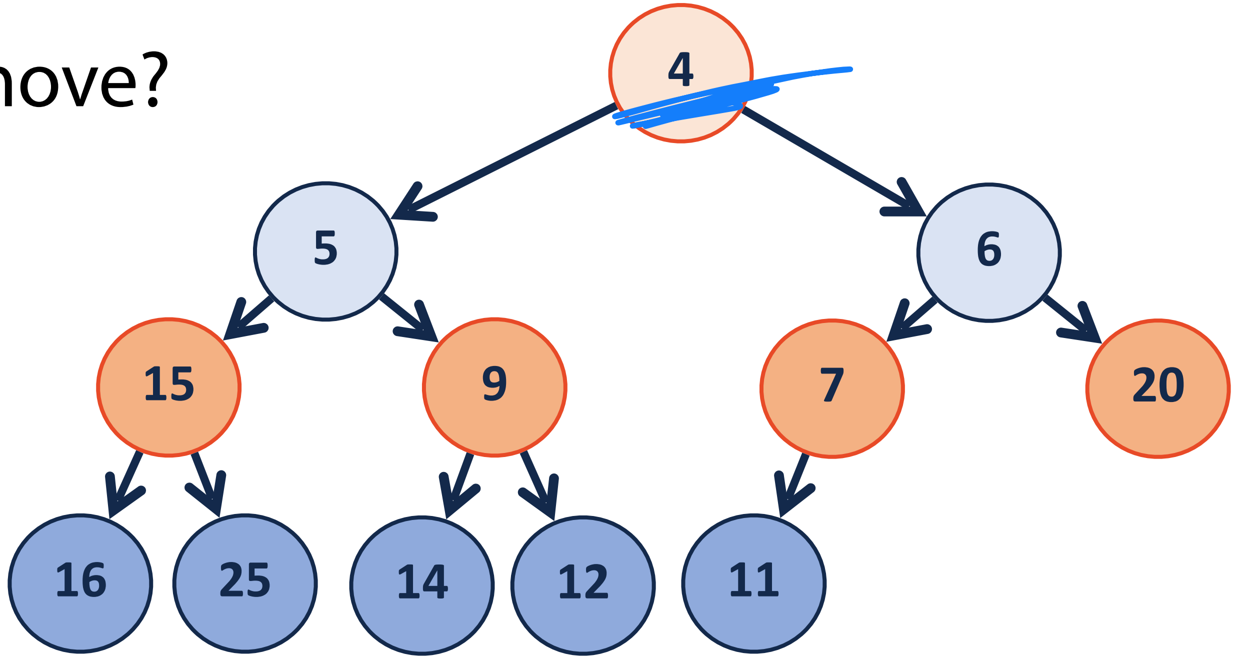
At same time:

↳ Remove min

↳ Replace w/ next smallest item

↳ Preserve complete tree property

Goal: O(log n) to do both things!



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

↑ access this in O(1)

removeMin

1) Swap root w/ the last item in array

↳ Then remove the min value

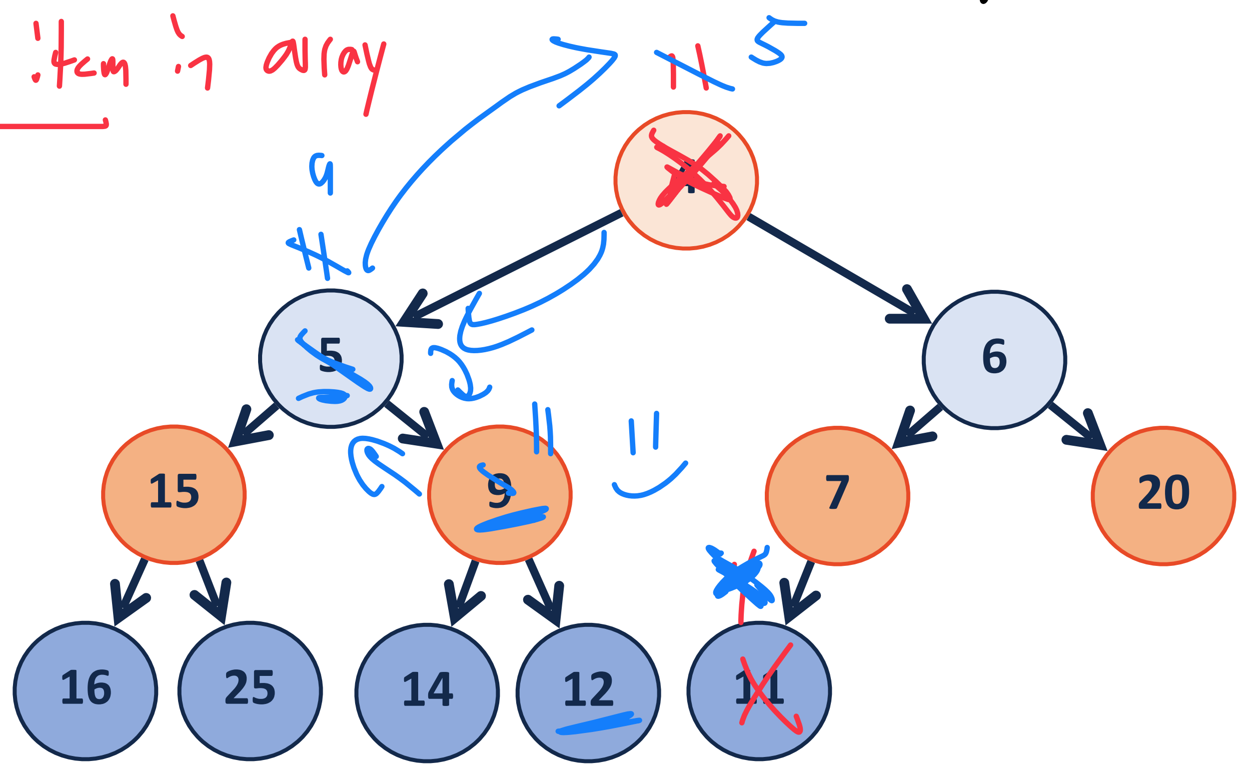
↳ size--

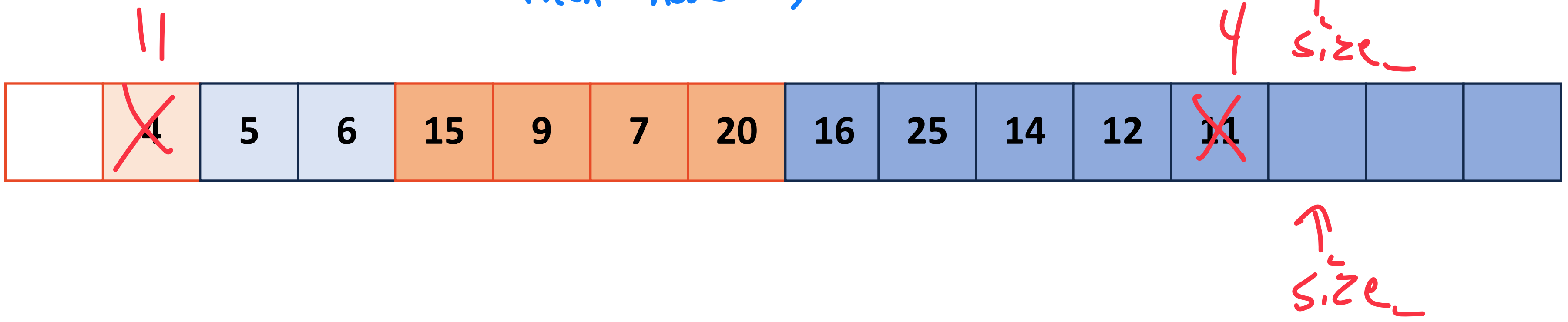
2) Restore heap property

↳ Heapify Down my new root

At each node, swap w/ smallest child (if child smaller than node)

Tradeoffs!





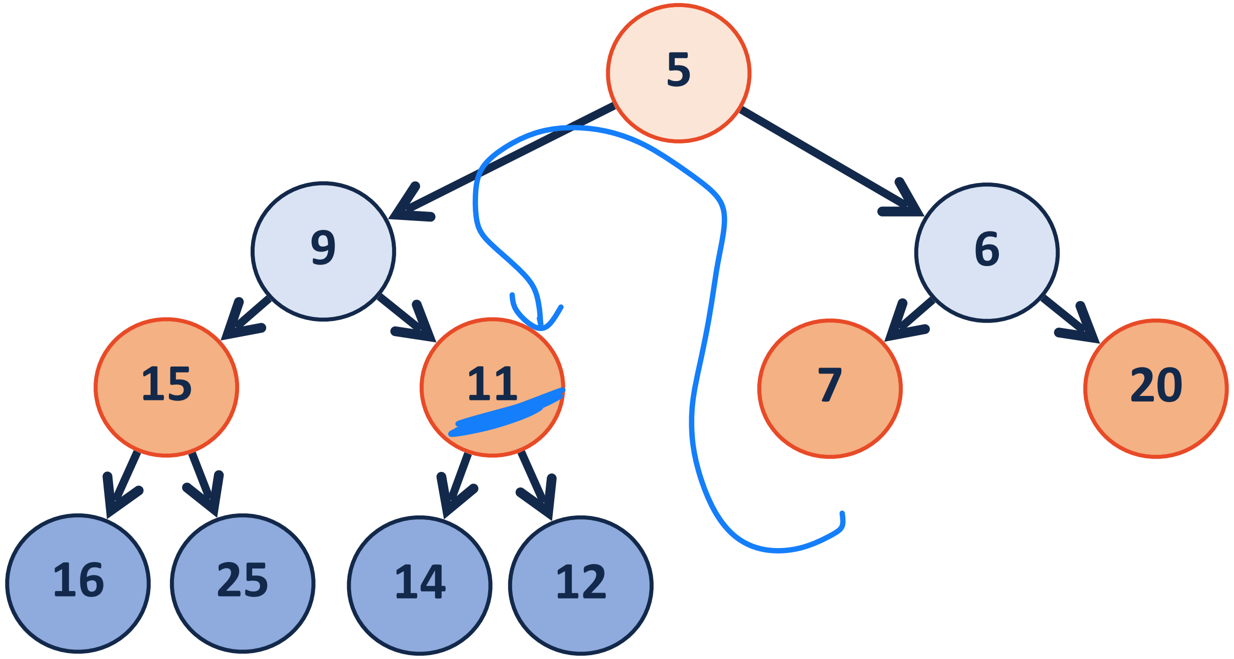
removeMin

1) Swap root with last item

   (and remove)

   (and modify size)

2) HeapifyDown( ) root

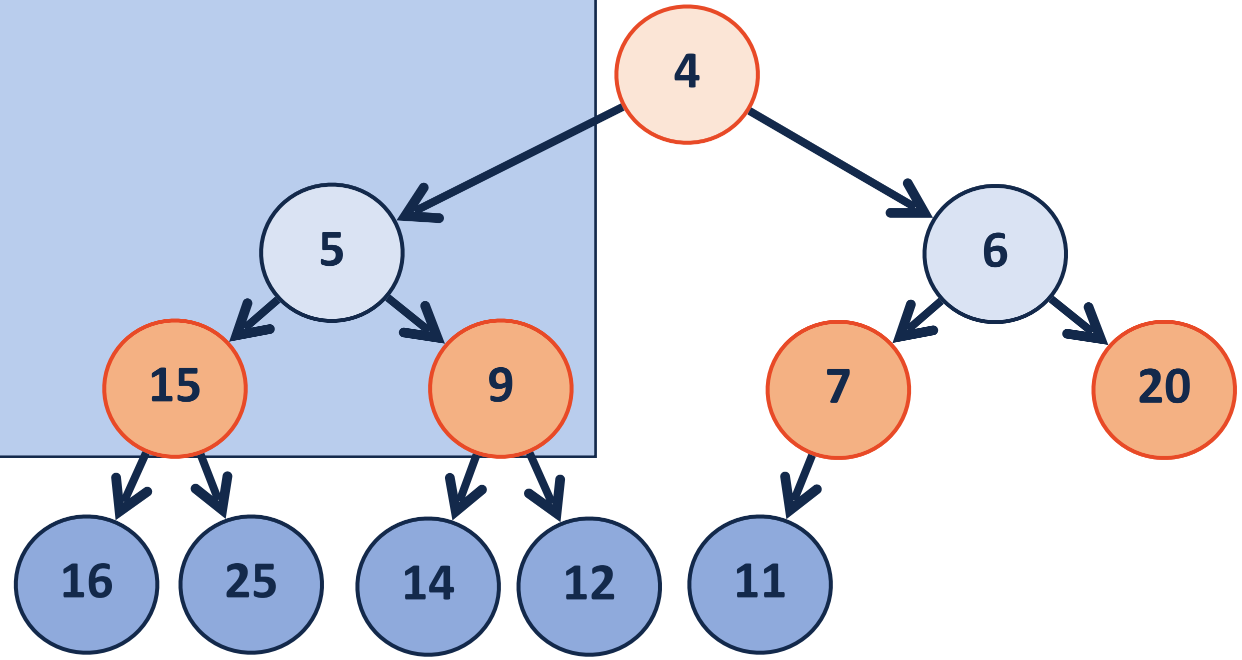


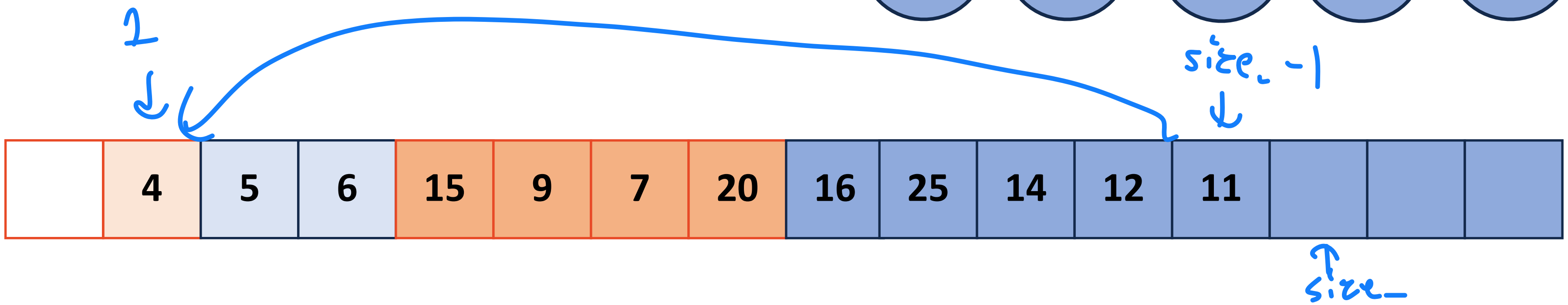
	5	9	6	15	11	7	20	16	25	14	12				

removeMin

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

  // Restore the heap property
  _heapifyDown();

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

root , item





removeMin - heapifyDown

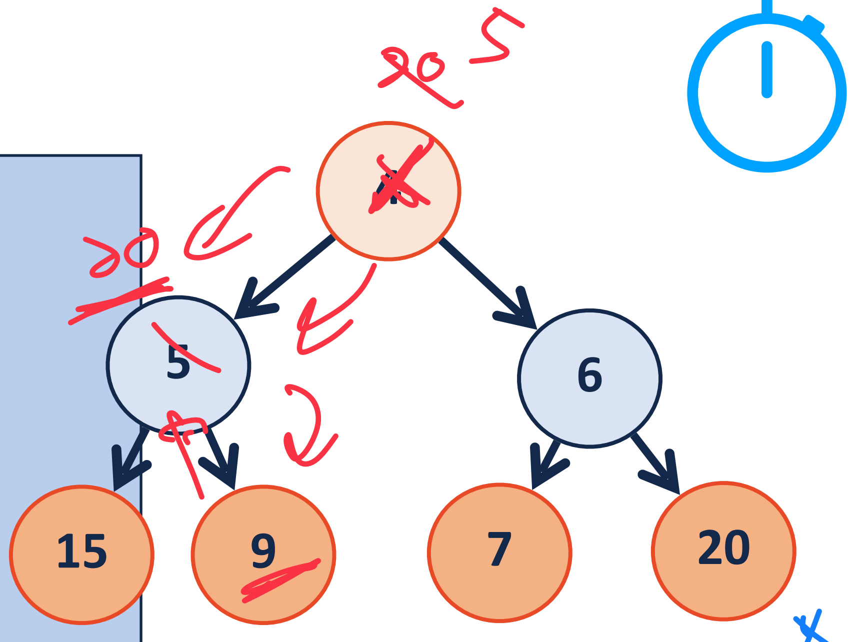
```
template <class T>
T Heap<T>::_removeMin() {
  // Swap with the last value
  T minValue = item_[1];
  item_[1] = item_[size_ - 1];
  size--;

  // Restore the heap property
  _heapifyDown(1);

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

```
template <class T>
void Heap<T>::_heapifyDown(size_t index) {
  if ( !_isLeaf(index) ) {
    size_t minChildIndex = _minChild(index);

    if ( item_[index] ___>___ item_[minChildIndex] ) {
      std::swap( item_[index], item_[minChildIndex] );

      _heapifyDown( __minChildIndex__ );
    }
  }
}
```

← Base case

Handles 1 child case
2 child case

if parent large, then child, swap



0	1	2	3	4	5	6	7
	~~4~~ 20	5	6	15	9	7	~~20~~

buildHeap (minHeap Constructor)

How can I build a minHeap?

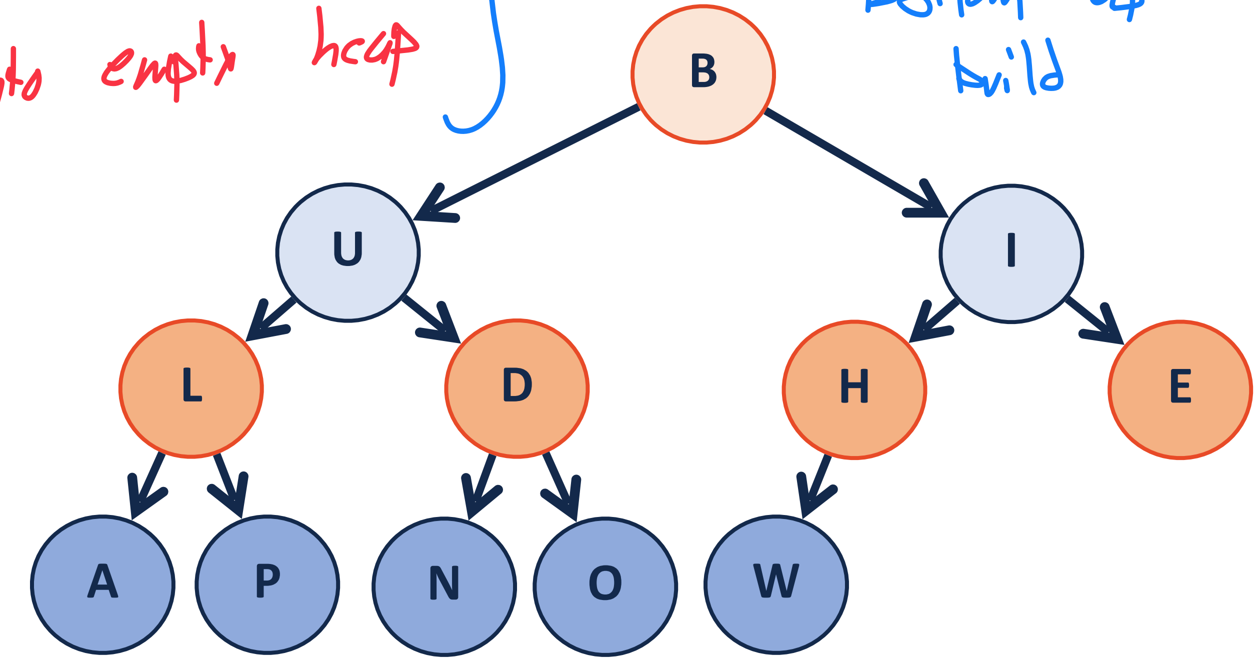
↳ lets chain insert items into empty heap

↳ Heapify up()

like a Btree bottom up build

↳ Sort the input array

↳ Build using Heapify Down()



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

buildHeap – sorted array

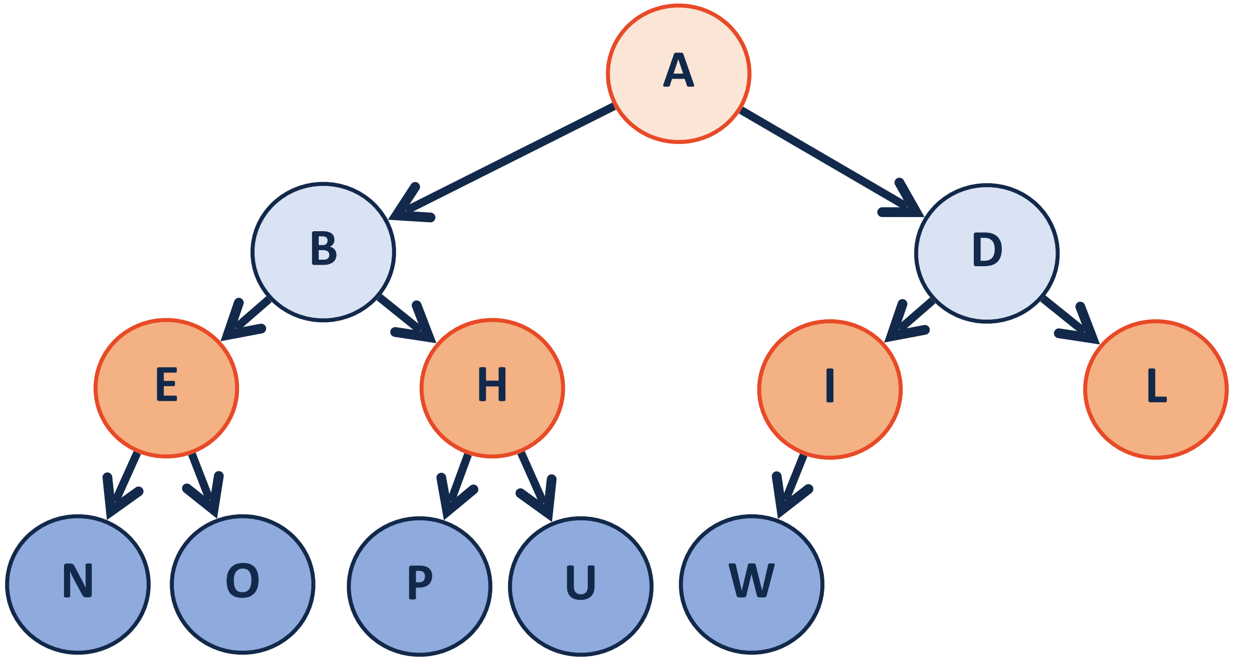
← N initial items

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

Sort Alg

O(n log n)

General optimal sort in place runtime



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

Smallest

every parent smaller than every child

largest

Join Code: 225

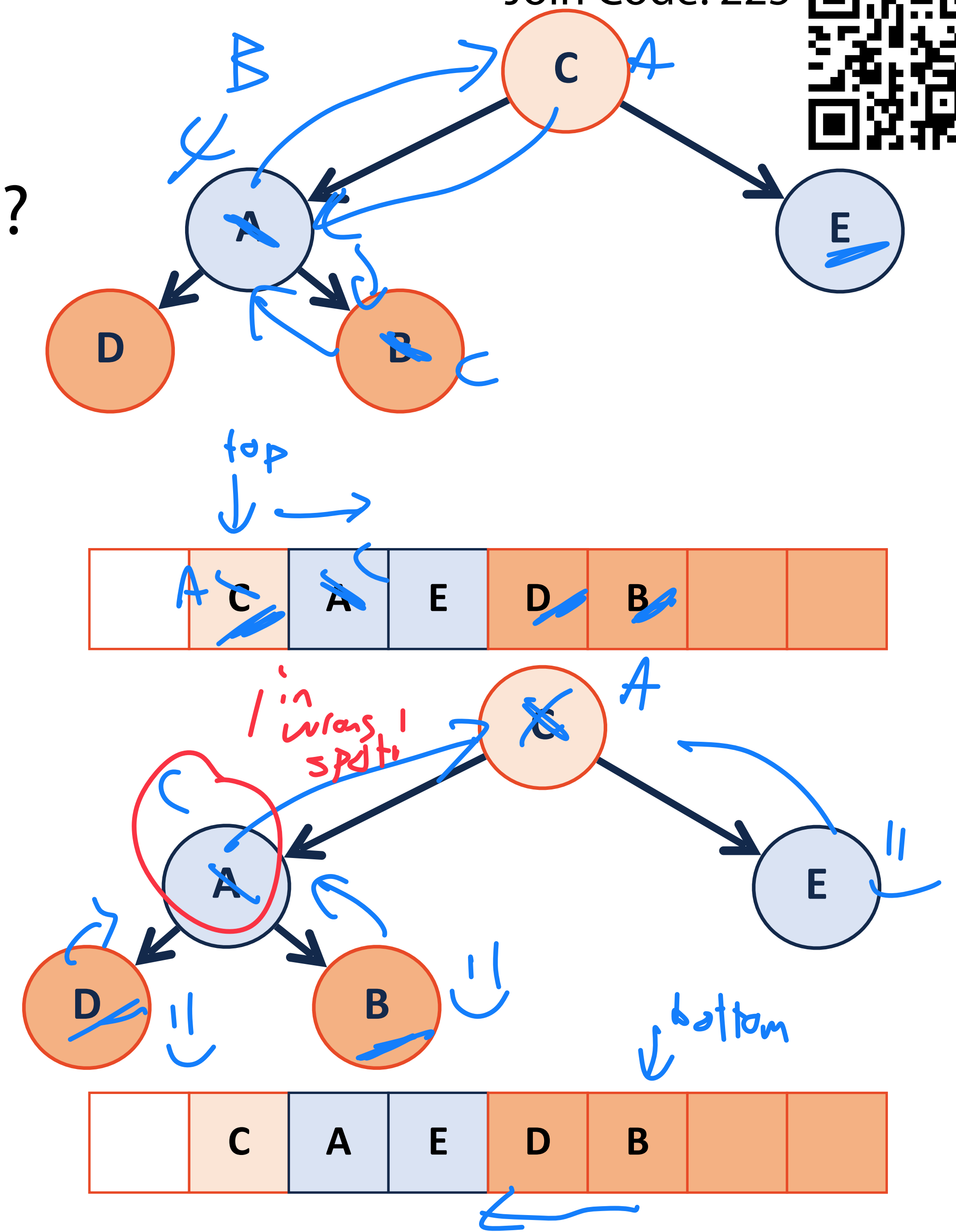
buildHeap - heapifyUp

Do we heapifyUp from top or bottom?

↳ when we heapifyUp we assume that all nodes above us are heap

heapify up

← Must be heap for swap to be correct



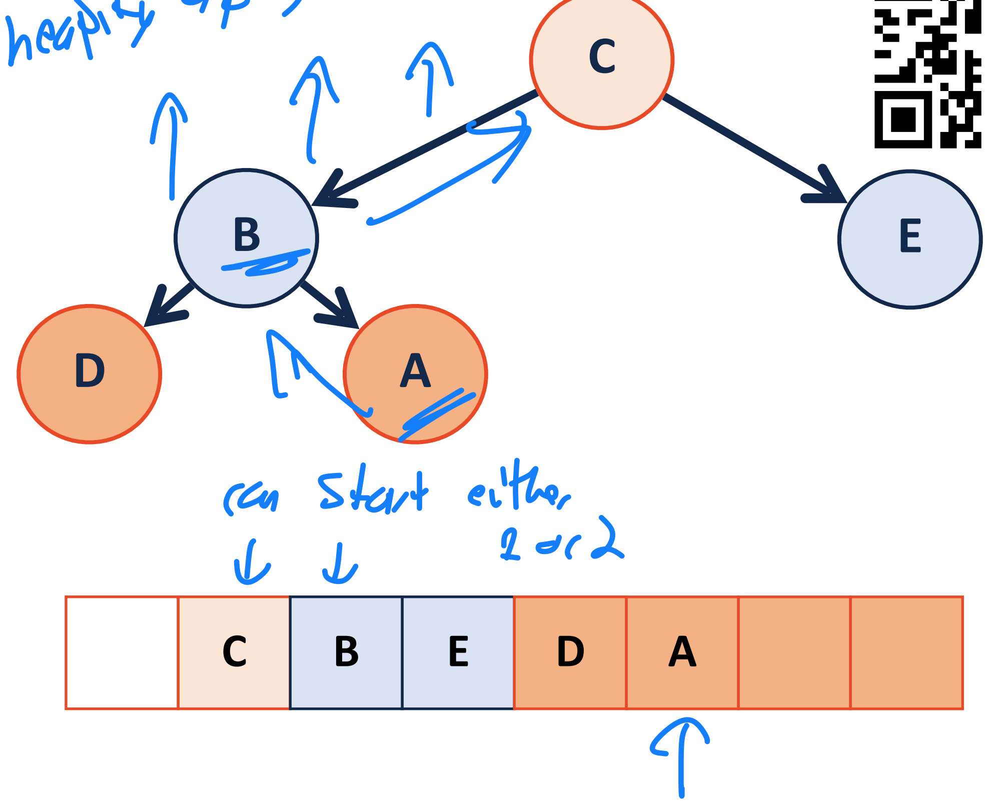
Join Code: 225

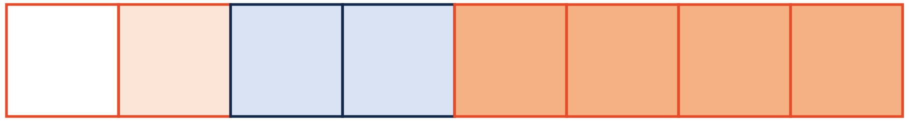
buildHeap - heapifyUp

Repeatedly `heapifyUp(i)`:

Starting at index 1 / 2

Ending at index size_ - 1

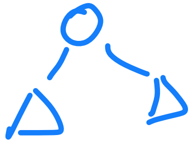




buildHeap - heapifyDown

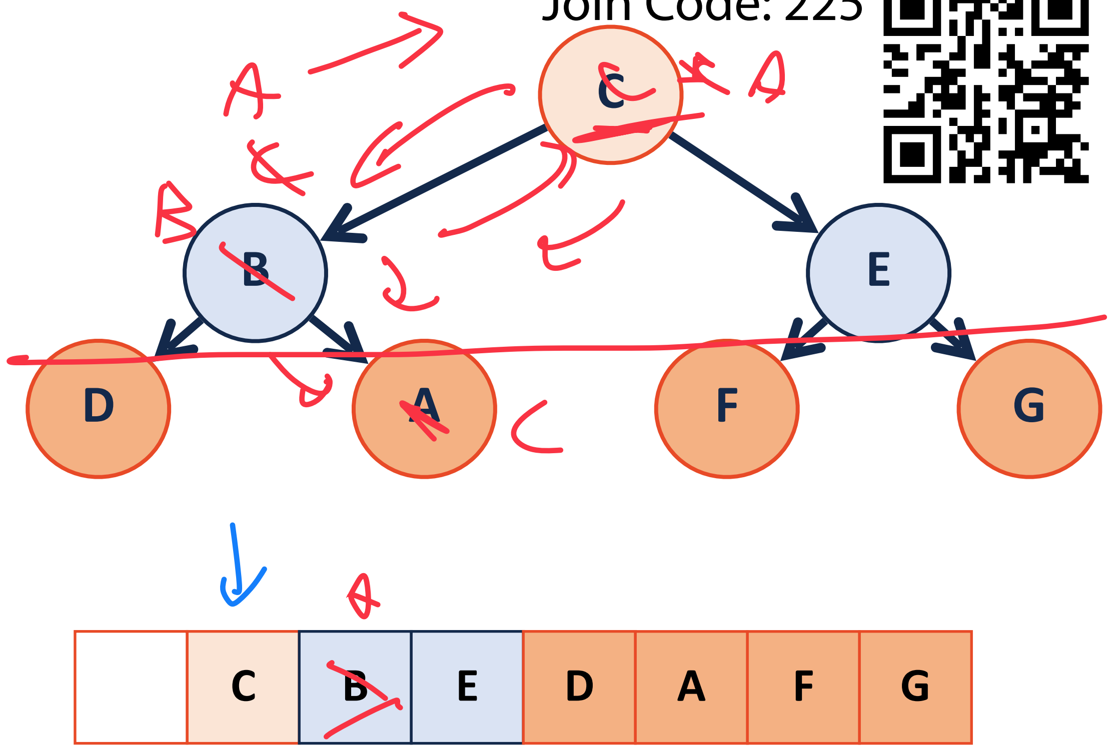
Join Code: 225

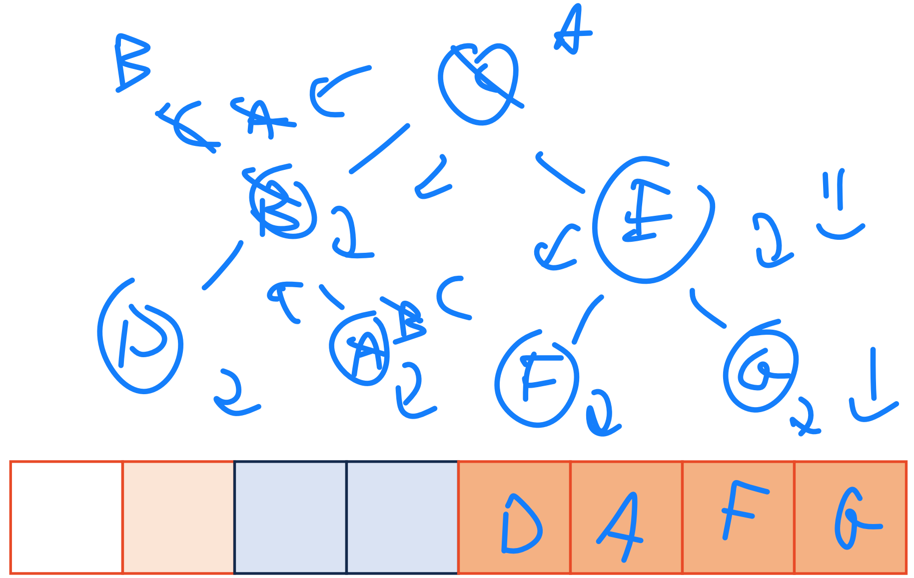
Do we start from top or bottom?



At a glance they both seem to work!

↳ But....





buildHeap - heapifyDown

Join Code: 225

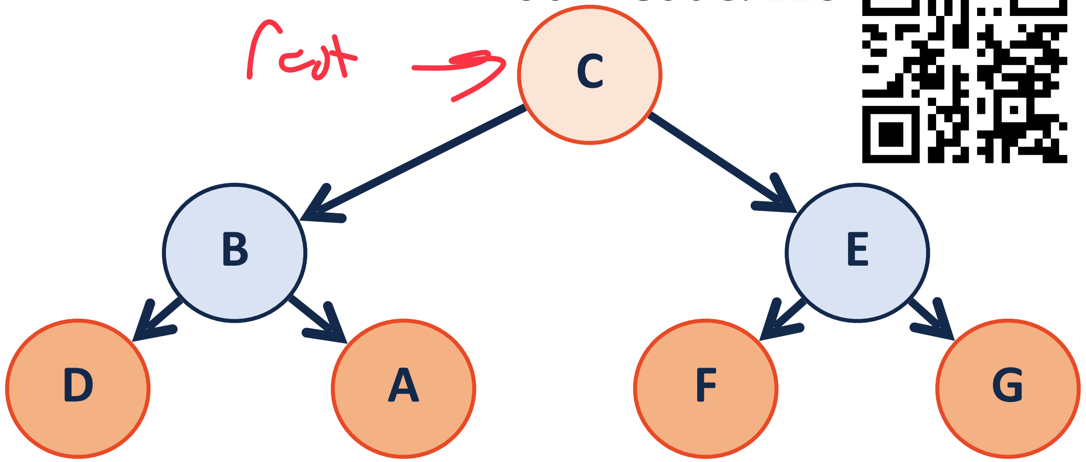
Repeatedly `heapifyDown(i)`:

Starting at index size/2

Ending at index 1

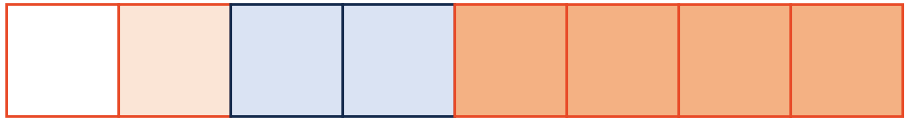
This is much faster than the opposite

root



first non leaf node from bottom up

	C	B	E	D	A	F	G



buildHeap

1. Sort the array — its a heap!

O(n log n)

2. heapifyUp()

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

n x

O(log n)

O(n log n)

3. heapifyDown()

```
template <class T>
void Heap<T>::buildHeap() {
  for (unsigned i = size/2; i > 0; i--) {
    heapifyDown(i);
  }
}
```

n/2 x

seems O(log n)

actually O(h)

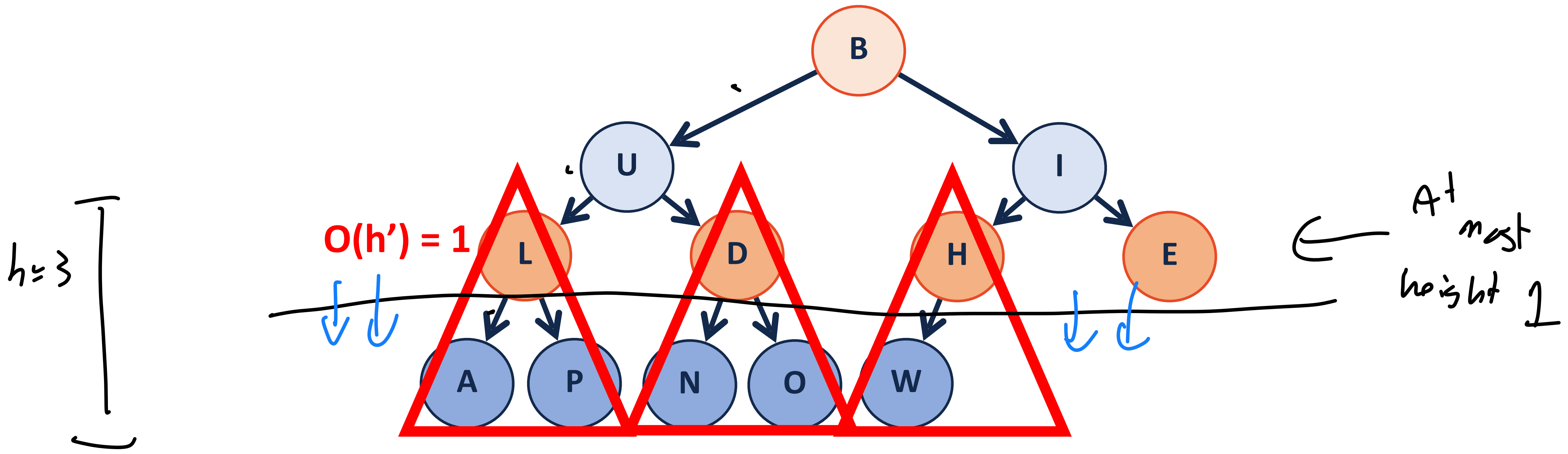
seems like O(n log n)

actually O(n) !!

buildHeap - heapifyDown

Lets break down the total 'amount' of work:

2^{h-1} nodes (4) each doing 1 work

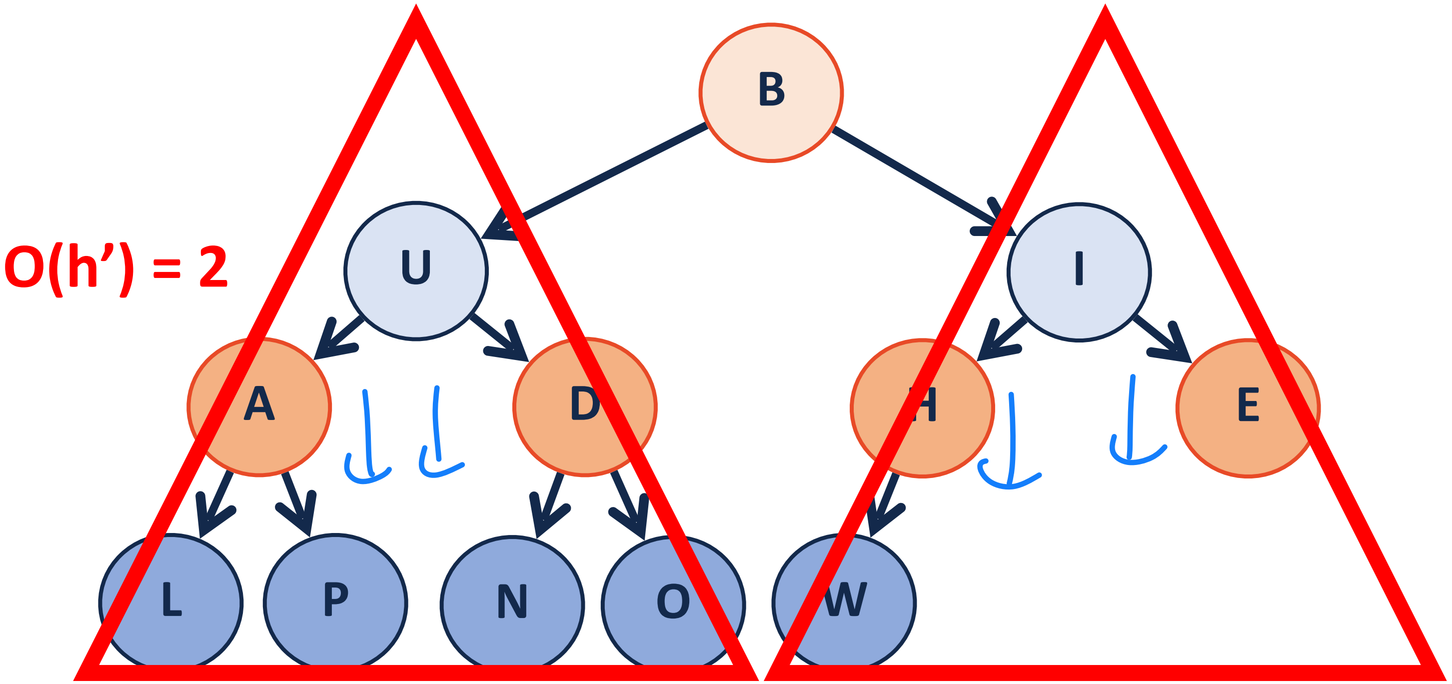


	B	U	I	L	D	H	E	A	P	N	O	W			

buildHeap - heapifyDown

Lets break down the total 'amount' of work:

2^{h-2} (2) each doing 2 work

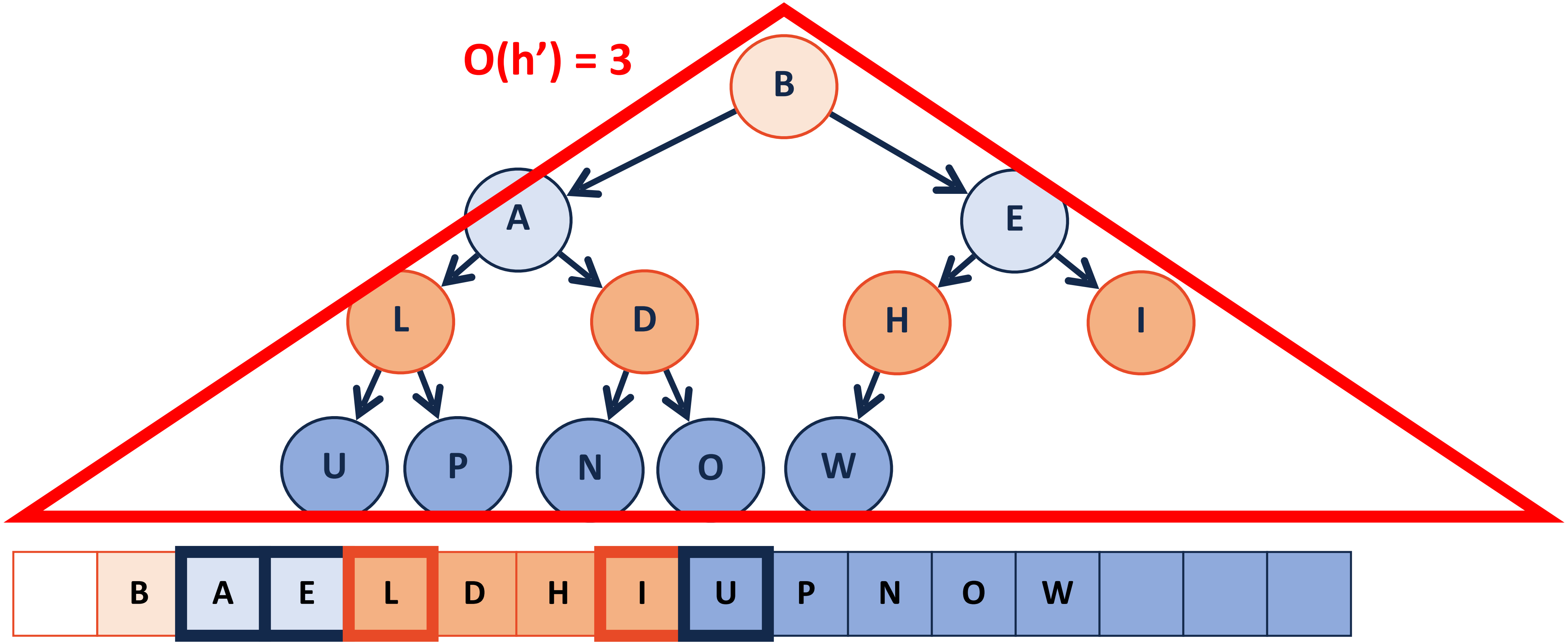


	B	U	I	A	D	H	E	L	P	N	O	W			

buildHeap - heapifyDown

Lets break down the total ‘amount’ of work:

2^{h-3} (1) node doing 3 work



Proving buildHeap Running Time

Theorem: The running time of buildHeap on array of size **n** is: $O(n)$

Strategy: We will add up all runtimes for heapifyDown

||

all heights of all internal nodes

Finish this proof Friday :)

Proving buildHeap Running Time

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

Strategy:

1) Call heapifyDown on every non-leaf node

2) Worst case work for any node is the height of node

3) To prove time, simply add up worst case swaps of every node

Proving buildHeap Running Time

S(h): Sum of the heights of all nodes in a **perfect** tree of height **h**.

S(0) =

S(1) =

S(2) =

S(h) =

Proving buildHeap Running Time

Claim: Sum of heights of all nodes in a perfect tree: $S(h) = 2^{h+1} - 2 - h$

Base Case:

Proving buildHeap Running Time

Claim: Sum of heights of all nodes in a perfect tree: $S(h) = 2^{h+1} - 2 - h$

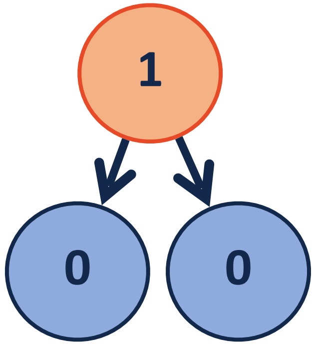
Base Case:

h = 0

$$2^{0+1} - 2 - 0 = 0$$

vs

h = 1



$$2^{1+1} - 2 - 1 = 1$$

Proving buildHeap Running Time

Claim: Sum of heights of all nodes in a perfect tree: $S(h) = 2^{h+1} - 2 - h$

Induction Step:

Proving buildHeap Running Time

Claim: Sum of heights of all nodes in a perfect tree: $S(h) = 2^{h+1} - 2 - h$

Induction Step: $S(i) = i + 2\ S(i-1)$ is true for all values $i < h$

$S(h-1) = 2^{h-1+1} - 2 - (h-1) = 2^h - h - 1$ (By IH)

$S(h) = h + 2\ S(h-1) = h + \left(2\ (2^h - h - 1)\right)$ (Plug in)

$S(h) = 2^{h+1} - 2 - h$ (Simplify)

Proving buildHeap Running Time

Theorem: The running time of buildHeap on array of size **n** is O(n)

$$S(h) = 2^{h+1} - 2 - h$$

How can we relate **h** and **n**?

How can we estimate running time?

Proving buildHeap Running Time

Theorem: The running time of buildHeap on array of size **n** is O(n)

$$S(h) = 2^{h+1} - 2 - h$$

How can we relate **h** and **n**? $h \leq log\ n$

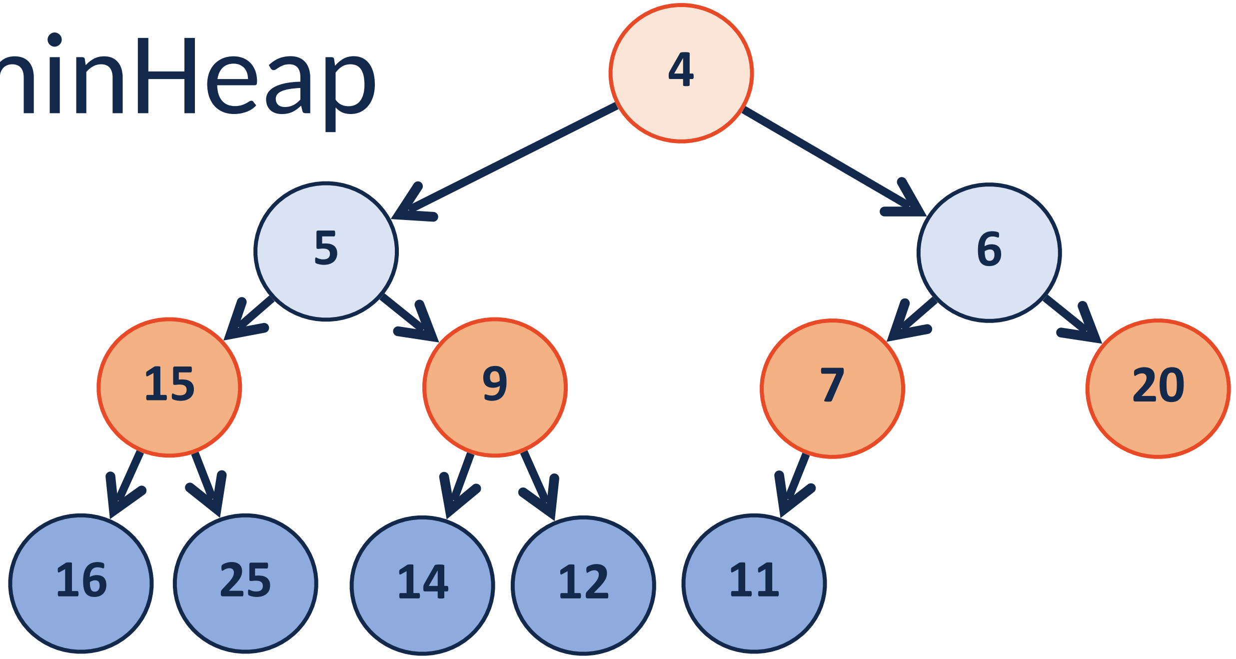
How can we estimate running time?

$2^{log\ n+1} - 2 - log\ n$ (Plug in)

$2 * 2^{log_2 n} - 2 - log\ n$ (Simplify)

$2\ n - log\ n - 2 \approx O(n)$ (Rearrange)

minHeap



1. Construction

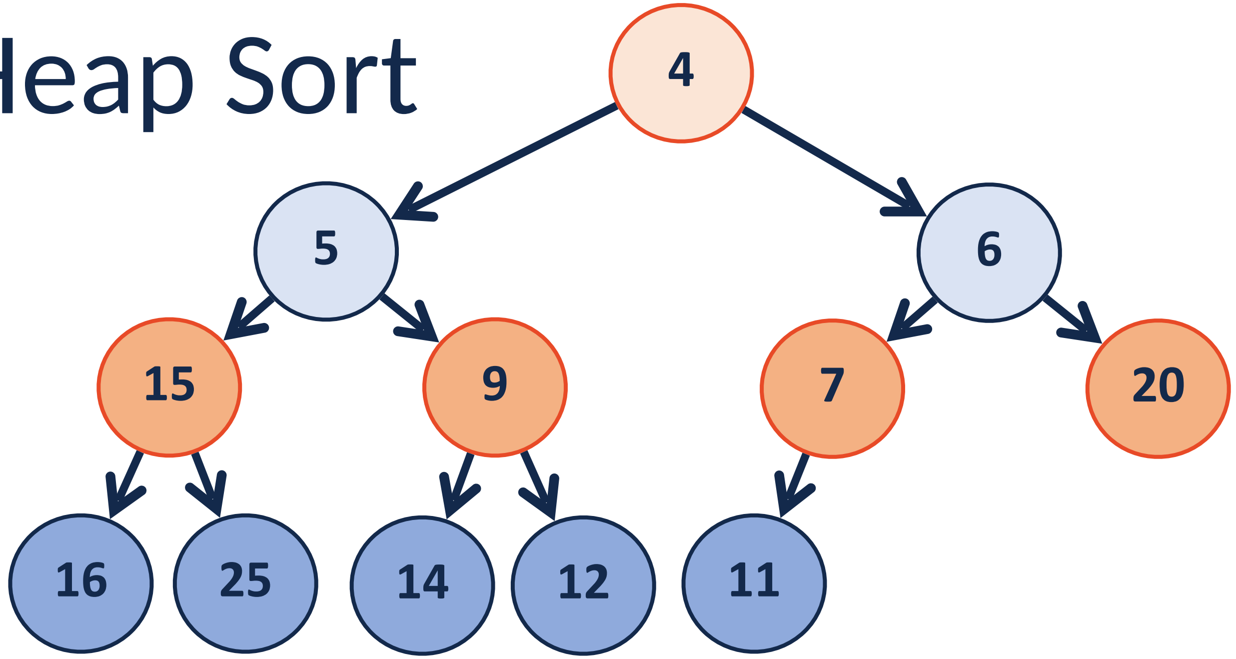
2. Insert

3. RemoveMin

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

minHeap is a good example of tradeoffs:

Heap Sort



1.

2.

3.

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

Running time?