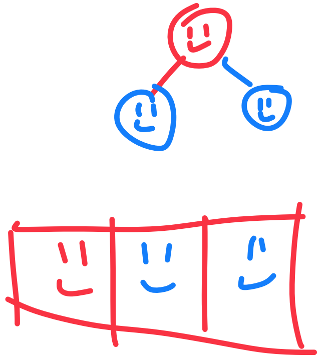
Data Structures

Heaps

CS 225
Brad Solomon

October 11, 2024



haha it was
lists all along

UNIVERSITY OF
ILLINOIS
URBANA-CHAMPAIGN

Department of Computer Science

Learning Objectives

Introduce the heap data structure

Discuss heap ADT implementations

Thinking conceptually: Sorting a queue

How might we build a ‘queue’ in which our front element is the min?

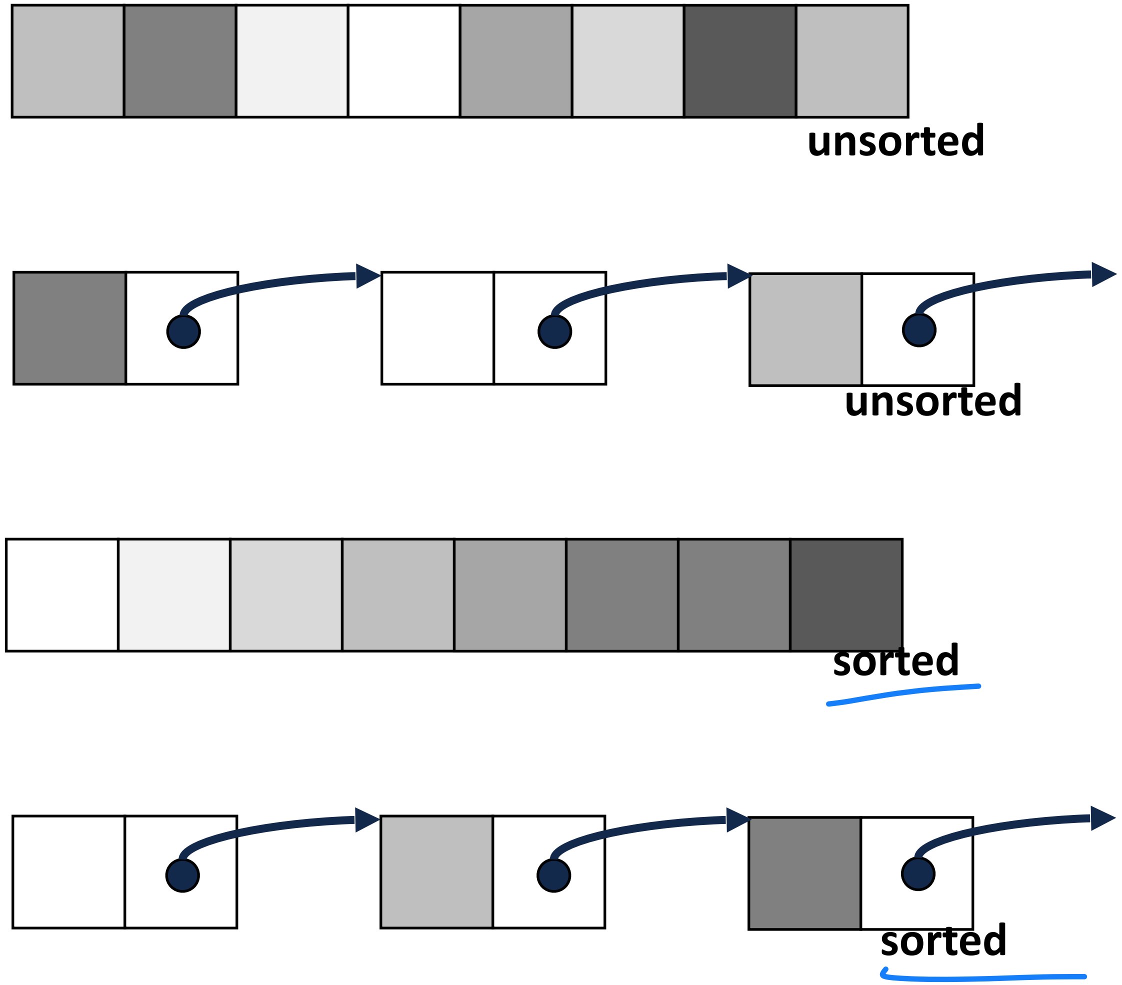
Interface
- ↳ Enqueue
- ↳ Dequeue ← will always return min item $\boxed{O(1)}$

Implementation
- ↳ Sorted List can find in $O(1)$
  - ↳ Array can remove in $O(1)^*$
  - ↳ Linked List

Priority Queue Implementation

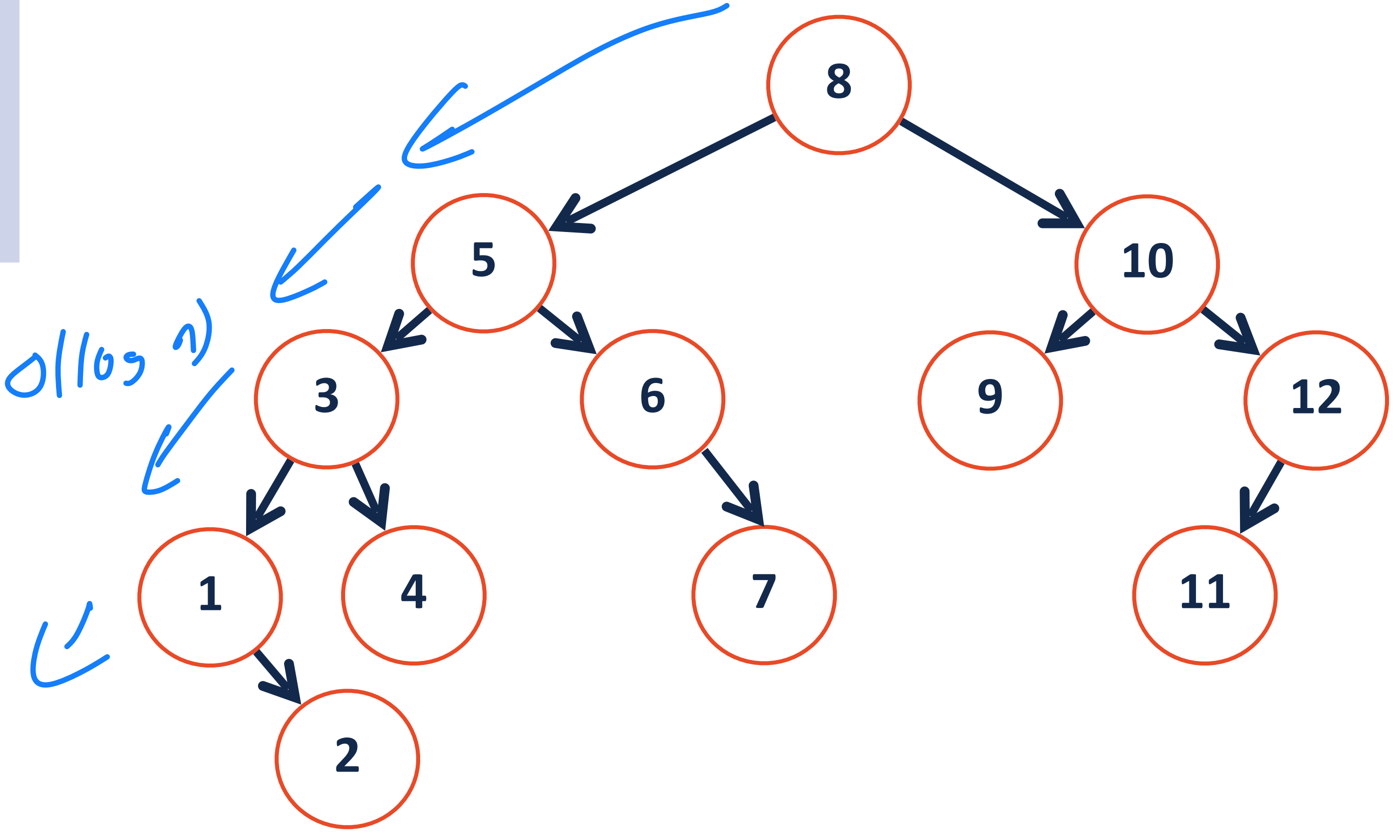
insert	removeMin
O(1*)	O(n)
O(1)	O(n)
O(n)	O(1)
O(n)	O(1)

O(1)*



Priority Queue Implementation

insert	removeMin
$O(\log n)$	$O(\log n)$



$O(h) \approx O(\log n)$

A different priority queue implementation...

1) Min value at root

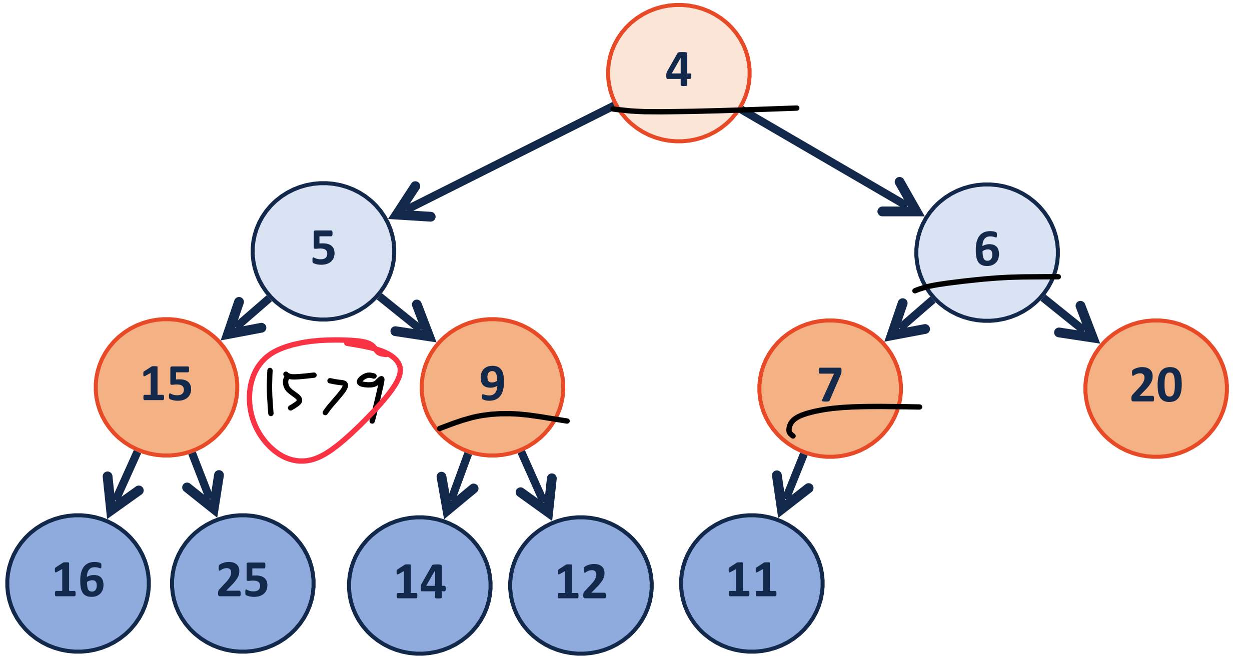
↑ recursively true

↳ A new kind of 'ordered'

2) Left vs right don't matter

3) Binary tree!

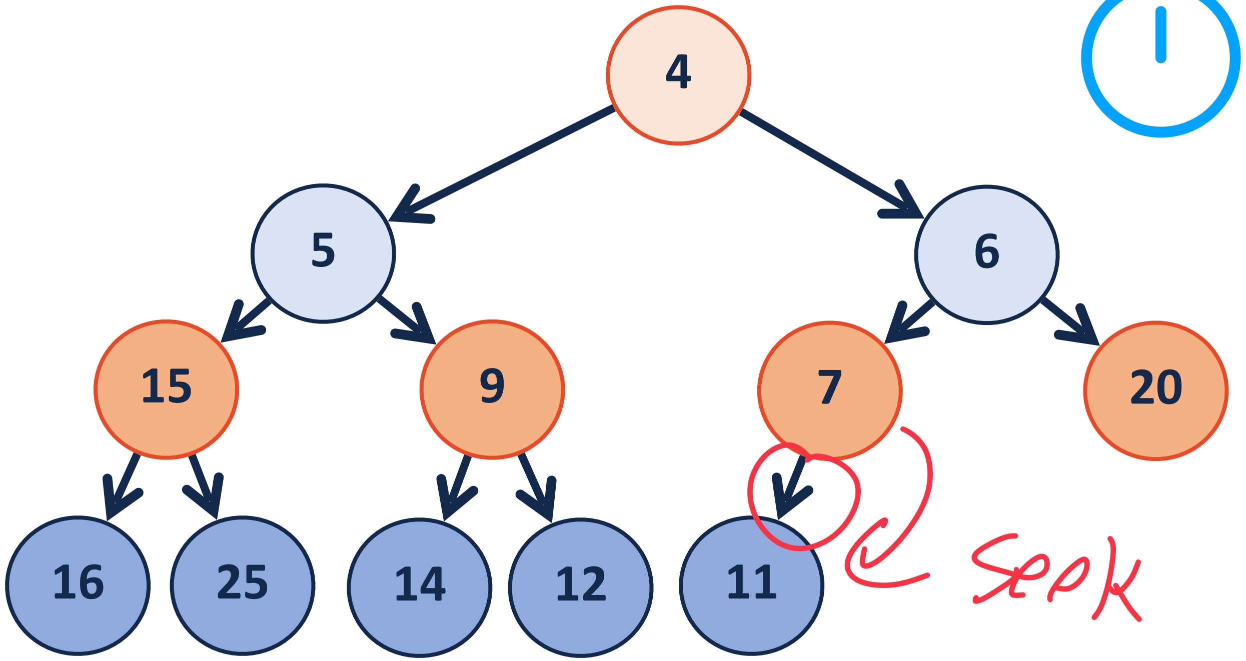
4) Is a complete tree



(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.



↳ This is good for priority queue

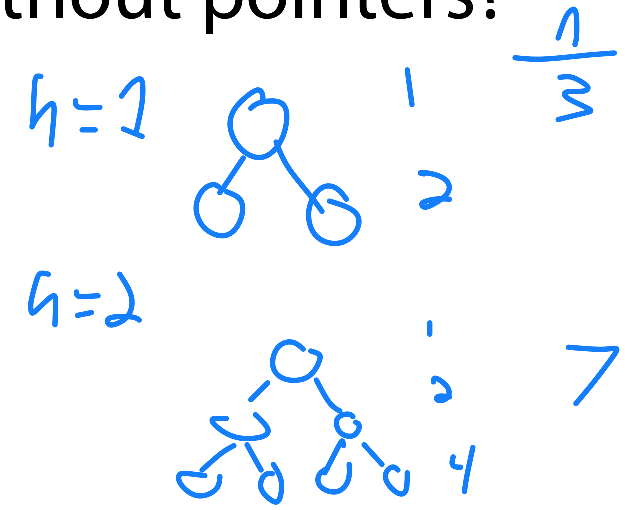
↳ B Tree 's tree seek ops are slow!

Thinking conceptually: A tree without pointers

What class of (non-trivial) trees can we describe without pointers?

↳ complete / perfect

↳ n vs h



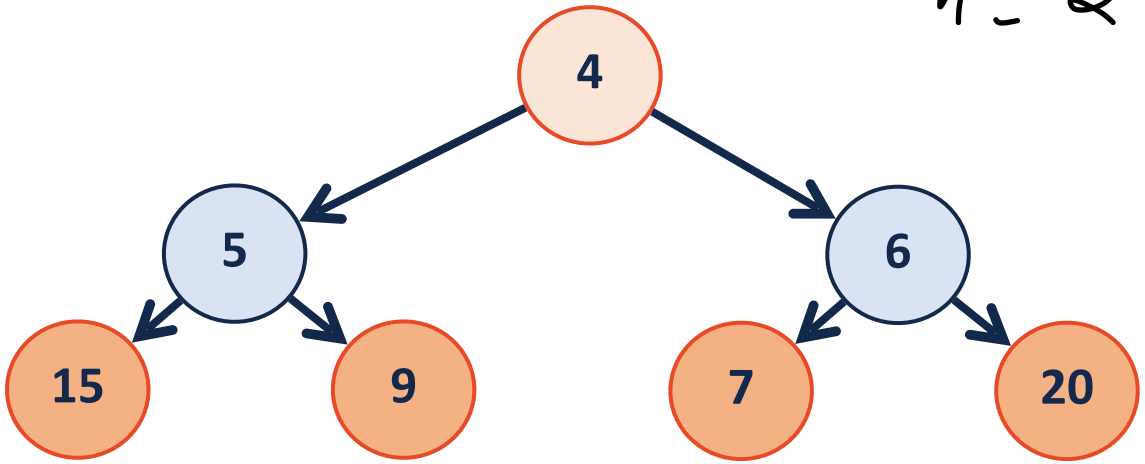
What is the relationship between nodes and height for these trees?

complete $\leq 2^{h+1} - 1$

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

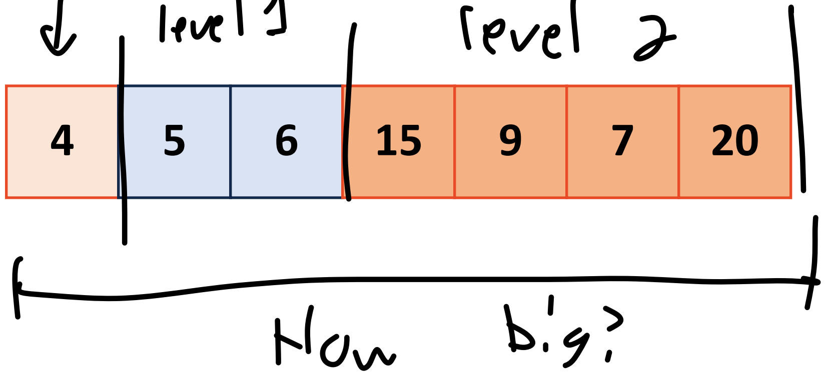
(min)Heap

h = 2



This is visual only

This is stored



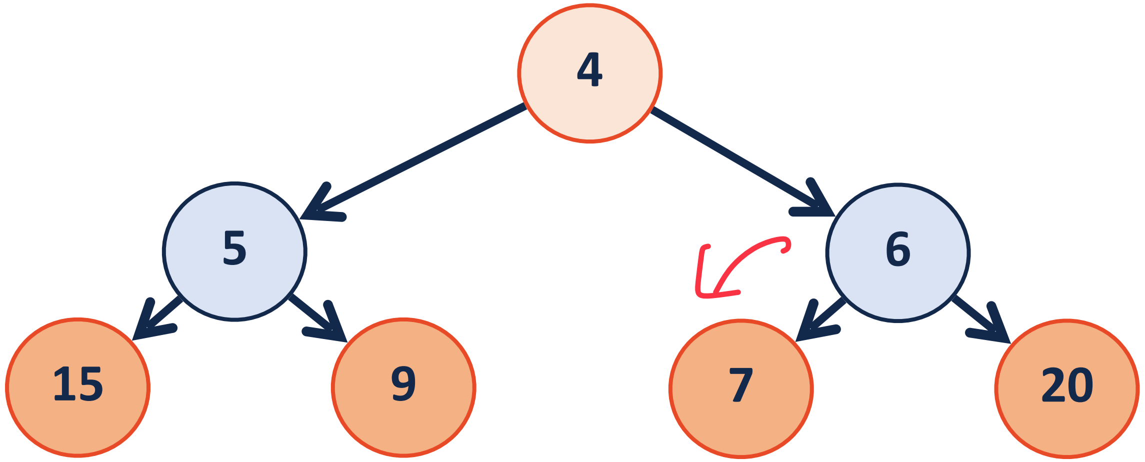
4	5	6	15	9	7	20

$2^3 - 1 = 7$

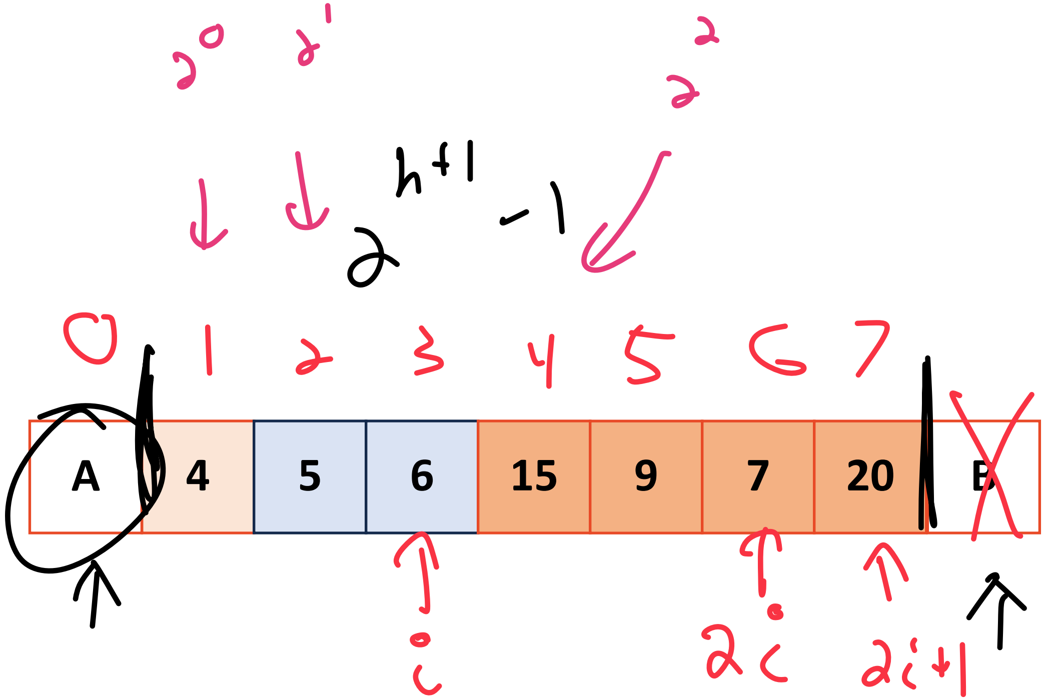
(min)Heap

Claim: Blank in front makes math easier

(Design decision)

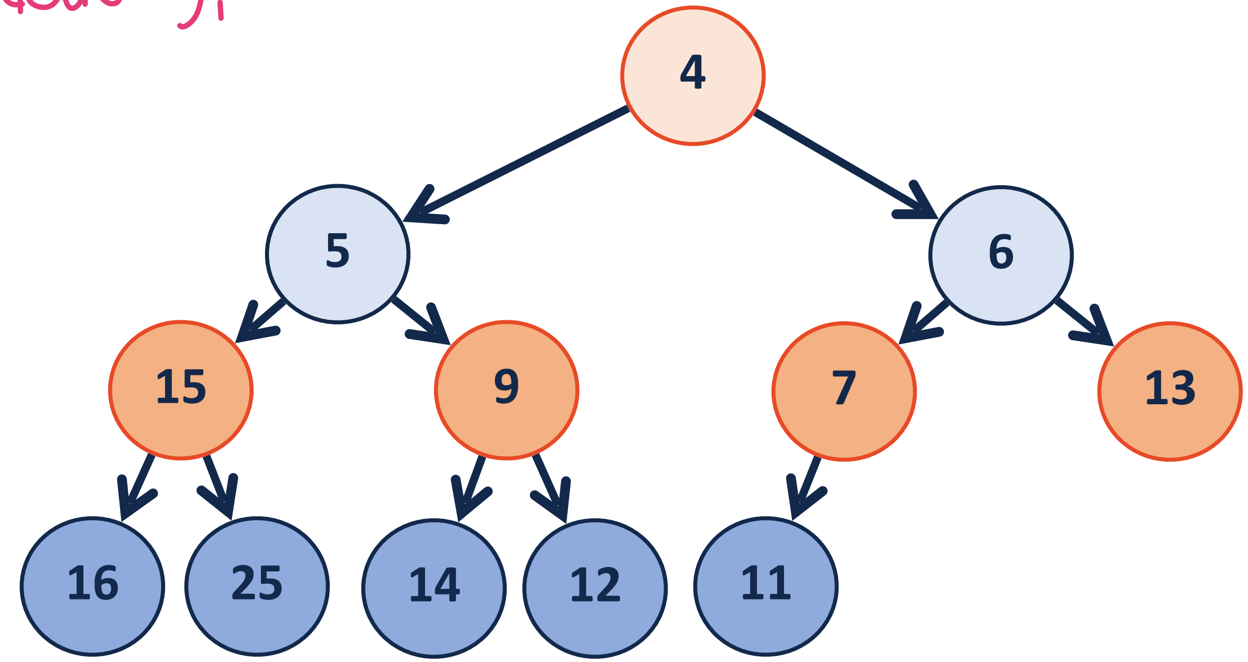


2^x is easier to allocate



growArray

New tree level == array doubling!



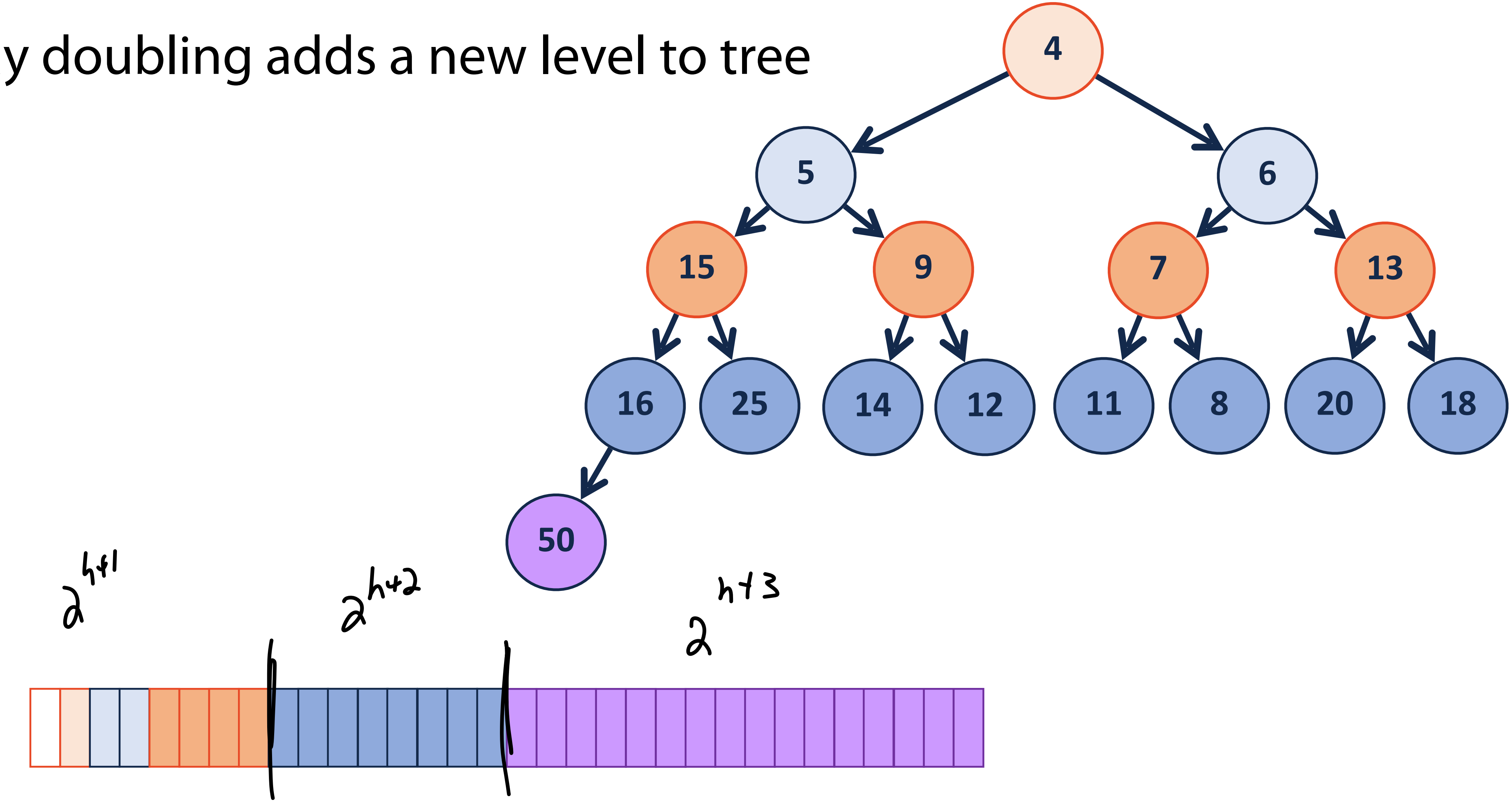
2^{h+2}

2^{h+1}

	4	5	6	15	9	7	20	16	25	14	12	11			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

growArray

Array doubling adds a new level to tree



(min)Heap

`leftChild(i):` 2i

index of node 7 is 6 → left child 12

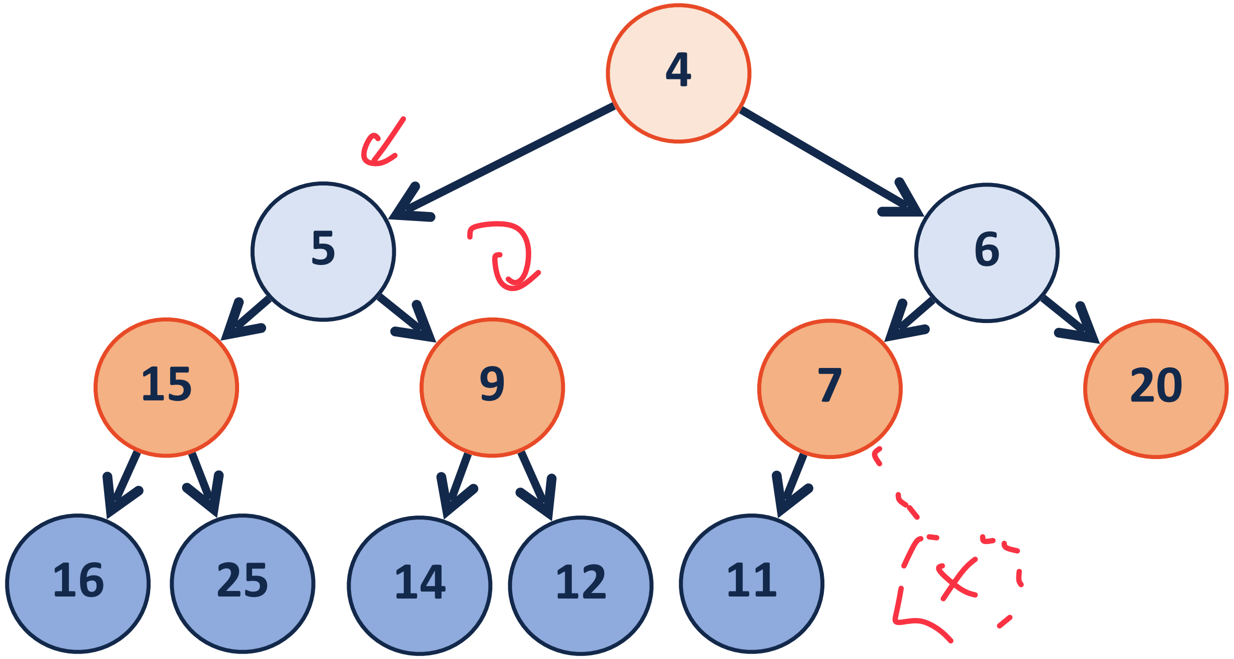
node 5 is 2 → left child 4

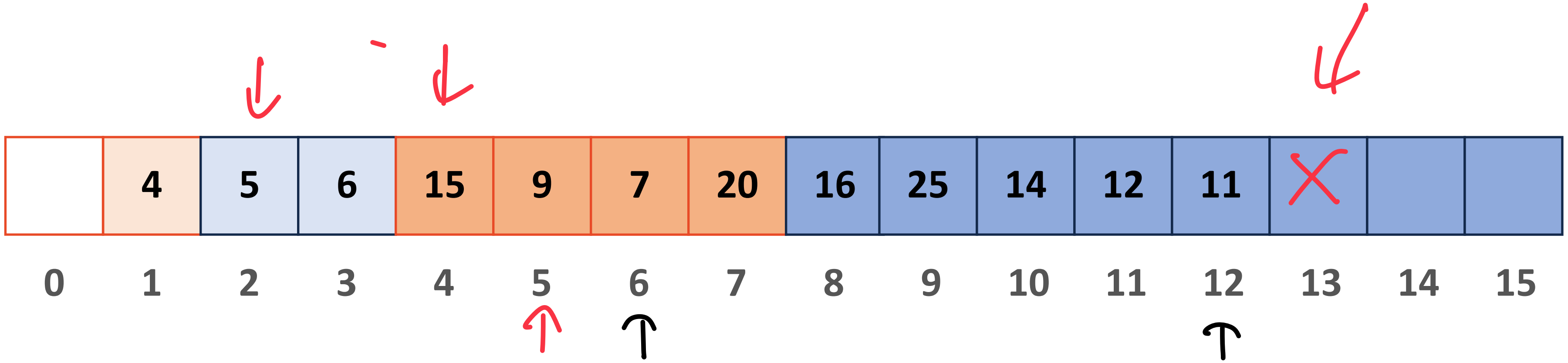
i is index of node

`rightChild(i):` 2i+1

6 → 13

5 → 5





	4	5	6	15	9	7	20	16	25	14	12	11	X		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

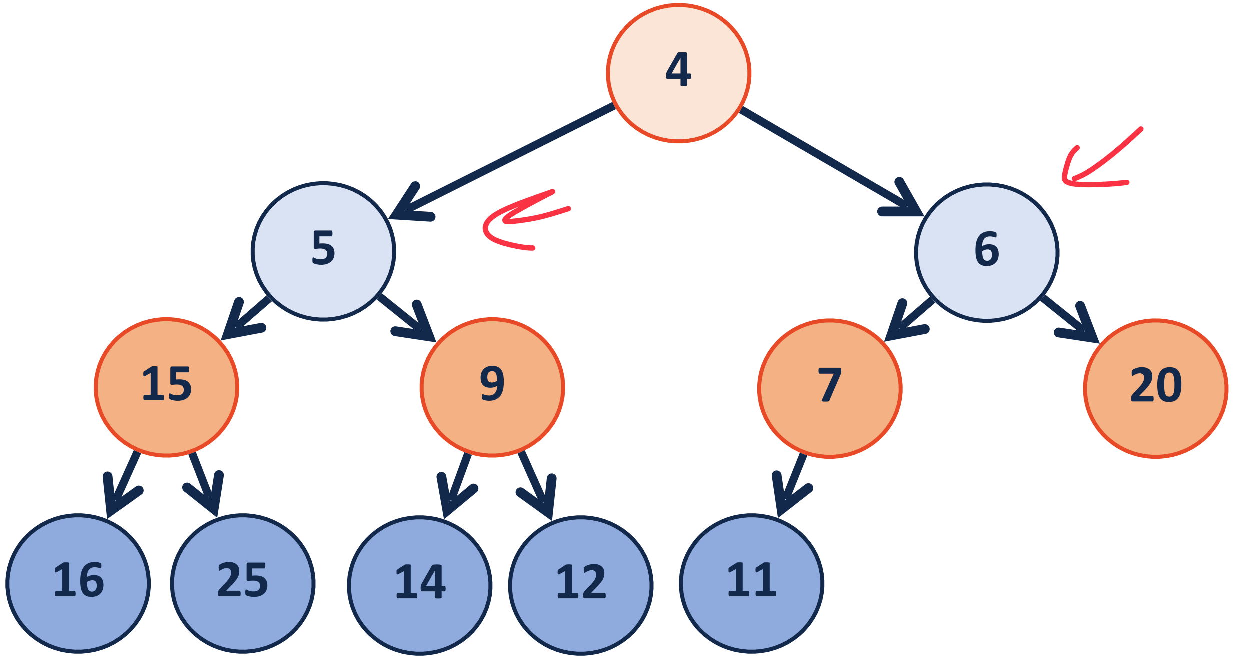
(min)Heap

`parent(i):`

$i=3$, parent @ $i=1$

$i=2$, parent @ $i=1$

$\text{int}\left(\frac{i}{2}\right)$ or $\left\lfloor \frac{i}{2} \right\rfloor$



	4	5	6	15	9	7	20	16	25	14	12	11			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

(min)Heap

(Perfect

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

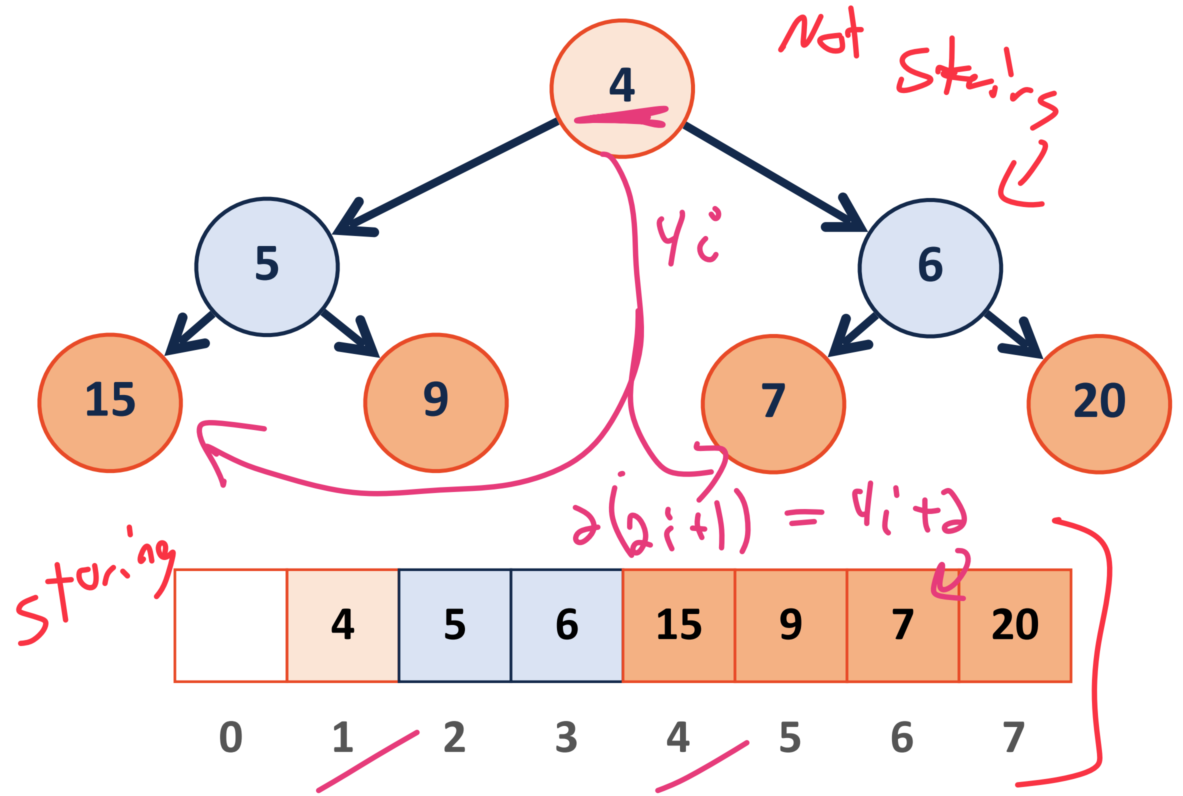
Can index from 0 or 1 (we will index from 1 in slides)

`leftChild(i): 2i`

O(1)
parent
&
children

`rightChild(i): 2i+1`

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



Not stairs

4i

2(2i+1) = 4i+2

storing

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

(min)Heap ADT → Priority queue

Insert

RemoveMin

Constructor

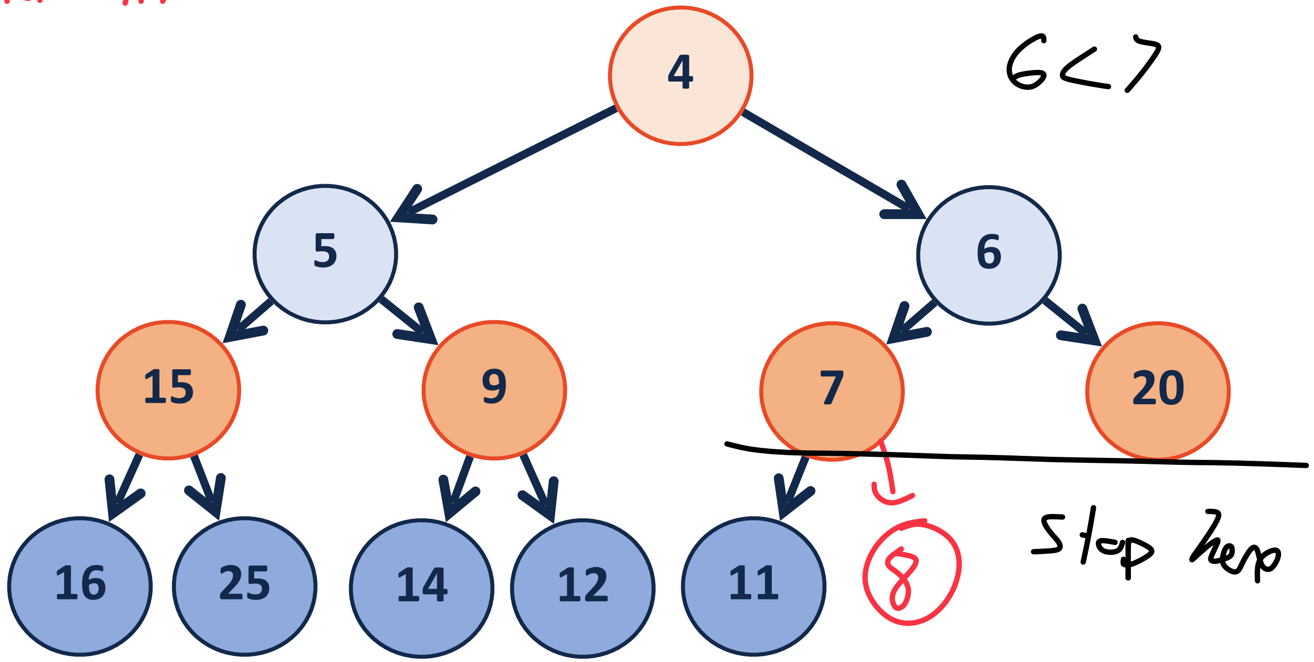
insert

Insert(8)

Insert Back in array (in minheap) is O(1)*

1) Insert at end of array
2) Check if minheap valid

≡ is node smaller than parent?



4<6

6<7

stop here

7<8

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	4	5	6	15	9	7	20	16	25	14	12	11	8		

insert

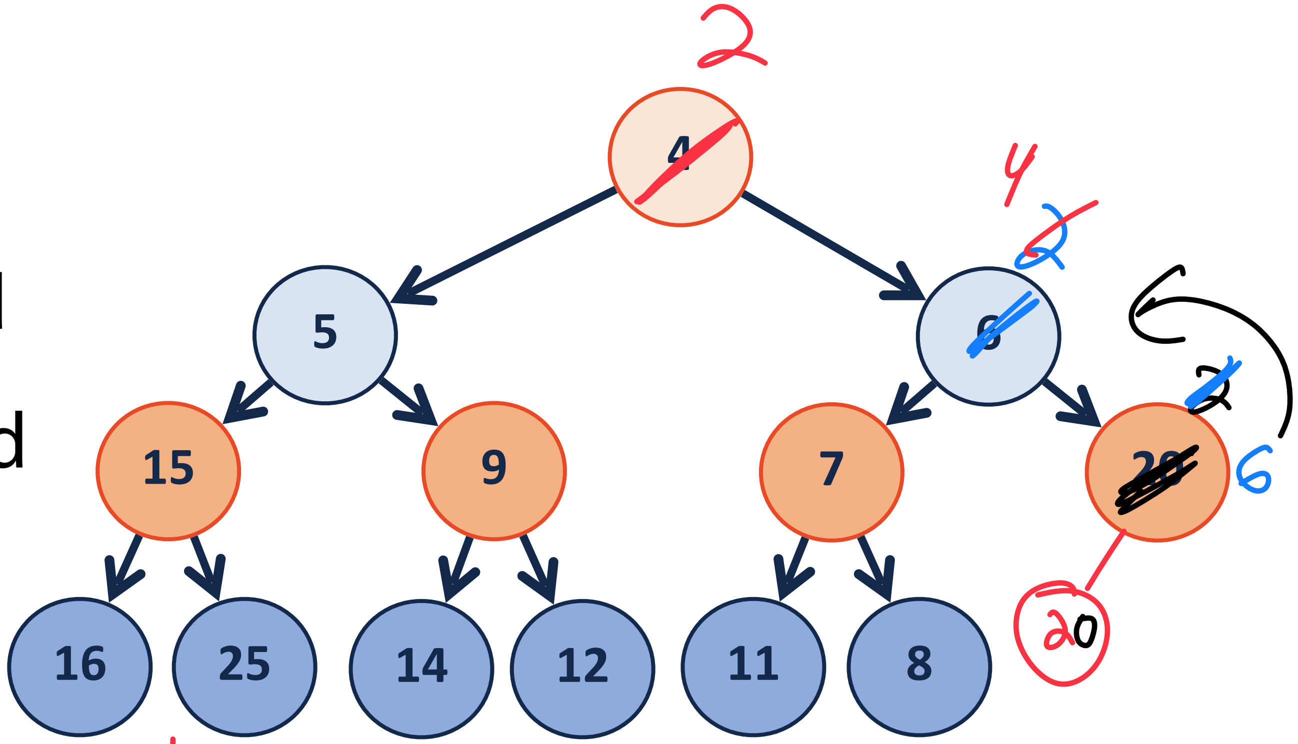
Insert(2)

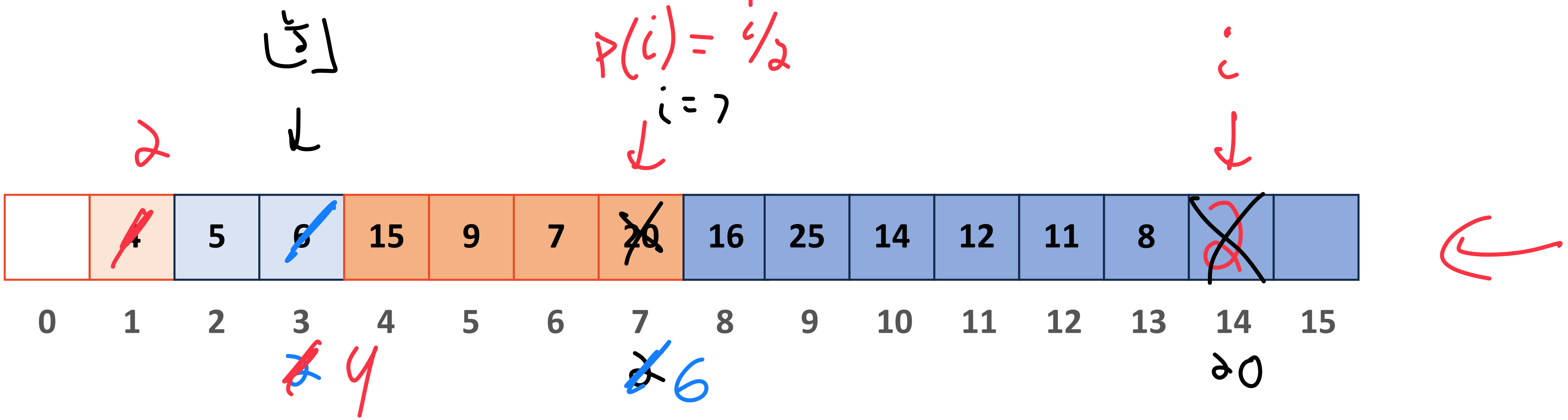
1) Insert at end of array

2) Check if minHeap still valid

3) Swap with parent if needed

Steps 2 and 3 are recursive!




$\lfloor \frac{i}{2} \rfloor$

$p(i) = i/2$

i = 7

i

insert

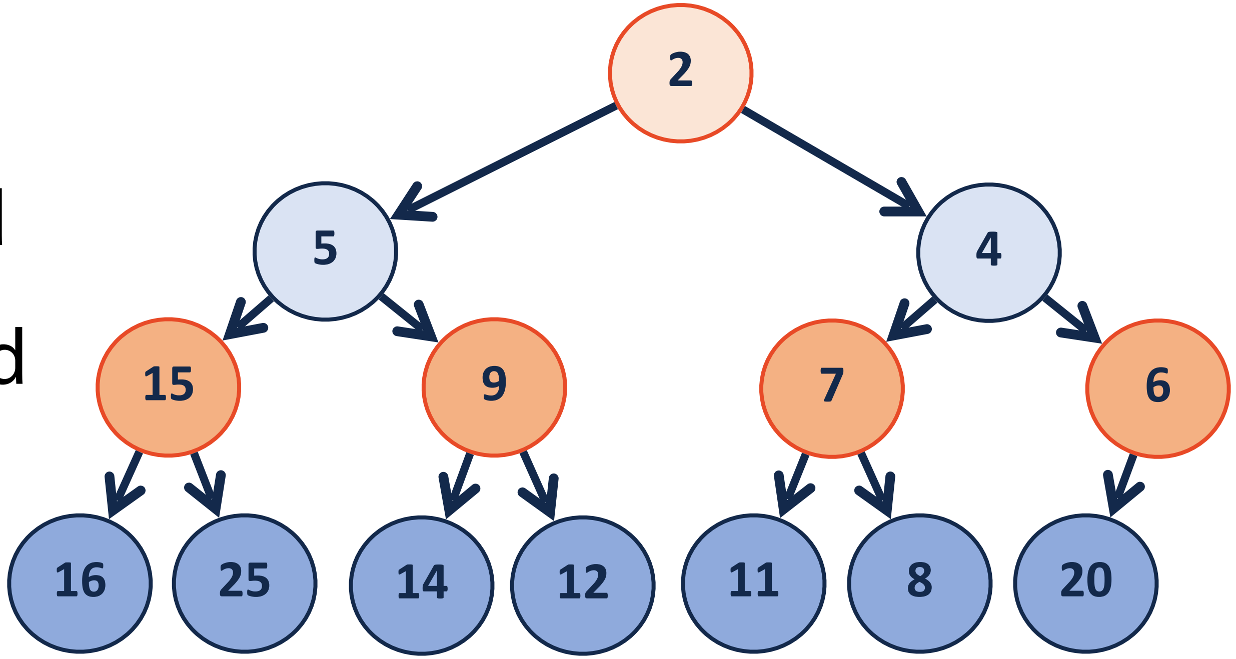
[After] Insert(2)

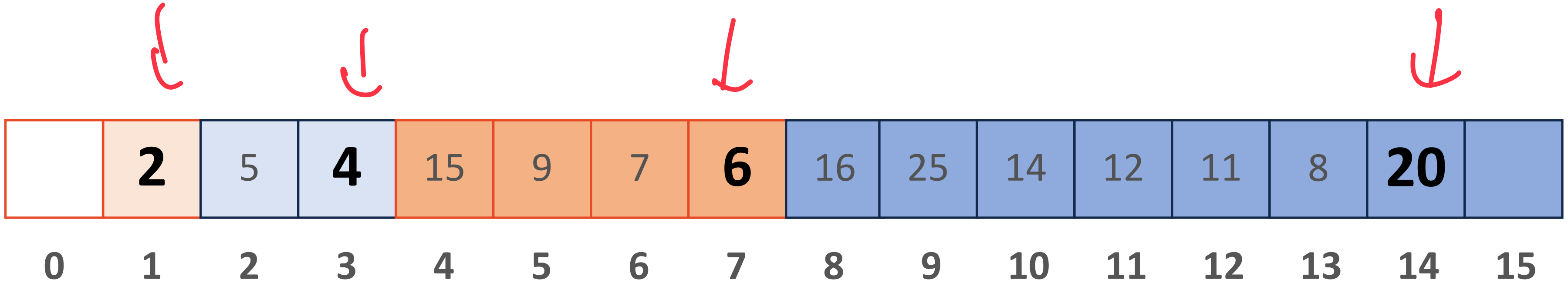
1) Insert at end of array

2) Check if minHeap still valid

3) Swap with parent if needed

Steps 2 and 3 are recursive!





	2	5	**4**	15	9	7	**6**	16	25	14	12	11	8	**20**	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

insert

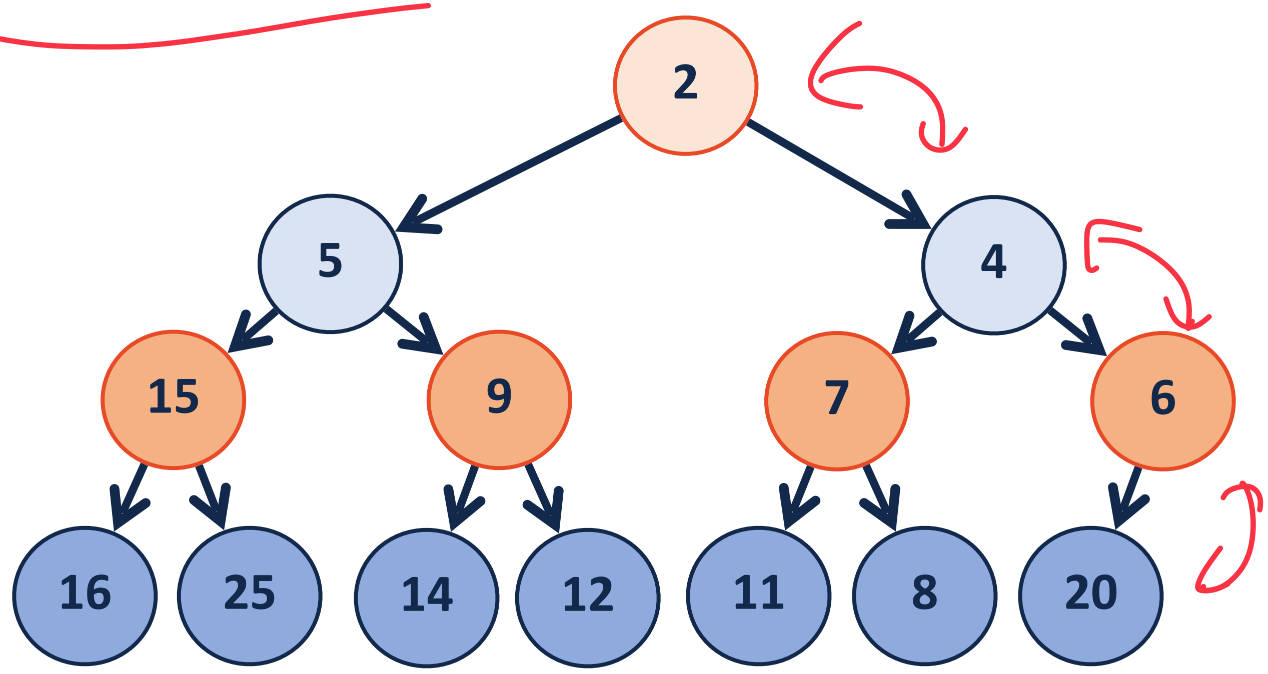
$O(\log n)$

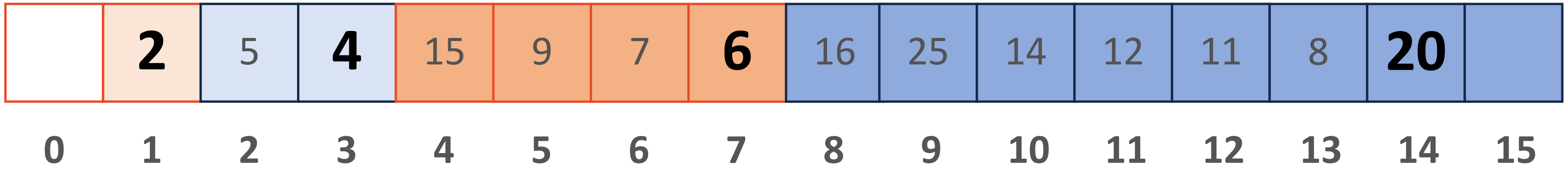
[After] Insert(2)

What is my height? 3

Number of swaps? 3

$h \approx O(\log n)$



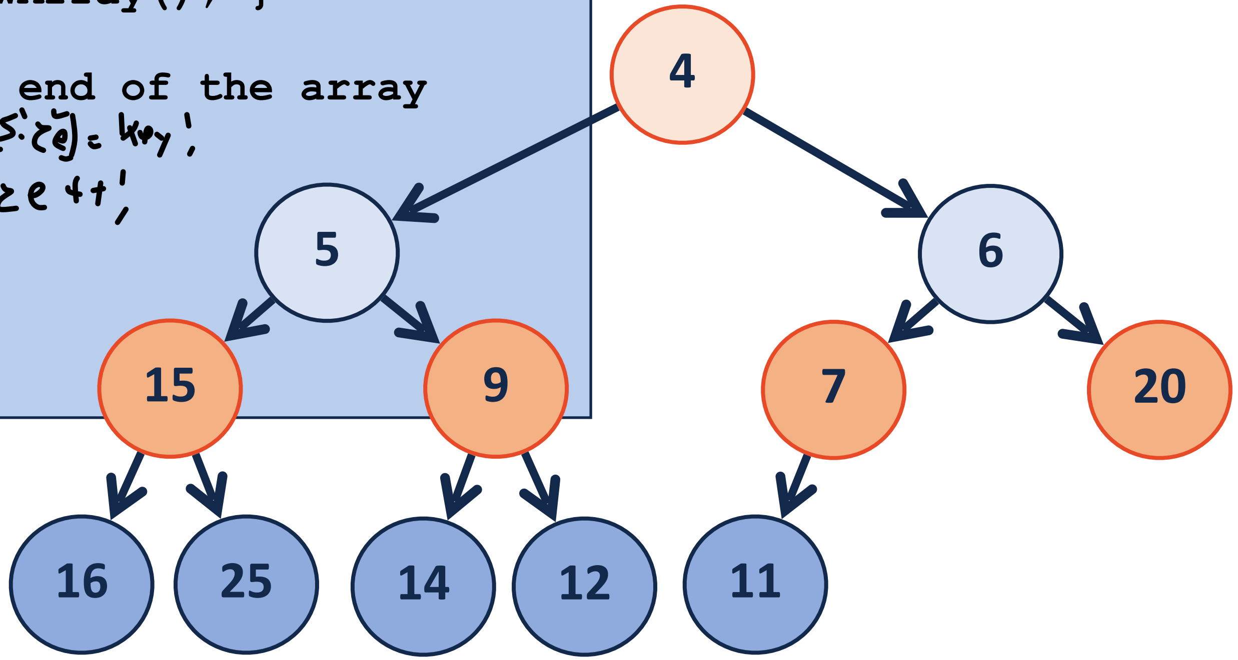


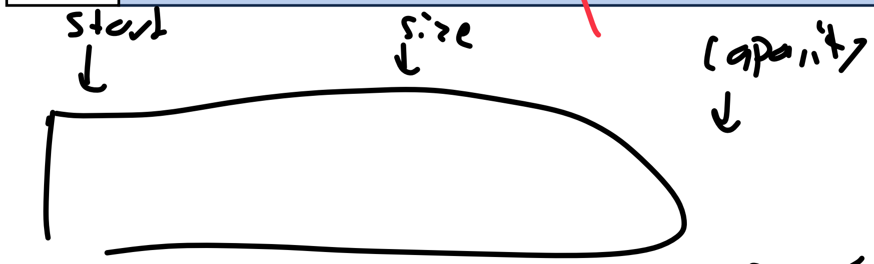
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	2	5	**4**	15	9	7	**6**	16	25	14	12	11	8	**20**	

double array
if full

```
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);
}
```

item[size_] = key;
size_++;





size is first open space

size_ ?++

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

insert - heapifyUp

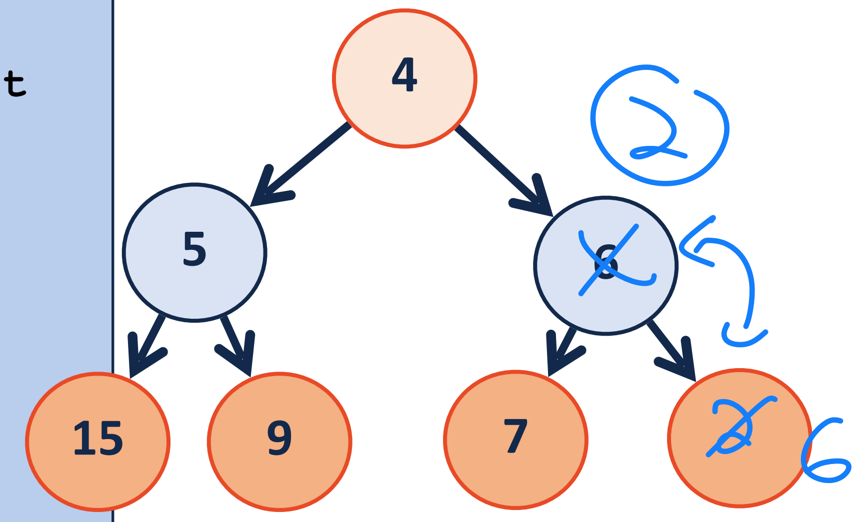
```
template <class T>
void Heap<T>::_insert(const T & key) {
  // Check to ensure there's space to insert an element
  // ...if not, grow the array
  if ( size_ == capacity_ ) { _growArray(); }

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

  // Restore the heap property
  _heapifyUp(size_ - 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( ____i/2____ );
    }
  }
}
```

← if ~~not~~ root base case

← swap if parent larger

recurse on "parent"



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