Data Structures

Heaps

CS 225

Brad Solomon

March 9, 2026

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Exam 3 (3/23 — 3/25)

Monday after spring break

Autograded MC and one coding question

Manually graded short answer prompt

Practice exam on PL ~~soon~~* this morning

Topics covered can be found on website

Registration started March 5

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

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

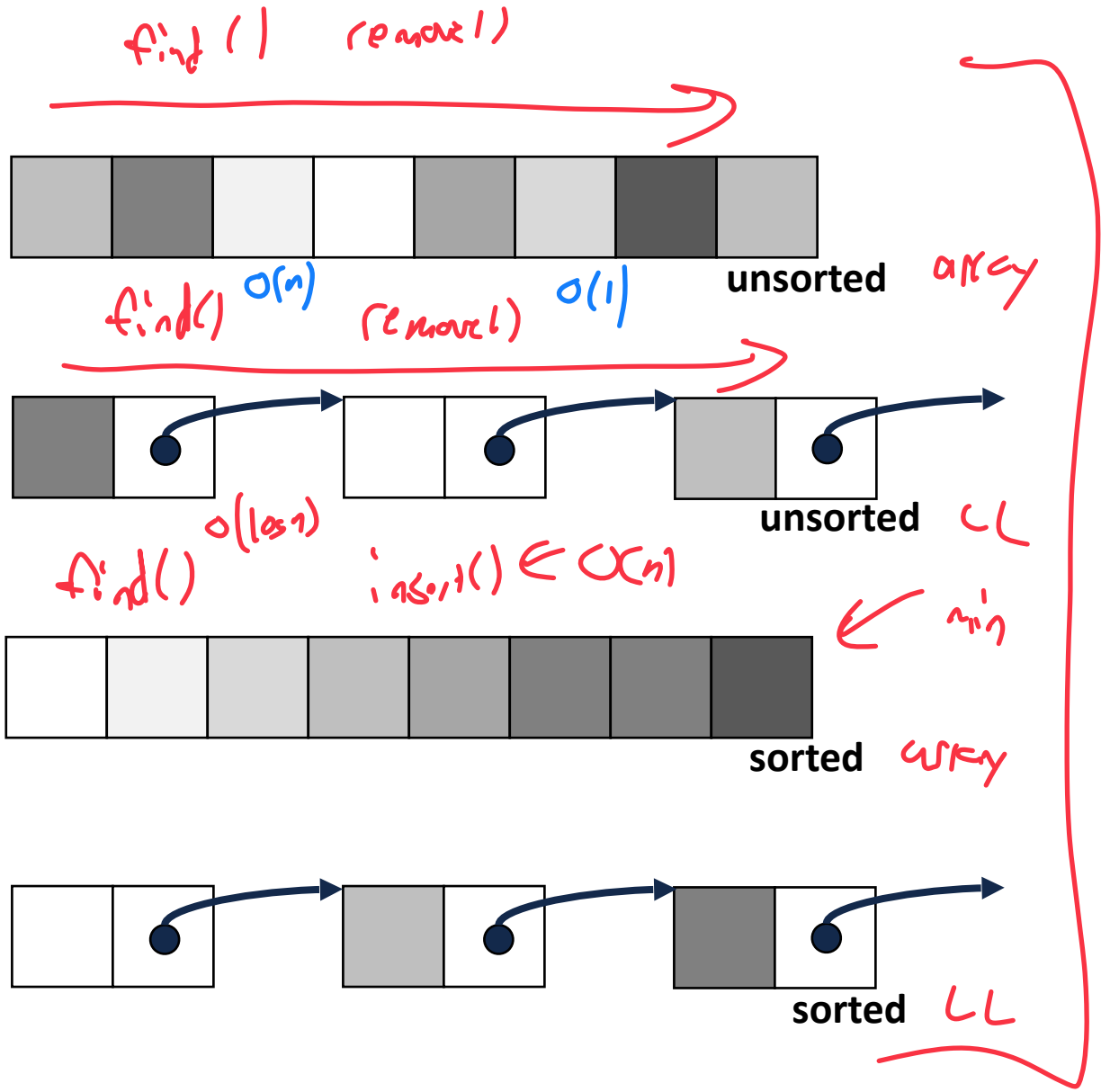
- Insert
- Remove Min()
  - No find() (No random access)!

→ Implementation

- Use linked list & sort every insert (insert sorted)
  - ↑ slow insert
  - ↑ fast remove min
- Array w/ knowledge of current min index
  - fast insert
  - slow removeMin.

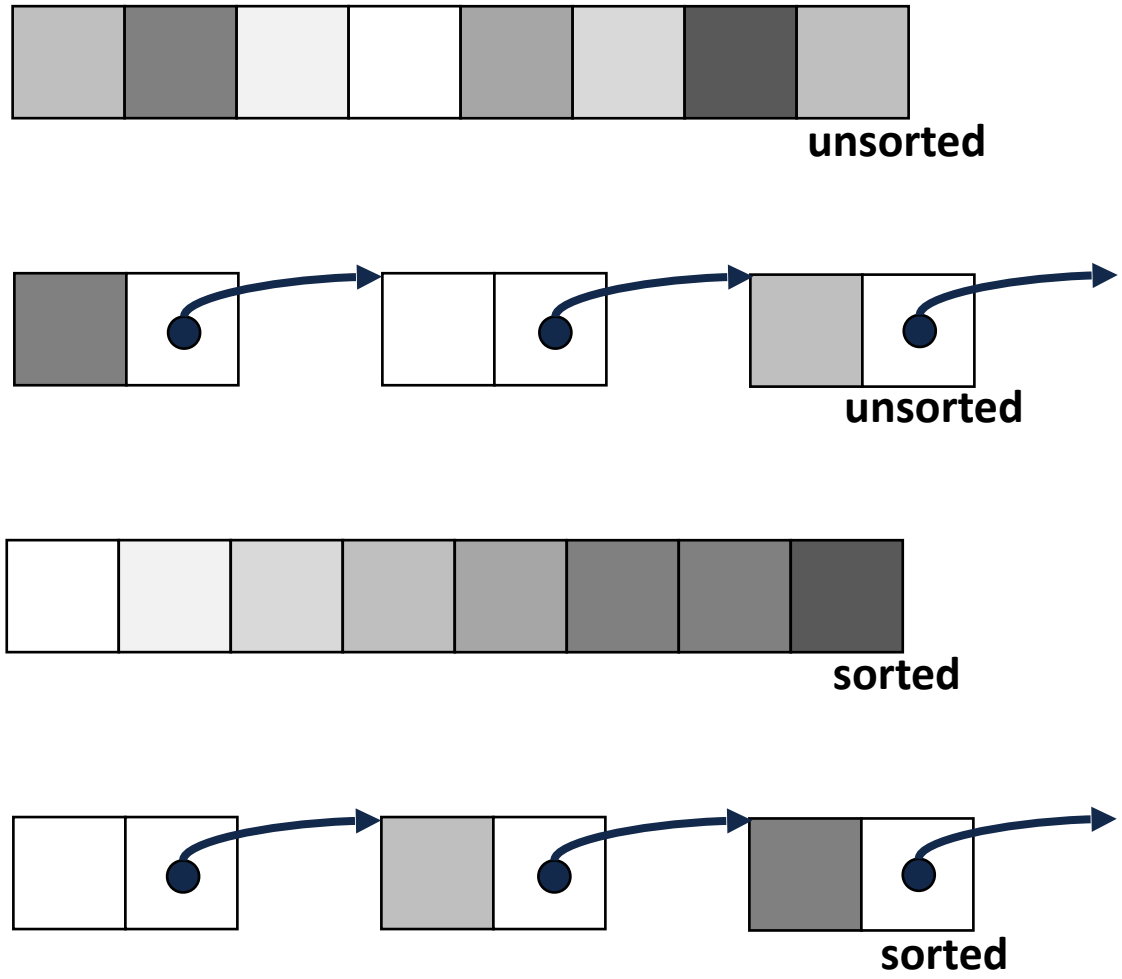
Priority Queue Implementation

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



Priority Queue Implementation

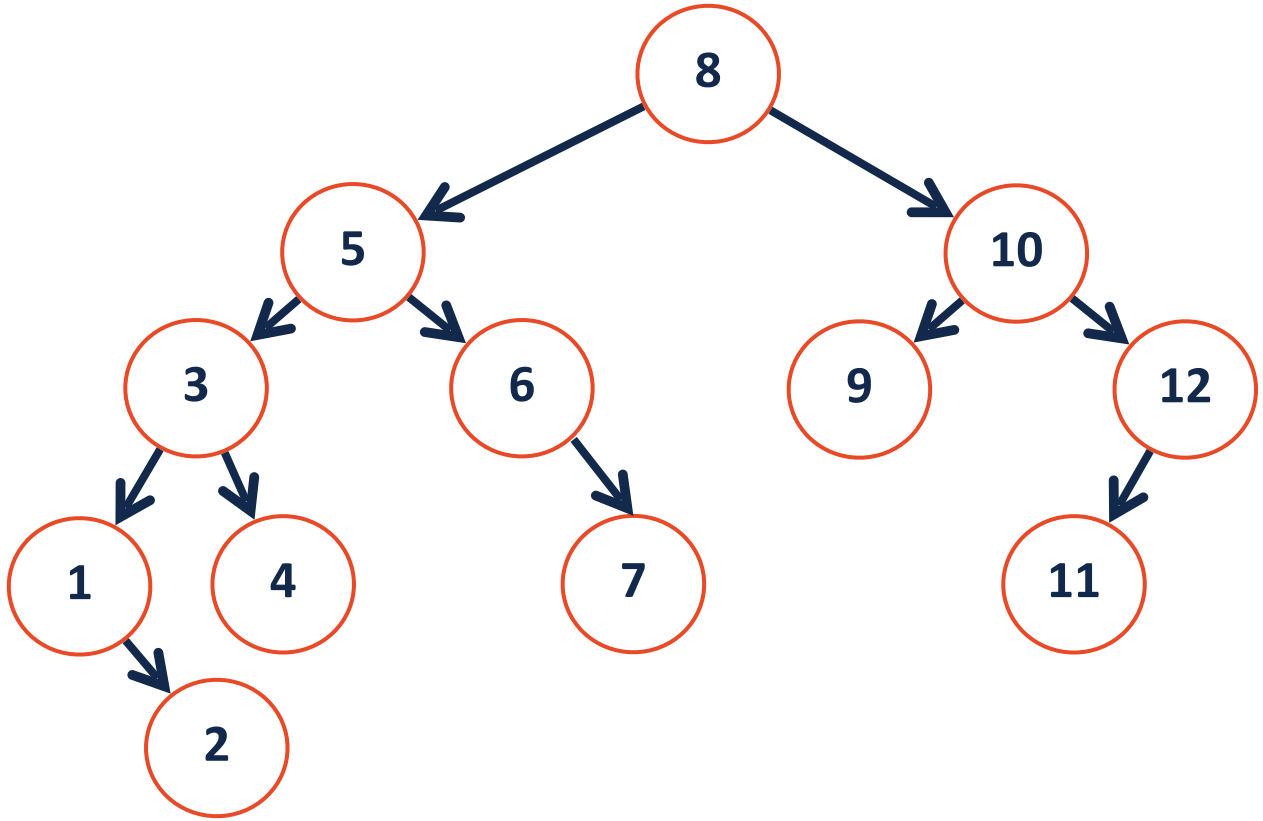
insert	removeMin
O(1)*	O(n)
O(1)	O(n)
O(n)	O(1)
O(n)	O(1)



Priority Queue Implementation

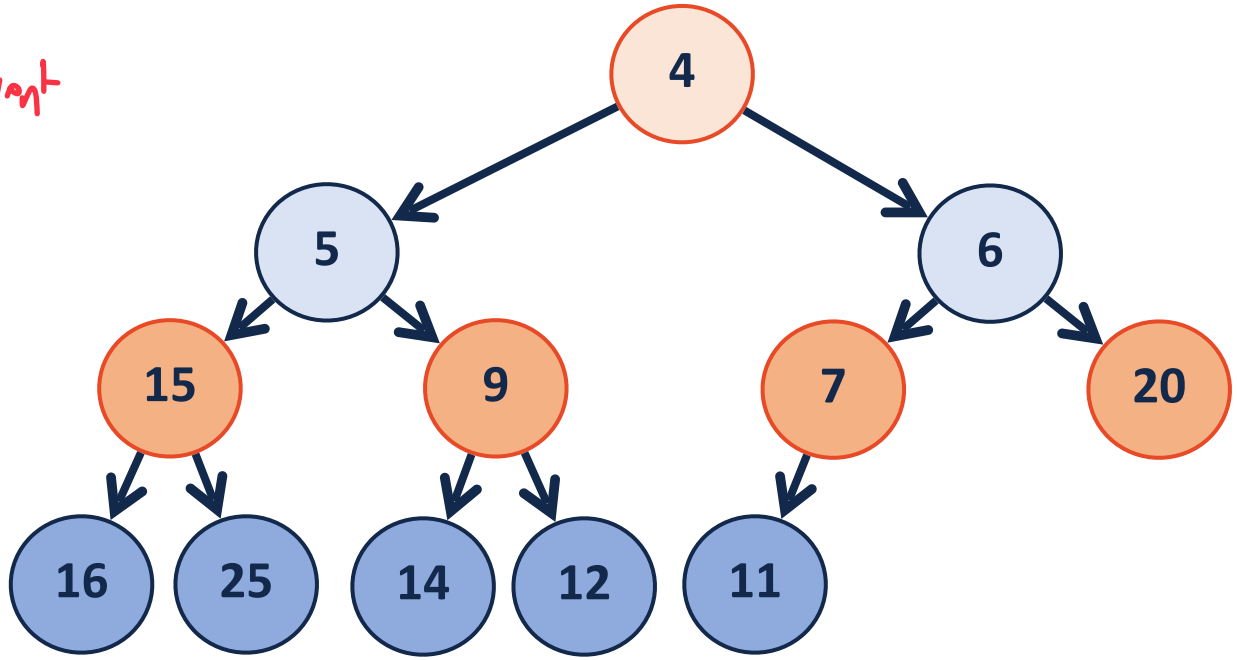
insert	removeMin
O(log n)	O(log n)

Goal: good at both



A different priority queue implementation...

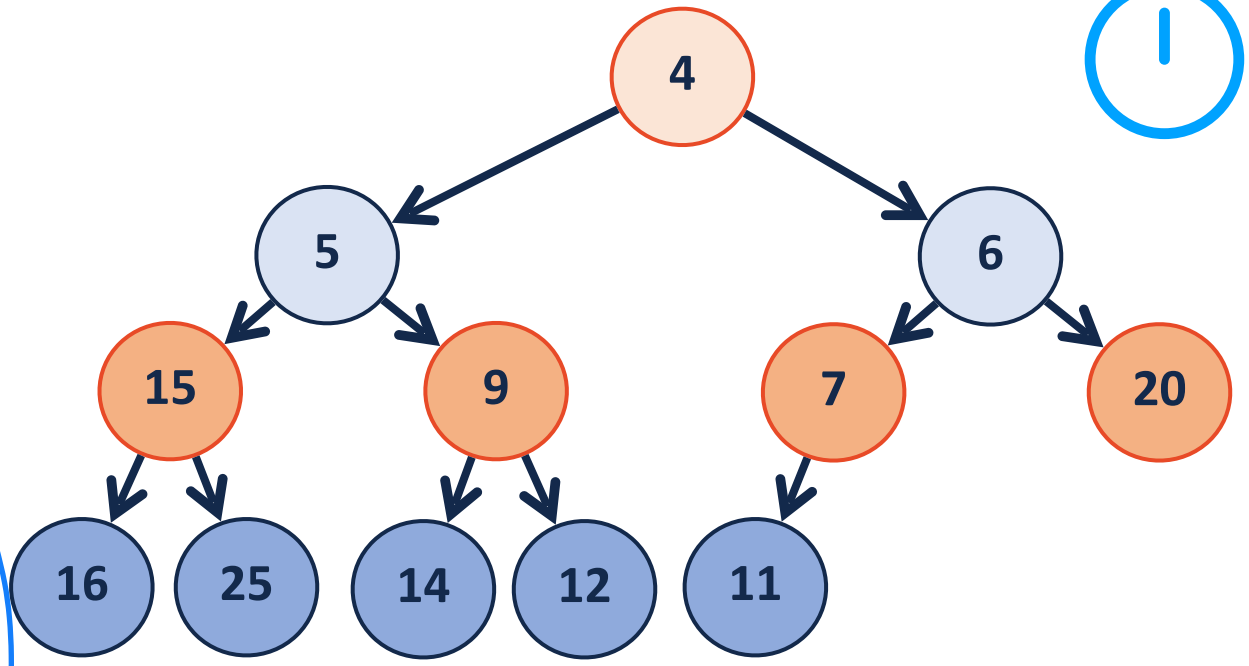
- Ordered tree but new order
  - Values of children larger than parent
- Complete binary tree *



(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.



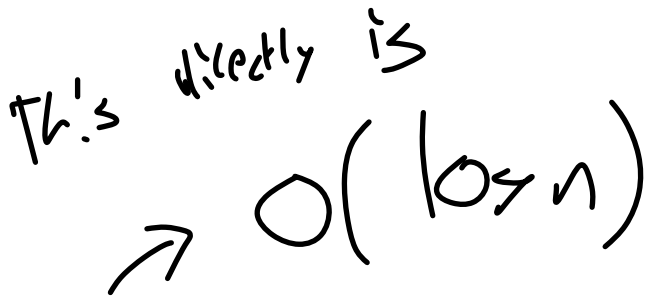
why is complete tree important? → Because I hate pointers / trees

Thinking conceptually: A tree without pointers

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

↳ A complete tree can be described w/o pointers!

This directly is



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

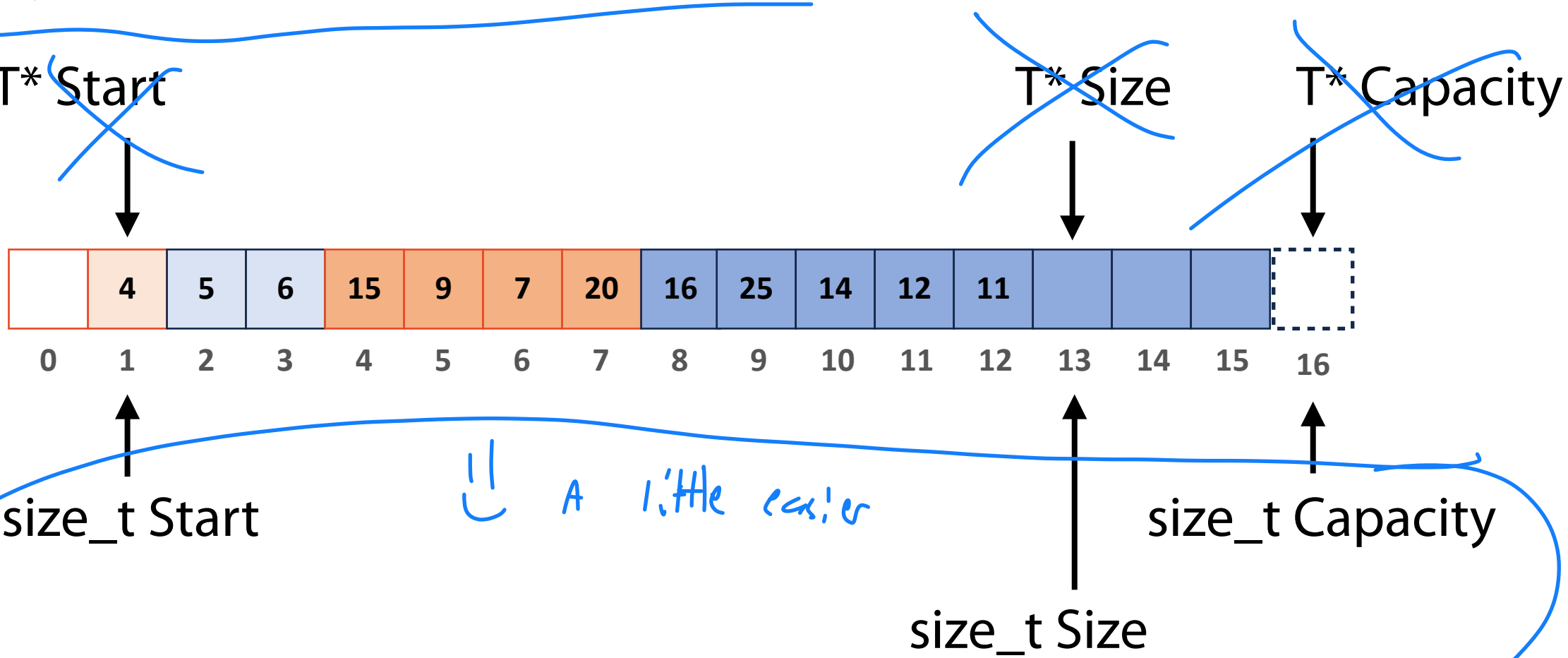
↳ $n \leq 2^{h+1} - 1$

Perfect tree of height h

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

Implementation of heap array

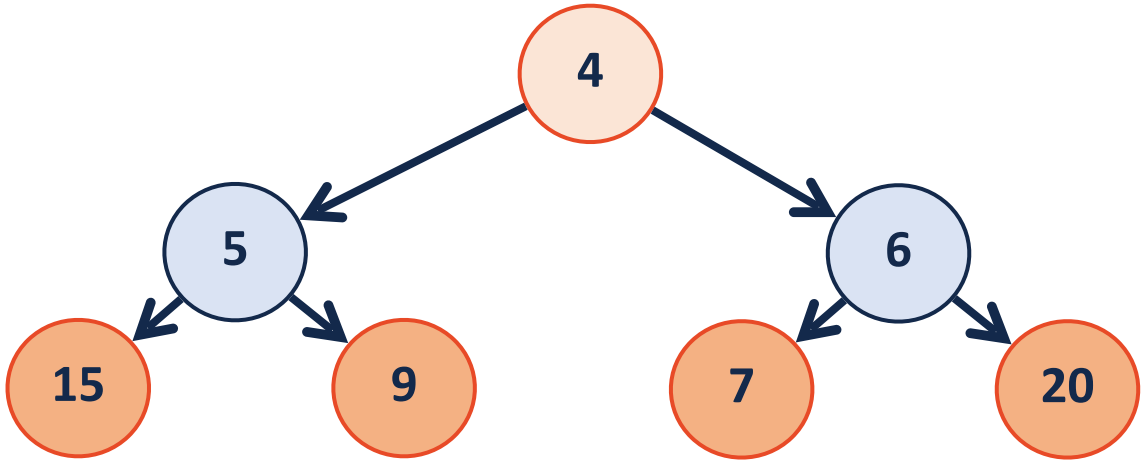
Array List (Pointer implementation)



Array List (Index implementation)

(min)Heap

Max of $2^{h+1}-1$ nodes

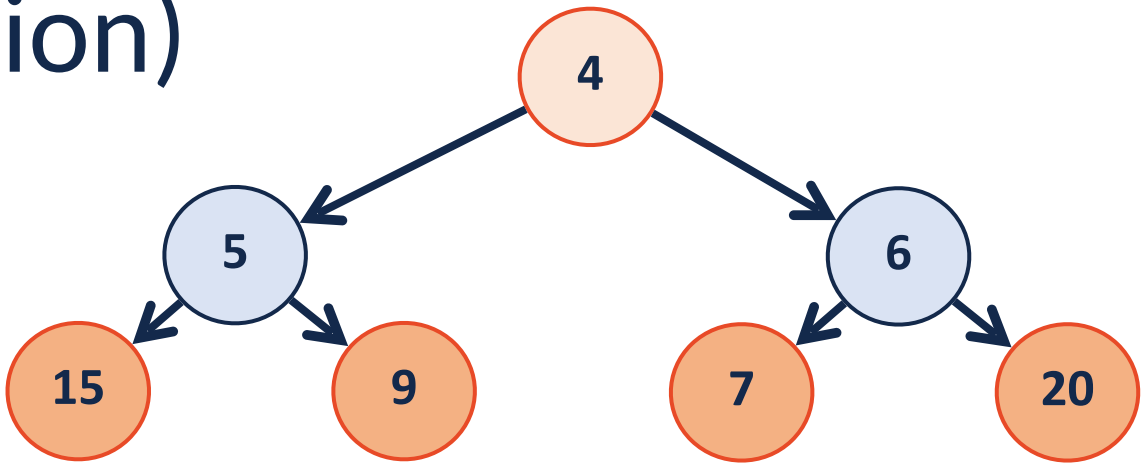


↓ lets allocate 2^{h+1} nodes!

4	5	6	15	9	7	20

(min)Heap (Design Decision)

Would you rather have A or B?



we will do this

Not true

B is better because back is more efficient in array?

A	4	5	6	15	9	7	20	B

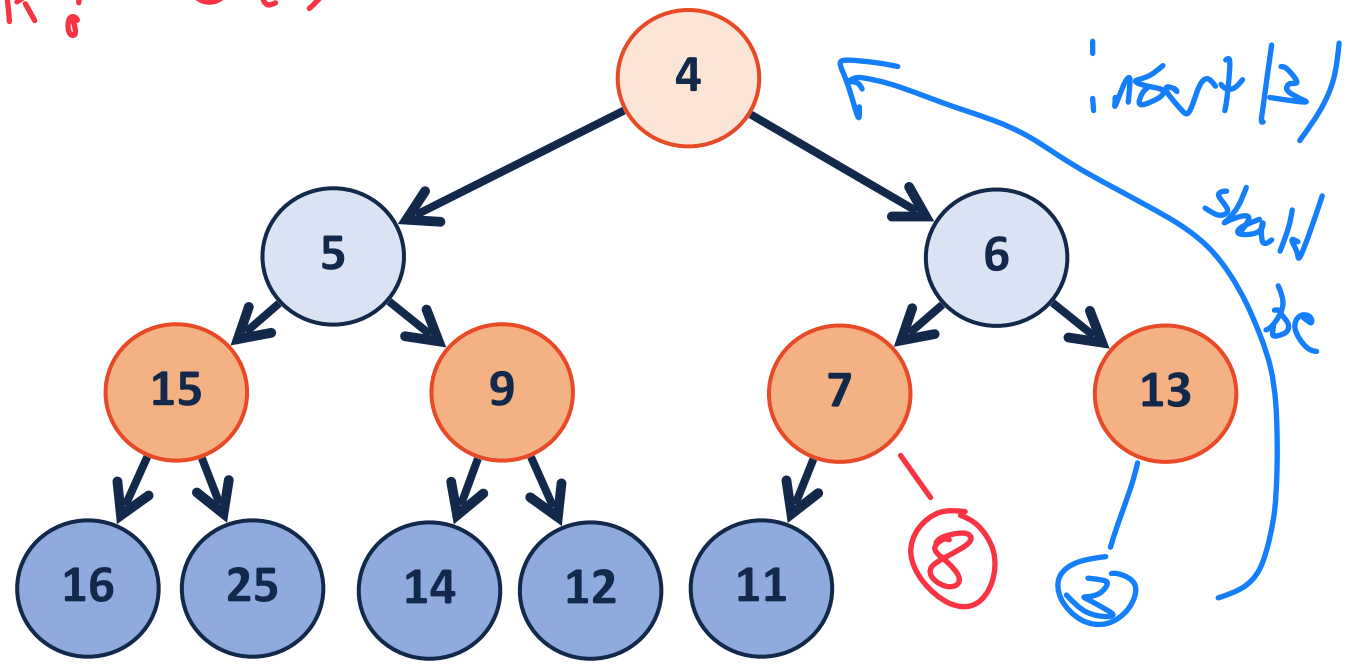
counter point: This is either always blank or... do is my last item

~~growArray~~

insert

Take advantage of insertBack! O(1) !!

insert(8)



Black in front makes math nicer

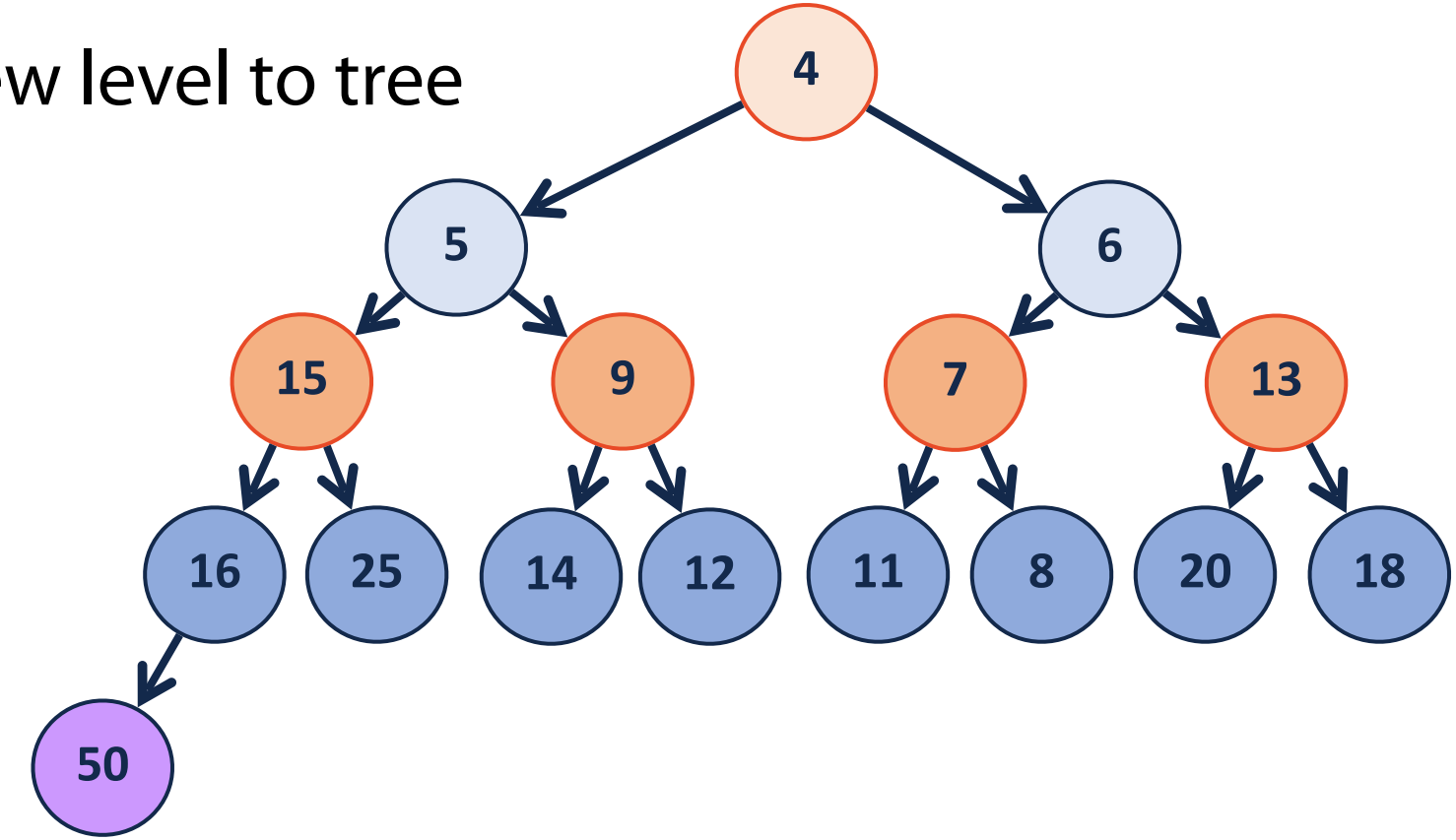
↑ ↑ (start at 0 or at 1)

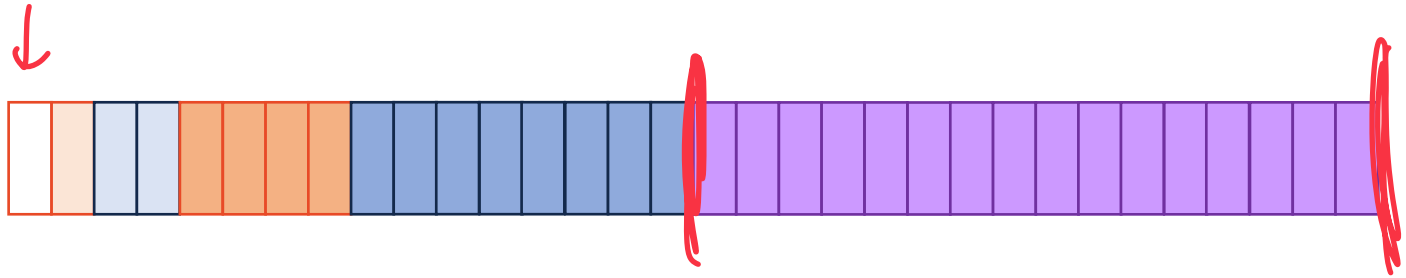
is not when it needs to be

In like five slides see that

growArray

Array doubling adds a new level to tree

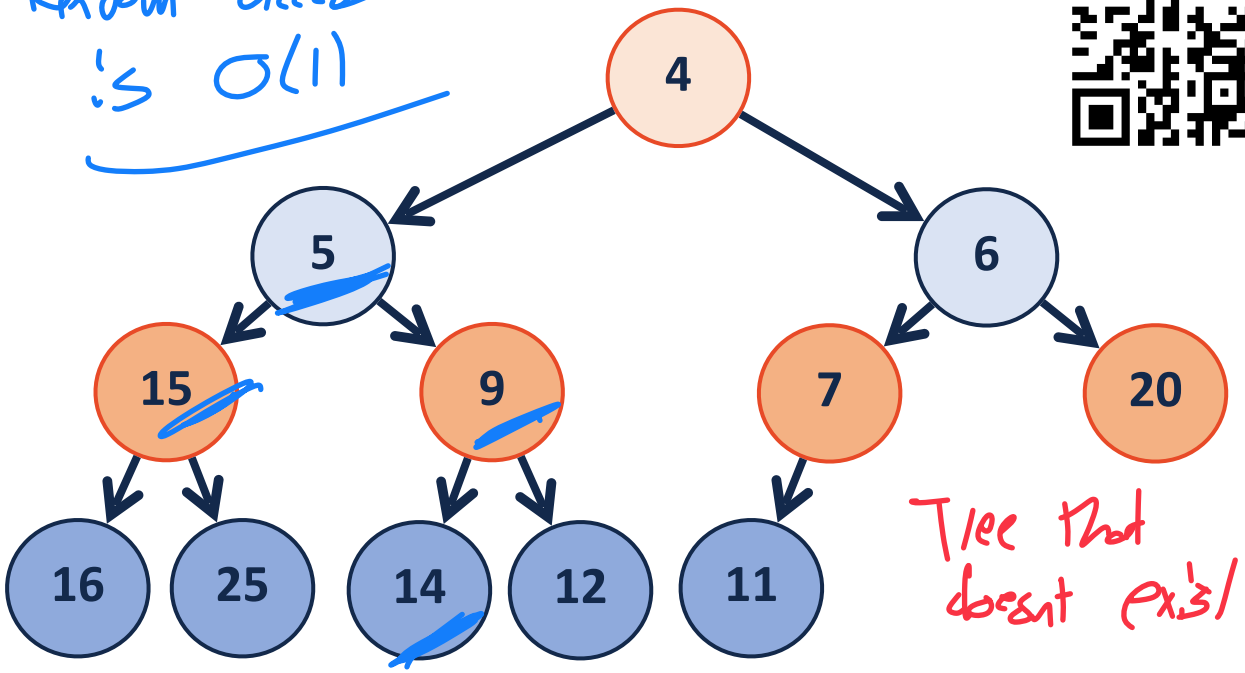




(min)Heap

Array Random access is O(1)

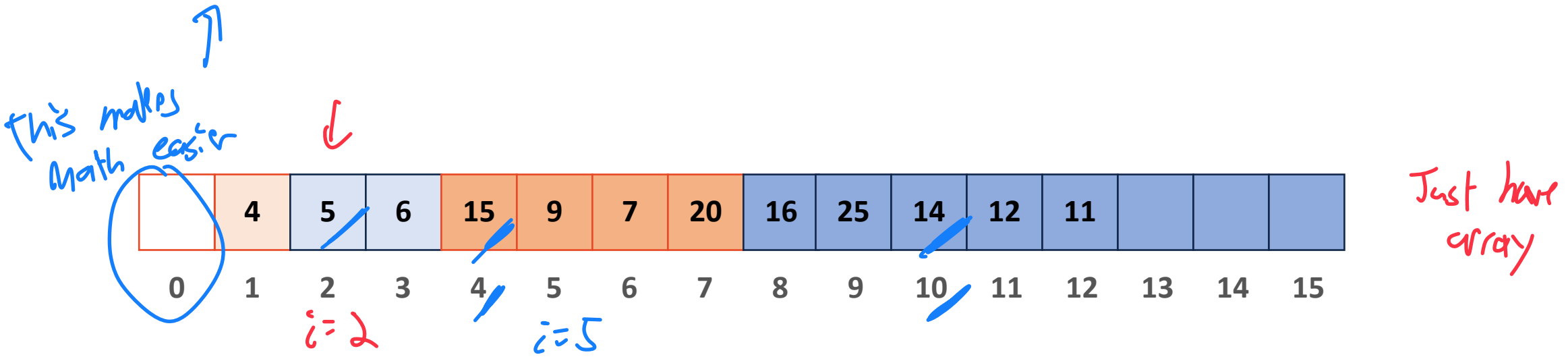
Join Code: 225



leftChild(i): 2i

rightChild(i): 2i+1

This makes math easier



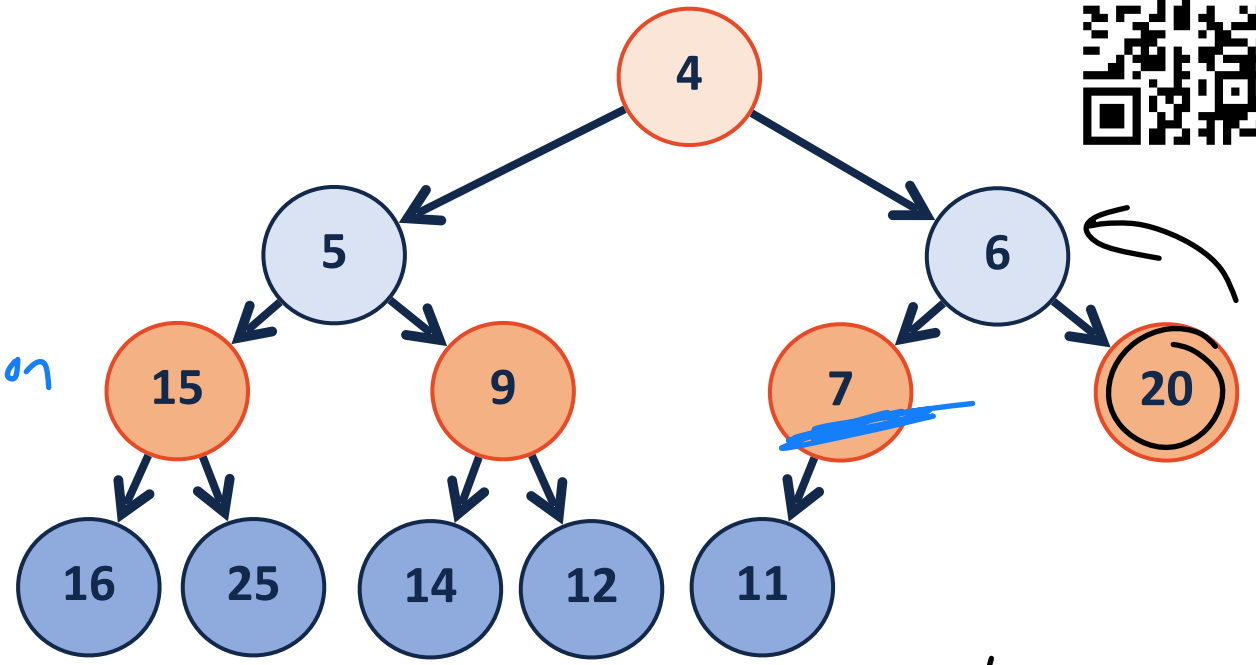
(min)Heap

Join Code: 225

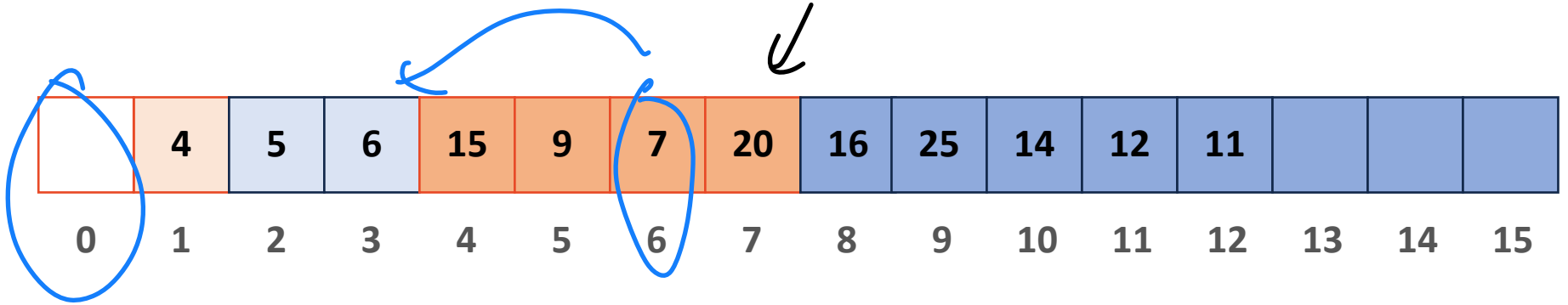
`parent(i):`

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

int $(\frac{i}{2})$ truncation



$i=7 \Rightarrow \frac{7}{2} = 3.5$ ↗ 4 ↘ 3



(min)Heap

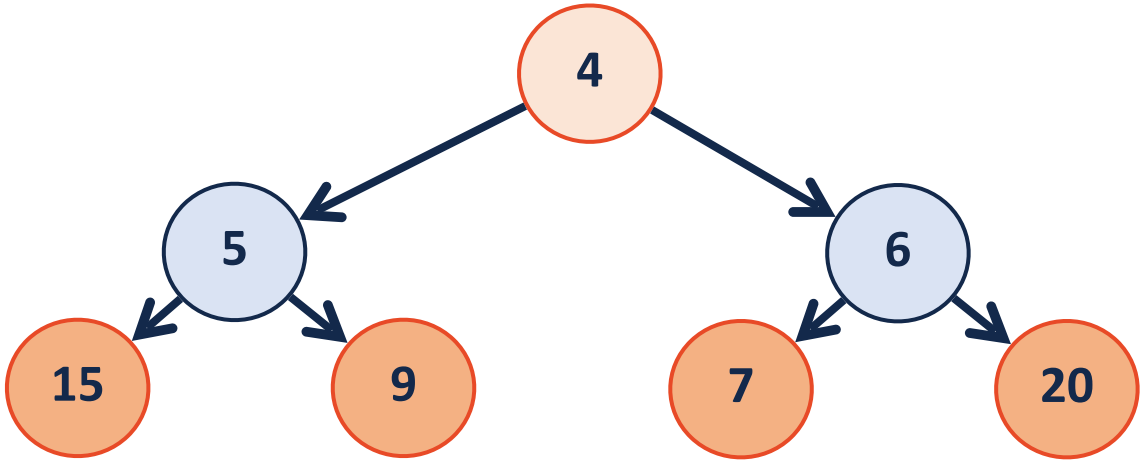
By storing as a complete tree, can avoid using pointers at all!

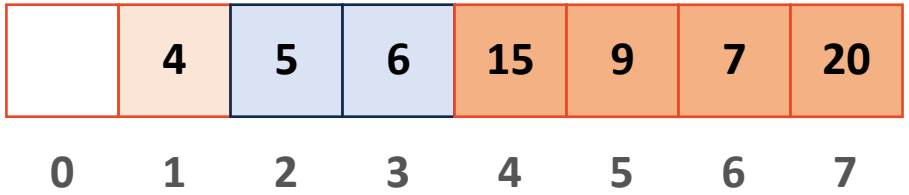
Can index from 0 or 1 (we will index from 1 in slides)

```
leftChild(i): 2i
```

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

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





(min)Heap ADT

Insert

RemoveMin

Constructor

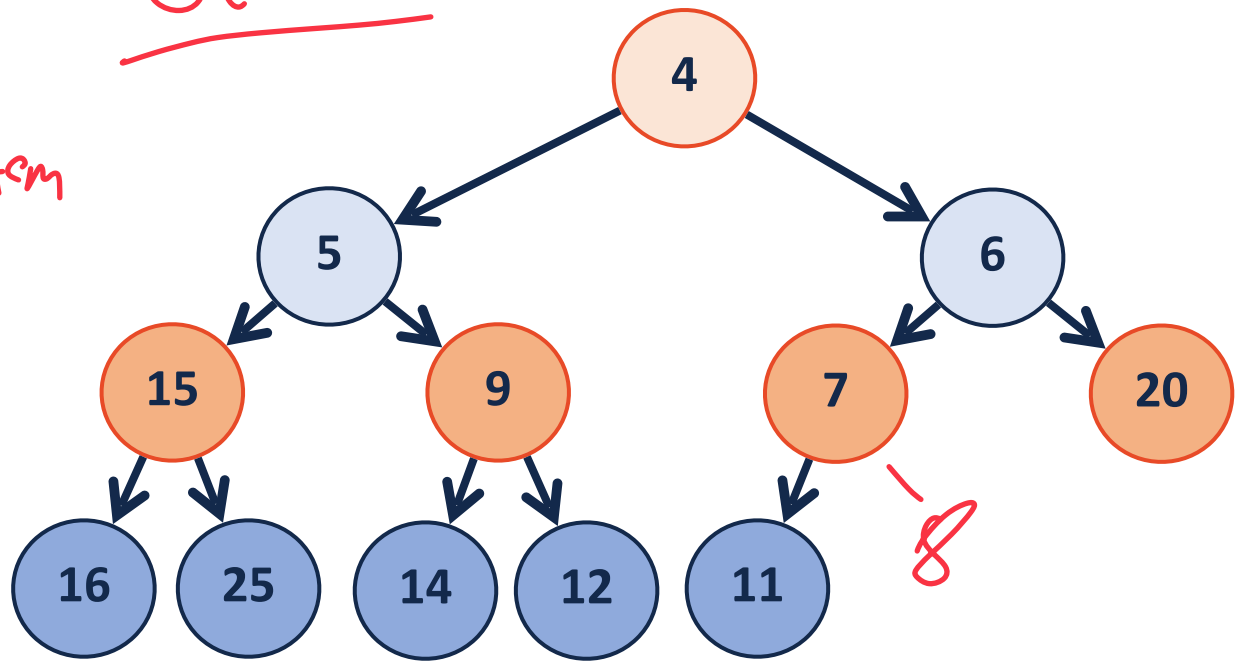
insert

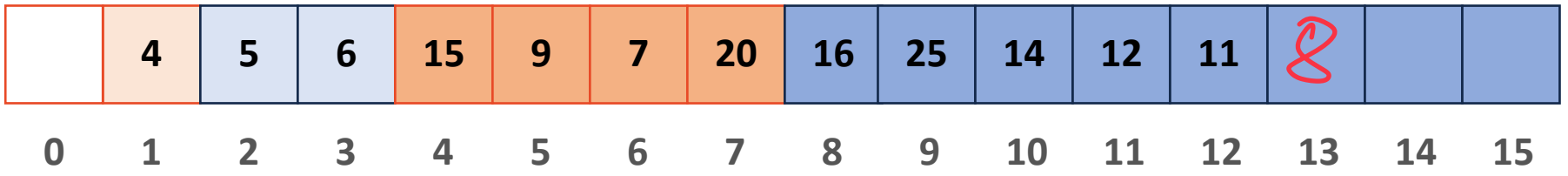
Insert(8)

1) Insert at back of array

2) correct minheap if new item breaks properties

O(1)





insert

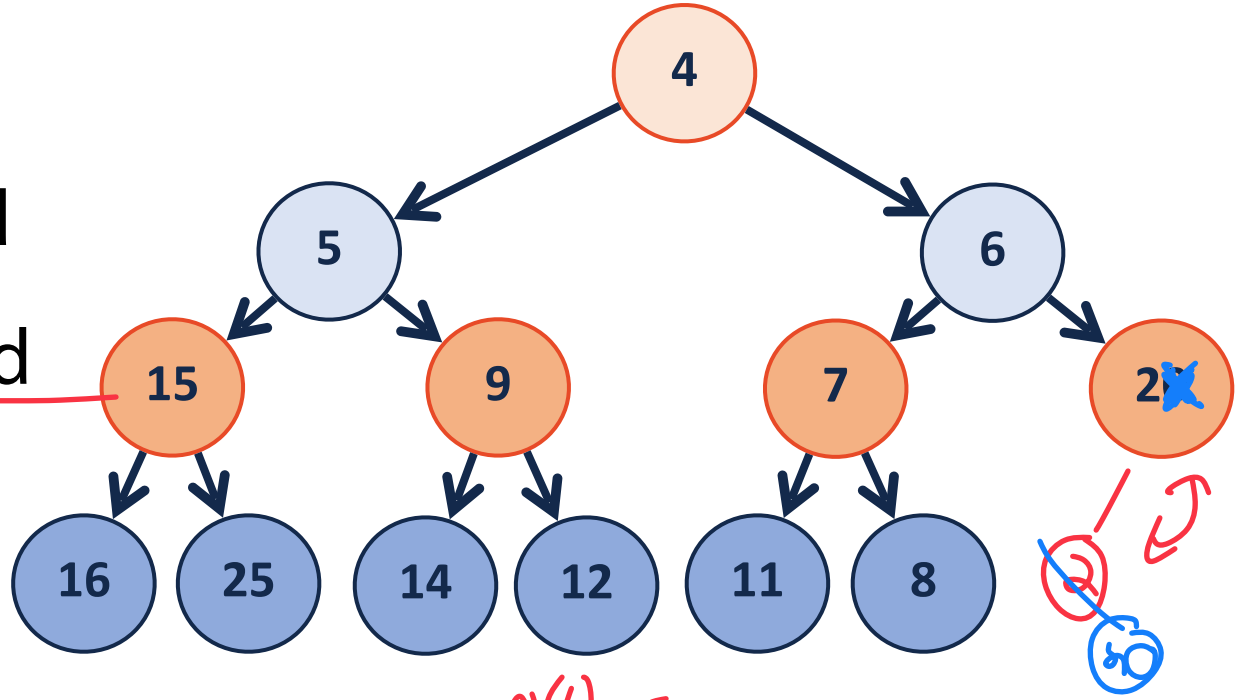
Insert(2)

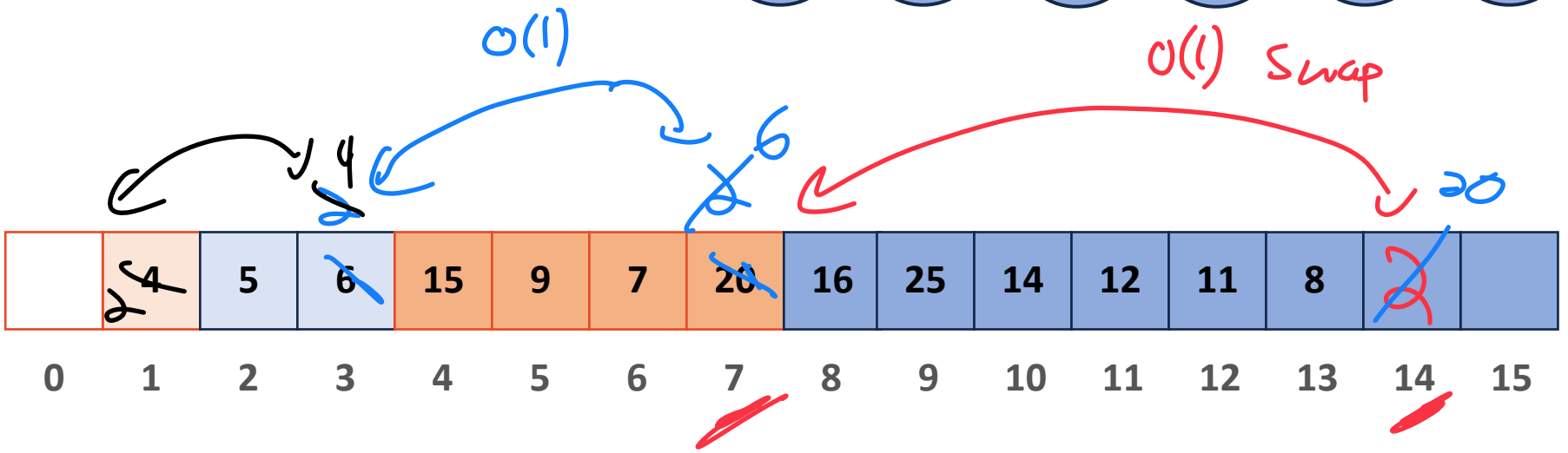
1) Insert at end of array

2) Check if minHeap still valid

3) Swap with parent if needed

Steps 2 and 3 are recursive!





insert

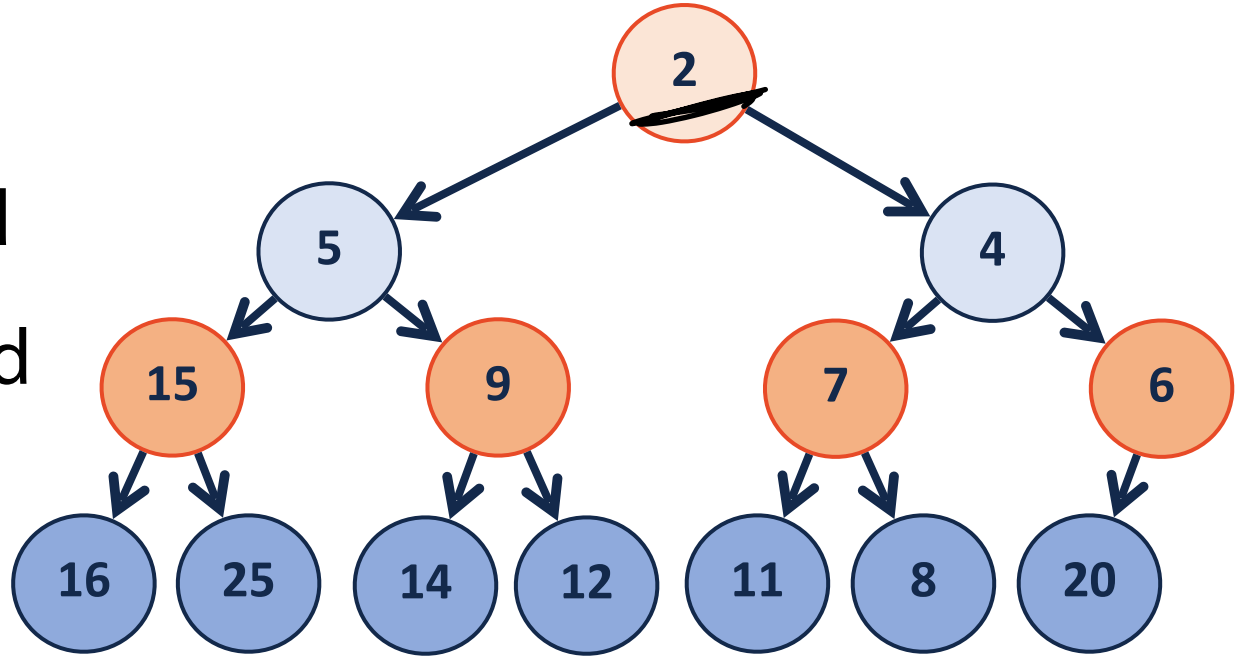
[After] Insert(2)

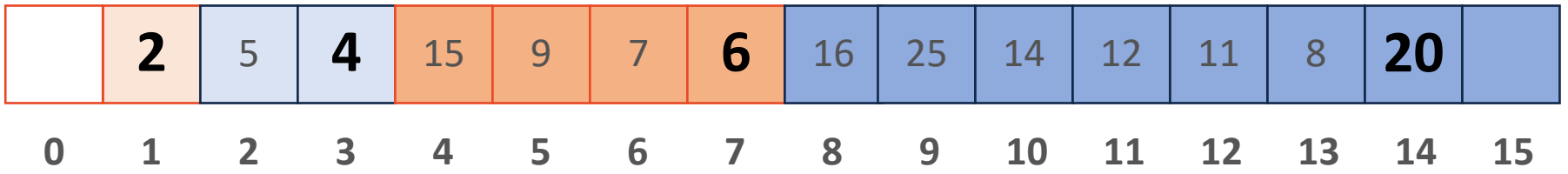
1) Insert at end of array

2) Check if minHeap still valid

3) Swap with parent if needed

Steps 2 and 3 are recursive!





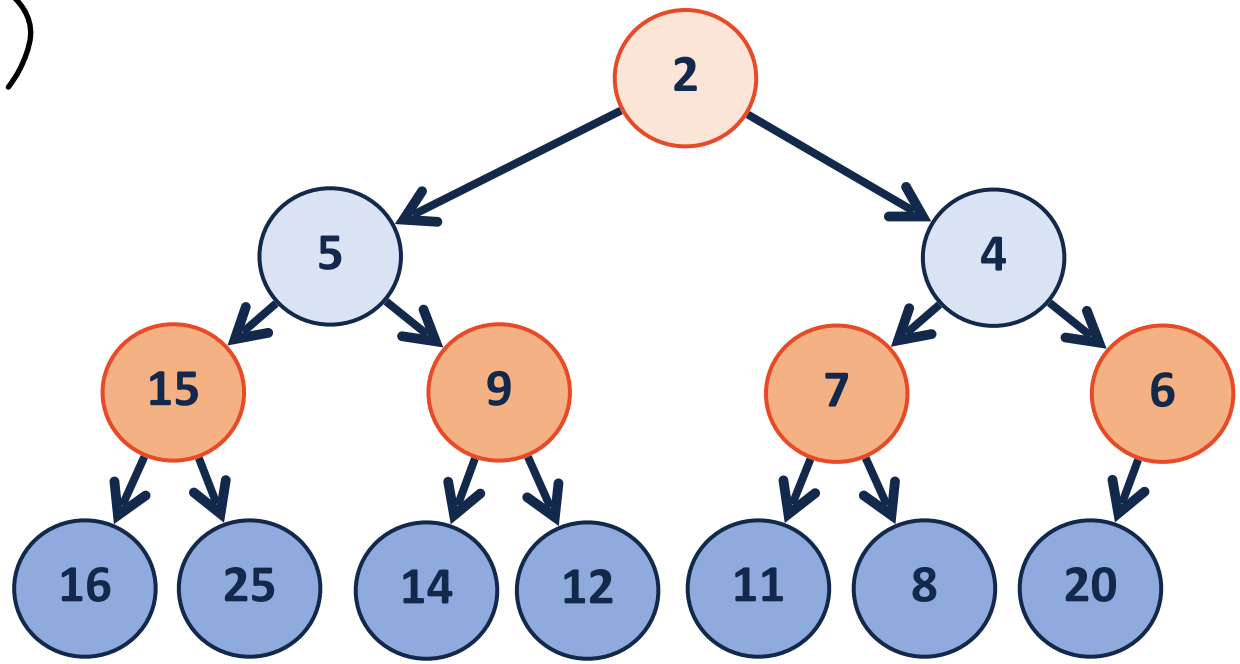
insert

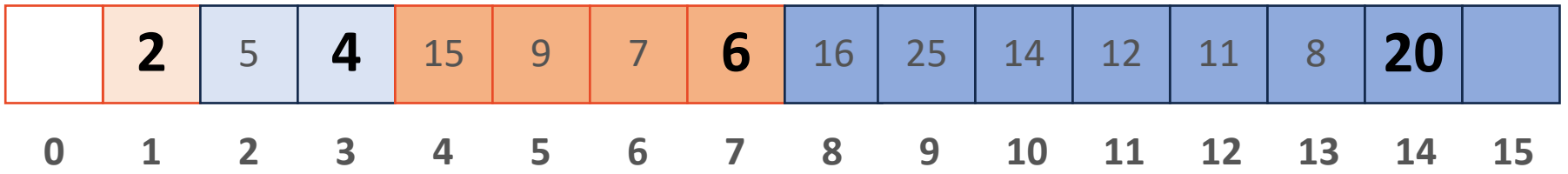
[After] Insert(2)

What is my height? $O(\log n)$

Number of swaps? $O(\log n)$

↳ height swaps





insert

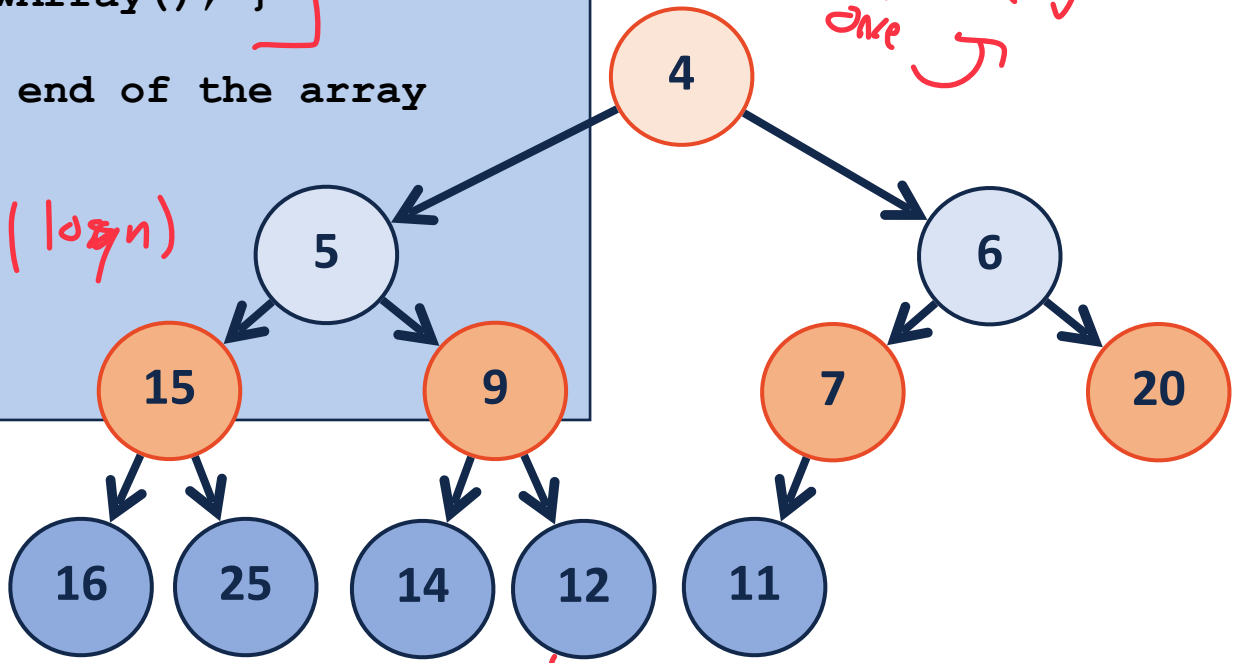
```
1  template <class T>
2  void Heap<T>::_insert(const T & key) {
3    // Check to ensure there's space to insert an element
4    // ...if not, grow the array
5    if ( size_ == capacity_ ) { _growArray(); }
6
7    // Insert the new element at the end of the array
8    item_[size_++] = key;
9
10  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

array doubling

O(n) / O(1)* once

O(1)

O(log n)



item_ 2

size 2++

insert - heapifyUp

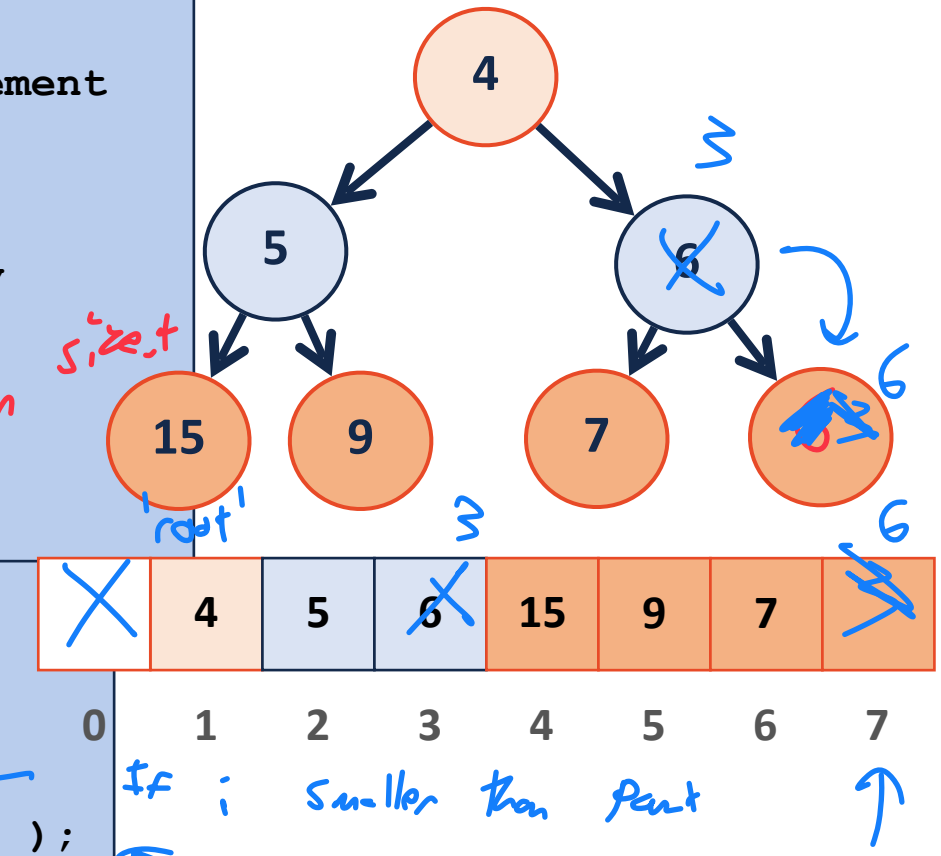
Join Code: 225


```
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_ - 1);
12 }
```

size_ is an size_t

```
1  template <class T>
2  void Heap<T>::_heapifyUp( size_t index ) {
3
4    if ( index > 1 ) {
5      if ( item_[index] < item_[ parent(index) ] ) {
6        std::swap( item_[index], item_[ parent(index) ] );
7
8        _heapifyUp( parent(index) );
9      }
10   }
11 }
```

index

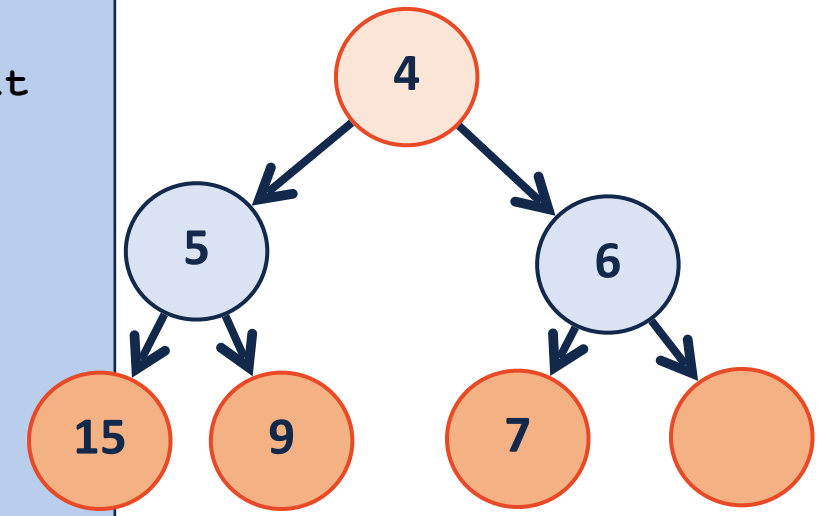


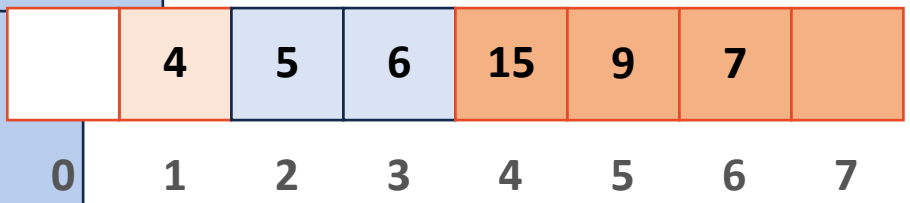
insert - heapifyUp

```
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_ - 1);
12 }
```

```
1  template <class T>
2  void Heap<T>::_heapifyUp( size_t index ) {
3
4    if ( index > 1 ) {
5      if ( item_[index] < item_[ parent(index) ] ) {
6        std::swap( item_[index], item_[ parent(index) ] );
7
8        _heapifyUp( parent(index) ); // index / 2;
9      }
10   }
11 }
```

The same





removeMin

What is the Big O of array remove?

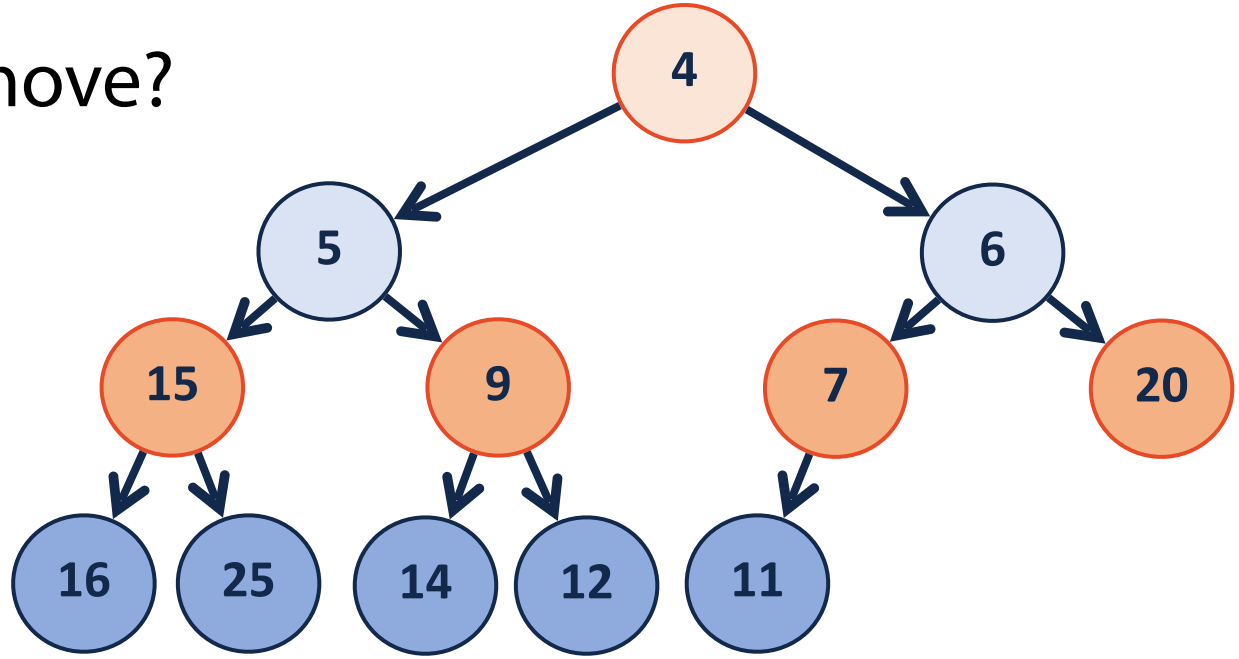
O(n) if remove front

What else can we do?

Goal: Remove min

replace rt and min

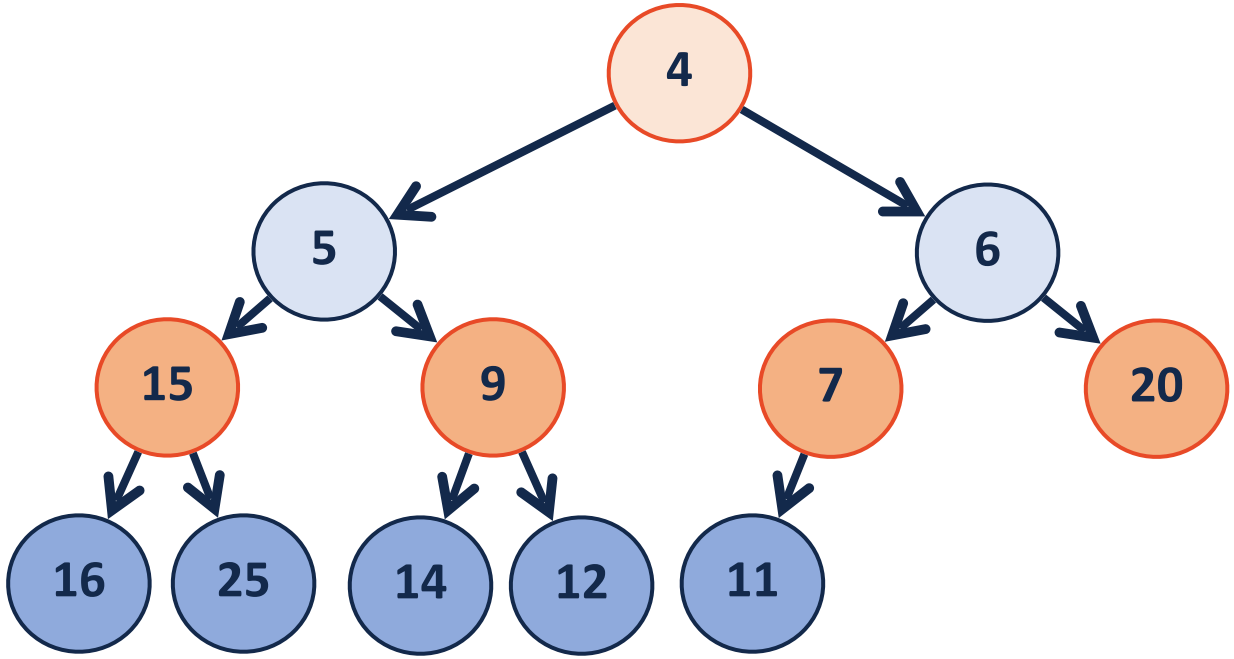
Do both at same time in < O(n)



Swap can do this!

min is front item! :)

removeMin



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