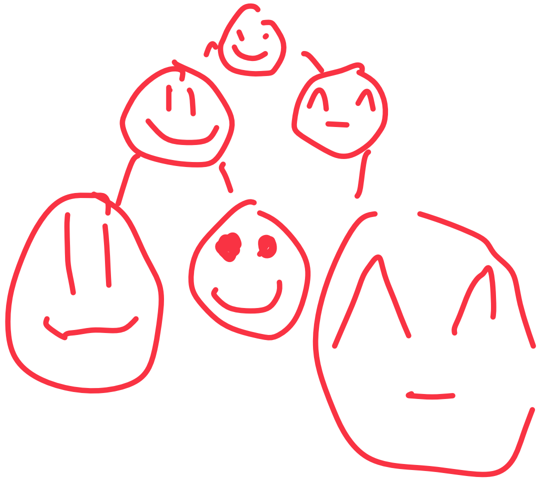
Data Structures

Heaps

CS 225
Brad Solomon & G Carl Evans

October 11, 2023



Department of Computer Science

Announcements

Reminder: Exam 3 October 16-18

Drop deadline: October 13

MP_Traversals out now! ← Start early!

As of this morning, less than 25% of students filled out IEF!

↳ 70% ☹

Learning Objectives

Introduce the min 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?

↳ priority queue

Insert items

remove Min (and return)

Interface

↳ Org is that min element is front / top

Implementation ← Saw this w/ Array / Sorted
LL \ unsorted

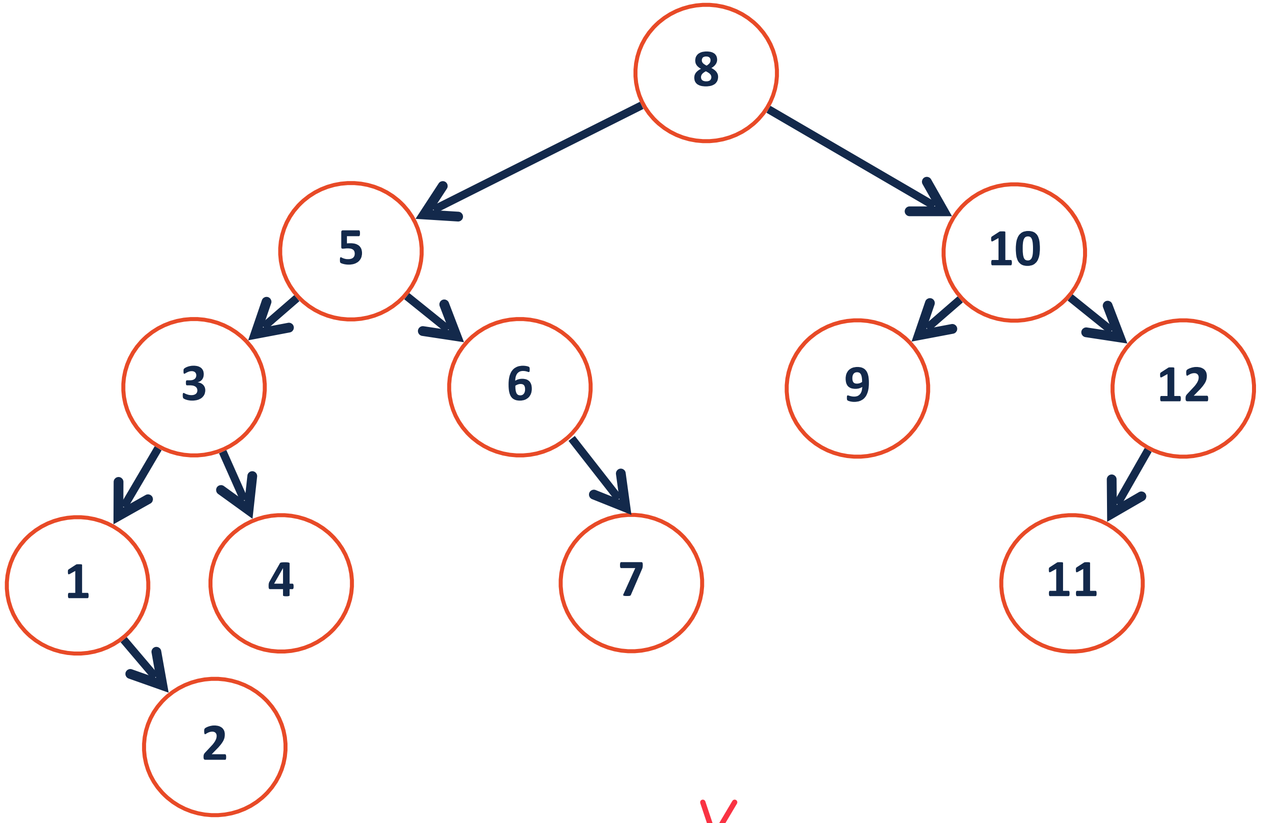
O(1) | O(n)

for today is minheap!

Priority Queue Implementation

AVL

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



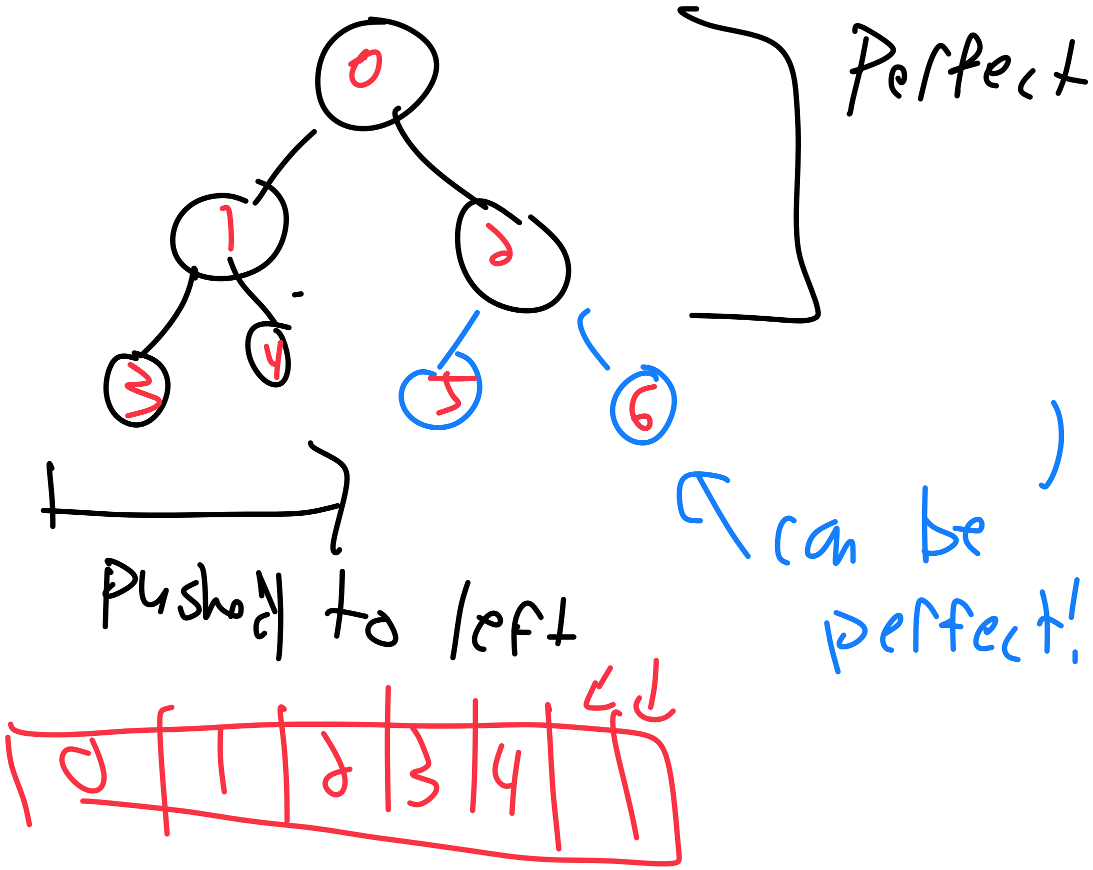
1) Dont need pointers in my life!

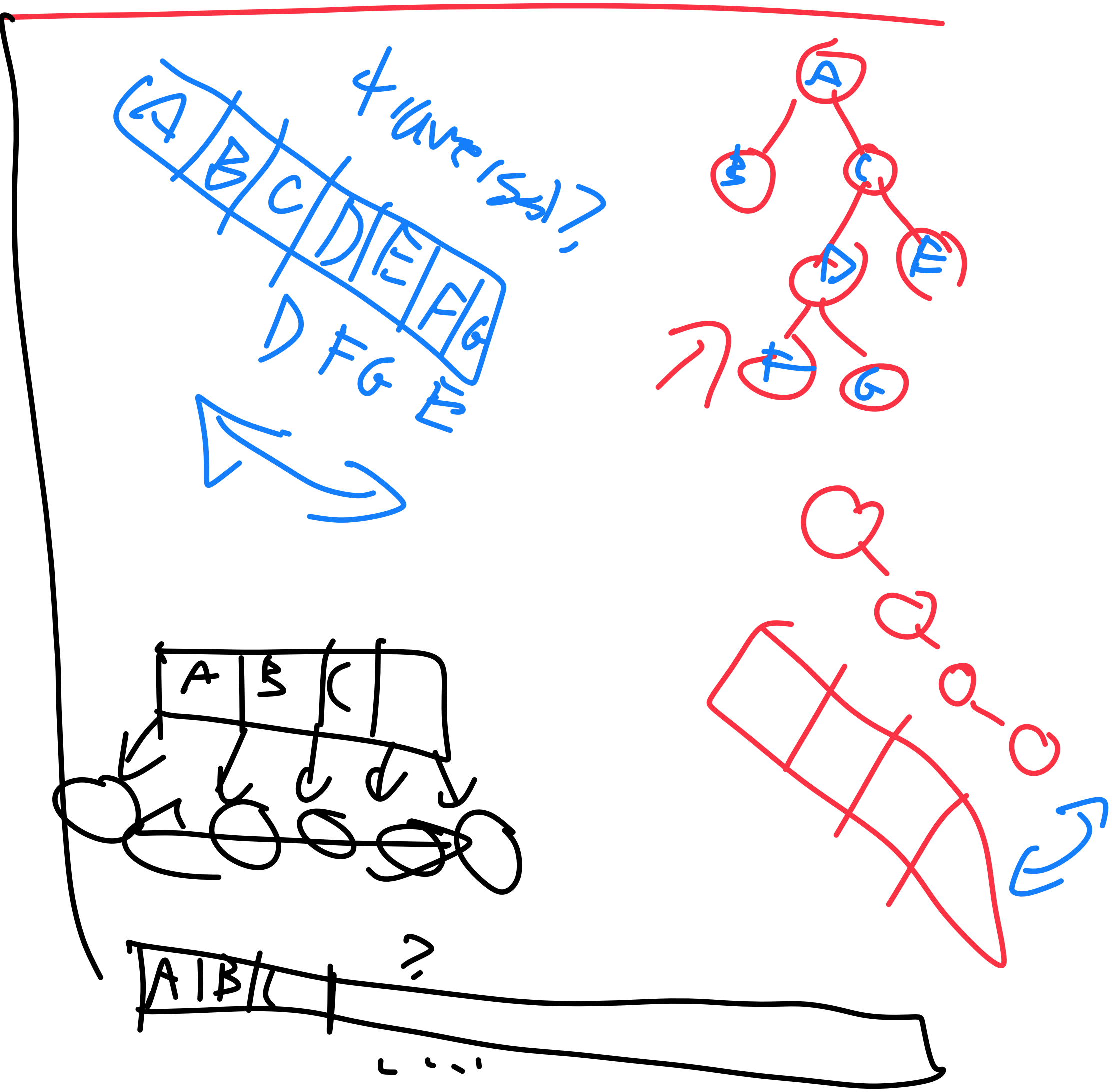
2) Constructing is a pain $O(n \log n)$

Thinking conceptually: A tree without pointers

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

A complete tree!





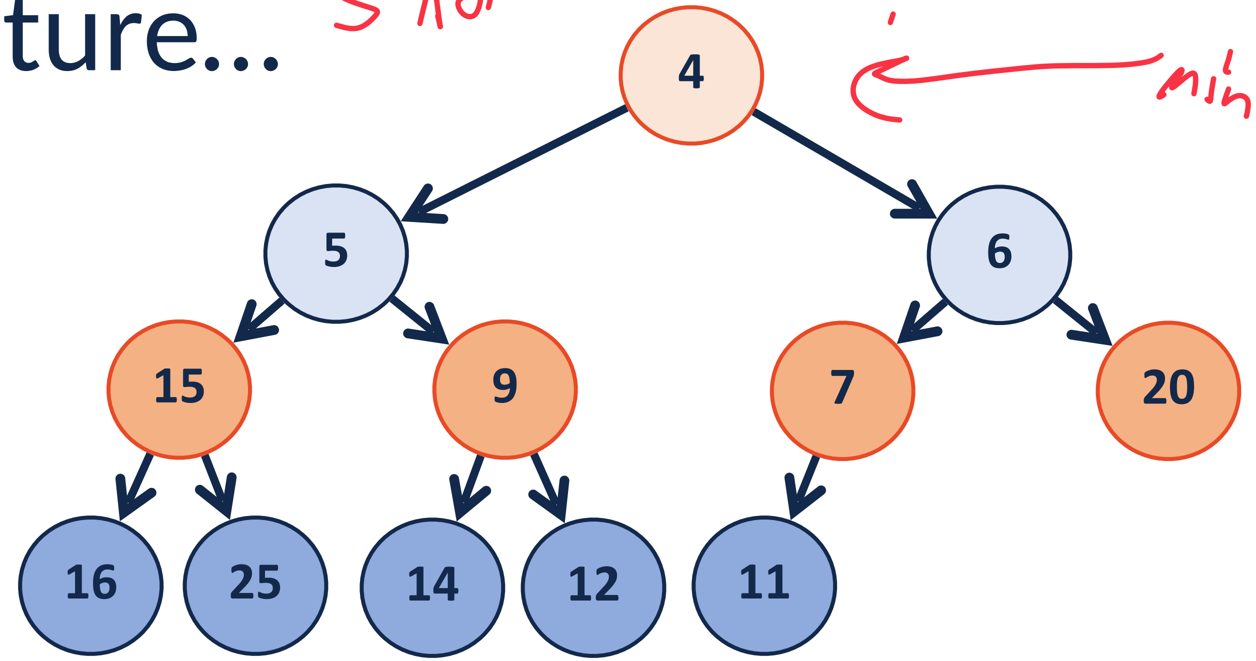
Another possibly structure…

↳ for priority queue

1) Binary tree (Complete)

2) Keys are ordered s.t. descendants of a key are larger than key

5 not smaller than 4!



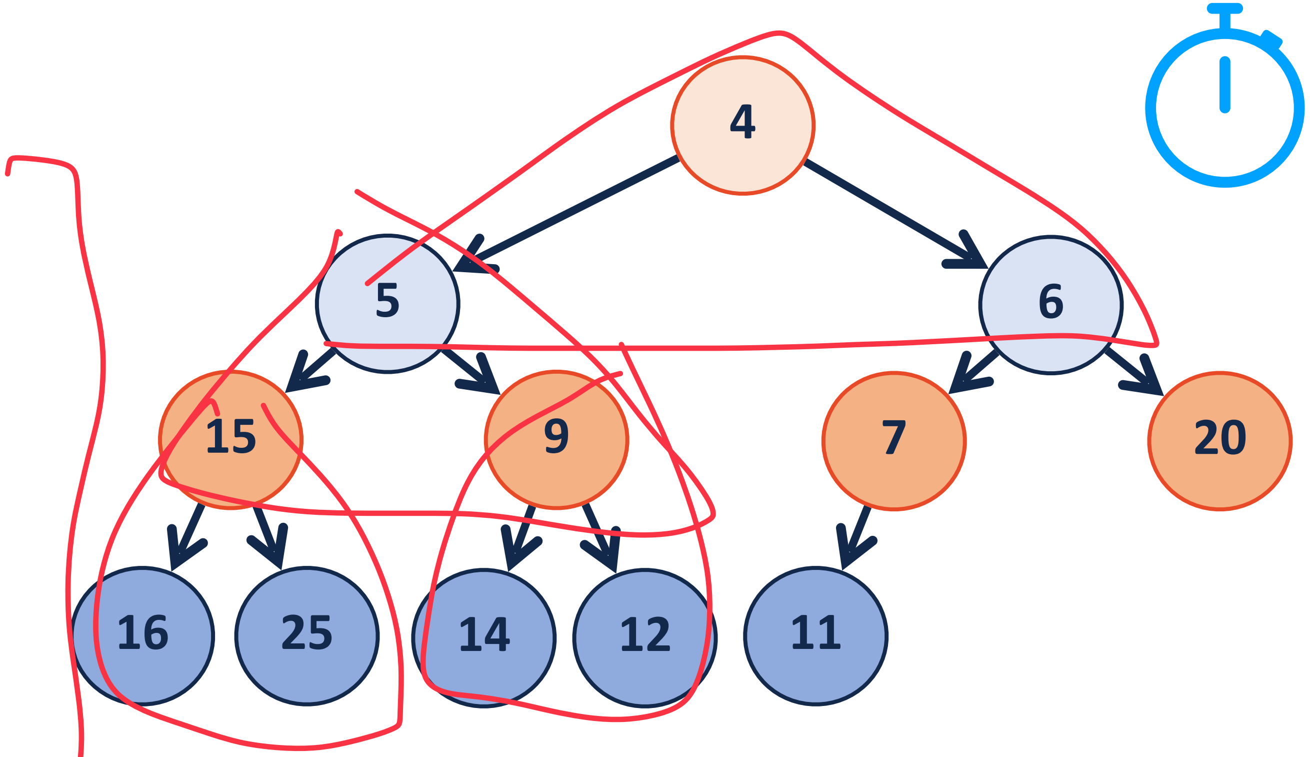
min

Children are larger than my parent (current)

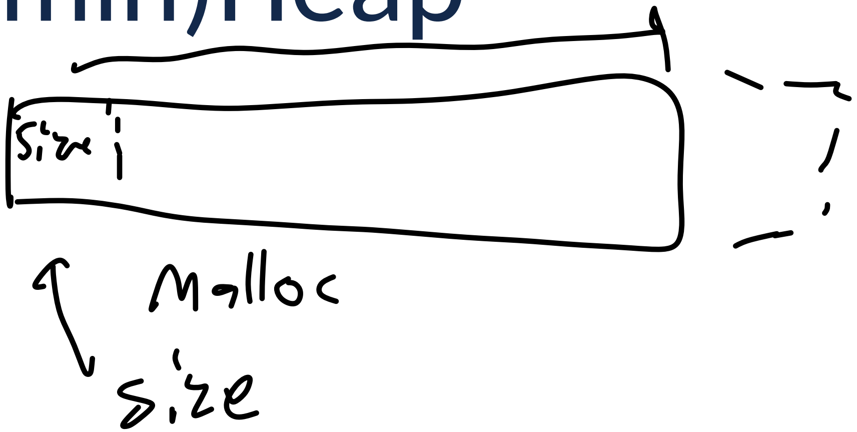
(min)Heap

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.



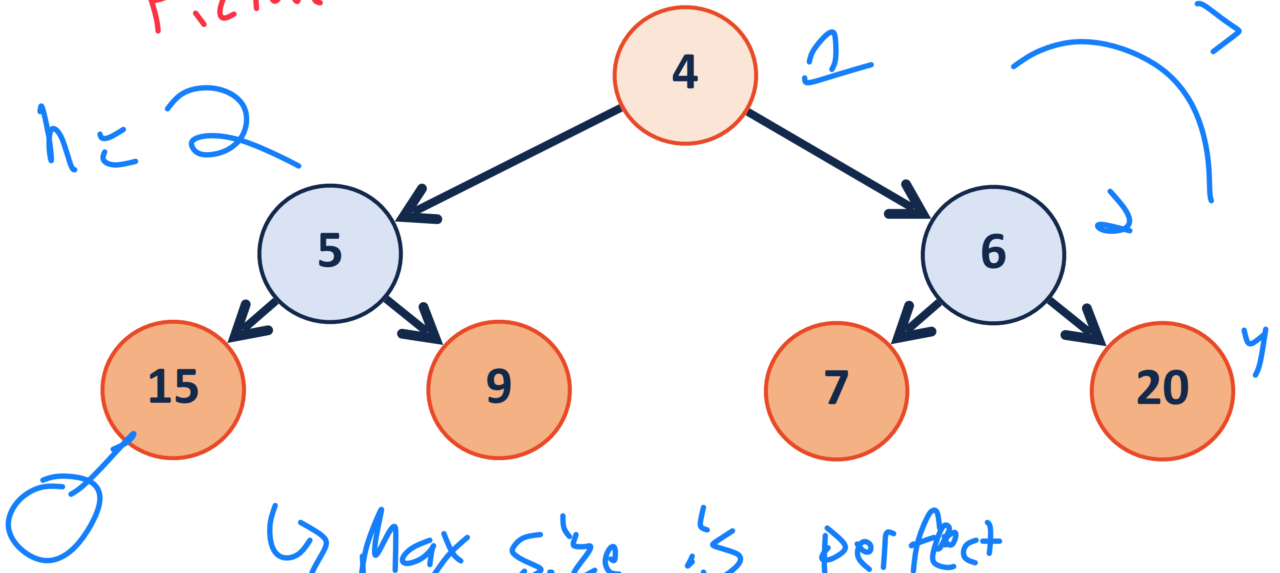
(min)Heap



Malloc

size

Picture for your convenience



↳ Max size is perfect tree height h

# of nodes: $2^{(h+1)}-1$

root value is 1

empty value @ 0

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

$2^{(h+1)}$ bits

4	5	6	15	9	7	20

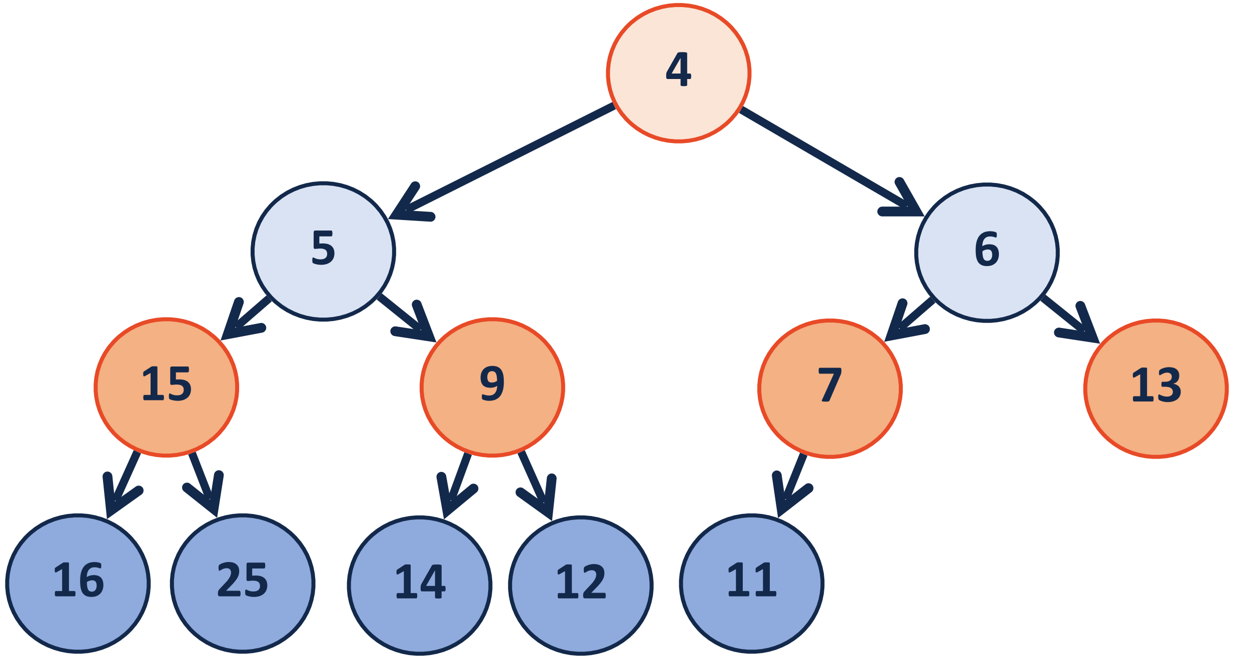
2^{h+1}

:)

2^{h+2}

The actual storage

growArray



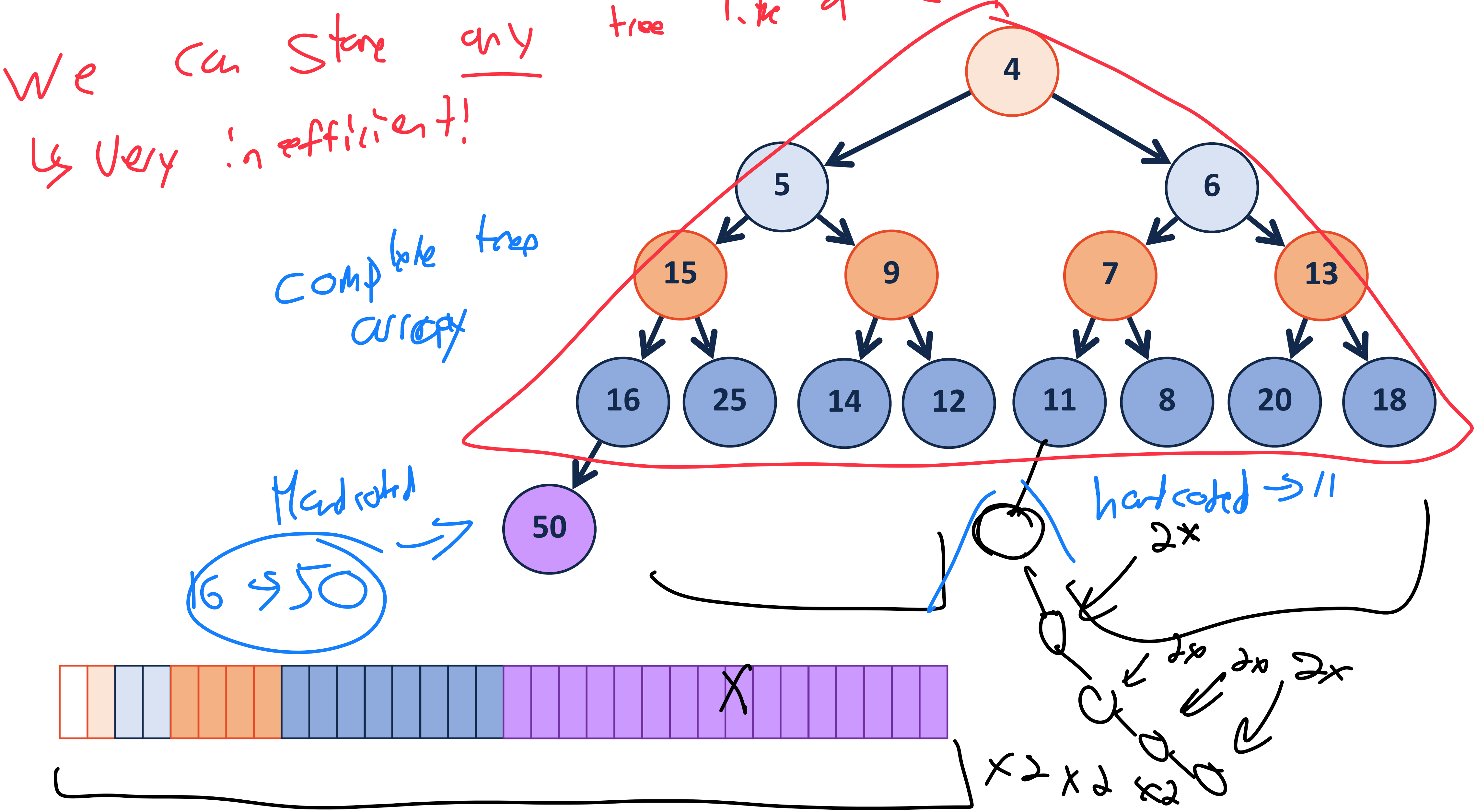
	4	5	6	15	9	7	20	16	25	14	12	11			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

~~growArray~~

Good question slide

We can store any tree like a complete tree

is very inefficient!



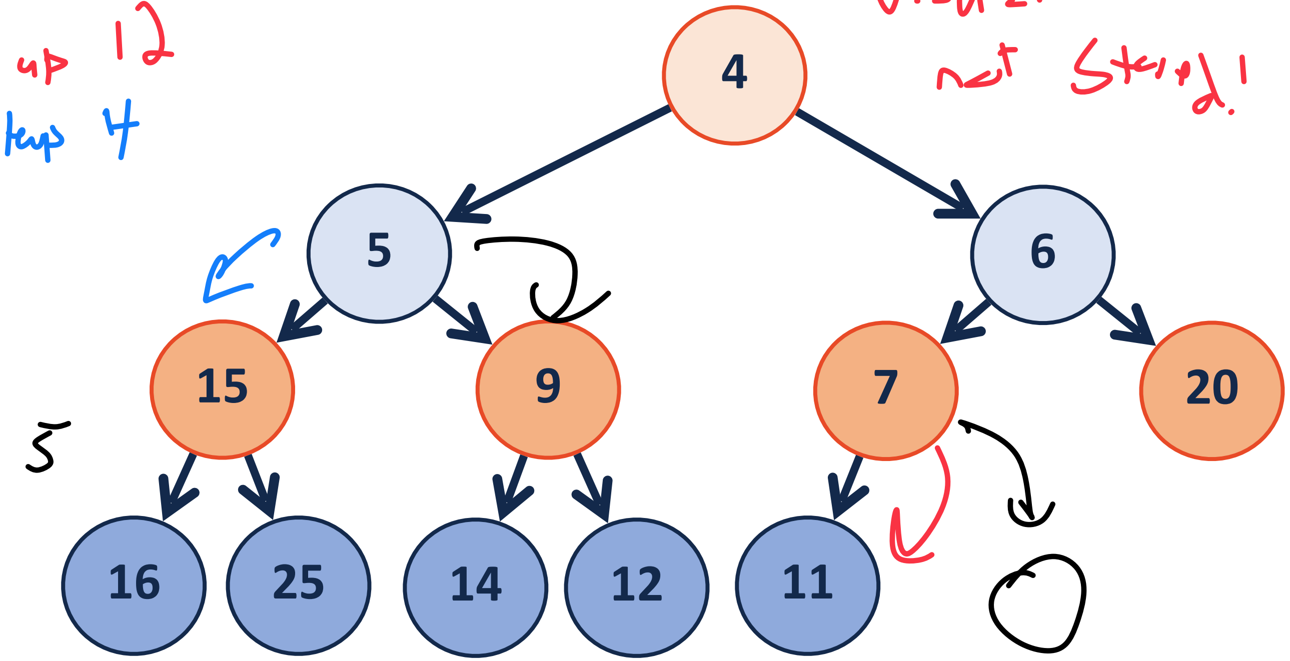
(min)Heap

leftChild(i): i = 6 → Look up 12
i = 2 → Look up 4

↳ 2i

rightChild(i): i = 2 → look up 5
i = 6 → 13

↳ 2i + 1



visual not stored!

	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

Design decision! #

(min)Heap

$i = 14$ or 15 → parent(i) = 7

parent(i):

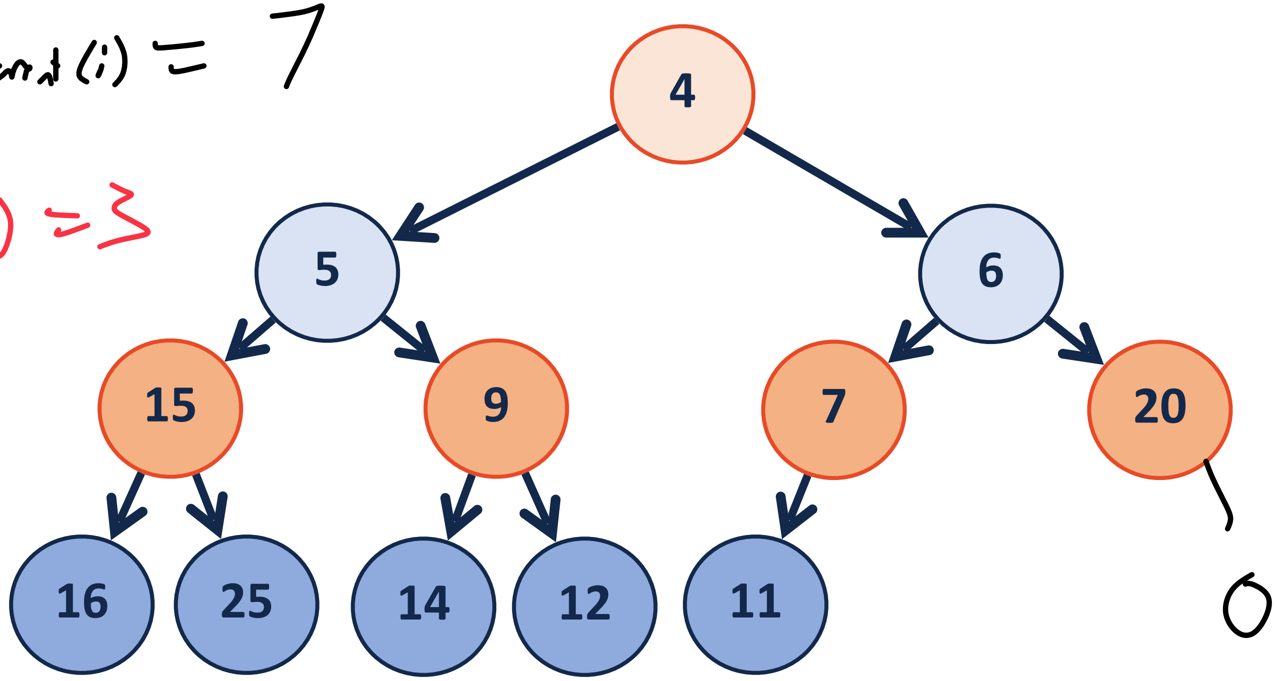
$i = 6$ or 7 → parent(i) = 3

If even: $i/2$

odd: $(i-1)/2$

$\lfloor i/2 \rfloor$

↑ floor of $i/2$



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			

(min)Heap

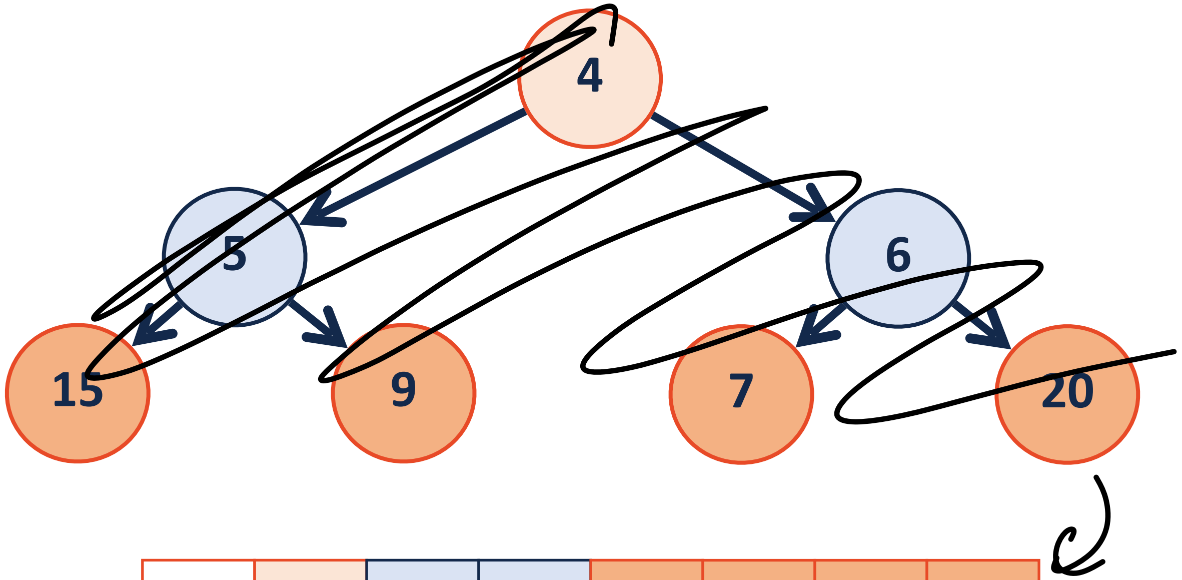
By storing as a complete tree, can avoid using pointers at all!

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)`



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

(min)Heap ADT → implementing priority Queue

Insert

RemoveMin

Constructor

insert

Insert(8)

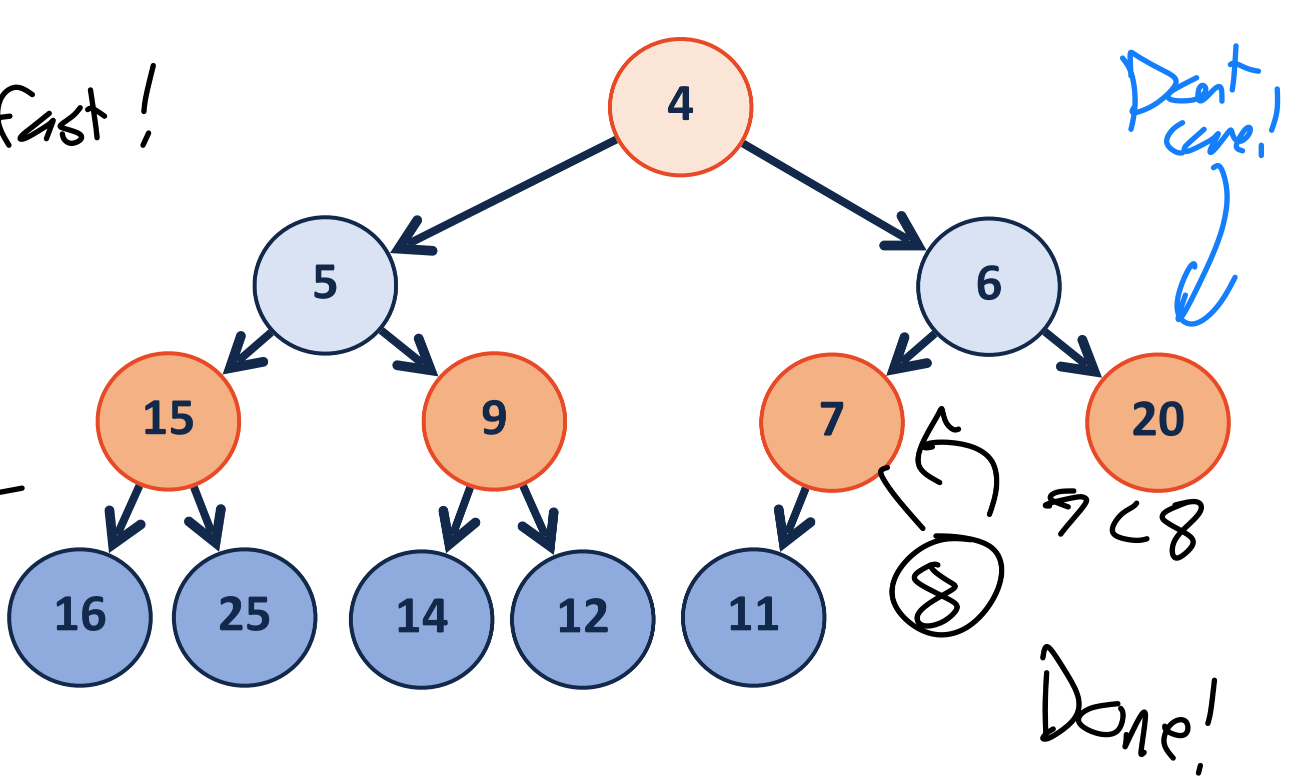
1) Array insert generally slow

↳ insert at end generally fast!

1) Insert value at end of array

2) check if insert broke heap

↳ equiv to check if parent smaller



	4	5	6	15	9	7	20	16	25	14	12	11	8		

insert

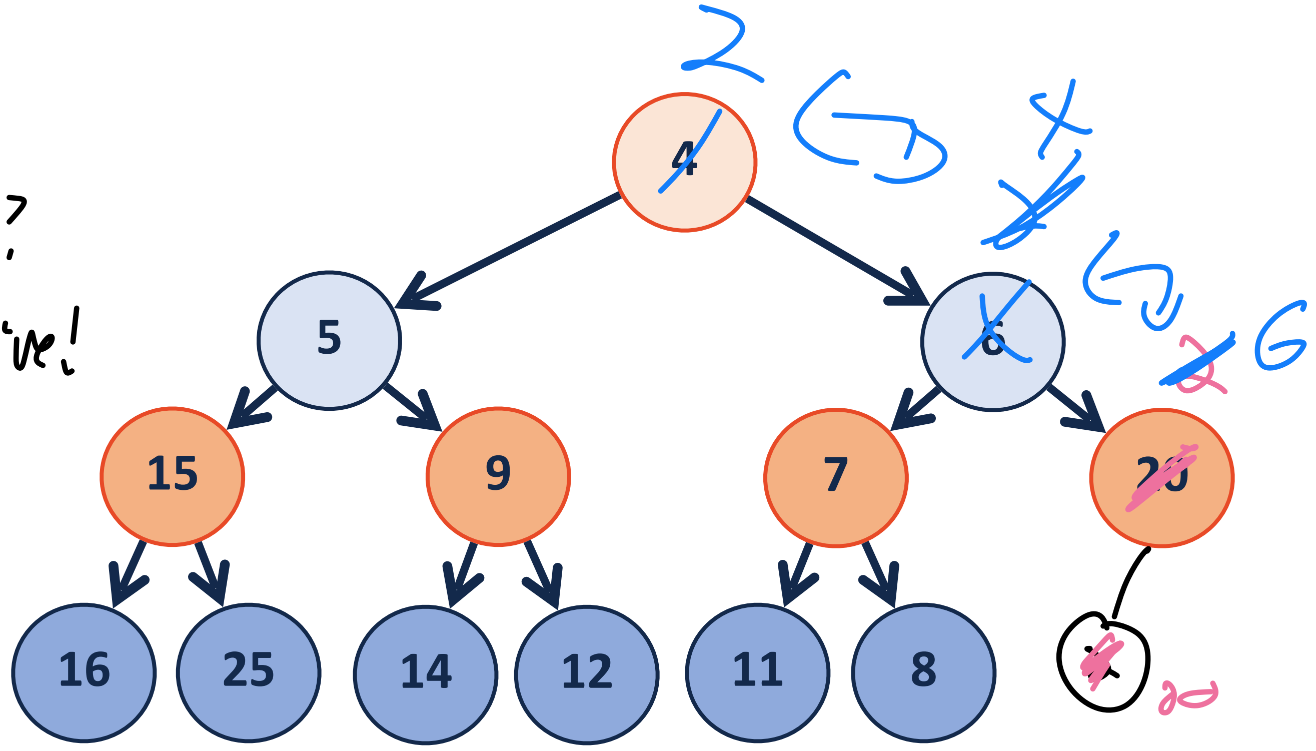
Insert(2)

1) Insert into array at end

2) Is my heap still a min heap?
↳ check parent(i) vs i
↳ if not, swap! } recursive!

recurse until parent(i).key < i.key

or i = 1 (root)



14/2 => 7

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

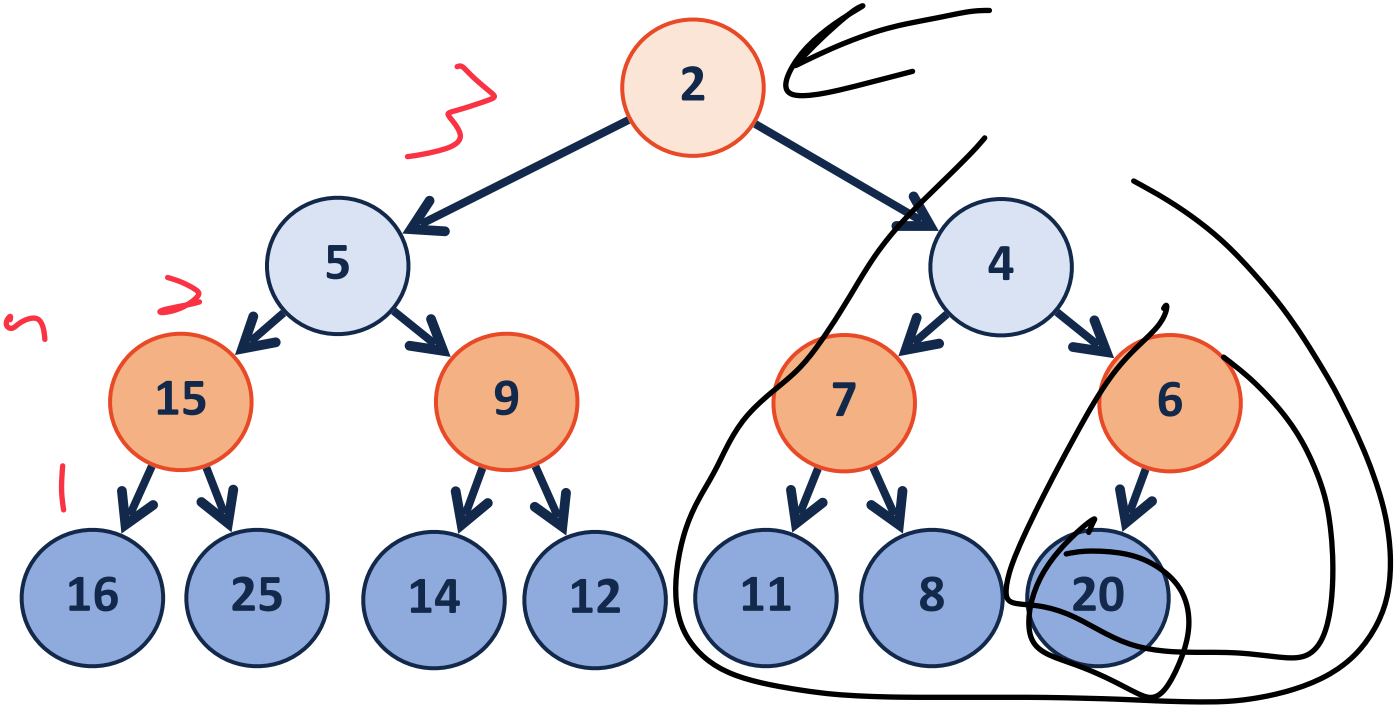
insert

[After] Insert(2)

Tree has height 3

Swaps: 3

Logically can't do more than height!



	2	5	4	15	9	7	6	16	25	14	12	11	8	20	

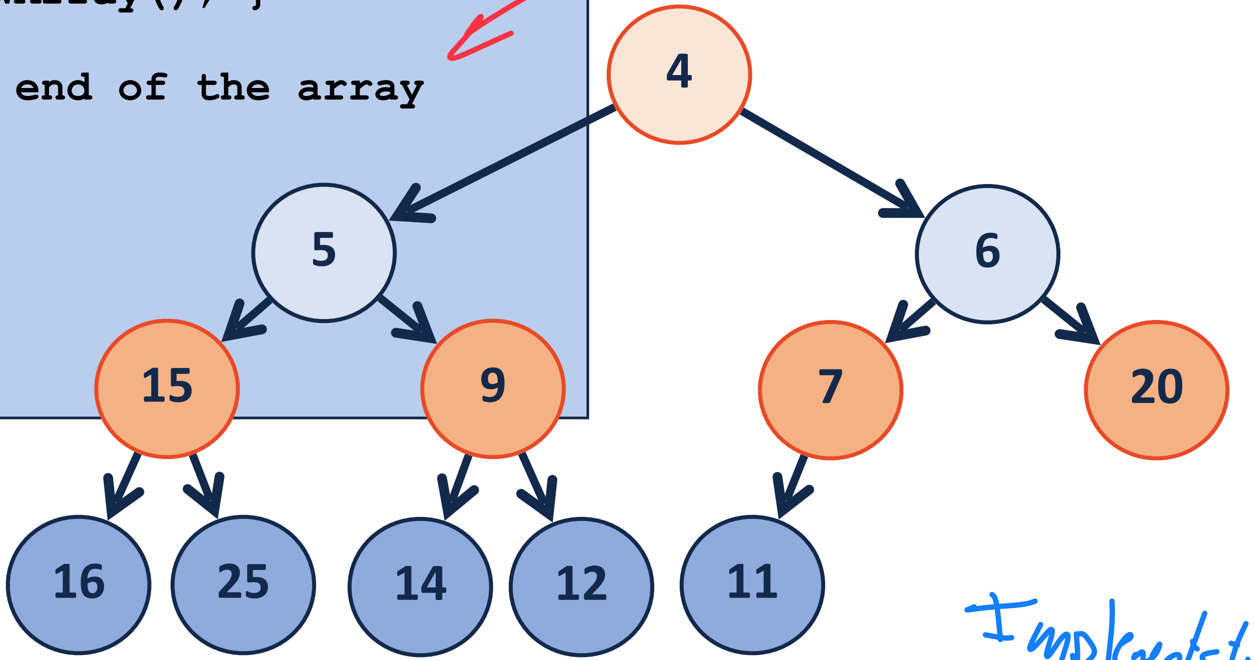
insert

```
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);
}
```

simple array resize

we have seen size before!

↳ Normally we have size = one after last



Implementation choice!

++size

2

size

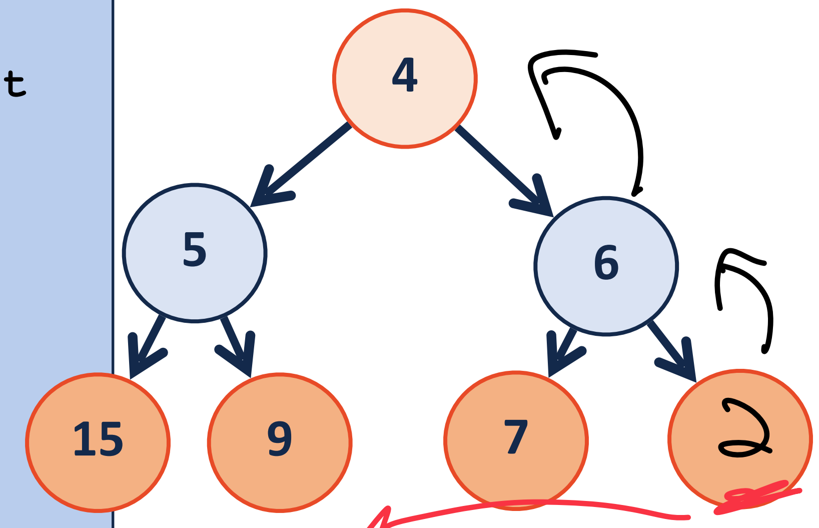
	4	5	6	15	9	7	20	16	25	14	12	11		

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( size_t index ) {

  if ( index > 1 ) {  ← base case of recursion
    if ( item_[index] < item_[ parent(index) ] ) {
      std::swap( item_[index], item_[ parent(index) ] );

      _heapifyUp( parent(i) );
    }
  }
}
```

Swap logic

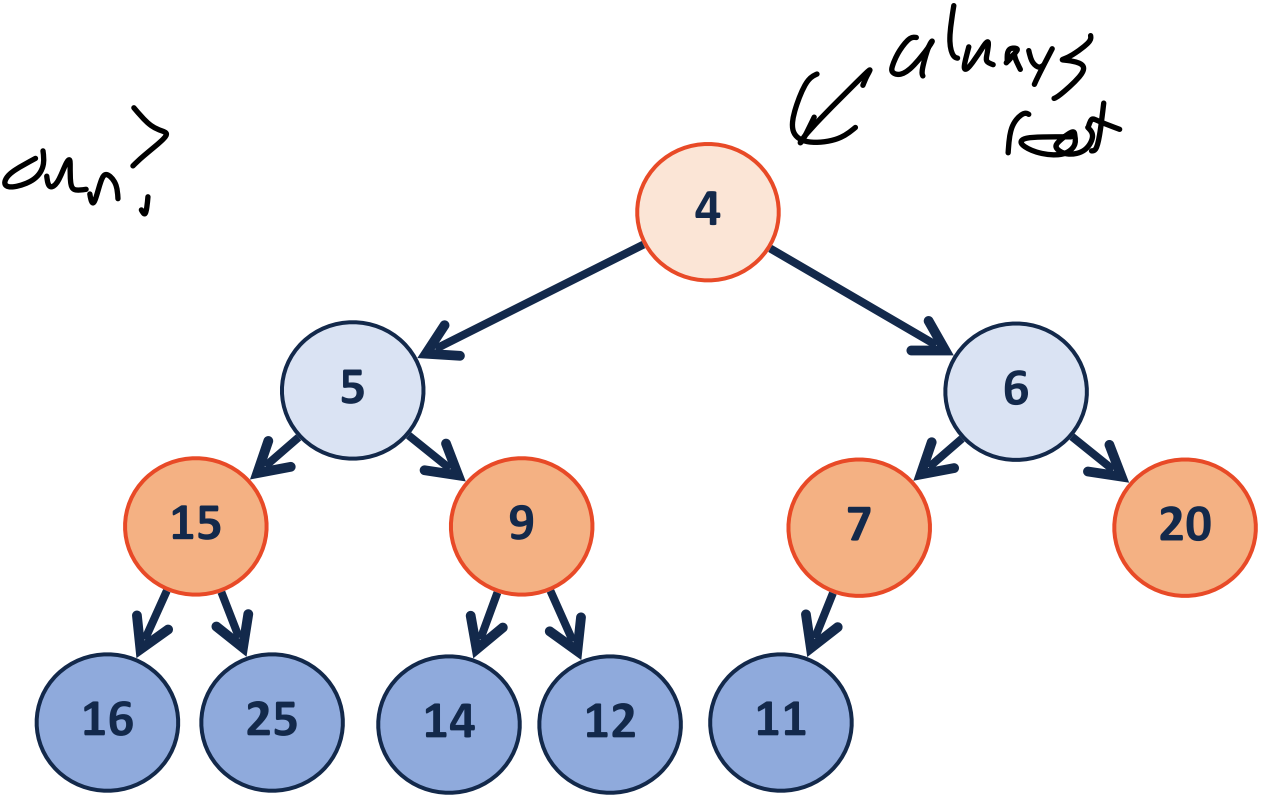


	4	5	6	15	9	7	2
0	1	2	3	4	5	6	7

removeMin

How to fix heap down?

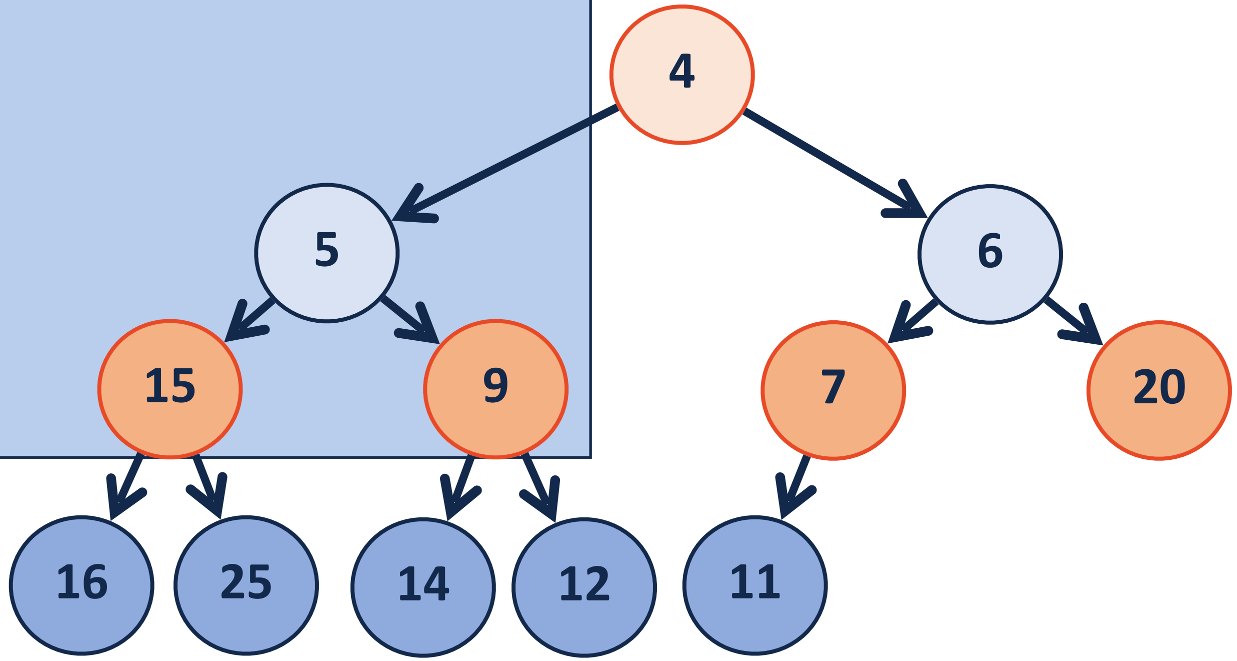
↳ heapify-down



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

removeMin

```
template <class T>
T 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;
}
```

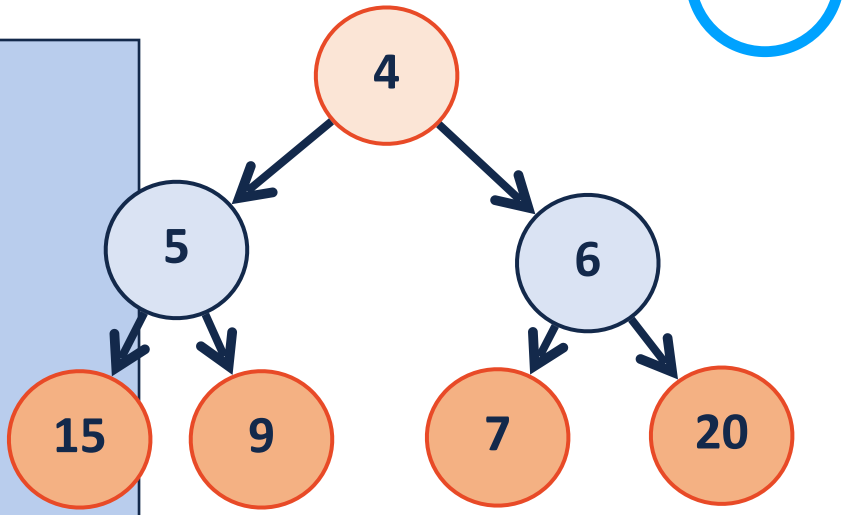


	4	5	6	15	9	7	20	16	25	14	12	11			

removeMin - heapifyDown

```
template <class T>
T 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;
}
```

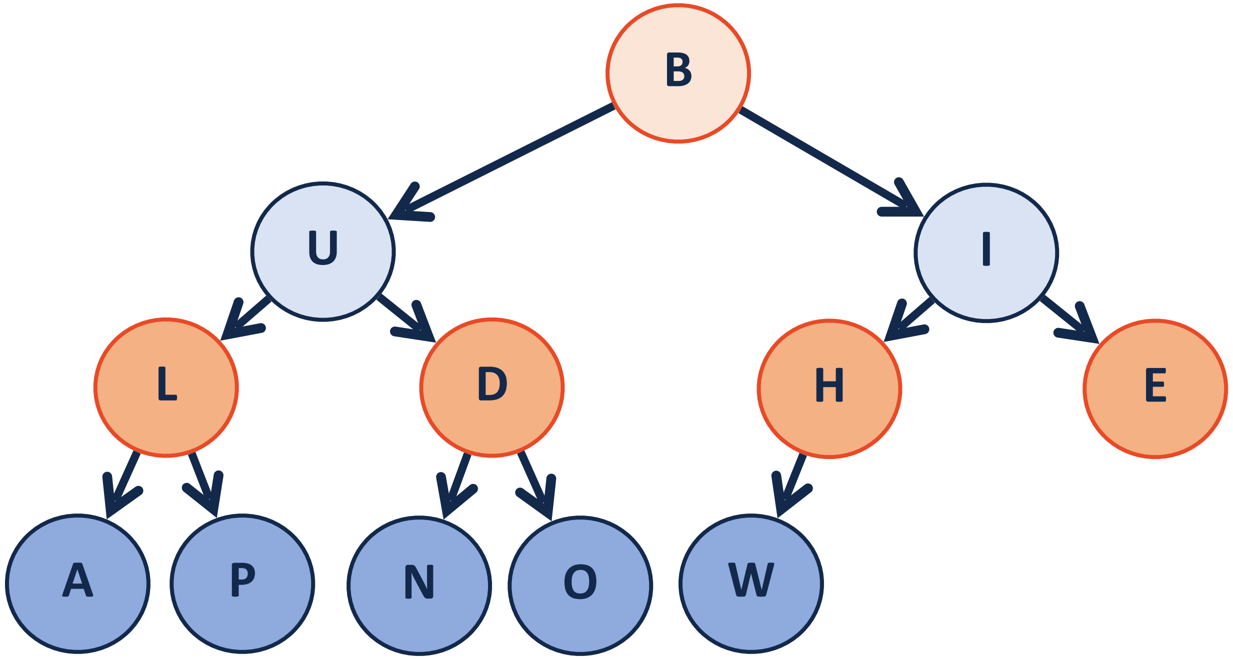
```
template <class T>
void Heap<T>::_heapifyDown(int index) {
  if ( !_isLeaf(index) ) {
    int minChildIndex = _minChild(index);

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

      _heapifyDown( ________________ );
    }
  }
}
```



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

buildHeap (minHeap Constructor)

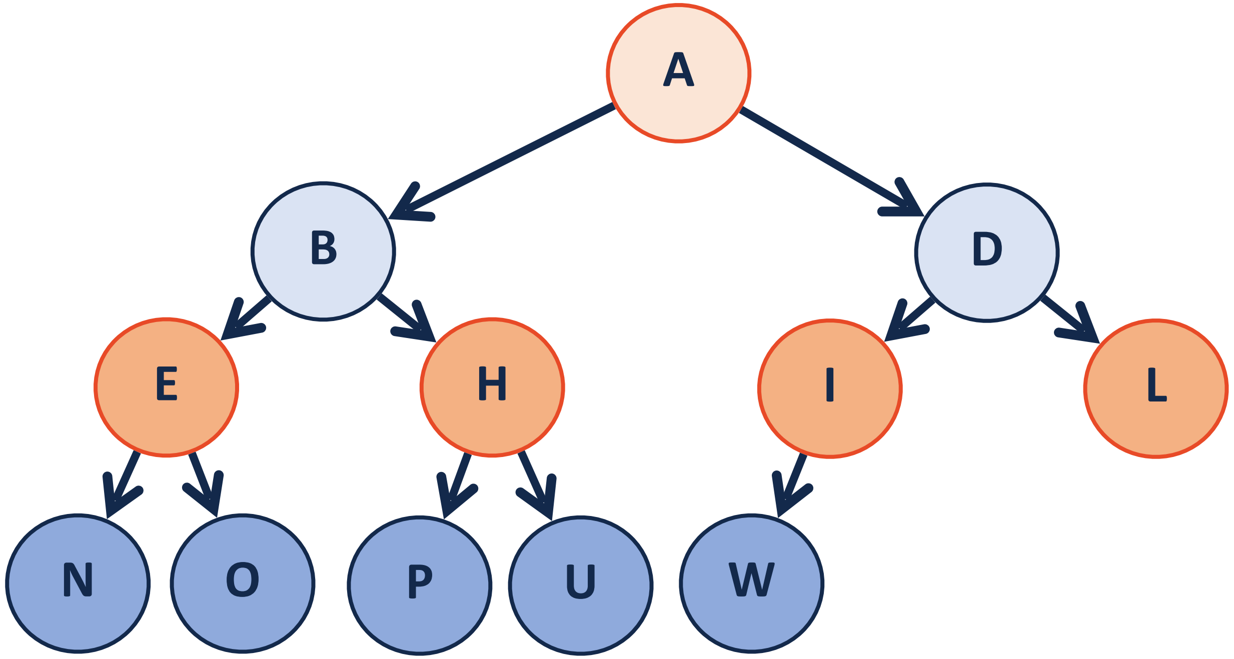
How can I build a minHeap?



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

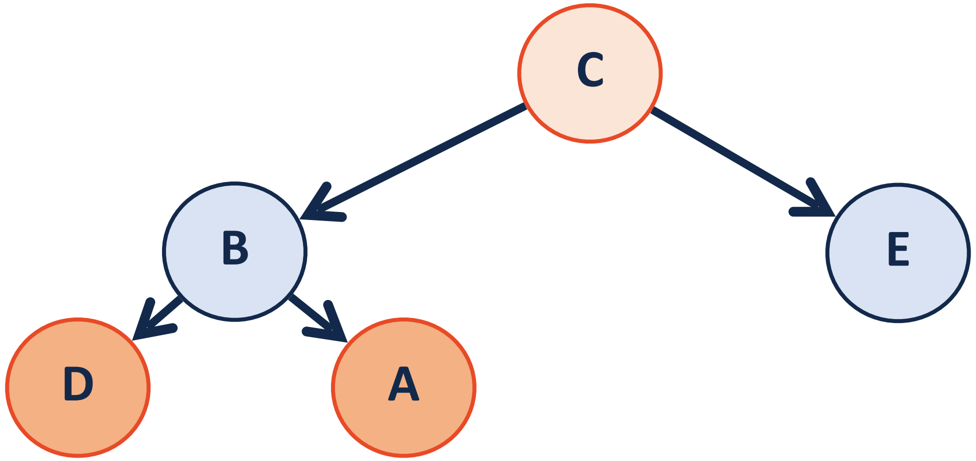
buildHeap – sorted array

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



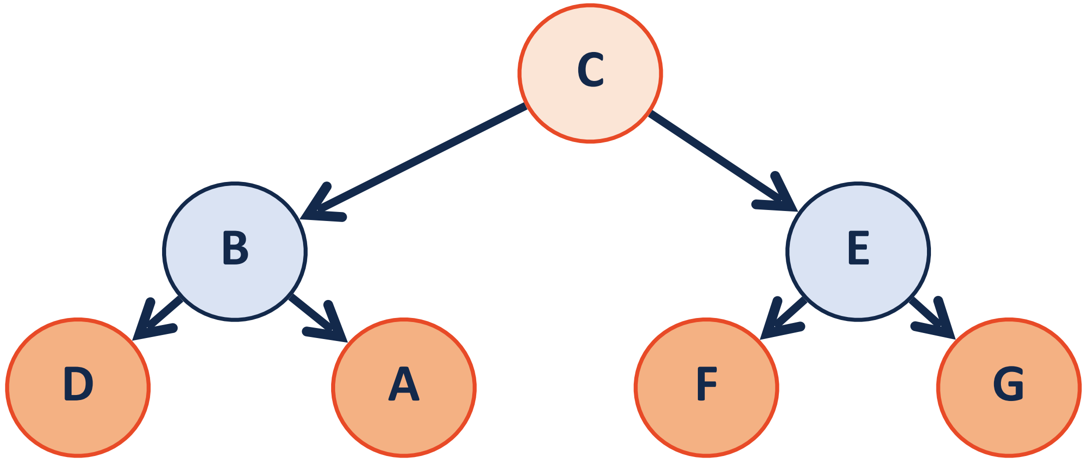
	A	B	D	E	H	I	L	N	O	P	U	W			

buildHeap - heapifyUp



	C	B	E	D	A		

buildHeap - heapifyDown



	C	B	E	D	A	F	G

buildHeap

1. Sort the array — its a heap!

2. heapifyUp()

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

3. heapifyDown()

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