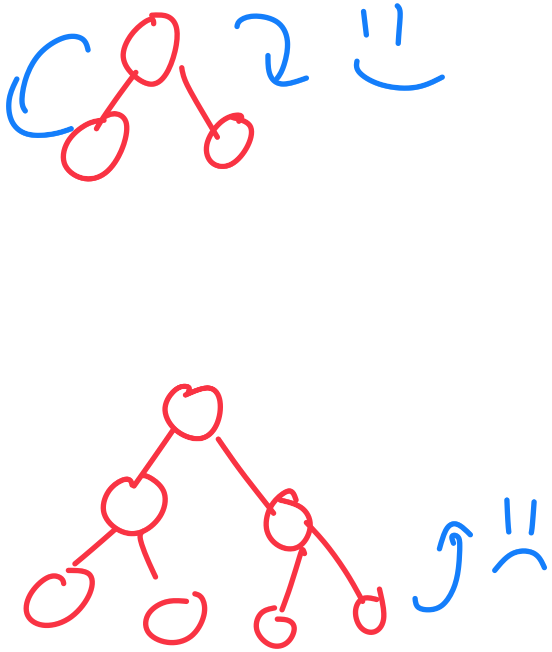
Data Structures
Heaps Analysis

CS 225
Brad Solomon

October 14, 2024



Department of Computer Science

Exam 3 (10/23 — 10/25)

Autograded MC and one coding question

Manually graded short answer prompt

Practice exam on PL

Topics covered can be found on website

Registration started October 10

https://courses.engr.illinois.edu/cs225/fa2024/exams/

Learning Objectives

Review the heap data structure

Discuss heap ADT implementations

Prove the runtime of the heap

(min)Heap (Priority Queue)

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

$\boxed{2^{h+1}} - 1$

~ n extra space

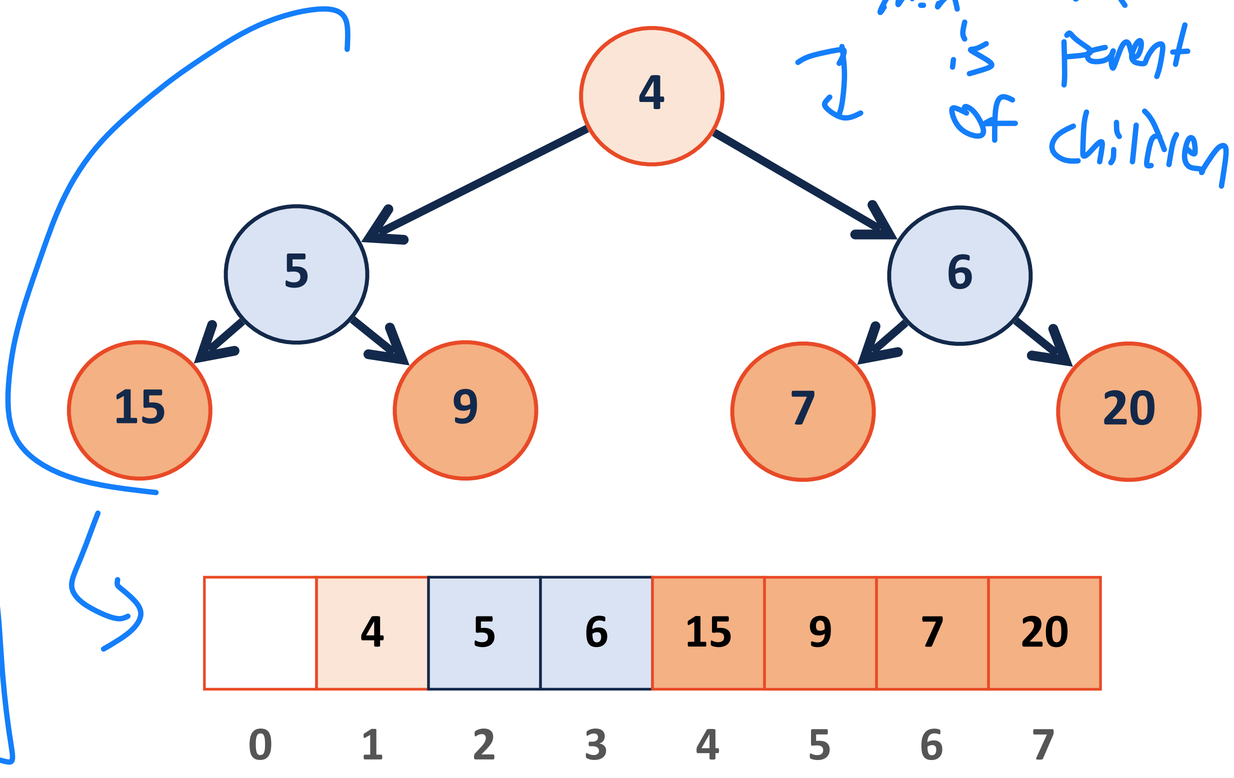
If index starts at 1:

`leftChild(i): 2i`

`rightChild(i): 2i+1`

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

min value ↓ is parent of children



(min)Heap

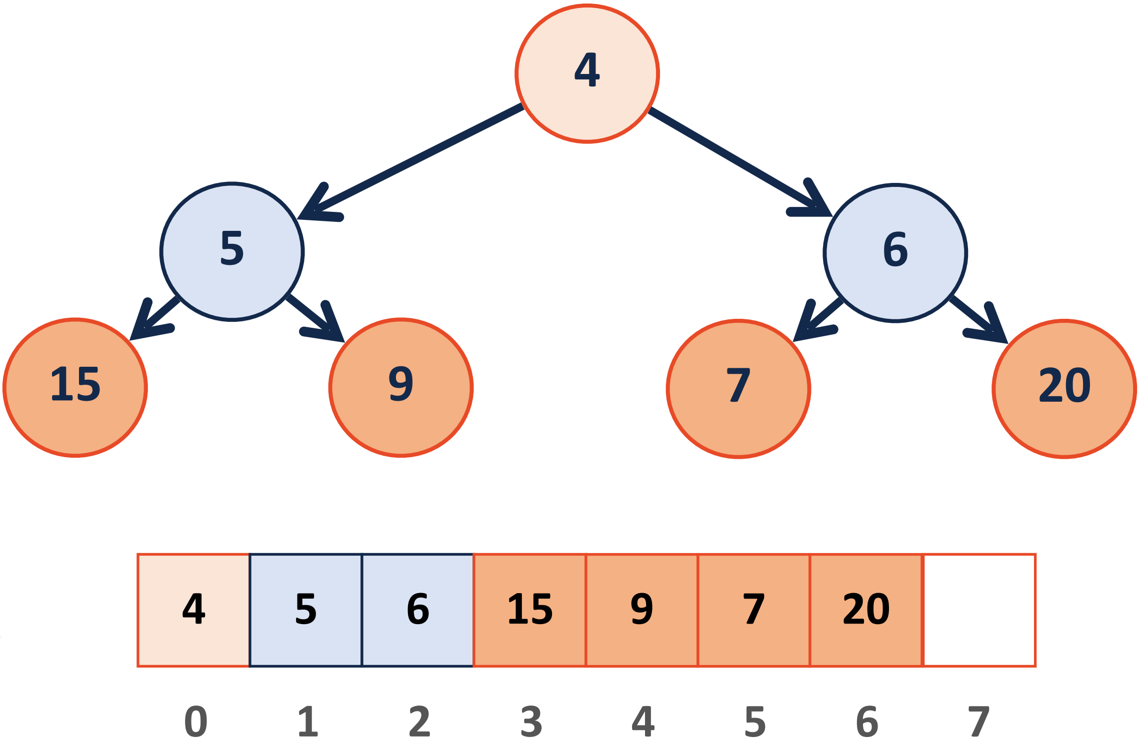
By storing as a complete tree, can avoid using pointers at all!

If Index starts at 0: Math was indeed nicer at i=1

`leftChild(i): 2i+1`

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

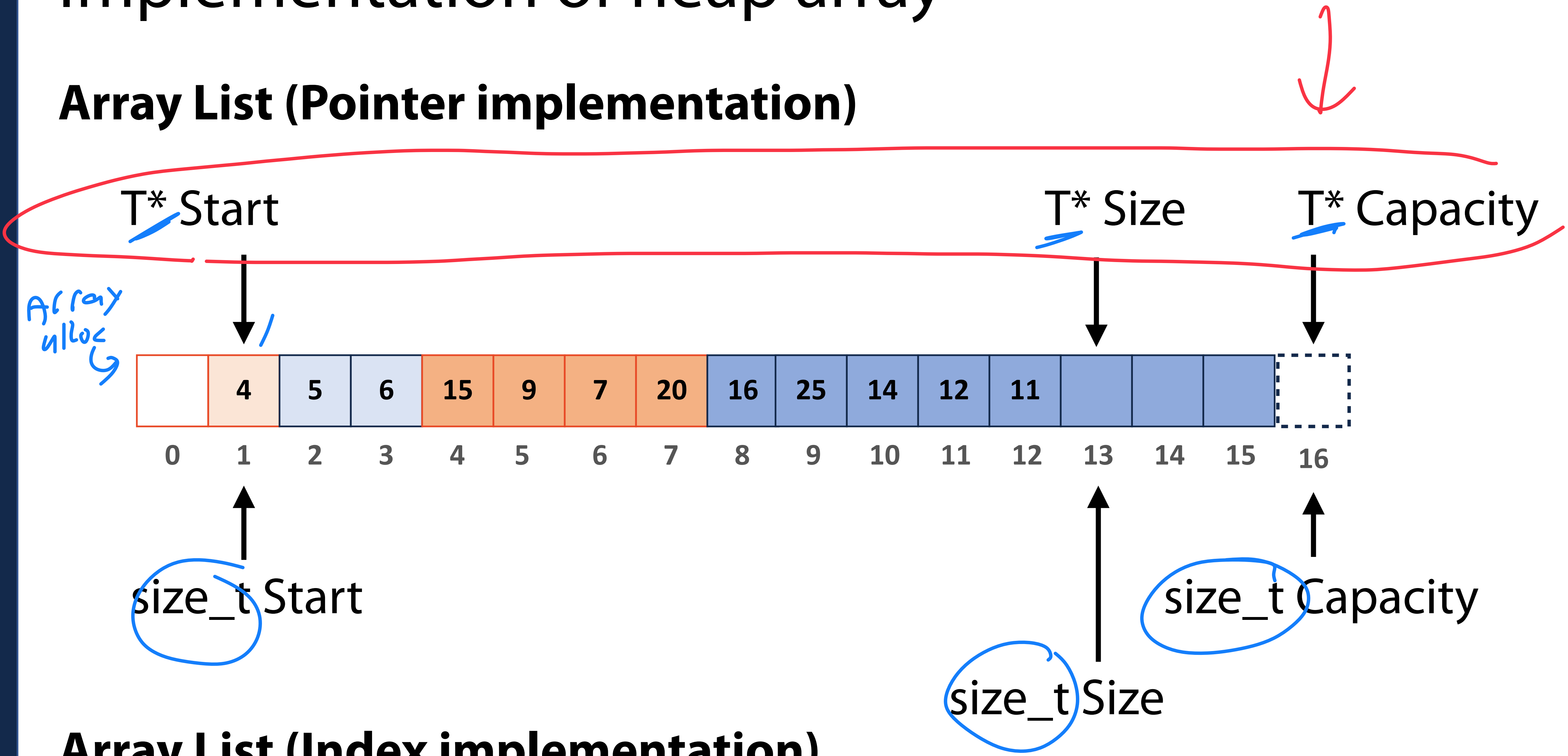
`parent(i): floor((i-1)/2)`



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

Implementation of heap array

Array List (Pointer implementation)



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

Array List (Index implementation)

insert - heapifyUp

$h = O(\log n)$

```
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);
}
```

$O(1)^*$

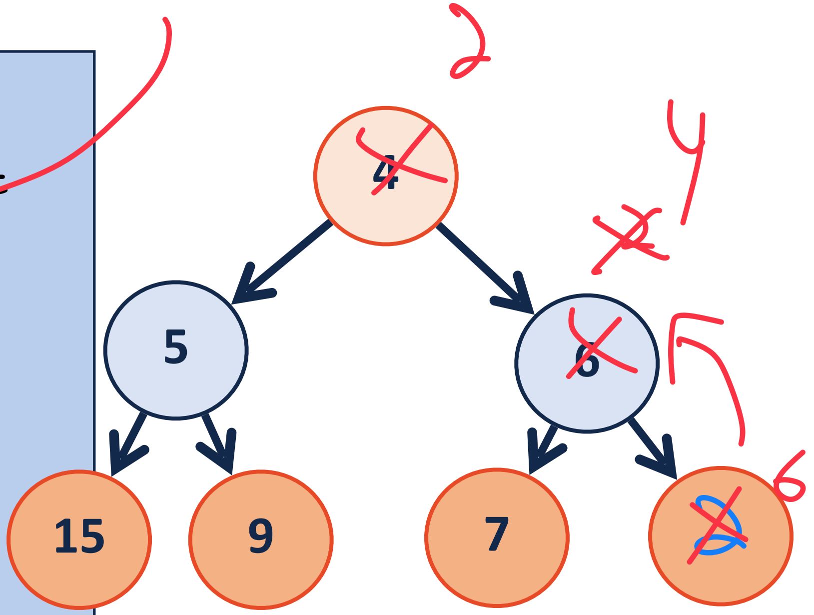
$O(1)$

$O(\log n)$

```
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;
    }
  }
}
```



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

$\boxed{O(\log n)}$

size = 7

size = 8

removeMin

Is there a "good" case for array remove?

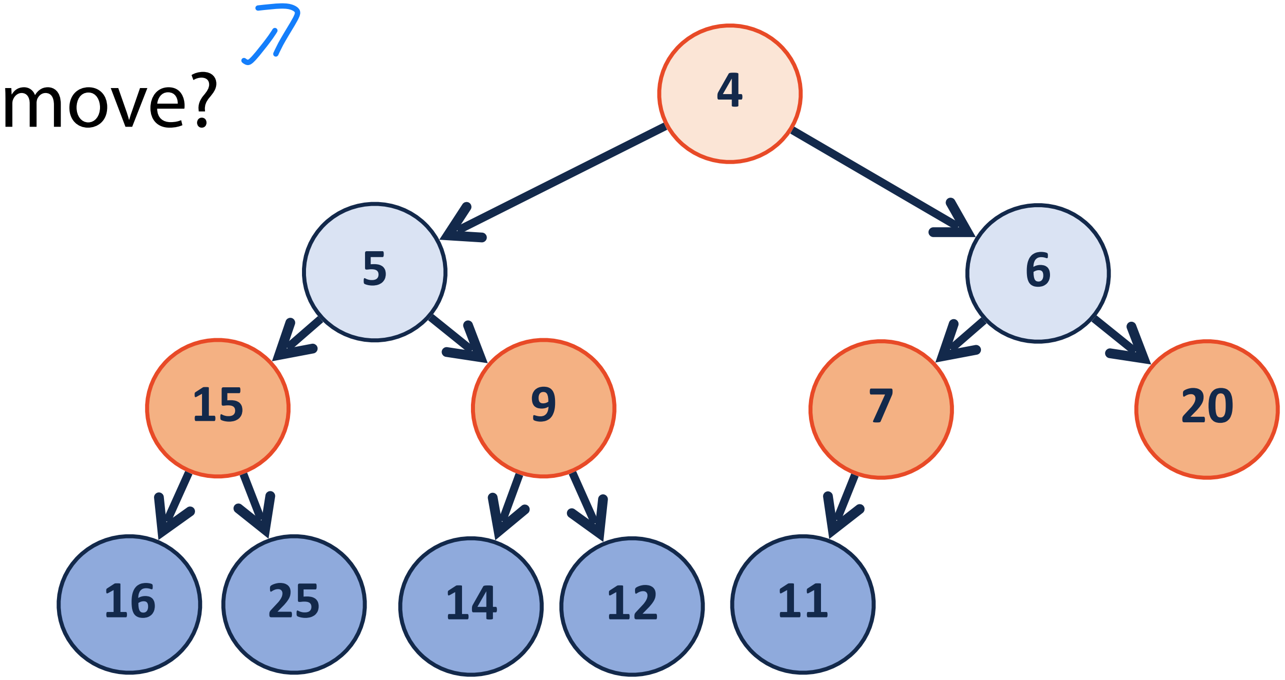
What is the Big O of array remove?

↳ O(n)

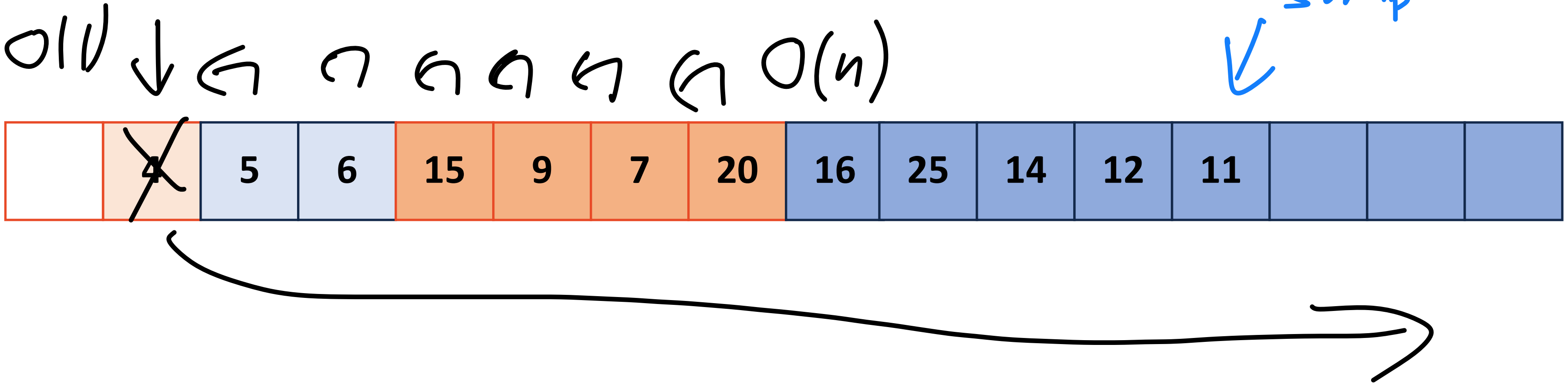
What else can we do?

↳ Chain swaps!

↳ Swap



swap here?



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

removeMin

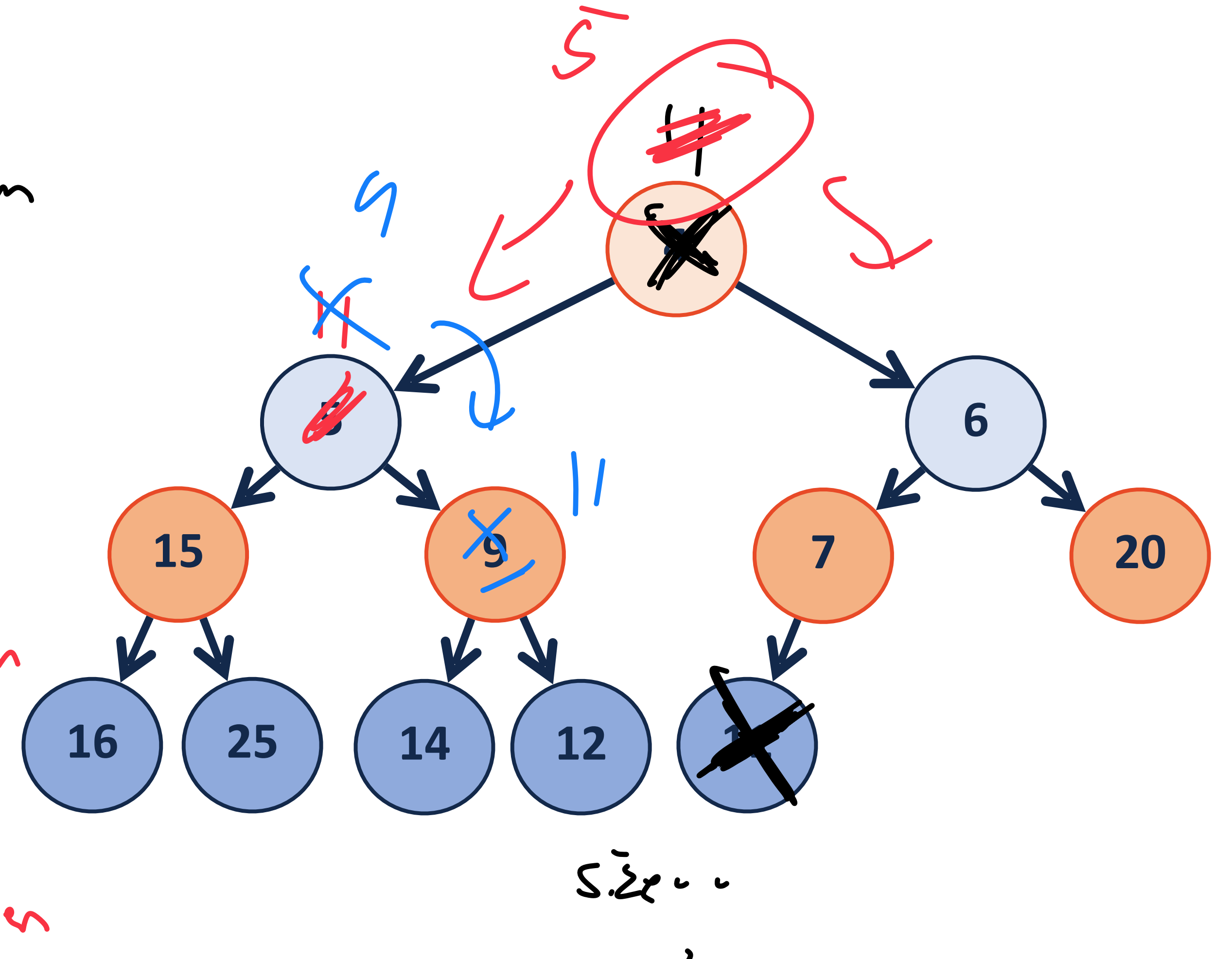
1) Swap root w/ last item

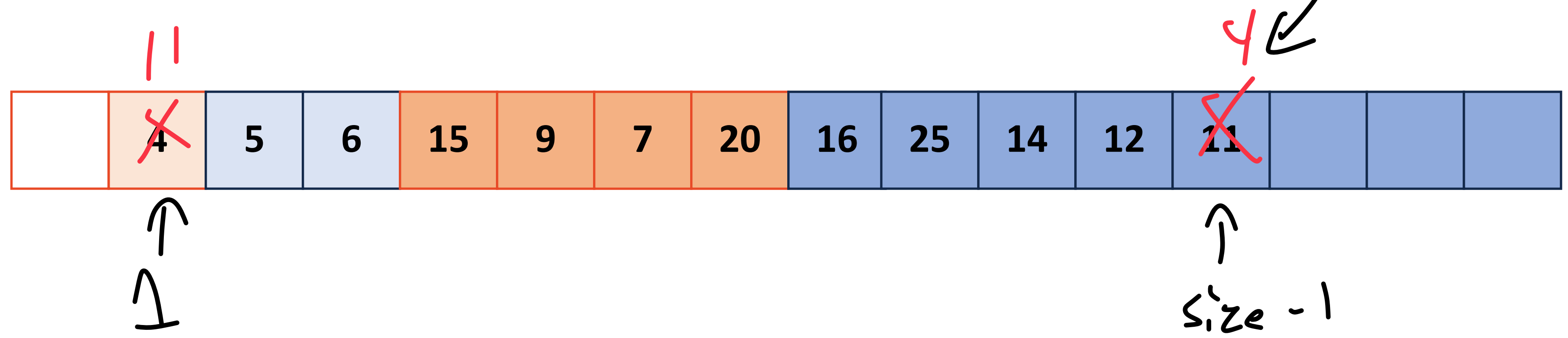
↳ Delete last item

↳ size--;

2) heapifyDown()

↳ Repeated swaps w/ min child until leaf or smaller than both children





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

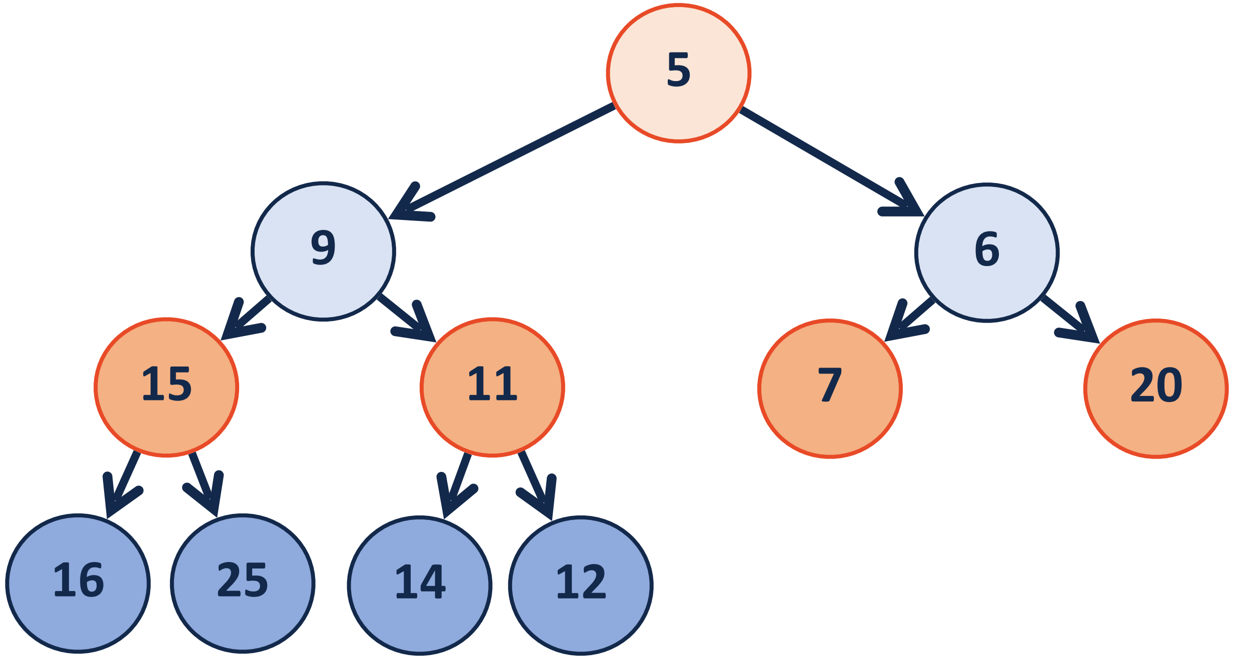
removeMin

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				

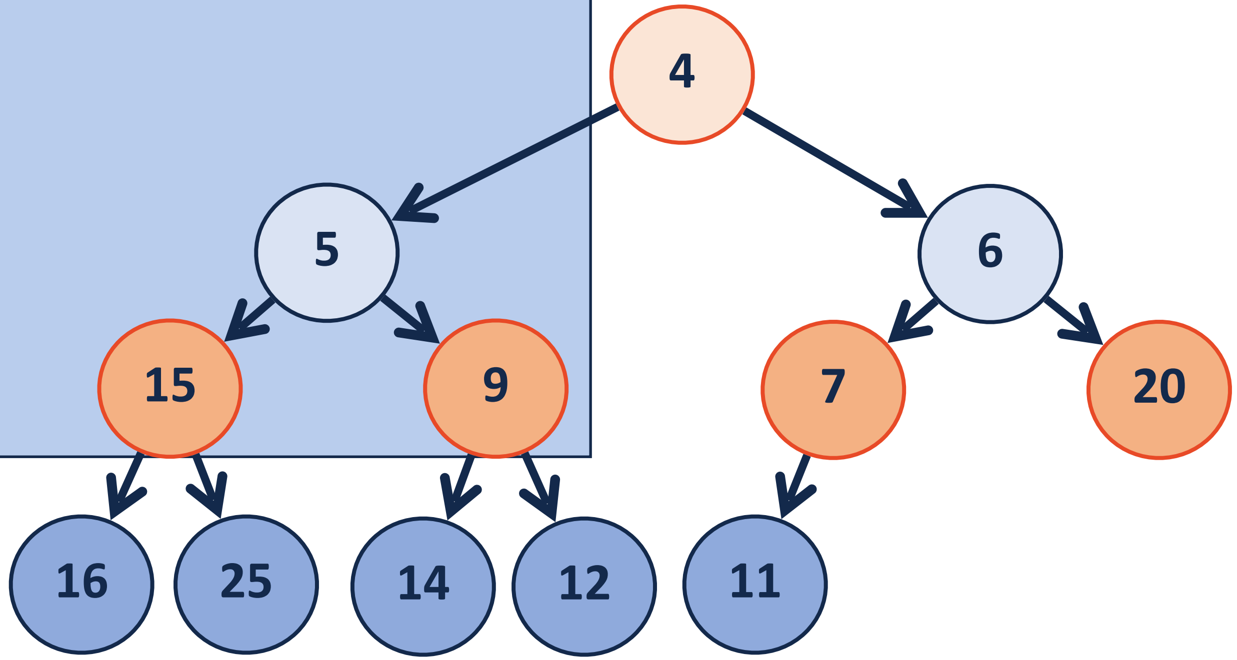
removeMin

```
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;
}
```

4 remove



11

← size

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

4

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;
}
```

O(1)

O(log n)

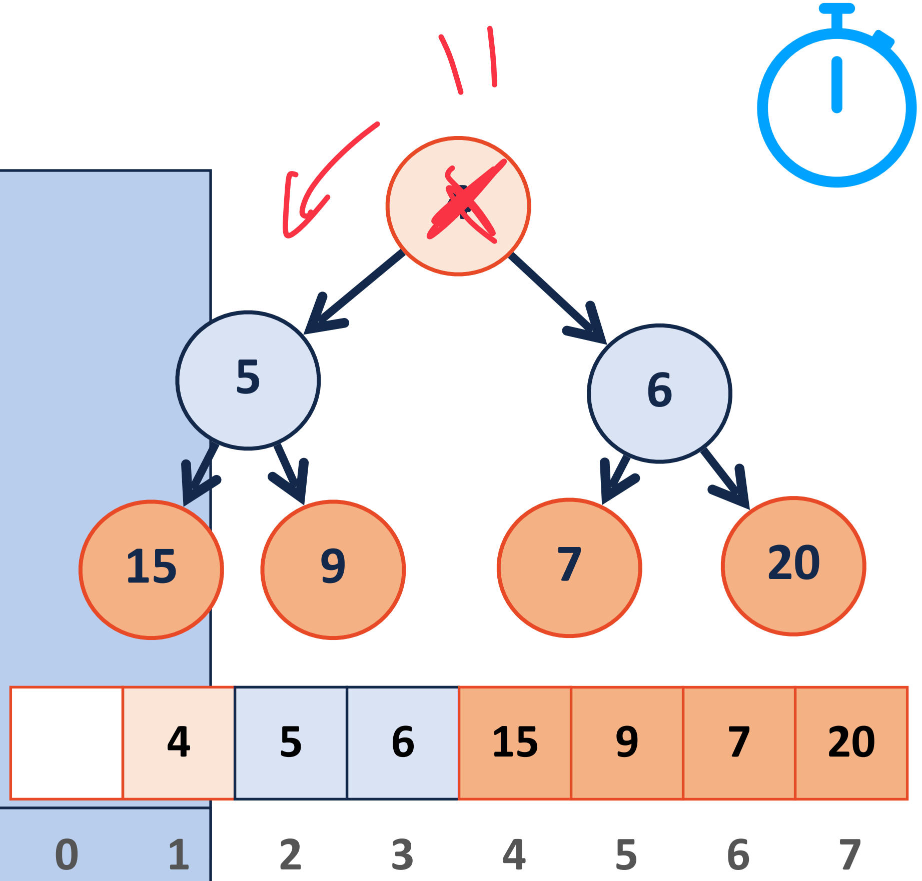
```
template <class T>
void Heap<T>::_heapifyDown(int index) {
  if ( !_isLeaf(index) ) {
    int minChildIndex = _minChild(index);

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

      _heapifyDown( minChildIndex );
    }
  }
}
```

← Base case

← get min child



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

O(log n)

buildHeap (minHeap Constructor)

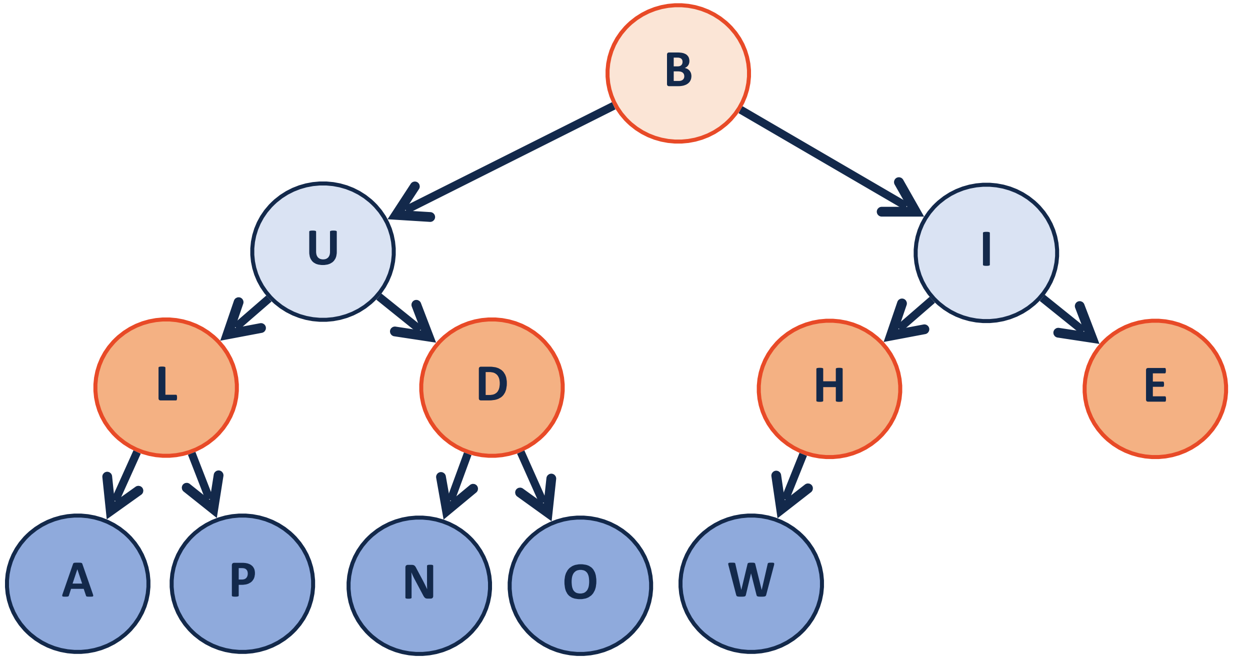
How can I build a minHeap?

1) Sort an array

2) Chain inserts

↳ heapifyUp()

3) Chain heapifyDown()



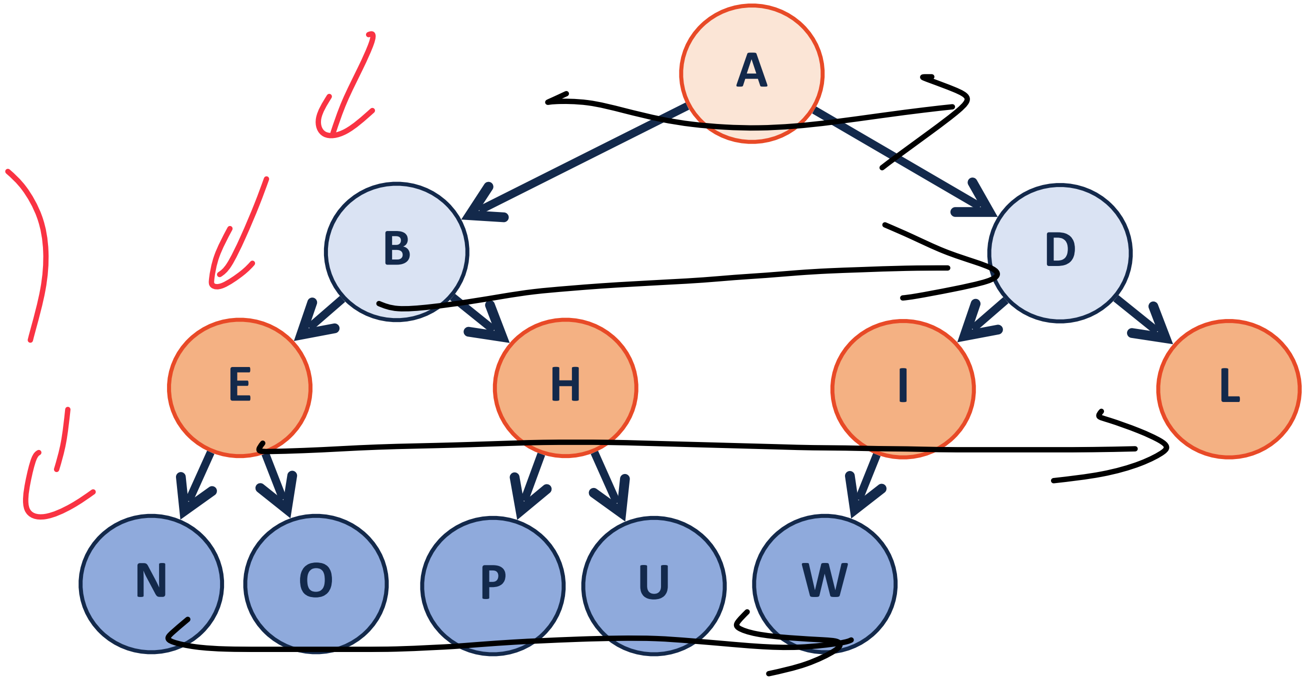
	B	U	I	L	D	H	E	A	P	N	O	W			

buildHeap – sorted array

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

Black Box Sort

O(n log n)

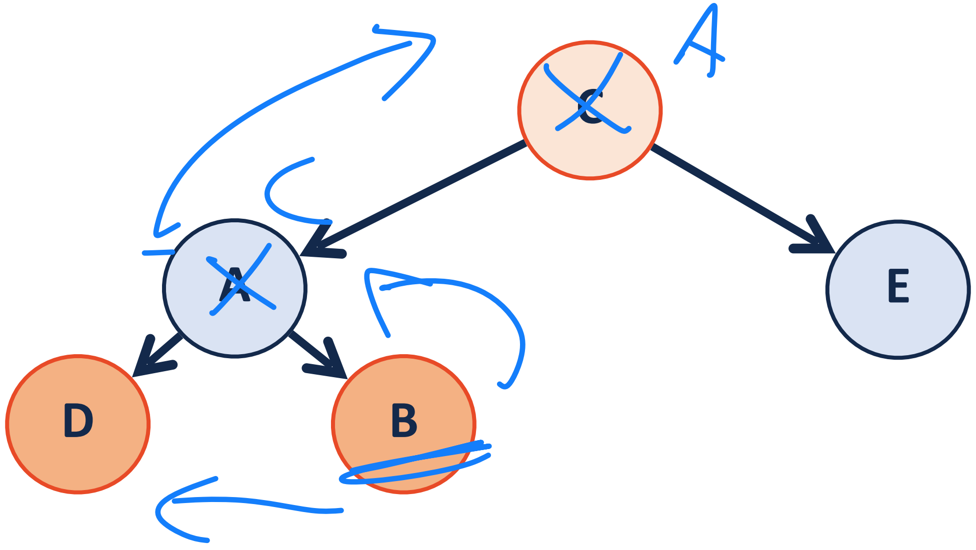


	A	B	D	E	H	I	L	N	O	P	U	W			

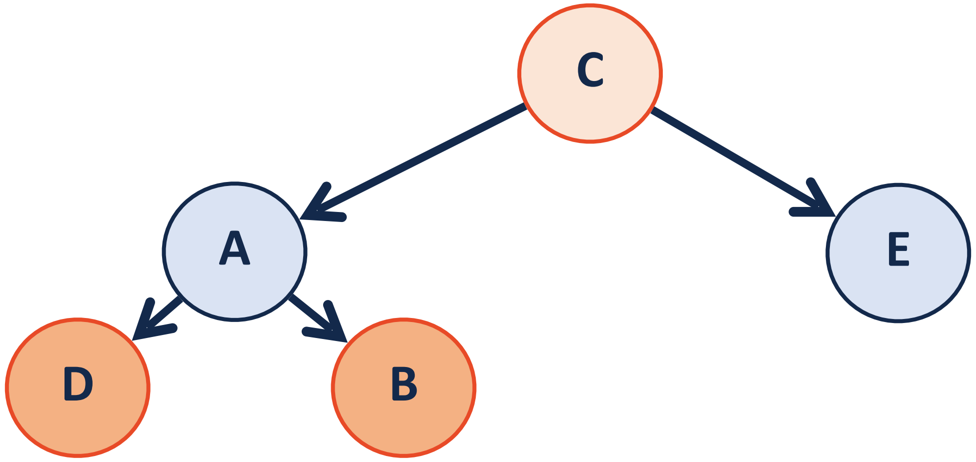
buildHeap - heapifyUp

Do we heapifyUp from top or bottom?

↳ B says no swap!



	C	A	E	D	B		



	C	A	E	D	B		

buildHeap - heapifyUp

Repeatedly `heapifyUp(i)`:

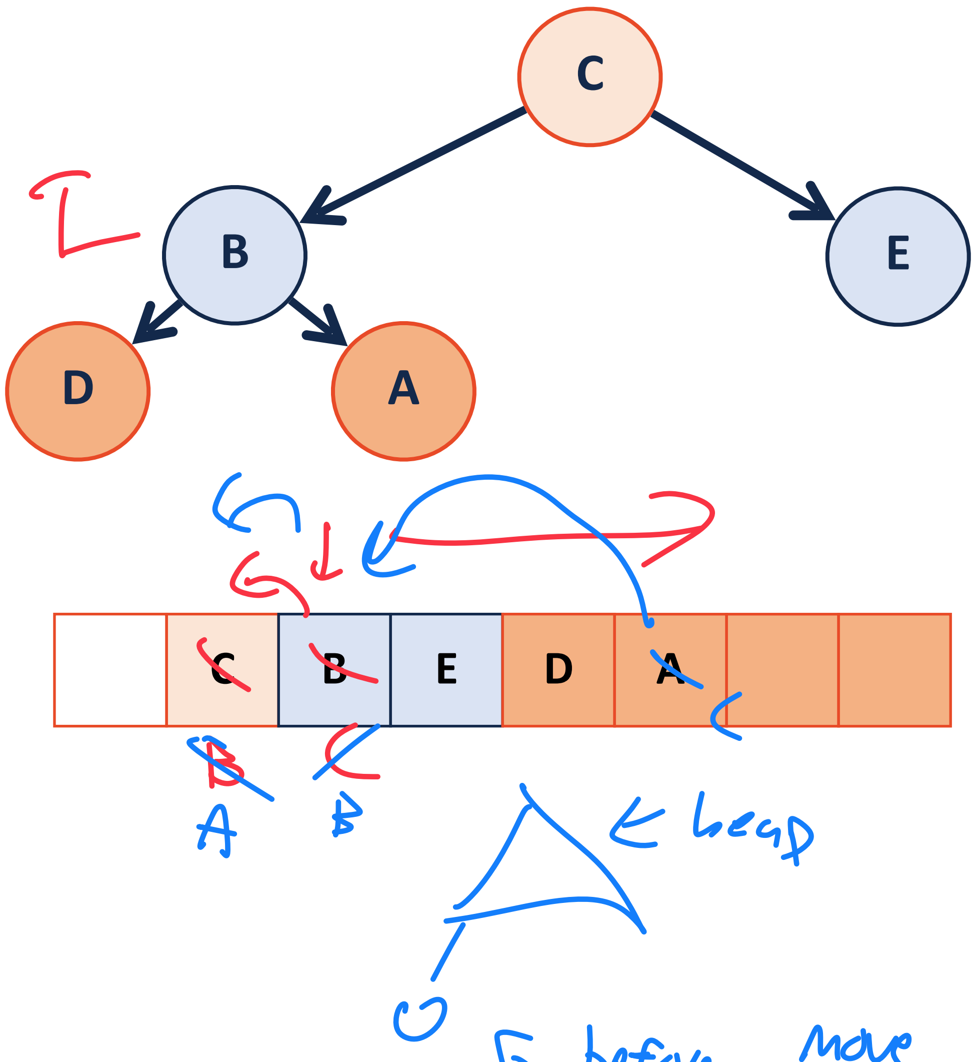
Starting at index 2

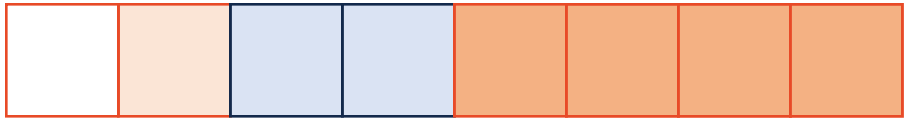
Ending at index size -1

Starting from top to bottom

why top to bottom?

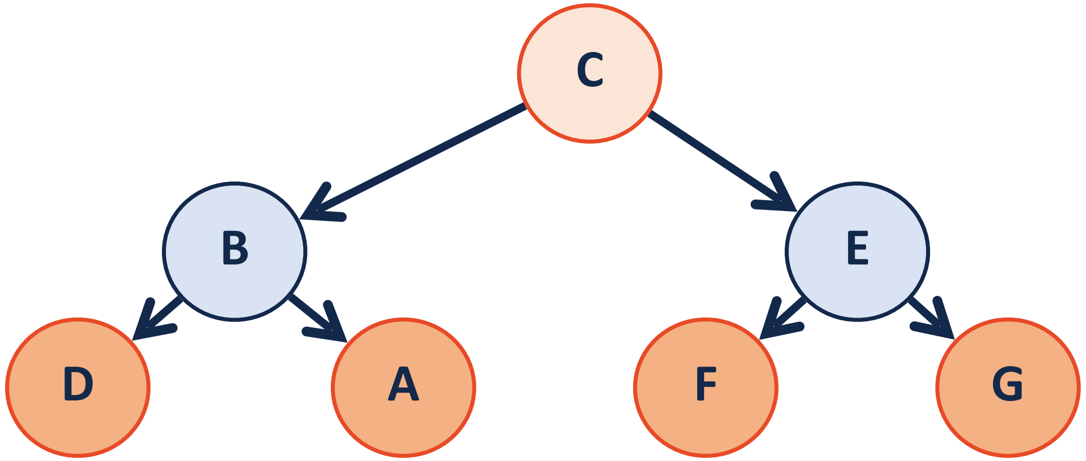
↳ heapify up & Down assume heap property

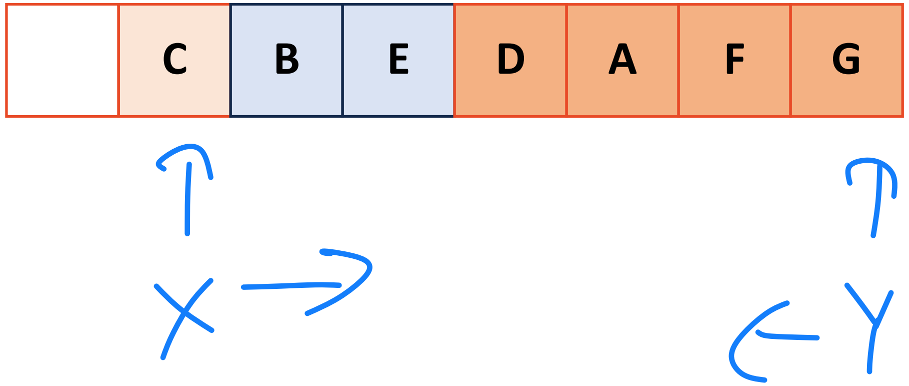




buildHeap - heapifyDown

Do we hDown from top or bottom?



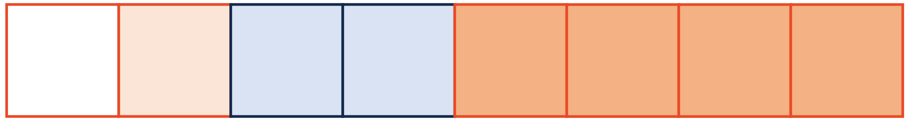


	C	B	E	D	A	F	G

before I can place this

↓ ↓ ↓ ↓ heapDown()

Must be heap



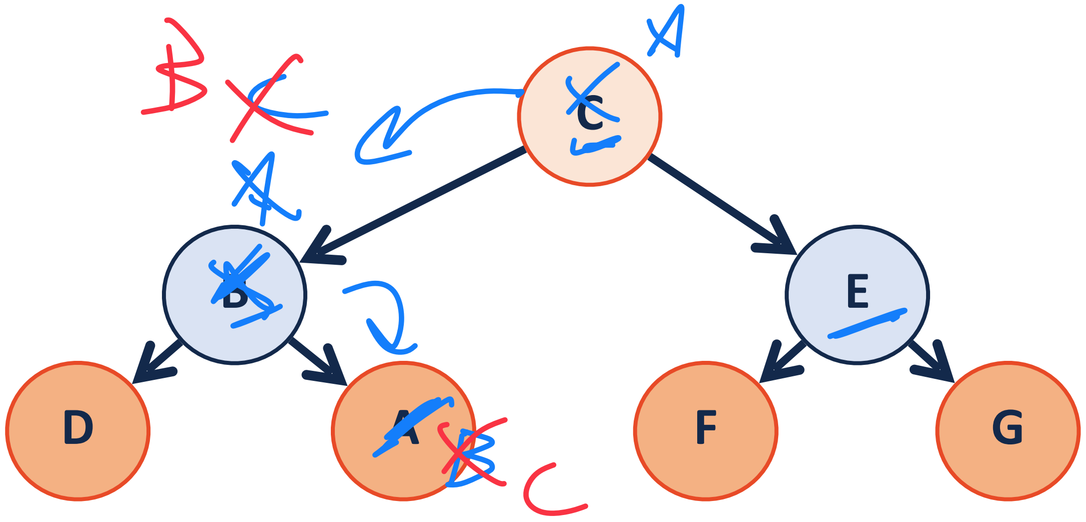
buildHeap - heapifyDown

Repeatedly `heapifyDown(i)`:

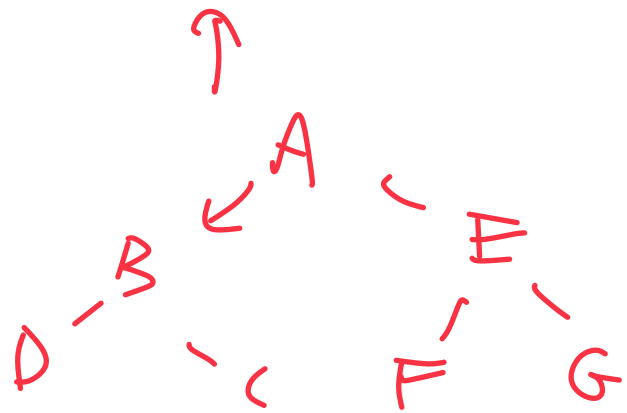
Starting at index capacity/2

Ending at index 1

No leaves



	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

$O(\log n)$

$O(n \log n)$

3. heapifyDown()

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

$n/2$

$O(\overset{h}{\cancel{\log n}})$

$O(\frac{n}{2} \log n) \rightarrow O(n)$

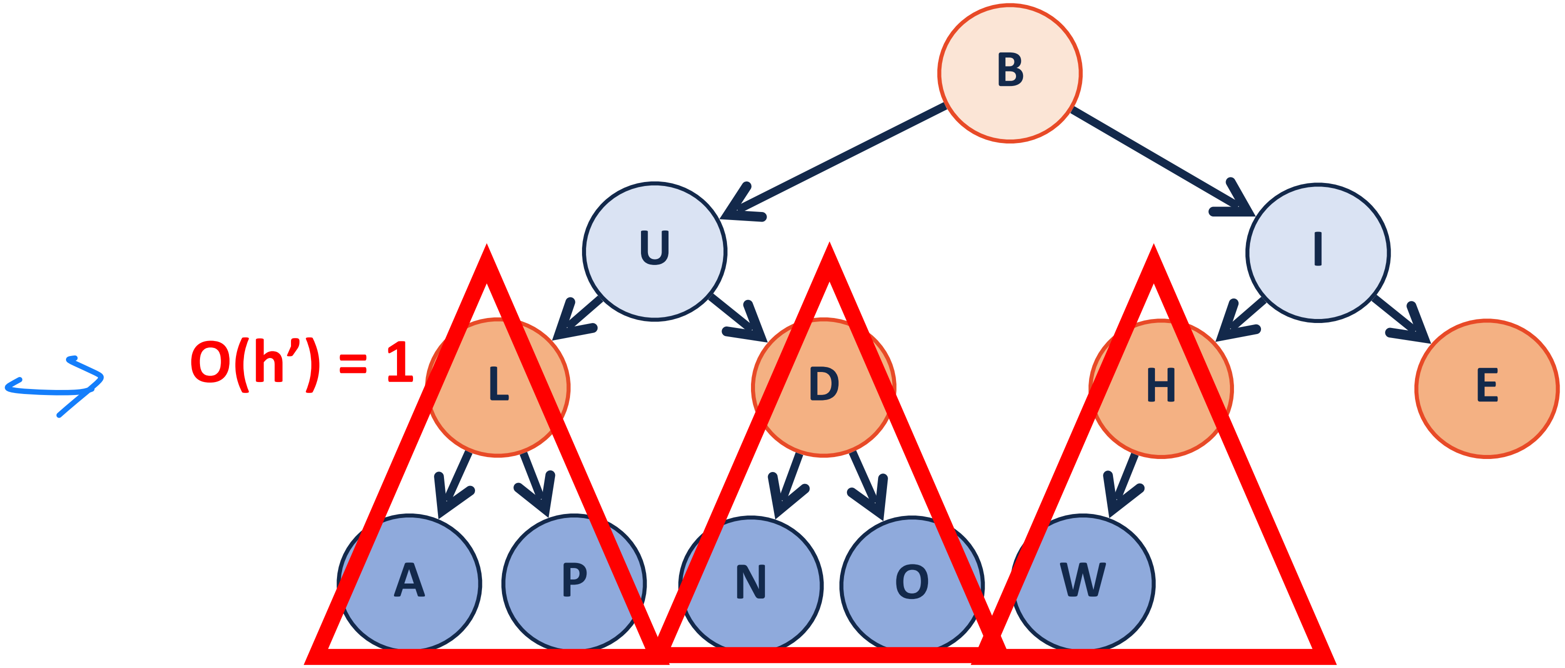
Wrong!

Not right! $O(h)$

buildHeap - heapifyDown

Lets break down the total ‘amount’ of work:

2^{h-1} each can swap at most 1 time

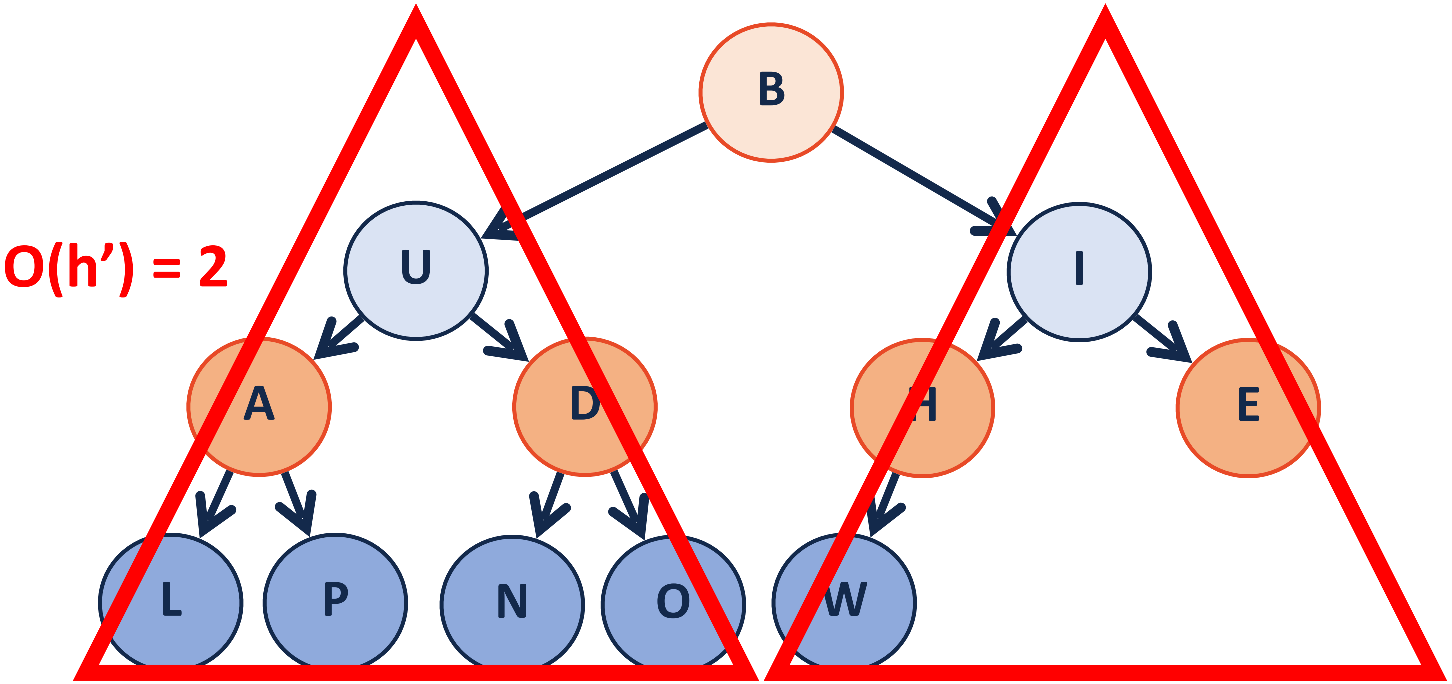


	B	U	I	L	D	H	E	A	P	N	O	W			

buildHeap - heapifyDown

Lets break down the total ‘amount’ of work:

2^{h-2} can do at most 2 swaps



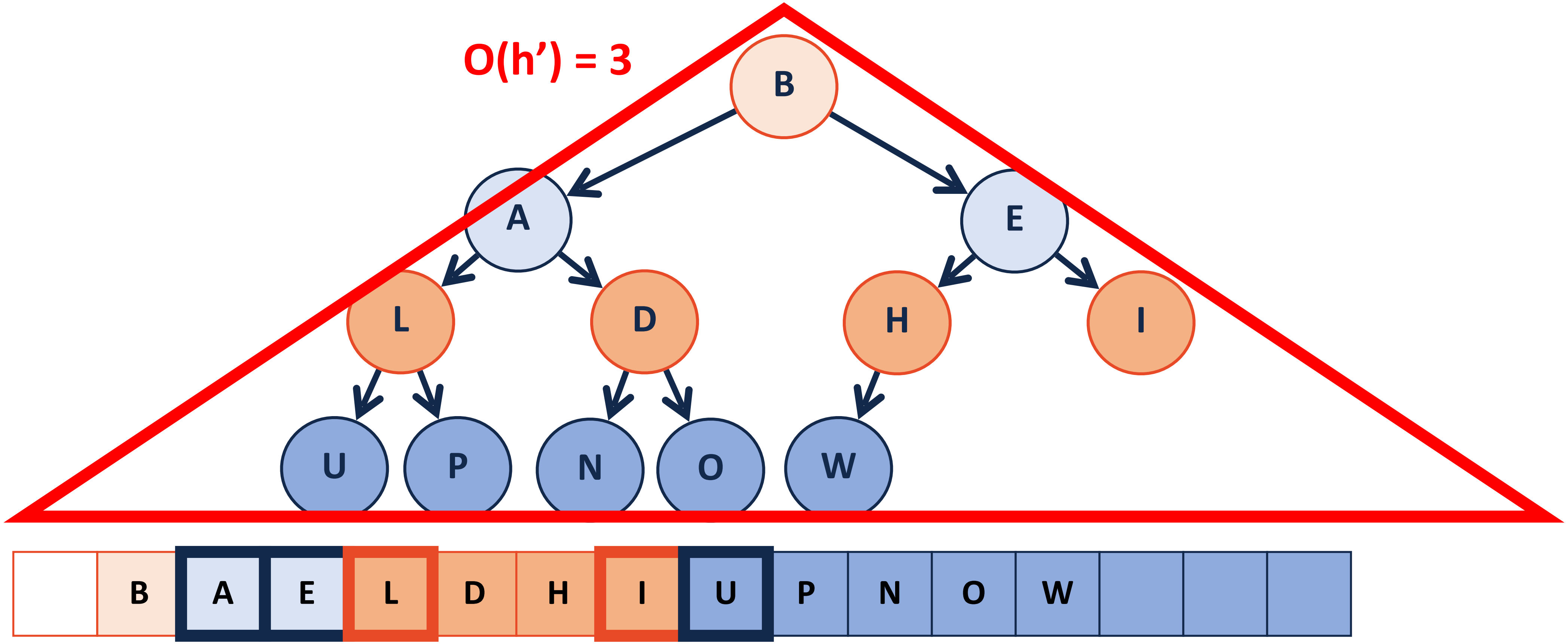
O(h’) = 2

	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} at most 3 swaps



Proving buildHeap Running Time

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

Strategy:

Proving buildHeap Running Time

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

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

$||$

adding up height of every node

Proving buildHeap Running Time

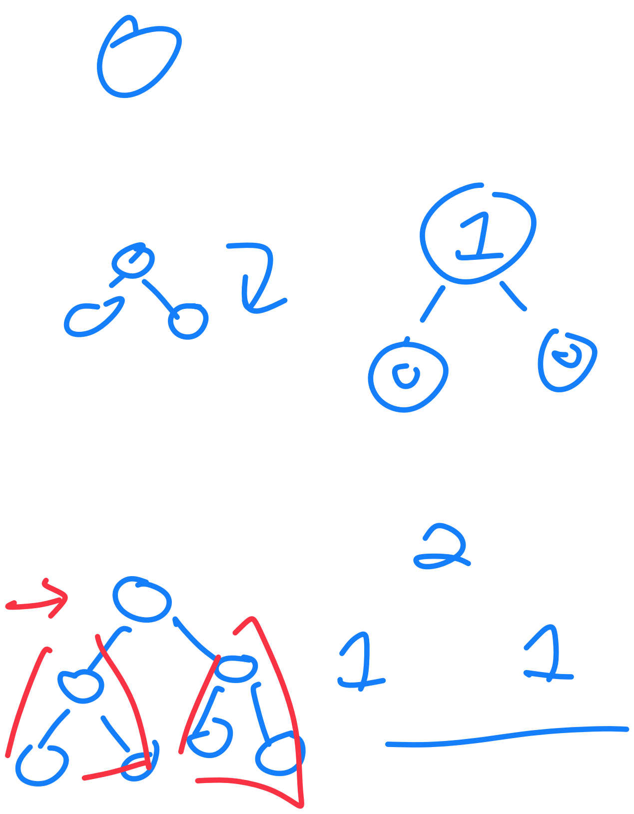
S(h): Sum of the heights of all nodes in a **perfect** tree of height **h**.

S(0) = 0 swaps

S(1) = 1

S(2) = 4

S(h) = $h + s(h-1) + s(h-1)$



Proving buildHeap Running Time

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

Base Case:

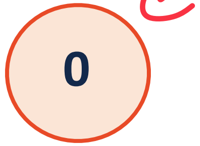
Proving buildHeap Running Time

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

Base Case:

0 swaps total

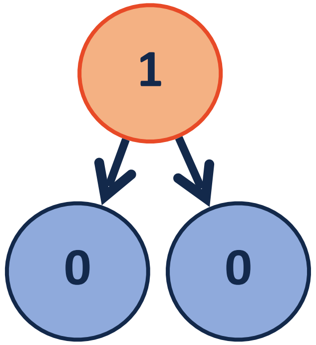
h = 0



$$\underline{2^{0+1} - 2 - 0 = 0}$$ ✓

vs

h = 1



$$2^{1+1} - 2 - 1 = 1$$ ✓

2+0+0

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)

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

$2 \cdot 2^h$

Proving buildHeap Running Time

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

$$S(h) = 2^{h+1} - 2 - h$$ ↙ proven!

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

How can we estimate running time?

Proving buildHeap Running Time

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

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

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

How can we estimate running time?

$$2^{\underline{\log n}+1} - 2 - \underline{\log n}$$

(Plug in)

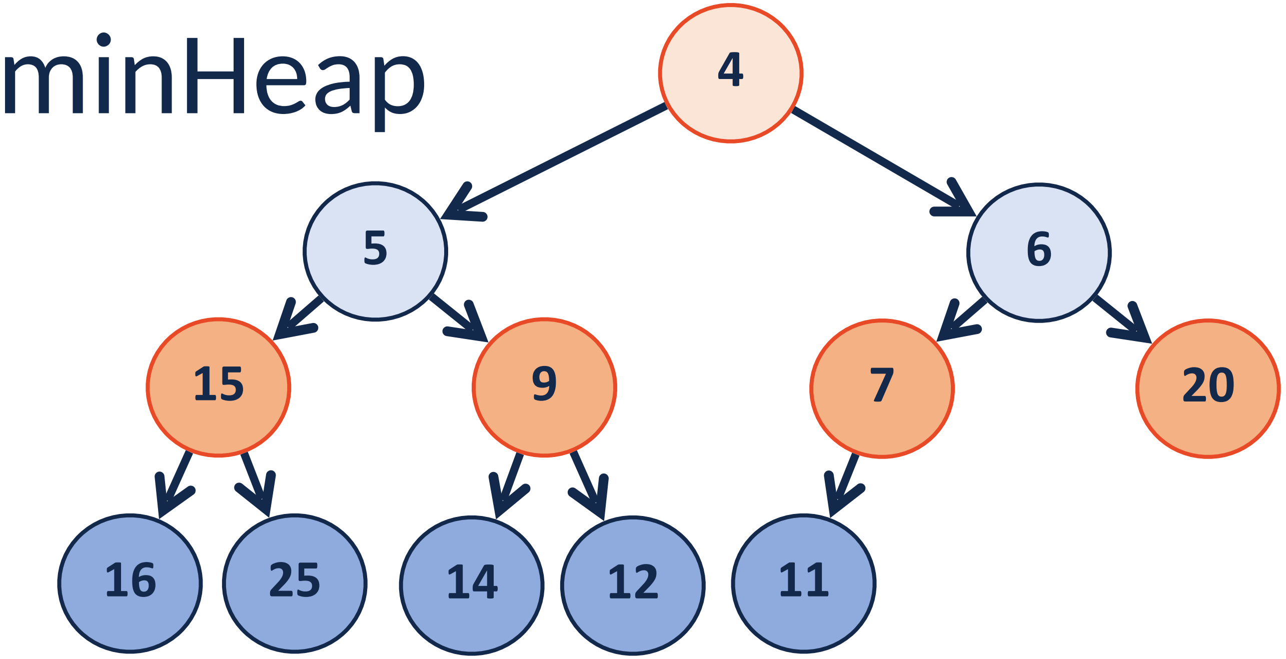
$$2 * 2^{\log_2 n} - 2 - \log n$$

(Simplify)

$$\boxed{2\, n} - \log n - 2 \approx O(n)$$

(Rearrange)

minHeap



1. **Construction**
   ↳ $O(n)$
2. **Insert** → $O(\log n)$
3. **RemoveMin** → $O(\log n)$

On an array!

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

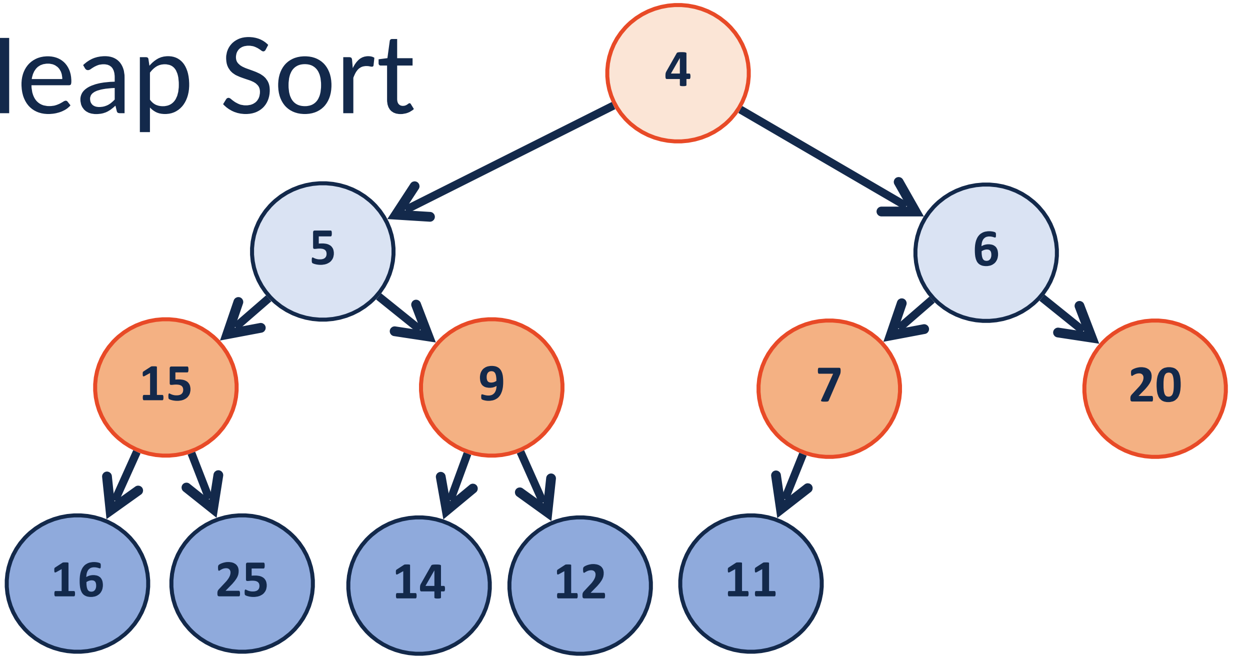
minHeap is a good example of tradeoffs:

Array is faster than tree (memory)

+ Improved construction

↪ No random access ??

Heap Sort



1. Build heap $O(n)$

2. call removeMin() n times
↳ $n \log n$

3. Reverse the array

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

Running time? $O(n \log n)$