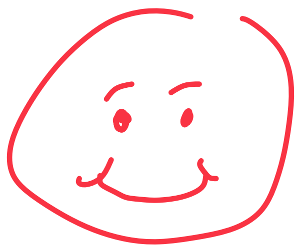
Data Structures

Heaps

CS 225

Brad Solomon & G Carl Evans

October 13, 2023



UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN
Department of Computer Science

Announcements

Reminder: Exam 3 October 16-18

Drop deadline: October 13

MP_Traversals out now!

Hand writing :(

PL!

expiration date?

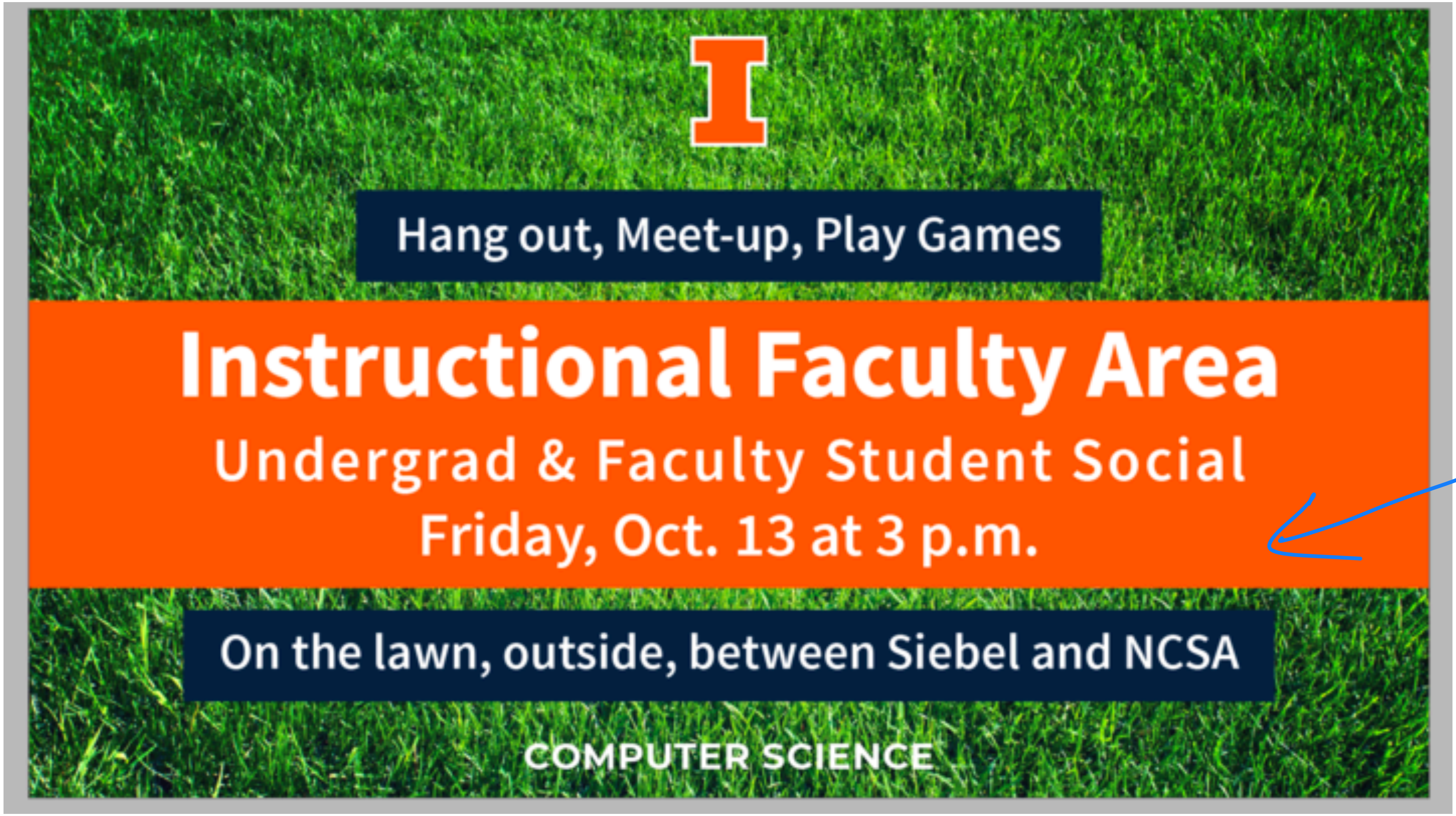
As of this morning, 1/3 of students filled out IEF!

↳ Sent an email!

↳ if 70% fill out everyone gets 5 points

CS Social Event Today!

Probably canceled due to rain :(



Learning Objectives

Review heap ADT → simple but powerful!

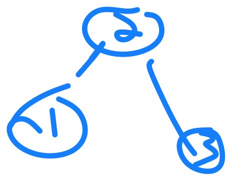
Analyze efficiency of minHeap implementations

(min)Heap

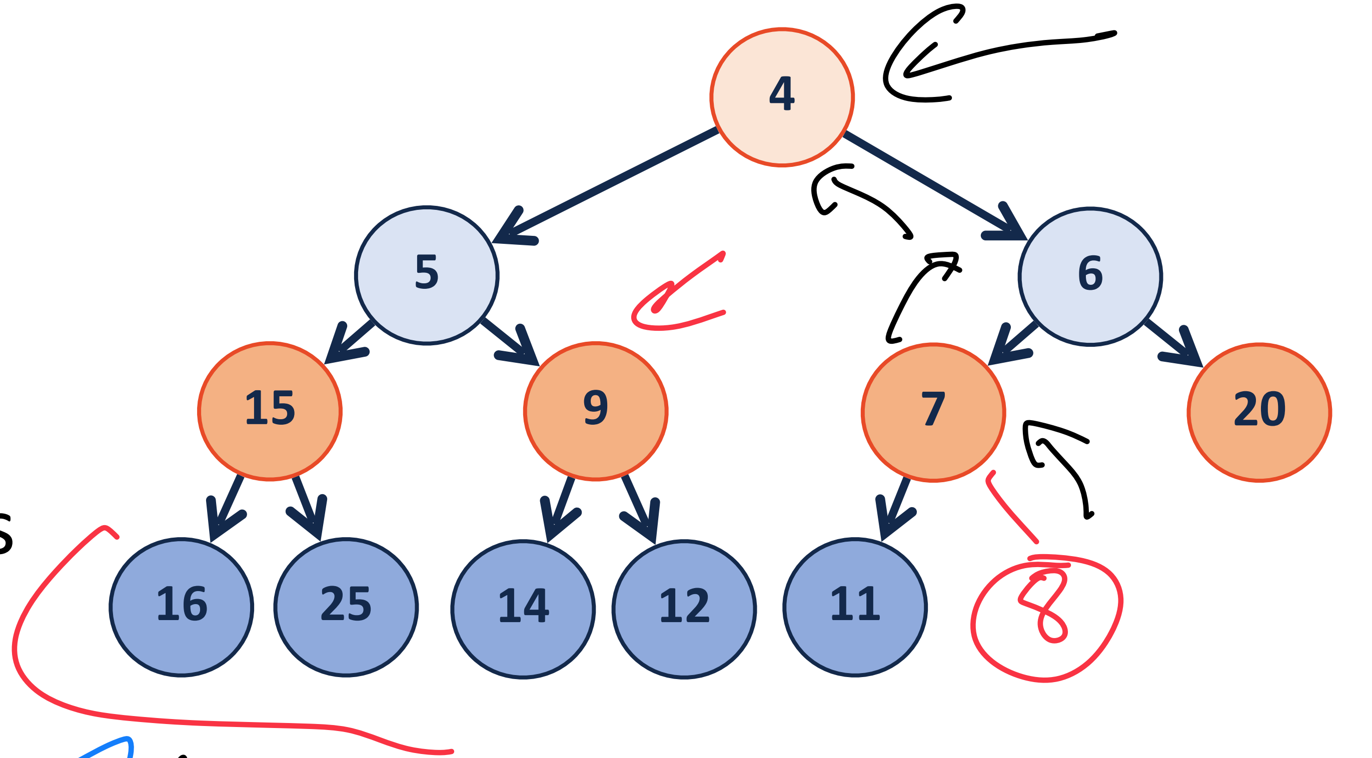
A complete binary tree T is a min-heap if:

- **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.

BST



ordered left to right



Not a search tool!

Order top to bottom

Loose order

(min)Heap

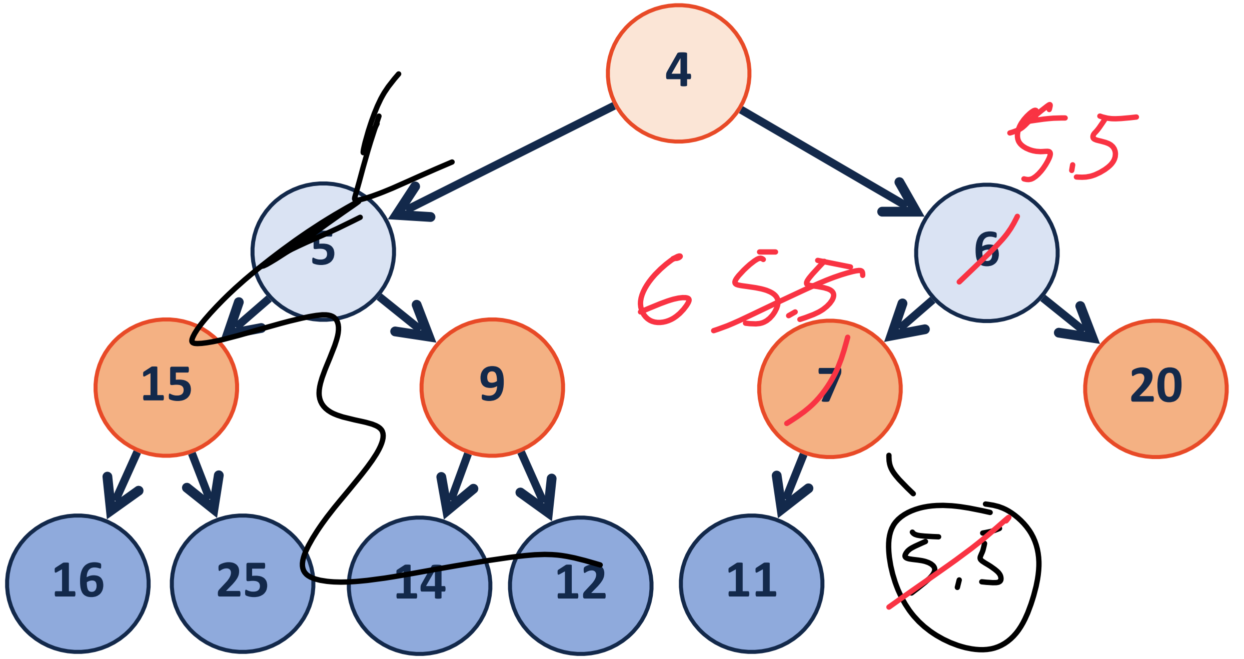
Insert (i)

↳ Array push_back (i)

↳ Swap until heap again

↳ 'heapify up'

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



complete tree is balanced

5.5

	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

This is a design decision!

removeMin

→ and return

1) Swap min value (root) w/ last value

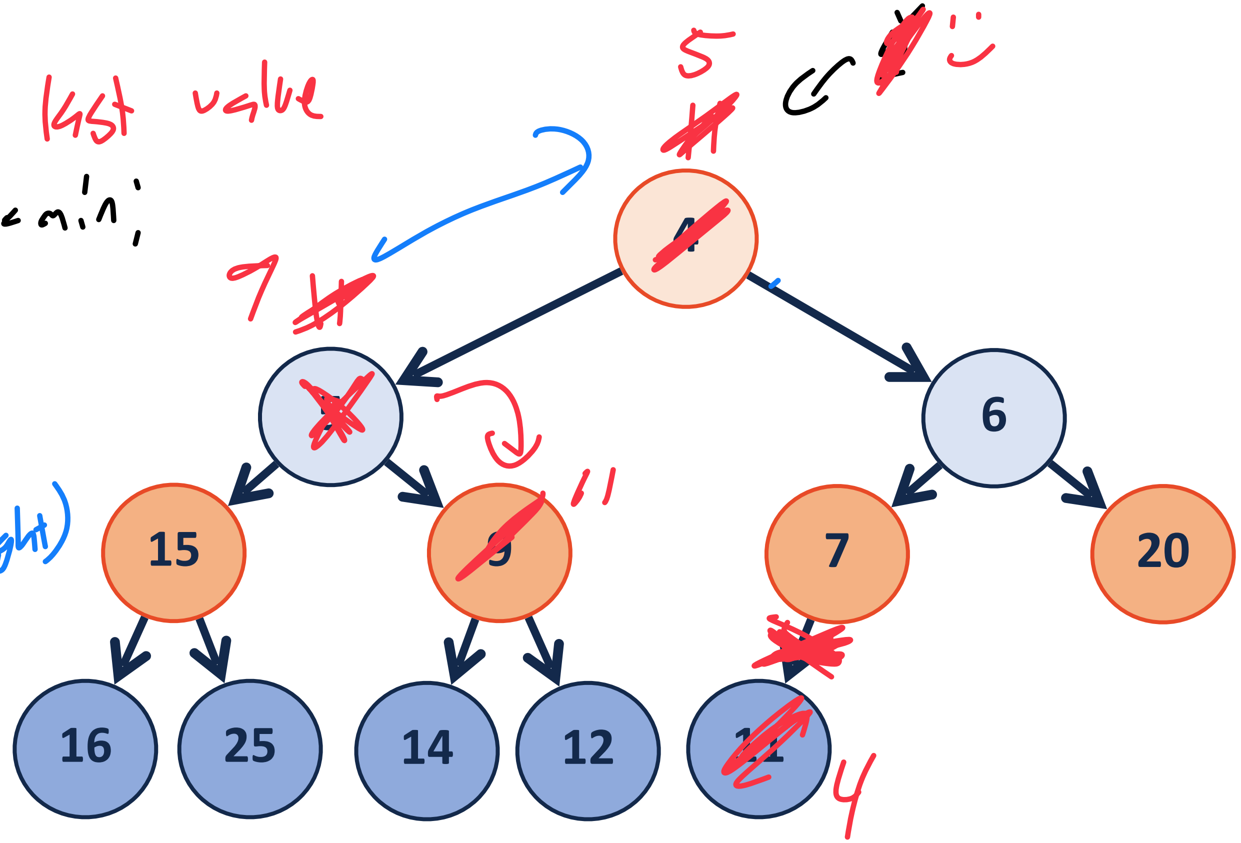
→ tmp = min;

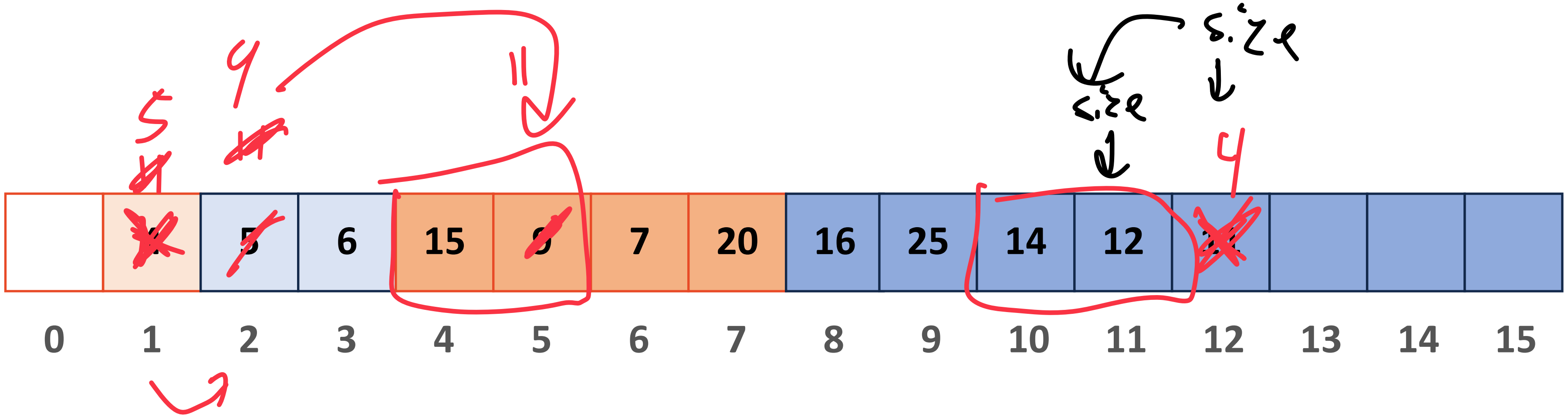
2) Delete min by size--

3) heapifyDown(root)

↳ Swap root w/ min(left, right)

recursively





removeMin

O(log n)

↕

O(h) swaps

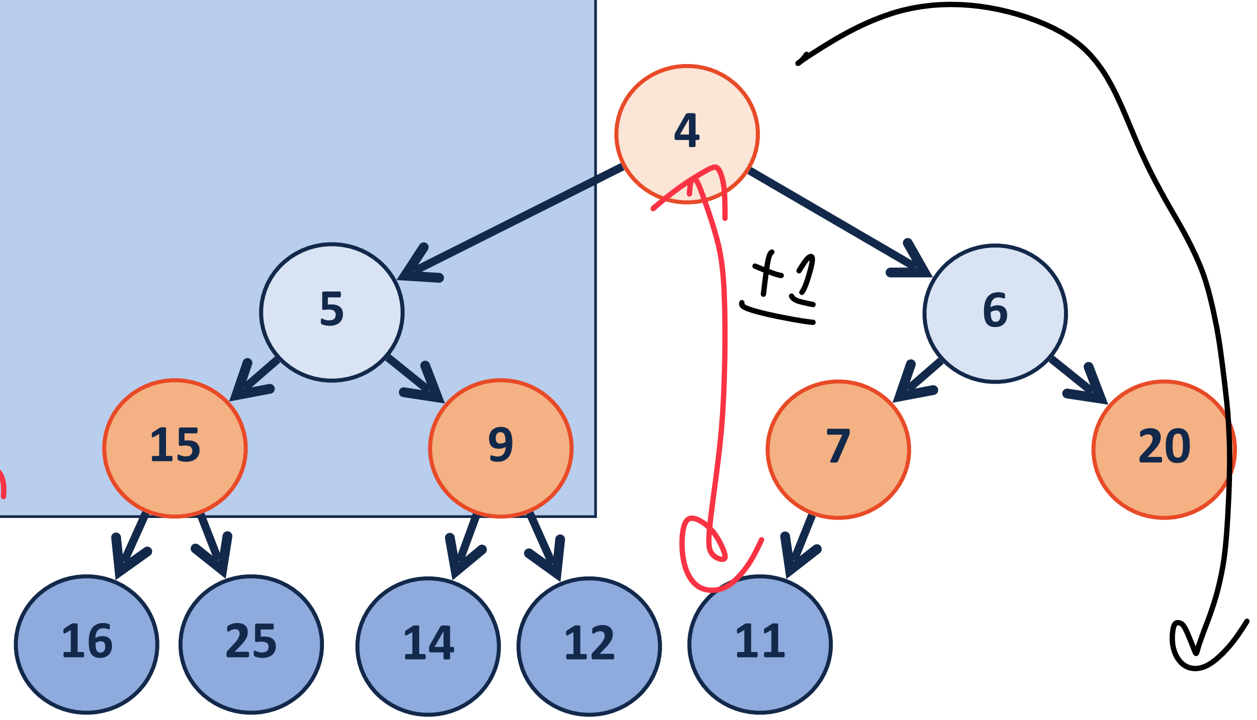
```
template <class T>
T Heap<T>::_removeMin() {
  // Swap with the last value
  T minValue = item_[1];
  item_[1] = item_[size_];
  size--;

  // Restore the heap property
  heapifyDown();

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

Store the actual min

return the min



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	~~4~~ 11	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_];
  size--;

  // Restore the heap property
  _heapifyDown();

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

i=1

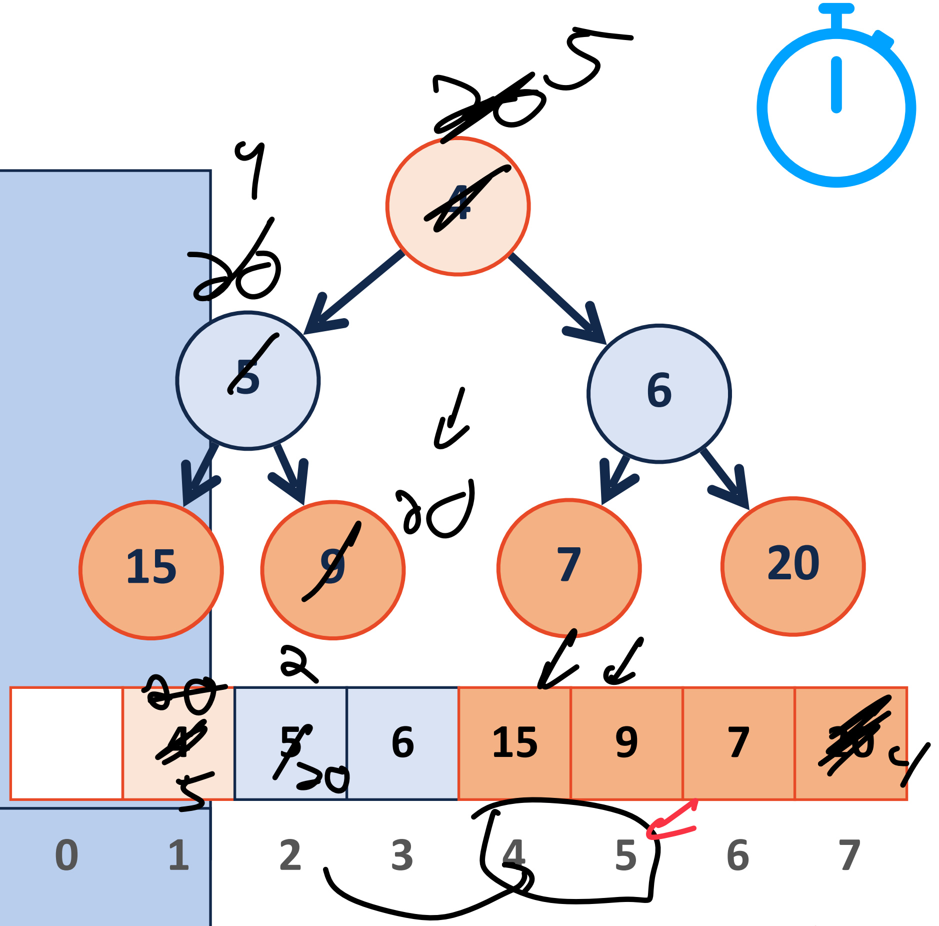
i=2

i=1

```
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 for down
- I am leaf
or
my children both larger than me

buildHeap (minHeap Constructor)

If you give me an array of data, how to build?

1) Sort the array (and add blank at front)

↳ $O(n \log n)$:'(

2) heapify up every item

↳ n items   log (n)

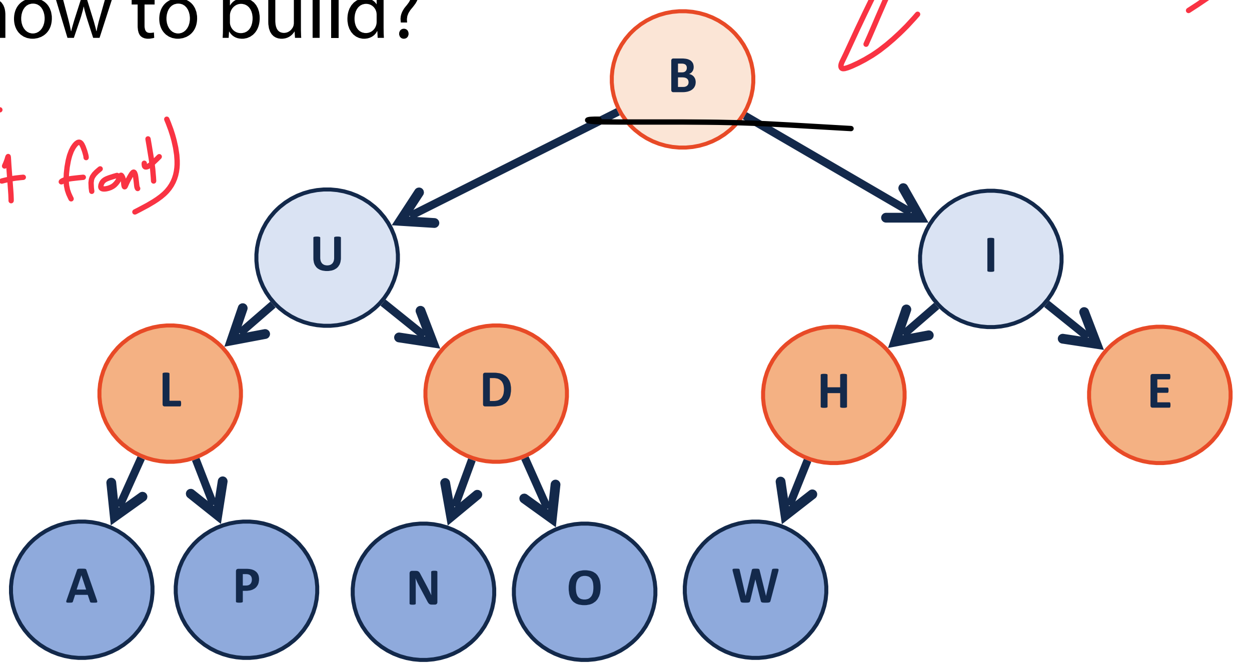
↳ n log n

3) heapify Down

↳ n items   log(n)

↳ n log n   :/

Imaginary



Heap is array

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

buildHeap – sorted array

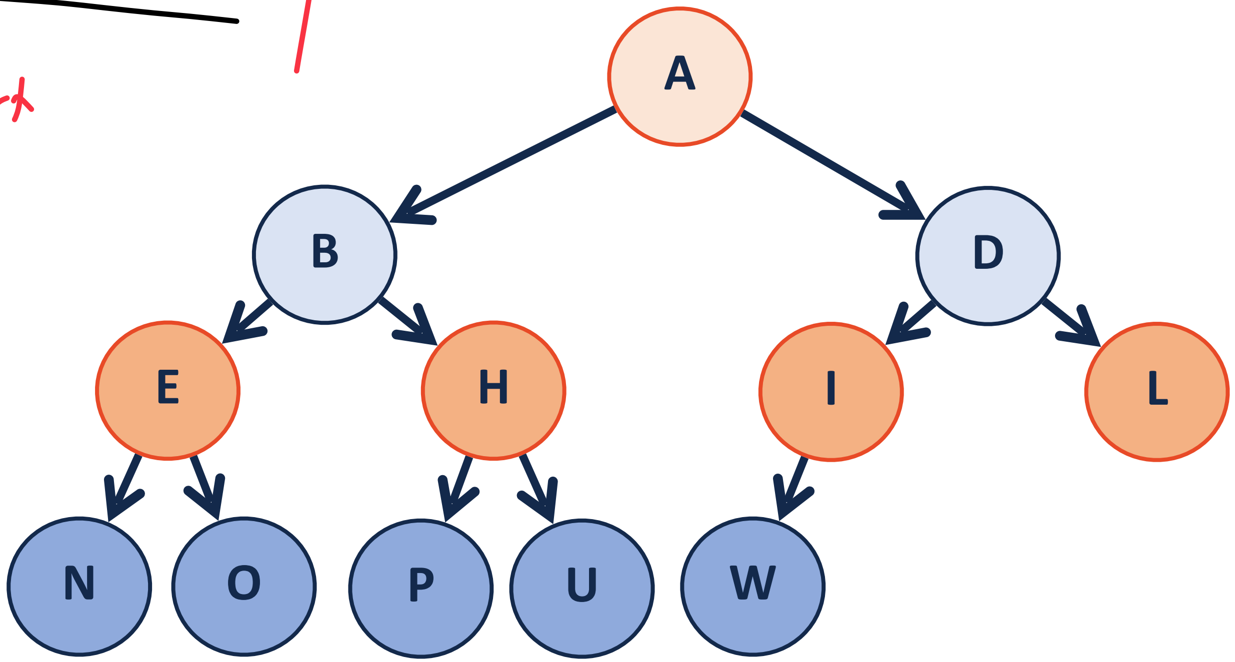
log(n)

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

→ n log n

merge sort

↳ faster sorts on specific data

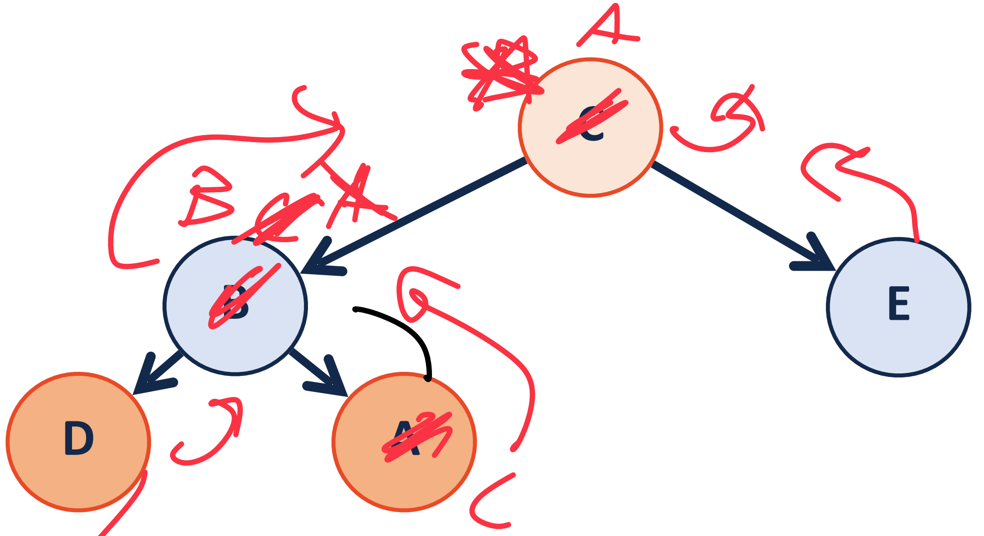


	A	B	D	E	H	I	L	N	O	P	U	W			

buildHeap - heapifyUp

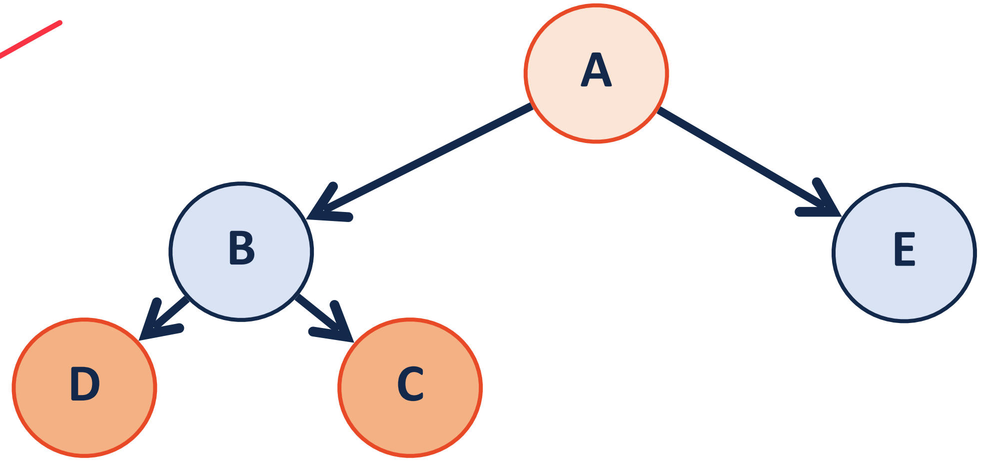
If heapifying up I need to start at index 2 (or 1 but root skips)

↳ Assume everything above me has heap property



	C	B	E	D	A		

This is doing a bunch of inserts (more or less)

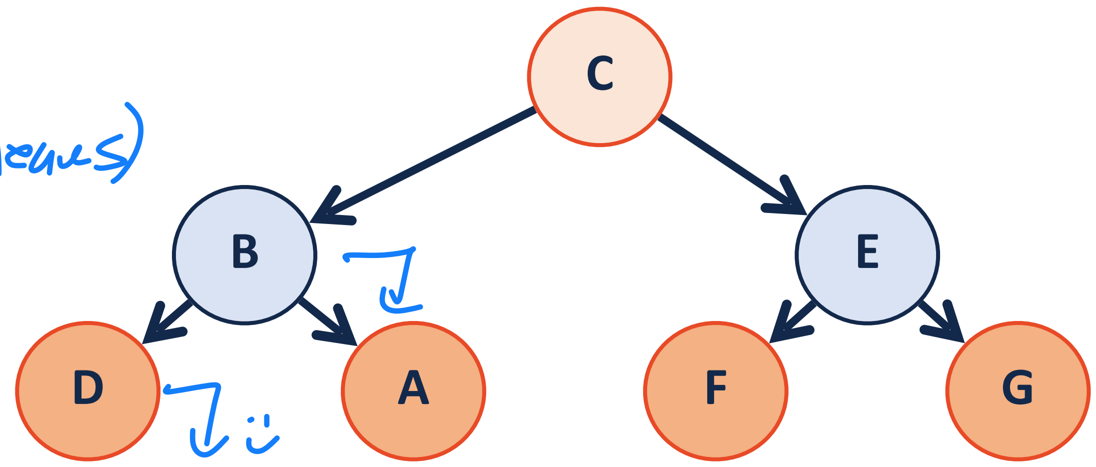


	A	B	E	D	C		

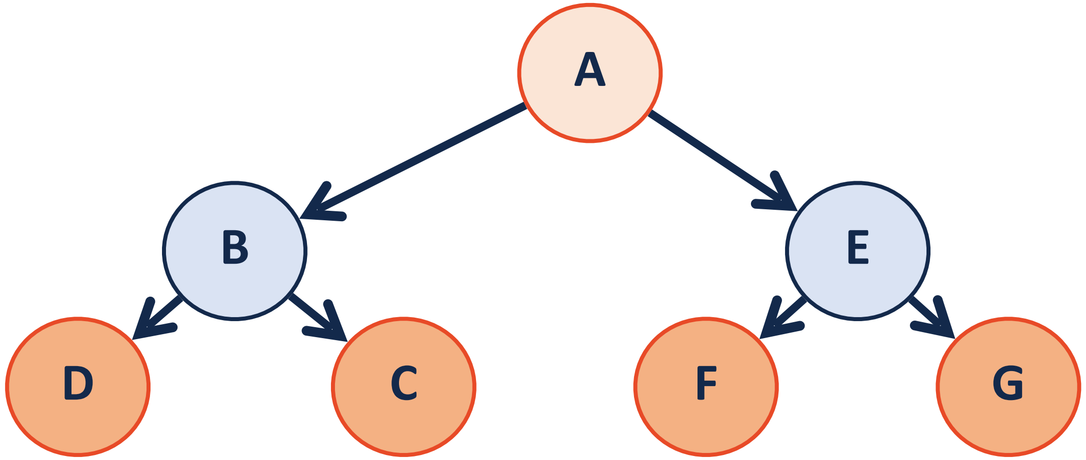
buildHeap - heapifyDown

Start from my leaves (but I can skip leaves)

↳ $\frac{size}{2} = parent(size)$



	C	B	E	D	A	F	G



	A	B	E	D	C	F	G

buildHeap

1. Sort the array — its a heap!   n log n

2. heapifyUp()   n-1

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

n log n

Said n log n

3. heapifyDown()   n/2

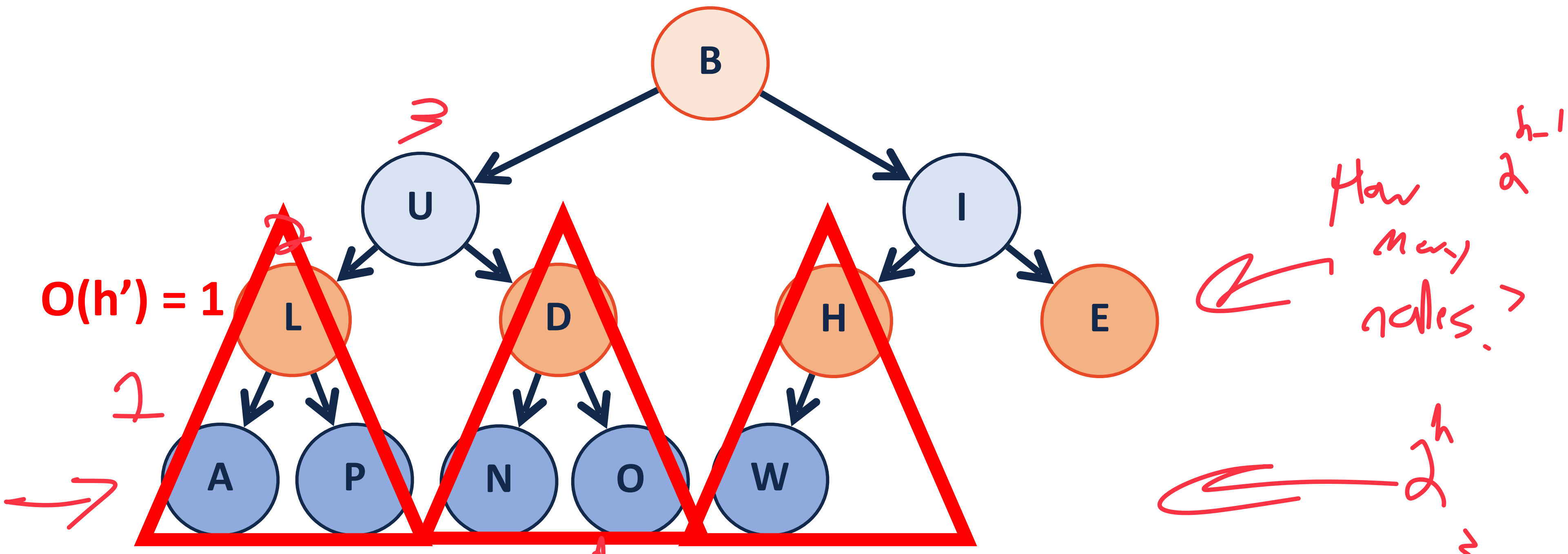
```
template <class T>
void Heap<T>::buildHeap() {
  for (unsigned i = parent(size); i > 0; i--) {
    heapifyDown(i);
  }
}
```

Be aware size of subproblem

buildHeap - heapifyDown

Lets break down the total 'amount' of work:

2^{h-1} nodes → 1 op

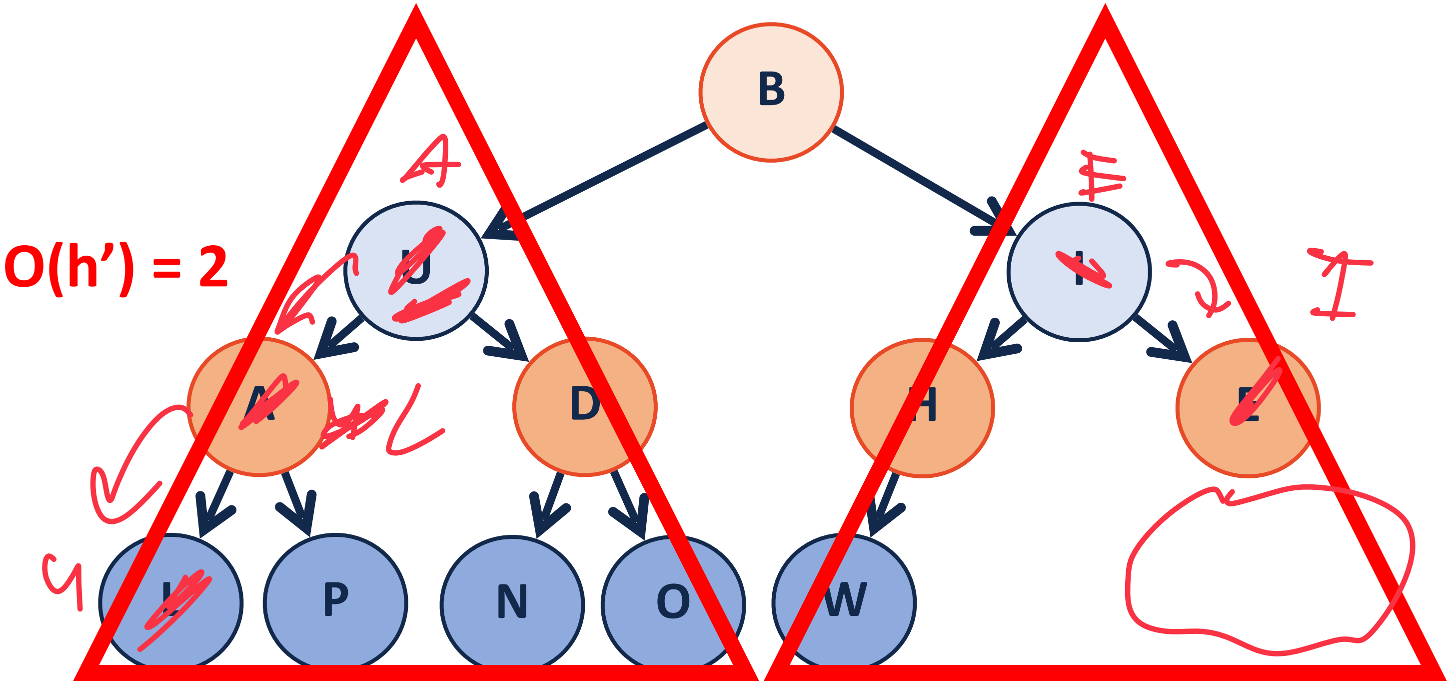


	B	U	I	L	D	H	E	A	P	N	O	W			

buildHeap - heapifyDown

Lets break down the total ‘amount’ of work:

$2^{h-2} \rightarrow 2$ ops



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

buildHeap - heapifyDown

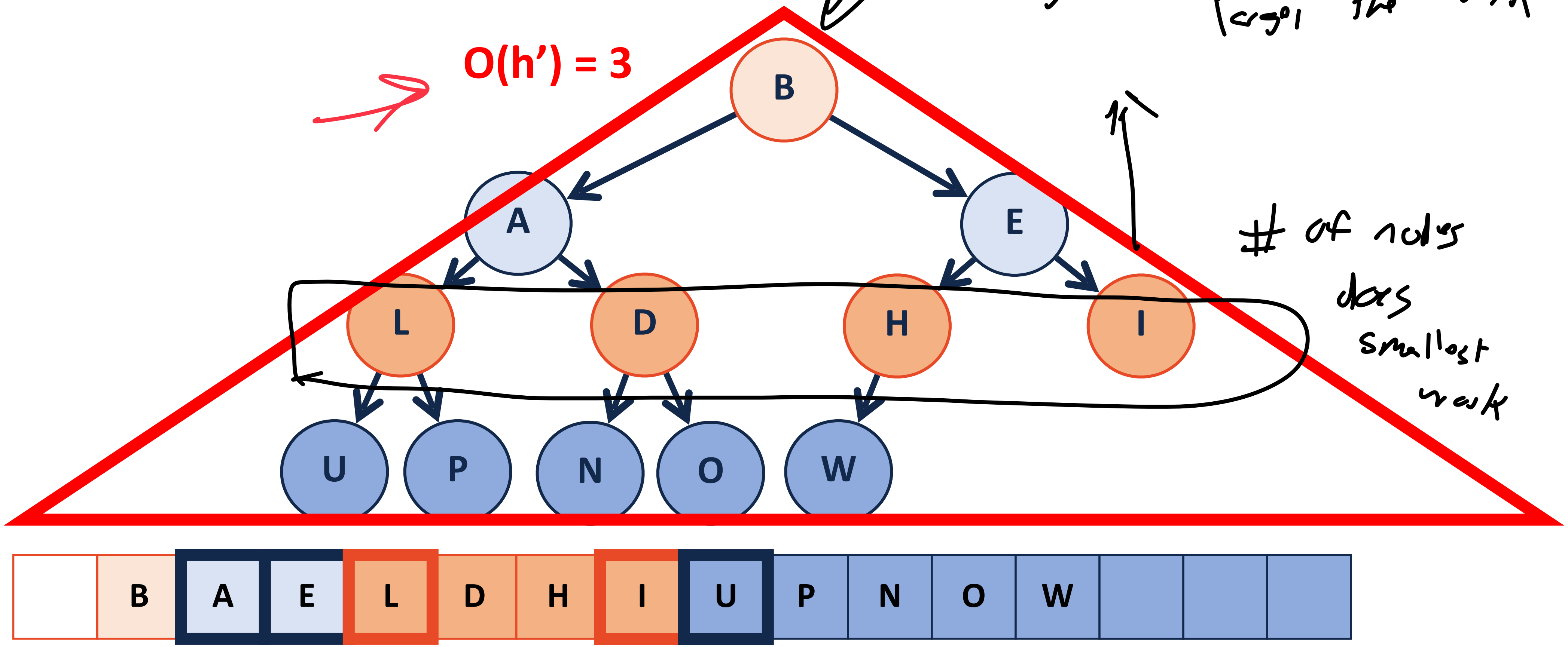
Lets break down the total 'amount' of work:

$2^{h-3} \rightarrow 3$ ops

item can do at most O(h)

smaller node

larger the work



# of nodes does smallest work

Proving buildHeap Running Time

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

Strategy:

1) Do heapify Down on every non-leaf node ← ~$n/2$

2) Worst case work for any node is its height (height of subtree)

3) Our worst case is every node swaps its max

$\parallel$

The sum of the height of every node

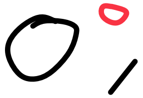
Proving buildHeap Running Time

worst case complexity is perfect

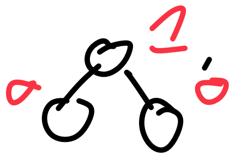
S(h): Sum of the heights of all nodes in a **perfect** tree of height **h**.

only draw one picture

S(0) = 0



S(1) = 1

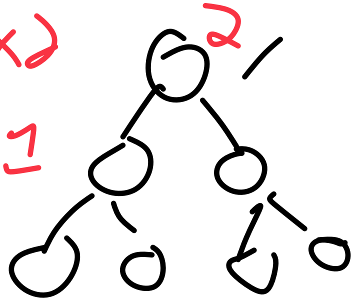


S(2) = 4

1×2

2×1

4×0



guess form

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

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

Proving buildHeap Running Time

proving

recurrence

vs induction

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

Base Case:

$h = 0$ $\quad 2^1 - 2 - 0 = 0$ ✓

$h = 1$ (1) $\quad 2^2 - 2 - 1 = 1$ ✓

eq holds for base case!

Proving buildHeap Running Time

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

↳ Assume true up to h-1

Induction Step: $S(h) = 2S(h-1) + h$

↓ we know

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

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

So plug it in!

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

$2^{h+1} - 4 - 2h + 2 + h$

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

???

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

Proving buildHeap Running Time

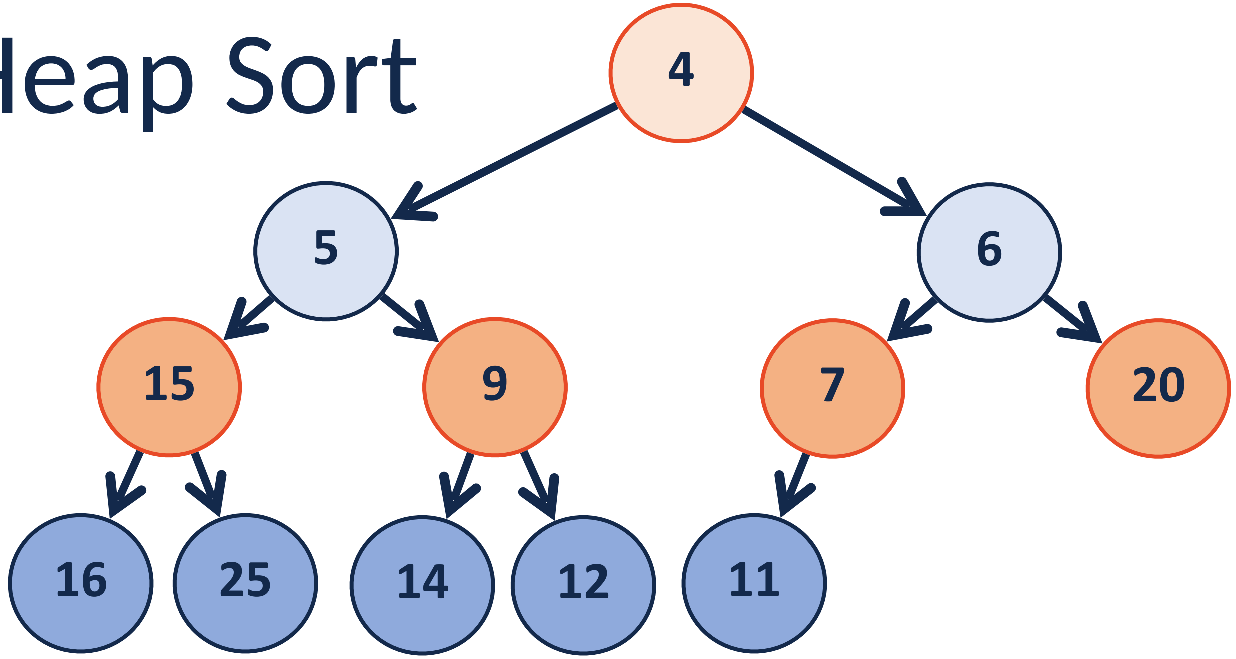
Theorem: The running time of buildHeap on array of size **n** is O(n)

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

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

How can we estimate running time?

Heap Sort



1. $O(n)$

2.

3.

$n \log n$

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

Running time?

minHeap is a good example of tradeoffs: