AVL Trees
Heaps
And Complexity

D. Thiebaut
CSC212 — Fall 2014

Some material taken from http://cseweb.ucsd.edu/~kube/cls/100/Lectures/lec4.avl/lec4.pdf

Complexity
Of BST Operations

or

"Why Should
We Use BST
Data Structures"

What We're After:

- What's the **worst** amount of work we can expect
    - when we insert?
    - when we delete?
    - when we search, successfully or unsuccessfully?
- What's the **average** amount of work we can expect?
- We'd like to know about the **best** case, but in general it doesn't occur that often.

Which Two-Operations In BSTs are Similar?

- Search/Find
  - Successful
  - Unsuccessful
- Insert
- Delete

Worst Case for *Inserts* or Unsuccessful *Searches*

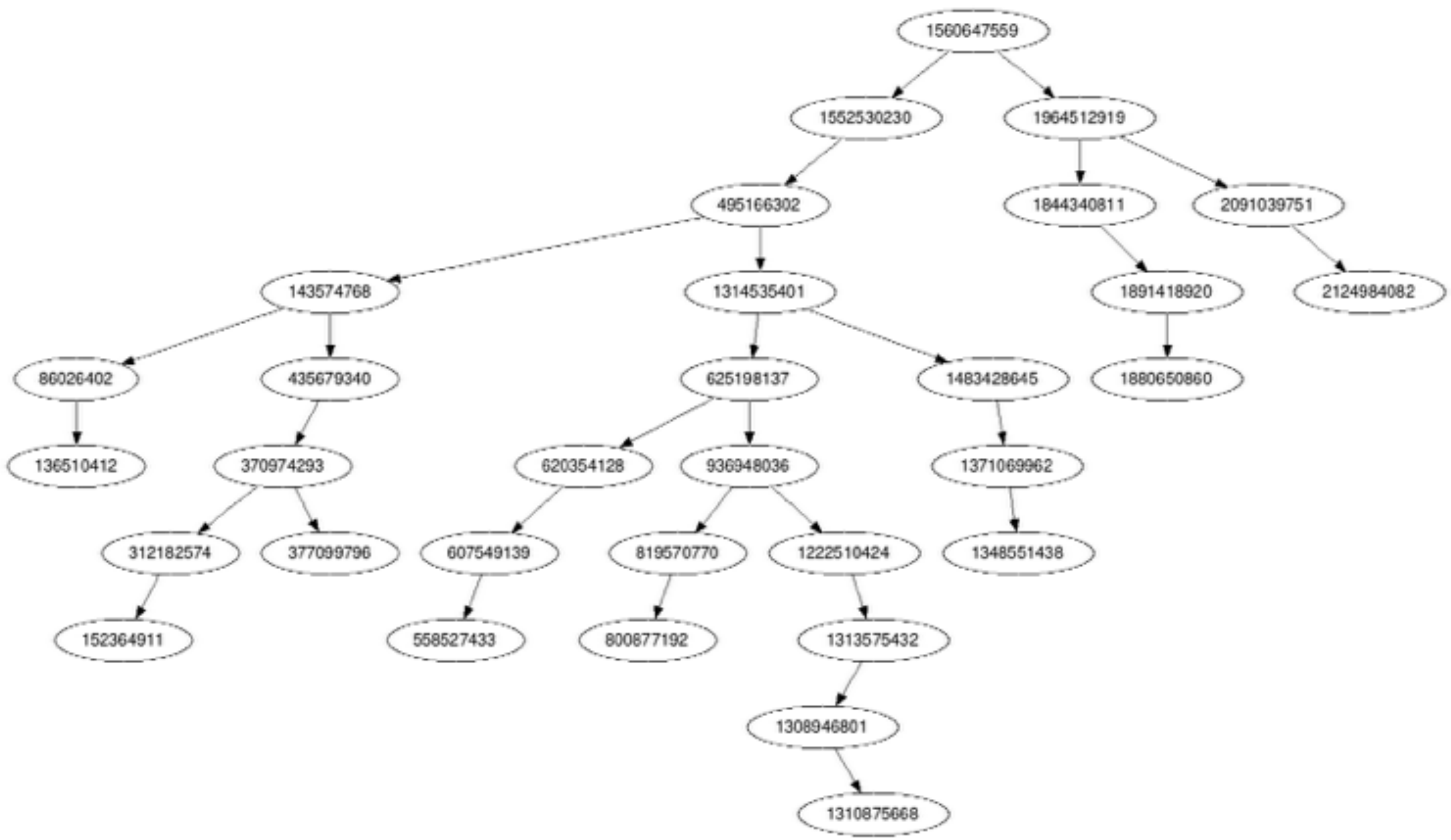
- **Definition:** The depth of a node *x*, *d(x)*, is the # of nodes from the root to that node. *d(x)* = 0-based level of the node plus 1.
- What is the worst possible number of nodes visited for searching or inserting in a BST?
- So, what is the worst-case complexity for insert or unsuccessful operations?

Operation	Outcome	Worst Case Complexity	Average Complexity	Best Complexity
Insertion	*X*	***O(N)***		
Search	*Successful*			
	Unsuccessful	***O(N)***		
Deletion	*X*			

Best-Case complexity for search?

Operation	Outcome	Worst Case Complexity	Average Complexity	Best Complexity
Insertion	*X*	*O(N)*		
Search	*Successful*			***O(1)***
	Unsuccessful	*O(N)*		***O(1)***
Deletion	*X*			

Average Case for *Successful Searches*



Need More Definitions

- BST of N nodes
- key_i = key residing in node x_i, i=1, 2… N
- $d(x_i)$ = depth of node x_i
- p_i = probability of searching for key_i

What is the average number of nodes visited when searching a BST of N nodes?

$$D_{avg}(N) = \sum_{i=1}^{N} d_i p_i$$

So, we need to know p_i…

- if all the keys are **equally likely**, then all the p_i are identical.

- if the p_i are **not identical**, then some keys are more likely than others, and we can take advantage of this → *self-adjusting trees* (Section 6.8)

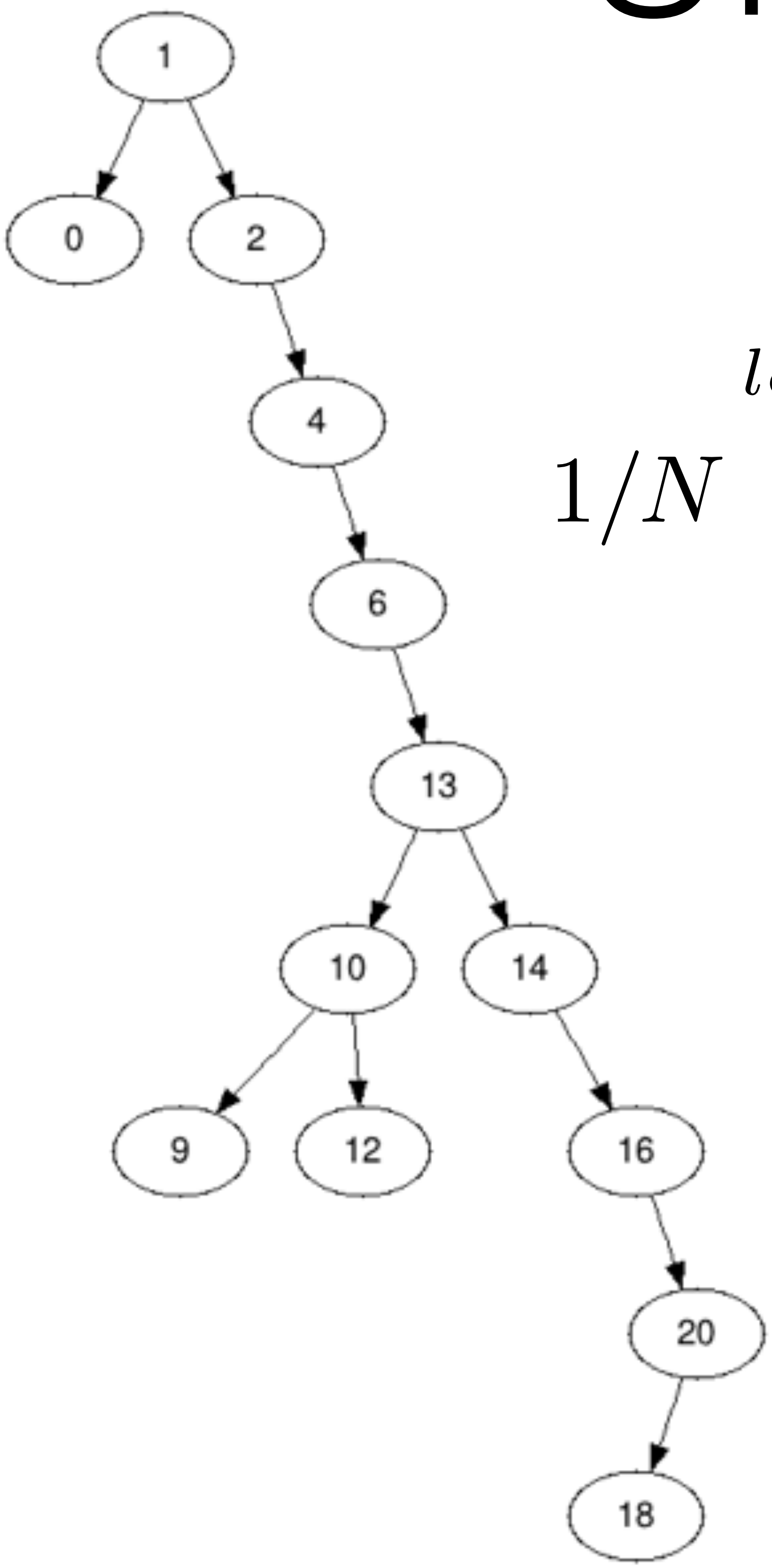
Equally Likely Keys

- equally likely keys: $p_i = 1/N$

$$D_{avg}(N) = 1/N \sum_{i=1}^{N} d_i = \text{total node depth}$$

- $D_{avg}(N)$ depends on tree shape!

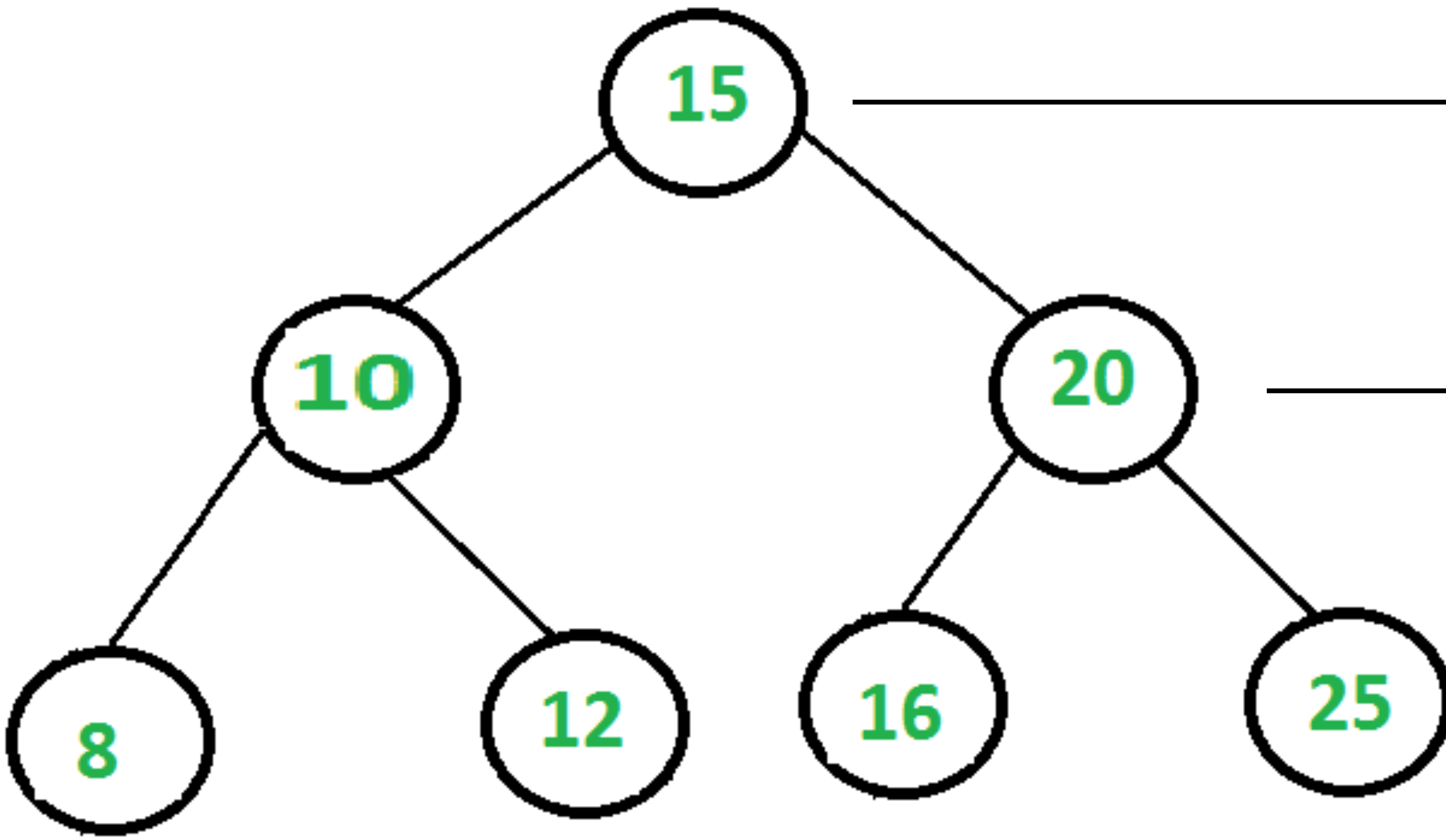
Unbalanced Tree



$$1/N \sum_{i=1}^{log(N+1)} d_i p_i = 1/N(1+2+3+...+N)$$

= ?

Fully Balanced Tree



1x1 = 1x2$^{(1-1)}$

2x2 = 2x2$^{(2-1)}$

3x4 = 3x2$^{(3-1)}$

*i*x2$^{(i-1)}$

$$1/N \sum_{i=1}^{log(N+1)} i2^{(i-1)} < \log(N+1) = O(\log N)$$

Successful Search:

- O(N) Unbalanced Trees
- O(log N)? Balanced Trees

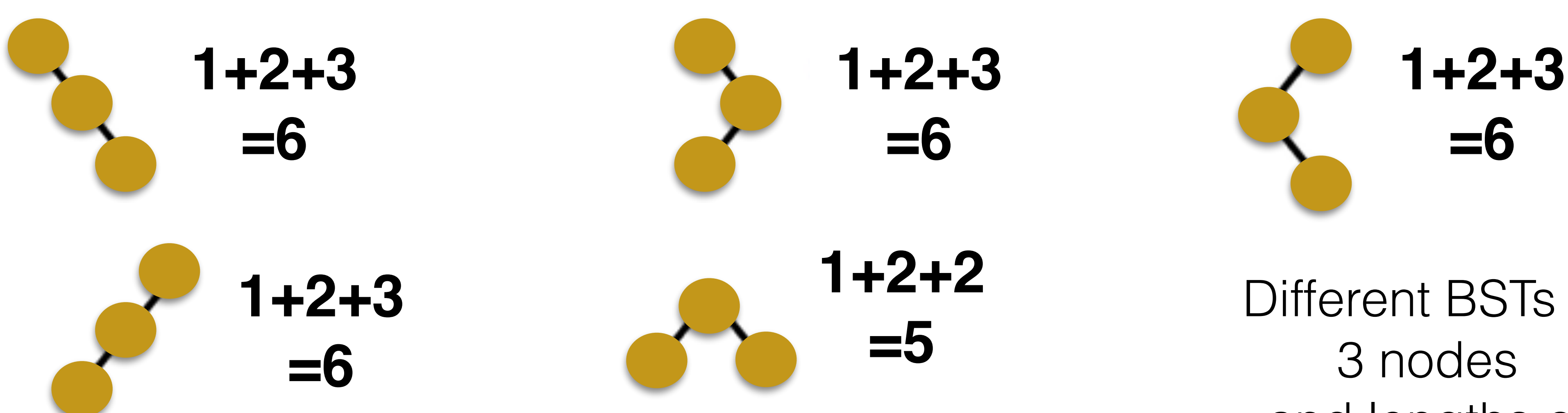
But…
Are average
BSTs tall or
fat?

What about Successful Search in a Random BST?

What about Successful Search in a Random BST?

- We need to look at all possible shapes the BST of *N* nodes can have and compute the average number of probes required for each successful search, and average over all possible shapes of the BST…

- Huge amount of combinations!



Different BSTs of
3 nodes
and lengths of
different paths for each

Average # of Probes
for Successful Searches in a "random" BST
of *N* nodes:

$$D_{avg}(N) \approx 1.386 \log_2 N$$

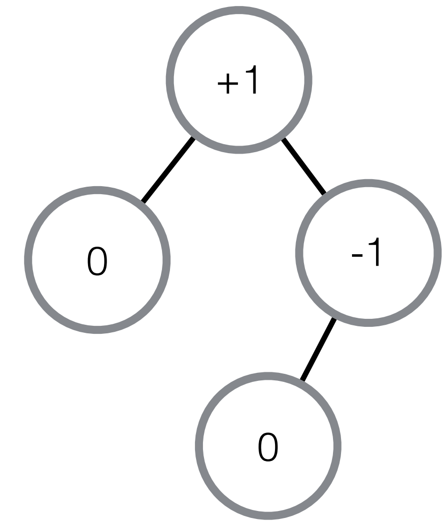
We don't really care

Operation	Outcome	Worst Case Complexity	Average Complexity	Best Complexity
Insertion	*X*	*O(N)*	O(log *N*)	O(1)
Search	*Successful*	*O(N)*	O(log *N*)	*O(1)*
	Unsuccessful	*O(N)*	O(log *N*)	*O(1)*
Deletion	*X*	*O(N)*	O(log *N*)	*O(1)*

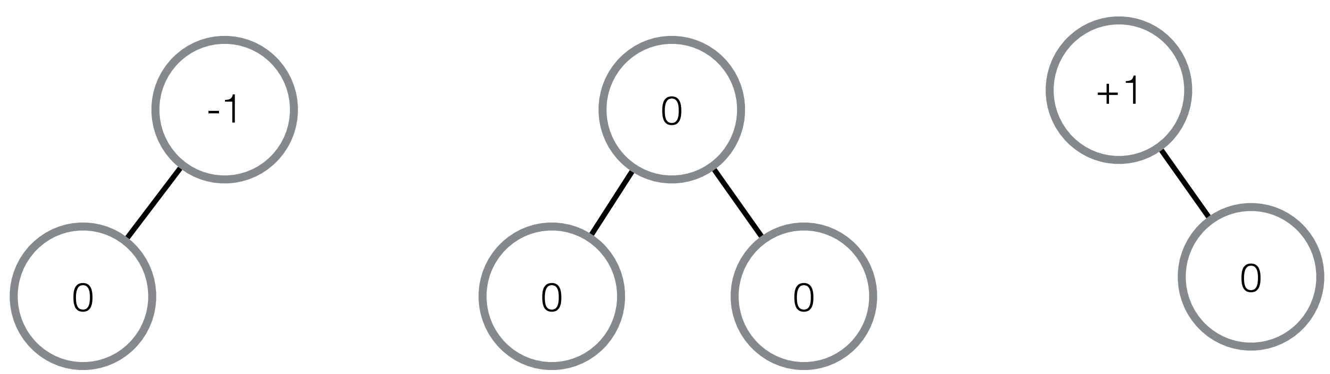
Unlikely
to happen

That's what
we expect

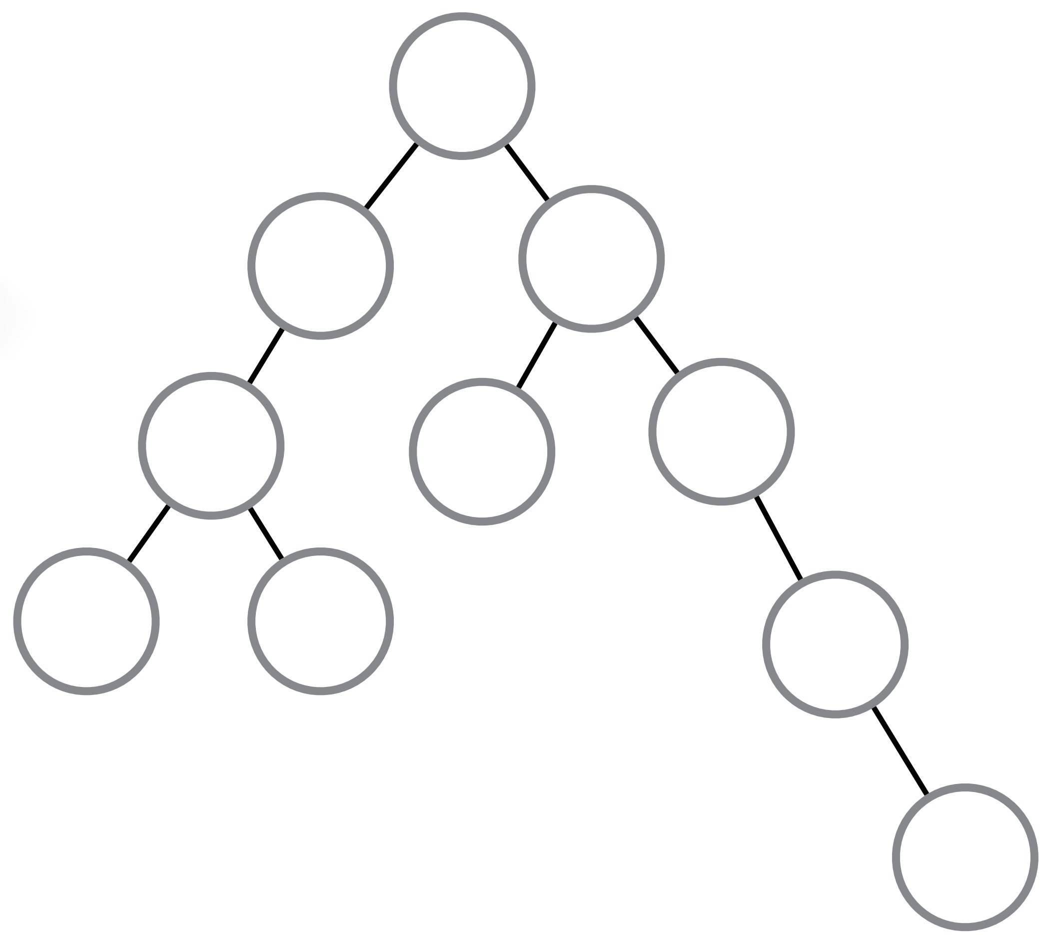
AVL-Trees



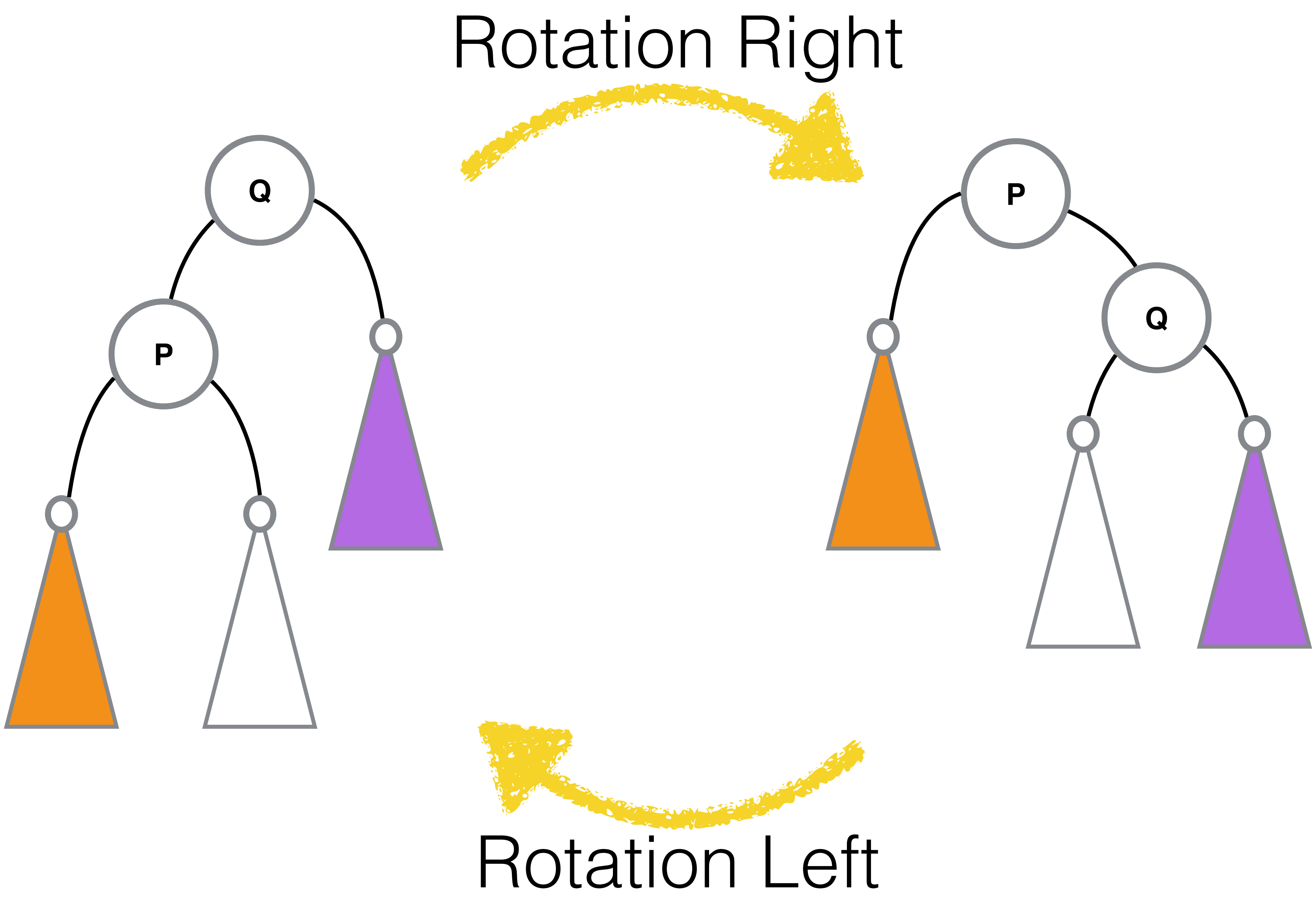
- Named for Adelson-Velskii and Landis
- 1962
- Important property: For any node X in the tree, the heights of the left and right subtrees of X differ by at most 1



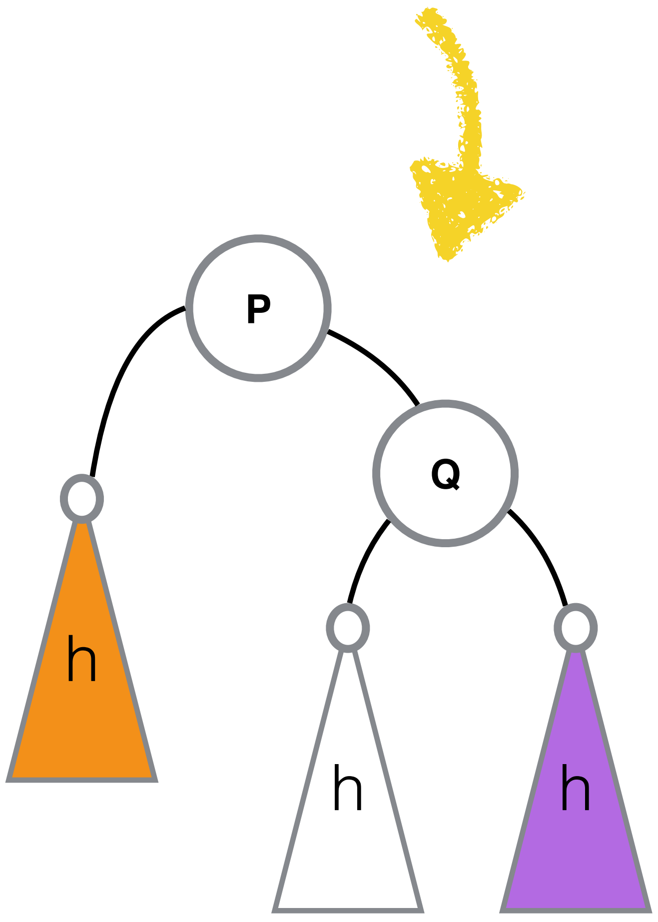
AVL

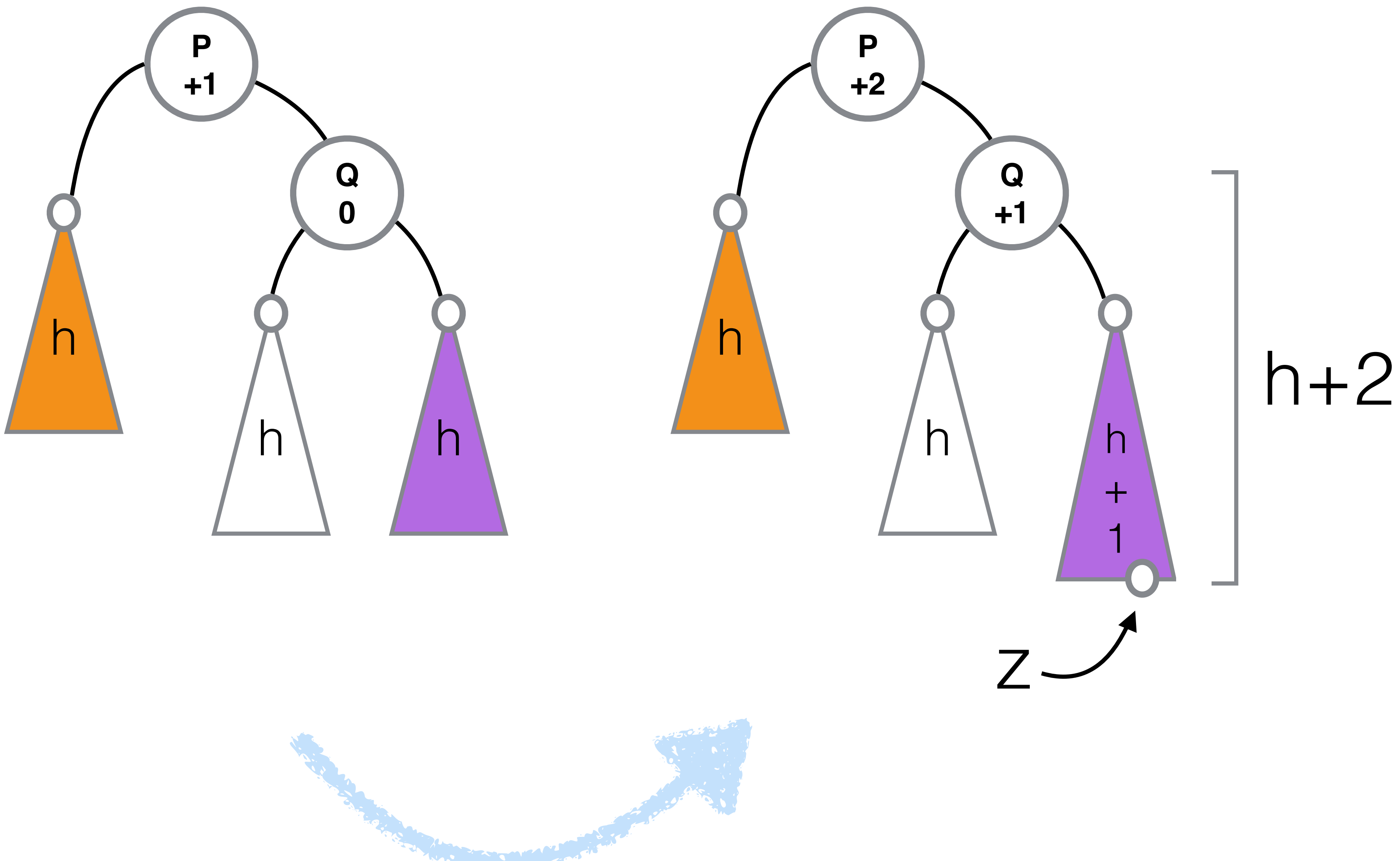


AVL Trees Rely on *Rotations*

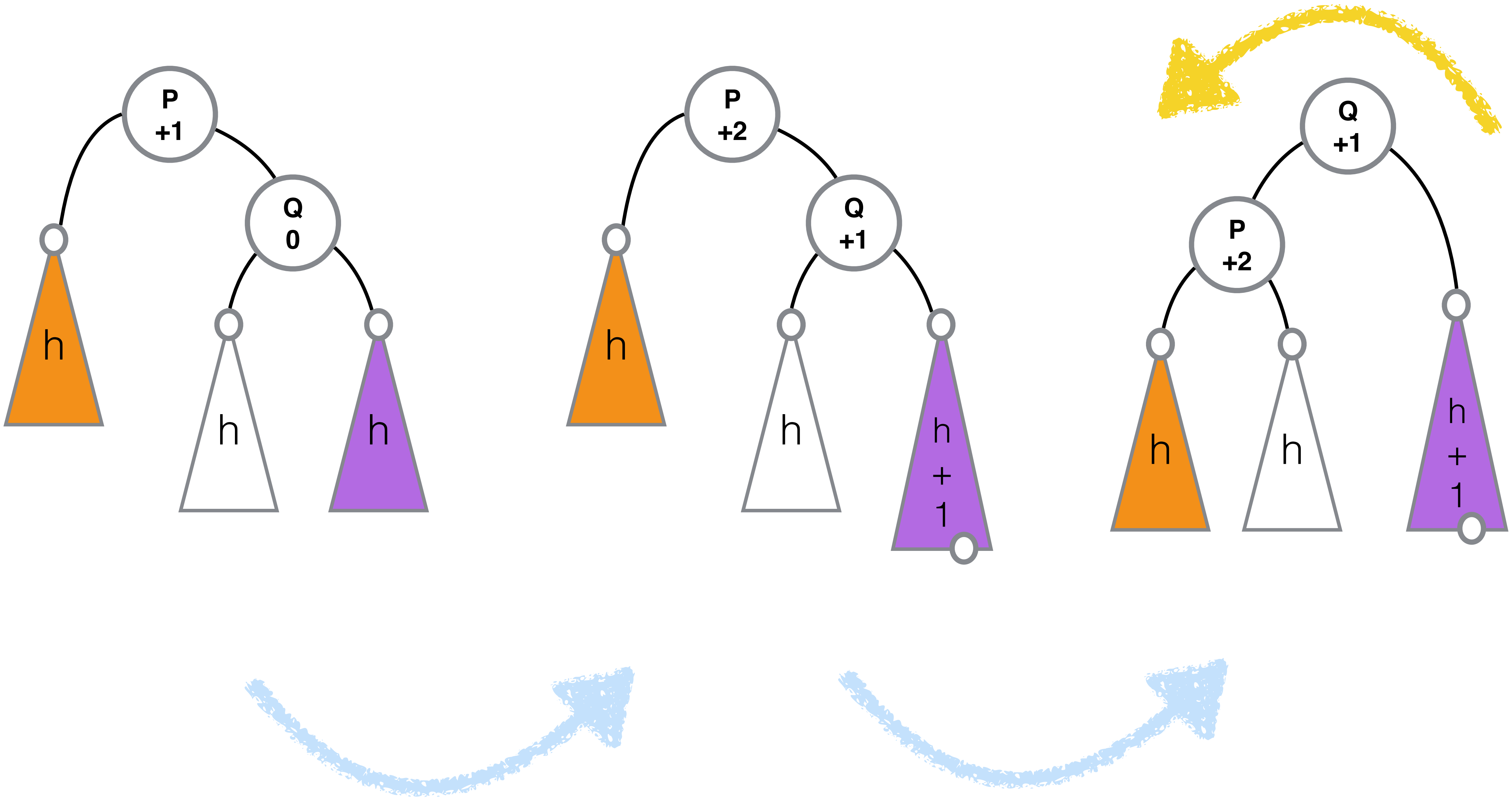


add 'Z'




P
+1
Q
0
h
h
h
P
+2
Q
+1
h
h
h
+
1
h+2
Z

Rotation Left



AVL Time Complexity

Operation	Worst Case Complexity	Average Complexity
Insertion	O(log *N*)	O(log *N*)
Search	O(log *N*)	O(log *N*)
Deletion	O(log *N*)	O(log *N*)

Java

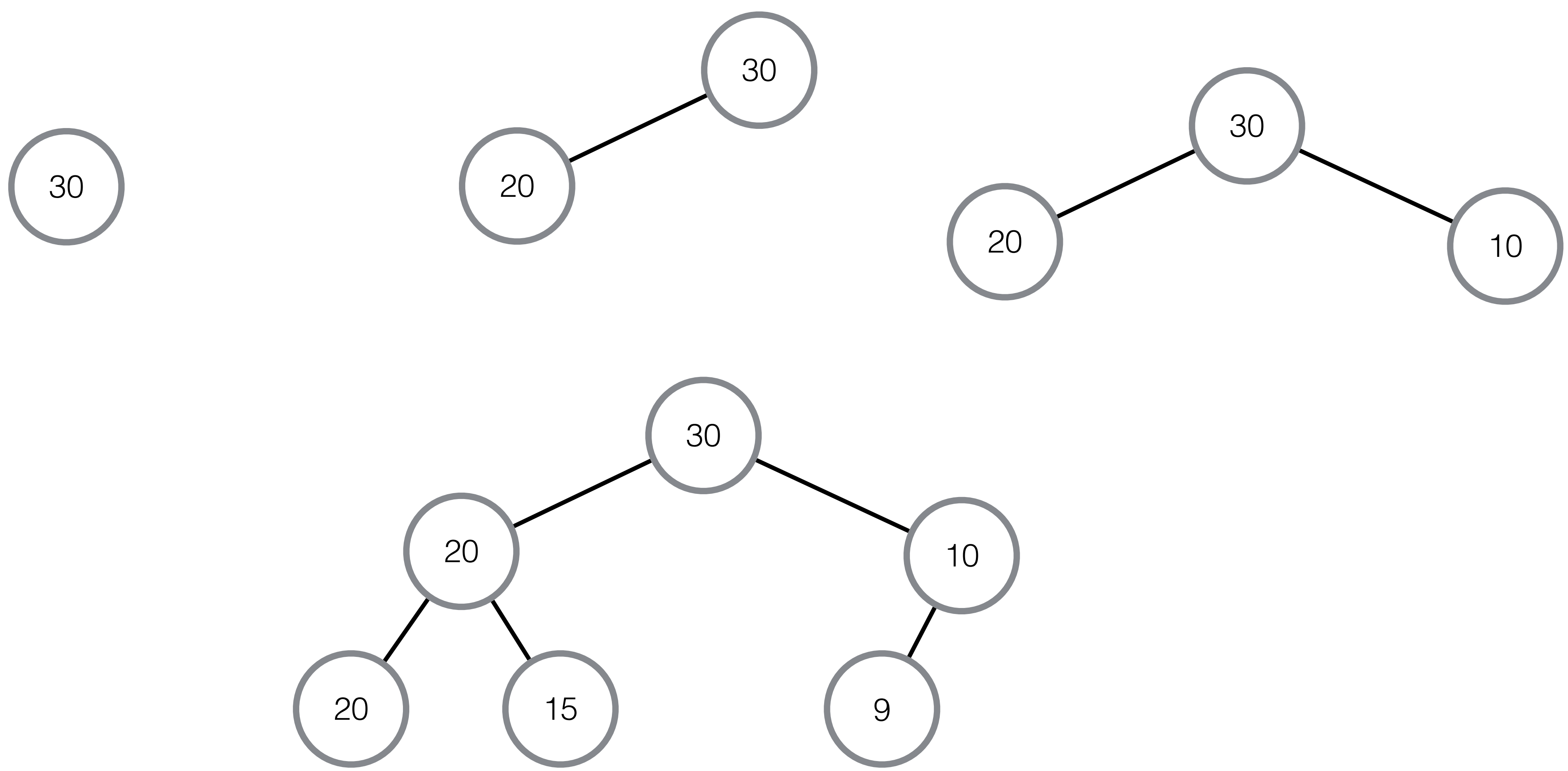
- Not that many "pure" trees
- **javax.swing.tree**
- **javax.swing.tree** TreeModel
- **javax.swing.tree** TreeNode


Heaps

A heap is:

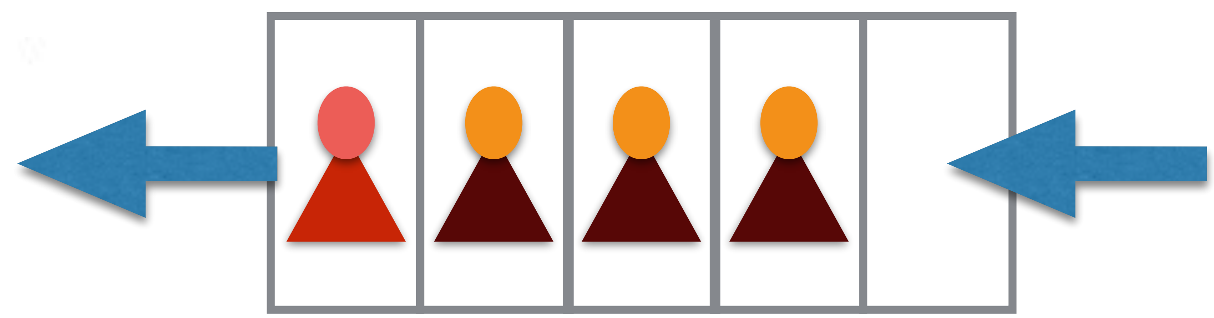
- A **fully balanced binary tree**, will all leaves on the **left-most** inner nodes

- The key of a *parent* is **larger** than or equal to the key of its *children*



Typically used for…

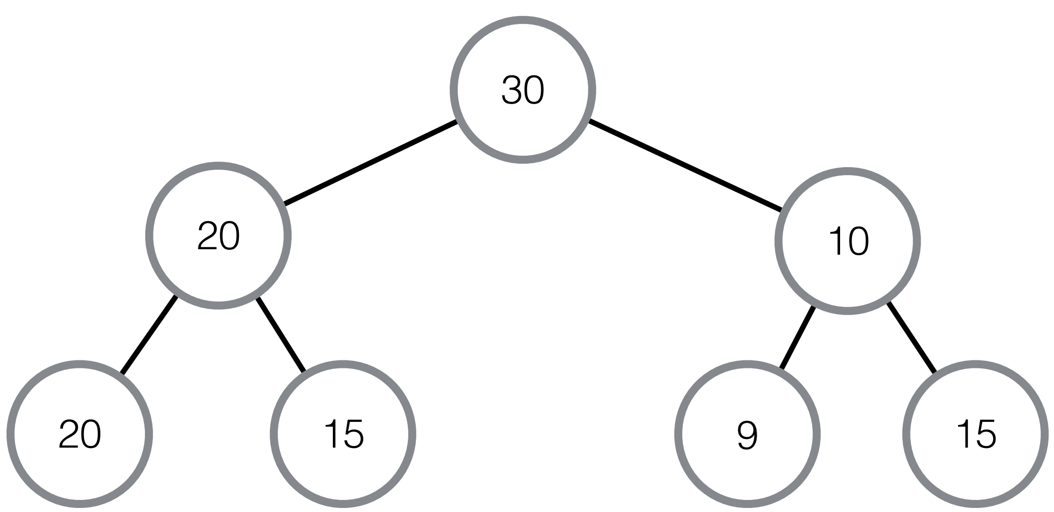
- **Priority queues**: get the next element with the highest priority



First-In…
Highest-Priority Out

Properties

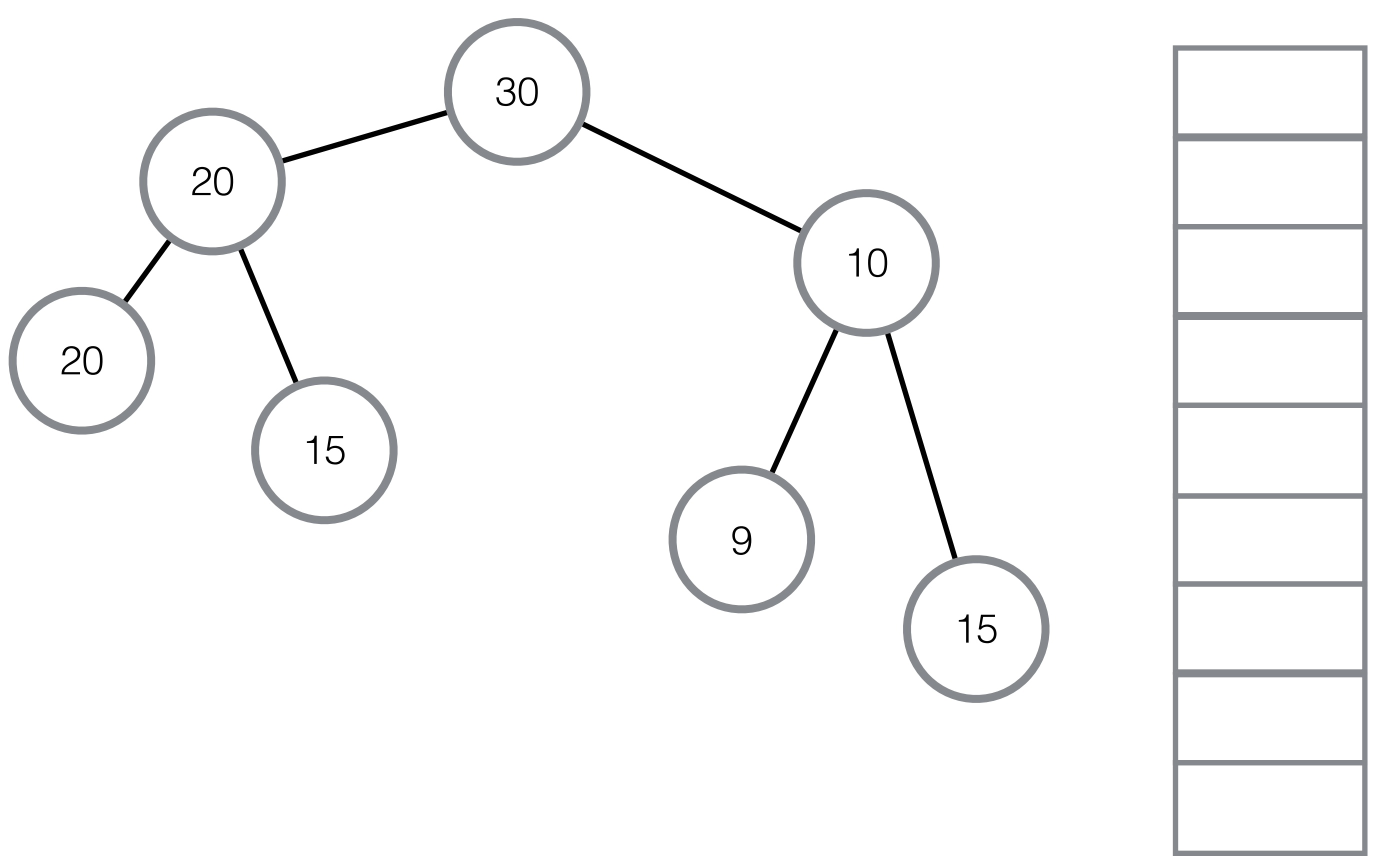
- The largest element is in the root, always
- The heap folds nicely…



What can you say of the keys other than the root?

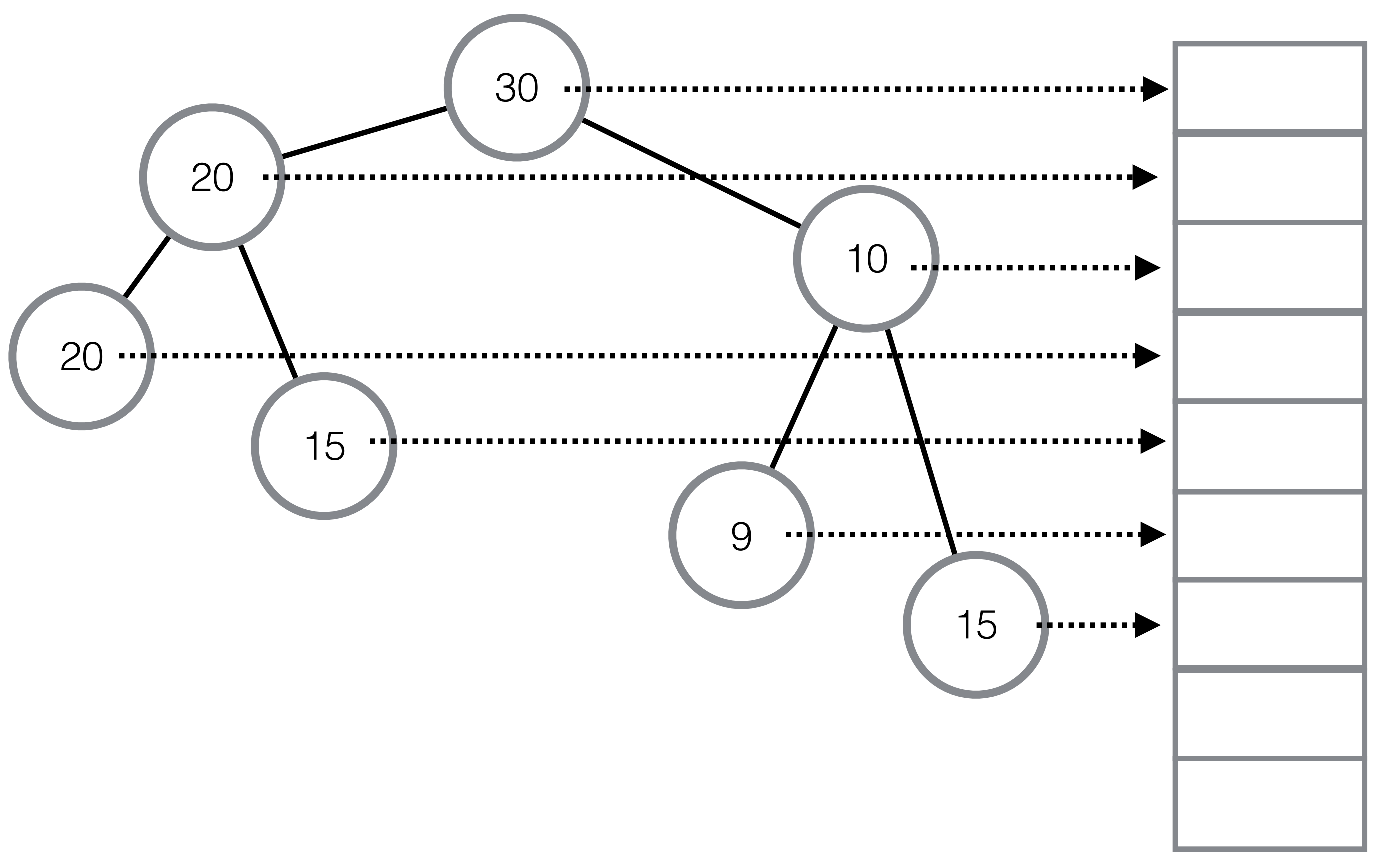
Properties

- The largest element is in the root, always
- The heap folds nicely…



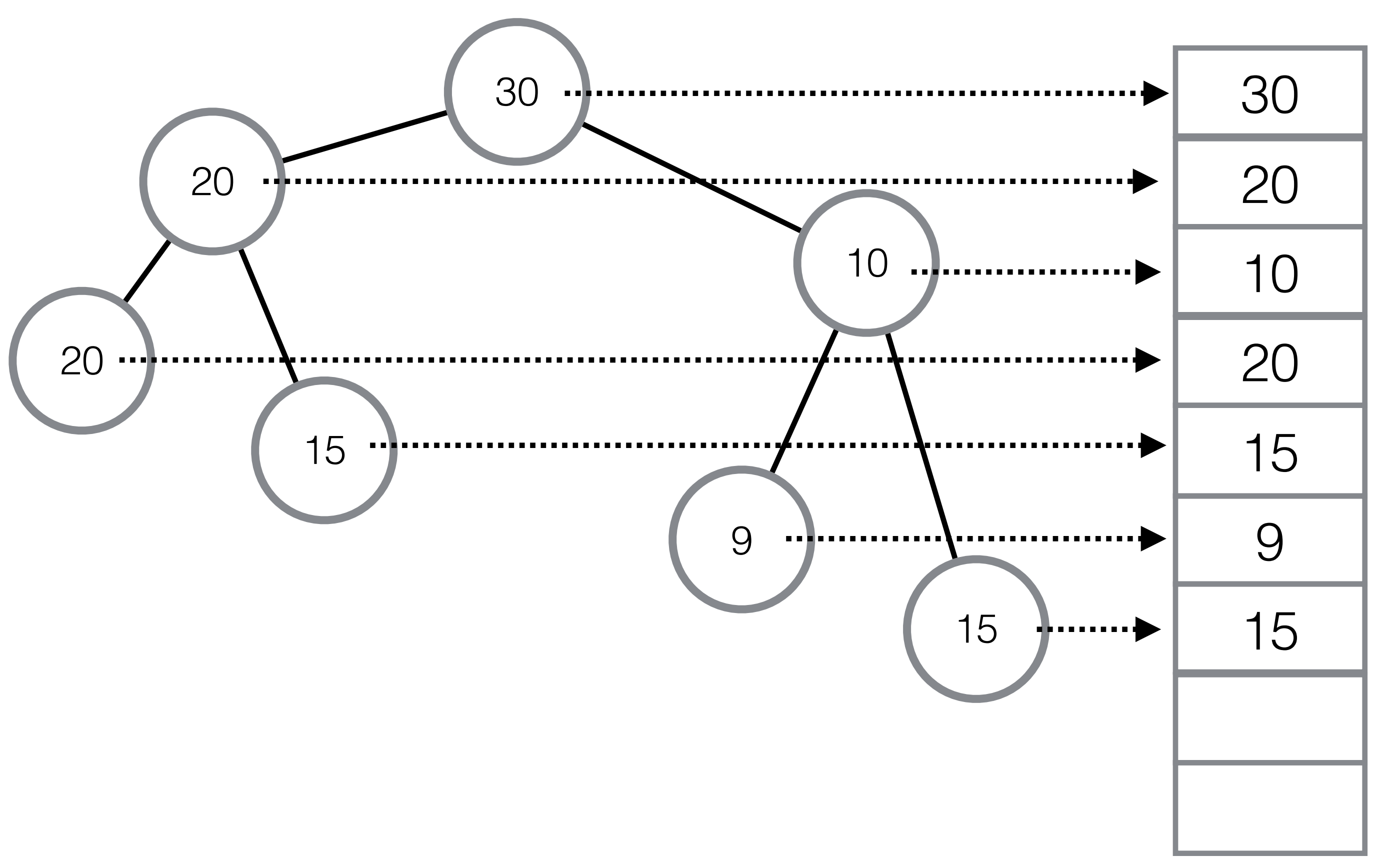
Properties

- The largest element is in the root, always
- The heap folds nicely…



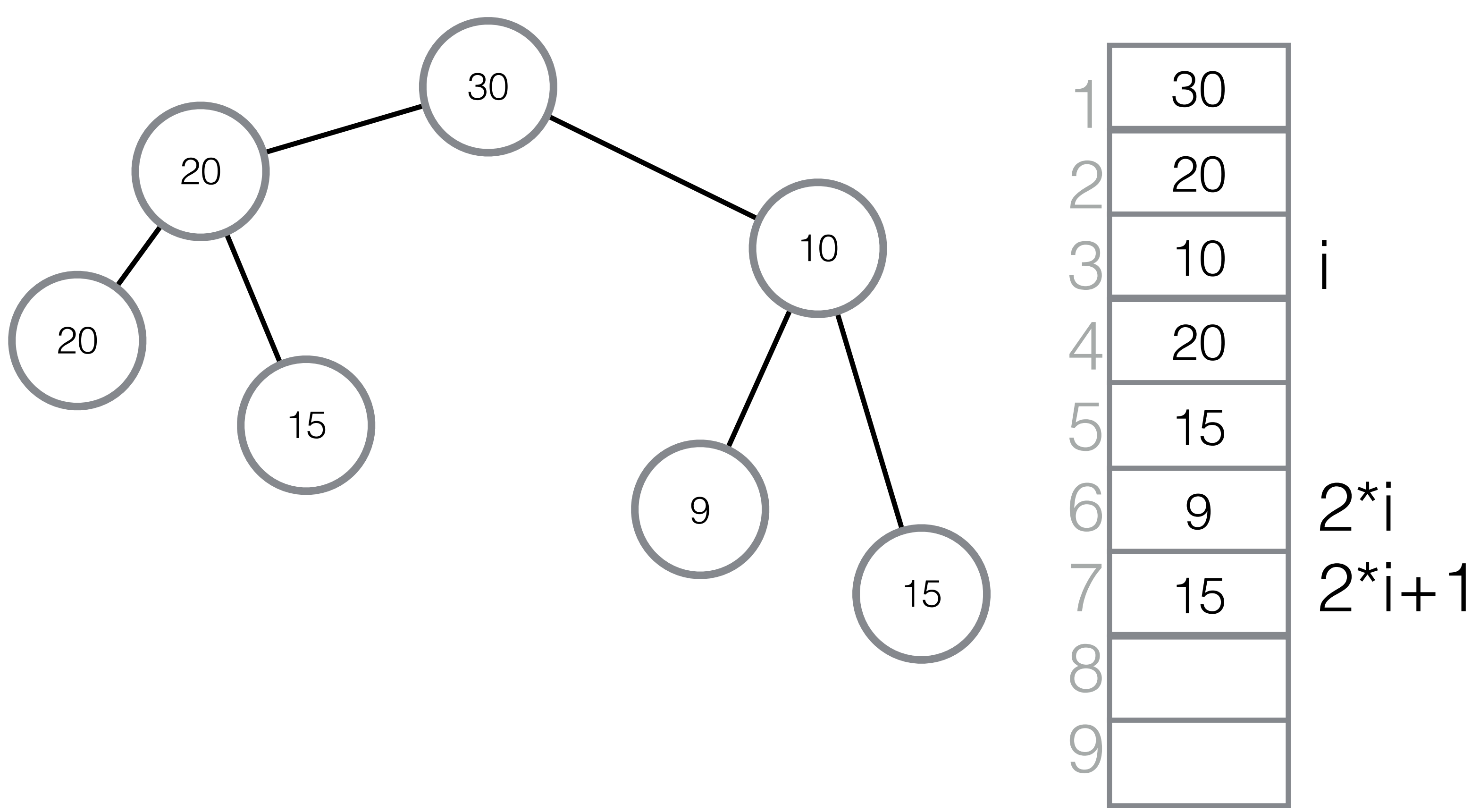
Properties

- The largest element is in the root, always
- The heap folds nicely…

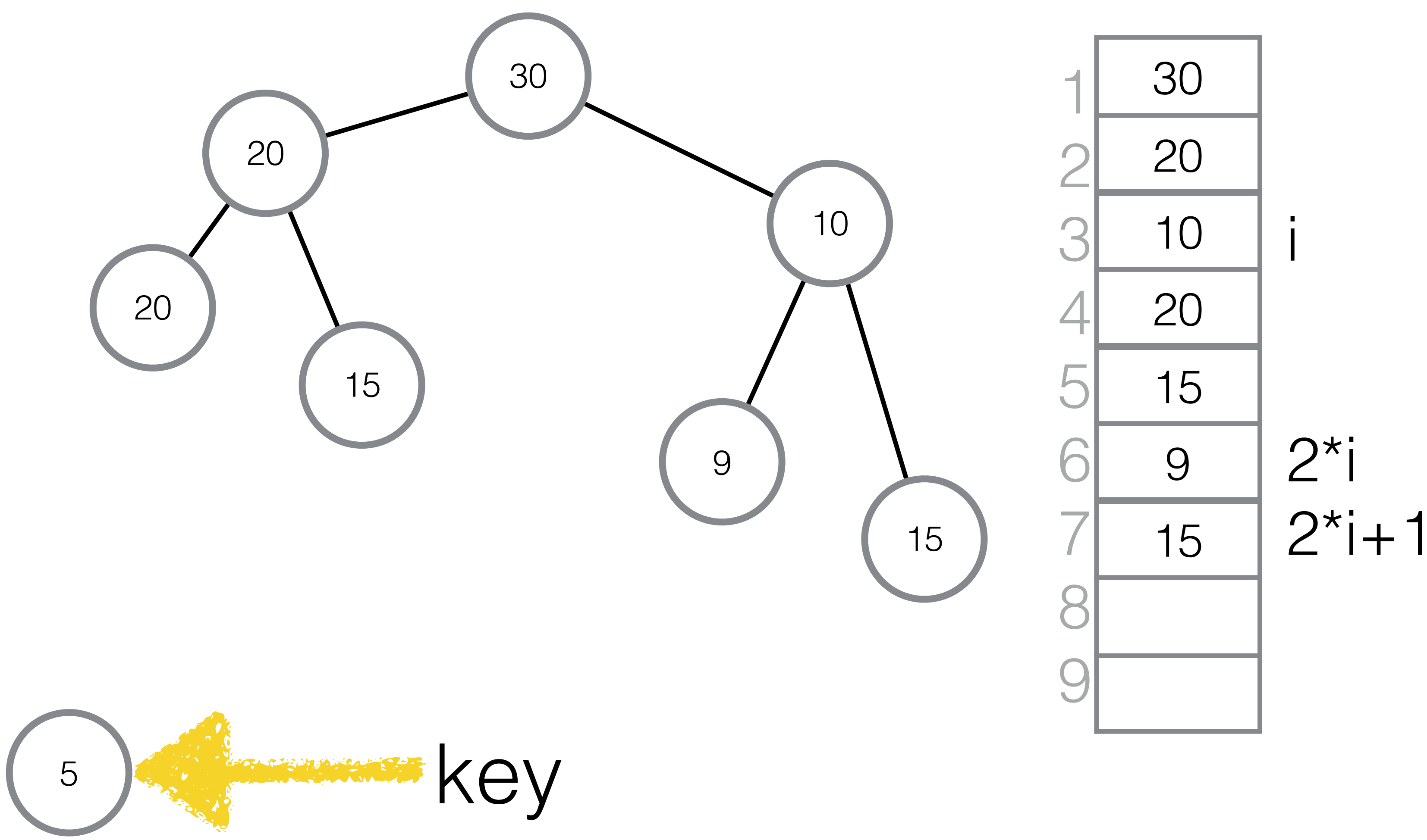


Properties

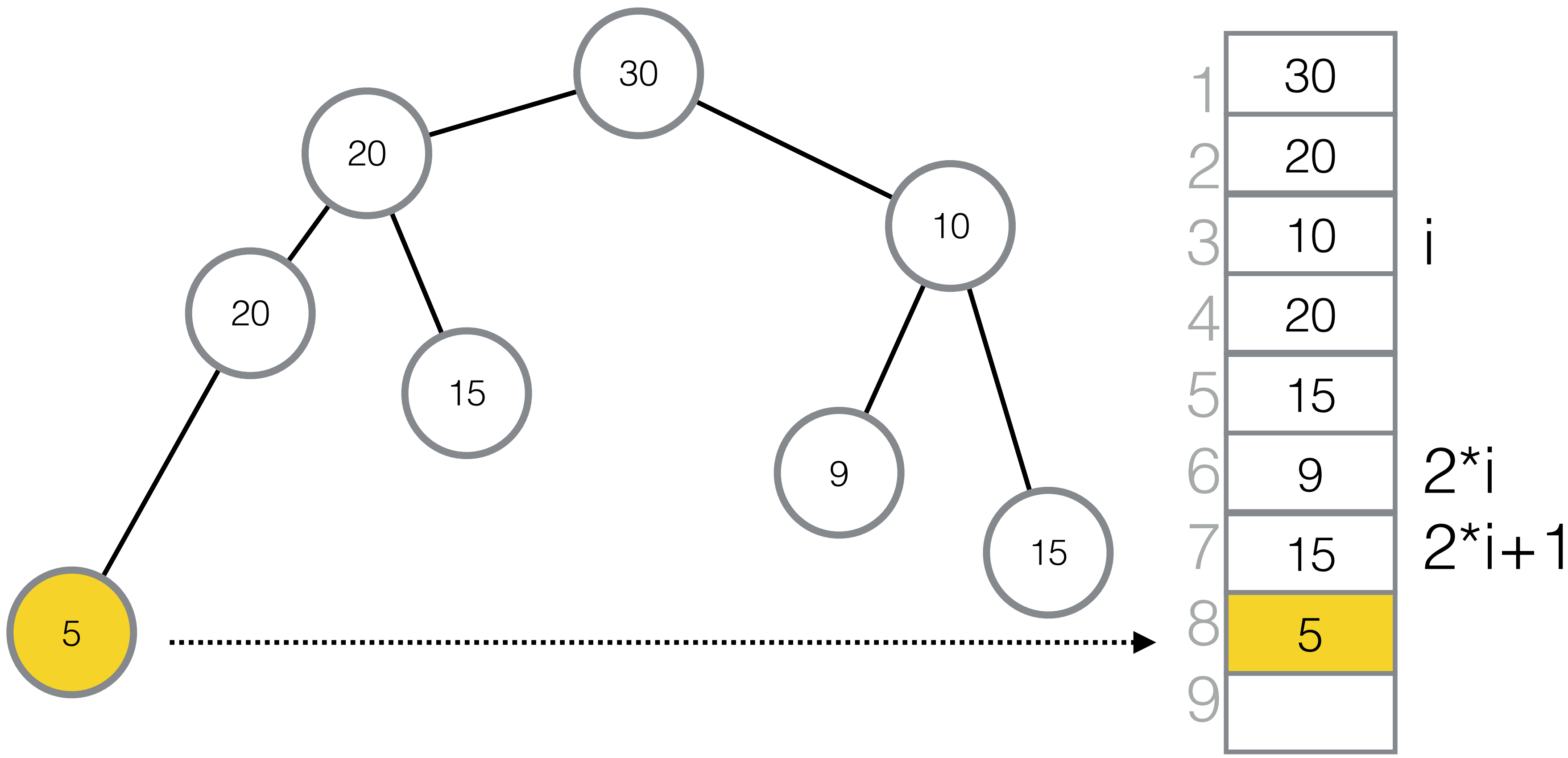
- The largest element is in the root, always
- The heap folds nicely…



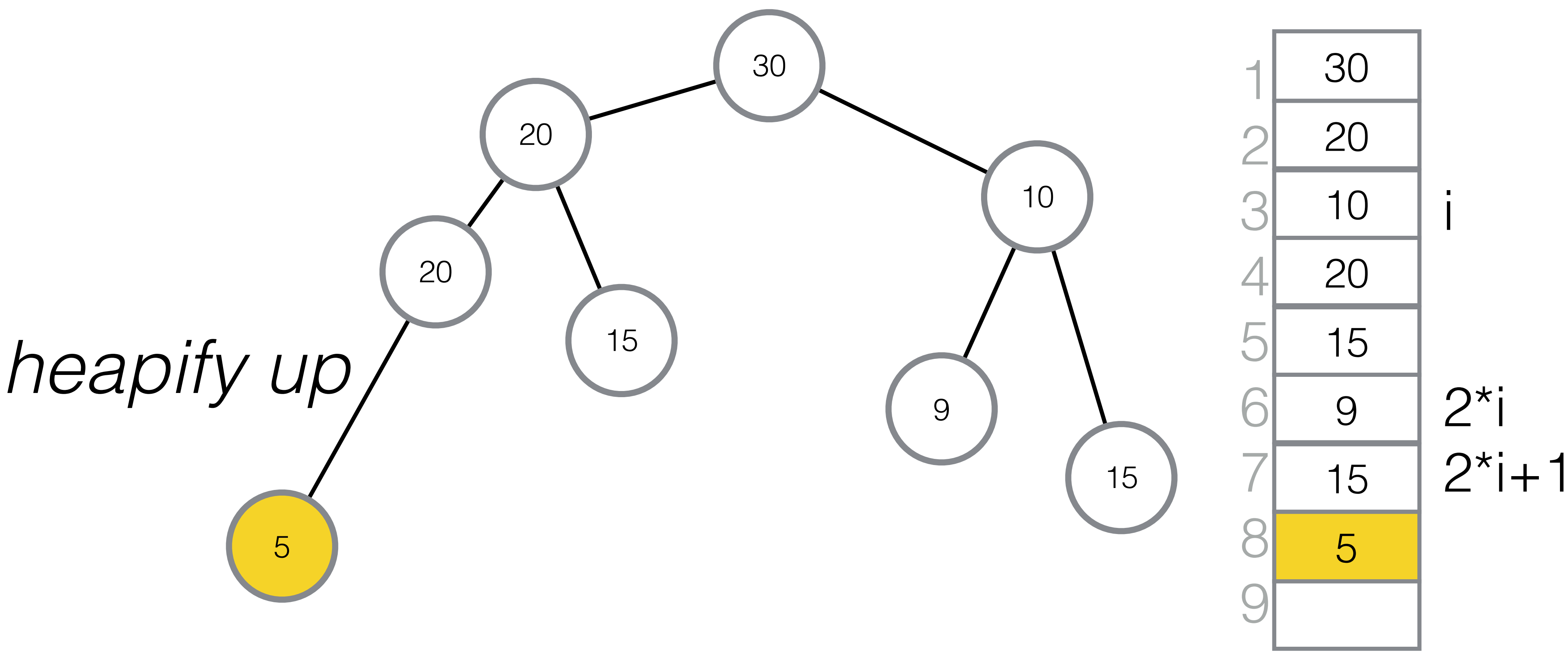
insertion of a new key



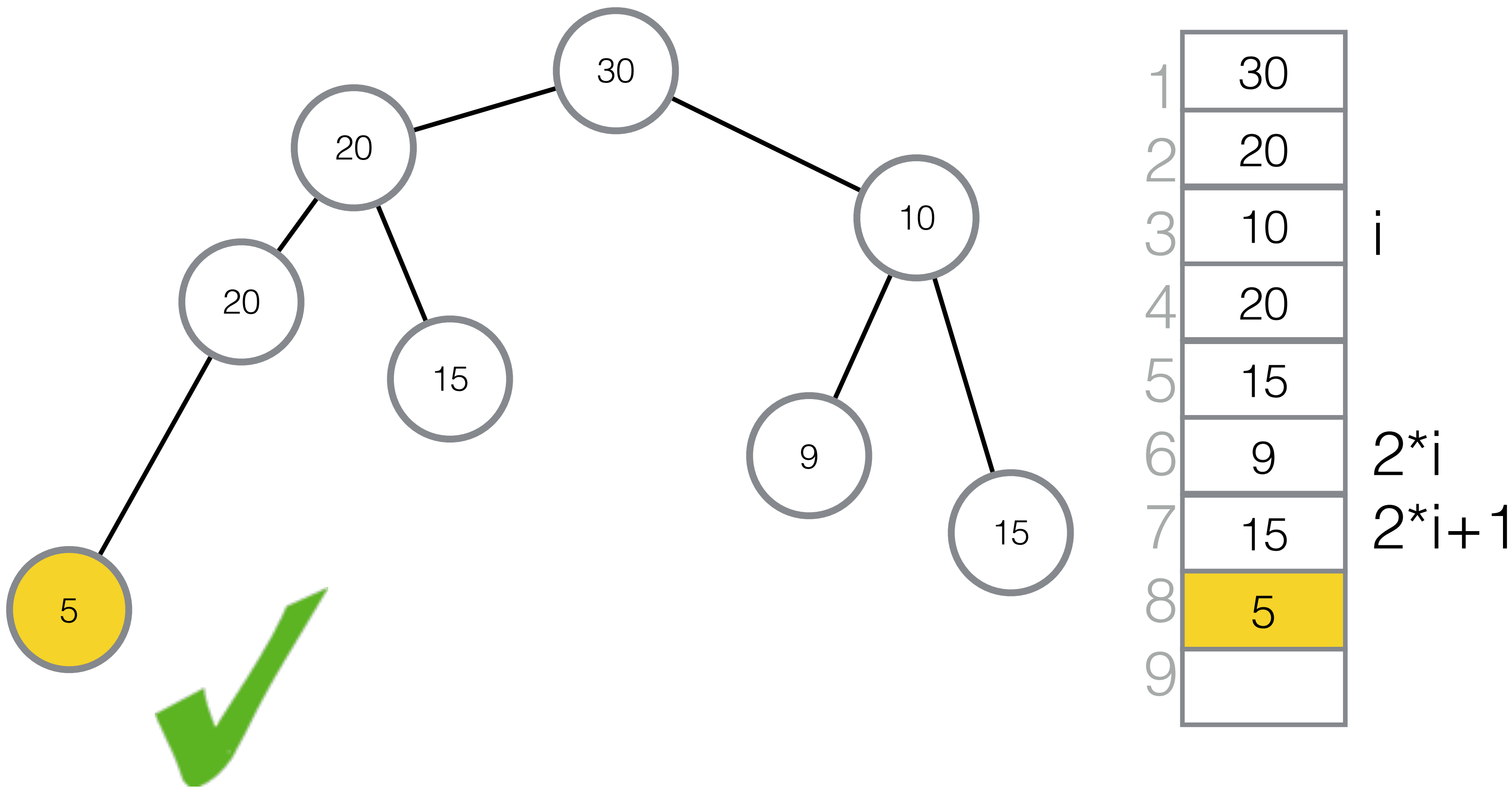
insertion of a new key



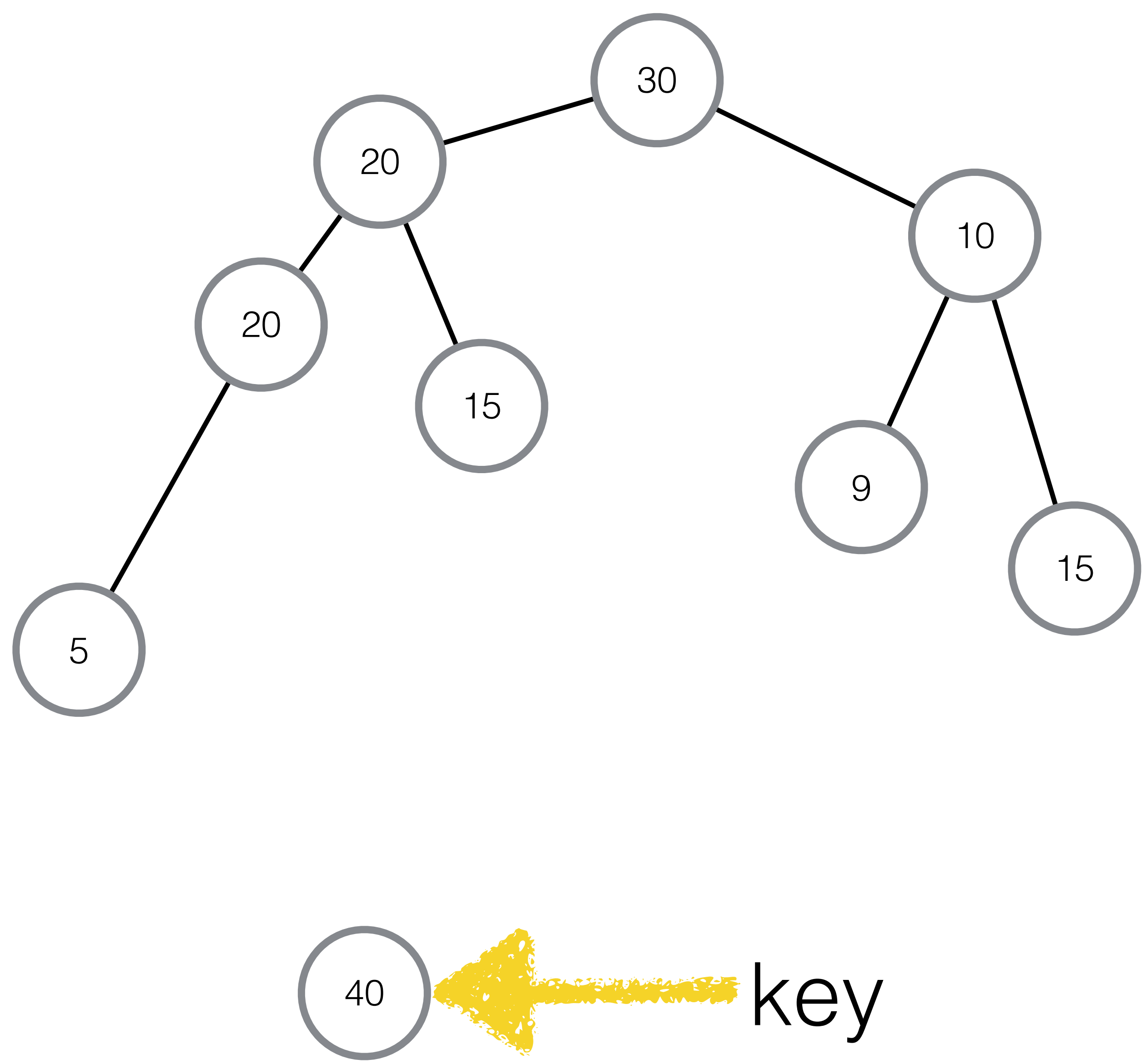
insertion of a new key



insertion of a new key

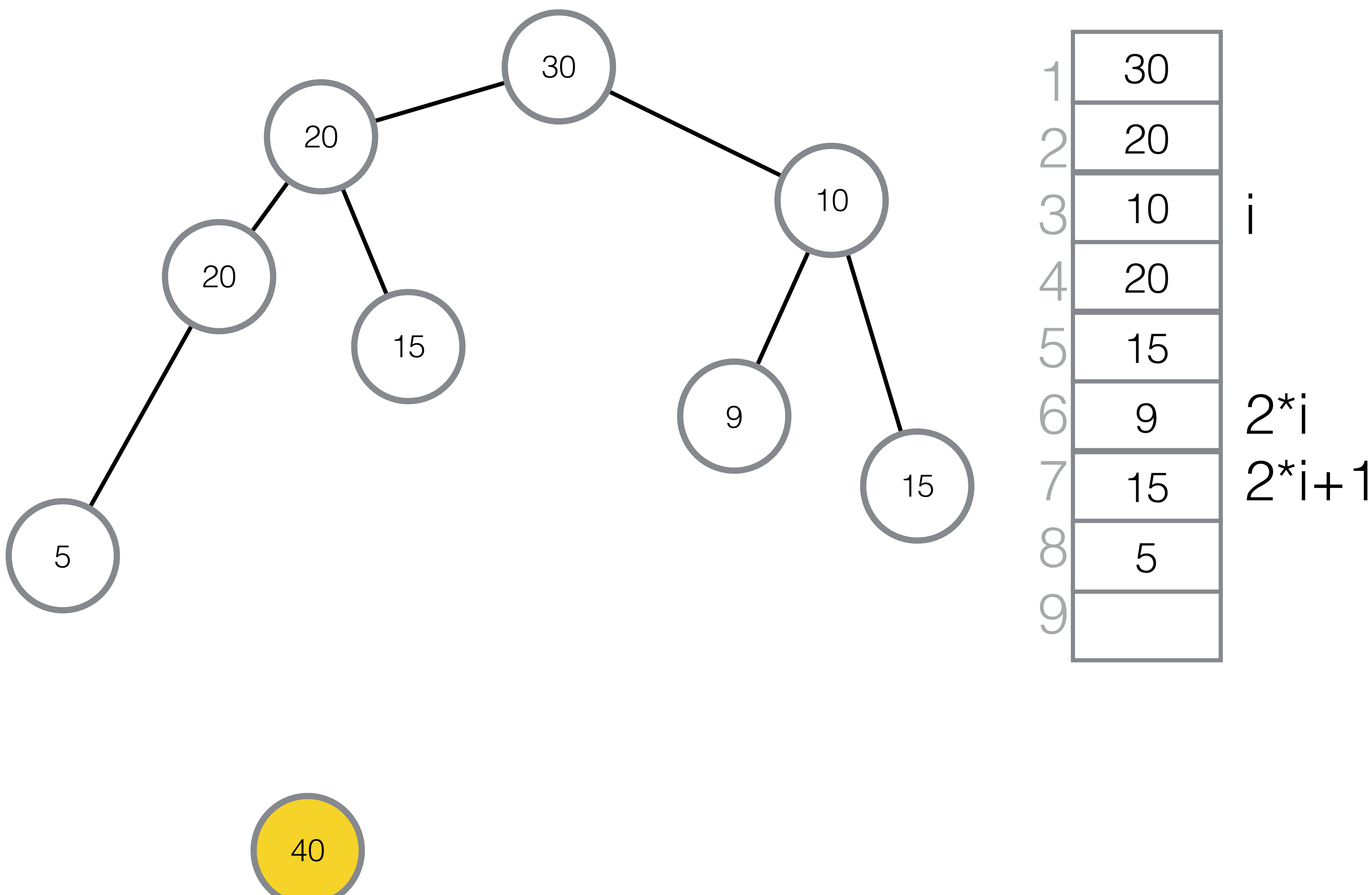


insertion of a new key

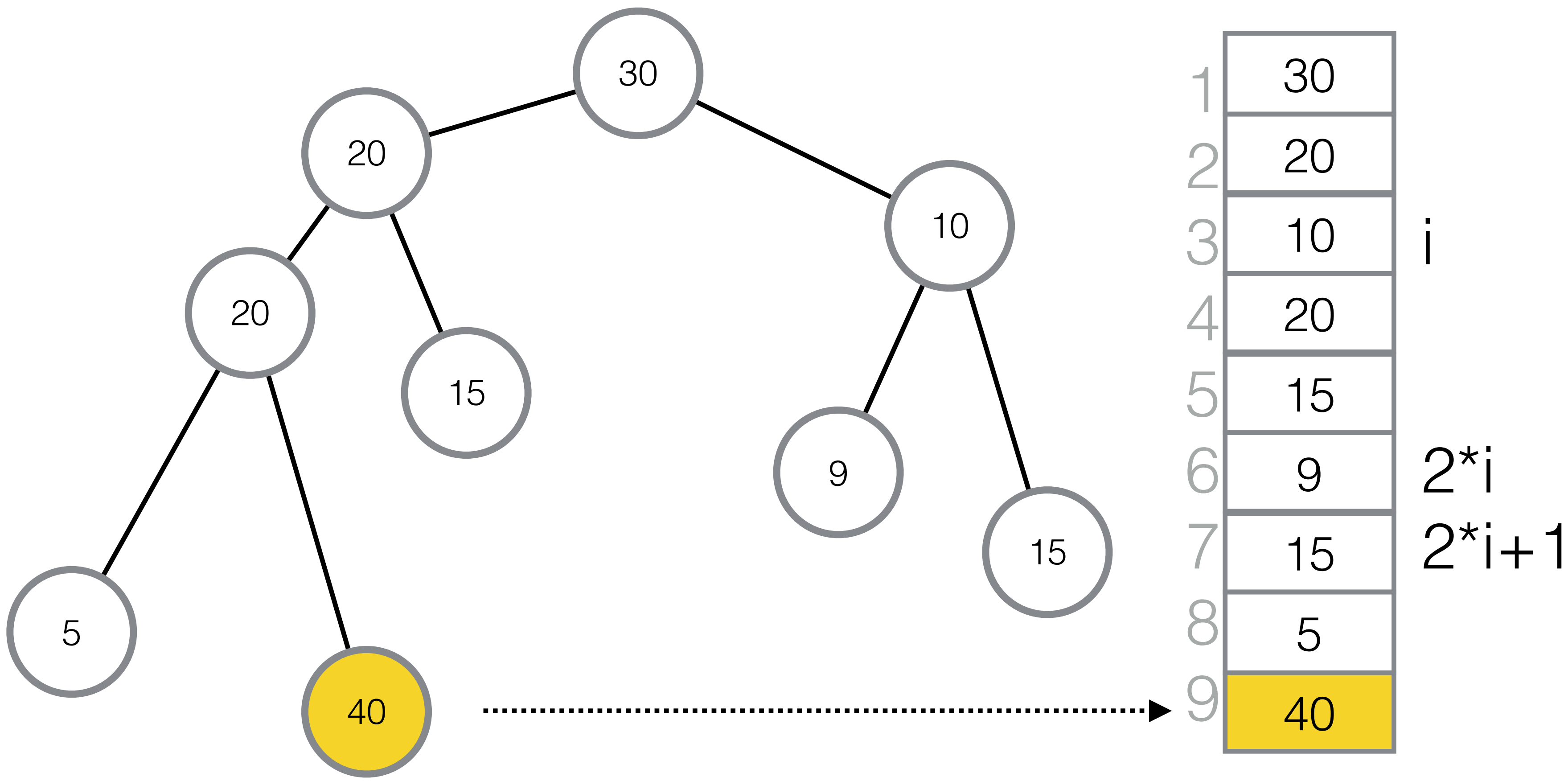


1	30	
2	20	
3	10	i
4	20	
5	15	
6	9	2*i
7	15	2*i+1
8	5	
9		

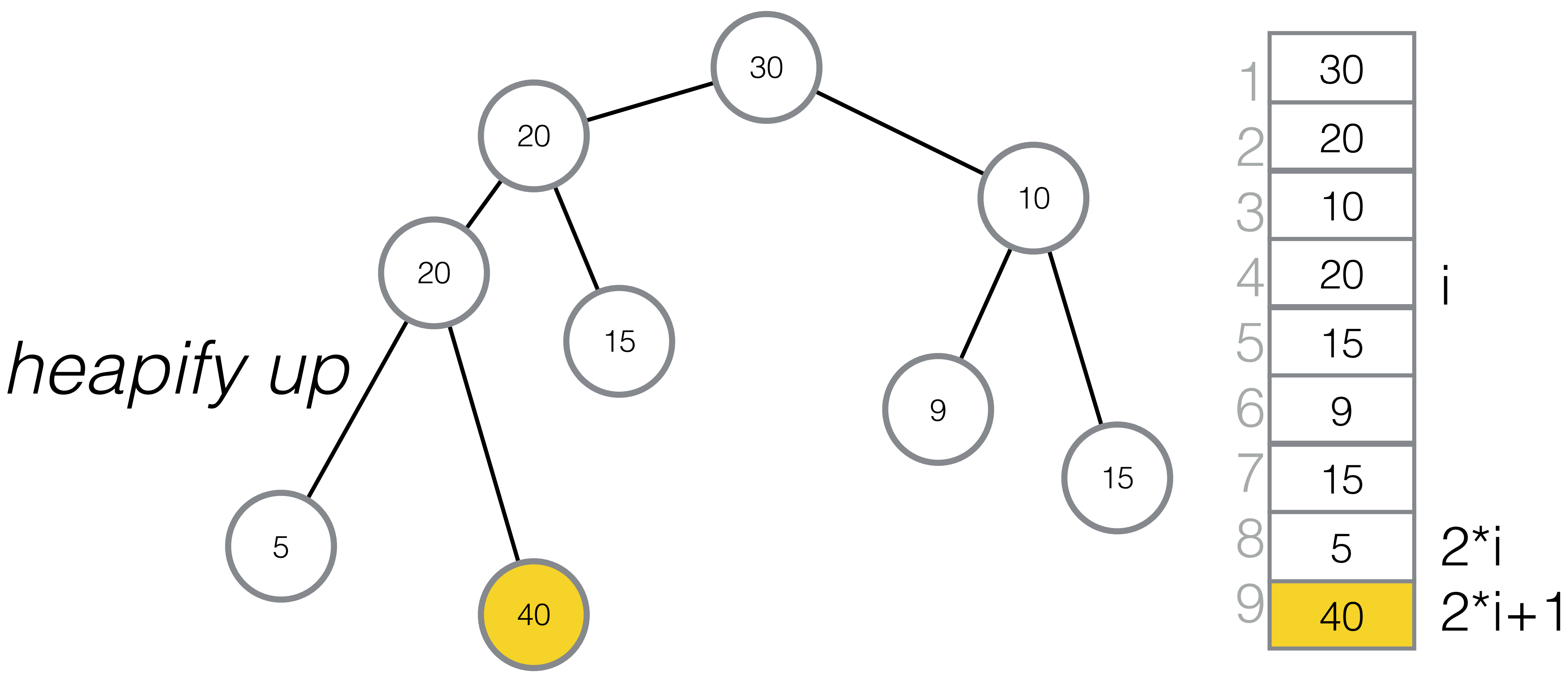
insertion of a new key



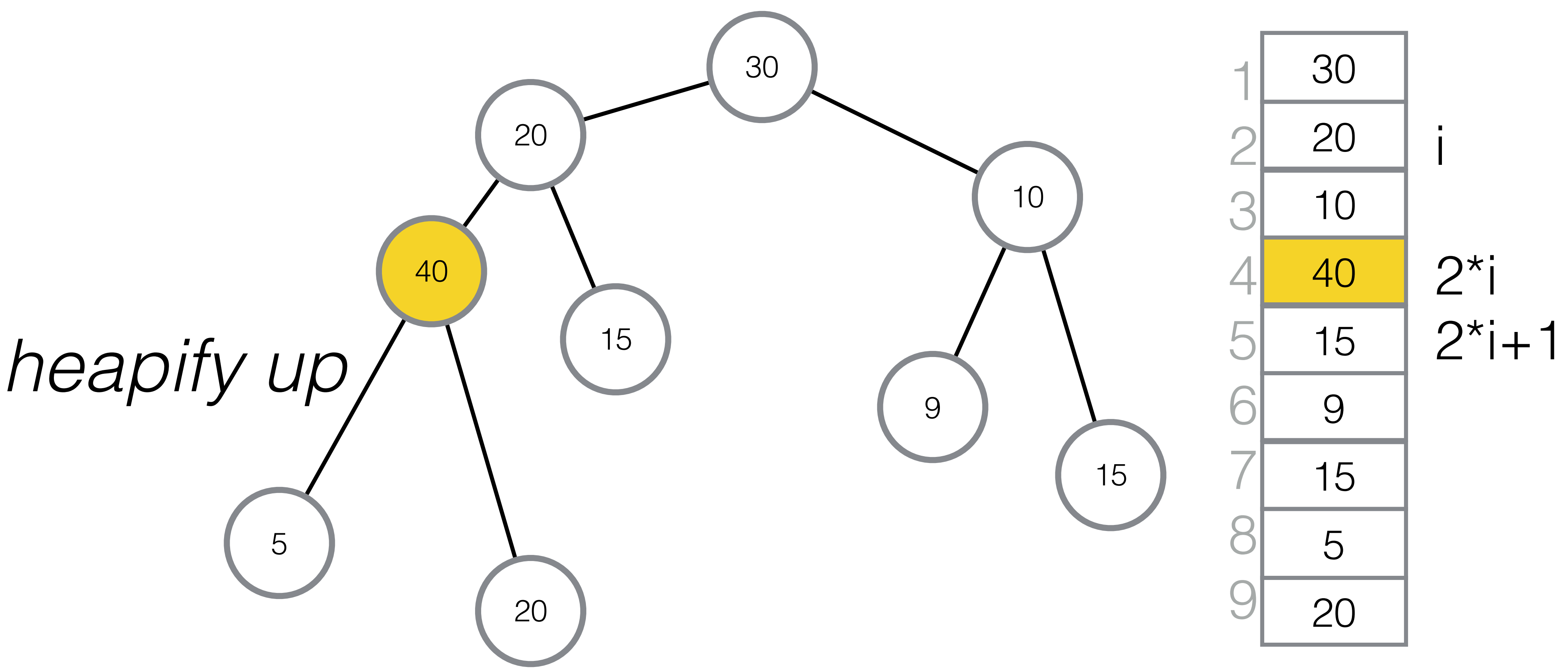
insertion of a new key



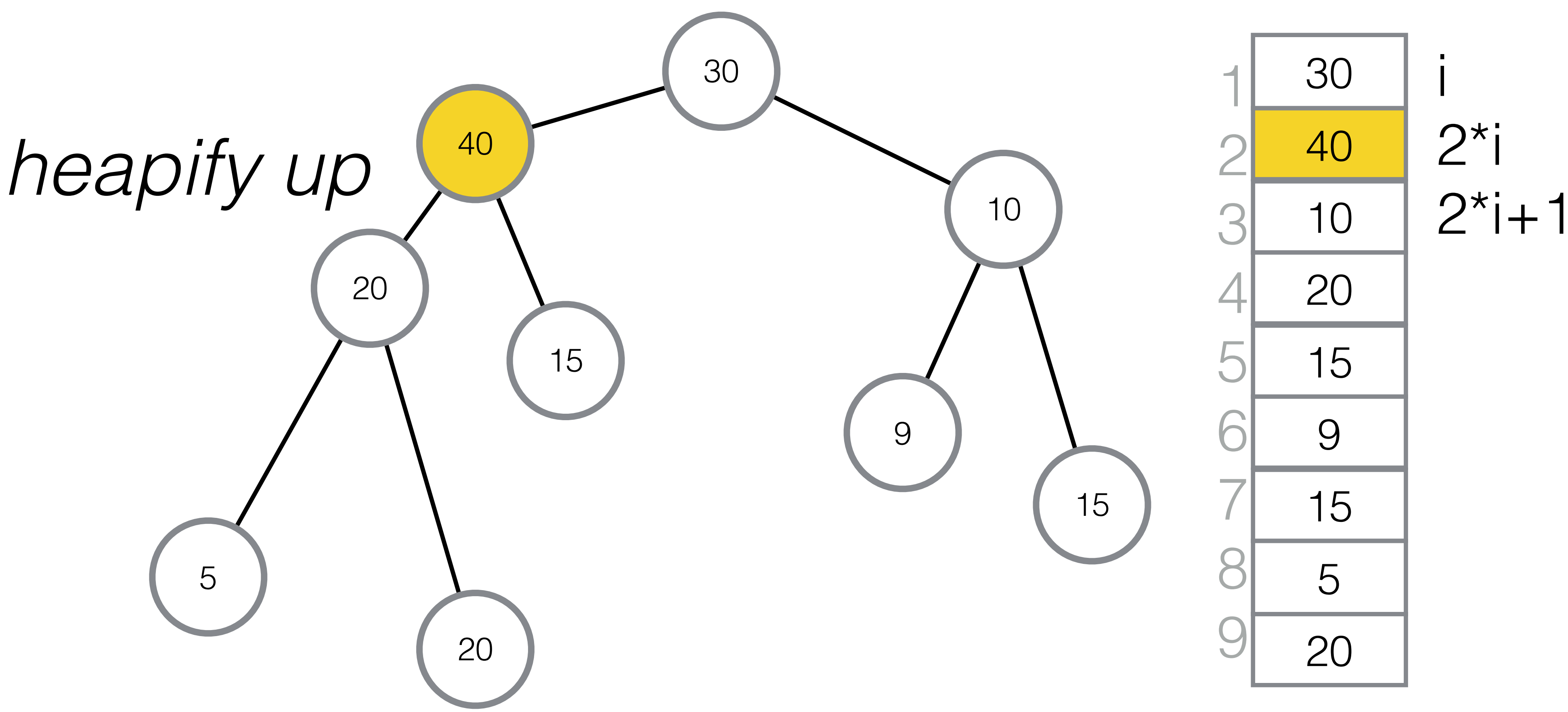
insertion of a new key



insertion of a new key

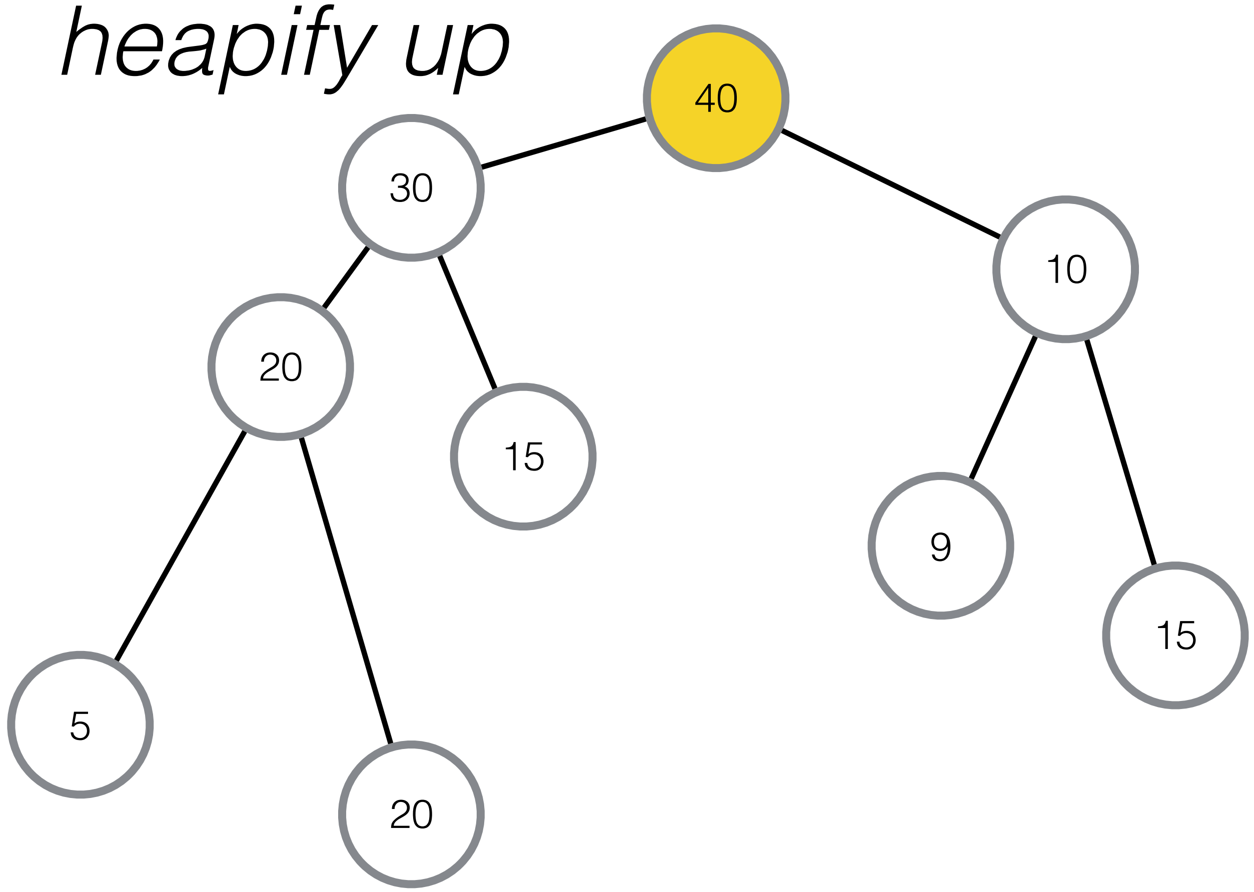


insertion of a new key



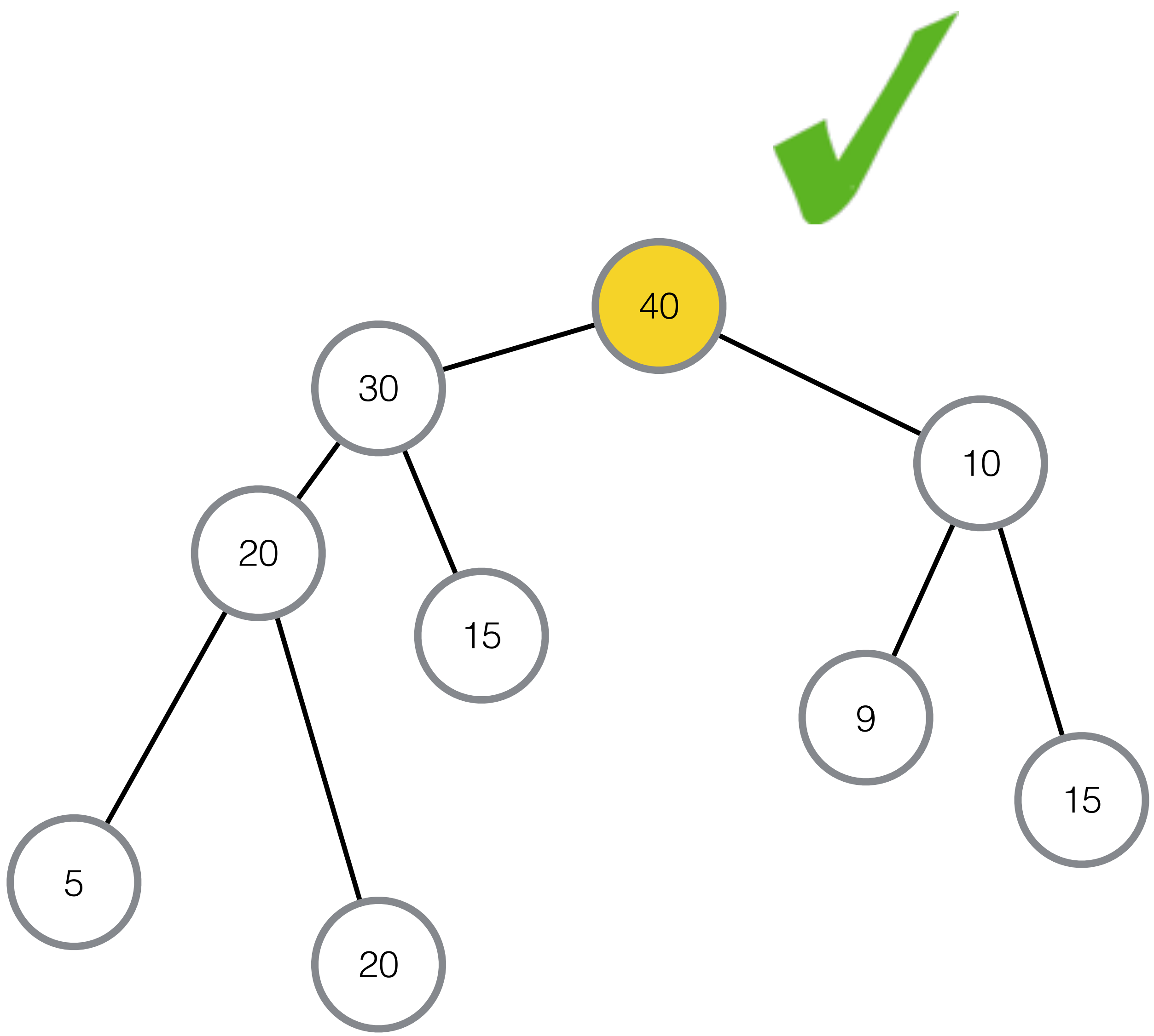
insertion of a new key

heapify up



1	40	i
2	30	2*i
3	10	2*i+1
4	20	
5	15	
6	9	
7	15	
8	5	
9	20	

insertion of a new key

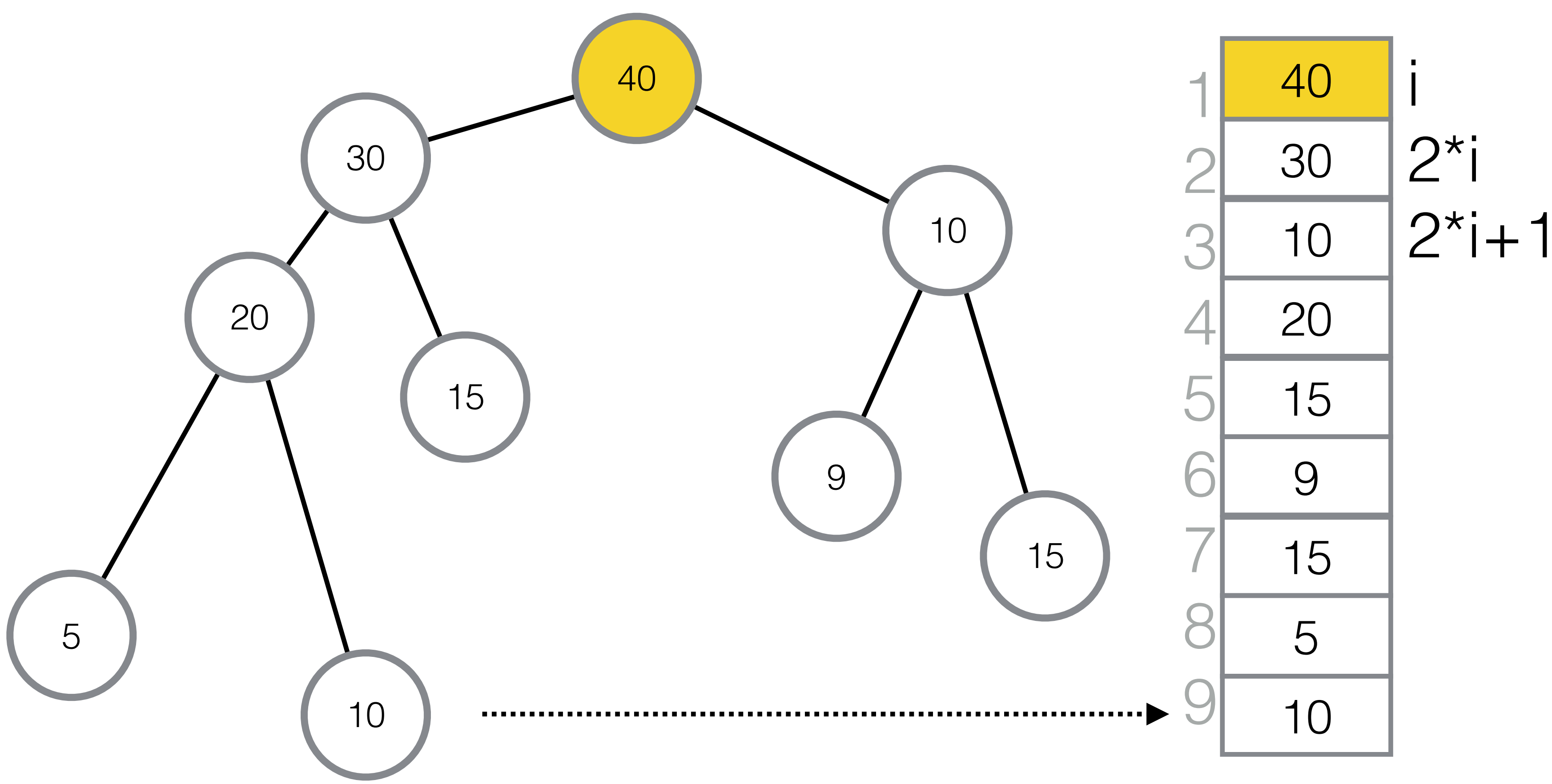


1	40	i
2	30	2*i
3	10	2*i+1
4	20	
5	15	
6	9	
7	15	
8	5	
9	20	

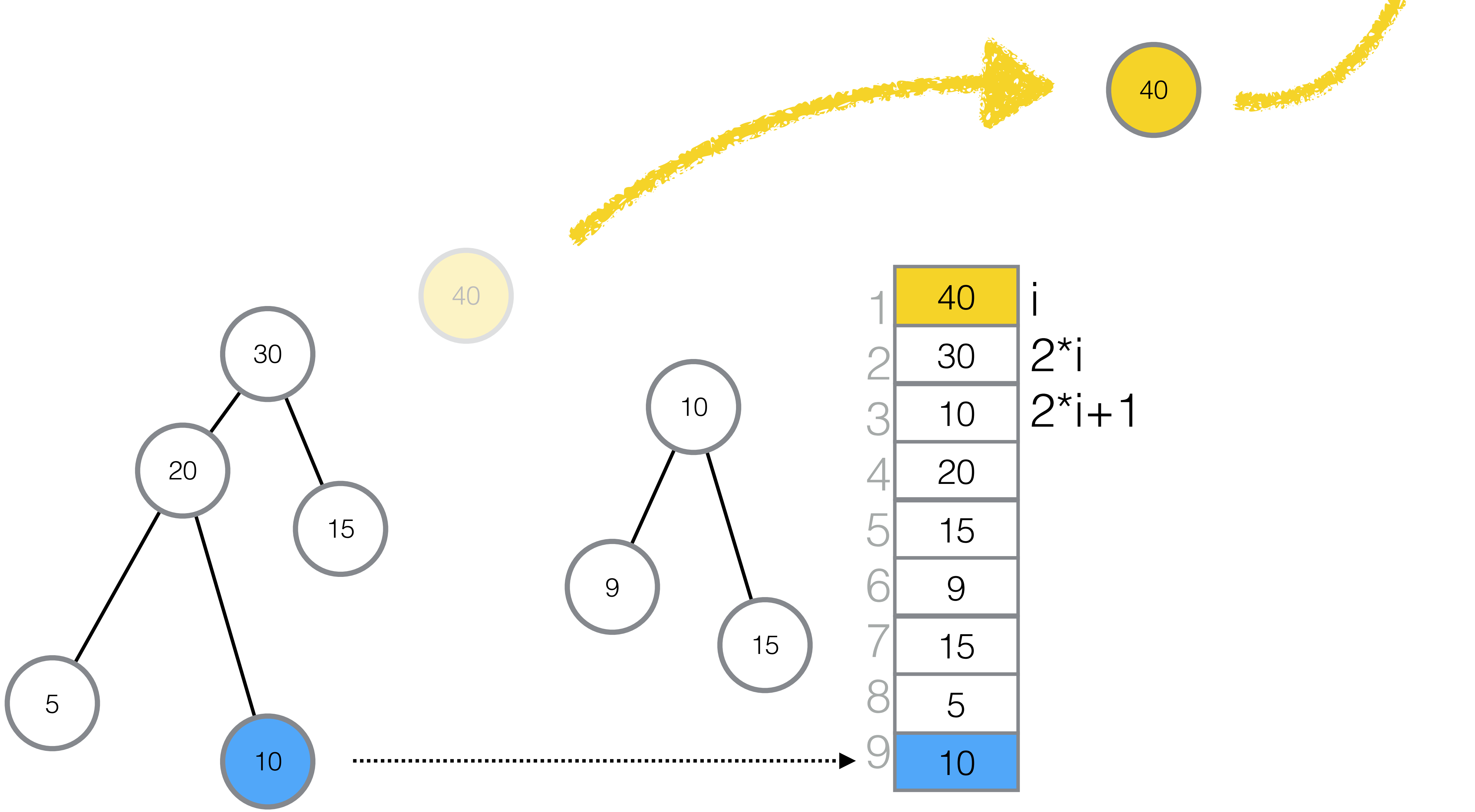
Algorithm

```
heapifyUp( node )
    begin
    while ( node has a parent ) and ( node.key > parent.key )
        begin
        swap( node and its parent )
        // node is now in its parent's original position
        end
     end
```

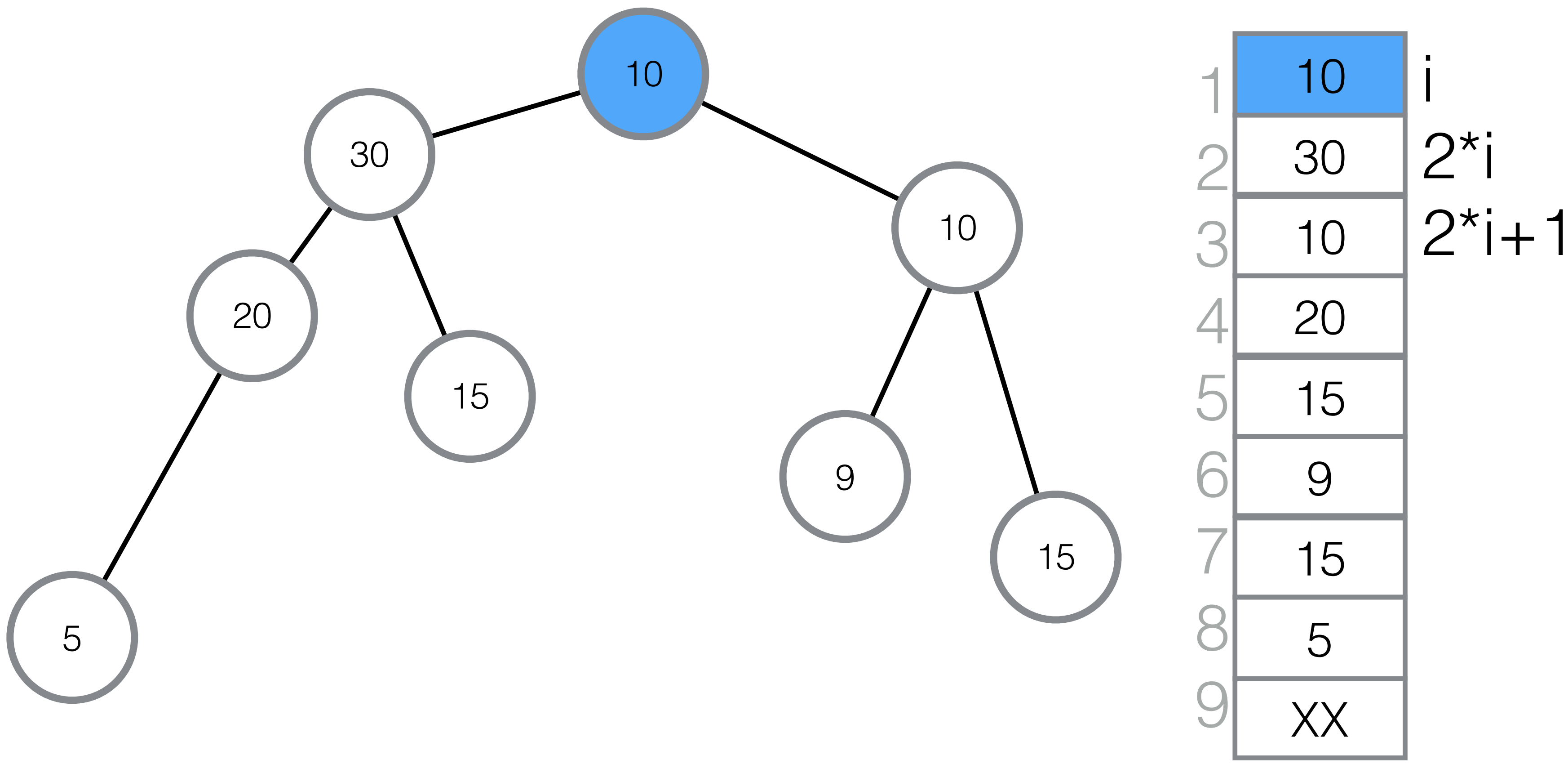
Deleting the Root



Deleting the Root

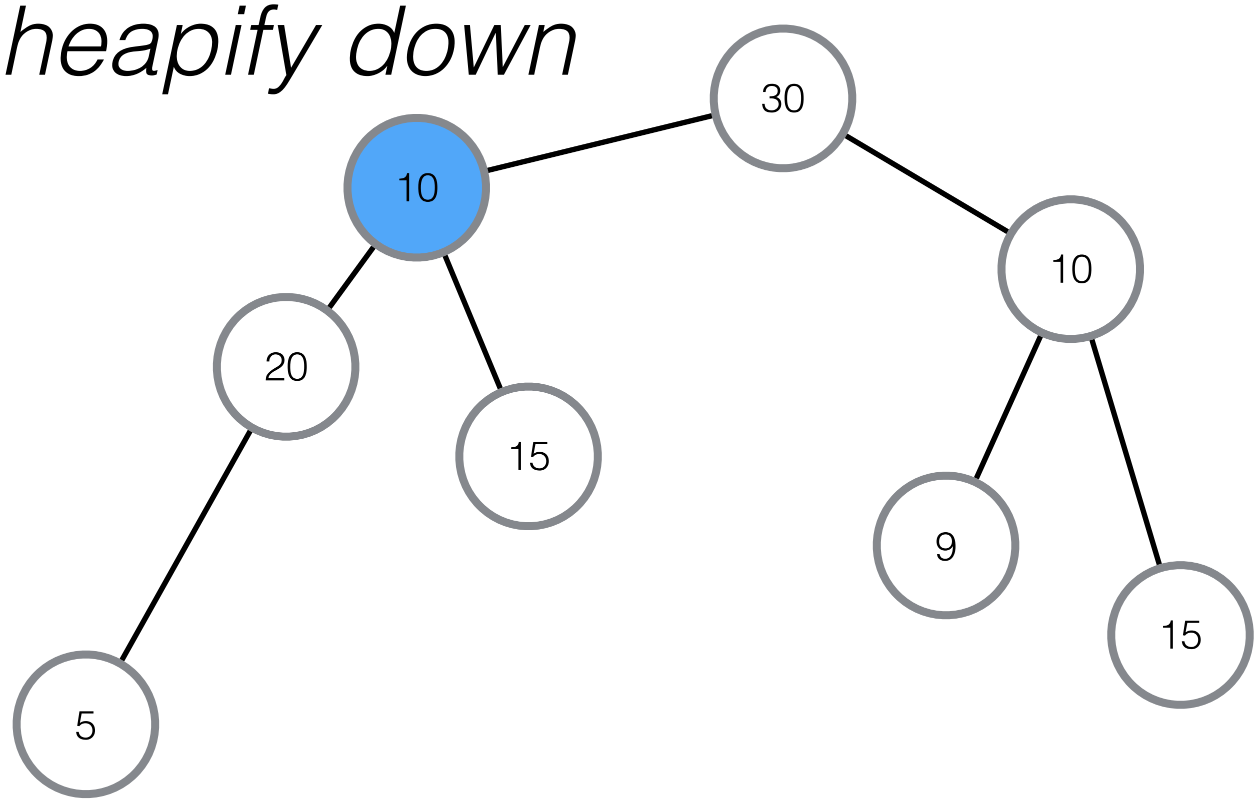


Deleting the Root



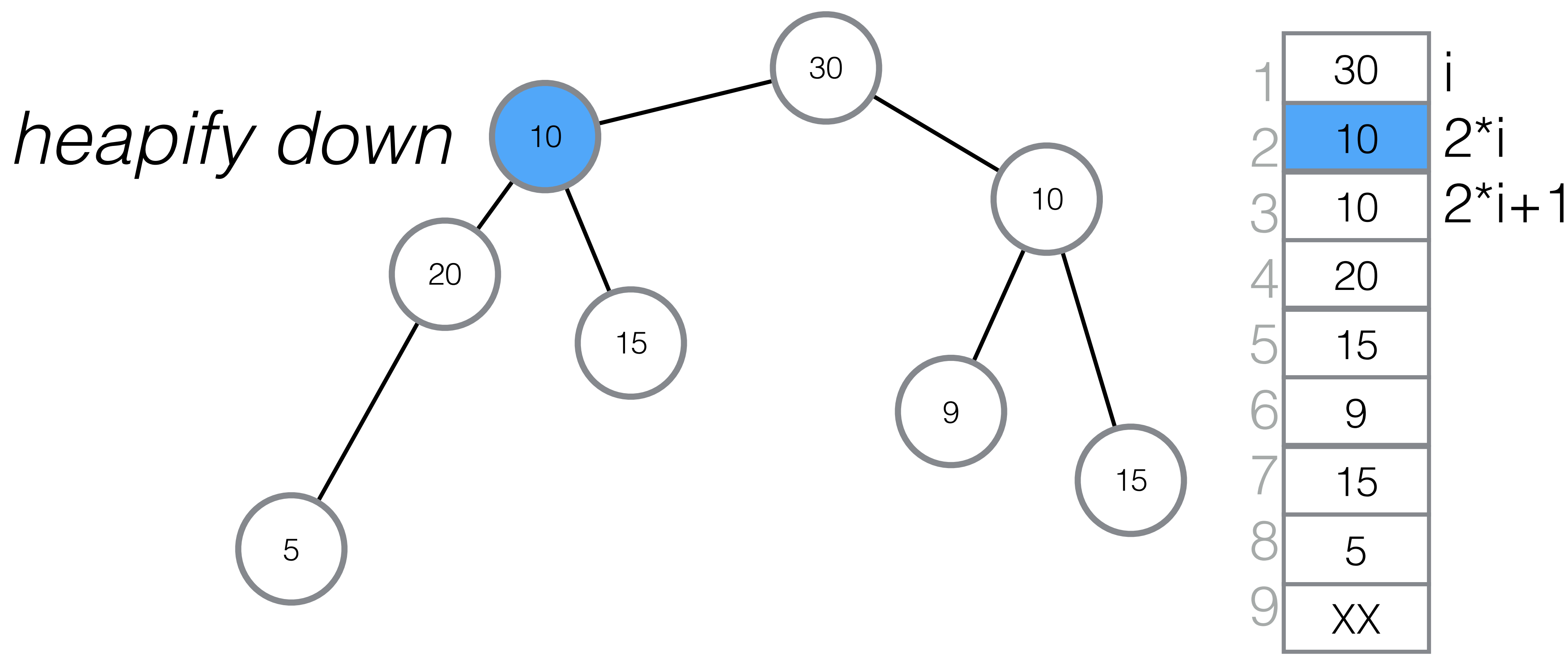
Deleting the Root

heapify down



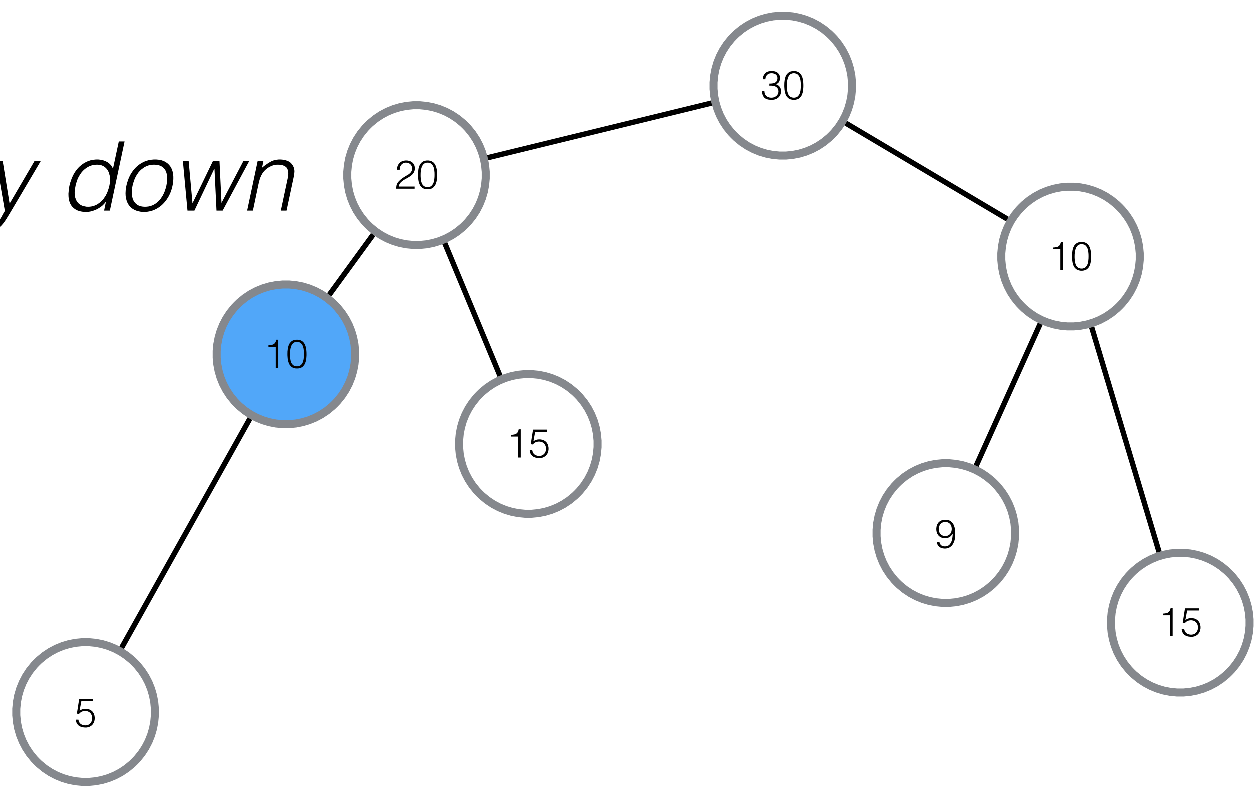
1	30	i
2	10	2*i
3	10	2*i+1
4	20	
5	15	
6	9	
7	15	
8	5	
9	XX	

Deleting the Root



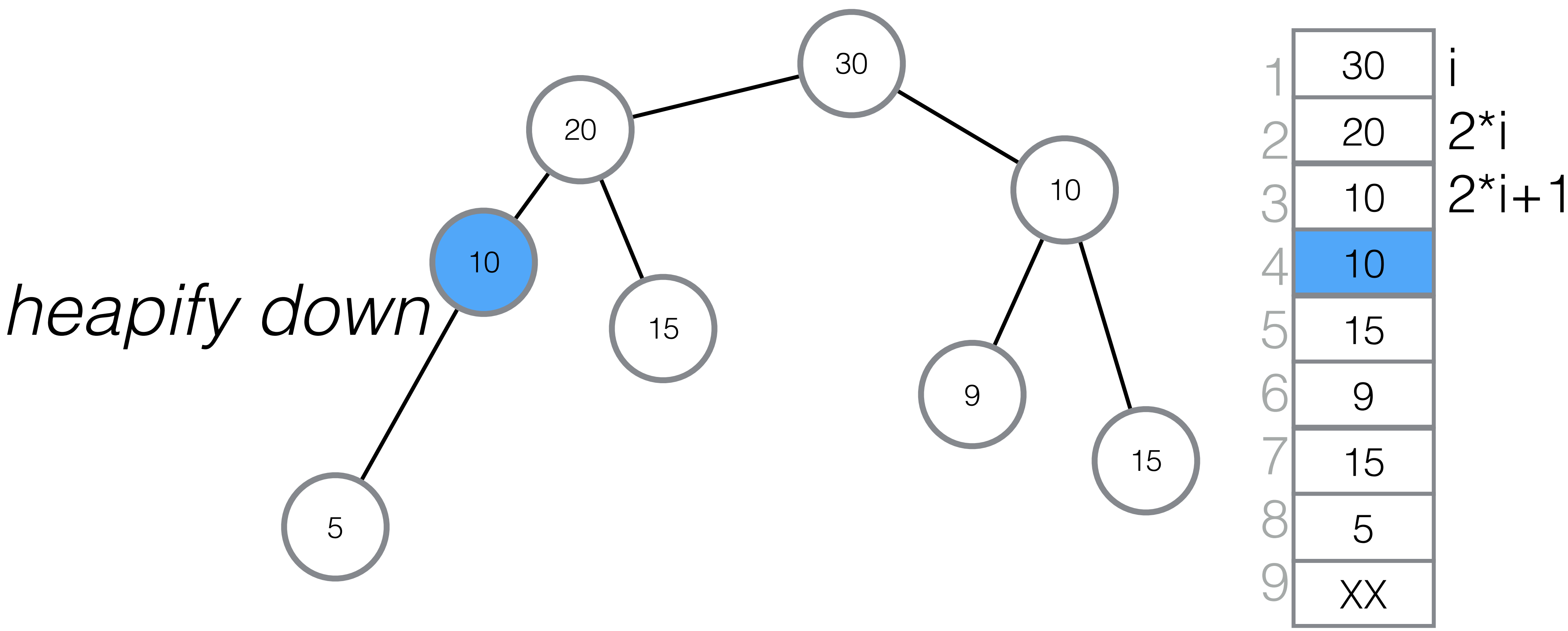
Deleting the Root

heapify down

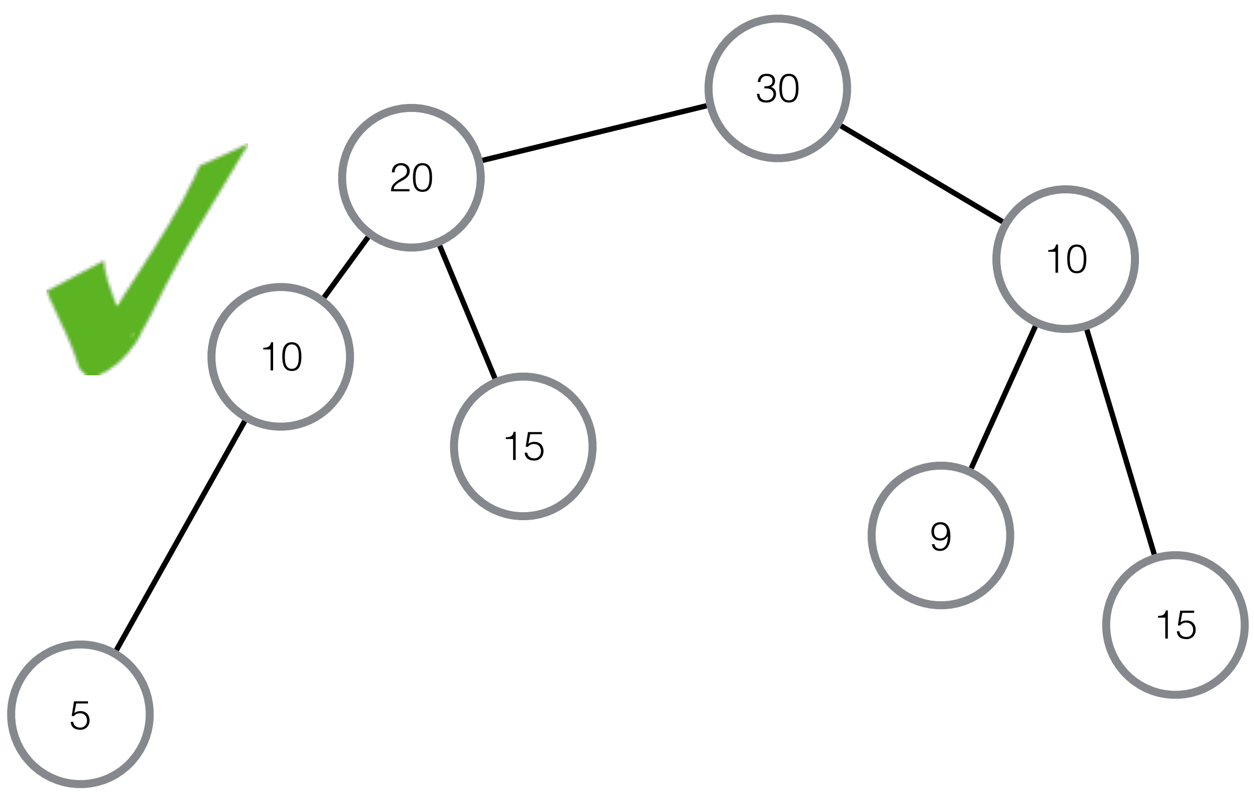


1	30	i
2	20	2*i
3	10	2*i+1
4	10	
5	15	
6	9	
7	15	
8	5	
9	XX	

Deleting the Root

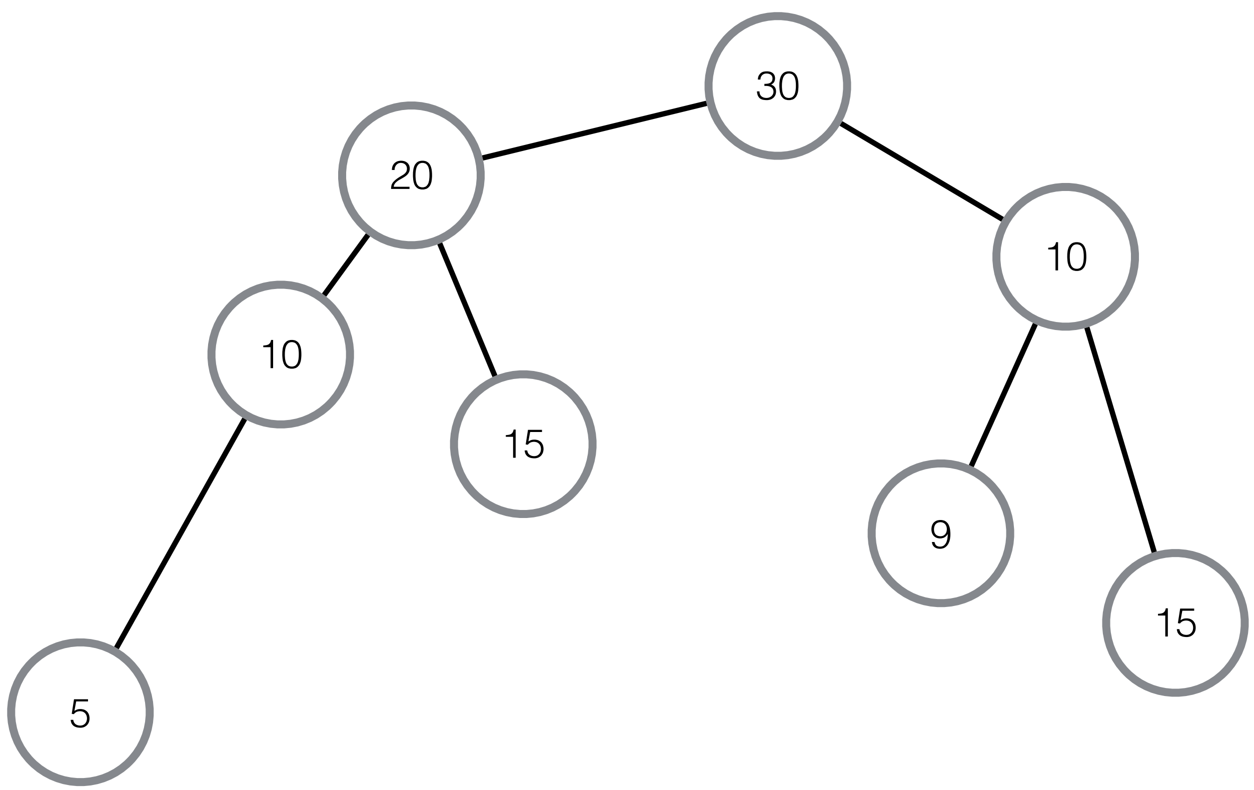


Deleting the Root



1	30
2	20
3	10
4	10
5	15
6	9
7	15
8	5
9	XX

Deleting the Root



1	30
2	20
3	10
4	10
5	15
6	9
7	15
8	5
9	XX

Algorithm

```
heapifyDown( node )
    begin
    if node is a leaf then
        return
    maxNode = node's child with highest key
    if maxNode.key > node.key then
        begin
        swap( node and maxNode )
        // now node is in place of maxNode's original position
        heapifyDown( node )   // recurse down
        end
    end
```

Operation	Worst Time Complexity	Average Time Complexity	Best Time Complexity
Insert key	O(log N)	O(log N)	O(log N)
delete root	O(log N)	O(log N)	O(log N)