Competitive Algorithm Design and Practice
Binary Search Tree & Binary Heap
2014/02/26

Yi Long, Lu (mike199250)

mike199250@gmail.com

http://myweb.ncku.edu.tw/~f74991073/2014_02_26_BST_BH.zip

Department of Computer Science and Information Engineering
National Cheng Kung University
Tainan, Taiwan

Outline

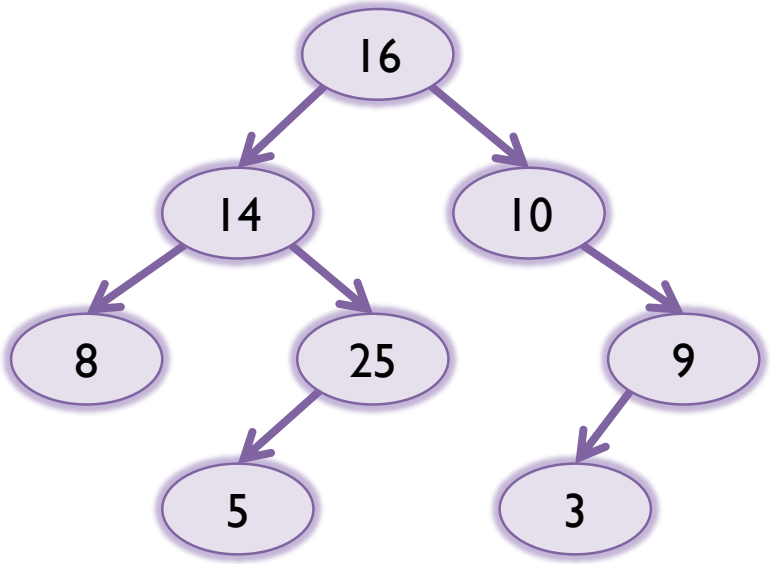
- Binary tree
- Binary search tree
- Practice - POJ 3481
- Heap
- Binary heap
- Practice - Uva 10954
- Problems

Binary Tree

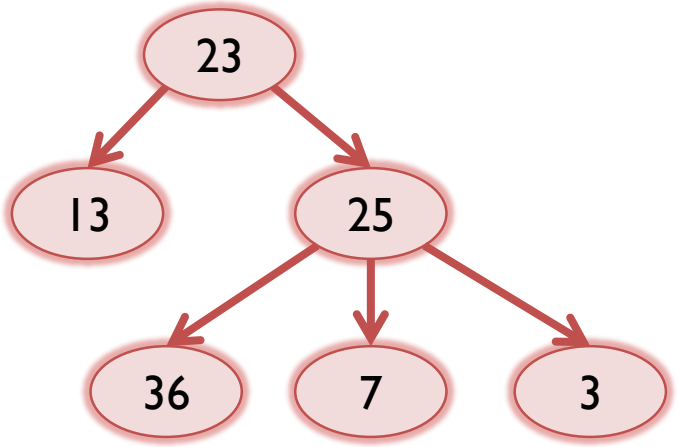
Made By mike199250

Binary Tree

- Each node has at most two children(left child and right child).



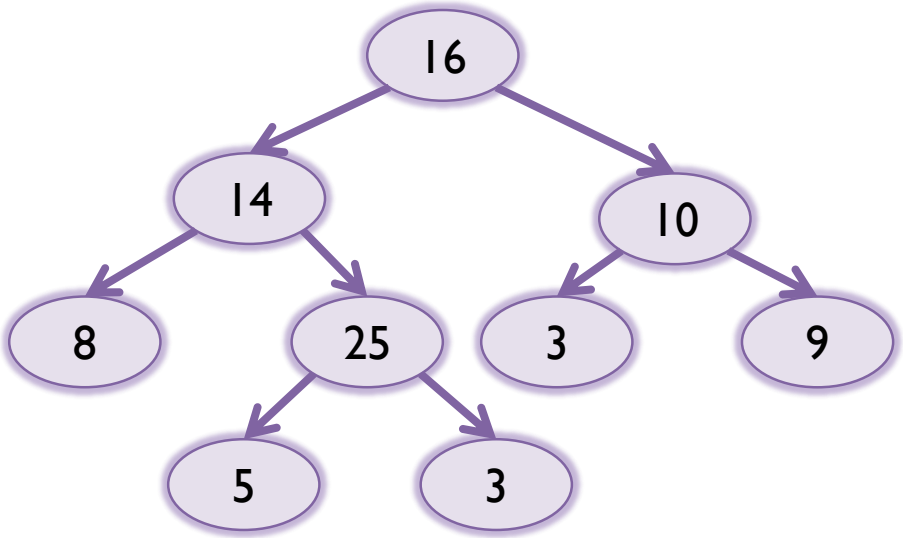
Binary tree

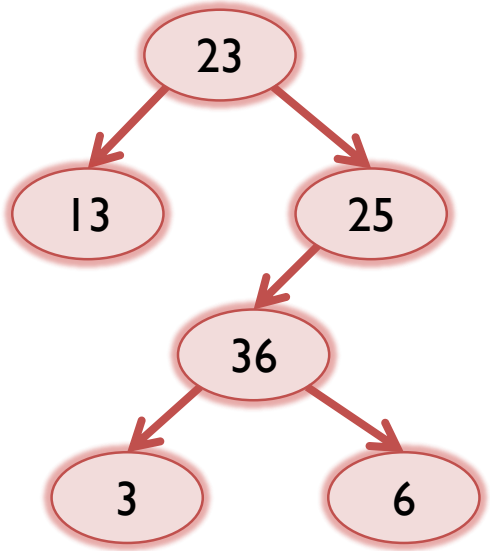


Not Binary tree

Full Binary Tree

- Every node other than the leaves has two children.

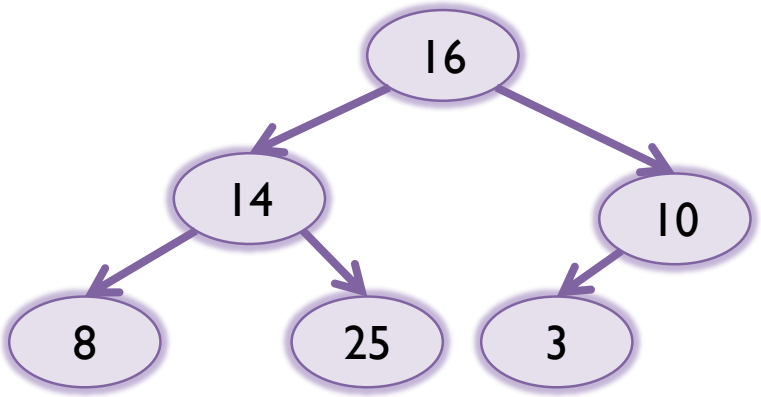




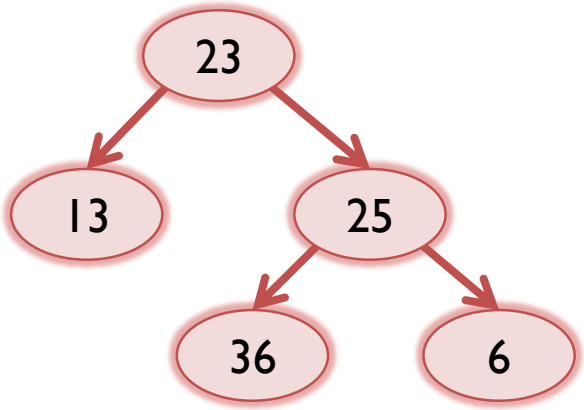
Full binary tree Not full binary tree

Complete Binary Tree

- Every level, except the last, is completely filled, and all nodes are as far left as possible.

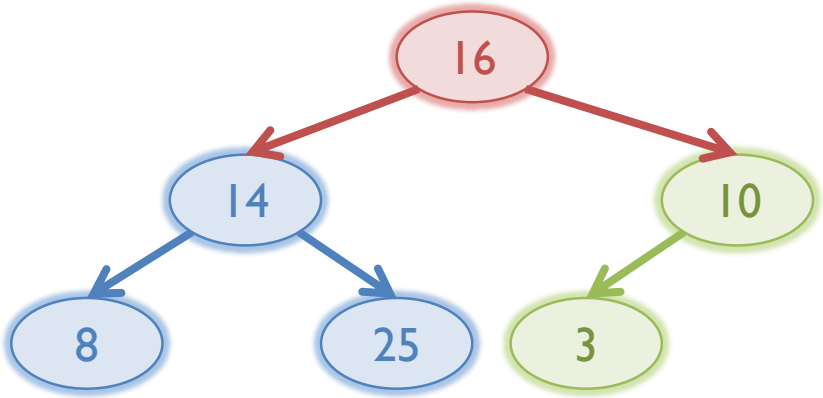


Complete binary tree



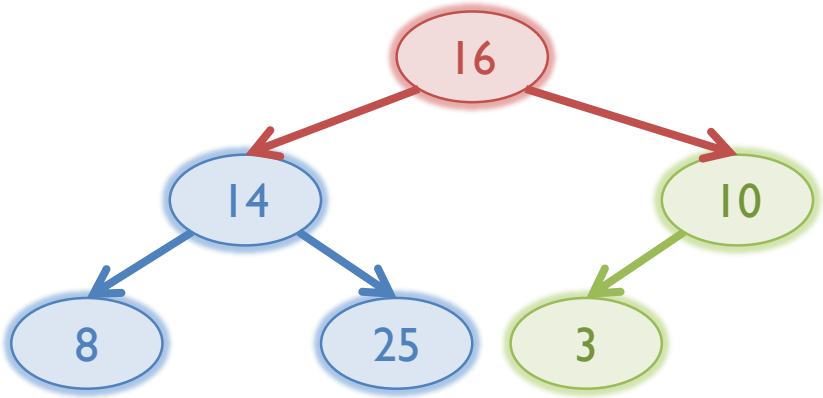
Not complete binary tree

Traversal



- Pre-order: 16 14 8 25 10 3
- In-order: 8 14 25 16 3 10
- Post-order: 8 25 14 3 10 16

Traversal




```
In-order-Tree-Walk(x)
1.  if x is not NULL
2.       then In-order-Tree-Walk(left[x])
3.            print key[x]
4.            In-order-Tree-Walk(right[x])
```

Binary Search Tree

Made By mike199250

Binary Search Tree

- The left sub-tree of a node contains only nodes with keys less than the node`s key.

Binary Search Tree

- The left sub-tree of a node contains only nodes with keys less than the node`s key.
- The right sub-tree of a node contains only nodes with keys greater than the node`s key.

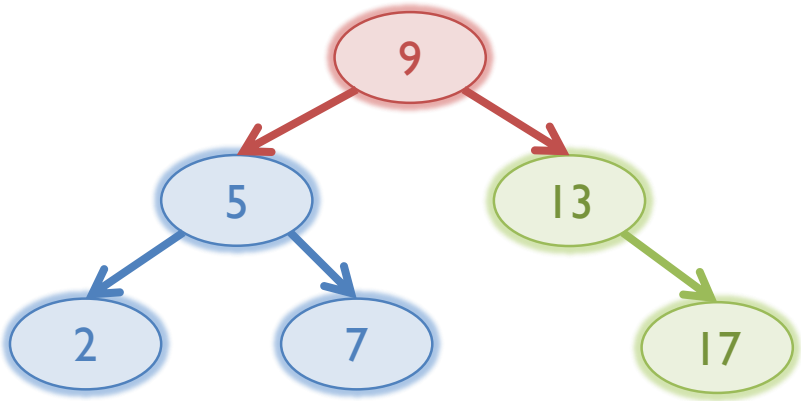
Binary Search Tree

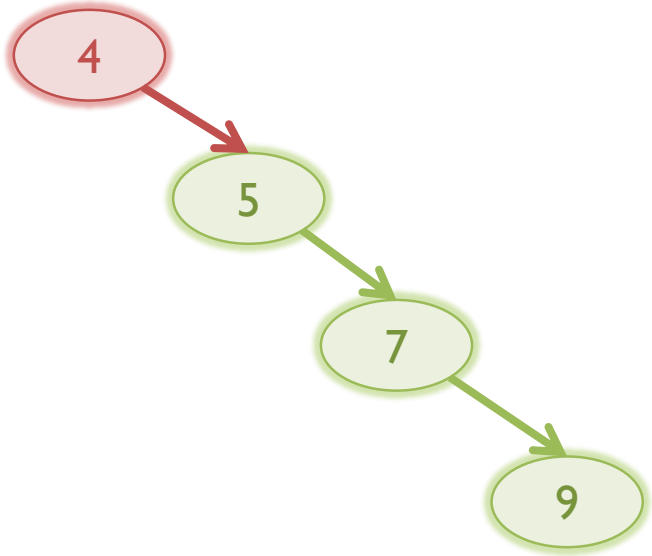
- The left sub-tree of a node contains only nodes with keys less than the node`s key.
- The right sub-tree of a node contains only nodes with keys greater than the node`s key.
- The left and right sub-tree are also binary search tree.

Binary Search Tree

- The left sub-tree of a node contains only nodes with keys less than the node`s key.
- The right sub-tree of a node contains only nodes with keys greater than the node`s key.
- The left and right sub-tree are also binary search tree.
- There must be no duplicate nodes.

Binary Search Tree





- Left key < node`s key
- Right key > node`s key

Binary Search Tree

- Operations: **Searching**, **Insertion**, **Deletion** can be performed in O(*h*) time, where *h* is the height of the tree.

Binary Search Tree

- Operations: **Searching**, **Insertion**, **Deletion** can be performed in O(*h*) time, where *h* is the height of the tree.

- Worst case: *h*=O(N)
- Balanced BST: *h*=O(log N)

Binary Search Tree

- Operations: **Searching**, **Insertion**, **Deletion** can be performed in O(*h*) time, where *h* is the height of the tree.

- Worst case: *h*=O(N)
- Balanced BST: *h*=O(log N)

- How to guarantee *h*=O(log N)?

Binary Search Tree

- Operations: **Searching**, **Insertion**, **Deletion** can be performed in O(*h*) time, where *h* is the height of the tree.

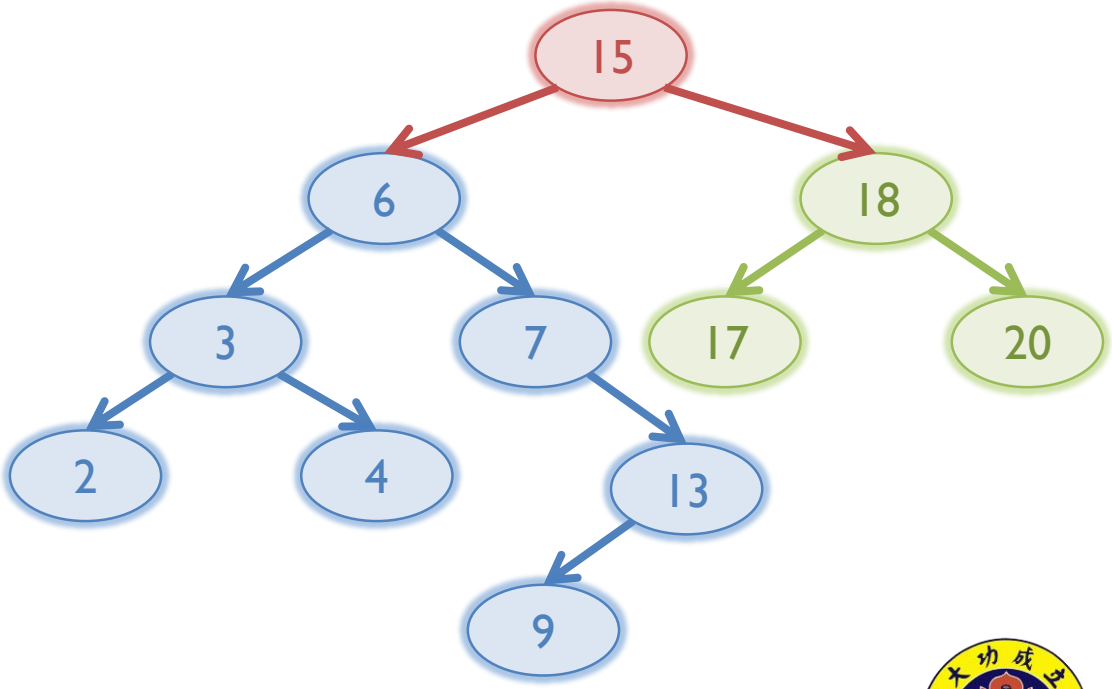
- Worst case: *h*=O(N)
- Balanced BST: *h*=O(log N)

- How to guarantee *h*=O(log N)?
  - Self-balancing Binary Search Tree!
  - AVL tree, Red-black tree, Treap, etc.

Searching

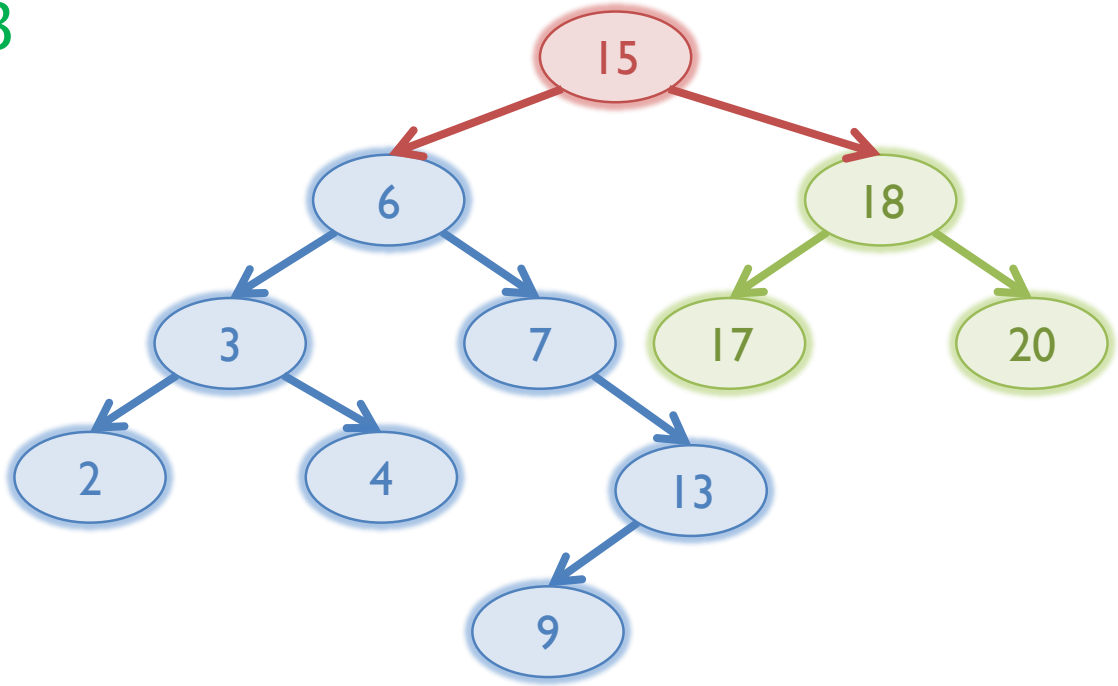
- **Searching** a binary search tree for a specific key.
- Can be a recursive or an iterative process.

- Example:
- Searching 13
- Searching 14



Searching

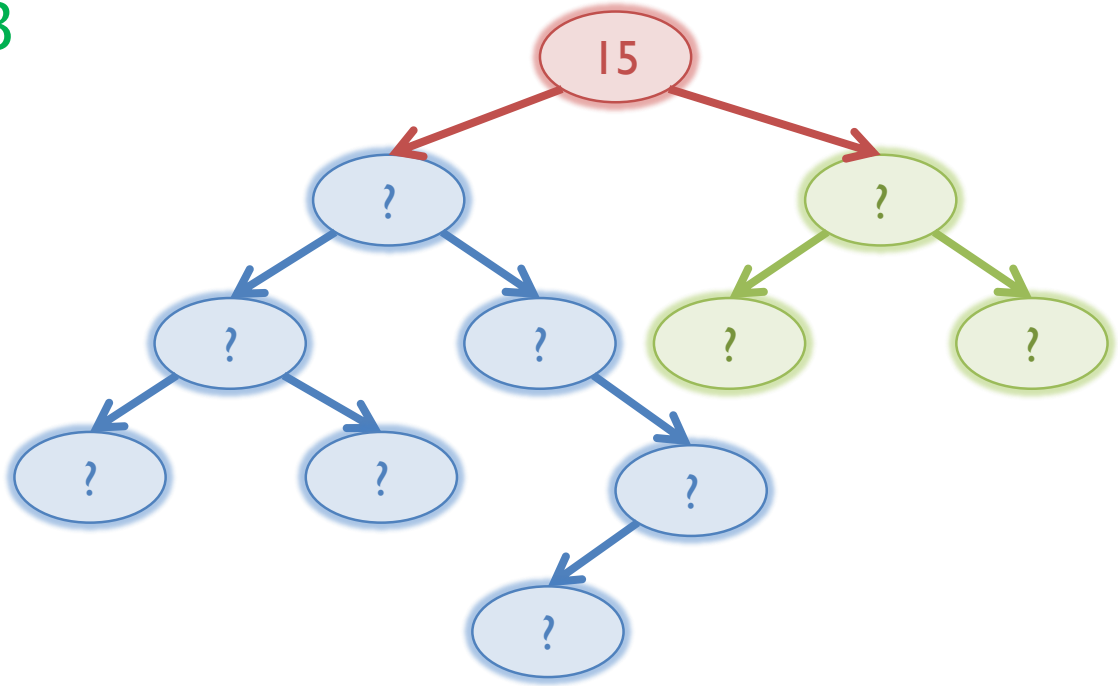
- Searching 13



Searching

- Searching 13

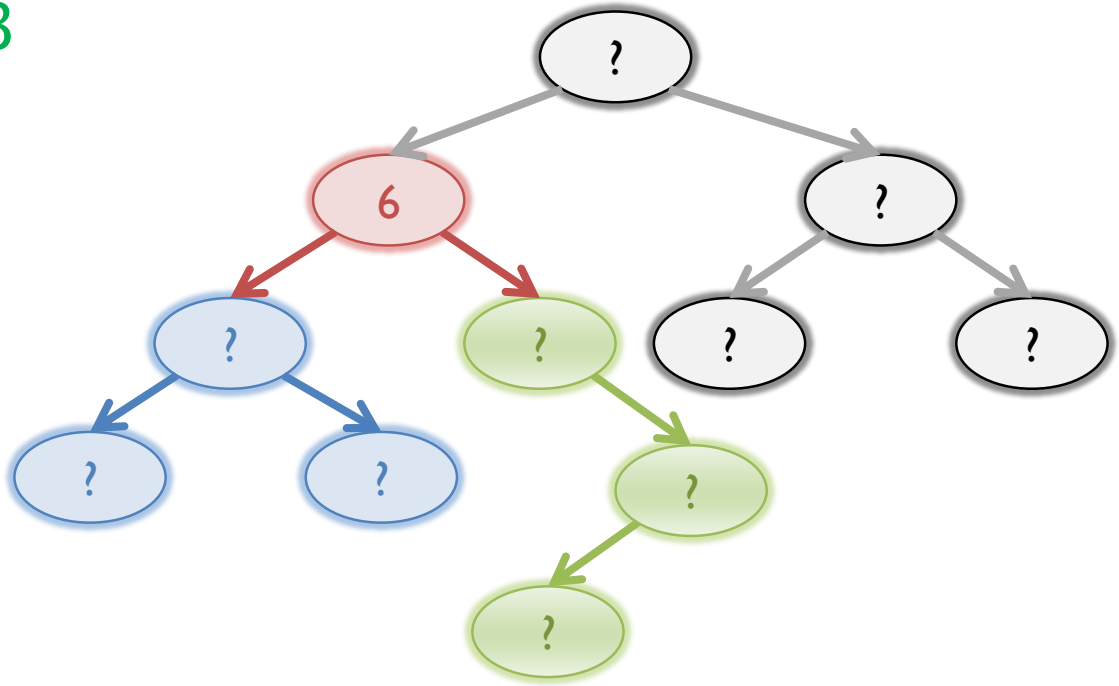
- 13 = 15 ?
  - No
- 13 < 15 ?
  - Yes



Searching

- Searching 13

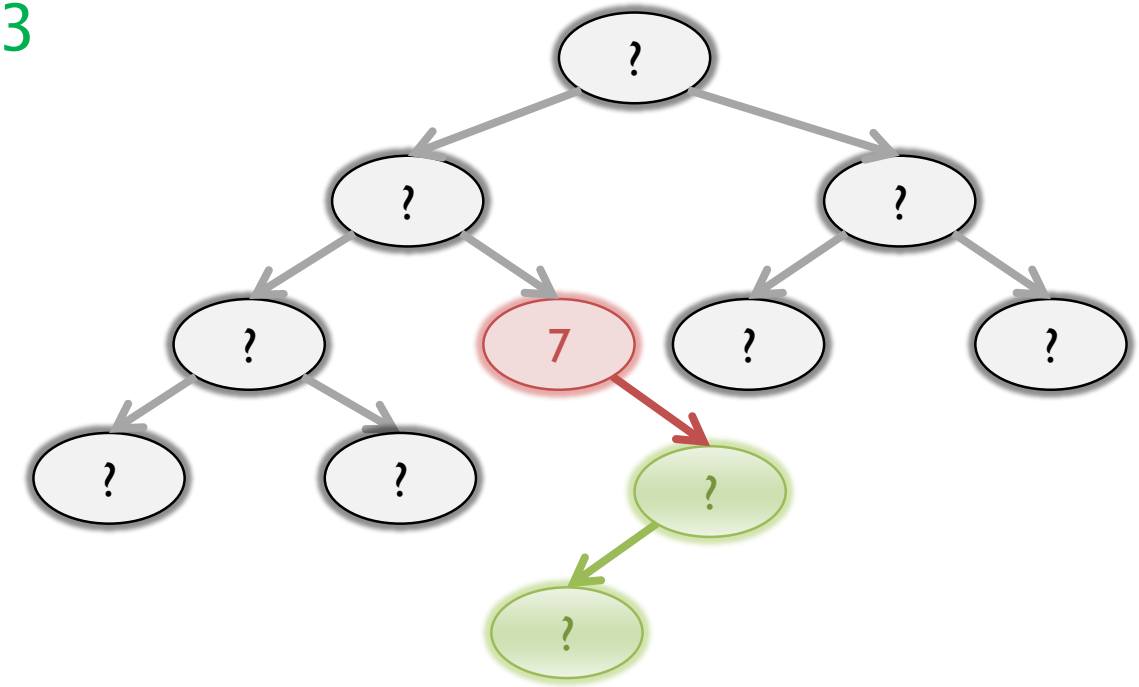
- 13 = 6 ?
  - No
- 13 < 6 ?
  - No
- 13 > 6 ?
  - Yes



Searching

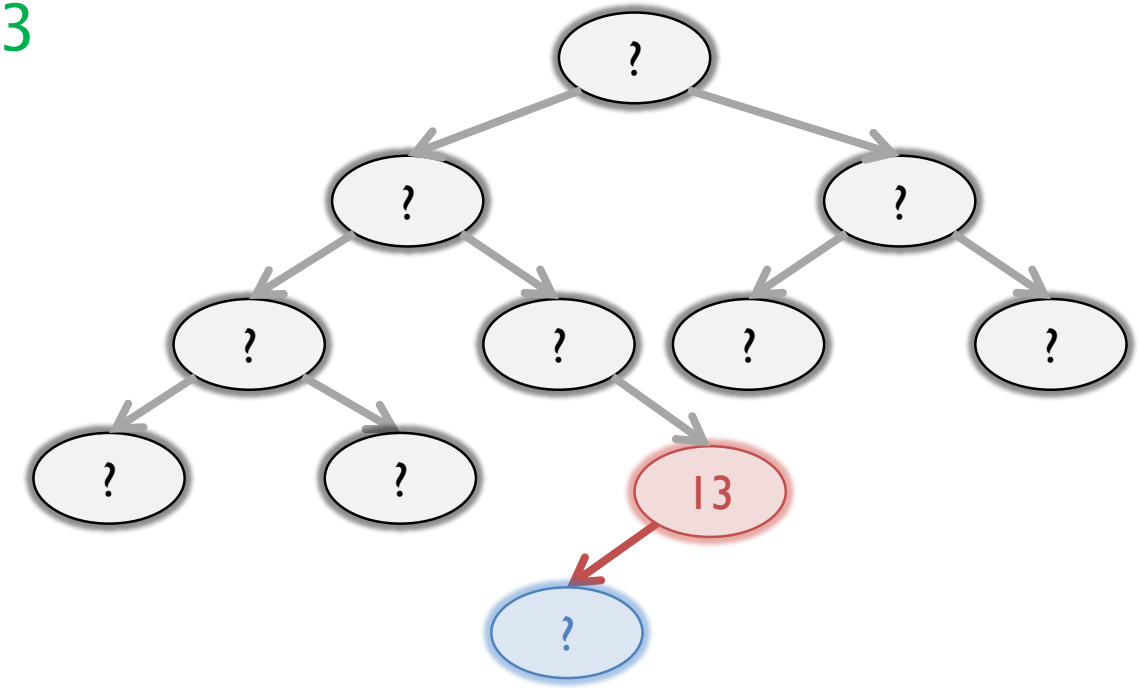
- Searching 13

- 13 = 7 ?
  - No
- 13 < 7 ?
  - No
- 13 > 7 ?
  - Yes



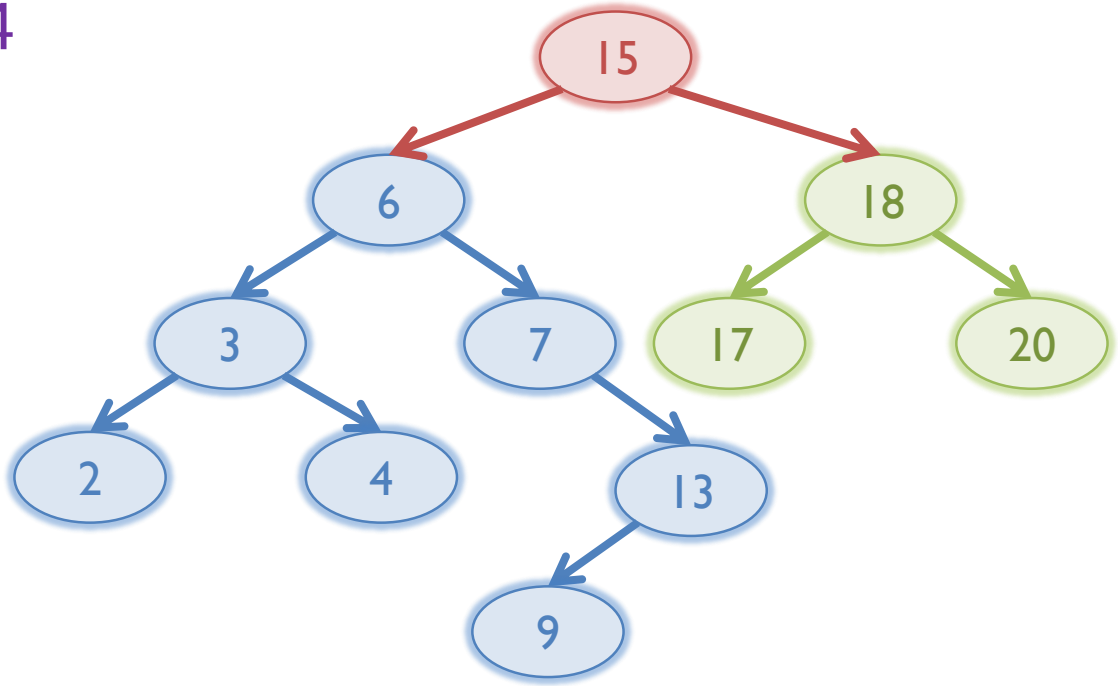
Searching

- Searching 13
- 13 = 13 ?
  - Yes
- Find!



Searching

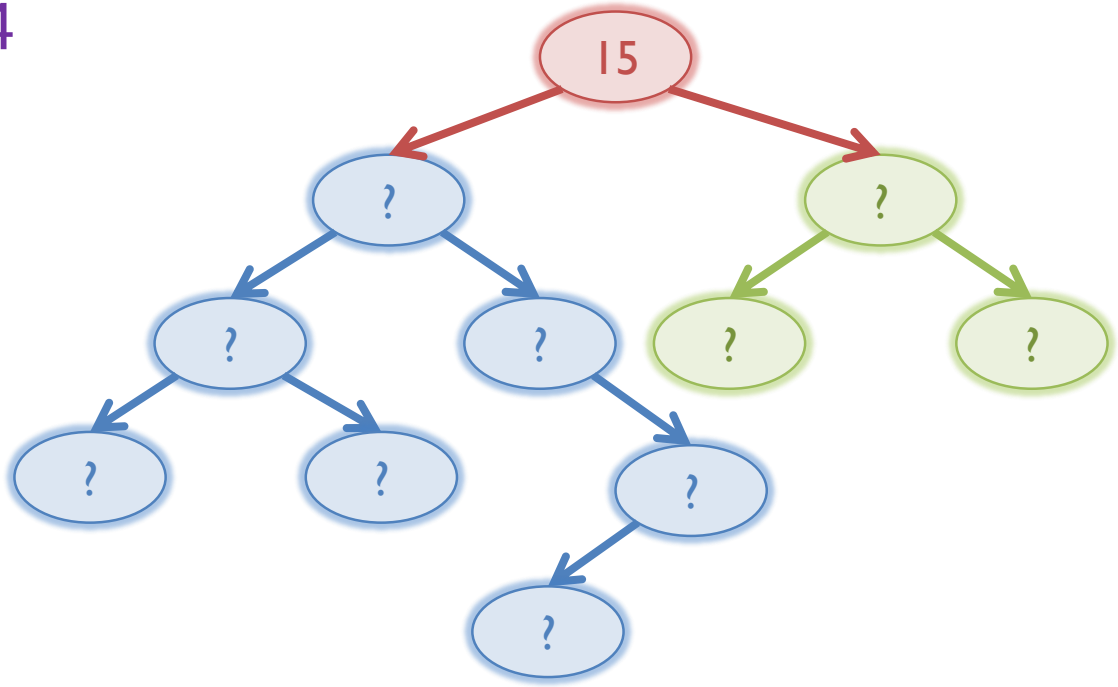
- Searching 14



Searching

- Searching 14

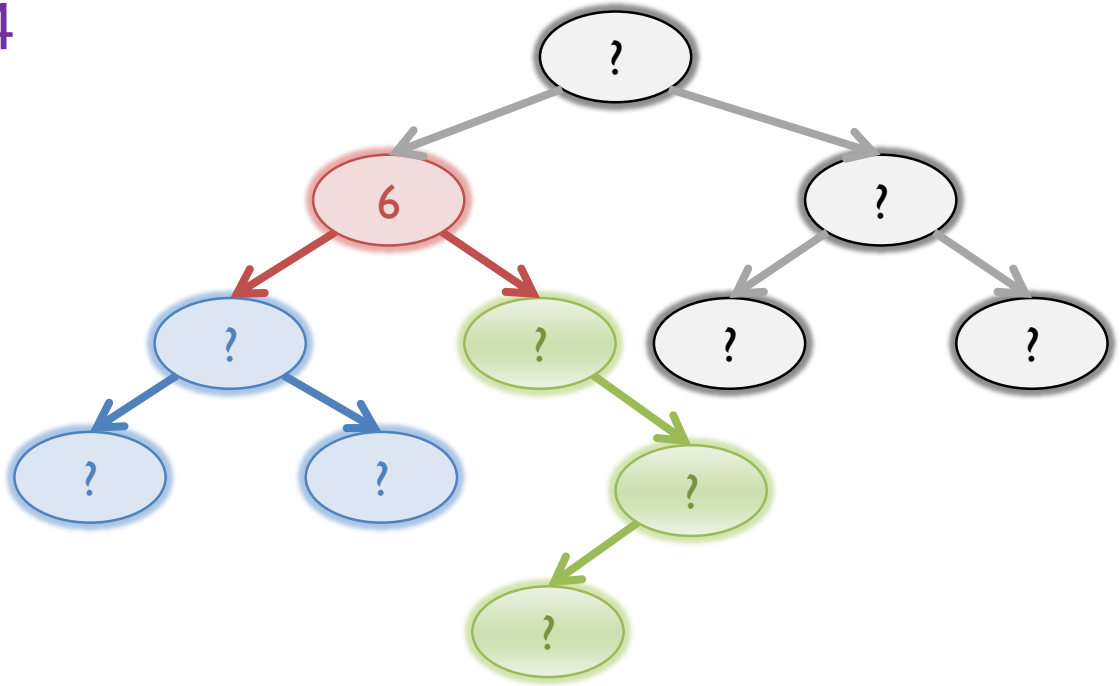
- 14 = 15 ?
  - No
- 14 < 15 ?
  - Yes



Searching

- Searching 14

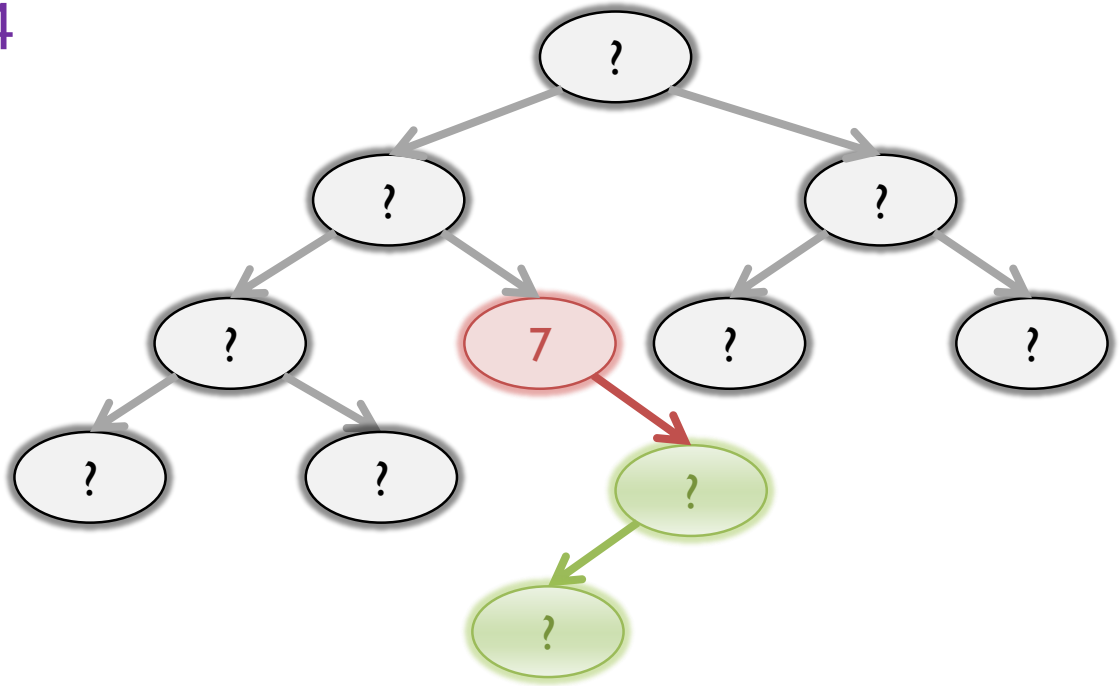
- 14 = 6 ?
  - No
- 14 < 6 ?
  - No
- 14 > 6 ?
  - Yes



Searching

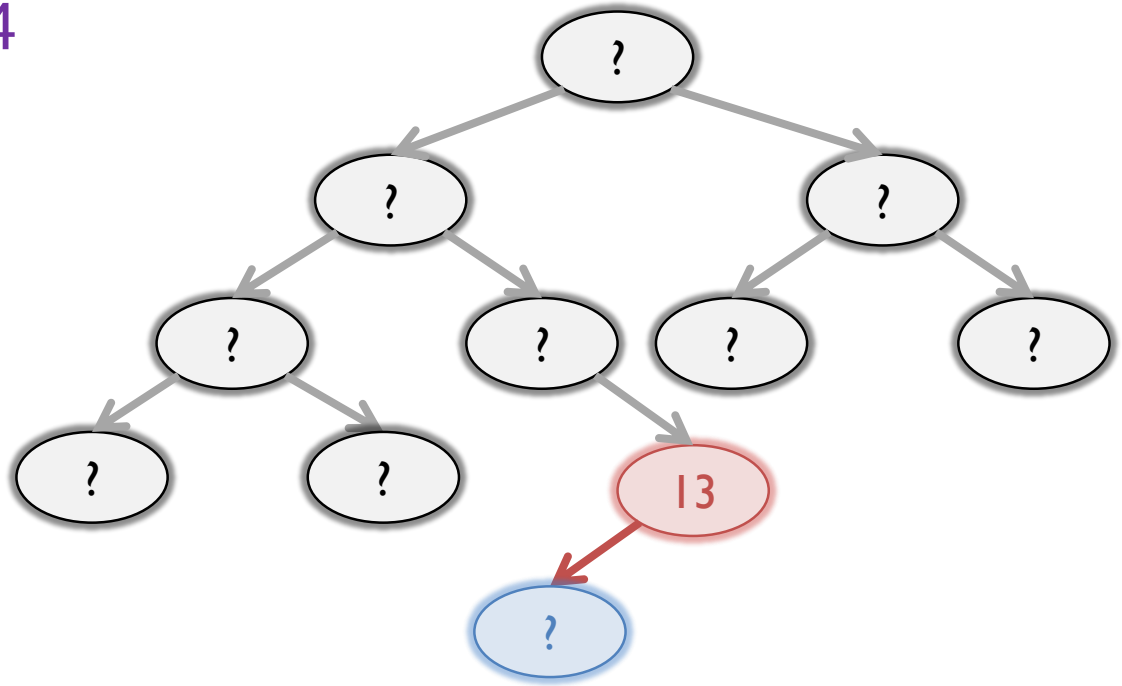
- Searching 14

- 14 = 7 ?
  - No
- 14 < 7 ?
  - No
- 14 > 7 ?
  - Yes



Searching

- Searching 14



- 14 = 13 ?
  - No
- 14 < 13 ?
  - No
- 14 > 13 ?
  - Yes
- But there is no sub-tree……
- Not exist!

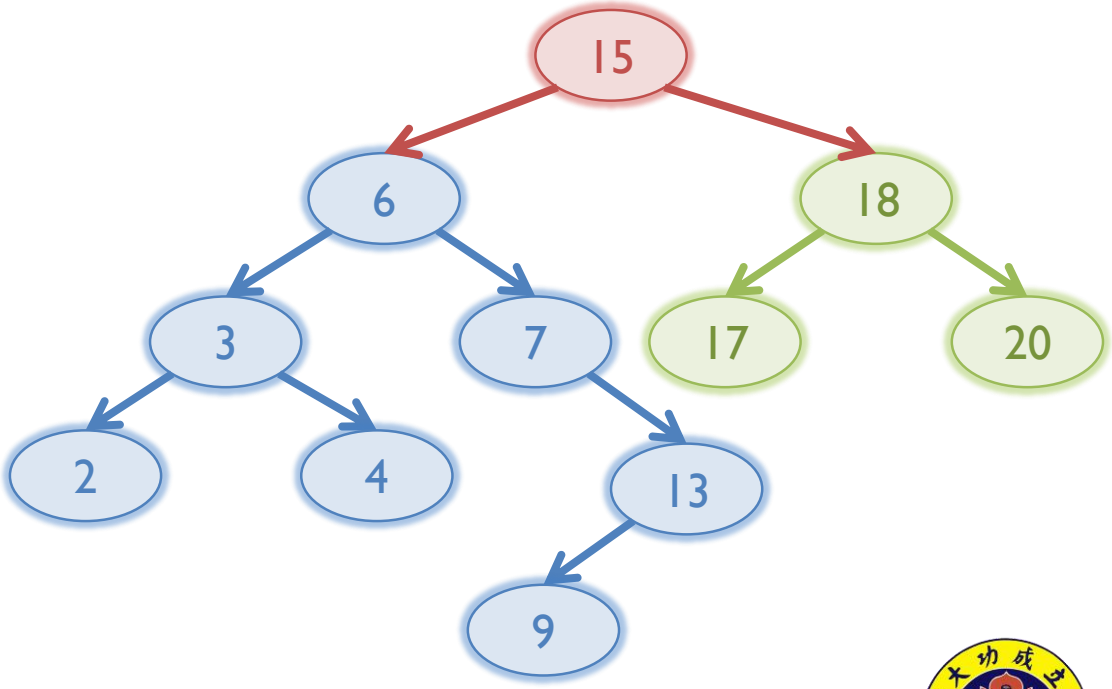
Searching

```
Recursive-Search(x, k)
1.   if x = NULL or key[x]= k
2.        return x
3.   else if k < key[x]
4.        return Recursive-search(left[x],k)
5.   else
6.        return Recursive-search(right[x],k)
```

Insertion

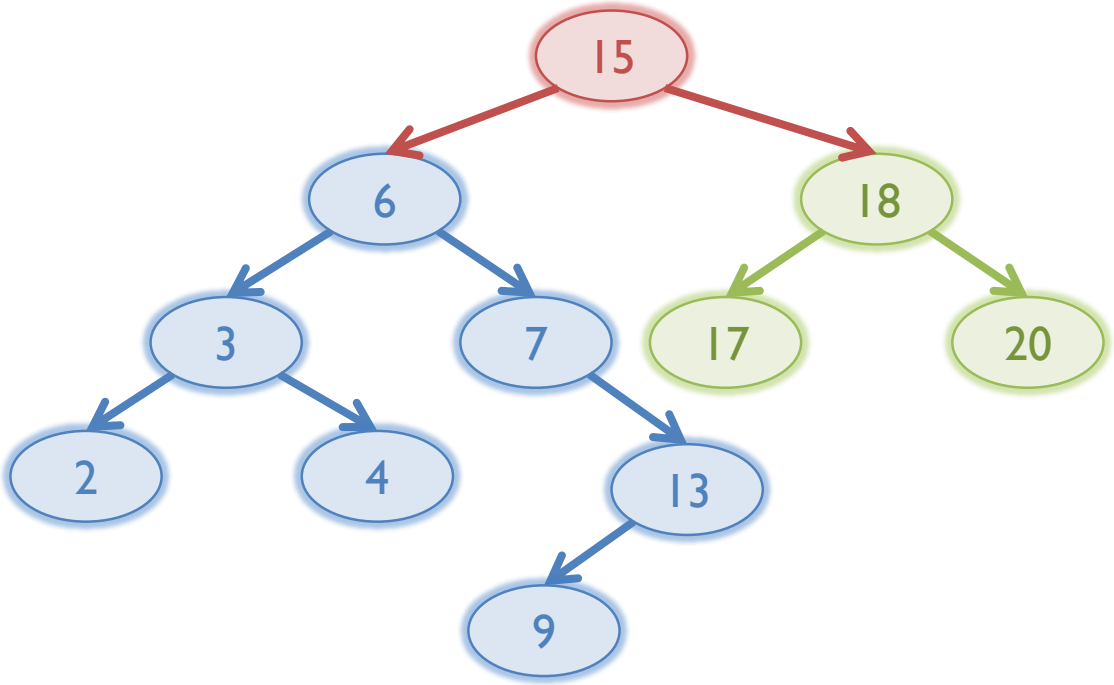
- **Insert** a key to a binary search tree.
- Searching the key, then add a new node with the key value at the external place.

- Example:
- Insert 14
- Insert 19



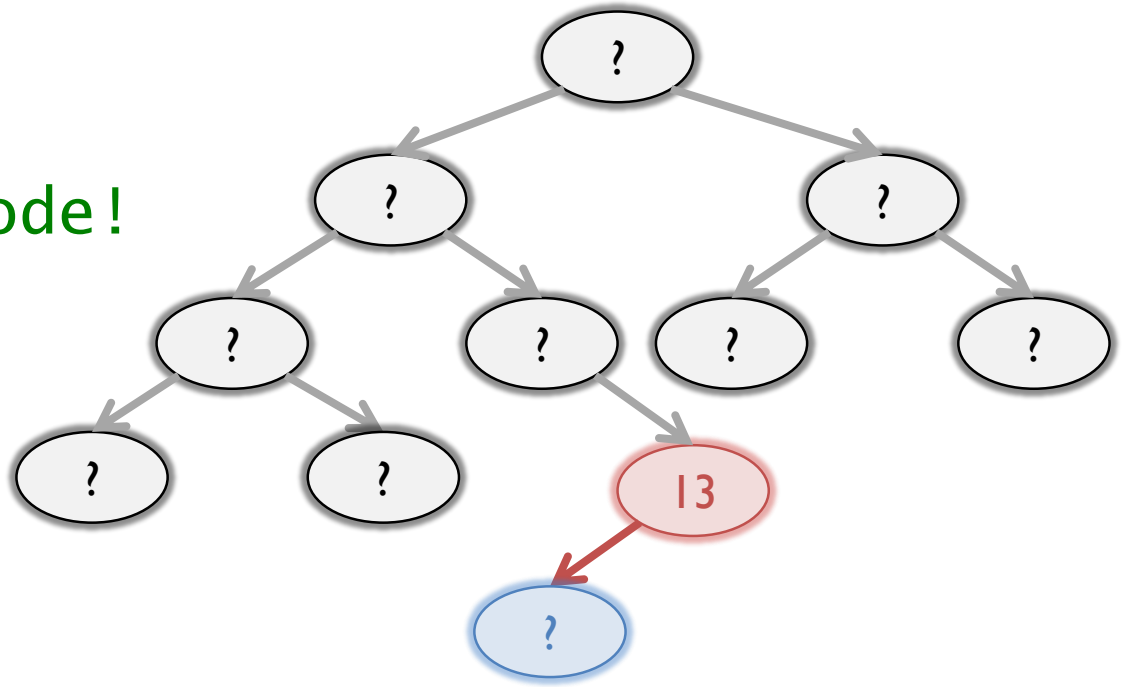
Insertion

- `Insert` 14



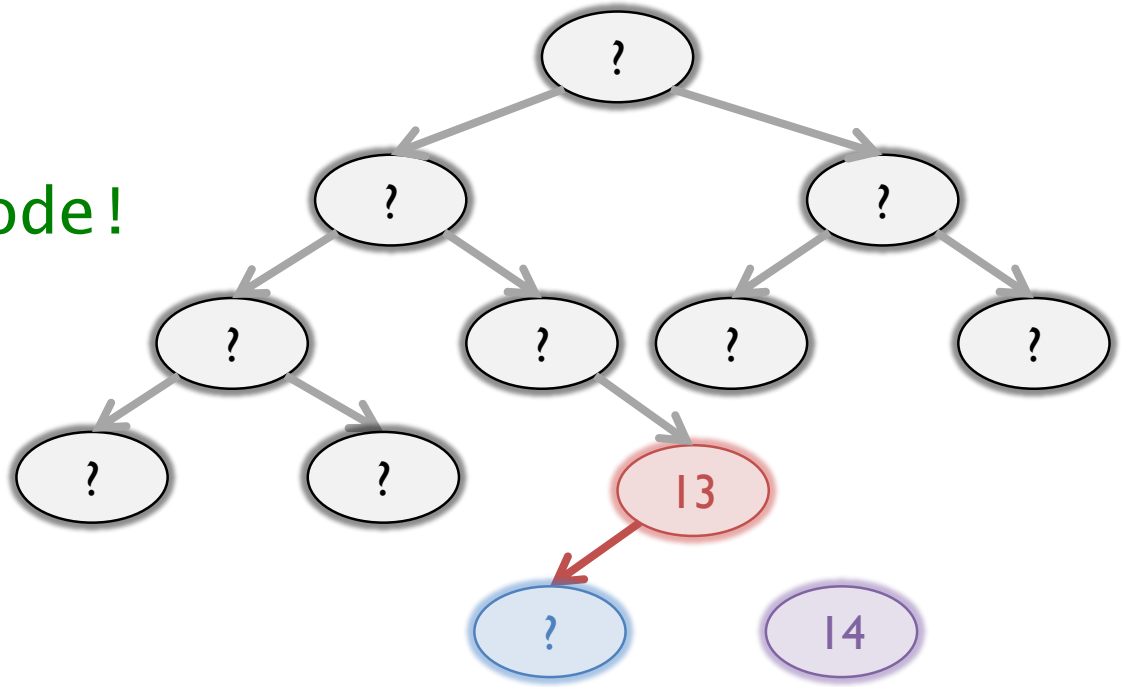
Insertion

- `Insert 14`
- `Add a new node!`



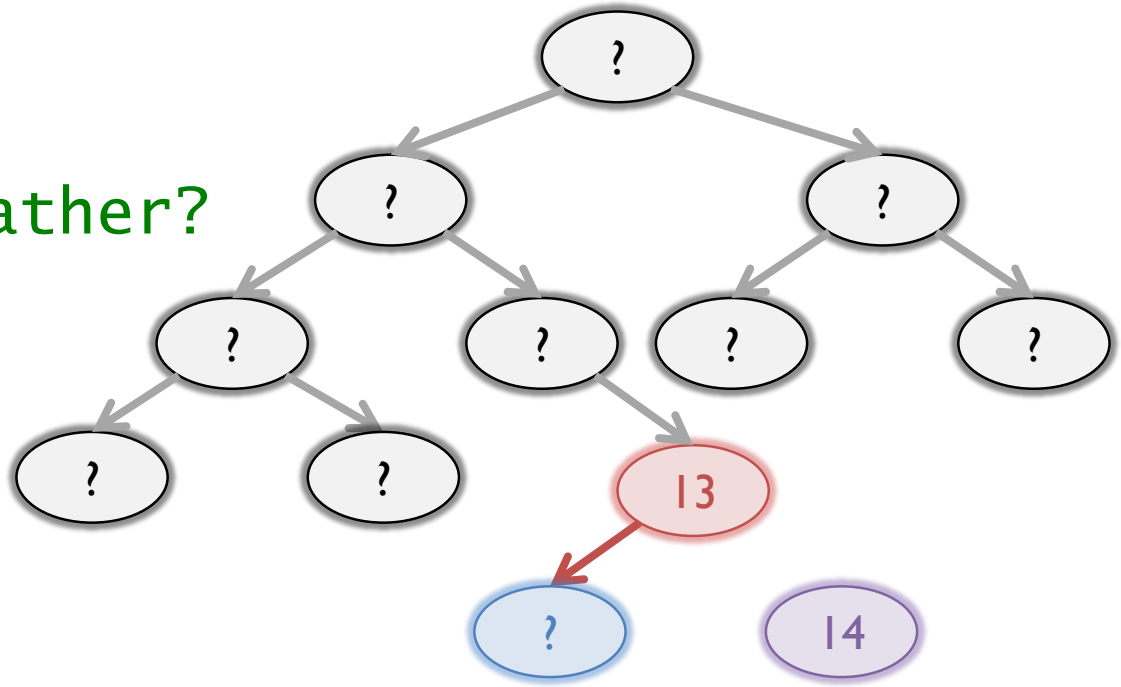
Insertion

- `Insert` 14
- Add a new node!



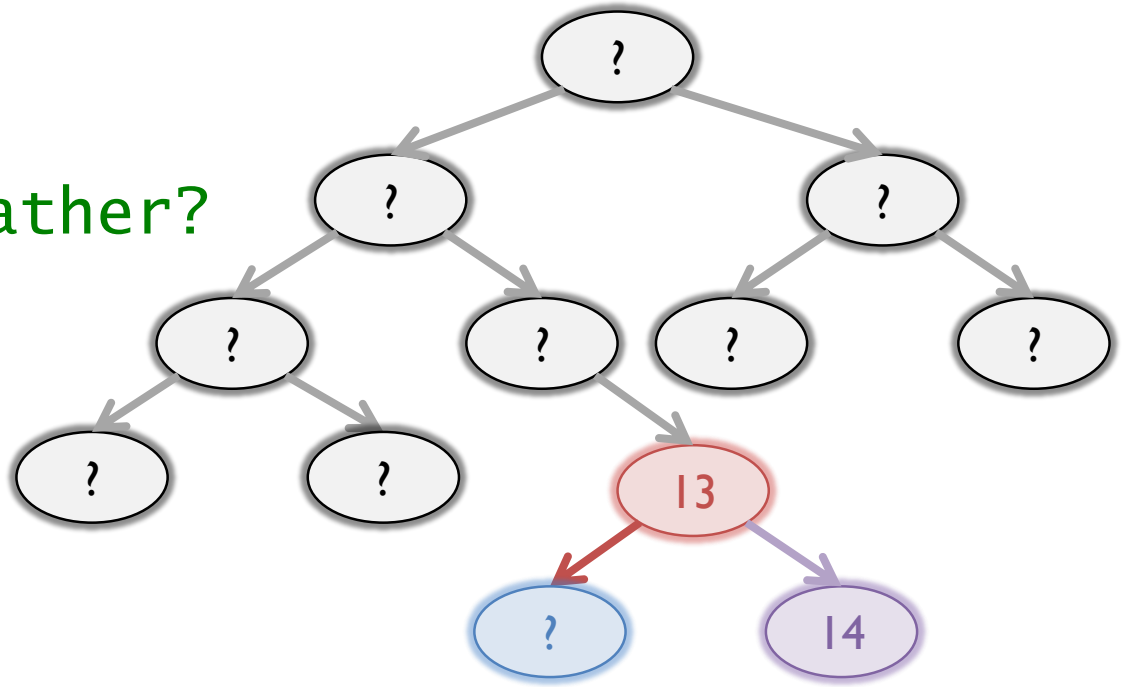
Insertion

- Insert 14
- Who is my father?



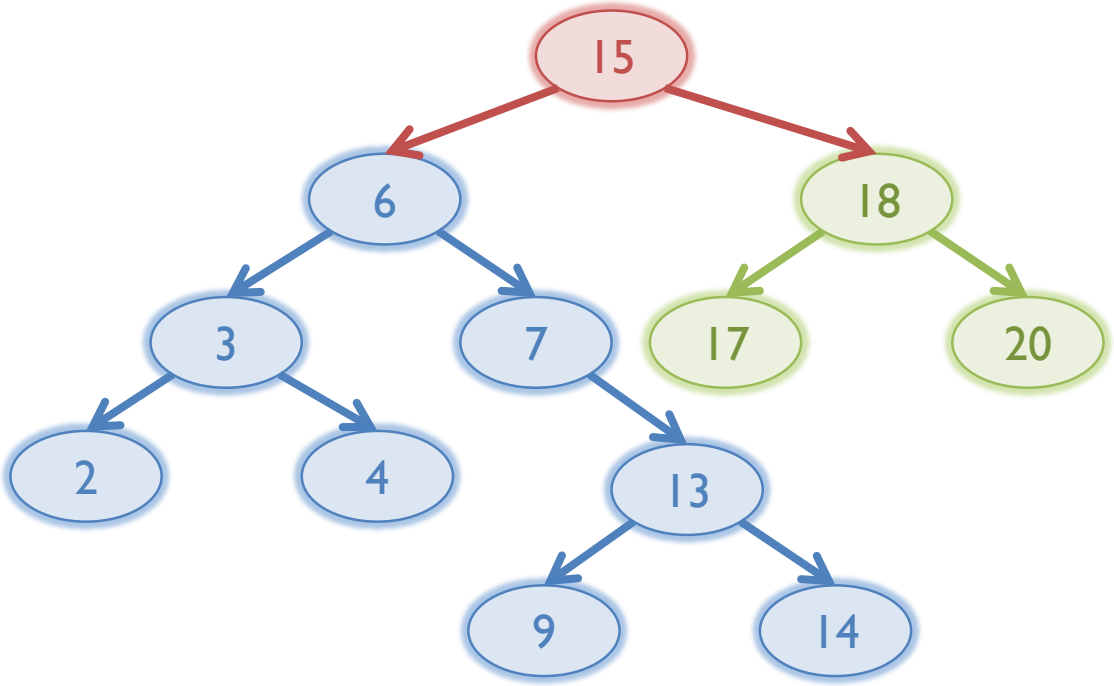
Insertion

- Insert 14
- Who is my father?



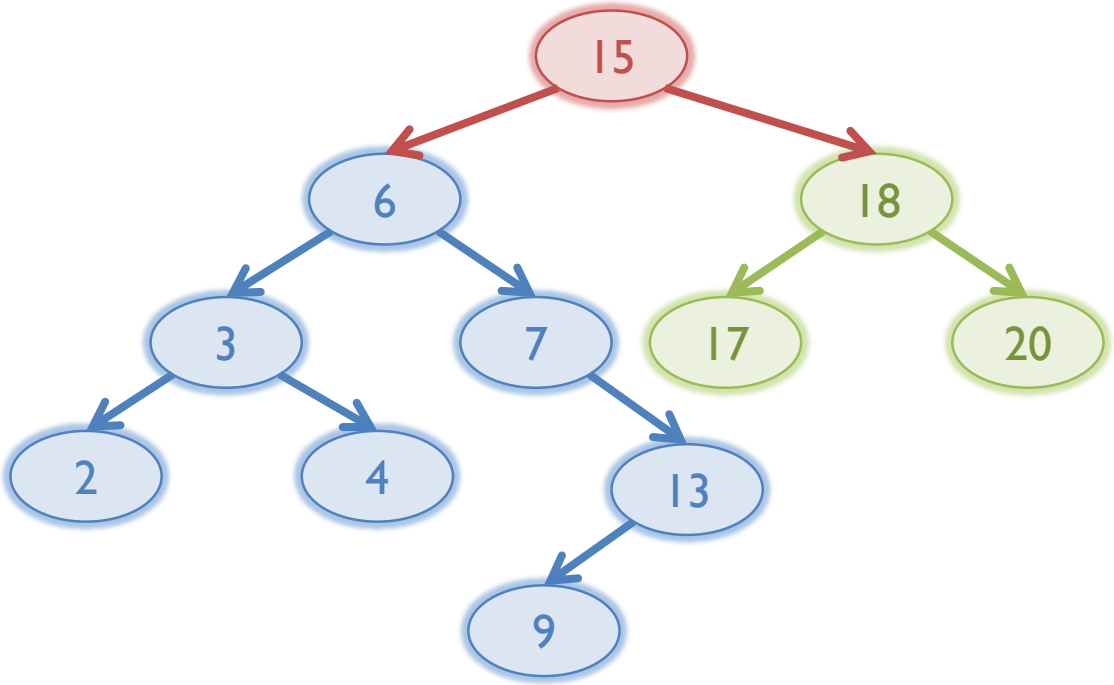
Insertion

- `Insert` 14
- Done!



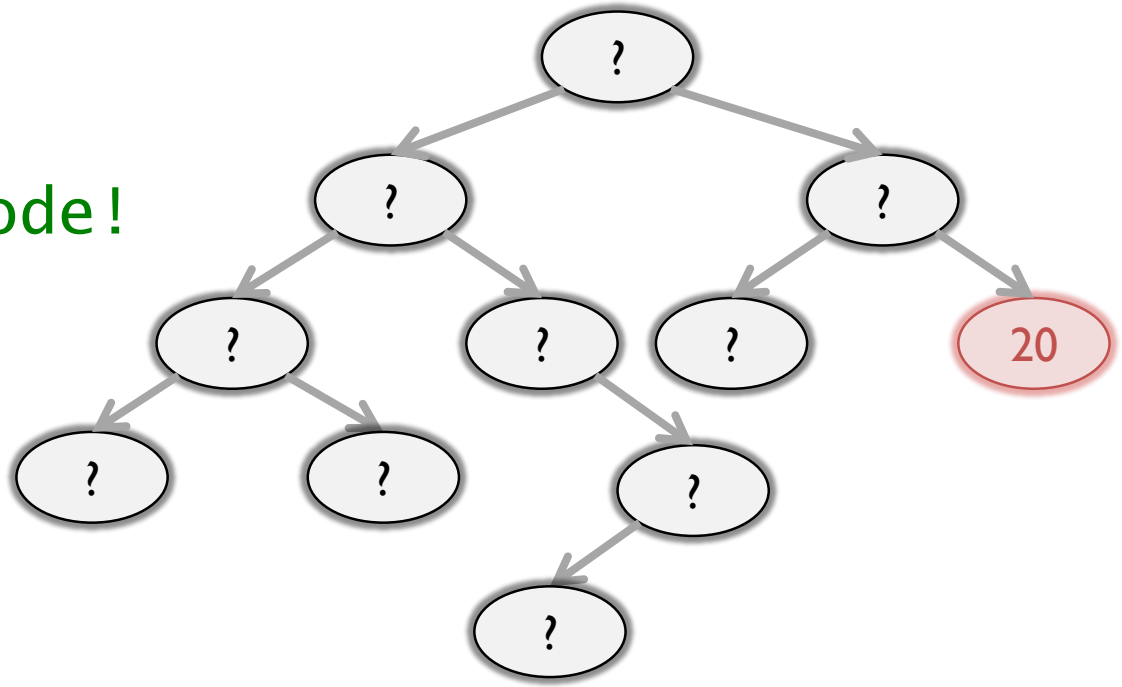
Insertion

- `Insert` 19



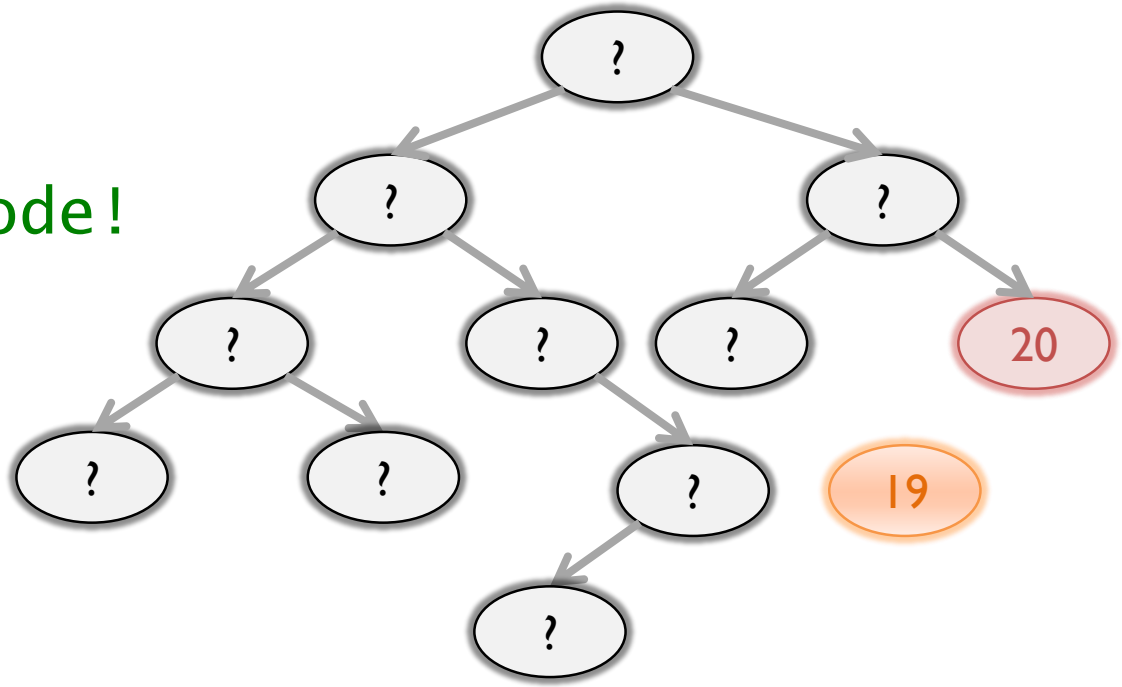
Insertion

- `Insert 19`
- `Add a new node!`



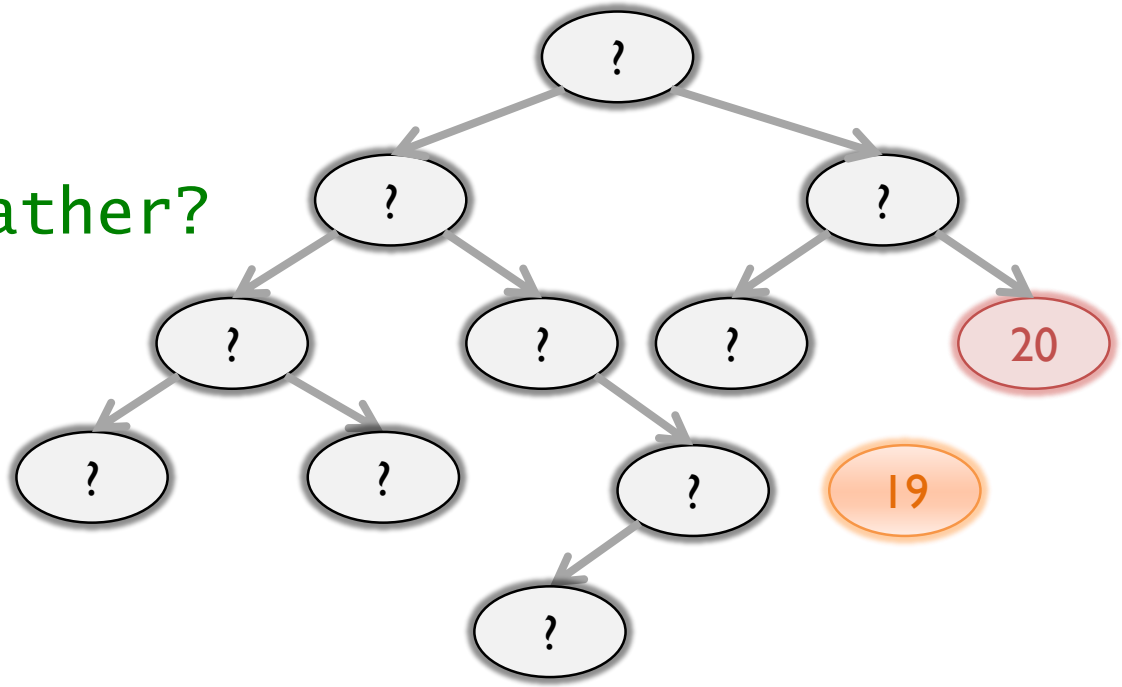
Insertion

- Insert 19
- Add a new node!



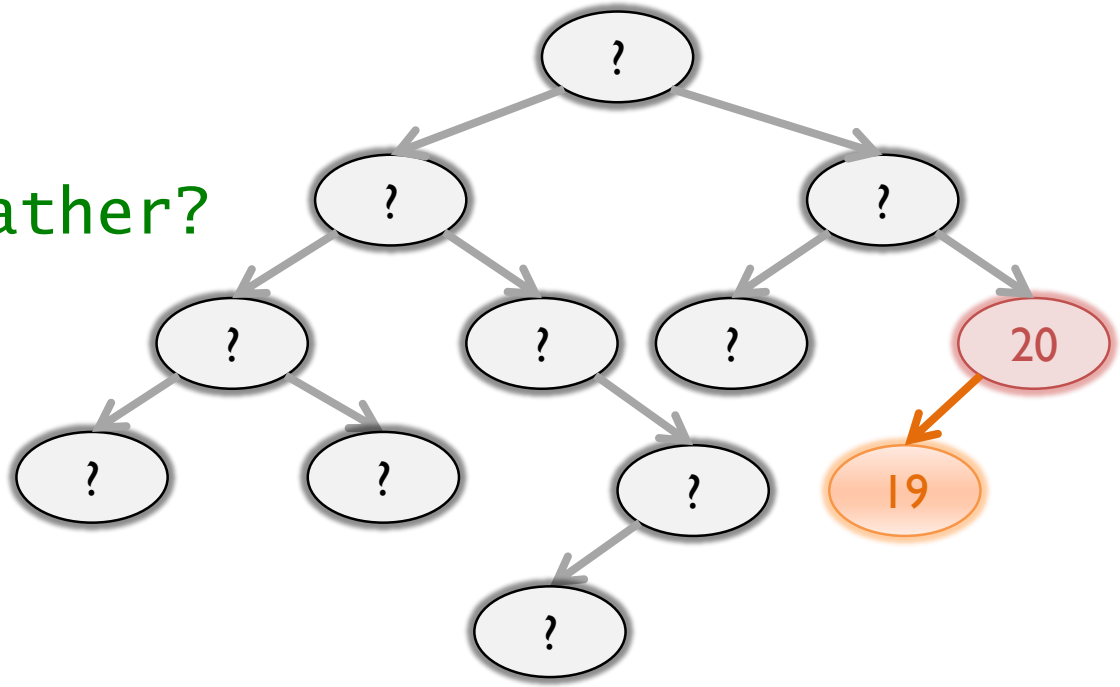
Insertion

- Insert 19
- Who is my father?



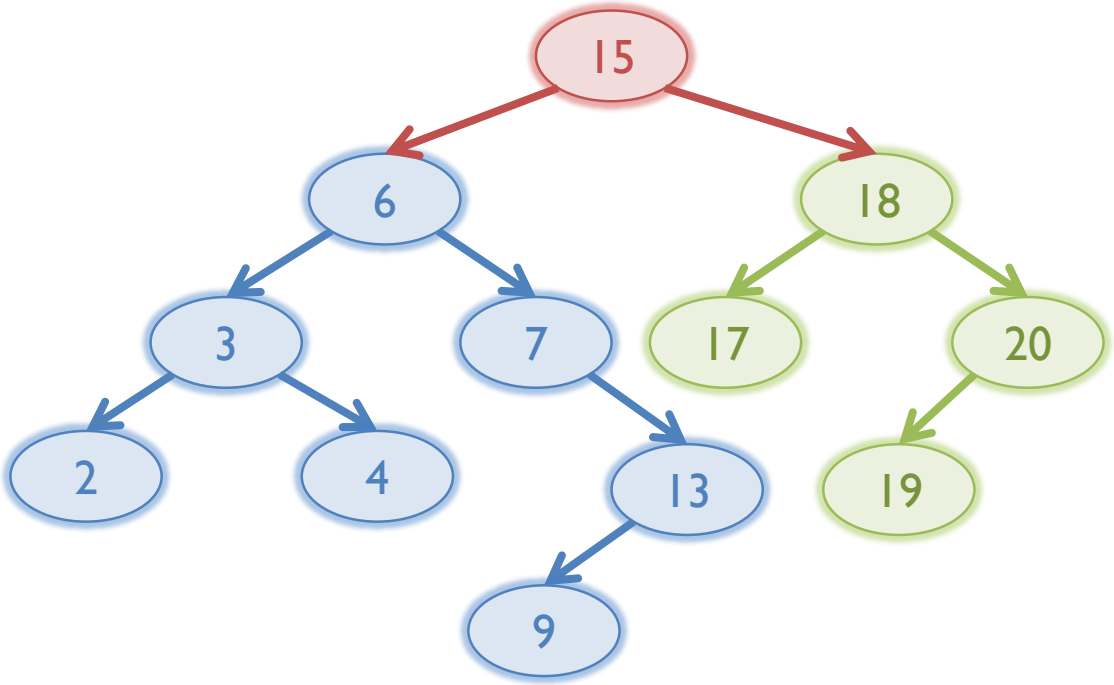
Insertion

- Insert 19
- Who is my father?



Insertion

- `Insert` 19
- Done!



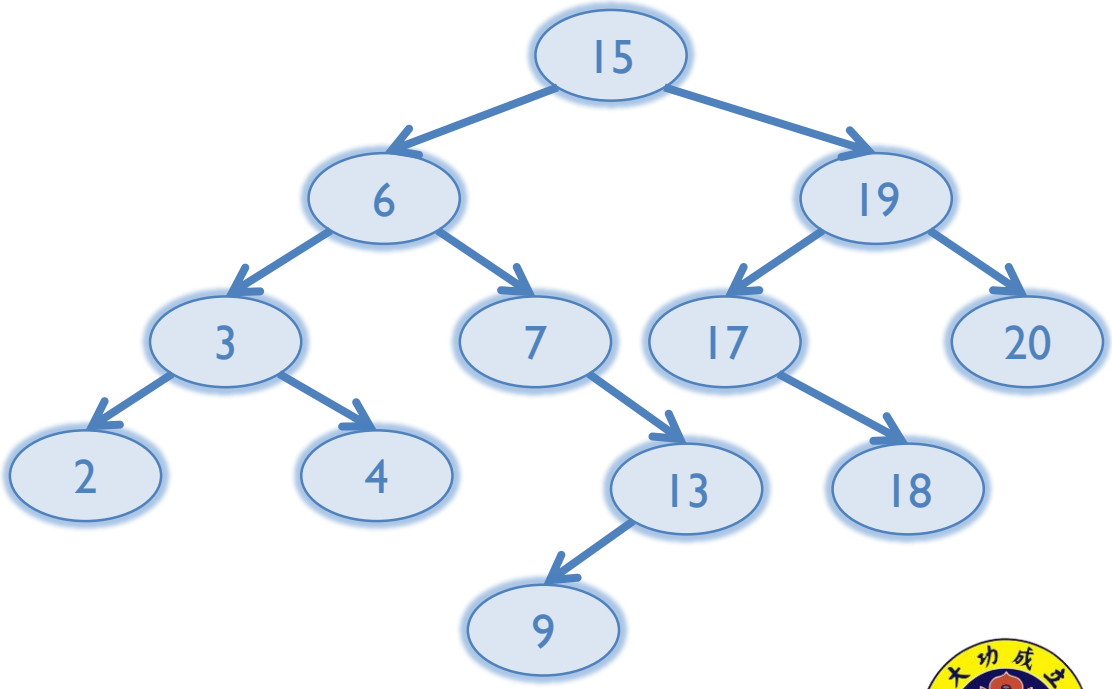
Insertion

```
Insert(Tree, k)
1.  p ← NULL
2.  x ← root[Tree]
3.  while x ≠ NULL
4.    do p ← x
5.       if k < key[x]
6.         x ← left[x]
7.       else
8.         x ← right[x]
9.  z ← new node
10. if p = NULL
11.    root[Tree] ← z
12.    /* Tree was empty */
13. else if k < key[p]
14.    left[p] ← z
15.    /* ex: 19 < 20 */
16. else
17.    right[p] ← z
18.    /* ex: 14 > 13 */
```

Deletion

- **Deleting** a node on a binary search tree.

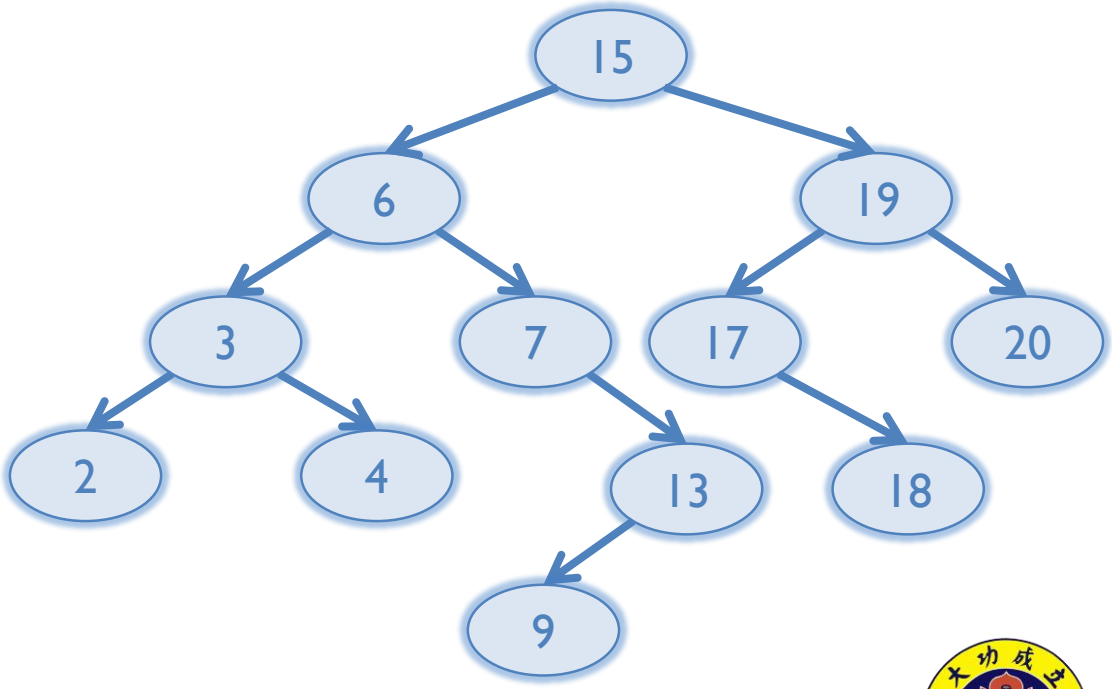
- Example:
- Deleting 4
- Deleting 13
- Deleting 19



Deletion

- **Deleting** a node on a binary search tree.

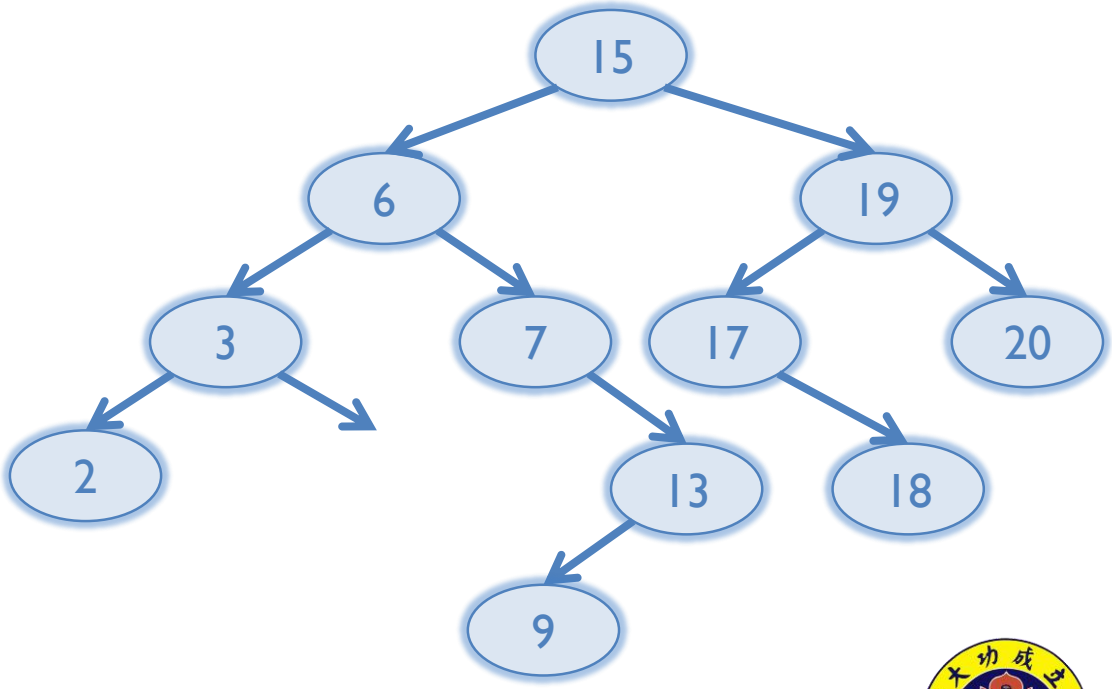
- Example:
- Deleting 4



Deletion

- **Deleting** a node on a binary search tree.

- Example:
- Deleting 4

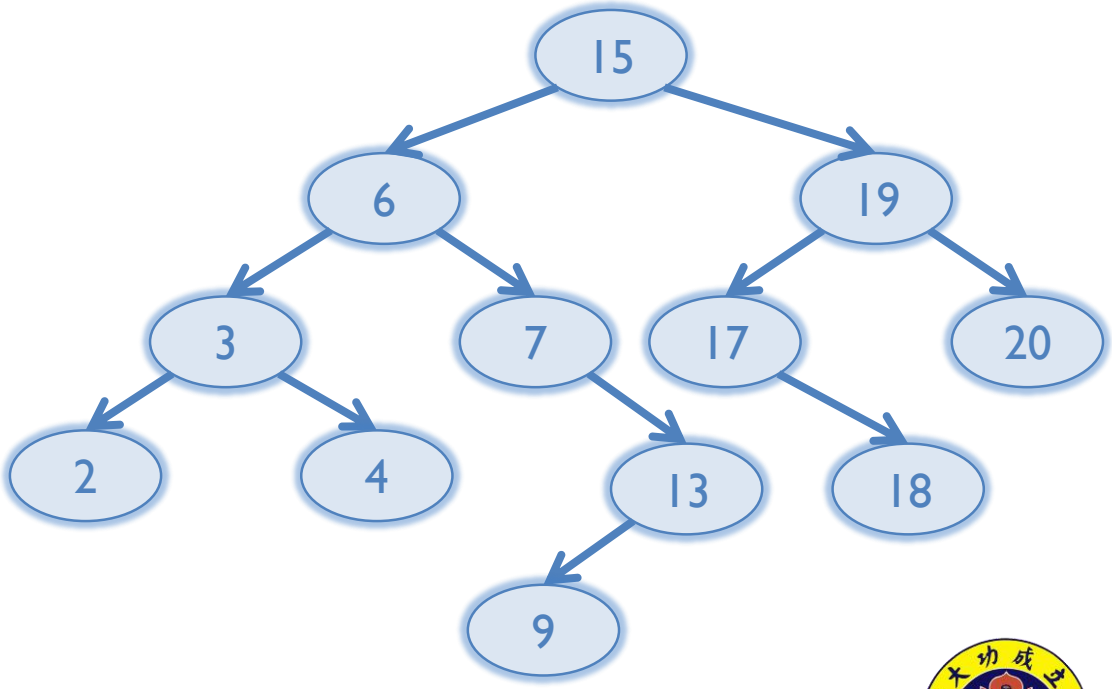


Deletion

- **Deleting** a node on a binary search tree.

- Example:

- Deleting 13

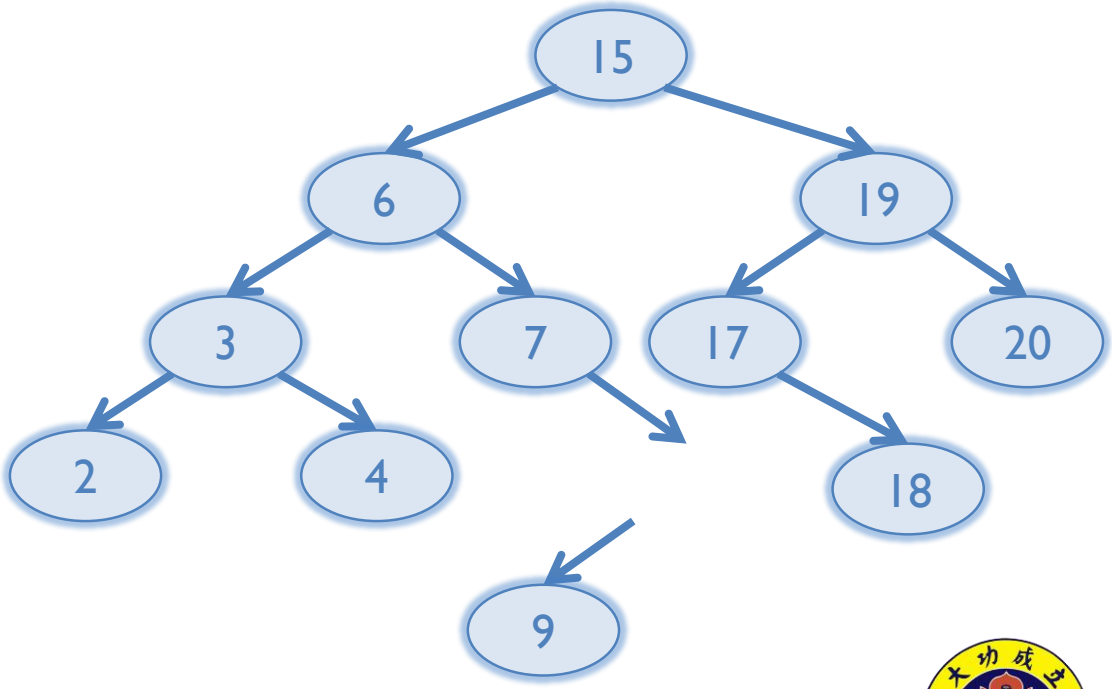


Deletion

- **Deleting** a node on a binary search tree.

- Example:

- Deleting 13

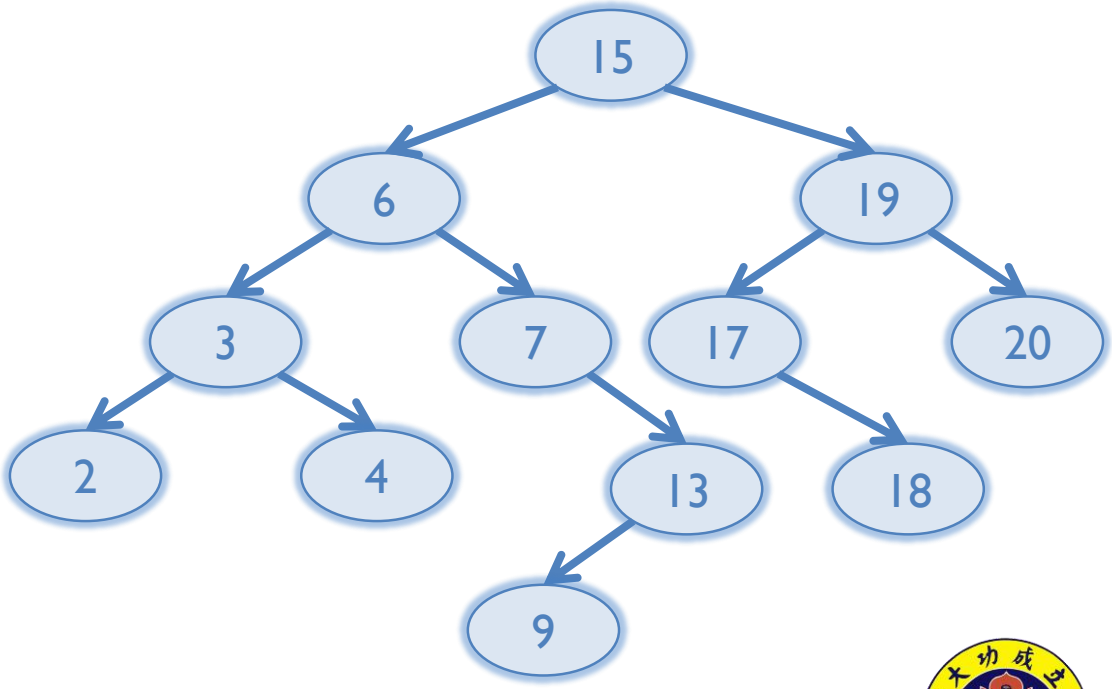


Deletion

- **Deleting** a node on a binary search tree.

- Example:

- Deleting 19

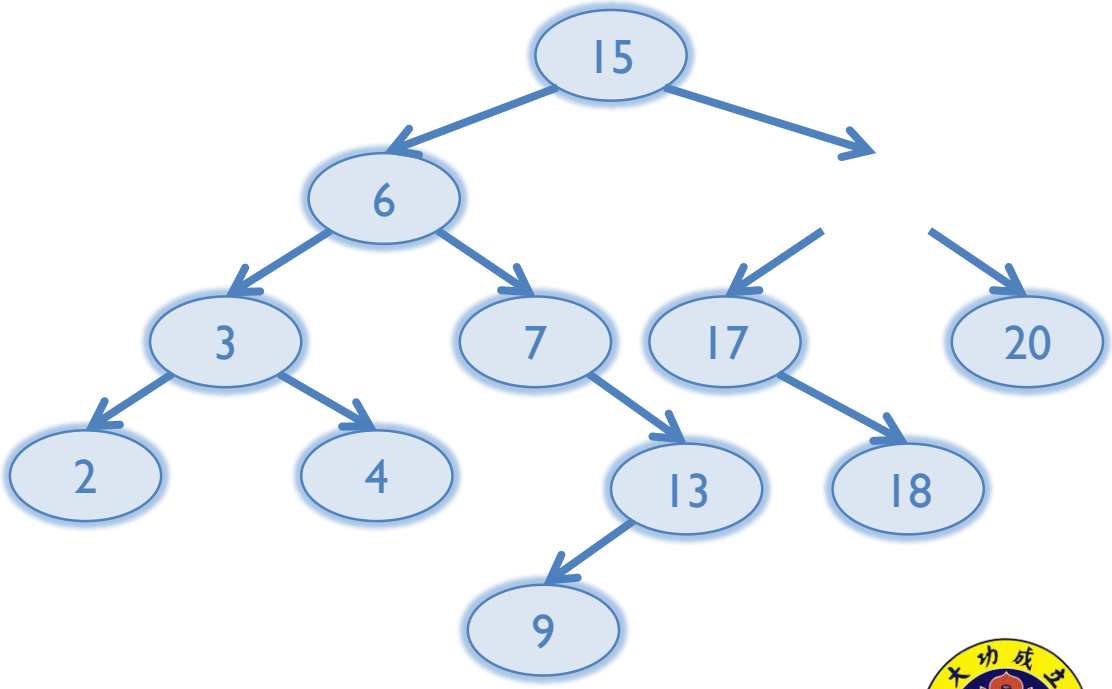


Deletion

- **Deleting** a node on a binary search tree.

- Example:

- Deleting 19



Deletion

- **Deleting** a node on a binary search tree.

Deletion

- **Deleting** a node on a binary search tree.
- Deleting a leaf: Simply remove it.

Deletion

- **Deleting** a node on a binary search tree.

- Deleting a leaf: Simply remove it.
- Deleting a node with 1 child: Remove the node and replace it with its child.

Deletion

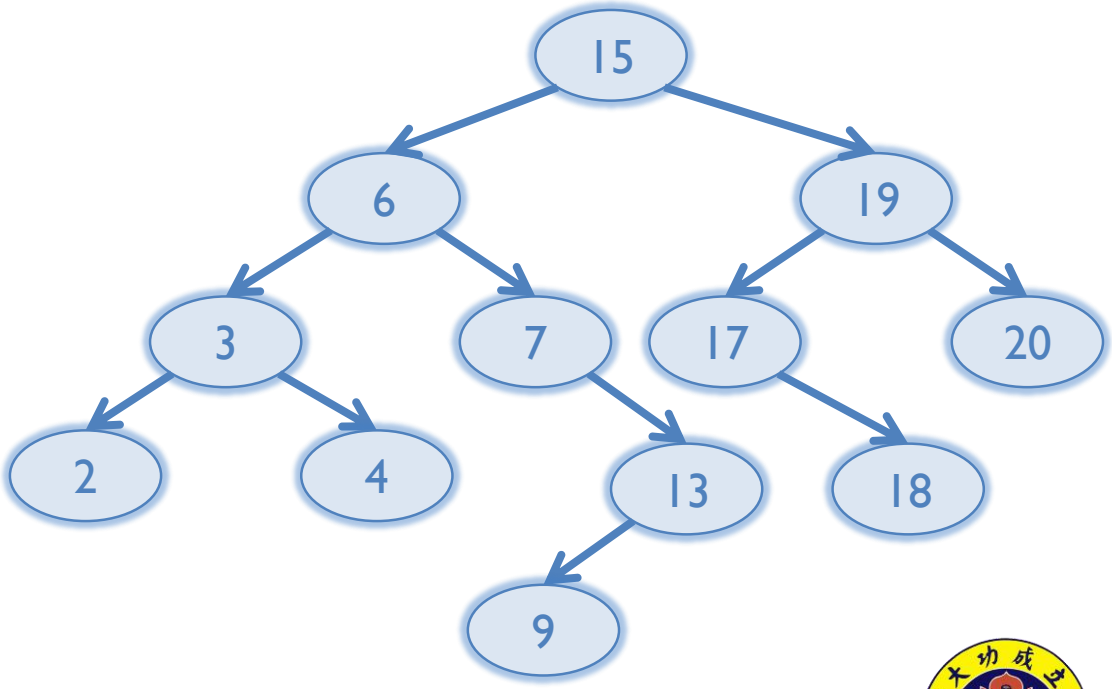
- **Deleting** a node on a binary search tree.

- Deleting a leaf: Simply remove it.
- Deleting a node with 1 child: Remove the node and replace it with its child.

- Deleting a node with 2 children: Call the node to be deleted N. Choose its in-order successor or predecessor node, R. Replace the data of N with the data of R, then delete R.

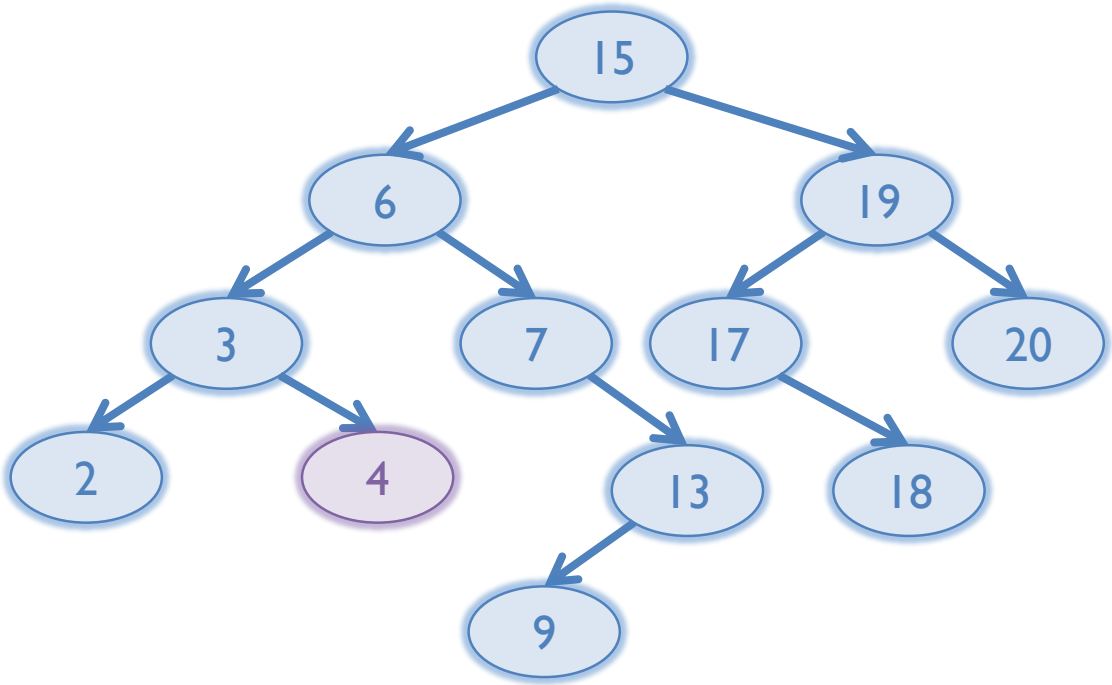
Deletion

- **Deleting** a node on a binary search tree.
- Three cases!
- Example:
- Deleting 4
- Deleting 13
- Deleting 15



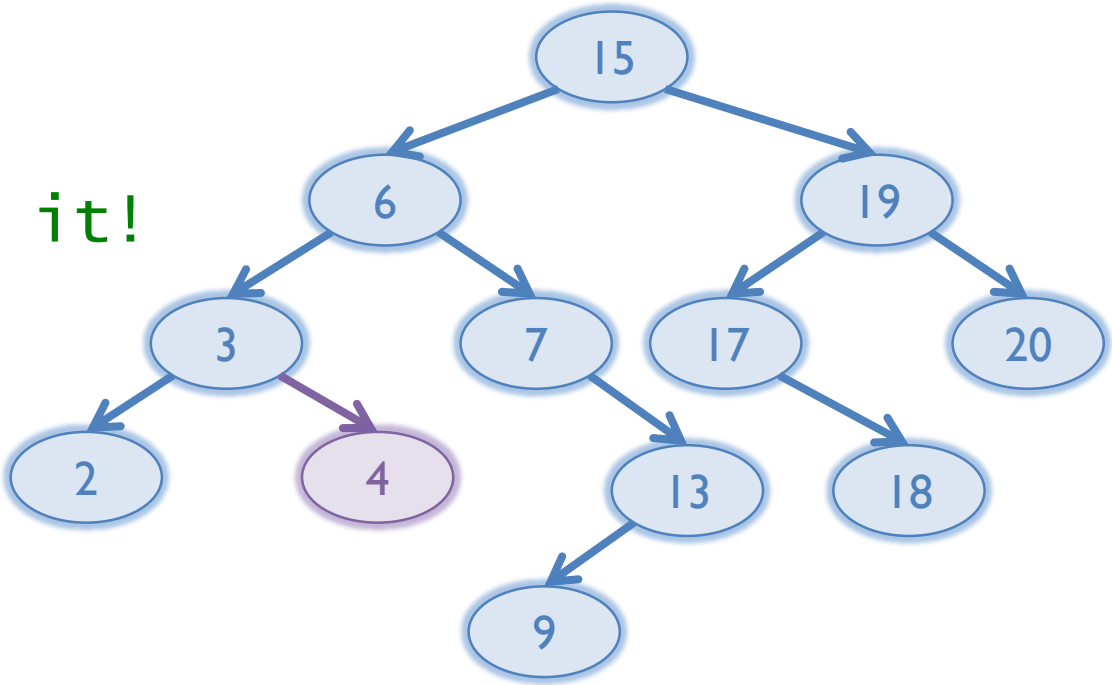
Deletion

- Deleting 4



Deletion

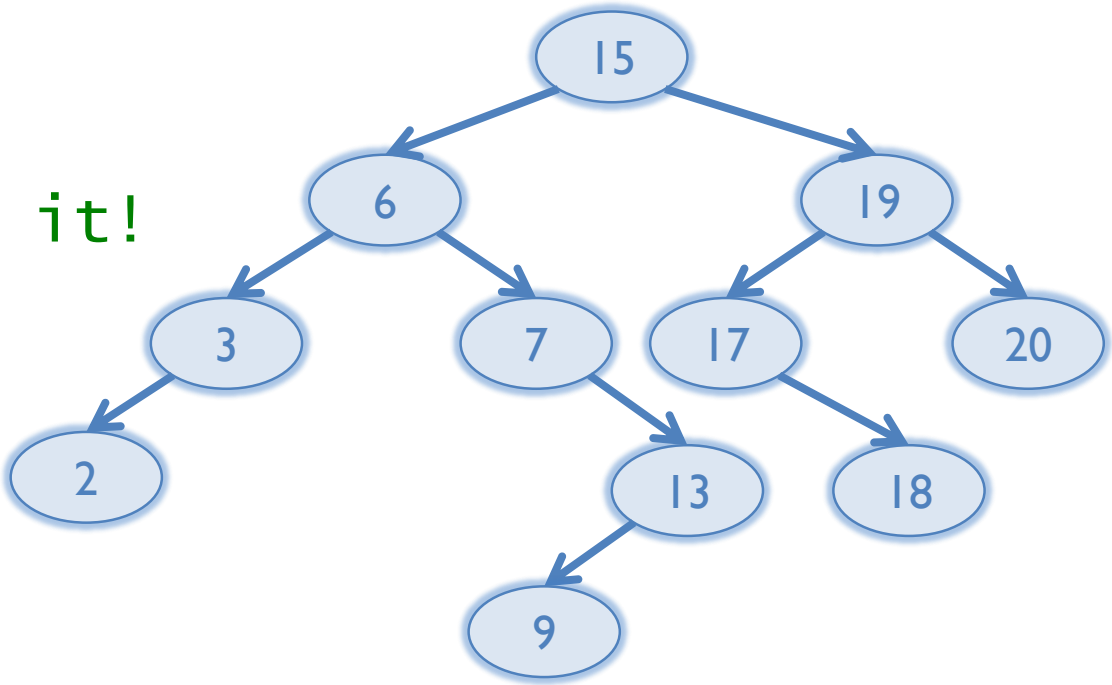
- Deleting 4
- Simply remove it!



Deletion

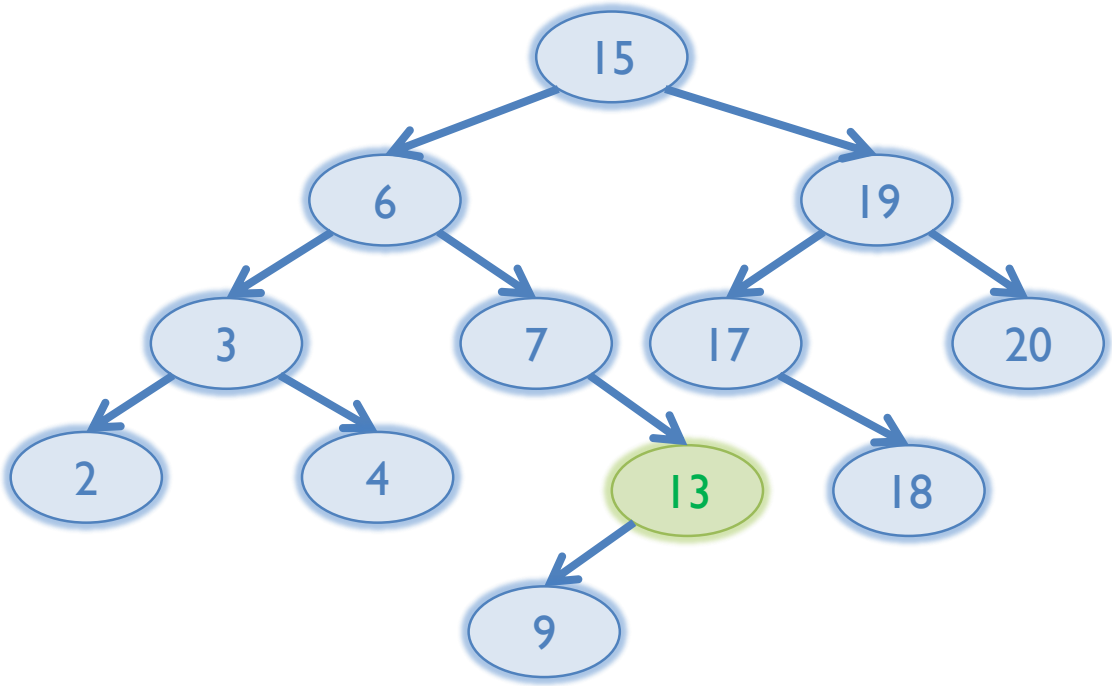
- Deleting 4

- Simply remove it!
- Done!



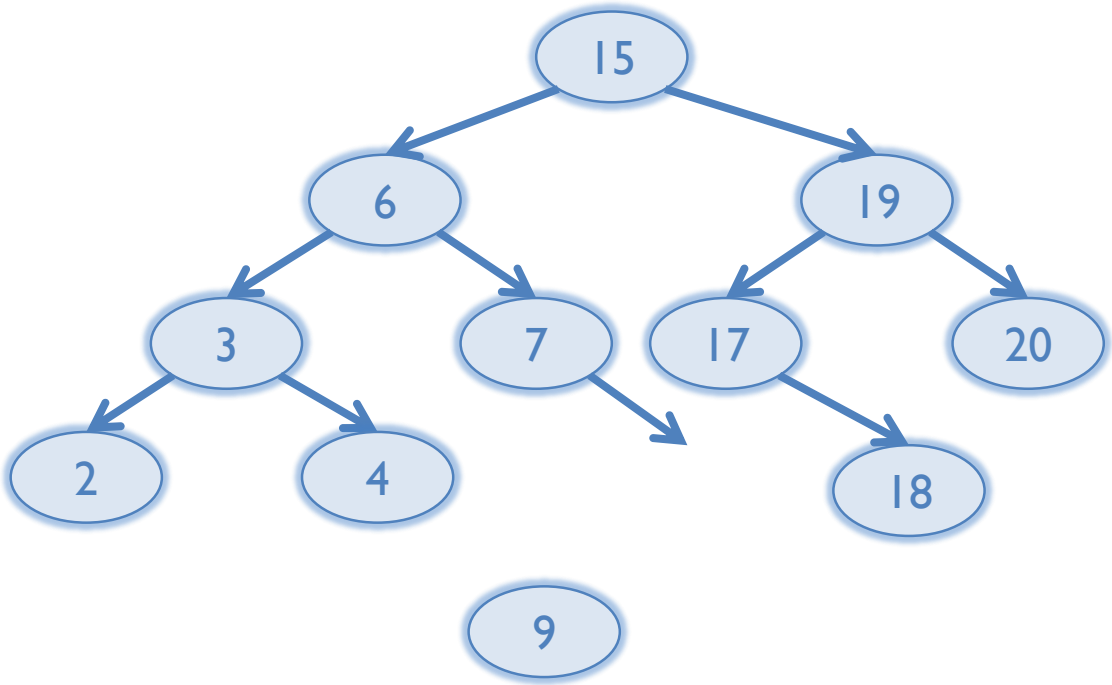
Deletion

- Deleting 13
- Remove it!



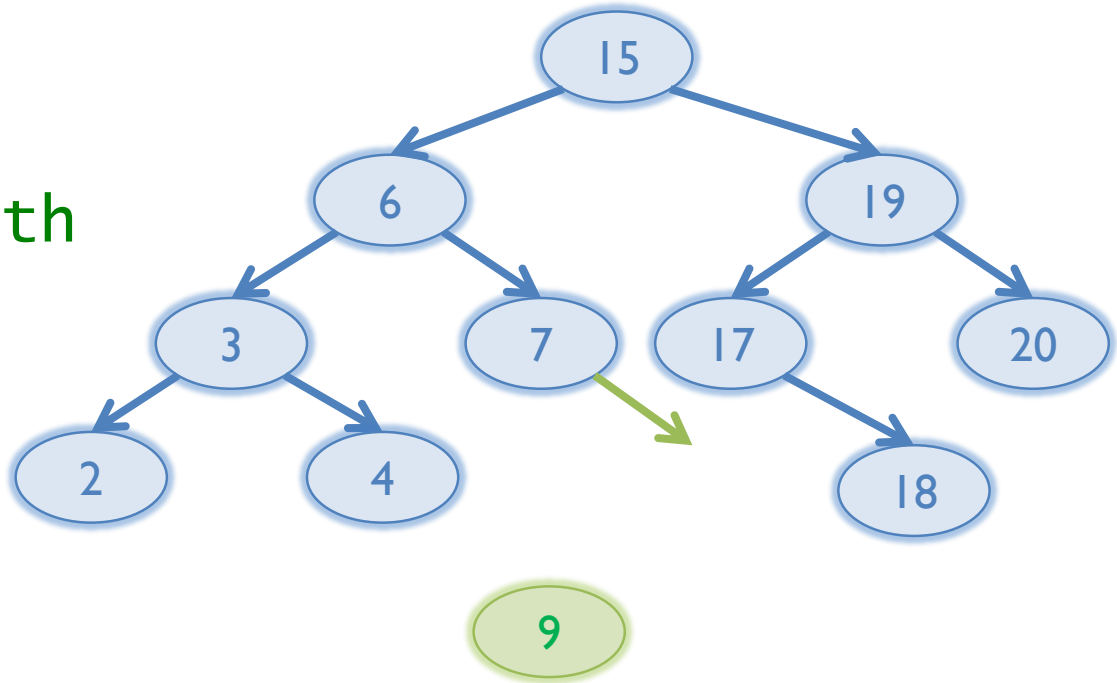
Deletion

- Deleting 13
- Remove it!



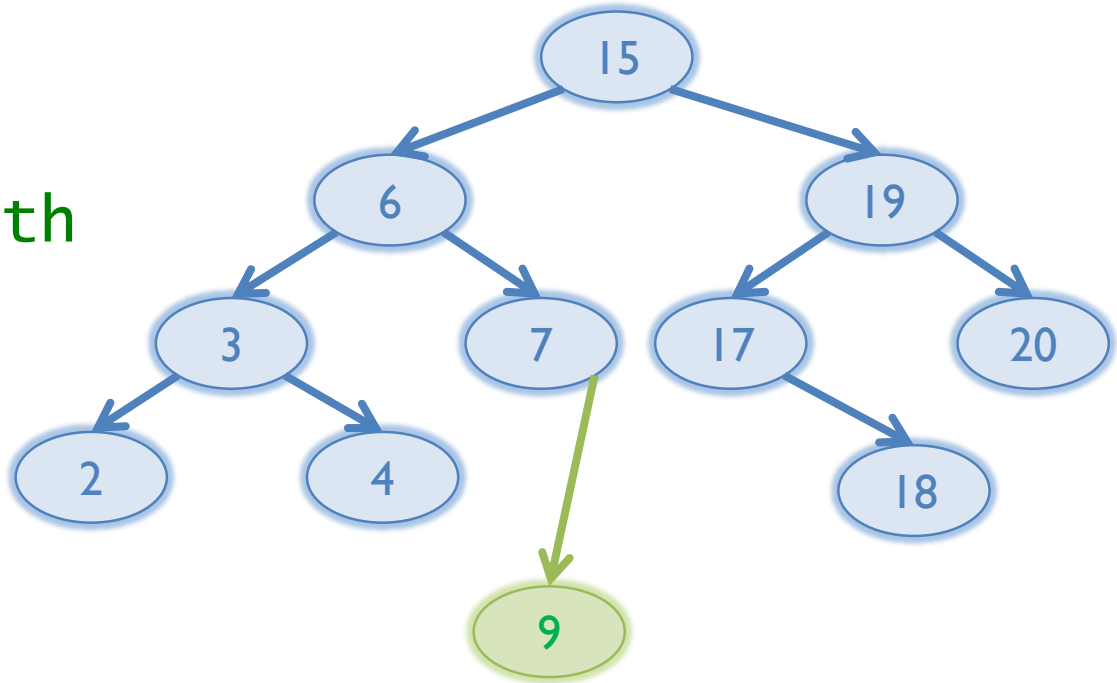
Deletion

- Deleting 13
- Replace it with its child



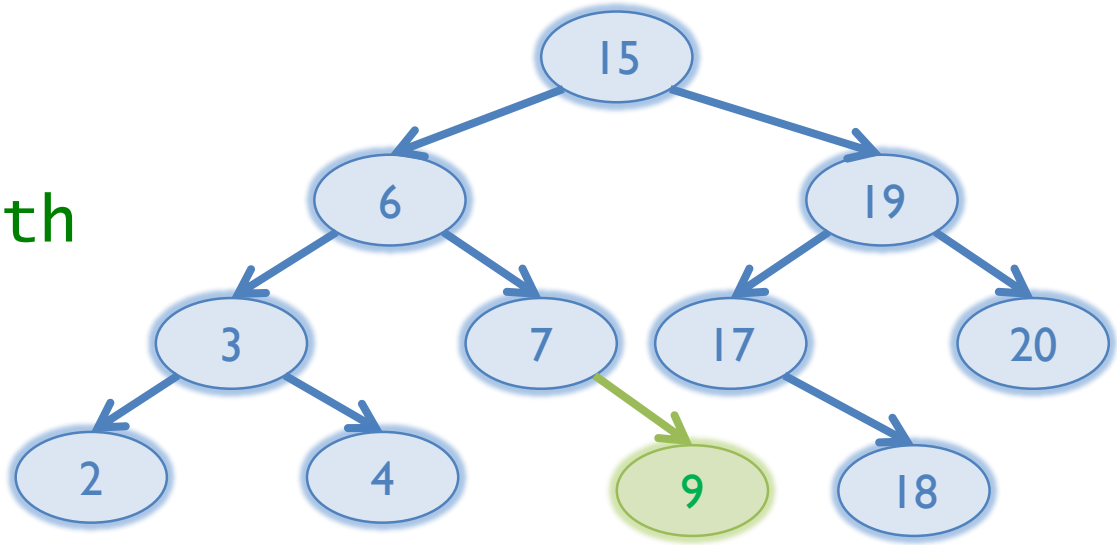
Deletion

- Deleting 13
- Replace it with its child



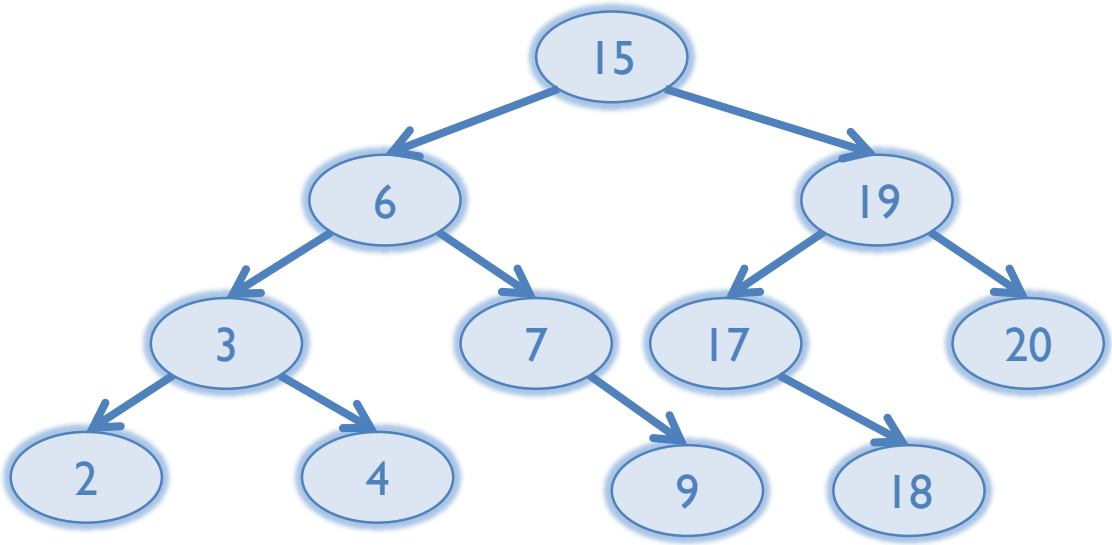
Deletion

- Deleting 13
- Replace it with its child



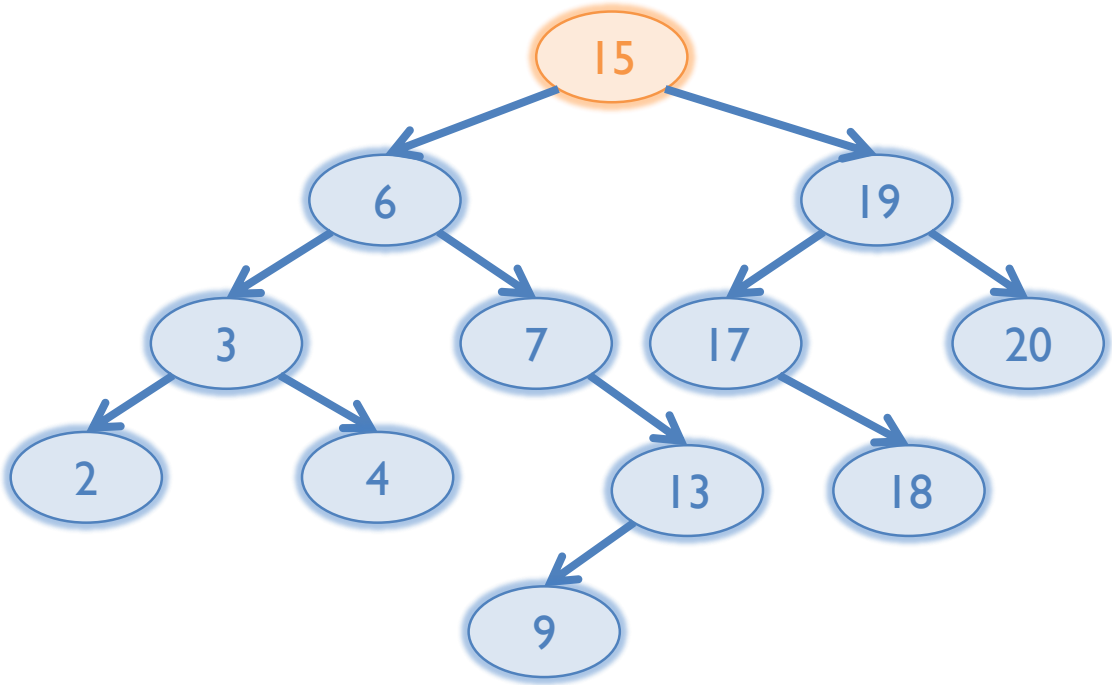
Deletion

- Deleting 13
- Done!



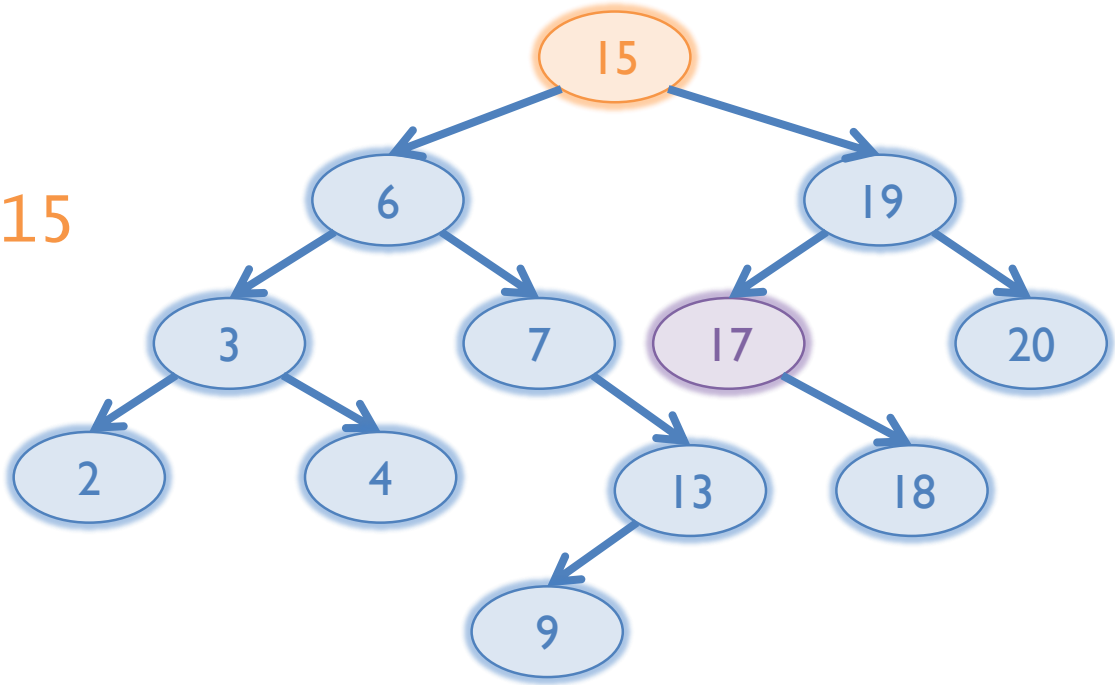
Deletion

- Deleting 15



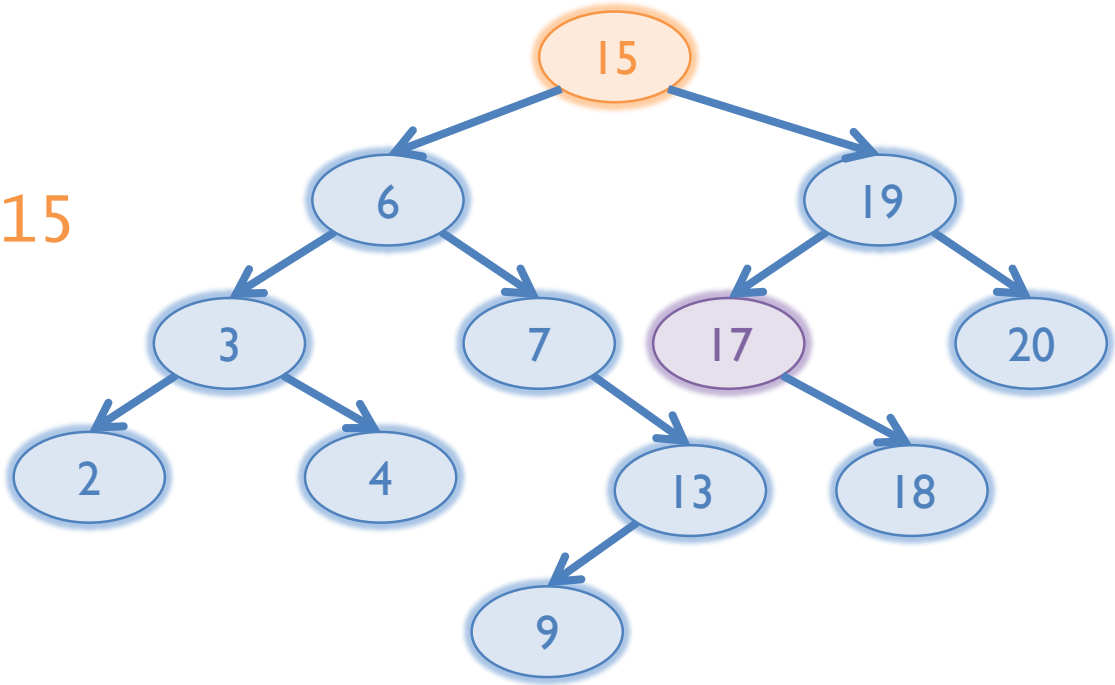
Deletion

- Deleting 15
- Successor of 15 is 17



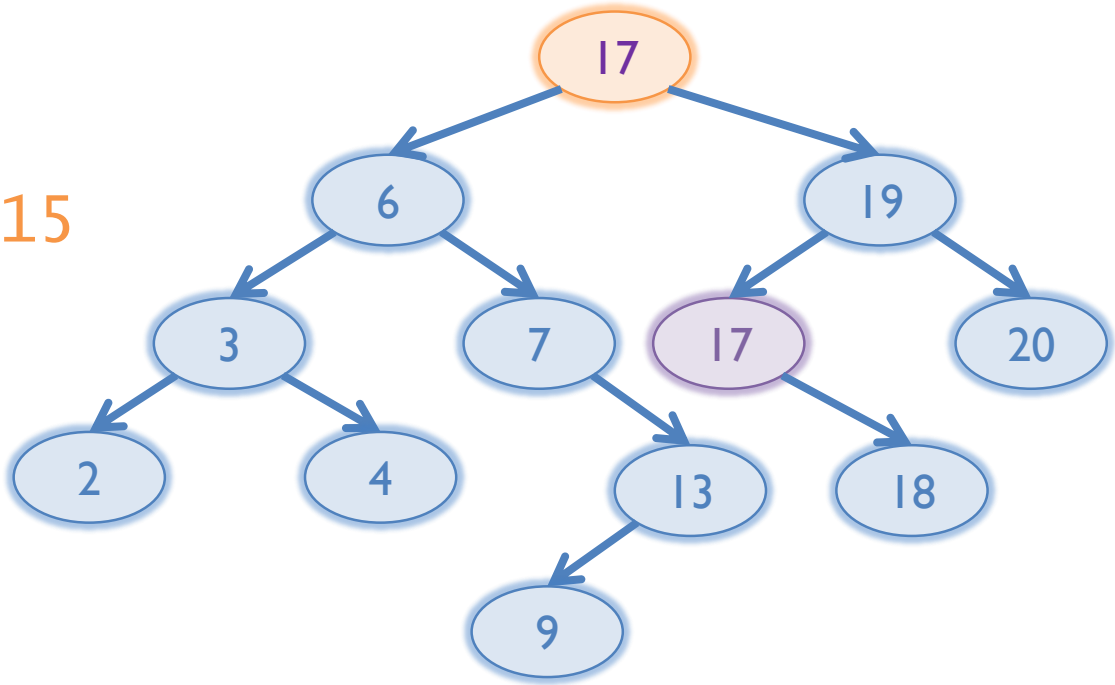
Deletion

- Deleting 15
- Successor of 15 is 17
- Replace!



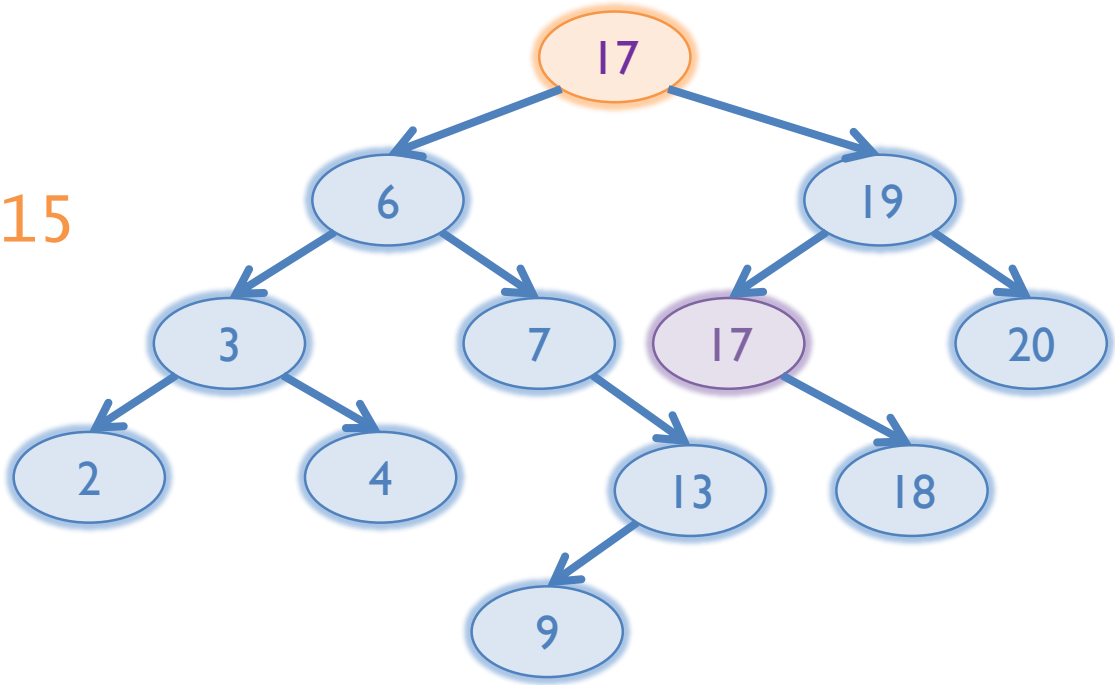
Deletion

- Deleting 15
- Successor of 15 is 17
- Replace!



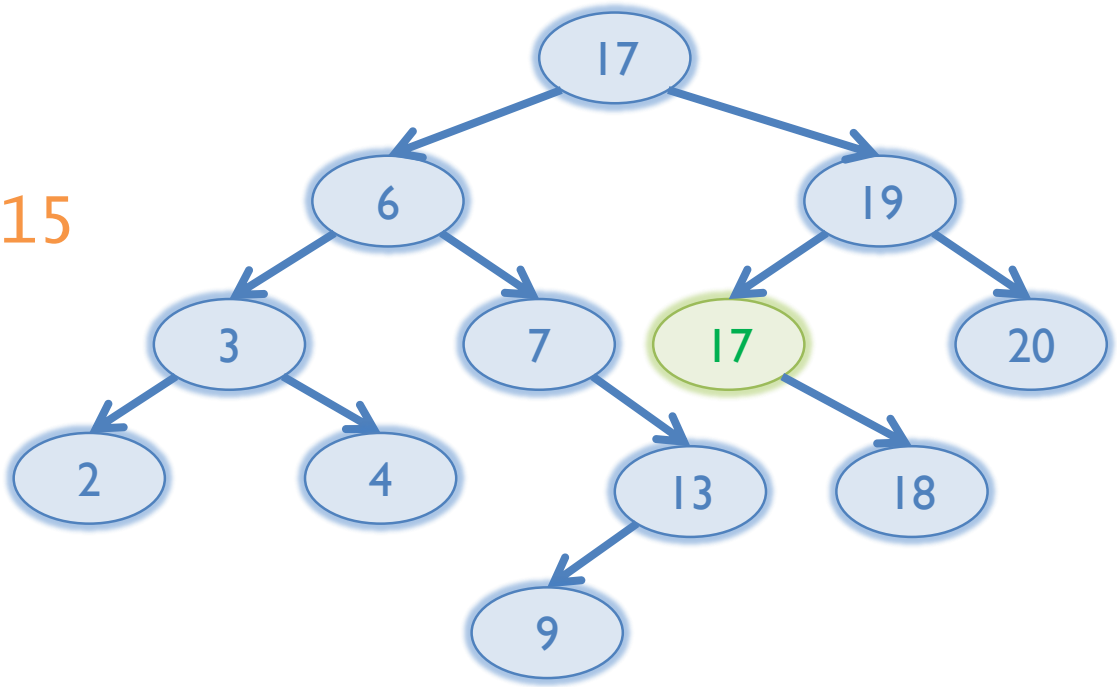
Deletion

- Deleting 15
- Successor of 15 is 17
- Delete R(17)!
- Case two!



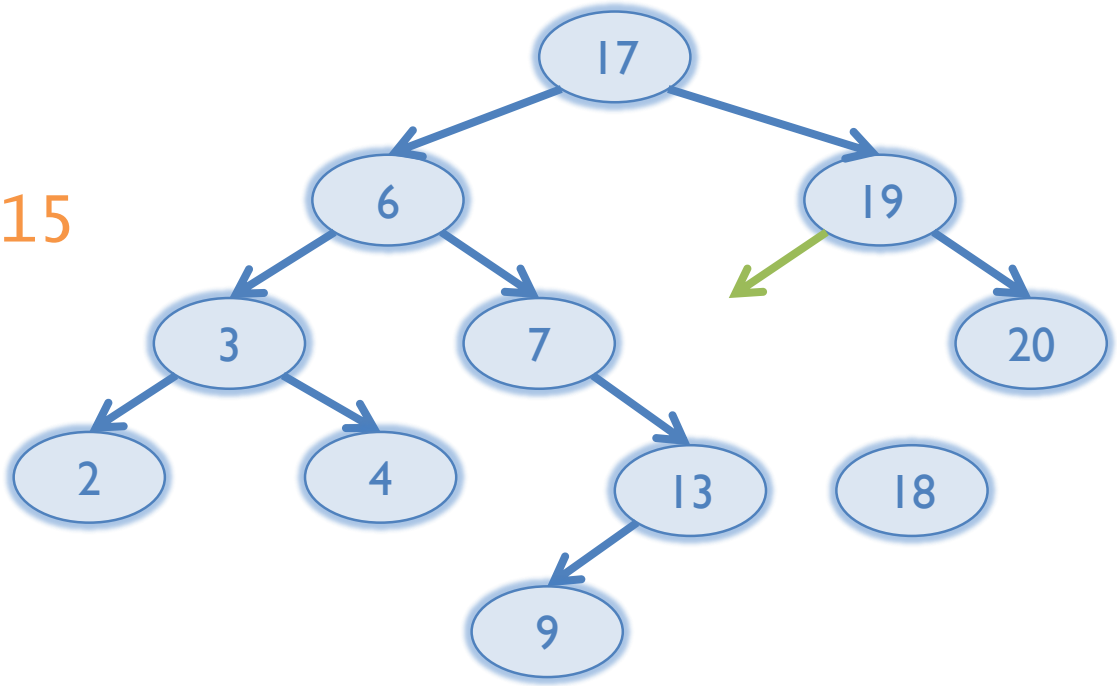
Deletion

- Deleting 15
- Successor of 15 is 17
- Delete R(17)!
- Case two!



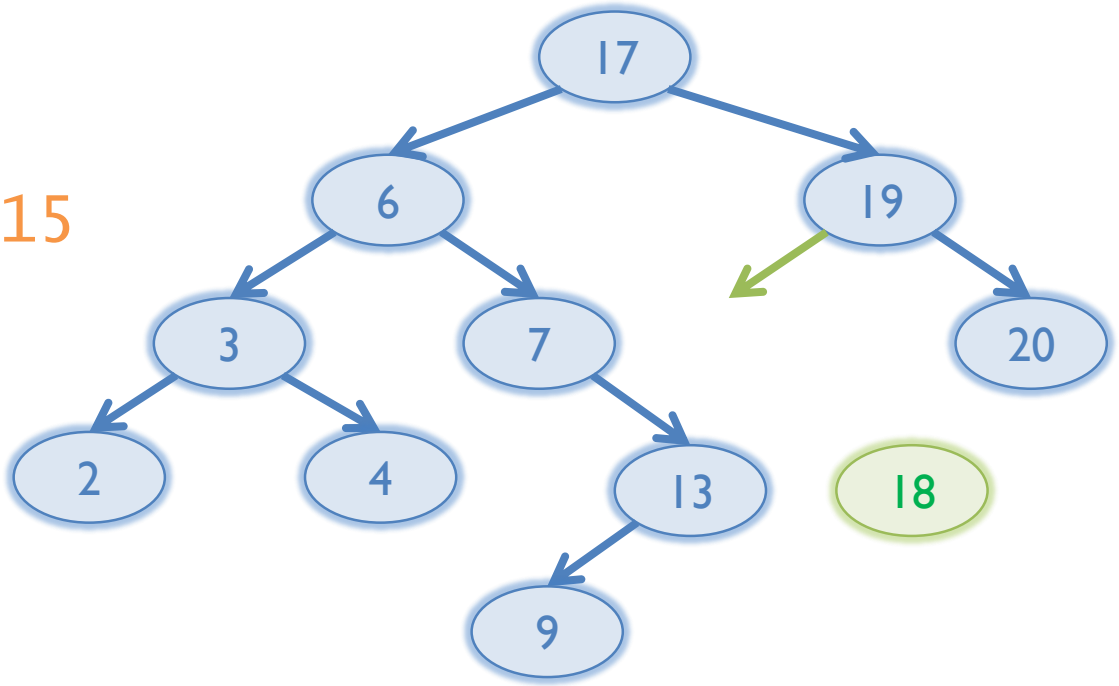
Deletion

- Deleting 15
- Successor of 15 is 17
- Delete R(17)!
- Case two!



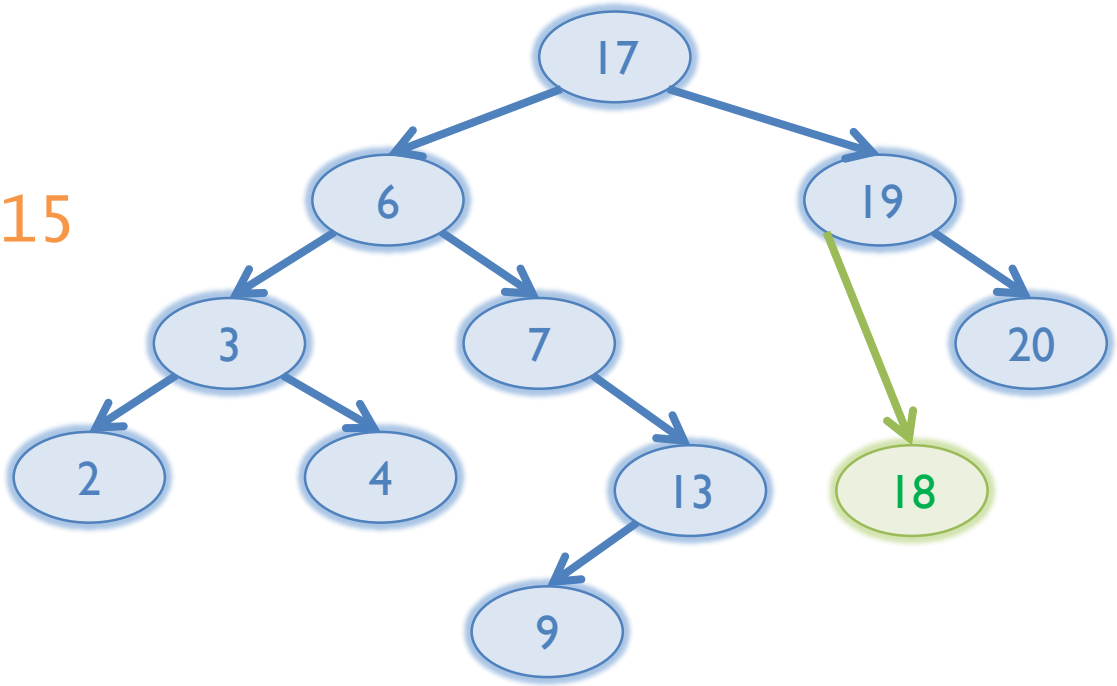
Deletion

- Deleting 15
- Successor of 15 is 17
- Delete R(17)!
- Case two!



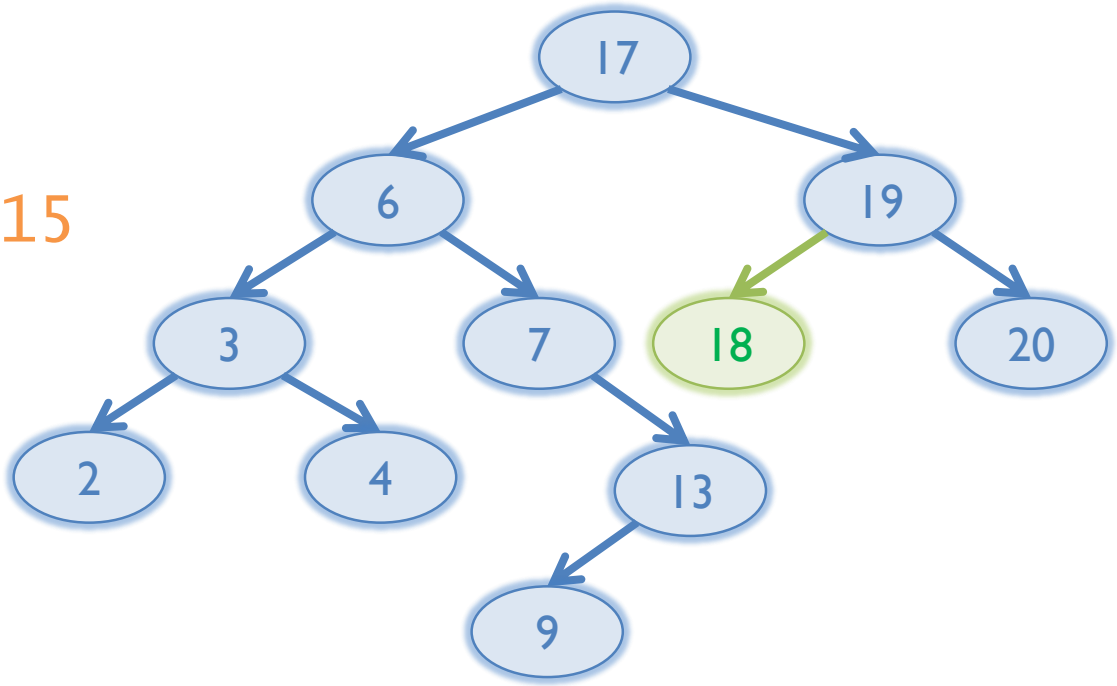
Deletion

- Deleting 15
- Successor of 15 is 17
- Delete R(17)!
- Case two!



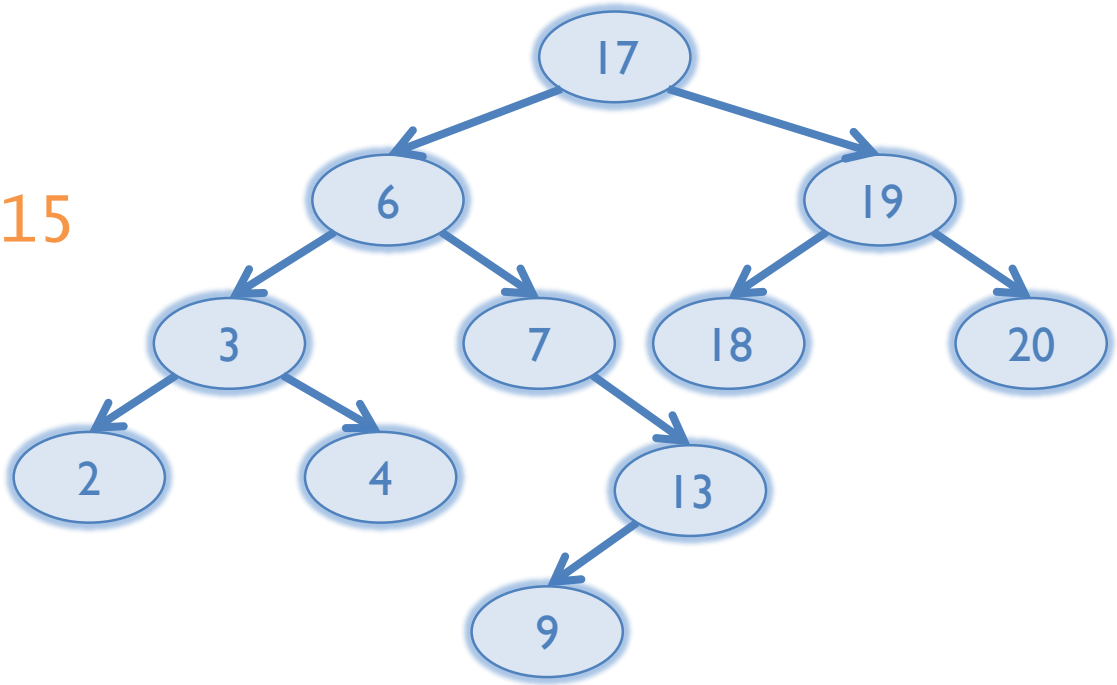
Deletion

- Deleting 15
- Successor of 15 is 17
- Delete R(17)!
- Case two!



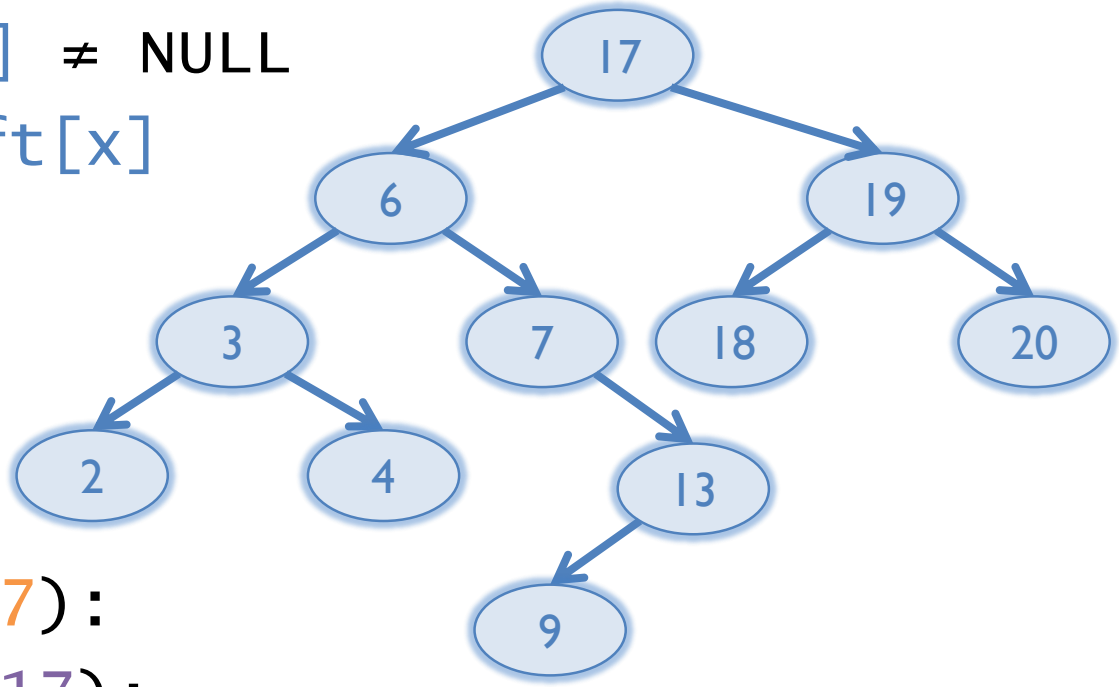
Deletion

- Deleting 15
- Successor of 15 is 17
- Delete R(17)!
- Case two!
- Done!



Deletion

```
Tree-Minimum(x)
1. while left[x] ≠ NULL
2.    do x ← left[x]
3. return x
```

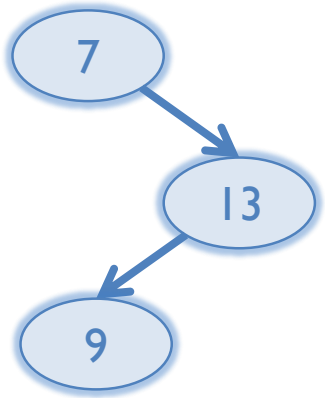


- Example:
- Tree-Minimum(7):
- Tree-Minimum(17):
- Tree-Minimum(19):

Deletion

```
Tree-Minimum(x)
1. while left[x] ≠ NULL
2.    do x ← left[x]
3. return x
```

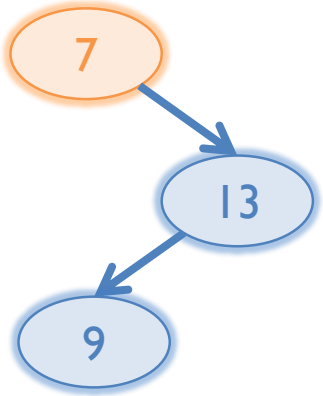
- Example:
- **Tree-Minimum(7):**



Deletion

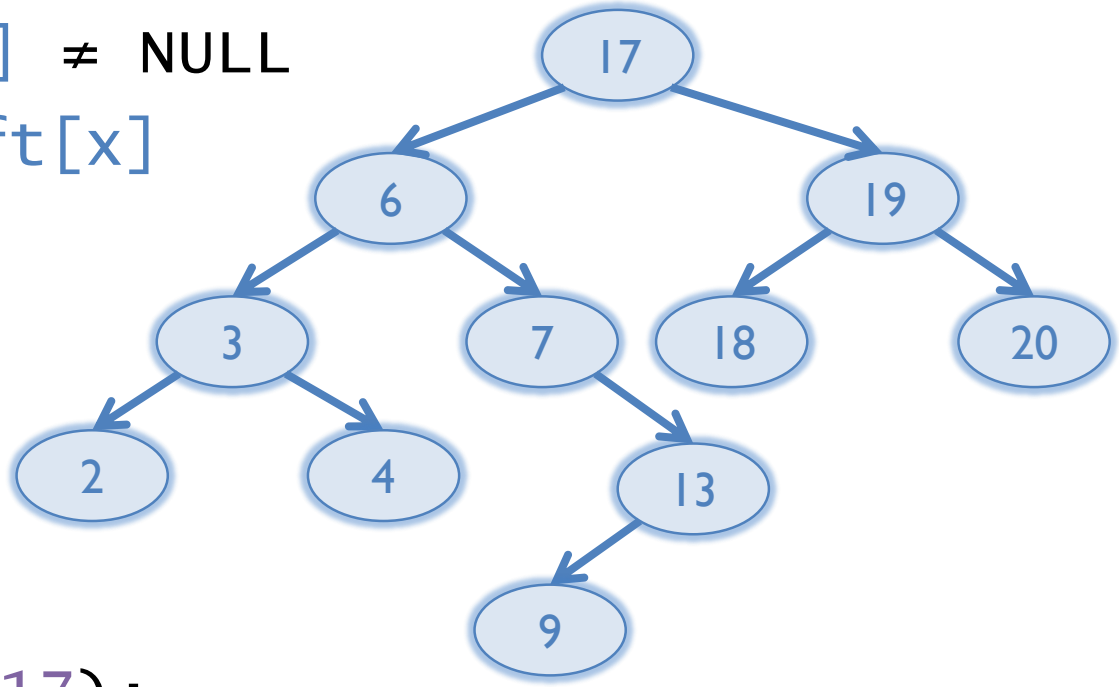
```
Tree-Minimum(x)
1. while left[x] ≠ NULL
2.    do x ← left[x]
3. return x
```

- Example:
- Tree-Minimum(7): 7



Deletion

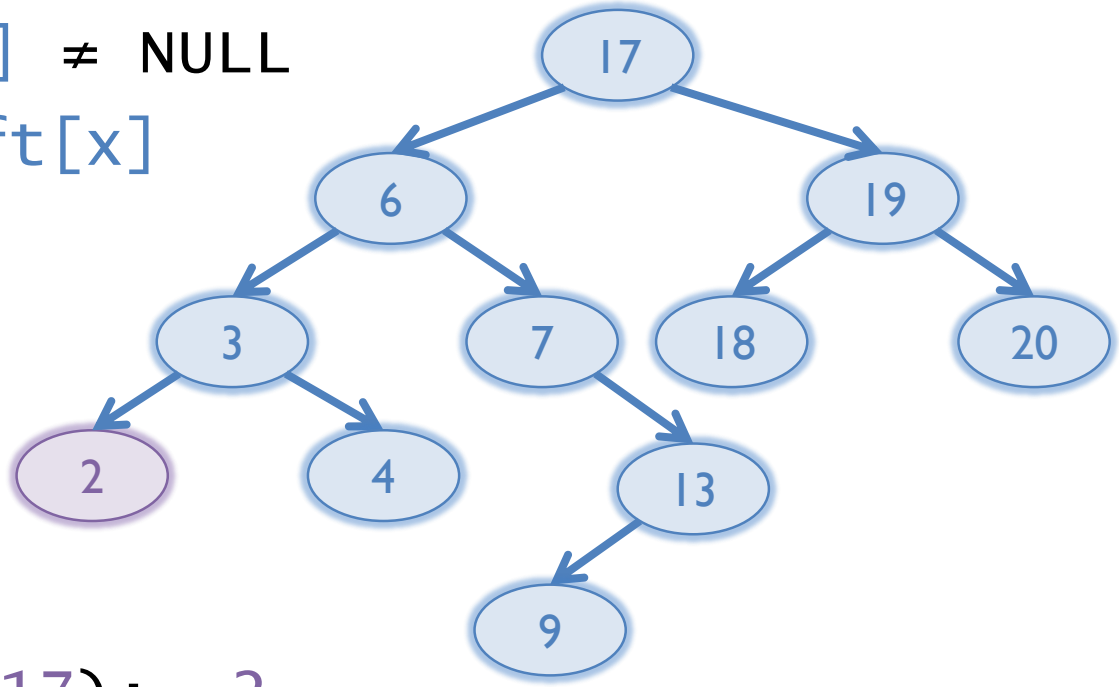
```
Tree-Minimum(x)
1. while left[x] ≠ NULL
2.    do x ← left[x]
3. return x
```



- Example:
-
- Tree-Minimum(17):

Deletion

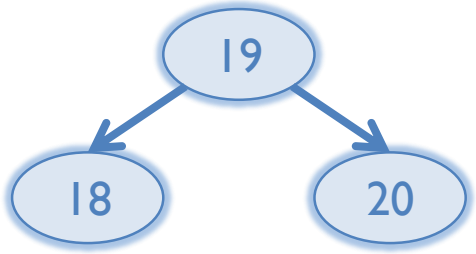
```
Tree-Minimum(x)
1. while left[x] ≠ NULL
2.    do x ← left[x]
3. return x
```



- Example:
-
- **Tree-Minimum(17):** 2

Deletion

```
Tree-Minimum(x)
1. while left[x] ≠ NULL
2.    do x ← left[x]
3. return x
```

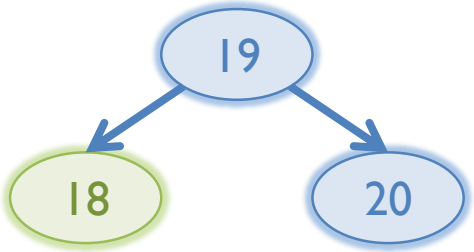


- Example:

- **Tree-Minimum(19):**

Deletion

```
Tree-Minimum(x)
1. while left[x] ≠ NULL
2.    do x ← left[x]
3. return x
```



- Example:

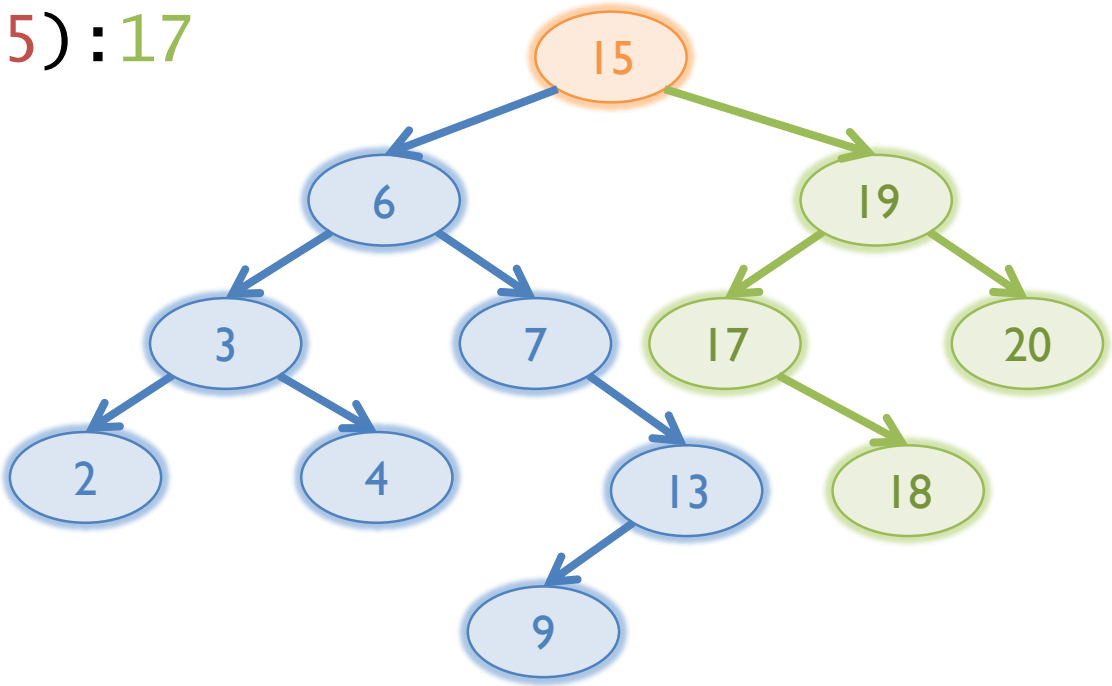
- **Tree-Minimum(19): 18**

Deletion

```
Tree-Successor(x)
1.  if right[x] ≠ NULL
2.     then return Tree-Minimum (right[x])
3.  y ← p[x]
4.  While y ≠ NULL and x = right[y]
5.     do x ← y
6.        y ← p[y]
7.  return y
```

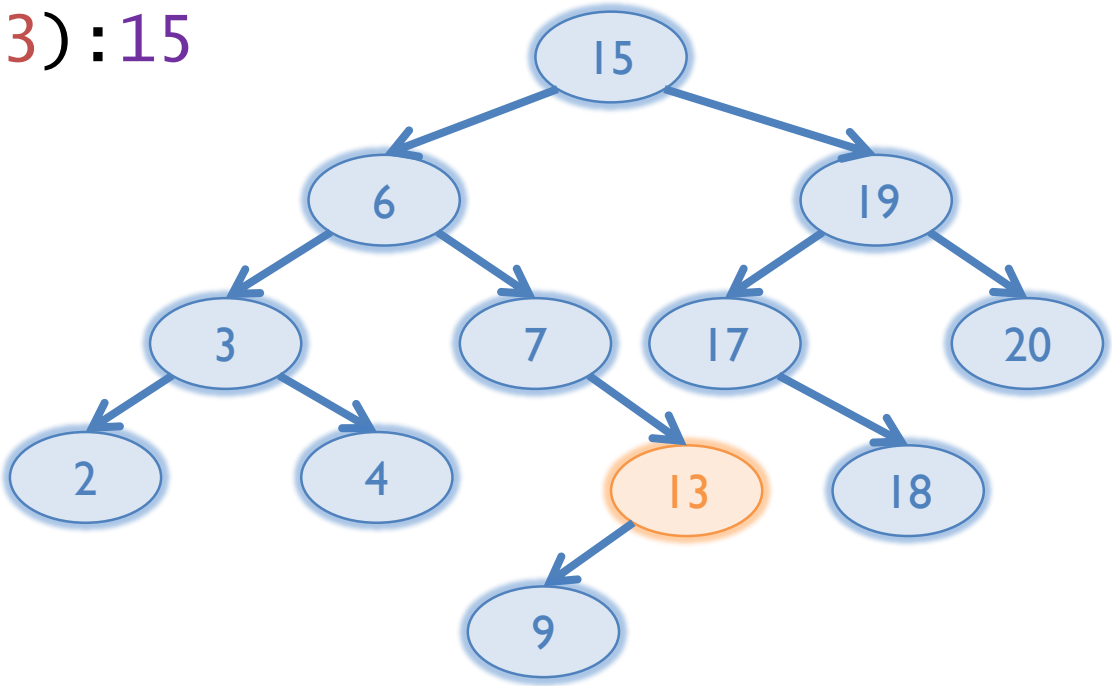
Deletion

`Tree-Successor(15):17`

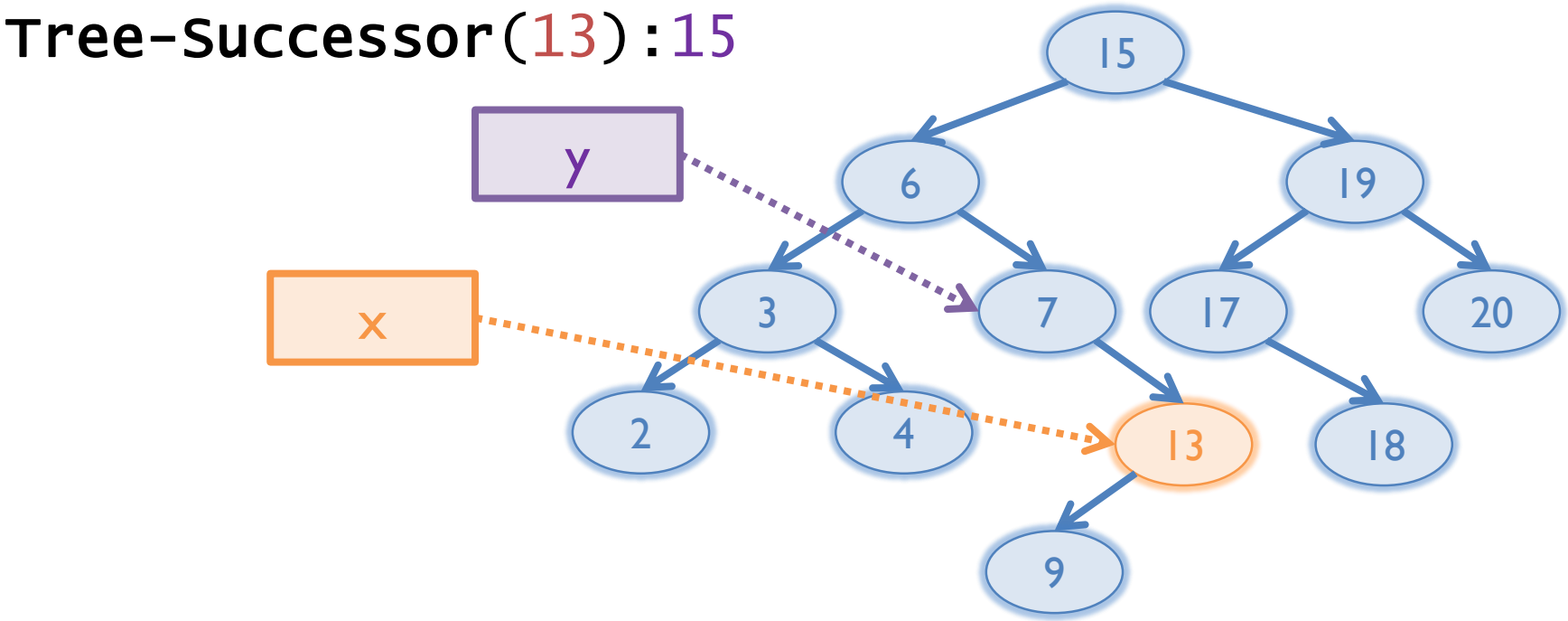


Deletion

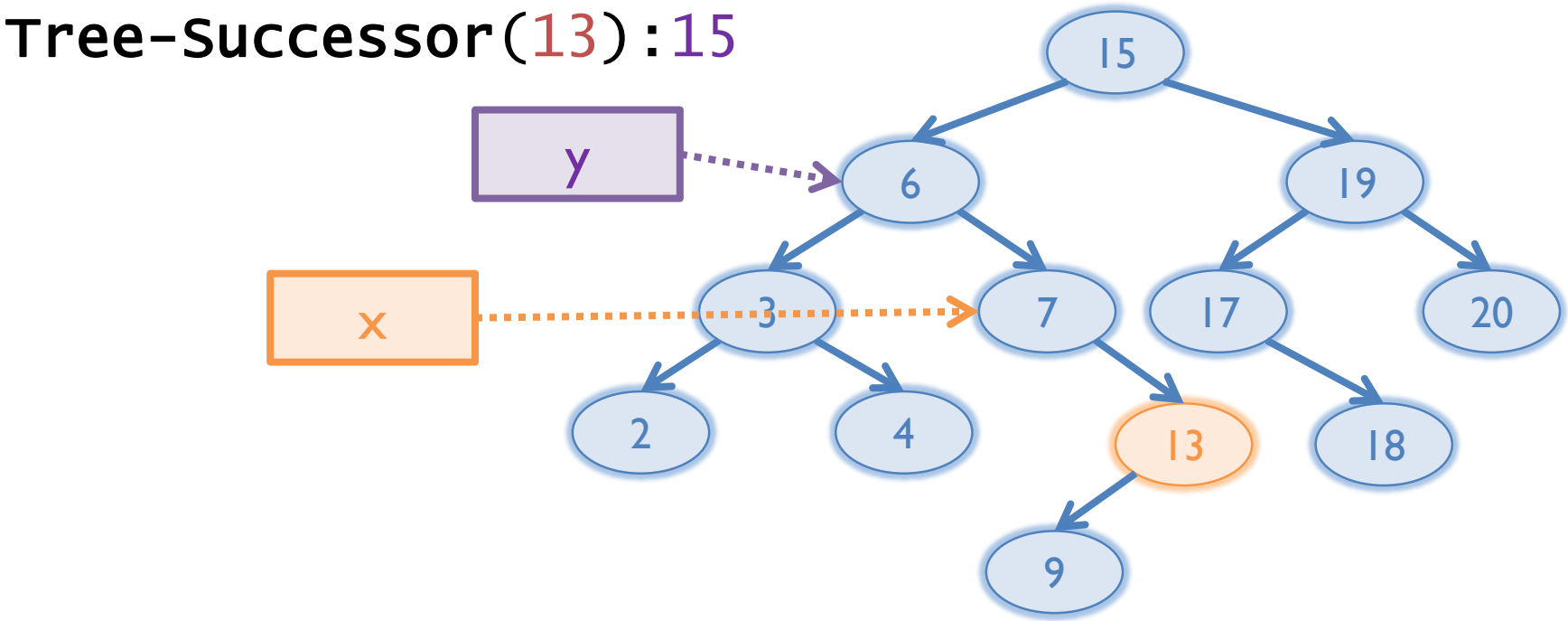
`Tree-Successor(13):15`



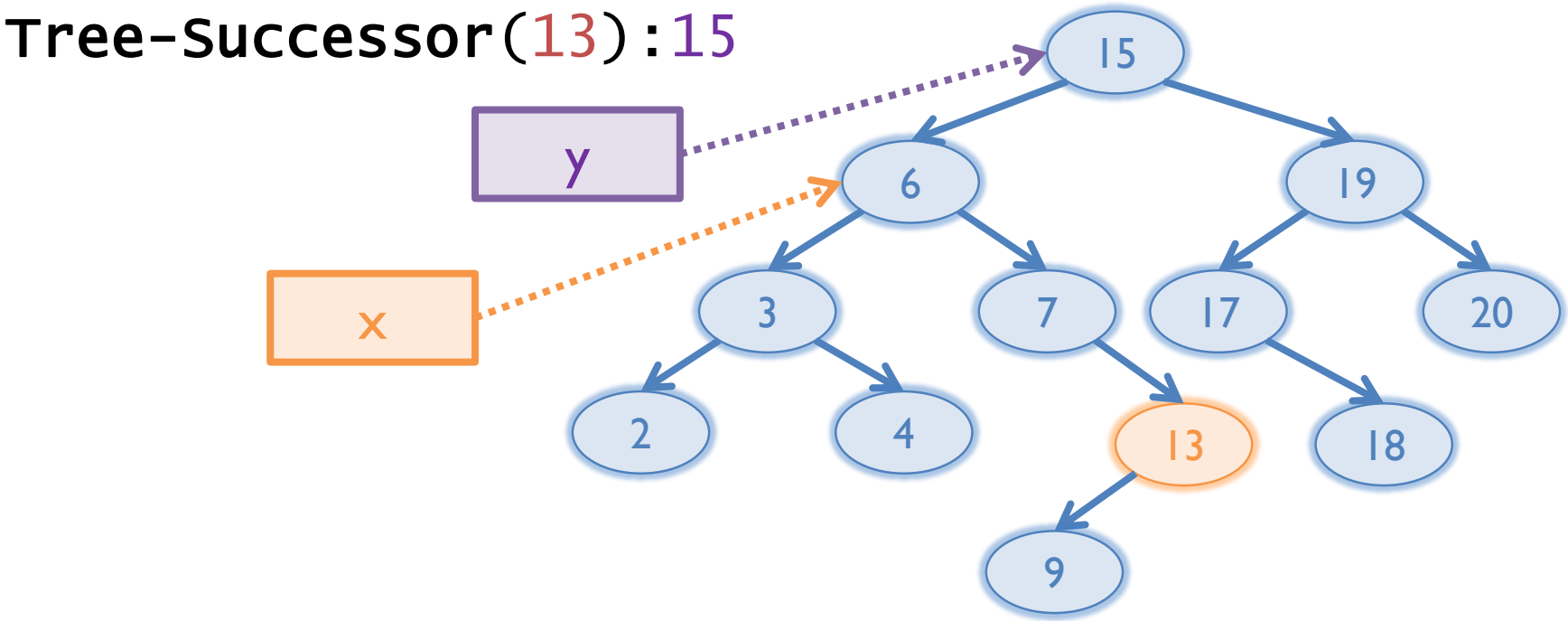
Deletion



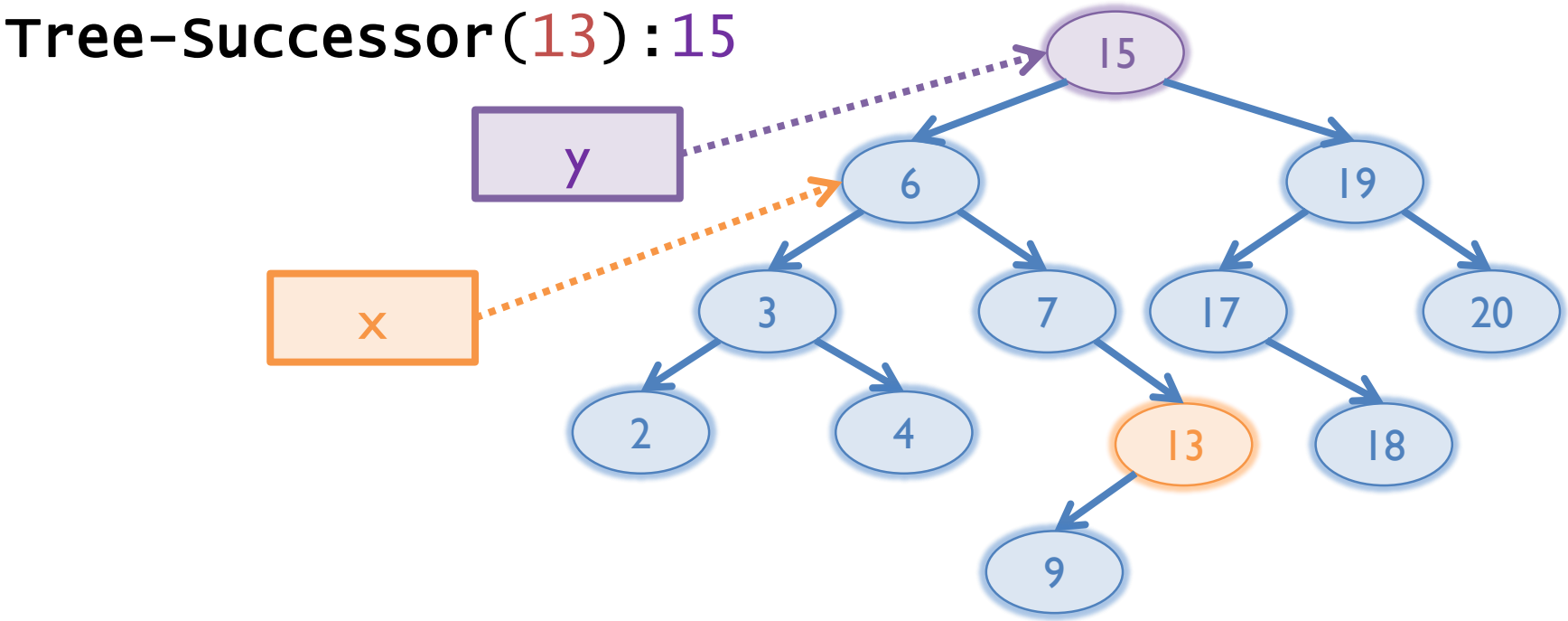
Deletion



Deletion



Deletion



Deletion

- **Deleting** a node on a binary search tree.

- Deleting a leaf: Simply remove it.
- Deleting a node with 1 child: Remove the node and replace it with its child.

- Deleting a node with 2 children: Call the node to be deleted N. Choose its in-order successor or predecessor node, R. Replace the data of N with the data of R, then delete R.

Deletion

```
Replace(N, R)
1. if p[N] ≠ NULL
2.     if N = left[ p[N] ]    /* N is a left  child */
3.         left[ p[N] ] ← R
4.     else                   /* N is a right child */
5.         right[ p[N] ] ← R
6. if R ≠ NULL
7.     p[R] ← p[N]
```

```
Deletion(Tree, N)
1. if right[N] = NULL and left[N] = NULL
2.     Replace(N, NULL)
3. else if left[N] ≠ NULL or right[N] ≠ NULL
4.     Replace(N, left[N] or right[N] )
5. else
6.     R ← Tree-Successor(N)
7.     copy R`s data into N
8.     Deletion(Tree, R)
```

Made By mike199250

Binary Search Tree

- It is so complicated, and it is not self-balancing binary search tree!

Binary Search Tree

- It is so complicated, and it is not self-balancing binary search tree!
- Besides, I am so lazy!

Binary Search Tree

- It is so complicated, and it is not self-balancing binary search tree!
- Besides, I am so lazy!
- Moreover, time is money!

Binary Search Tree

- It is so complicated, and it is not self-balancing binary search tree!
- Besides, I am so lazy!
- Moreover, time is money!
- Don`t worry! Be Happy!
  - C++ STL map, set might fulfill your dream!

Binary Search Tree

- How to use them?

Binary Search Tree

- How to use them?

- Search the usage of insert, erase, iterator, count, begin, end, clear, find, , operator overriding, etc.

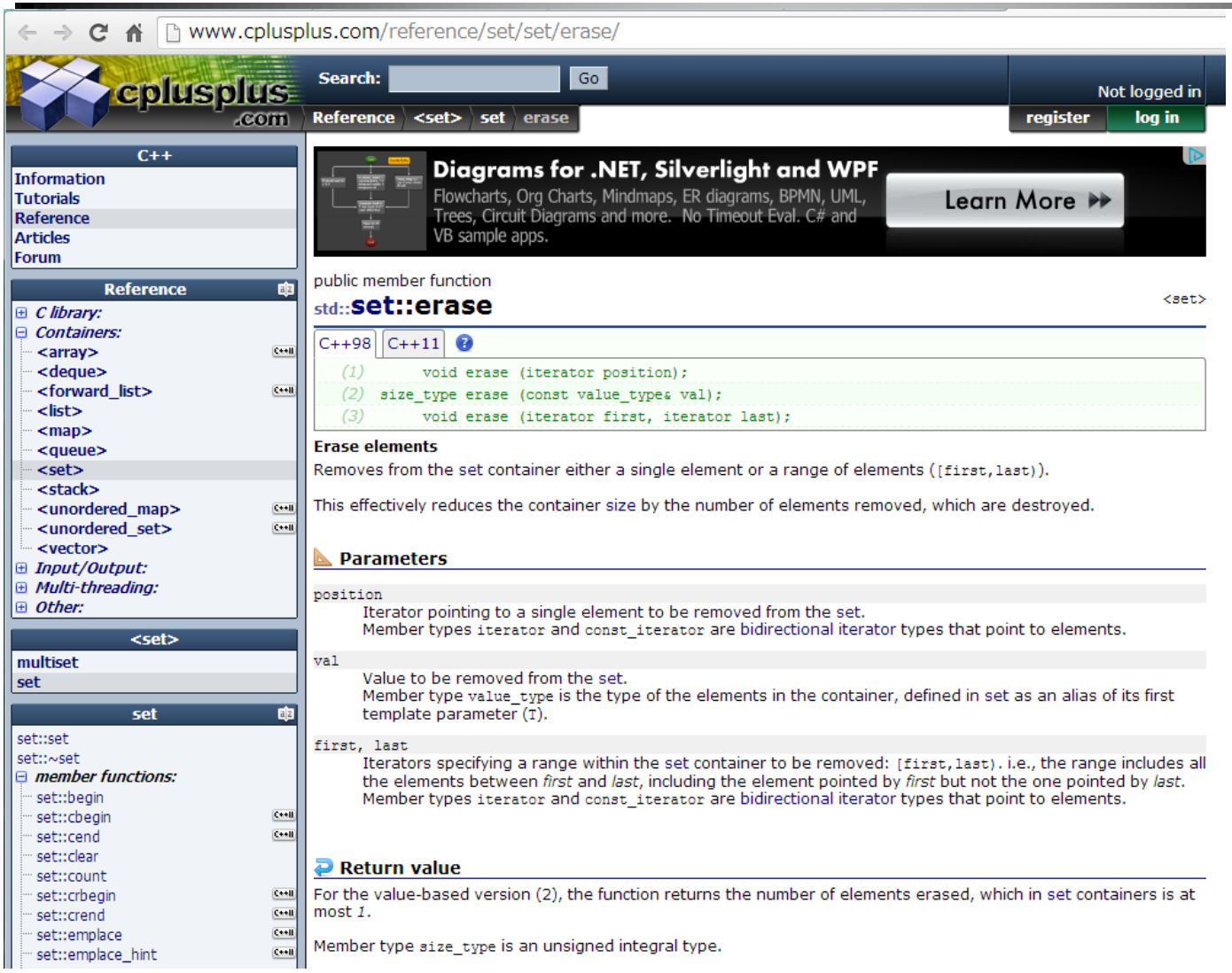
Binary Search Tree

- How to use them?
- Search the usage of insert, erase, iterator, count, begin, end, clear, find, , operator overriding, etc.
- Learn whatever you need.

Binary Search Tree

- How to use them?
- Search the usage of insert, erase, iterator, count, begin, end, clear, find, , operator overriding, etc.
- Learn whatever you need.
- However sometimes you may need your own self-balancing binary search tree……
  - AVL tree, Red-black tree, treap, etc.

Binary Search Tree



Made By mike199250

Binary Search Tree

Example

```
1  // erasing from set
2  #include <iostream>
3  #include <set>
4
5  int main ()
6  {
7    std::set<int> myset;
8    std::set<int>::iterator it;
9
10   // insert some values:
11   for (int i=1; i<10; i++) myset.insert(i*10);  // 10 20 30 40 50 60 70 80 90
12
13   it = myset.begin();
14   ++it;                                         // "it" points now to 20
15
16   myset.erase (it);
17
18   myset.erase (40);
19
20   it = myset.find (60);
21   myset.erase (it, myset.end());
22
23   std::cout << "myset contains:";
24   for (it=myset.begin(); it!=myset.end(); ++it)
25     std::cout << ' ' << *it;
26   std::cout << '\n';
27
28   return 0;
29 }
```

Output:

```
myset contains: 10 30 50
```

Made By mike199250

~Let`s Practice~

Made By mike199250

POJ 3481

- Link: http://poj.org/problem?id=3481

- Think flowing questions:
  1. What do we need?
  2. What can we do?

~Take a Break~

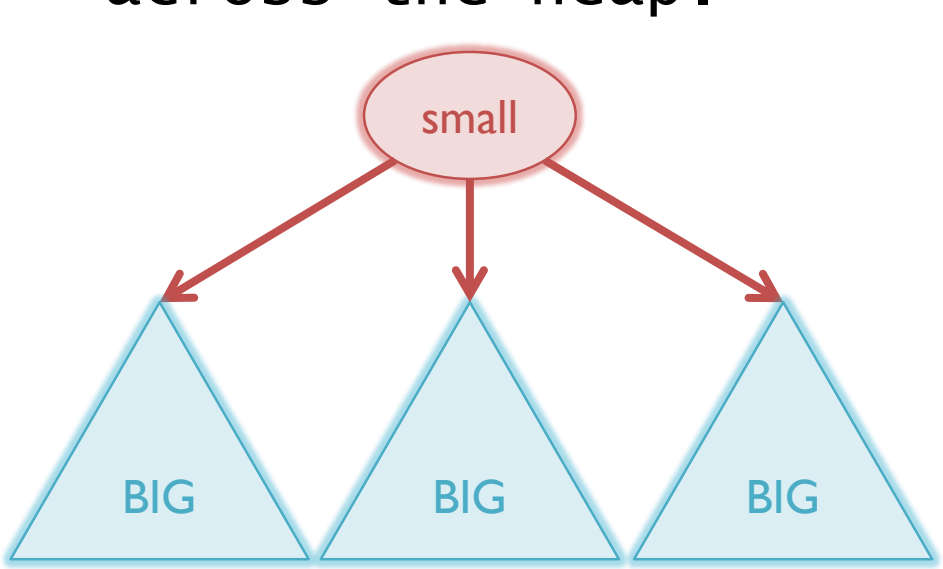
Made By mike199250

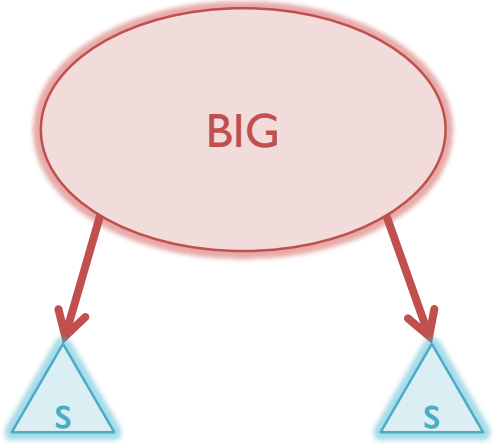
Heap

Made By mike199250

Heap

- A specialized tree-based data structure that satisfies the heap property: If A is a parent node of B then the key of node A is ordered with respect to the key of node B with the same ordering applying across the heap.





Heap

- Some operations:
- **heapify:**
- **find-max or find-min:**
- **delete-max or delete-min**
- **increase-key or decrease-key**
- **insert**
- **merge**

- Different types of heaps implement the operations in different ways.

Heap

- Some Variants:
- Binary heap
- Binomial heap
- Fibonacci heap
- etc.

- Different types of heaps implement the operations in different ways.

Binary Heap

Made By mike199250

Binary Heap

- Using a binary tree.

Binary Heap

- Using a binary tree.
- Shape property: A complete binary tree!

Binary Heap

- Using a binary tree.
- Shape property: A complete binary tree!
- Heap property: All nodes are either ≥ or ≤ each of its children.

Binary Heap

- Using a binary tree.
- Shape property: A complete binary tree!
- Heap property: All nodes are either ≥ or ≤ each of its children.

- max-heaps: ≥ children.
- min-heaps: ≤ children.

Binary Heap

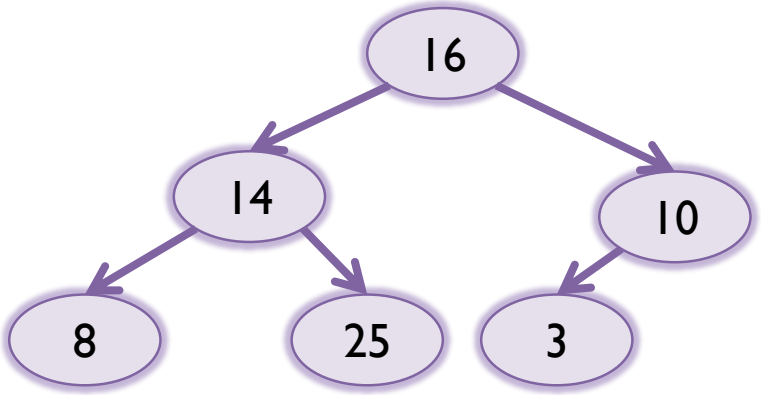
- Using a binary tree.
- Shape property: A complete binary tree!
- Heap property: All nodes are either ≥ or ≤ each of its children.

- max-heaps: ≥ children.
- min-heaps: ≤ children.

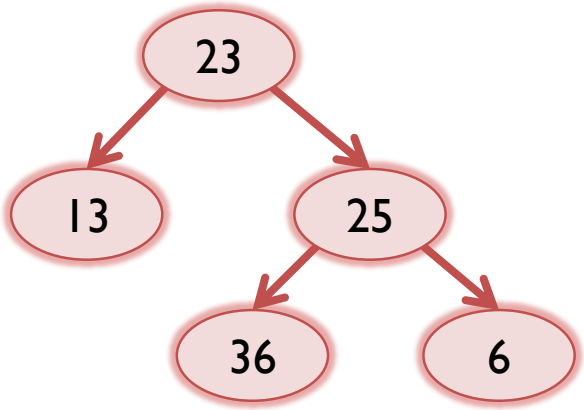
- Ordering of siblings in a heap is not specified!

Complete Binary Tree

- Every level, except the last, is completely filled, and all nodes are as far left as possible.



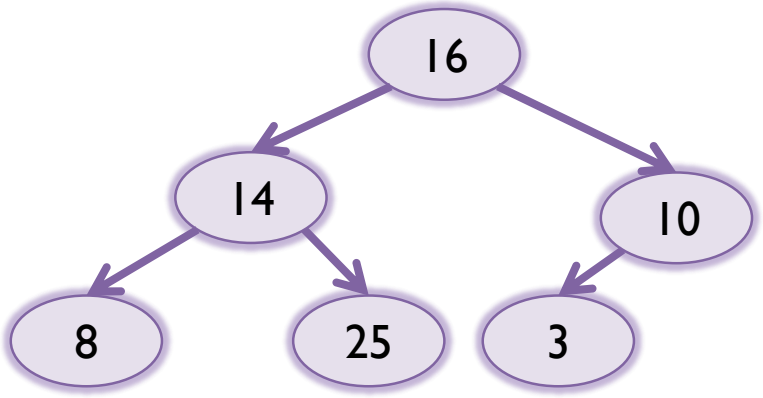
Complete binary tree



Not complete binary tree

Binary Heap

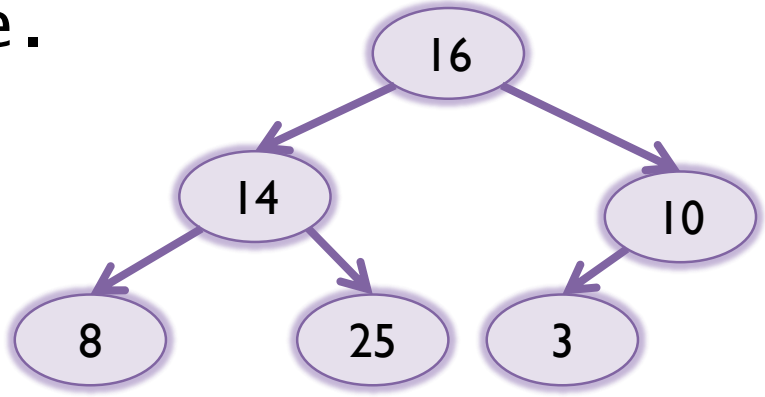
- It is not easy to write tree structure. However, we are more familiar with array.



Binary Heap

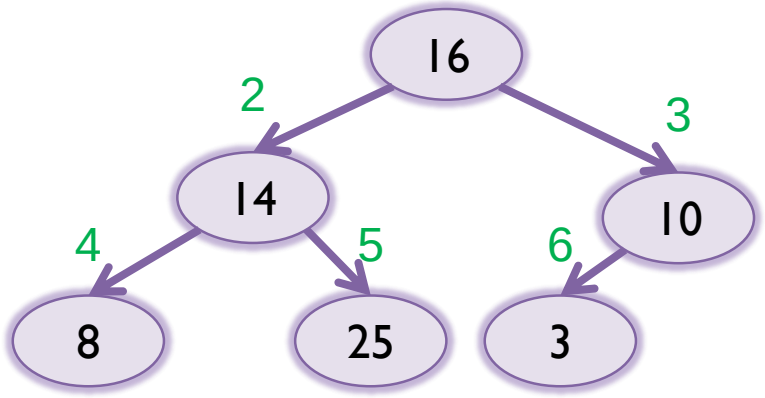
- It is not easy to write tree structure. However, we are more familiar with array.

- Don`t worry! Be Happy!
- We can use the index to represent the node on the tree.



Binary Heap

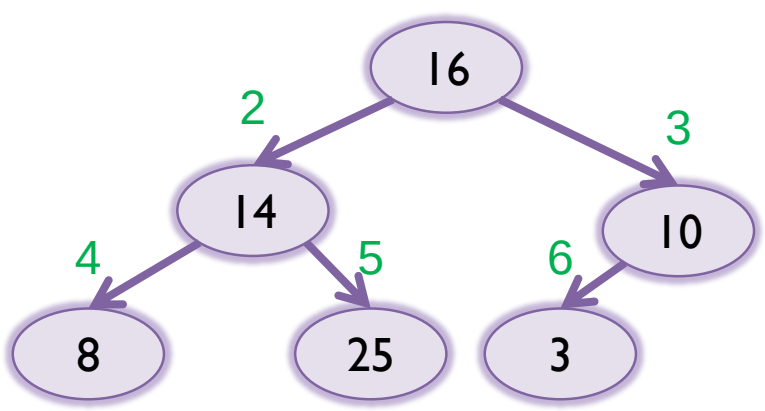
- root is A[1]
- For A[i]
- Left child is A[i*2]
- Right child is A[i*2+1]
- Parent is A[floor(i/2)]



Binary Heap

- length[A]: number of elements in A.
- heap-length[A]: number of elements in the heap stored within A.

- heap-length[A] ≤ length[A]

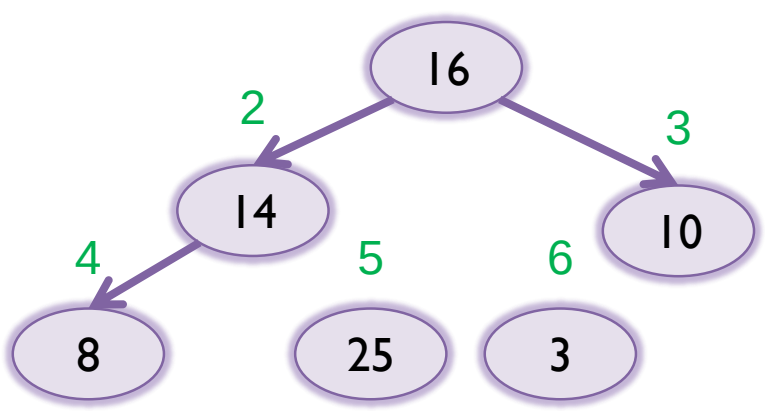


Binary Heap

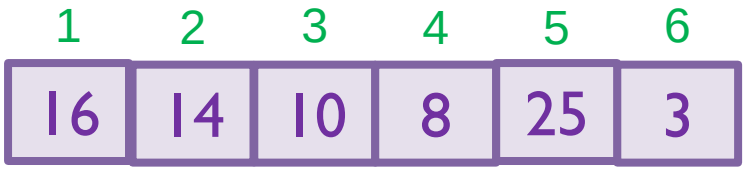
- length[A]: number of elements in A.
- heap-length[A]: number of elements in the heap stored within A.

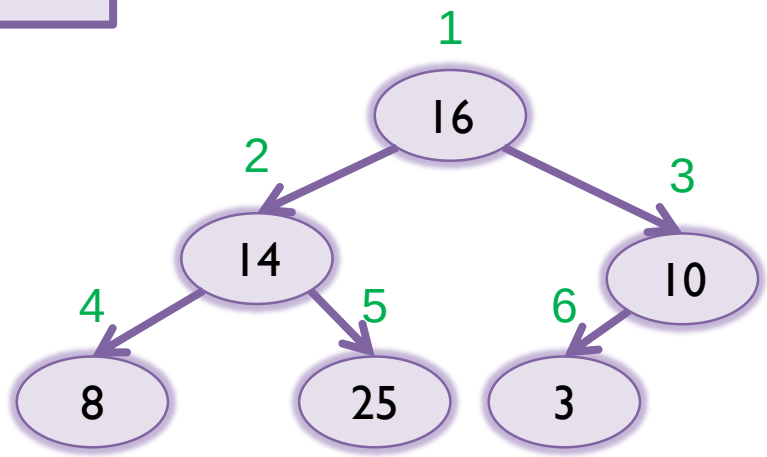
- heap-length[A] ≤ length[A]

- 4 ≤ 6



Binary Heap





Max-Heap

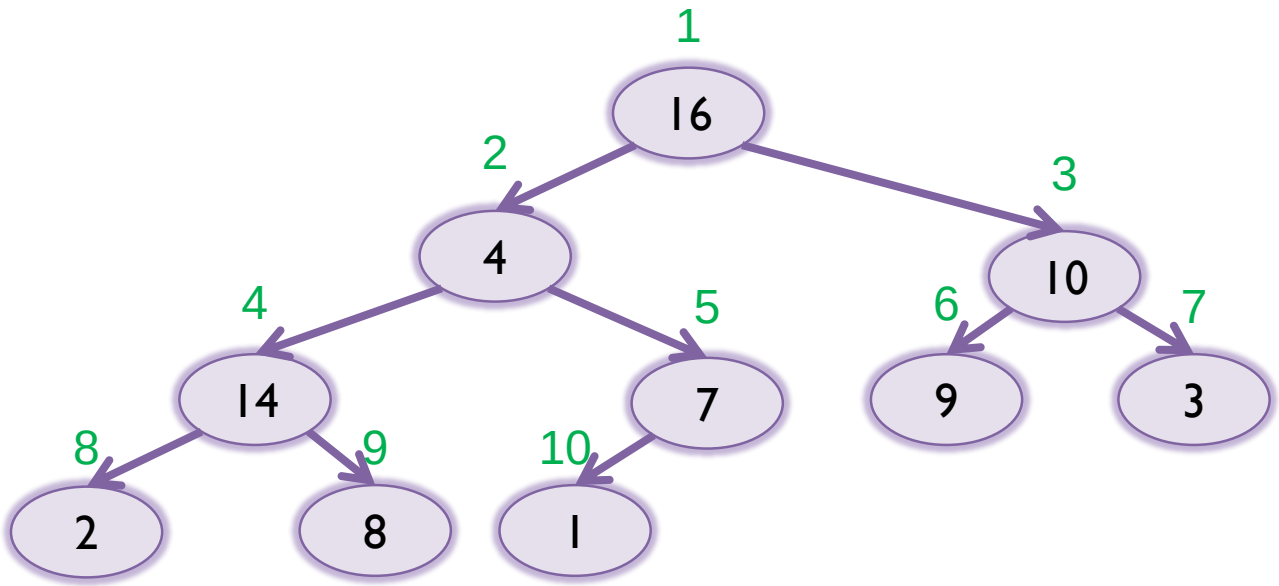
- Assume that sub-trees are heaps, but A[i] may be smaller than its children.

Max-Heap

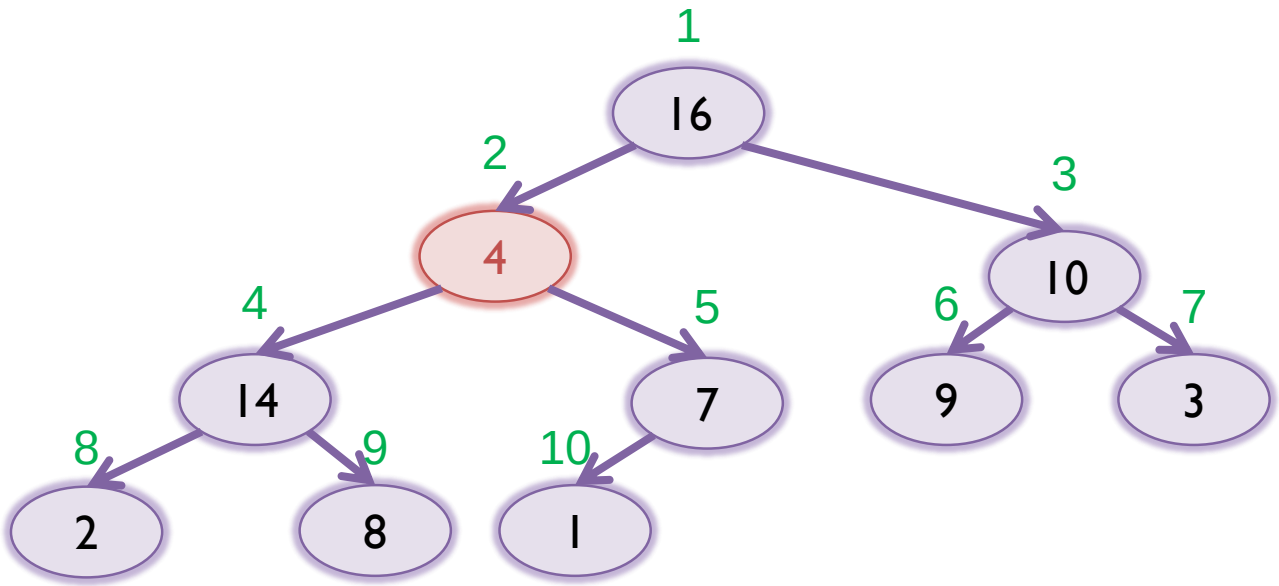
- Assume that sub-trees are heaps, but A[i] may be smaller than its children.

- **Max-Heapify**(A, i): A[i] will down-ward-move, so that the sub-tree rooted at A[i] becomes a heap.

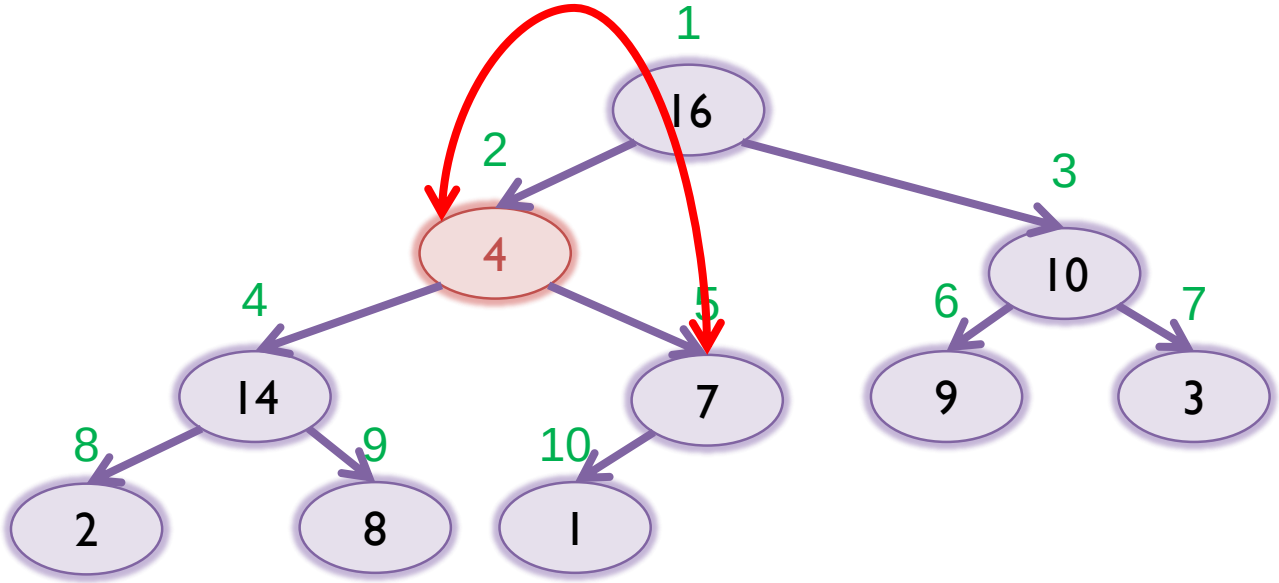
Max-Heap



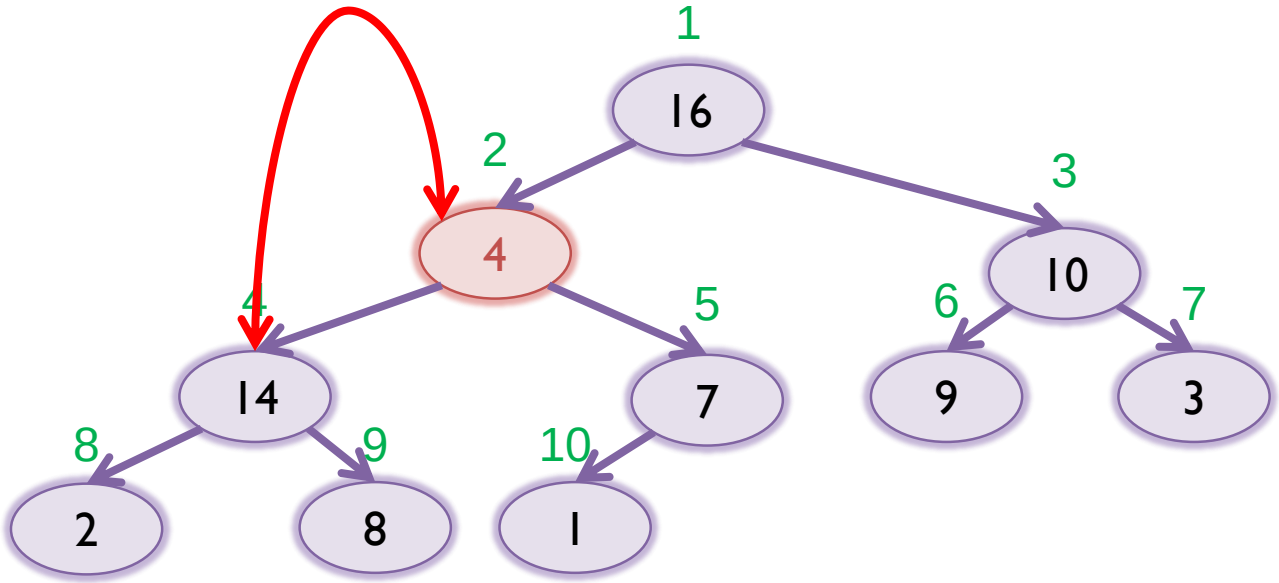
Max-Heap



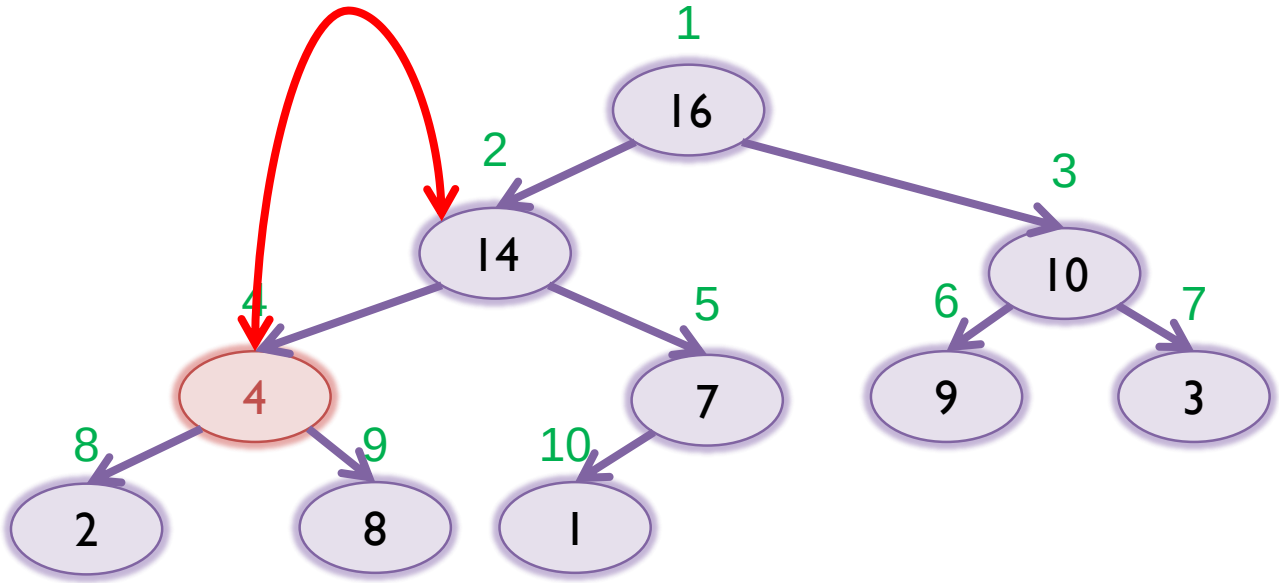
Max-Heap



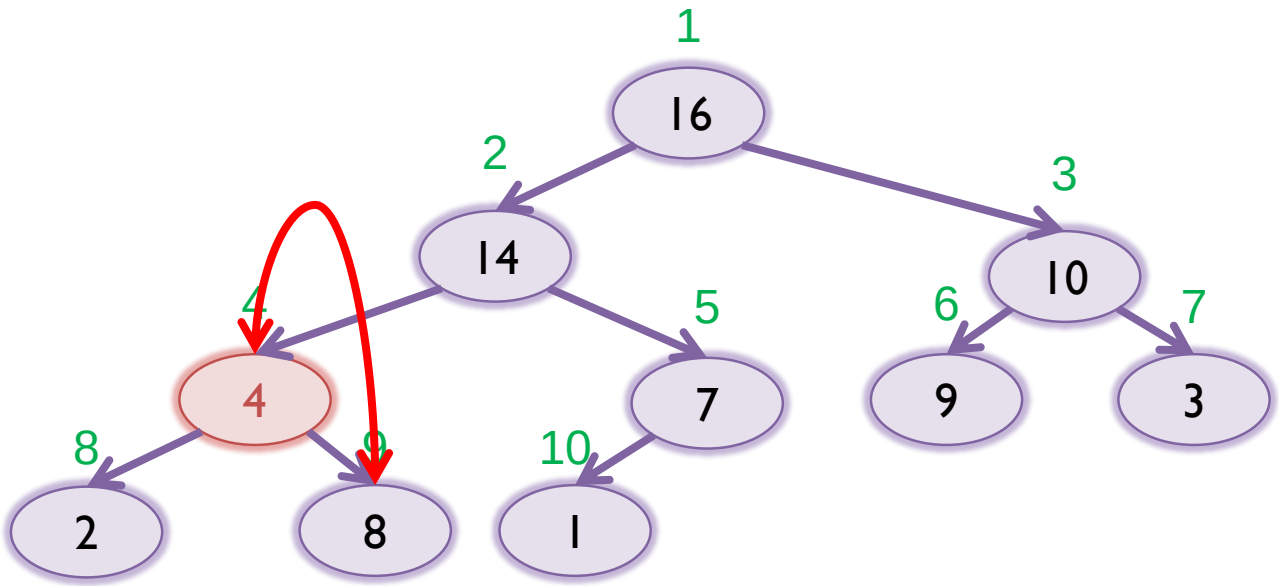
Max-Heap



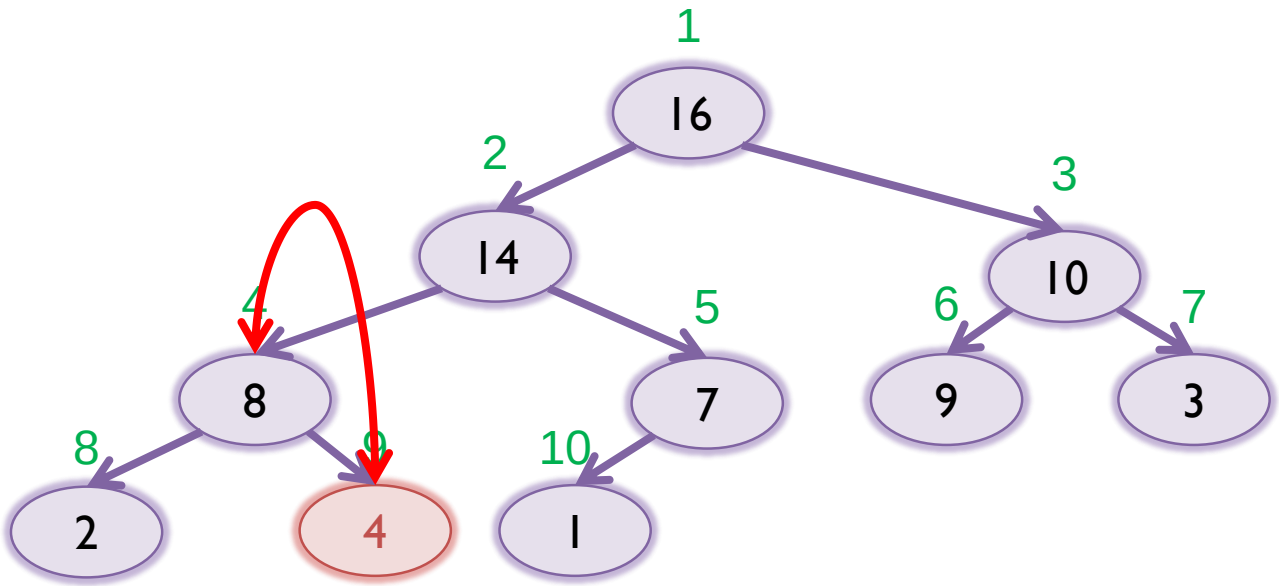
Max-Heap



Max-Heap



Max-Heap



Max-Heap

```
Max-Heapify(A, i)
1. left ← i*2
2. right ← i*2+1
3. largest ← i
4. if left ≤ heap-length[A] and A[left] > A[largest]
5.     largest ← left
6. if right ≤ heap-length[A] and A[right] > A[largest]
7.     largest ← right
8. if largest ≠ i
9.     swap(A[i], A[largest])
10.    Max-heapify(A, largest)
```

Max-Heap

- Time complexity of **Max-Heapify(A, i)**?

Max-Heap

- Time complexity of **Max-Heapify(A, i)**?

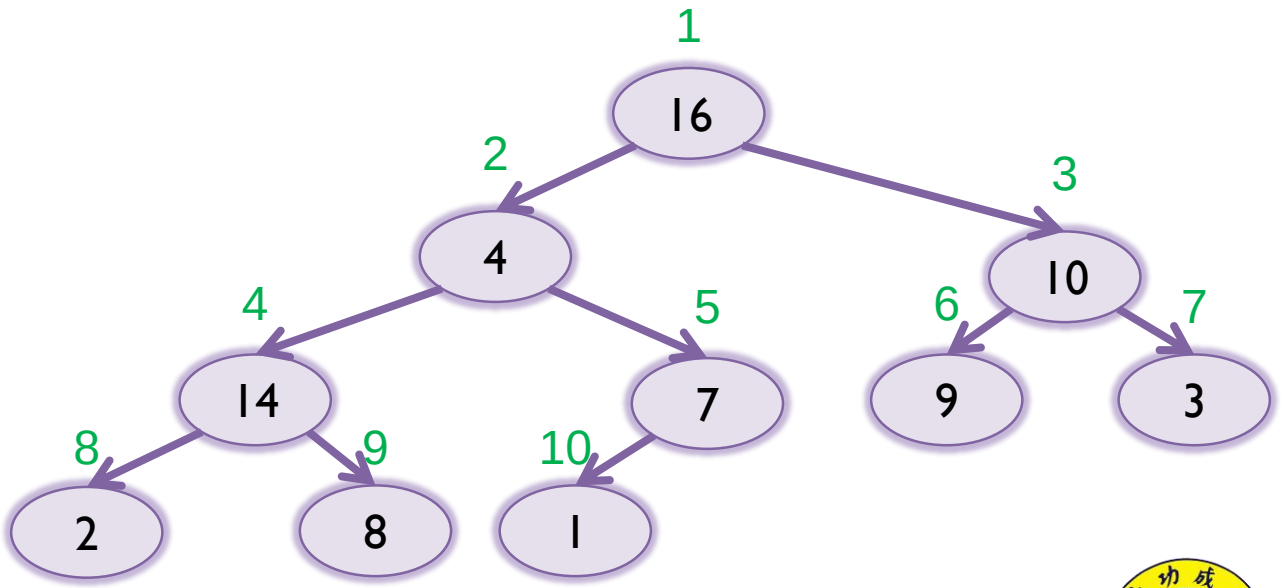
- In the worst case, the node has to be swapped with its child on each level until it reaches the bottom level.

Max-Heap

- Time complexity of **Max-Heapify(A, i)**?
- In the worst case, the node has to be swapped with its child on each level until it reaches the bottom level.
- So the time complexity relative to the height of the tree. O(*h*) or O(log N)

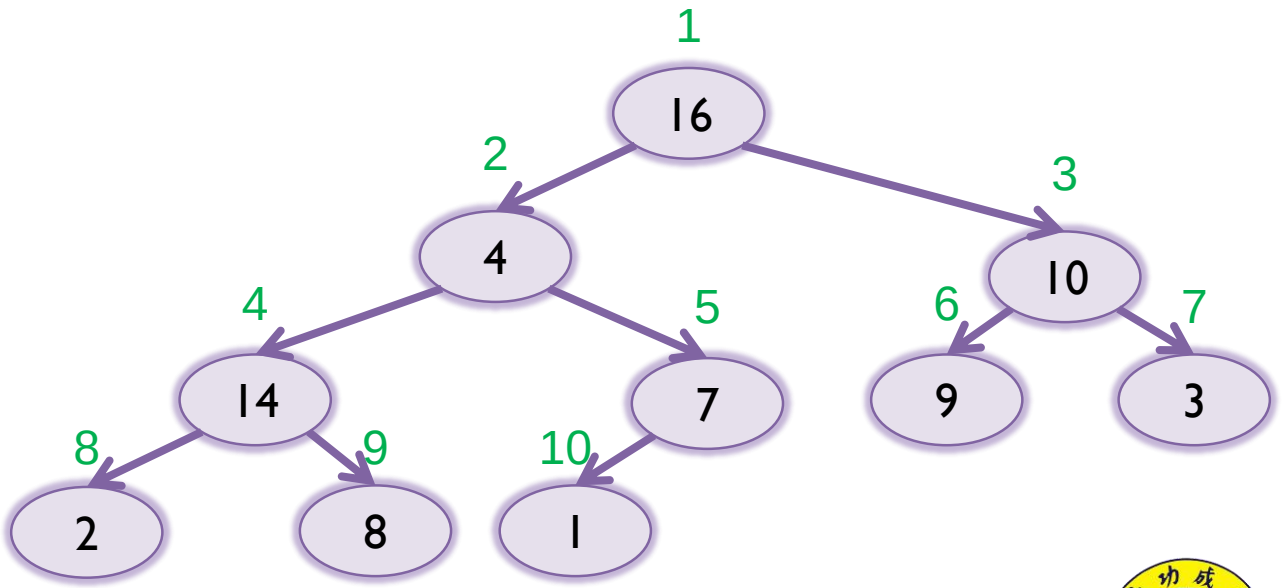
Max-Heap

- How to build the entire heap?



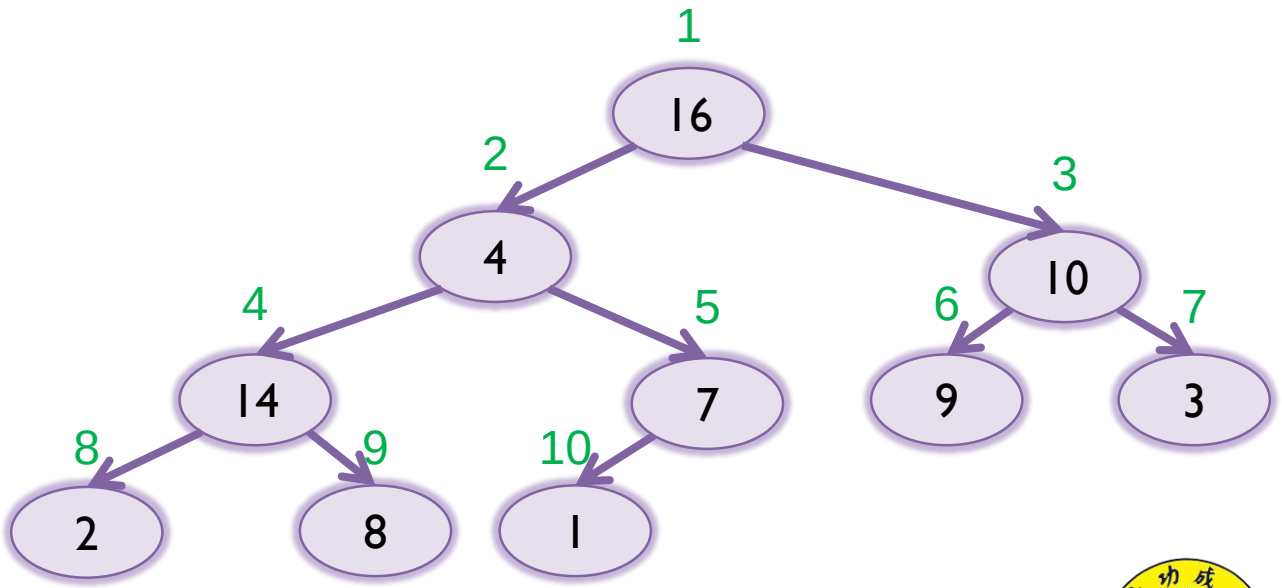
Max-Heap

- How to build the entire heap?
- Recursively build the sub-trees first, then do **max-heapify**?



Max-Heap

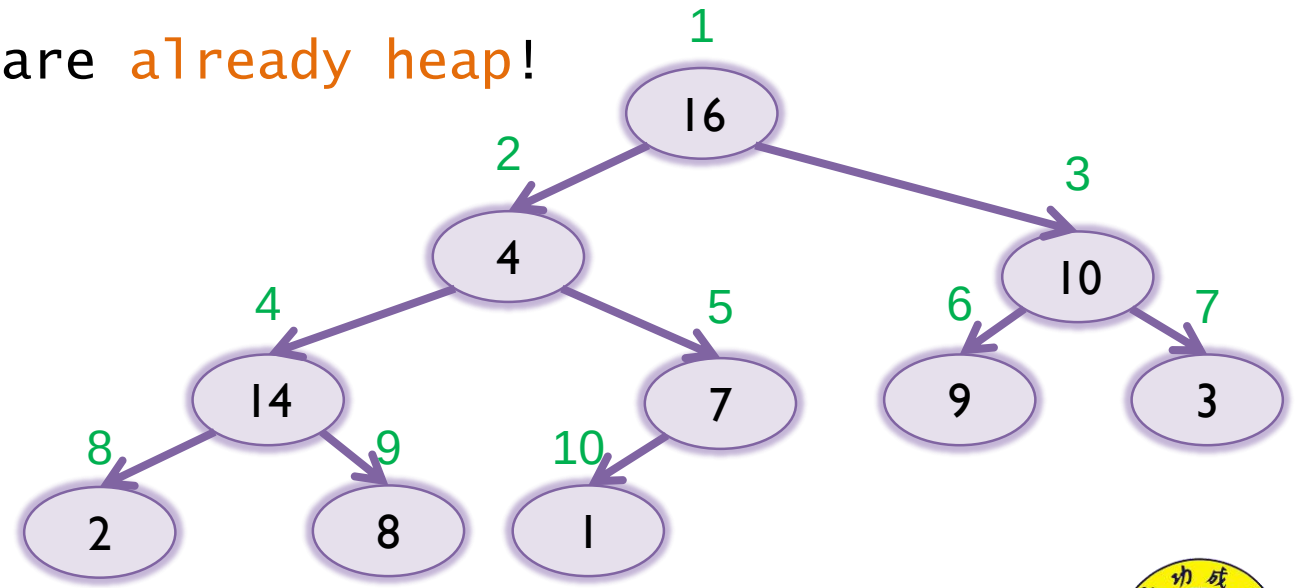
- How to build the entire heap?
- Recursively build the sub-trees first, then do **max-heapify**?
- We can do **max-heapify**(A, i) from i = length[A] to 1!



Max-Heap

- How to build the entire heap?
- Recursively build the sub-trees first, then do **max-heapify**?
- We can do **max-heapify**(A, i) from i = length[A] to 1!

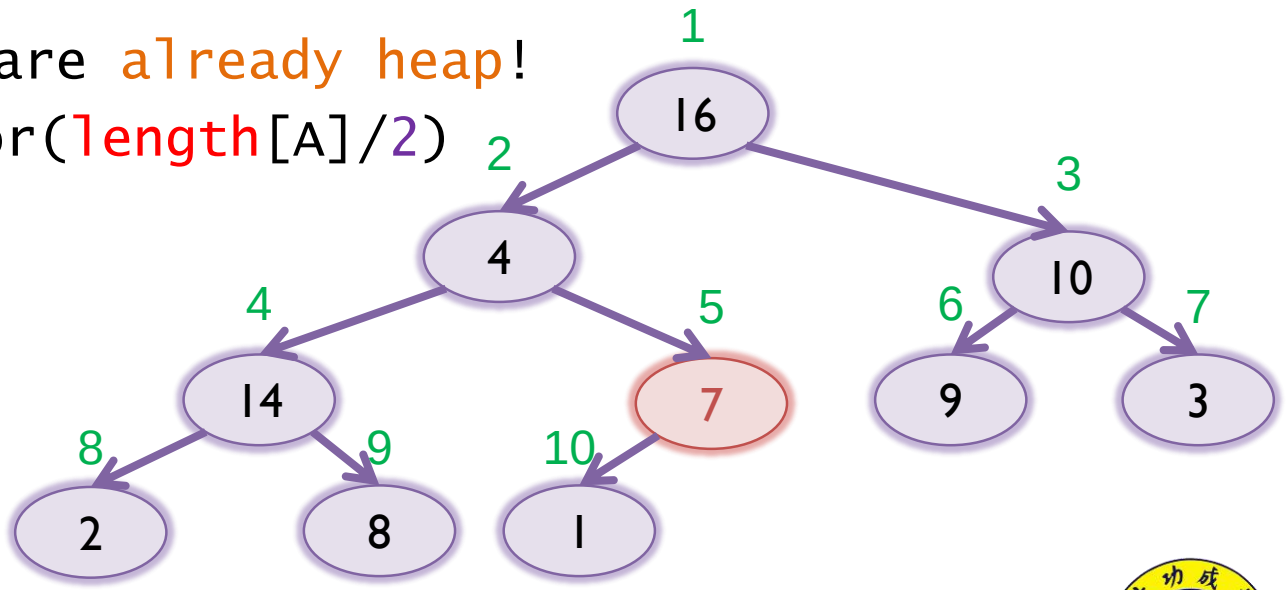
- The leaves are already heap!



Max-Heap

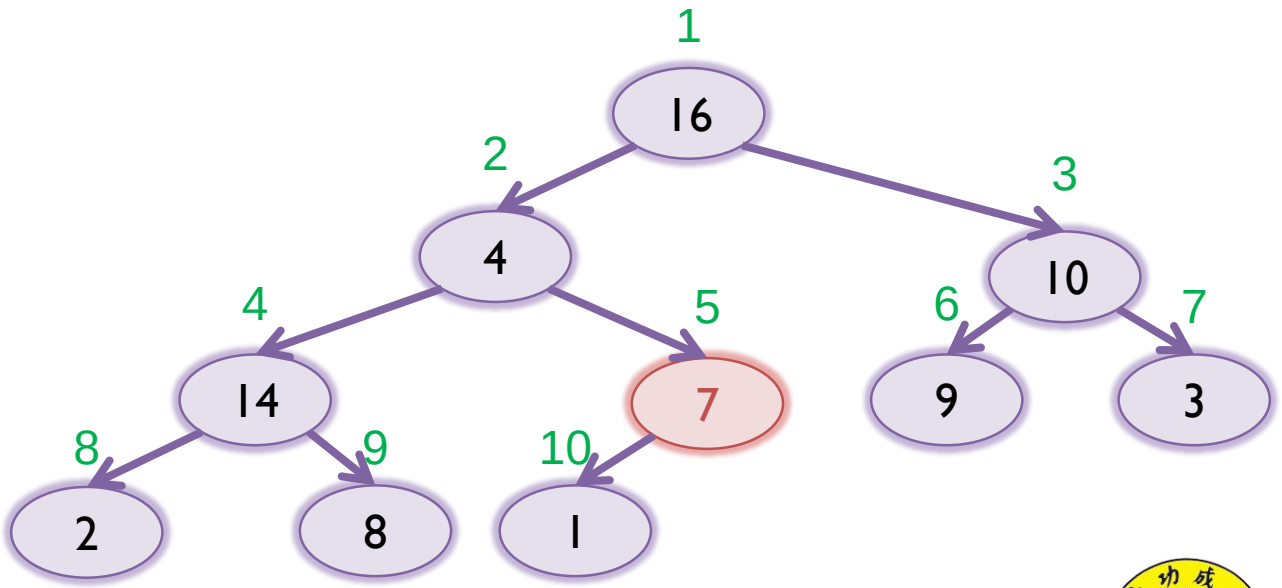
- How to build the entire heap?
- Recursively build the sub-trees first, then do **max-heapify**?
- We can do **max-heapify**(A, i) from i = length[A] to 1!

- The leaves are already heap!
  → from floor(length[A]/2)



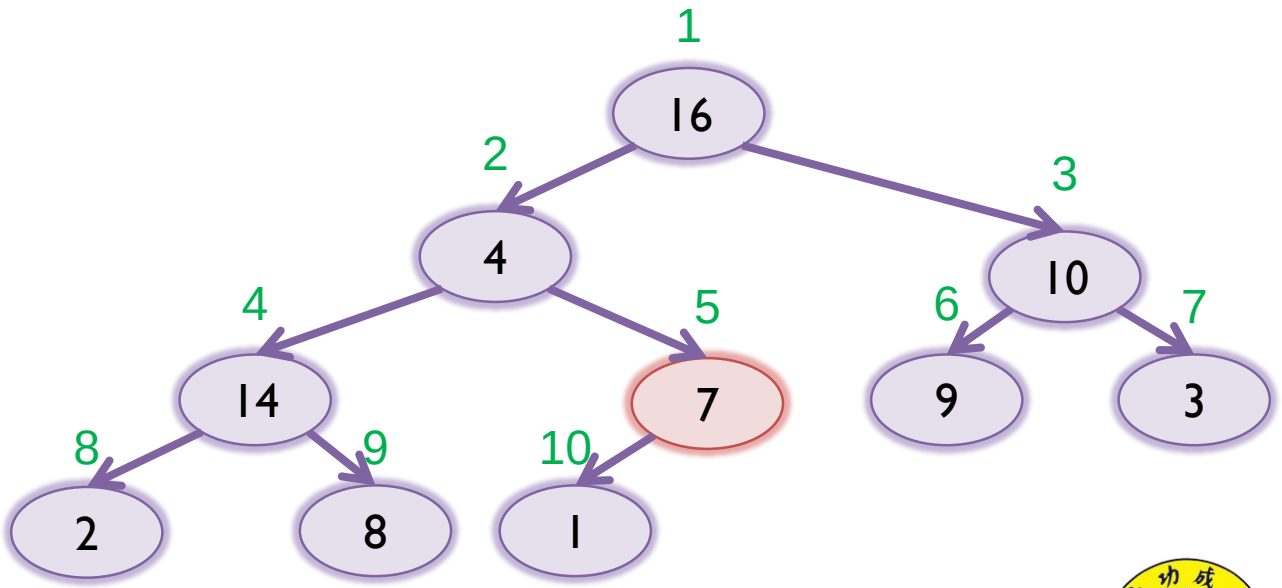
Max-Heap

```
Build-Max-Heap(A):
1. heap-length[A]←length[A]
2. for i ← floor(length[A]/2) down to 1 do
3.     Max-Heapify(A, i)
```



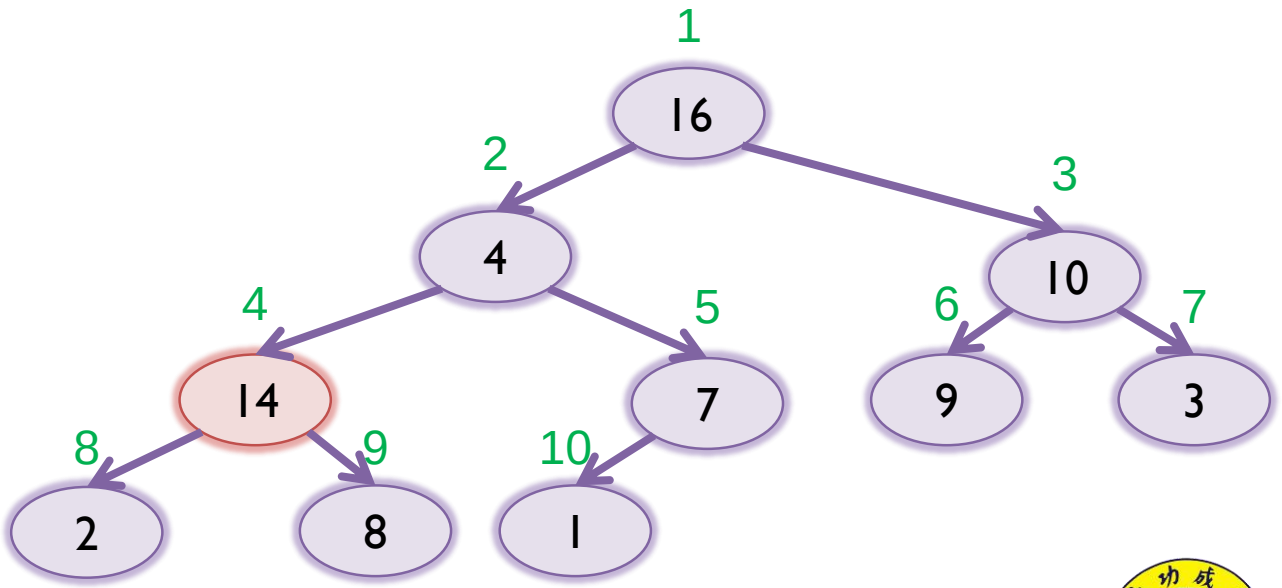
Max-Heap

```
Build-Max-Heap(A):
1. heap-length[A]←length[A]
2. for i ← floor(length[A]/2) down to 1 do
3.     Max-Heapify(A, 5)
```



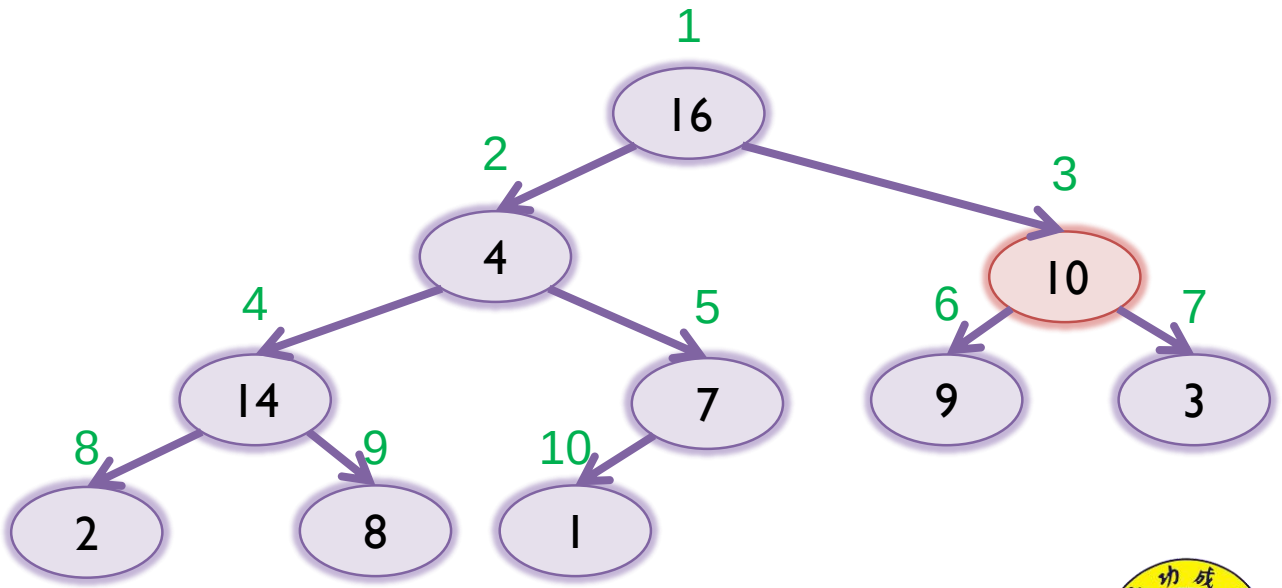
Max-Heap

```
Build-Max-Heap(A):
1. heap-length[A]←length[A]
2. for i ← floor(length[A]/2) down to 1 do
3.     Max-Heapify(A, 4)
```



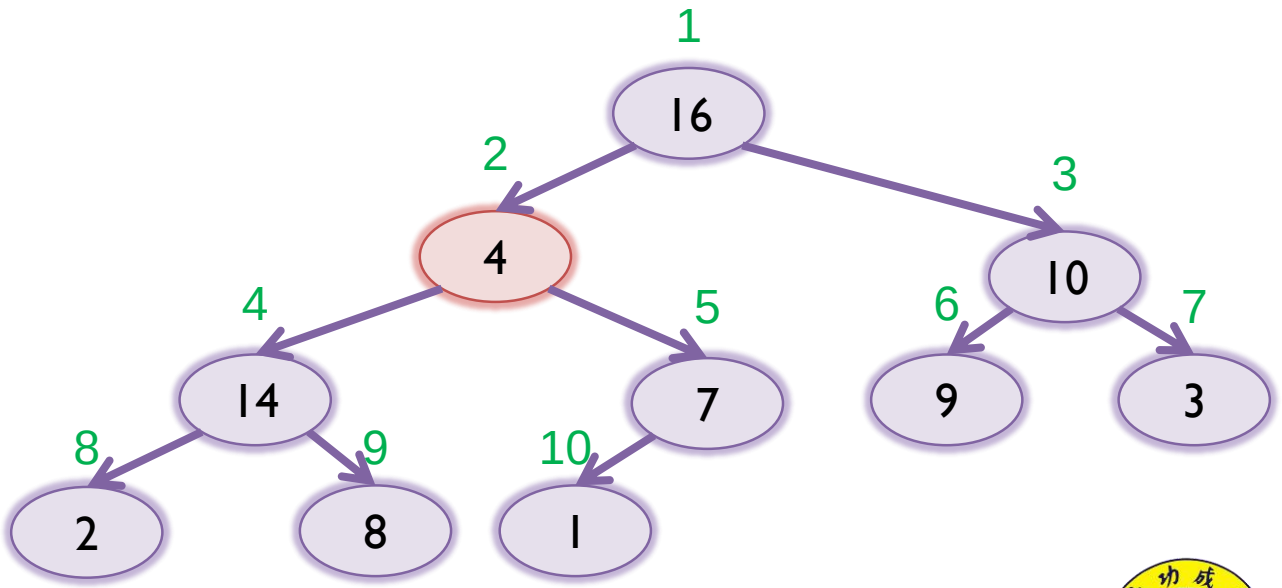
Max-Heap

```
Build-Max-Heap(A):
1. heap-length[A]←length[A]
2. for i ← floor(length[A]/2) down to 1 do
3.     Max-Heapify(A, 3)
```



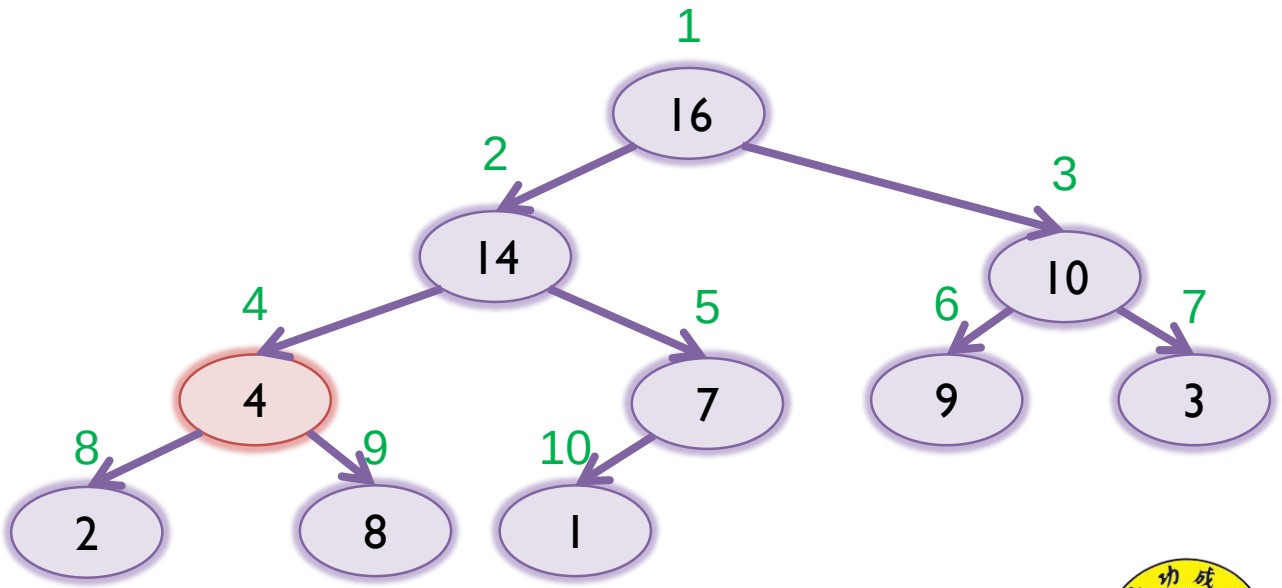
Max-Heap

```
Build-Max-Heap(A):
1. heap-length[A]←length[A]
2. for i ← floor(length[A]/2) down to 1 do
3.     Max-Heapify(A, 2)
```



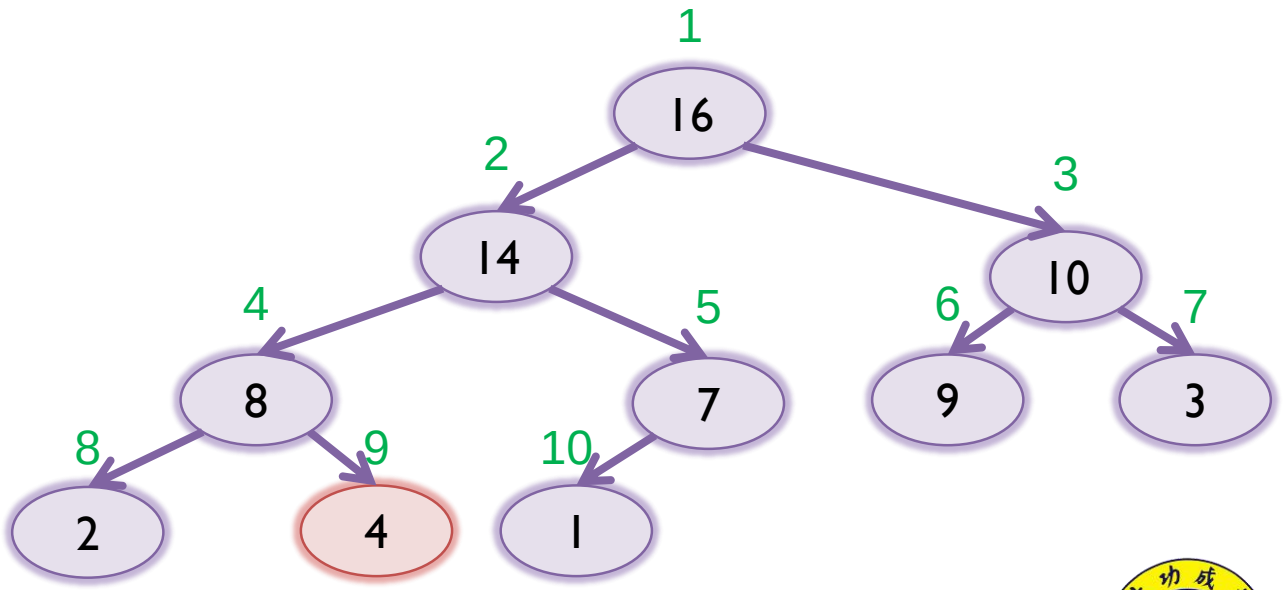
Max-Heap

```
Build-Max-Heap(A):
1. heap-length[A]←length[A]
2. for i ← floor(length[A]/2) down to 1 do
3.     Max-Heapify(A, 2)
```



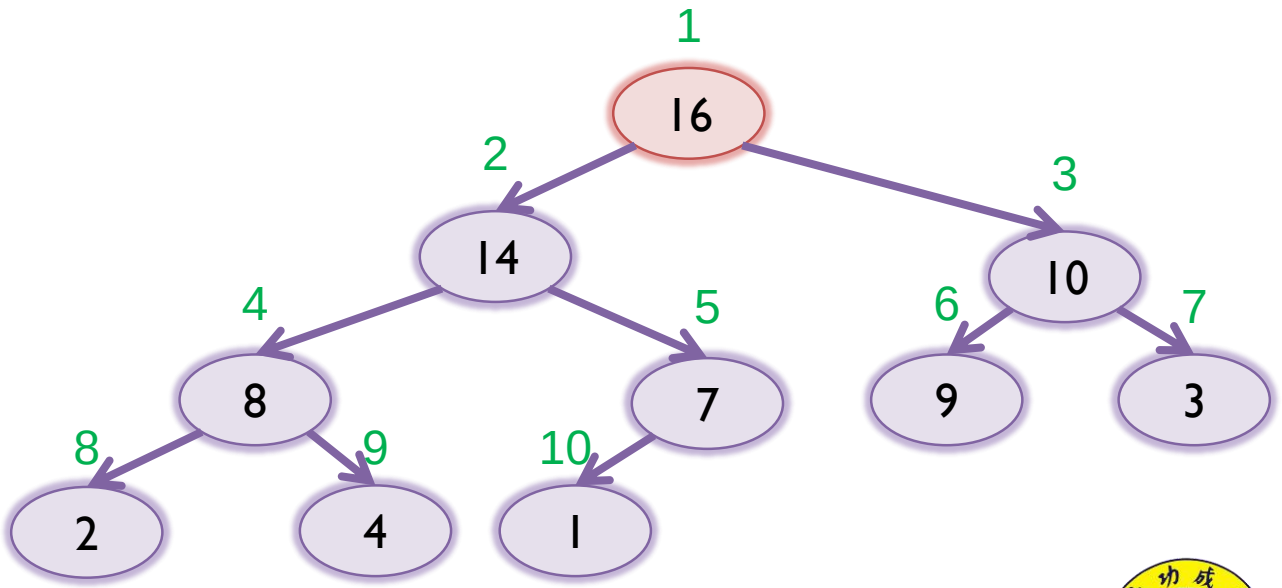
Max-Heap

```
Build-Max-Heap(A):
1. heap-length[A]←length[A]
2. for i ← floor(length[A]/2) down to 1 do
3.     Max-Heapify(A, 2)
```



Max-Heap

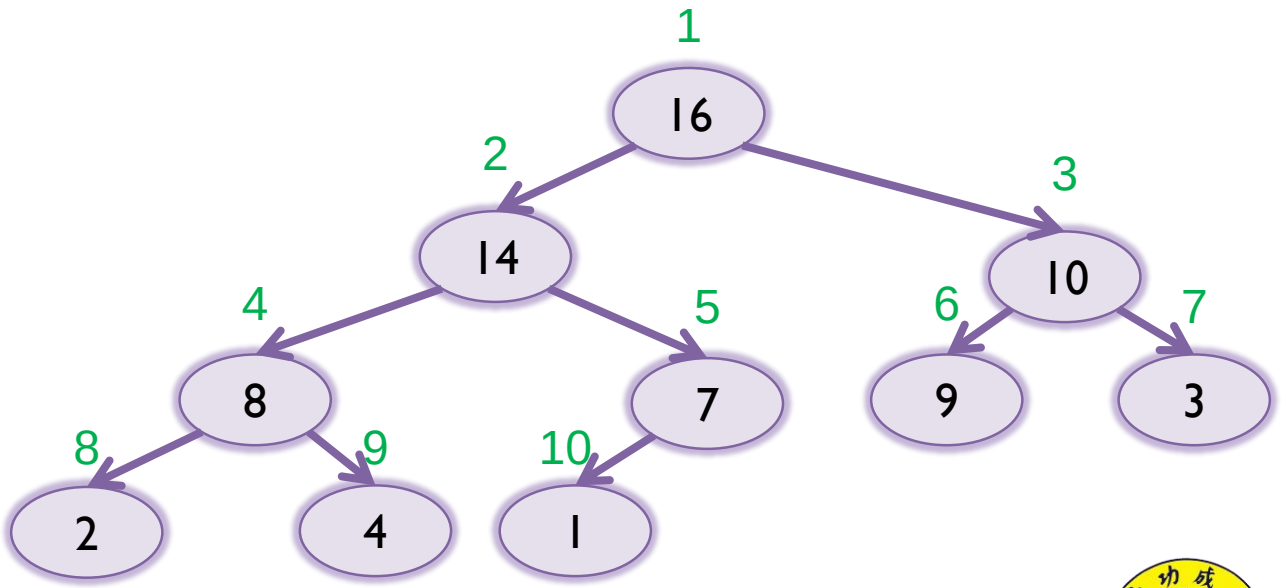
```
Build-Max-Heap(A):
1. heap-length[A]←length[A]
2. for i ← floor(length[A]/2) down to 1 do
3.     Max-Heapify(A, 1)
```



Max-Heap

```
Build-Max-Heap(A):
1. heap-length[A]←length[A]
2. for i ← floor(length[A]/2) down to 1 do
3.     Max-Heapify(A, i)
```

Done!



Max-Heap

- Time complexity of **Build-Max-Heap(A)**?

Max-Heap

- Time complexity of **Build-Max-Heap(A)**?
- In the first glimpse:
  - About n/2 calls to **Max-Heapify**(A, i).
  - Each takes O(log N) time.

Max-Heap

- Time complexity of **Build-Max-Heap(A)**?
- In the first glimpse:
  - About n/2 calls to **Max-Heapify**(A, i).
  - Each takes O(log N) time.
- The total is O(N*log N)!

Max-Heap

- Time complexity of **Build-Max-Heap(A)**?
- In the first glimpse:
  - About n/2 calls to **Max-Heapify**(A, i).
  - Each takes O(log N) time.
- The total is O(N*log N)!

- But… the **Max-Heapify**(A, i) is not always O(log N)!

Max-Heap

- Time complexity of **Build-Max-Heap(A)**?
- In the first glimpse:
  - About n/2 calls to **Max-Heapify**(A, i).
  - Each takes O(log N) time.
- The total is O(N*log N)!

- But… the **Max-Heapify**(A, i) is not always O(log N)!
- The height of heap is $\lfloor \log(N) \rfloor$, the number of nodes at height h is $\leq \lceil n/2^{h+1} \rceil$.

Max-Heap

- Time complexity of **Build-Max-Heap(A)**?
- In the first glimpse:
  - About n/2 calls to **Max-Heapify**(A, i).
  - Each takes O(log N) time.
- The total is O(N*log N)!

- But… the **Max-Heapify**(A, i) is not always O(log N)!
- The height of heap is $\lfloor \log(N) \rfloor$, the number of nodes at height h is $\leq \lceil n/2^{h+1} \rceil$.

- And then

Max-Heap

- Time complexity of **Build-Max-Heap(A)**?
- In the first glimpse:
  - About $n/2$ calls to **Max-Heapify**(A, i).
  - Each takes $O(\log N)$ time.
- The total is $O(N*\log N)$!

- But… the **Max-Heapify**(A, i) is not always $O(\log N)$!
- The height of heap is $\lfloor \log(N) \rfloor$, the number of nodes at height h is $\leq \lceil n/2^{h+1} \rceil$.

- And then…

Max-Heap

- Time complexity of **Build-Max-Heap(A)**?
- In the first glimpse:
  - About n/2 calls to **Max-Heapify**(A, i).
  - Each takes O(log N) time.
- The total is O(N*log N)!

- But… the **Max-Heapify**(A, i) is not always O(log N)!
- The height of heap is $\lfloor \log(N) \rfloor$, the number of nodes at height h is $\leq \lceil n/2^{h+1} \rceil$.

- And then……

Max-Heap

- Time complexity of **Build-Max-Heap(A)**?
- In the first glimpse:
  - About n/2 calls to **Max-Heapify**(A, i).
  - Each takes O(log N) time.
- The total is O(N*log N)!

- But… the **Max-Heapify**(A, i) is not always O(log N)!
- The height of heap is $\lfloor \log(N) \rfloor$, the number of nodes at height h is $\leq \lceil n/2^{h+1} \rceil$.

- And then…… O(N)!

Max-Heap

- Did we forget something?
  - Array → Max-heap or Min-heap

Max-Heap

- Did we forget something?
  - Array → Max-heap or Min-heap

- **Insertion** and **Deletion**?

Max-Heap

- Did we forget something?
  - Array → Max-heap or Min-heap

- **Insertion** and **Deletion**?
- Add an element to the heap
- Delete the root from the heap

Max-Heap

- Did we forget something?
  - Array → Max-heap or Min-heap

- **Insertion** and **Deletion**?
- Add an element to the heap
- Delete the root from the heap

- Conform the shape property first, then restore the heap property by traversing up or down.

Max-Heap

- Did we forget something?
  - Array → Max-heap or Min-heap

- **Insertion** and **Deletion**?
- Add an element to the heap
- Delete the root from the heap

- Conform the shape property first, then restore the heap property by traversing up or down.

- The last element of the bottom level!

Max-Heap

Insertion algorithm

1. Add the element to the bottom level of the heap.

2. Compare the added element with its parent.
   If they are in the correct order, stop.

3. If not, swap the element with its parent and return
   to step 2.

- Up-heap

Max-Heap

Deletion algorithm

1. Replace the root of the heap with the last element of the bottom level.
2. Compare the new root with its children.
   If they are in the correct order, stop.

3. If not, swap the element with one of its children and return to step 2.(swap with larger child for Max-heap, swap with smaller child for Min-heap).

- Down-heap, **Max-heapify** is similar to it.

Max-Heap

- Time complexity of Insertion and Deletion?
- In the worst case, the node would traverse from bottom to root or from root to bottom.
- So the time complexity relative to the height of the tree. O(h) or O(log N)

Max-Heap

- HeapSort

- in-place algorithm.
- not a stable sort.
- Time complexity:
  - worst case: O(Nlog N)
  - average: O(Nlog N)

HeapSort algorithm

1. build the heap
2. repeatedly removing the root, and inserting into the array.

Max-Heap

- Previous question again!

Max-Heap

- Previous question again!
- I am so lazy!

Max-Heap

- Previous question again!
- I am so lazy!
- Time is money!

Max-Heap

- Previous question again!
- I am so lazy!
- Time is money!
- Don`t worry! Be Happy!
  - C++ STL priority_queue might fulfill your dream!

~Let`s Practice~

Made By mike199250

UVa 10954

- Link: http://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=21&page=show_problem&problem=1895

UVa 10954

- Hint:

  1. If there are only three numbers a, b, c and $a \le b \le c$, how should we add?

  2. Two numbers after addition, are still two numbers?

Problems

- POJ(6)
  - 1338, 2255, 2431, 2442, 3253, 3481
- Uva(8)
  - 501, 712, 10821, 10909, 10954, 11995, 11997, 12347

- 基本題為5題
- 第二次修課的同學請至少完成上列紅字題號中的5題來達成基本題