

Data Structures & Algorithms

Subodh Kumar

(subodh@iitd.ac.in, Bharti 422)

Dept of Computer Sc. & Engg.



Basic Data Structures



Basic Data Structures

- Primitive types



Basic Data Structures

■ Primitive types



memory: sequence of bytes



Basic Data Structures

■ Primitive types


bits memory: sequence of bytes

The diagram illustrates memory as a sequence of bytes. It shows a row of eight small squares, each representing a byte. The first square contains vertical lines representing bits. To the right of the squares is a three-dot ellipsis, indicating that the sequence continues. Below the squares is the word "bits". To the right of the ellipsis is the text "memory: sequence of bytes".



Basic Data Structures

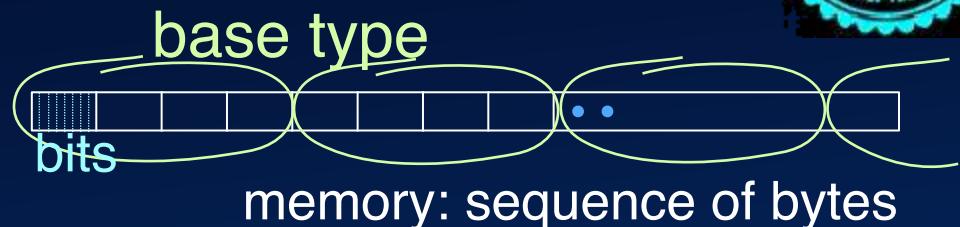
■ Primitive types





Basic Data Structures

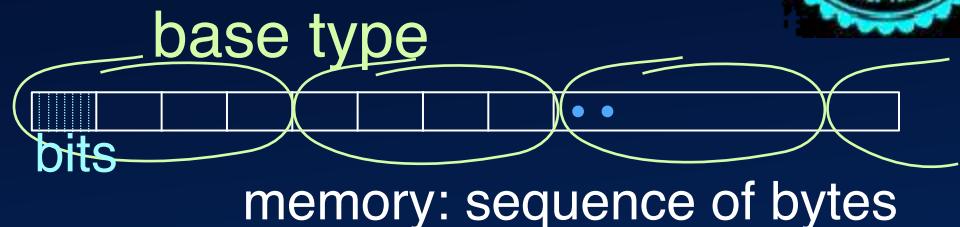
■ Primitive types





Basic Data Structures

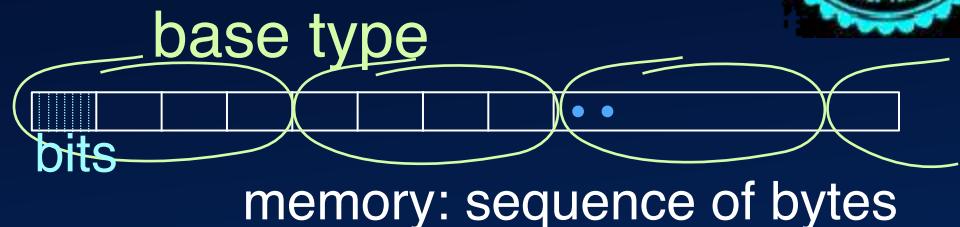
■ Primitive types





Basic Data Structures

- Primitive types
- List





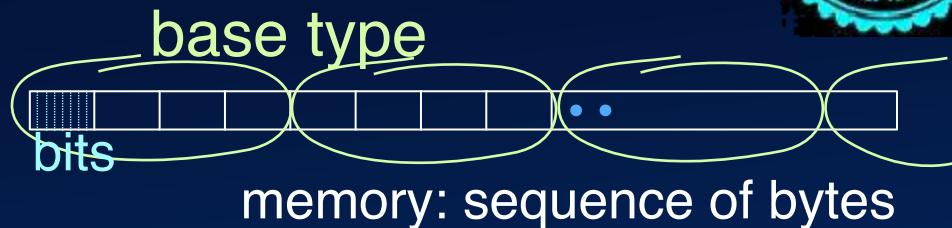
Basic Data Structures

- **Primitive types**

- **List**

- **Global rank access**

- Access only at a single rank
 - **Array, Vector**





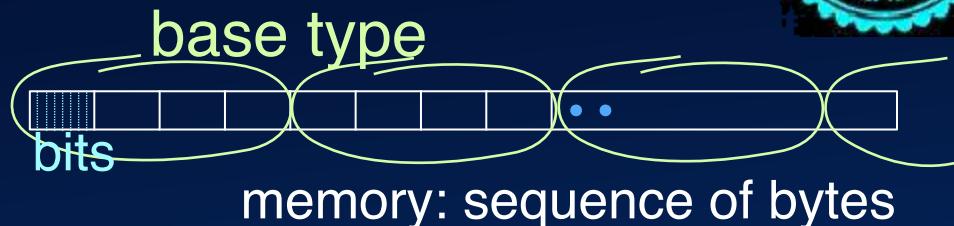
Basic Data Structures

- **Primitive types**

- **List**

- **Global rank access**

- Access only at a single rank
 - **Array, Vector**





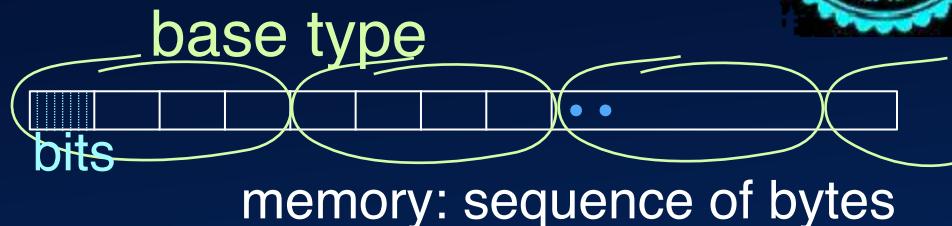
Basic Data Structures

■ Primitive types

■ List

■ Global rank access

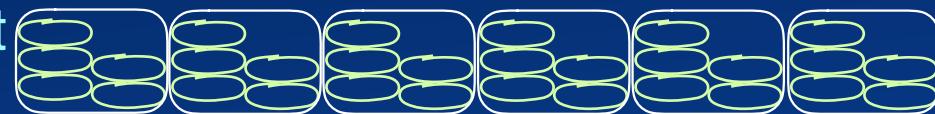
- Access only at a single rank
- Array, Vector



Reference
Array



Object
Array





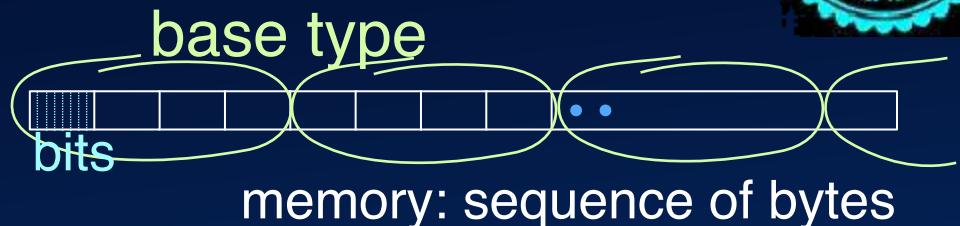
Basic Data Structures

■ Primitive types

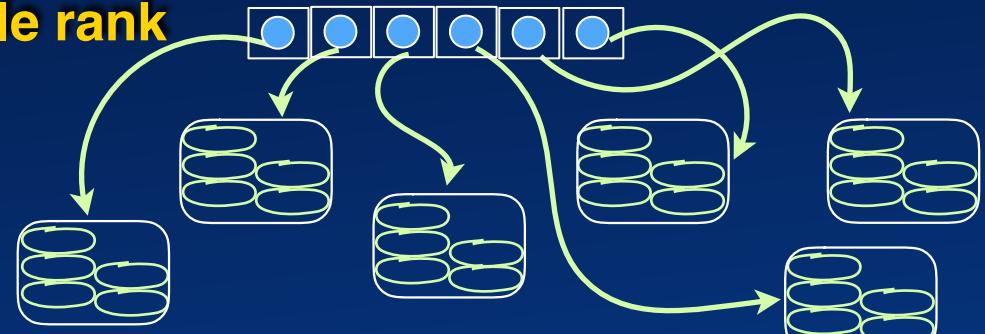
■ List

■ Global rank access

- Access only at a single rank
- Array, Vector



Reference
Array



Object
Array





Basic Data Structures

■ Primitive types

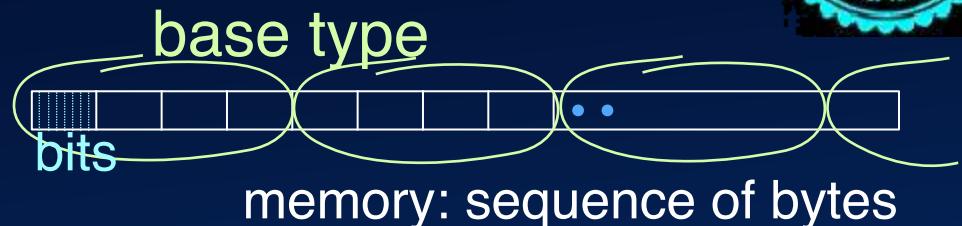
■ List

■ Global rank access

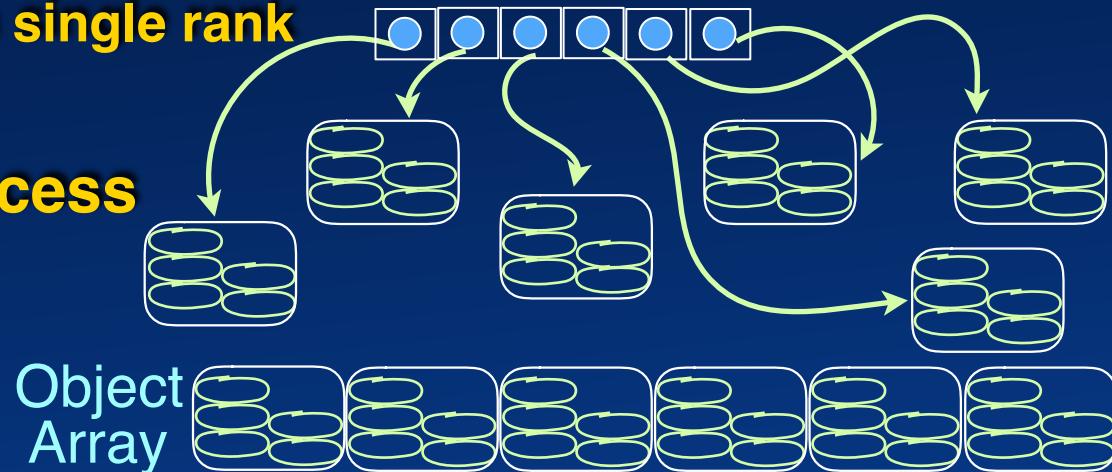
- Access only at a single rank
- Array, Vector

■ Local relative access

- Local update
- Linked lists



Reference
Array





Basic Data Structures

■ Primitive types

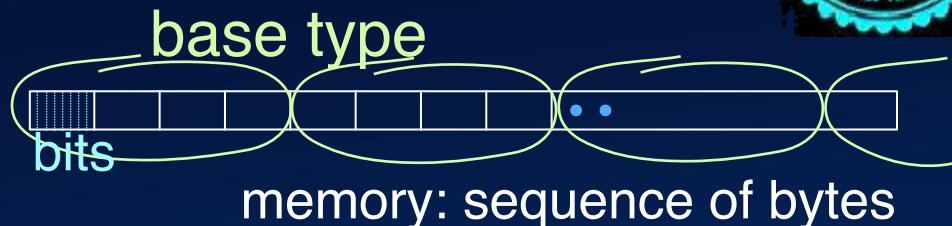
■ List

■ Global rank access

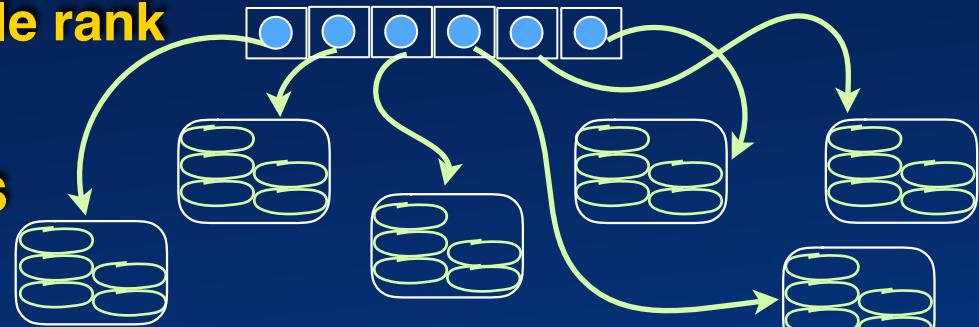
- Access only at a single rank
- Array, Vector

■ Local relative access

- Local update
- Linked lists



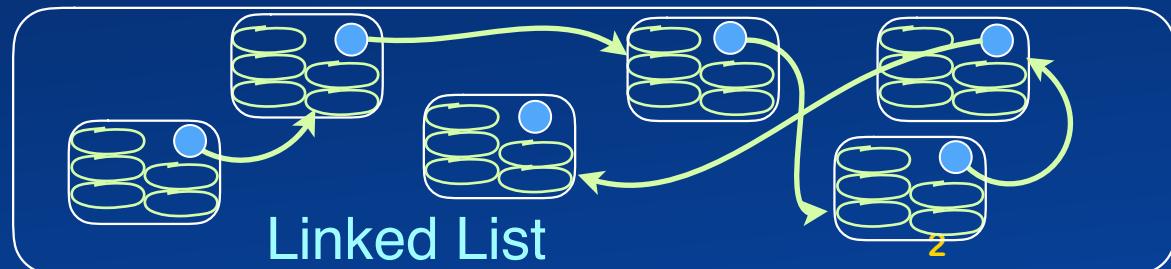
Reference
Array



Object
Array



Linked List





Basic Data Structures

■ Primitive types

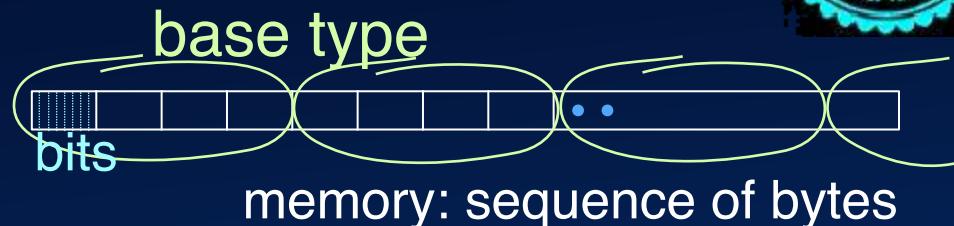
■ List

■ Global rank access

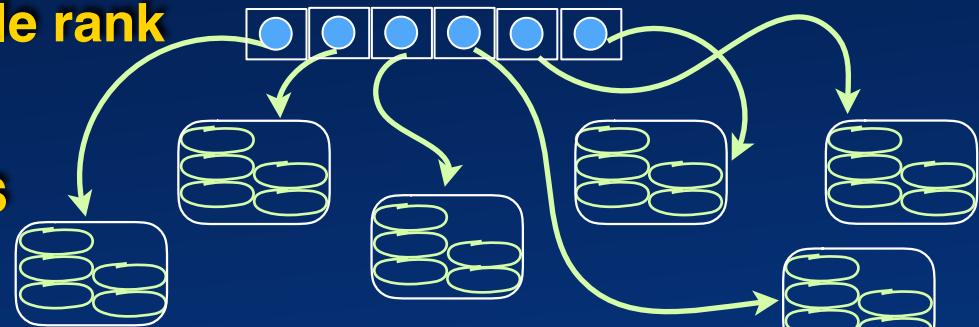
- Access only at a single rank
- Array, Vector

■ Local relative access

- Local update
- Linked lists



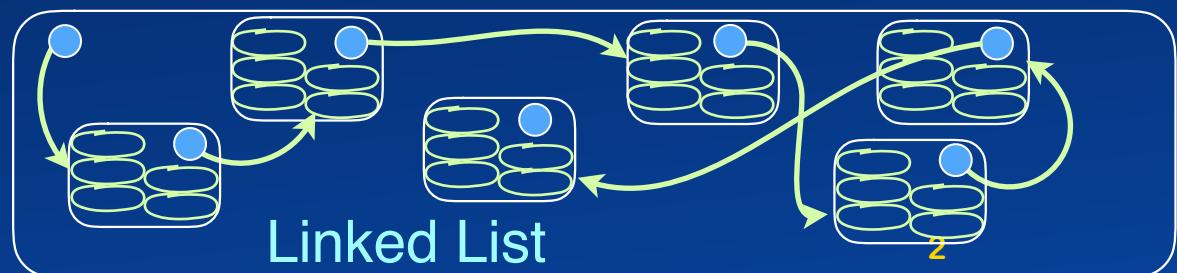
Reference
Array



Object
Array



Linked List





Basic Data Structures

■ Primitive types

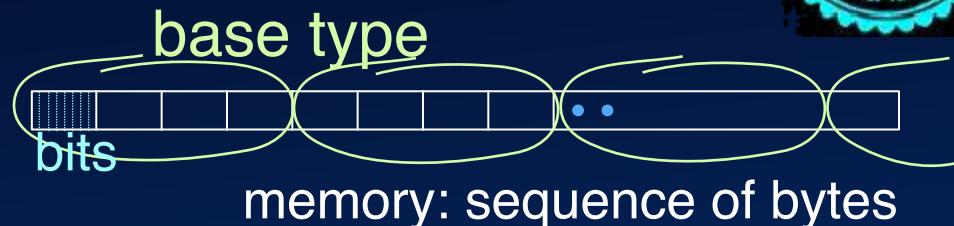
■ List

■ Global rank access

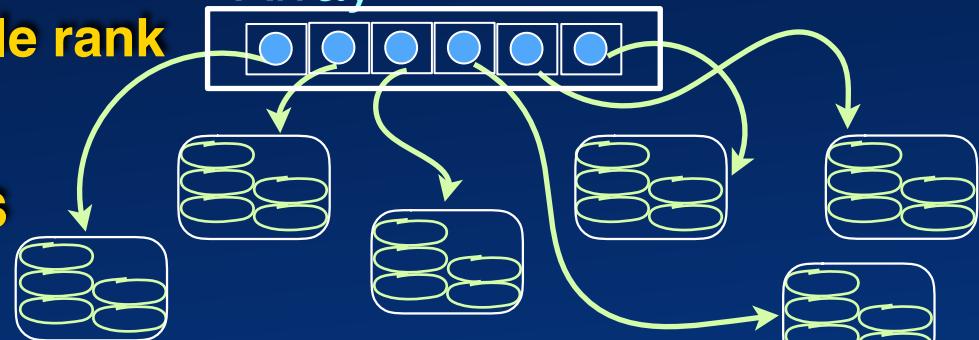
- Access only at a single rank
- Array, Vector

■ Local relative access

- Local update
- Linked lists



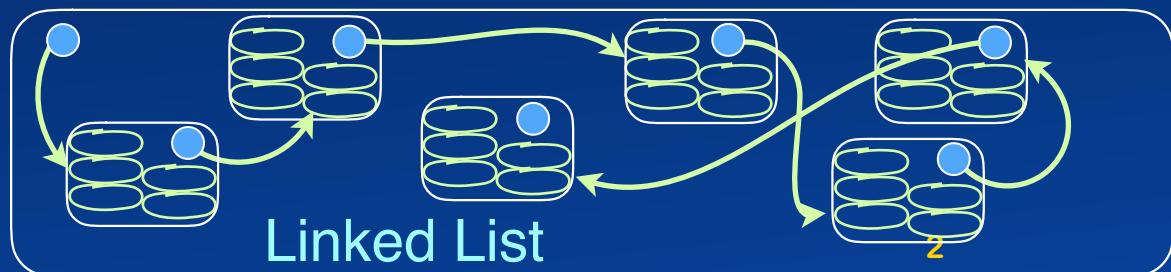
Reference
Array



Object
Array



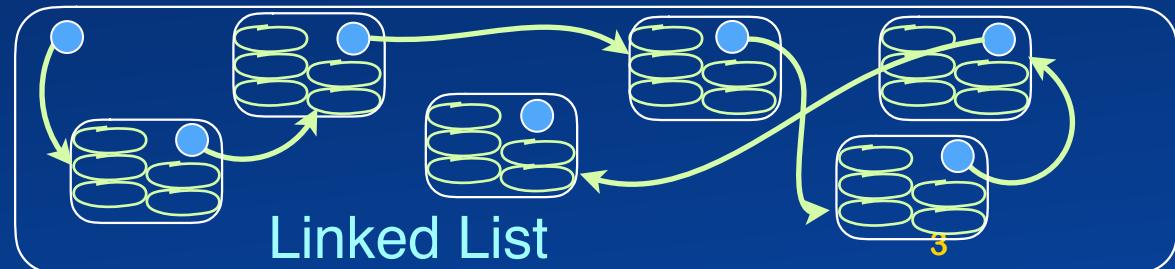
Linked List





Basic Data Structures

- Primitive types
- List
 - Global rank access
 - Access only at a single rank
 - Array, Vector
 - Local relative access
 - Local update
 - Linked lists
 - Both
 - Sequence





Basic Data Structures

- Primitive types

- List

- Global rank access

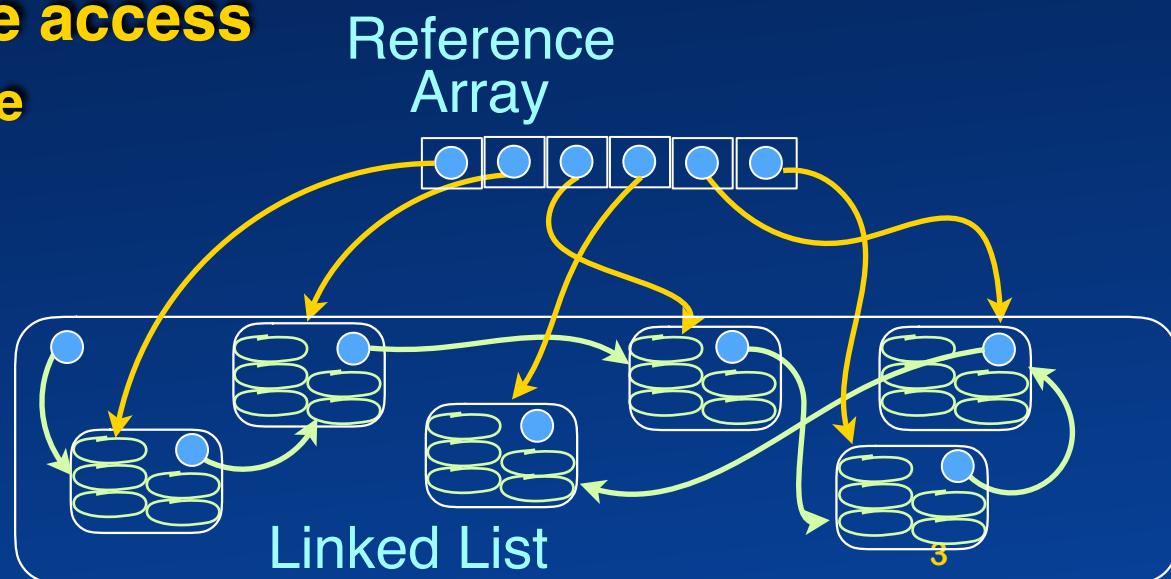
- Access only at a single rank
 - Array, Vector

- Local relative access

- Local update
 - Linked lists

- Both

- Sequence





Basic Data Structures

- Primitive types
- List
 - Global rank access
 - Access only at a single rank
 - Array, Vector
 - Local relative access
 - Local update
 - Linked lists
 - Both
 - Sequence
 - Queue/Deque
 - Stack



Basic Data Structures

- Primitive types
- List
 - Global rank access
 - Access only at a single rank
 - Array, Vector
 - Local relative access
 - Local update
 - Linked lists
 - Both
 - Sequence
 - Queue/Deque
 - Stack

Restrict usage

True or False?



- $\log(n^k) = O(n)$
- $(\log n)^k = O(n)$
- Average case complexity = O(worst case complexity)
- Average case complexity = o(worst case complexity)



Collection

■ Bag of Objects

- Need a way to identify objects
- Add a new object
- Delete an identified object
- Check if an identified object is present

Dictionary

```
public interface Dictionary<K, V> {  
    public V get(K key);  
    public void put(K key, V value);  
    public V remove(K key);  
    public iterator<V> allvalues();  
    public iterator<K> allkeys();  
}
```



Array Implementation

```
class Pair<A,B> {  
    A one;  
    B two;  
}  
public class ArrayMap<K,V> implements Dictionary<K, V> {  
    private Vector<Pair<K,V>> bag;  
    public void put(K key, V value):  
        // Find an empty space and put Pair<K,V> there  
    public V get(K key):  
        // Iterate of map looking for bag[i].one == key  
        // Return bag[found].two  
    // Etc.  
}
```



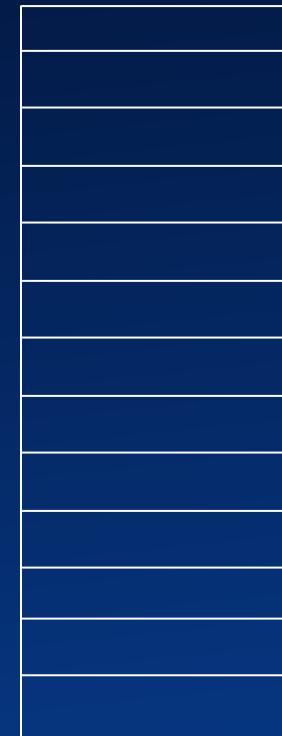
Hashing

- **index = int $i(key)$**



Hashing

- **index = int $i(key)$**

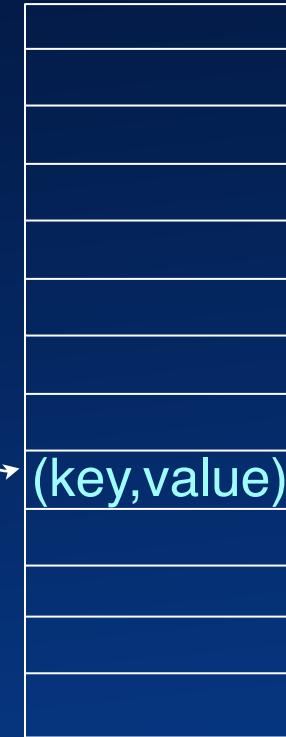


Array



Hashing

- $\text{index} = \text{int } i(\text{key})$ key



Array



Hashing

- $\text{index} = \text{int } i(\text{key})$ key

Another key



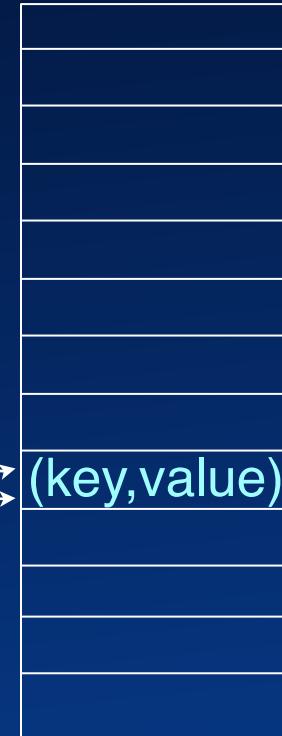


Hashing

- $\text{index} = \text{int } i(\text{key})$ key

Another key

Collision



Array



Hashing

- $\text{index} = \text{int } i(\text{key})$ key

Another key

Collision

Every key should have a different index.



Array

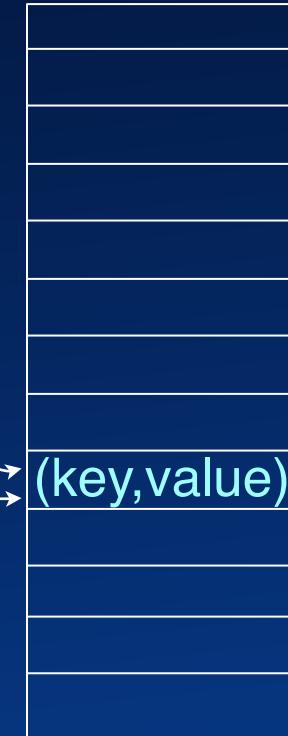


Hashing

- $\text{index} = \text{int } i(\text{key})$ key

Another key

Collision



~~Every key should have a different index.~~

Array



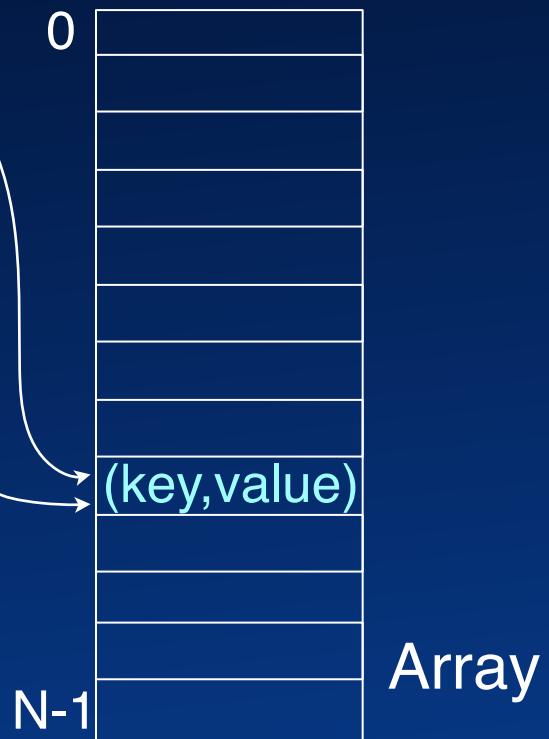
Hashing

- **index = int $i(key)$** key
- **hashcode = int $h(key)$**
- **index $i = \text{hashcode \% N}$**

Another key

Collision

~~Every key should have a different index.~~





Hashing

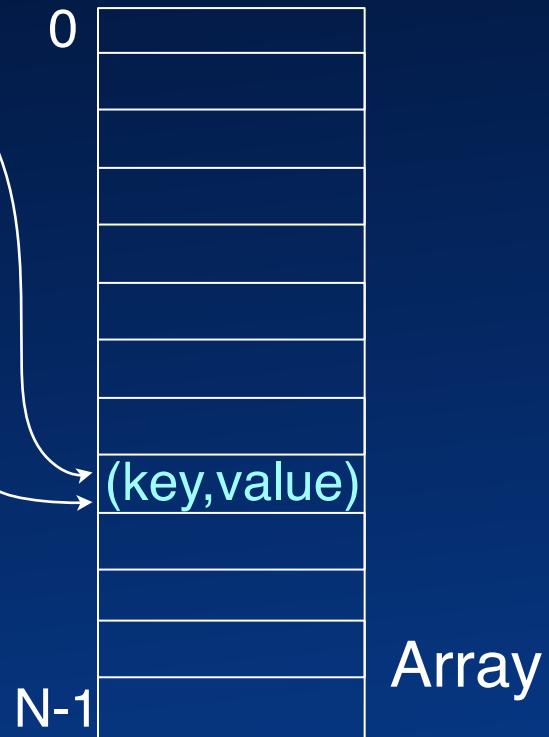
- **index = int $i(key)$** key
- **hashcode = int $h(key)$**
- **index $i = \text{hashcode \% N}$**

Another key

Collision

~~Every key should have a different index.~~

Every key should have a different hashcode
and then index also?





Hashing

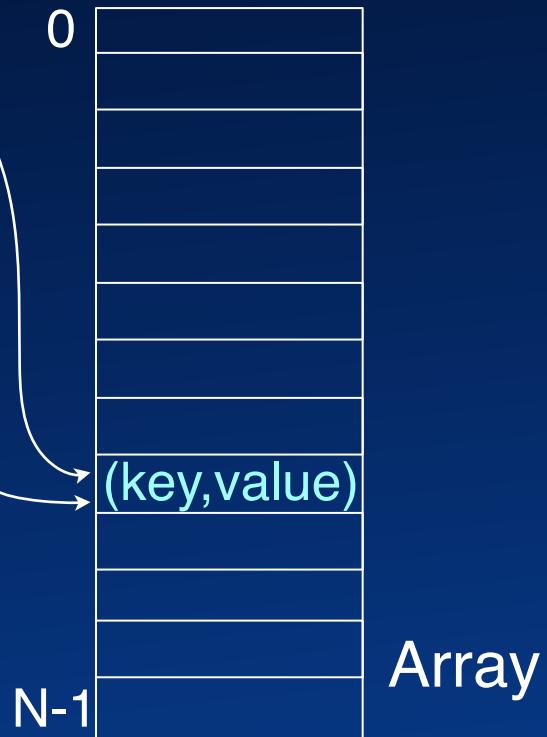
- **index = int $i(key)$** key
- **hashcode = int $h(key)$**
- **index $i = \text{hashcode \% N}$**

Another key

Collision

~~Every key should have a different index.~~

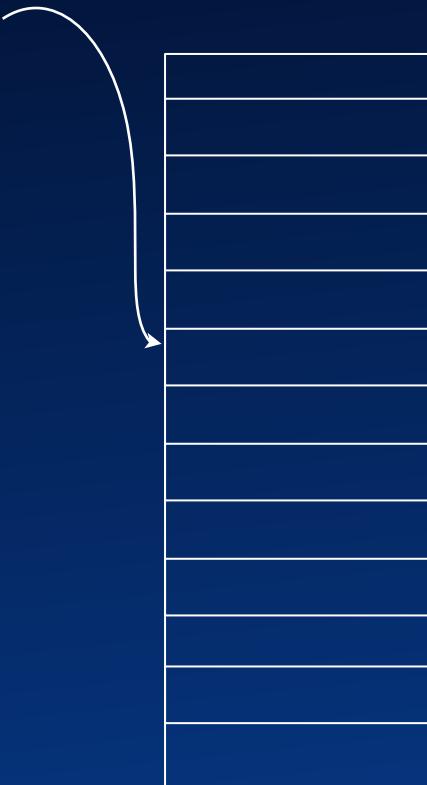
~~Every key should have a different hashcode
and then index also?~~





Collision Resolution

- Separate chaining

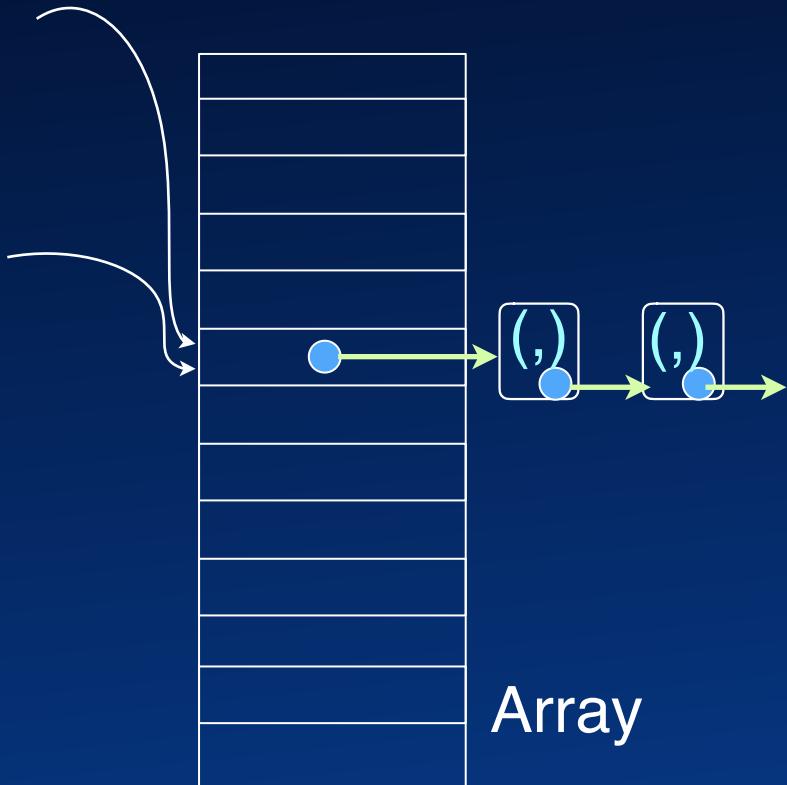


Array



Collision Resolution

■ Separate chaining

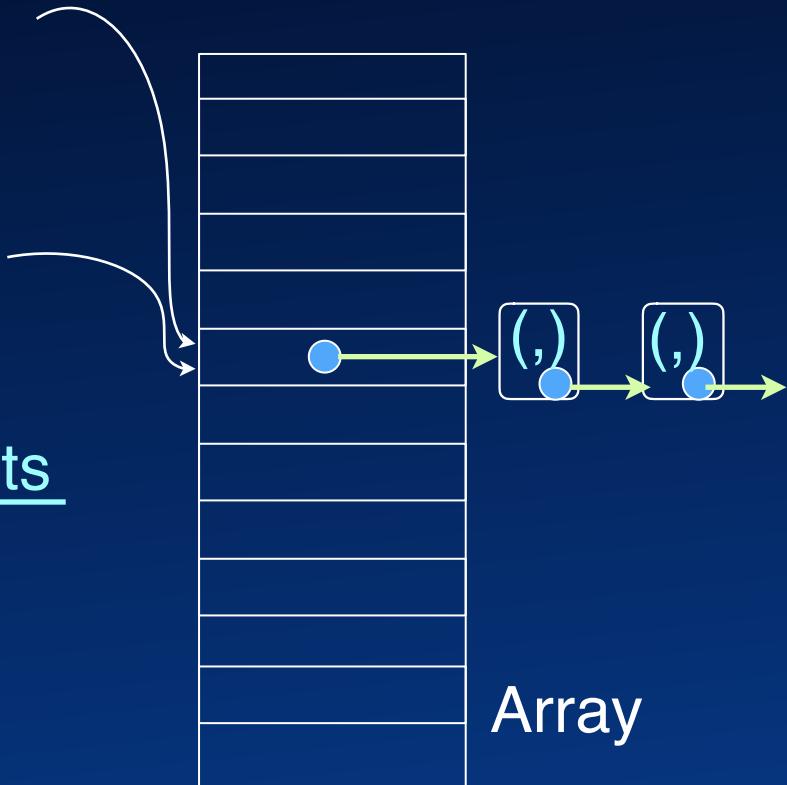




Collision Resolution

■ Separate chaining

Load Factor $\lambda = \frac{\text{No. elements}}{\text{No. slots}}$





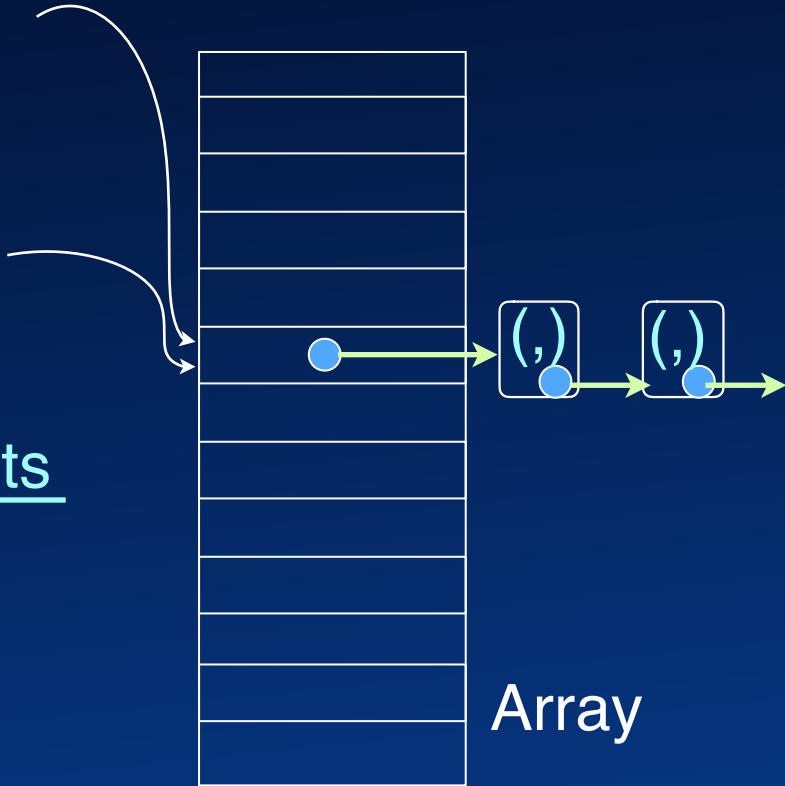
Collision Resolution

■ Separate chaining

$$\text{Load Factor } \lambda = \frac{\text{No. elements}}{\text{No. slots}}$$

Uniformly random keys \Rightarrow

Probability of collision $\leq \min(\lambda, 1)$





Collision Resolution

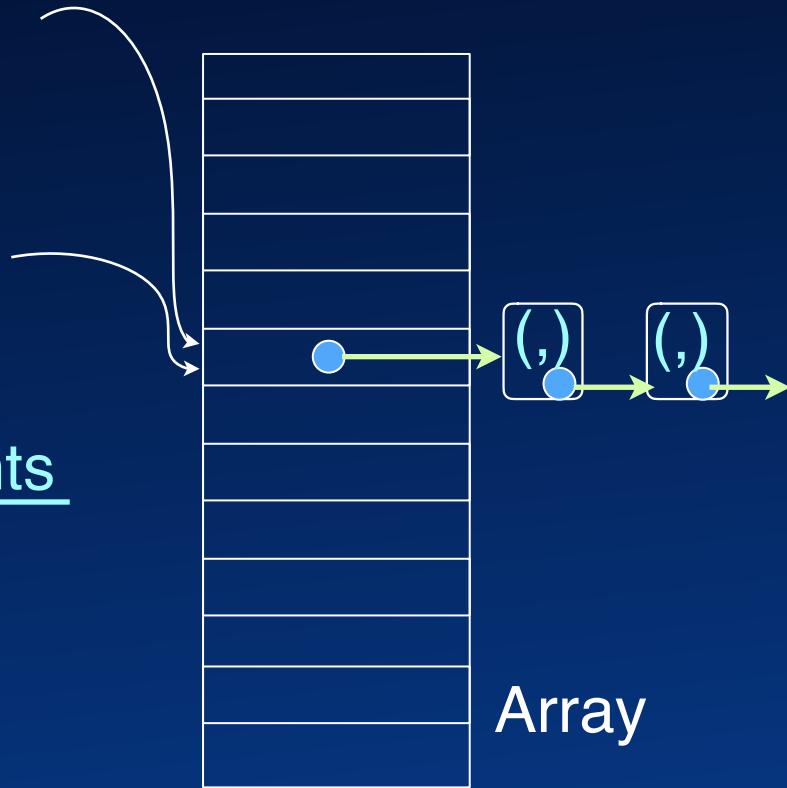
■ Separate chaining

$$\text{Load Factor } \lambda = \frac{\text{No. elements}}{\text{No. slots}}$$

Uniformly random keys \Rightarrow

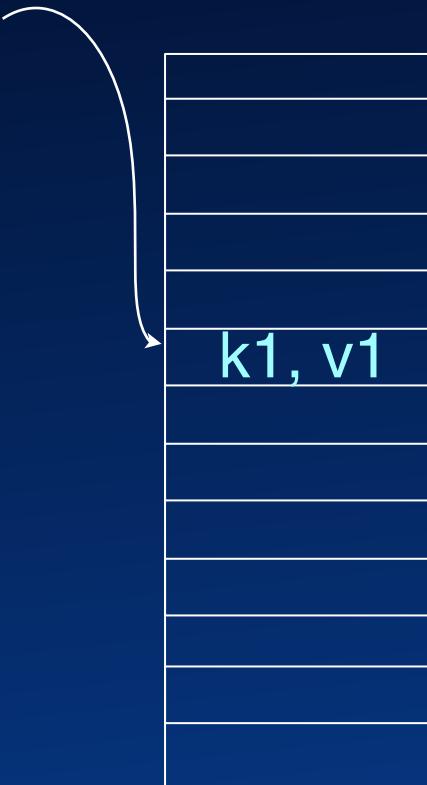
Probability of collision $\leq \min(\lambda, 1)$

= average chain length





Collision Resolution

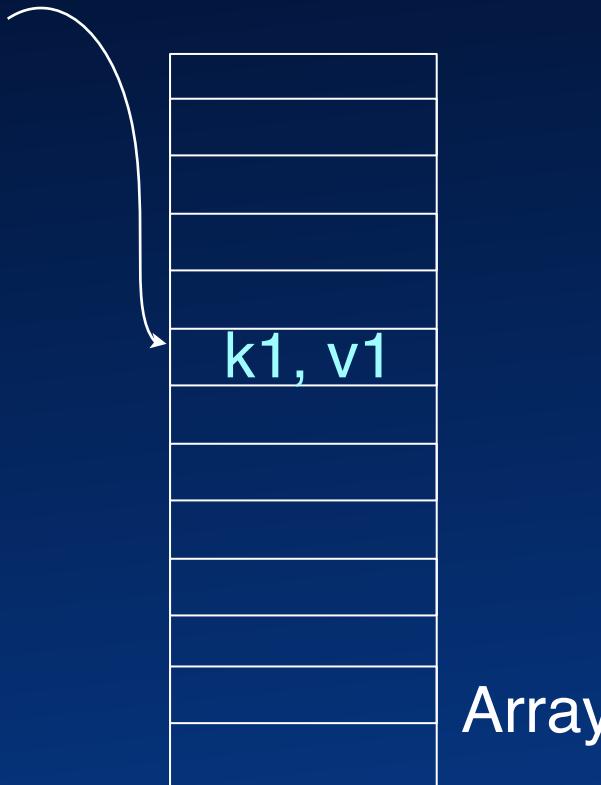


Array



Collision Resolution

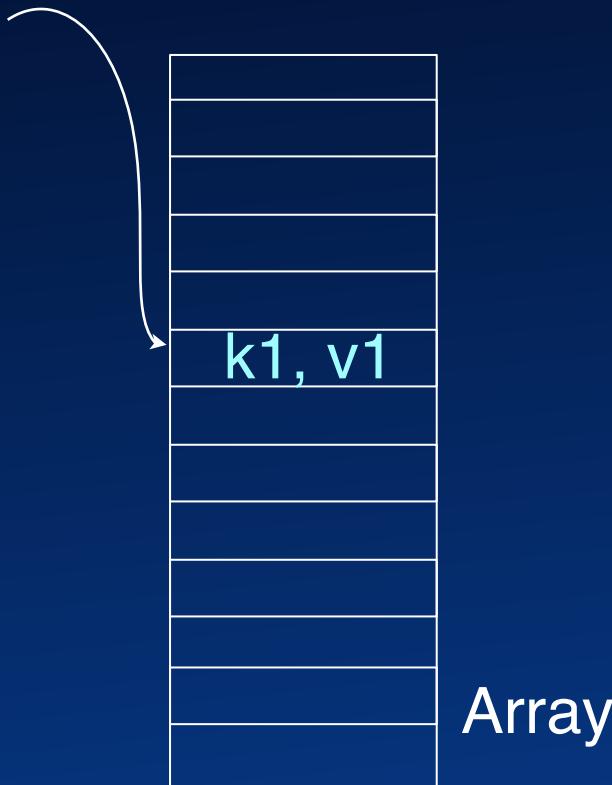
- Separate chaining





Collision Resolution

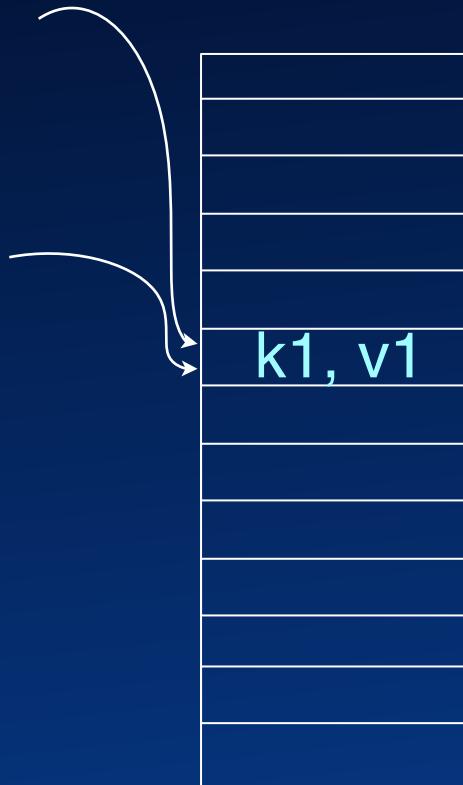
- Separate chaining
- Open Addressing





Collision Resolution

- Separate chaining



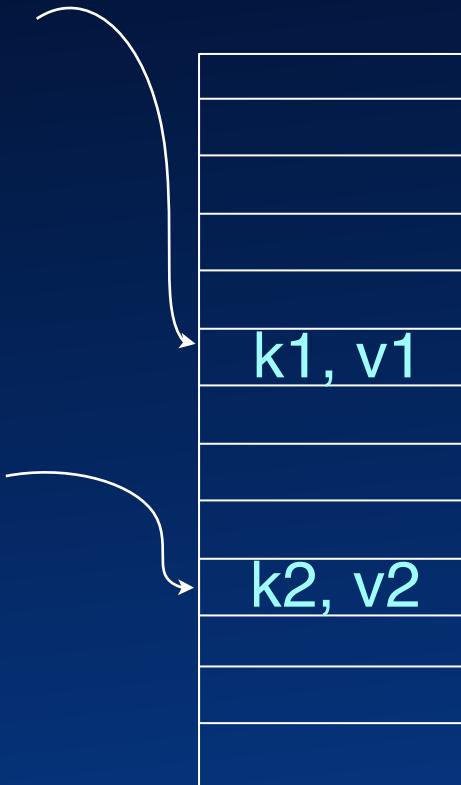
- Open Addressing

Array



Collision Resolution

- Separate chaining

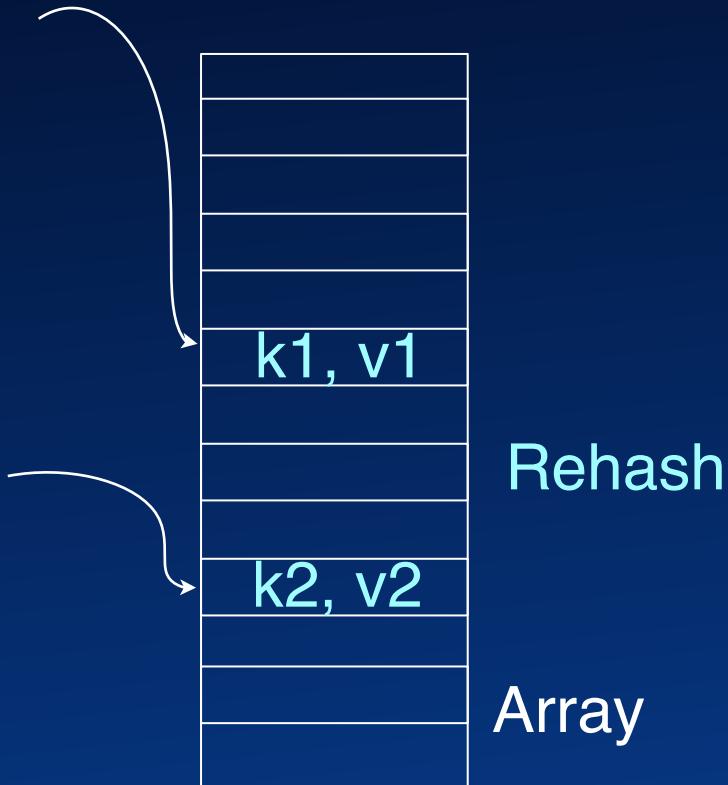


- Open Addressing



Collision Resolution

- Separate chaining

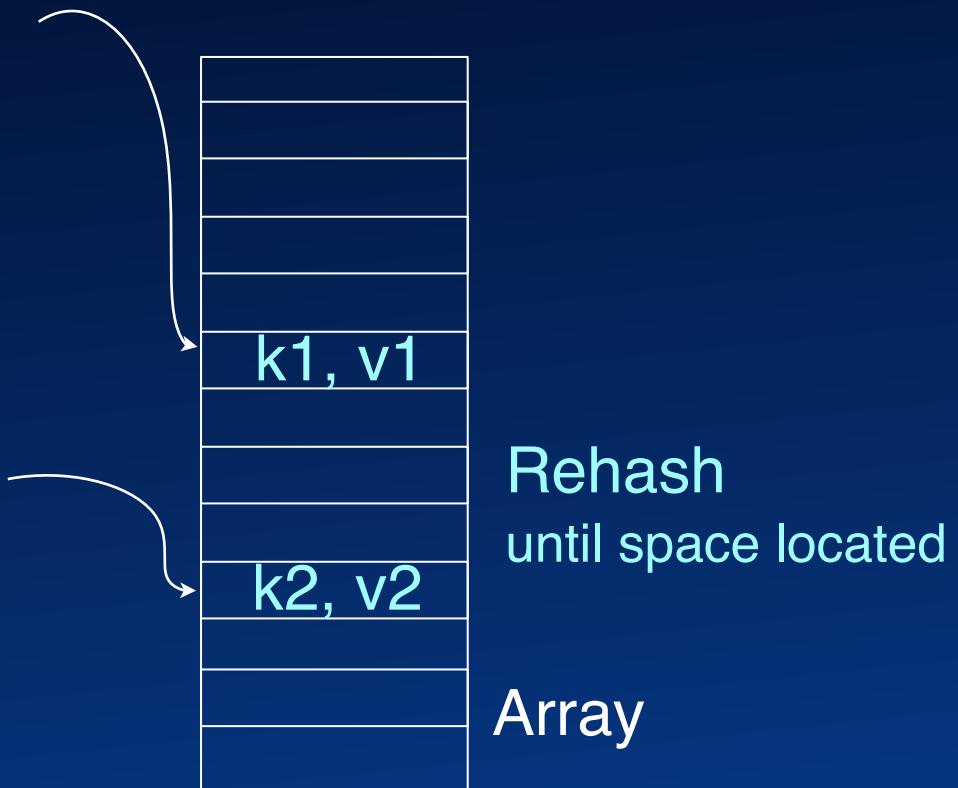


- Open Addressing



Collision Resolution

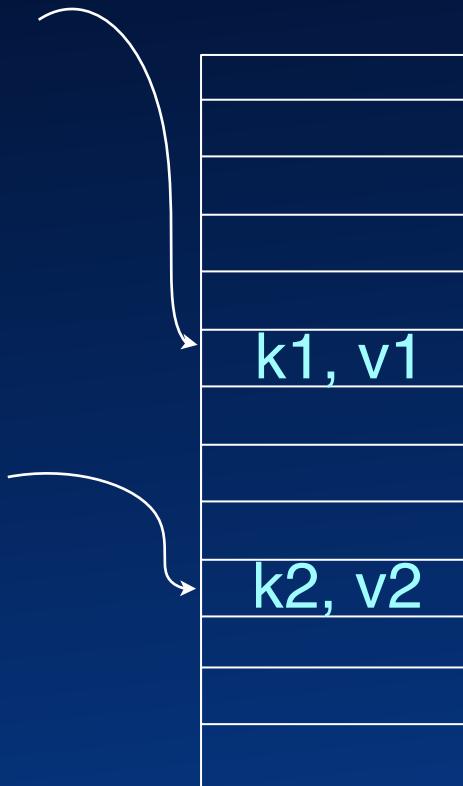
- Separate chaining
- Open Addressing





Collision Resolution

- Separate chaining



- Open Addressing

h_1
 h_2
 h_3

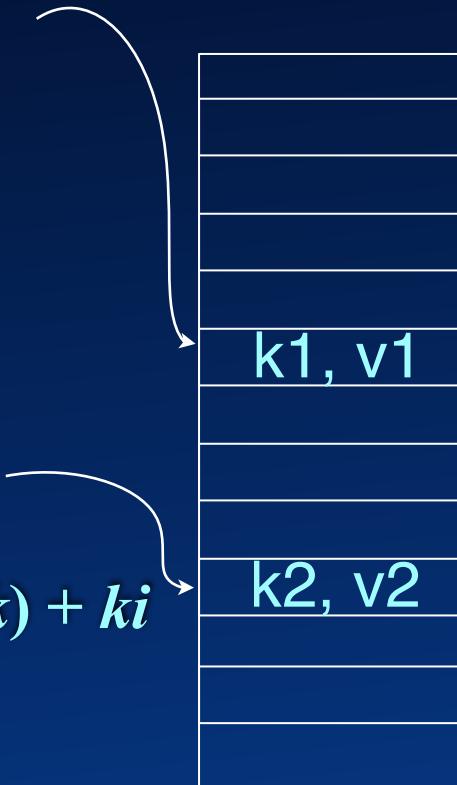
Rehash
until space located

Array



Collision Resolution

- Separate chaining



- Open Addressing

- Linear probing

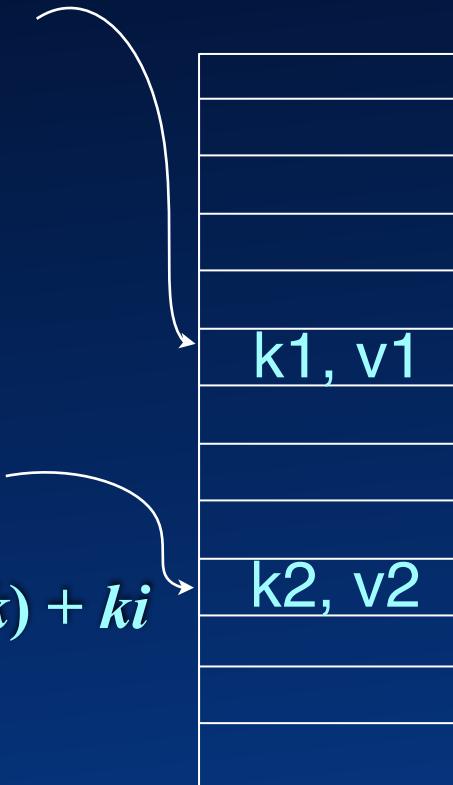
$$h_i(k) = h(k) + ki$$

h_1
 h_2
 h_3
Rehash
until space located
Array



Collision Resolution

- Separate chaining



- Open Addressing

- Linear probing

$$h_i(k) = h(k) + ki$$

h_1
 h_2
 h_3

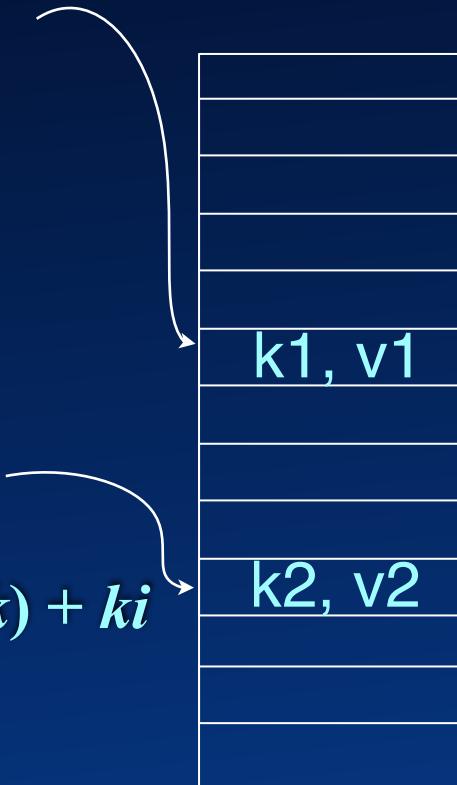
Rehash
until space located

Array size: N



Collision Resolution

- Separate chaining



- Open Addressing

- Linear probing

$$h_i(k) = h(k) + ki$$

h_1
 h_2
 h_3

Rehash
until space located

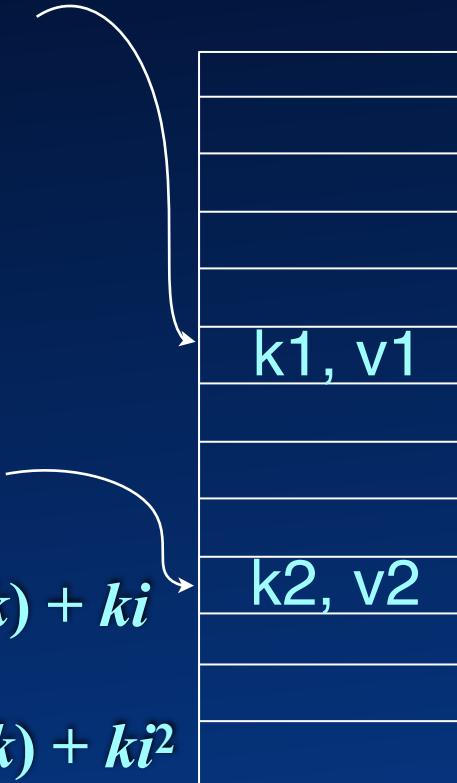
Array size: N

Index is "%N"
Choose Prime N



Collision Resolution

■ Separate chaining



■ Open Addressing

■ Linear probing

$$h_i(k) = h(k) + ki$$

■ Quadratic probing

$$h_i(k) = h(k) + ki^2$$

h_1

h_2

h_3

Rehash
until space located

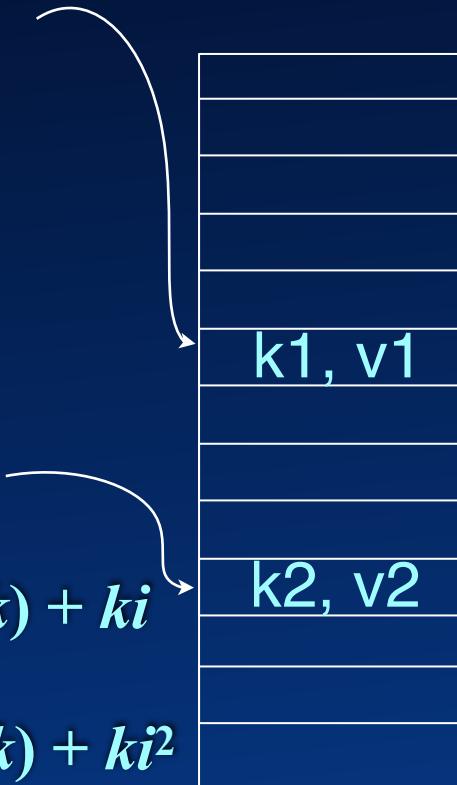
Array size: N

Index is "%N"
Choose Prime N



Collision Resolution

■ Separate chaining



■ Open Addressing

■ Linear probing

$$h_i(k) = h(k) + ki$$

■ Quadratic probing

$$h_i(k) = h(k) + ki^2$$

■ Double Hashing

$$h_i(k) = h(k) + h'(k)i$$

h_1

h_2

h_3

Rehash
until space located

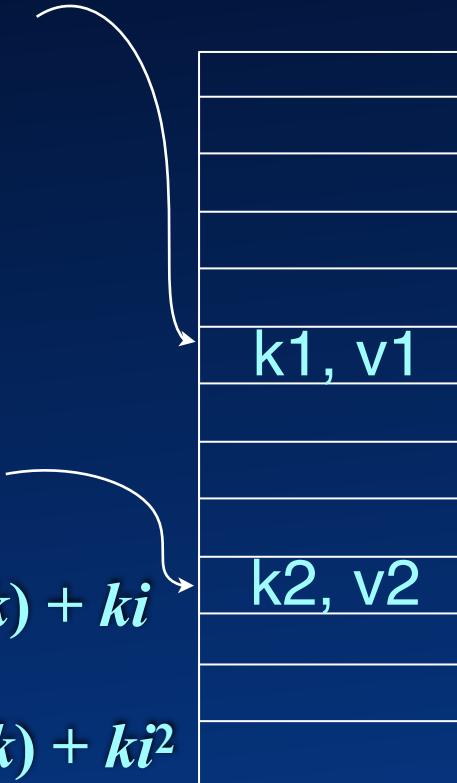
Array size: N

Index is "%N"
Choose Prime N



Collision Resolution

■ Separate chaining



■ Open Addressing

■ Linear probing

$$h_i(k) = h(k) + ki$$

■ Quadratic probing

$$h_i(k) = h(k) + ki^2$$

■ Double Hashing

$$h_i(k) = h(k) + h'(k)i$$

■ (Pseudo) Random probing

$$h_i(k) = h(k) + \text{random}(h(k), i)$$

h_1

h_2

h_3

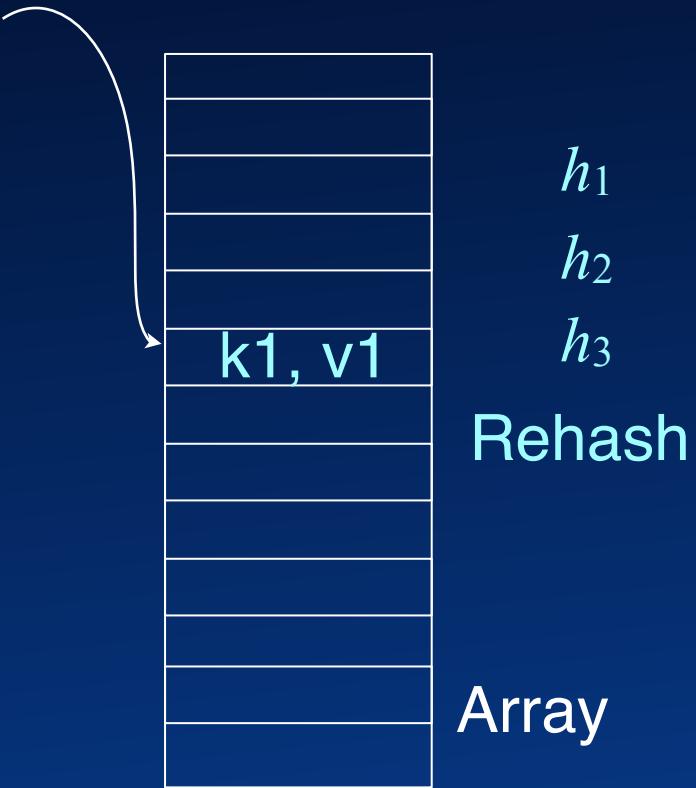
Rehash
until space located

Array size: N

Index is "%N"
Choose Prime N

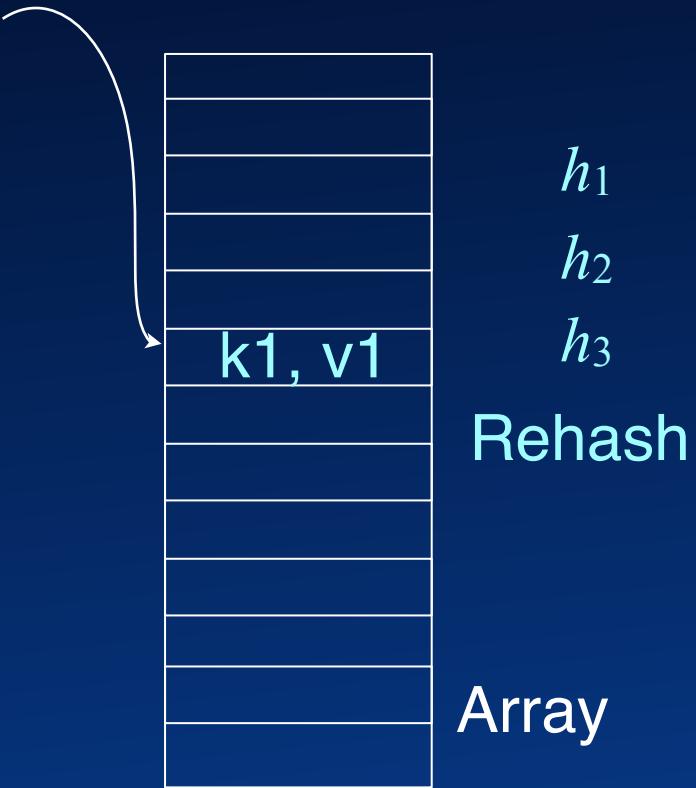


Cuckoo Hashing



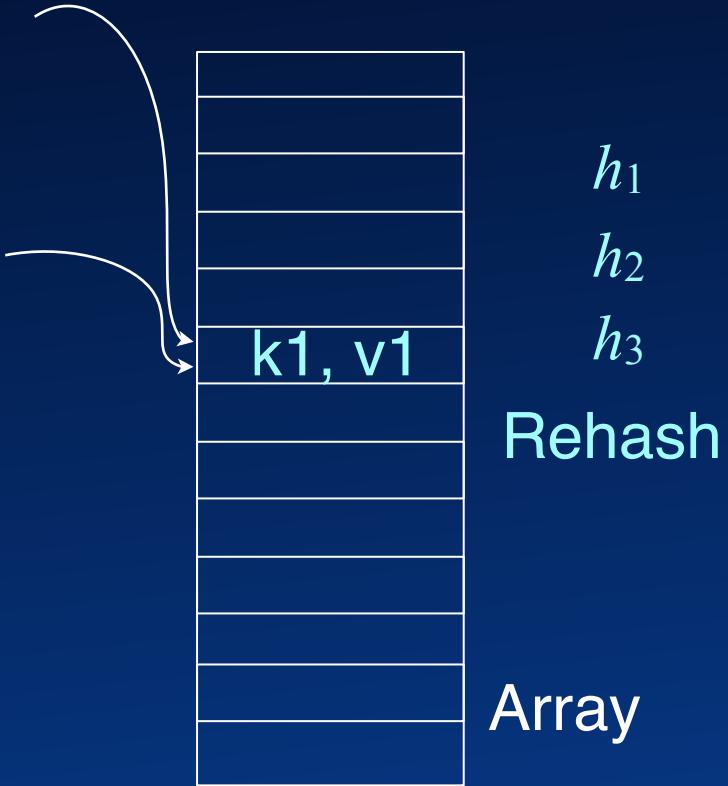


Cuckoo Hashing



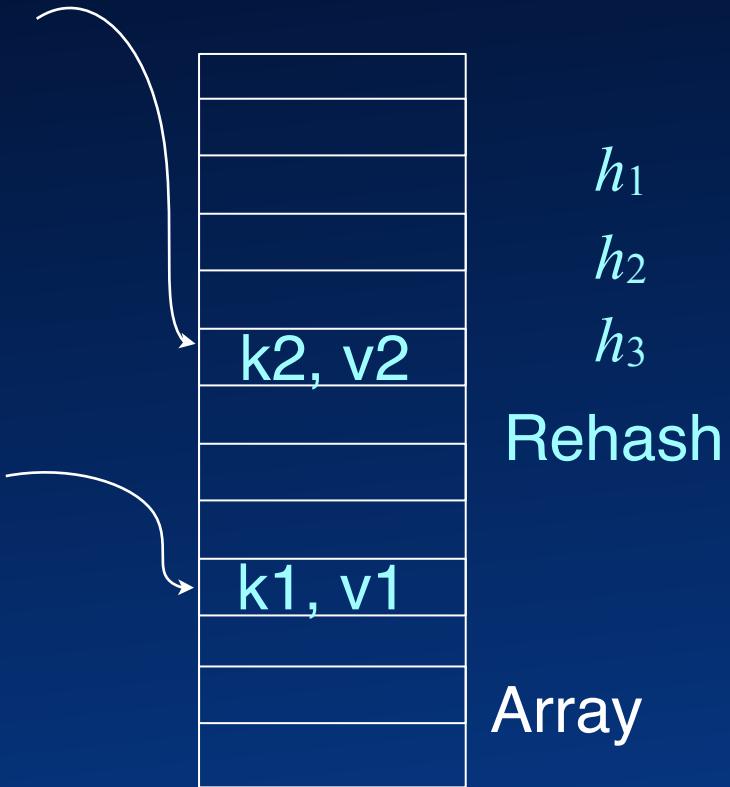


Cuckoo Hashing





Cuckoo Hashing





True or False?

- **Worst case query time for hash tables with separate chaining is $O(n^2)$**
- **Worst case query time for hash tables with linear probing is $\Theta(n)$**
- **Worst case insertion time for hash tables with separate chaining is $O(1)$**
- **Worst case deletion time for hash tables with separate chaining is $O(1)$**



Hash of Integer

■ $h(\text{Integer } i)$

- $[(a_0 i + a_1) \% P] \% N$
- P is a large prime $> N$
- $(a_0, a_1) \in [0 .. P-1]$



Hash Functions

- Middle r -bits
- Add them up
- Polynomial function



- Cyclic shift and add

$r+1$ parts

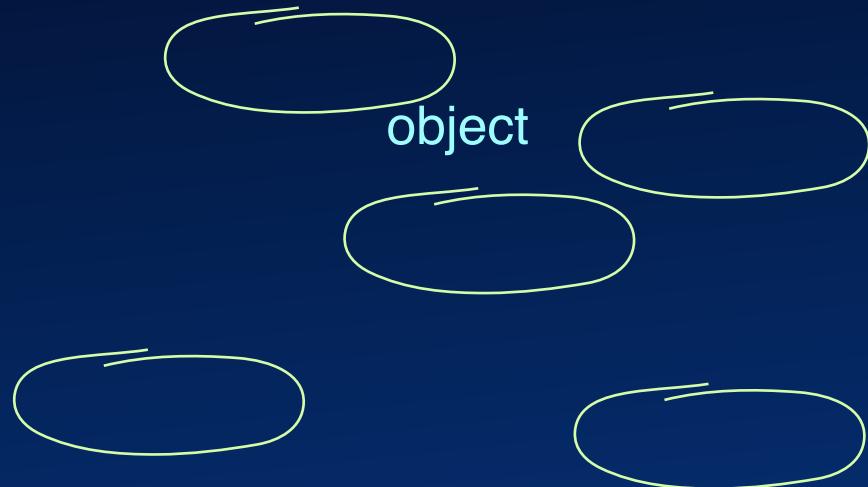
- Universal hashing

0101000011100110111



Hash Functions

- Middle r -bits
- Add them up
- Polynomial function



- Cyclic shift and add

$r+1$ parts

- Universal hashing

0101000011100110111

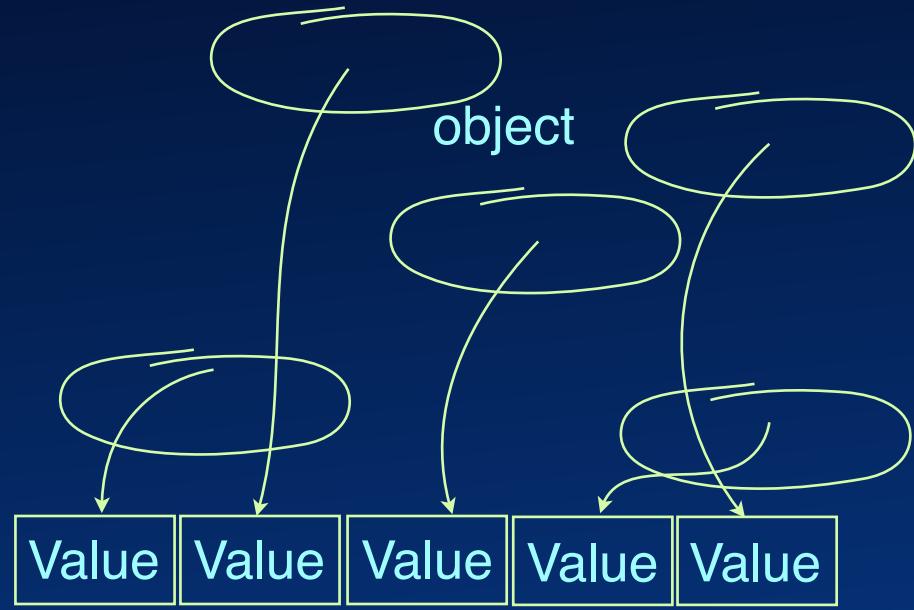


Hash Functions

- Middle r-bits
- Add them up
- Polynomial function

- Cyclic shift and add

- Universal hashing



0101000011100110111



Hash Functions

- Middle r-bits
- Add them up
- Polynomial function



- Cyclic shift and add

- Universal hashing

0101000011100110111

Value



Hash Functions

- Middle r-bits
- Add them up
- Polynomial function

With prime a : $h_a(k) = \sum_{i=0}^r a^i k_i \% N$



- Cyclic shift and add

- Universal hashing

0101000011100110111

Value



Hash Functions

- Middle r-bits
- Add them up
- Polynomial function

With prime a : $h_a(k) = \sum_{i=0}^r a^i k_i \% N$



- Cyclic shift and add

$$h = 0$$

$$h += \text{Cyclic Shift}(h) + k_i$$

Value

0101000011100110111

- Universal hashing



Hash Functions

- Middle r-bits
- Add them up
- Polynomial function

With prime a : $h_a(k) = \sum_{i=0}^r a^i k_i \% N$



- Cyclic shift and add

$$h = 0$$

$$h += \text{Cyclic Shift}(h) + k_i$$

Value

01010000011100110111

- Universal hashing



Hash Functions

- Middle r-bits
- Add them up
- Polynomial function

With prime a : $h_a(k) = \sum_{i=0}^r a^i k_i \% N$



- Cyclic shift and add

$$h = 0$$

$$h += \text{Cyclic Shift}(h) + k_i$$

Value

0000110011011101010

- Universal hashing



Hash Functions

- Middle r-bits
- Add them up
- Polynomial function

With prime a : $h_a(k) = \sum_{i=0}^r a^i k_i \% N$



- Cyclic shift and add

$$h = 0$$

$$h += \text{Cyclic Shift}(h) + k_i$$

Value

00001110011011101010

- Universal hashing



Universal Hash Function

- Choose prime N
- Divide key k into $r+1$ parts

$k_0, k_1, \dots k_r$ s.t. $\max(k_i) < N$

- Choose all possible hash functions

$$h_a(k) = \sum_{i=0}^r a_i k_i \% N$$

where $a_0, a_1, \dots a_r$, are each in $\{0 \dots N-1\}$

- There are N^{r+1} unique hash functions
- At hash creation time, randomly choose a
 - and let $h = h_a$
 - No need to explicitly enumerate the hash functions

Probability ($h(k1) = h(k2)$) = $1/N$



True or False?

- **Expected query time for hash tables with load factor = 0.5 is O(1) for separate chaining.**
- **Expected query time for hash tables with load factor = 0.5 is O(1) for open addressing.**
- **Worst case deletion time for a hash table is O(1) if cuckoo hashing (open addressing) is used.**



Ordered Keys

- Put it in a sorted list
 - Insertion sort





Ordered Keys

- Put it in a sorted list
 - Insertion sort



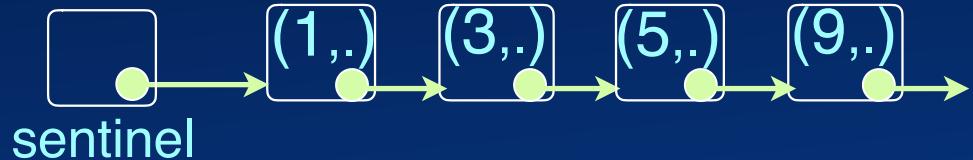


Ordered Keys

- Put it in a sorted list
 - Insertion sort



Insert 8



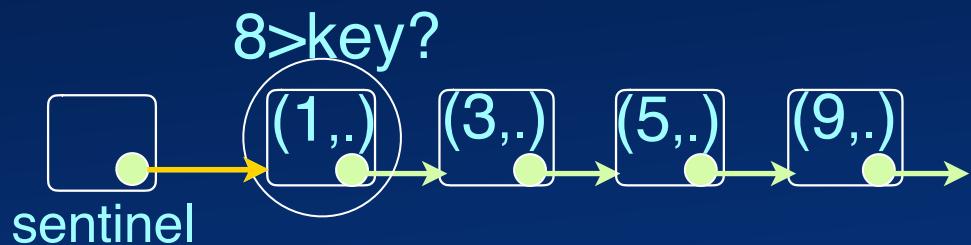


Ordered Keys

- Put it in a sorted list
 - Insertion sort



Insert 8



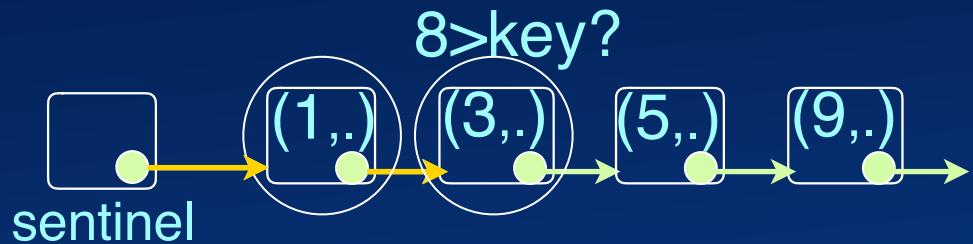


Ordered Keys

- Put it in a sorted list
 - Insertion sort



Insert 8



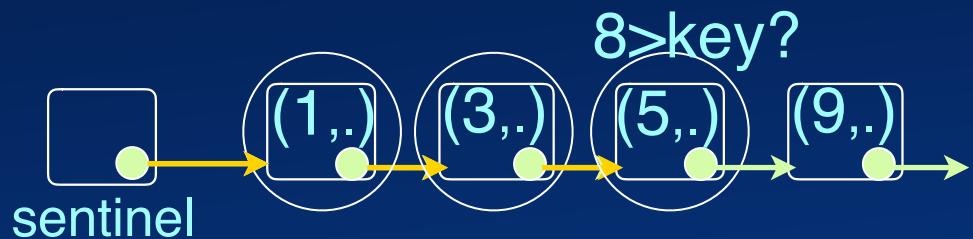


Ordered Keys

- Put it in a sorted list
 - Insertion sort



Insert 8



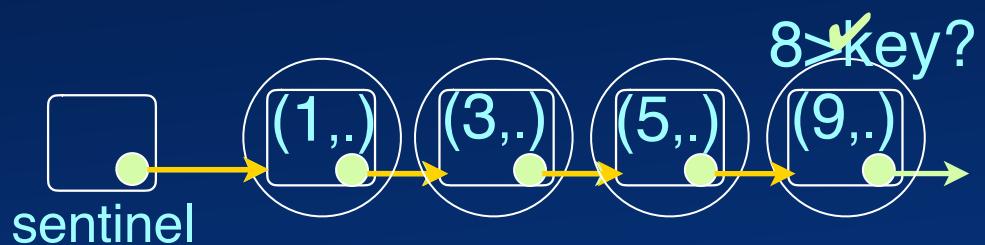


Ordered Keys

- Put it in a sorted list
 - Insertion sort



Insert 8



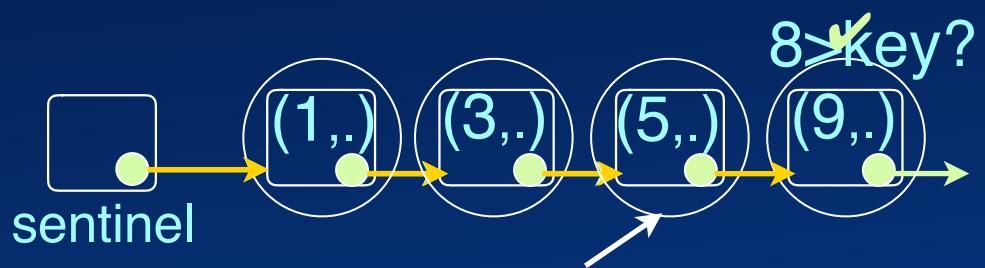


Ordered Keys

- Put it in a sorted list
 - Insertion sort



Insert 8



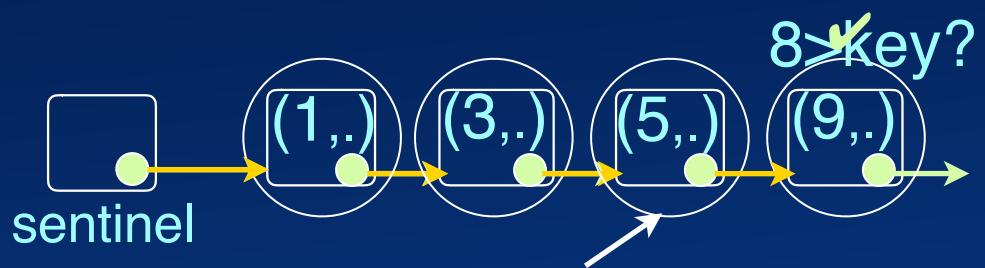


Ordered Keys

- Put it in a sorted list
 - Insertion sort



Insert 8



```
node p = sentinel;  
node n = sentinel.next;  
while ( n.key < k) {  
    p = n; n = n.next;  
}  
p.insertAfter(p, Pair(k, v))
```

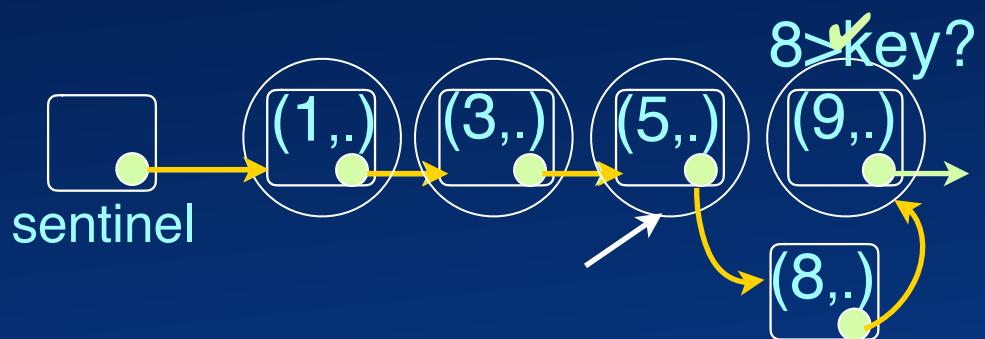


Ordered Keys

- Put it in a sorted list
 - Insertion sort



Insert 8



```
node p = sentinel;  
node n = sentinel.next;  
while ( n.key < k) {  
    p = n; n = n.next;  
}  
p.insertAfter(p, Pair(k, v))
```

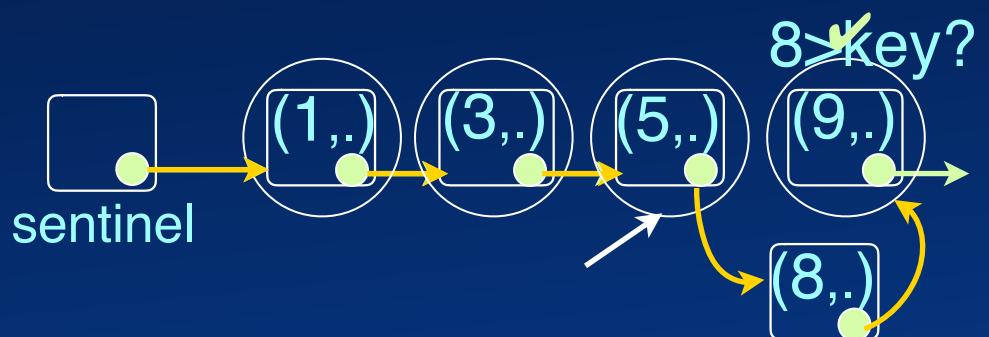


Ordered Keys

- Put it in a sorted list
 - Insertion sort



Insert 8



```
node p = sentinel;  
node n = sentinel.next;  
while ( n != null && n.key < k) {  
    p = n; n = n.next;  
}  
p.insertAfter(p, Pair(k, v))
```

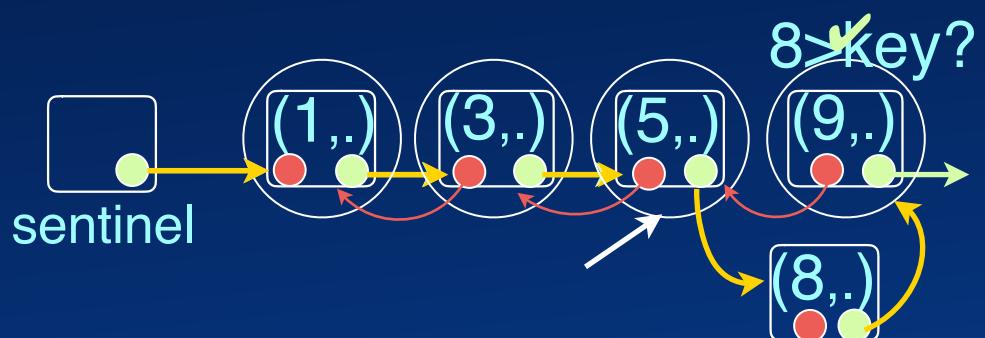


Ordered Keys

- Put it in a sorted list
 - Insertion sort



Insert 8



```
node p = sentinel;  
node n = sentinel.next;  
while ( n != null && n.key < k) {  
    p = n; n = n.next;  
}  
p.insertAfter(p, Pair(k, v))
```



Skip-list

∞

∞

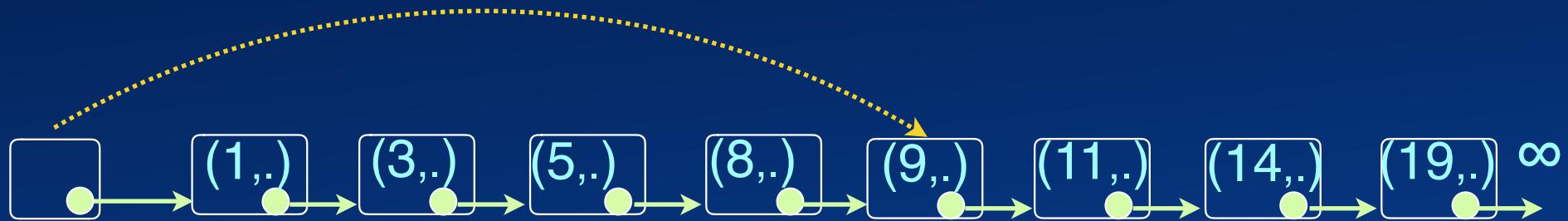




Skip-list

∞

∞

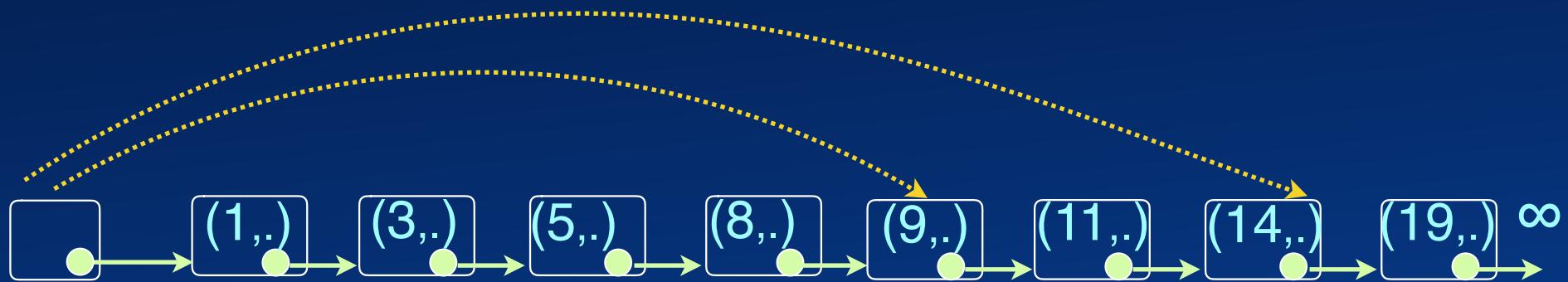




Skip-list

∞

∞

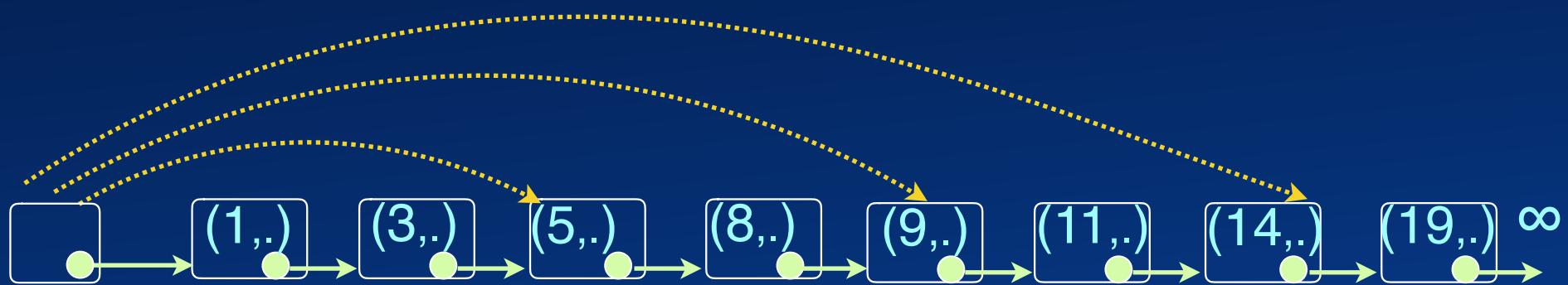




Skip-list

∞

∞

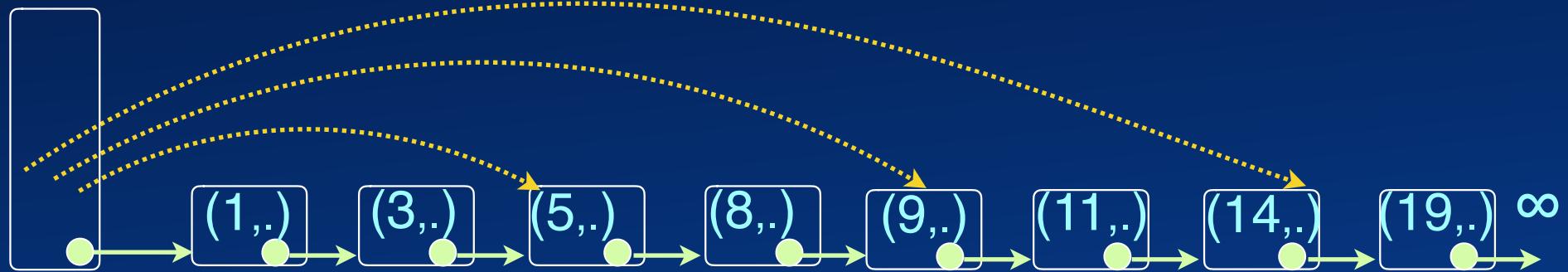




Skip-list

∞

∞

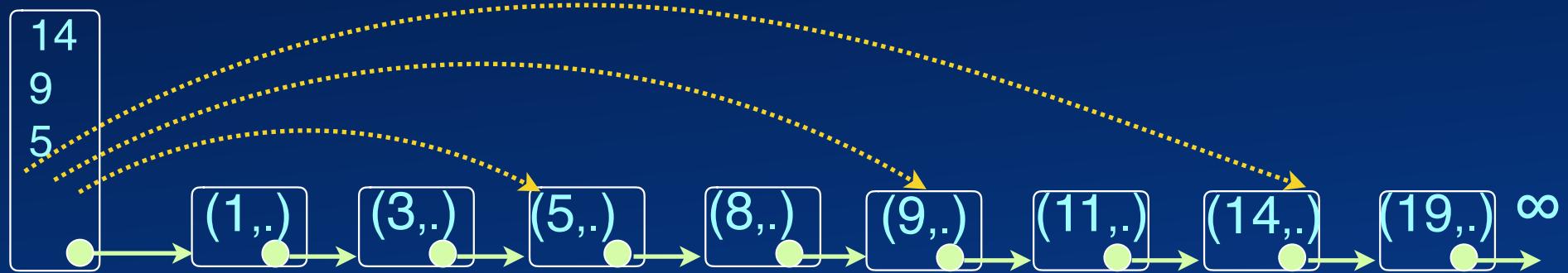




Skip-list

∞

∞





Skip-list

∞

∞





Skip-list

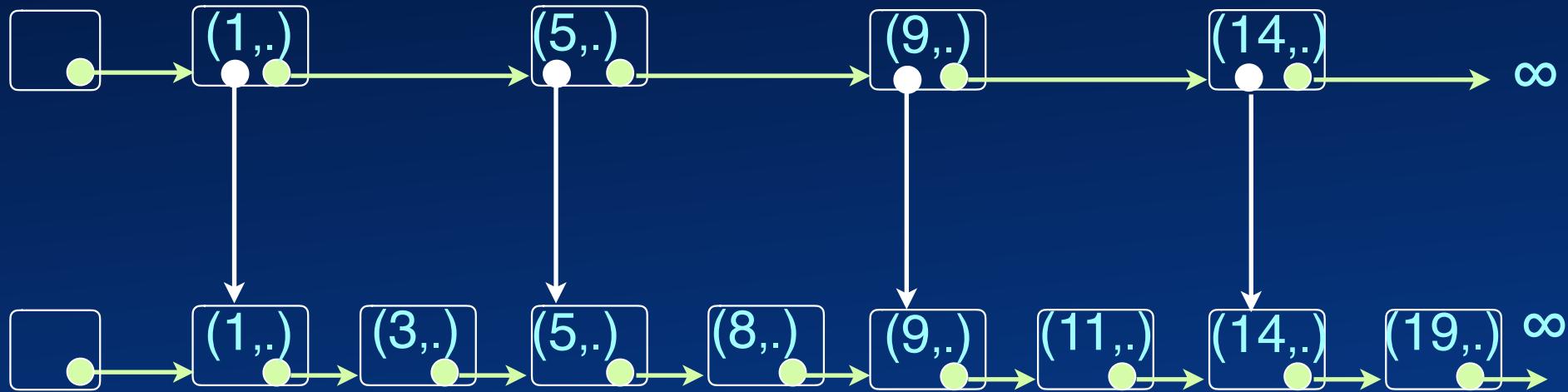
∞





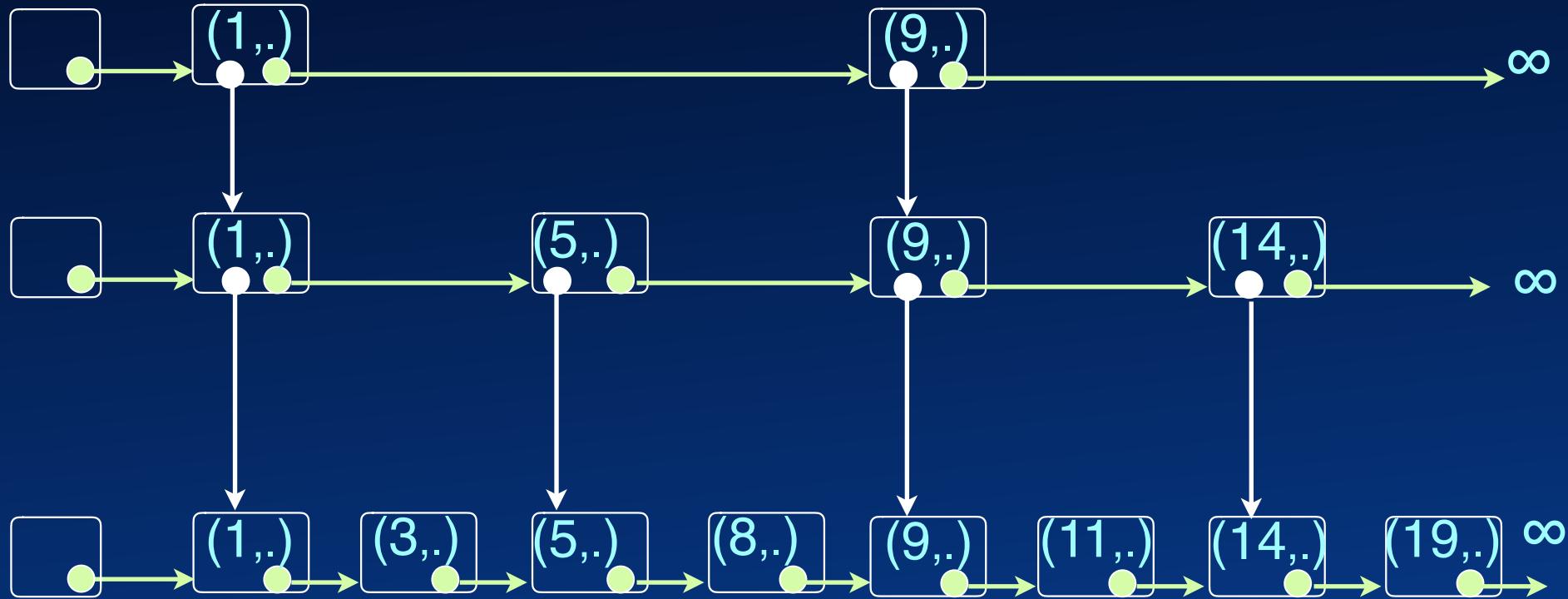
Skip-list

∞



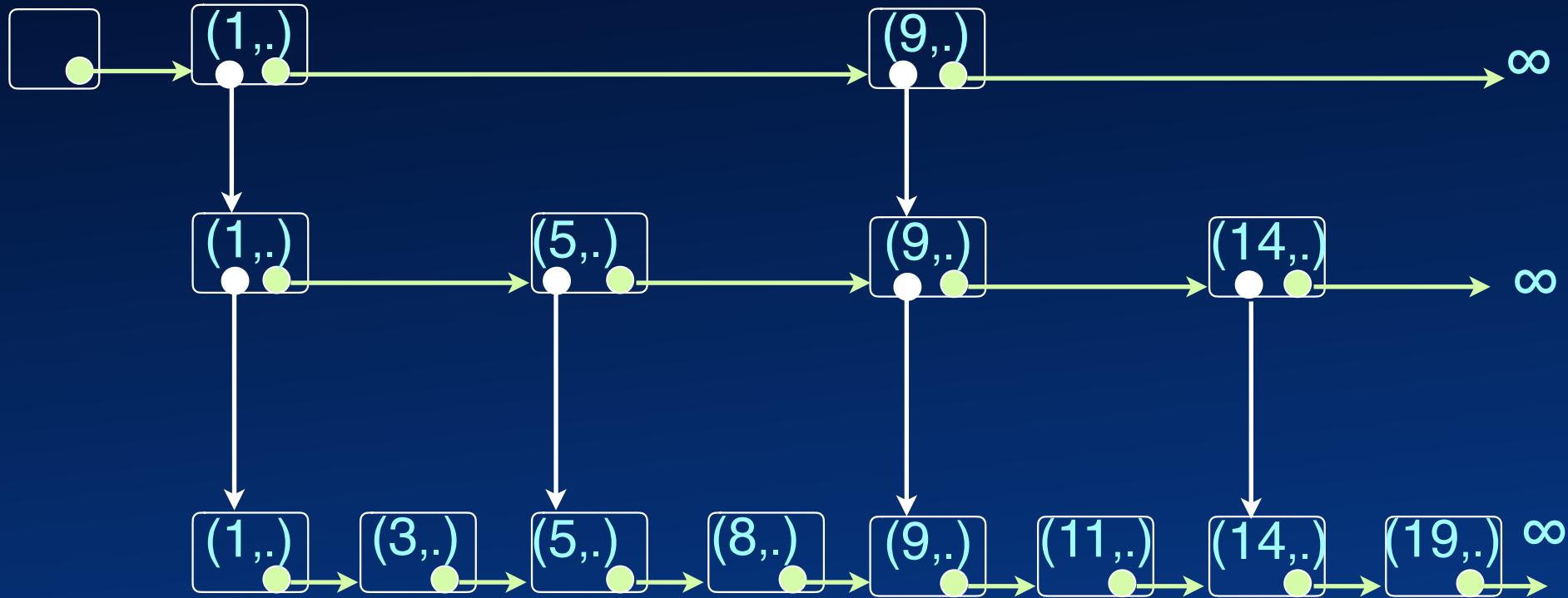


Skip-list



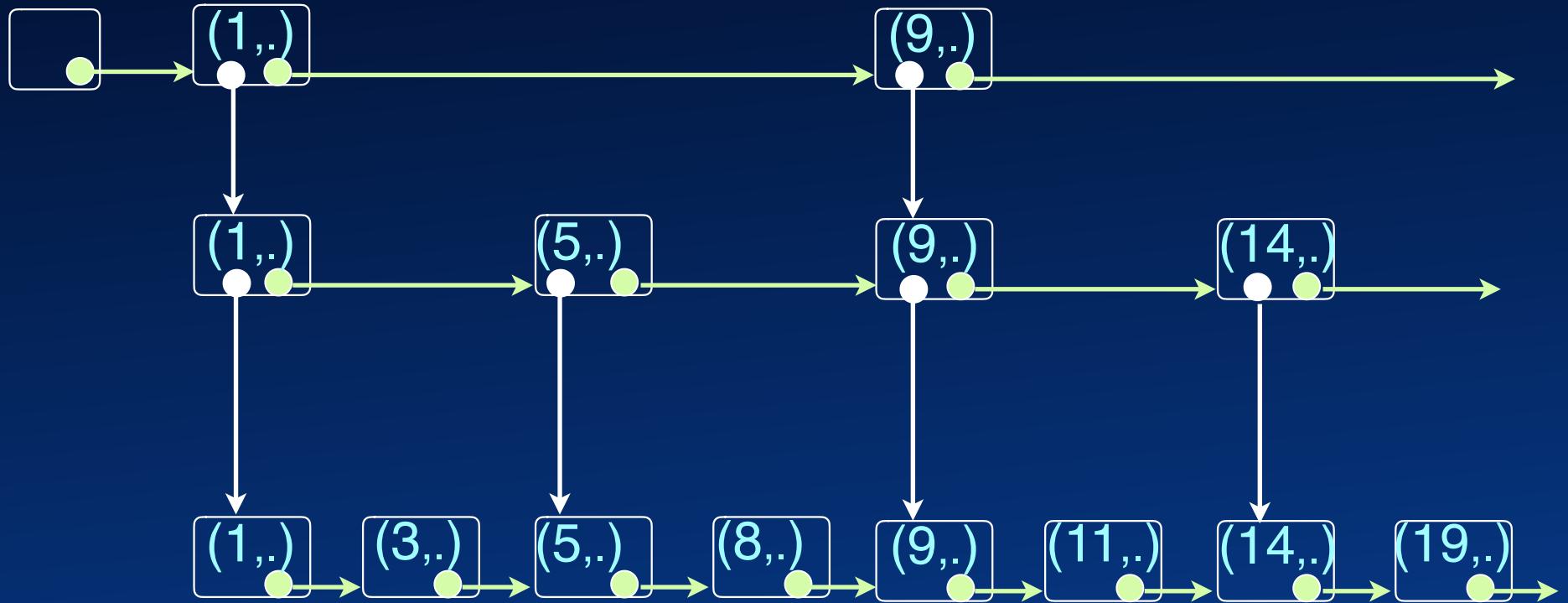


Skip-list



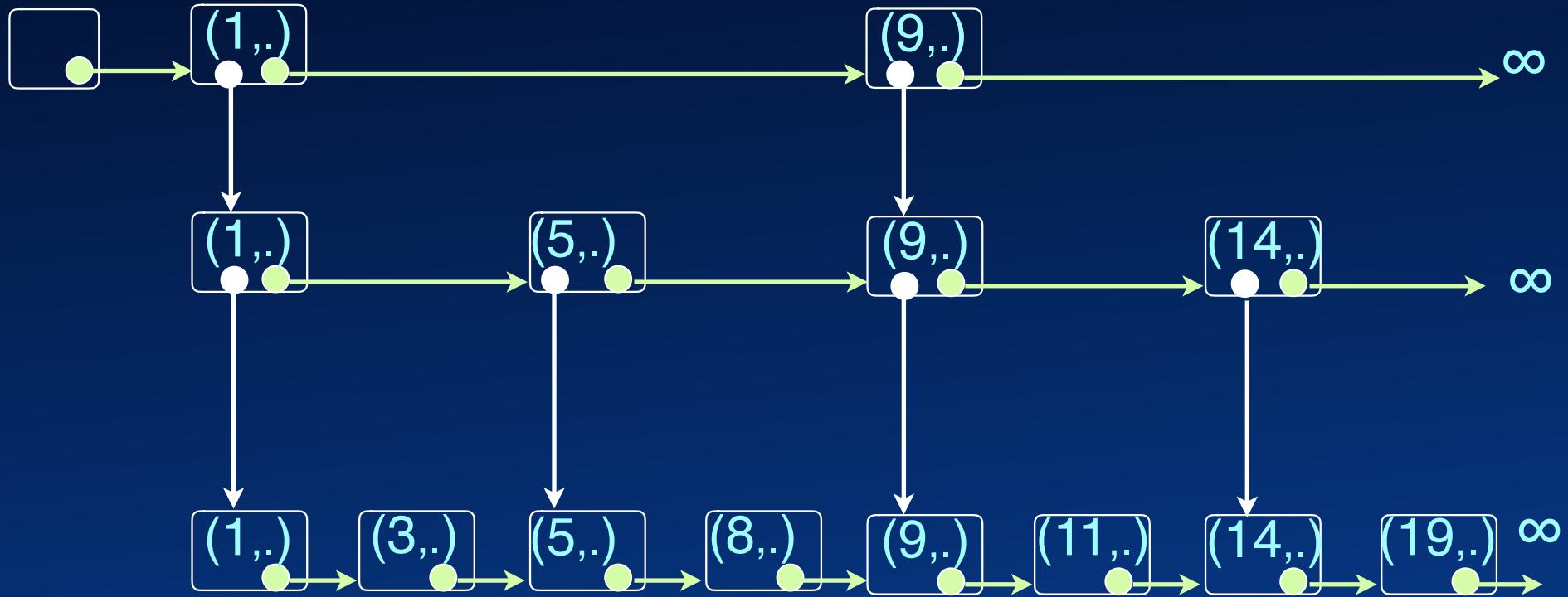


Skip-list



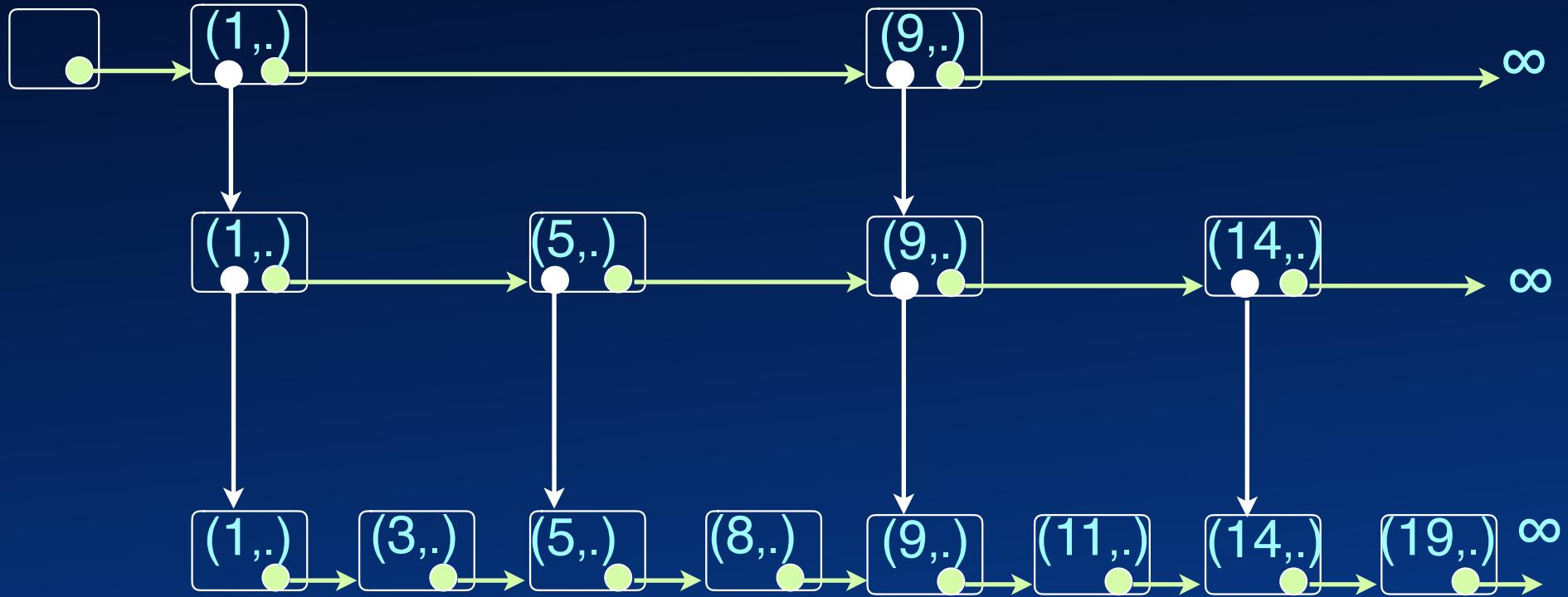


Skip-list





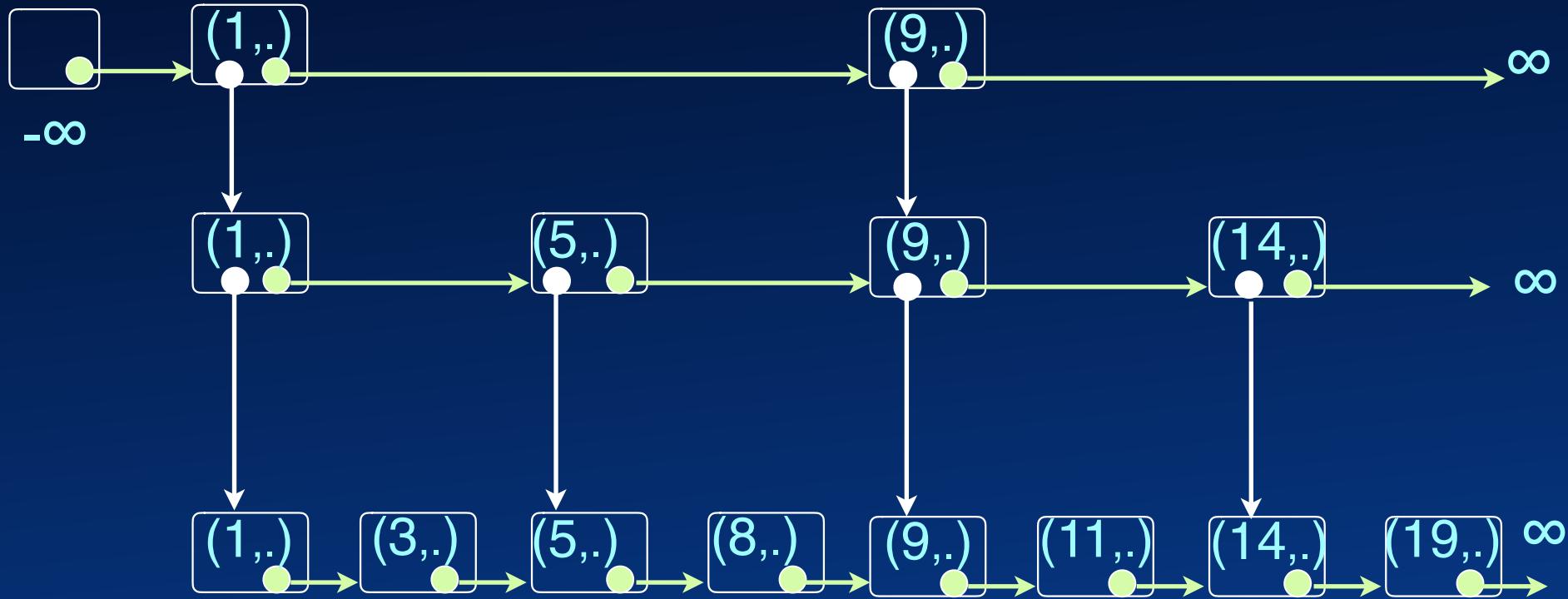
Skip-list



Find 10



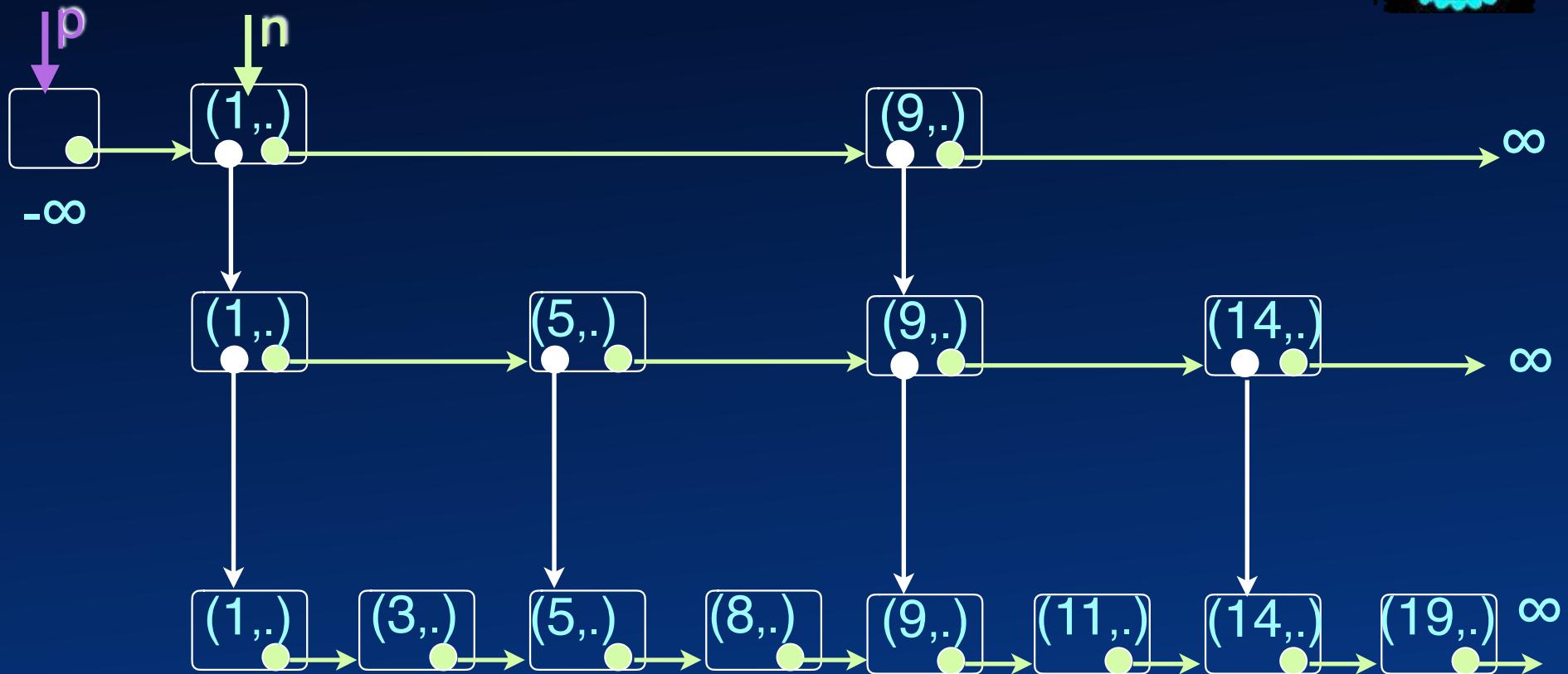
Skip-list



Find 10



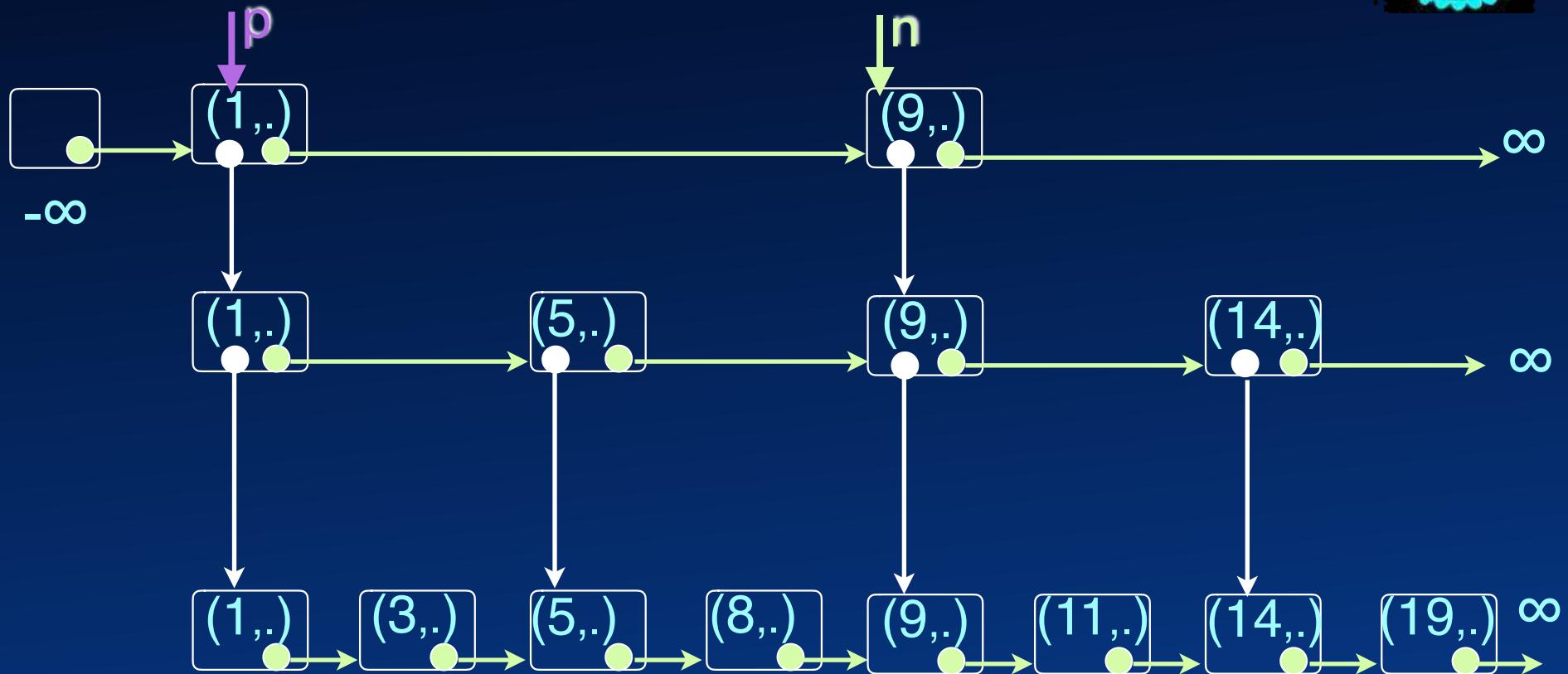
Skip-list



Find 10



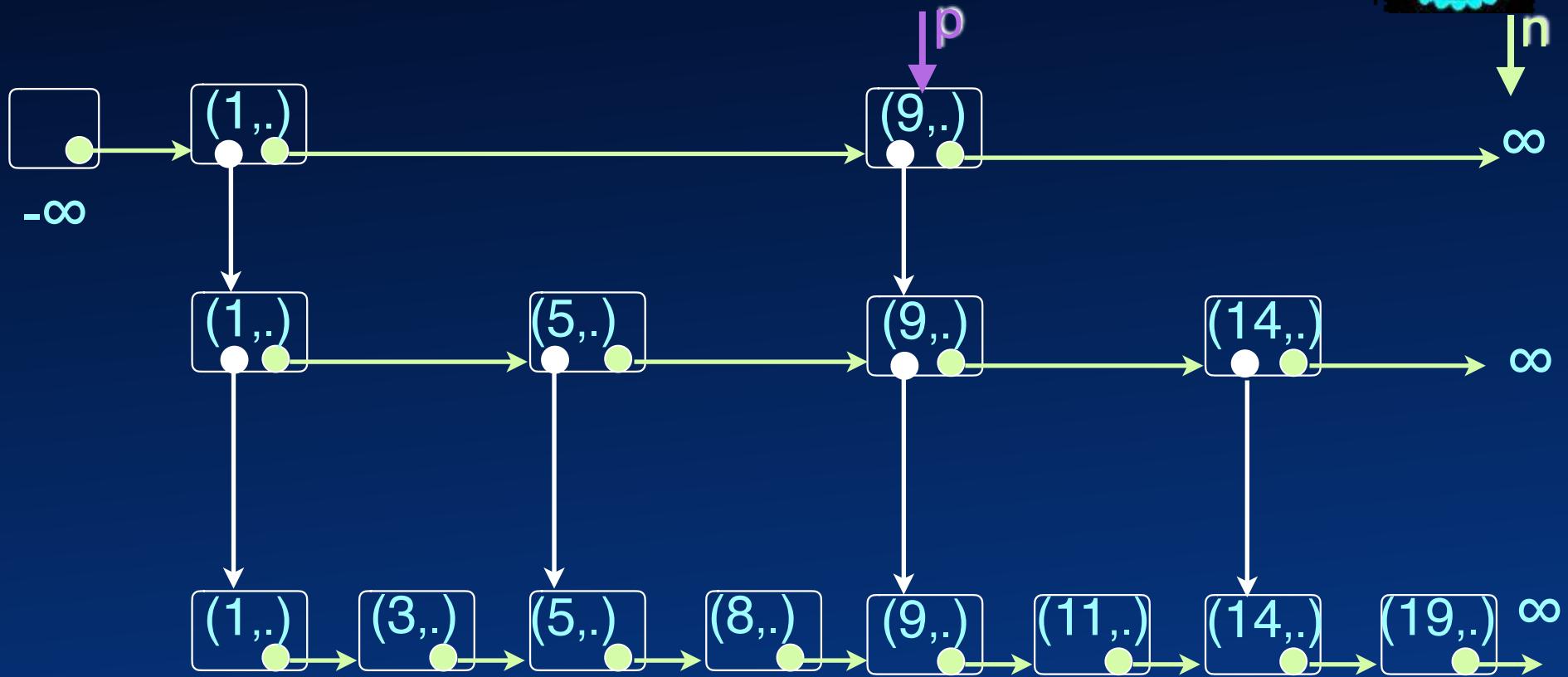
Skip-list



Find 10



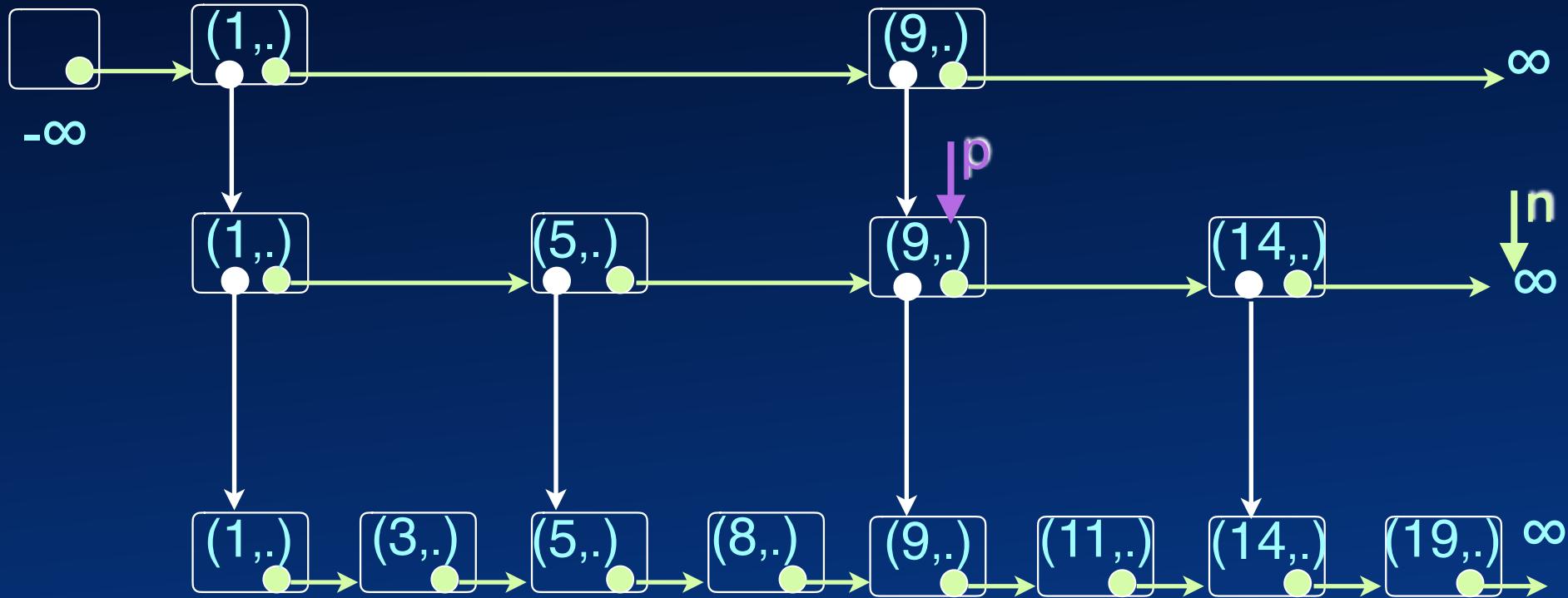
Skip-list



Find 10



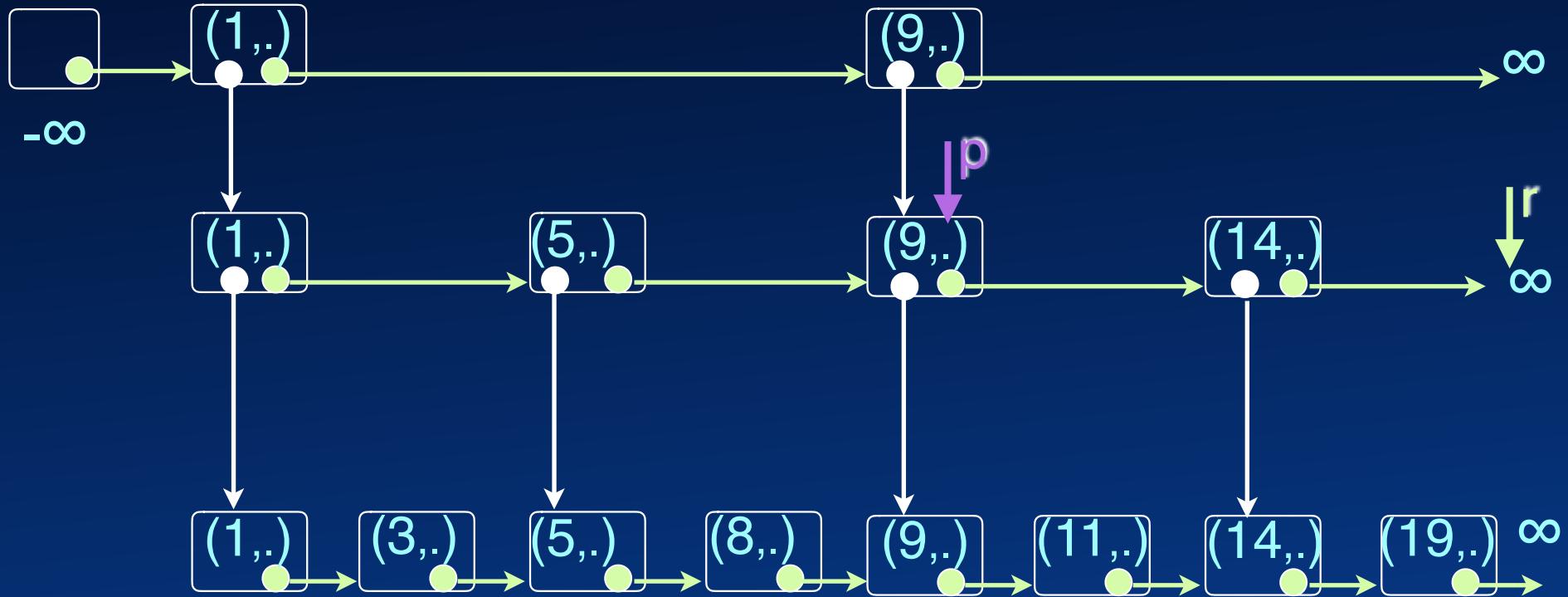
Skip-list



Find 10



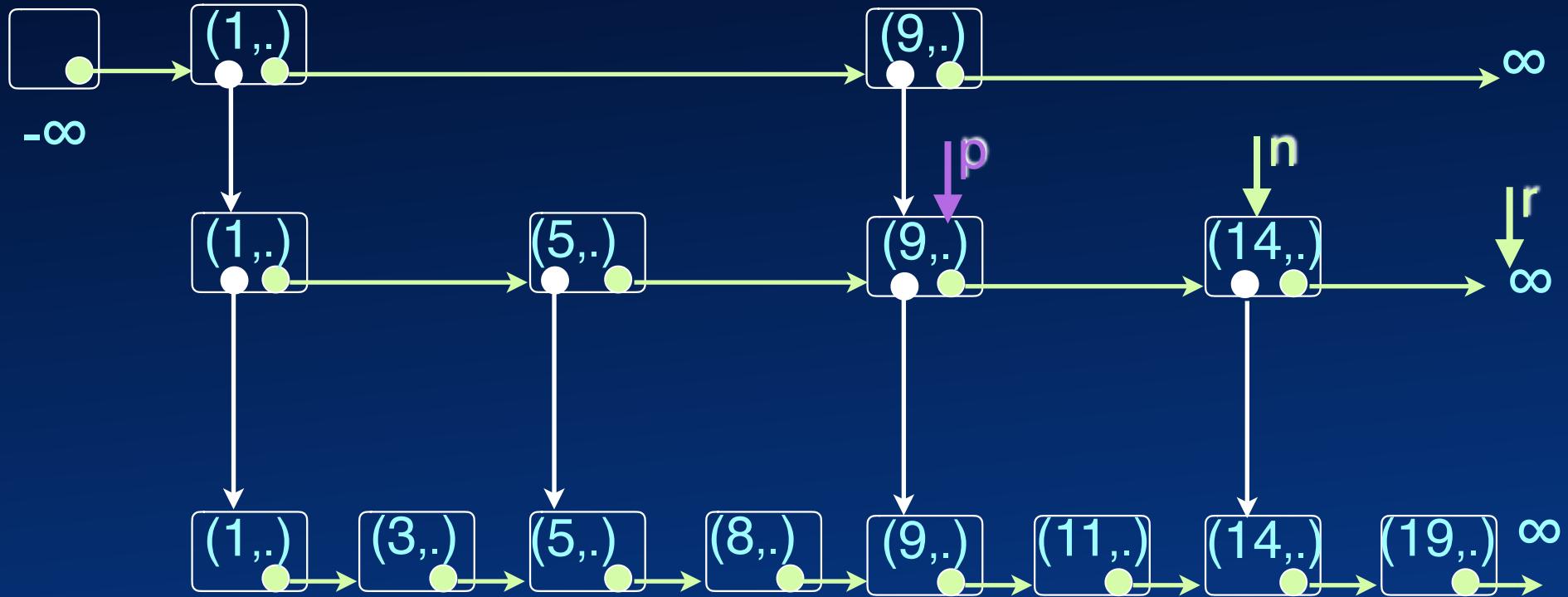
Skip-list



Find 10



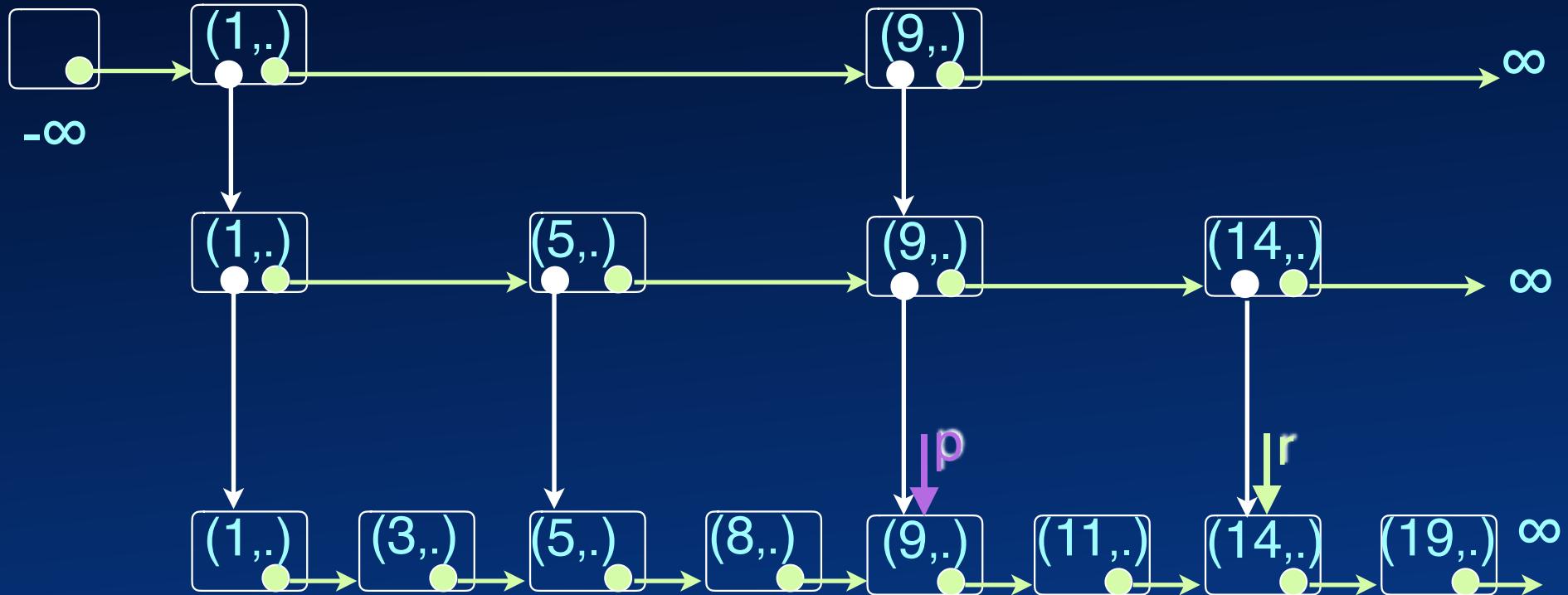
Skip-list



Find 10



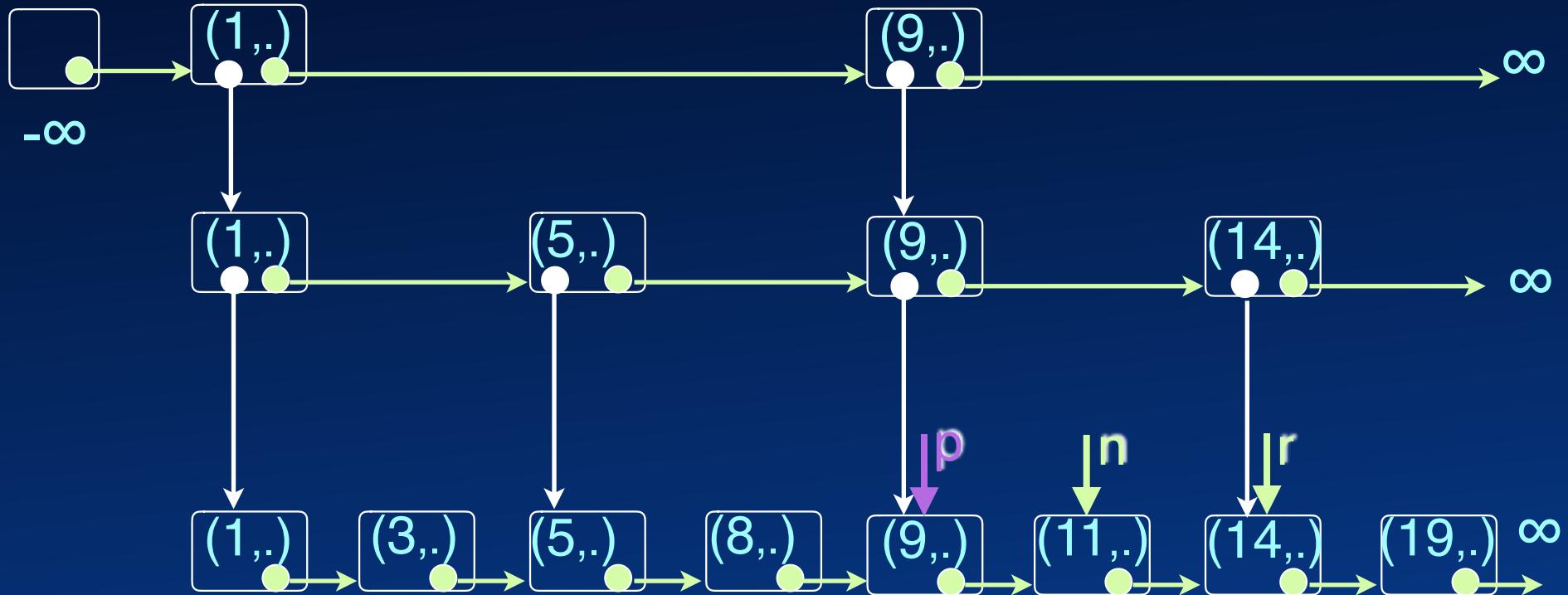
Skip-list



Find 10



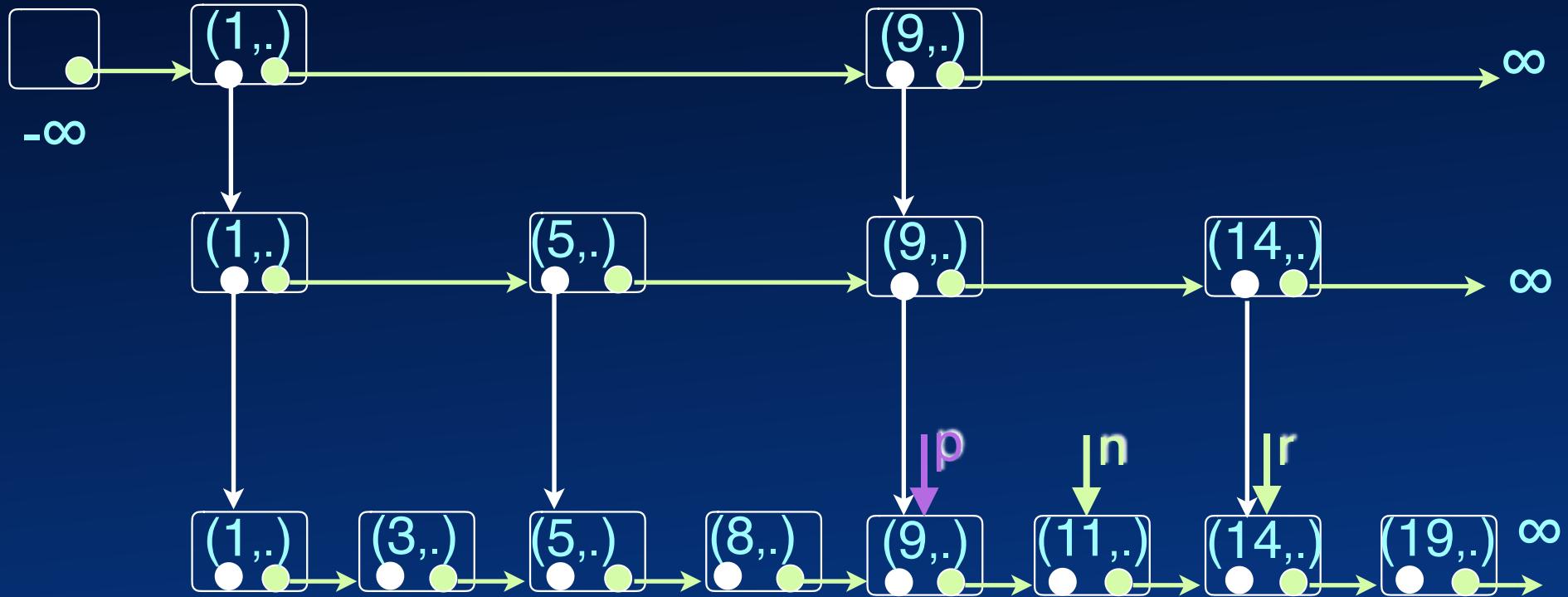
Skip-list



Find 10



Skip-list

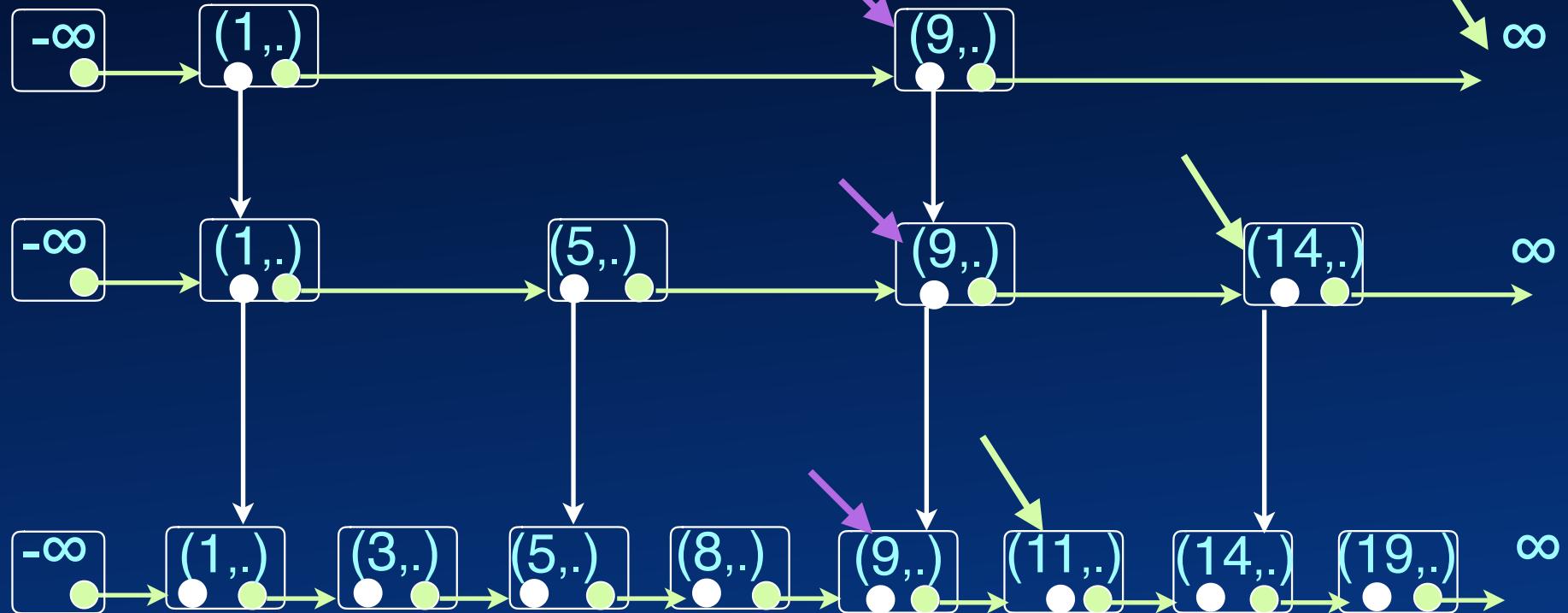


Find 10



Skip-list

sentinel

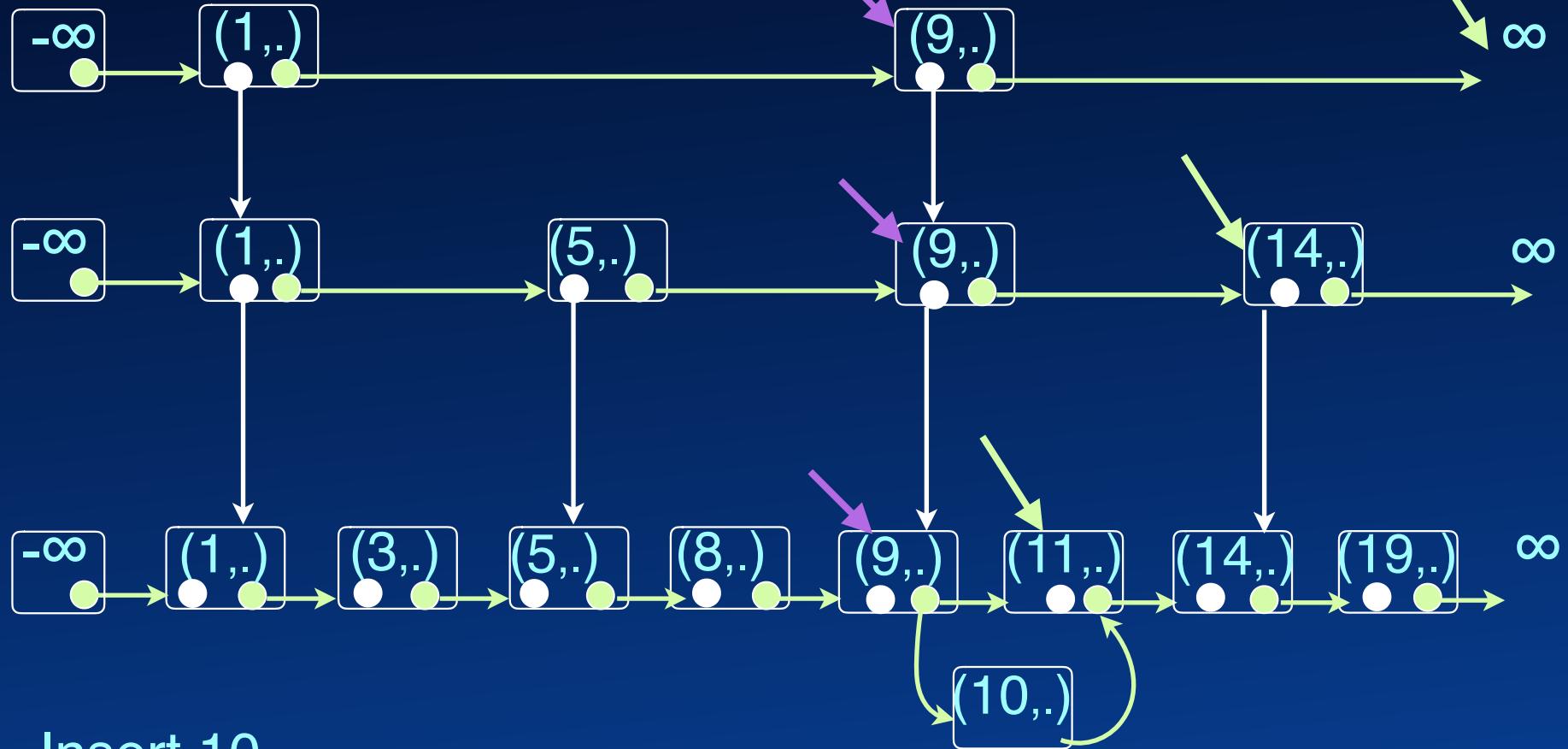


Insert 10



Skip-list

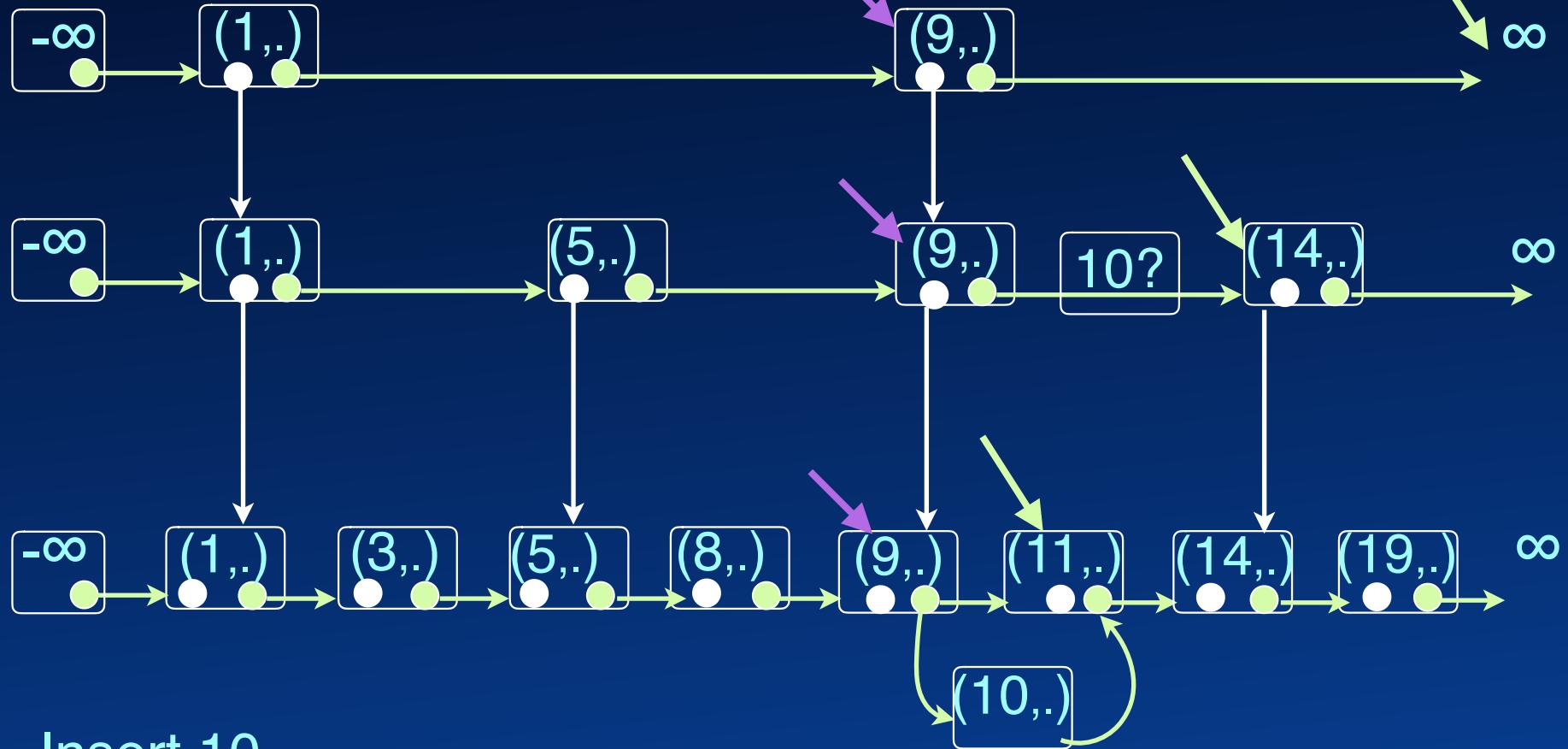
sentinel





Skip-list

sentinel

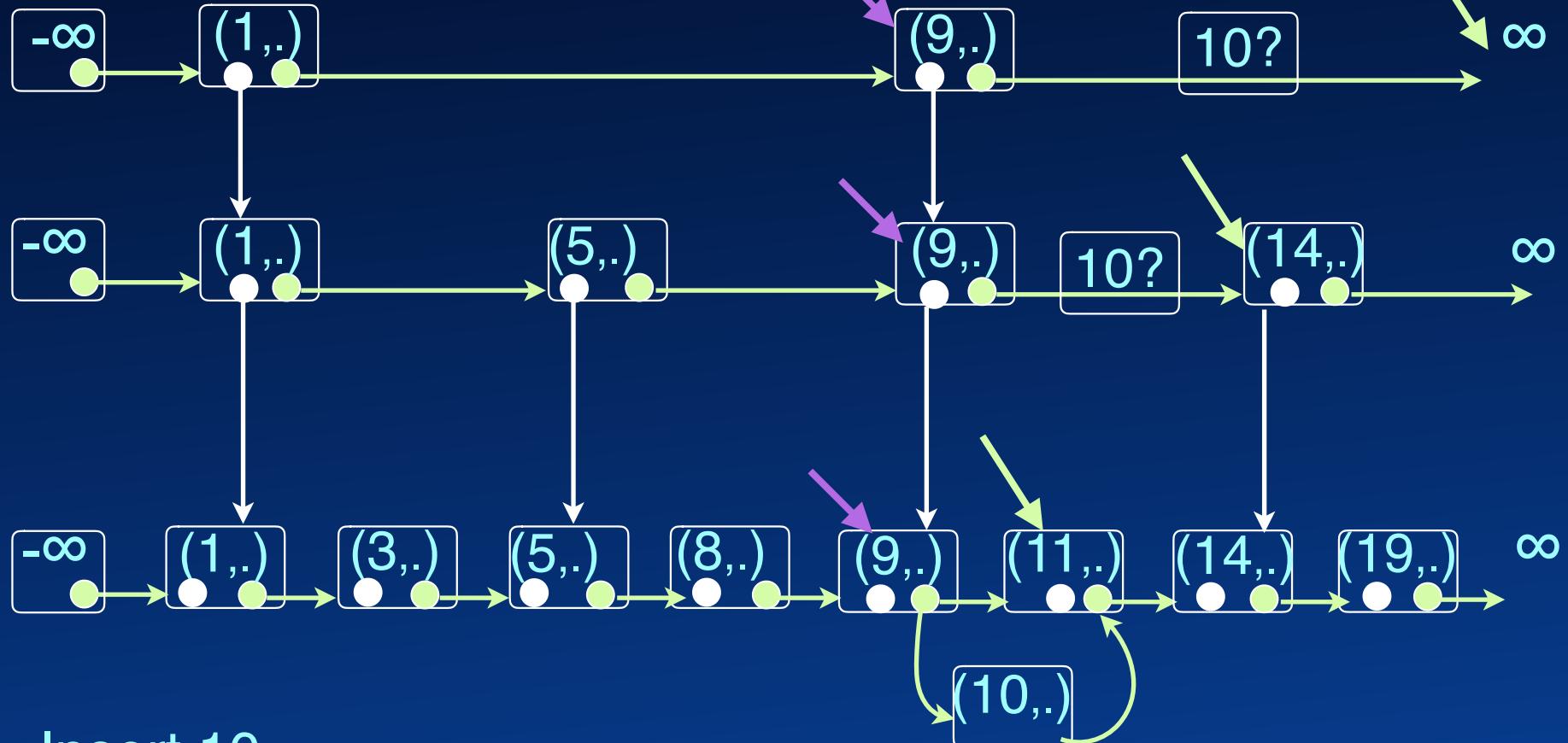


Insert 10



Skip-list

sentinel





Search in a Skip List

search(n, k):

```
node p = n;  
n = n.next;  
while (n != null && n.key < k) {  
    p = n; n = n.next;  
}  
if(n.key == k) return “answer”;  
search(p.below, k);
```



Search in a Skip List

```
search(n, k):  
    if(n == null) return “Fail”;  
    node p = n;  
    n = n.next;  
    while (n != null && n.key < k) {  
        p = n; n = n.next;  
    }  
    if(n.key == k) return “answer”;  
    search(p.below, k);
```



Search in a Skip List

```
search(n, k):  
    if(n == null) return “Fail”;  
    node p = n;  
    n = n.next;  
    while (n != null && n.key < k) {  
        p = n; n = n.next;  
    }  
    if(n.key == k) return “answer”;  
    search(p.below, k);
```

```
search(sentinel, k);
```



Search in a Skip List

```
search(n, k):  
    if(n == null) return “Fail”;  
    node p = n;  
    n = n.next;  
    while (n != null && n.key < k) {  
        p = n; n = n.next;  
    }  
    if(n != null && n.key == k) return “answer”;  
    search(p.below, k);  
  
search(sentinel, k);
```



Insert in a Skip List

```
insert(n, k, topinsert):  
    if(n == null) return;  
    node p = n;  
    n = n.next;  
    while (n != null && n.key < k) {  
        p = n; n = n.next;  
    } // Handle duplicate if necessary  
    if(level < h)  
        insertafternode(p, k);  
    insert(p.below, k, topinsert);
```

```
topinsert = toss_coins_until_tails(maxtosses);  
insert(sentinel, k, topinsert);
```



Insert in a Skip List

```
insert(n, k, topinsert):  
    if(n == null) return;  
    node p = n;  
    n = n.next;  
    while (n != null && n.key < k) {  
        p = n; n = n.next;  
    } // Handle duplicate if necessary  
    if(level < h)  
        insertafternode(p, k);  
    insert(p.below, k, topinsert);
```

```
topinsert = toss_coins_until_tails(maxtosses);  
insert(sentinel, k, topinsert);
```



Insert in a Skip List

```
insert(n, k, topinsert):  
    if(n == null) return;  
    node p = n;  
    n = n.next;  
    while (n != null && n.key < k) {  
        p = n; n = n.next;  
    } // Handle duplicate if necessary  
    if(level < h)  
        insertafternode(p, k); What about the below ref?  
    insert(p.below, k, topinsert);
```

```
topinsert = toss_coins_until_tails(maxtosses);  
insert(sentinel, k, topinsert);
```



Insert in a Skip List

```
insert(n, k, topinsert):  
    if(n == null) return;  
    node p = n;  
    n = n.next;  
    while (n != null && n.key < k) {  
        p = n; n = n.next;  
    } // Handle duplicate if necessary  
    if(level < h)  
        insertafternode(p, k); What about the below ref?  
    insert(p.below, k, topinsert);  
    add below ref
```

```
topinsert = toss_coins_until_tails(maxtosses);  
insert(sentinel, k, topinsert);
```



Insert in a Skip List

```
insert(n, k, topinsert, level):  
    if(n == null) return;  
    node p = n;  
    n = n.next;  
    while (n != null && n.key < k) {  
        p = n; n = n.next;  
    } // Handle duplicate if necessary  
    if(level < h)  
        insertafternode(p, k); What about the below ref?  
    insert(p.below, k, topinsert);  
    add below ref
```

```
topinsert = toss_coins_until_tails(maxtosses);  
insert(sentinel, k, topinsert);
```



Insert in a Skip List

```
insert(n, k, topinsert, level):  
    if(n == null) return;  
    node p = n;  
    n = n.next;  
    while (n != null && n.key < k) {  
        p = n; n = n.next;  
    } // Handle duplicate if necessary  
    if(level < h)  
        insertafternode(p, k); What about the below ref?  
    insert(p.below, k, topinsert, level-1);  
    add below ref
```

```
topinsert = toss_coins_until_tails(maxtosses);  
insert(sentinel, k, topinsert);
```



Insert in a Skip List

```
insert(n, k, topinsert, level):  
    if(n == null) return;  
    node p = n;  
    n = n.next;  
    while (n != null && n.key < k) {  
        p = n; n = n.next;  
    } // Handle duplicate if necessary  
    if(level < h)  
        insertafternode(p, k); What about the below ref?  
    insert(p.below, k, topinsert, level-1);  
    add below ref
```

```
topinsert = toss_coins_until_tails(maxtosses);  
insert(sentinel, k, topinsert, height);
```



Insert in a Skip List

```
insert(n, k, topinsert, level):  
    if(n == null) return;  
    node p = n;  
    n = n.next;  
    while (n != null && n.key < k) {  
        p = n; n = n.next;  
    } // Handle duplicate if necessary  
    if(level < h)  
        insertafternode(p, k); What about the below ref?  
    insert(p.below, k, topinsert, level-1);  
    add below ref
```

```
topinsert = toss_coins_until_tails(maxtosses);  
insert(sentinel, k, topinsert, height);
```

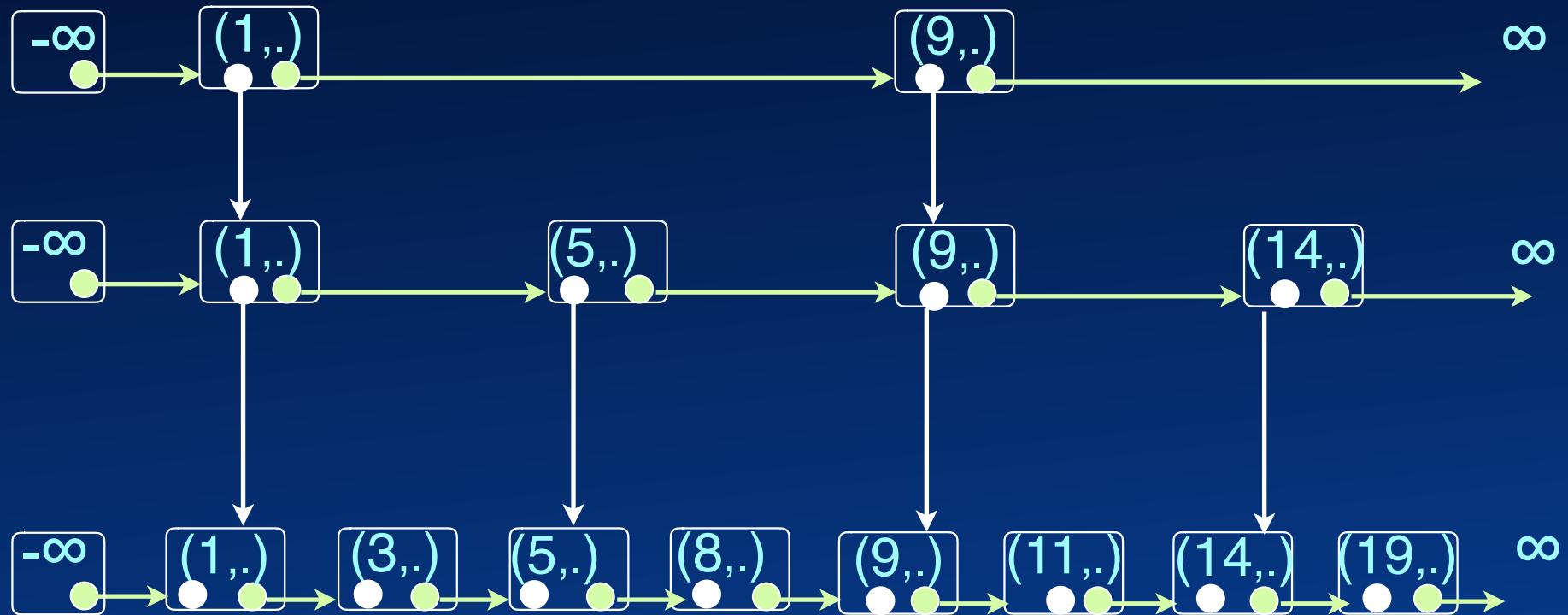
23

What if topinsert > height?



Skip-list: Increase height

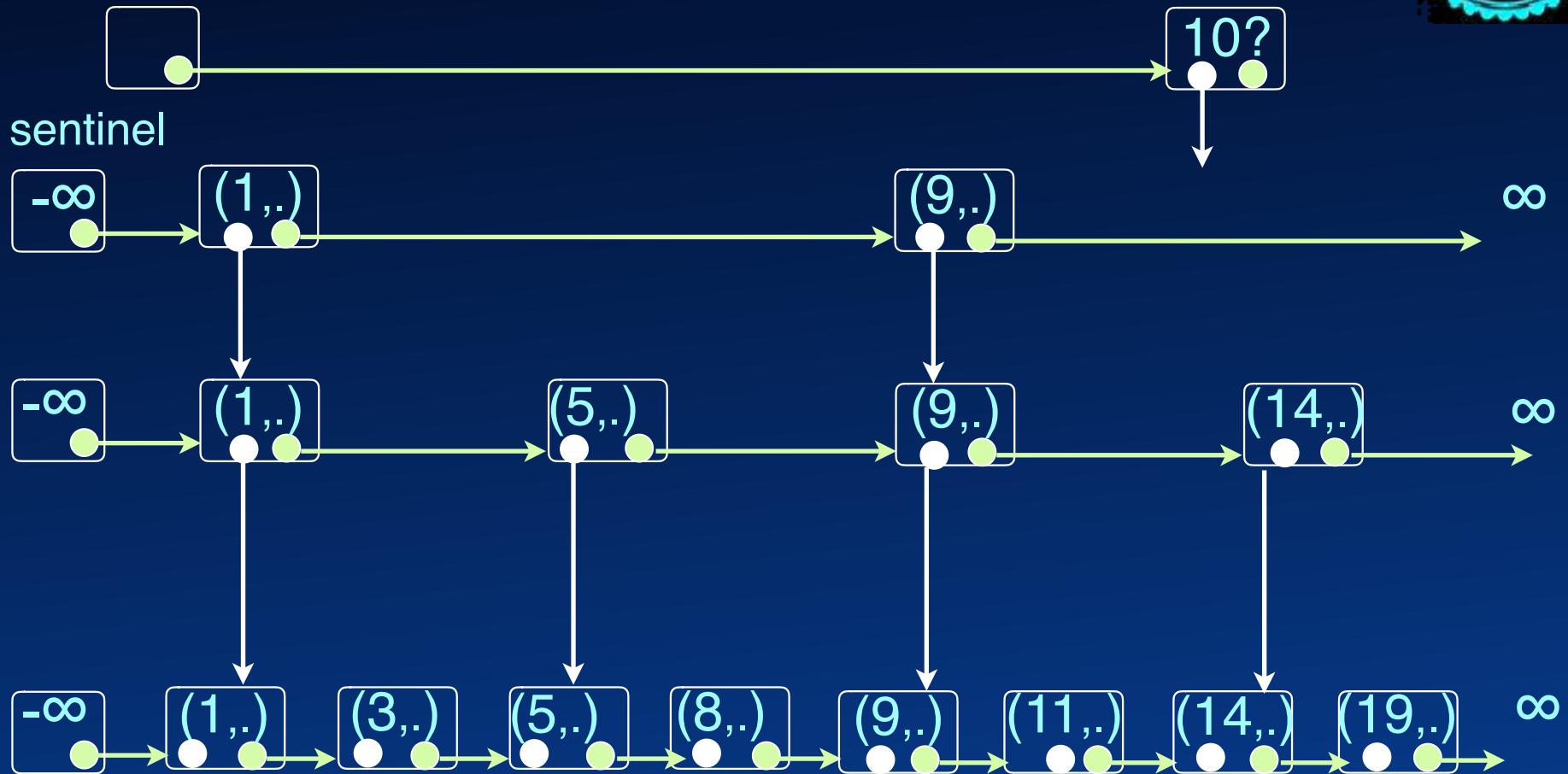
sentinel



Insert 10



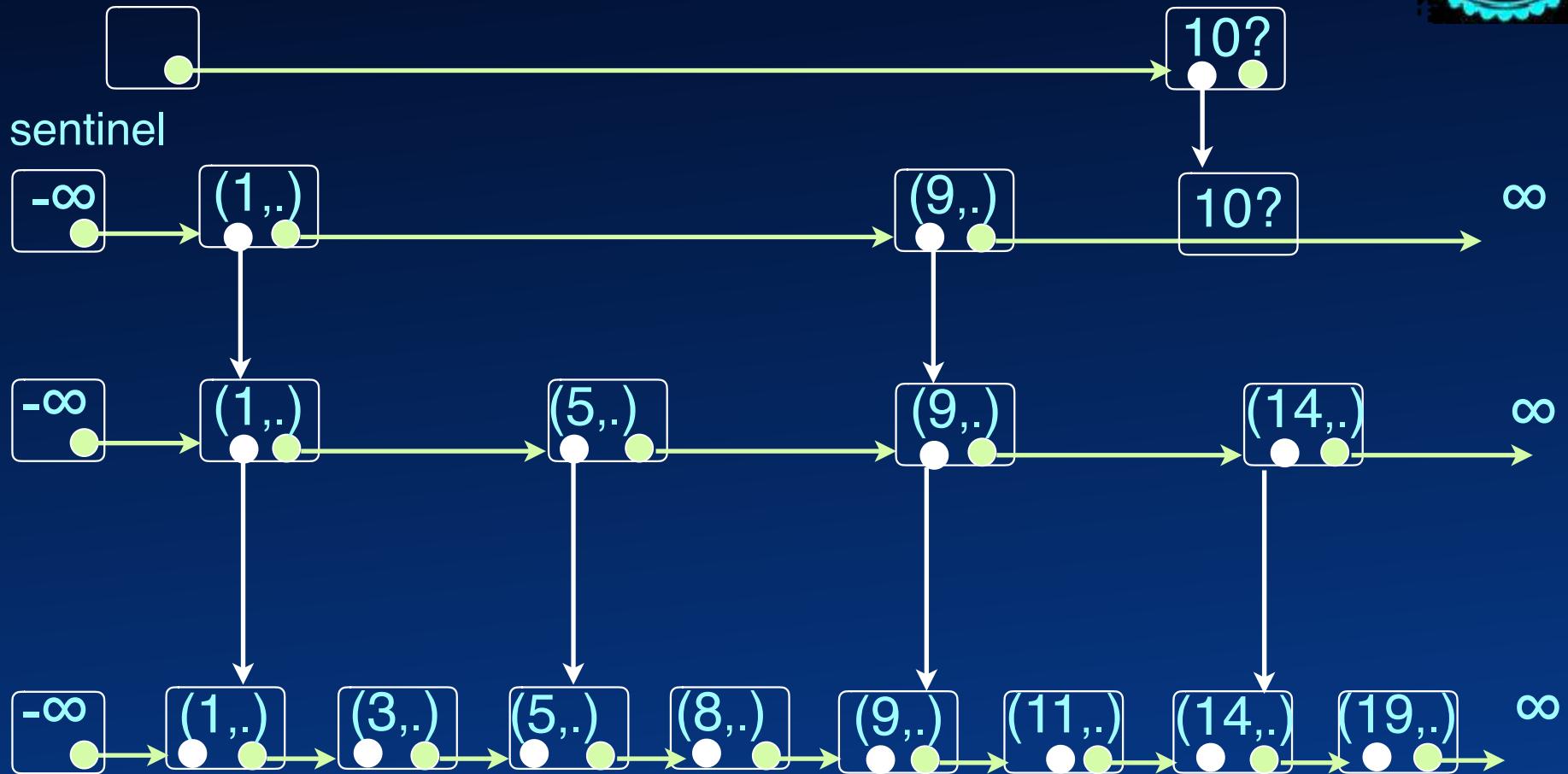
Skip-list: Increase height



Insert 10



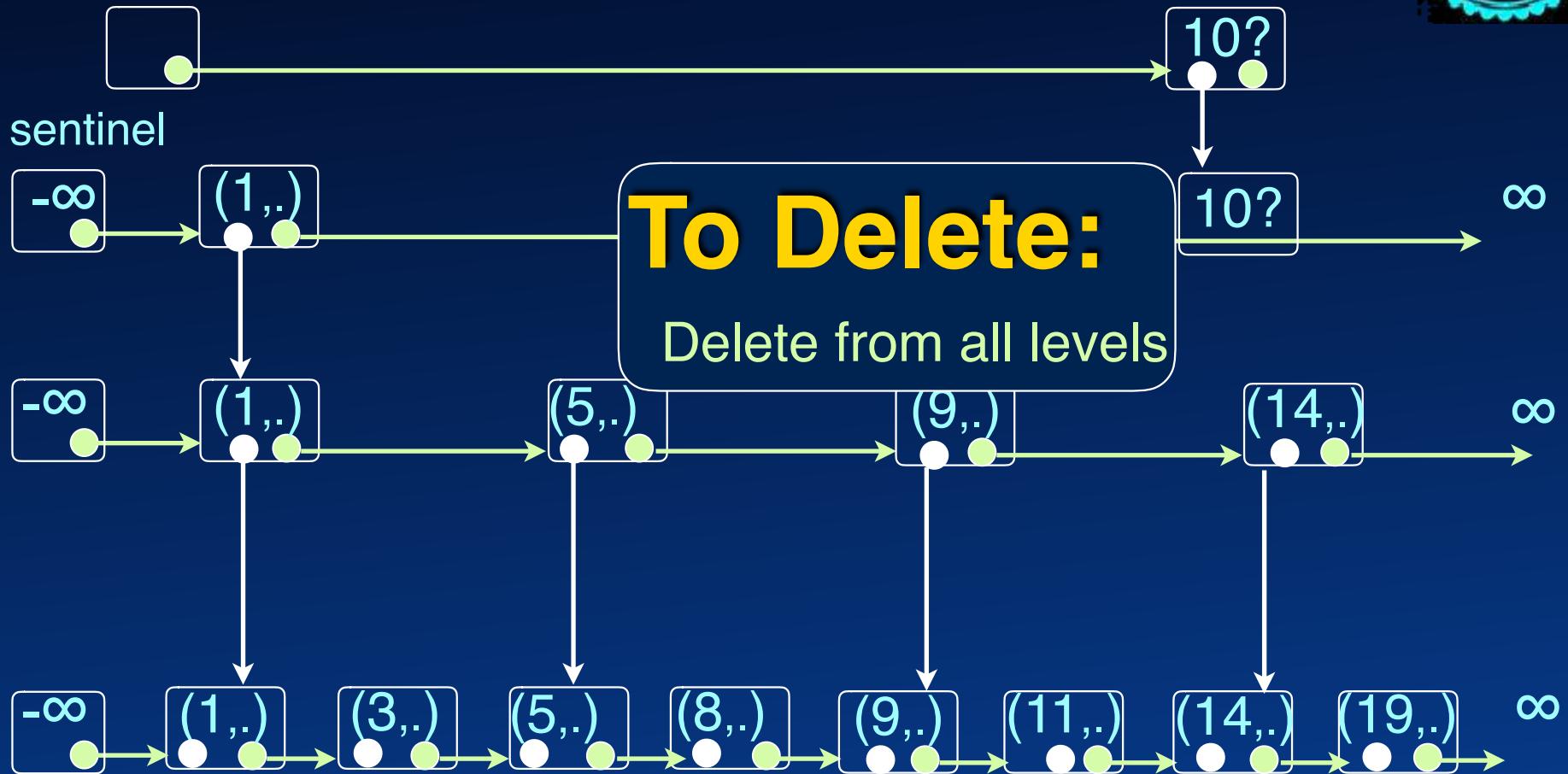
Skip-list: Increase height



Insert 10



Skip-list: Increase height



Insert 10



Skip-list: Analysis

$\text{Prob}(\text{level } i \text{ exists}) \leq n/2^i$



Skip-list: Analysis

$\text{Prob}(\text{level } i \text{ exists}) \leq n/2^i$

i successive heads in any of n experiments



Skip-list: Analysis

$\text{Prob}(\text{level } i \text{ exists}) \leq n/2^i$

i successive heads in any of n experiments

$\text{Prob}(\text{level } \log n \text{ exists}) \leq n/2^{\log n} = n/n$



Skip-list: Analysis

$\text{Prob}(\text{level } i \text{ exists}) \leq n/2^i$

i successive heads in any of n experiments

$\text{Prob}(\text{level } \log n \text{ exists}) \leq n/2^{\log n} = n/n$

$\text{Prob}(\text{level } k \log n \text{ exists}) \leq n/2^{k \log n} = n/n^k = 1/n^{(k-1)}$



Skip-list: Analysis

$\text{Prob}(\text{level } i \text{ exists}) \leq n/2^i$

i successive heads in any of n experiments

$\text{Prob}(\text{level } \log n \text{ exists}) \leq n/2^{\log n} = n/n$

$\text{Prob}(\text{level } k \log n \text{ exists}) \leq n/2^{k \log n} = n/n^k = 1/n^{(k-1)}$

\Downarrow
height = $k \log n$



Skip-list: Analysis

$\text{Prob}(\text{level } i \text{ exists}) \leq n/2^i$

i successive heads in any of n experiments

$\text{Prob}(\text{level } \log n \text{ exists}) \leq n/2^{\log n} = n/n$

$\text{Prob}(\text{level } k \log n \text{ exists}) \leq n/2^{k \log n} = n/n^k = 1/n^{(k-1)}$

\Downarrow
height = $k \log n$

Expected no. of nodes visited at any level = 2



Skip-list: Analysis

$\text{Prob}(\text{level } i \text{ exists}) \leq n/2^i$

i successive heads in any of n experiments

$\text{Prob}(\text{level } \log n \text{ exists}) \leq n/2^{\log n} = n/n$

$\text{Prob}(\text{level } k \log n \text{ exists}) \leq n/2^{k \log n} = n/n^k = 1/n^{(k-1)}$

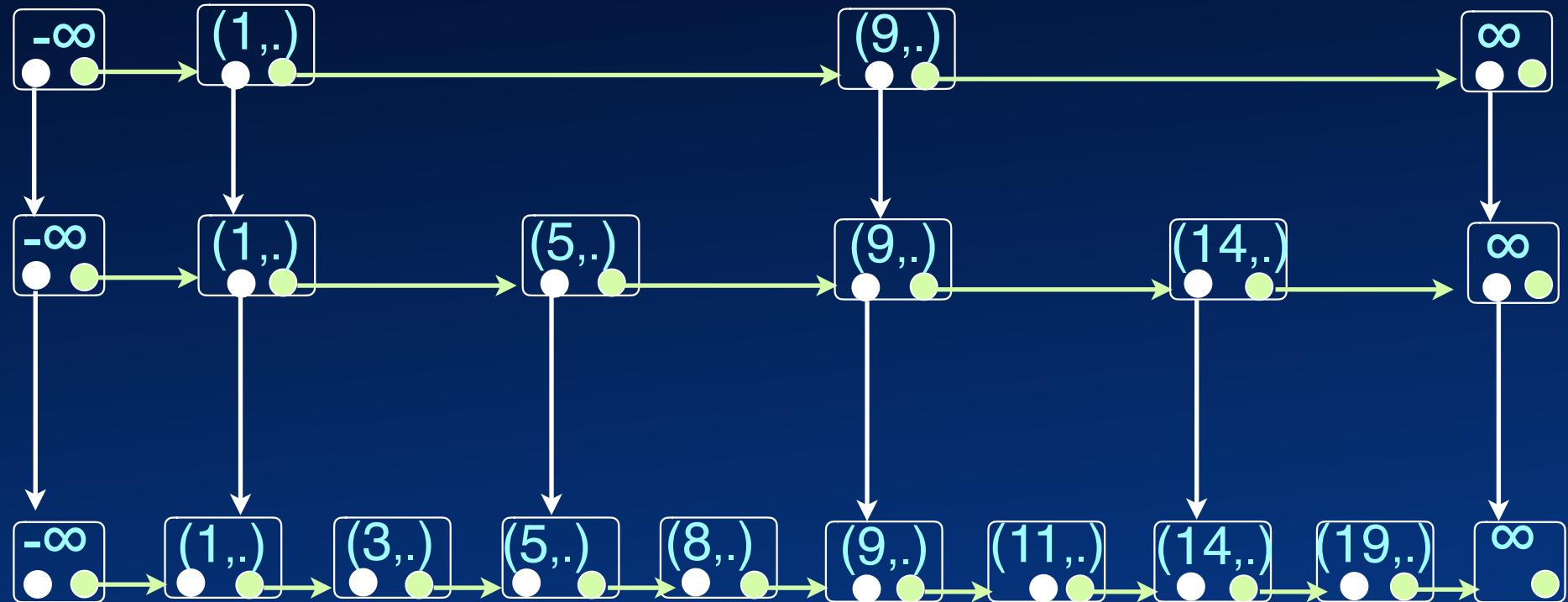
\Downarrow
height = $k \log n$

Expected no. of nodes visited at any level = 2
= Expected number of tosses before *heads*



Skip-list

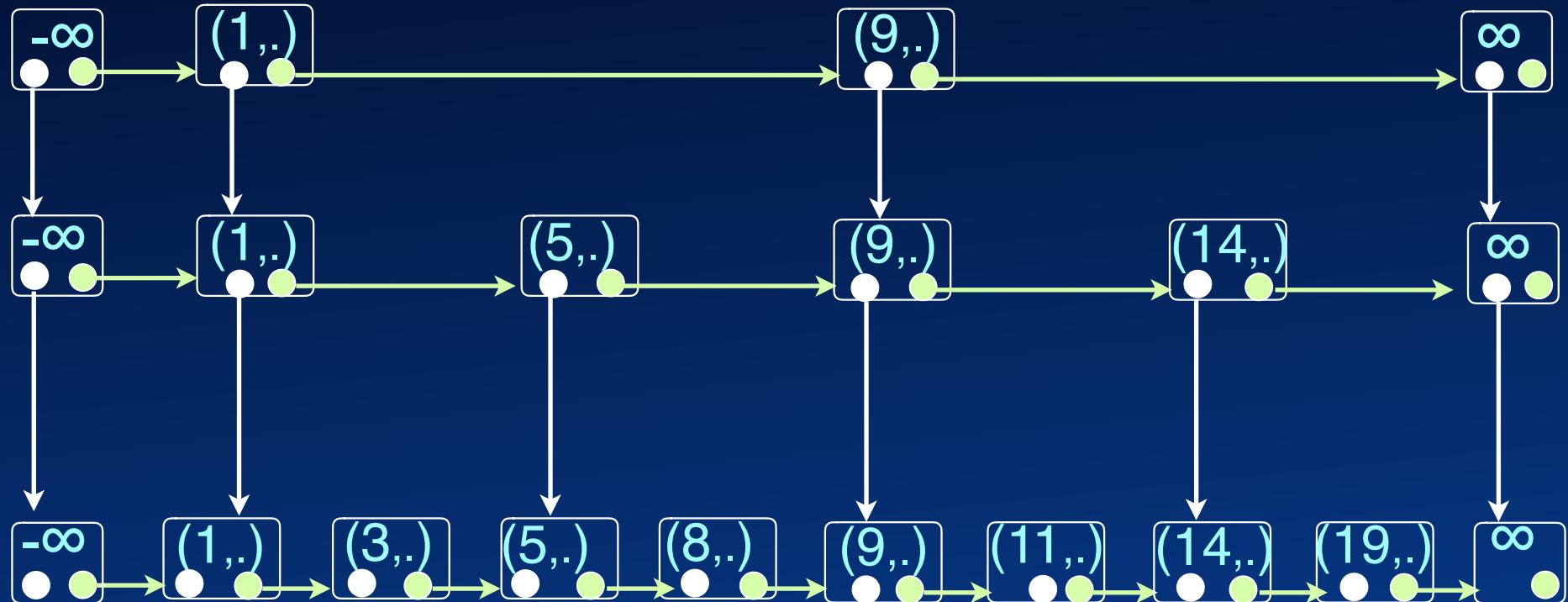
sentinel





Skip-list

sentinel

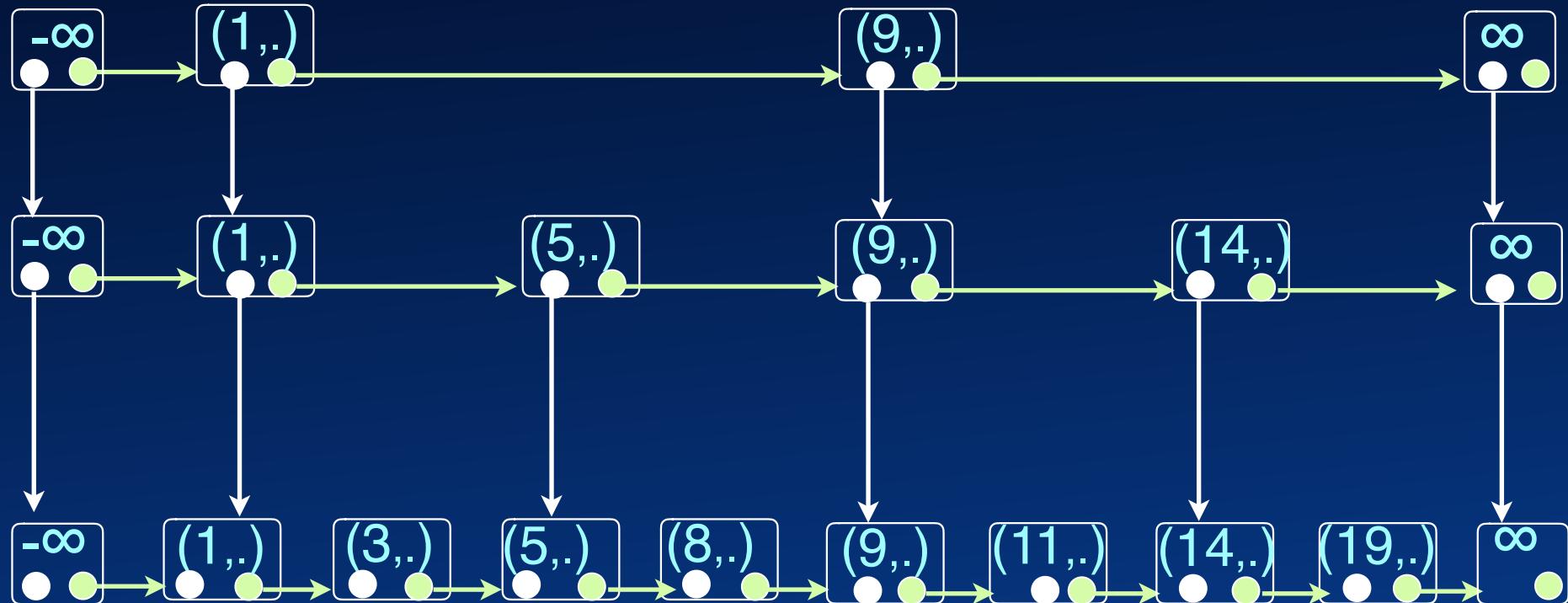


Next to 10



Skip-list

sentinel



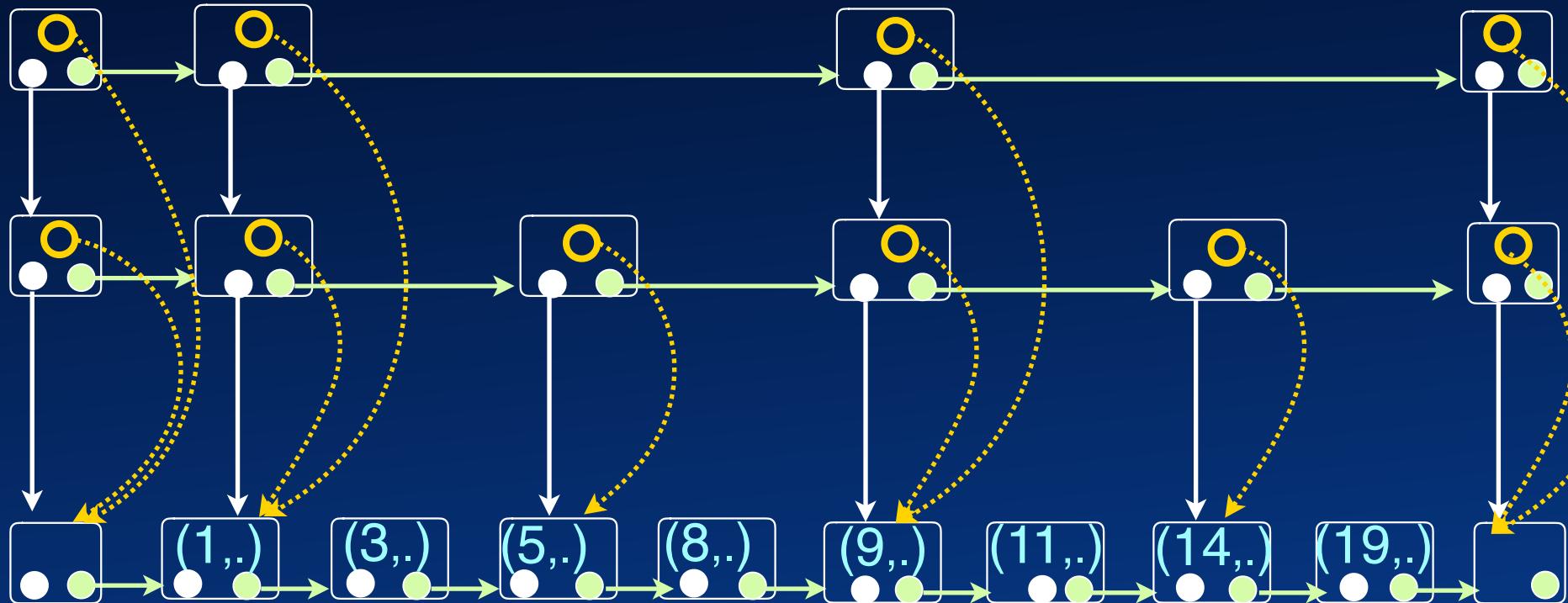
Next to 10

Just before 10?



Skip-list: Save Space

sentinel



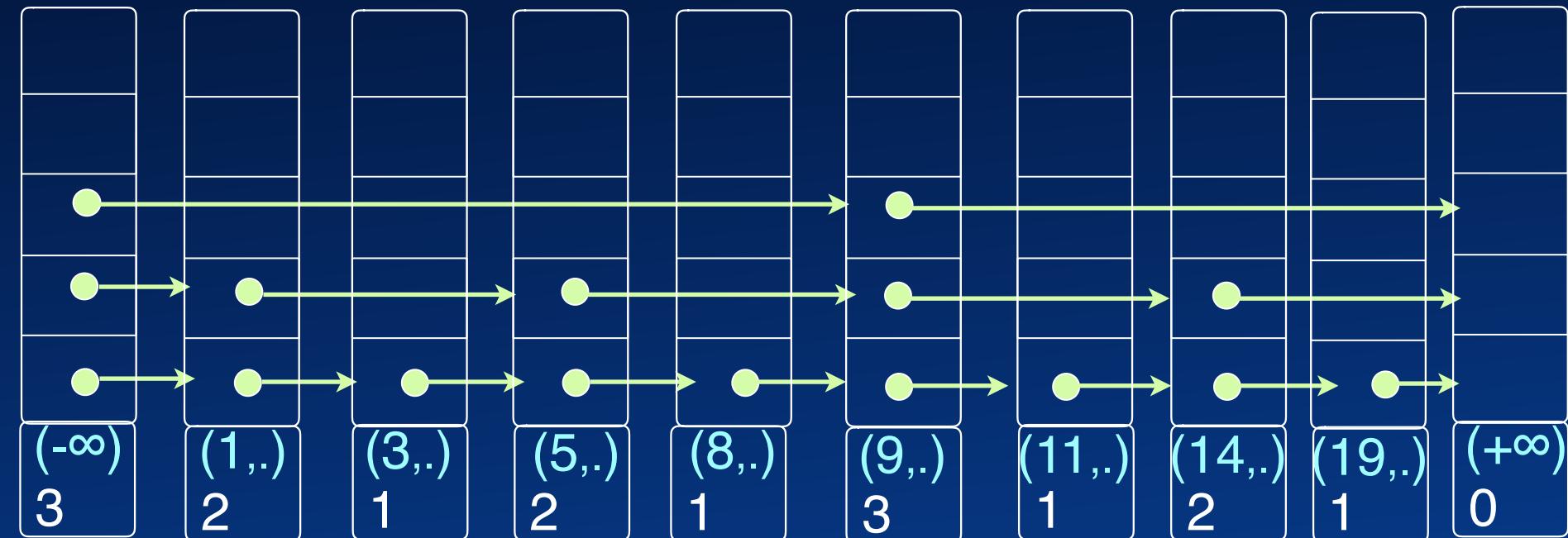
May store only references at the upper levels



Skip-list: Array

sentinel

esent



Could even store in arrays

“Linked” Data structures





“Linked” Data structures

```
class A {  
    B b;  
    C c;  
    int i;  
}
```



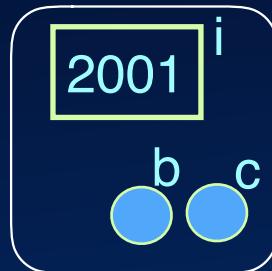
“Linked” Data structures

```
class A {  
    B b;      A a = new A(..)  
    C c;  
    int i;  
}
```



“Linked” Data structures

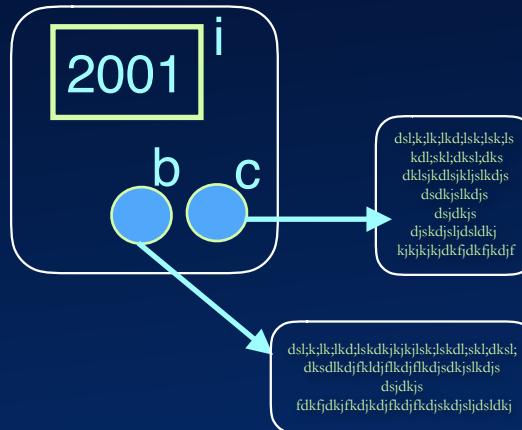
```
class A {  
    B b;      A a = new A(..)  
    C c;  
    int i;  
}
```





“Linked” Data structures

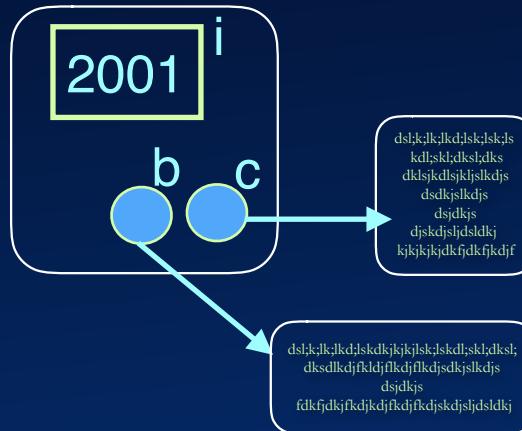
```
class A {  
    B b;      A a = new A(..)  
    C c;  
    int i;  
}
```





“Linked” Data structures

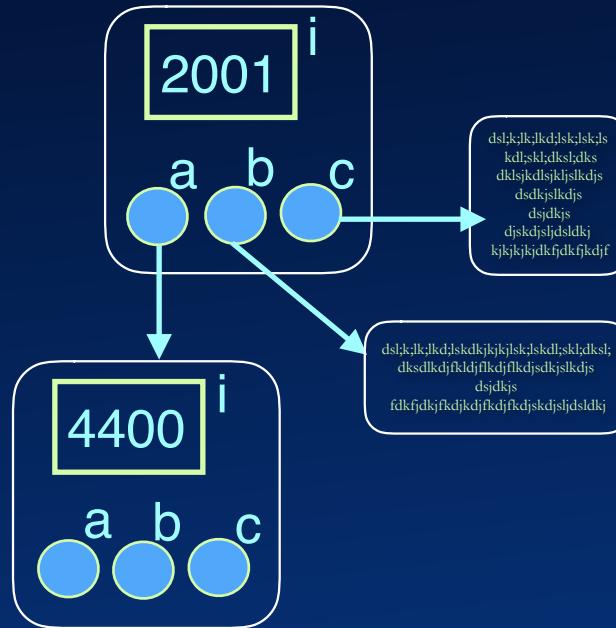
```
class A {  
    B b;        A a = new A(..)  
    C c;  
    int i;  
} A a;  
}
```





“Linked” Data structures

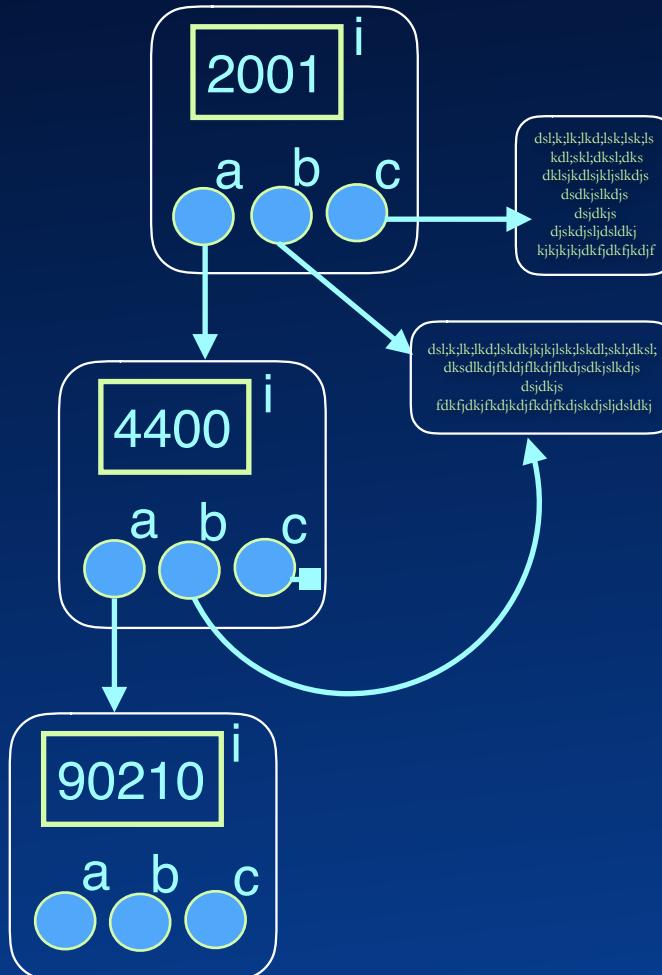
```
class A {  
    B b;      A a = new A(..)  
    C c;  
    int i;  
}  
A a;  
}
```





“Linked” Data structures

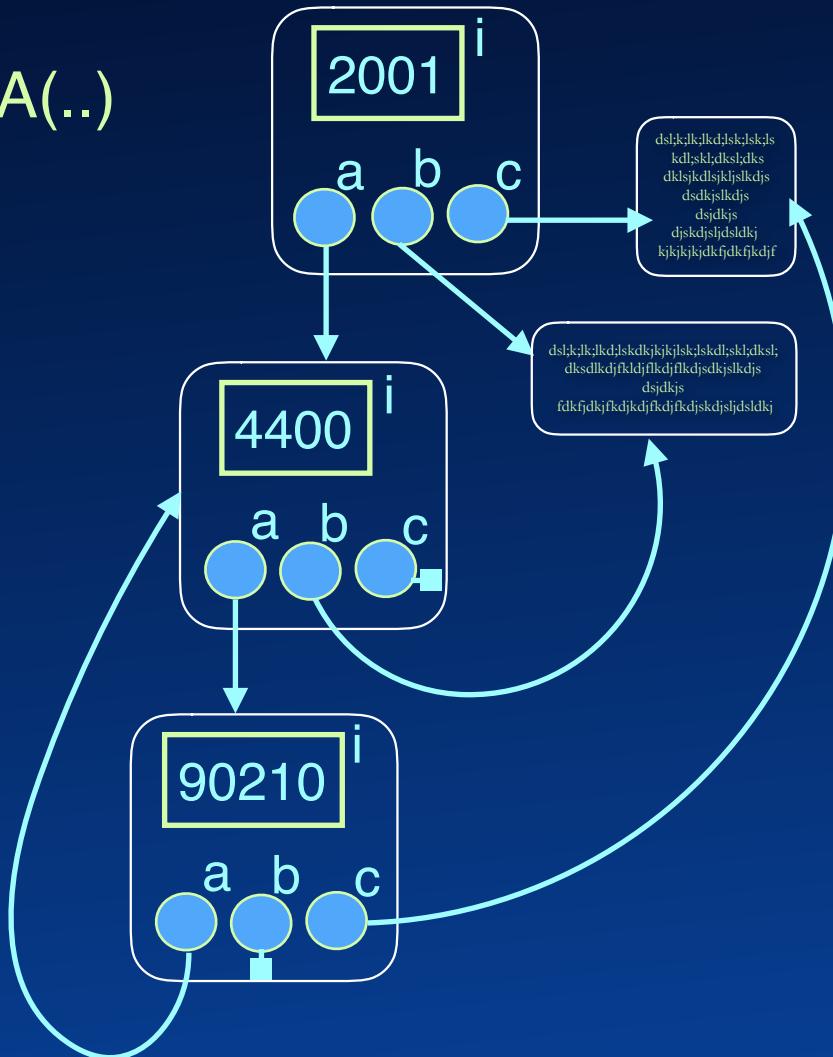
```
class A {  
    B b;      A a = new A(..)  
    C c;  
    int i;  
}  
A a;  
}
```



“Linked” Data structures



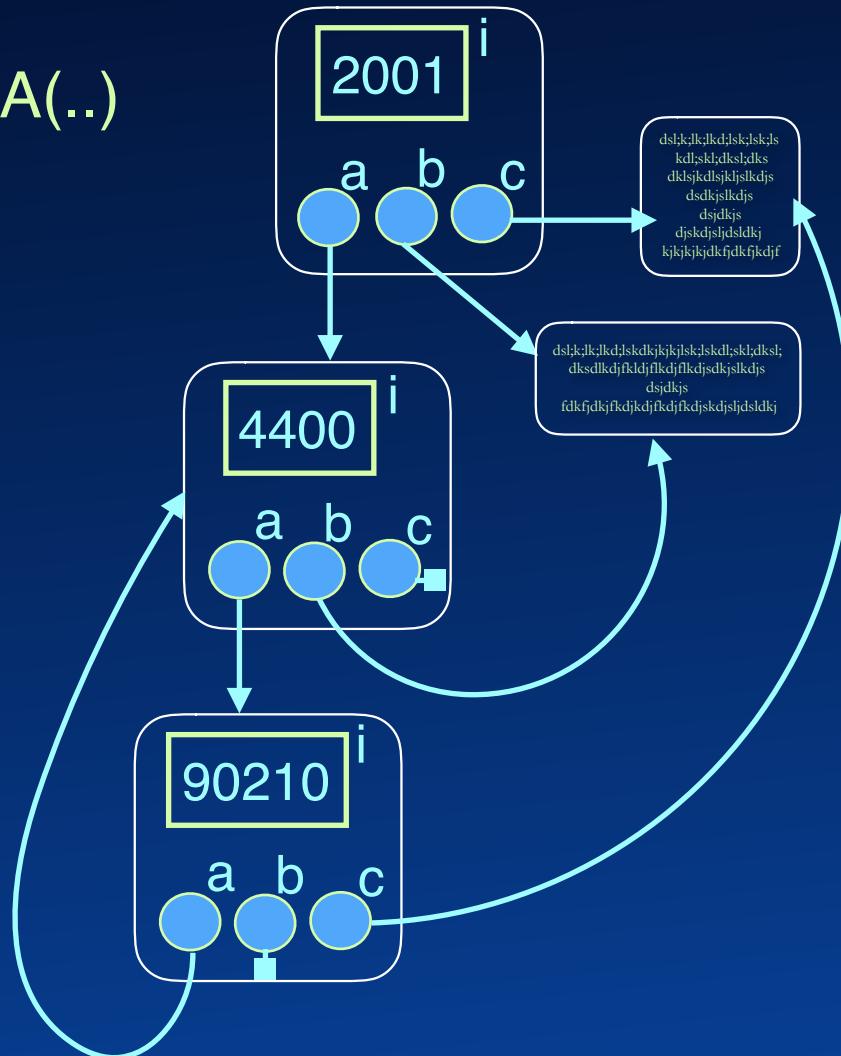
```
class A {  
    B b;        A a = new A(..)  
    C c;  
    int i;  
} A a;  
}
```





“Linked” Data structures

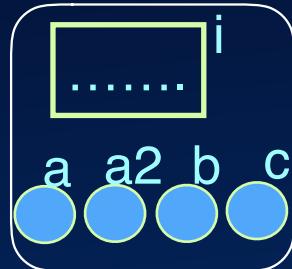
```
class A {  
    B b;      A a = new A(..)  
    C c;  
    int i;  
}  
A a;  
A a2;  
}
```





“Linked” Data structures

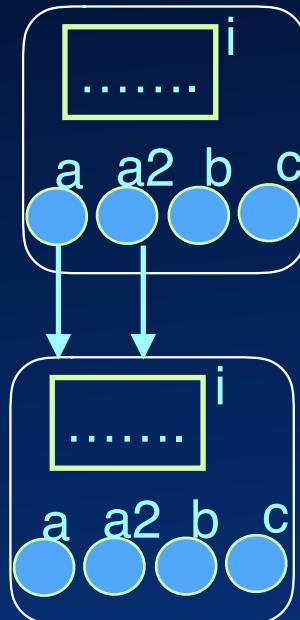
```
class A {  
    B b;      A a = new A(..)  
    C c;  
    int i;  
    A a;  
    A a2;  
}
```





“Linked” Data structures

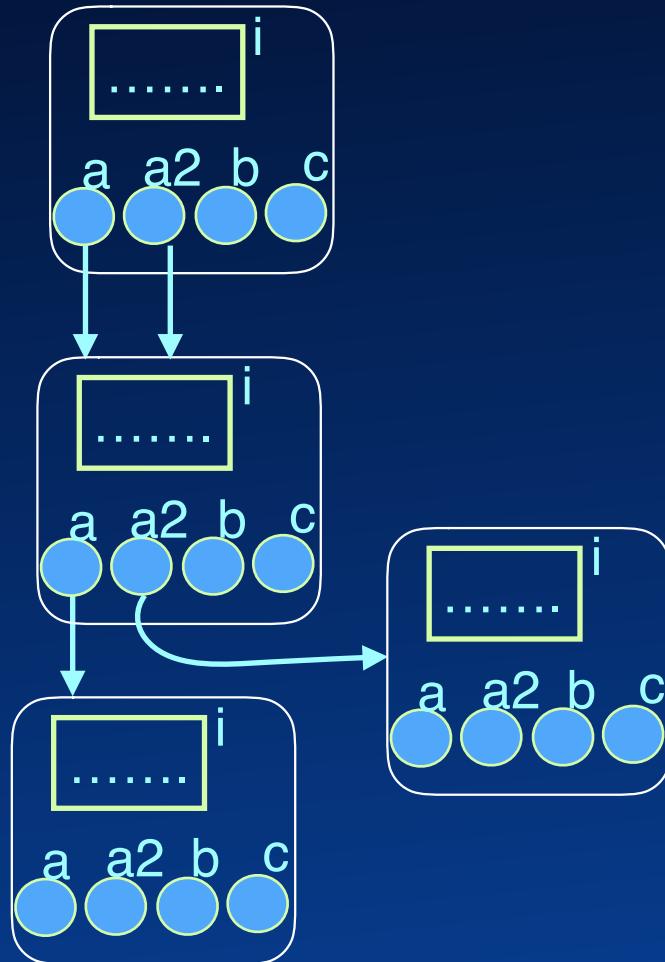
```
class A {  
    B b;      A a = new A(..)  
    C c;  
    int i;  
    A a;  
    A a2;  
}
```





“Linked” Data structures

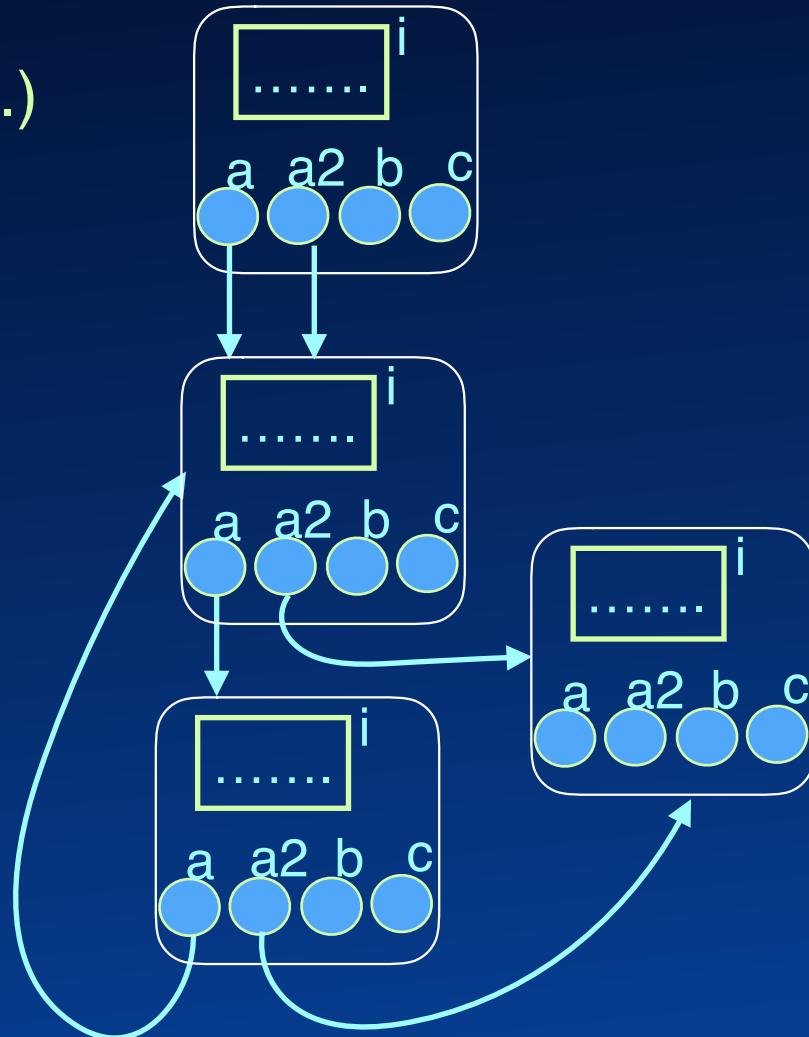
```
class A {  
    B b;      A a = new A(..)  
    C c;  
    int i;  
    A a;  
    A a2;  
}
```





“Linked” Data structures

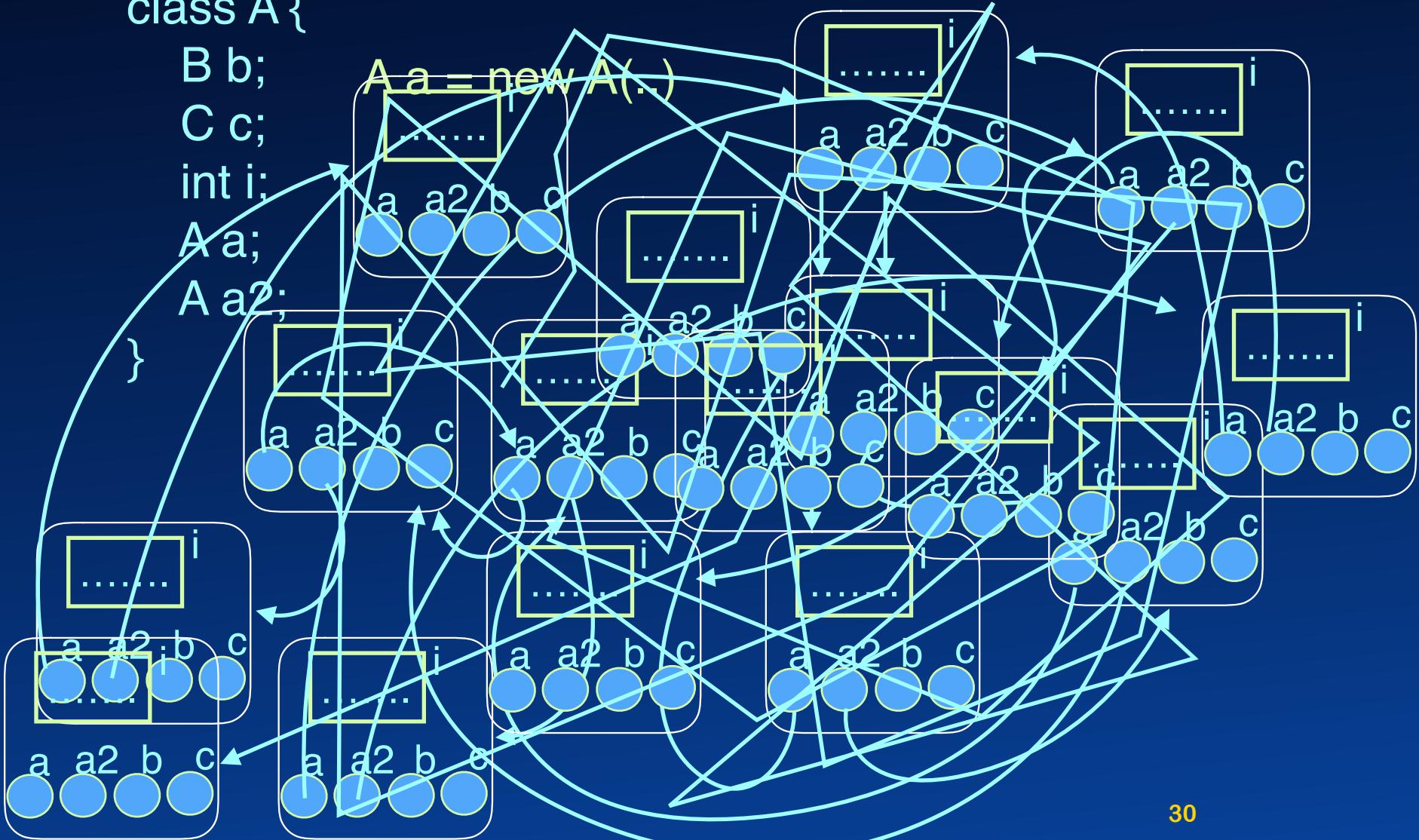
```
class A {  
    B b;      A a = new A(..)  
    C c;  
    int i;  
    A a;  
    A a2;  
}
```





“Linked” Data structures

```
class A {  
    B b;  
    C c;  
    int i;  
    A a;  
    A a2;  
}
```



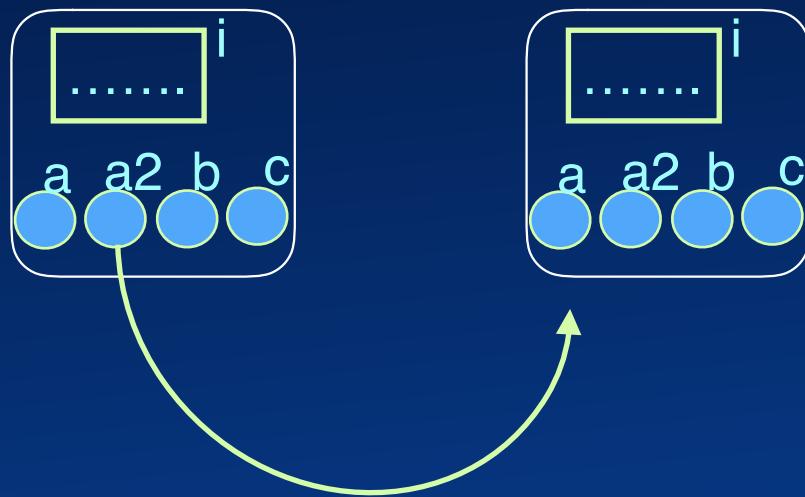


Doubly Linked List



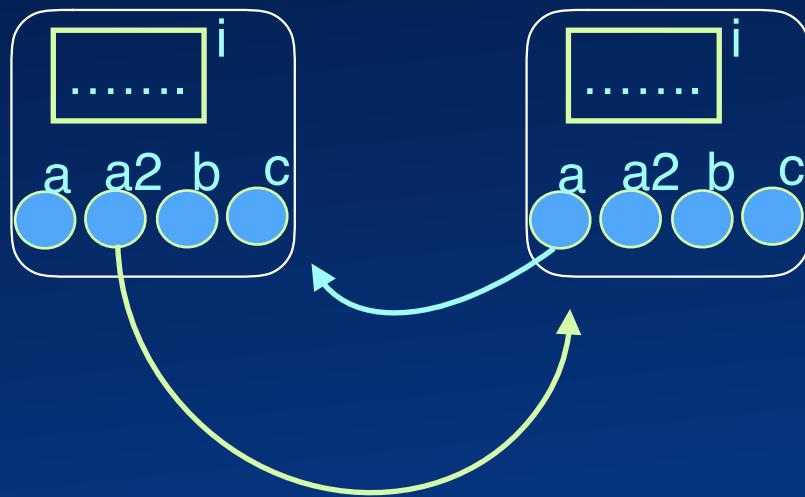


Doubly Linked List



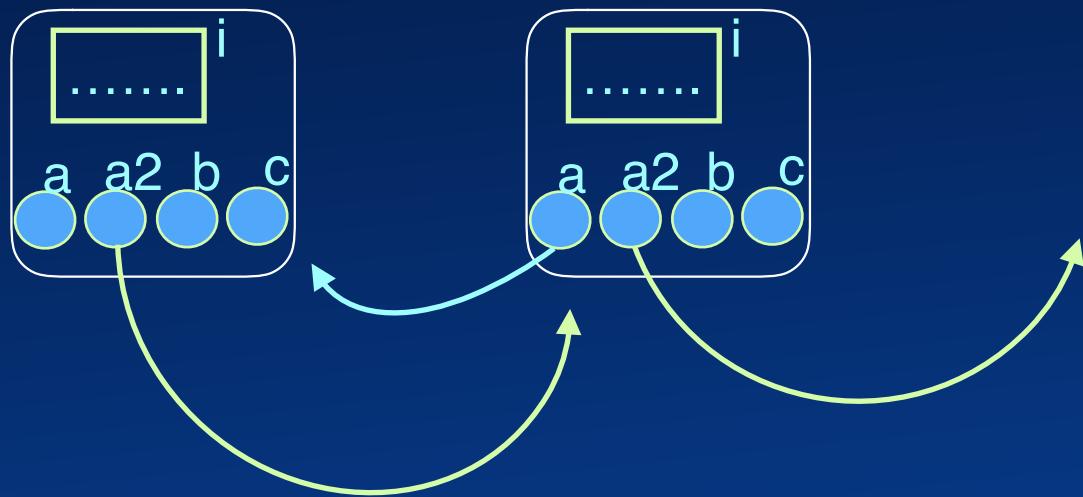


Doubly Linked List



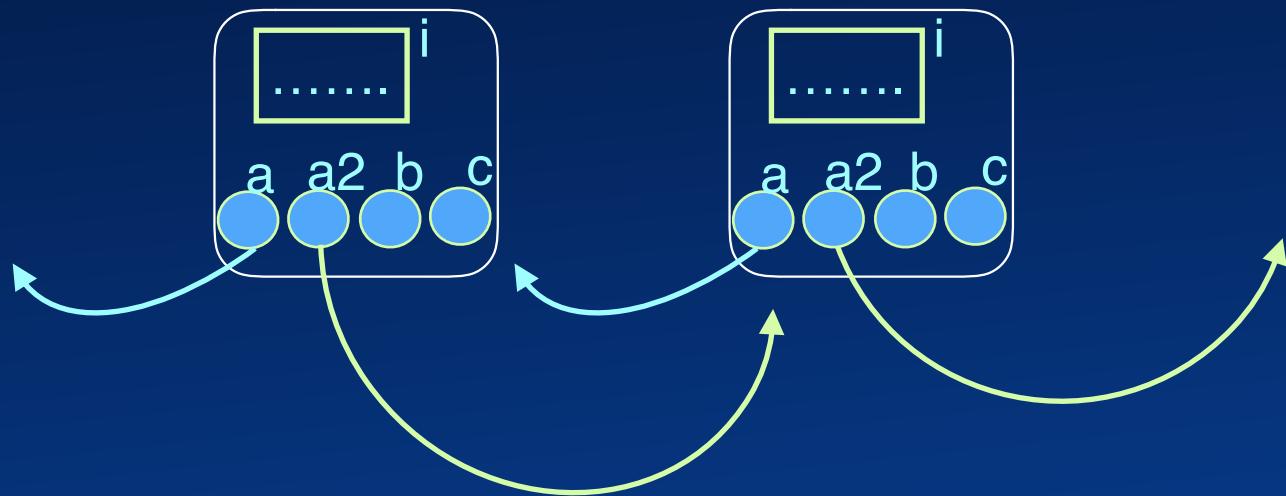


Doubly Linked List



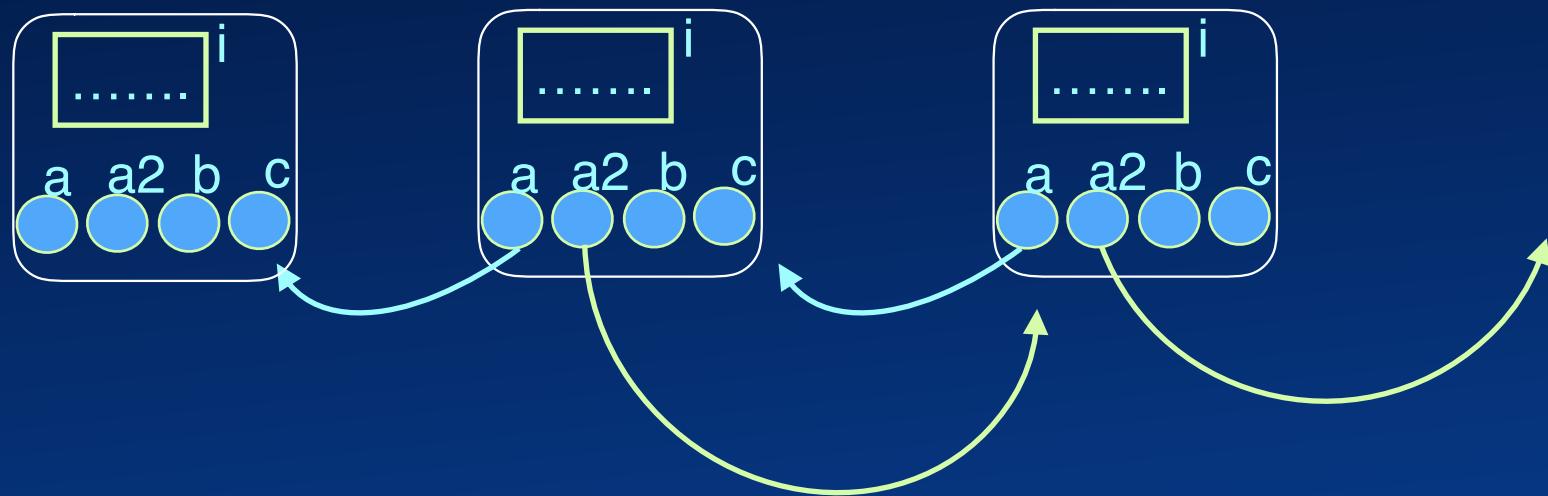


Doubly Linked List



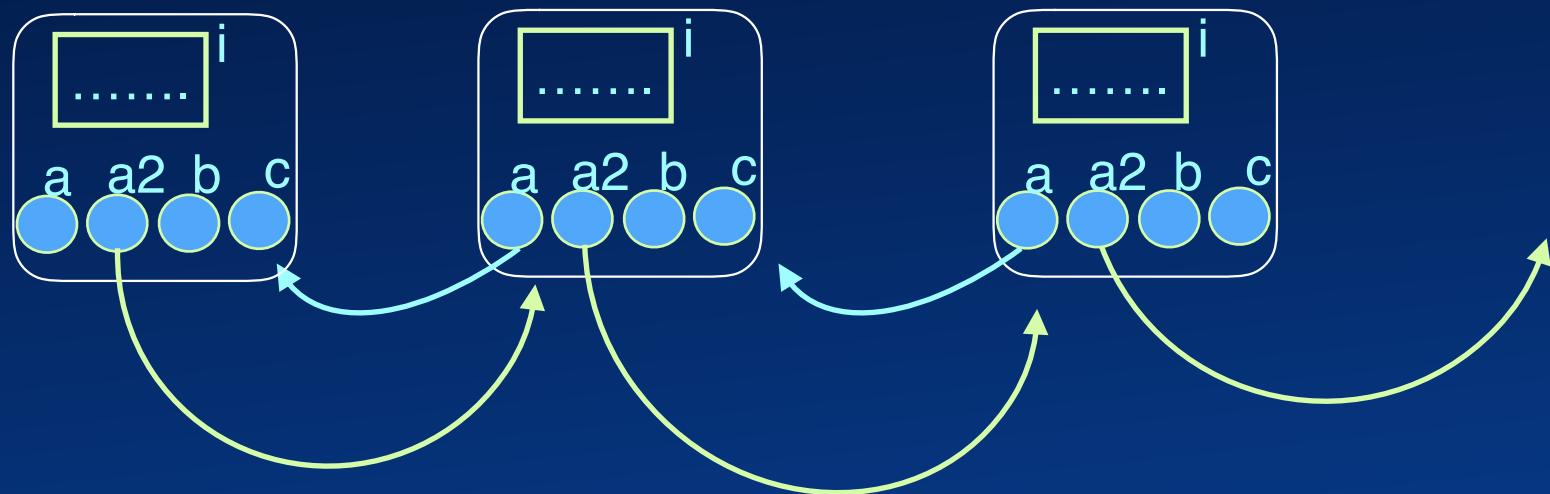


Doubly Linked List



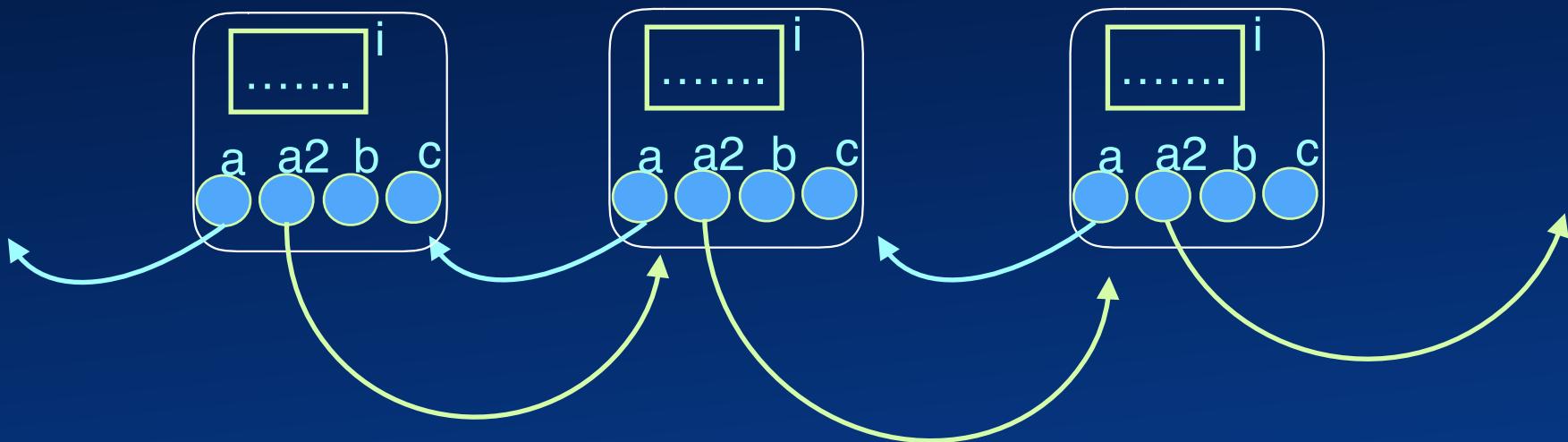


Doubly Linked List



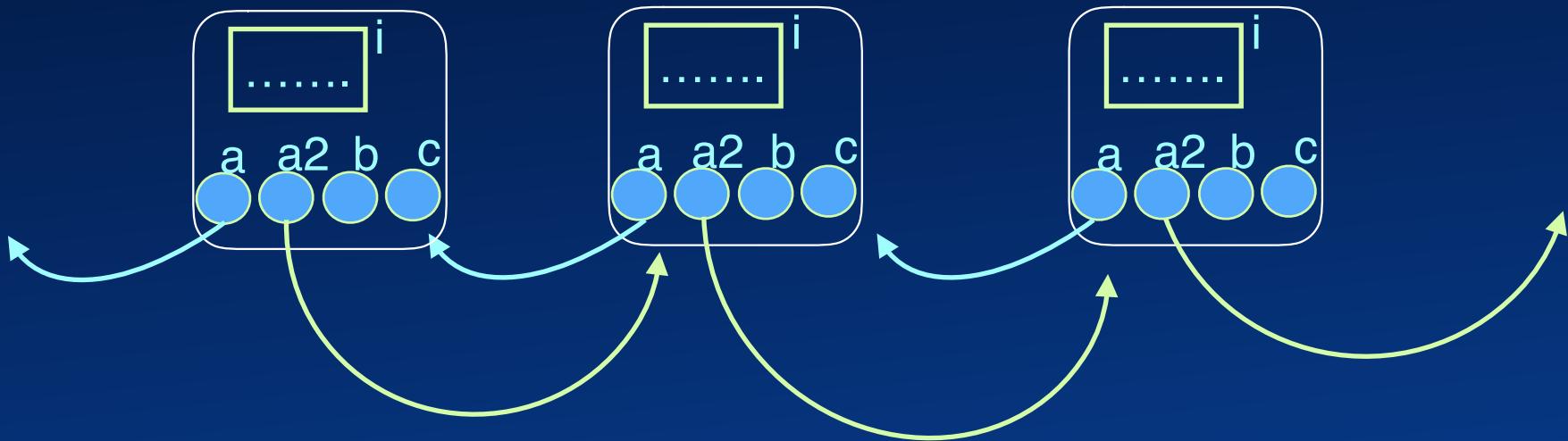


Doubly Linked List





Doubly Linked List



At most two nodes refer to any given node, and are reciprocated.



Binary Tree

```
class BinaryNode<T> {  
    BinaryNode left;  
    BinaryNode right;  
    T value;  
    methods etc.  
}
```



Binary Tree

```
class BinaryNode<T> {  
    BinaryNode left;  
    BinaryNode right;  
    T value;  
    methods etc.  
}
```

```
class BinaryNode<T> {  
    BinaryNode left;  
    BinaryNode right;  
    T value;  
    methods etc.  
}
```

```
class BinaryNode<T> {  
    BinaryNode left;  
    BinaryNode right;  
    T value;  
    methods etc.  
}
```



Binary Tree

```
class BinaryNode<T> {  
    BinaryNode left;  
    BinaryNode right;  
    T value;  
    methods etc.  
}
```

```
class BinaryNode<T> {  
    BinaryNode left;  
    BinaryNode right;  
    T value;  
    methods etc.  
}
```

```
class BinaryNode<T> {  
    BinaryNode left;  
    BinaryNode right;  
    T value;  
    methods etc.  
}
```

```
class BinaryNode<T> {  
    BinaryNode left;  
    BinaryNode right;  
    T value;  
    methods etc.  
}
```

```
class BinaryNode<T> {  
    BinaryNode left;  
    BinaryNode right;  
    T value;  
    methods etc.  
}
```

```
class BinaryNode<T> {  
    BinaryNode left;  
    BinaryNode right;  
    T value;  
    methods etc.  
}
```

```
class BinaryNode<T> {  
    BinaryNode left;  
    BinaryNode right;  
    T value;  
    methods etc.  
}
```



Binary Tree

```
class BinaryNode<T> {  
    BinaryNode left;  
    BinaryNode right;  
    T value;  
    methods etc.  
}
```

```
class BinaryNode<T> {  
    BinaryNode left;  
    BinaryNode right;  
    T value;  
    methods etc.  
}
```

```
class BinaryNode<T> {  
    BinaryNode left;  
    BinaryNode right;  
    T value;  
    methods etc.  
}
```

```
class BinaryNode<T> {  
    BinaryNode left;  
    BinaryNode right;  
    T value;  
    methods etc.  
}
```

```
class BinaryNode<T> {  
    BinaryNode left;  
    BinaryNode right;  
    T value;  
    methods etc.  
}
```

```
class BinaryNode<T> {  
    BinaryNode left;  
    BinaryNode right;  
    T value;  
    methods etc.  
}
```

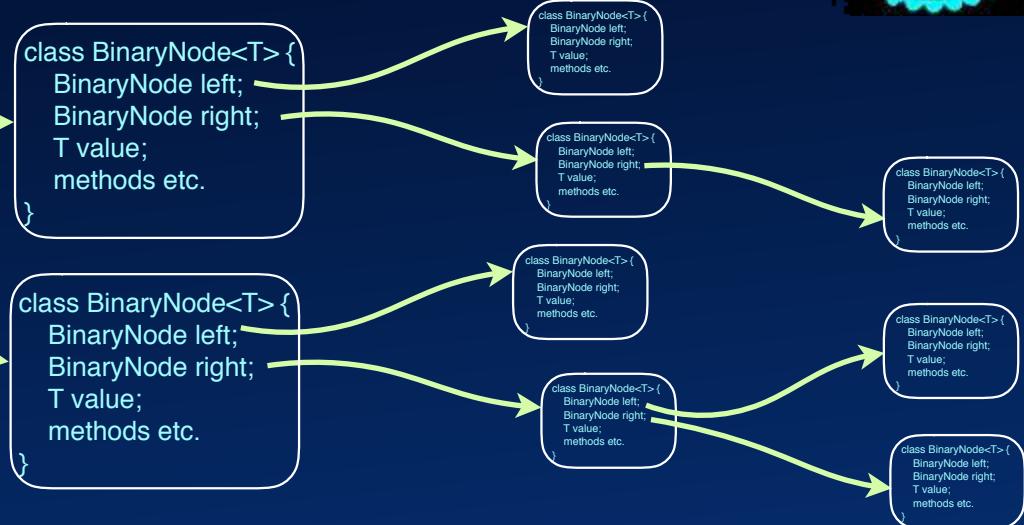
```
class BinaryNode<T> {  
    BinaryNode left;  
    BinaryNode right;  
    T value;  
    methods etc.  
}
```

```
class BinaryNode<T> {  
    BinaryNode left;  
    BinaryNode right;  
    T value;  
    methods etc.  
}
```



Binary Tree

```
class BinaryNode<T> {  
    BinaryNode left;  
    BinaryNode right;  
    T value;  
    methods etc.  
}
```

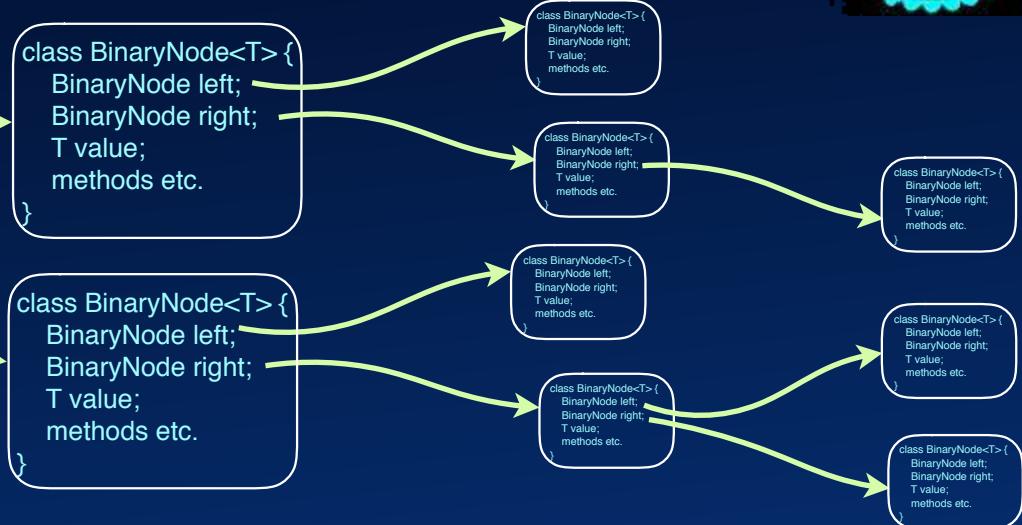


Exactly one other node contains reference to any given node.



Binary Tree

```
class BinaryNode<T> {  
    BinaryNode left;  
    BinaryNode right;  
    T value;  
    methods etc.  
}
```



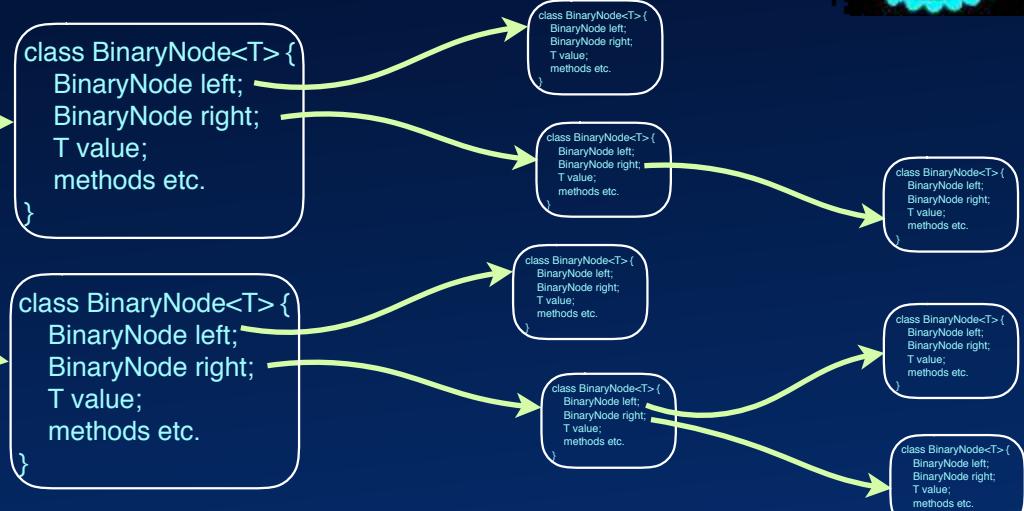
Exactly one other node contains reference to any given node.
One special node has no other node with reference to it.



Binary Tree

```
class BinaryNode<T> {  
    BinaryNode left;  
    BinaryNode right;  
    T value;  
    methods etc.  
}
```

root



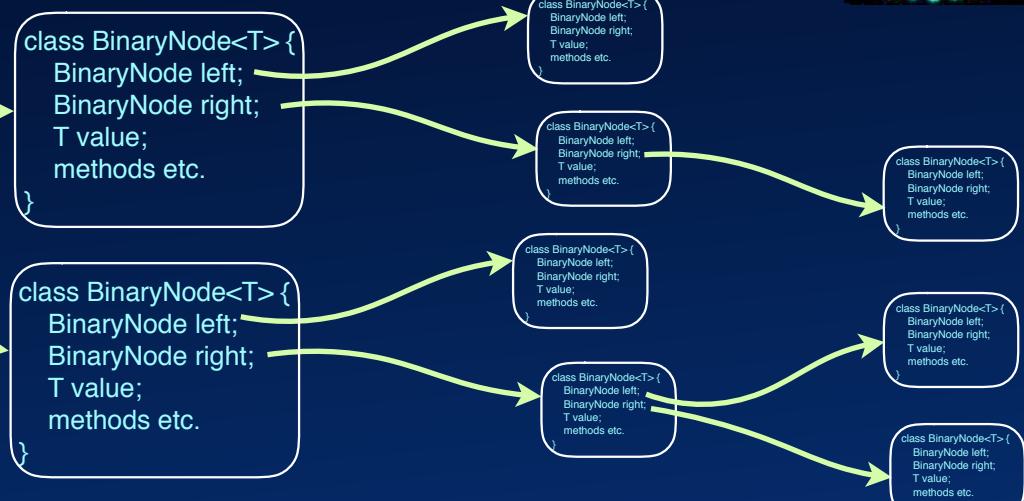
Exactly one other node contains reference to any given node.
One special node has no other node with reference to it.



Binary Tree

```
class BinaryNode<T> {  
    BinaryNode left;  
    BinaryNode right;  
    T value;  
    methods etc.  
}
```

root



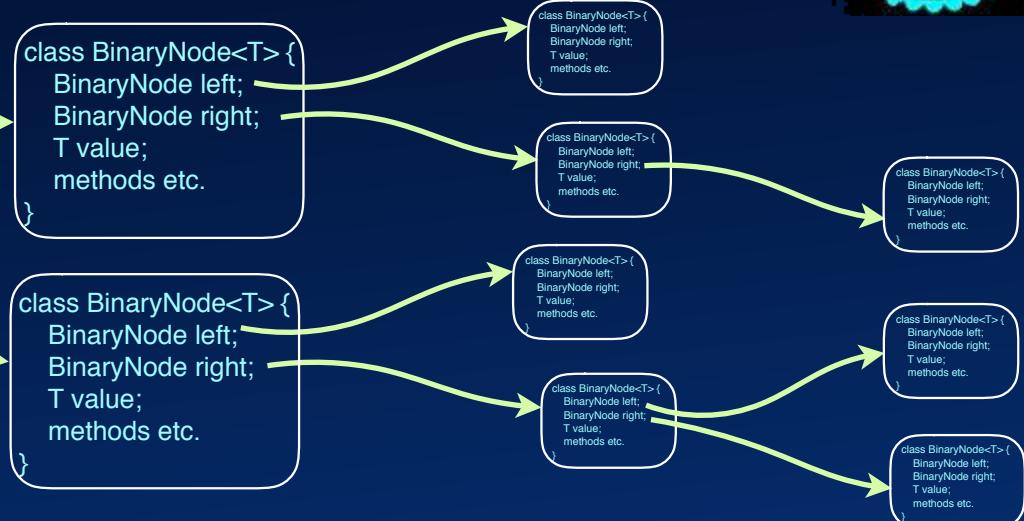
Exactly one other node contains reference to any given node.
One special node has no other node with reference to it.
Binary tree has two references per node



Binary Tree

```
class BinaryNode<T> {  
    BinaryNode left;  
    BinaryNode right;  
    T value;  
    methods etc.  
}
```

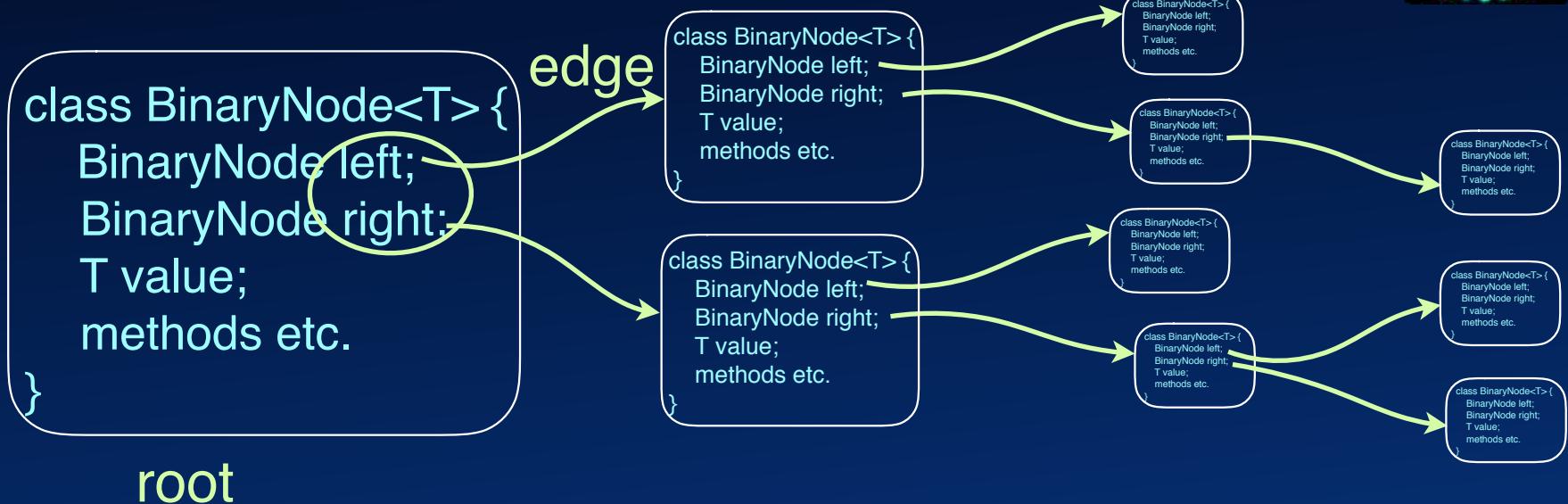
root



Exactly one other node contains reference to any given node.
One special node has no other node with reference to it.
Binary tree has two references per node



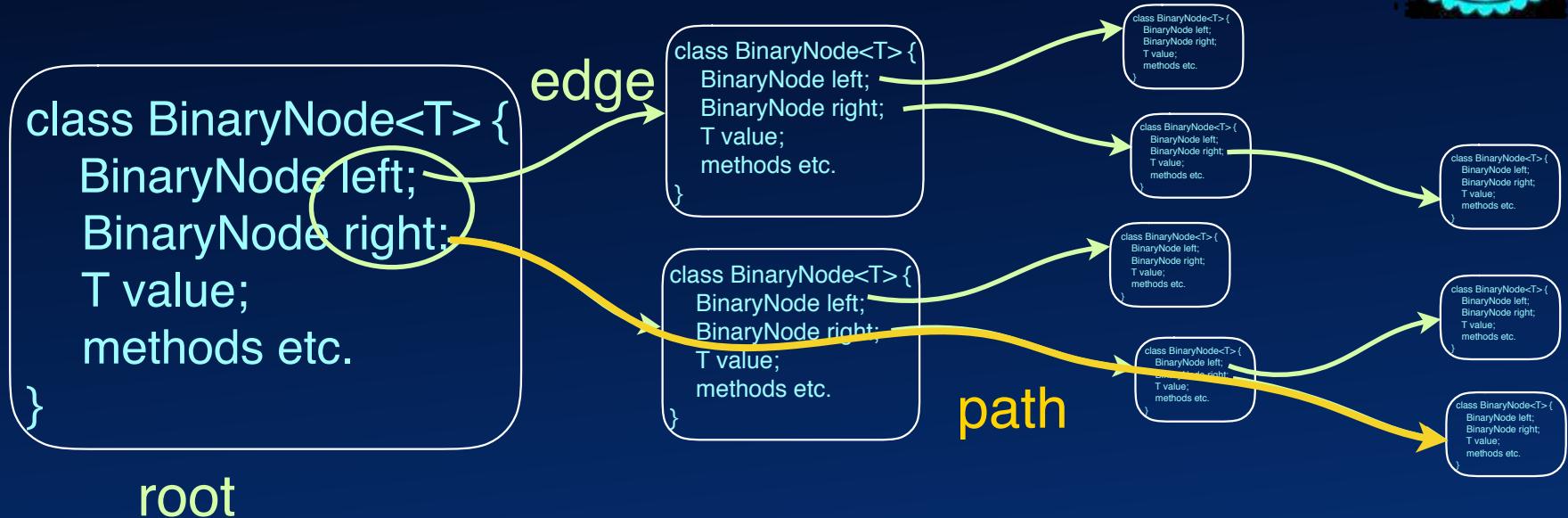
Binary Tree



Exactly one other node contains reference to any given node.
One special node has no other node with reference to it.
Binary tree has two references per node



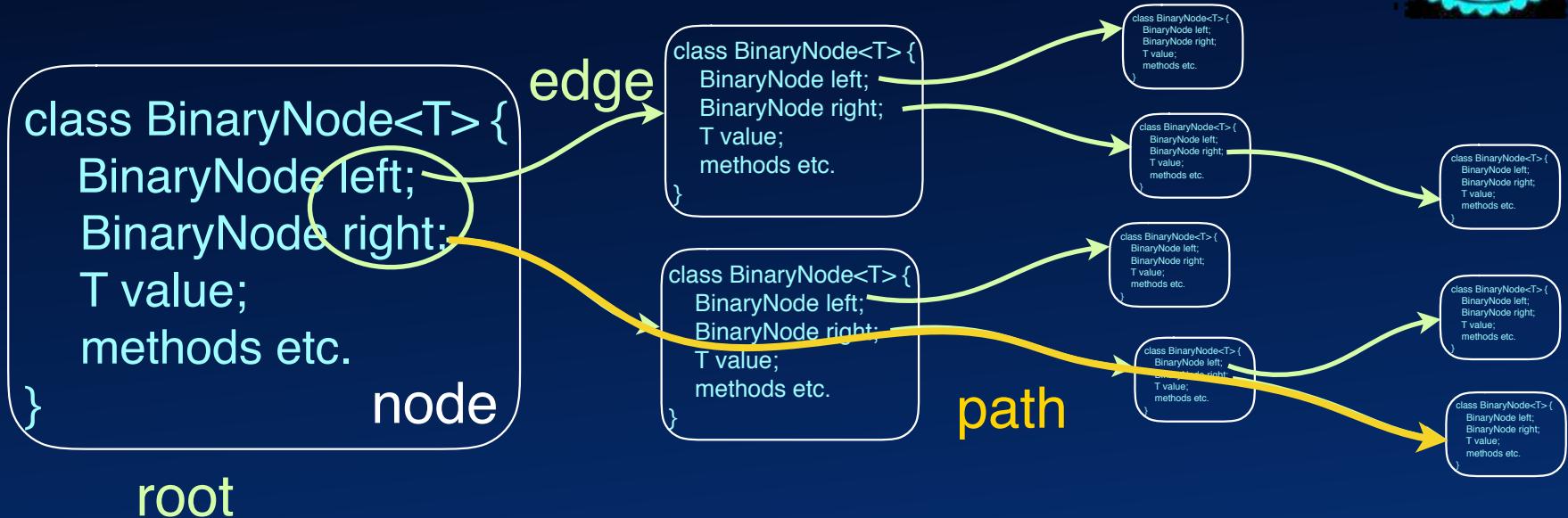
Binary Tree



Exactly one other node contains reference to any given node.
One special node has no other node with reference to it.
Binary tree has two references per node



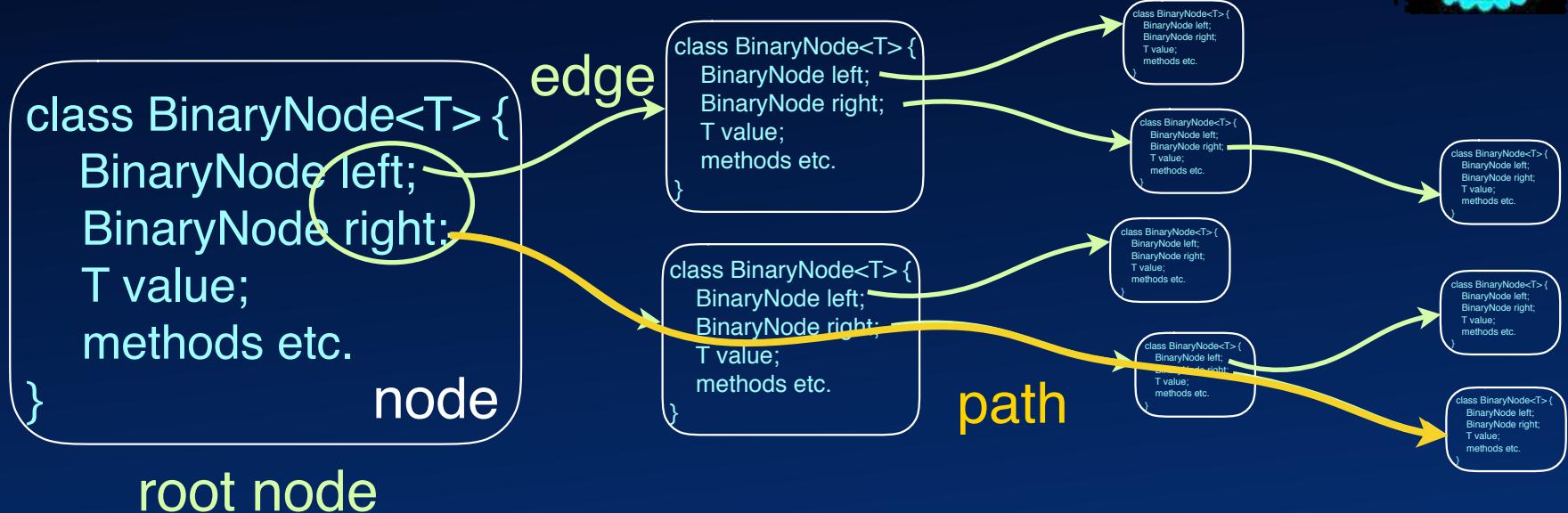
Binary Tree



Exactly one other node contains reference to any given node.
One special node has no other node with reference to it.
Binary tree has two references per node



Binary Tree



Exactly one other node contains reference to any given node.
One special node has no other node with reference to it.
Binary tree has two references per node



Quiz

- Keeping skip-list towers in arrays saves space. What else does it save? [1 word]
- How? [maximum 10 words]
- What deficiency does it suffer from? [Maximum 3 words]

col106quiz@cse.iitd.ac.in

Format: time, No separate fetch of below node, fixed count



Binary Tree

```
class BinaryNode<T> {  
    BinaryNode<T> left;  
    BinaryNode<T> right;  
    T value;  
    methods etc.  
}
```

```
class BinaryNode<T> {  
    BinaryNode<T> left;  
    BinaryNode<T> right;  
    T value;  
    methods etc.  
}
```

```
class BinaryNode<T> {  
    BinaryNode<T> left;  
    BinaryNode<T> right;  
    T value;  
    methods etc.  
}
```

```
class BinaryNode<T> {  
    BinaryNode<T> left;  
    BinaryNode<T> right;  
    T value;  
    methods etc.  
}
```

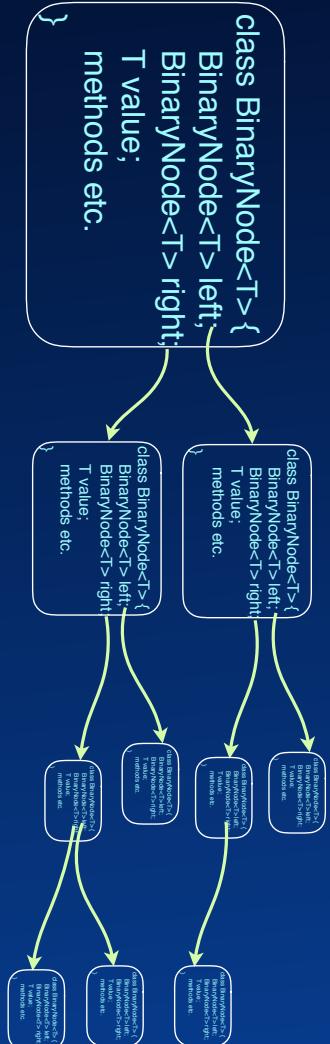
```
class BinaryNode<T> {  
    BinaryNode<T> left;  
    BinaryNode<T> right;  
    T value;  
    methods etc.  
}
```

```
class BinaryNode<T> {  
    BinaryNode<T> left;  
    BinaryNode<T> right;  
    T value;  
    methods etc.  
}
```

```
class BinaryNode<T> {  
    BinaryNode<T> left;  
    BinaryNode<T> right;  
    T value;  
    methods etc.  
}
```

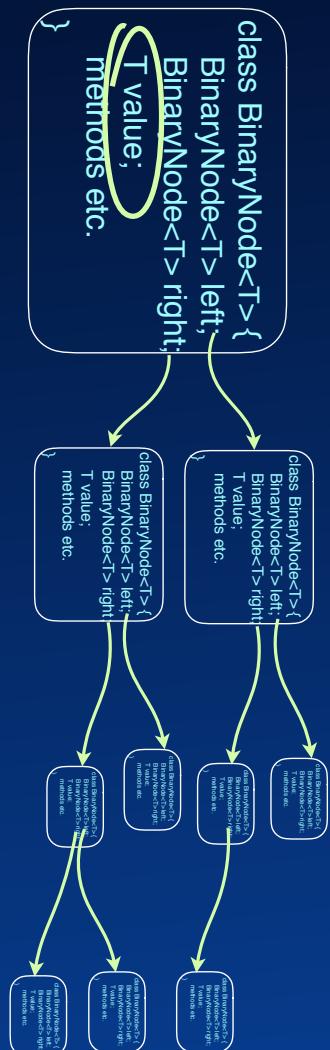


Binary Tree



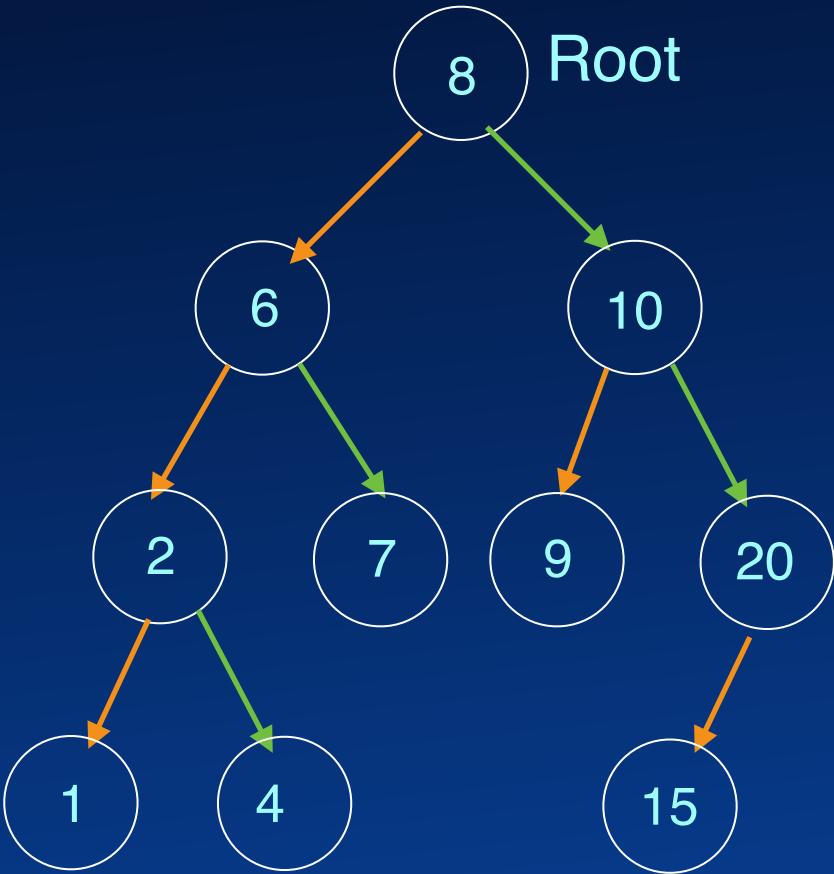
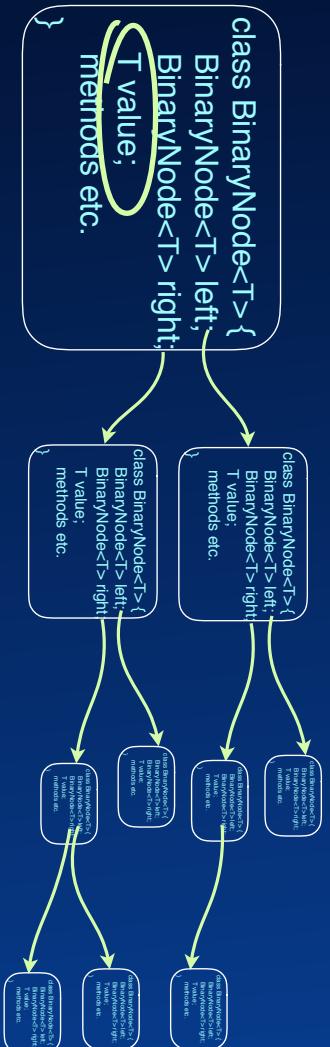


Binary Tree



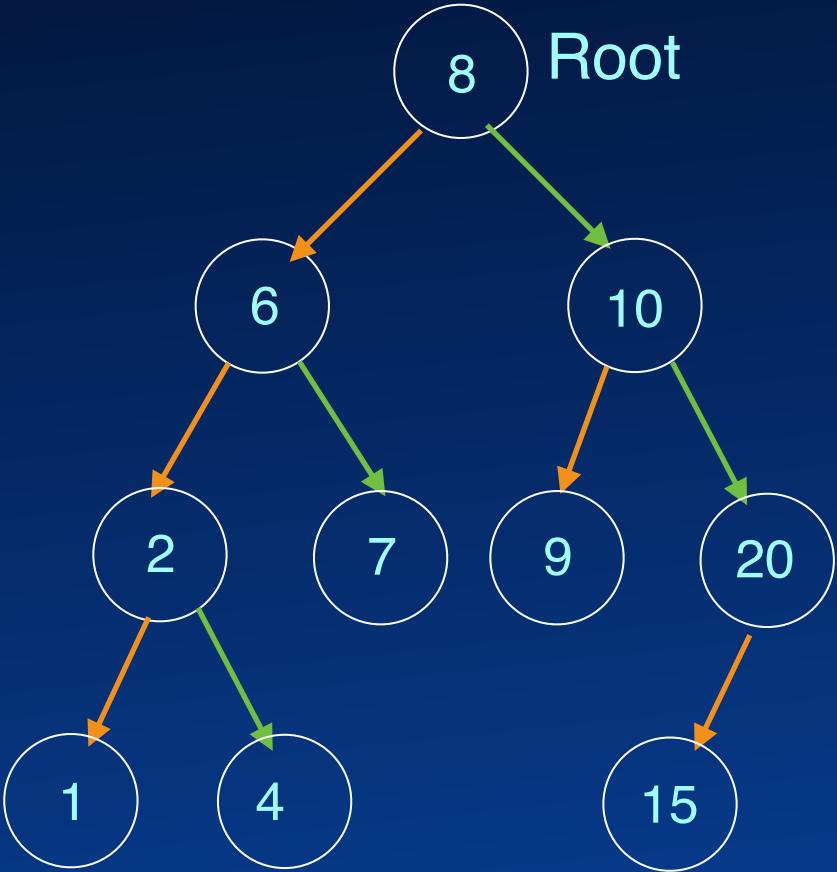
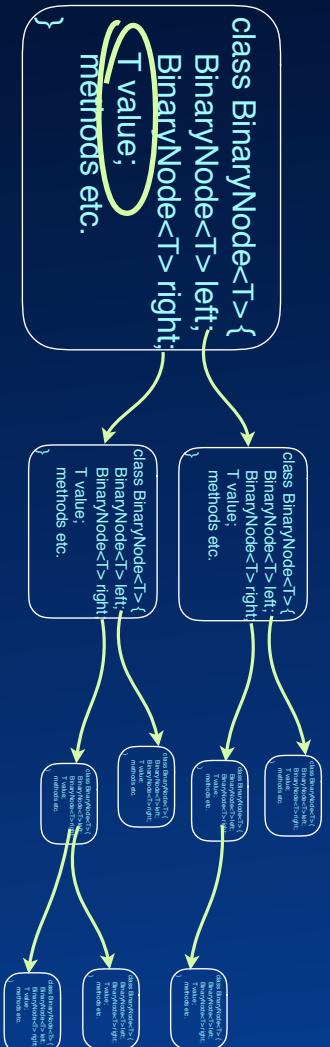


Binary Tree





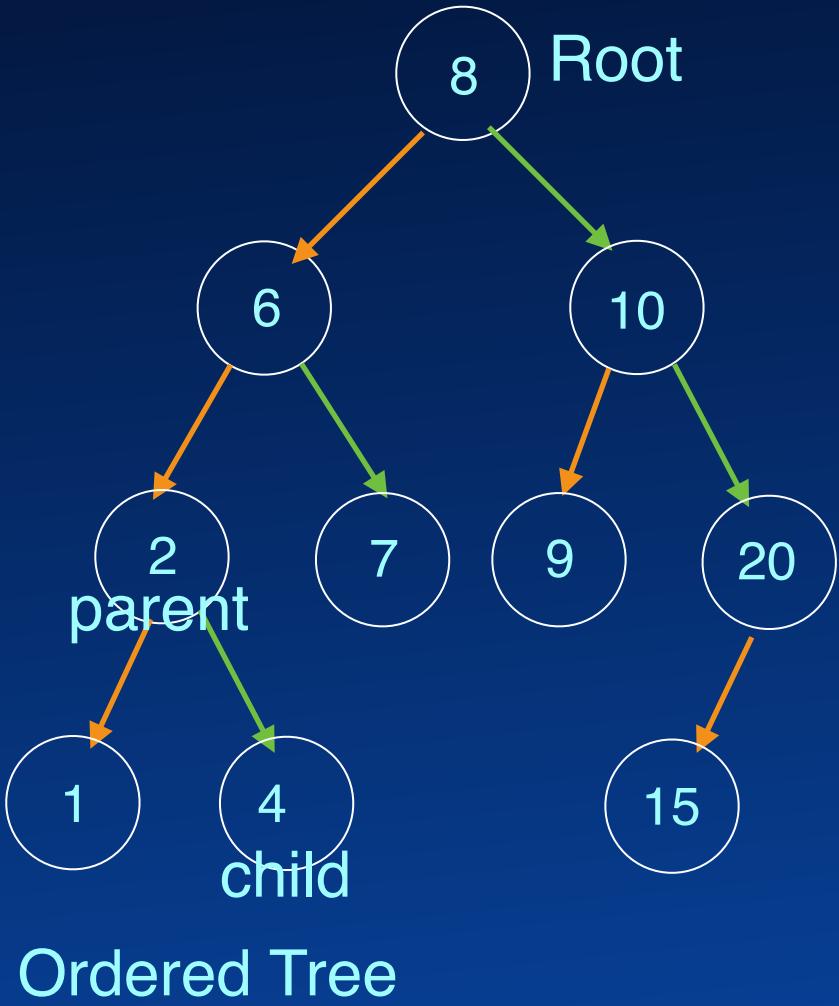
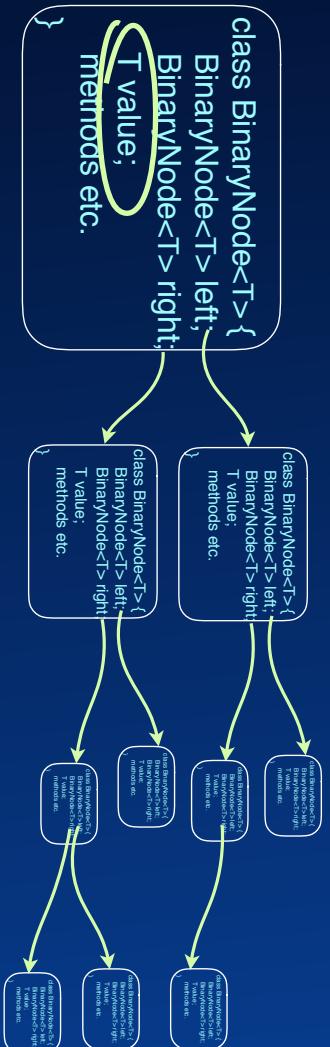
Binary Tree



Ordered Tree

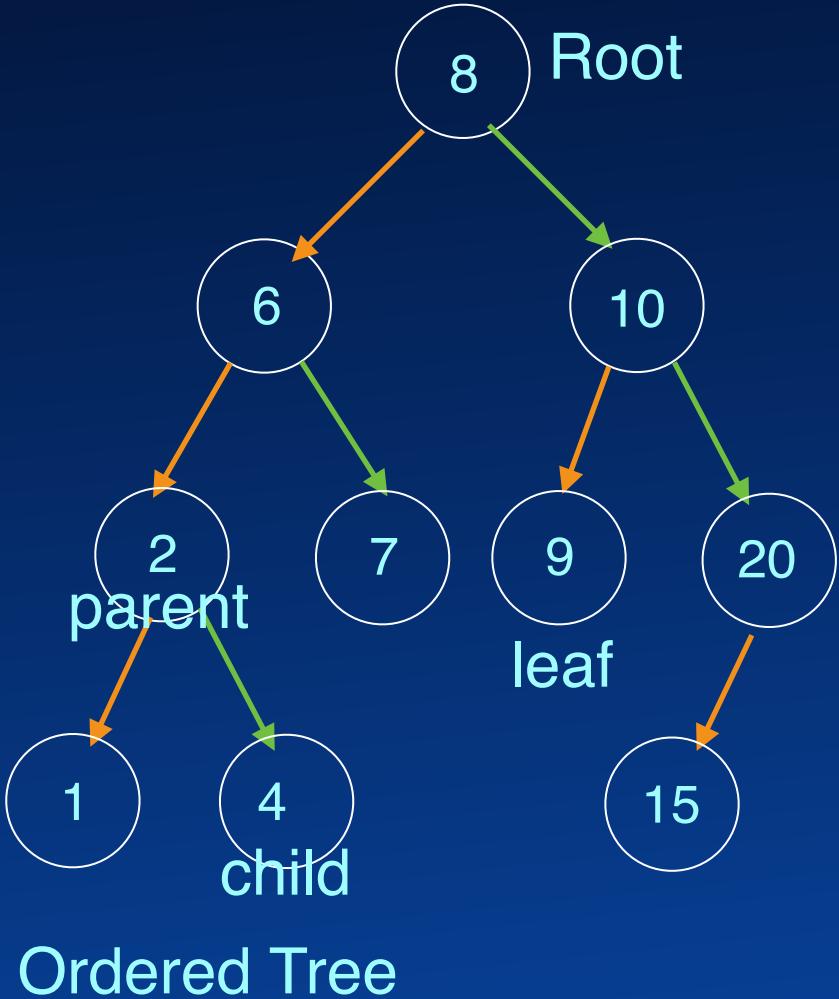
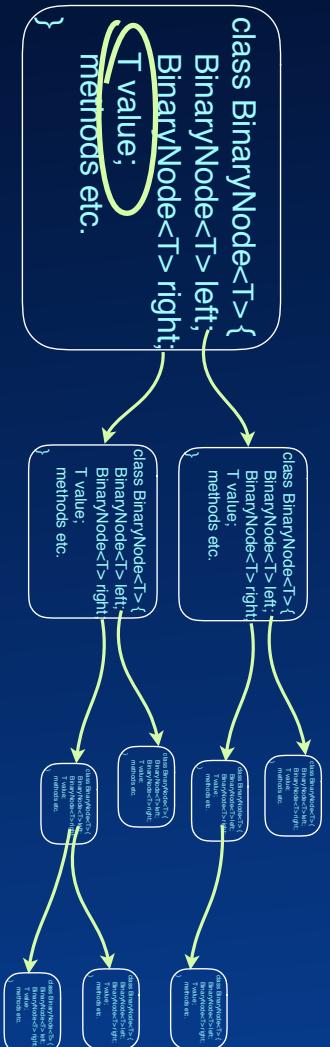


Binary Tree



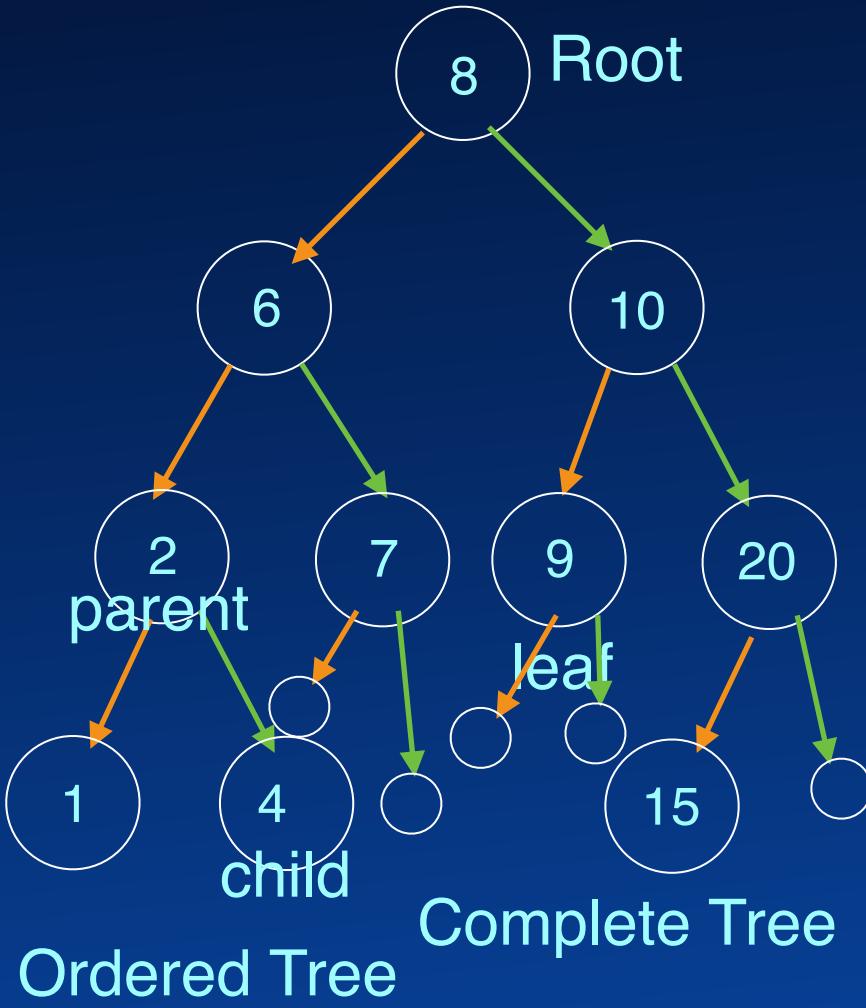
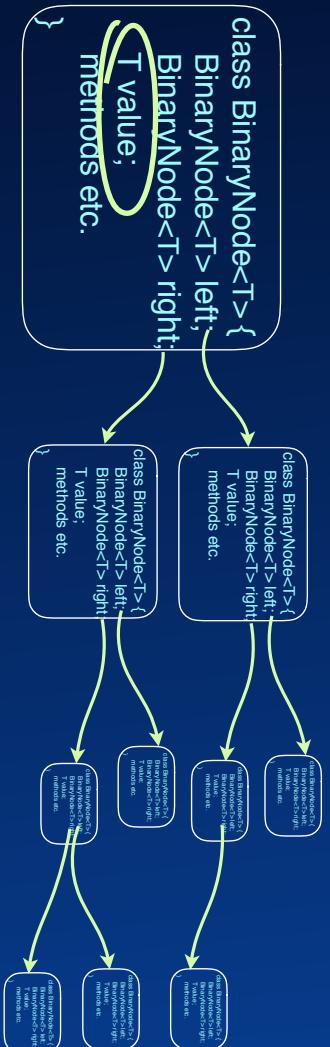


Binary Tree



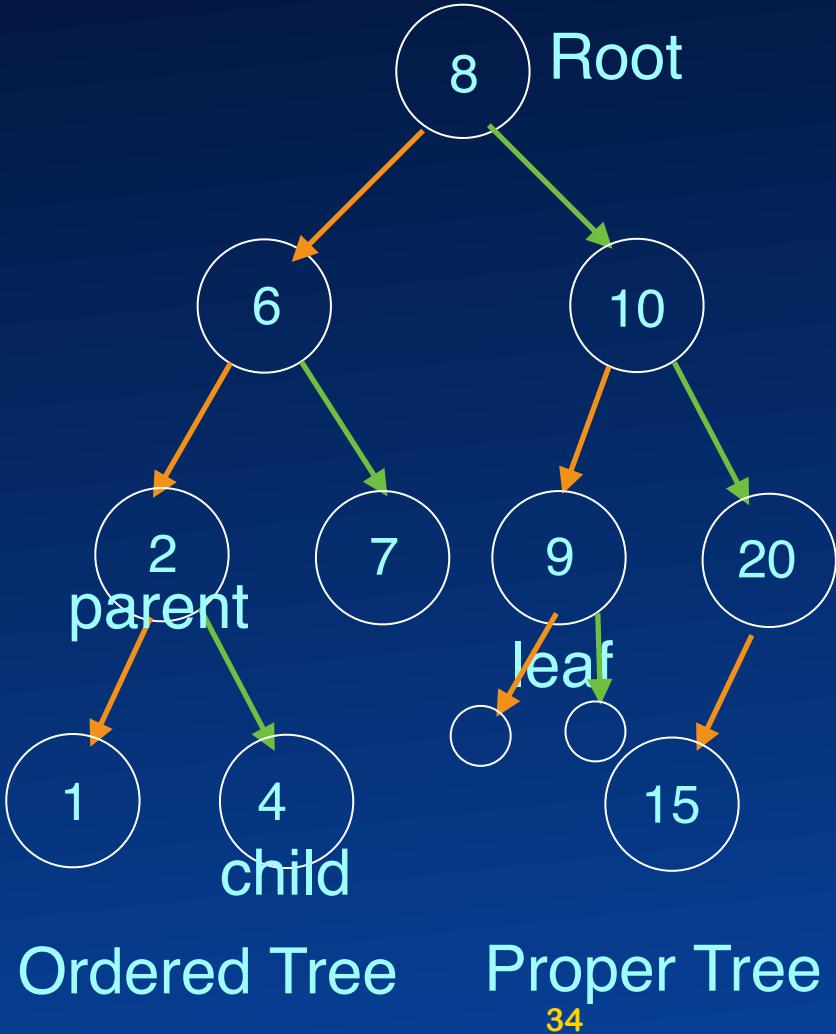
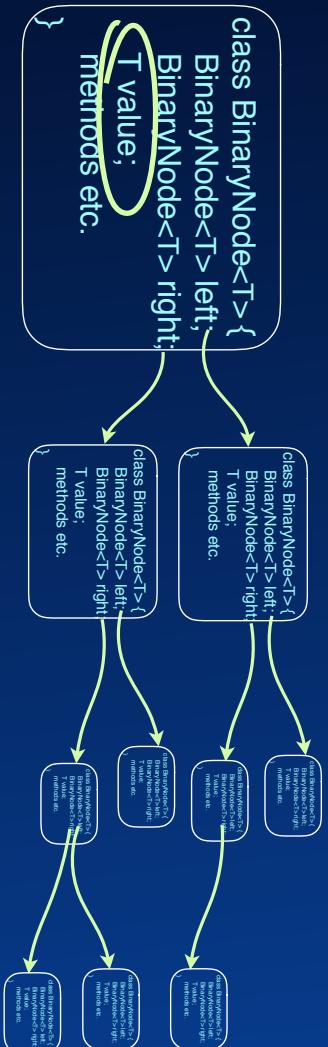


Binary Tree



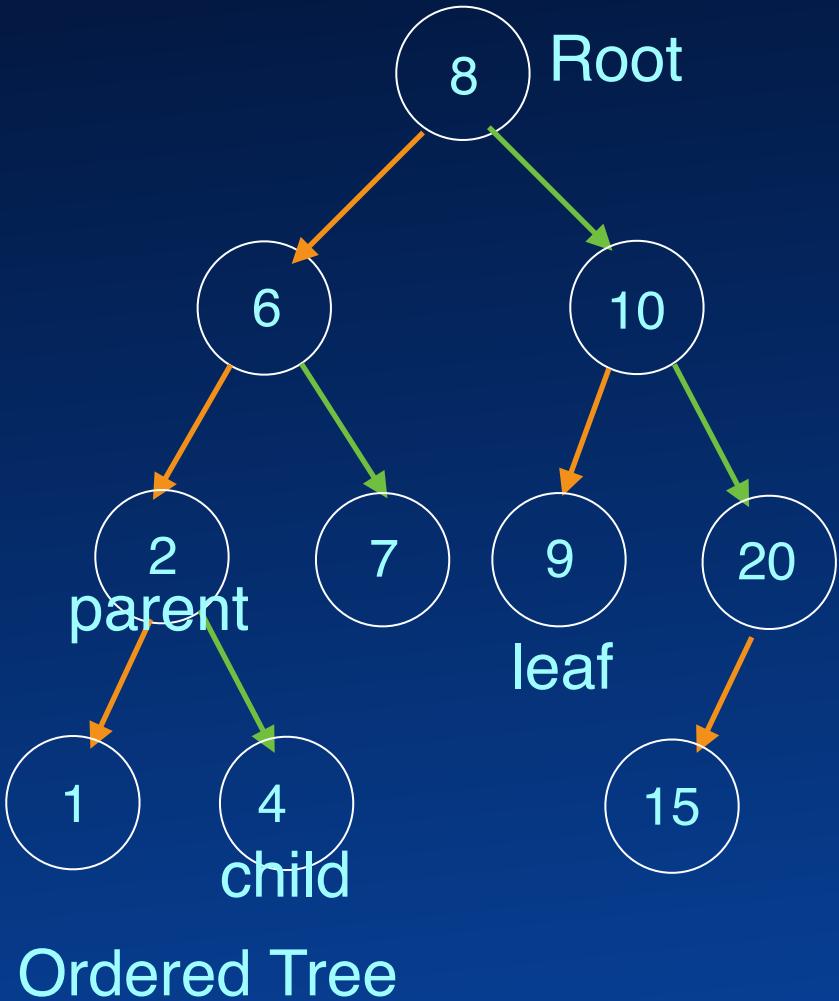
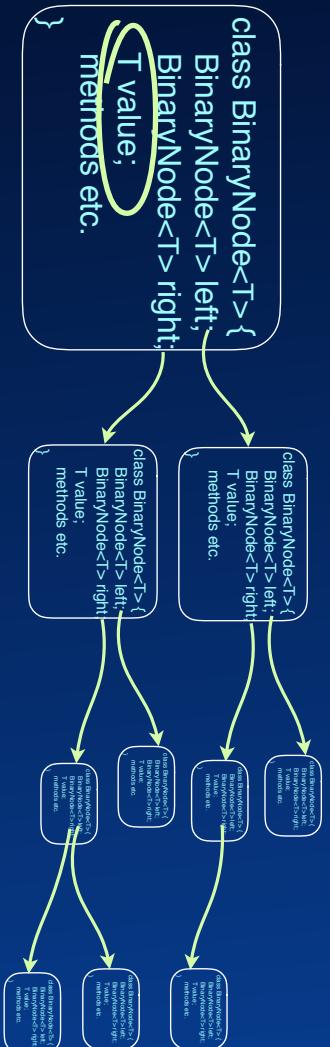


Binary Tree



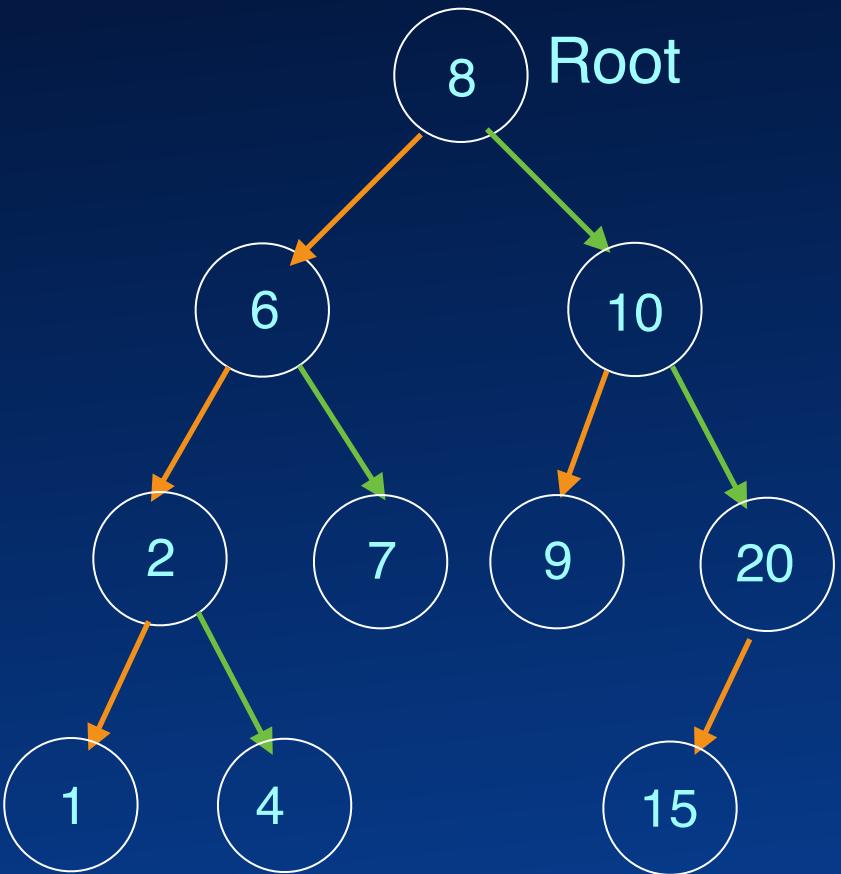


Binary Tree





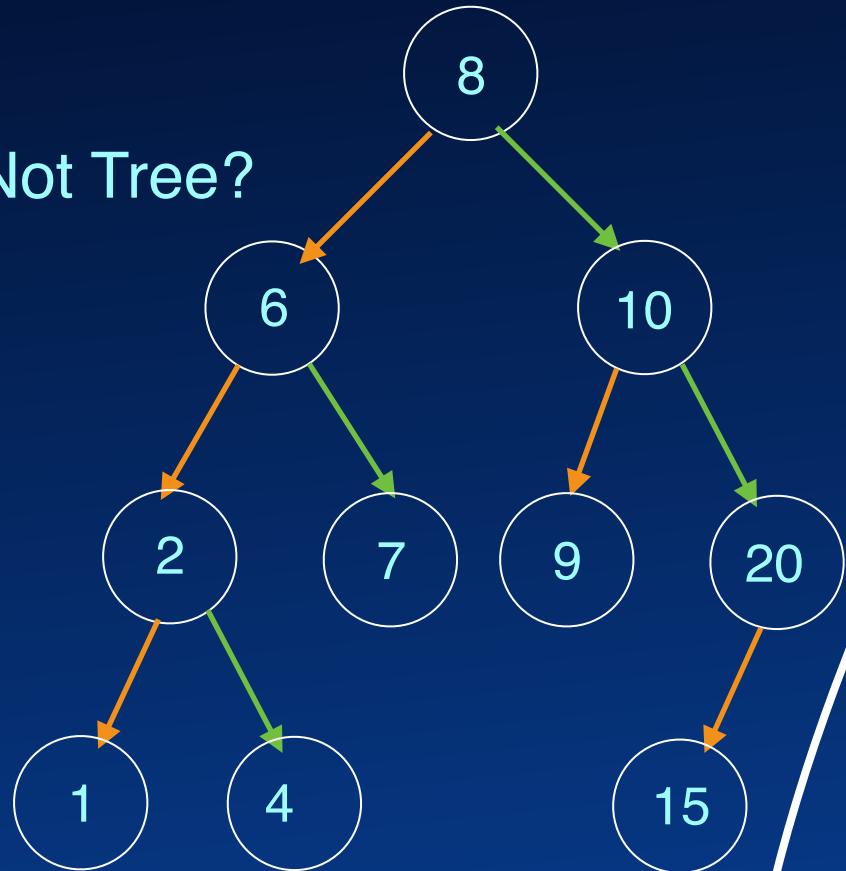
Tree



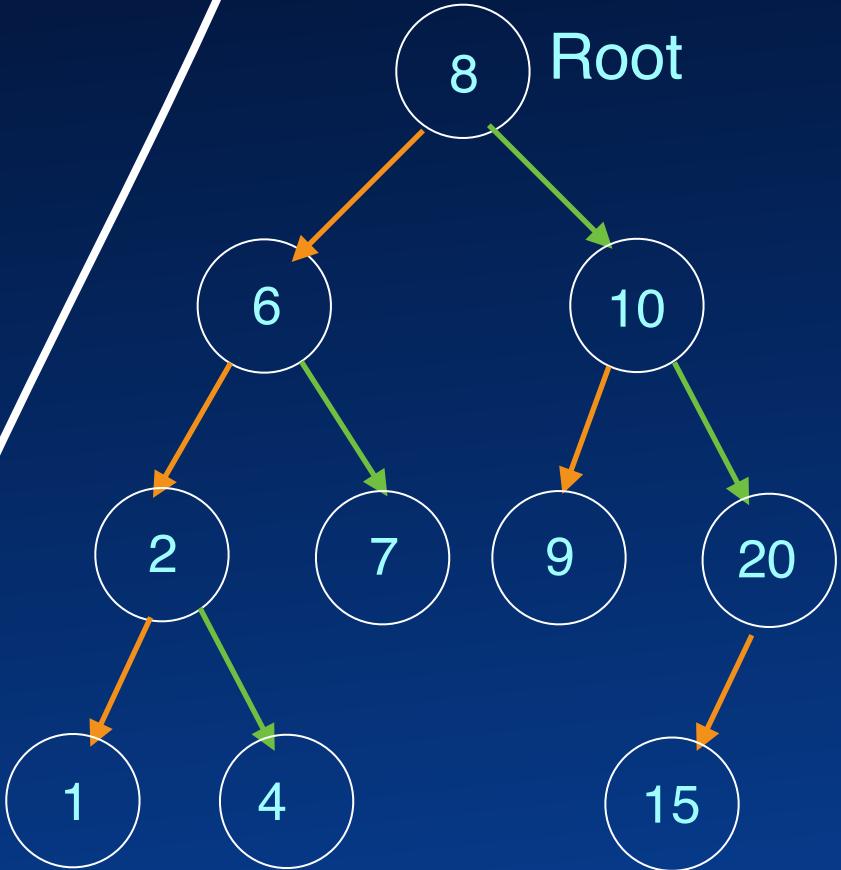


Tree

Not Tree?



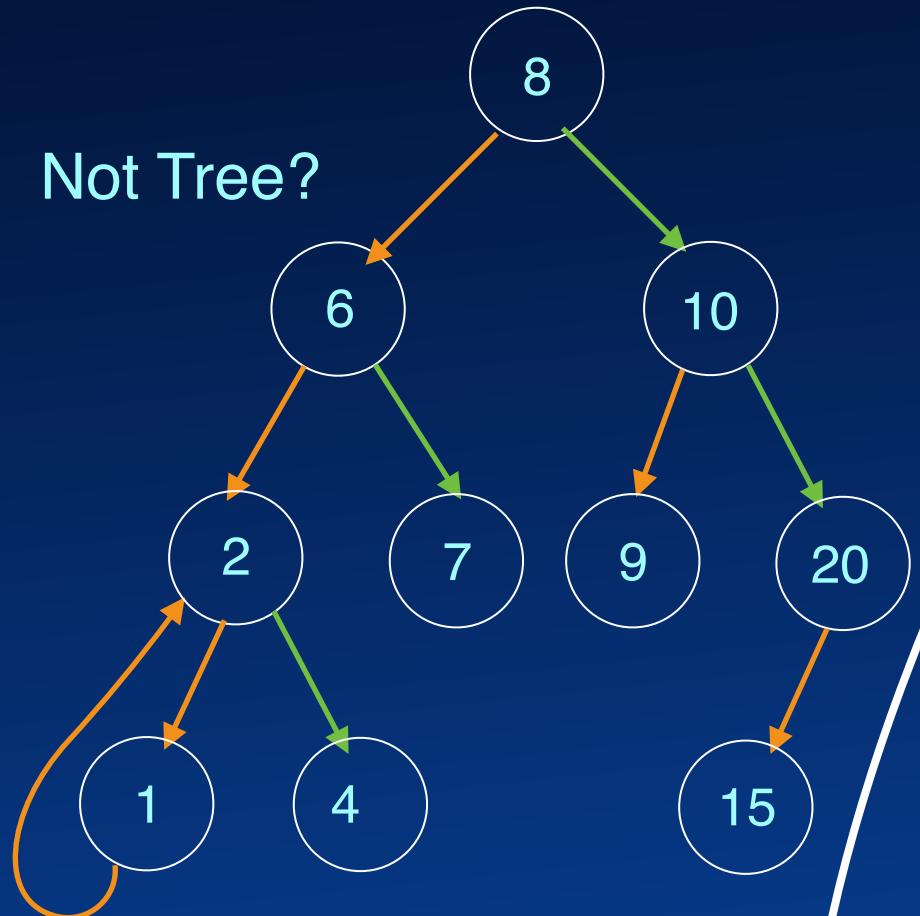
Root



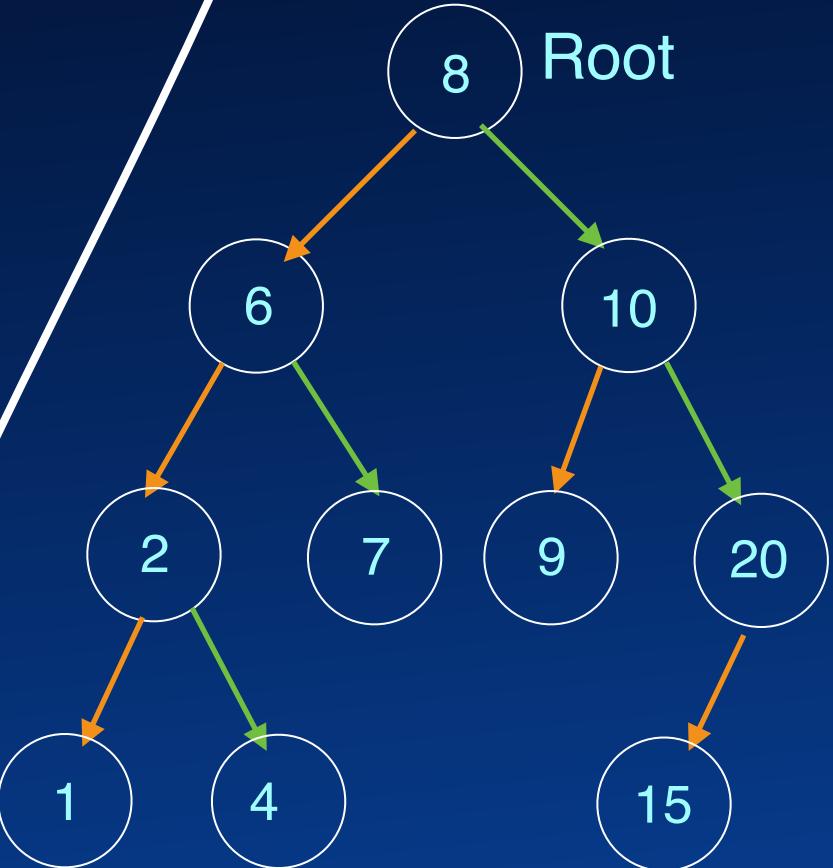


Tree

Not Tree?



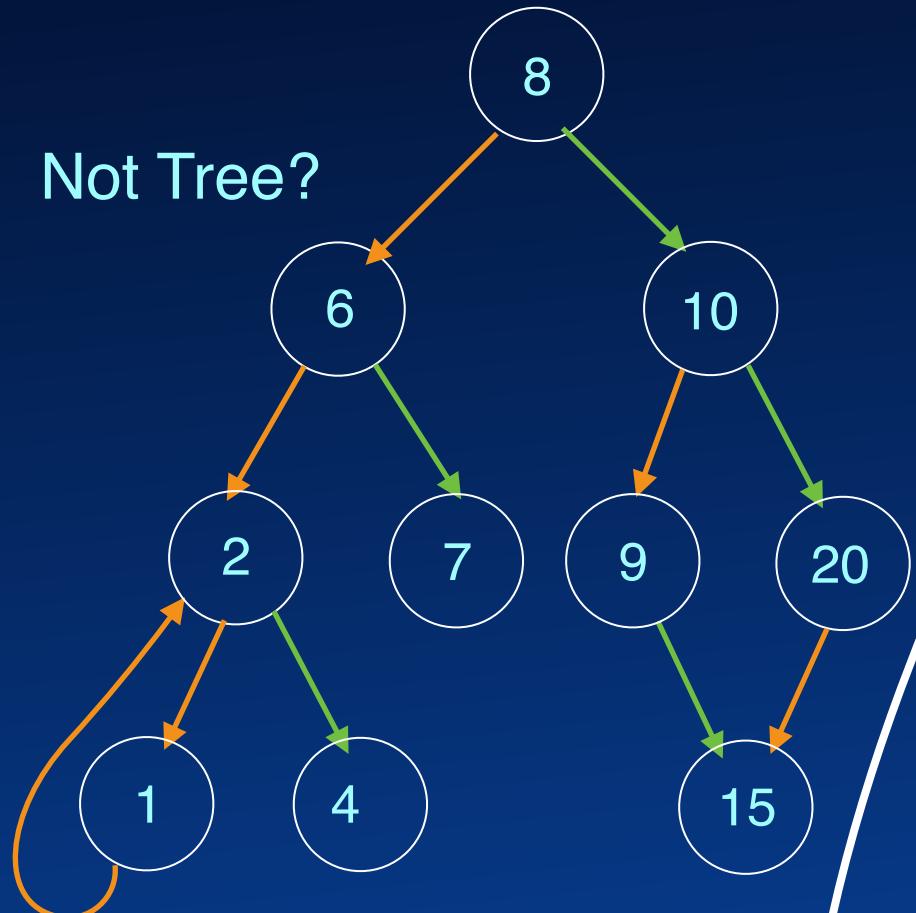
Root



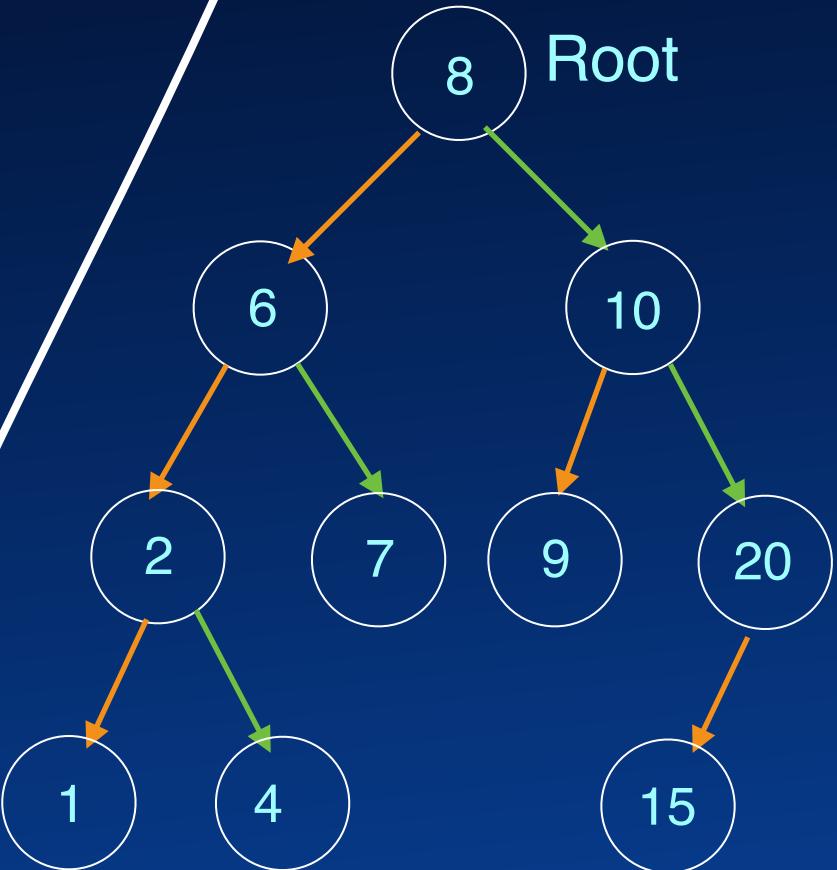


Tree

Not Tree?



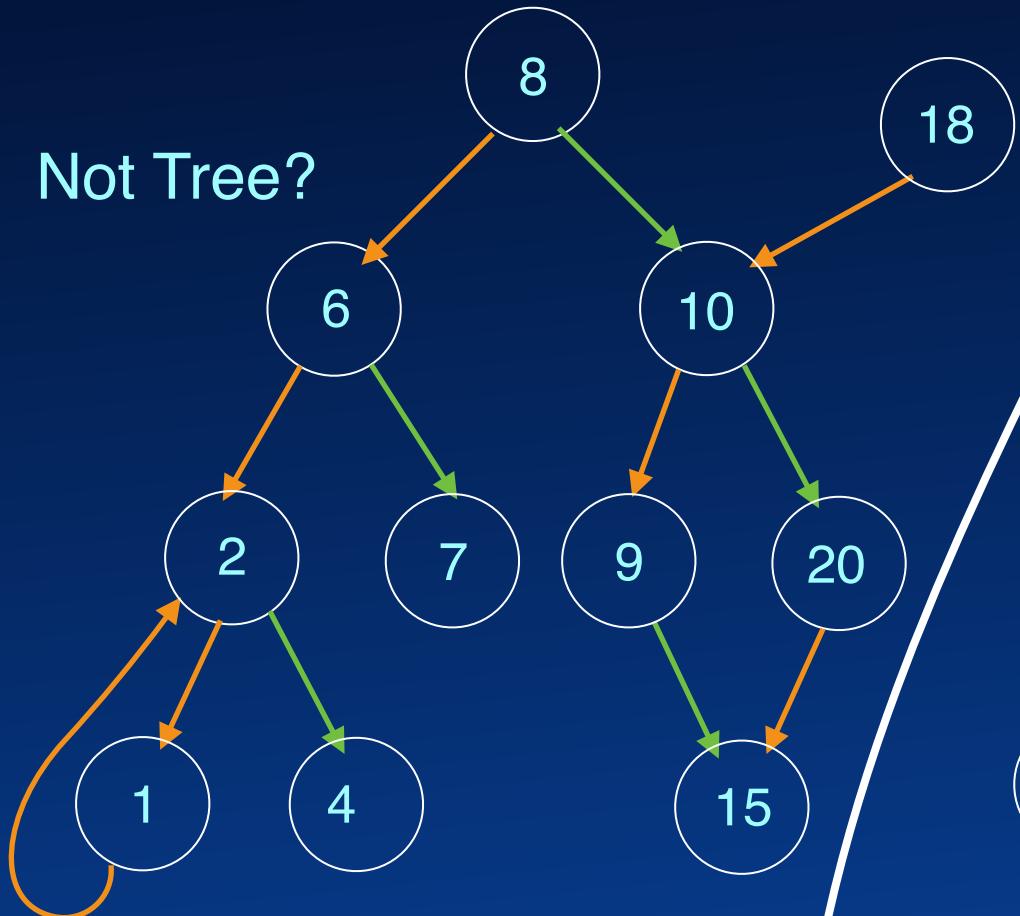
Root



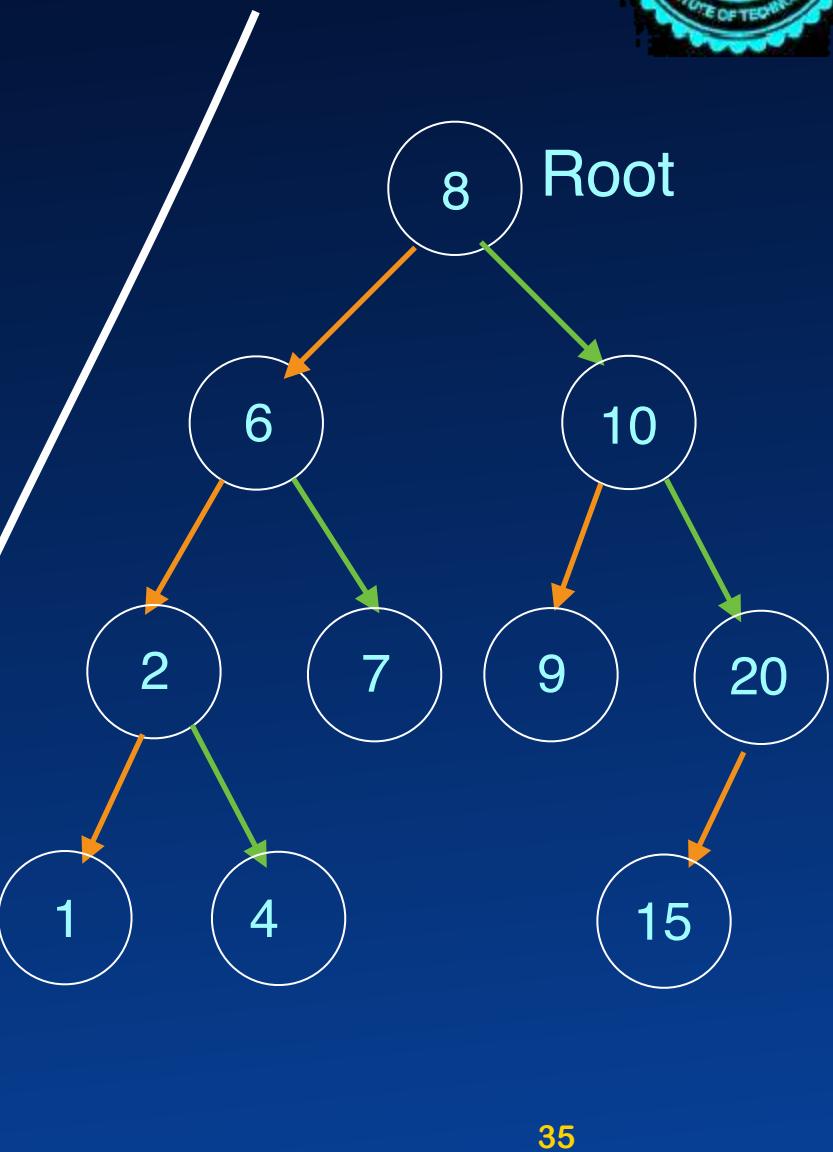


Tree

Not Tree?



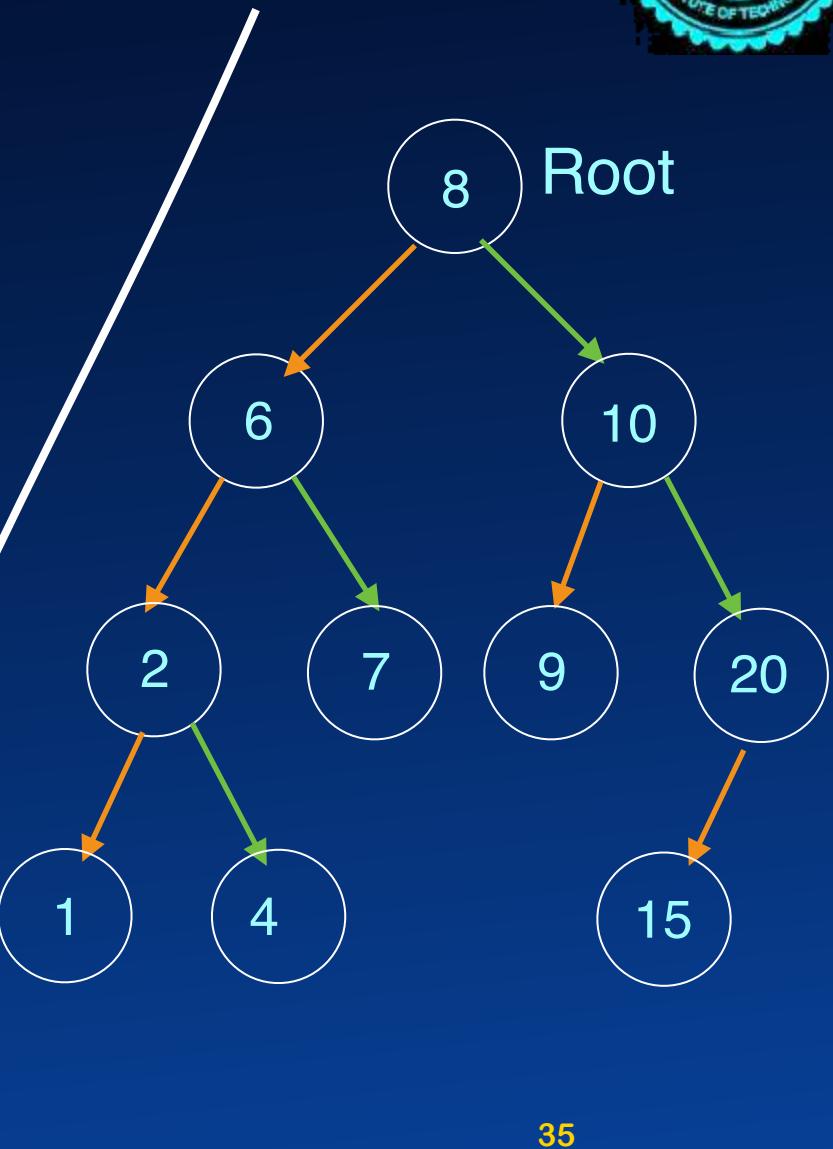
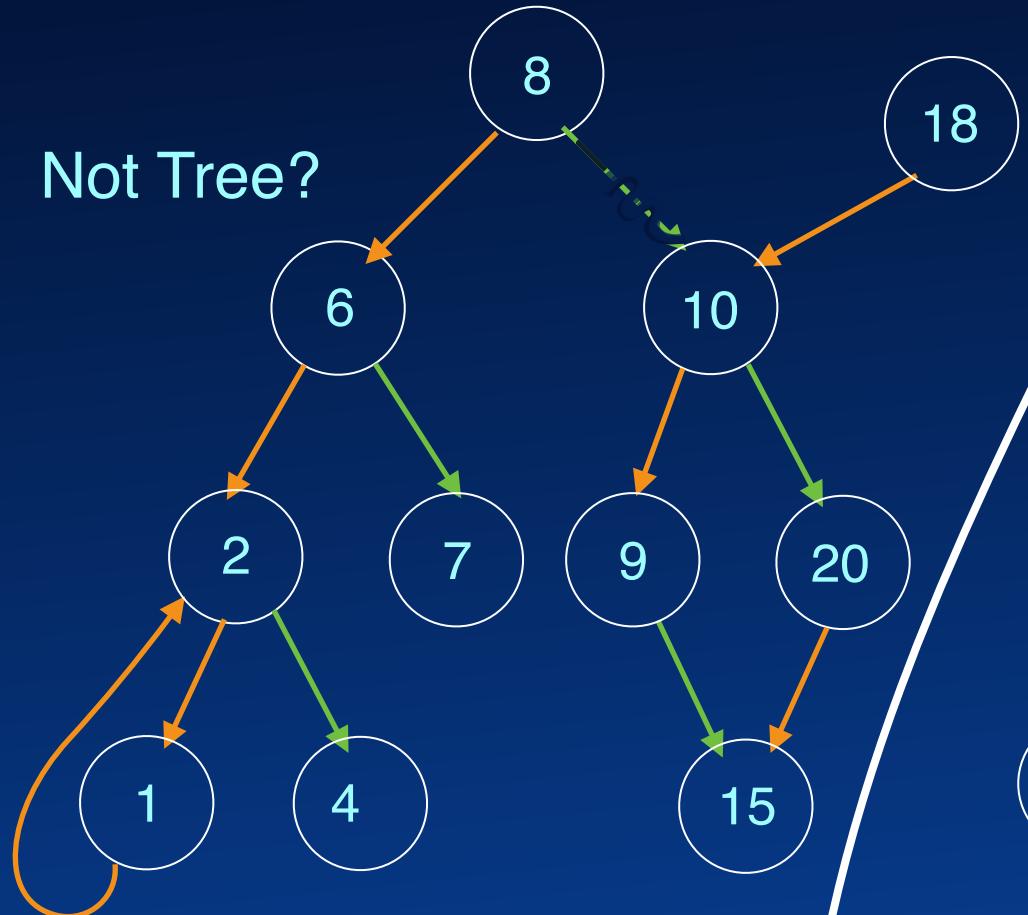
Root





Tree

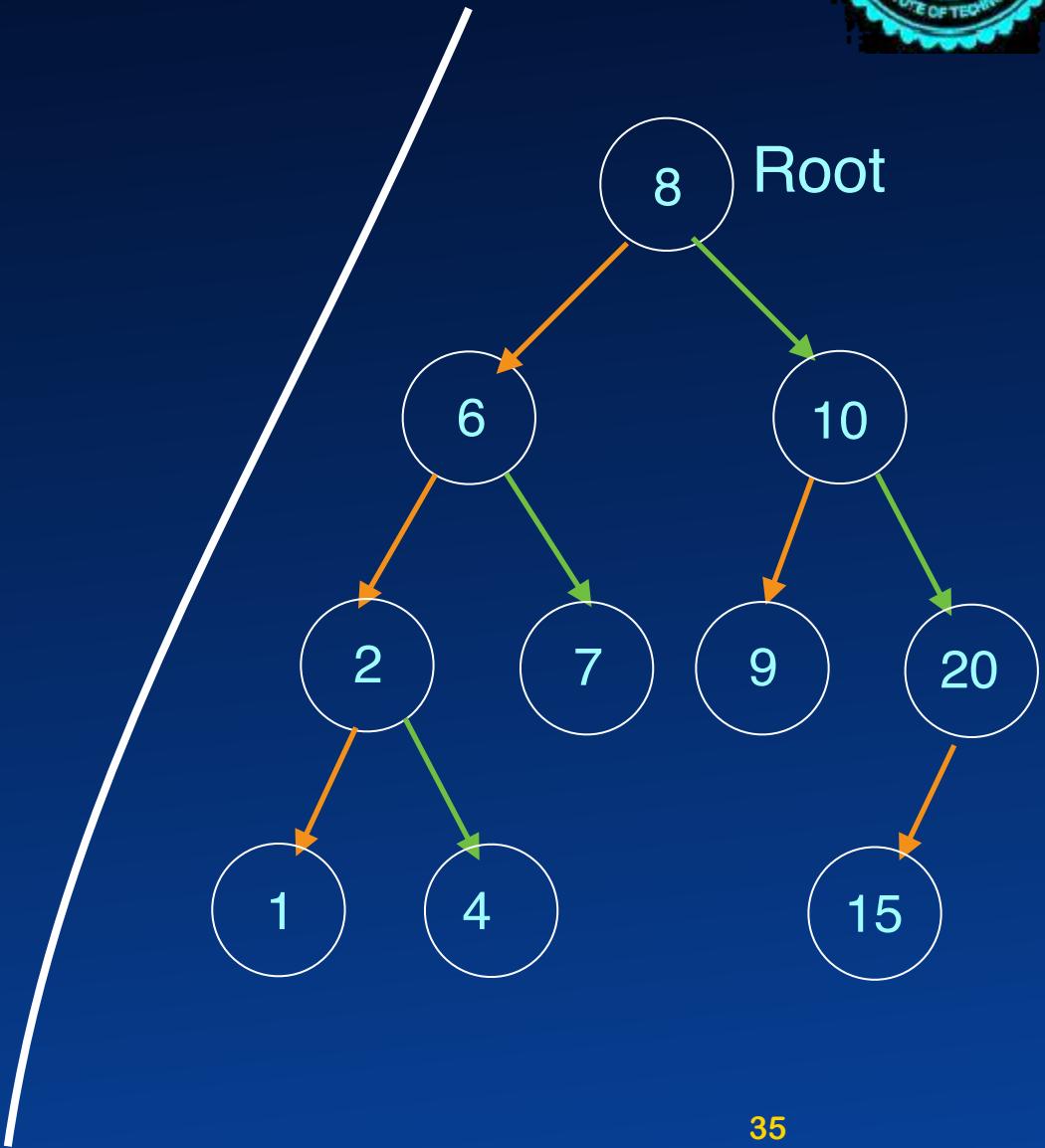
Not Tree?





Tree

Tree



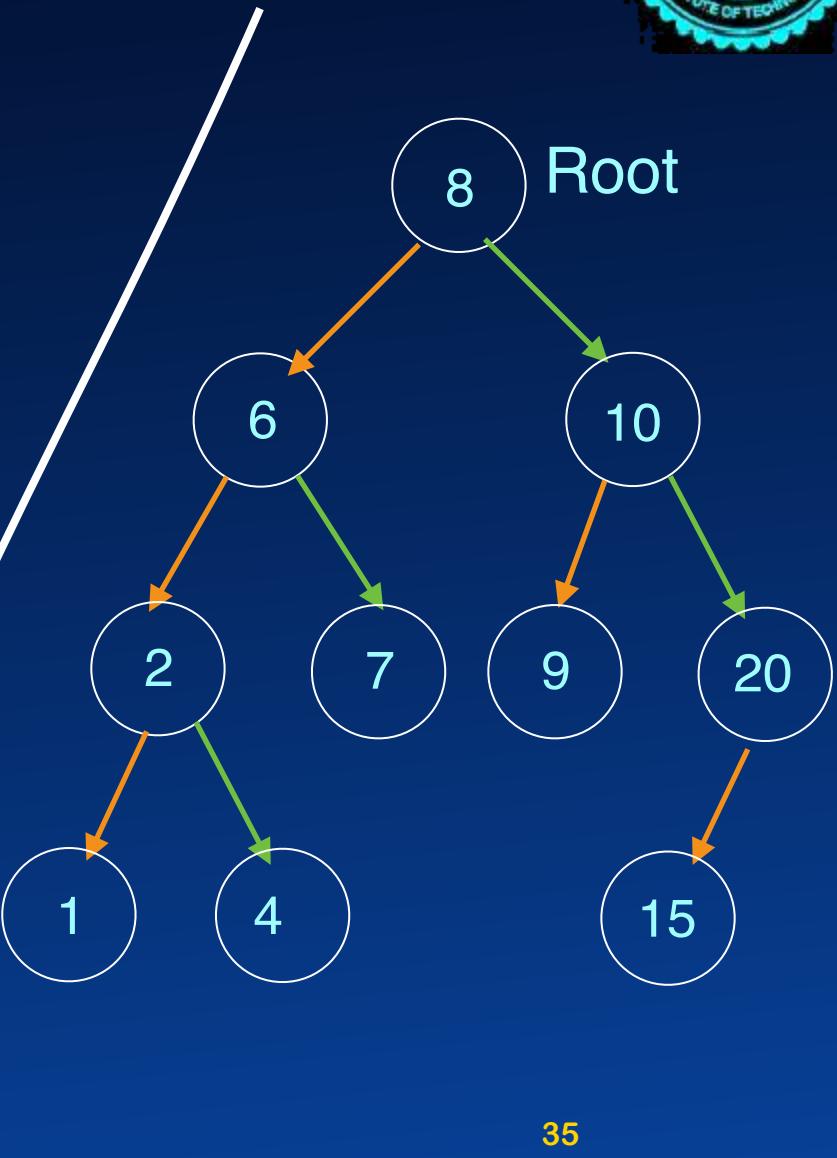


Tree

Tree



Root





Tree

Tree

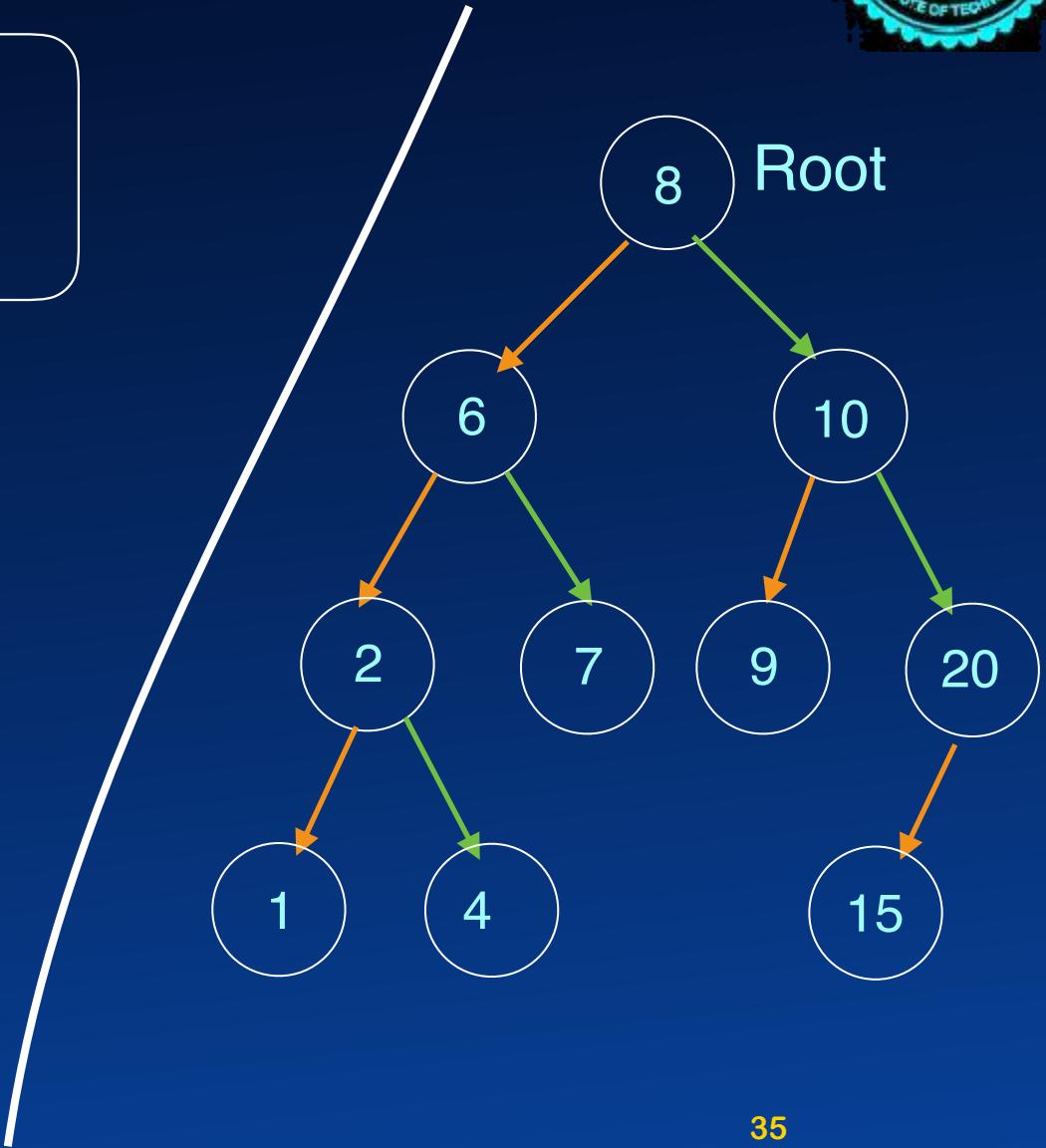


35



Tree

Tree



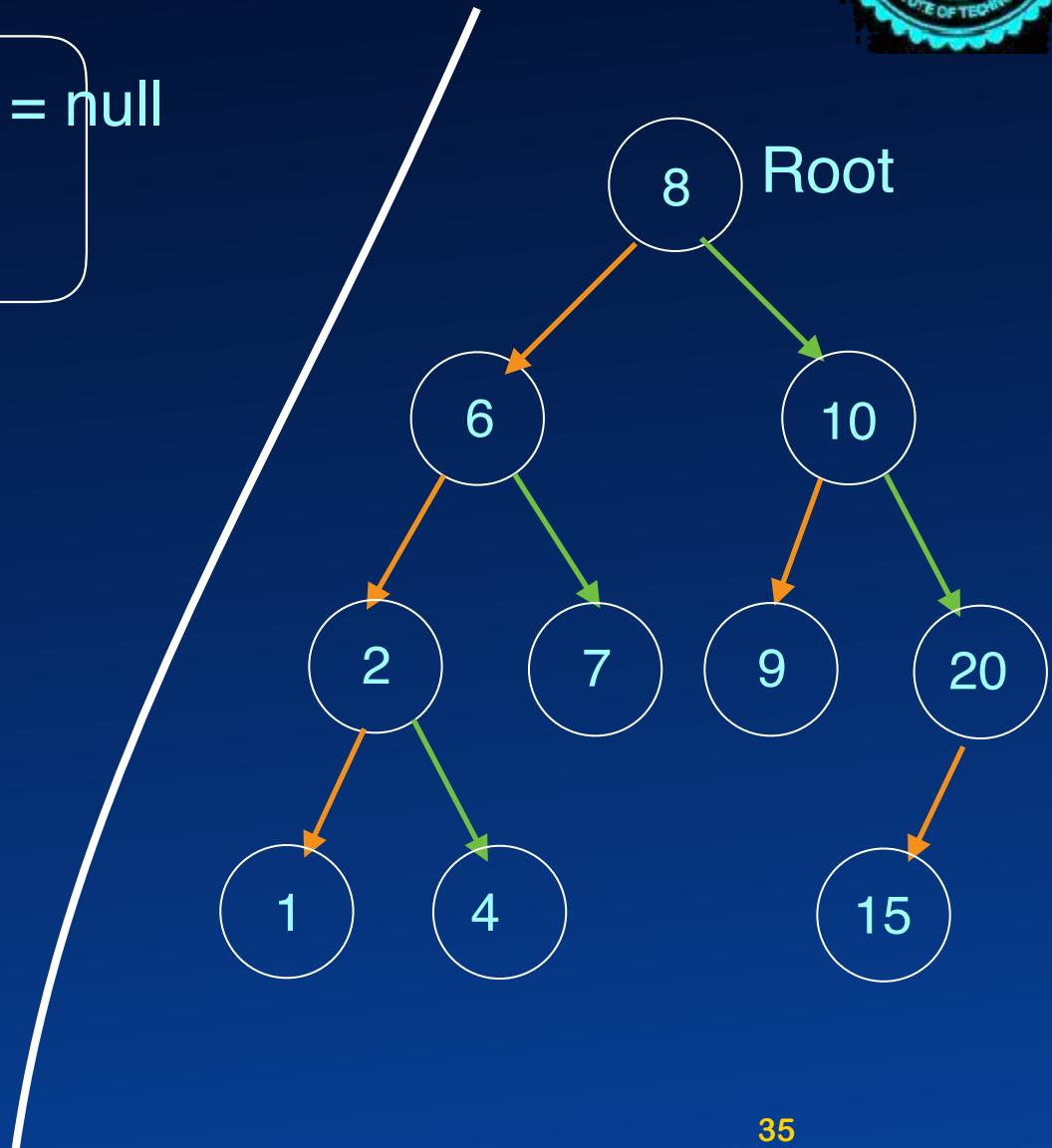


Tree

Tree

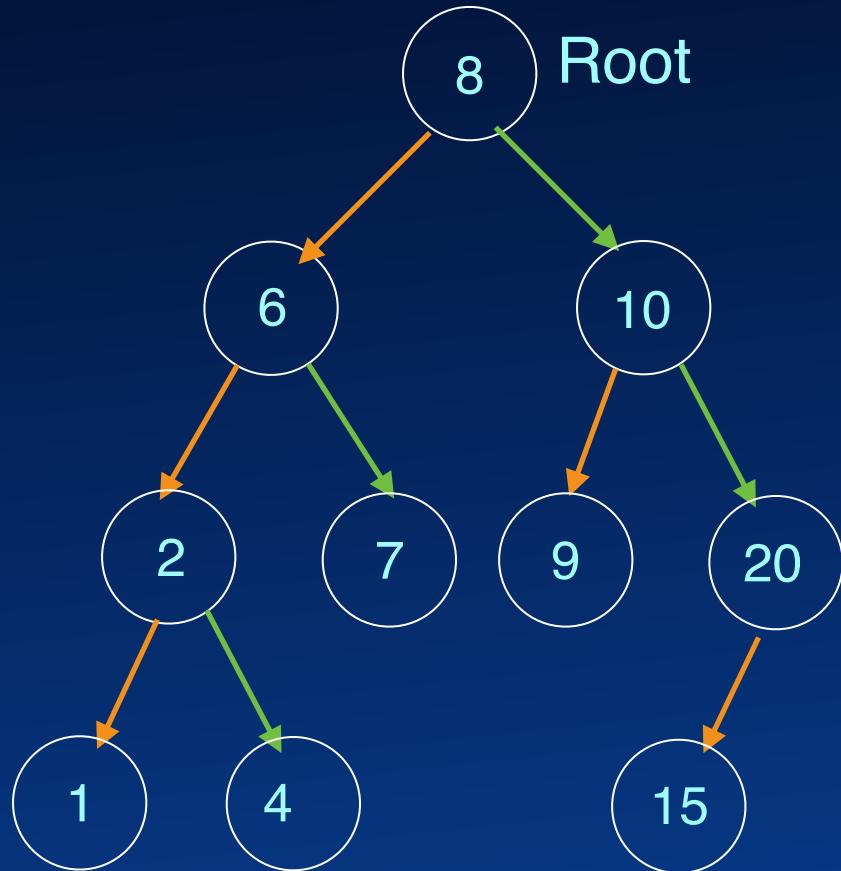


Root = null



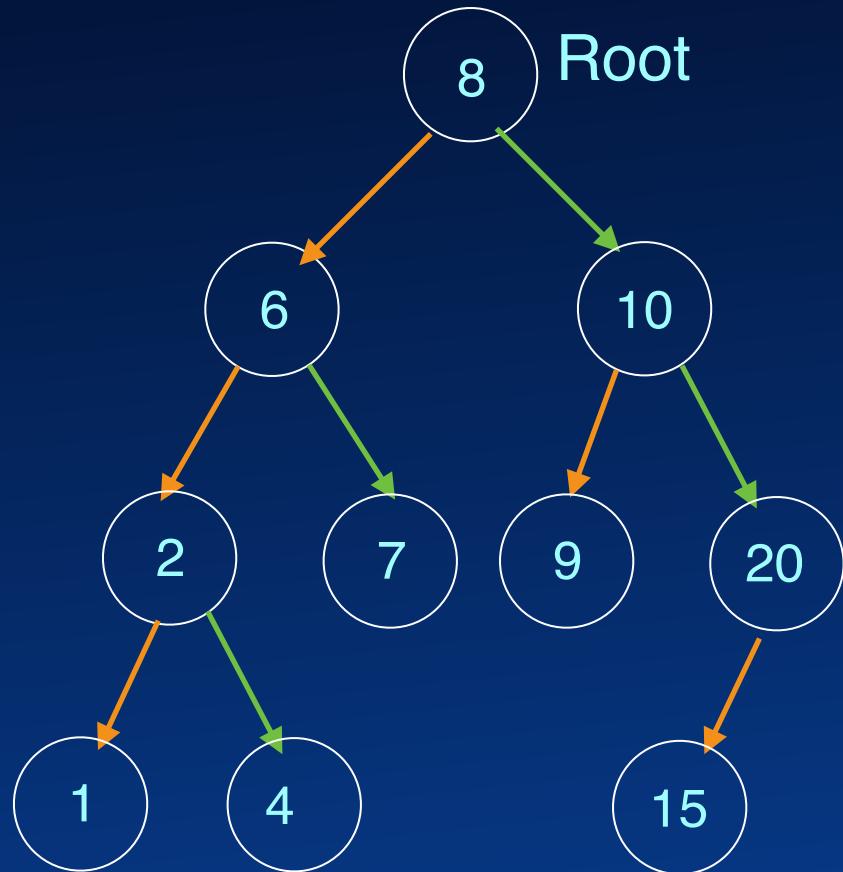


Tree



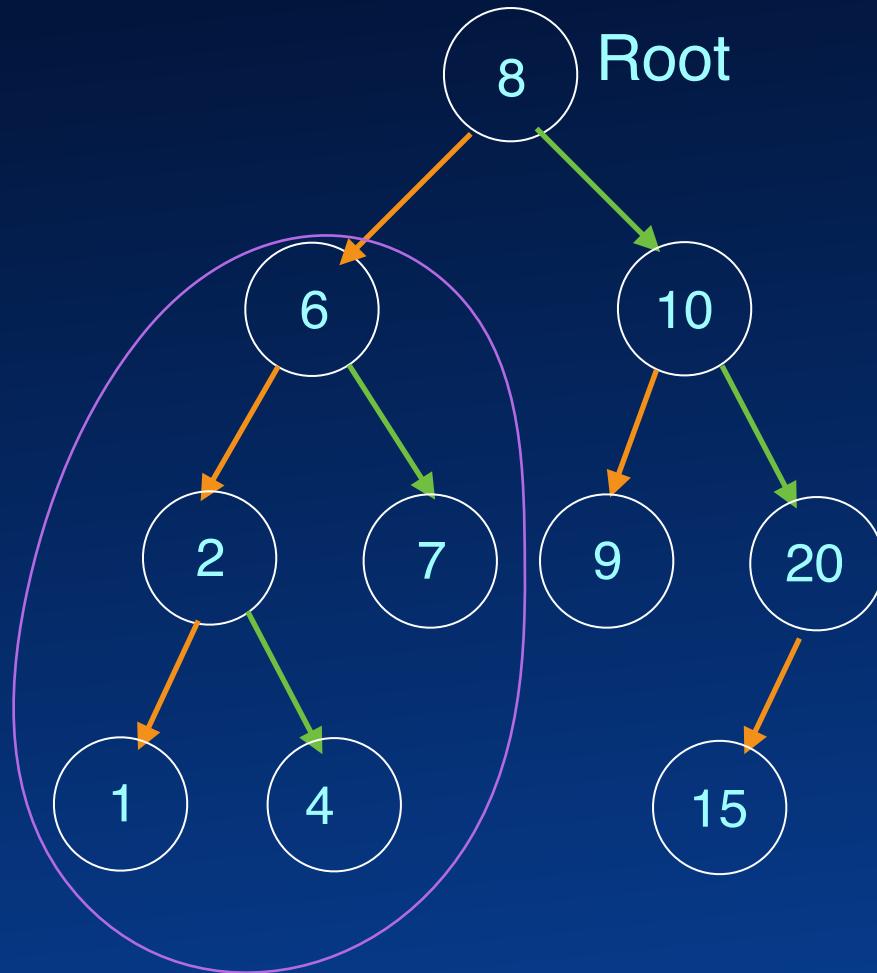


Tree



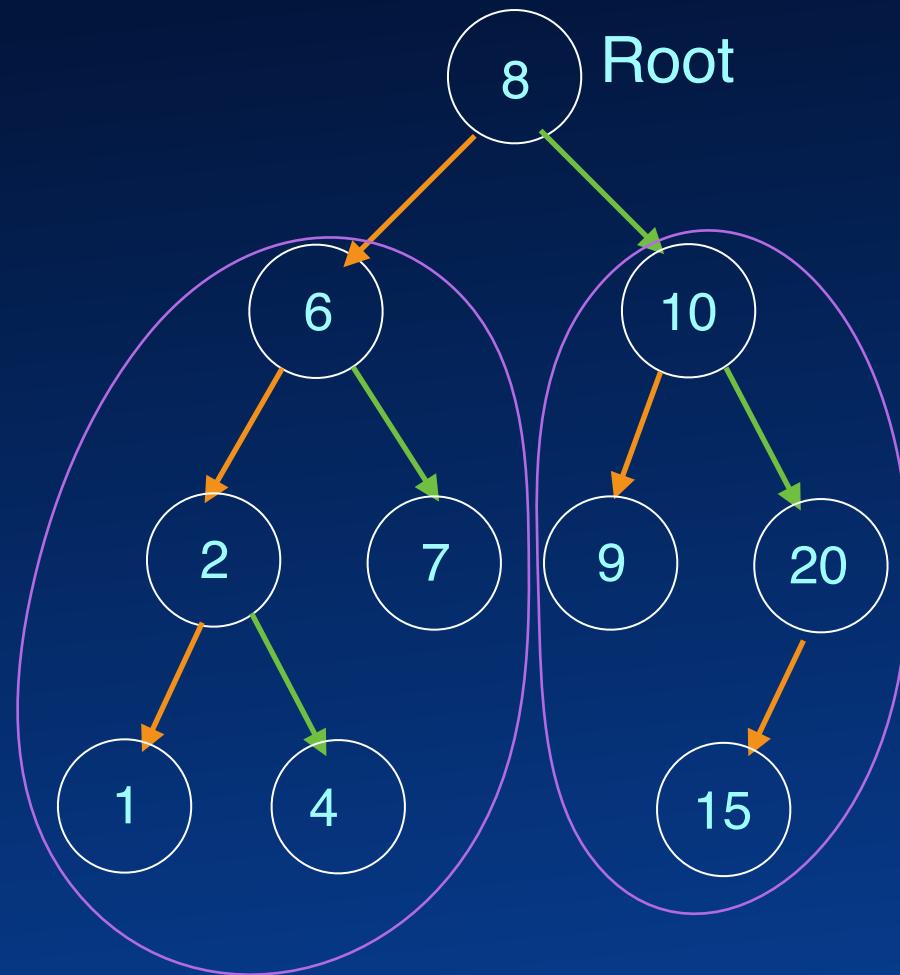


Tree



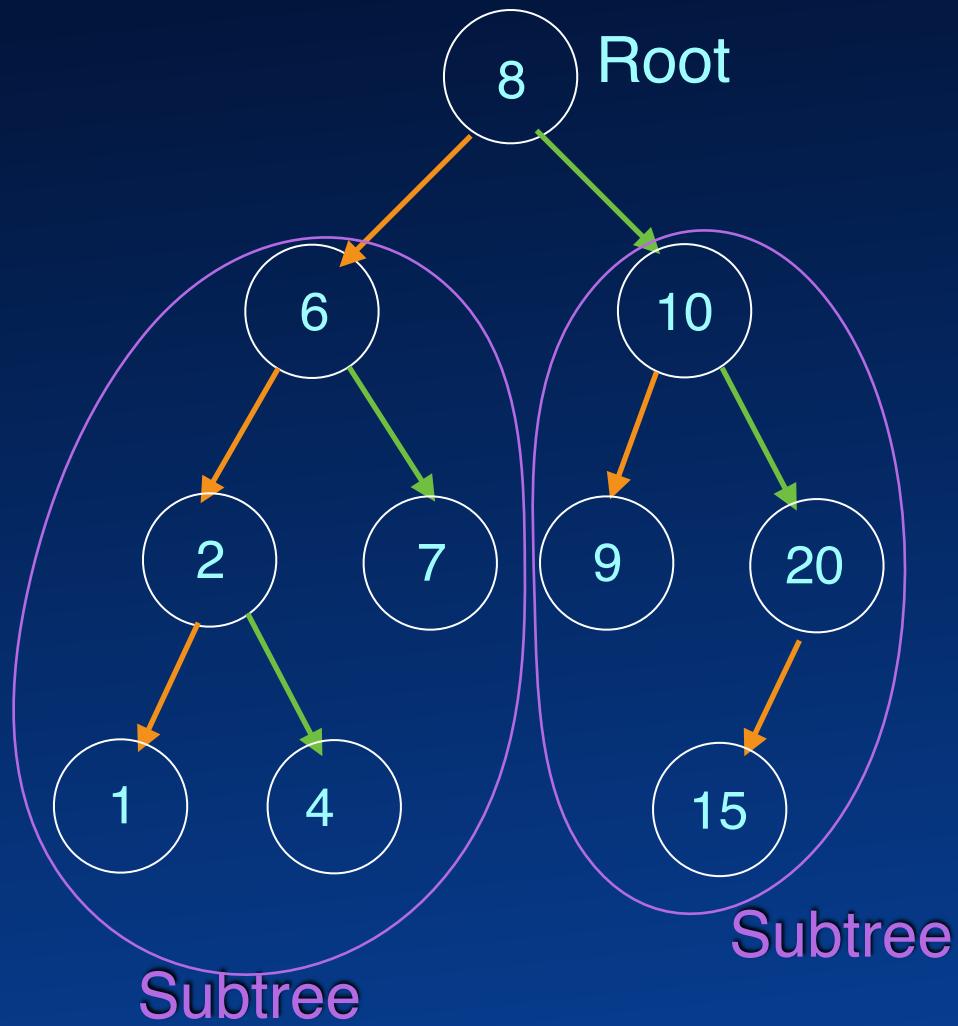


Tree



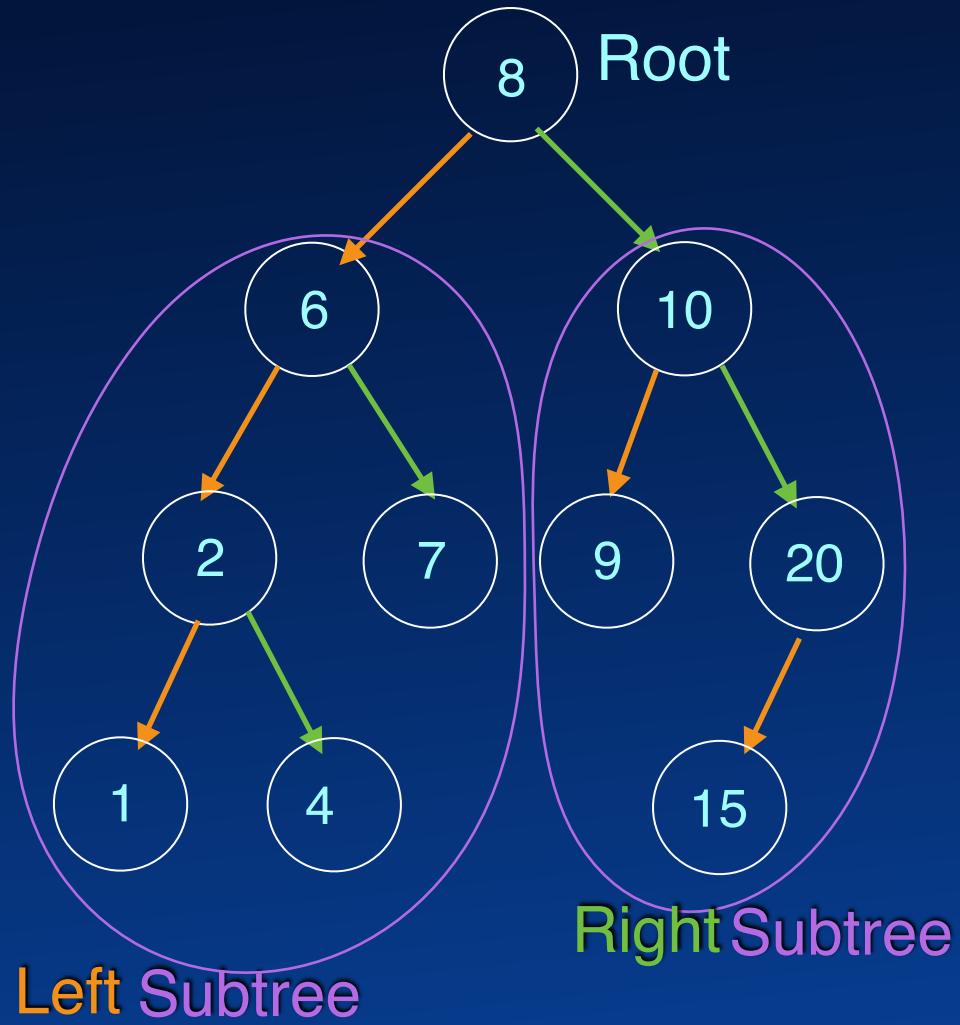


Tree



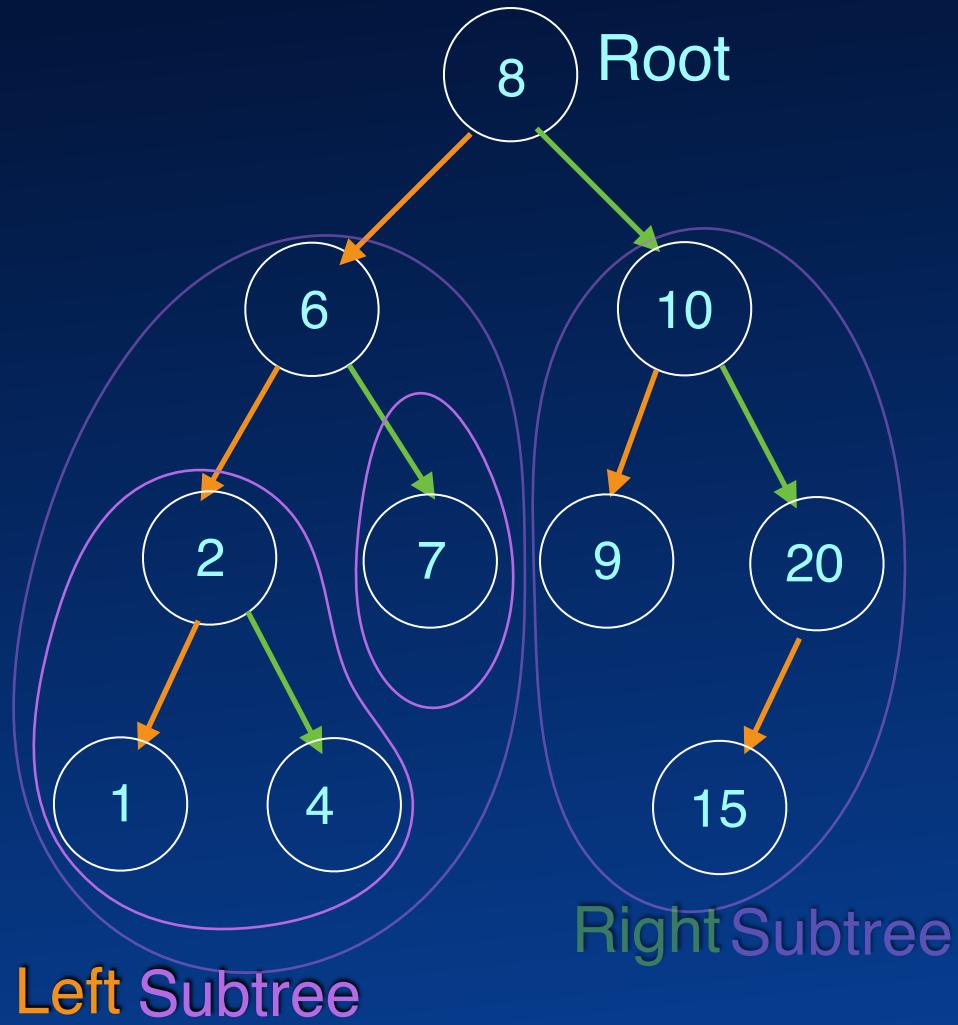


Tree



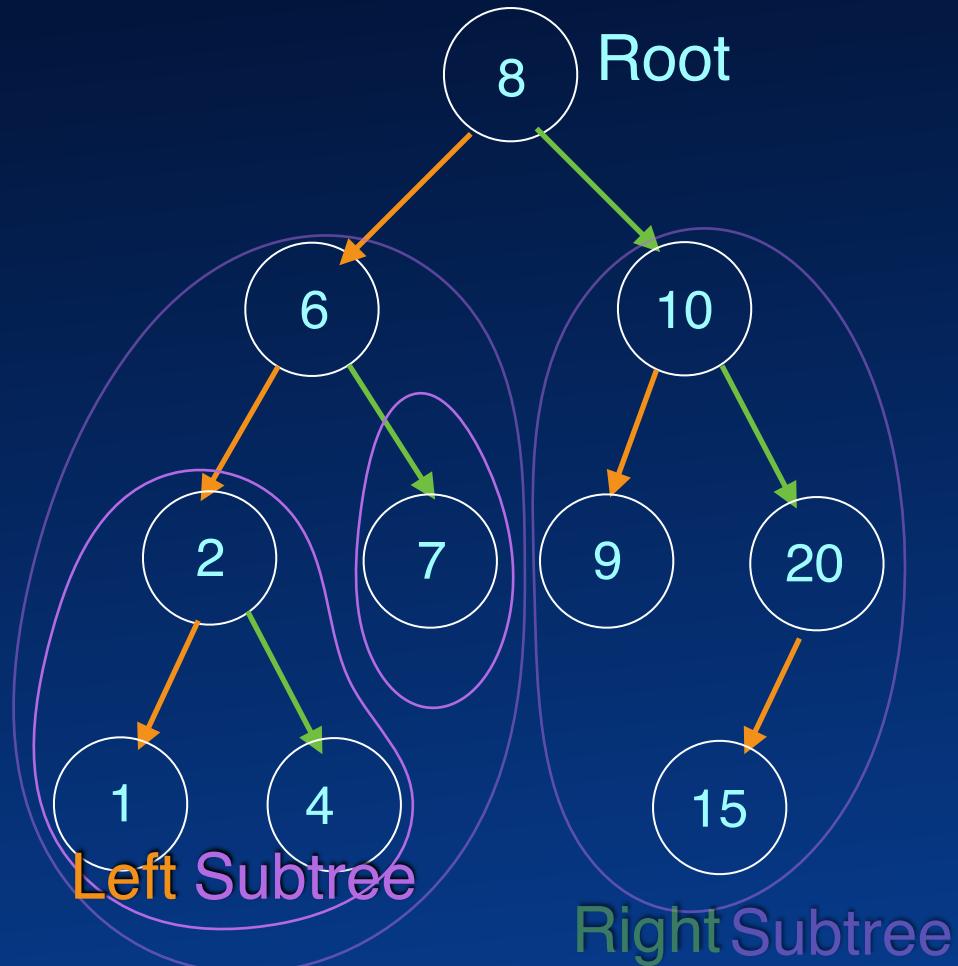


Tree



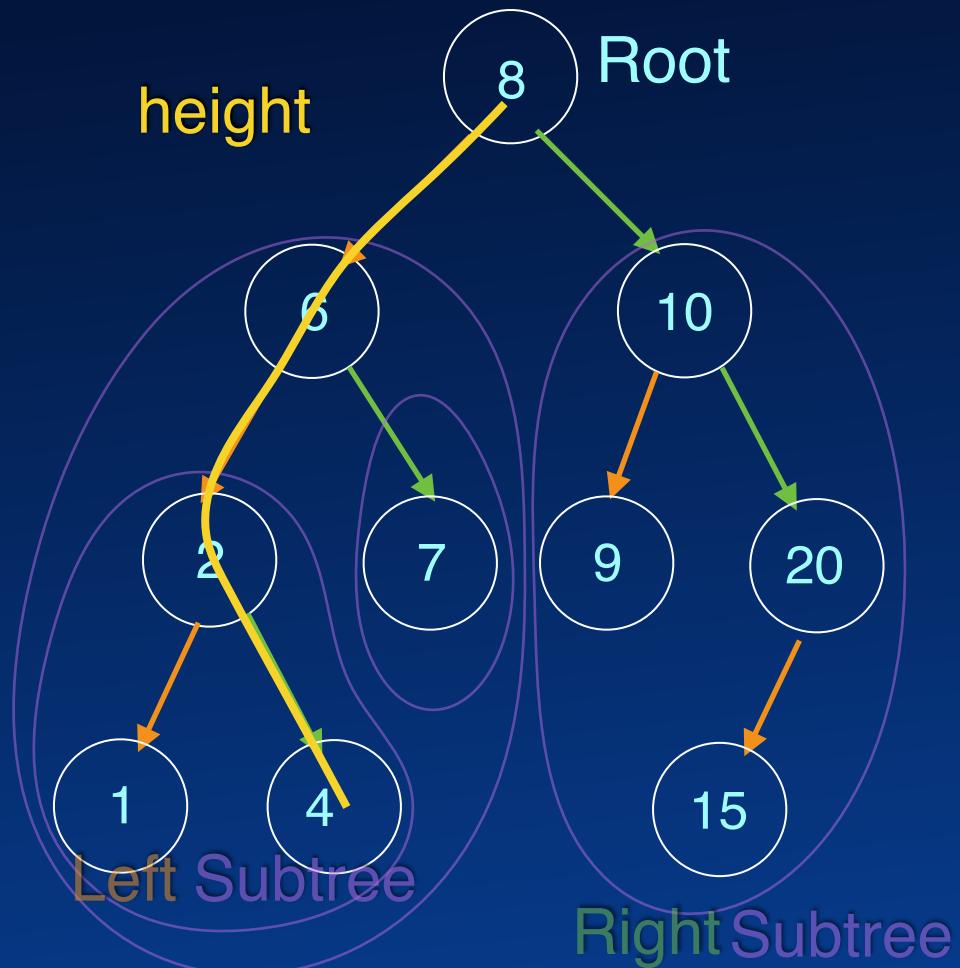


Tree



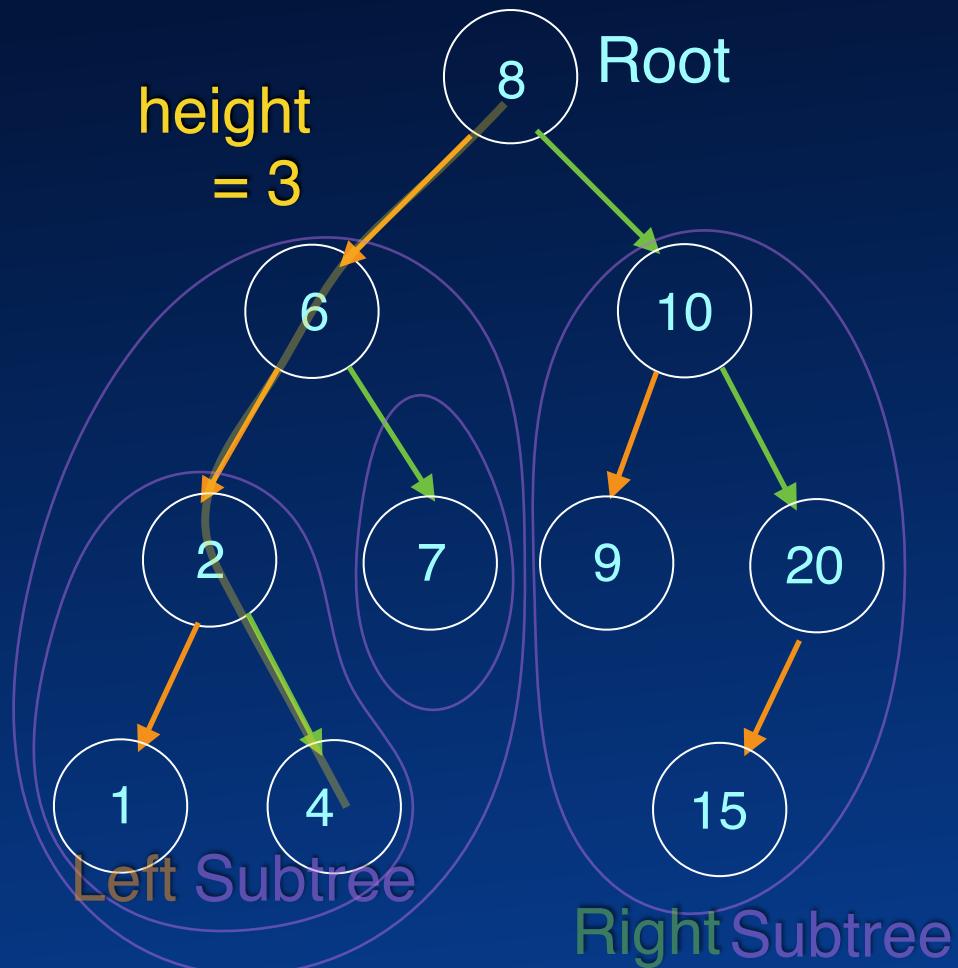


Tree





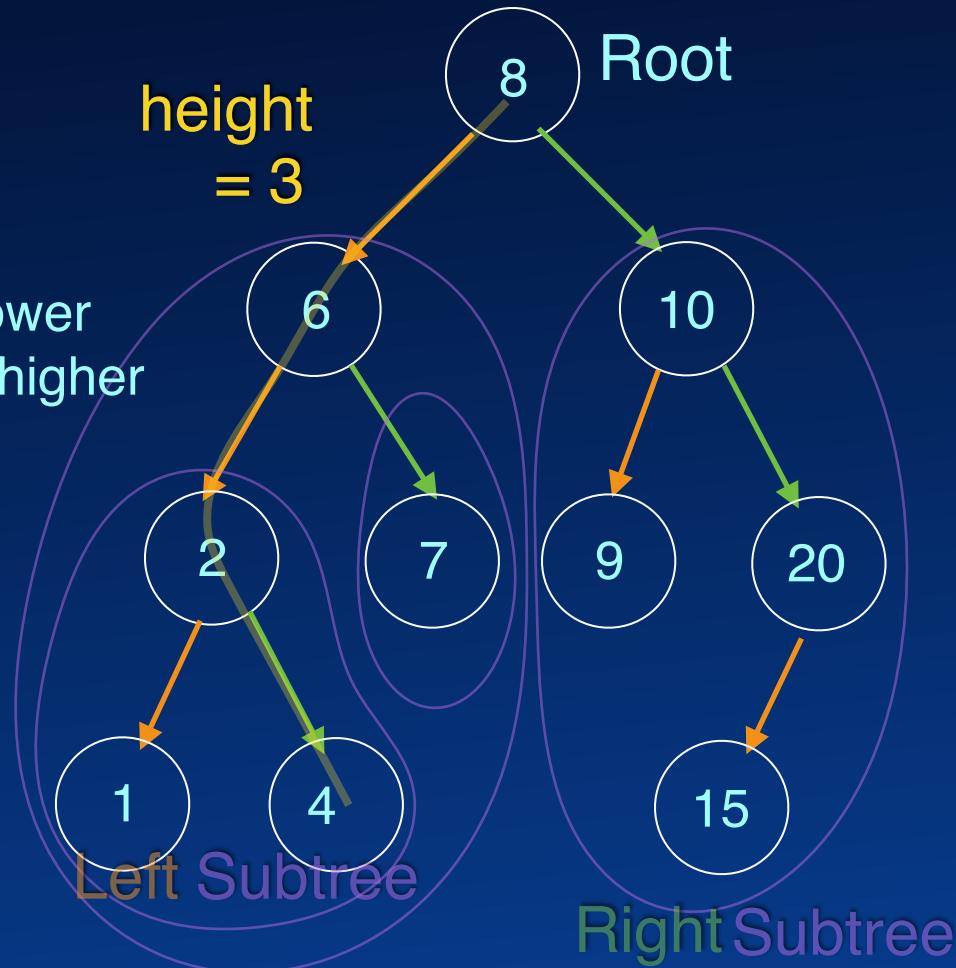
Tree





Tree

BST:
left subtree has lower
right subtree has higher





Tree Operations

```
class BinaryNode<T> {  
    BinaryNode<T> left;  
    BinaryNode<T> right;  
    T value;  
}
```

```
class BinaryTree<T> {  
    BinaryNode<T> root;  
    int height() { return height(root); }  
    int height(BinaryNode<T> node) {  
        if(node == null) return -1;  
        return(1 + max(  
            height(node.left), height(node.right)));  
    }  
    int count() { return count(root); }  
    int count(BinaryNode<T> node) {  
        if(node == null) return 0;  
        return(1 +  
            count(node.left) + count(node.right));  
    }  
}
```



Tree Operations

```
class BinaryNode<T> {  
    BinaryNode<T> left;  
    BinaryNode<T> right;  
    T value;  
}
```

```
class BinaryTree<T> {  
    BinaryNode<T> root;  
    int height() { return height(root); }  
    int height(BinaryNode<T> node) {  
        if(node == null) return -1;  
        return(1 + max(  
            height(node.left), height(node.right)));  
    }  
    int count() { return count(root); }  
    int count(BinaryNode<T> node) {  
        if(node == null) return 0;  
        return(1 +  
            count(node.left) + count(node.right));  
    }  
}
```



Tree Operations

```
class BinaryNode<T> {  
    BinaryNode<T> left;  
    BinaryNode<T> right;  
    T value;  
}
```

```
class BinaryTree<T> {  
    BinaryNode<T> root;  
    int height() { return height(root); }  
    int height(BinaryNode<T> node) {  
        if(node == null) return -1;  
        return(1 + max(  
            height(node.left), height(node.right)));  
    }  
    int count() { return count(root); }  
    int count(BinaryNode<T> node) {  
        if(node == null) return 0;  
        return(1 +  
            count(node.left) + count(node.right));  
    }  
}
```



Tree Operations

```
class BinaryNode<T> {  
    BinaryNode<T> left;  
    BinaryNode<T> right;  
    T value;  
}
```

```
class BinaryTree<T> {  
    BinaryNode<T> root;  
    int height() { return height(root); }  
    int height(BinaryNode<T> node) {  
        if(node == null) return -1;  
        return(1 + max(  
            height(node.left), height(node.right)));  
    }  
    int count() { return count(root); }  
    int count(BinaryNode<T> node) {  
        if(node == null) return 0;  
        return(1 +  
            count(node.left) + count(node.right));  
    }  
}
```



Tree Operations

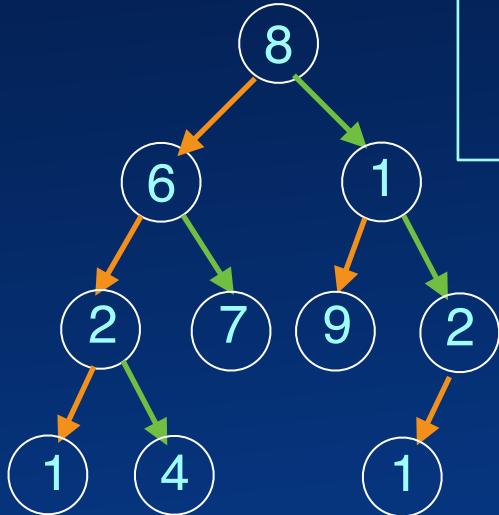
```
class BinaryNode<T> {  
    BinaryNode<T> left;  
    BinaryNode<T> right;  
    T value;  
}
```

```
class BinaryTree<T> {  
    BinaryNode<T> root;  
    int height() { return height(root); }  
    int height(BinaryNode<T> node) {  
        if(node == null) return -1;  
        return(1 + max(  
            height(node.left), height(node.right)));  
    }  
    int count() { return count(root); }  
    int count(BinaryNode<T> node) {  
        if(node == null) return 0;  
        return(1 +  
            count(node.left) + count(node.right));  
    }  
}
```



BST

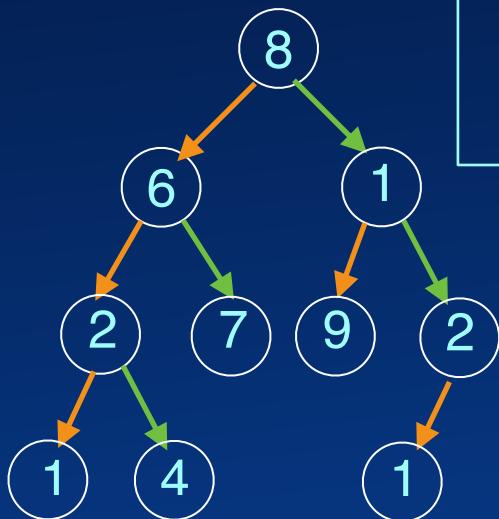
```
class BST<T> {  
    BinaryNode<T> root;  
    BinaryNode<T> find(T v) { return find(root, v); }  
    BinaryNode<T> find(BinaryNode<T> node, T v) {  
        if(node == null) return null;  
        if(node.value == v) return node;  
        BinaryNode n = find(node.left, v);  
        return (n == null)? find(node.right, v) : n;  
    }  
}
```





BST

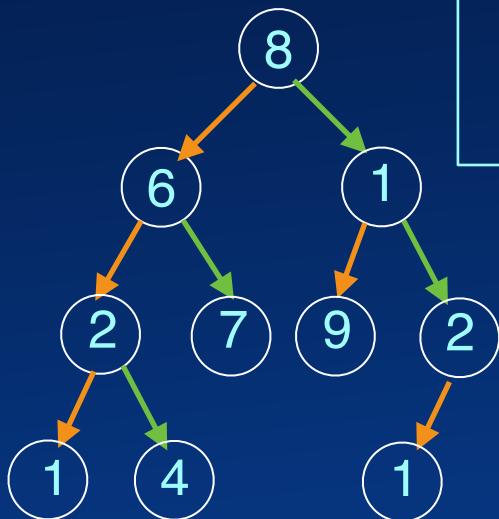
```
class BST<T> {  
    BinaryNode<T> root;  
    BinaryNode<T> find(T v) { return find(root, v); }  
    BinaryNode<T> find(BinaryNode<T> node, T v) {  
        → if(node == null) return null;  
        if(node.value == v) return node;  
        BinaryNode n = find(node.left, v);  
        return (n == null)? find(node.right, v) : n;  
    }  
}
```





BST

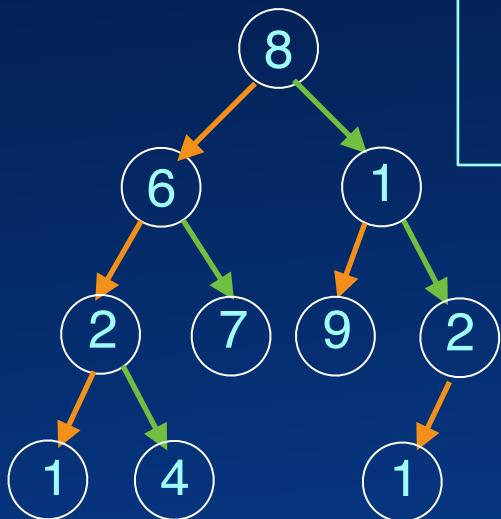
```
class BST<T> {  
    BinaryNode<T> root;  
    BinaryNode<T> find(T v) { return find(root, v); }  
    BinaryNode<T> find(BinaryNode<T> node, T v) {  
        if(node == null) return null;  
        ➔ if(node.value == v) return node;  
        BinaryNode n = find(node.left, v);  
        return (n == null)? find(node.right, v) : n;  
    }  
}
```





BST

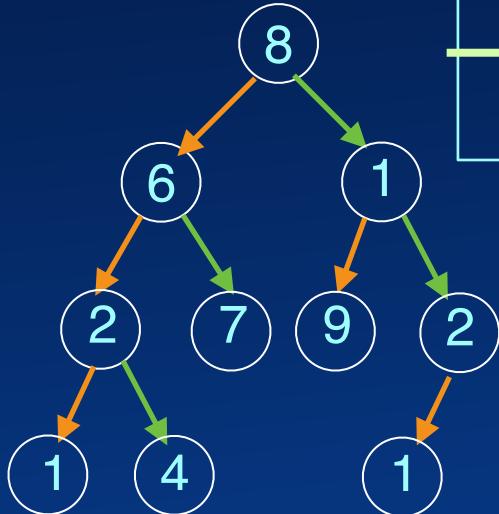
```
class BST<T> {  
    BinaryNode<T> root;  
    BinaryNode<T> find(T v) { return find(root, v); }  
    BinaryNode<T> find(BinaryNode<T> node, T v) {  
        if(node == null) return null;  
        if(node.value == v) return node;  
        → BinaryNode n = find(node.left, v);  
        return (n == null)? find(node.right, v) : n;  
    }  
}
```





BST

```
class BST<T> {  
    BinaryNode<T> root;  
    BinaryNode<T> find(T v) { return find(root, v); }  
    BinaryNode<T> find(BinaryNode<T> node, T v) {  
        if(node == null) return null;  
        if(node.value == v) return node;  
        BinaryNode n = find(node.left, v);  
        return (n == null)? find(node.right, v) : n;  
    }  
}
```

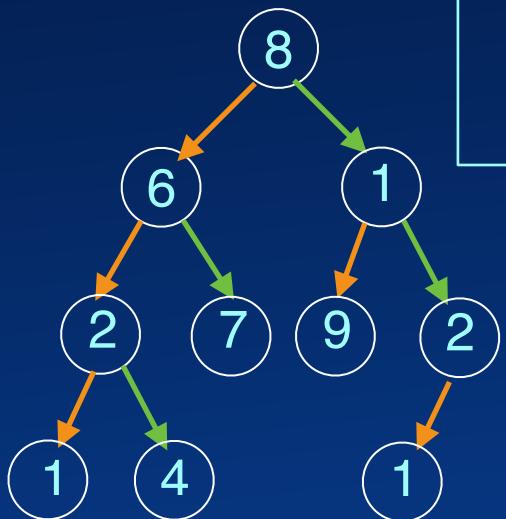




BST

```
class BST<T> {  
    BinaryNode<T> root;  
    BinaryNode<T> find(T v) { return find(root, v); }  
    BinaryNode<T> find(BinaryNode<T> node, T v) {  
        if(node == null) return null;  
        if(node.value == v) return node;
```

```
class BST<T extends Comparable<T>> {  
    BinaryNode<T> root;  
    BinaryNode<T> find(T v) { return find(root, v); }  
    BinaryNode<T> find(BinaryNode<T> node, T v) {  
        if(node == null) return null;  
        int compared = v.compareTo(node.value);  
        if(compared == 0) return node;  
        if(compared < 0) return find(node.left, v);  
        return find(node.right, v);  
    }
```

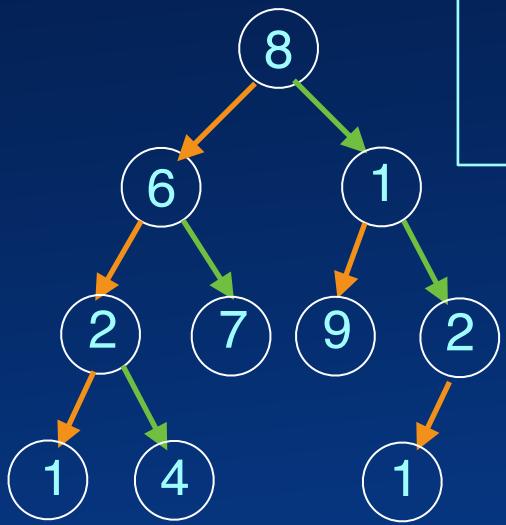




BST

```
class BST<T> {  
    BinaryNode<T> root;  
    BinaryNode<T> find(T v) { return find(root, v); }  
    BinaryNode<T> find(BinaryNode<T> node, T v) {  
        if(node == null) return null;  
        if(node.value == v) return node;
```

```
class BST<T extends Comparable<T>> {  
    BinaryNode<T> root;  
    BinaryNode<T> find(T v) { return find(root, v); }  
    BinaryNode<T> find(BinaryNode<T> node, T v) {  
        if(node == null) return null;  
        int compared = v.compareTo(node.value);  
        if(compared == 0) return node;  
        if(compared < 0) return find(node.left, v);  
        return find(node.right, v);  
    }
```

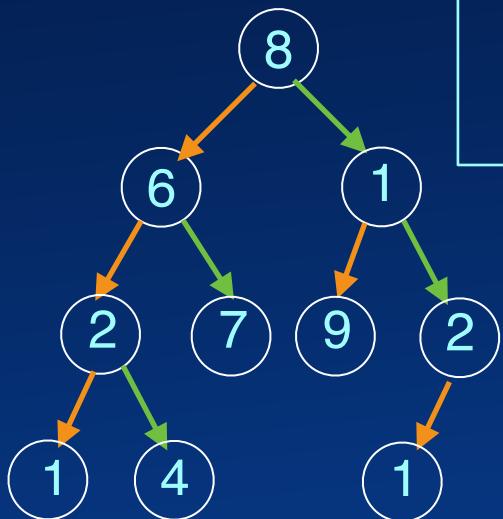




BST

```
class BST<T> {  
    BinaryNode<T> root;  
    BinaryNode<T> find(T v) { return find(root, v); }  
    BinaryNode<T> find(BinaryNode<T> node, T v) {  
        if(node == null) return null;  
        if(node.value == v) return node;
```

```
class BST<T extends Comparable<T>> {  
    BinaryNode<T> root;  
    BinaryNode<T> find(T v) { return find(root, v); }  
    BinaryNode<T> find(BinaryNode<T> node, T v) {  
        if(node == null) return null;  
        int compared = v.compareTo(node.value);  
        if(compared == 0) return node;  
        if(compared < 0) return find(node.left, v);  
        return find(node.right, v);  
    }
```

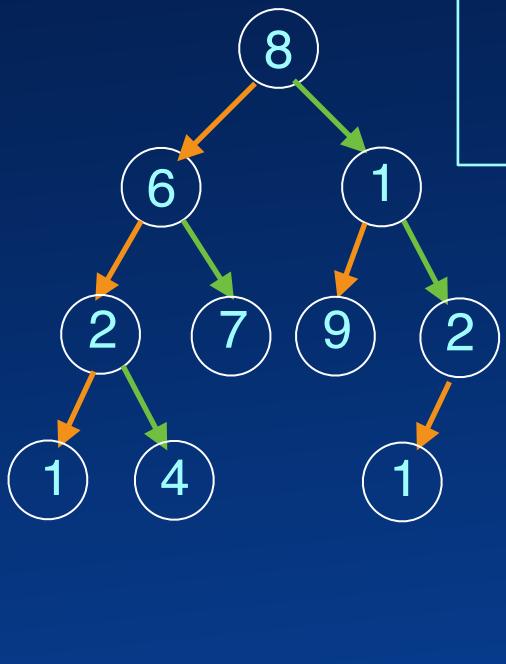




BST

```
class BST<T> {  
    BinaryNode<T> root;  
    BinaryNode<T> find(T v) { return find(root, v); }  
    BinaryNode<T> find(BinaryNode<T> node, T v) {  
        if(node == null) return null;  
        if(node.value == v) return node;
```

```
class BST<T extends Comparable<T>> {  
    BinaryNode<T> root;  
    BinaryNode<T> find(T v) { return find(root, v); }  
    BinaryNode<T> find(BinaryNode<T> node, T v) {  
        if(node == null) return null;  
        int compared = v.compareTo(node.value);  
        → if(compared == 0) return node;  
        if(compared < 0) return find(node.left, v);  
        return find(node.right, v);  
    }
```

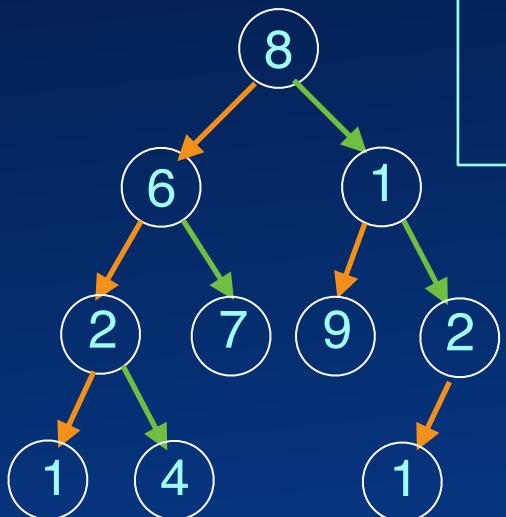




BST

```
class BST<T> {  
    BinaryNode<T> root;  
    BinaryNode<T> find(T v) { return find(root, v); }  
    BinaryNode<T> find(BinaryNode<T> node, T v) {  
        if(node == null) return null;  
        if(node.value == v) return node;
```

```
class BST<T extends Comparable<T>> {  
    BinaryNode<T> root;  
    BinaryNode<T> find(T v) { return find(root, v); }  
    BinaryNode<T> find(BinaryNode<T> node, T v) {  
        if(node == null) return null;  
        int compared = v.compareTo(node.value);  
        if(compared == 0) return node;  
        → if(compared < 0) return find(node.left, v);  
        return find(node.right, v);  
    }
```

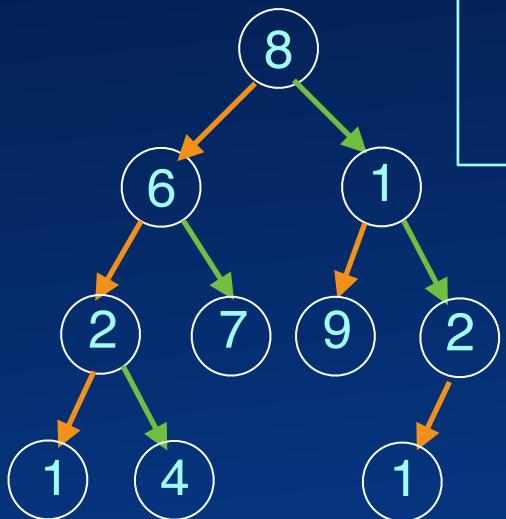




BST

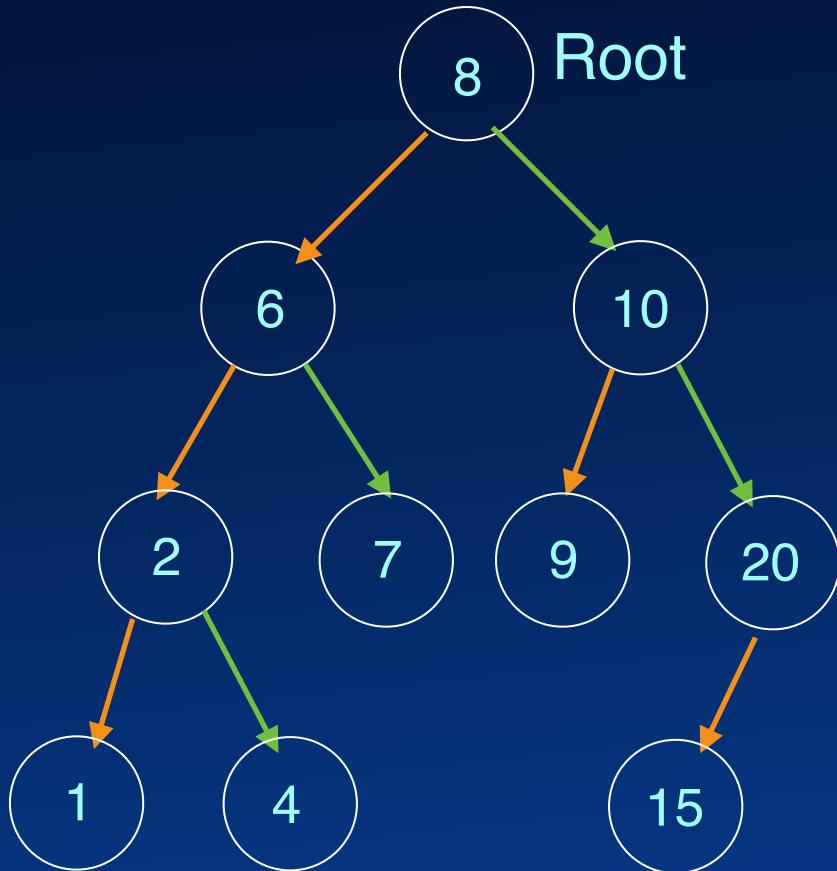
```
class BST<T> {  
    BinaryNode<T> root;  
    BinaryNode<T> find(T v) { return find(root, v); }  
    BinaryNode<T> find(BinaryNode<T> node, T v) {  
        if(node == null) return null;  
        if(node.value == v) return node;
```

```
class BST<T extends Comparable<T>> {  
    BinaryNode<T> root;  
    BinaryNode<T> find(T v) { return find(root, v); }  
    BinaryNode<T> find(BinaryNode<T> node, T v) {  
        if(node == null) return null;  
        int compared = v.compareTo(node.value);  
        if(compared == 0) return node;  
        if(compared < 0) return find(node.left, v);  
        →return find(node.right, v);  
    }
```





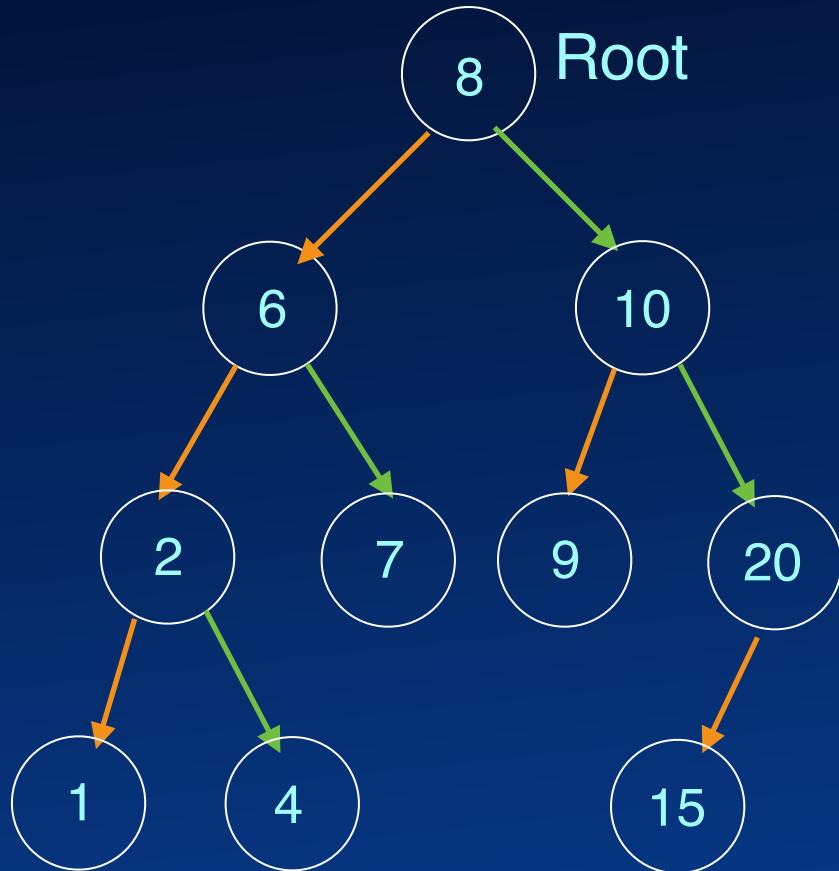
BST Operations





BST Operations

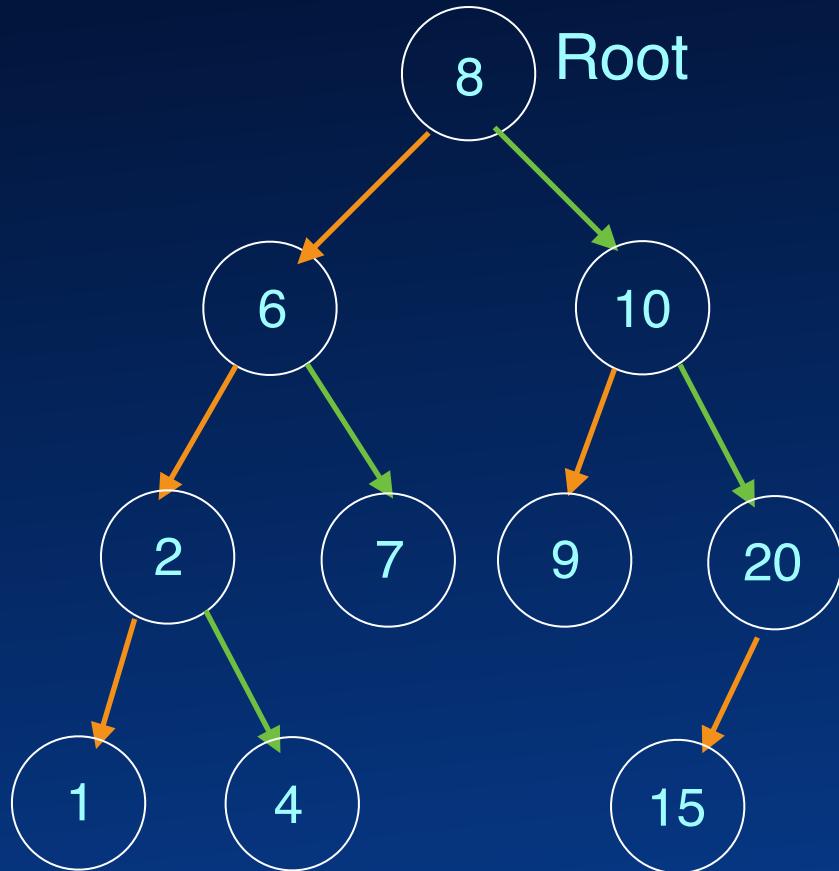
Insert 5





BST Operations

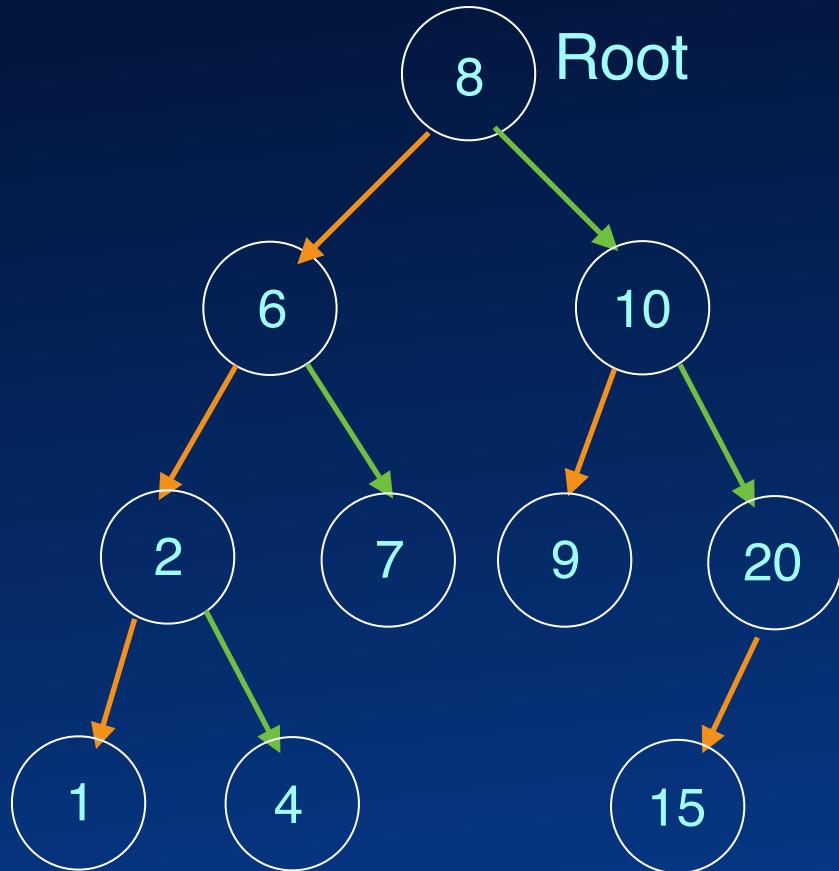
Insert 5





BST Operations

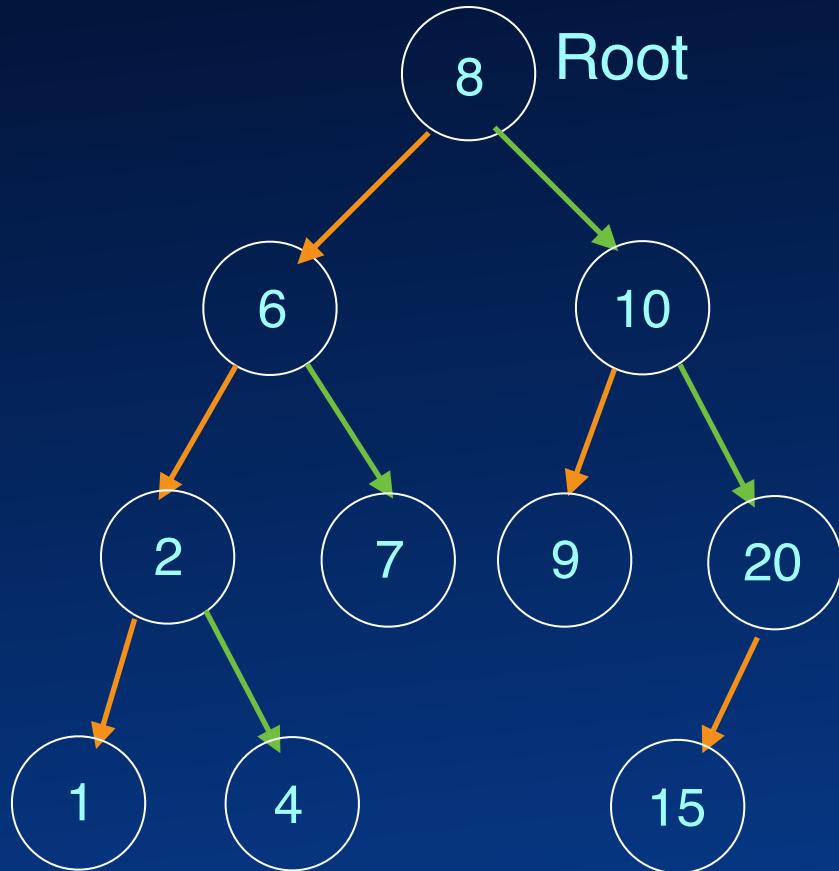
Insert 5





BST Operations

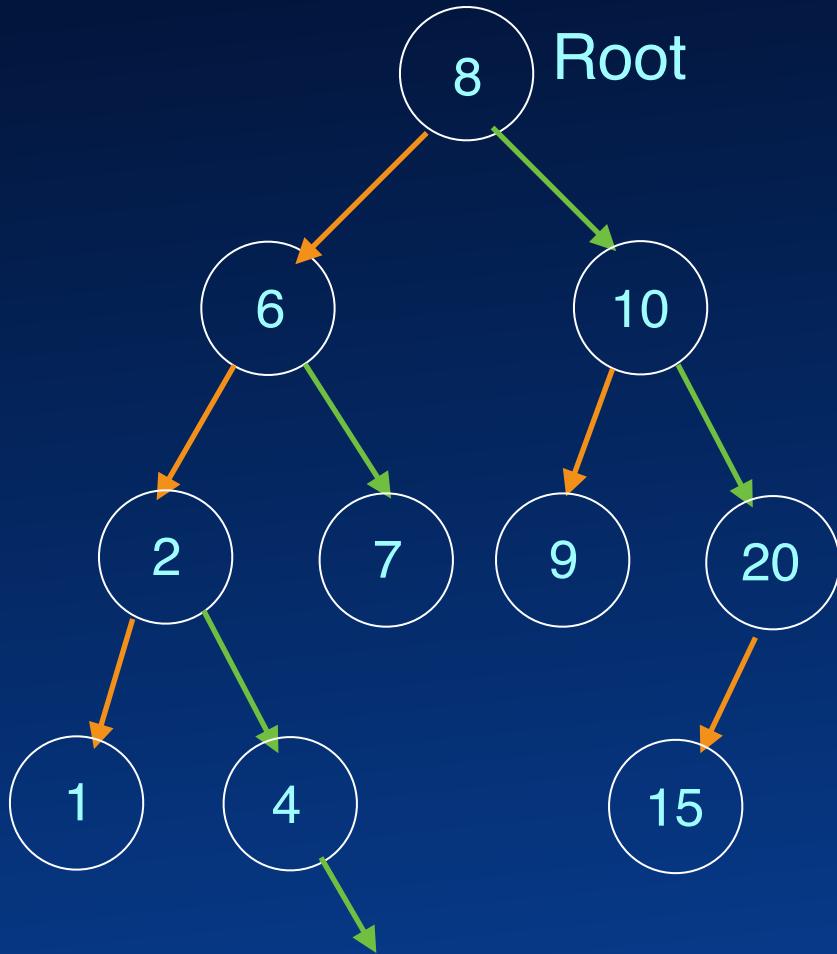
Insert 5





BST Operations

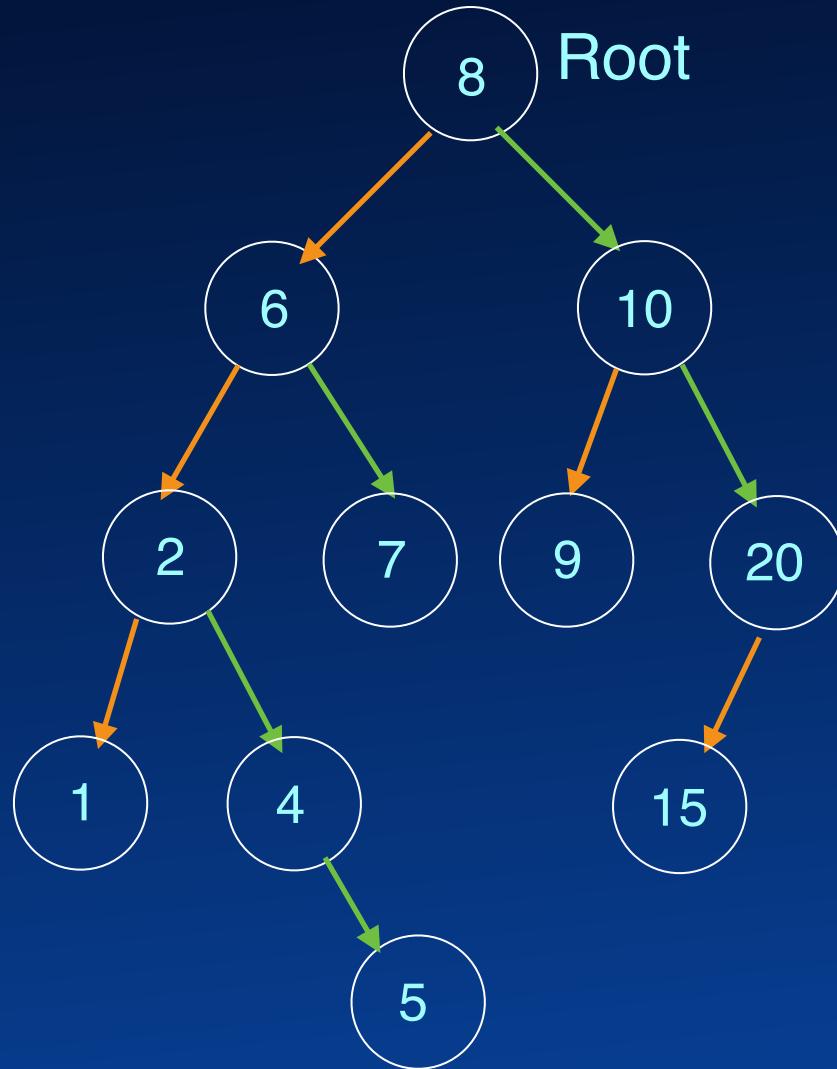
Insert 5





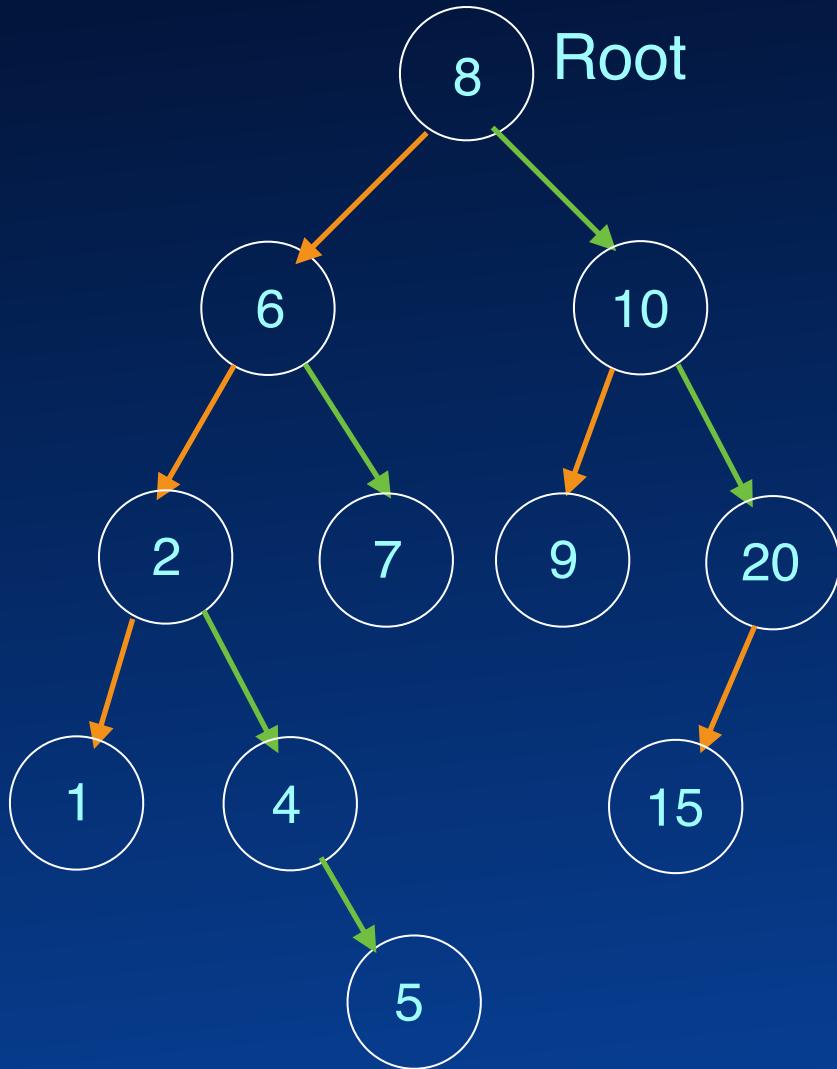
BST Operations

Insert 5



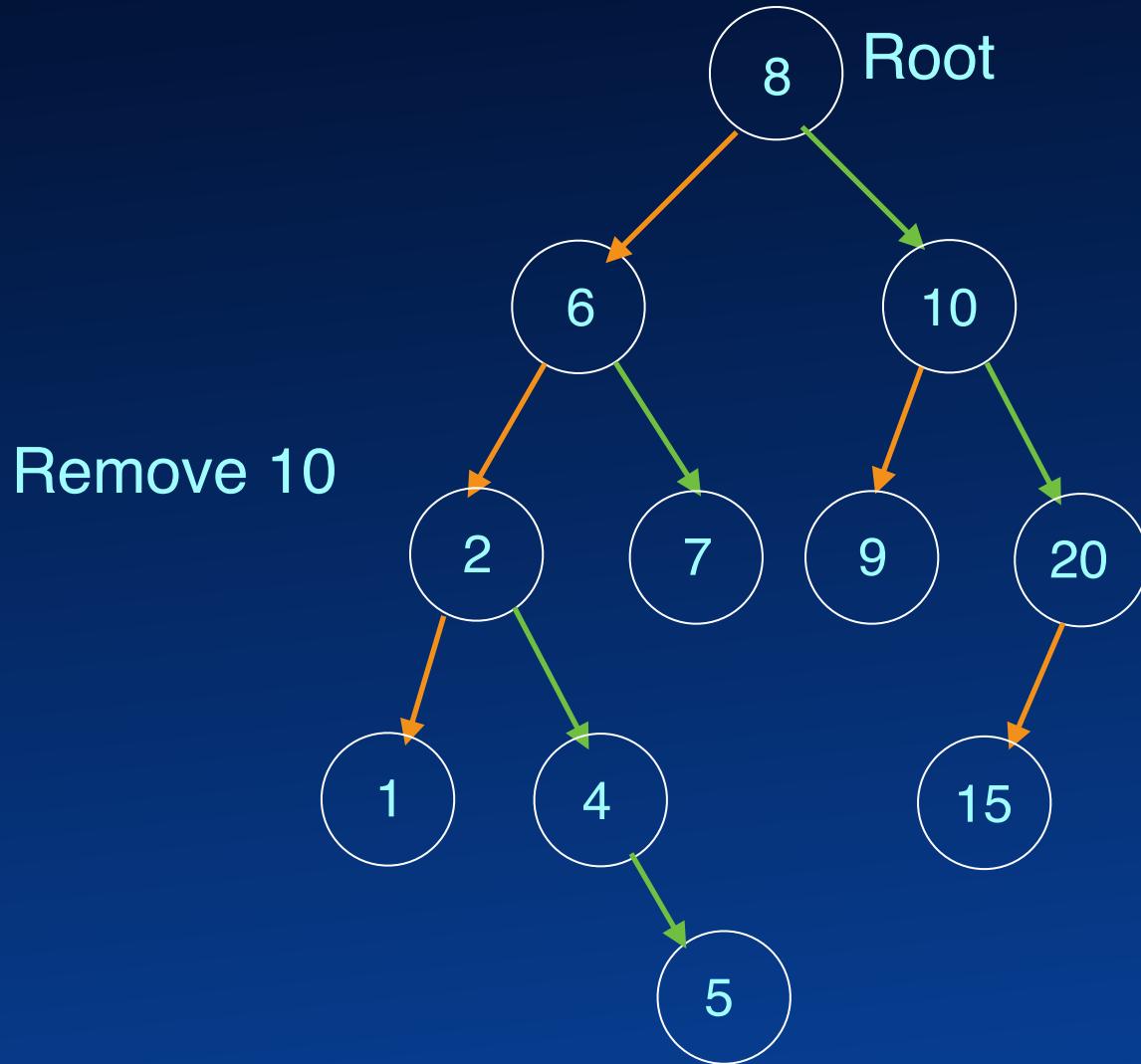


BST Operations



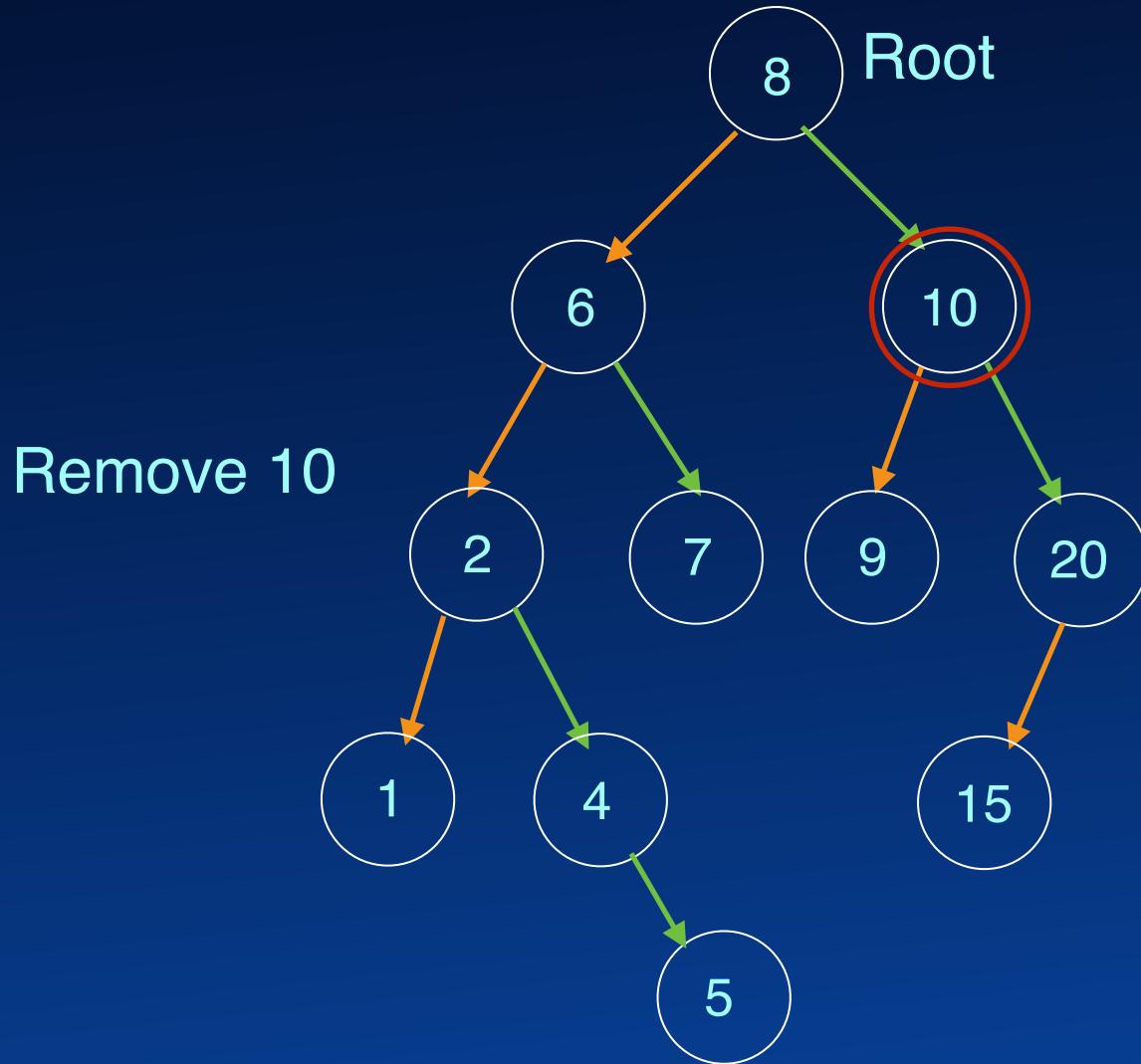


BST Operations



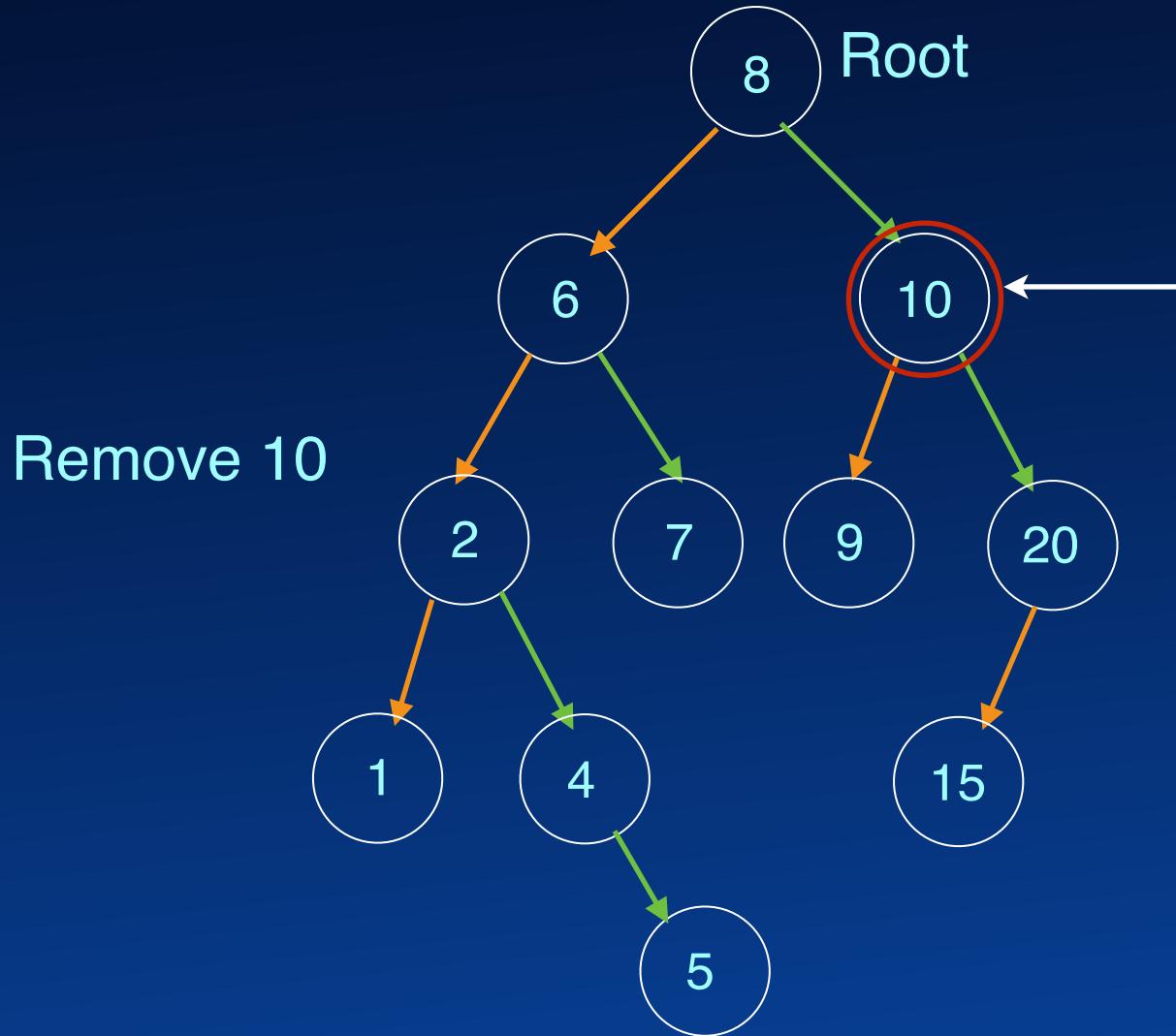


BST Operations



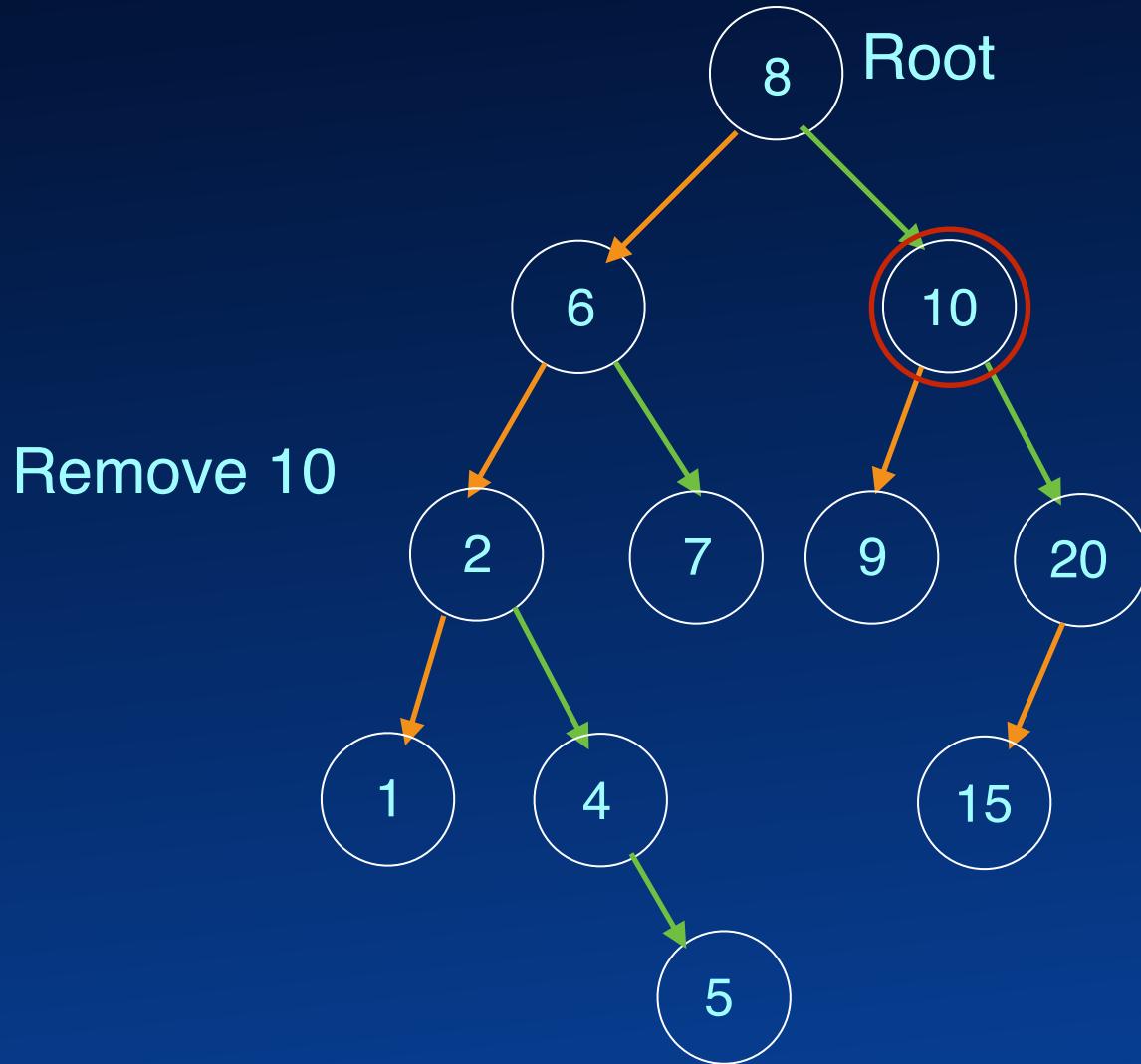


BST Operations



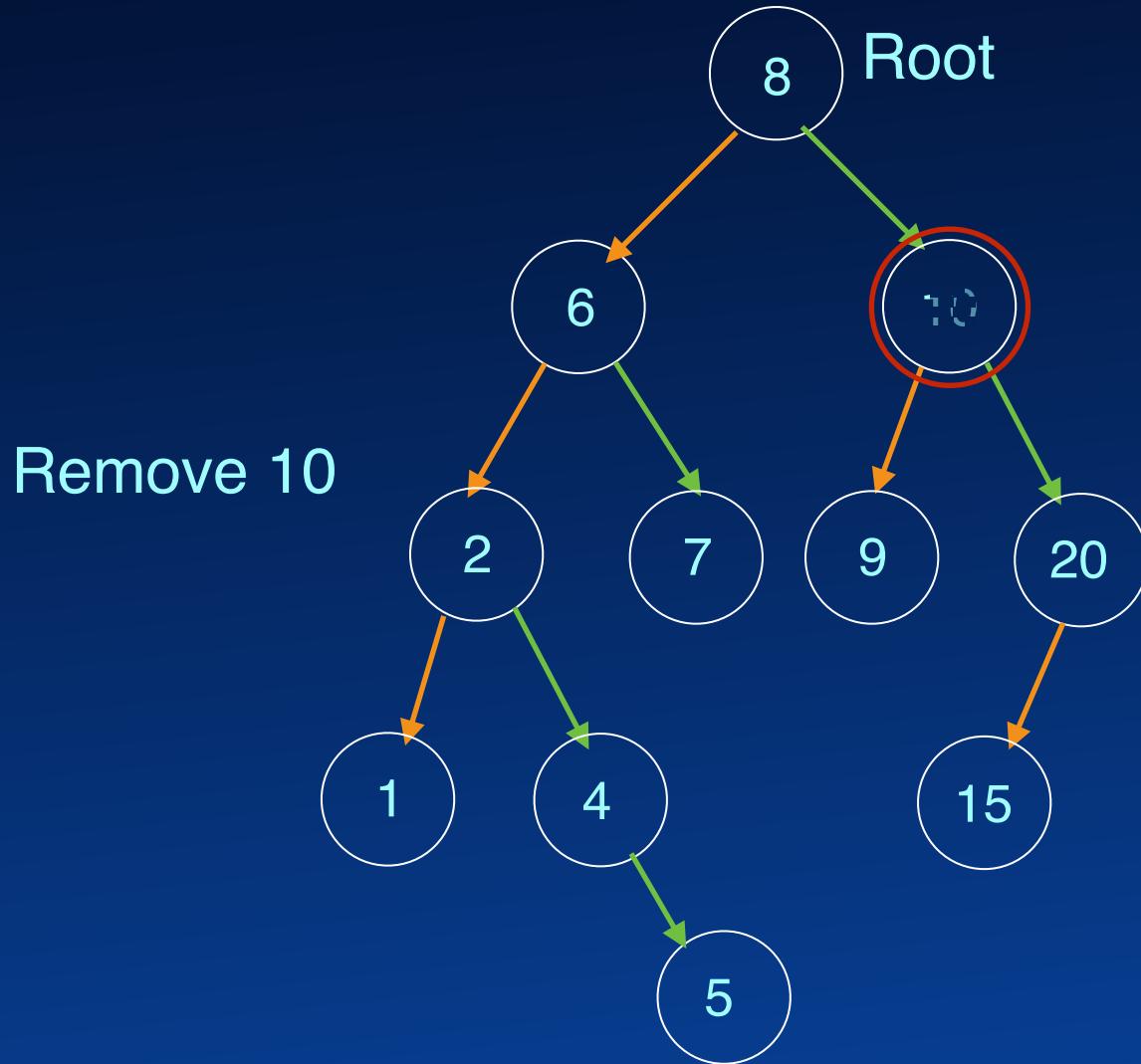


BST Operations



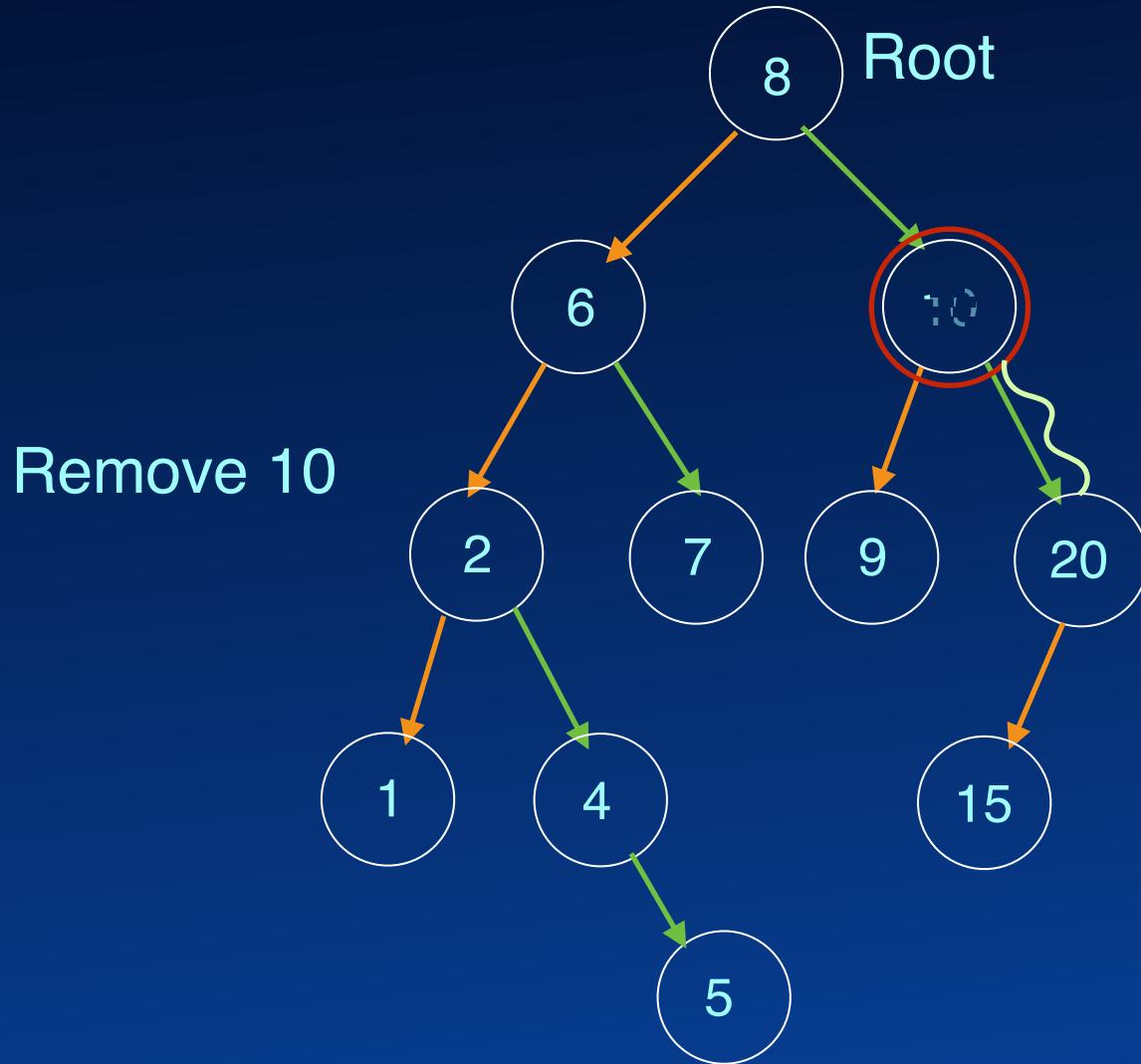


BST Operations



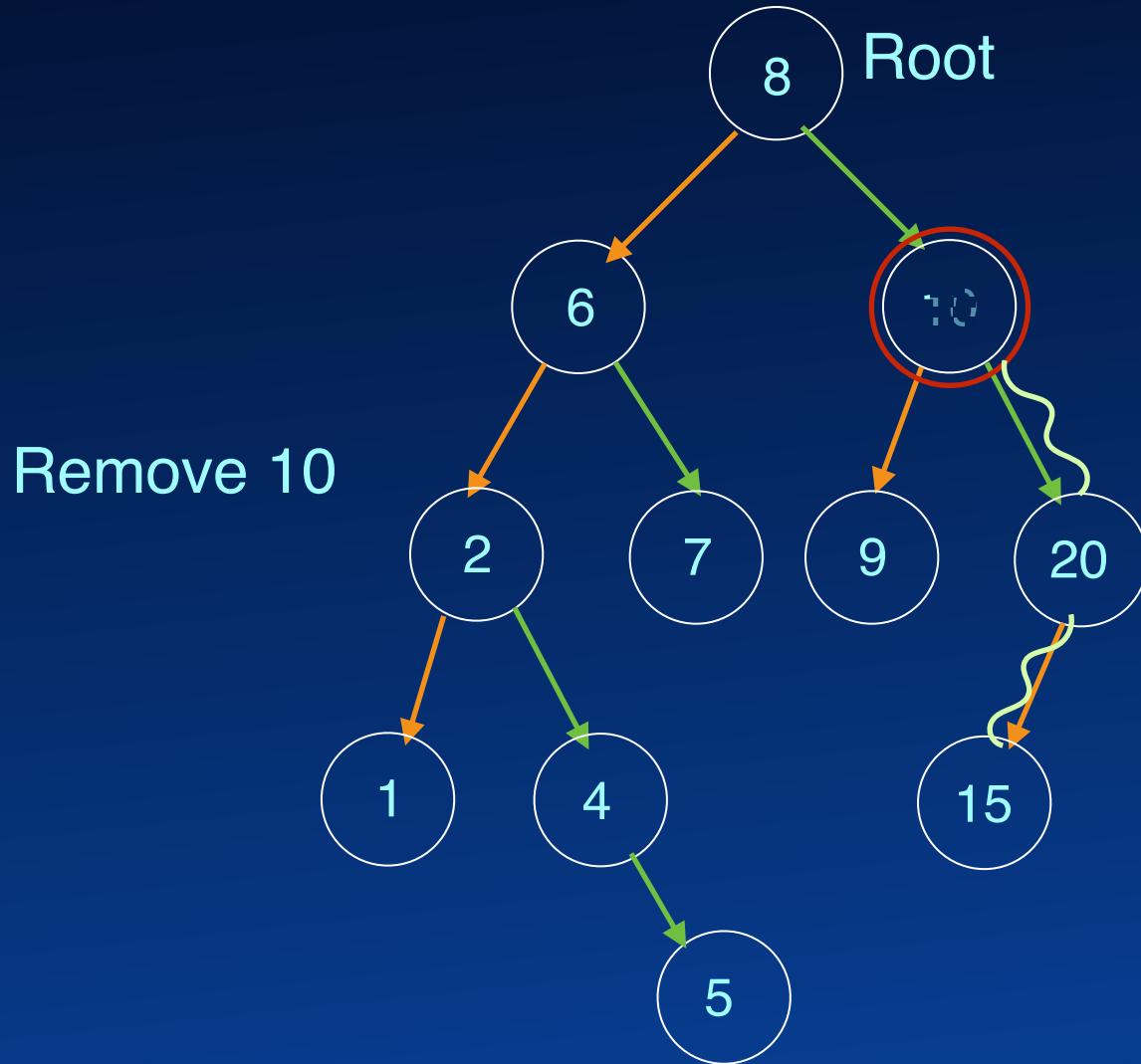


BST Operations



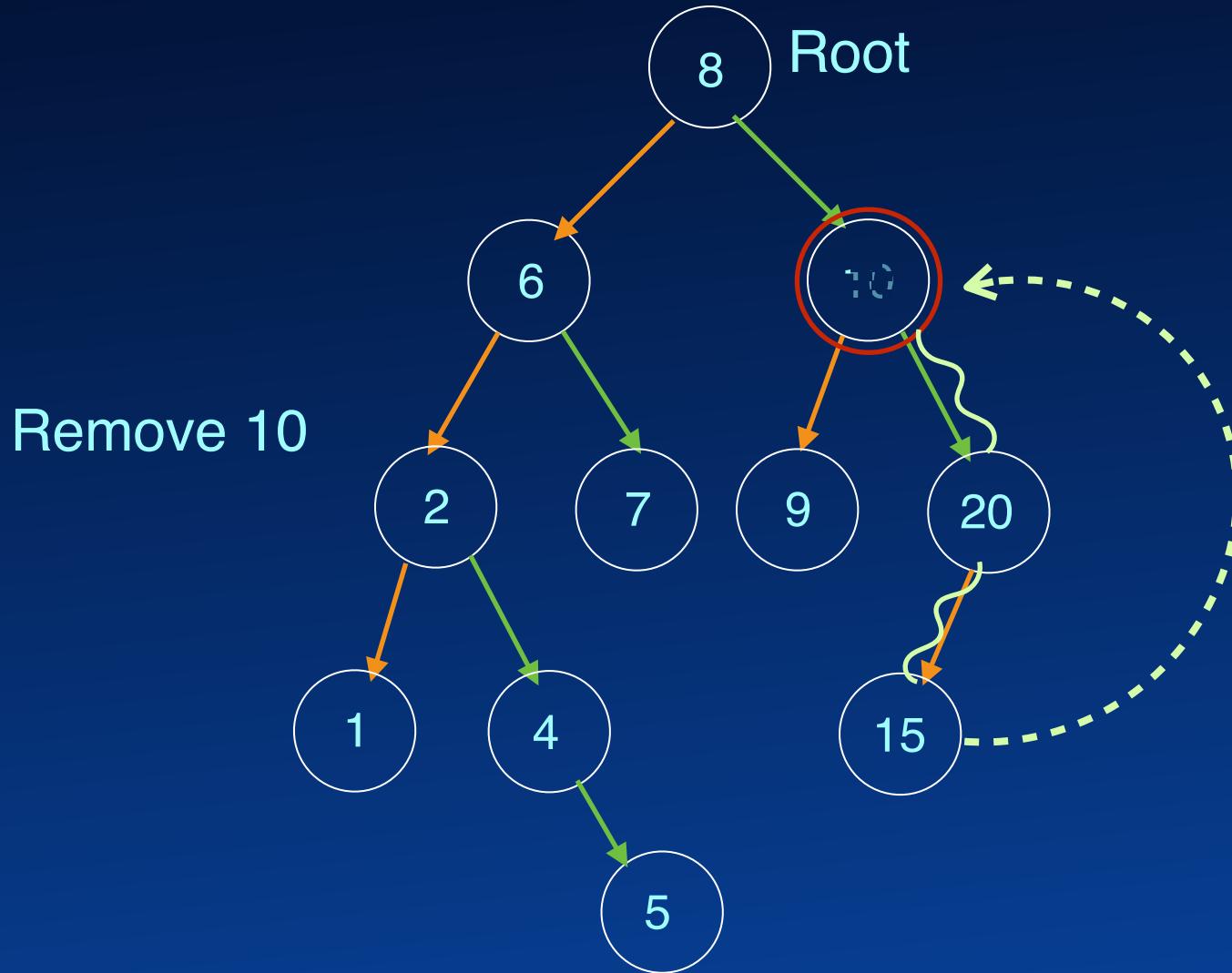


BST Operations



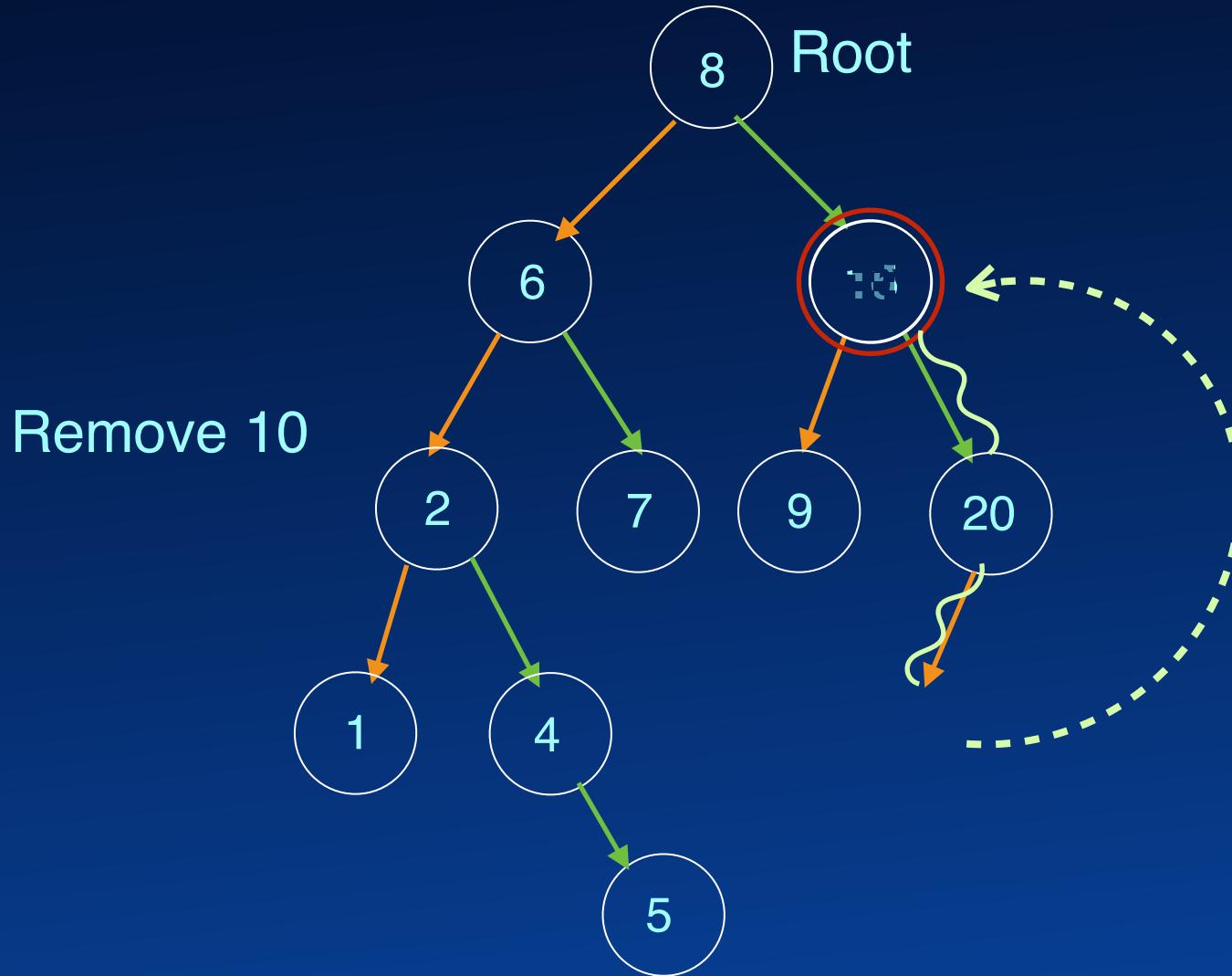


BST Operations



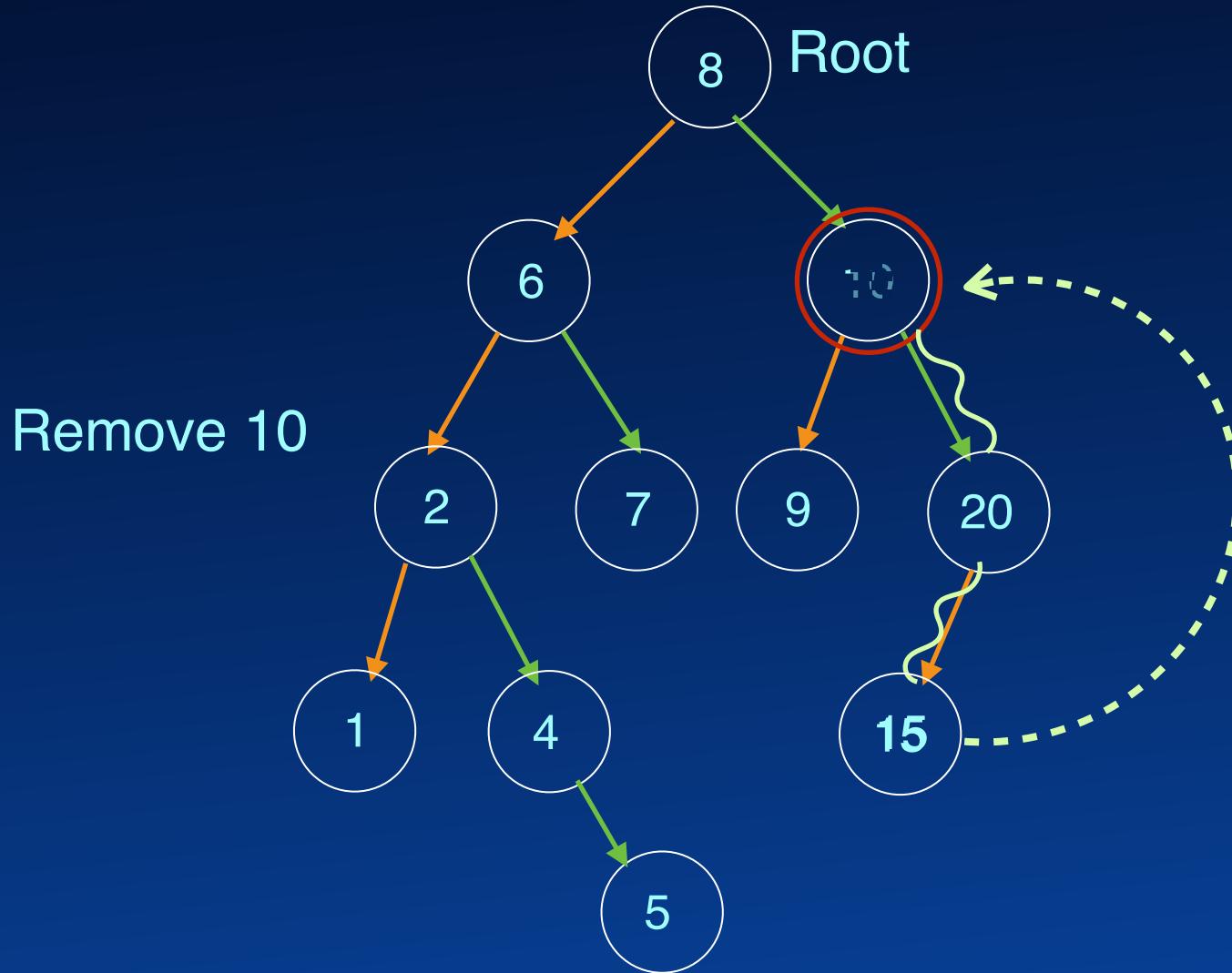


BST Operations



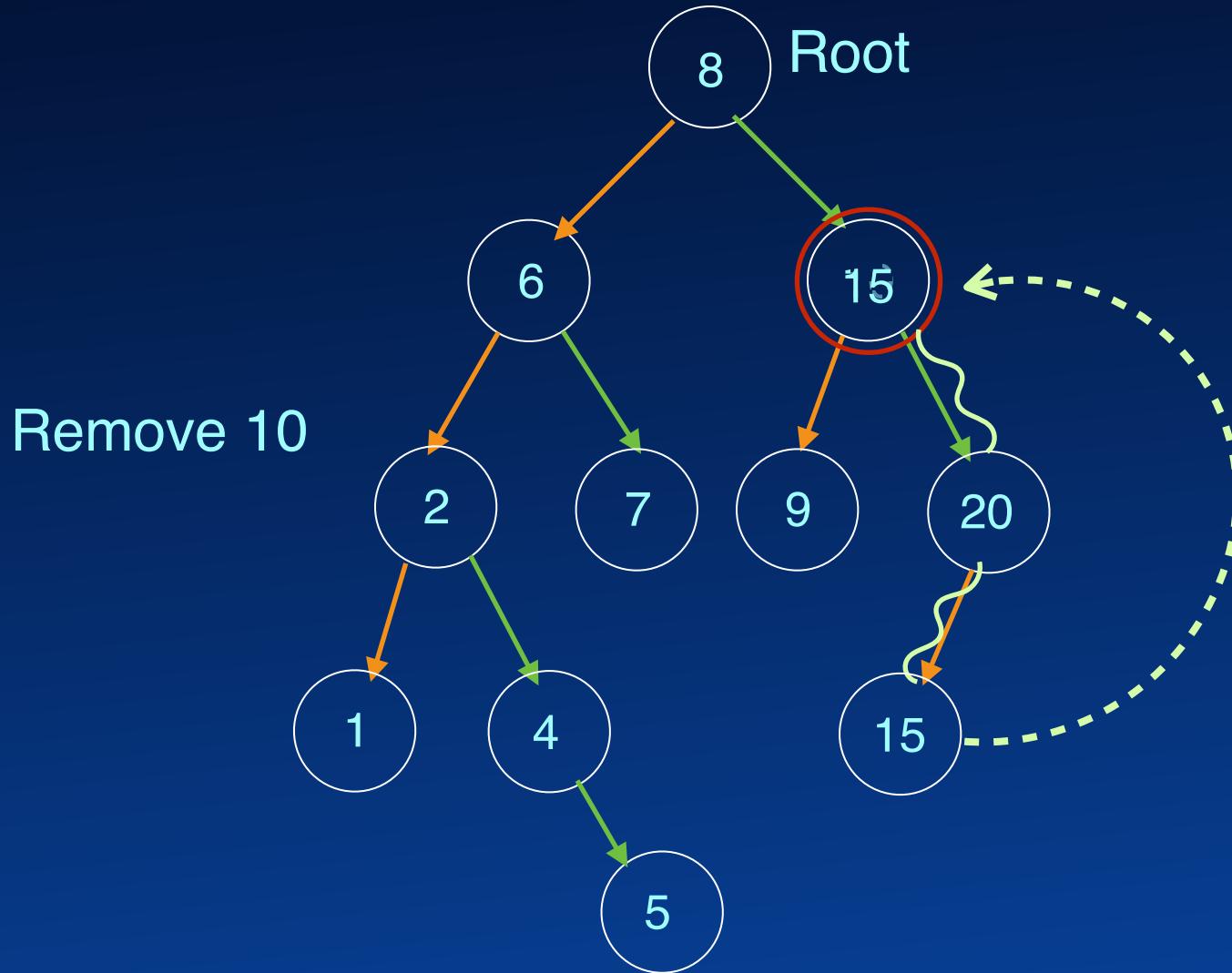


BST Operations



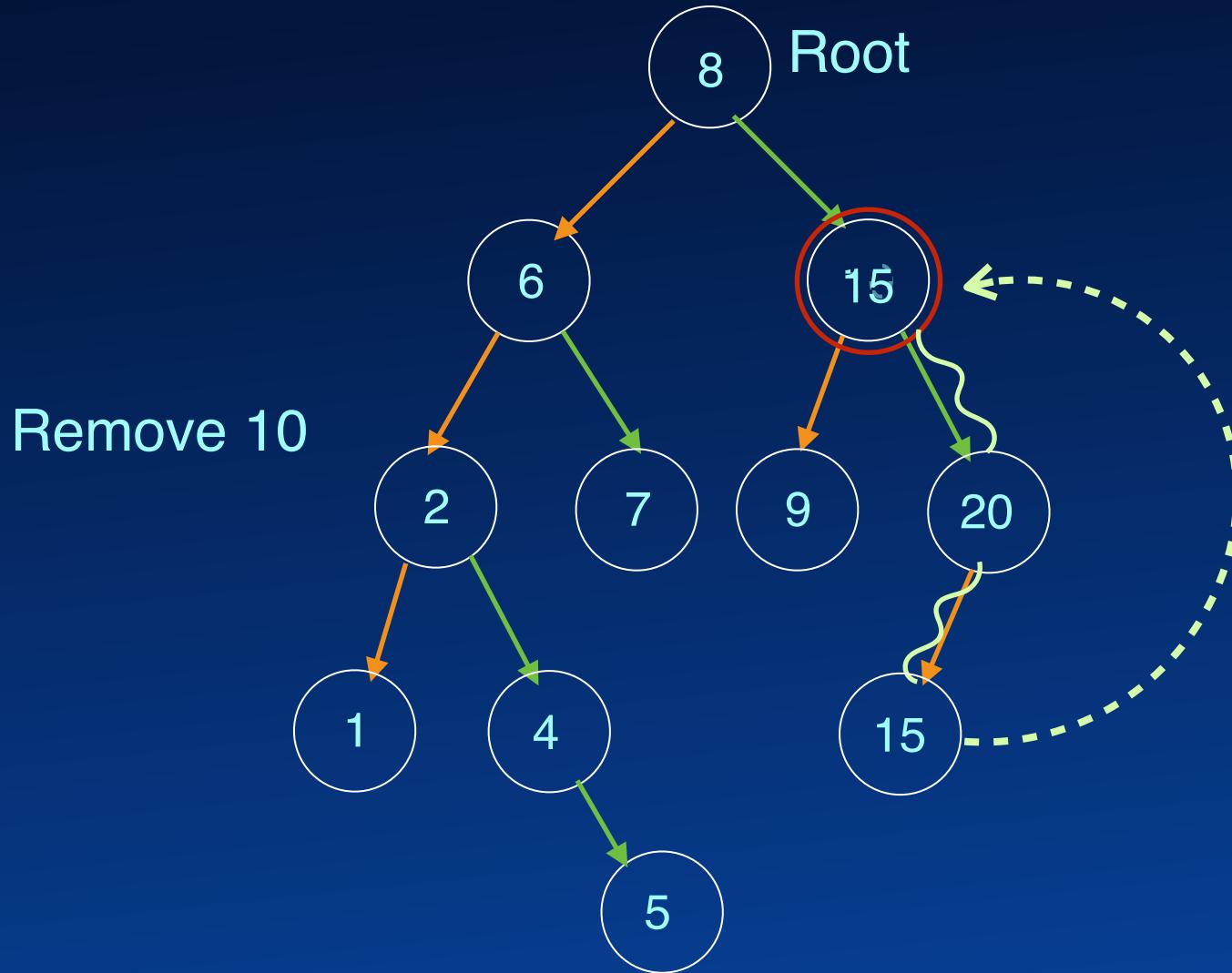


BST Operations



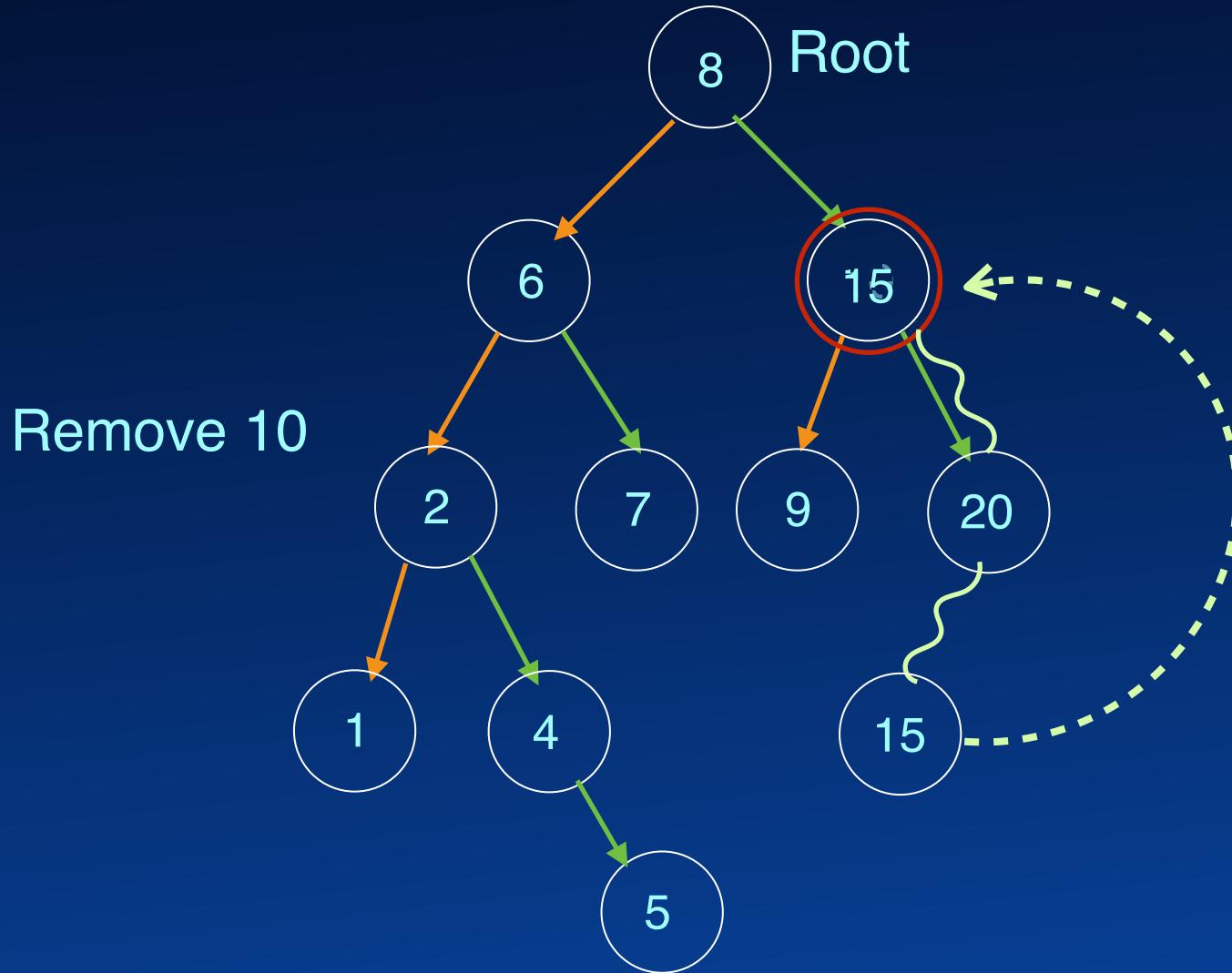


BST Operations



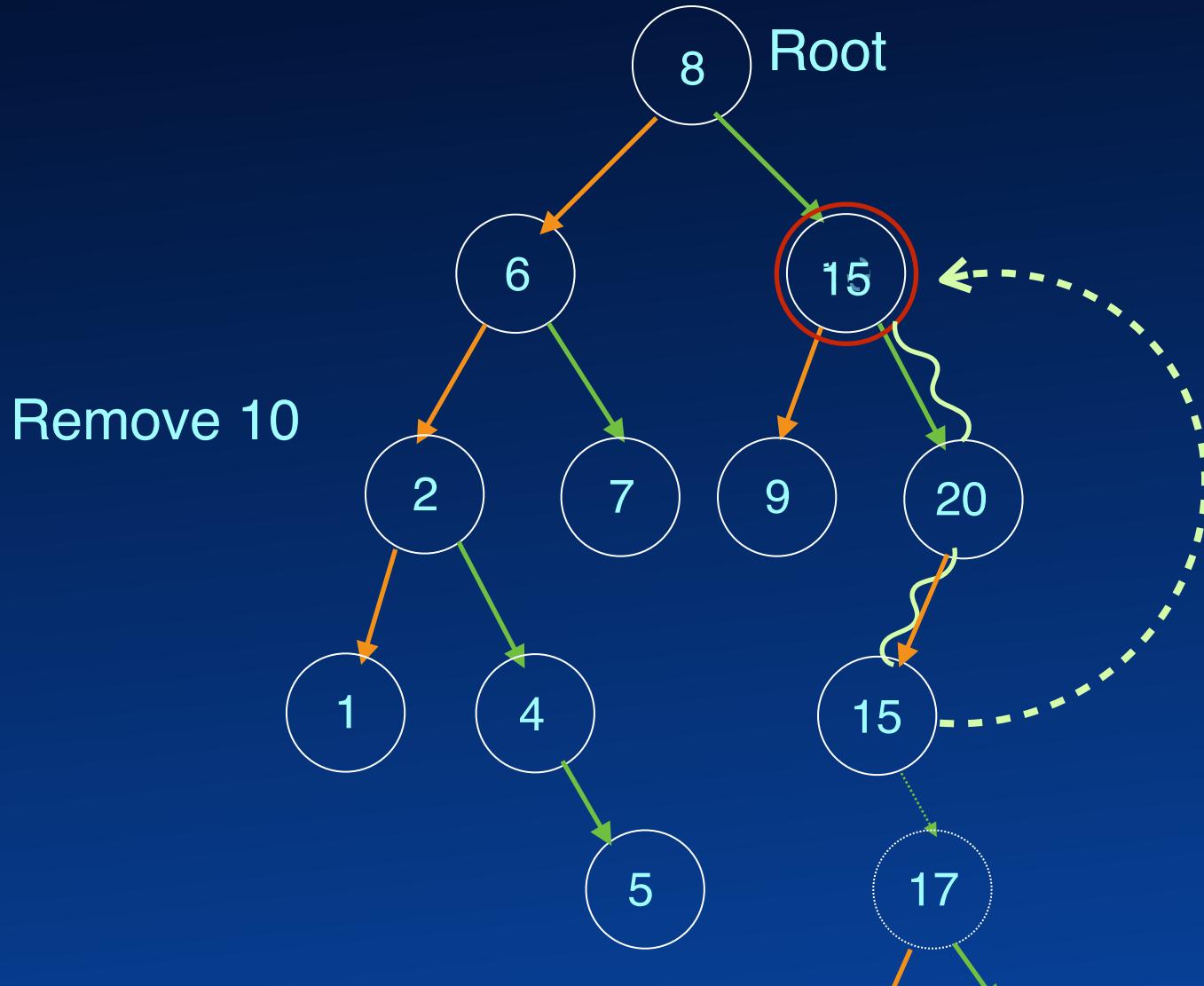


BST Operations



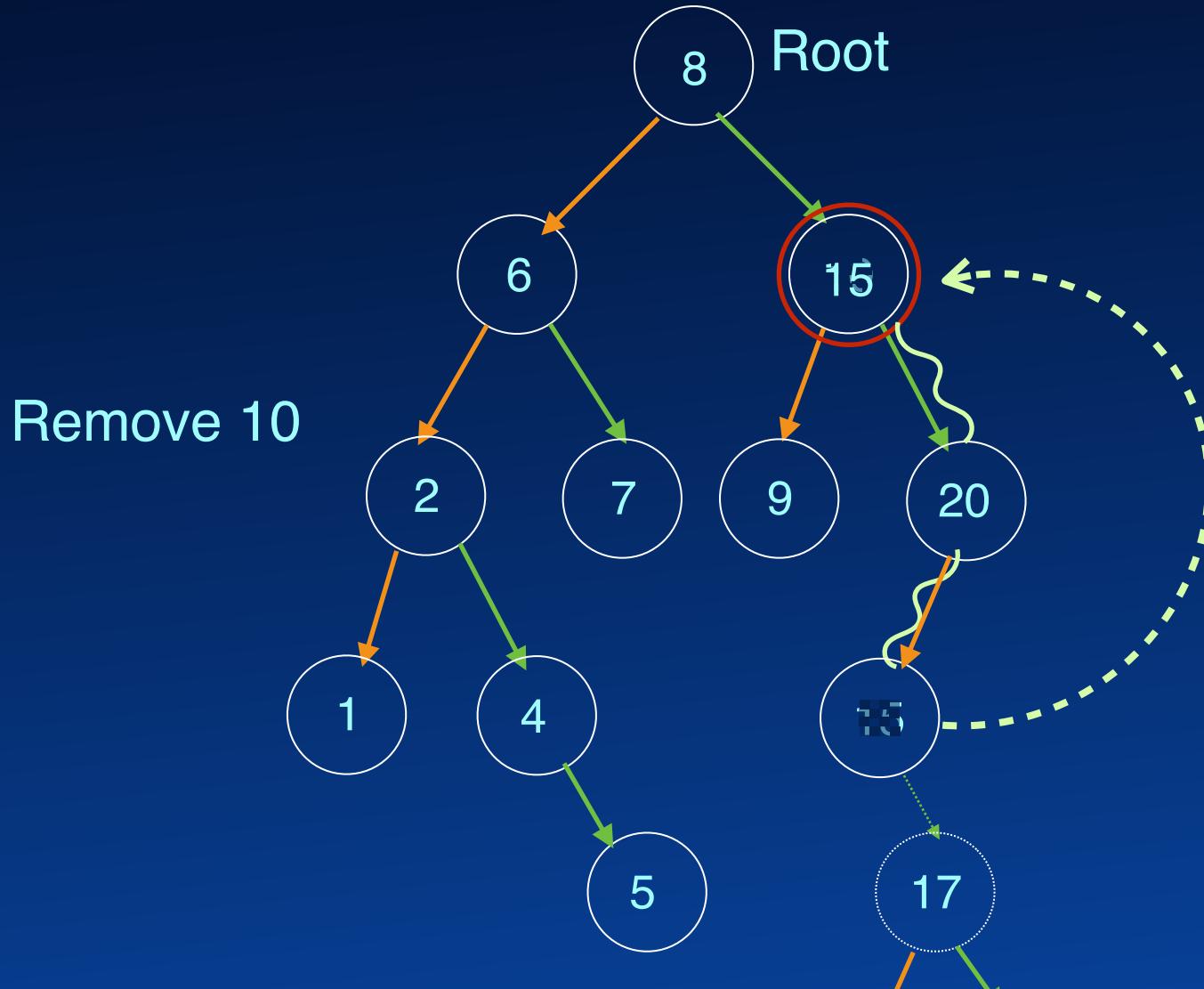


BST Operations



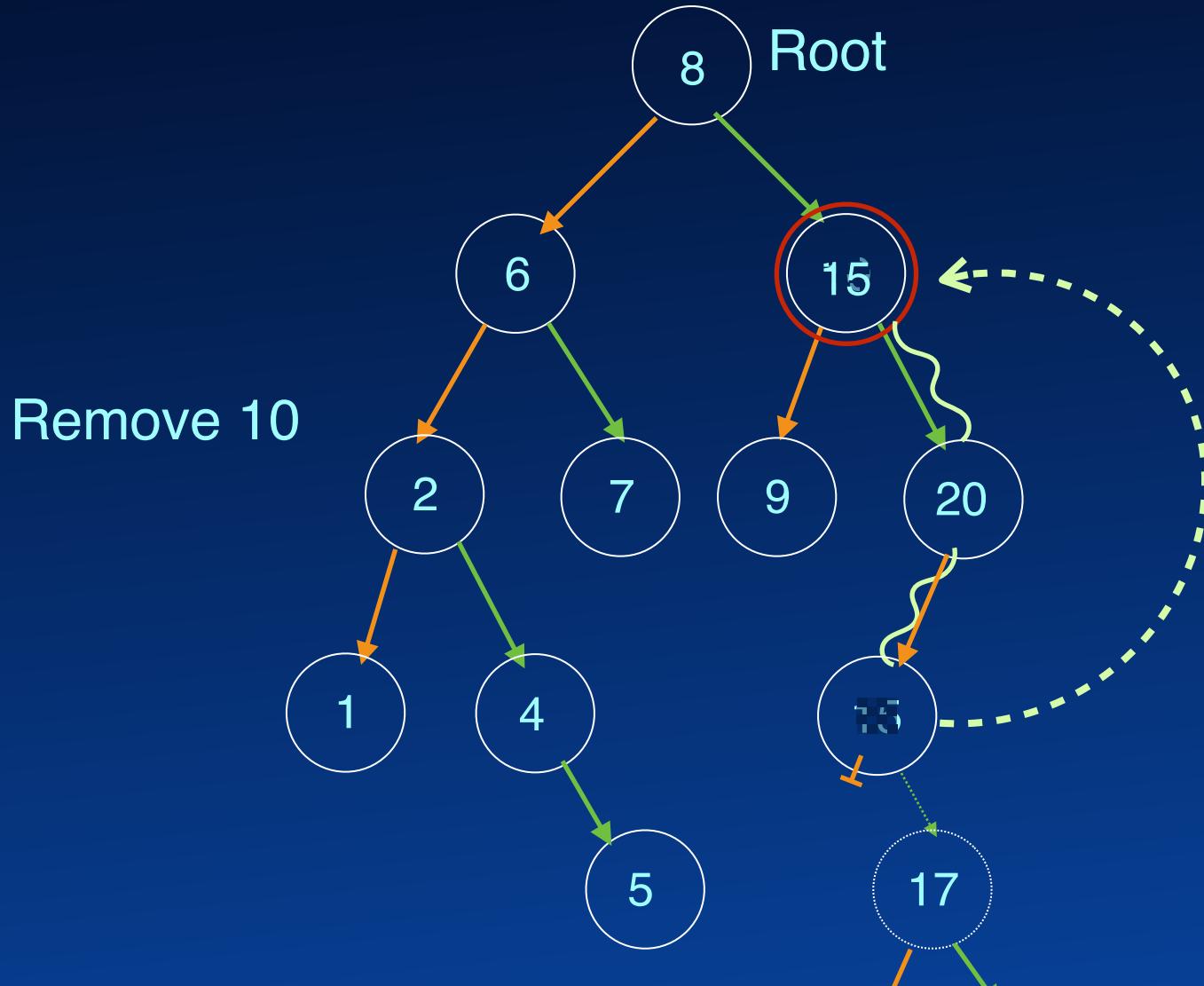


BST Operations





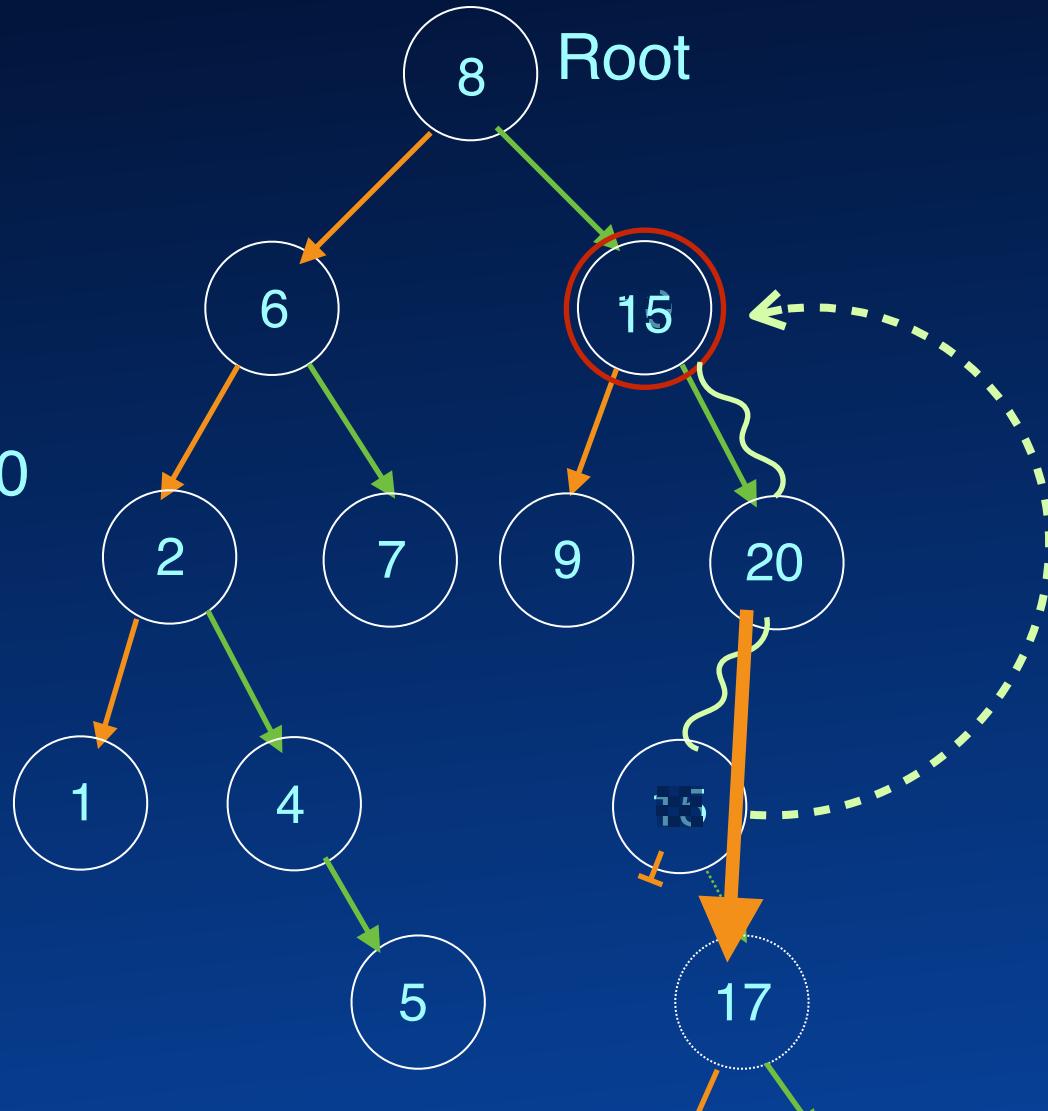
BST Operations



BST Operations

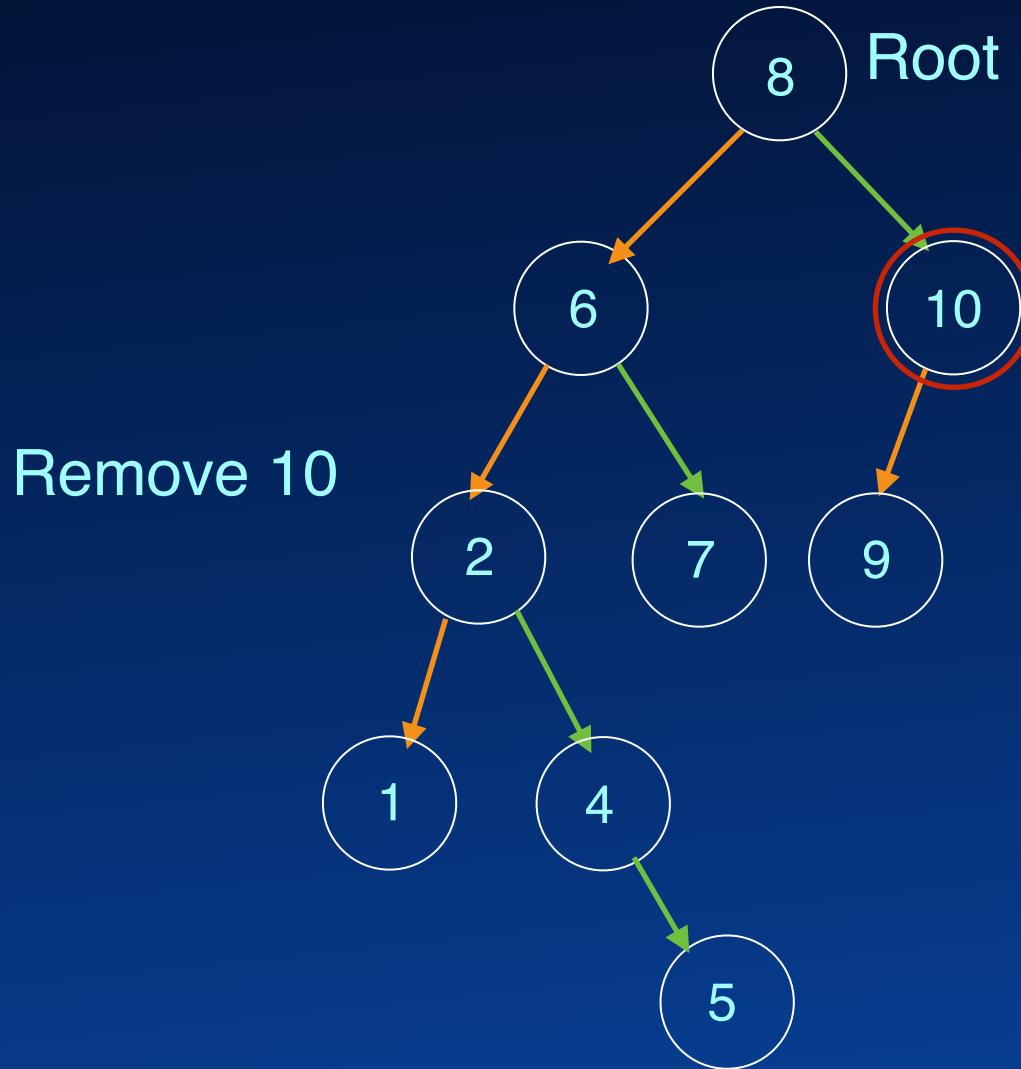


Remove 10



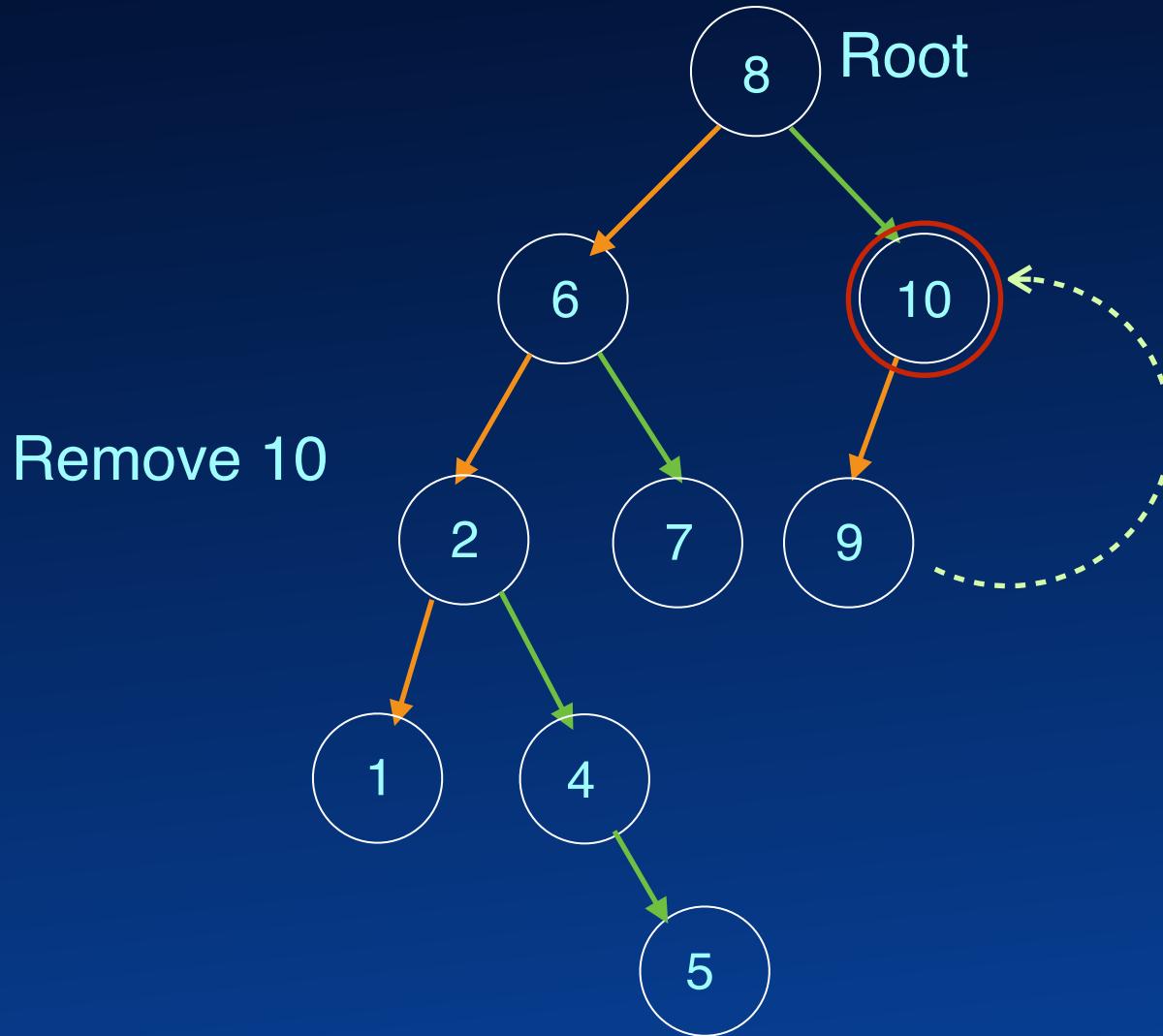


BST Operations



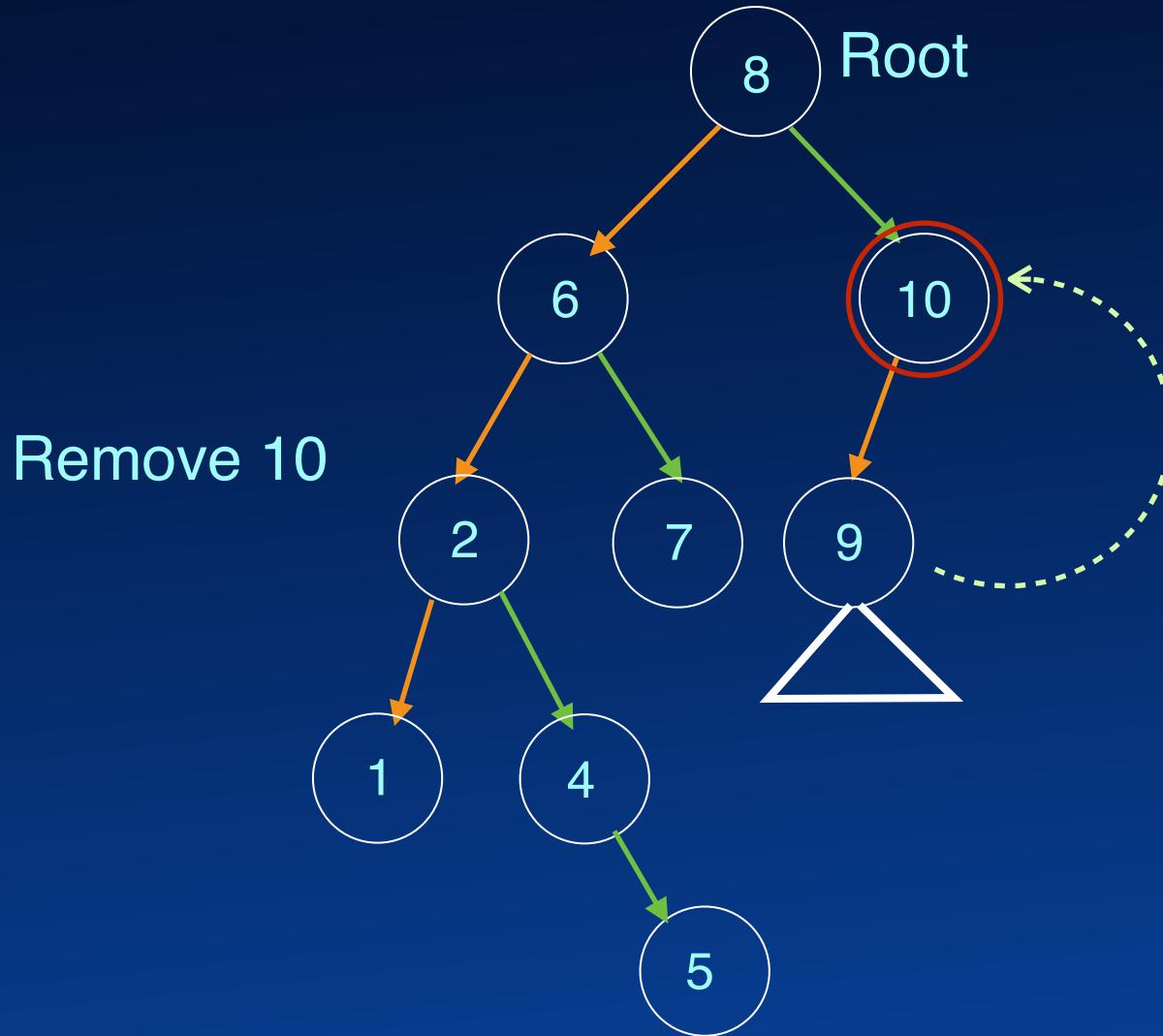


BST Operations



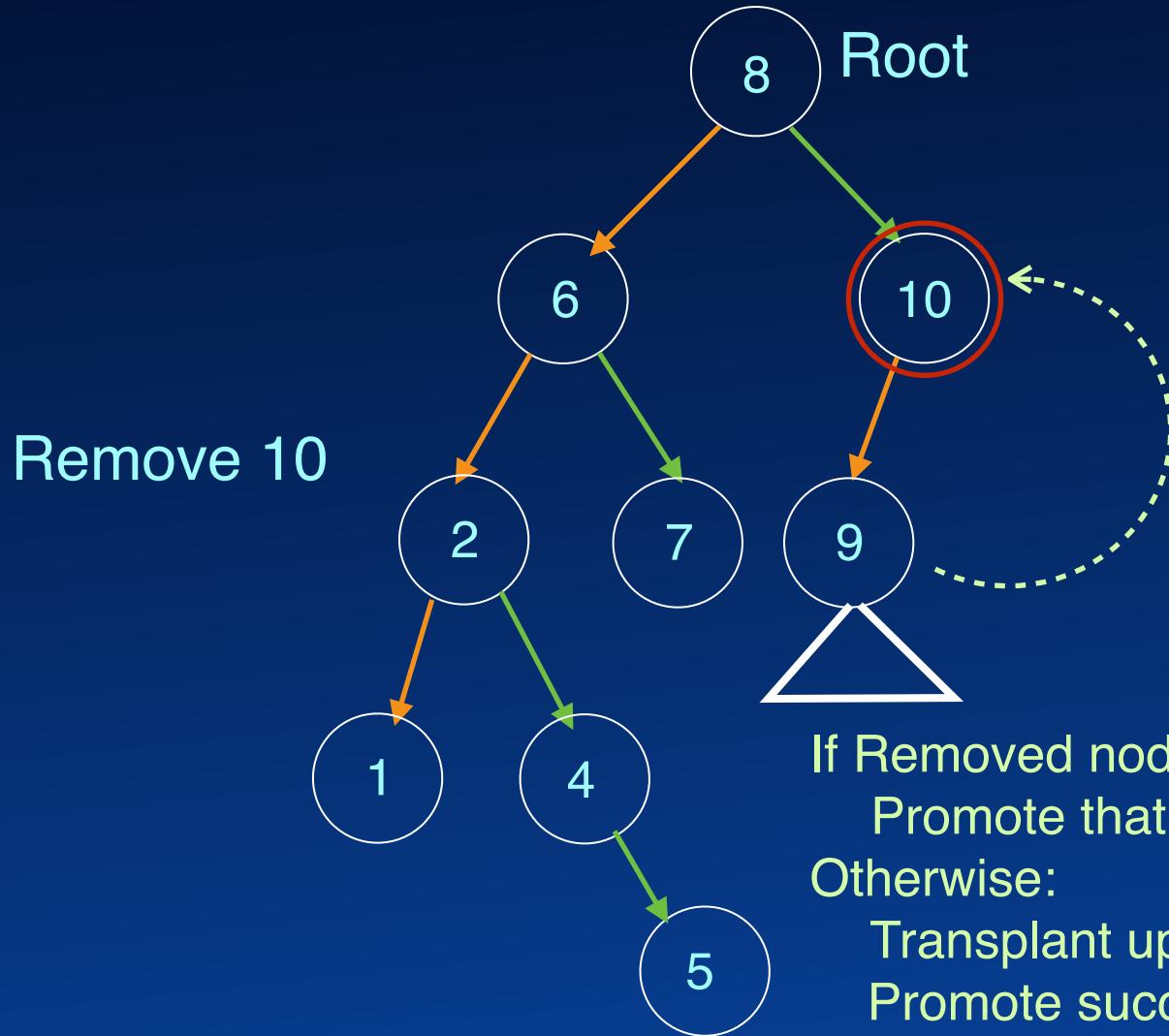


BST Operations



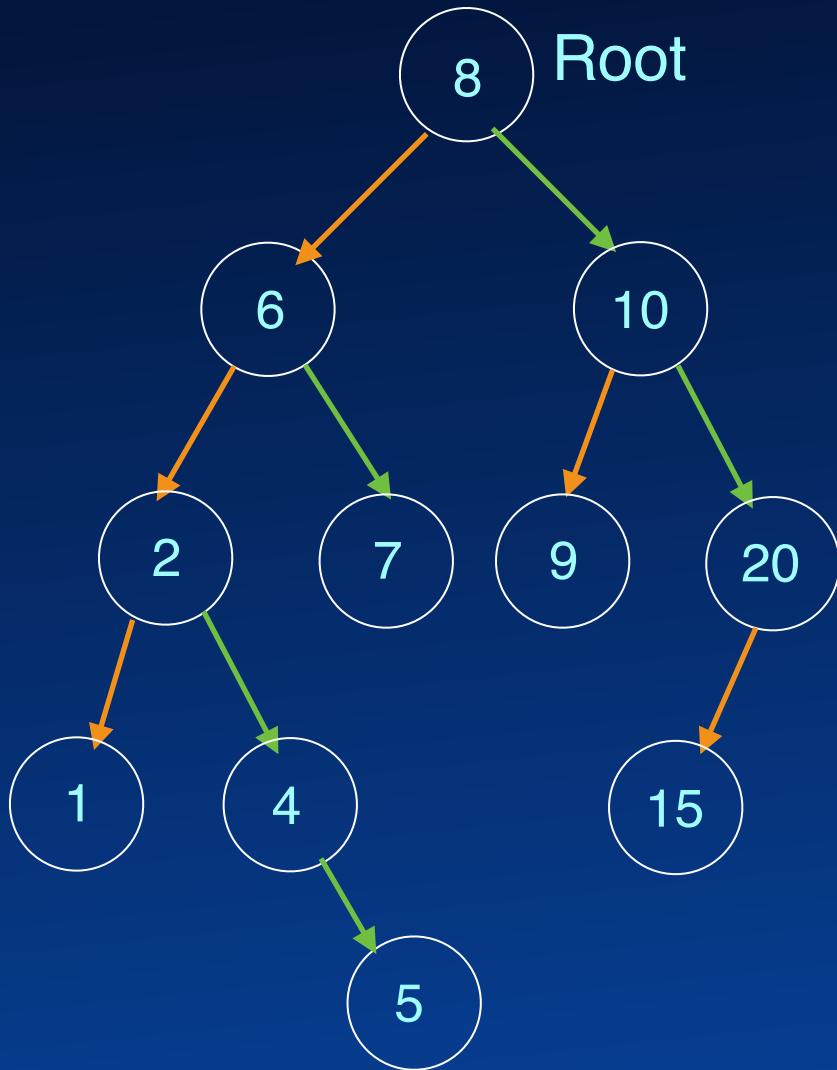


BST Operations



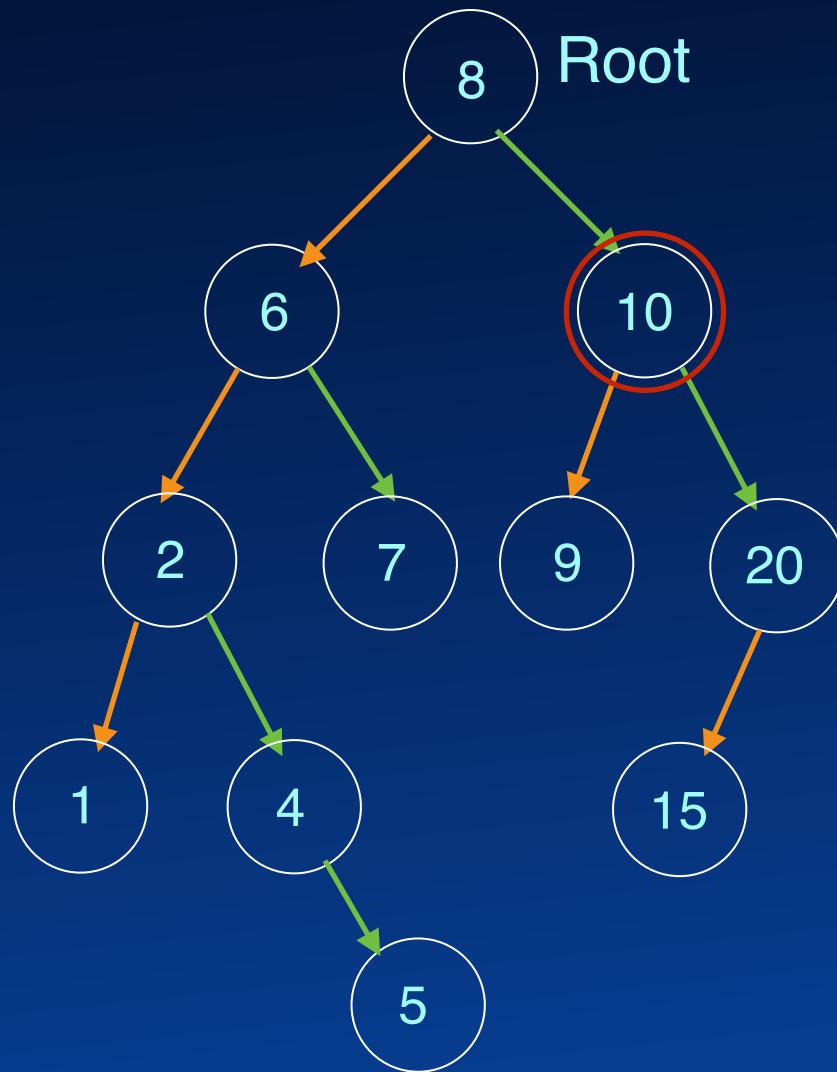


Successor



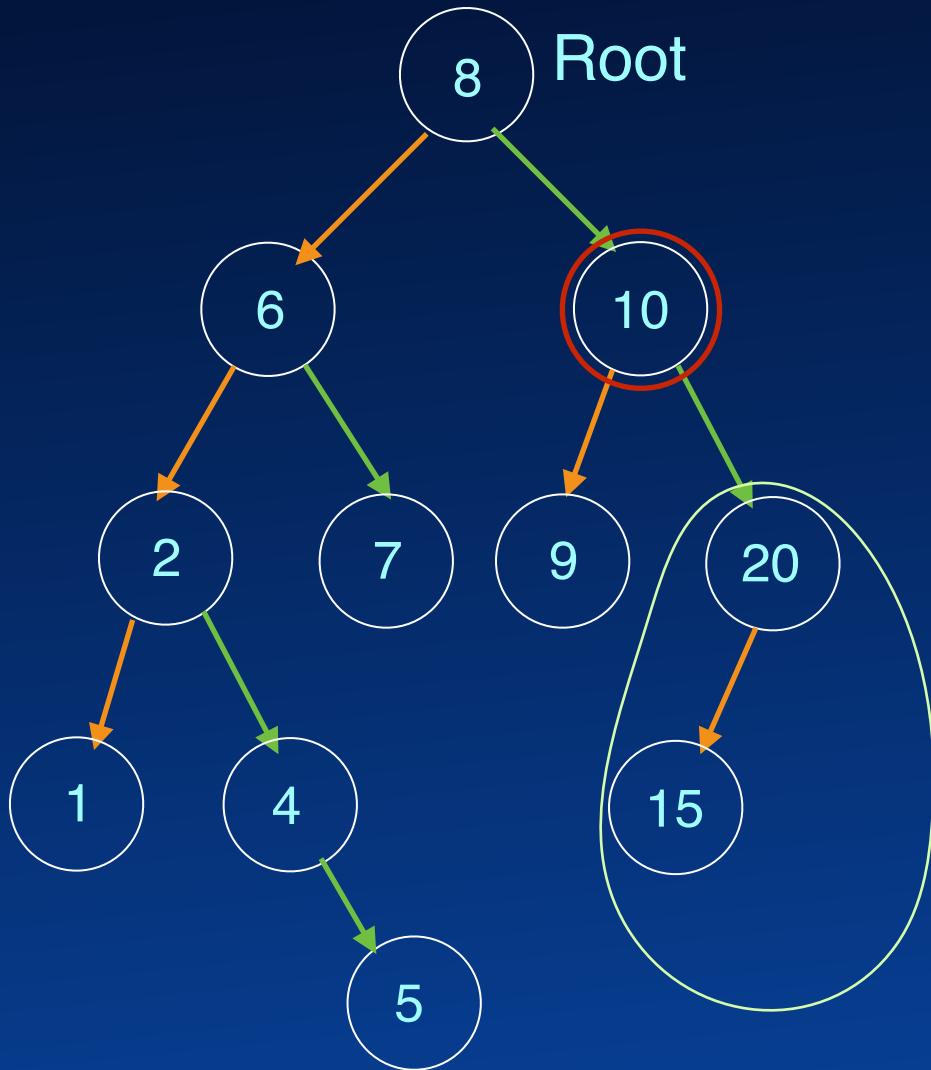


Successor



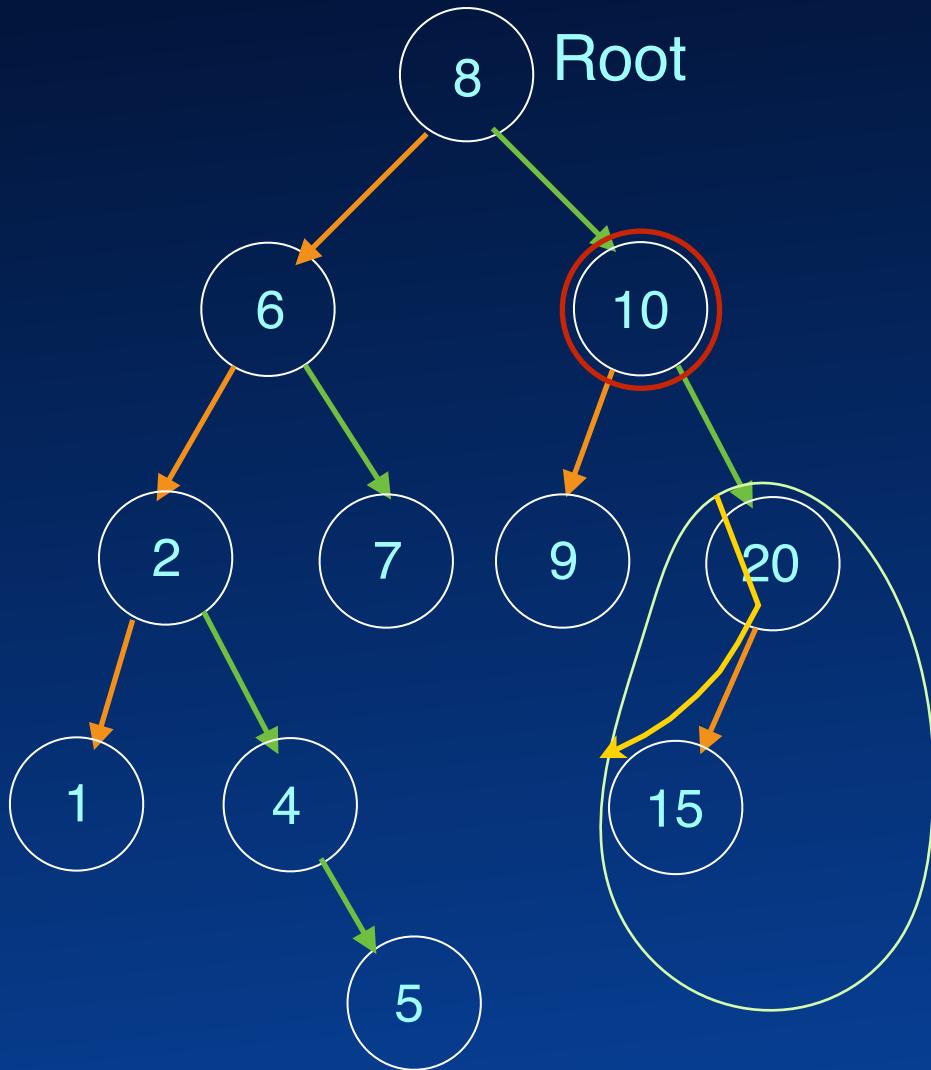


Successor



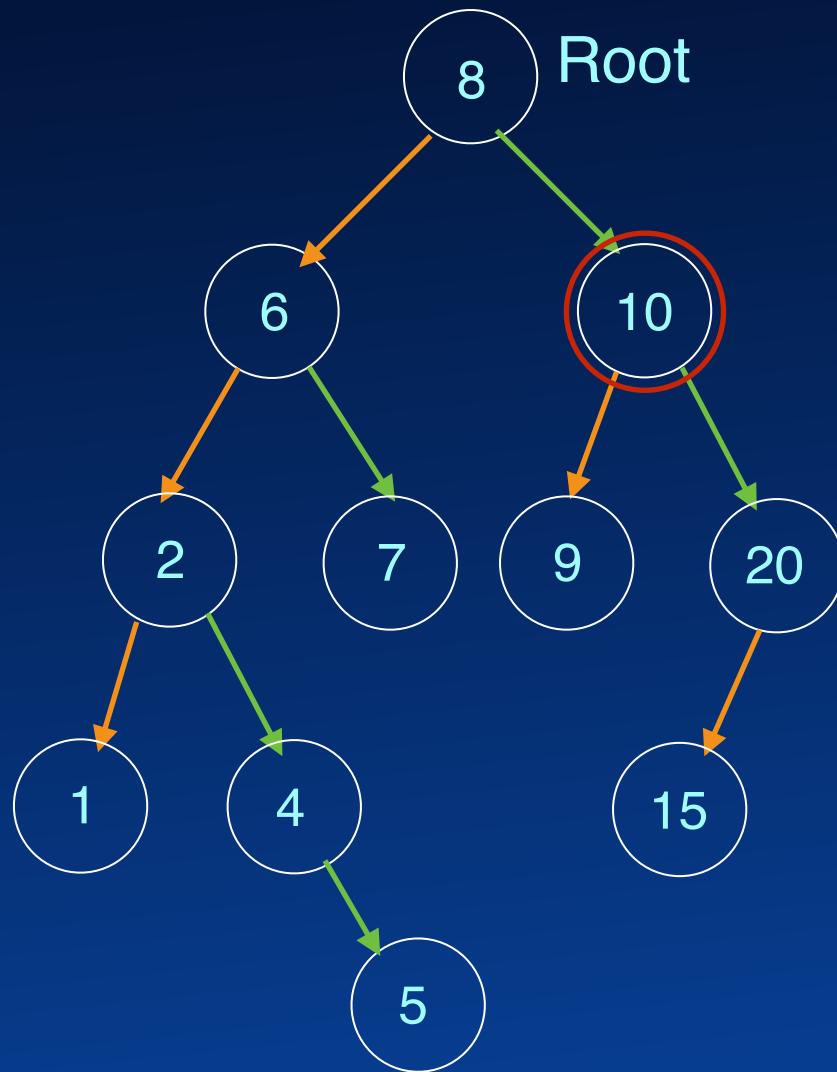


Successor



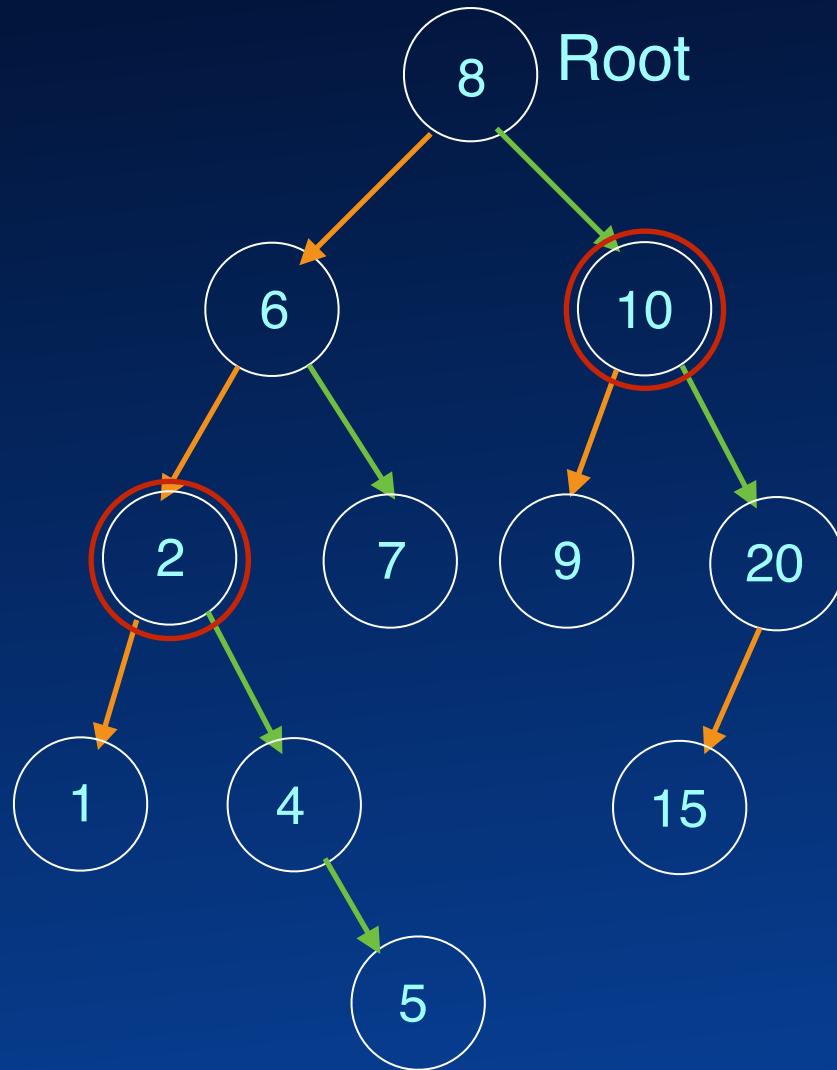


Successor



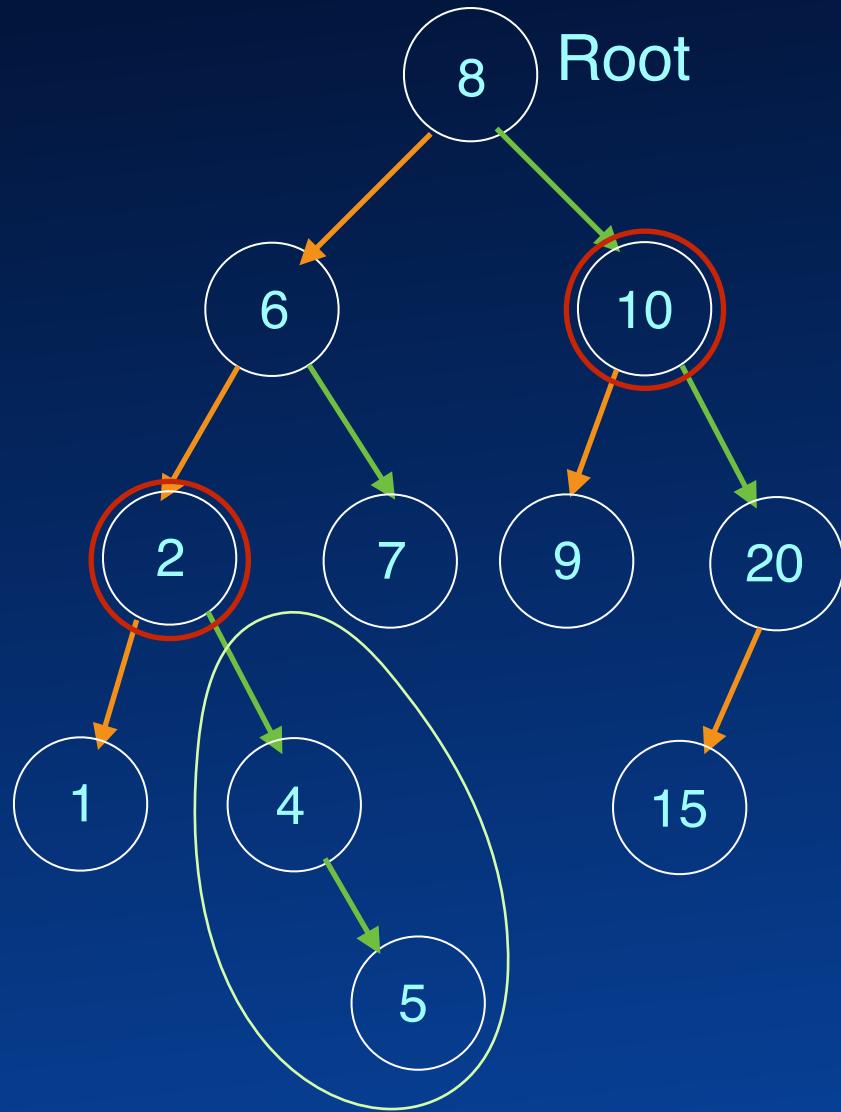


Successor



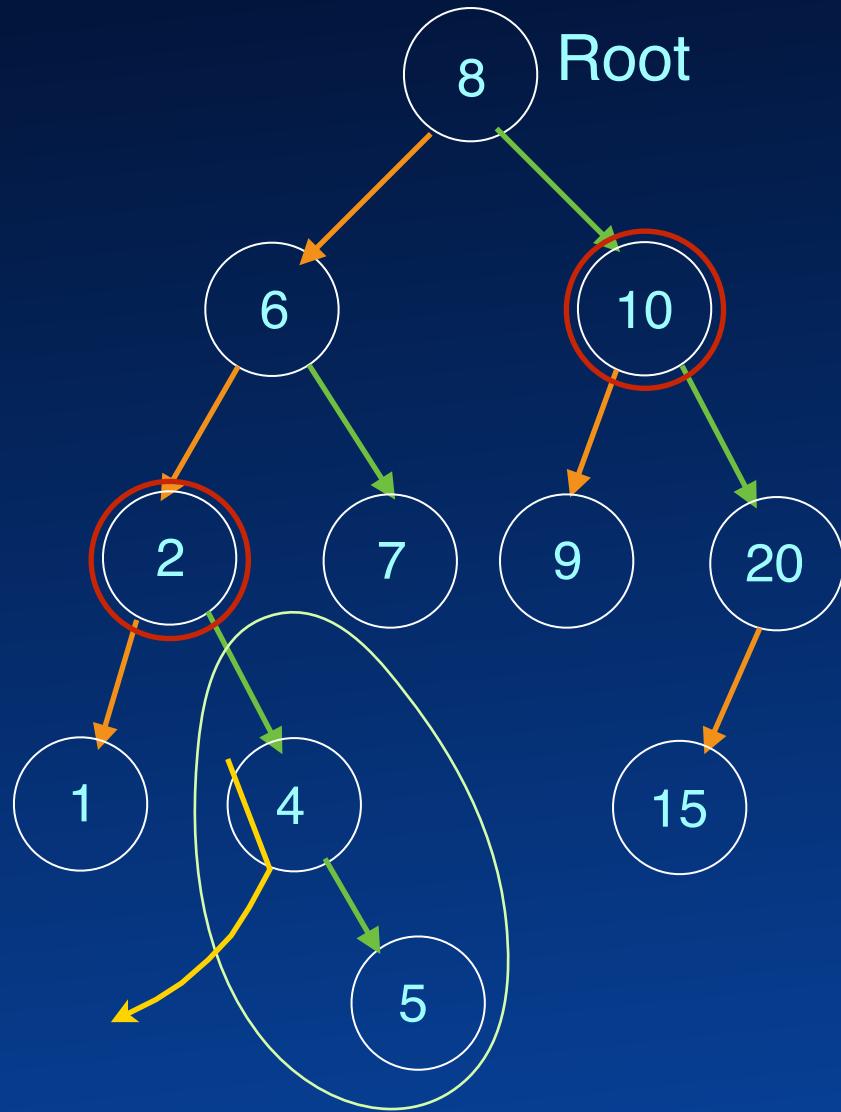


Successor





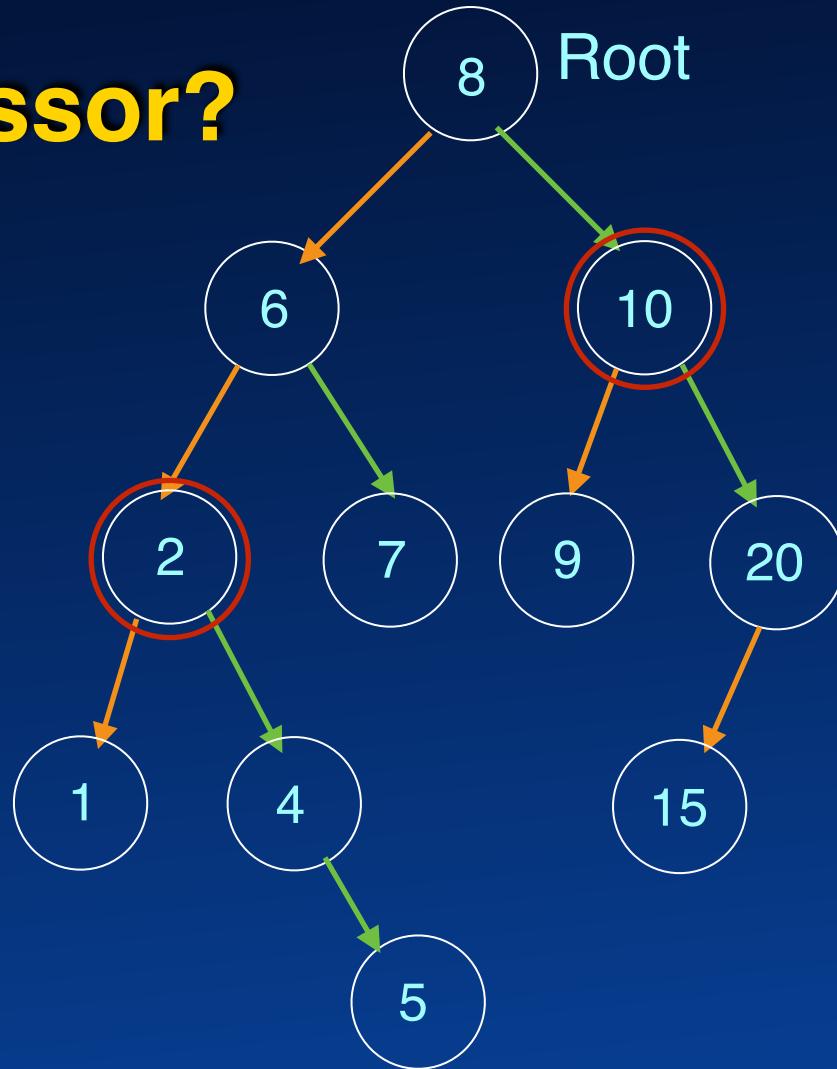
Successor





Successor

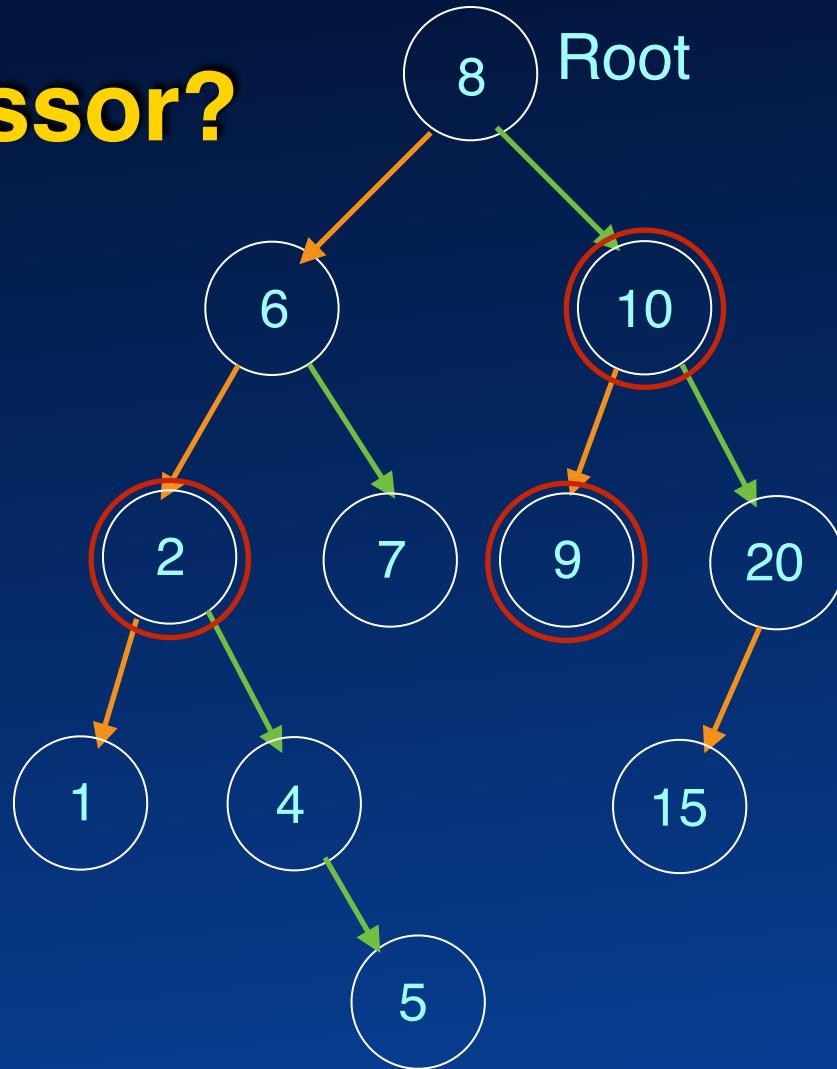
Predecessor?





Successor

Predecessor?





True or False?

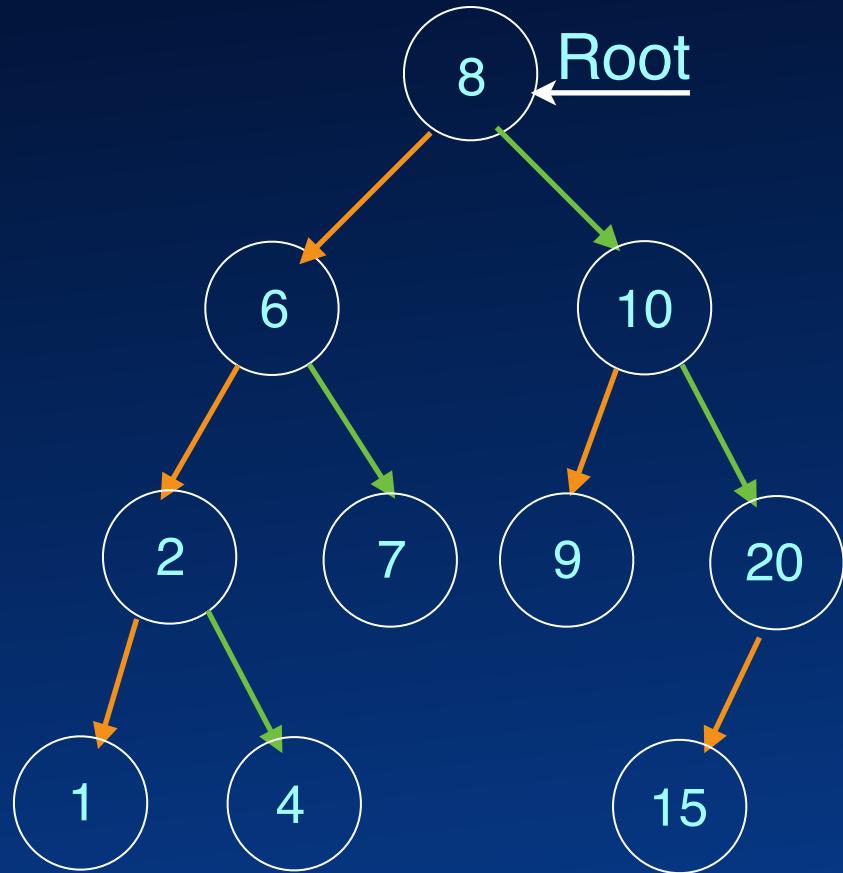
Format: t,f,f,f

Mail: col106quiz@cse.iitd.ac.in

- **The maximum height of a binary tree with n nodes is n-1**
- **The maximum height of a proper binary tree with n nodes is $2\log n + 1$**
- **The minimum height of a complete binary tree with n nodes, is $\text{ceil}(\log(n+1))$**
- **The number of non-leaf nodes in an n-node tree is always $\leq n/2$**

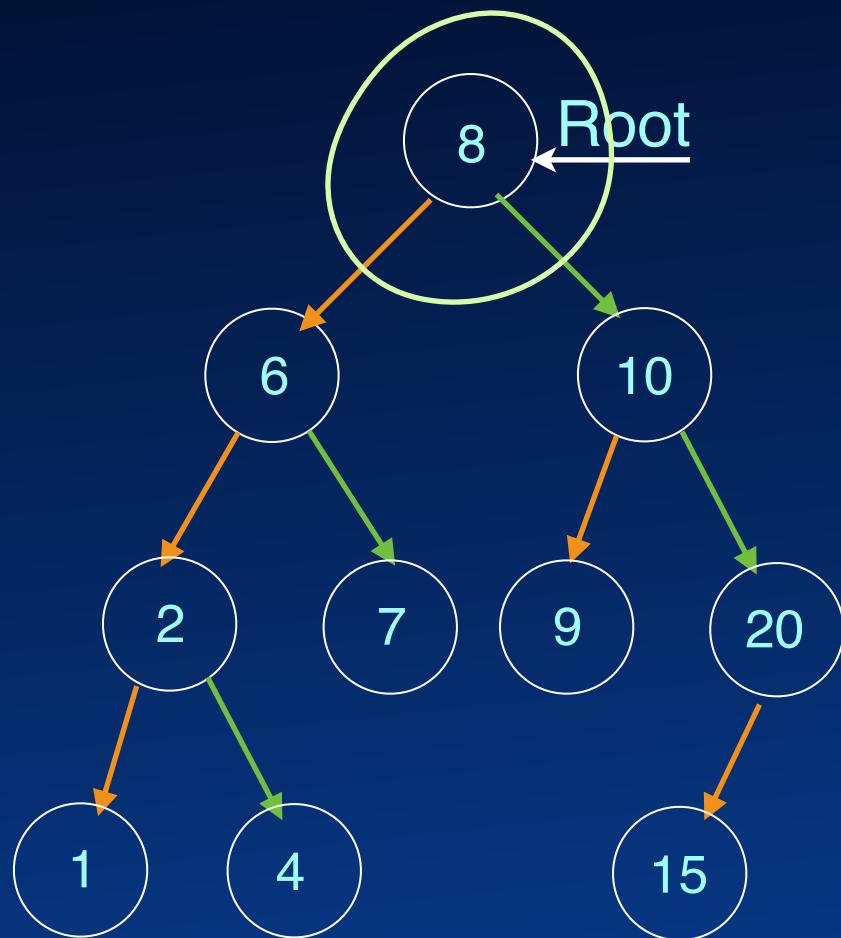


Traversal



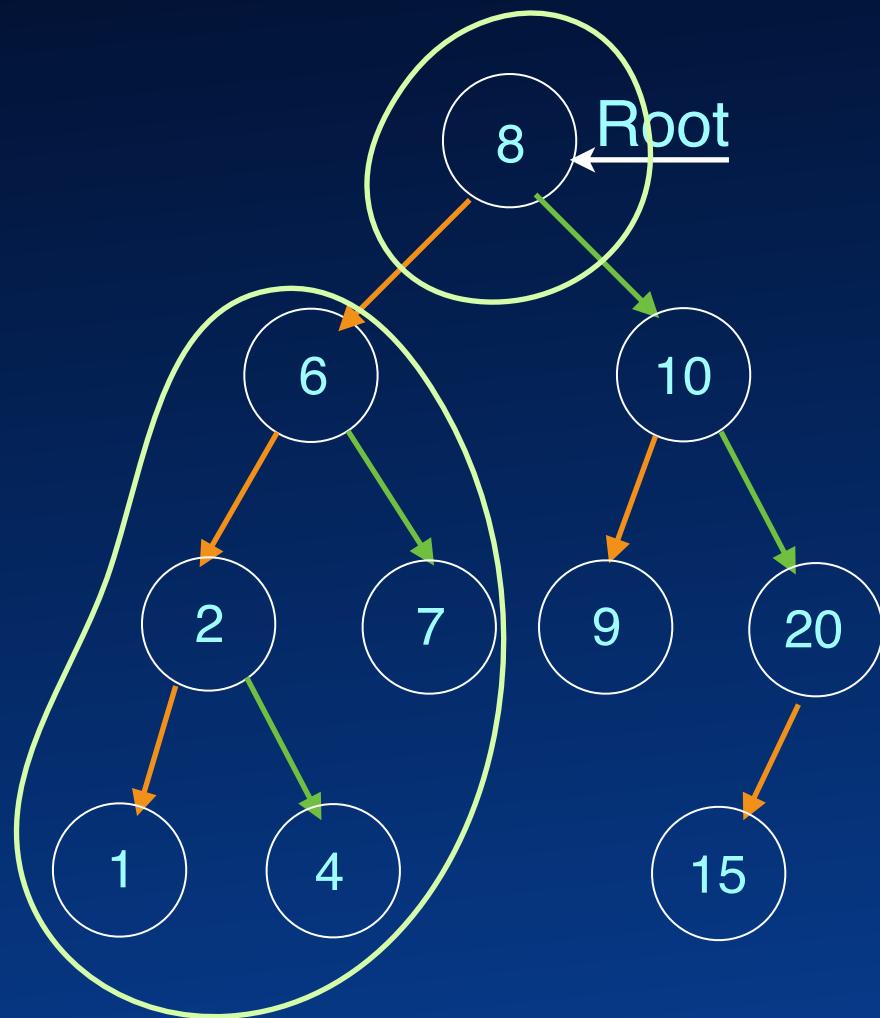


Traversal



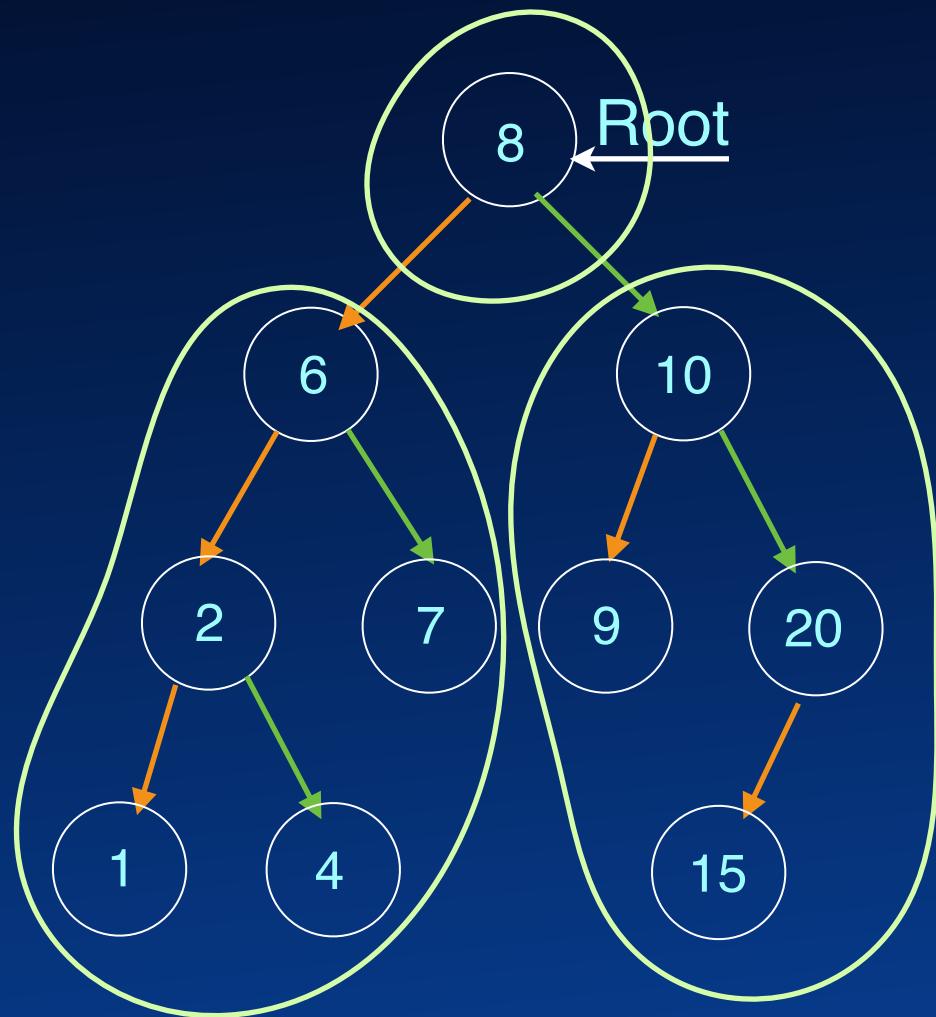


Traversal



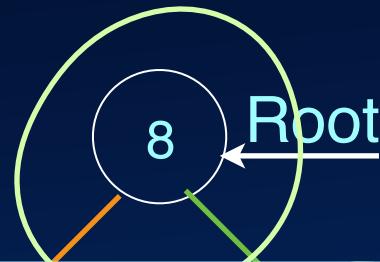


Traversal





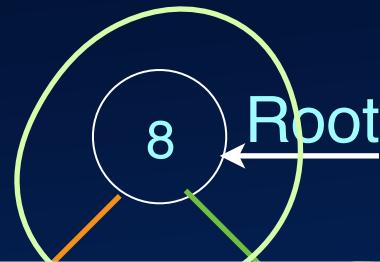
Traversal



```
class BST<T extends Comparable<T>> {
    BinaryNode<T> root;
    void iterate(Consumer<T> op) { iterate(root, op); }
    void iterate(BinaryNode<T> node, Consumer<T> op) {
        if(node == null) return;
        op.accept(node.value);
        iterate(node.left, op);
        iterate(node.right, op);
    }
    ...
}
```



Traversal

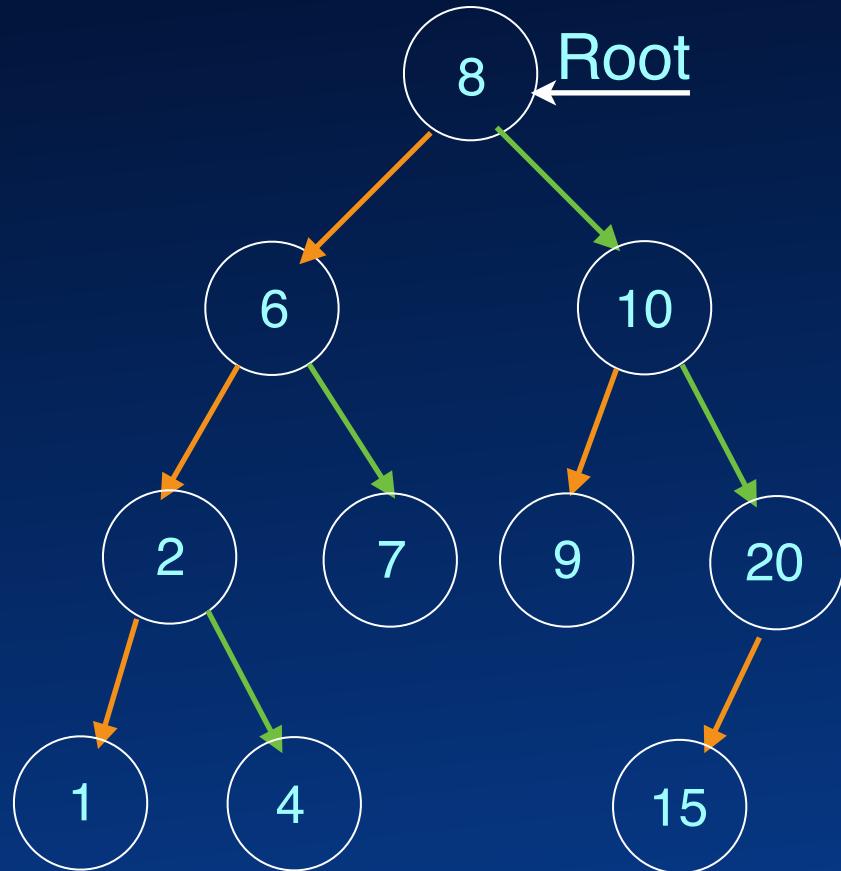


```
class BST<T extends Comparable<T>> {  
    BinaryNode<T> root;  
    void iterate(Consumer<T> op) { iterate(root, op); }  
    void iterate(BinaryNode<T> node, Consumer<T> op) {  
        if(node == null) return;  
        op.accept(node.value);  
        iterate(node.left, op);  
        iterate(node.right, op);  
    }  
    ...  
}
```

Pre-order Traversal



Traversal





Traversal



```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```



Traversal

```
interface SortablePair<K extends Comparable<K>, V> {  
    K key();  
    V value();  
}
```



```
class SortablePair < implements SortablePosition<PT> {  
    PT key;  
    PT value;  
    SortablePair left;  
    SortablePair right;  
    SortablePair parent;  
    int numChildren;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```



Traversal

```
interface SortablePair<K extends Comparable<K>, V> {
```

```
    K key();  
    V value();  
}
```

```
interface Position2way<T> {  
    Position2way<T> left();  
    Position2way<T> right();  
    T value();  
}
```



```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```

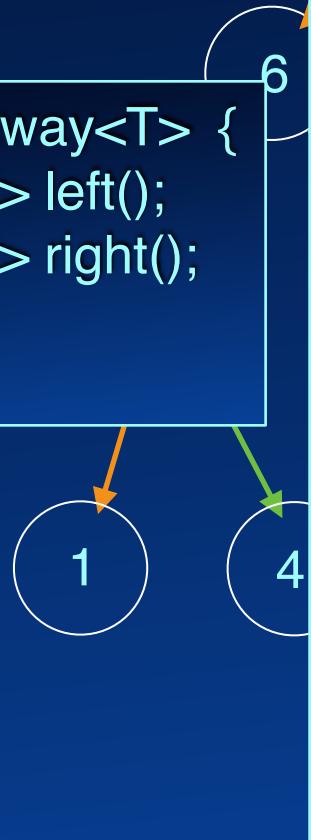


Traversal

```
interface SortablePair<K extends Comparable<K>, V> {
```

```
    K key();  
    V value();  
}
```

```
interface Position2way<T> {  
    Position2way<T> left();  
    Position2way<T> right();  
    T value();  
}
```



```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```

In-order Traversal

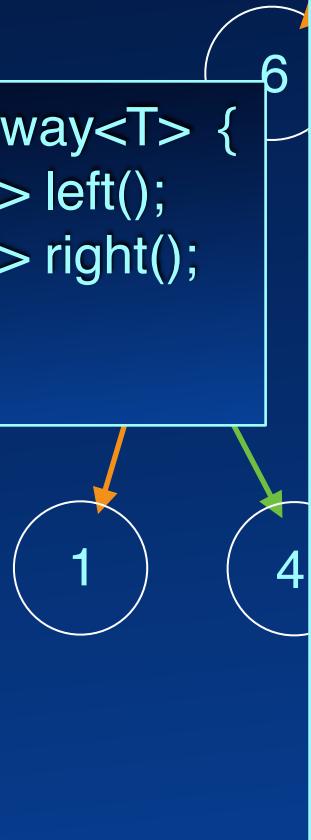


Traversal

```
interface SortablePair<K extends Comparable<K>, V> {
```

```
    K key();  
    V value();  
}
```

```
interface Position2way<T> {  
    Position2way<T> left();  
    Position2way<T> right();  
    T value();  
}
```



```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```

In-order Traversal



Traversal

```
interface SortablePair<K extends Comparable<K>, V> {
```

```
    K key();  
    V value();  
}
```

```
interface Position2way<T> {  
    Position2way<T> left();  
    Position2way<T> right();  
    T value();  
}
```



```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
}
```

```
tree.iterate(t -> {System.out.println(t.value());});
```

```
...  
}
```

In-order Traversal



Traversal

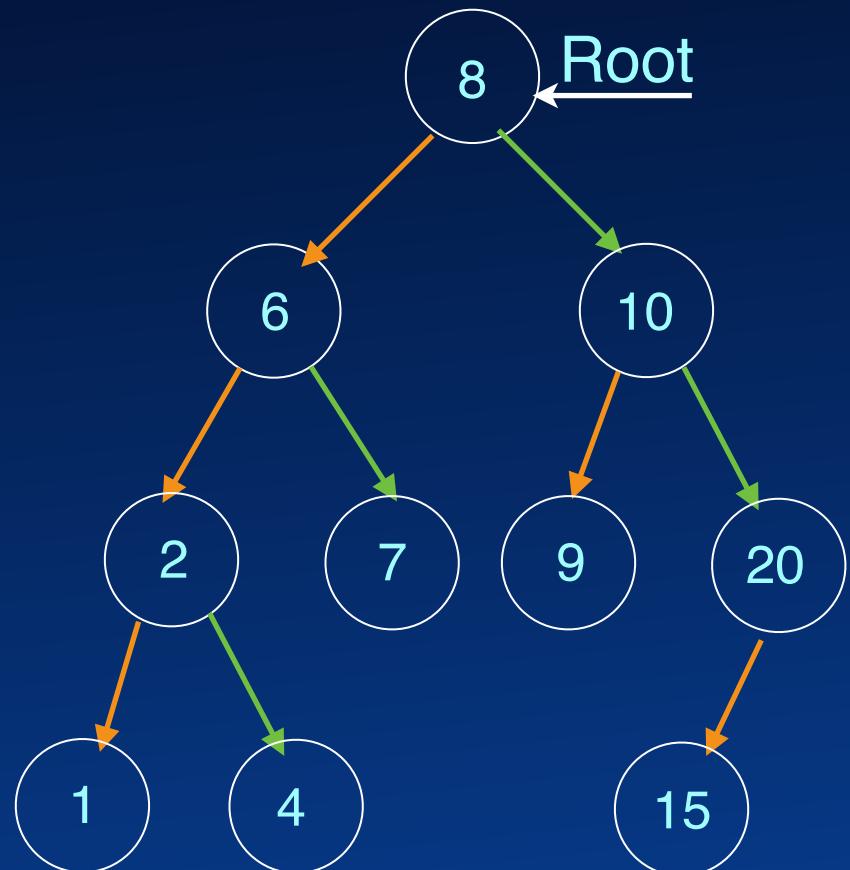


```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```



Traversal

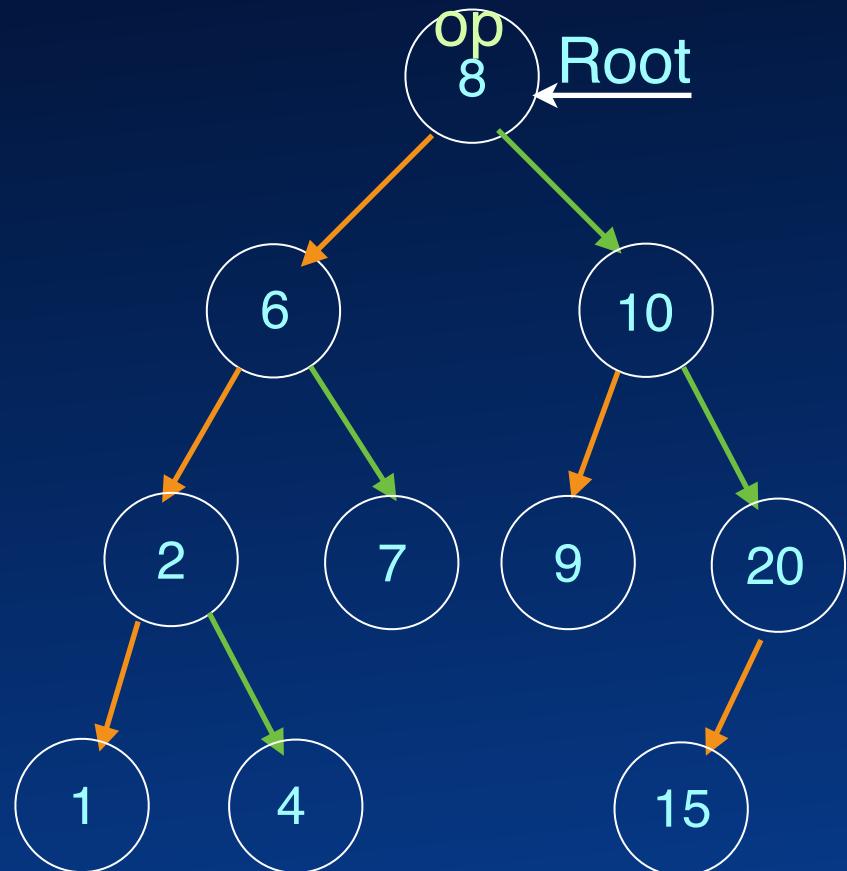
```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```





Traversal

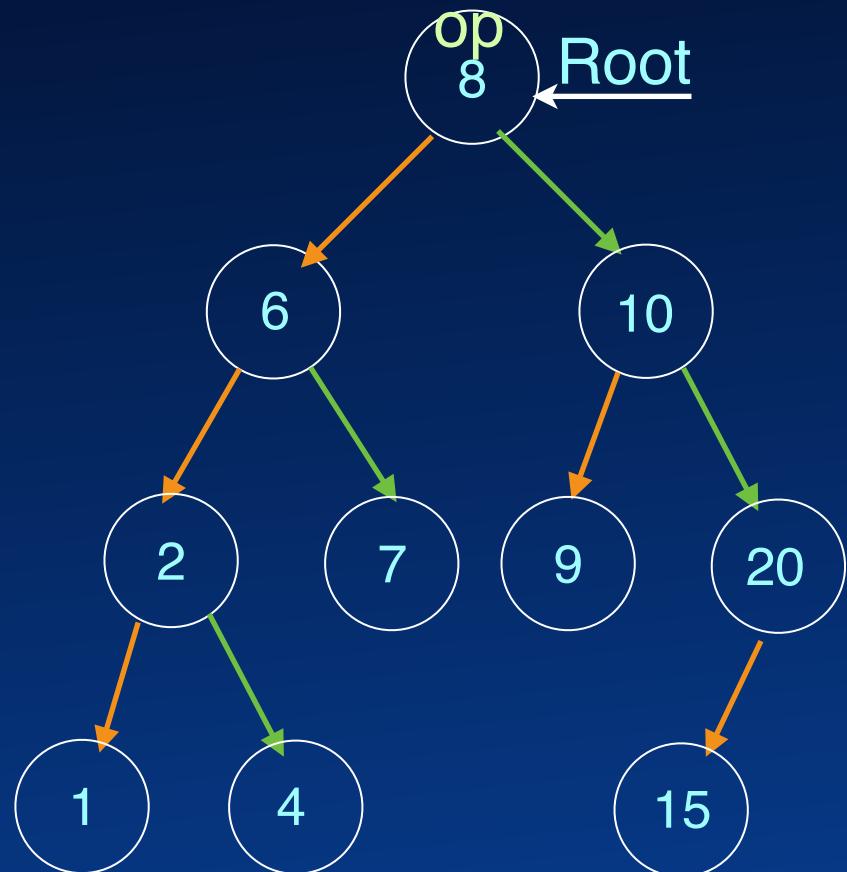
```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```





Traversal

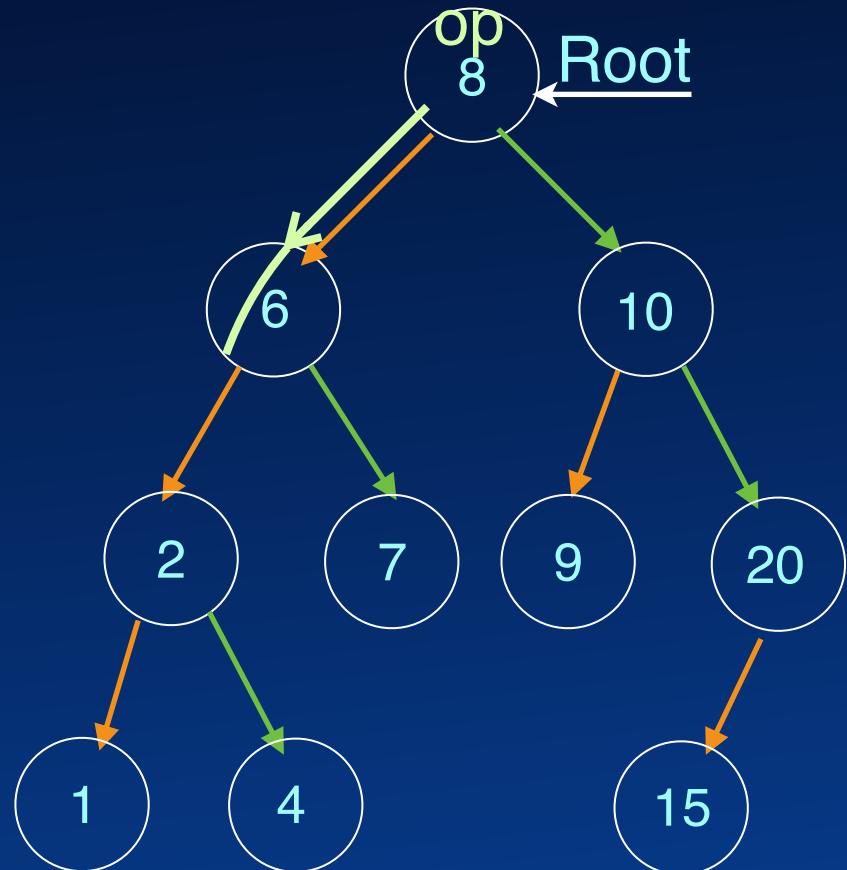
```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```





Traversal

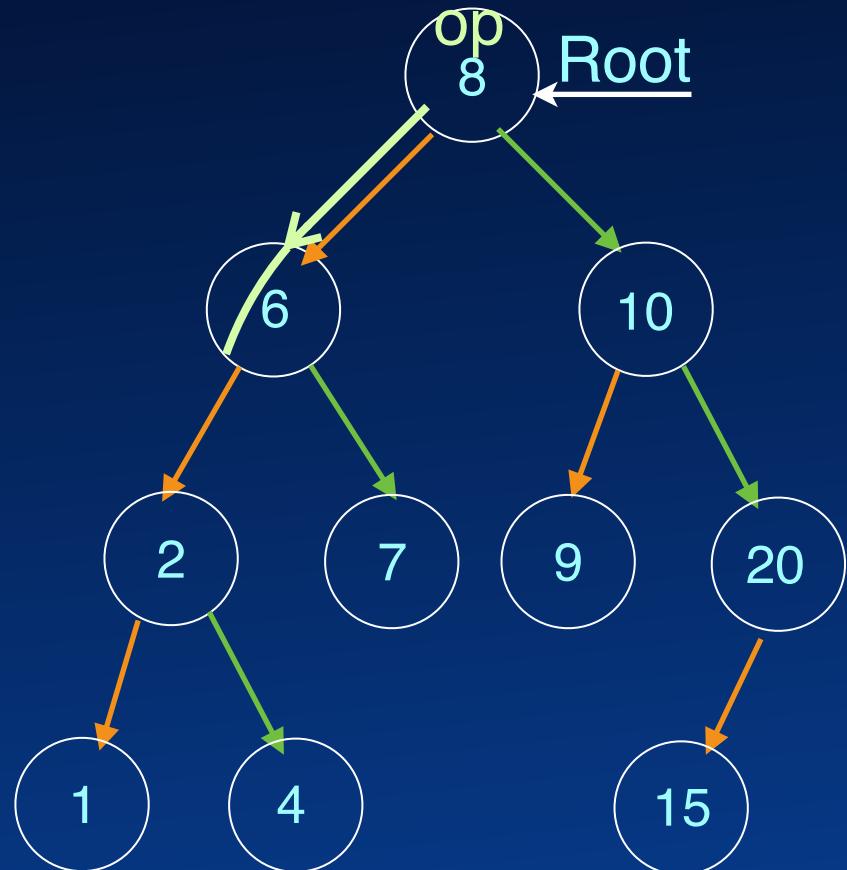
```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```





Traversal

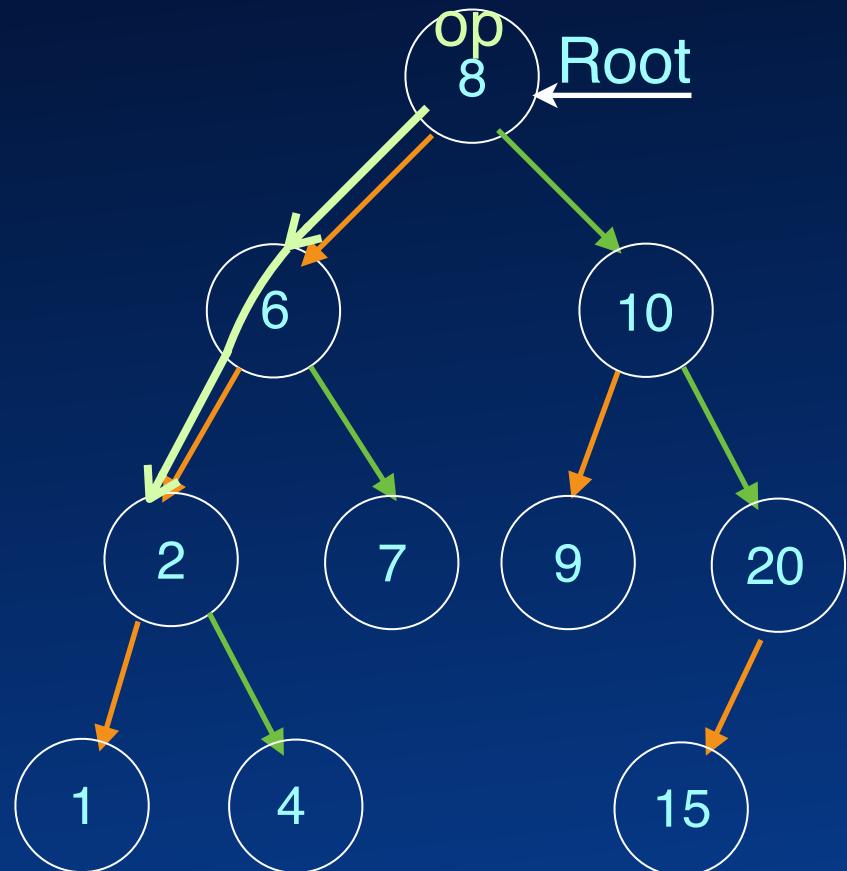
```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```





Traversal

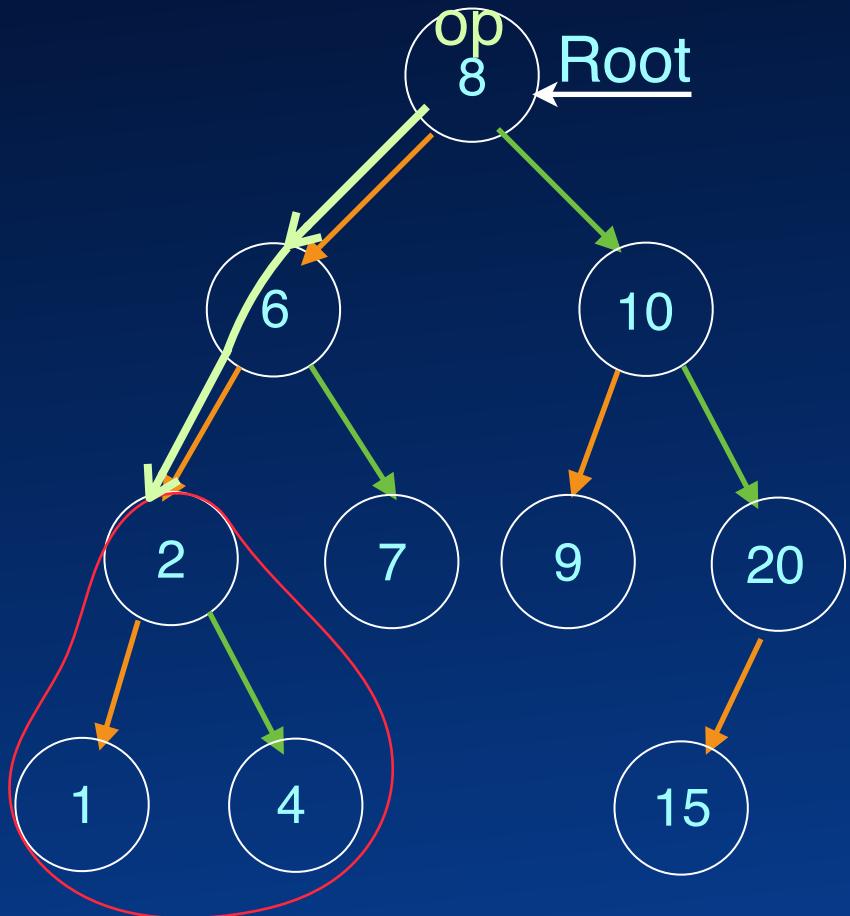
```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```





Traversal

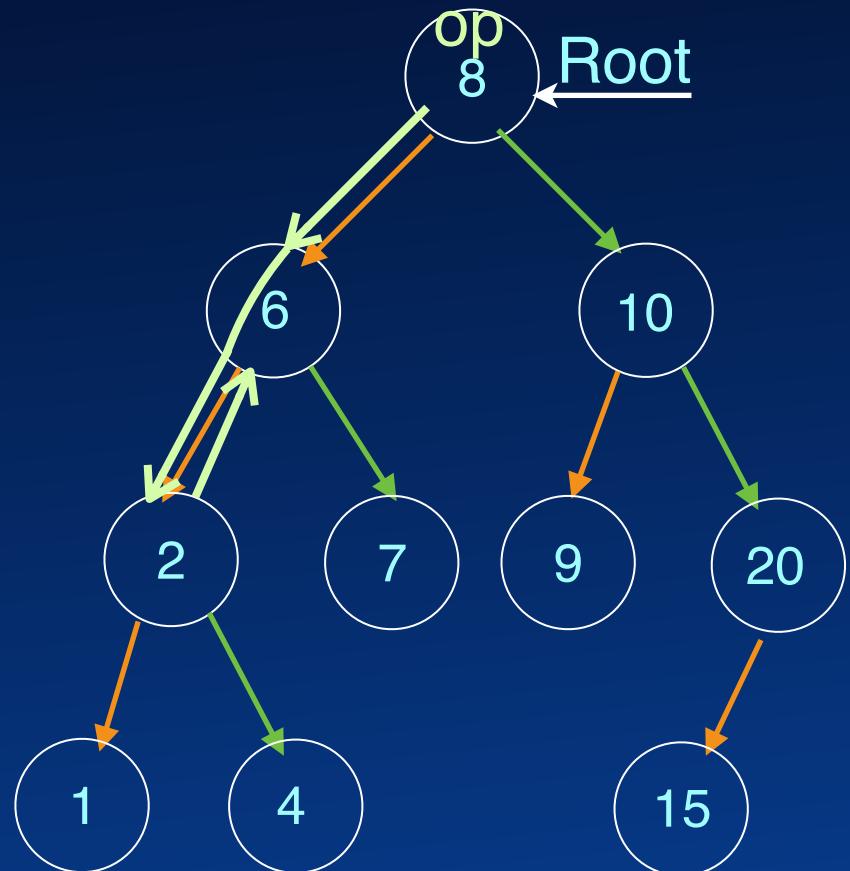
```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```





Traversal

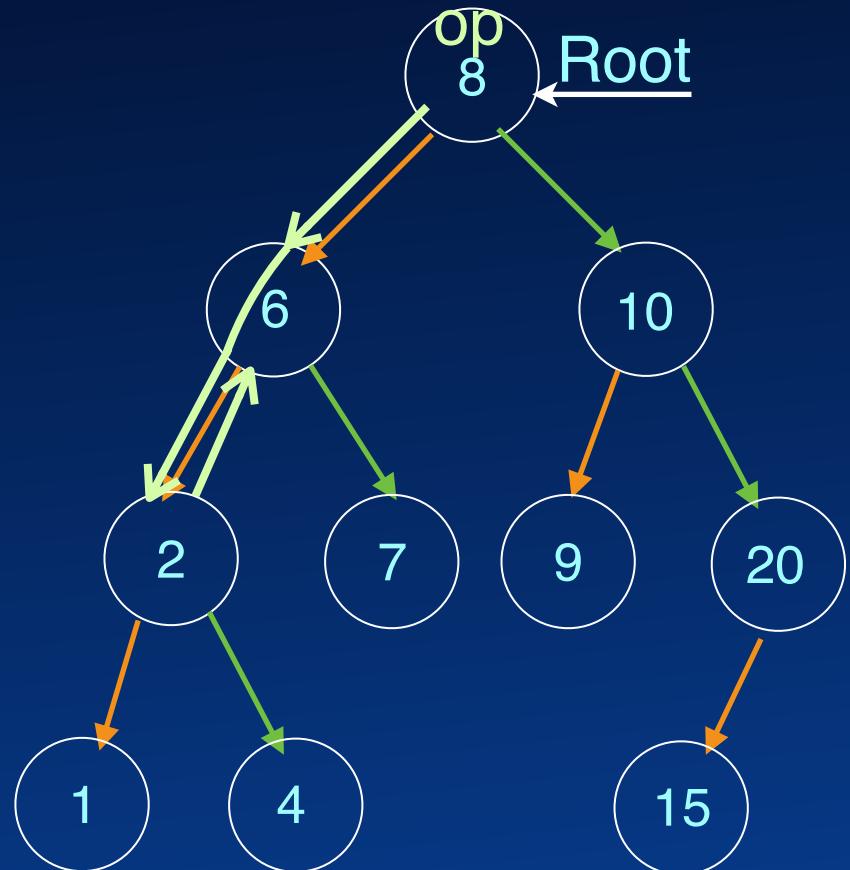
```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```





Traversal

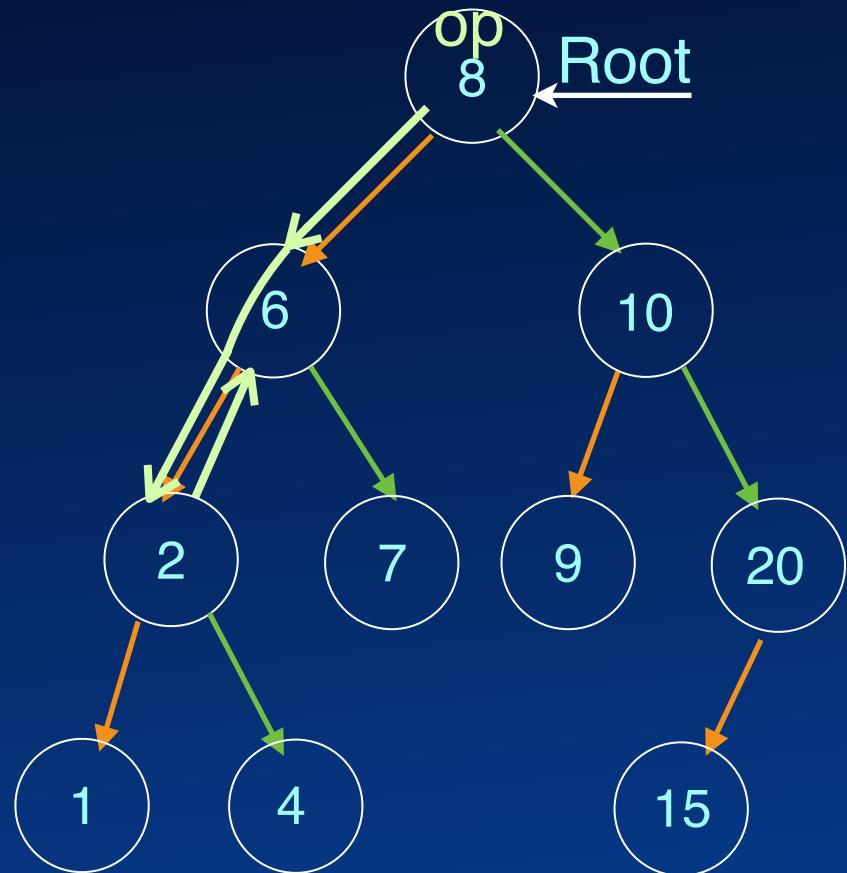
```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```





Traversal

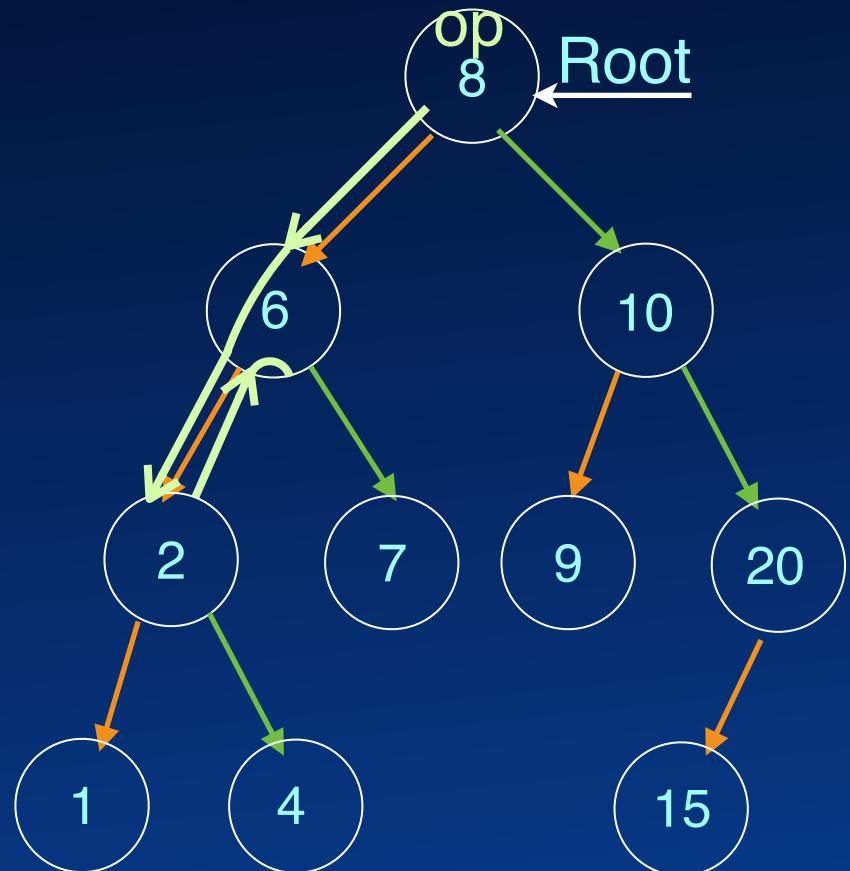
```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```





Traversal

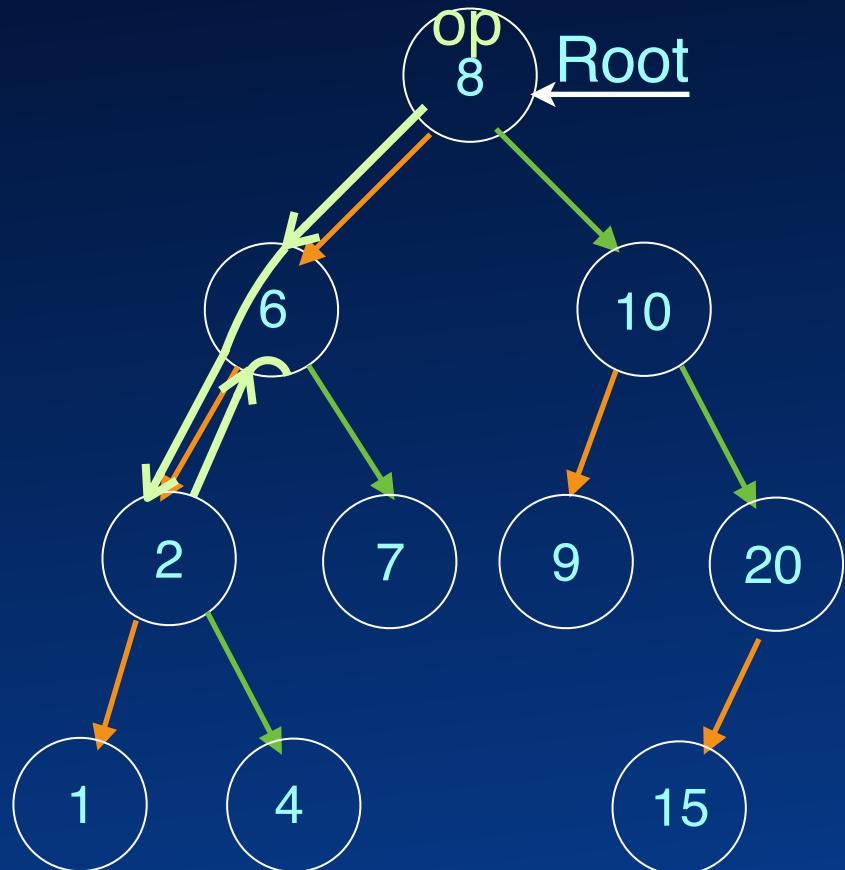
```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```





Traversal

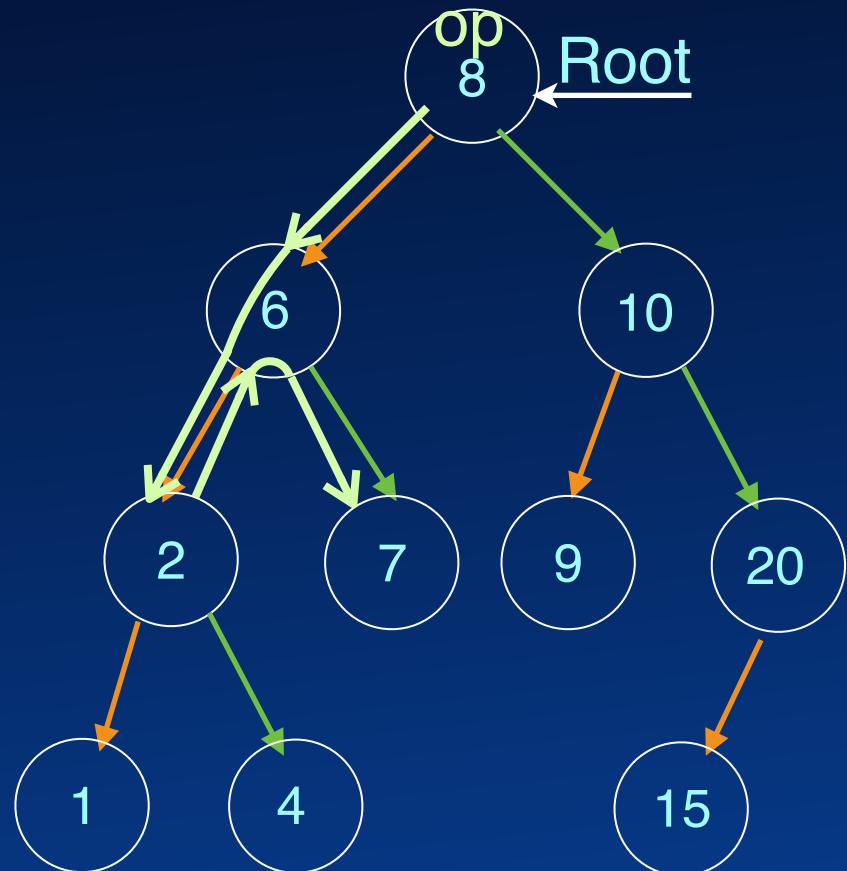
```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```





Traversal

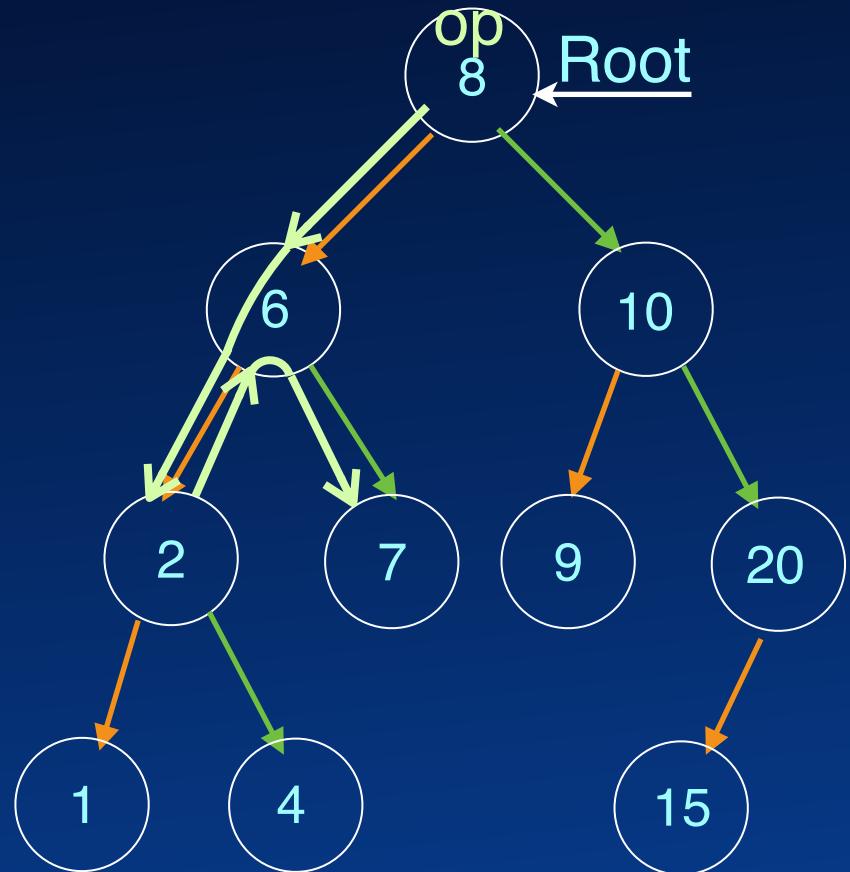
```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```





Traversal

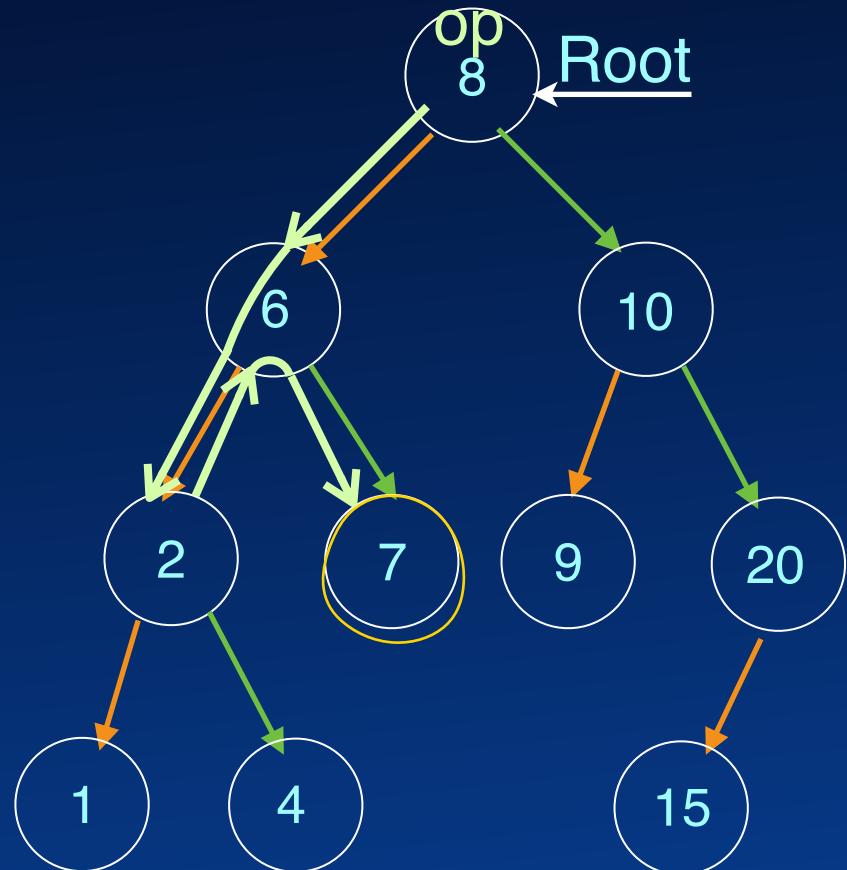
```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```





Traversal

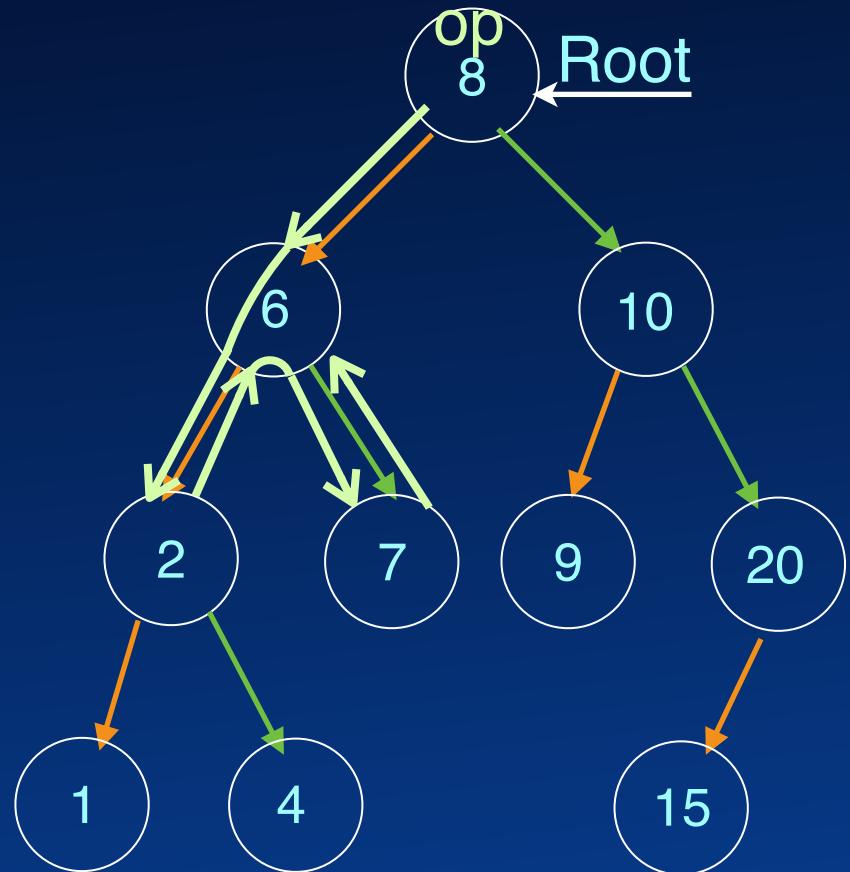
```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```





Traversal

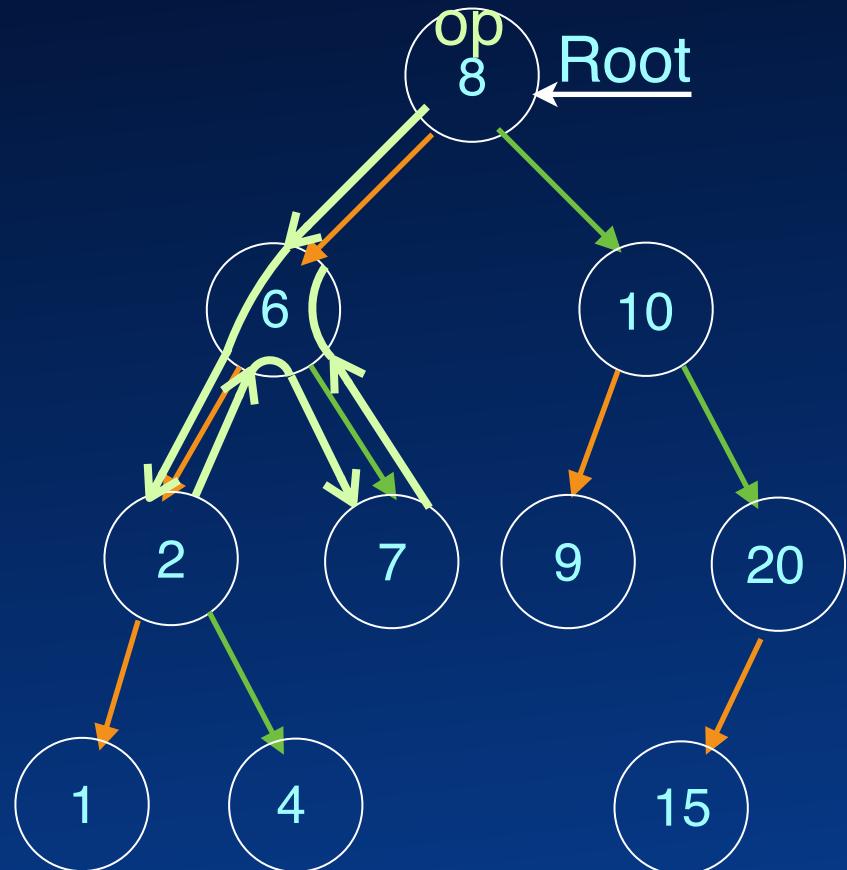
```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```





Traversal

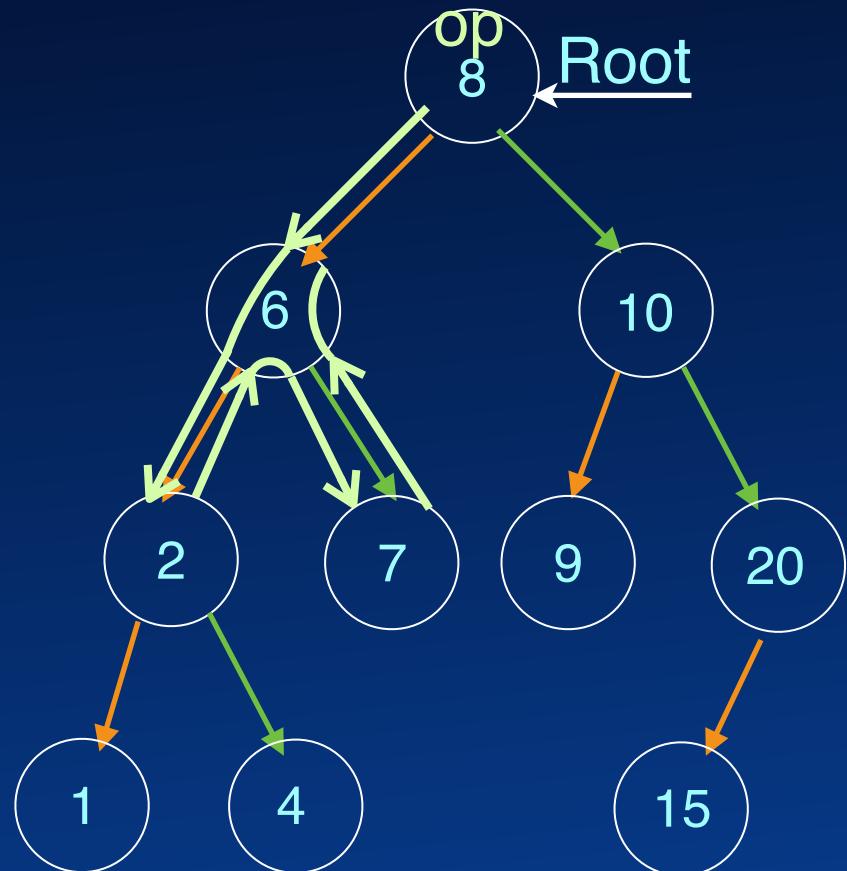
```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```





Traversal

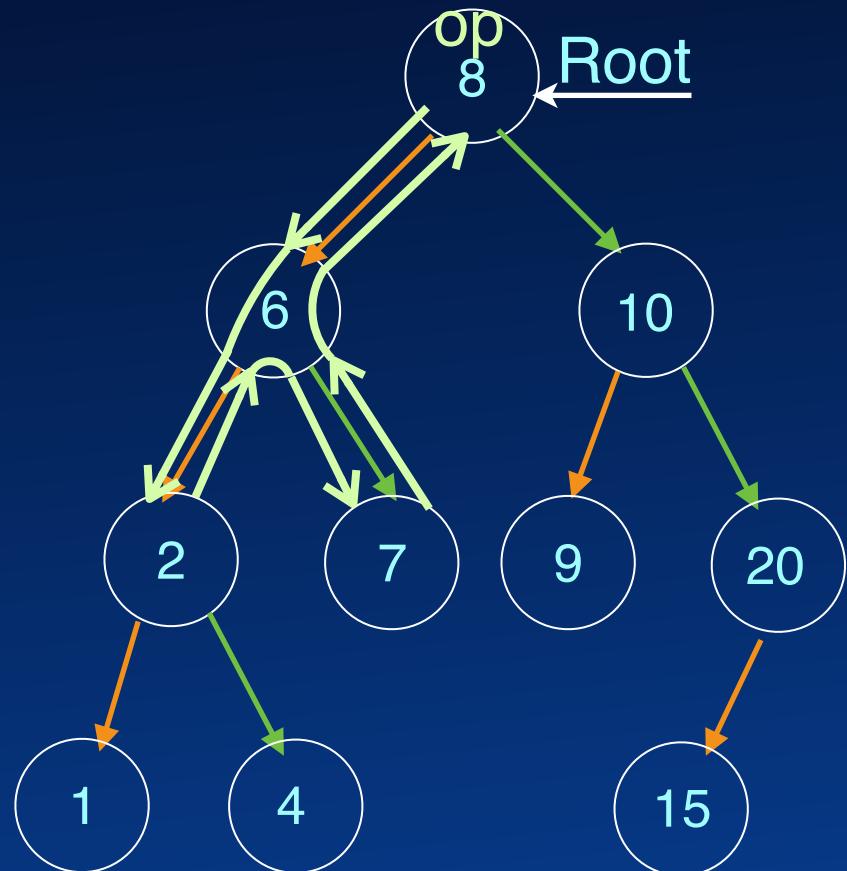
```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```





Traversal

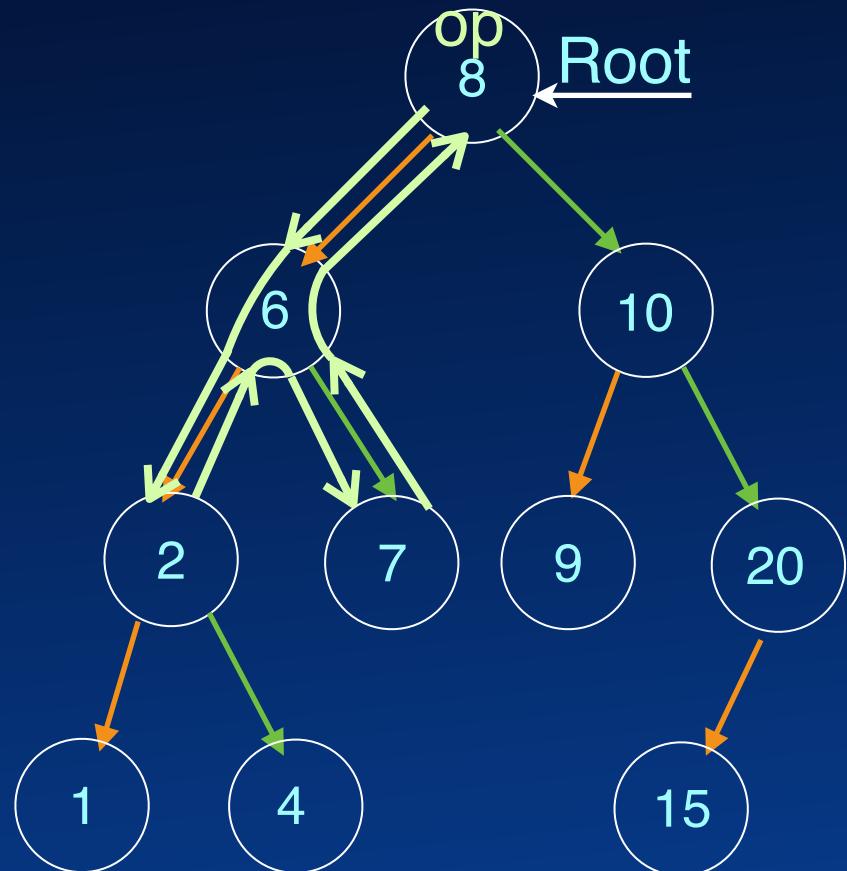
```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```





Traversal

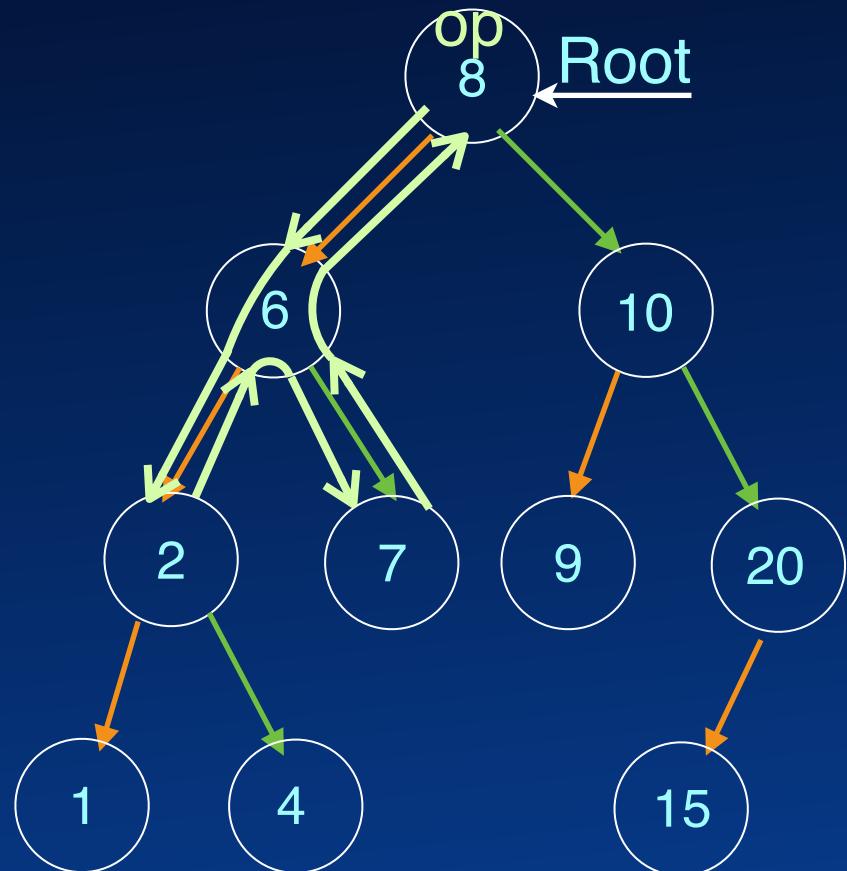
```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```





Traversal

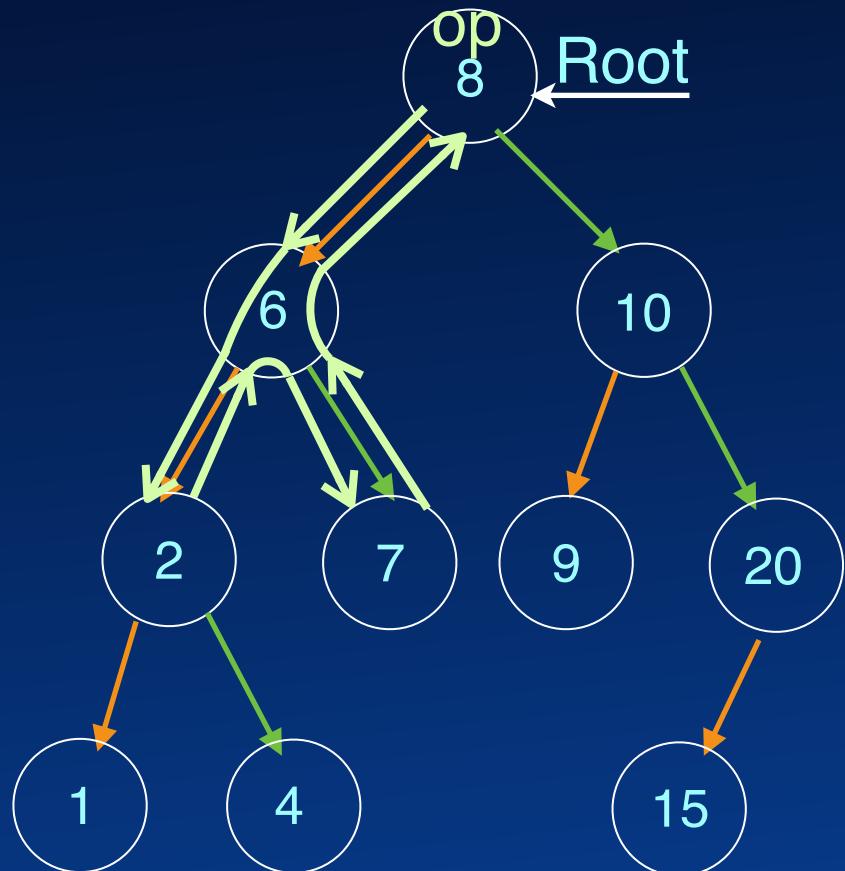
```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```





Traversal

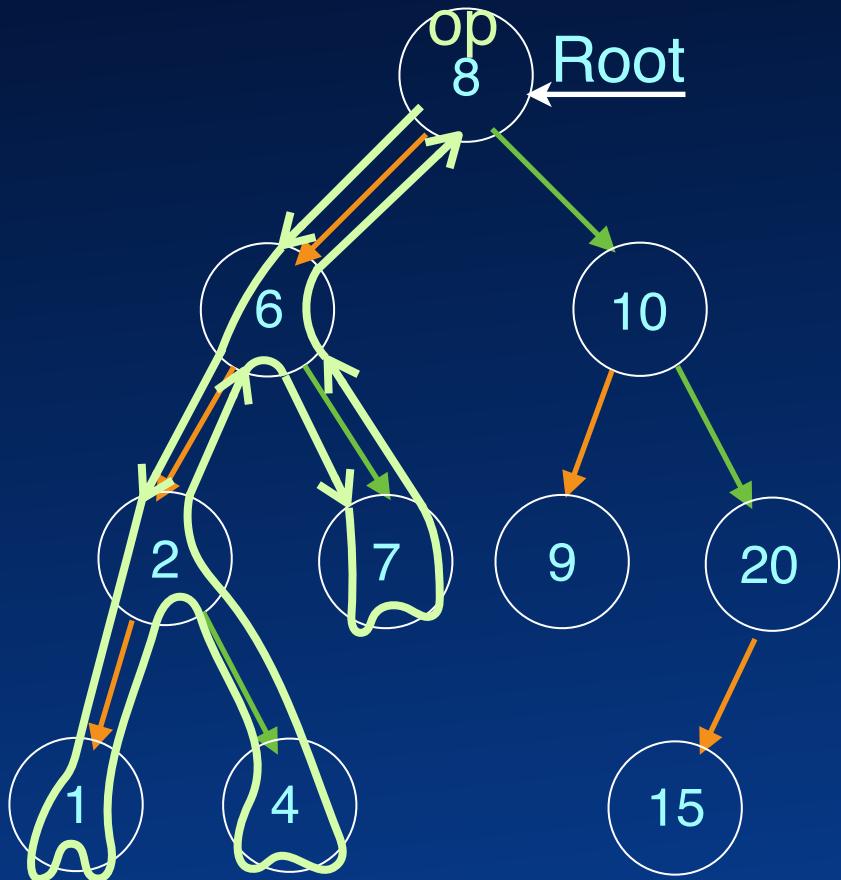
```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```





Traversal

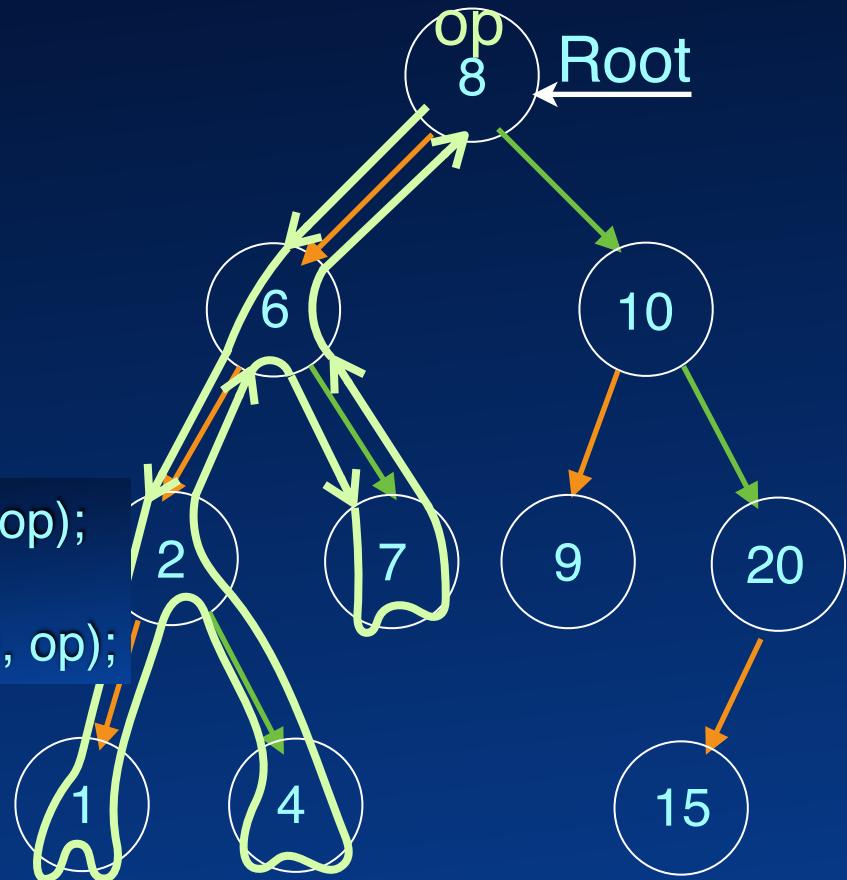
```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```





Traversal

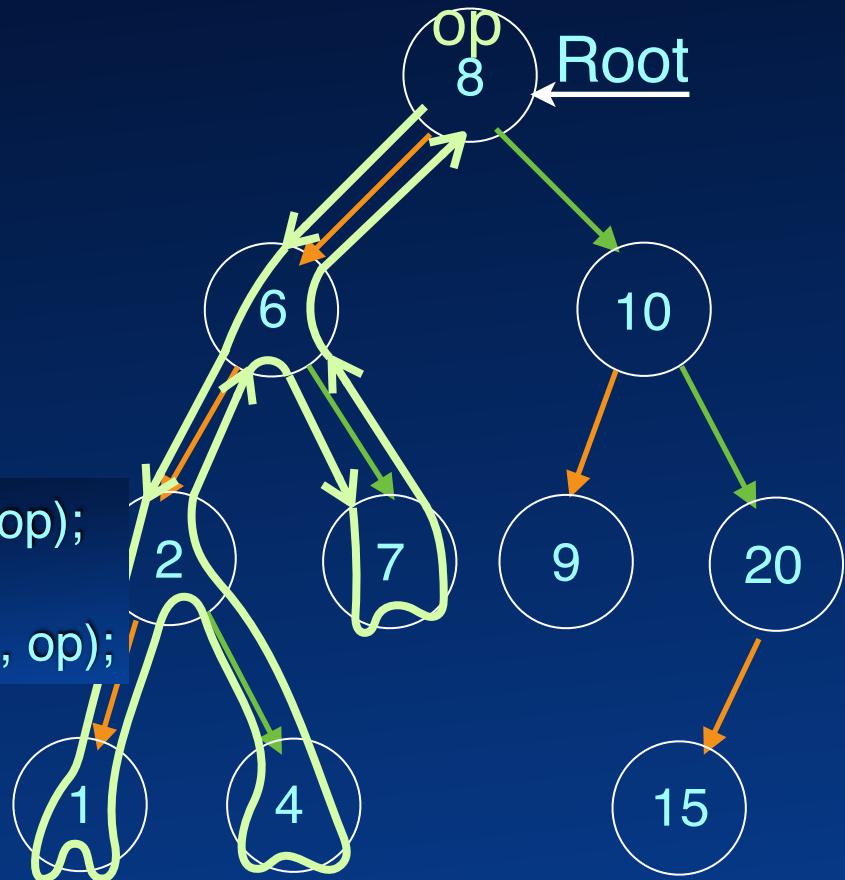
```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        if(node.left() != null) iterate(node.left(), op);  
        op.accept(node.value());  
        if(node.right() != null) iterate(node.right(), op);  
    }  
    ...  
}
```





Traversal

```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        if(root != null) iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        if(node.left() != null) iterate(node.left(), op);  
        op.accept(node.value());  
        if(node.right() != null) iterate(node.right(), op);  
    }  
    ...  
}
```

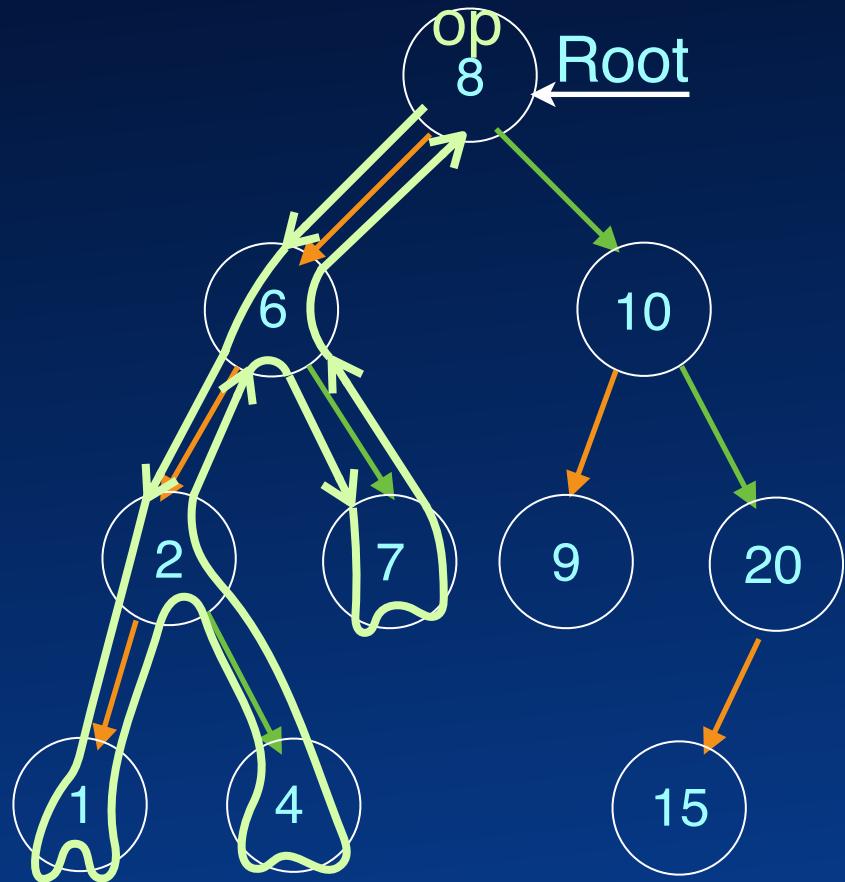




Traversal

```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```

Euler's Tour

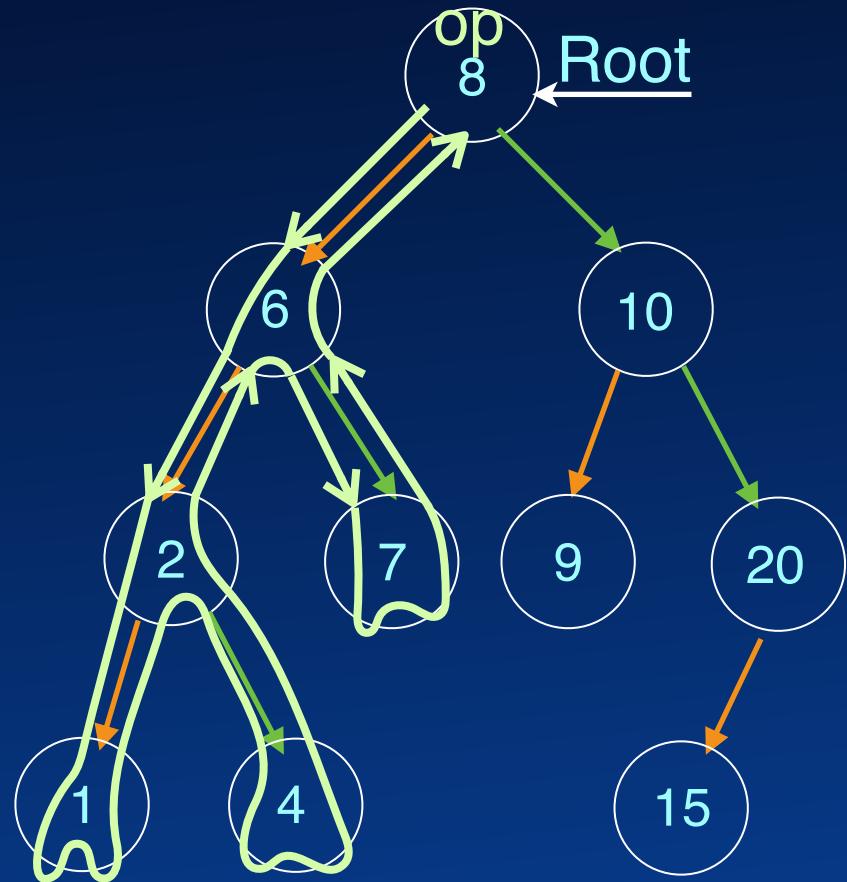




Traversal

```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```

Euler's Tour



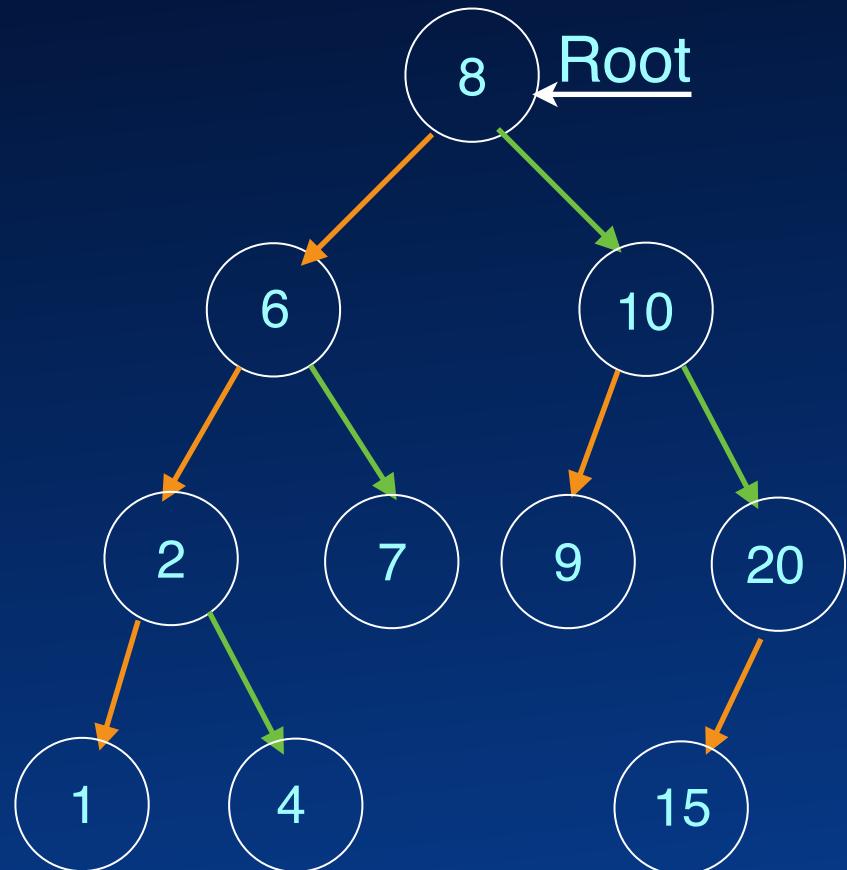
Depth First Traversal



Traversal

```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```

In-order Traversal

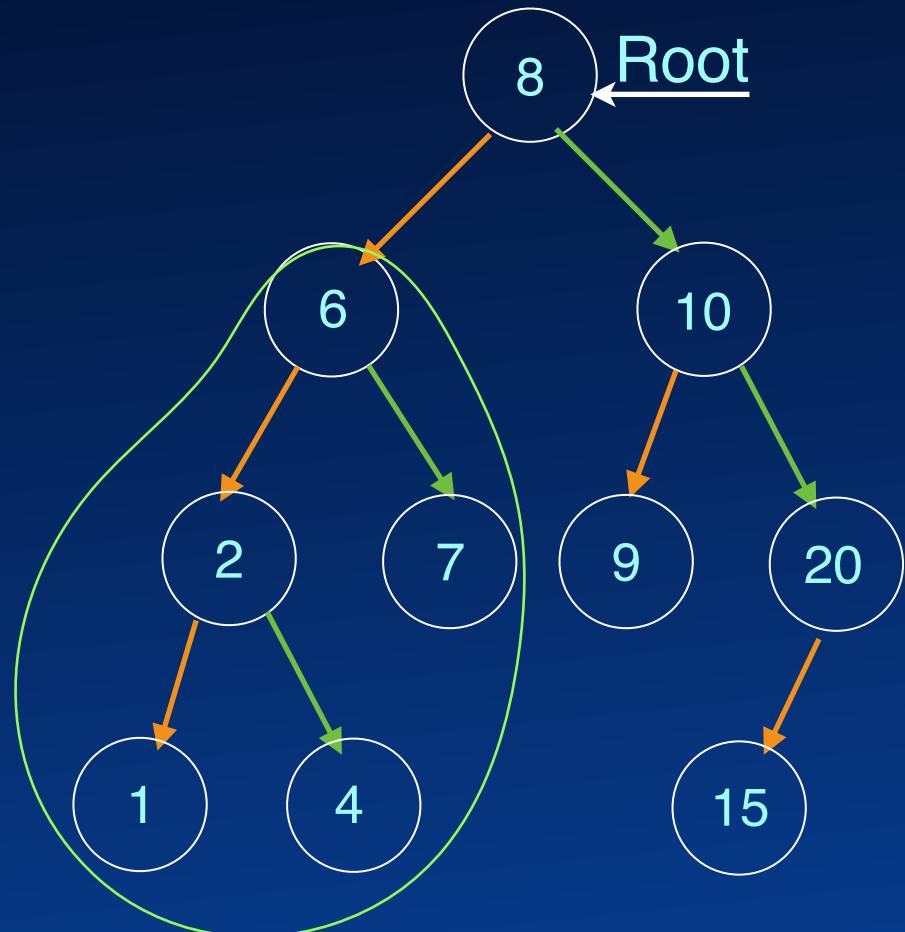




Traversal

```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```

In-order Traversal

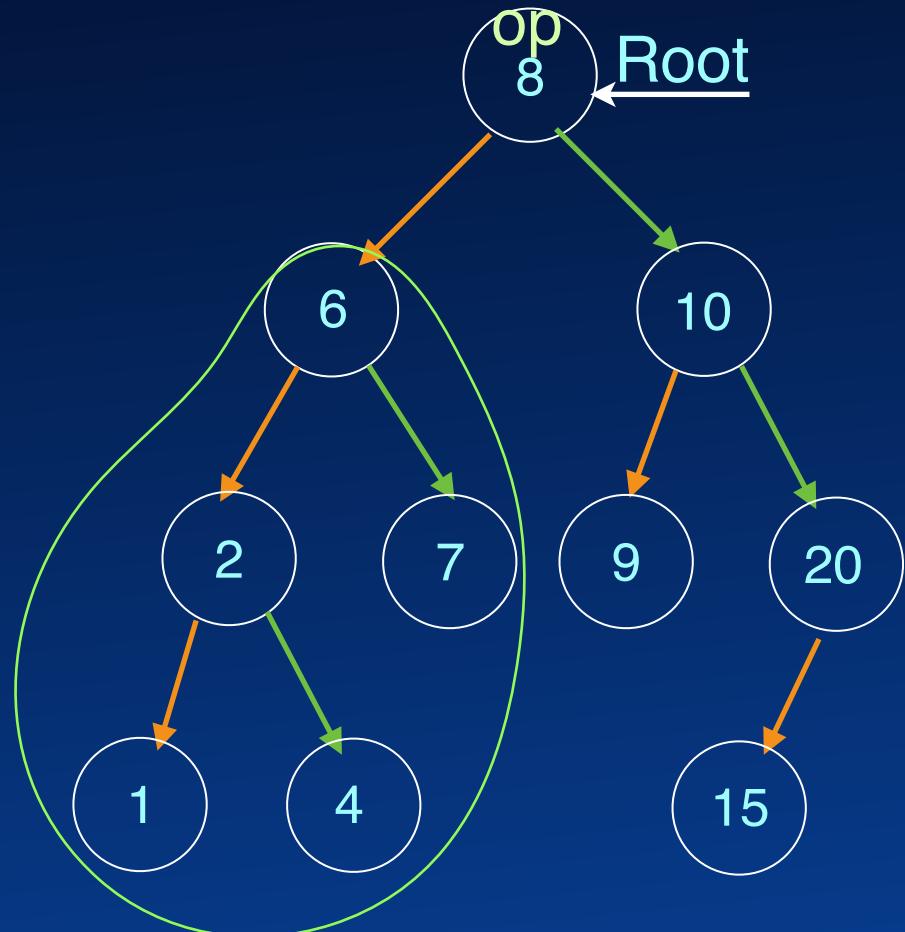




Traversal

```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```

In-order Traversal

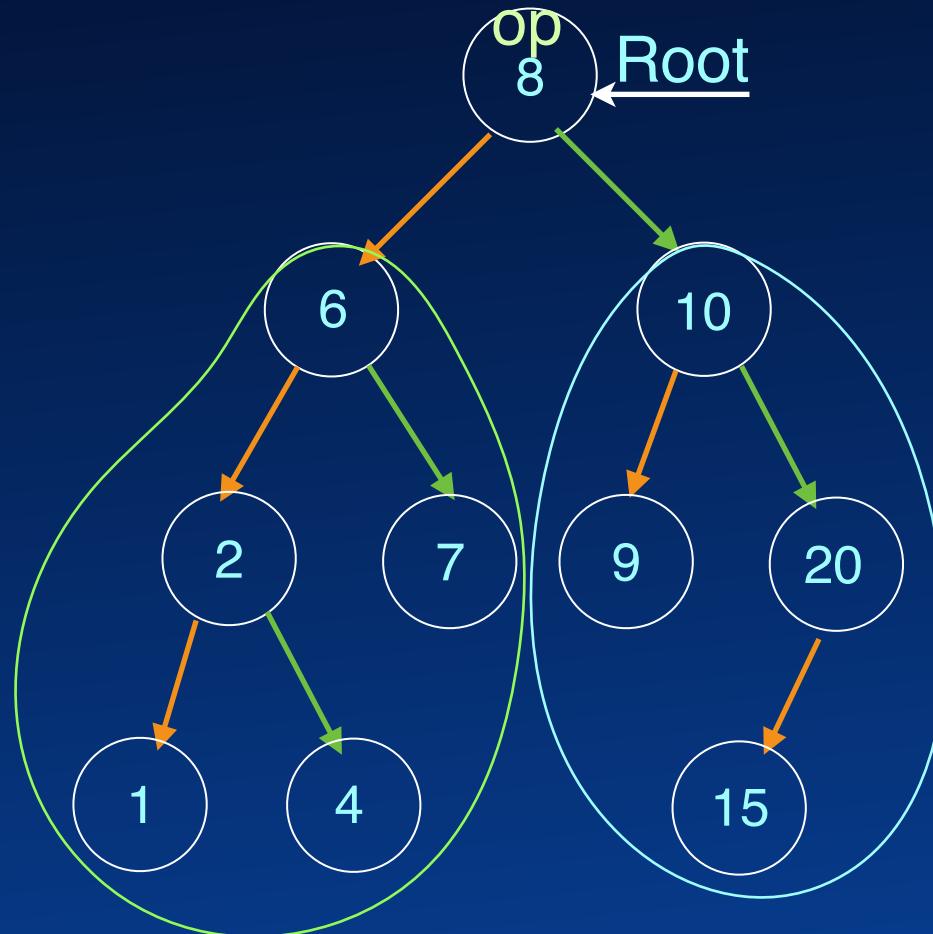




Traversal

```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```

In-order Traversal

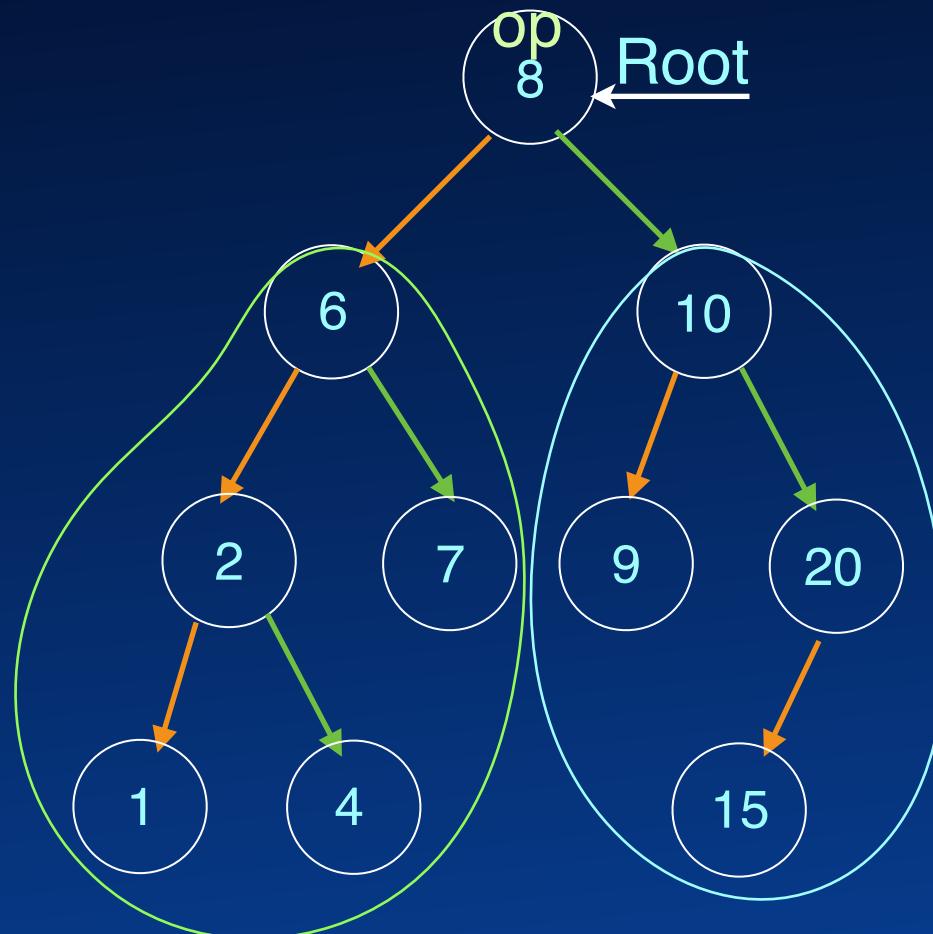




Traversal

```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```

In-order Traversal

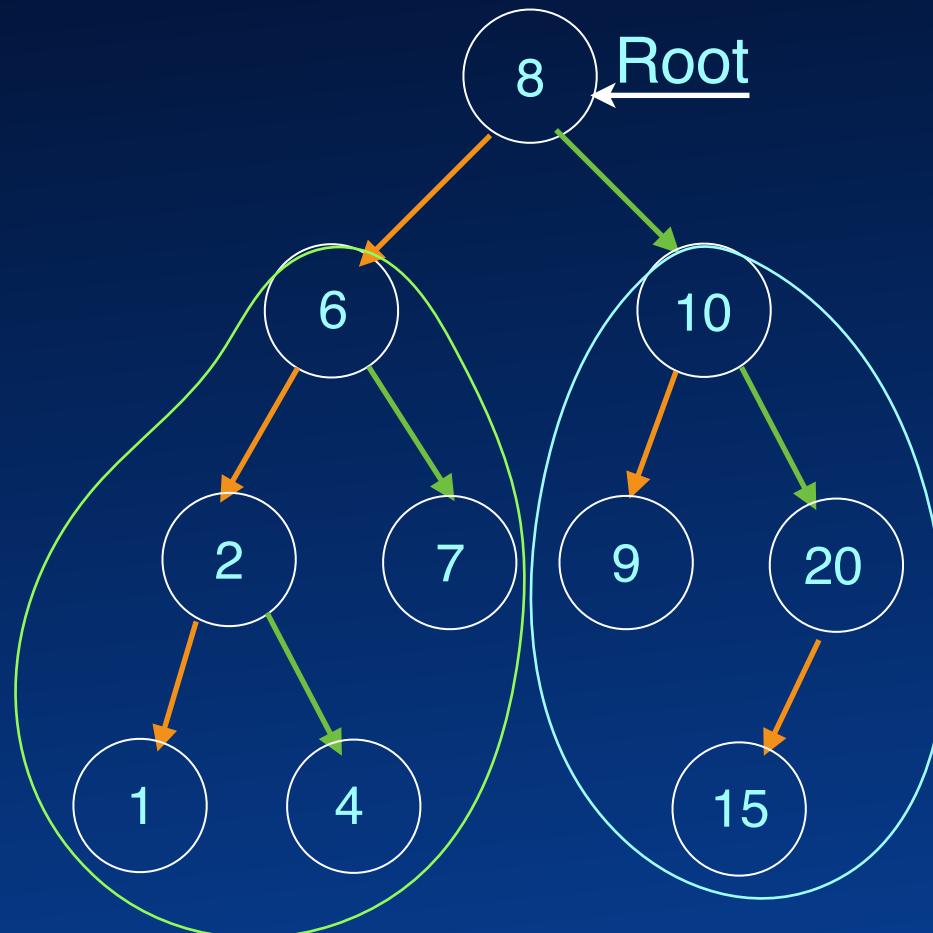




Traversal

```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```

In-order Traversal

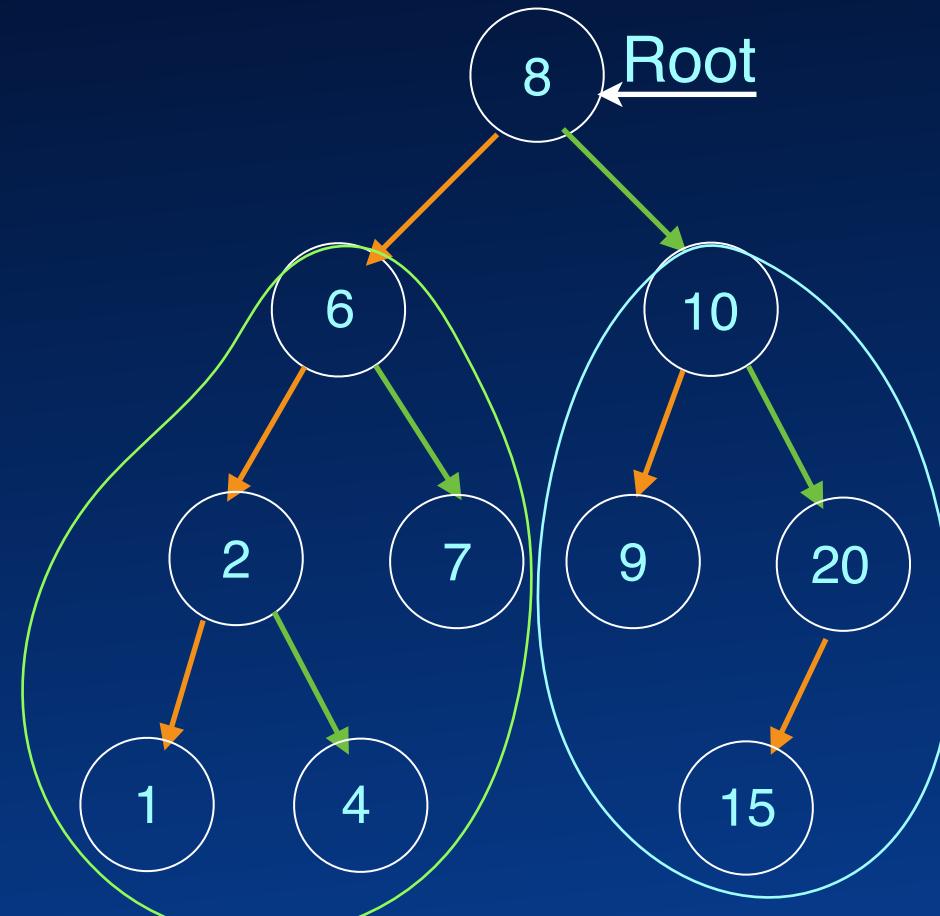




Traversal

```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```

In-order Traversal

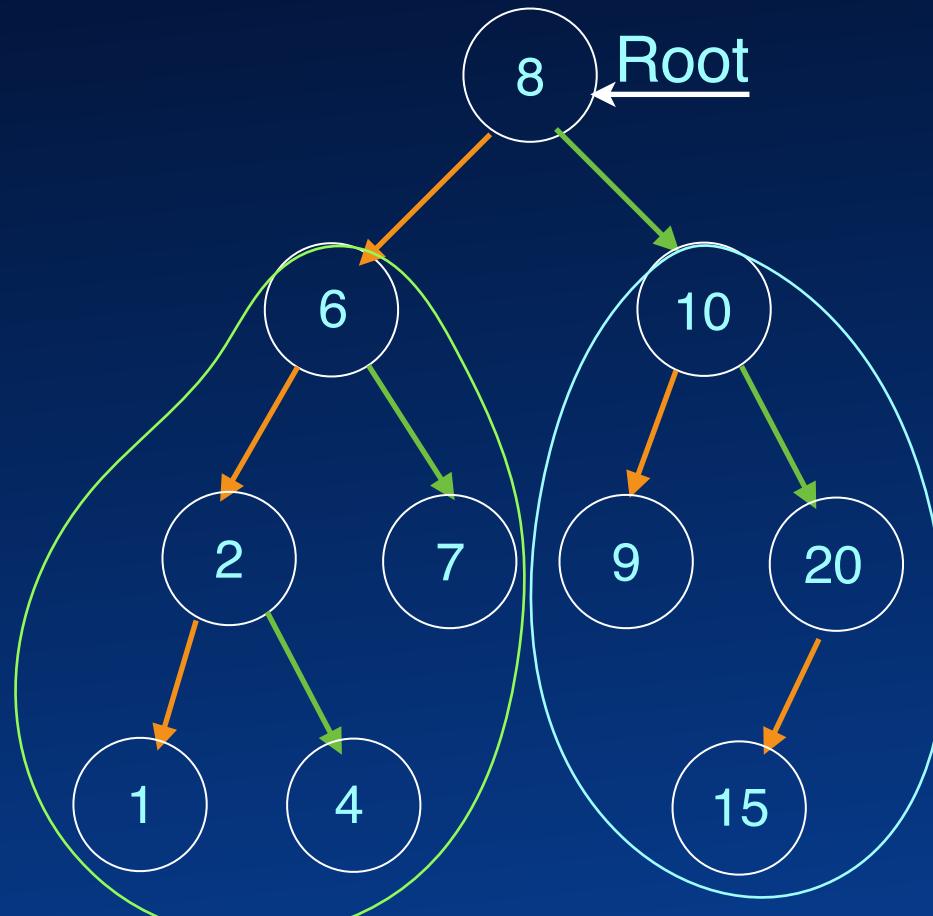
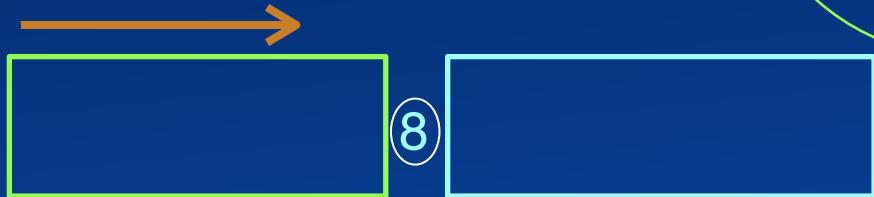




Traversal

```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
               Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```

In-order Traversal

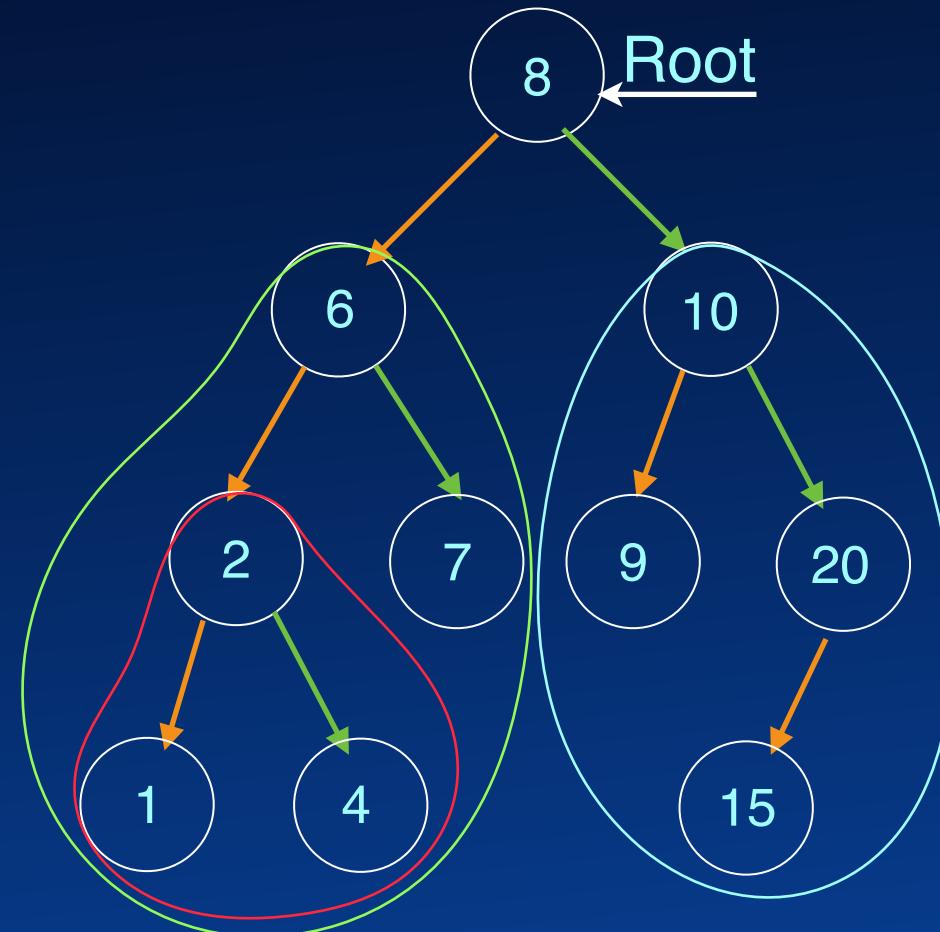




Traversal

```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
               Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```

In-order Traversal

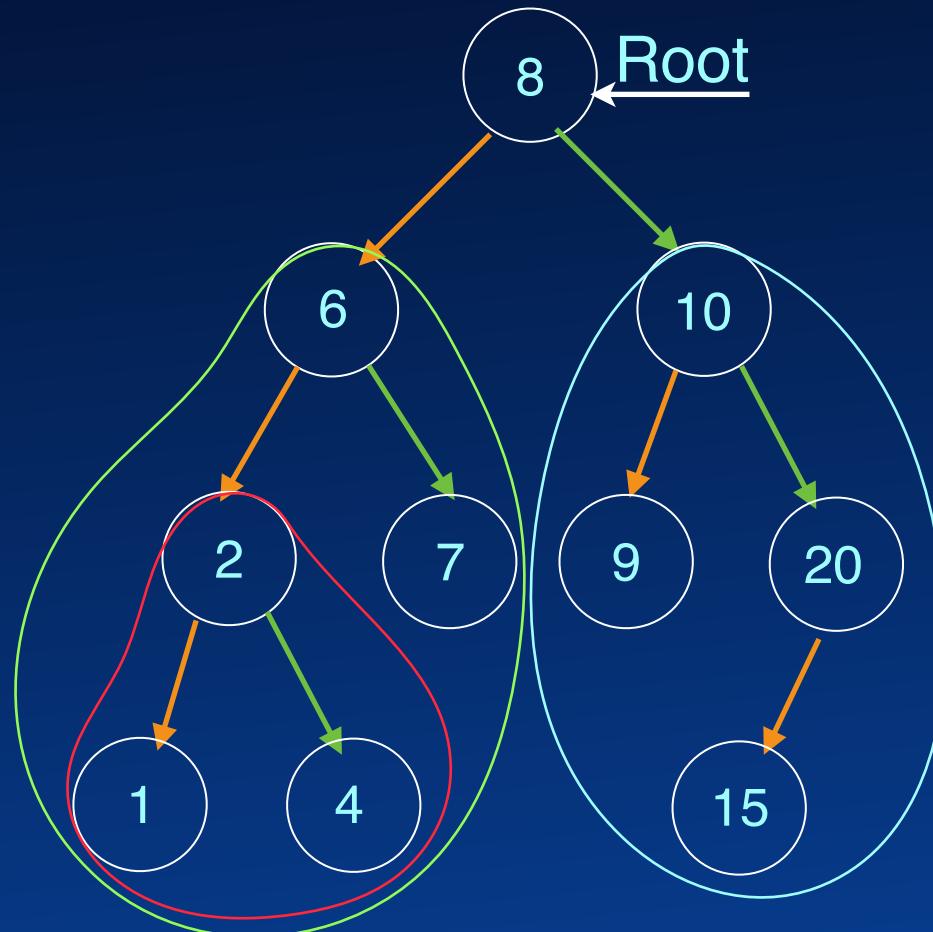




Traversal

```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```

In-order Traversal

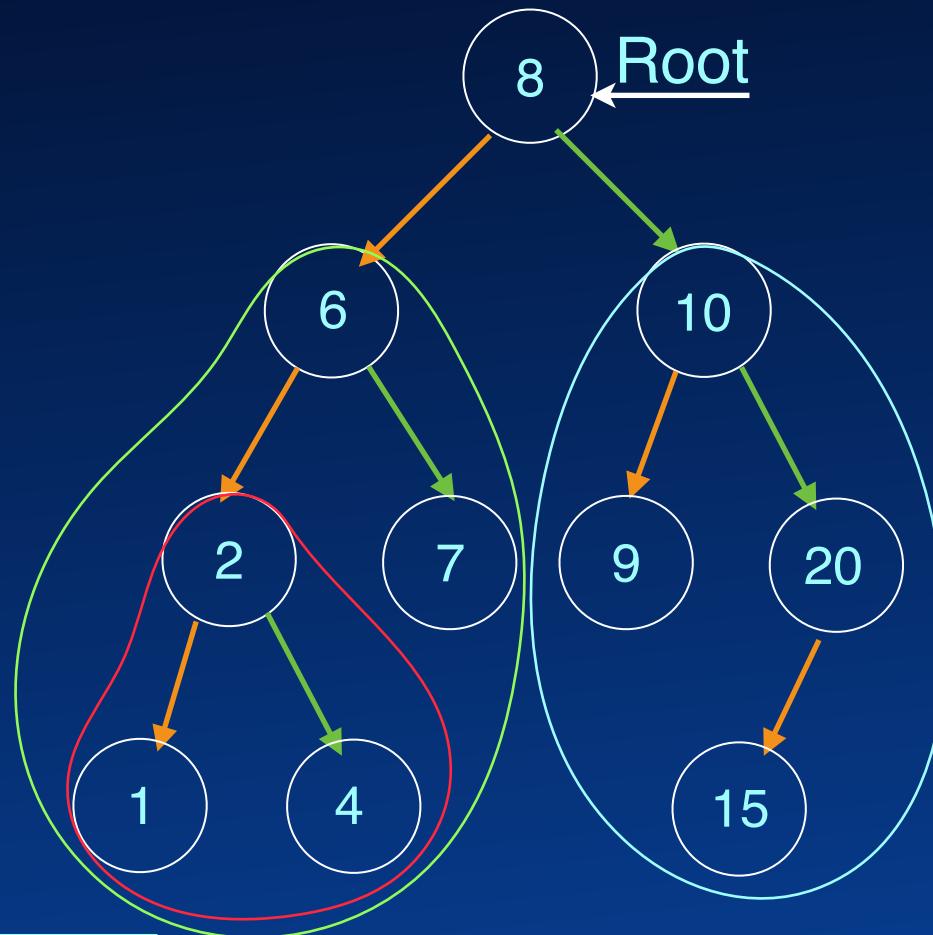




Traversal

```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```

In-order Traversal

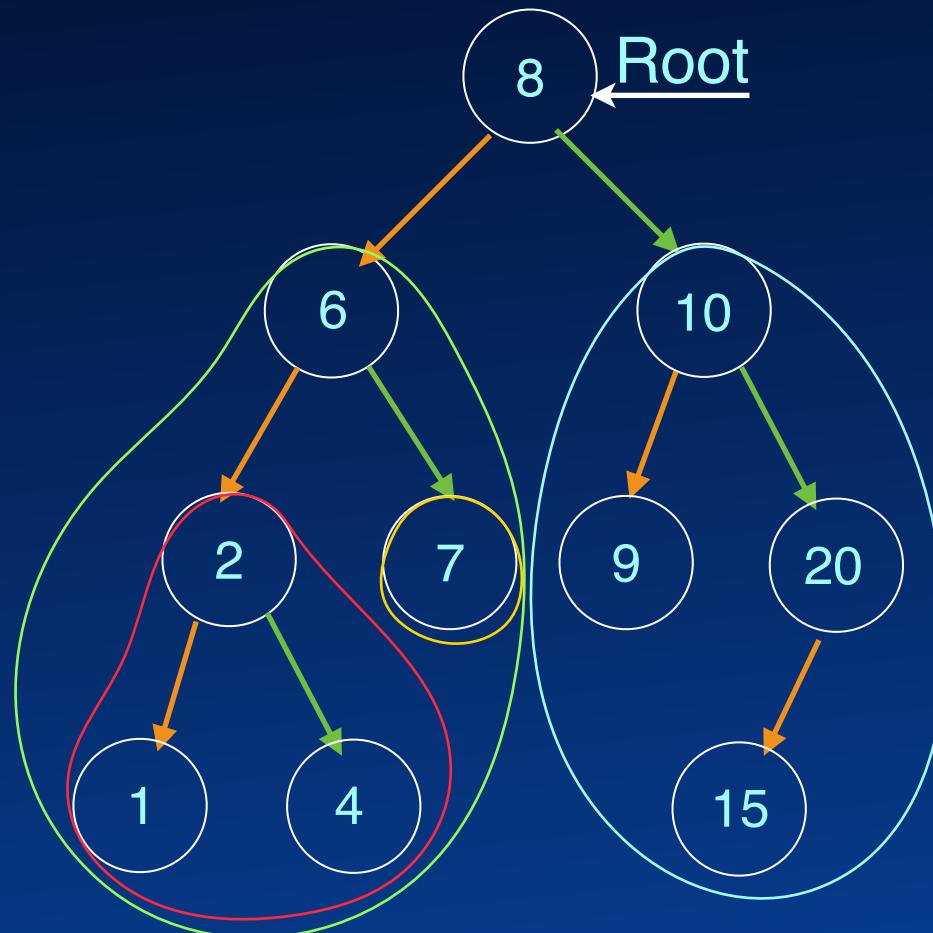
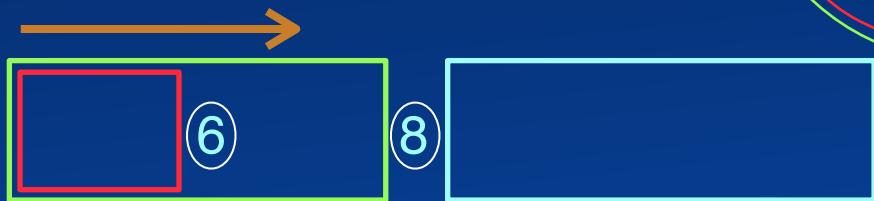




Traversal

```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
               Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```

In-order Traversal

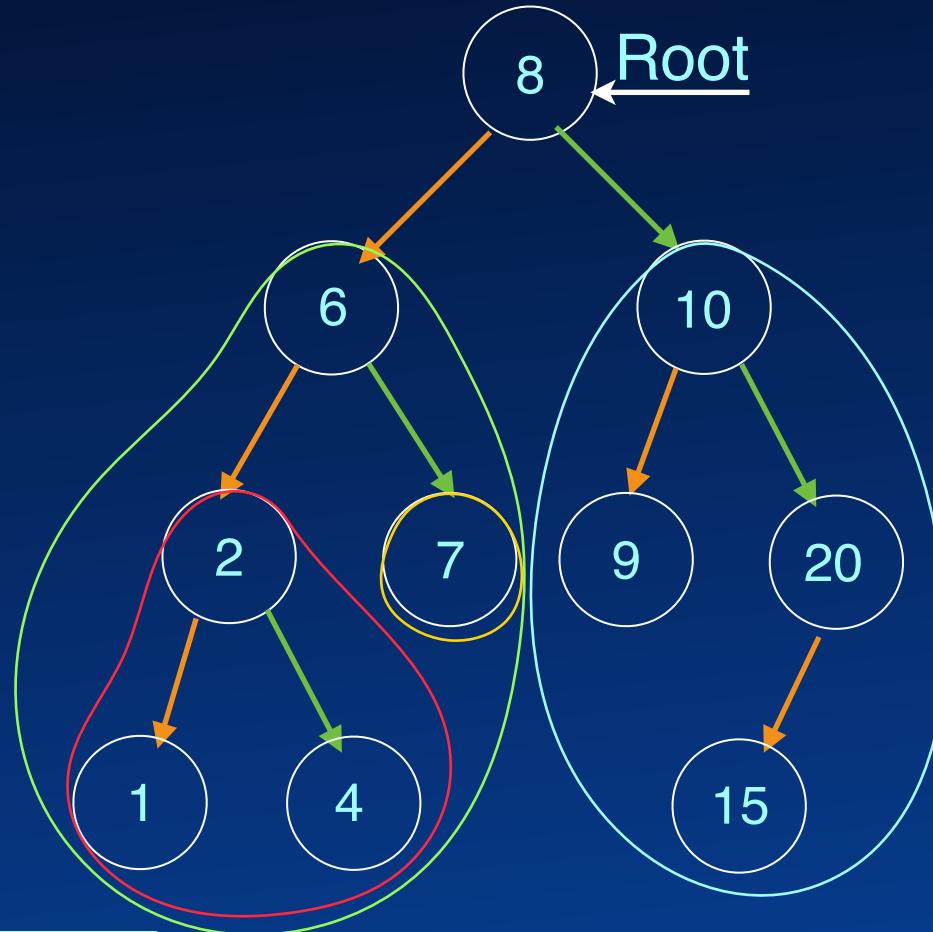
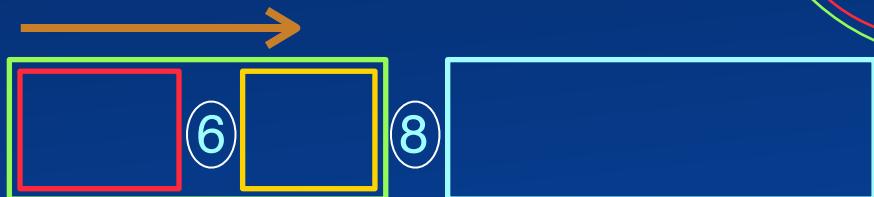




Traversal

```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```

In-order Traversal

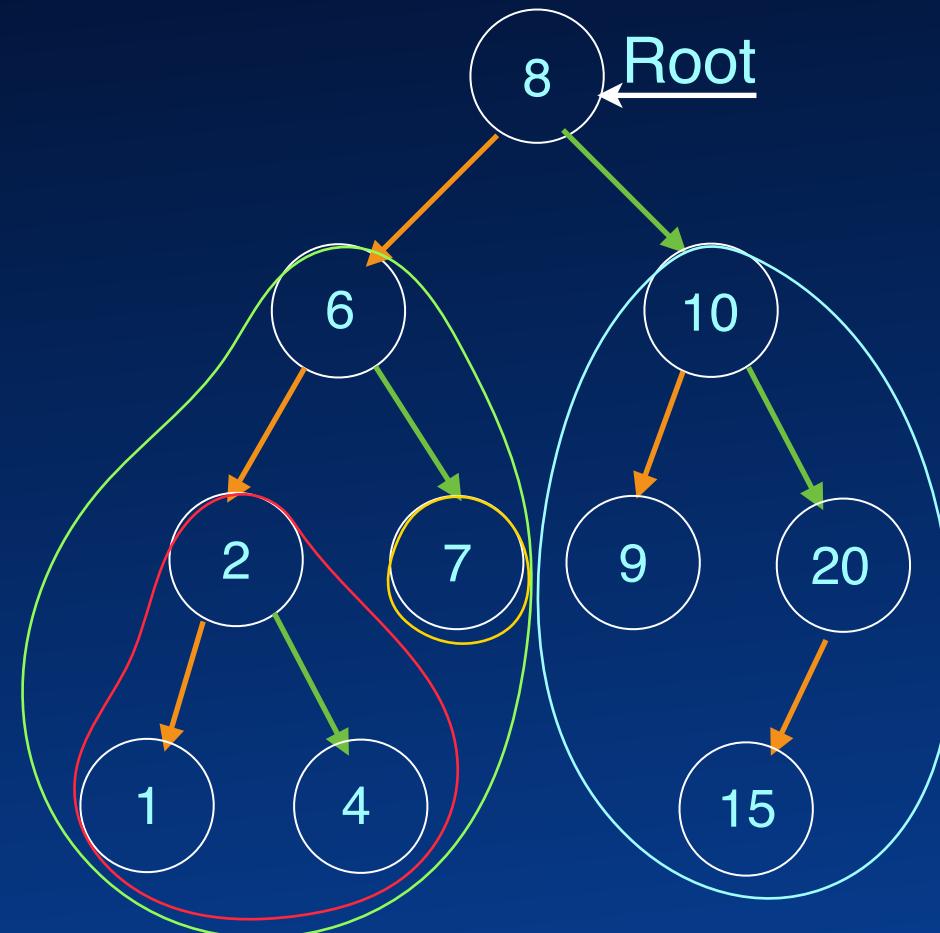




Traversal

```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```

In-order Traversal

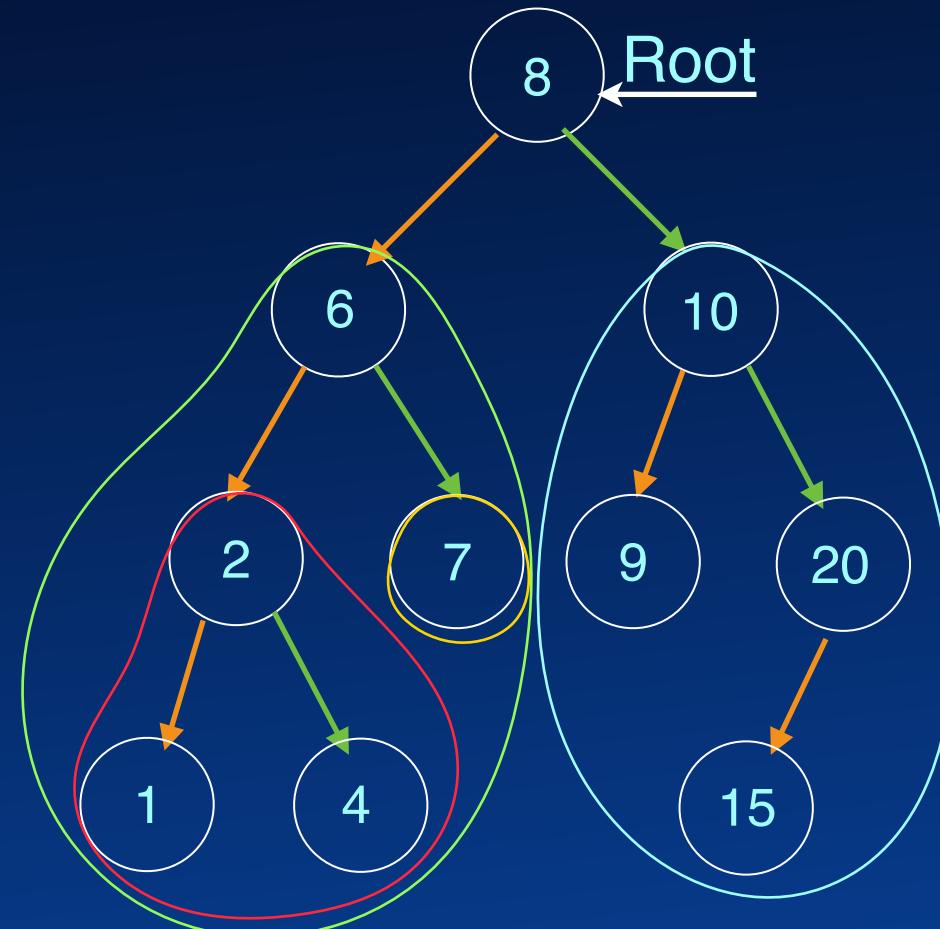
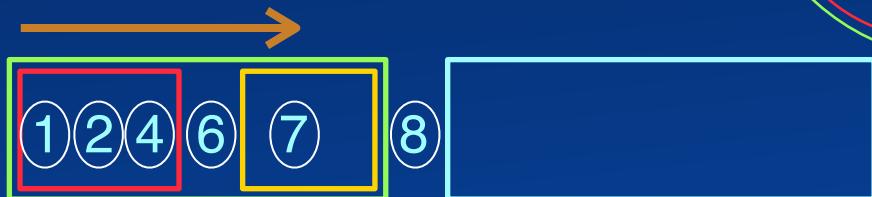




Traversal

```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```

In-order Traversal

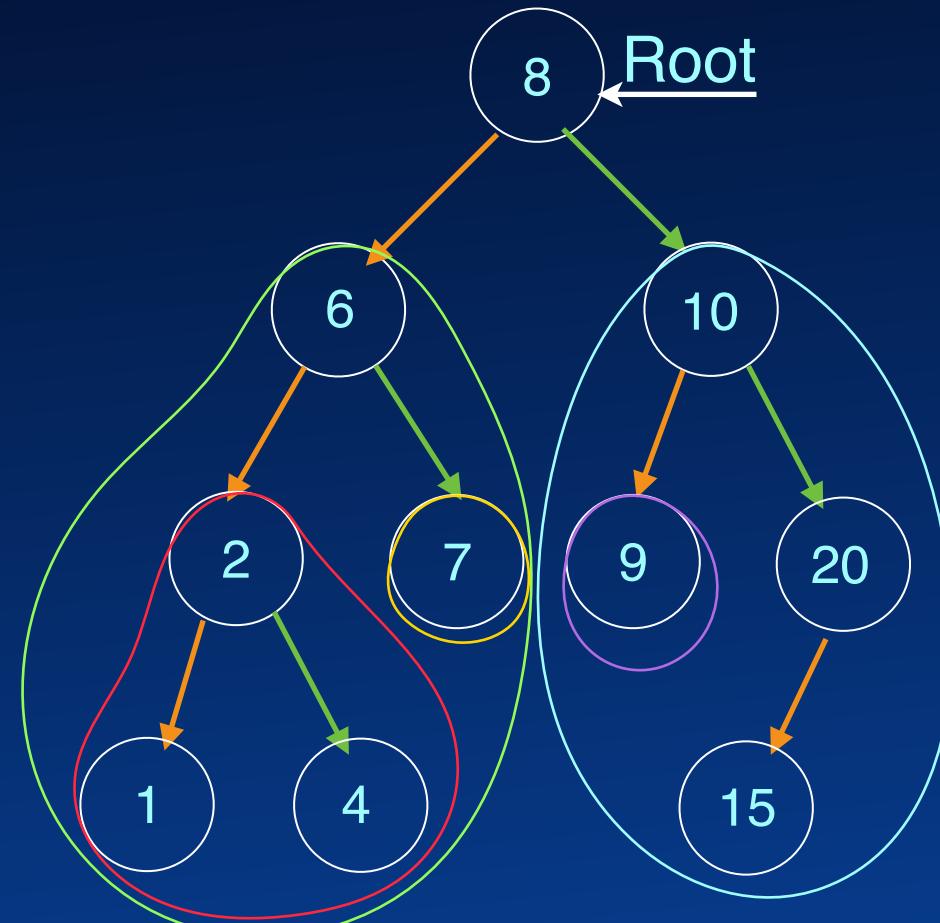
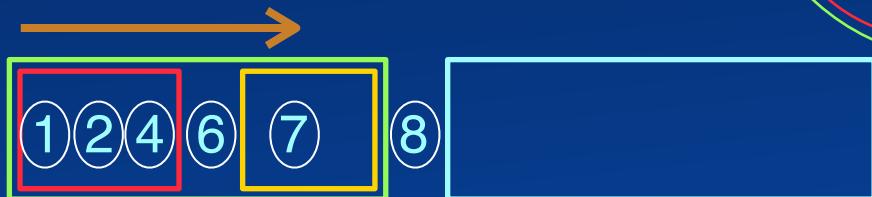




Traversal

```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
               Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```

In-order Traversal

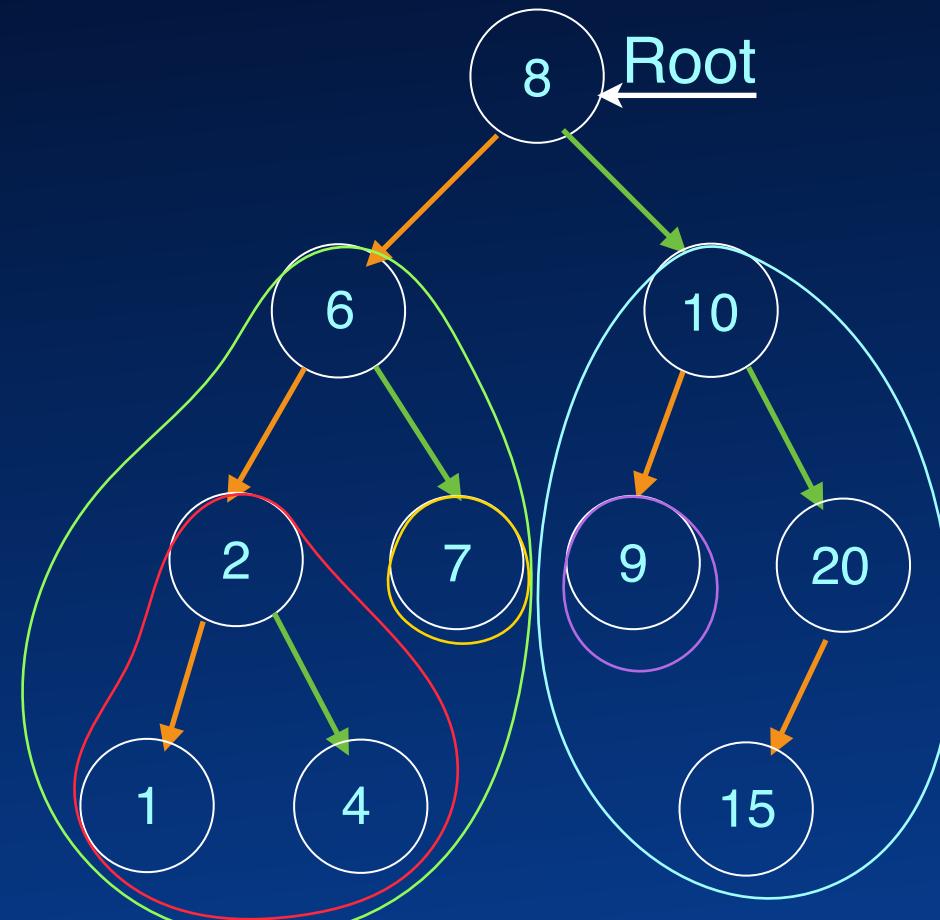
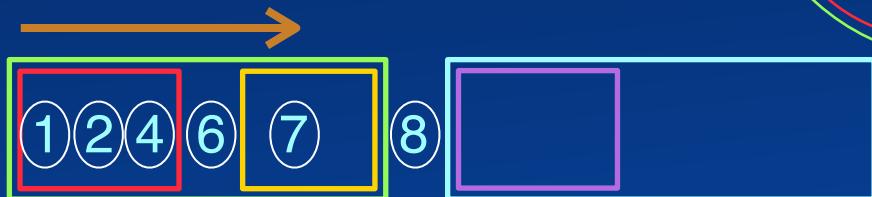




Traversal

```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```

In-order Traversal

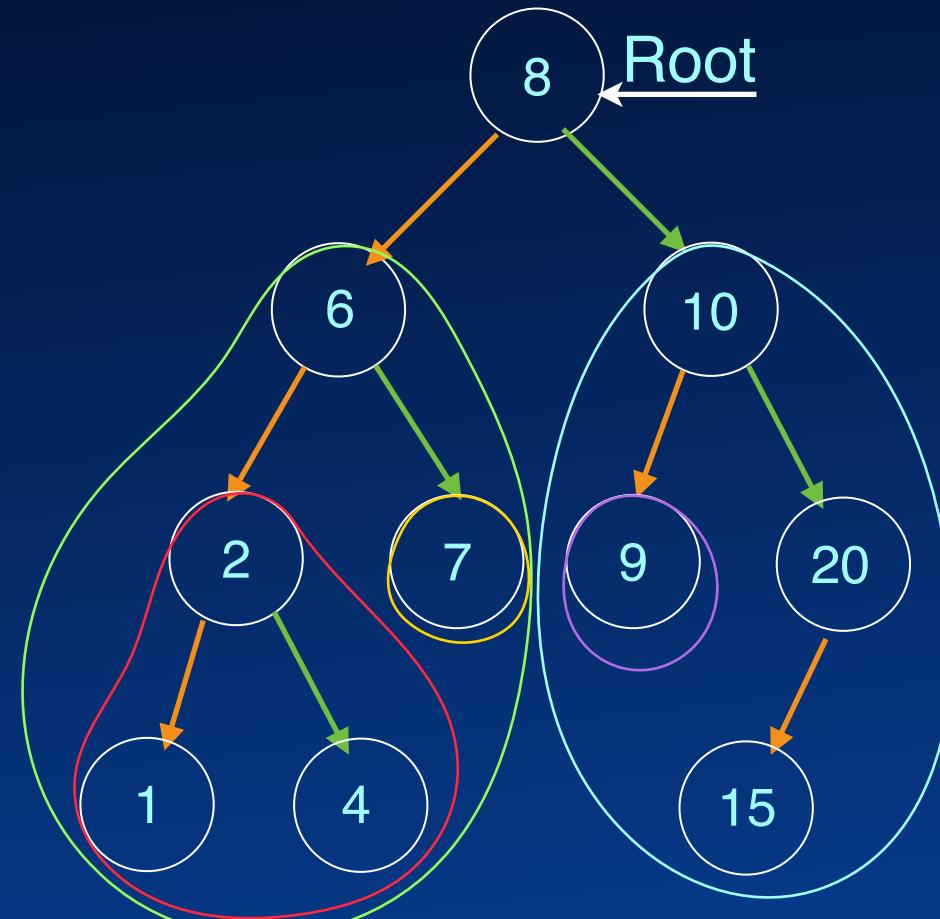
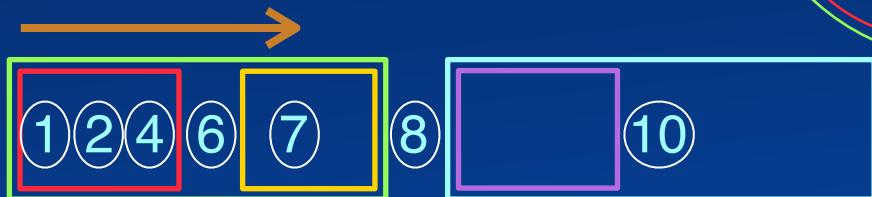




Traversal

```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
               Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```

In-order Traversal

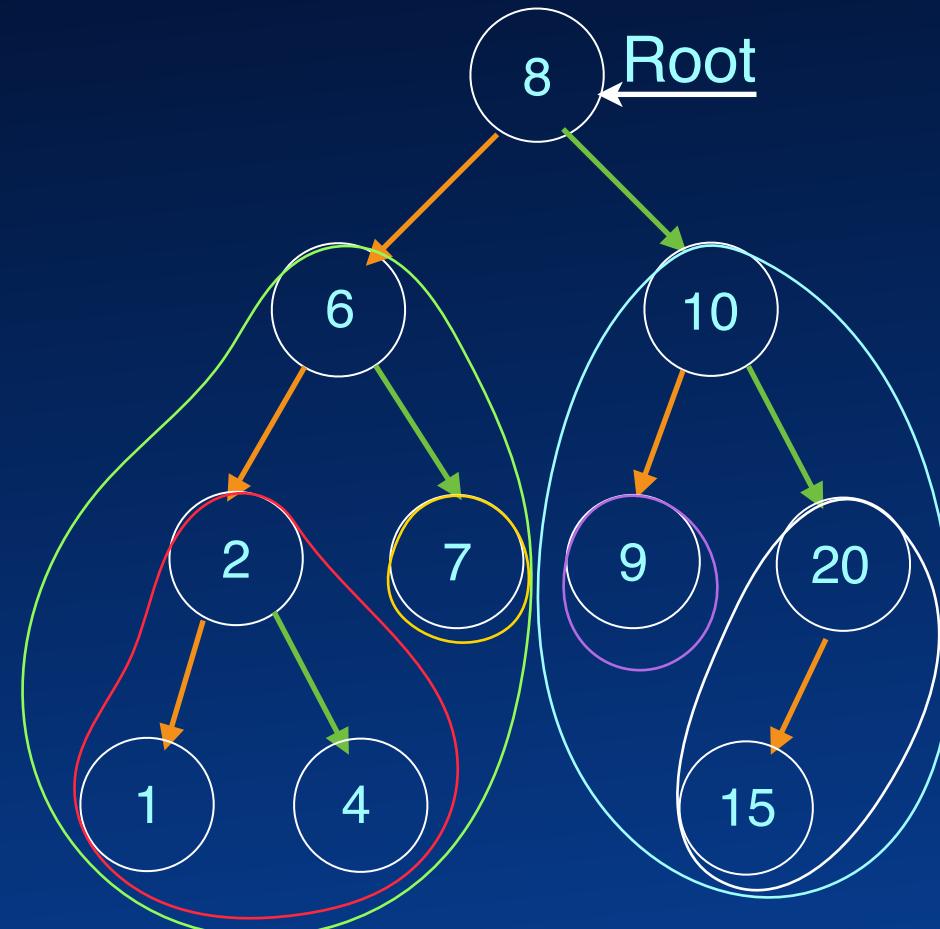
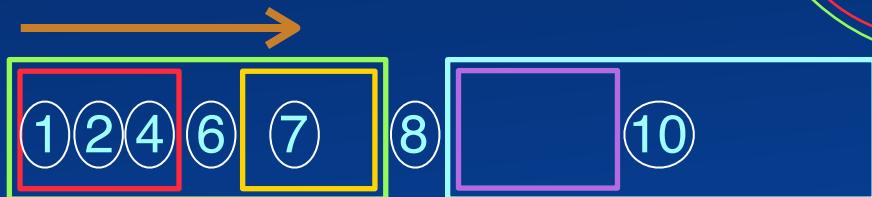




Traversal

```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```

In-order Traversal

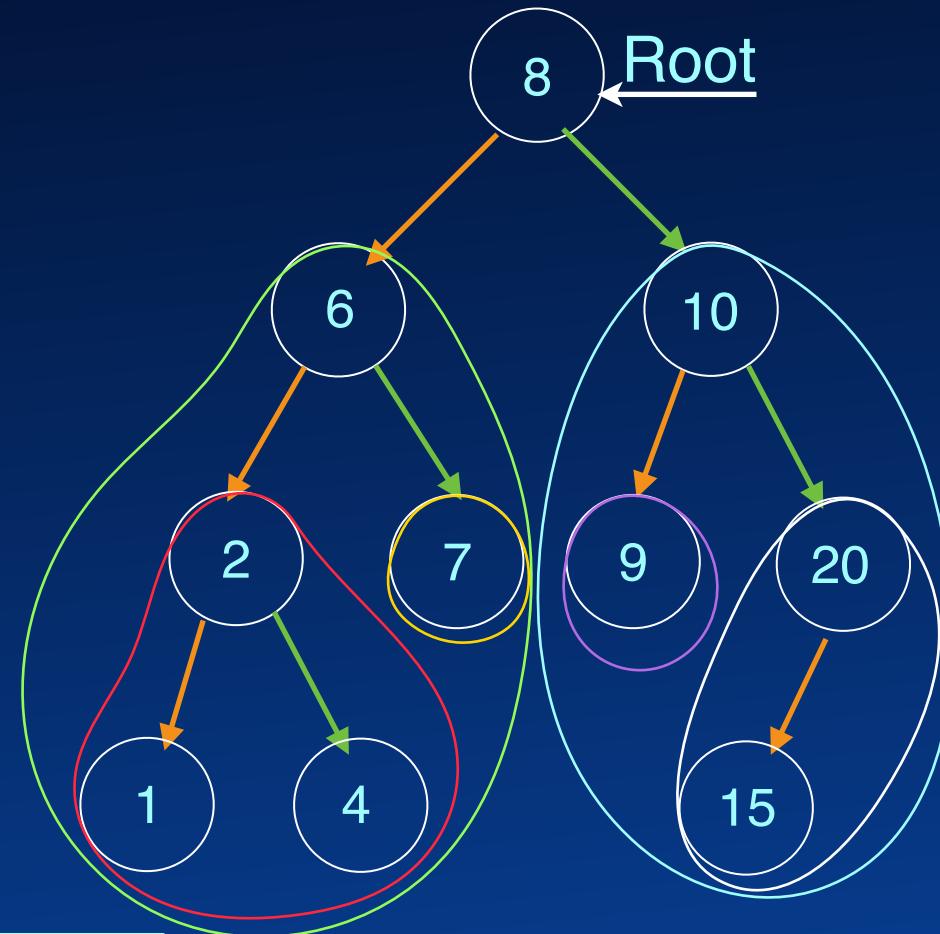
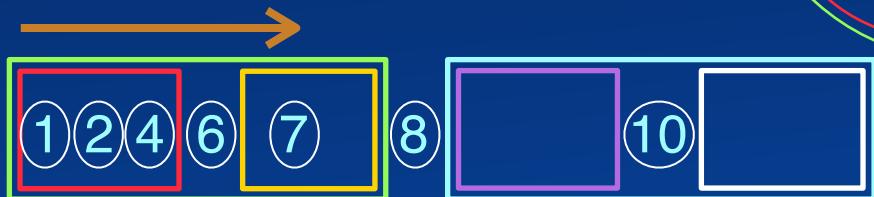




Traversal

```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```

In-order Traversal

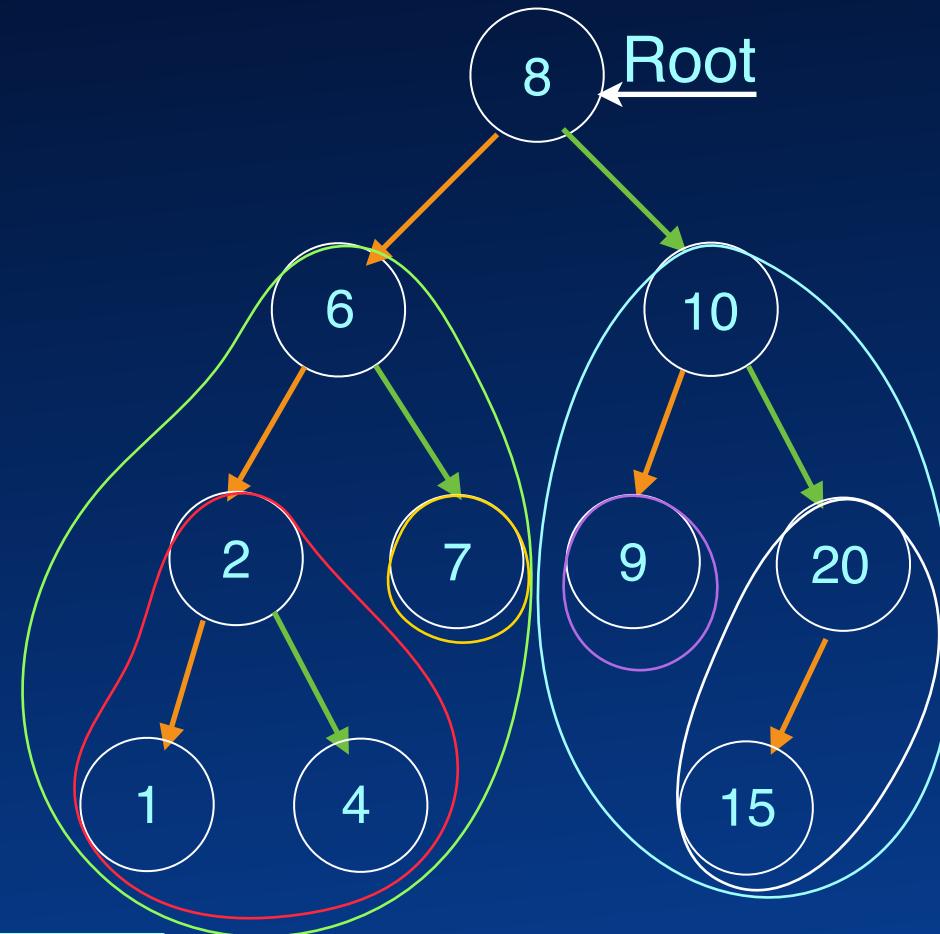




Traversal

```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
               Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```

In-order Traversal

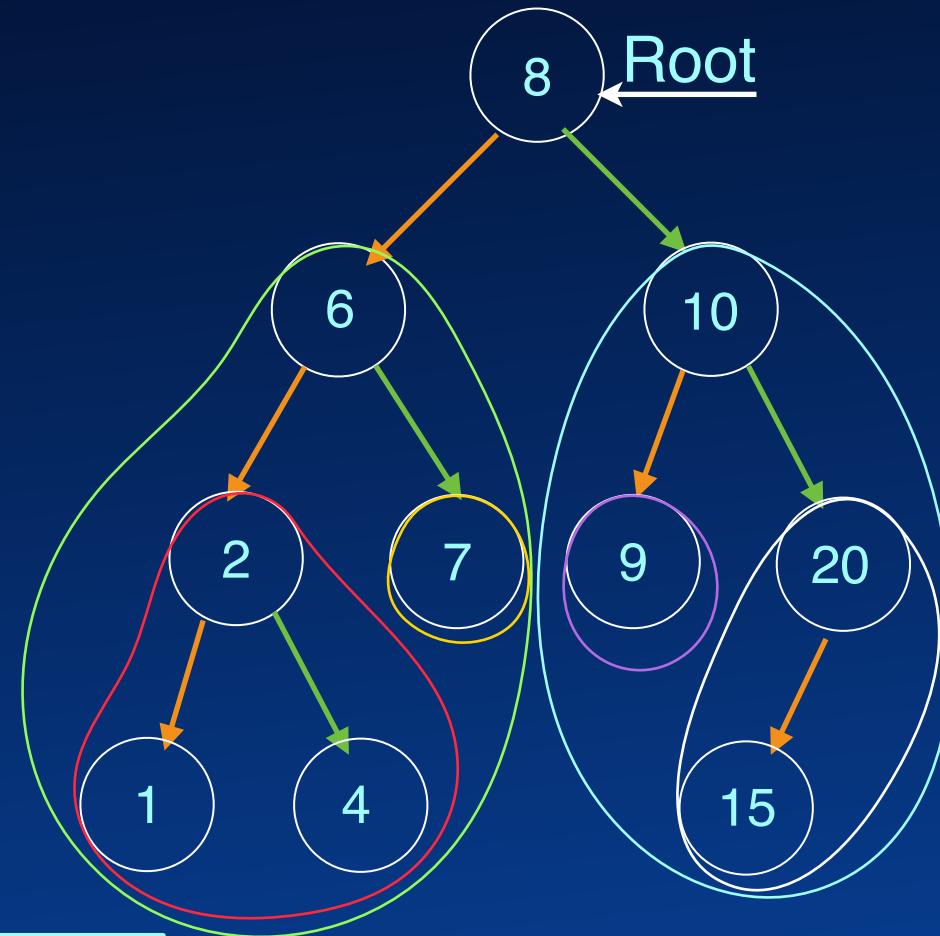




Traversal

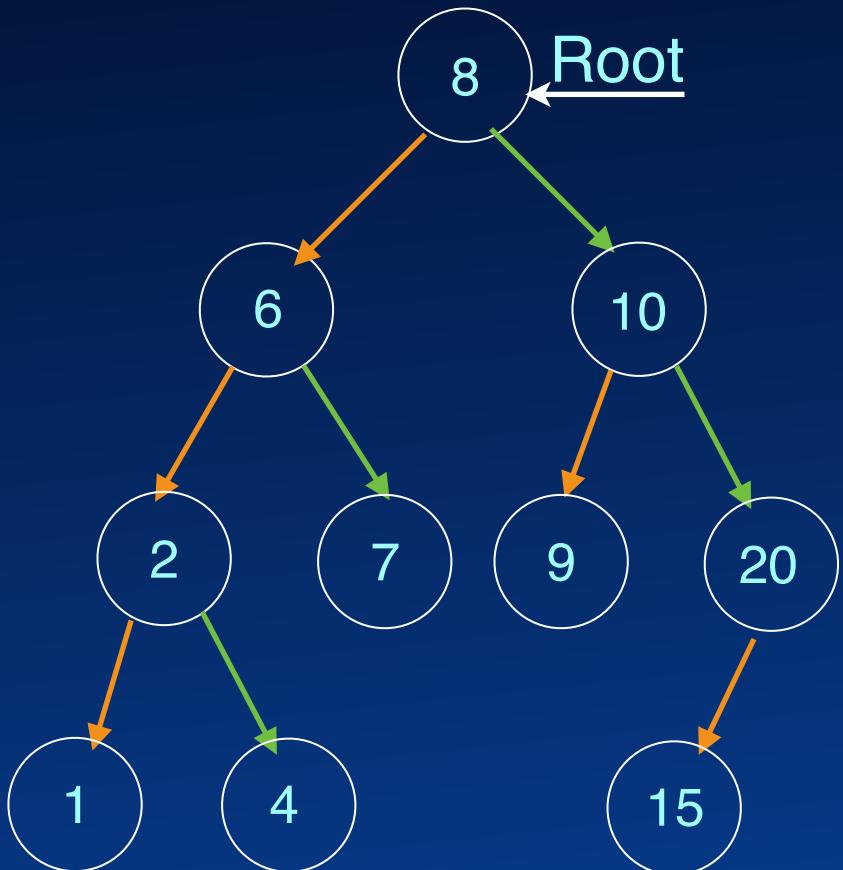
```
class BST<PT extends SortablePair> {  
    Position2way<PT> root;  
    int numPositions;  
    void iterate(Consumer<PT> op) {  
        iterate(root, op);  
    }  
    void iterate(Position2way<PT> node,  
                Consumer<PT> op) {  
        if(node == null) return;  
        iterate(node.left(), op);  
        op.accept(node.value());  
        iterate(node.right(), op);  
    }  
    ...  
}
```

In-order Traversal



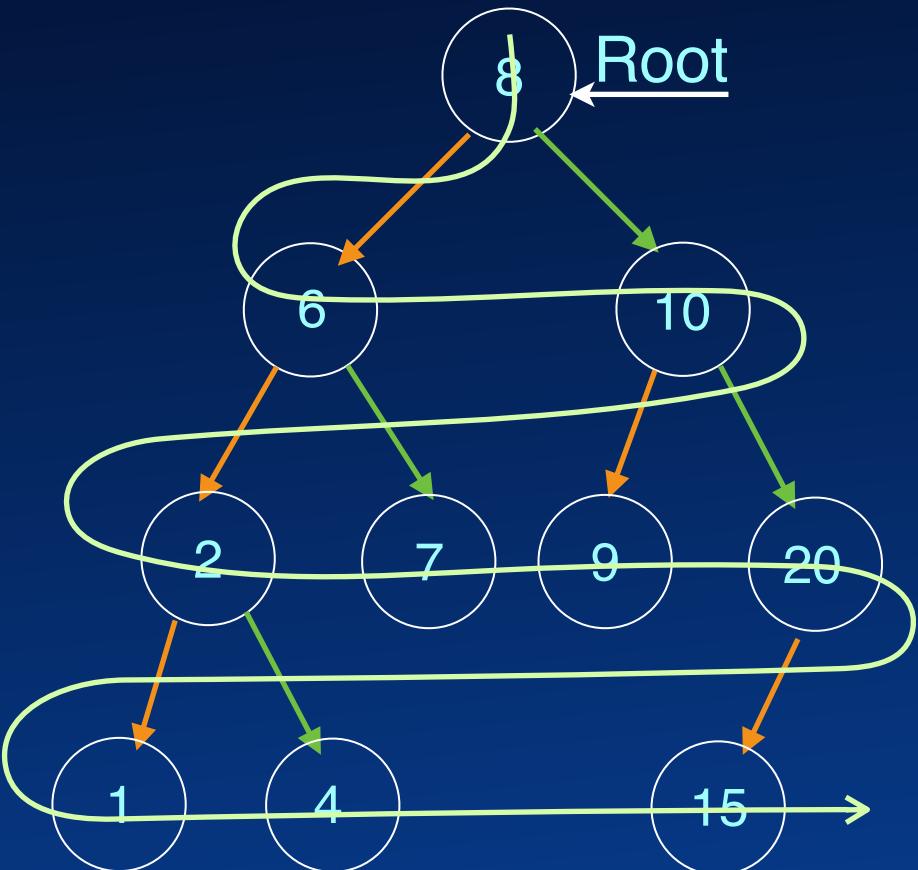


Breadth-first Traversal



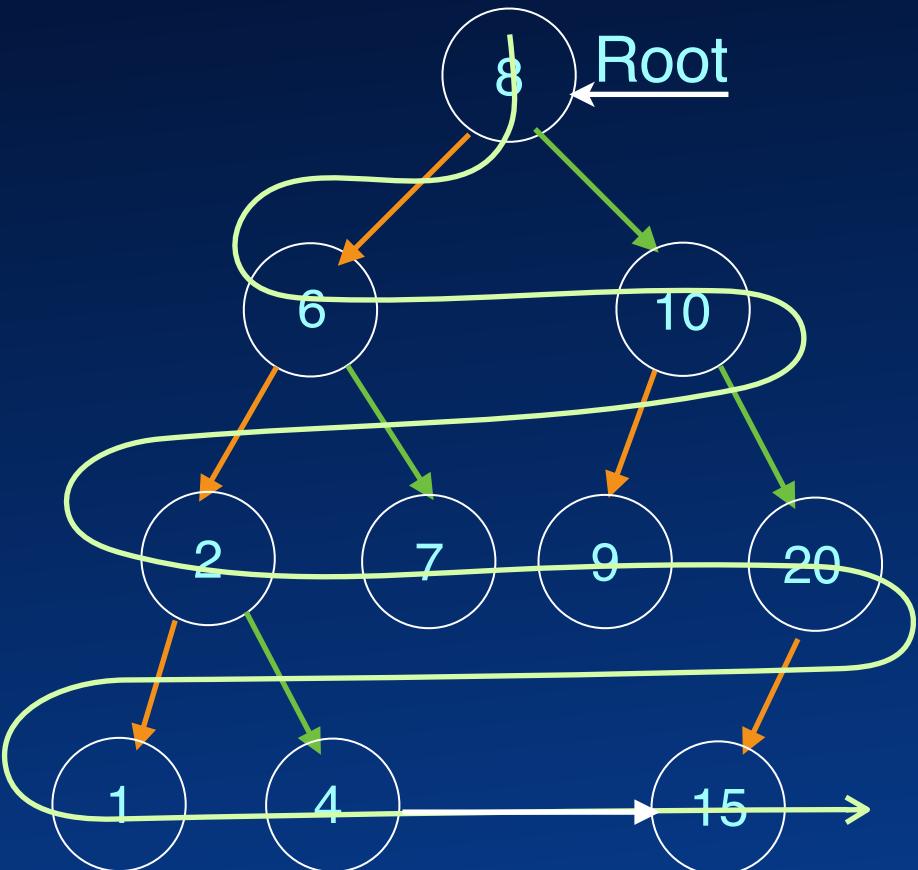


Breadth-first Traversal



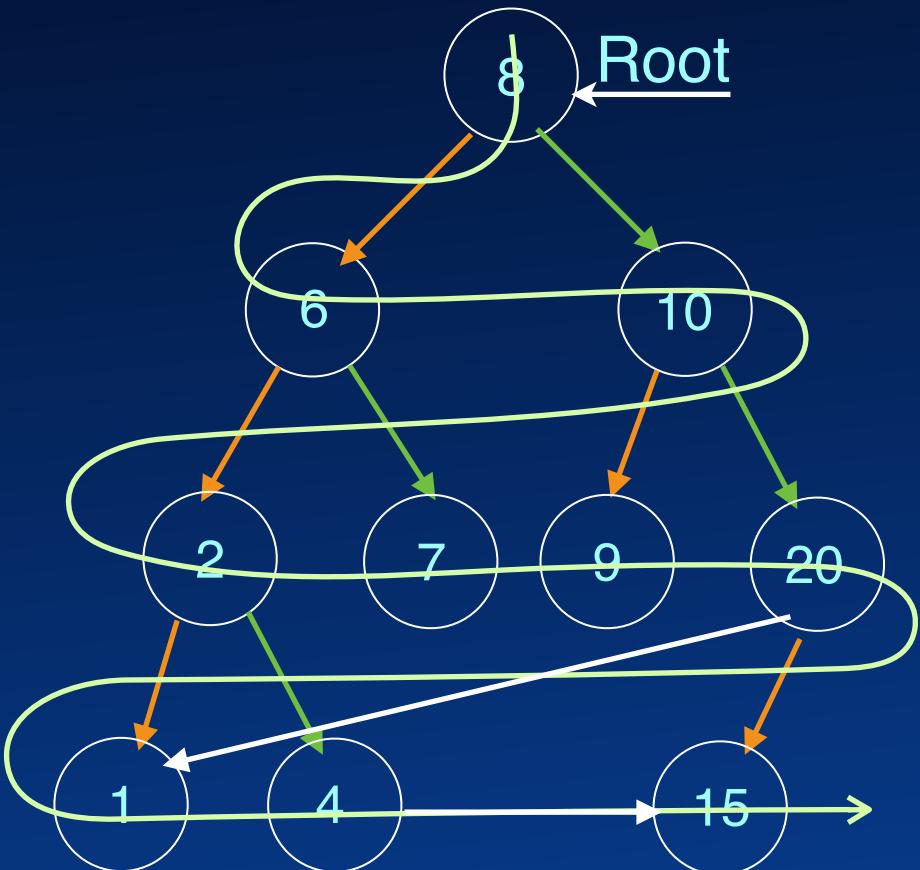


Breadth-first Traversal



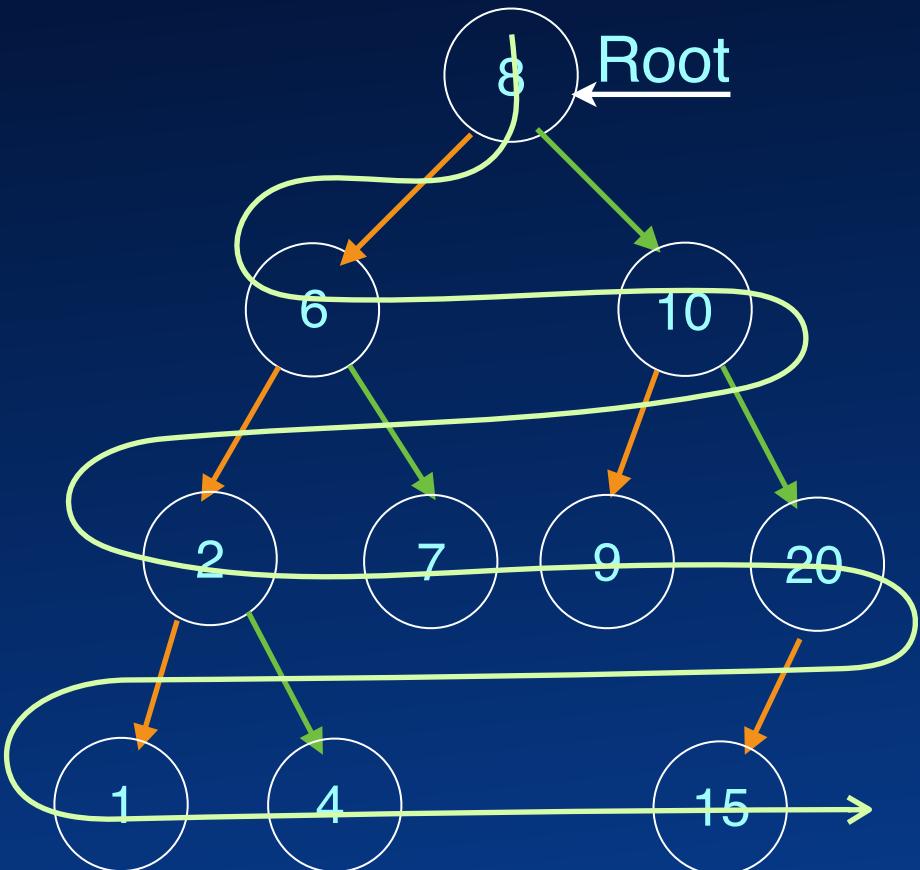


Breadth-first Traversal





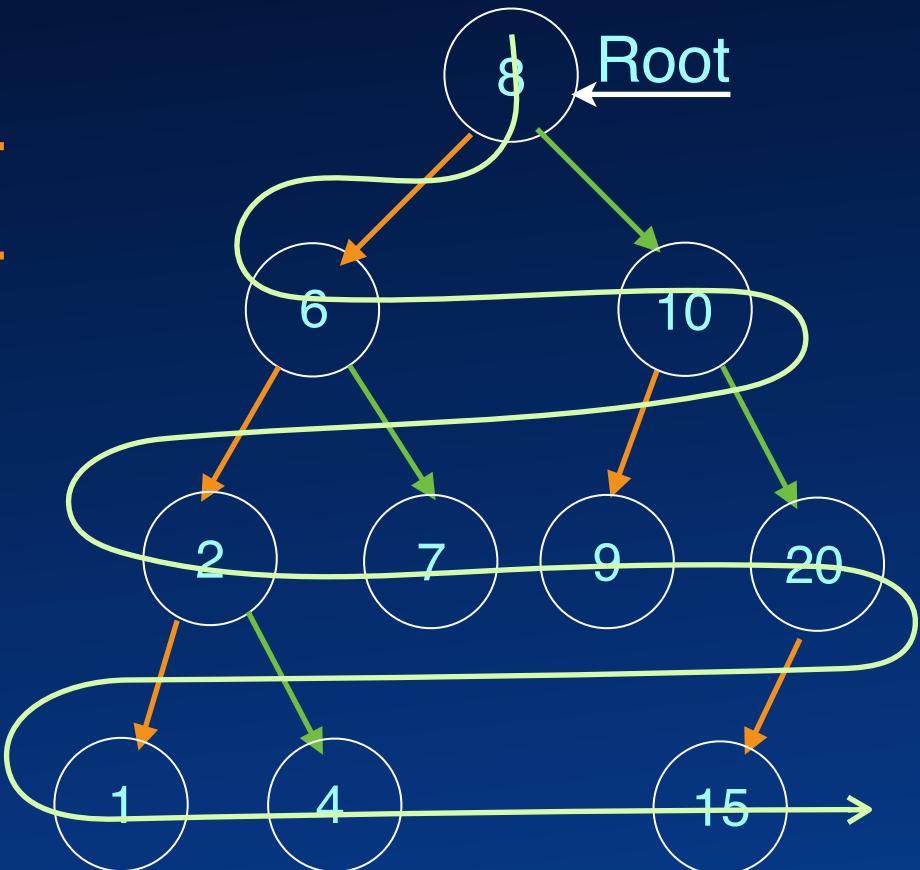
Breadth-first Traversal





Breadth-first Traversal

Queue

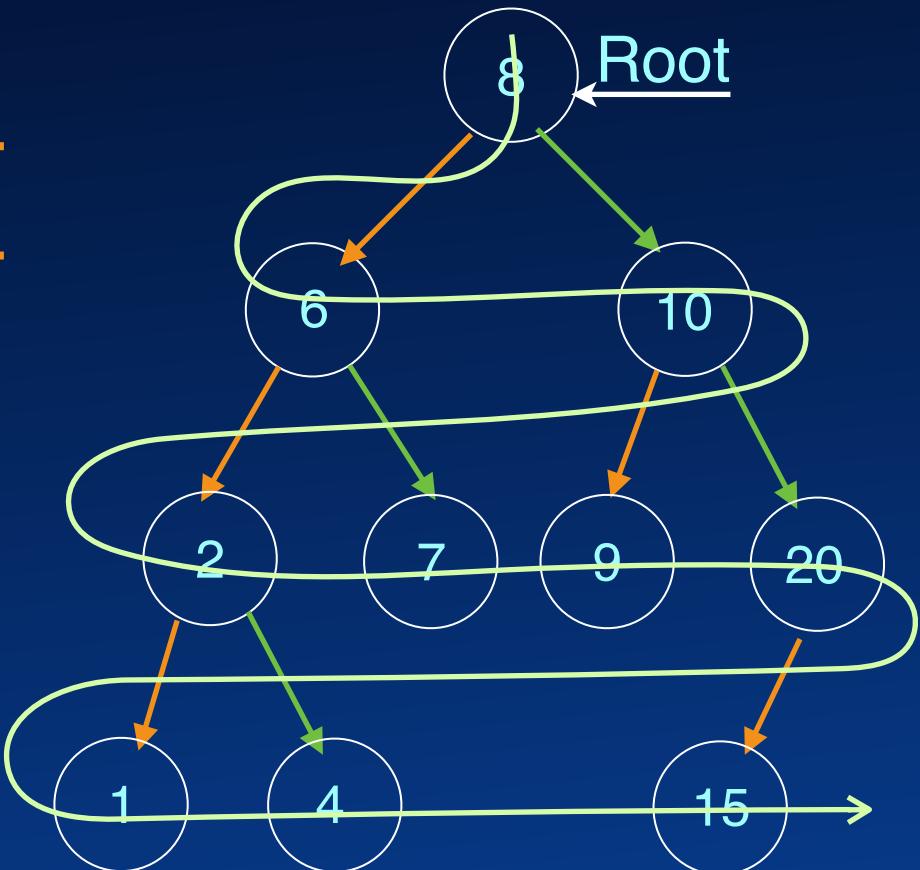




Breadth-first Traversal

Queue

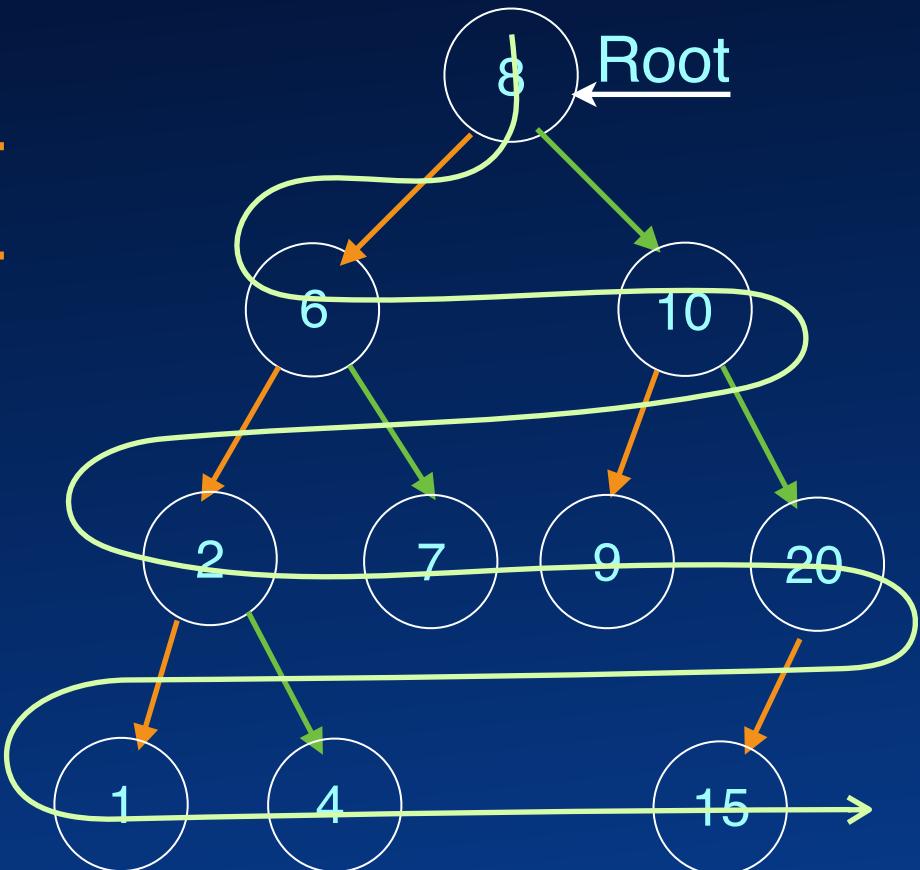
8





Breadth-first Traversal

Queue

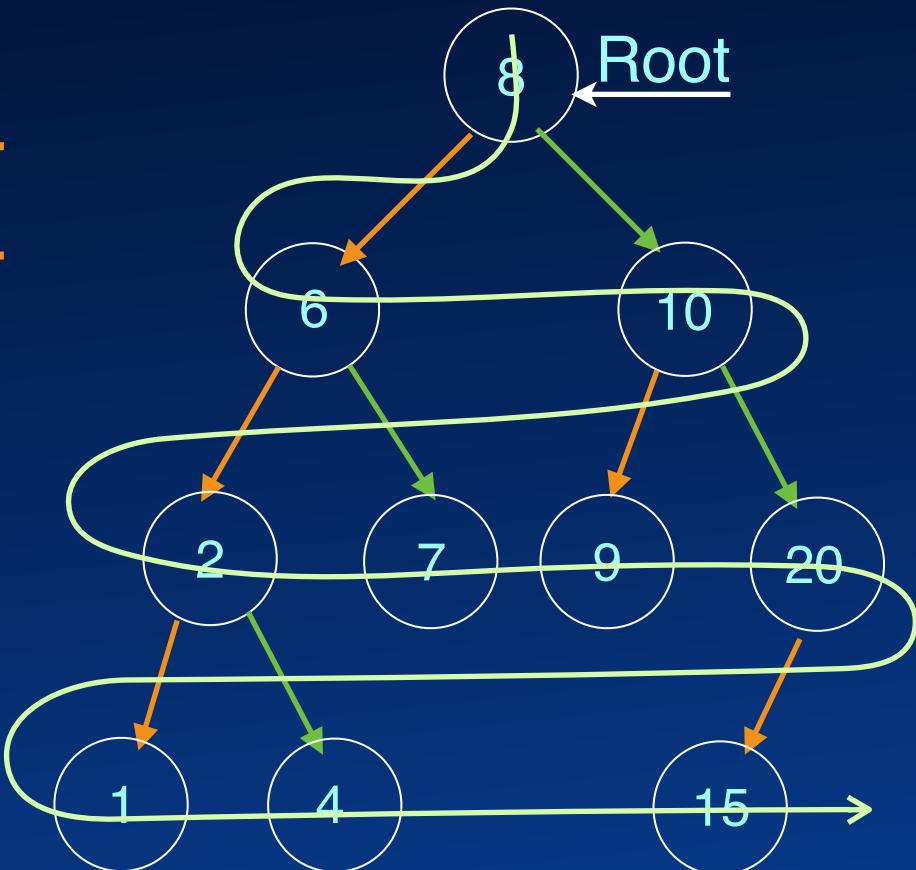




Breadth-first Traversal

Queue

6 10

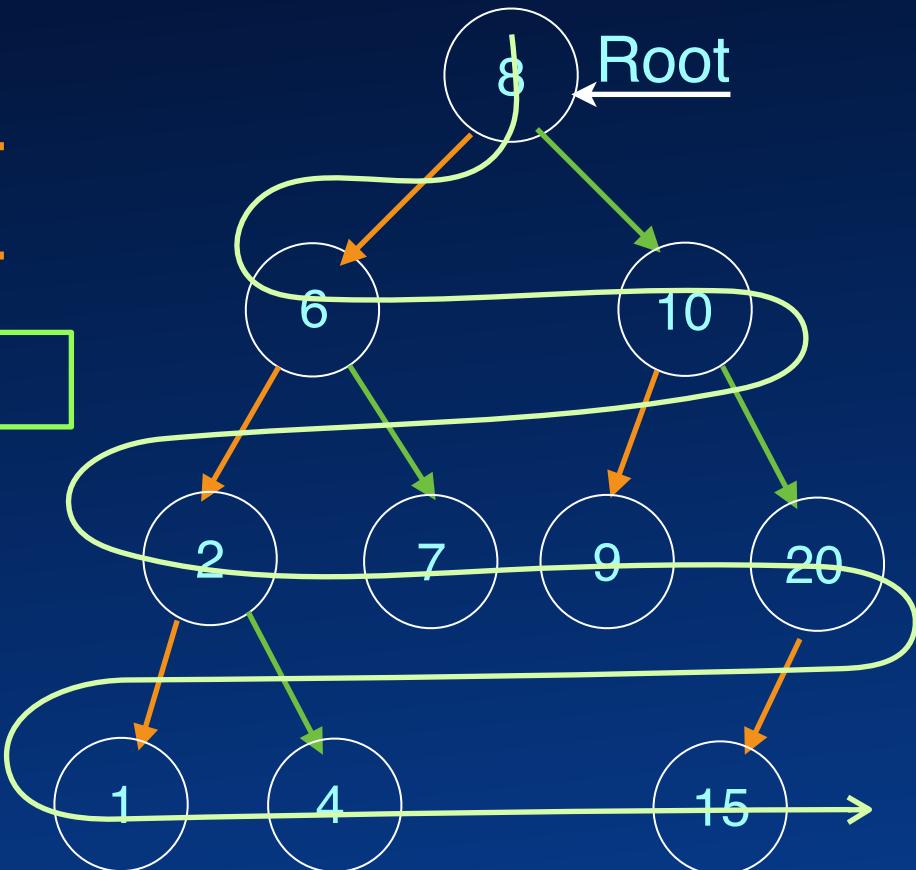




Breadth-first Traversal

Queue

6 10





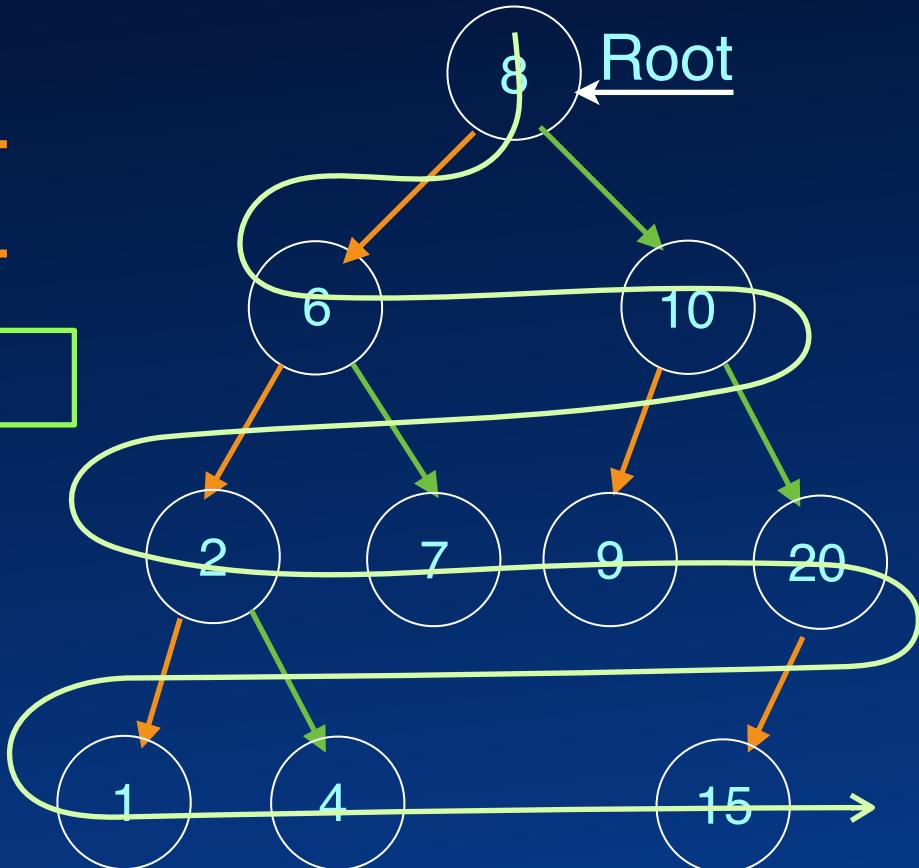
Breadth-first Traversal

Queue

6 10



8





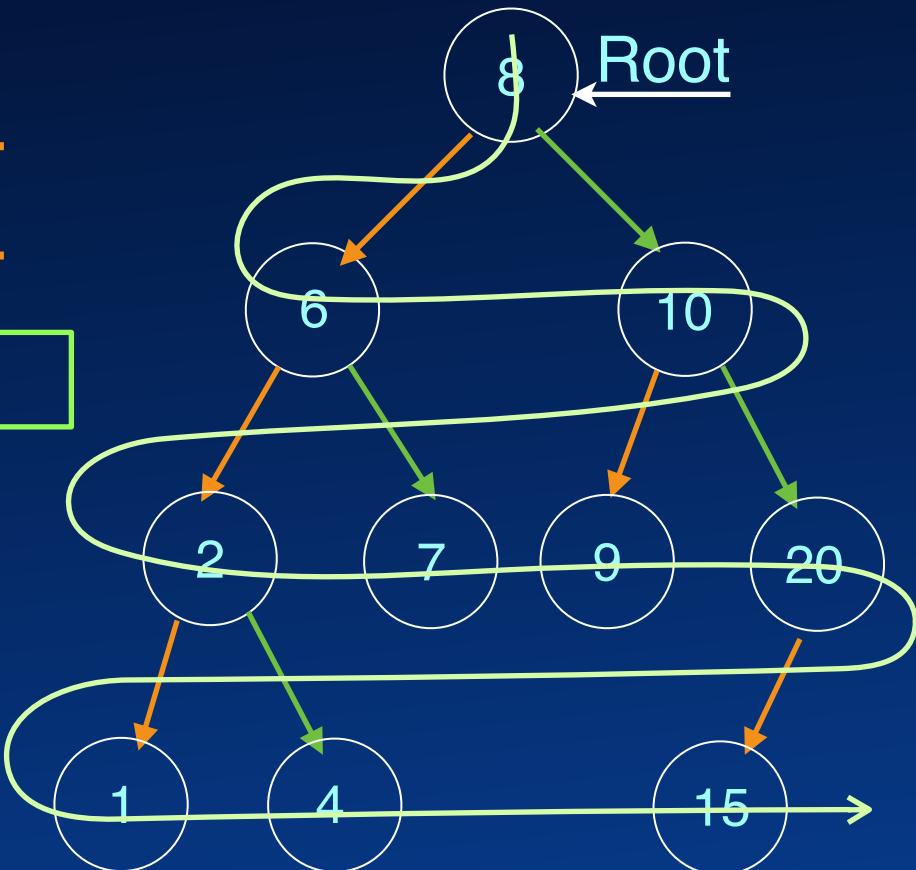
Breadth-first Traversal

Queue

10



8

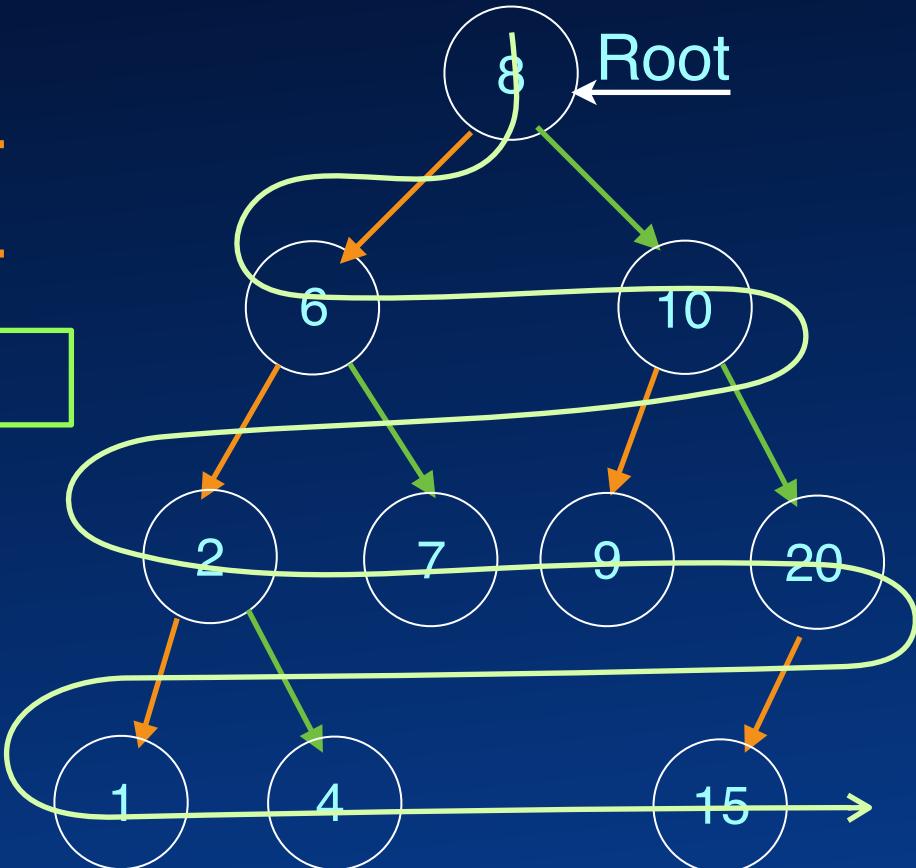




Breadth-first Traversal

Queue

10 2 7





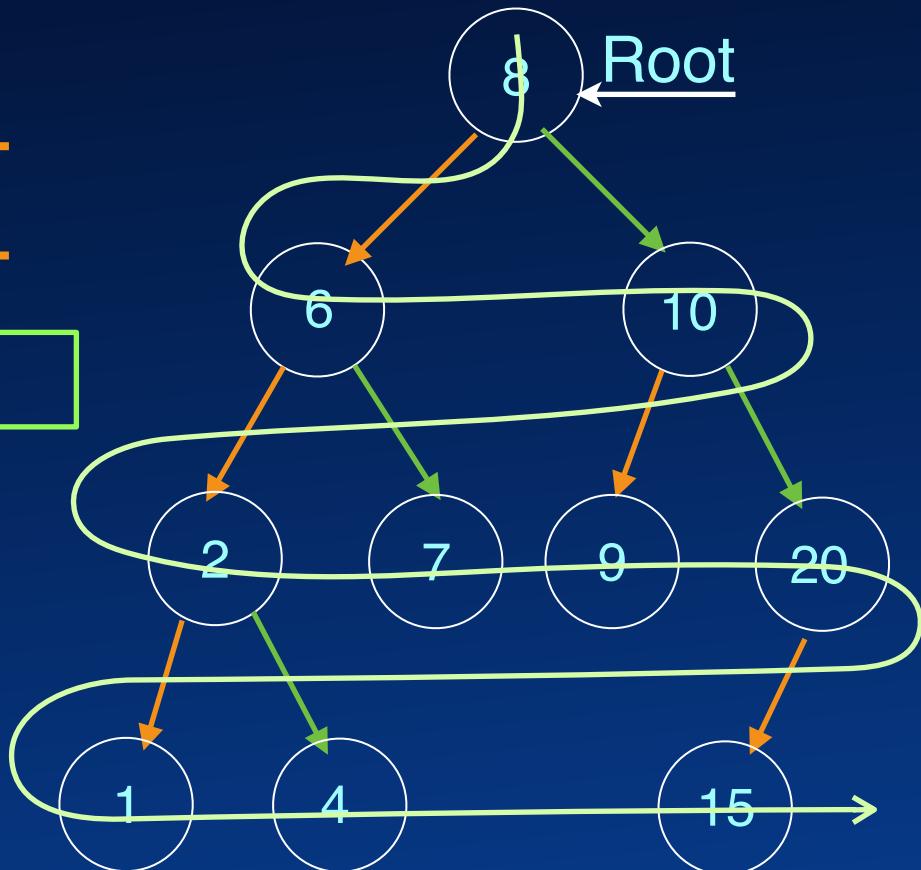
Breadth-first Traversal

Queue

10 2 7



8 6





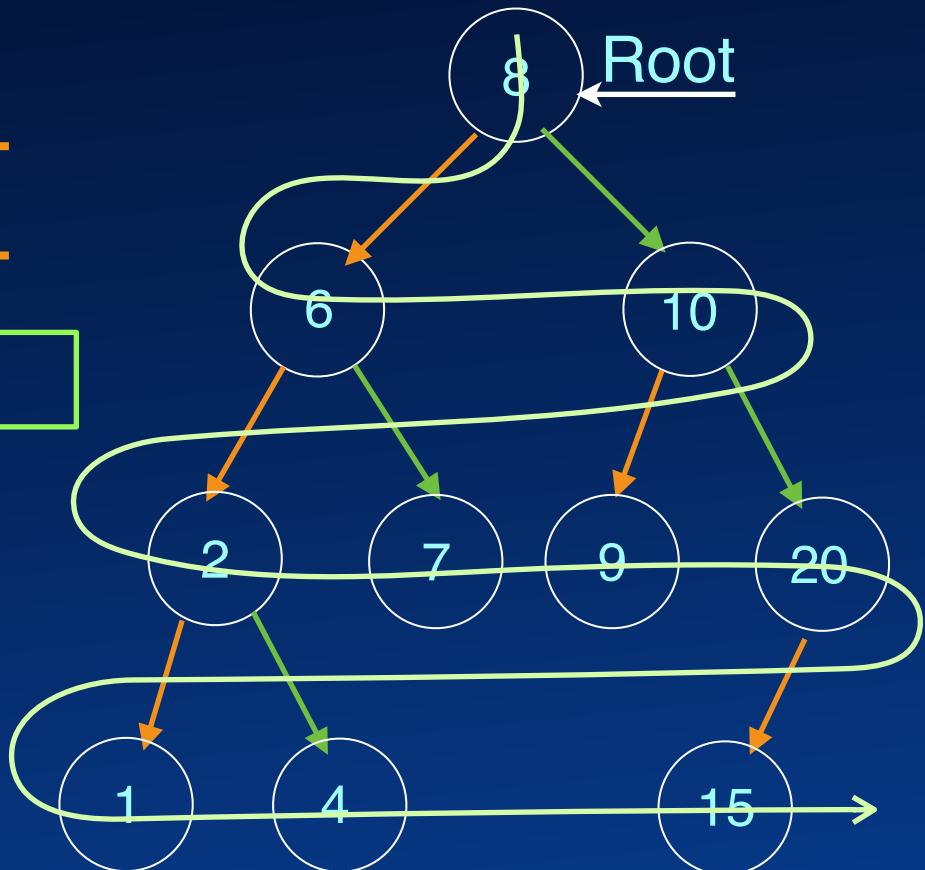
Breadth-first Traversal

Queue

2 7

10

8 6





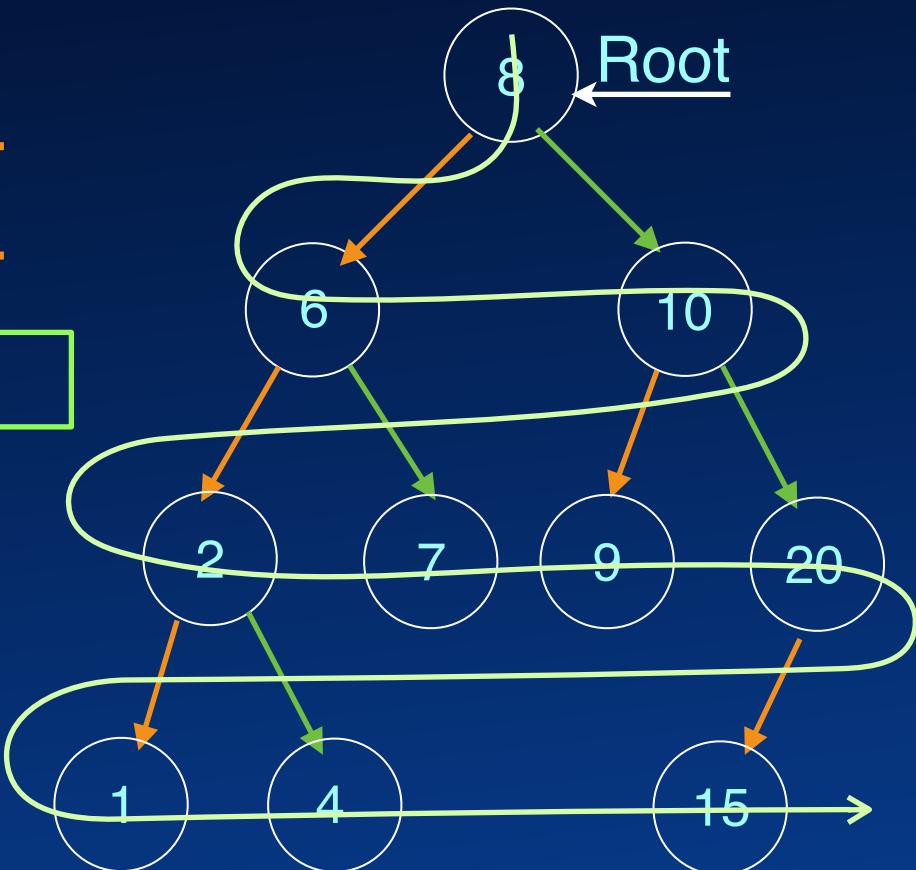
Breadth-first Traversal

Queue

2 7 9 20



8 6





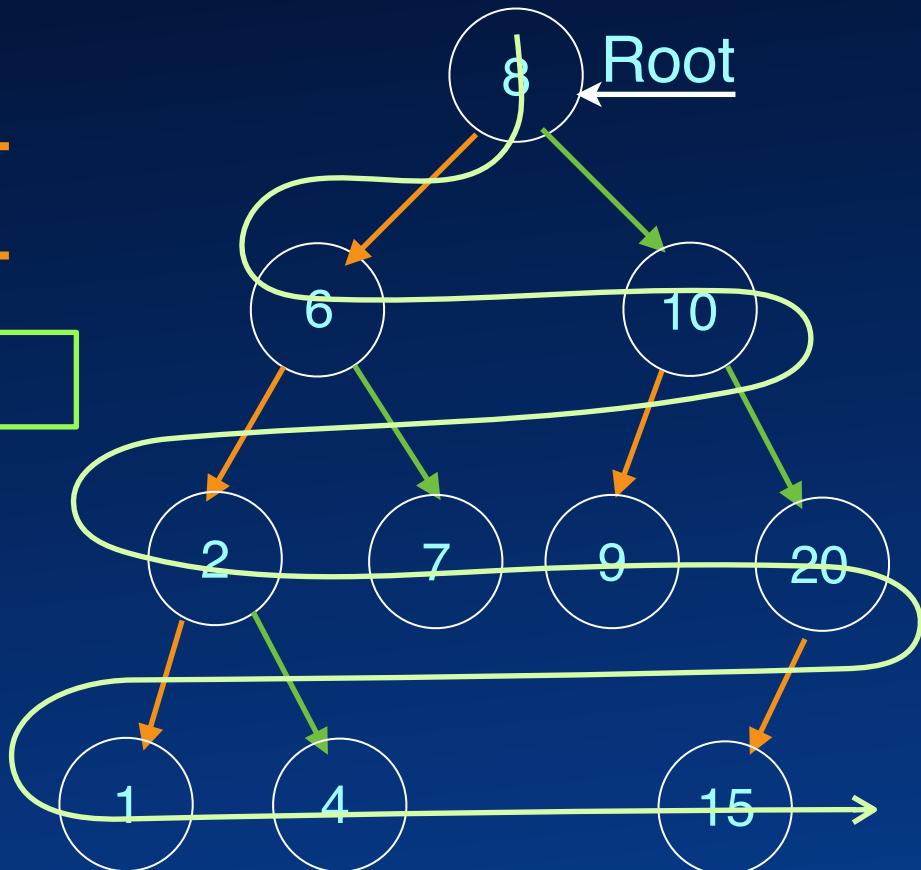
Breadth-first Traversal

Queue

2 7 9 20



8 6 10





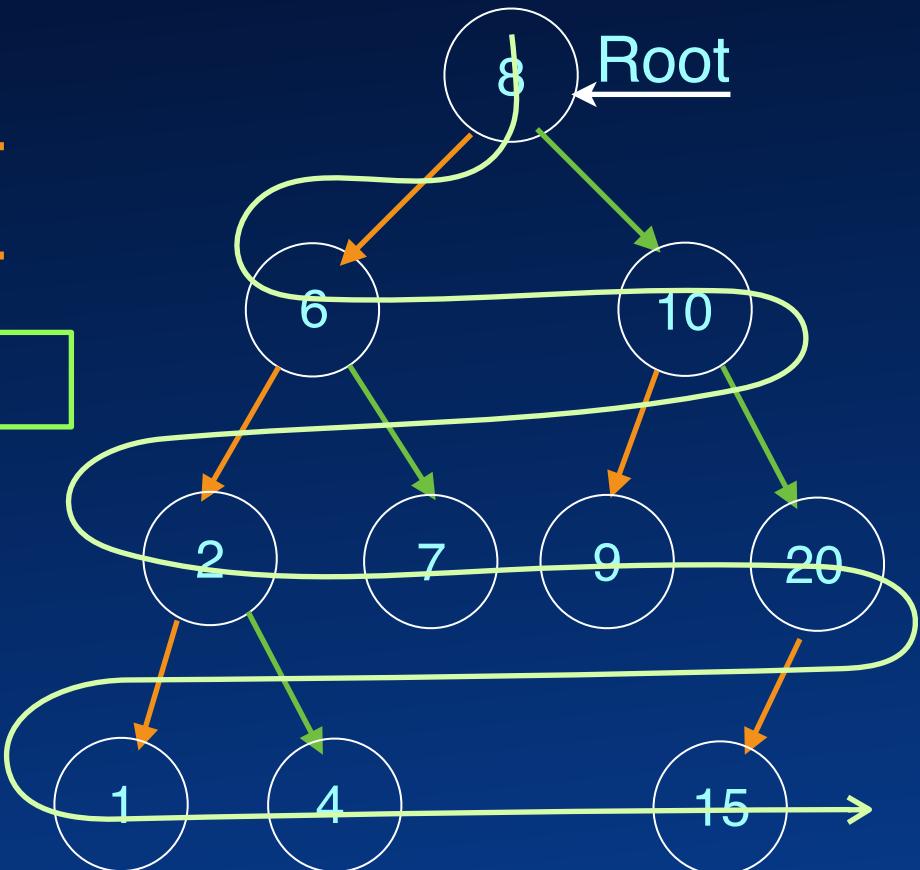
Breadth-first Traversal

Queue

7 9 20

2

8 6 10





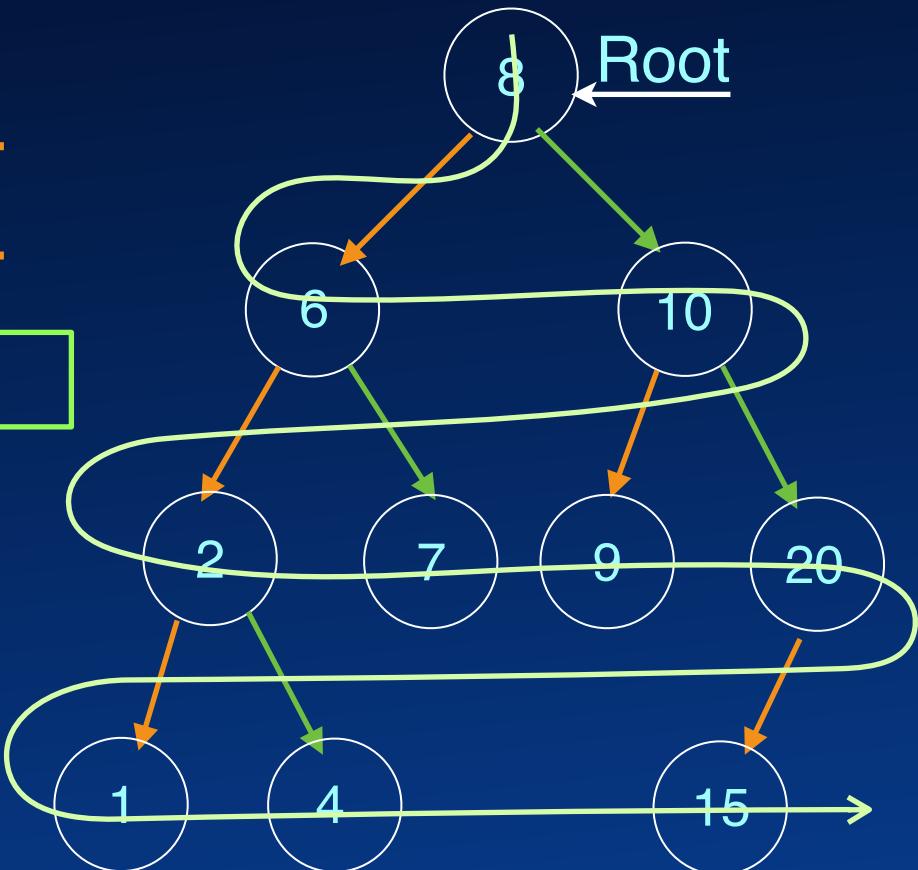
Breadth-first Traversal

Queue

7 9 20 1 4

2

8 6 10



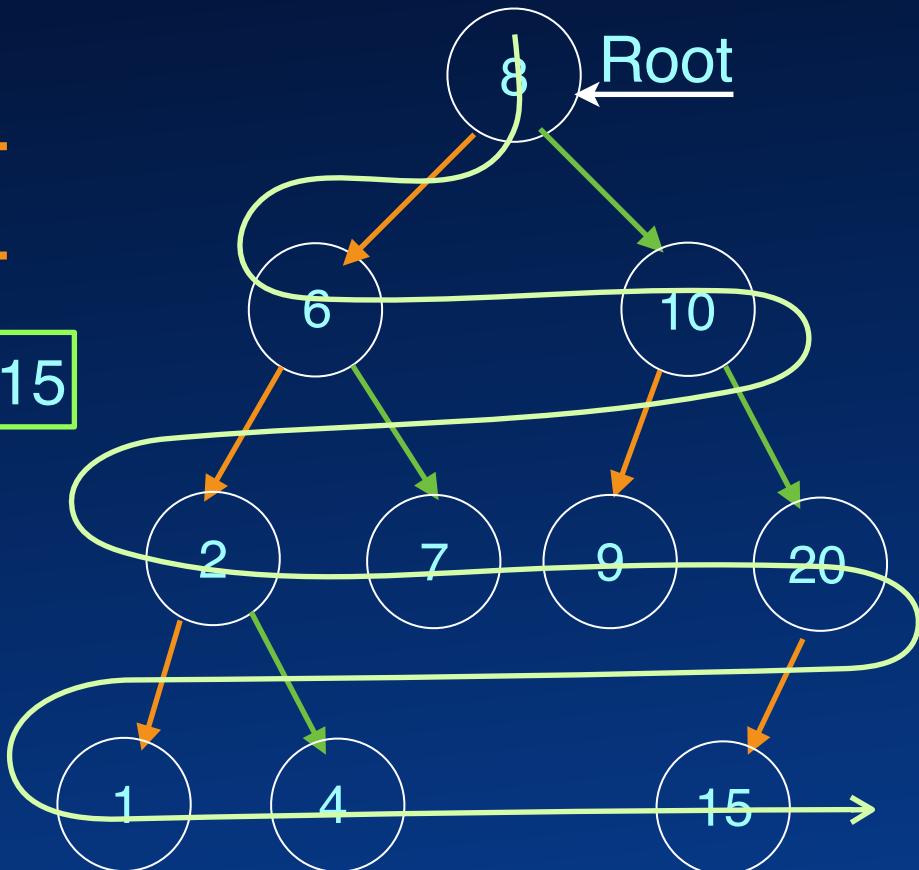


Breadth-first Traversal

Queue



8 6 10 2 7 9 20 1 4 15





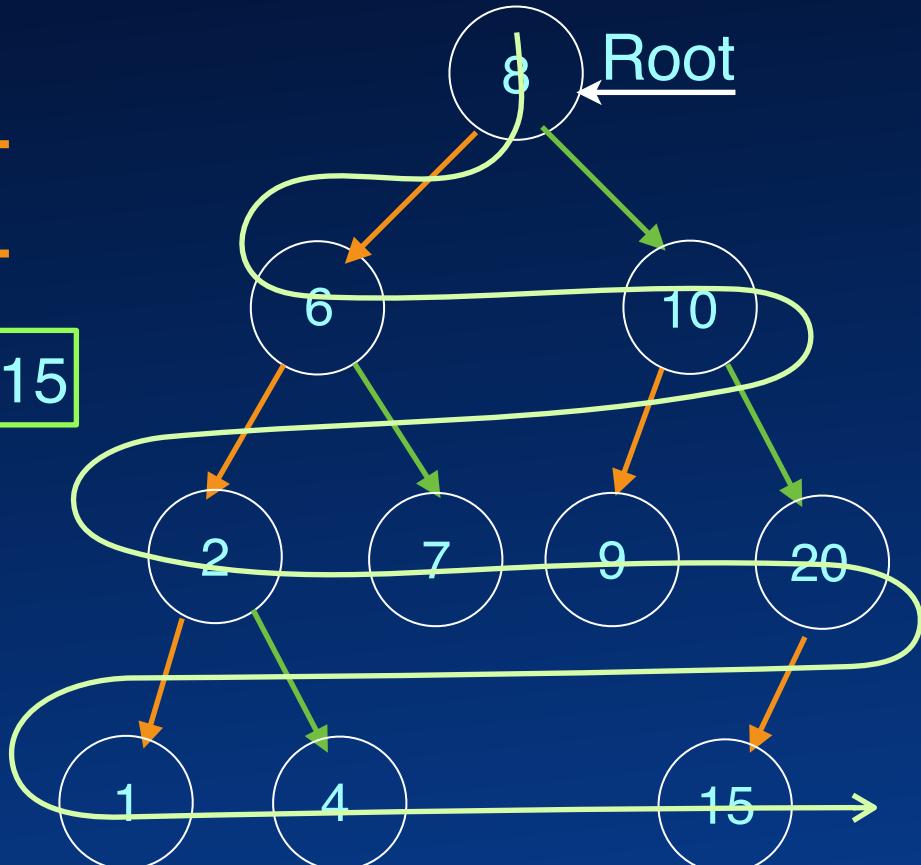
Breadth-first Traversal

Queue



8 6 10 2 7 9 20 1 4 15

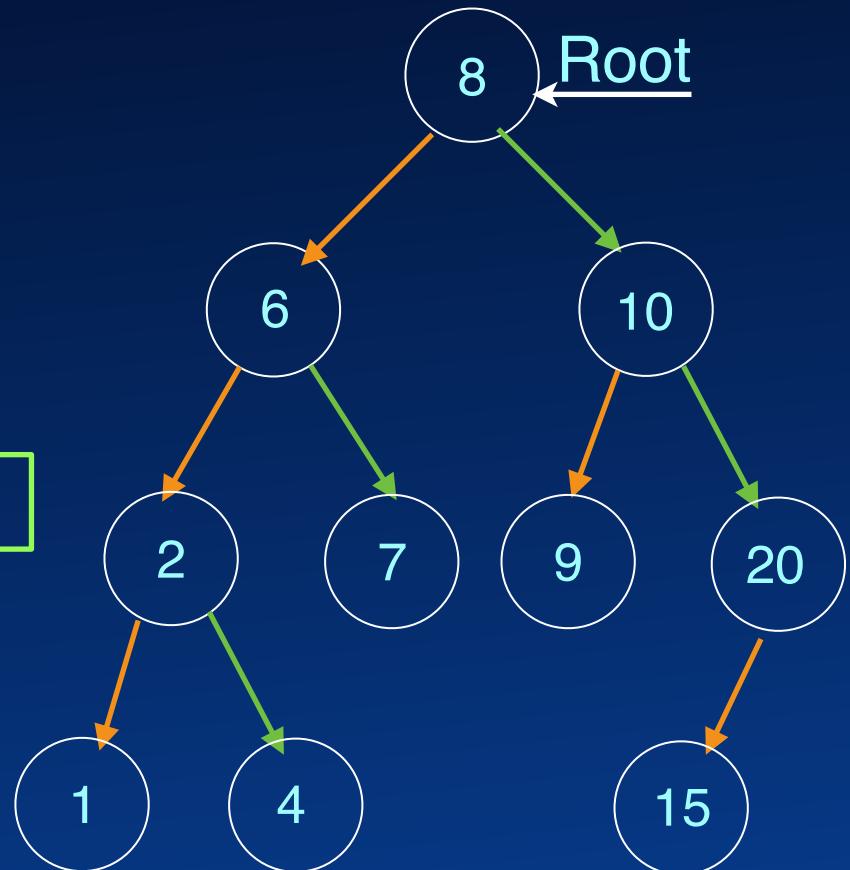
```
q.insert(root)
while(q.hasNext()) {
    x = q.next();
    q.insert(x.left);
    q.insert(x.right);
    op.accept(x);
}
```





Depth-first Traversal

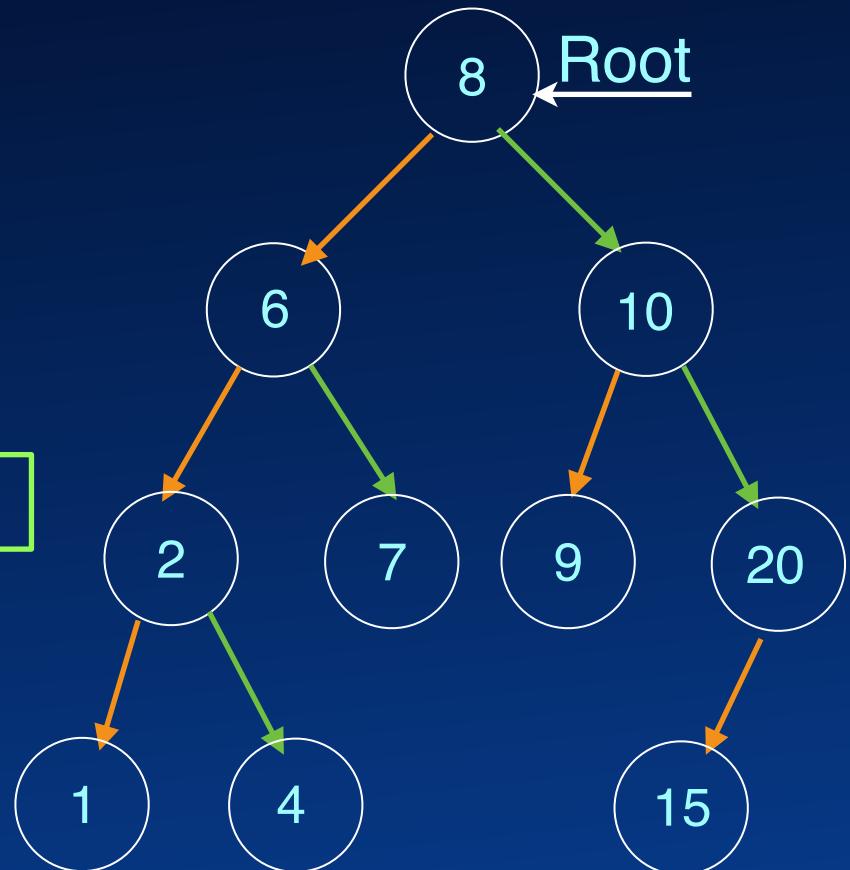
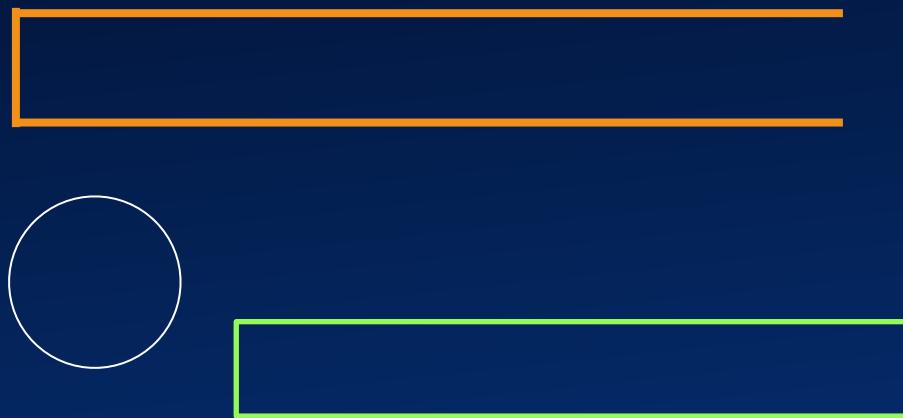
Queue





Depth-first Traversal

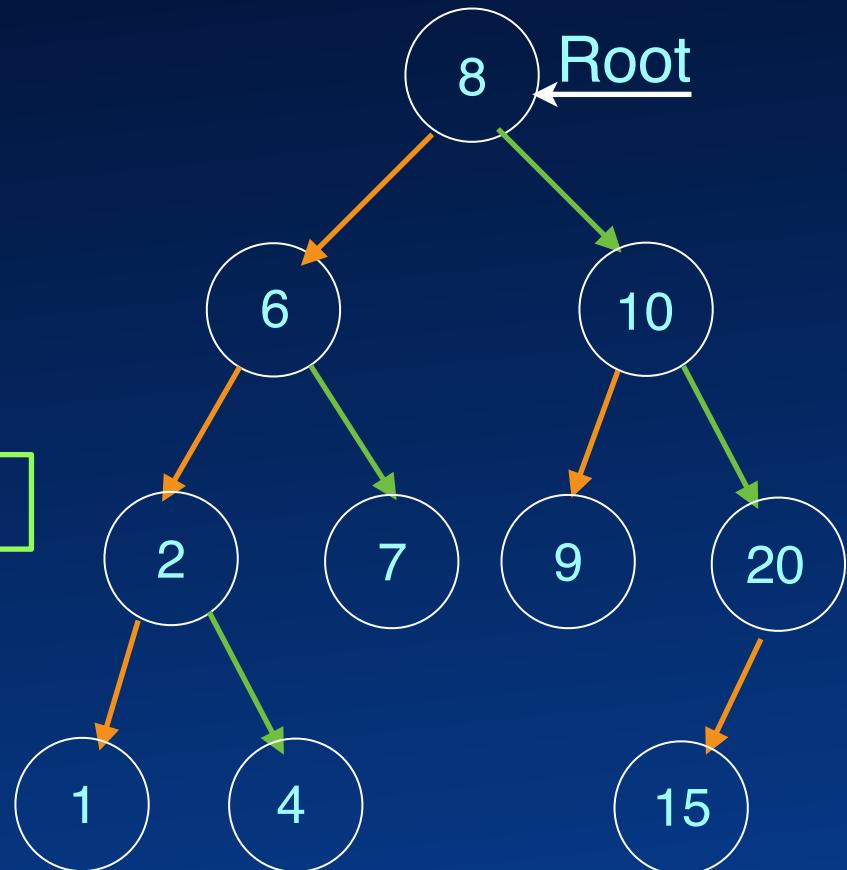
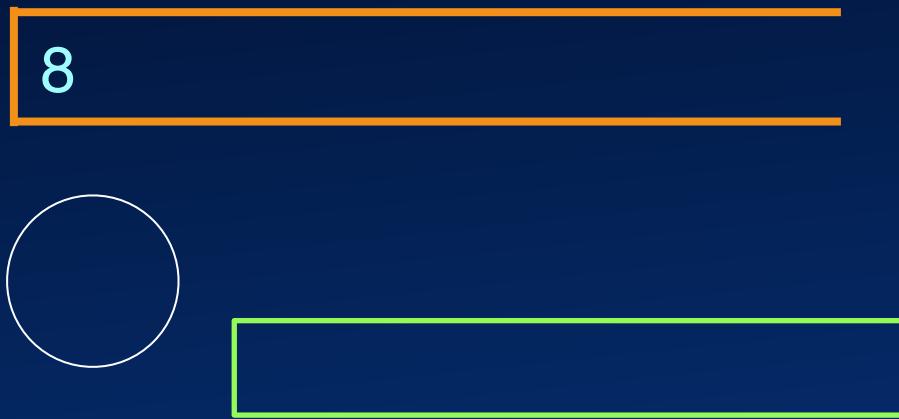
Stack





Depth-first Traversal

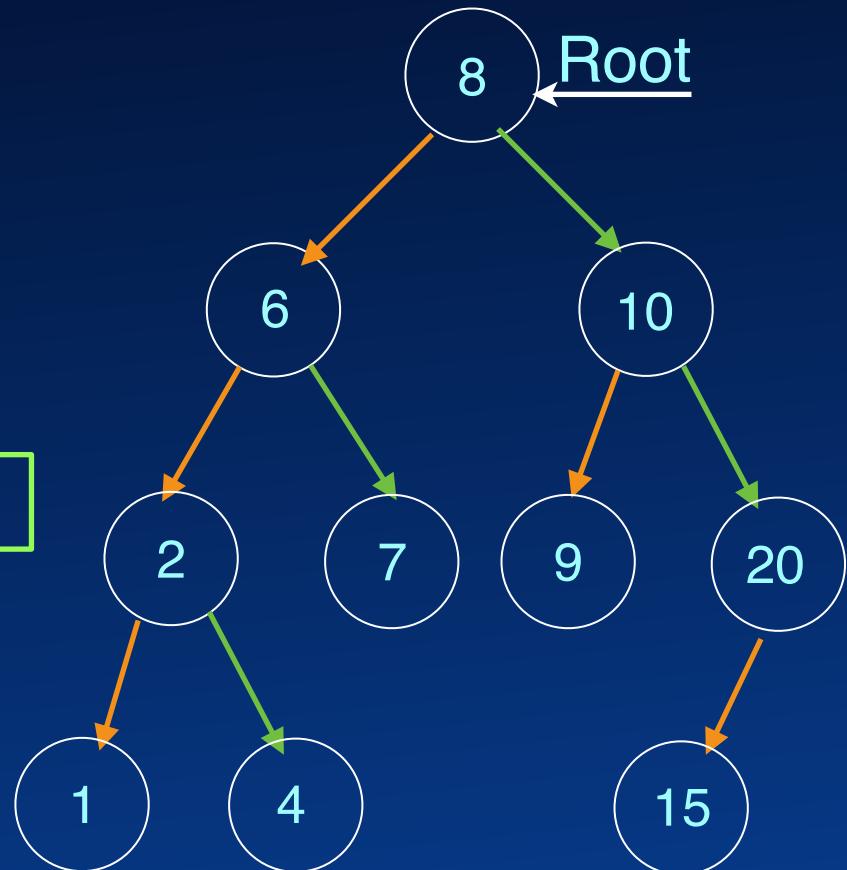
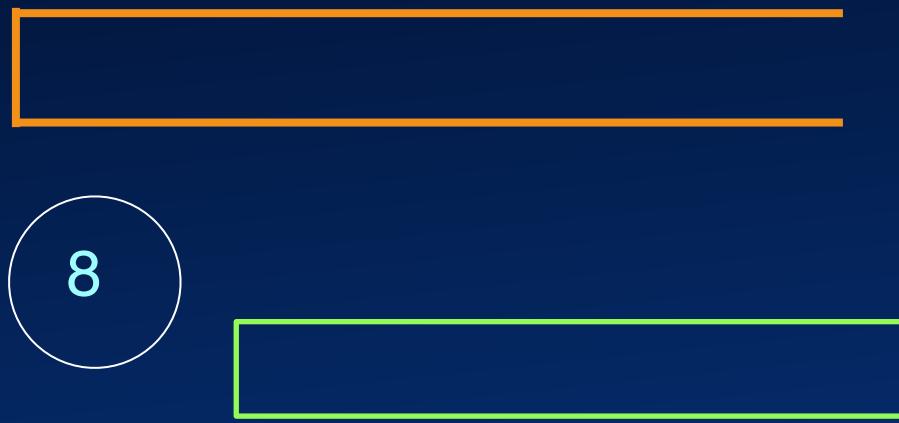
Stack





Depth-first Traversal

Stack

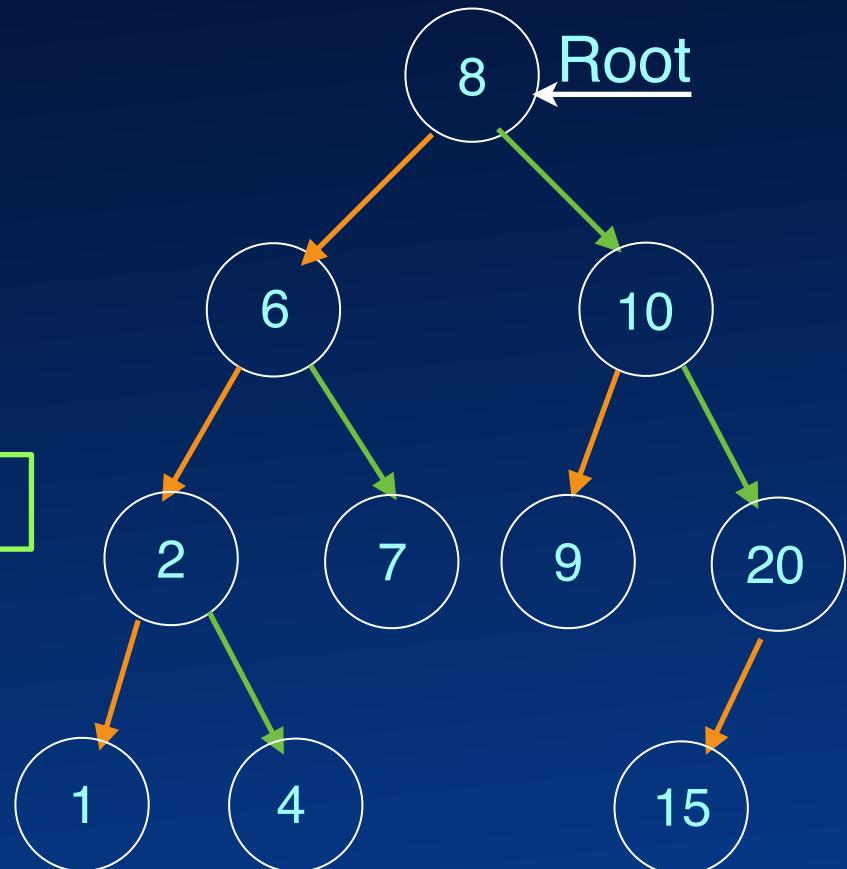




Depth-first Traversal

Stack

10 6





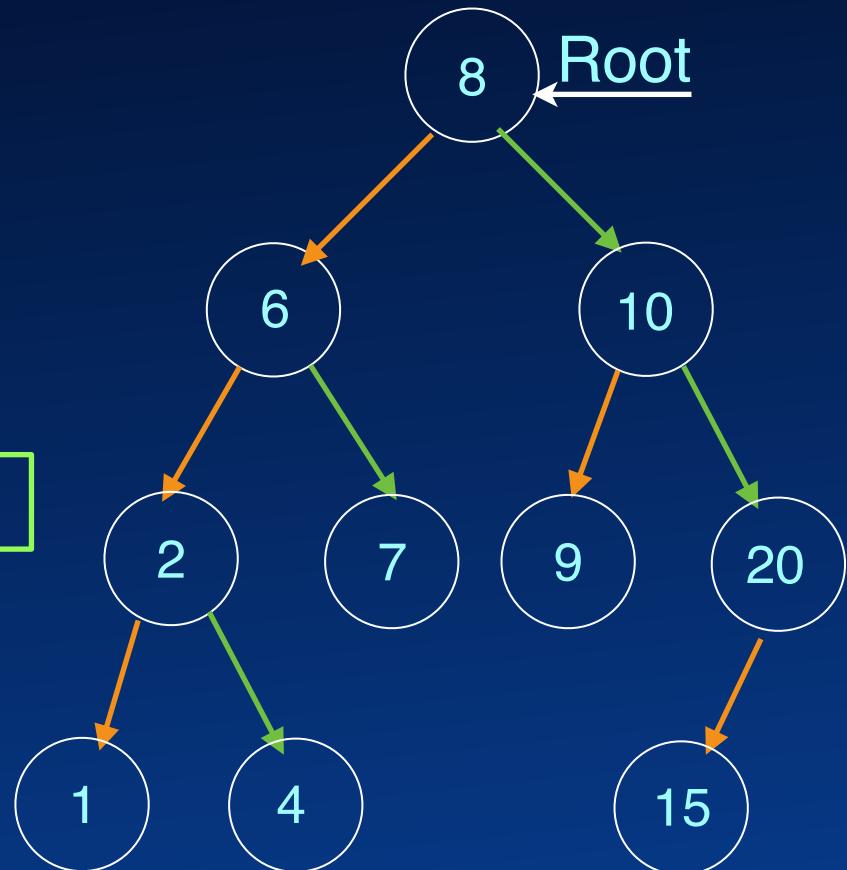
Depth-first Traversal

Stack

10 6



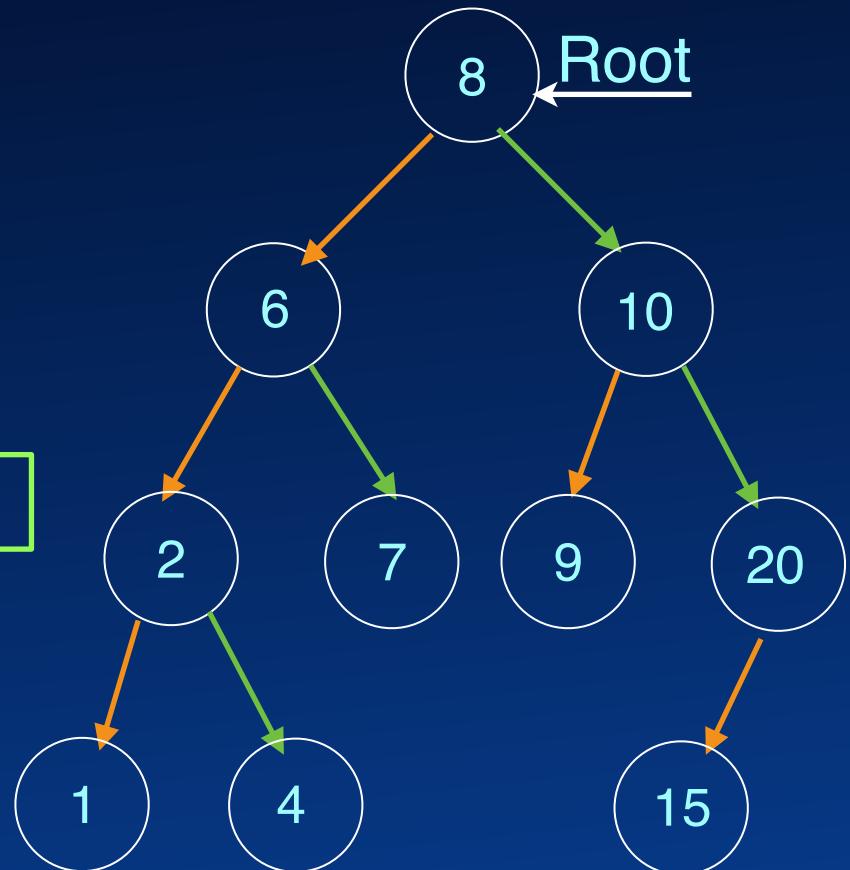
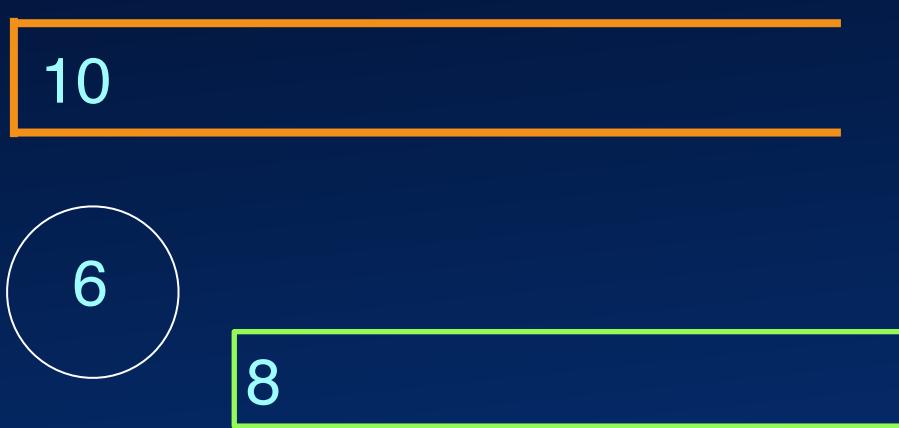
8





Depth-first Traversal

Stack





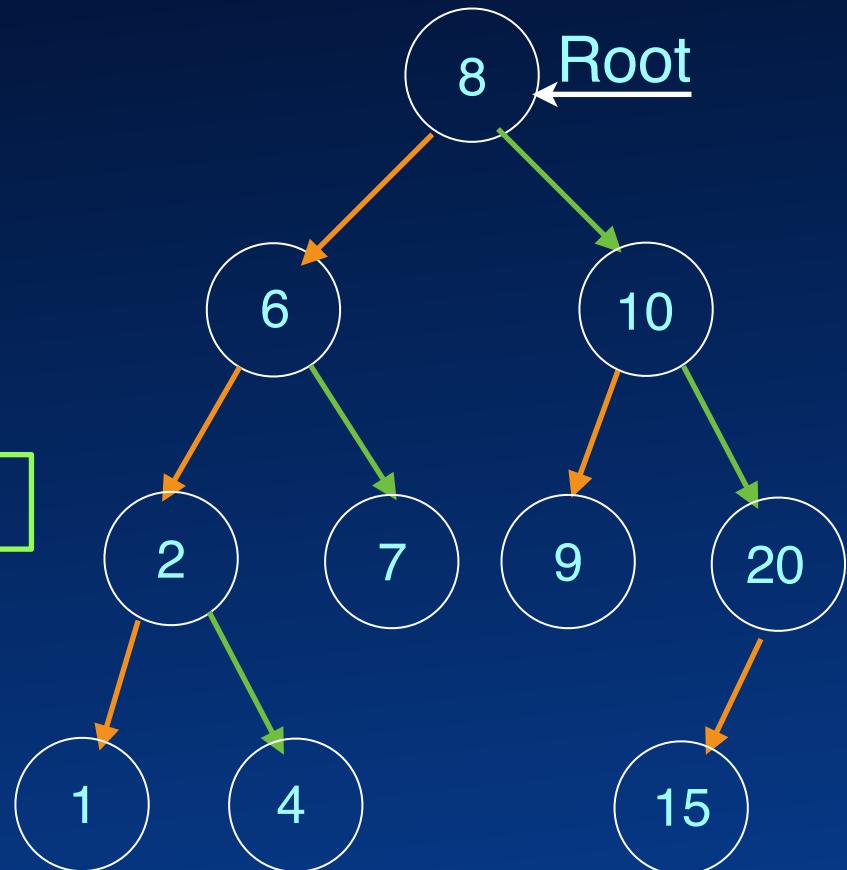
Depth-first Traversal

Stack

10 7 2



8





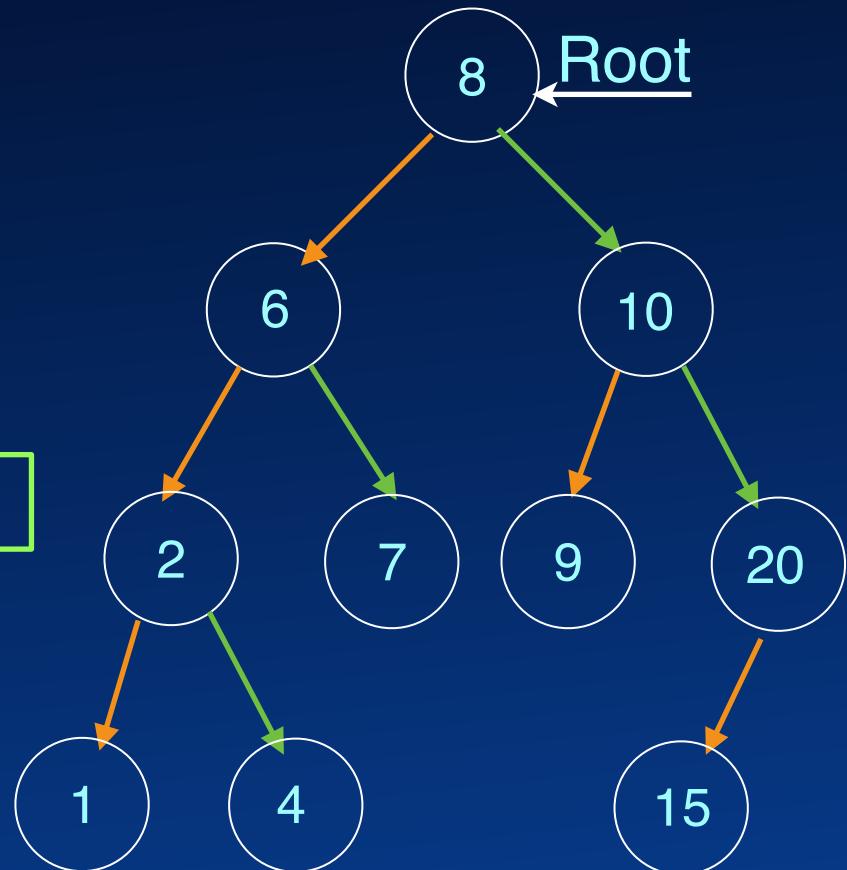
Depth-first Traversal

Stack

10 7 2



8 6





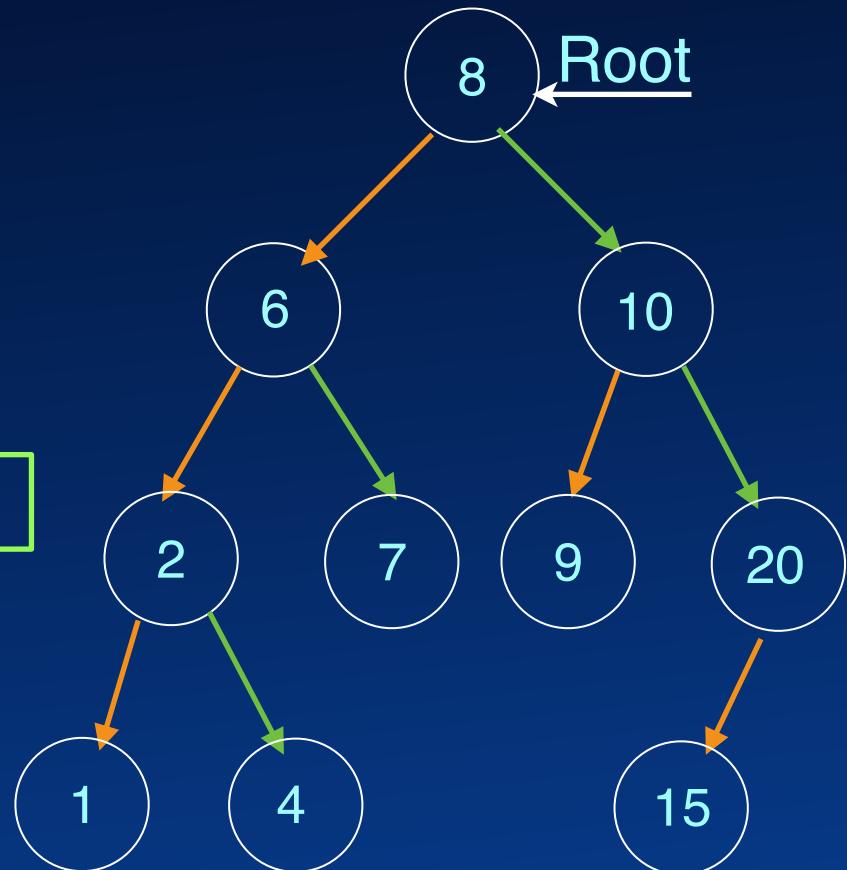
Depth-first Traversal

Stack

10 7



8 6





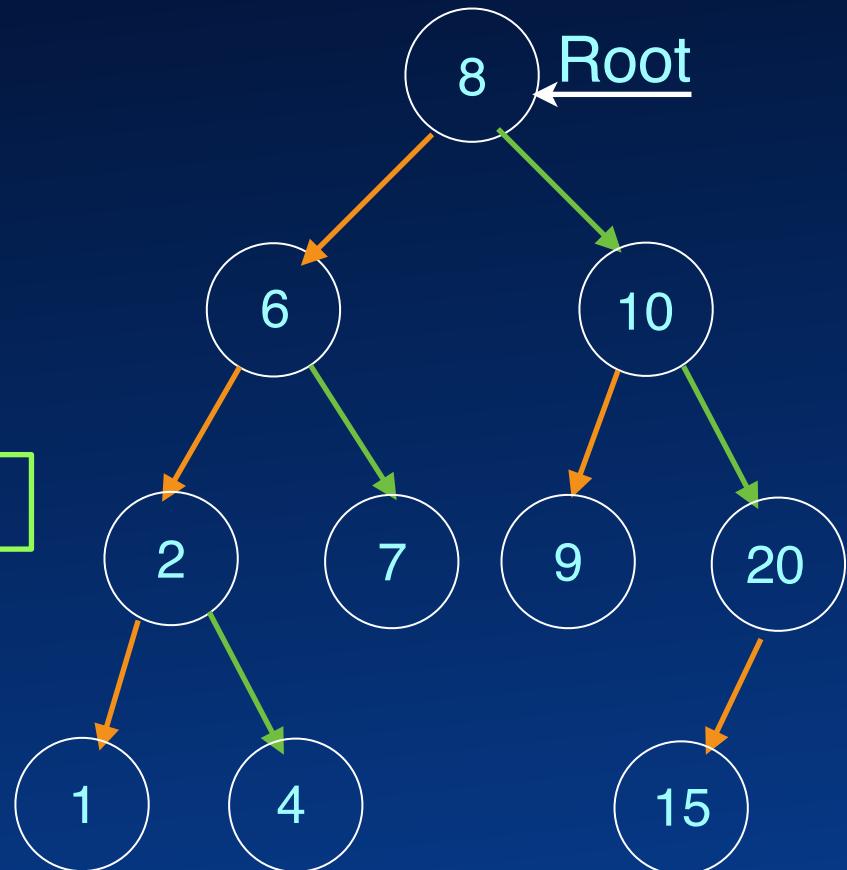
Depth-first Traversal

Stack

10 7 4 1



8 6





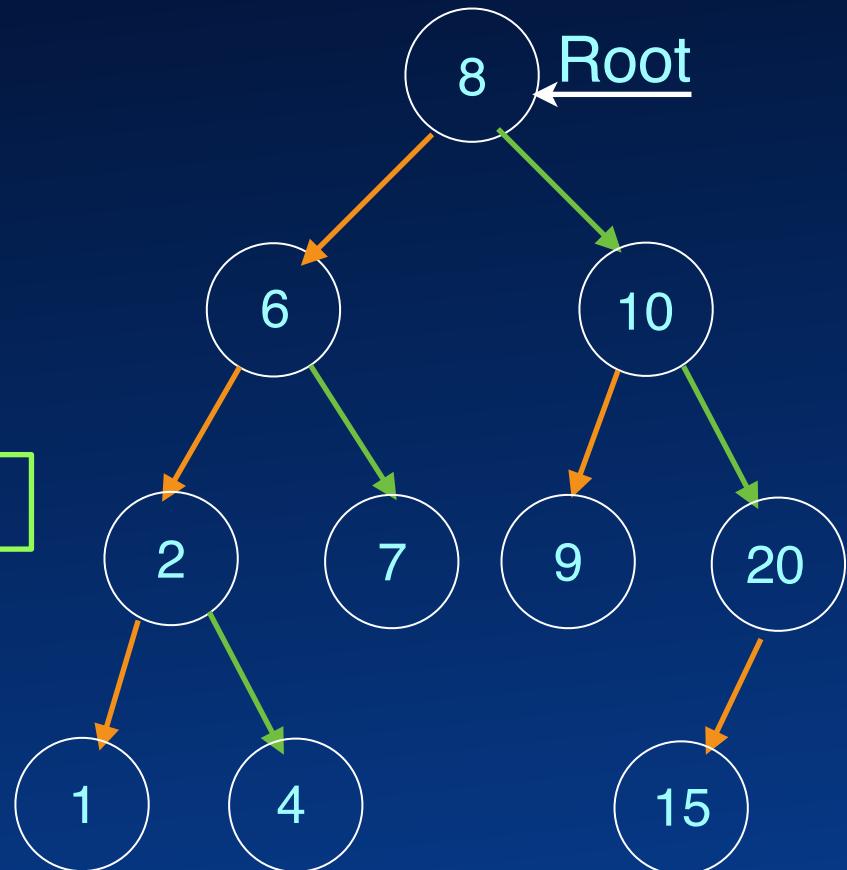
Depth-first Traversal

Stack

10 7 4 1



8 6 2





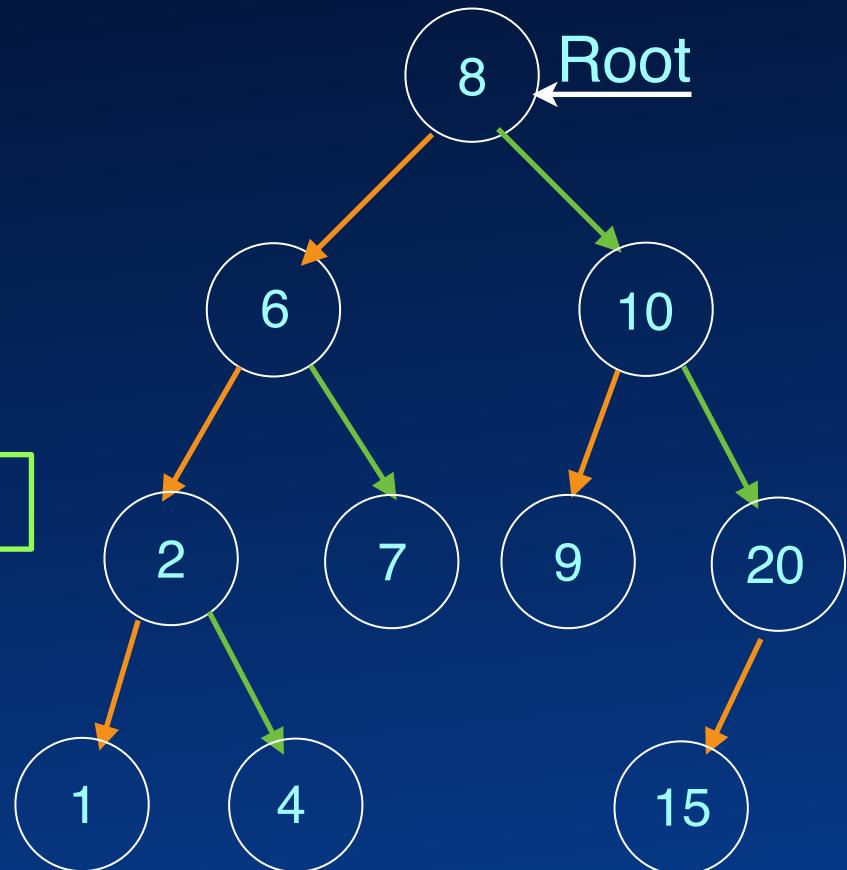
Depth-first Traversal

Stack

10 7 4



8 6 2





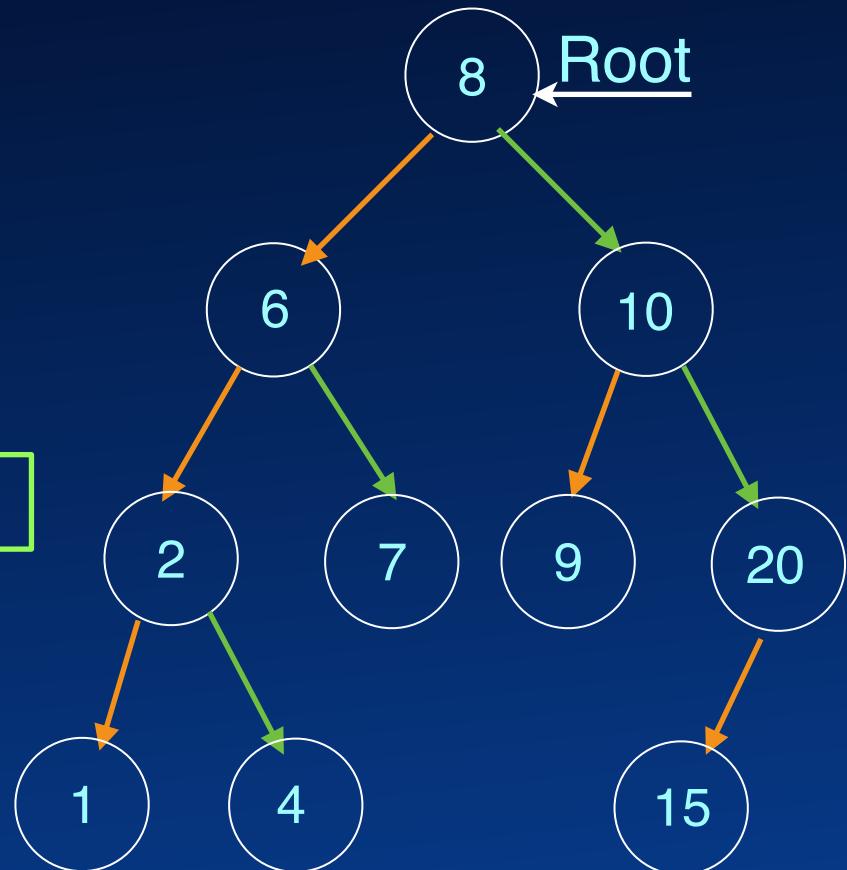
Depth-first Traversal

Stack

10 7 4



8 6 2 1





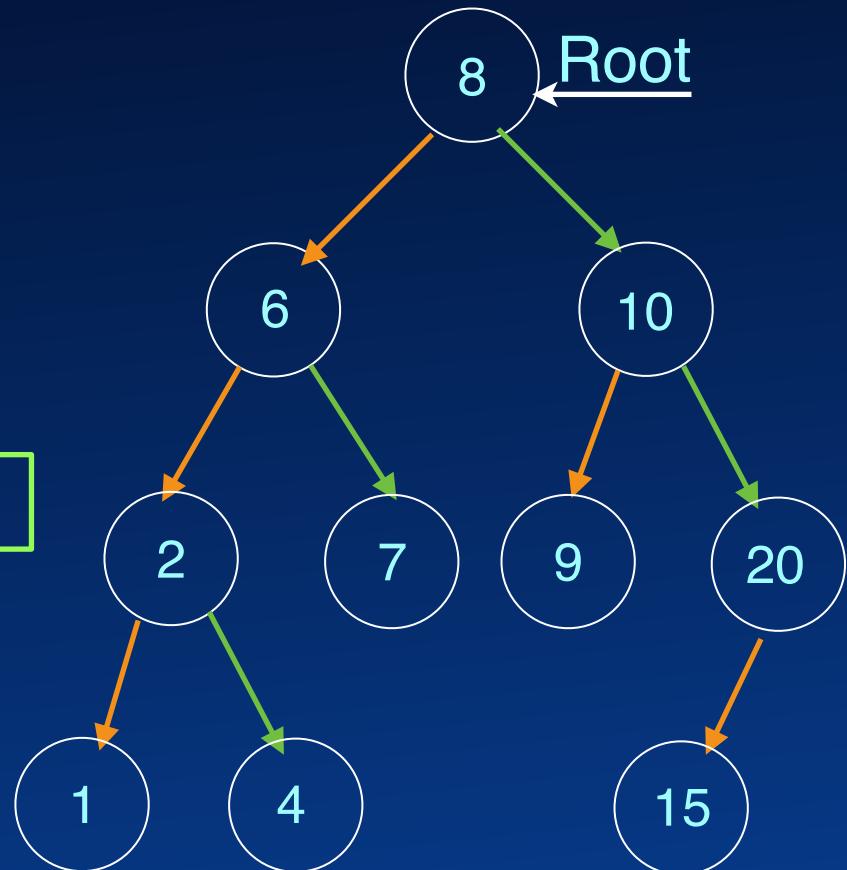
Depth-first Traversal

Stack

10 7



8 6 2 1 4





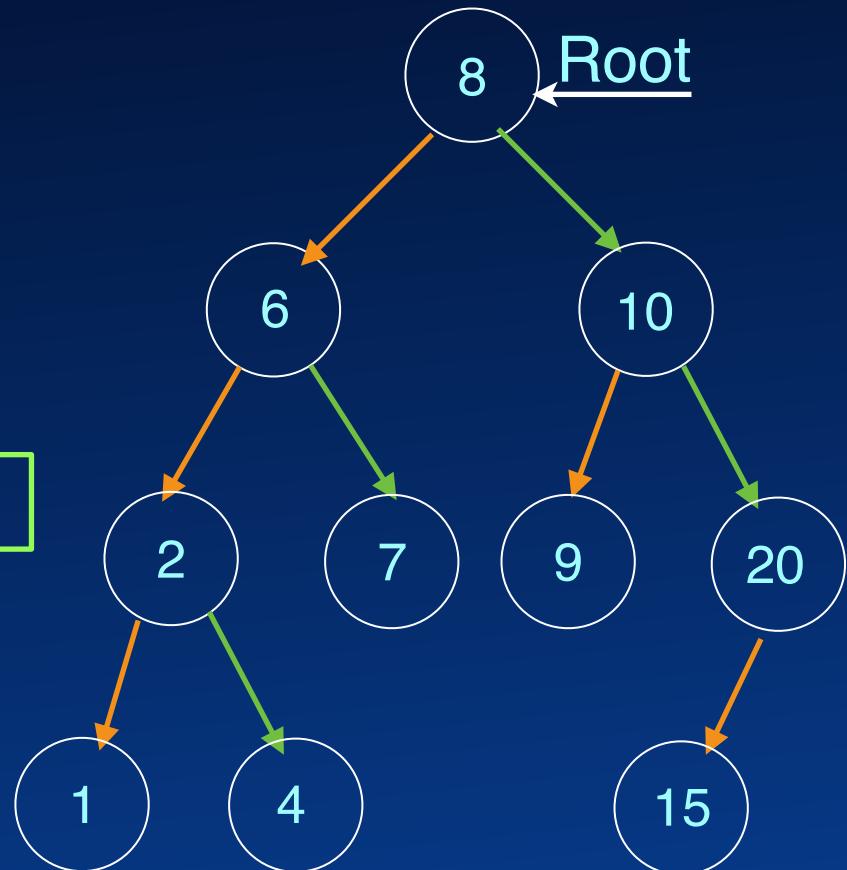
Depth-first Traversal

Stack

10



8 6 2 1 4 7





Depth-first Traversal

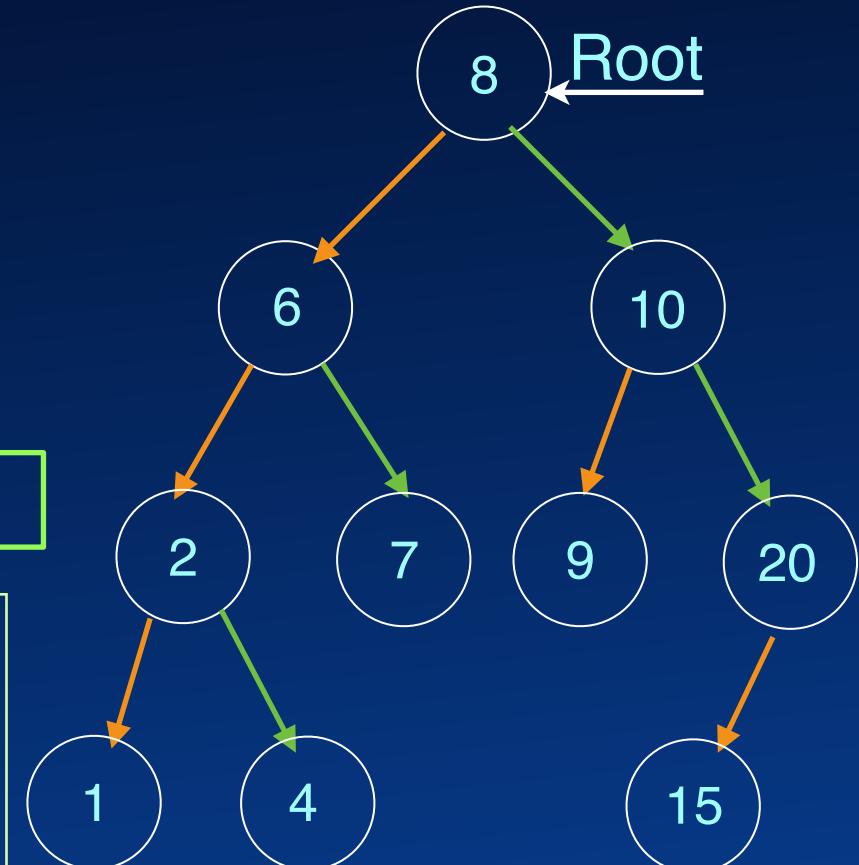
Stack

10



8 6 2 1 4 7

```
stack.push(root)
while(stack.hasany()) {
    x = stack.pop();
    stack.push(x.right);
    stack.push(x.left);
    op.accept(x);
}
```





True or False?

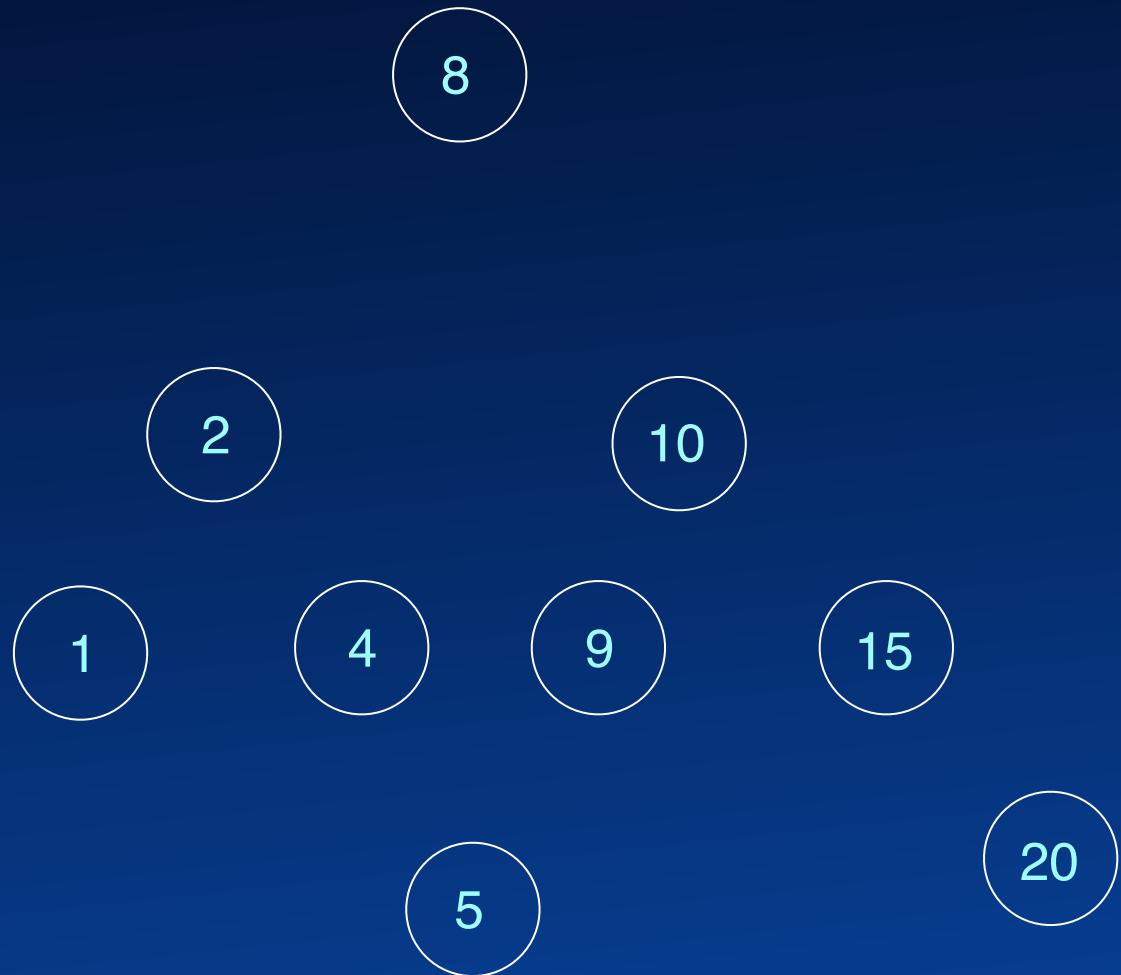
Format: f,f,f,f

Mail: col106quiz@cse.iitd.ac.in

- **The number of internal nodes in a proper binary tree is less than $n/2$.**
- **The Euler's tour of a binary tree enters each node 3 times.**
- **The height of a complete binary tree with n nodes, is $\text{ceil}(\log(n+1)) - 1$.**
- **In-order traversal of a binary search tree operates on the keys of the tree in an increasing order.**

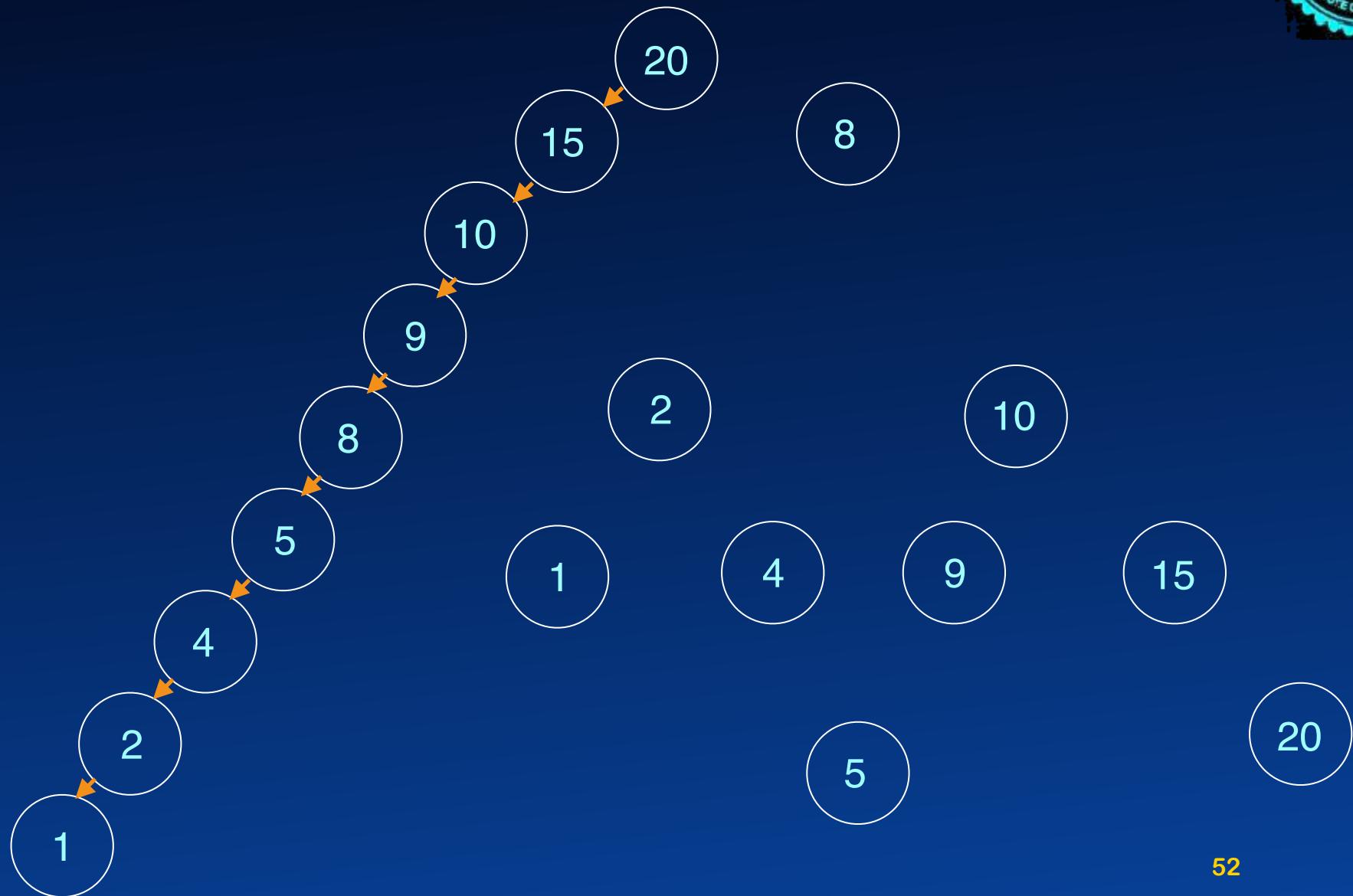


Tree Balance



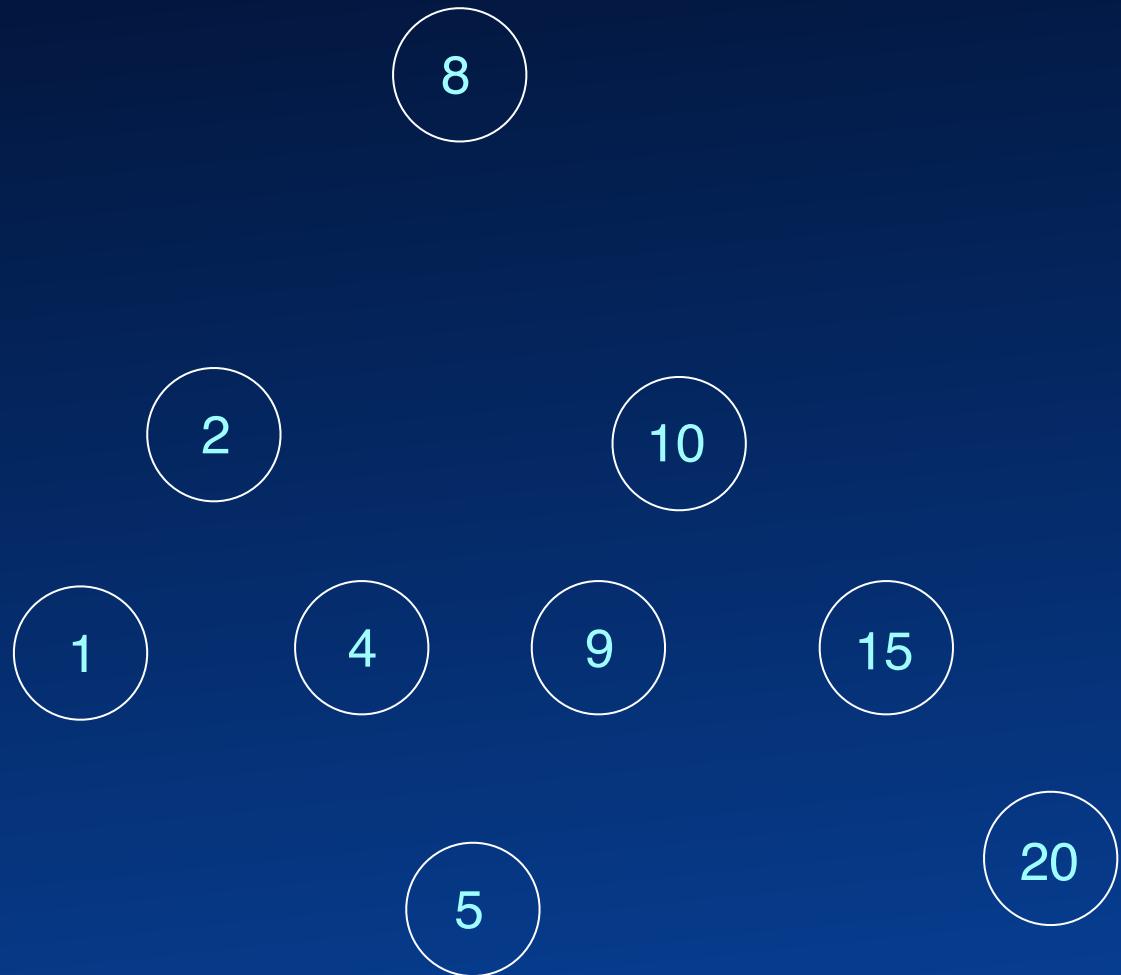


Tree Balance



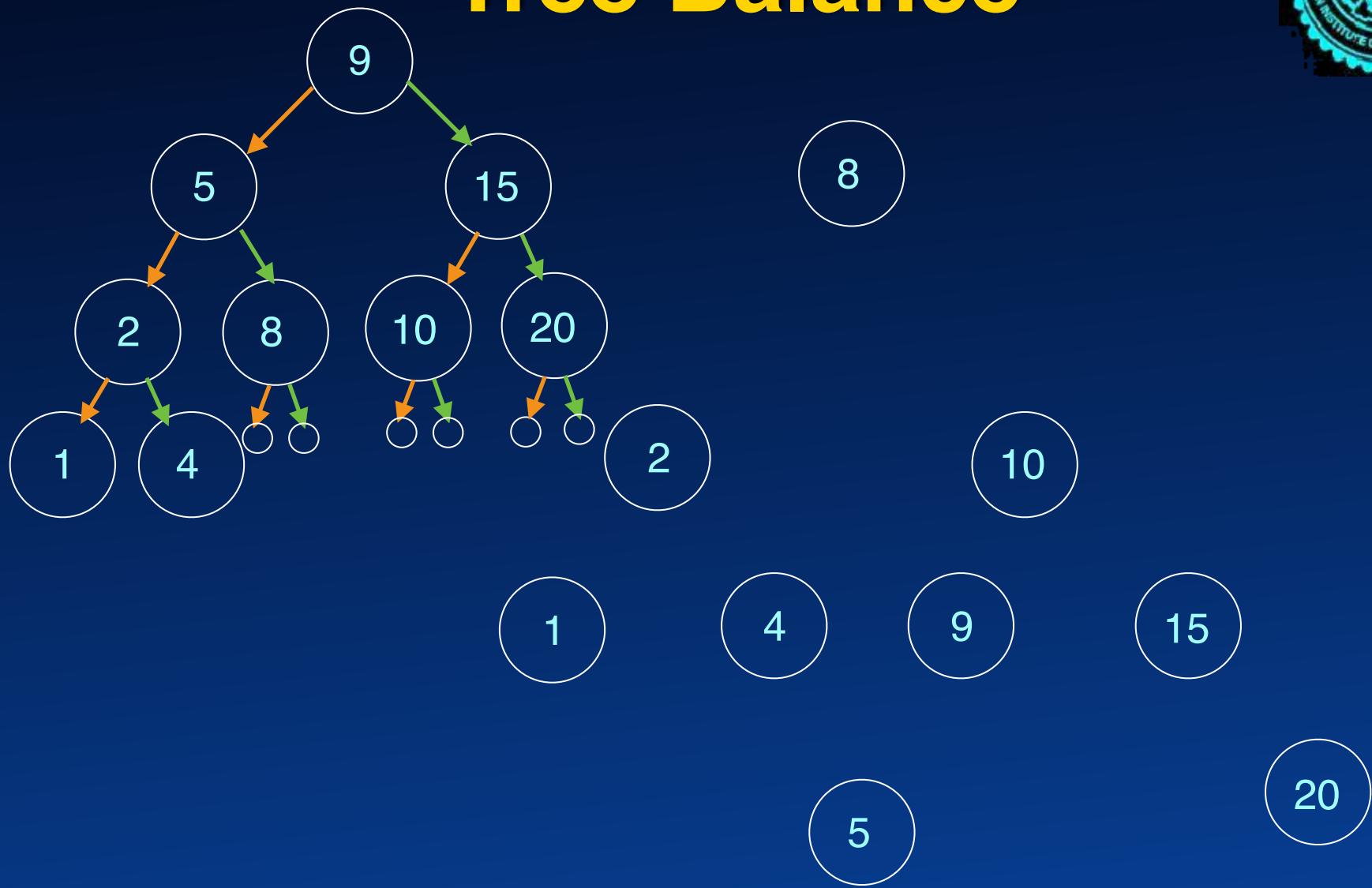


Tree Balance



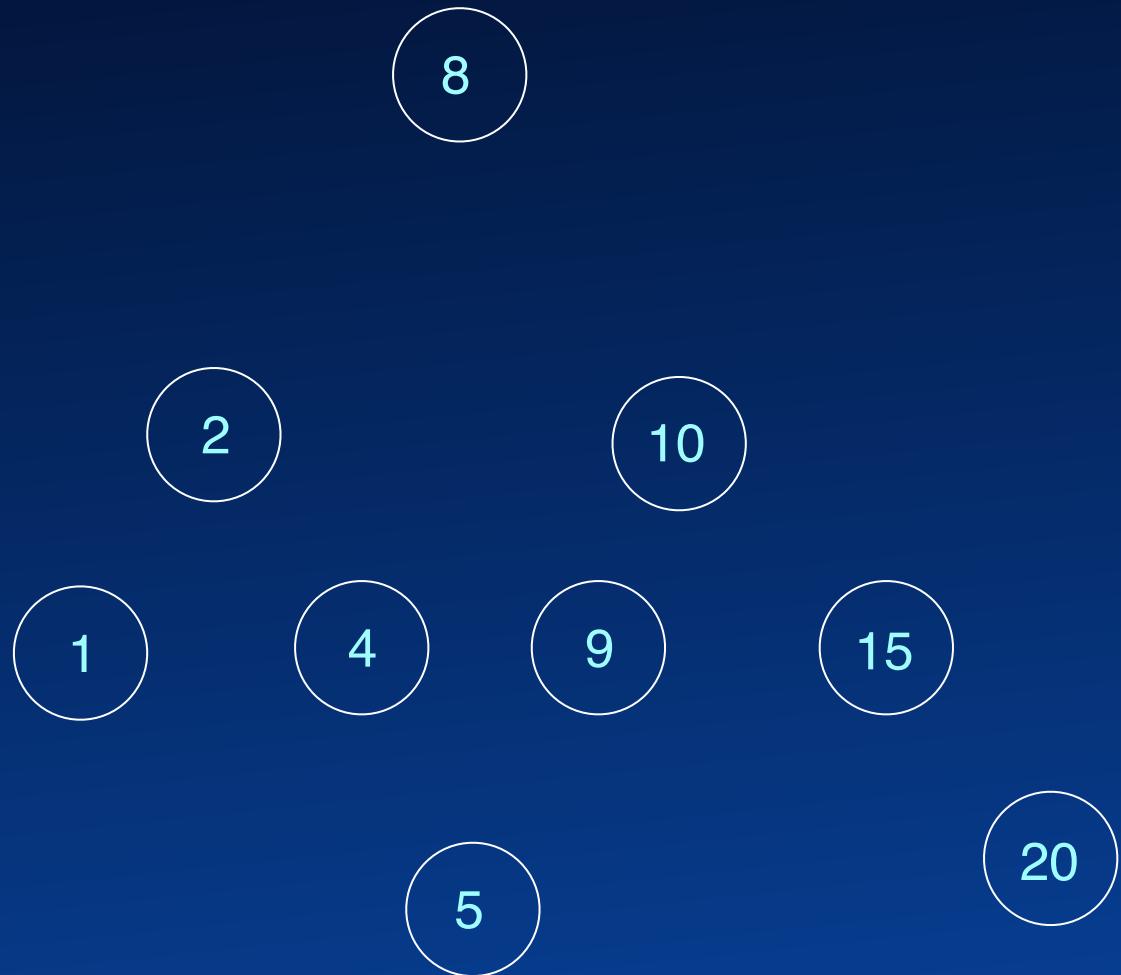


Tree Balance



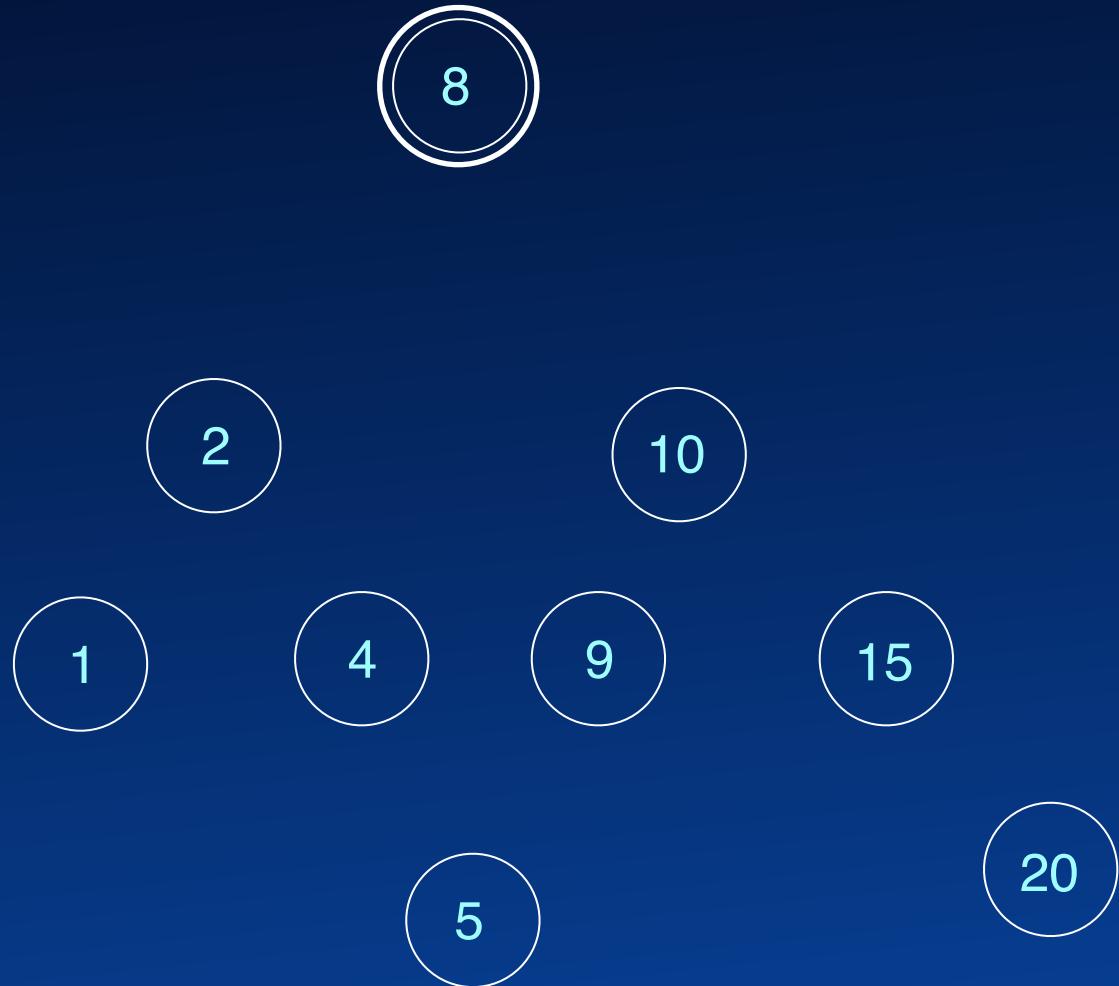


Tree Balance





Tree Balance





Tree Balance

8
Root





Tree Balance

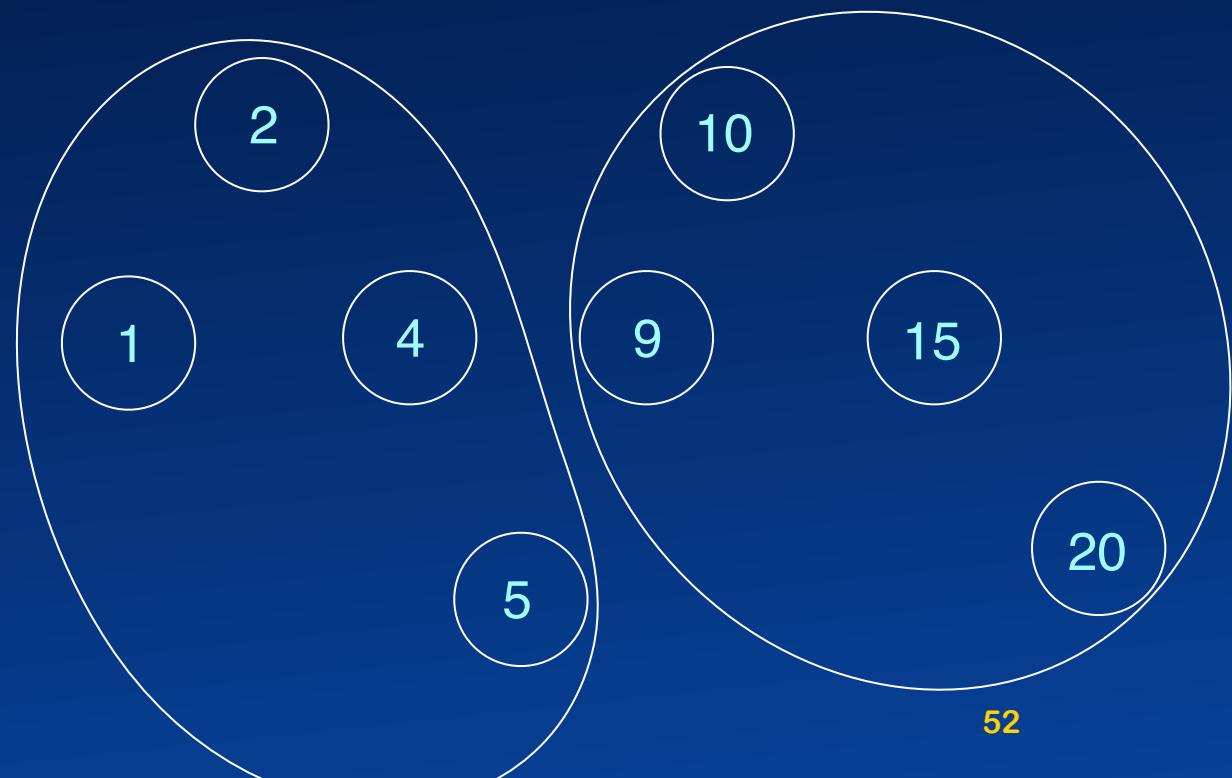
8
Root





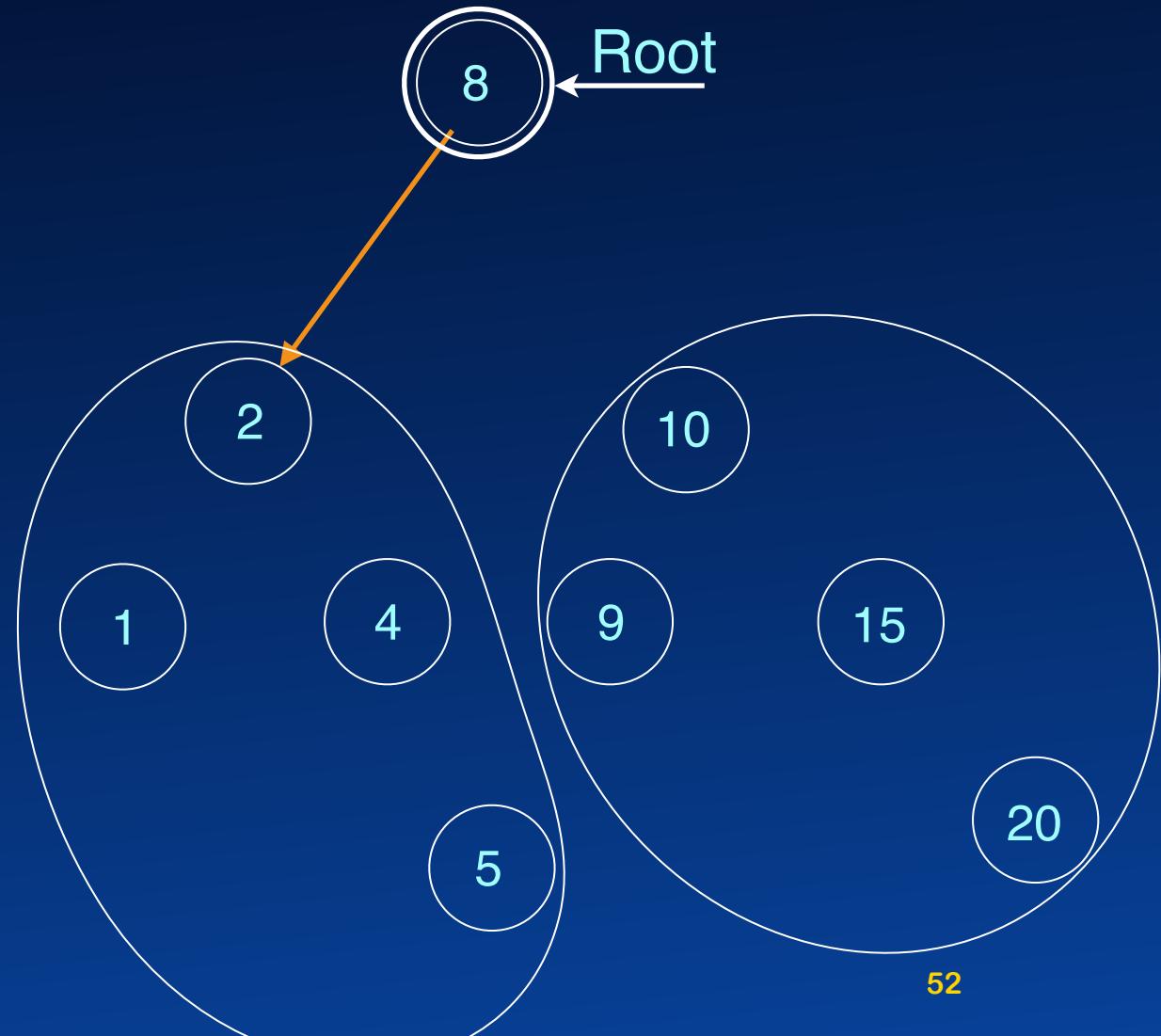
Tree Balance

8
Root



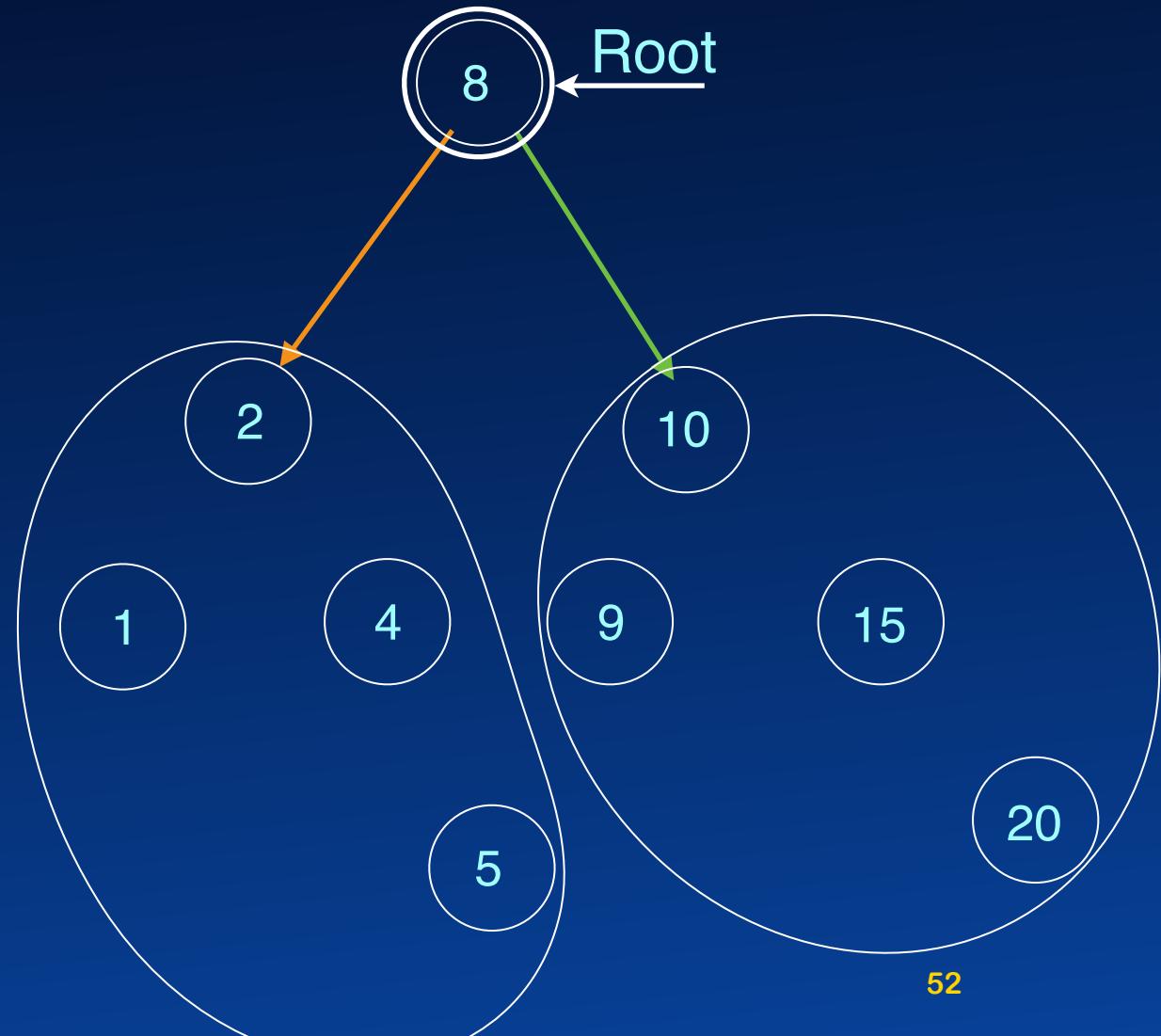


Tree Balance



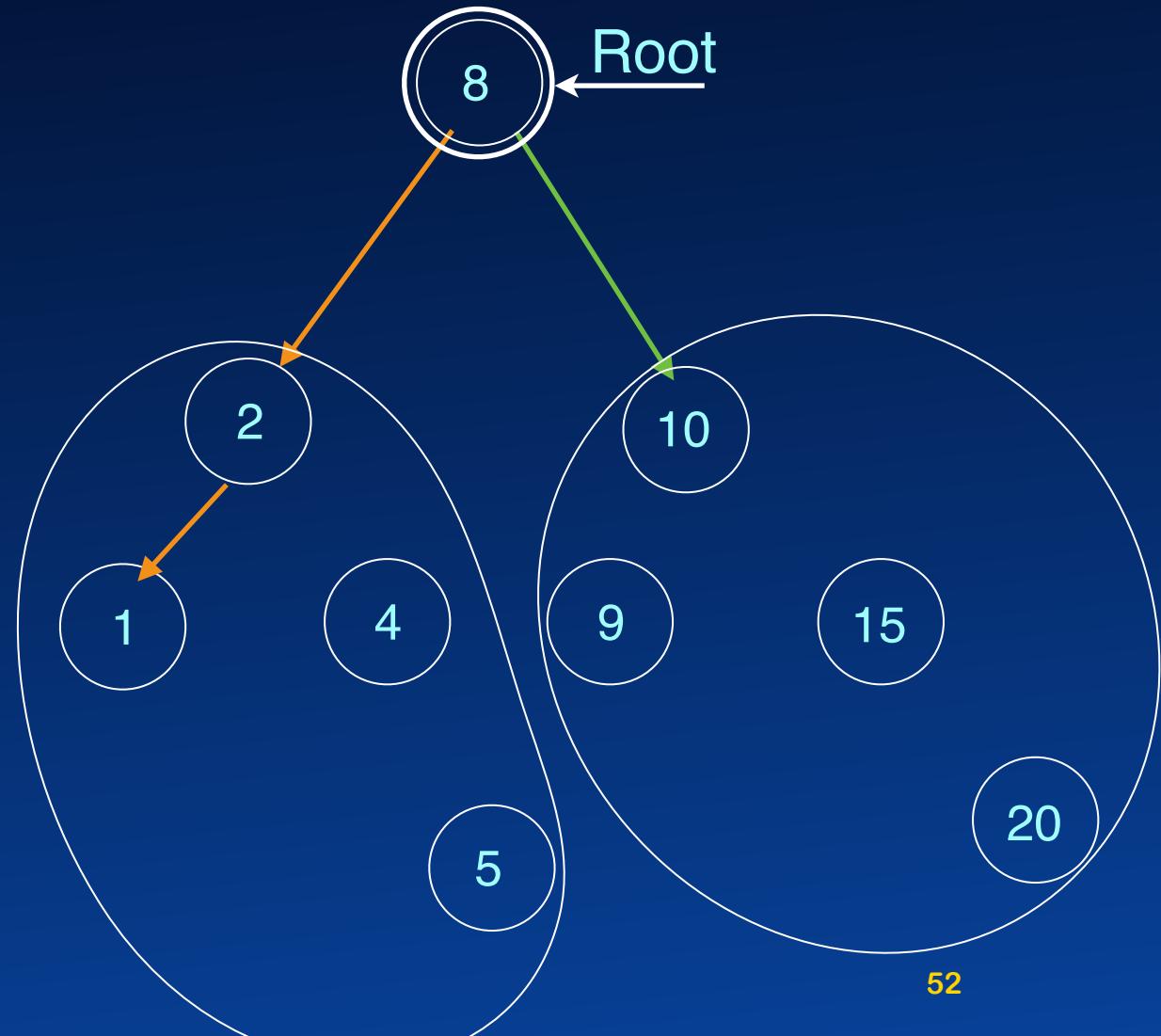


Tree Balance



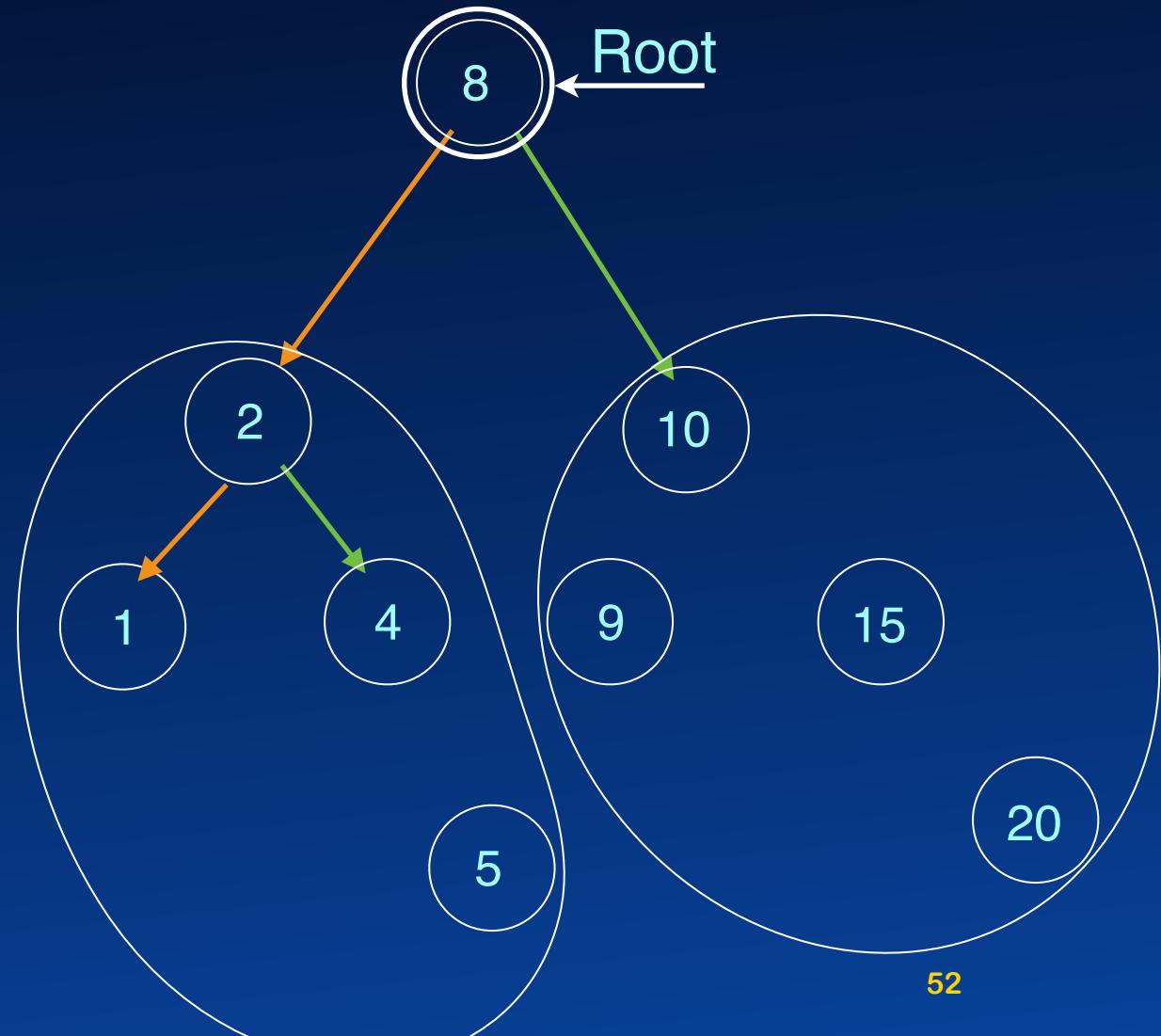


Tree Balance



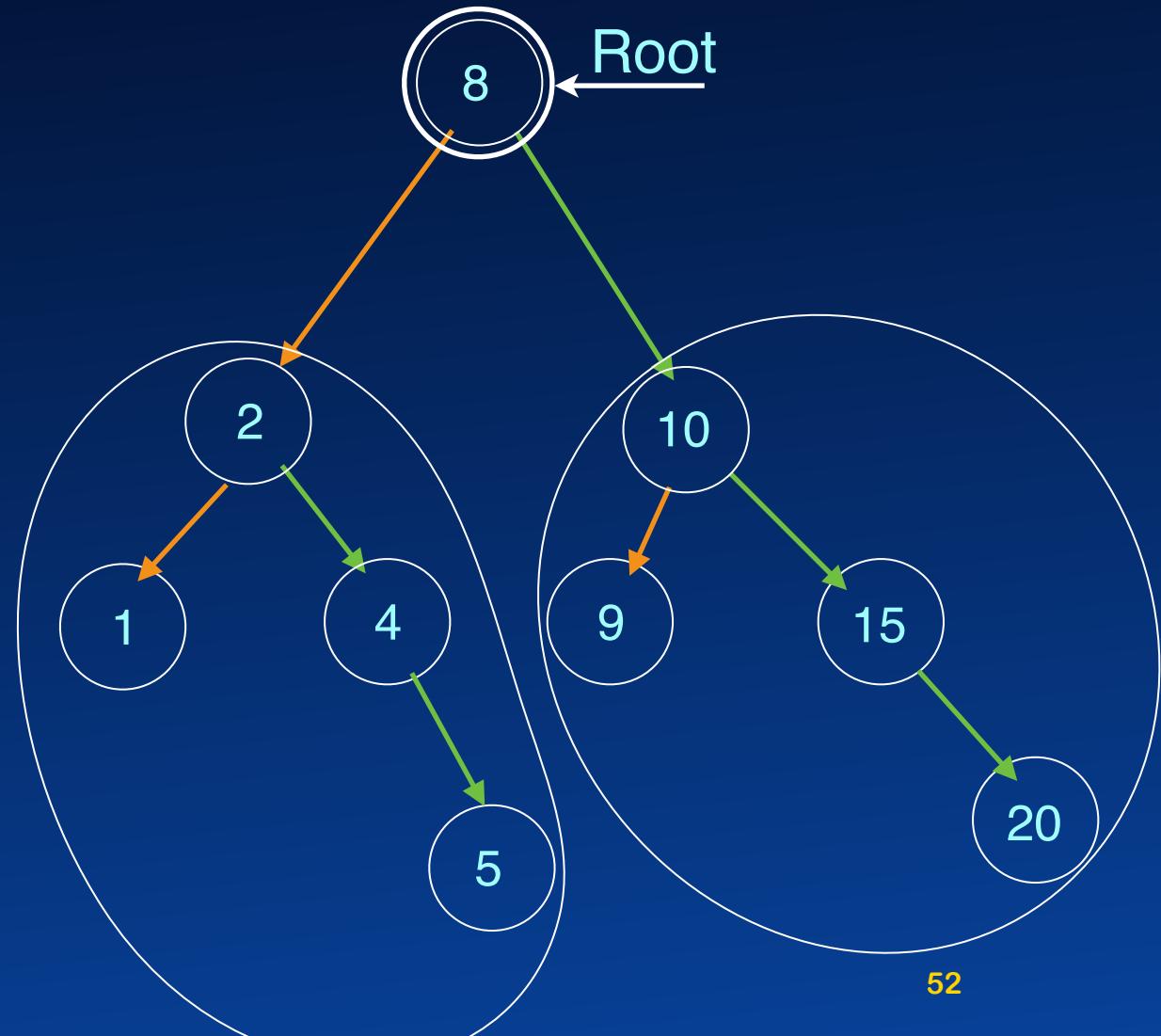


Tree Balance



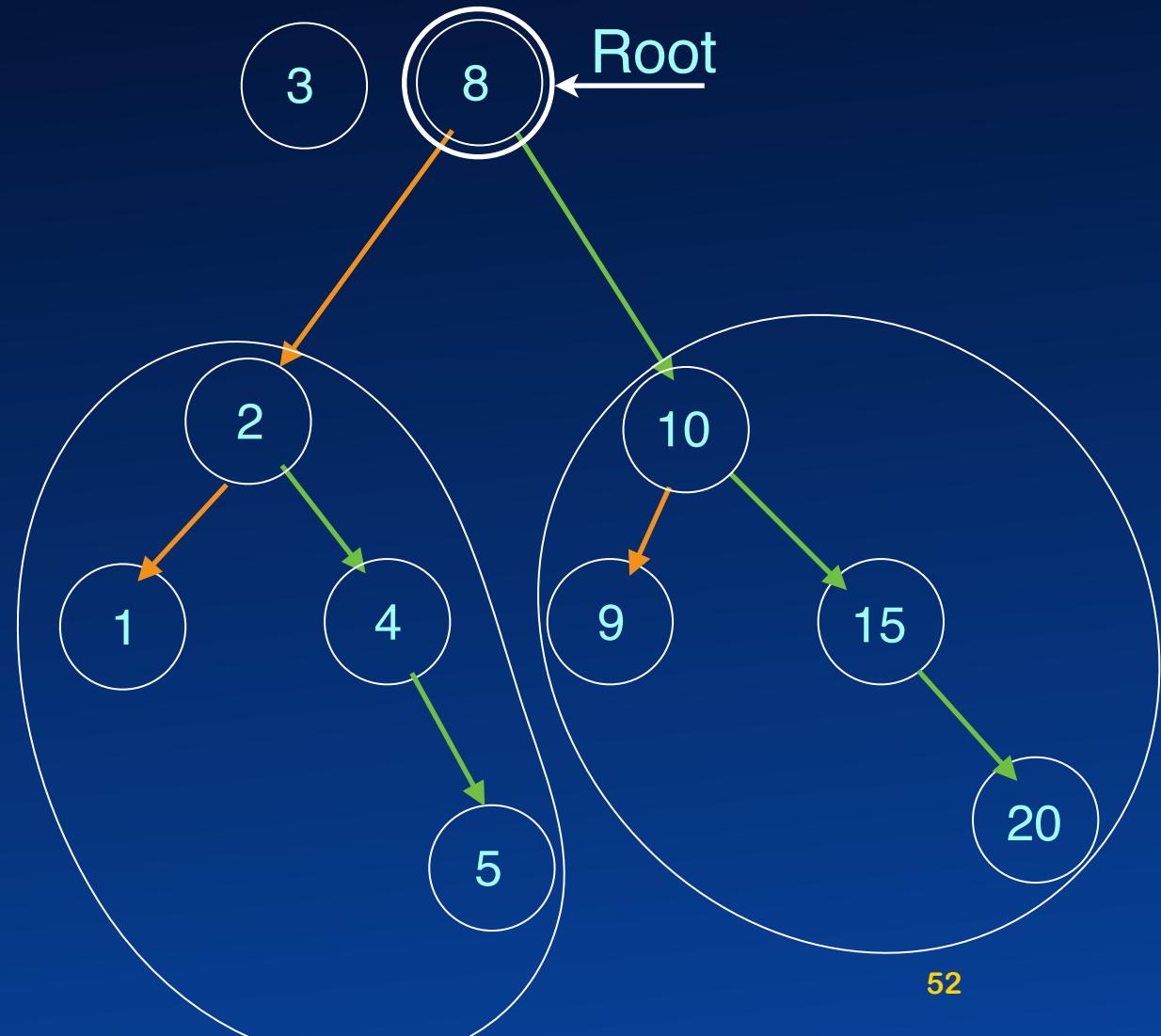


Tree Balance



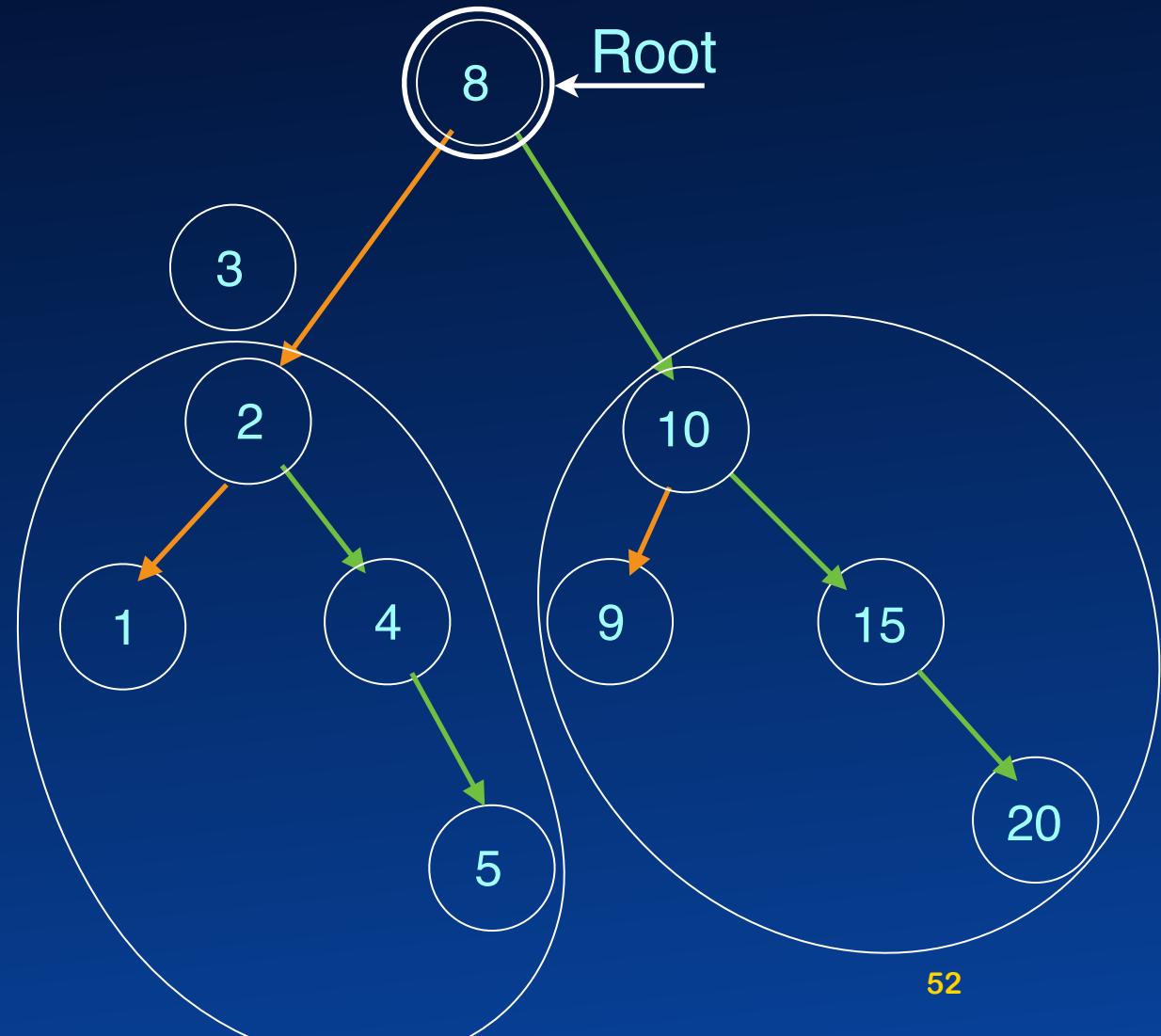


Tree Balance



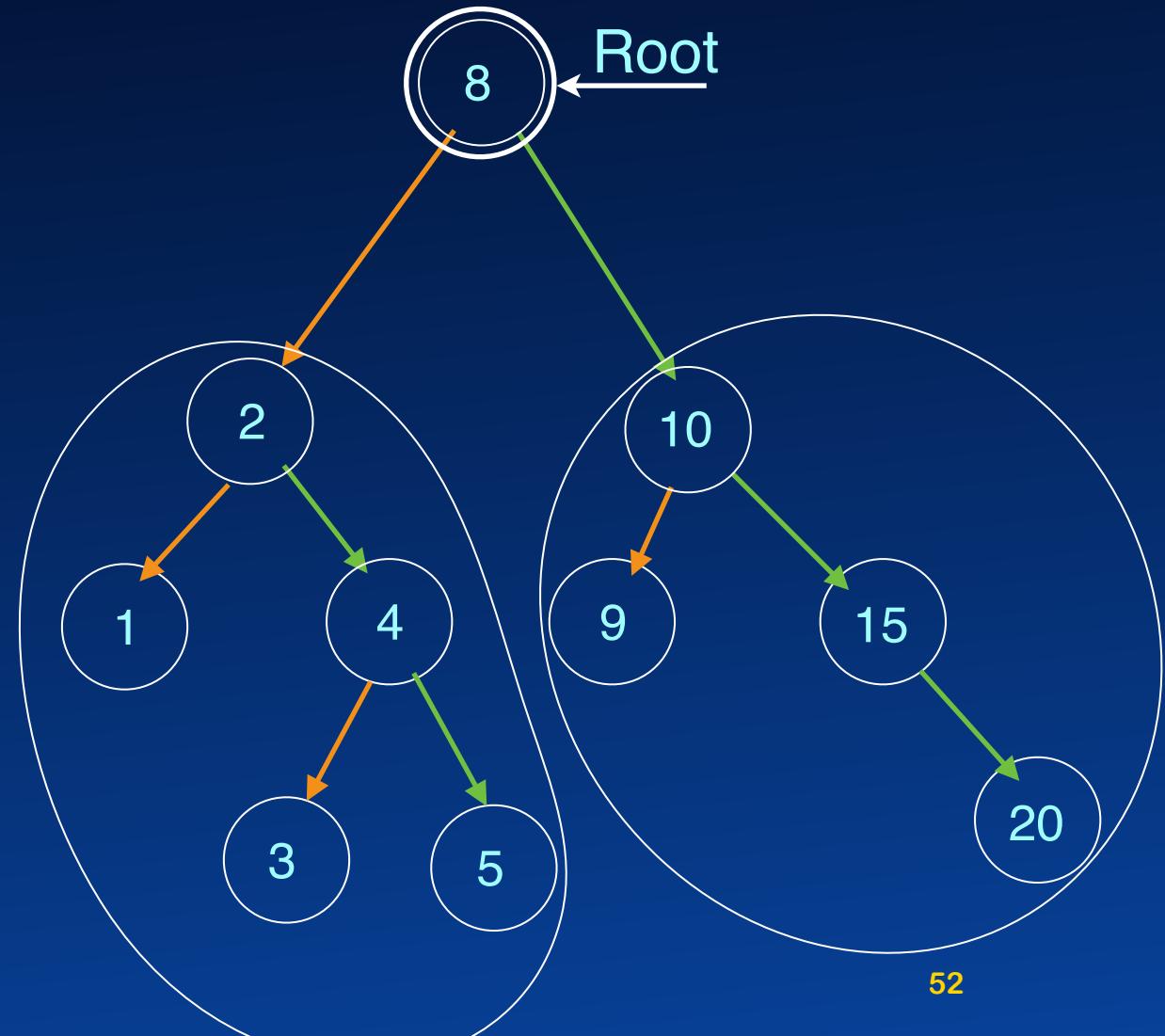


Tree Balance



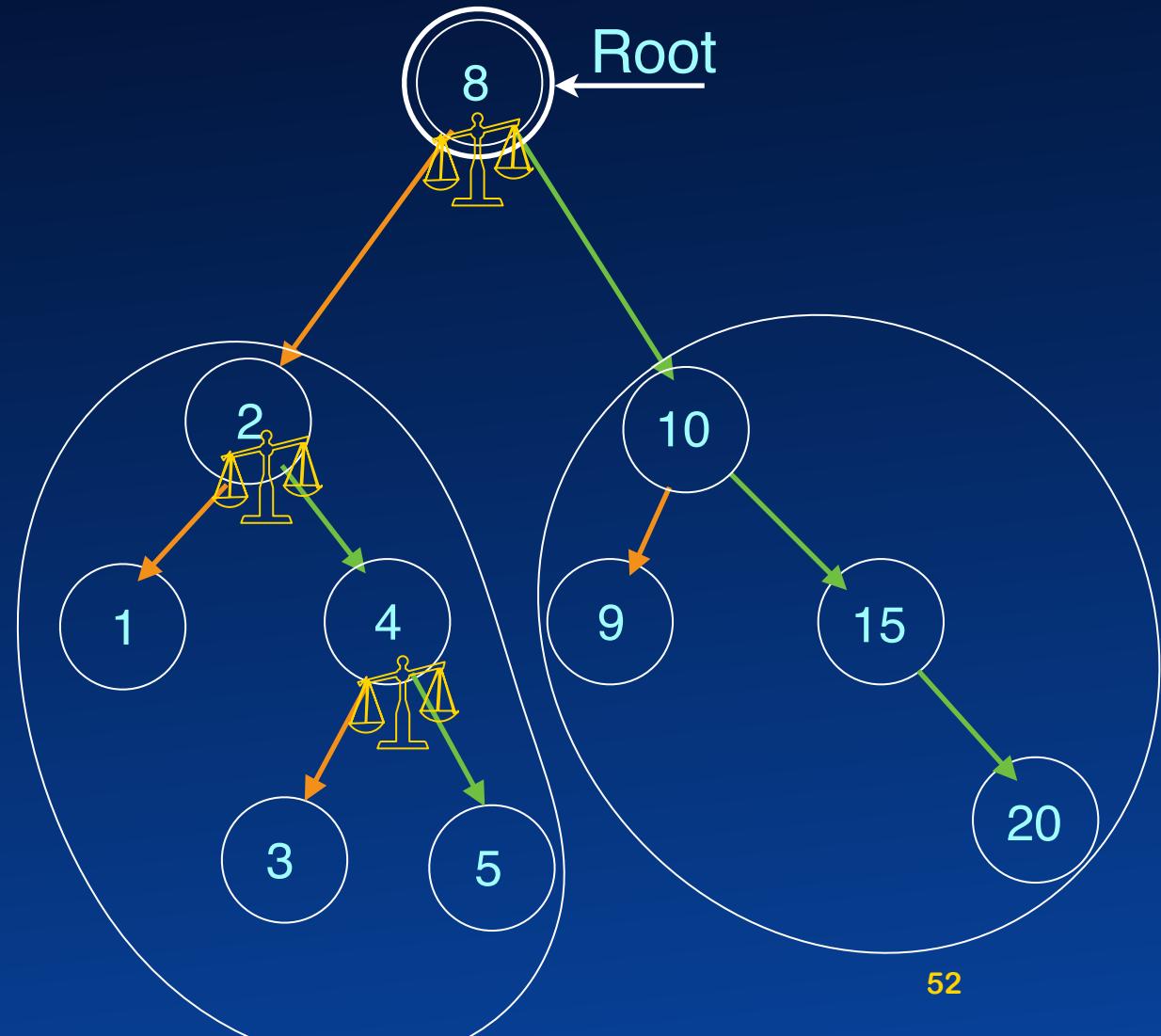


Tree Balance



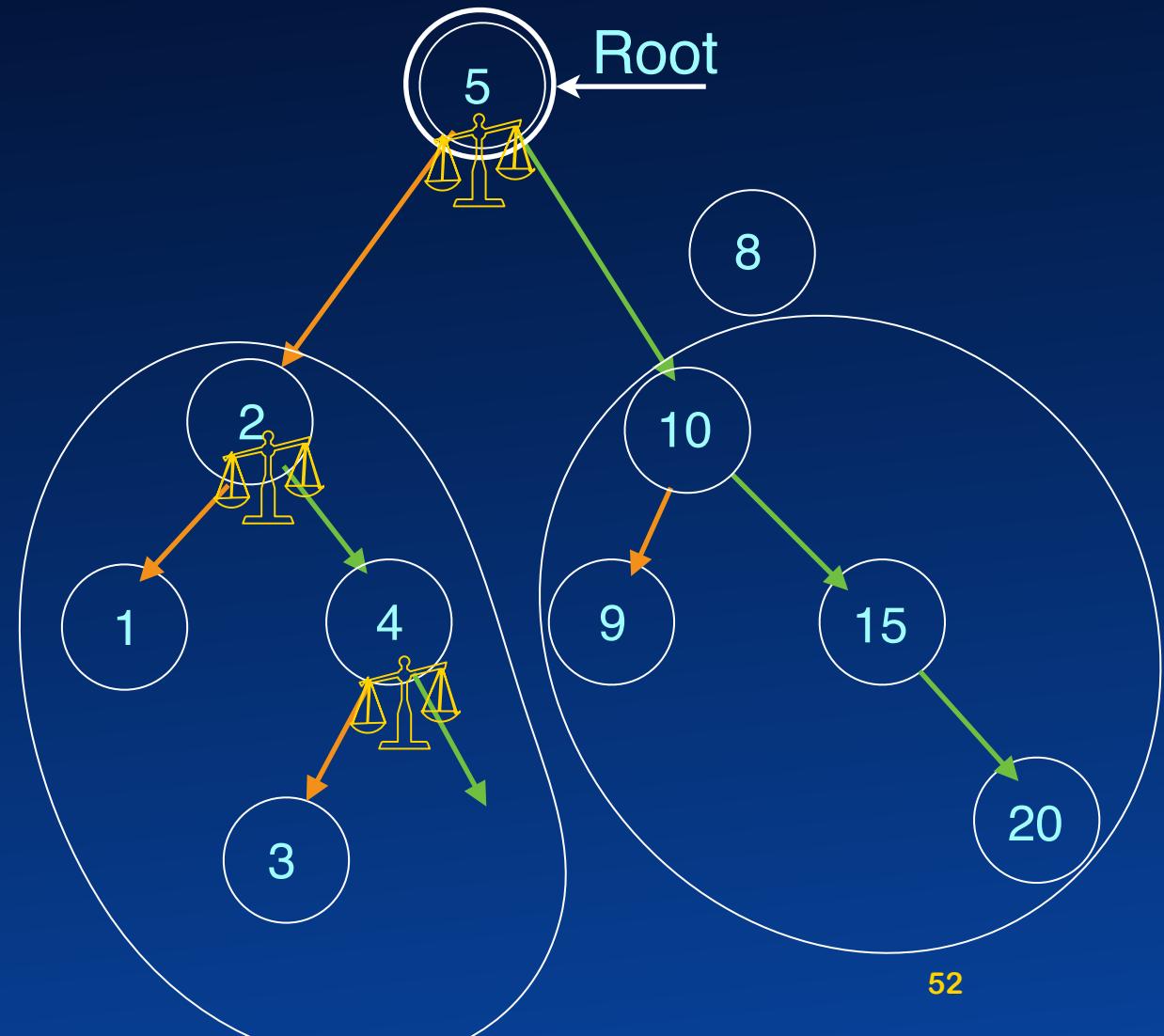


Tree Balance



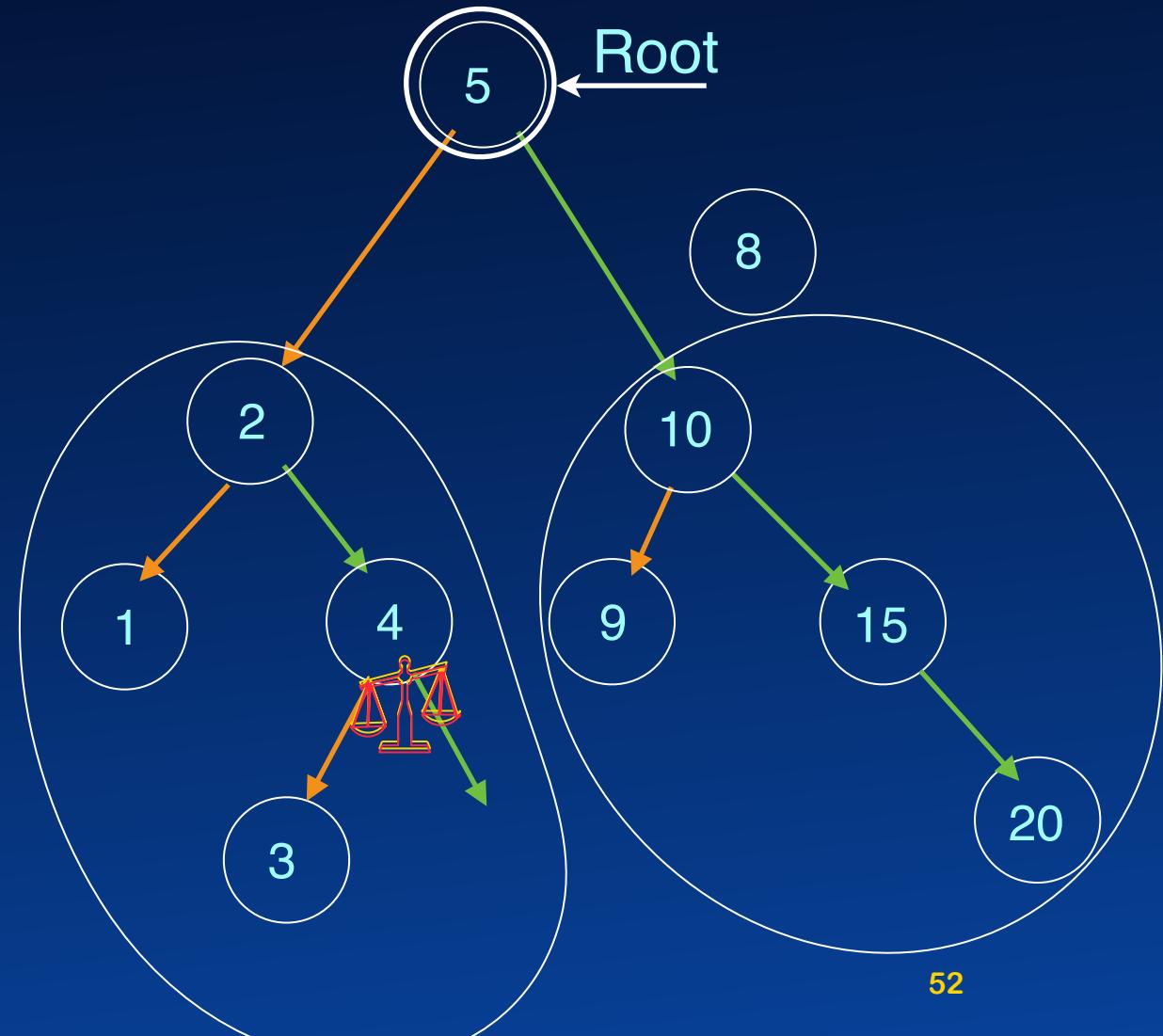


Tree Balance



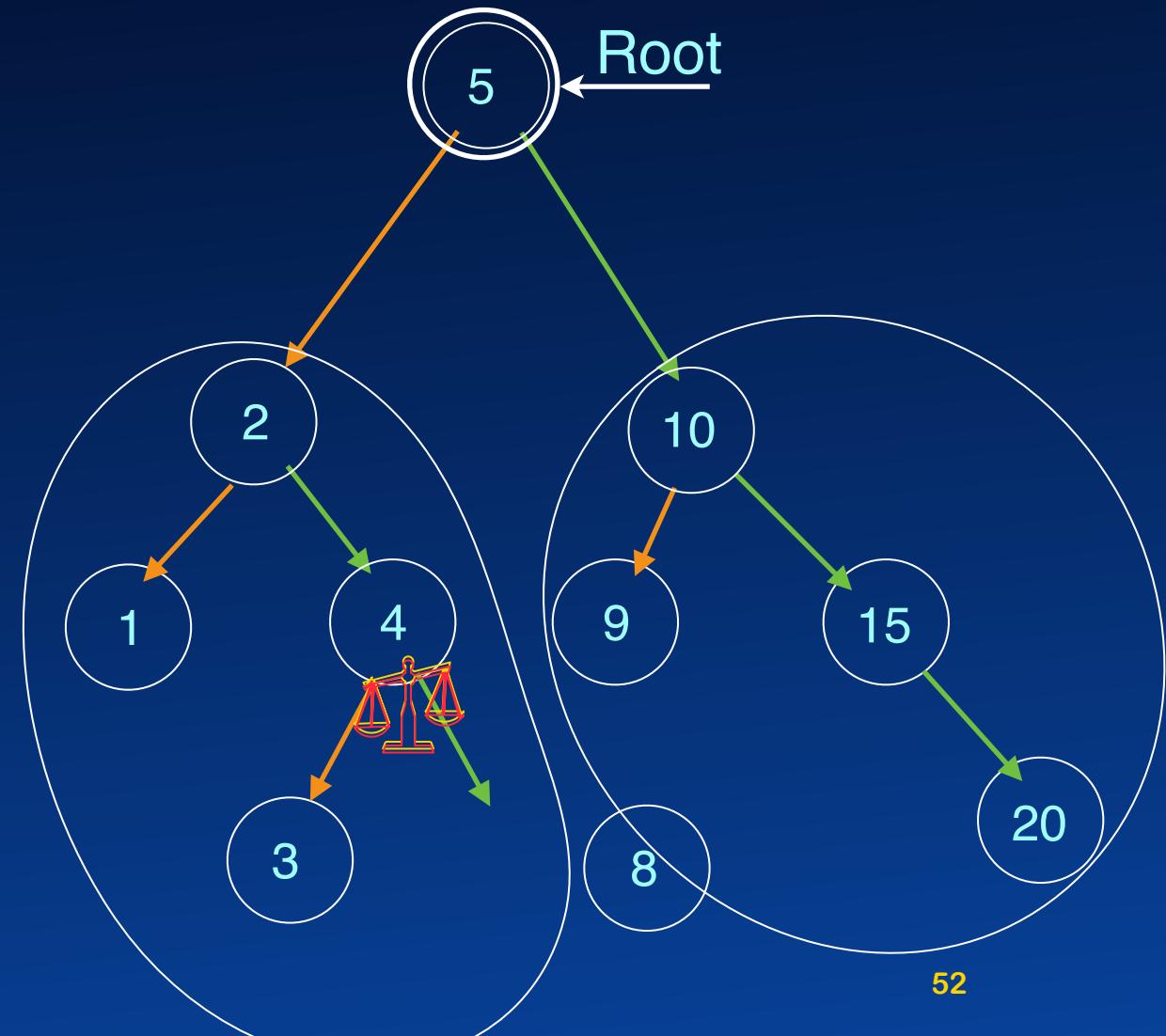


Tree Balance



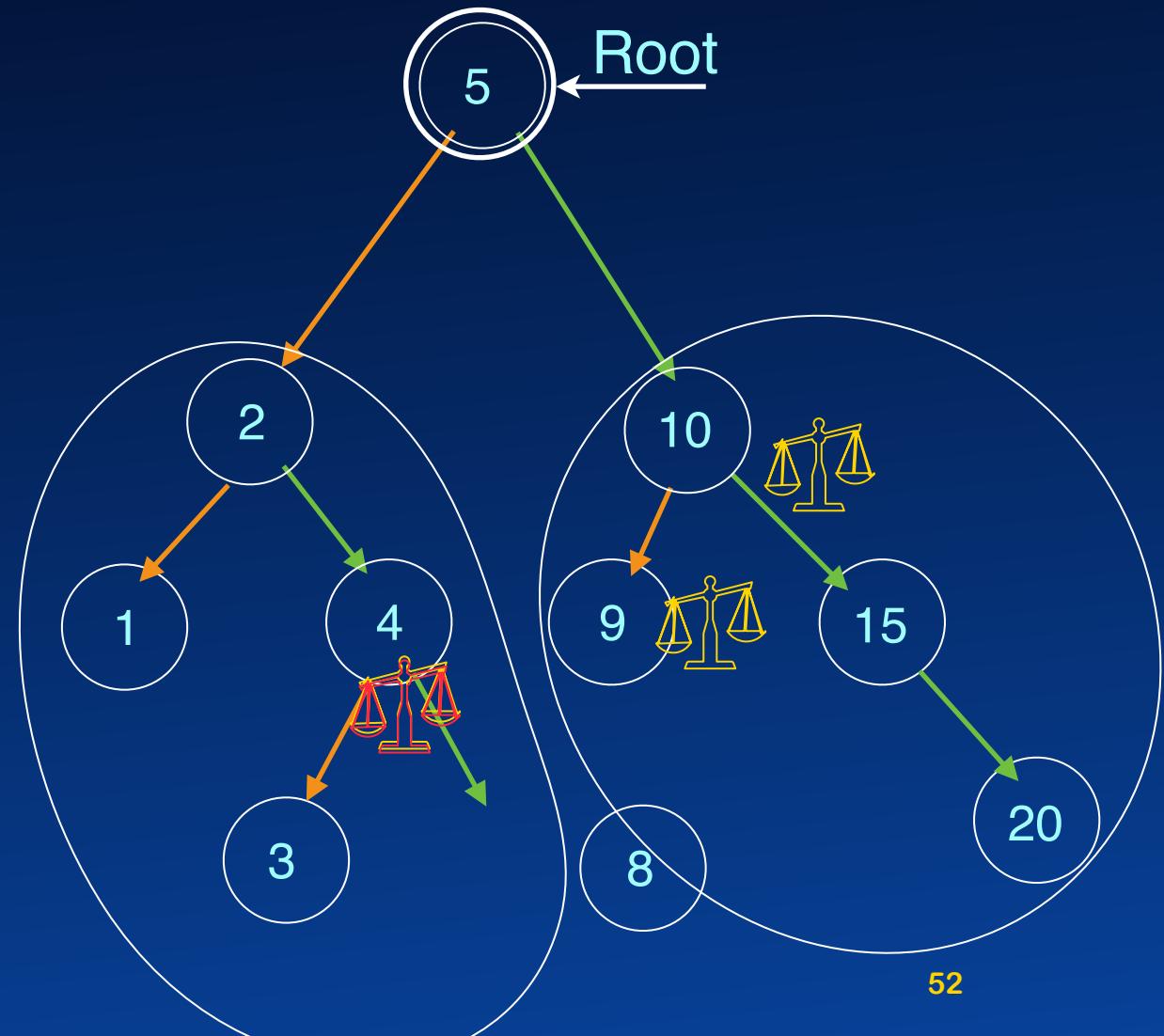


Tree Balance





Tree Balance





Balanced Tree

Count balanced (Weight balanced):

Weight of left subtree is at least α times the right subtree's

$$\alpha < .29$$

Height balanced:

Height of left subtree is within ± 1 of that of right subtree



Balanced Tree

Count balanced (Weight balanced):

Weight of left subtree is at least α times the right subtree's

$$\alpha < .29$$

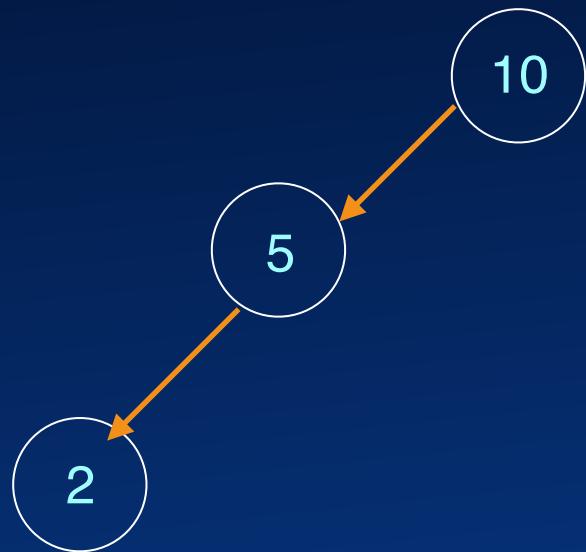
Height balanced:

Height of left subtree is within ± 1 of that of right subtree

AVL Tree

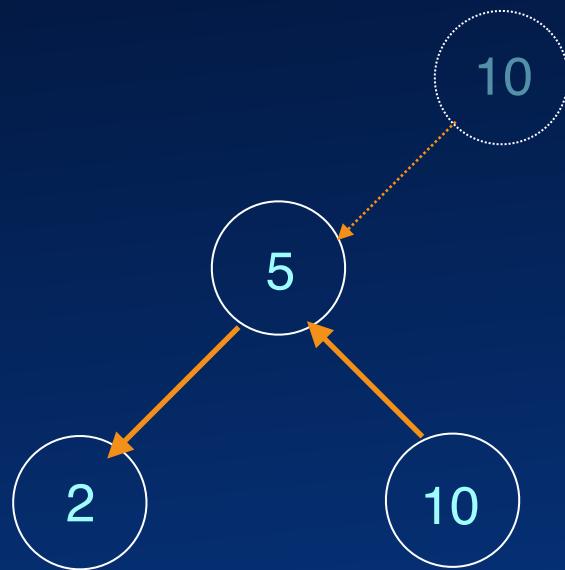


Rotation



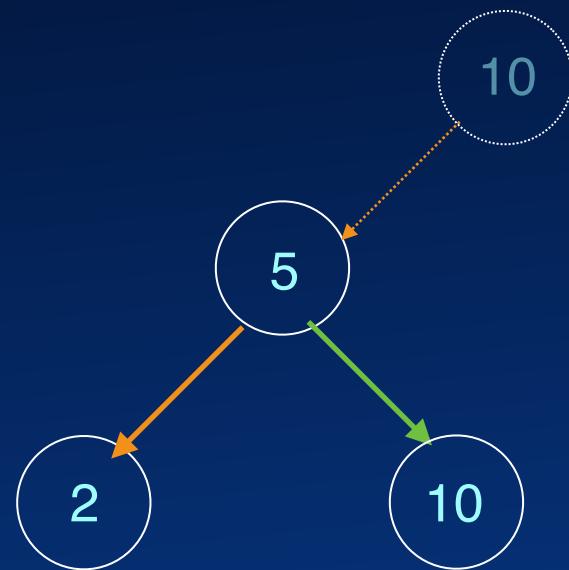


Rotation



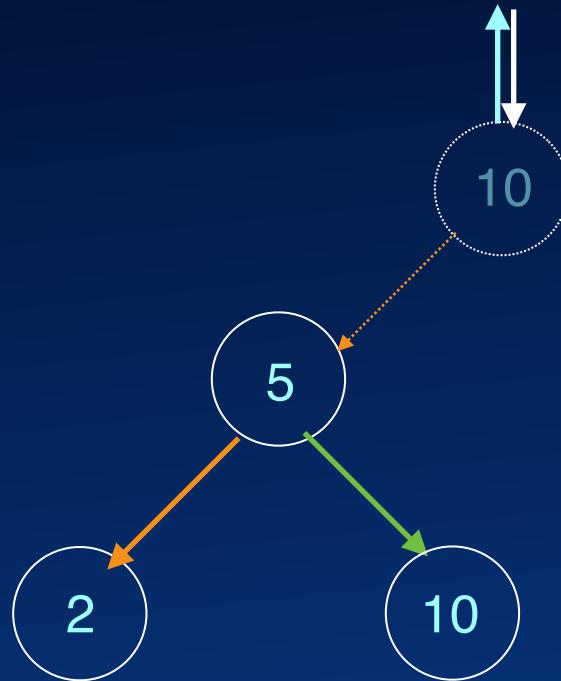


Rotation



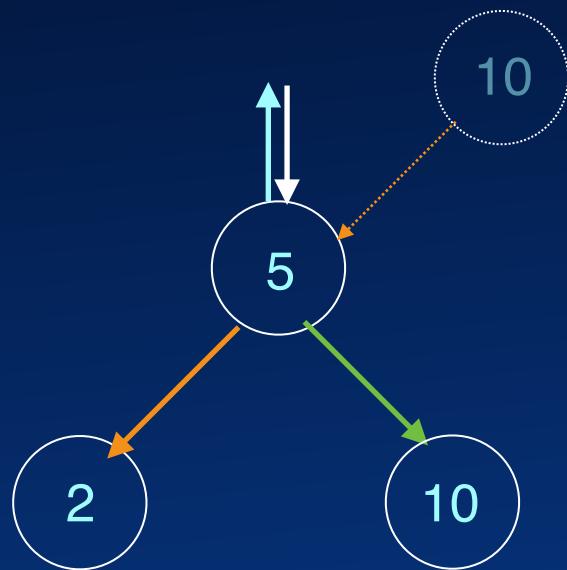


Rotation



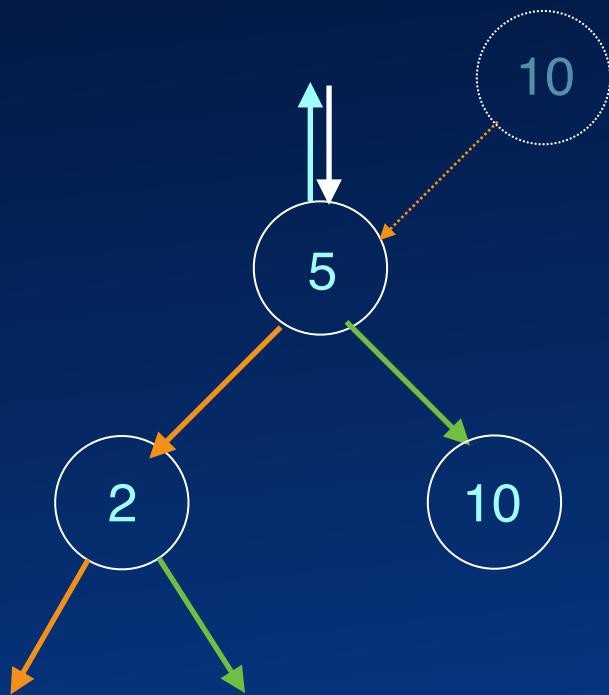


Rotation



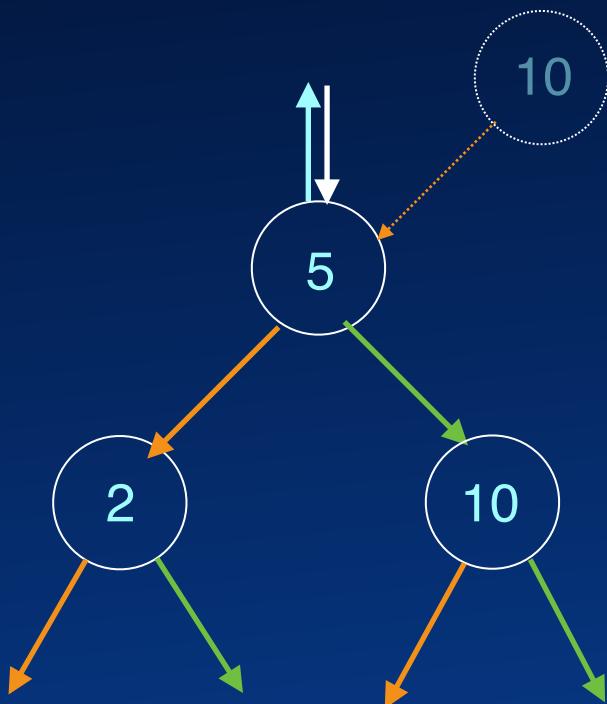


Rotation



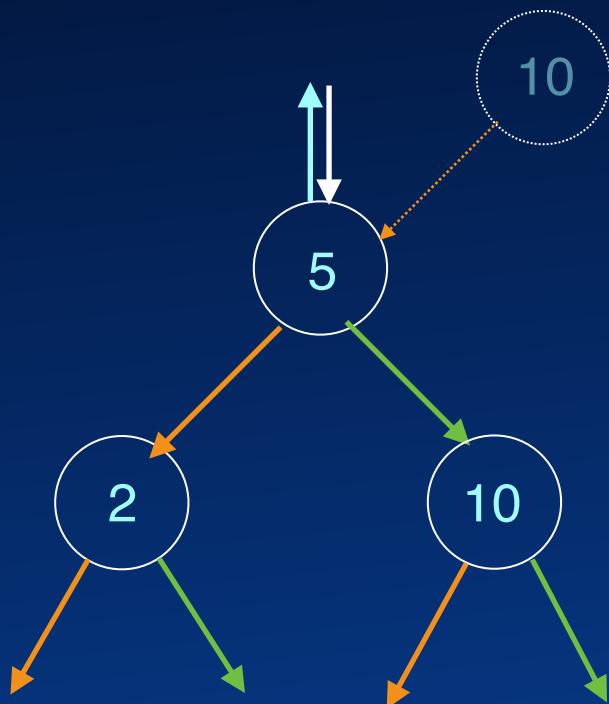


Rotation



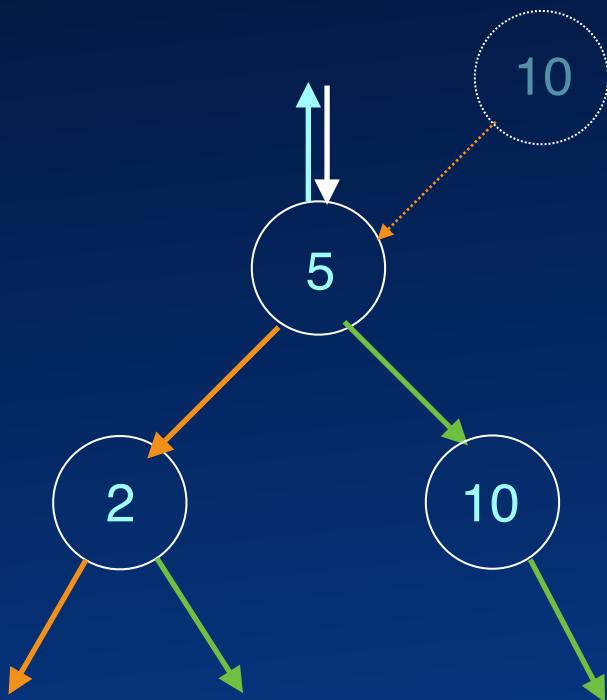


Rotation



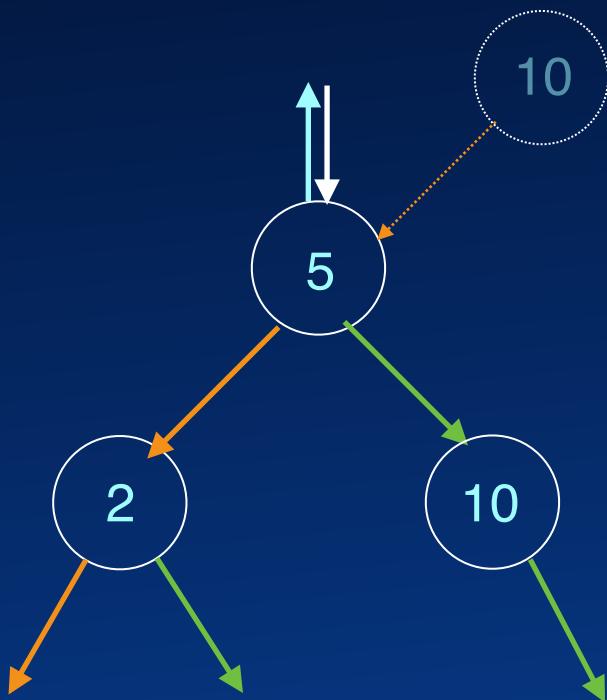


Rotation



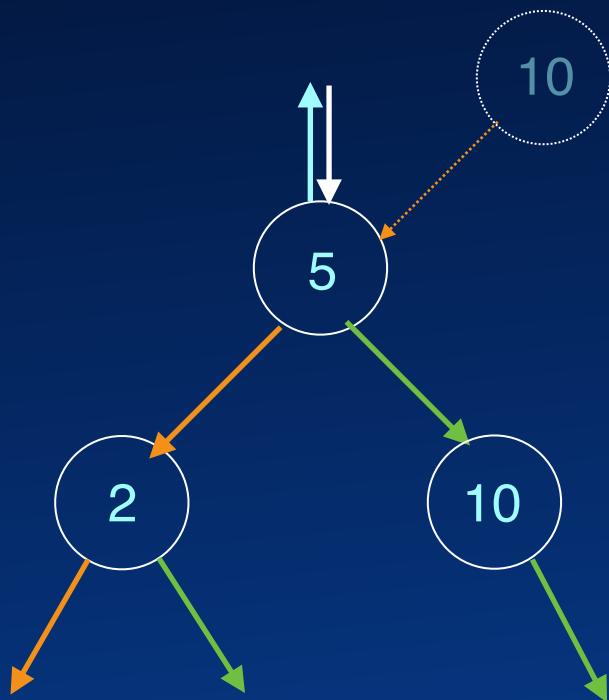


Rotation



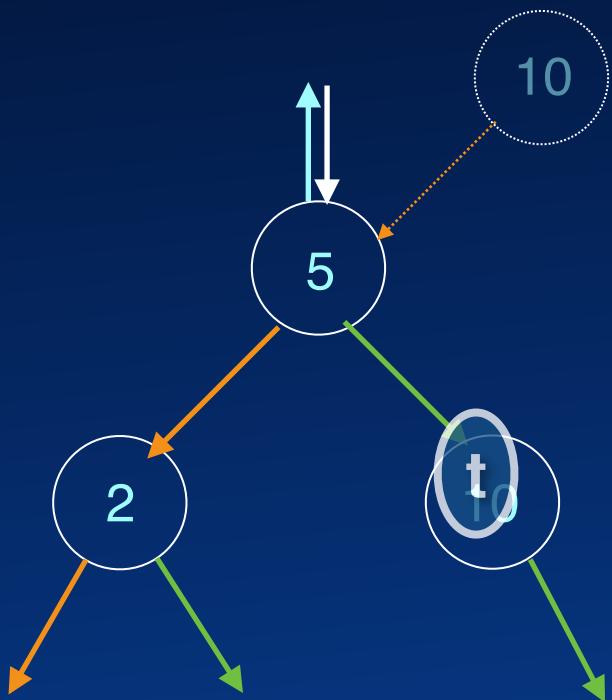


Rotation



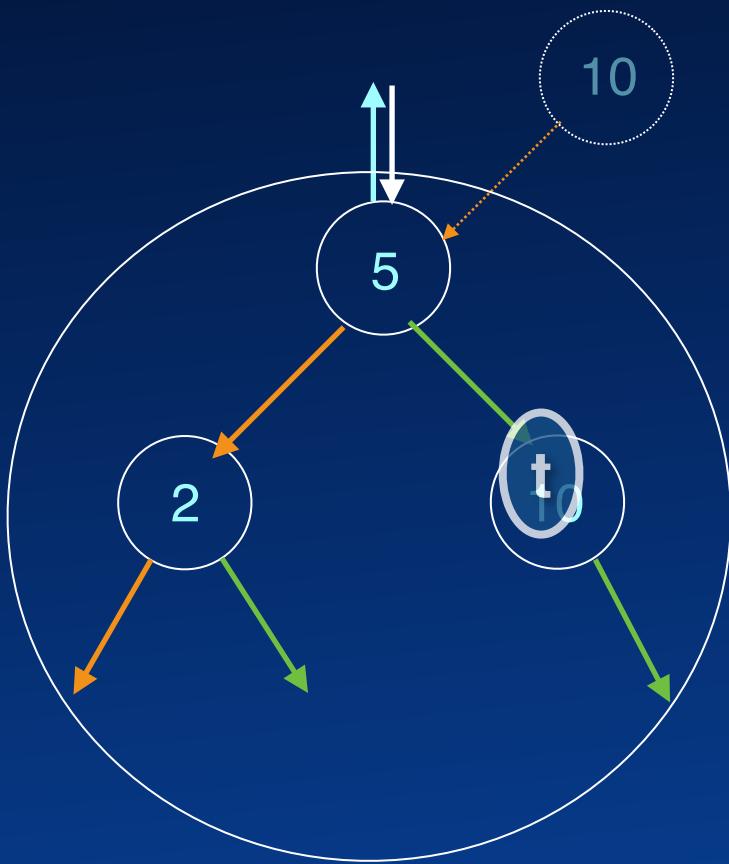


Rotation



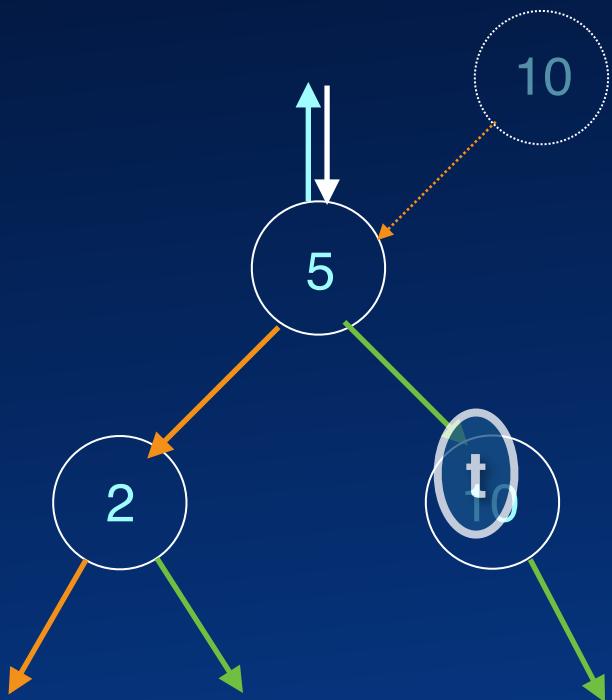


Rotation



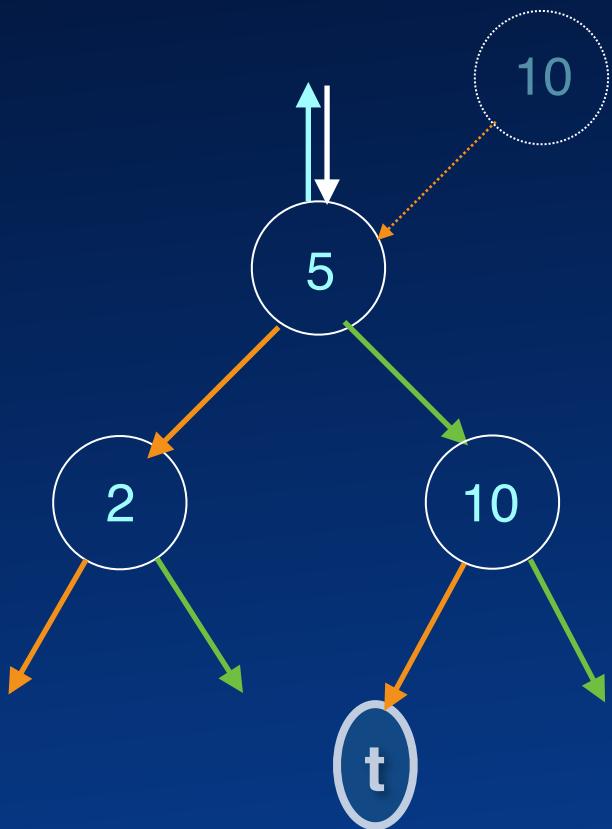


Rotation



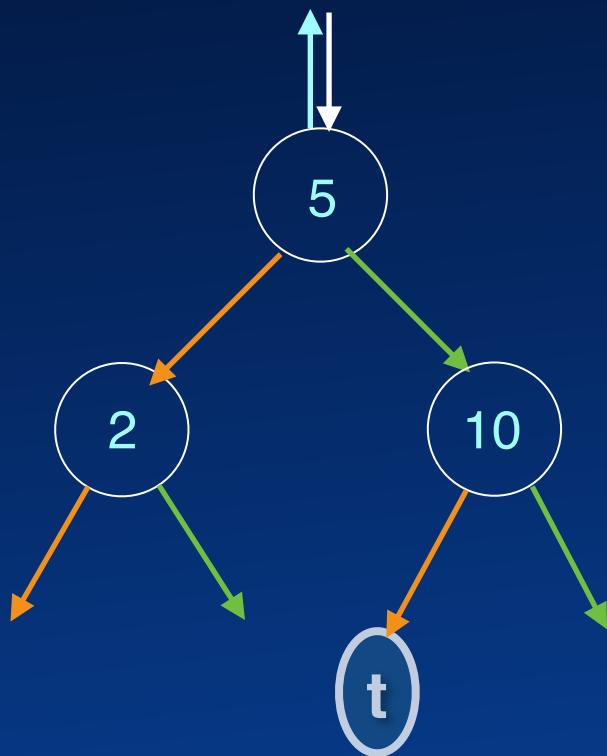


Rotation



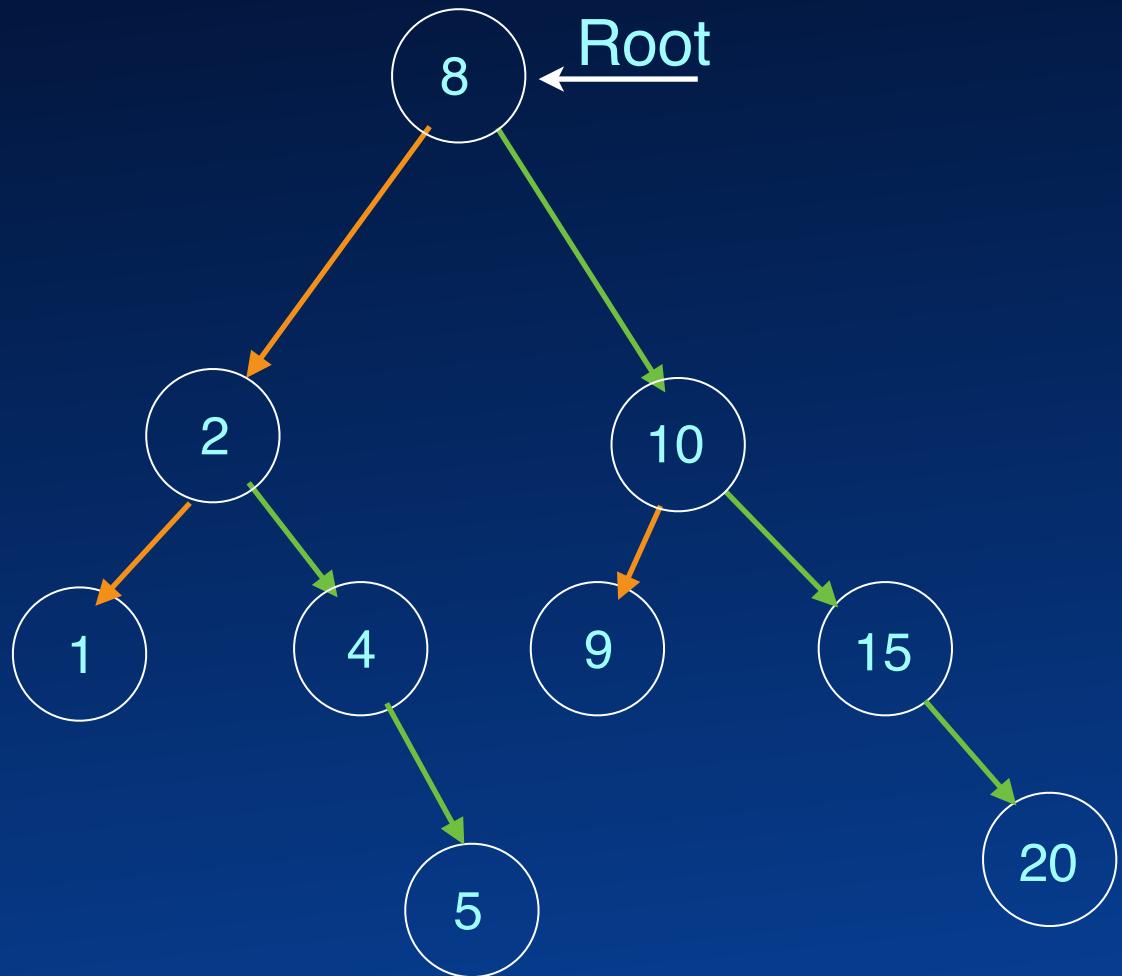


Rotation



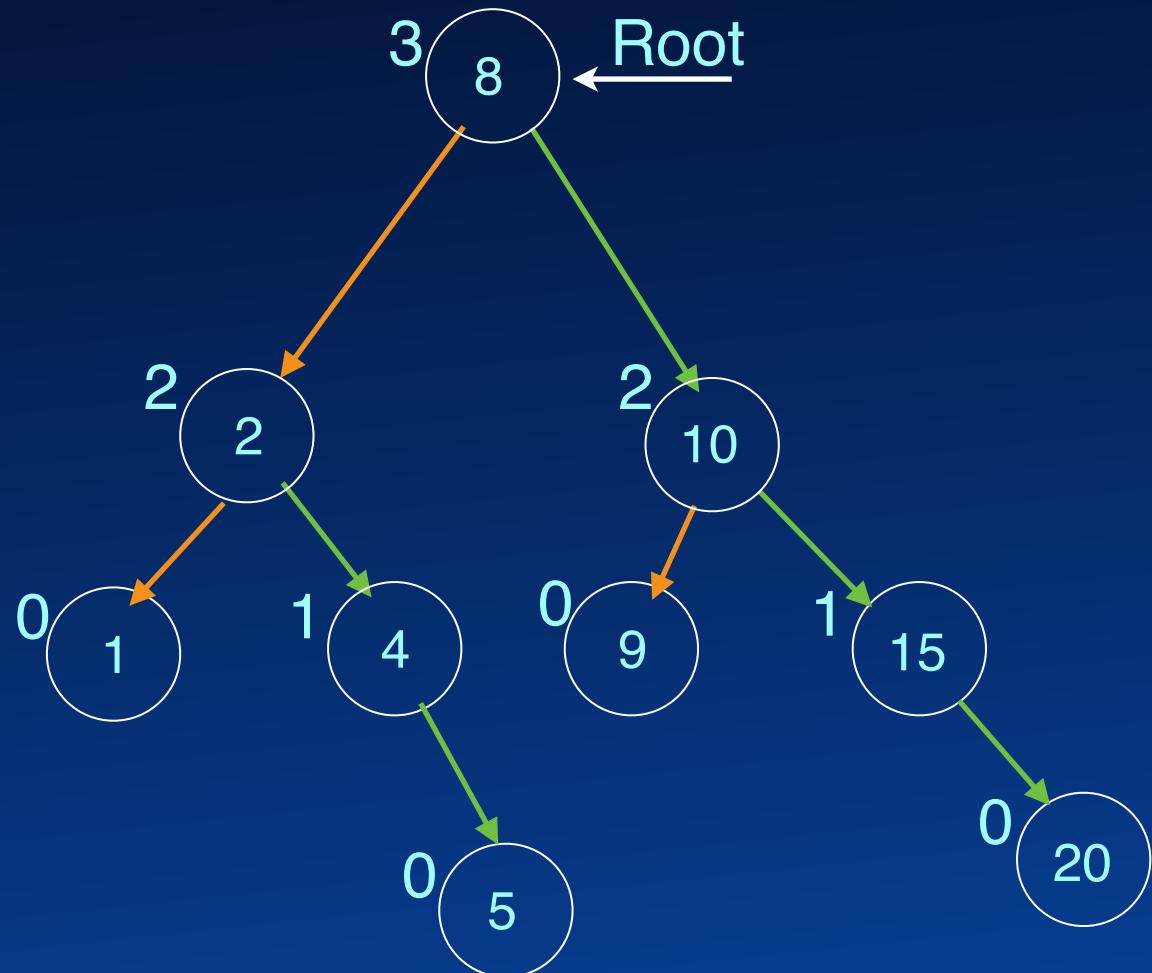


AVL Tree Height



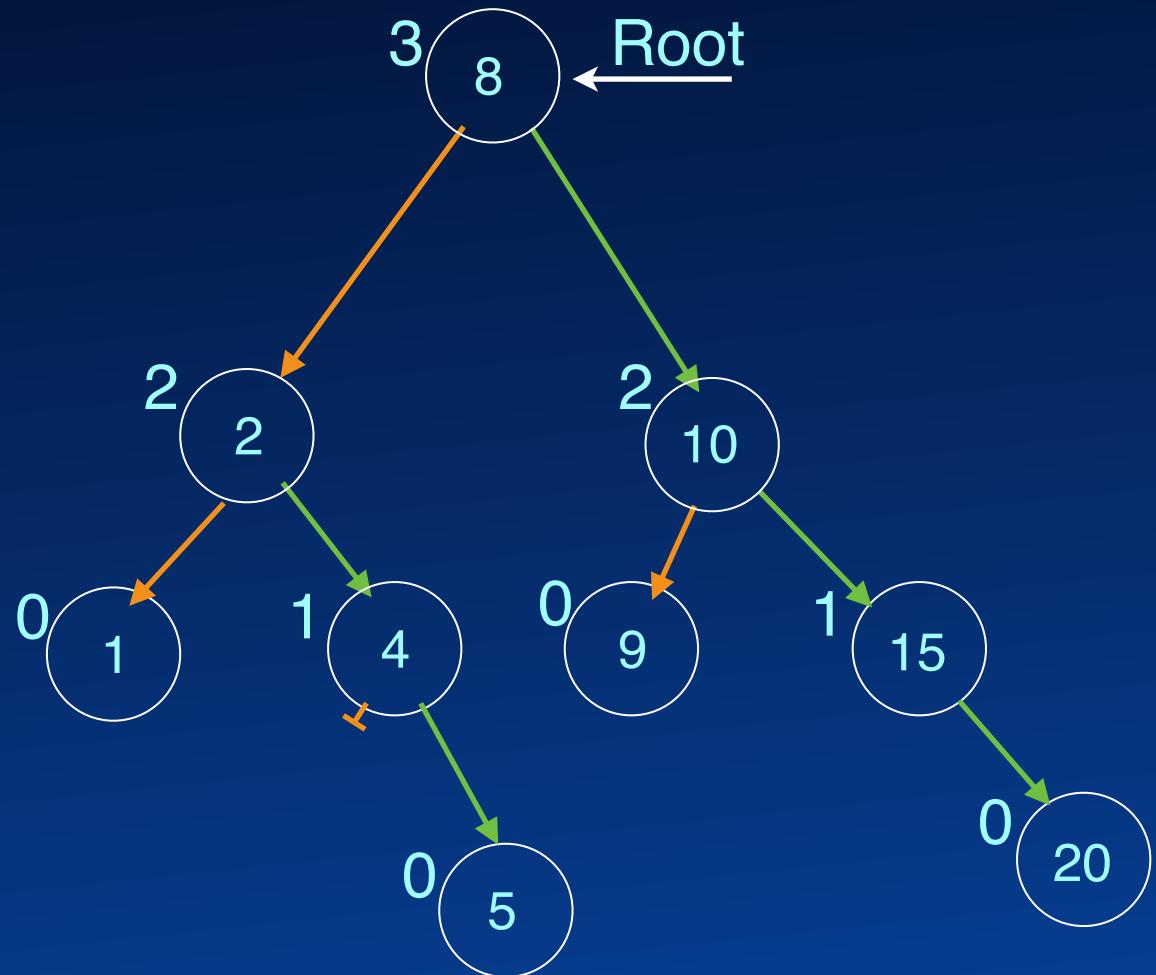


AVL Tree Height



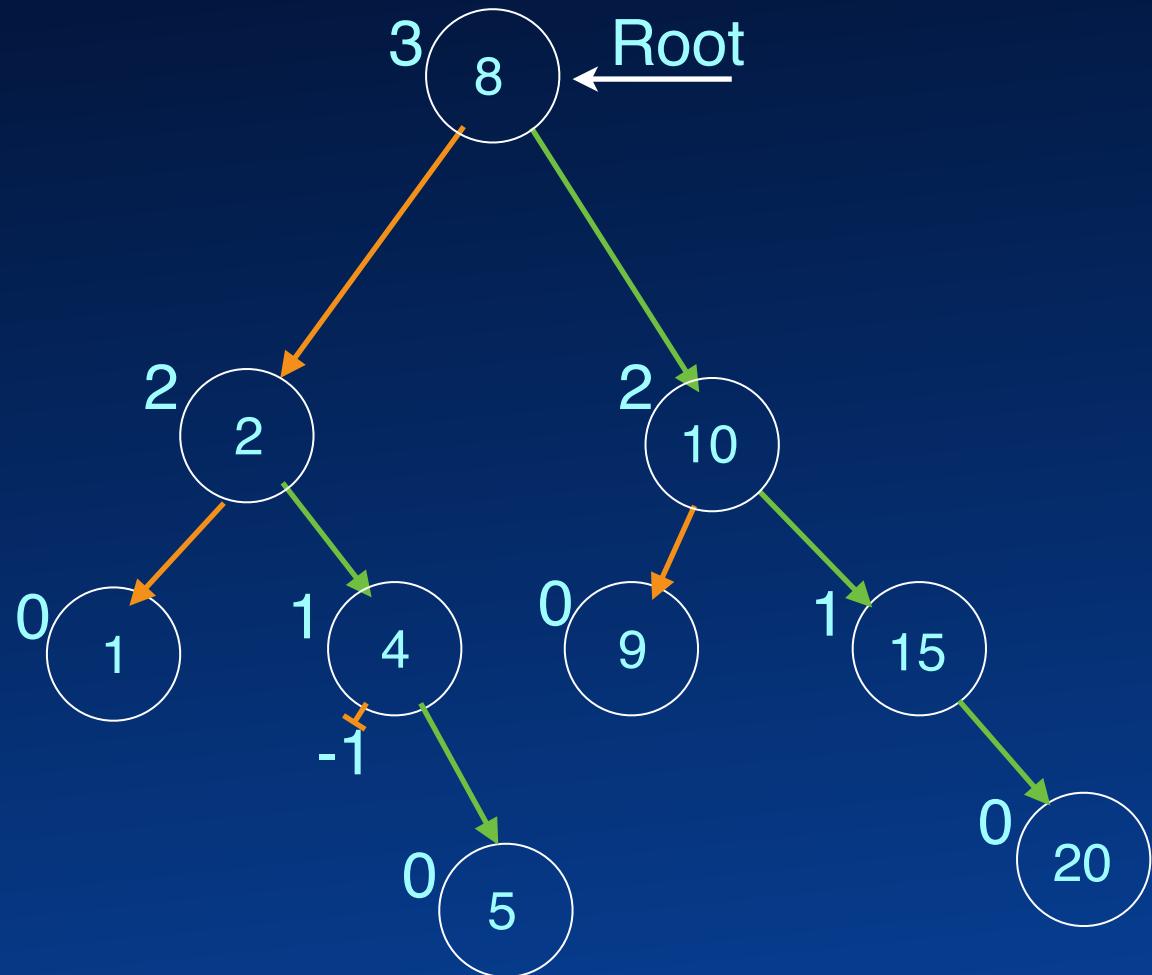


AVL Tree Height





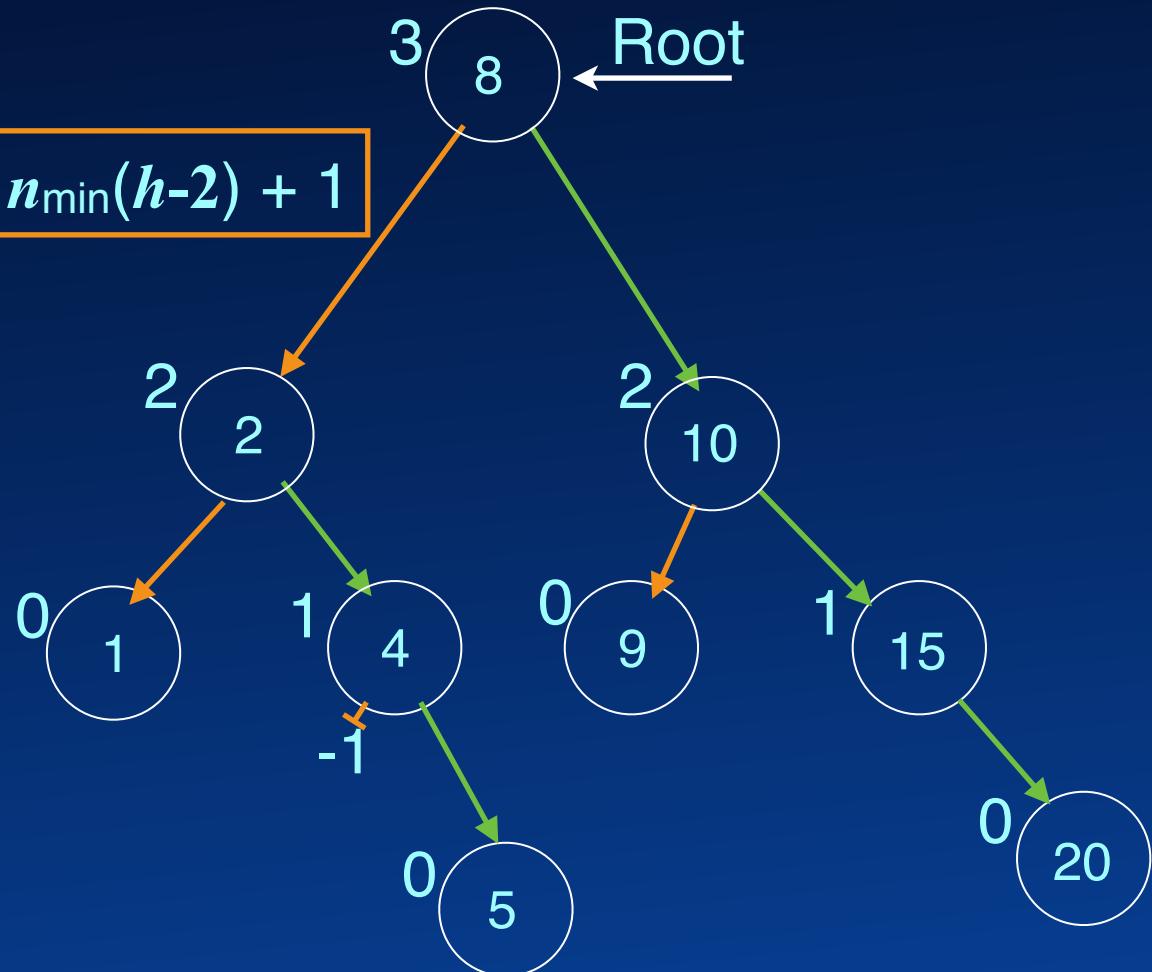
AVL Tree Height





AVL Tree Height

$$n_{\min}(h) = n_{\min}(h-1) + n_{\min}(h-2) + 1$$



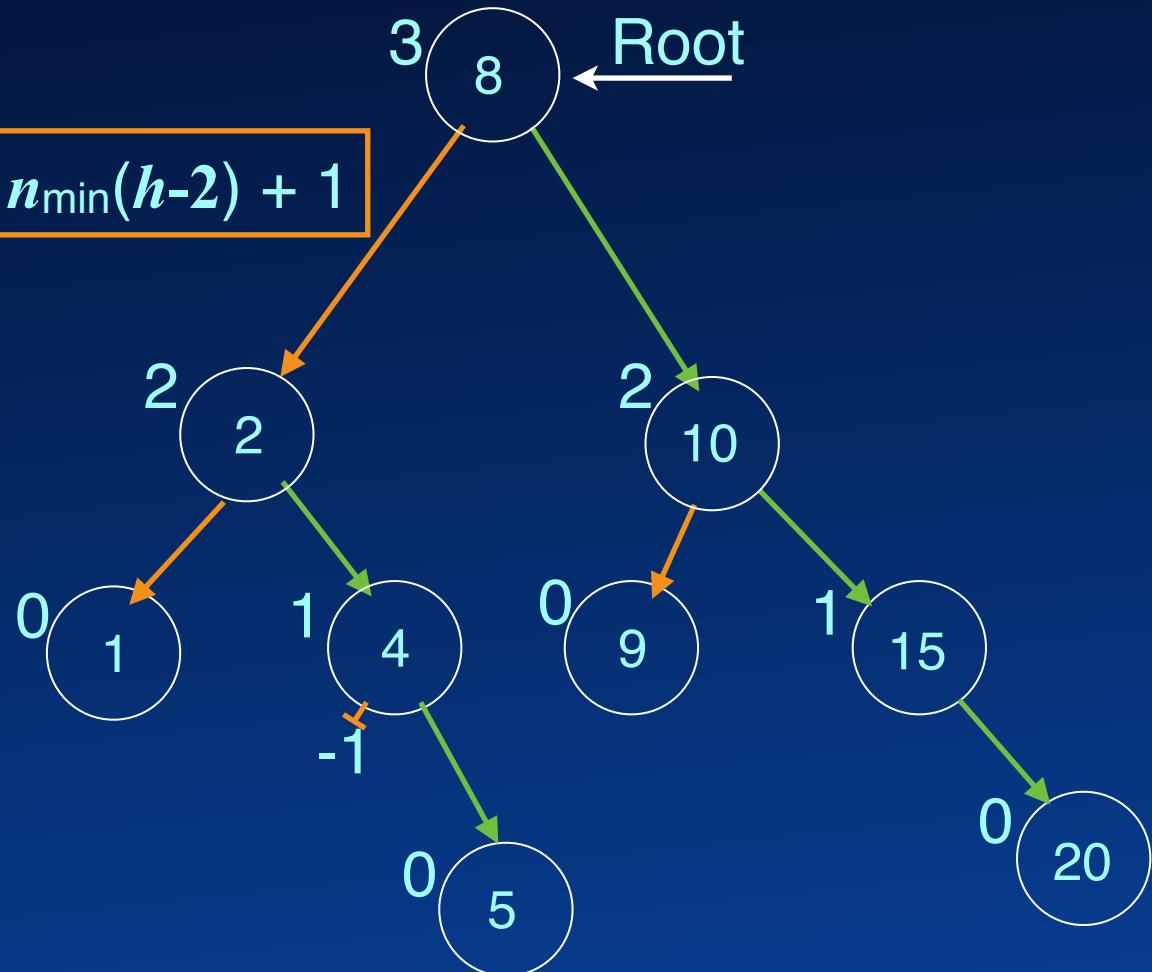


AVL Tree Height

$$n_{\min}(h) = n_{\min}(h-1) + n_{\min}(h-2) + 1$$

$$n_{\min}(0) = 1$$

$$n_{\min}(-1) = 0$$





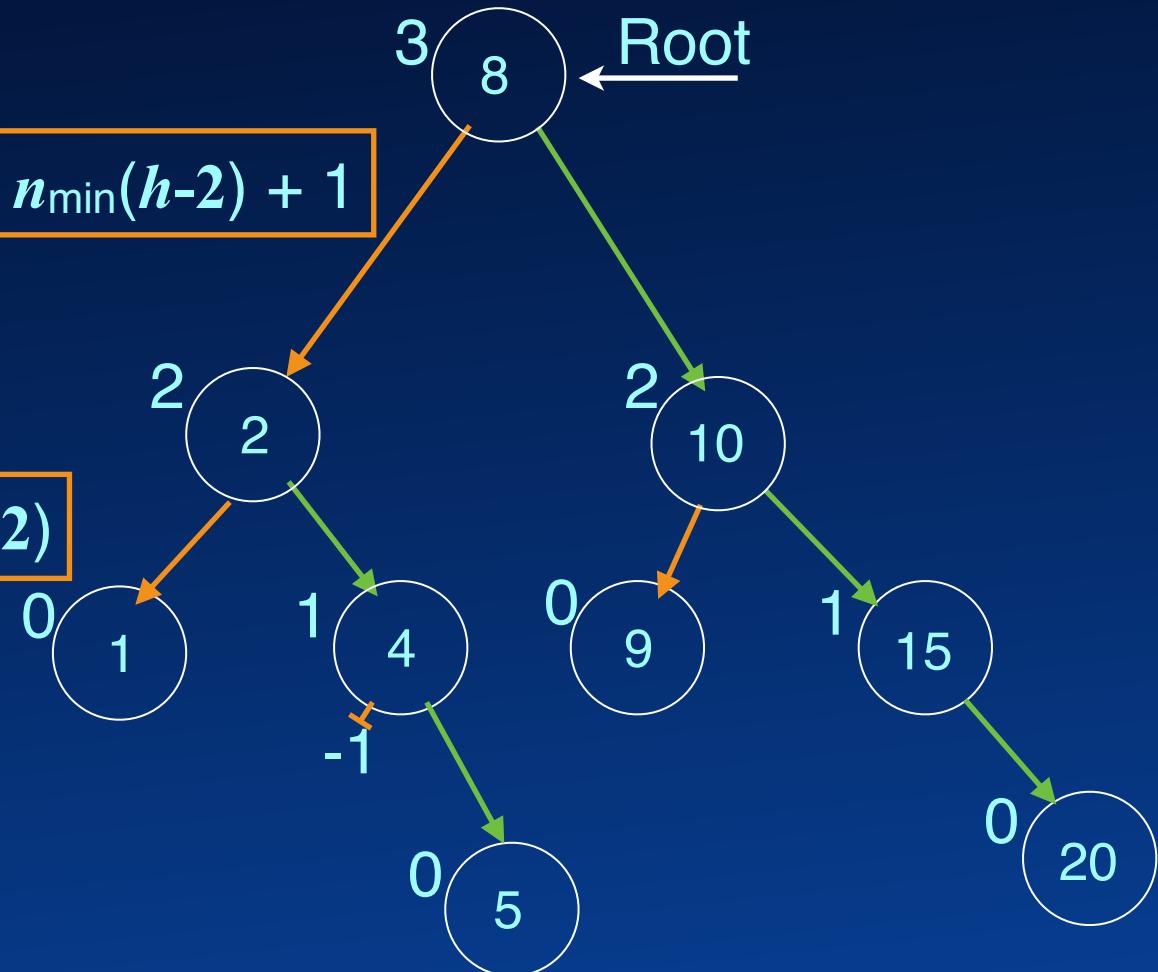
AVL Tree Height

$$n_{\min}(h) = n_{\min}(h-1) + n_{\min}(h-2) + 1$$

$$n_{\min}(0) = 1$$

$$n_{\min}(-1) = 0$$

$$n_{\min}(h) > 2 n_{\min}(h-2)$$





AVL Tree Height

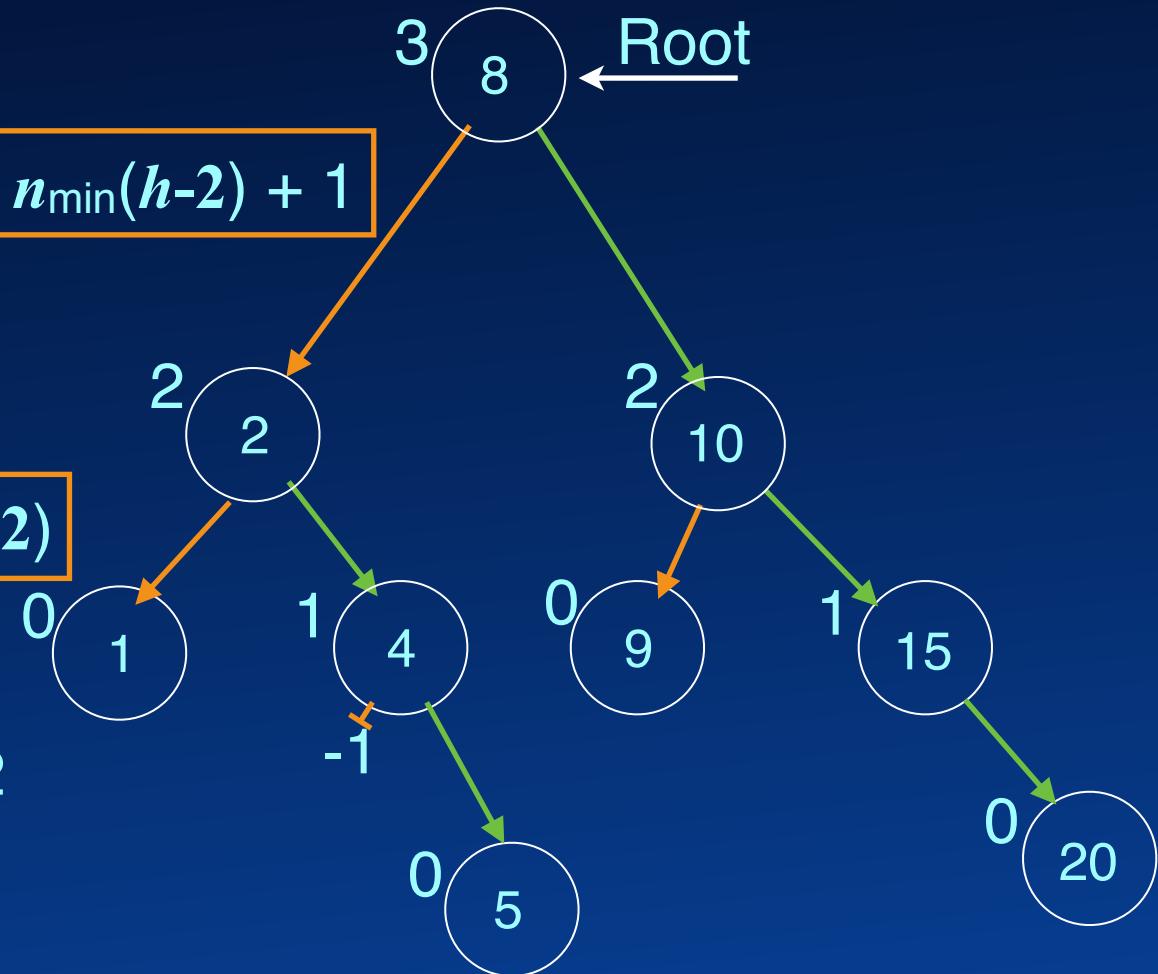
$$n_{\min}(h) = n_{\min}(h-1) + n_{\min}(h-2) + 1$$

$$n_{\min}(0) = 1$$

$$n_{\min}(-1) = 0$$

$$n_{\min}(h) > 2 n_{\min}(h-2)$$

$$n > 2^{(h+.328)*.694} - 2$$





AVL Tree Height

$$n_{\min}(h) = n_{\min}(h-1) + n_{\min}(h-2) + 1$$

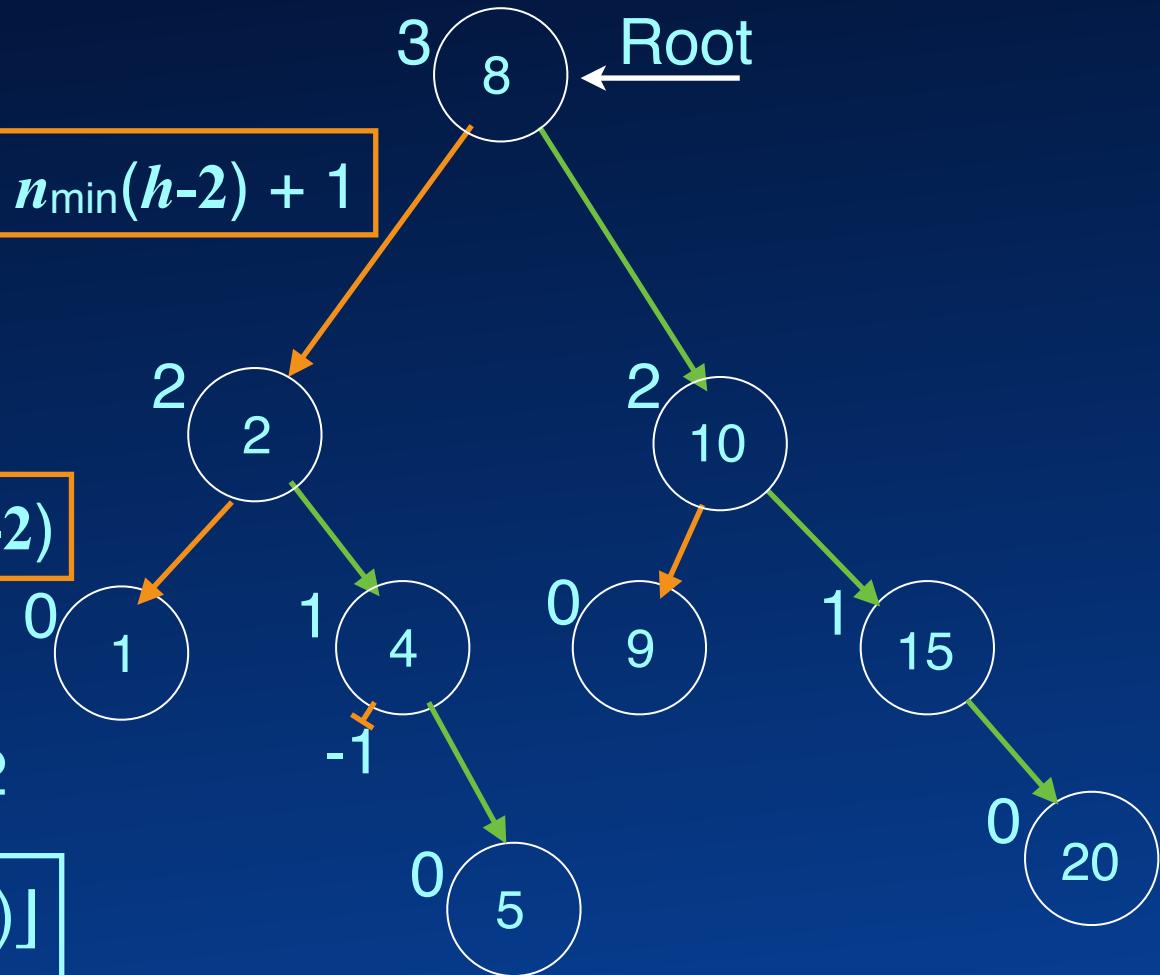
$$n_{\min}(0) = 1$$

$$n_{\min}(-1) = 0$$

$$n_{\min}(h) > 2 n_{\min}(h-2)$$

$$n > 2^{(h+.328)*.694} - 2$$

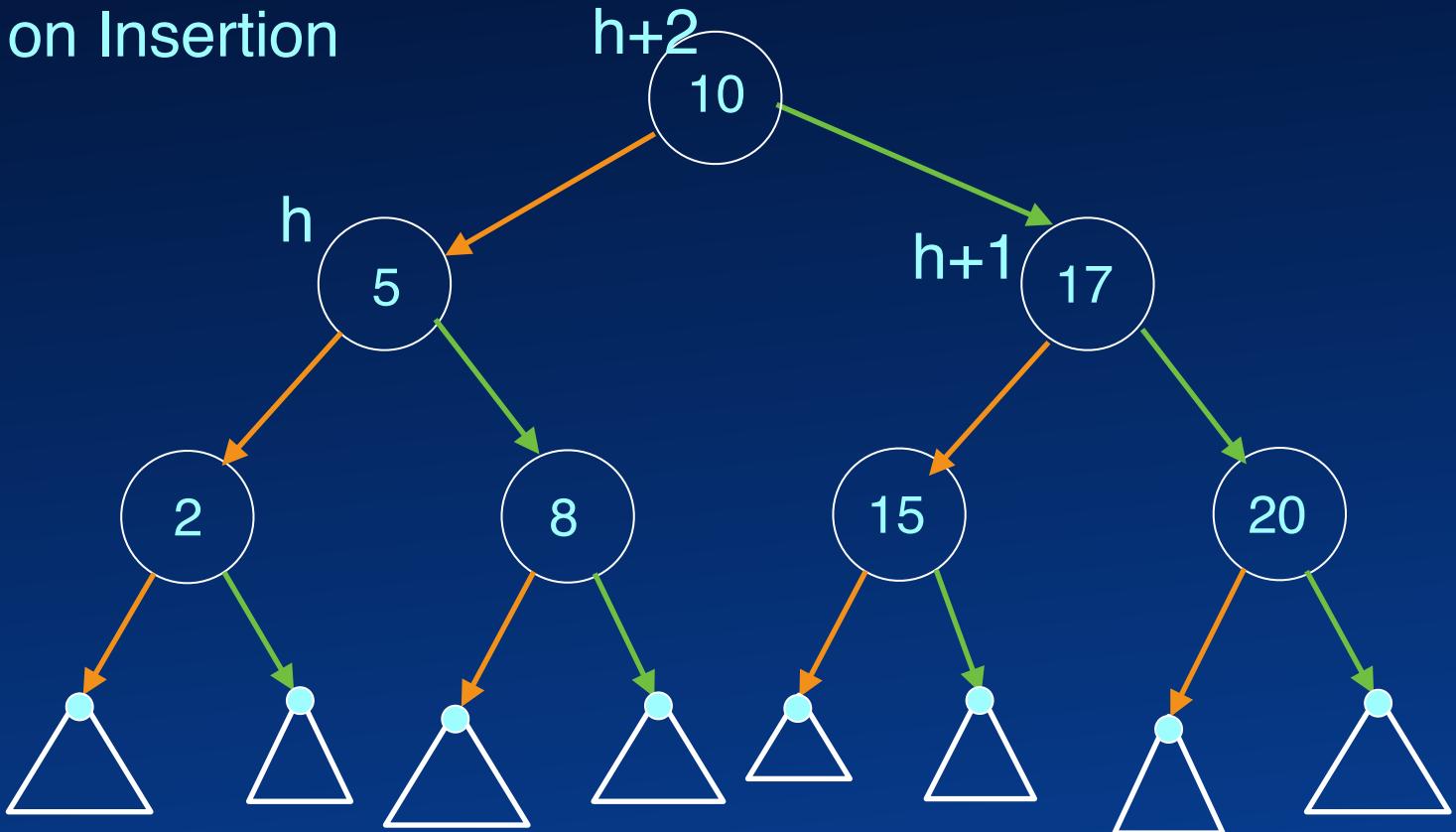
$$h < \lfloor 1.44 \log(n+2) \rfloor$$





Rotation

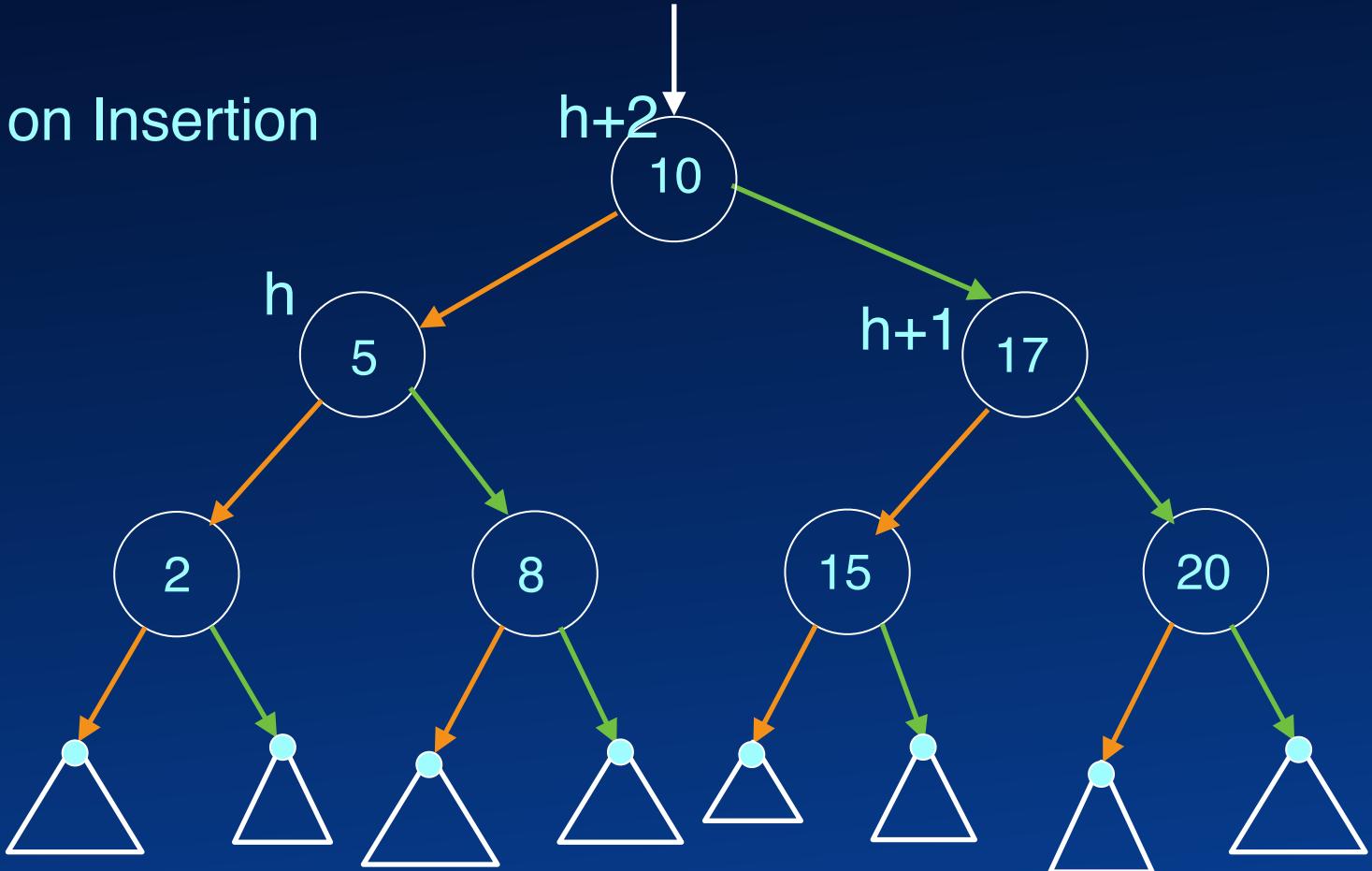
Imbalance on Insertion





Rotation

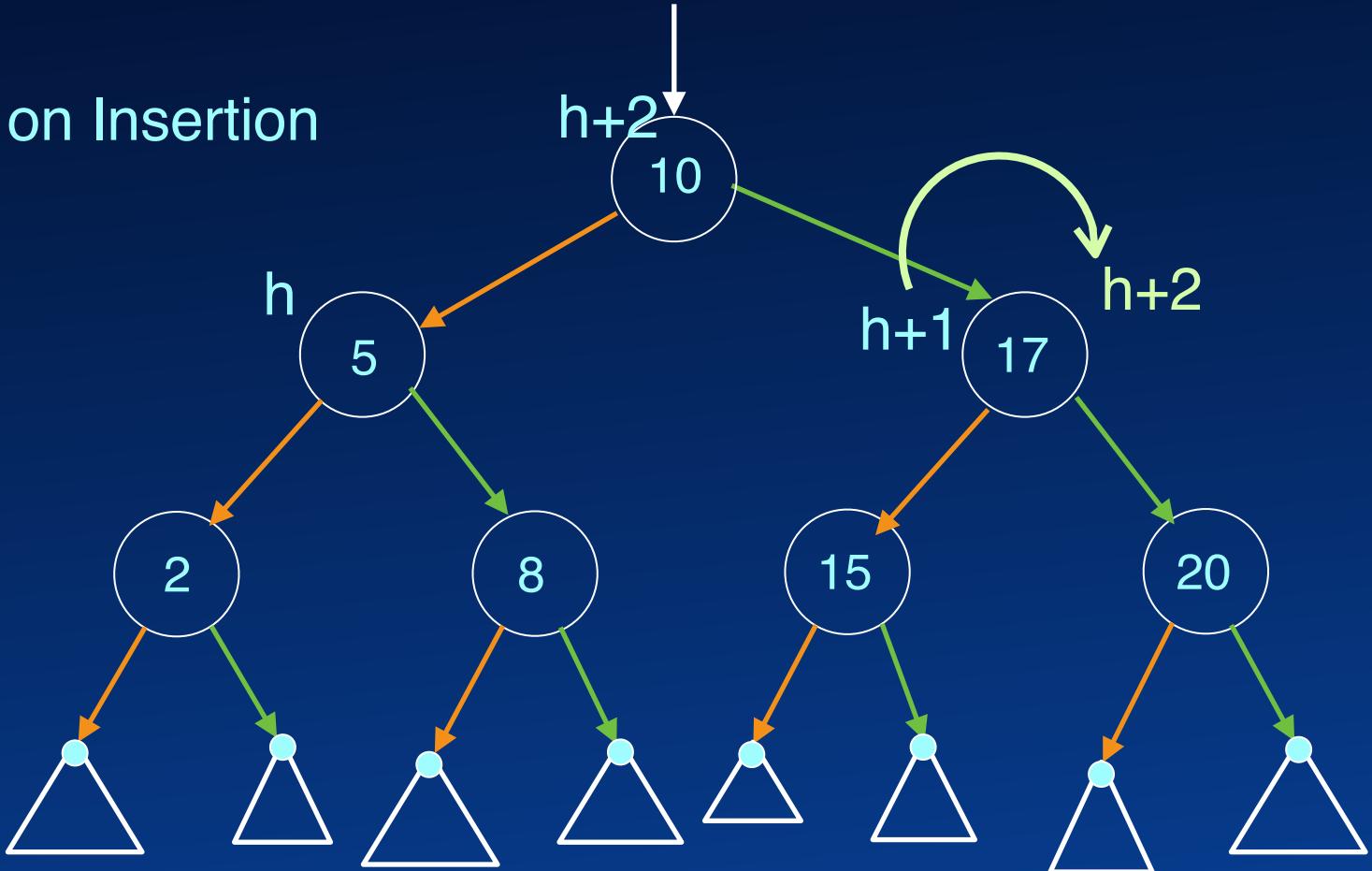
Imbalance on Insertion





Rotation

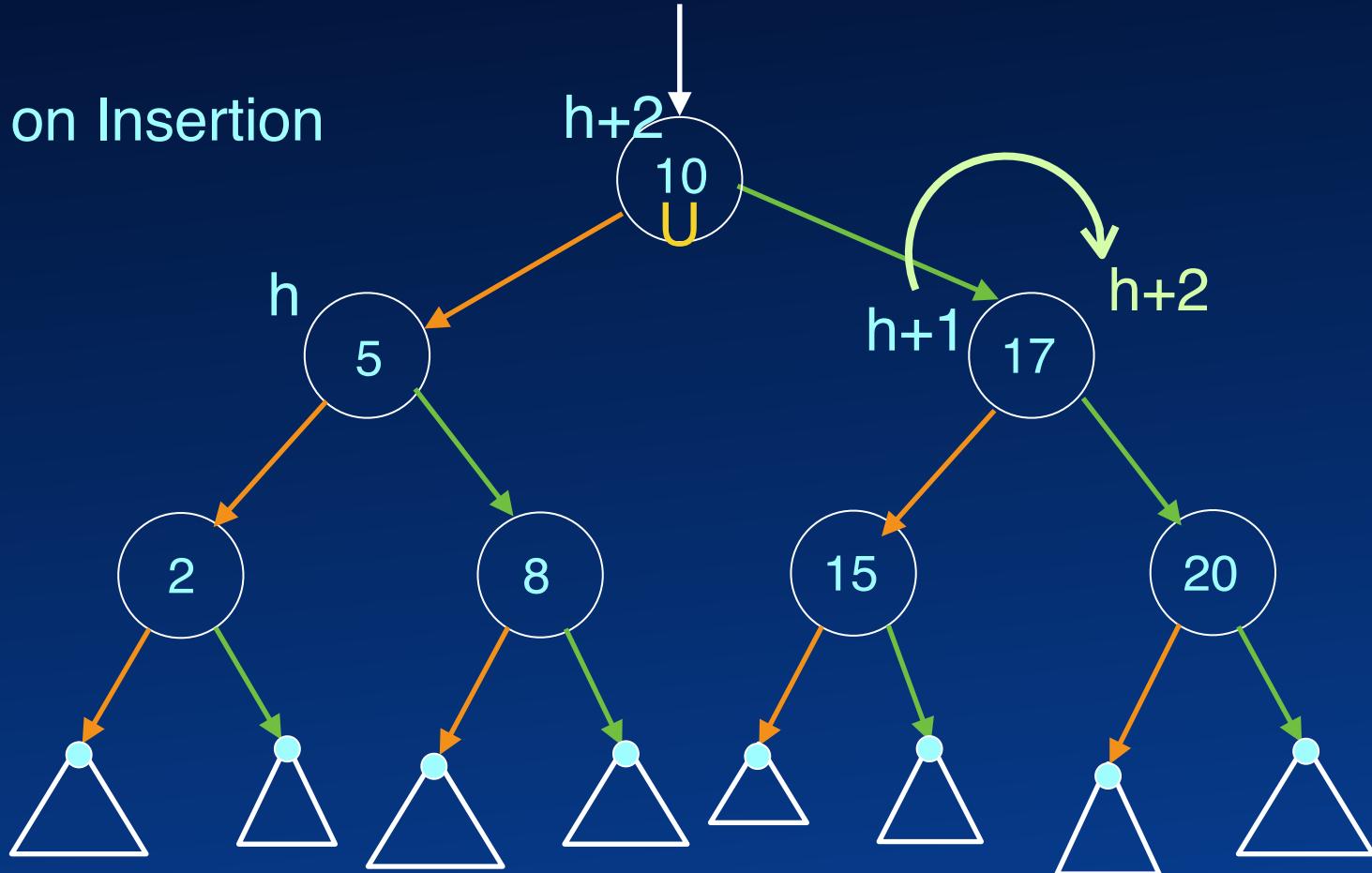
Imbalance on Insertion





Rotation

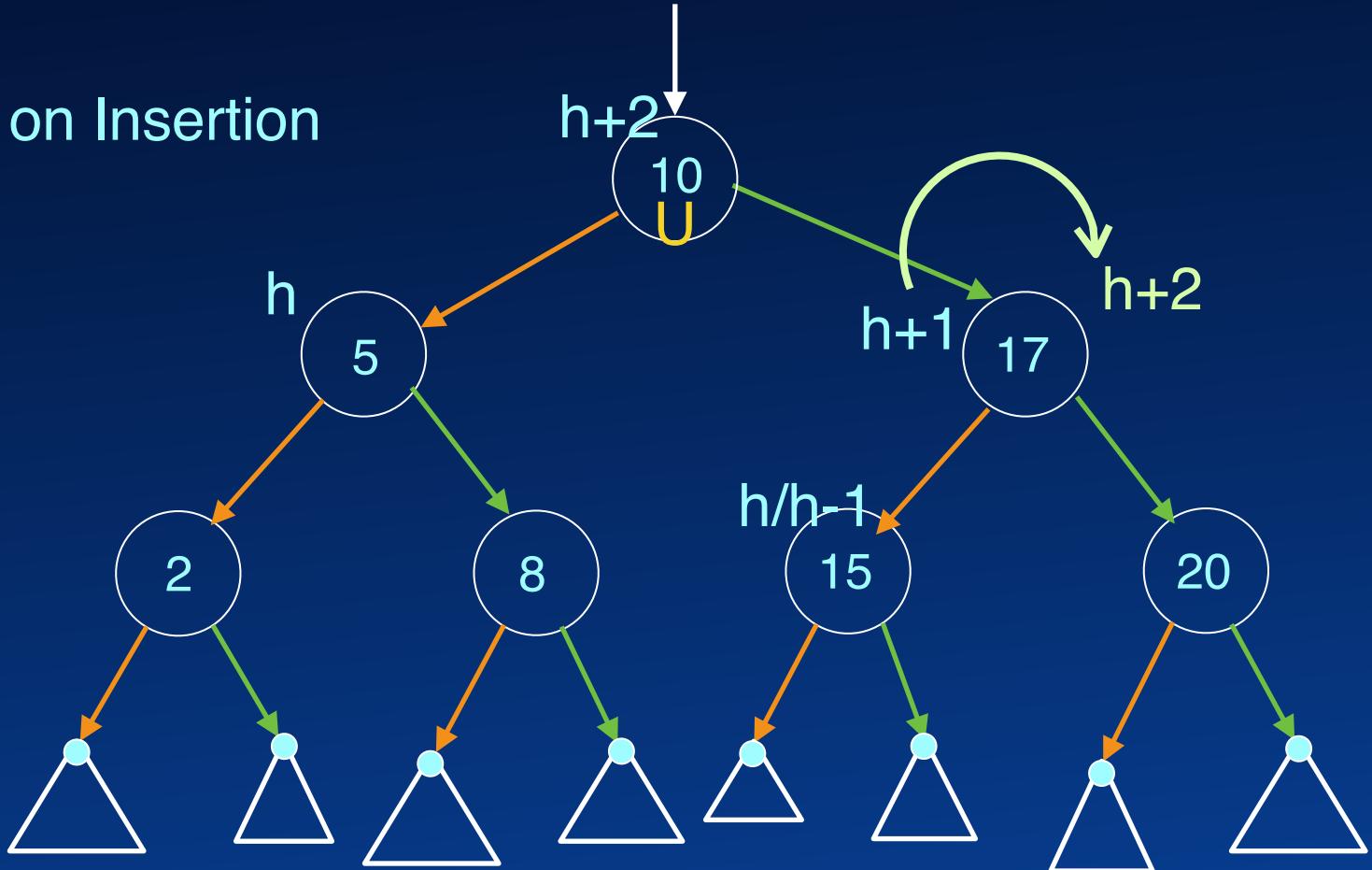
Imbalance on Insertion





Rotation

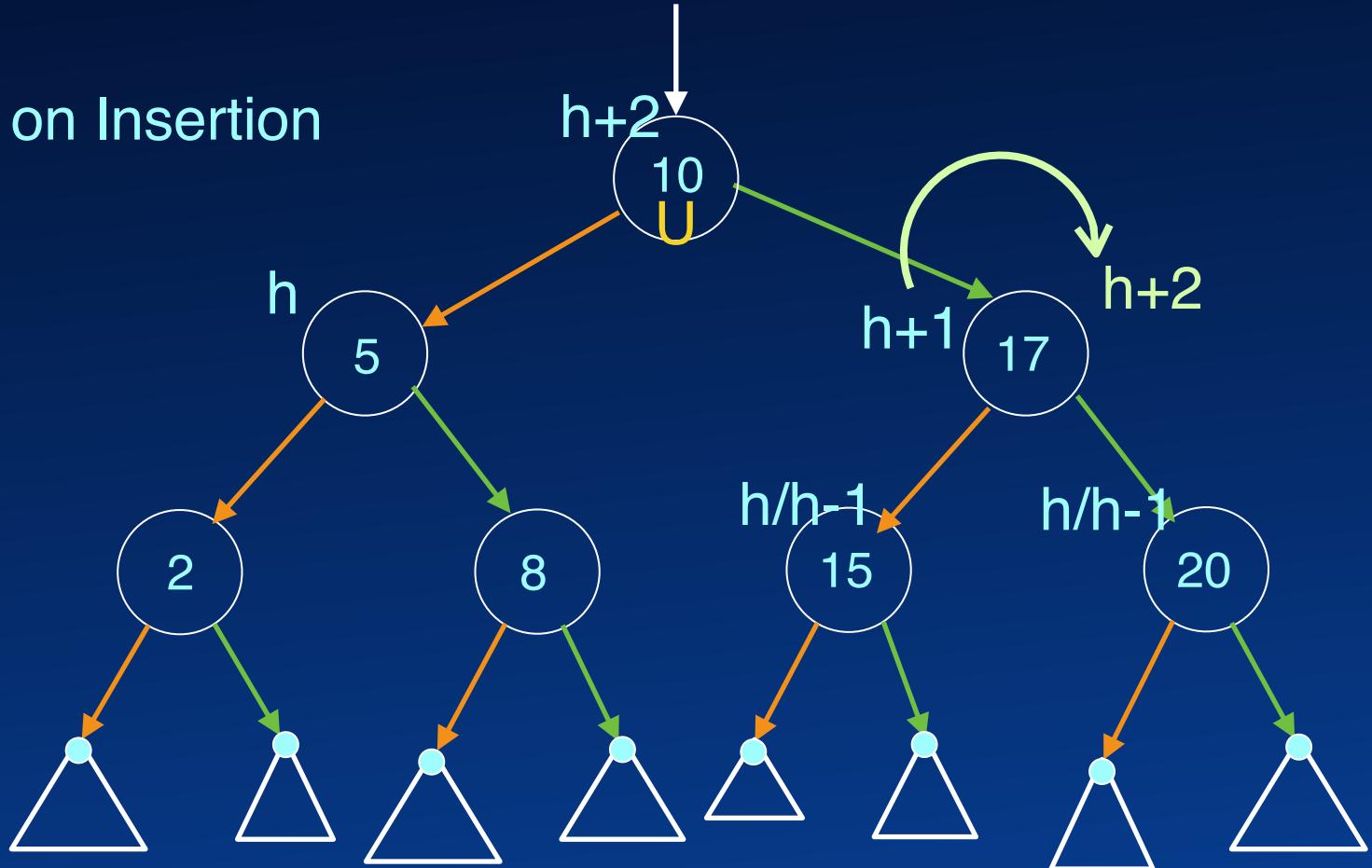
Imbalance on Insertion





Rotation

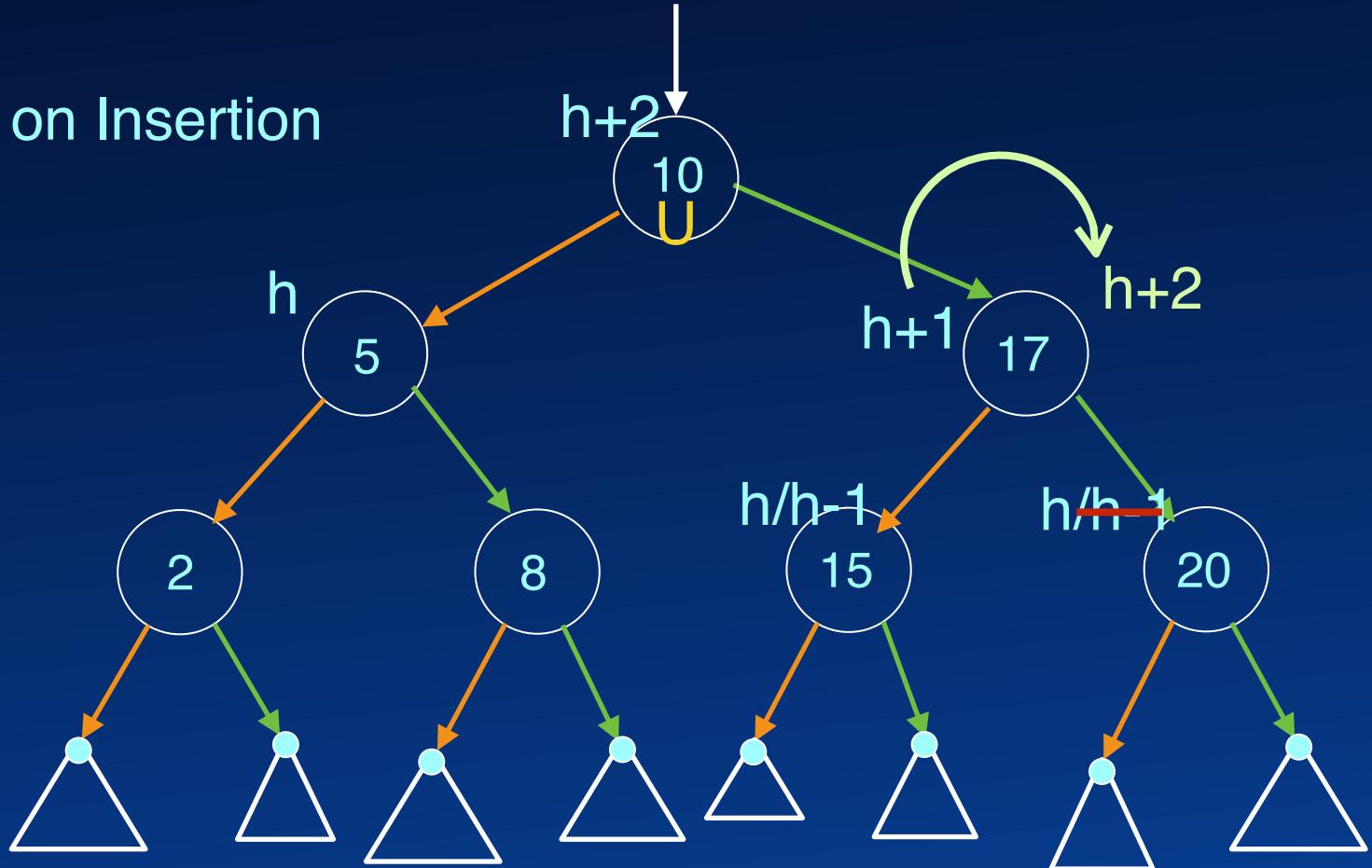
Imbalance on Insertion





Rotation

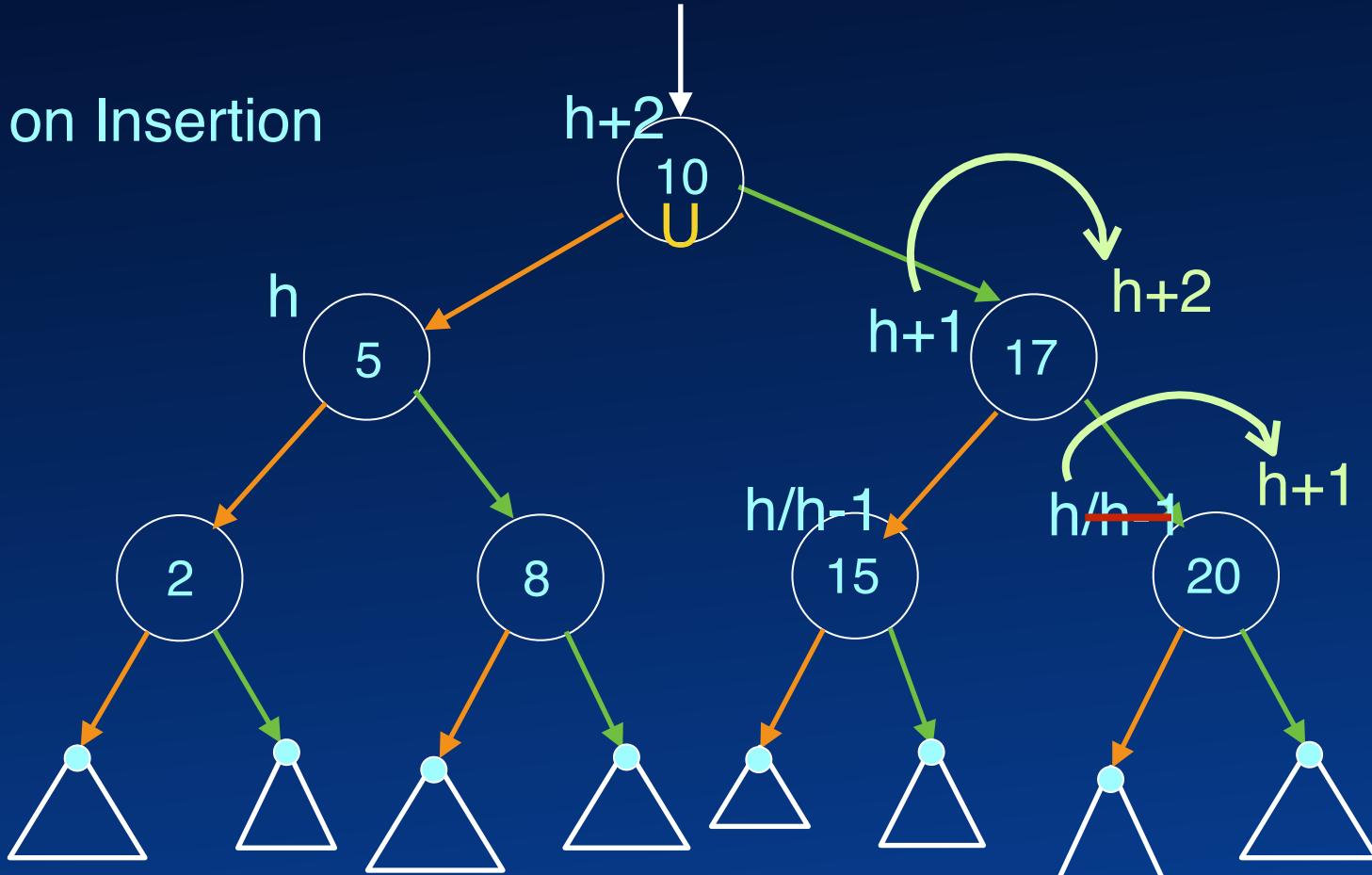
Imbalance on Insertion





Rotation

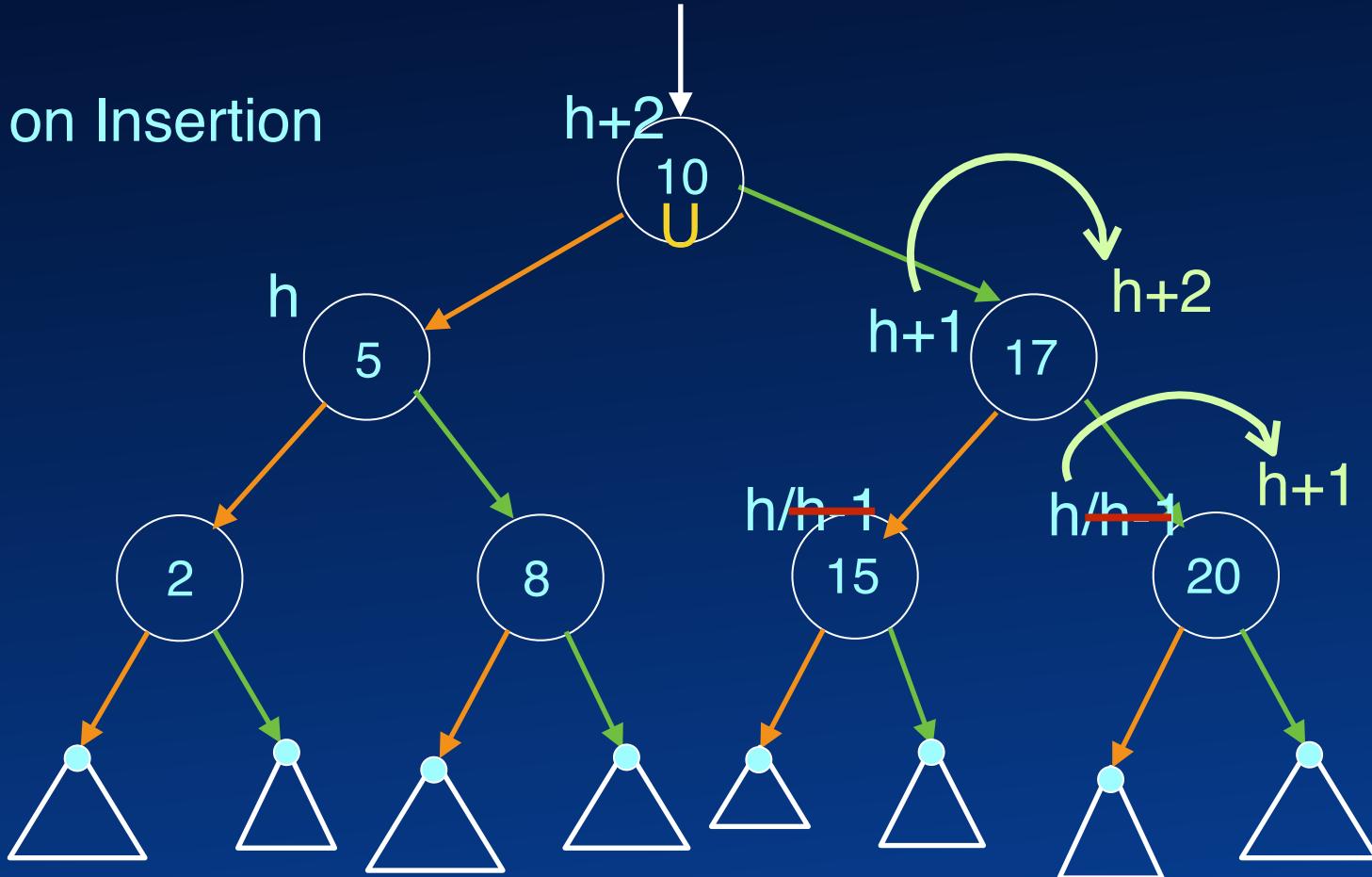
Imbalance on Insertion





Rotation

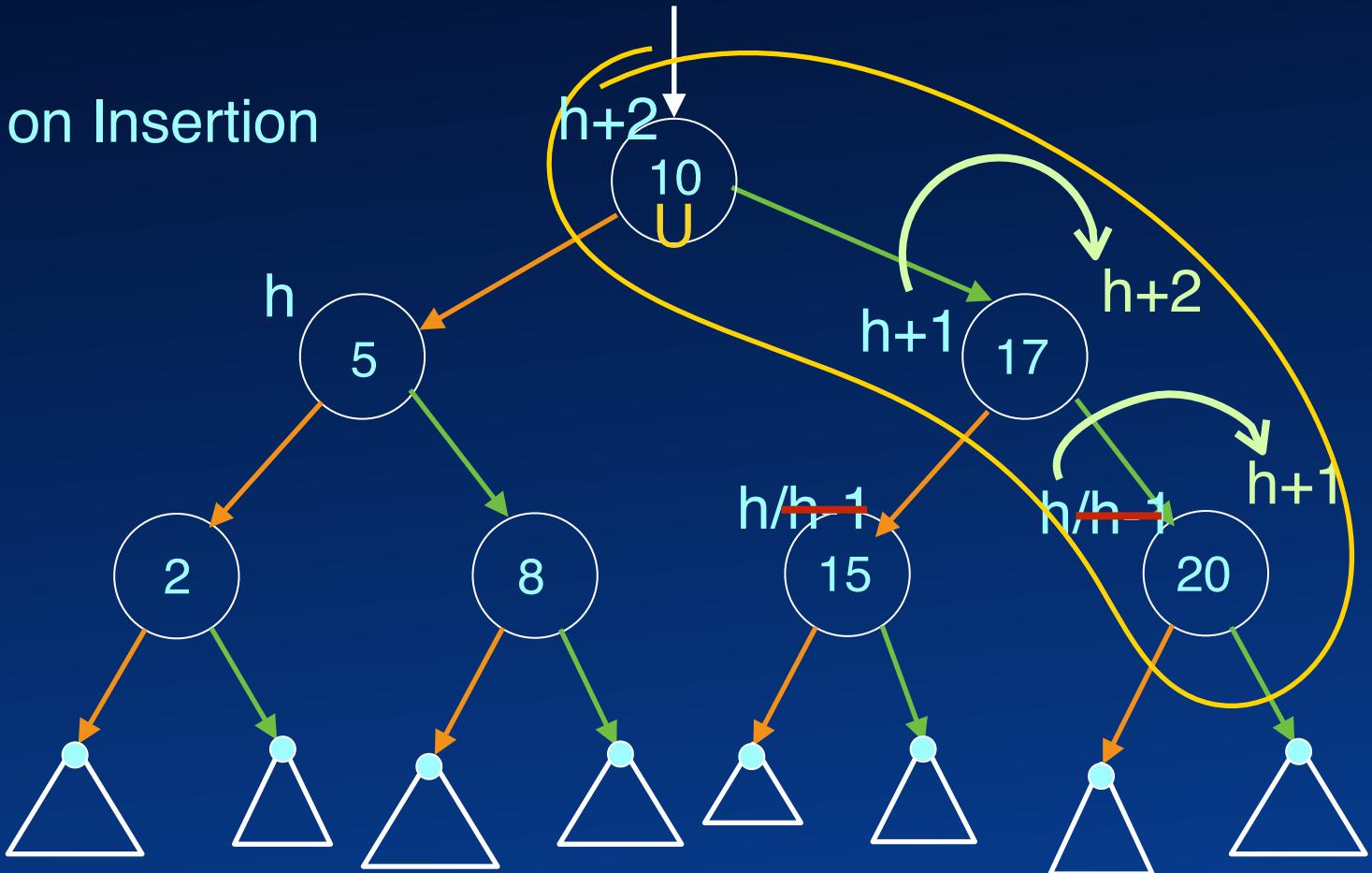
Imbalance on Insertion





Rotation

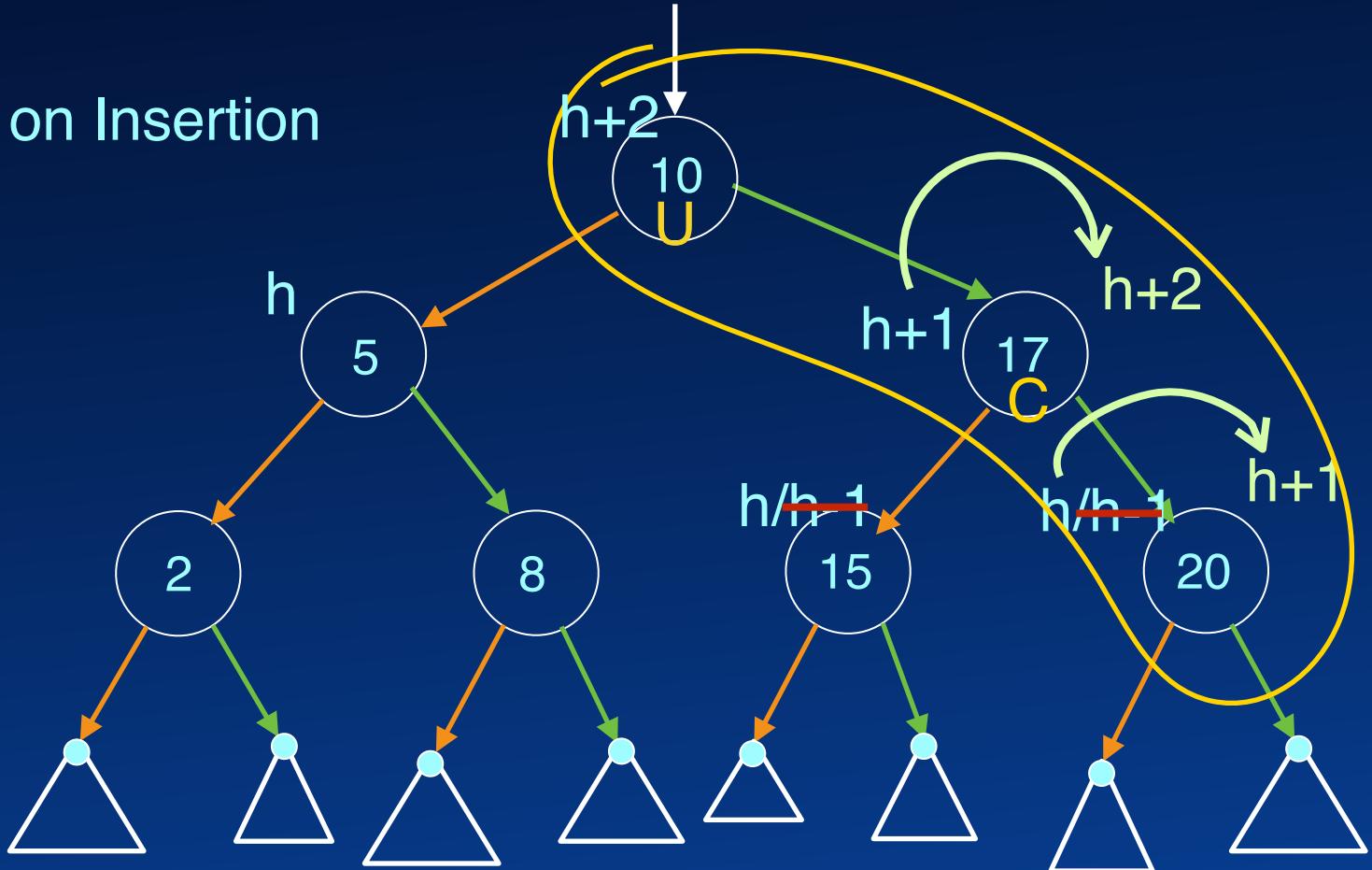
Imbalance on Insertion



Rotation



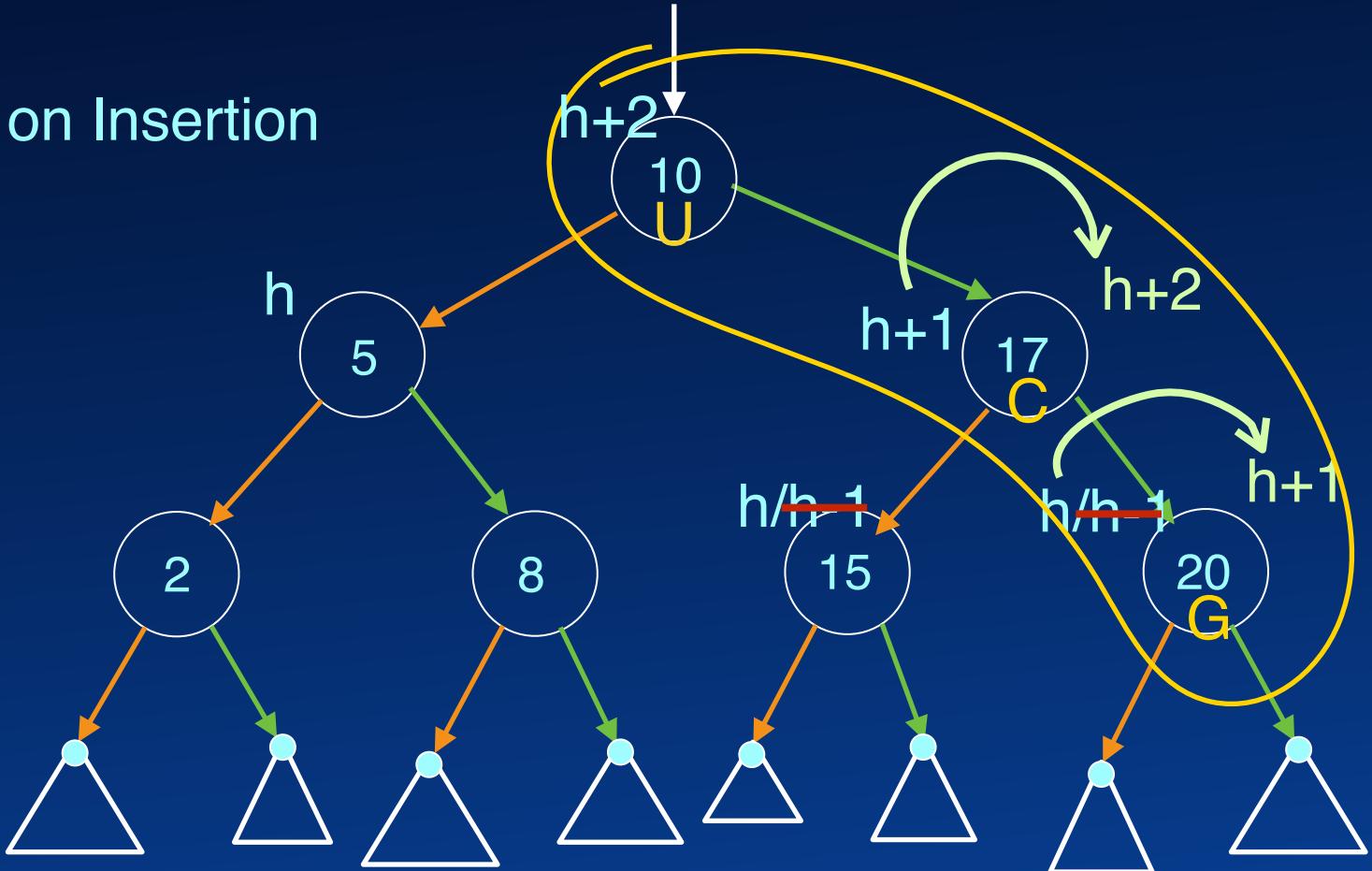
Imbalance on Insertion



Rotation



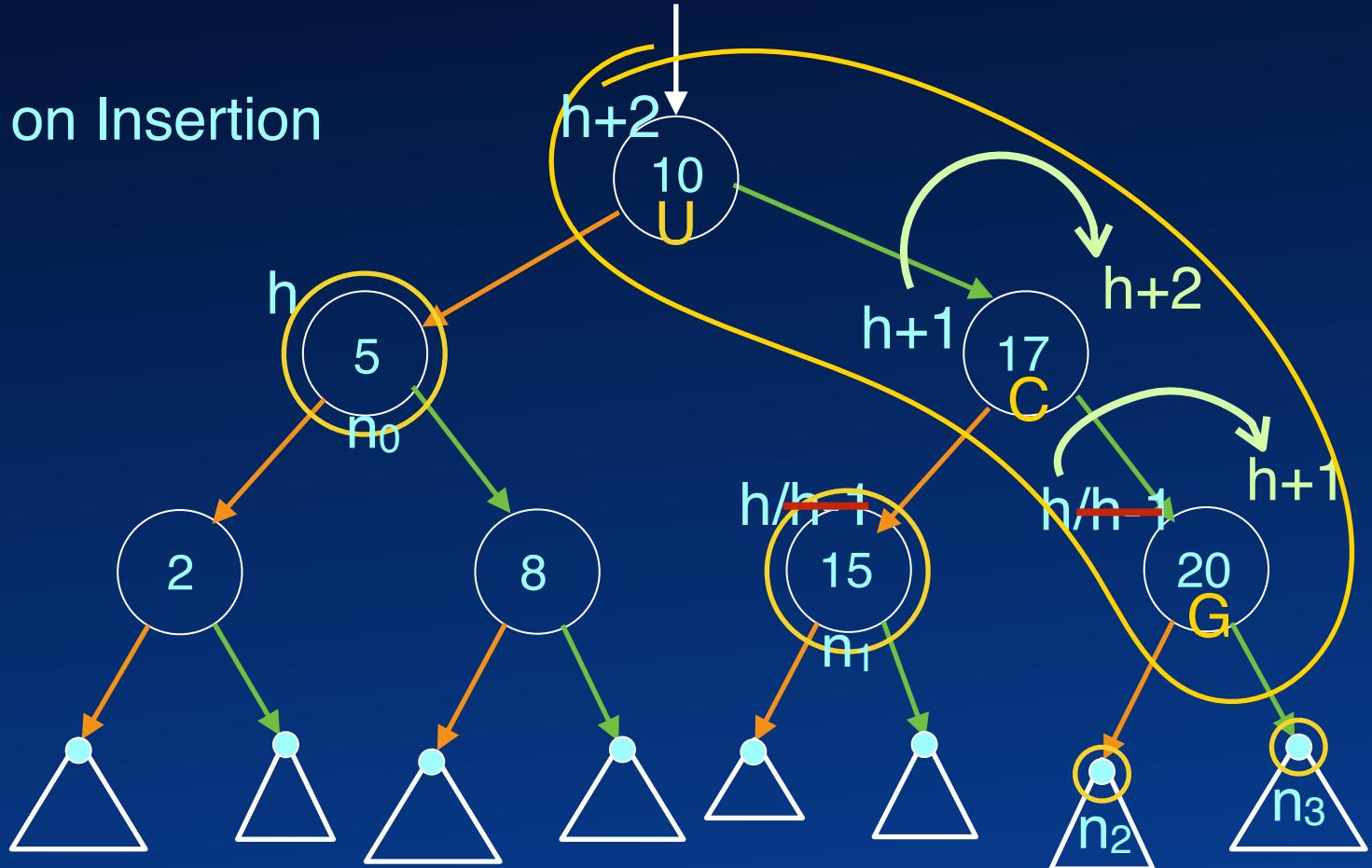
Imbalance on Insertion





Rotation

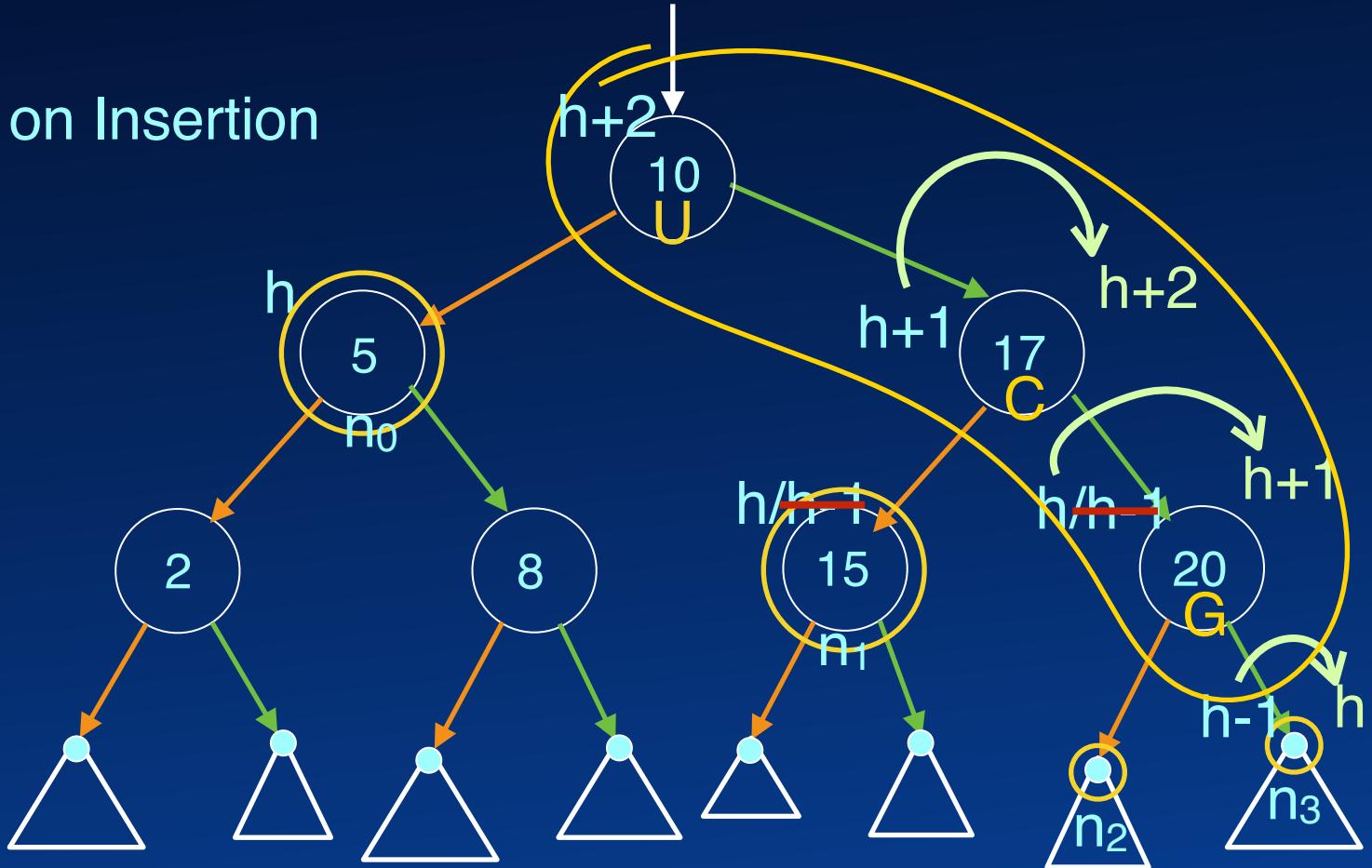
Imbalance on Insertion





Rotation

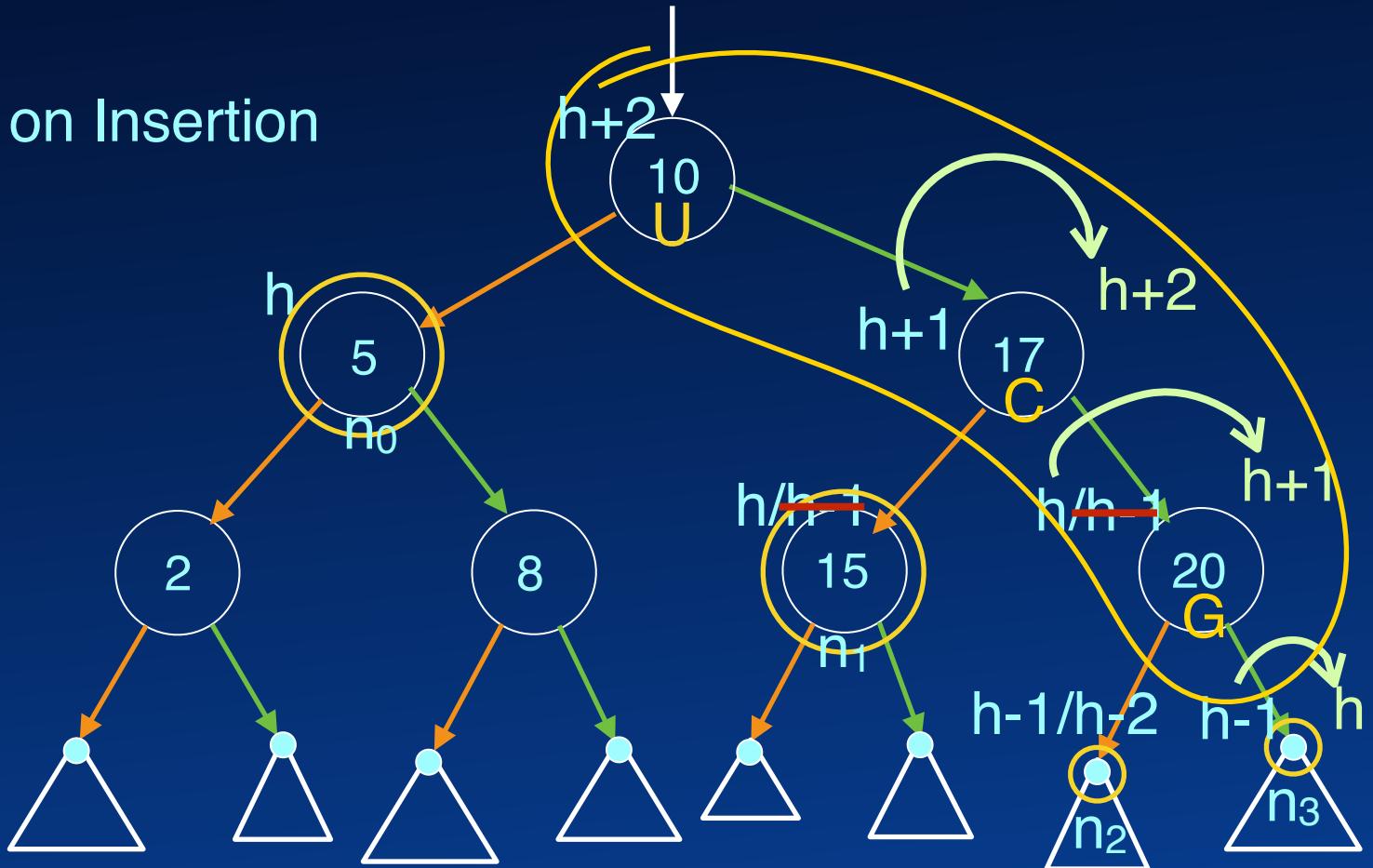
Imbalance on Insertion





Rotation

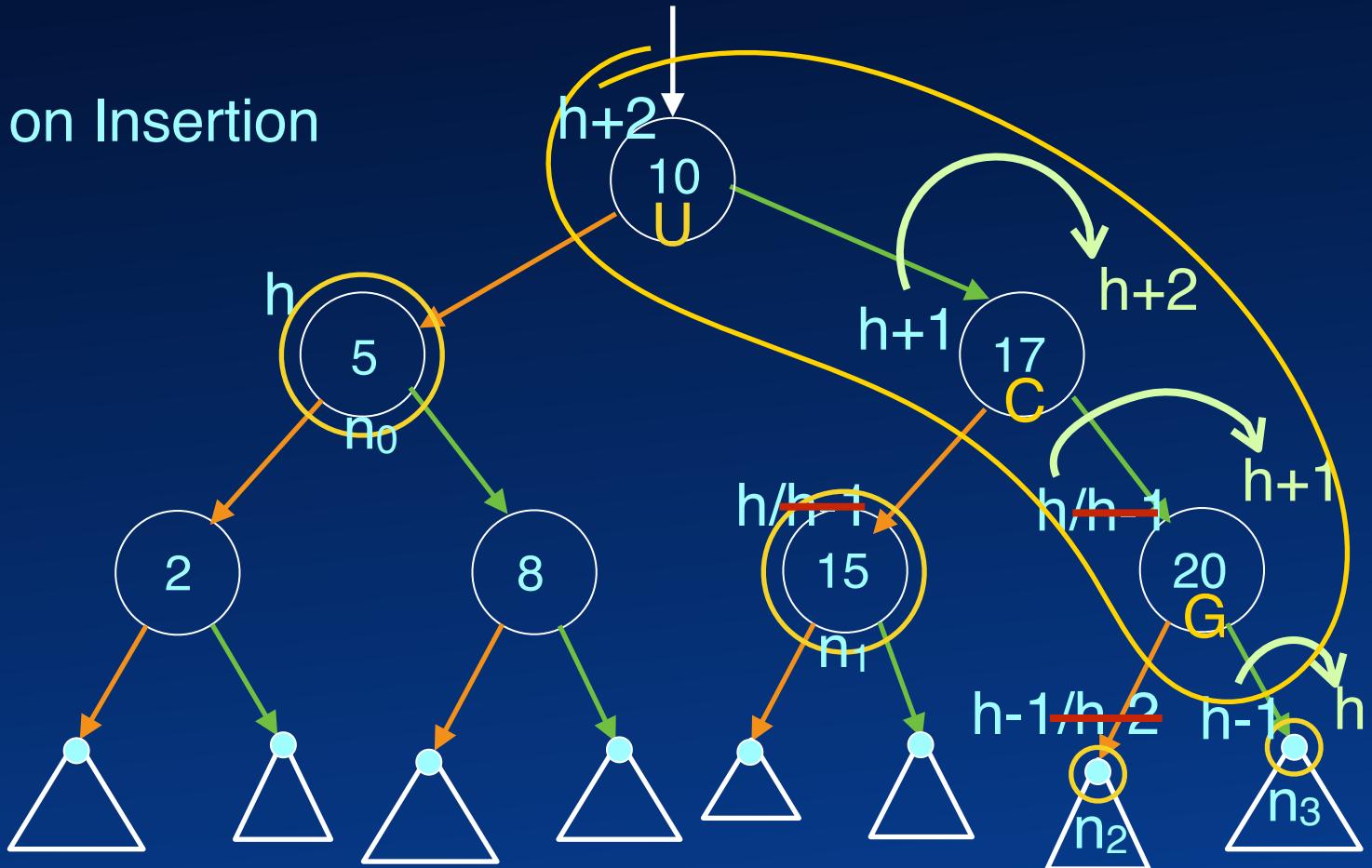
Imbalance on Insertion





Rotation

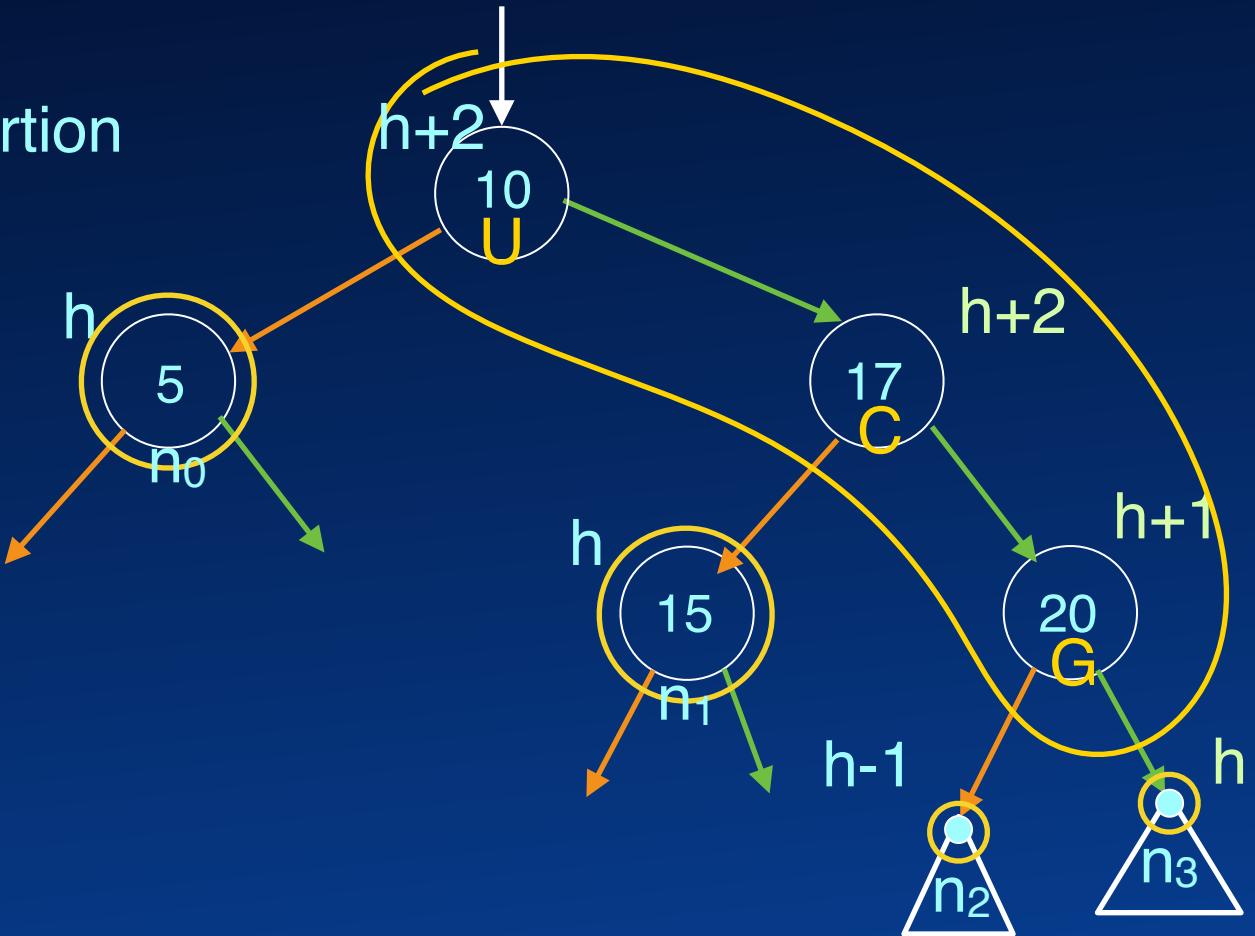
Imbalance on Insertion





Rotation

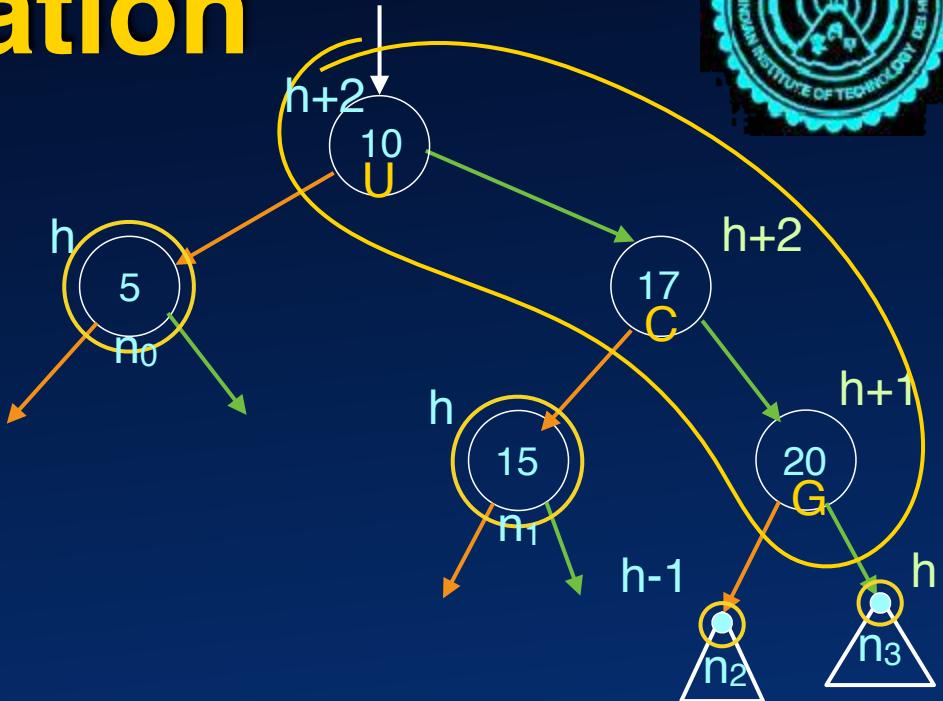
Imbalance on Insertion





Rotation

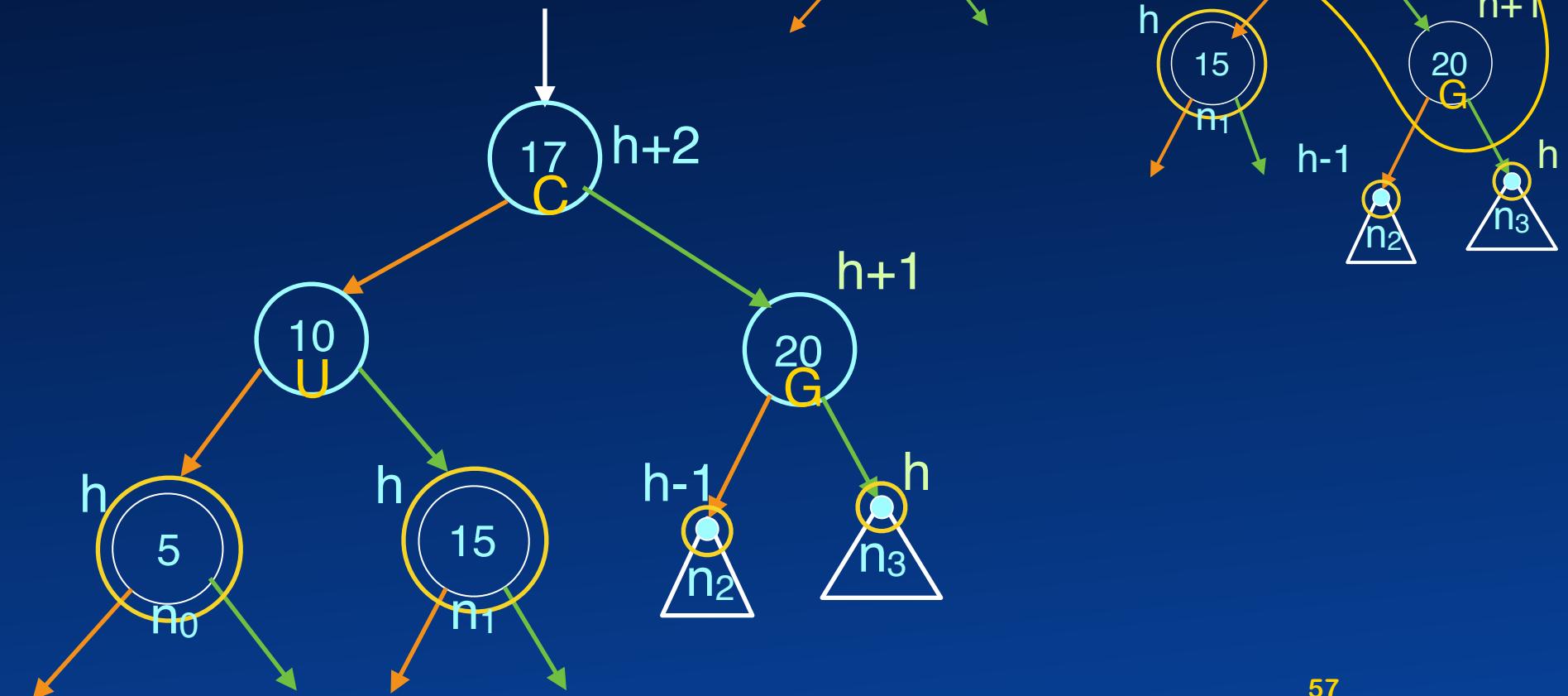
Imbalance on Insertion



Rotation



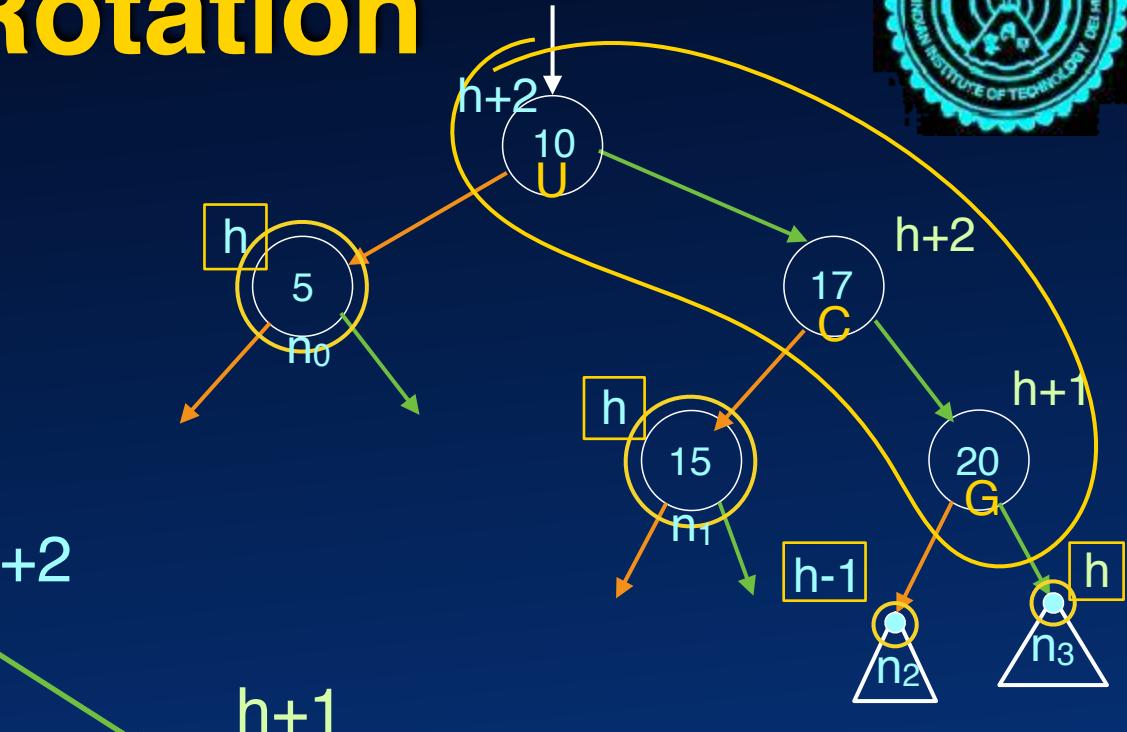
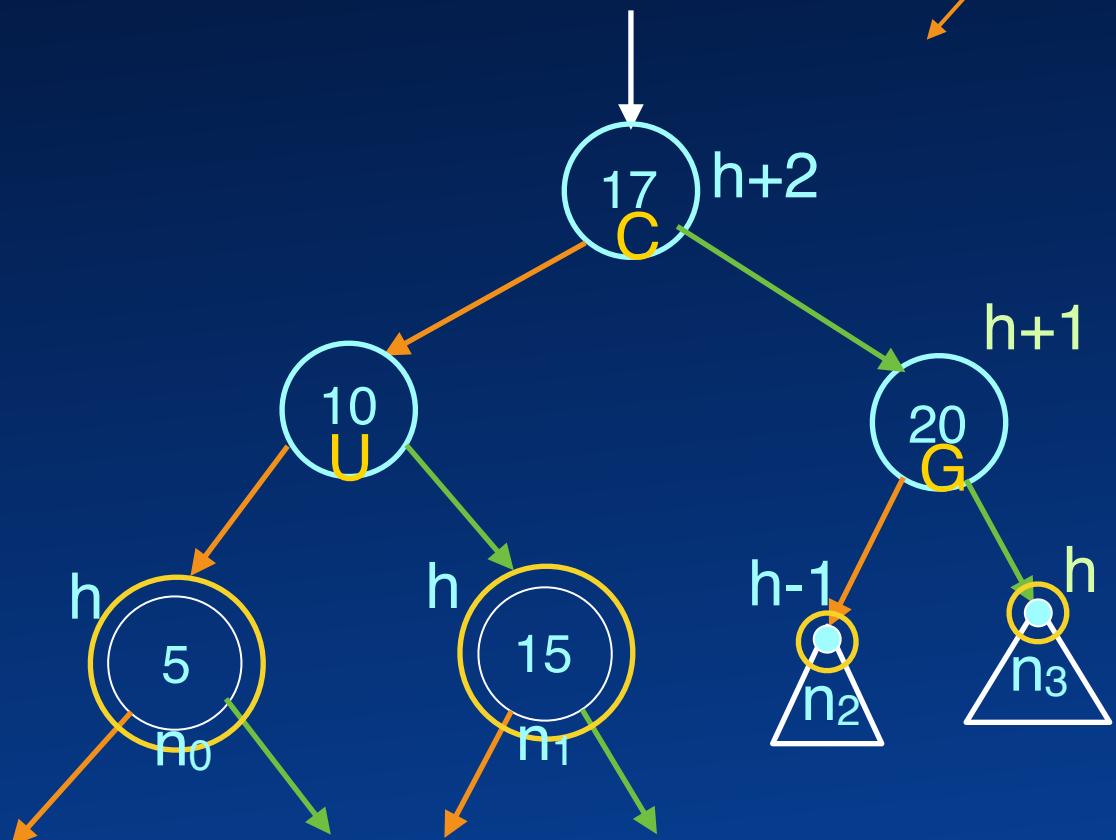
Imbalance on Insertion





Rotation

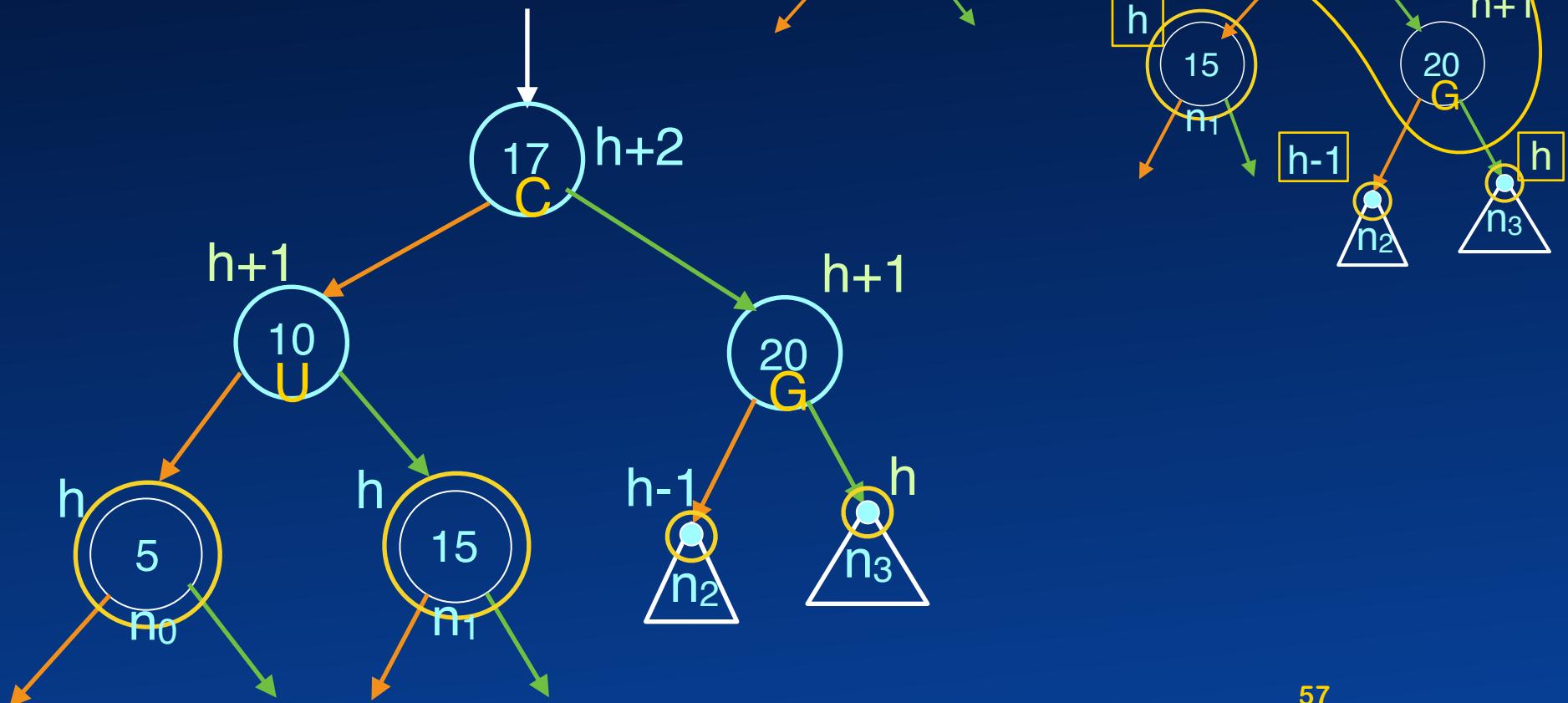
Imbalance on Insertion





Rotation

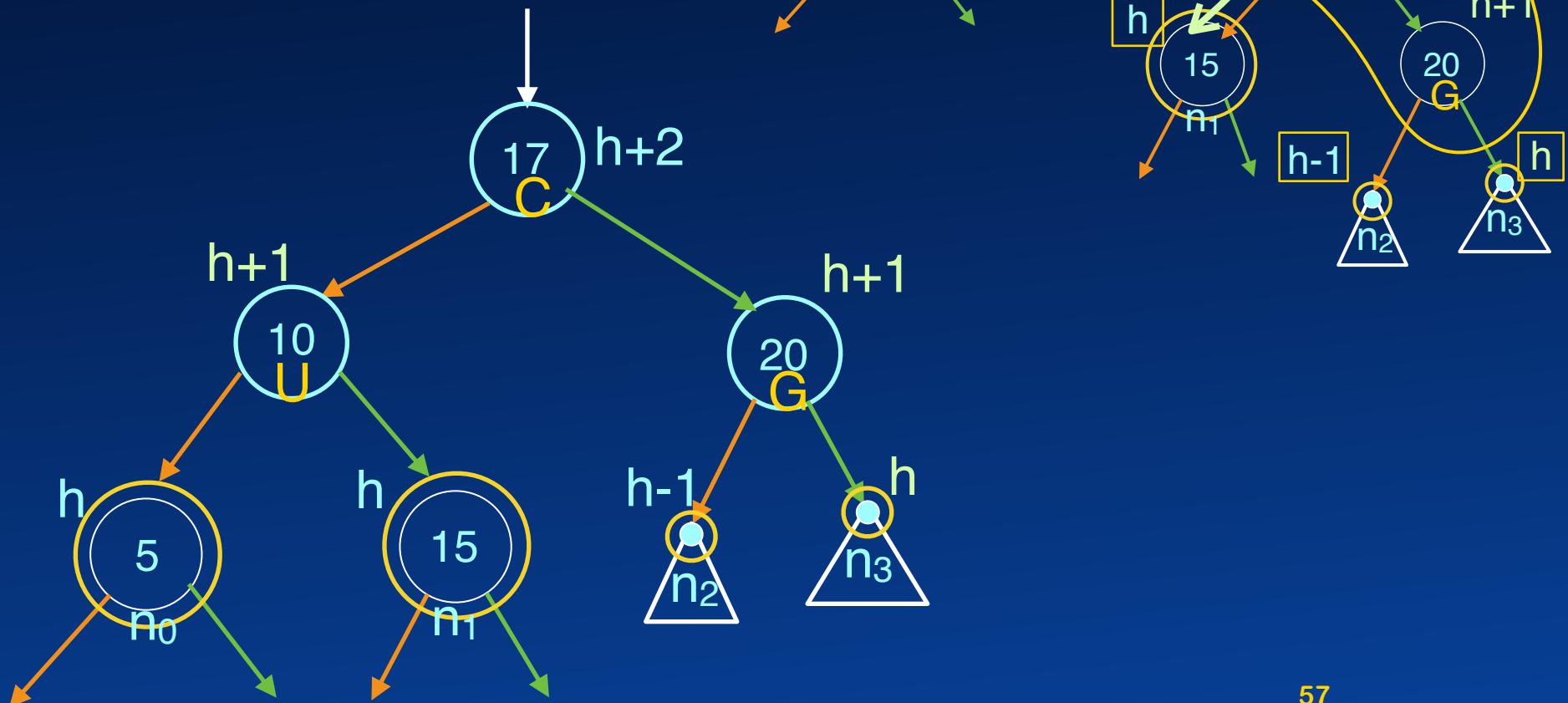
Imbalance on Insertion





Rotation

Imbalance on Insertion





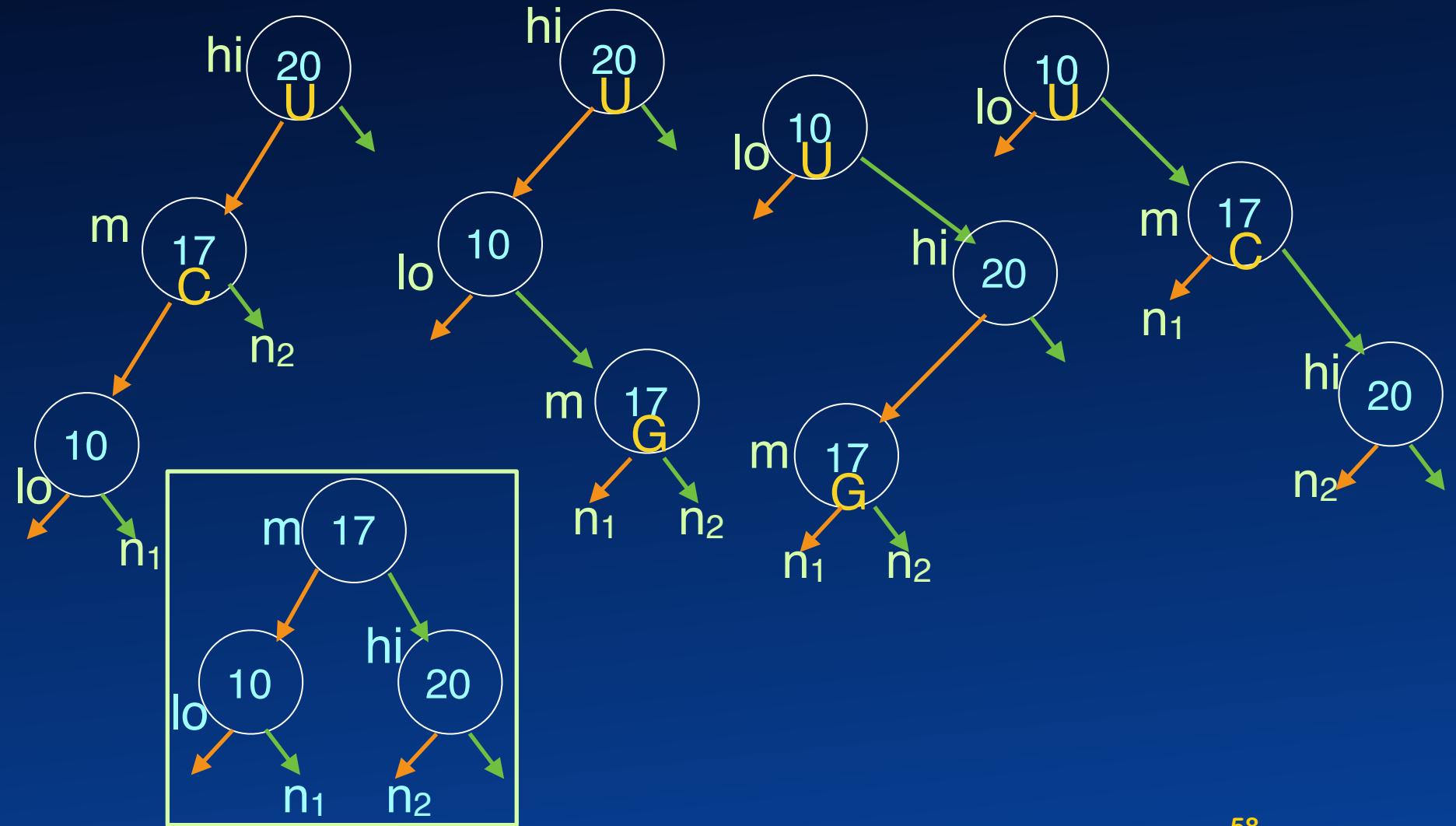
Rotation

Imbalance on Insertion



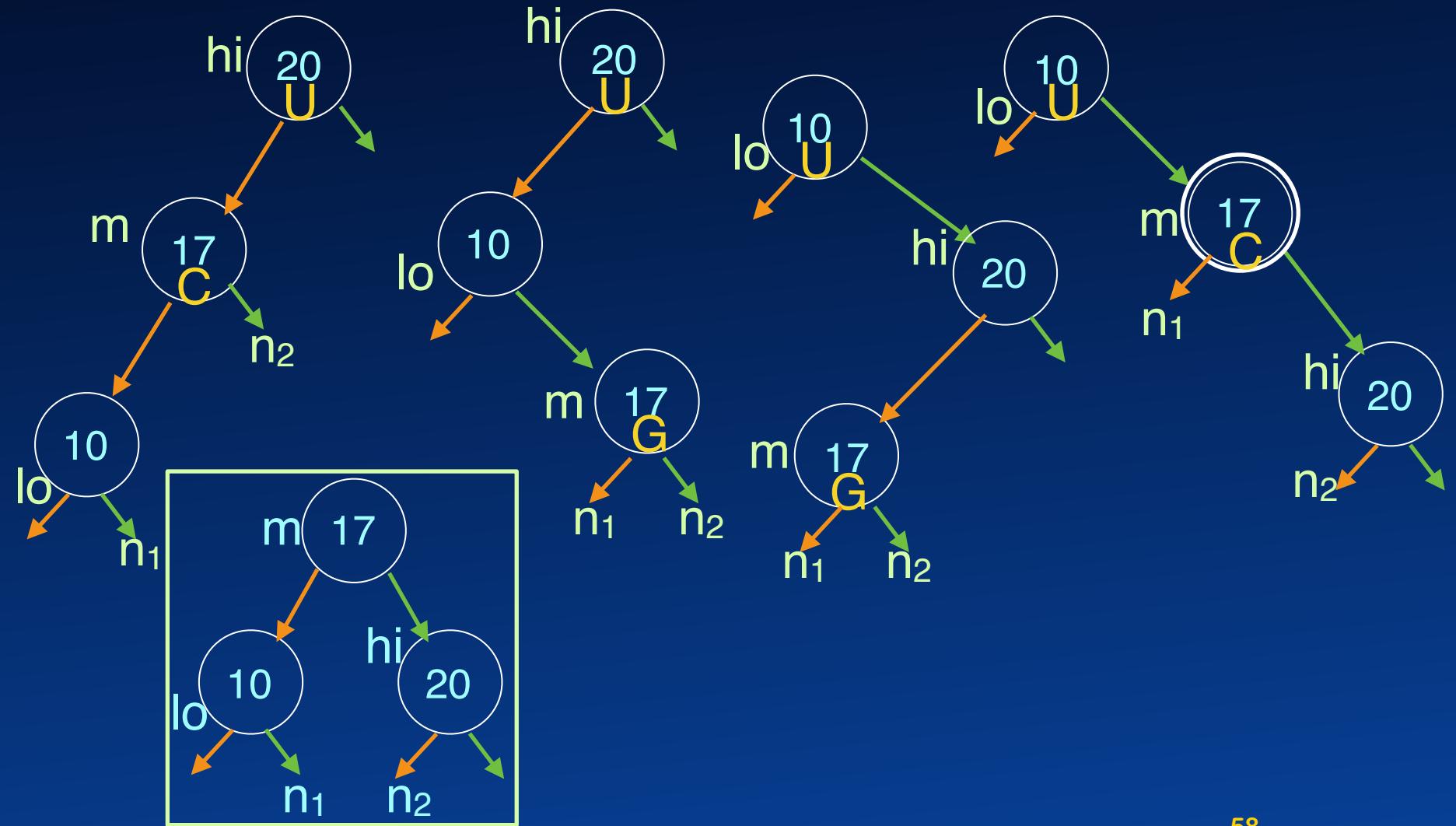


Rebalancing AVL Tree



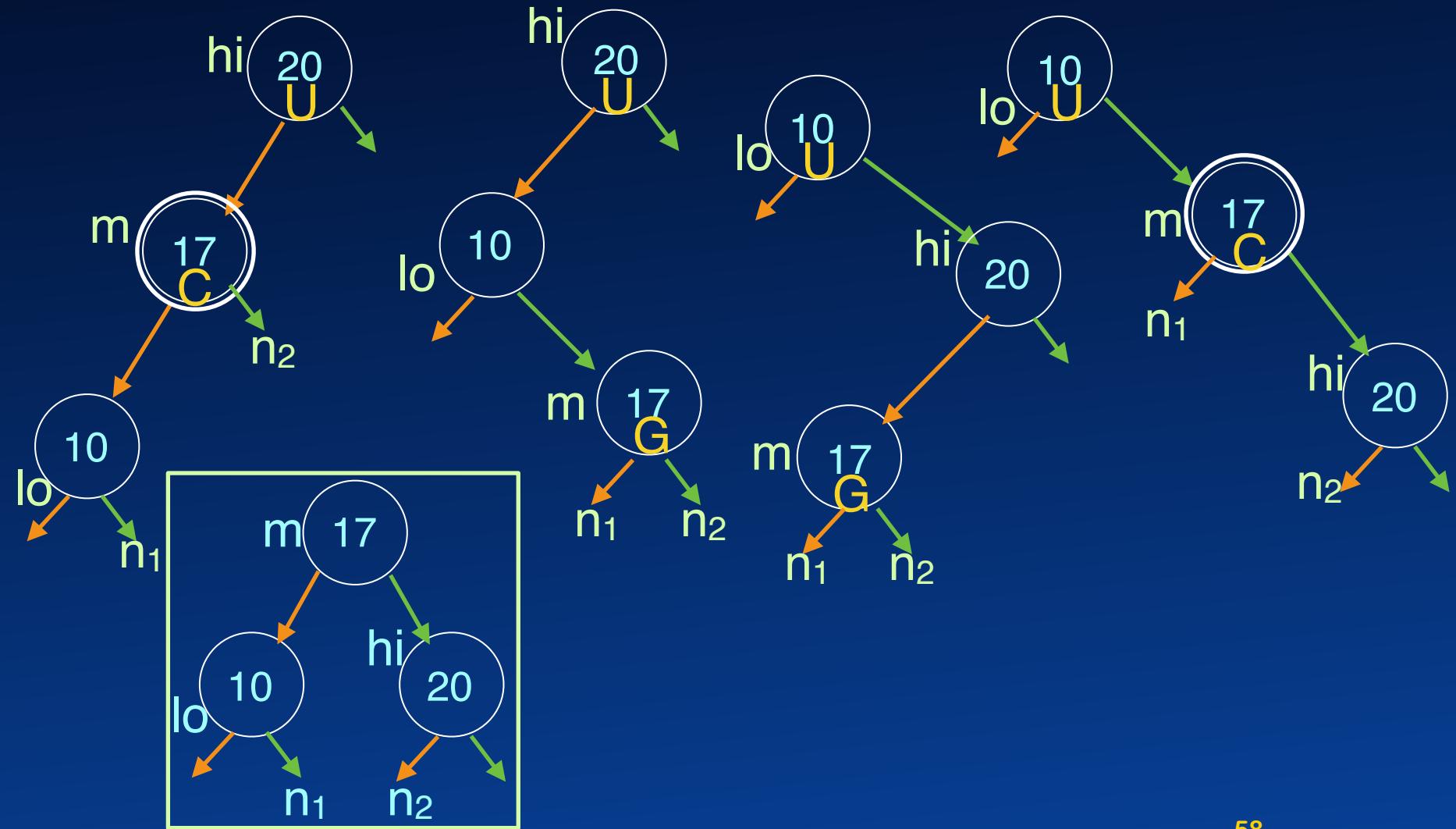


Rebalancing AVL Tree



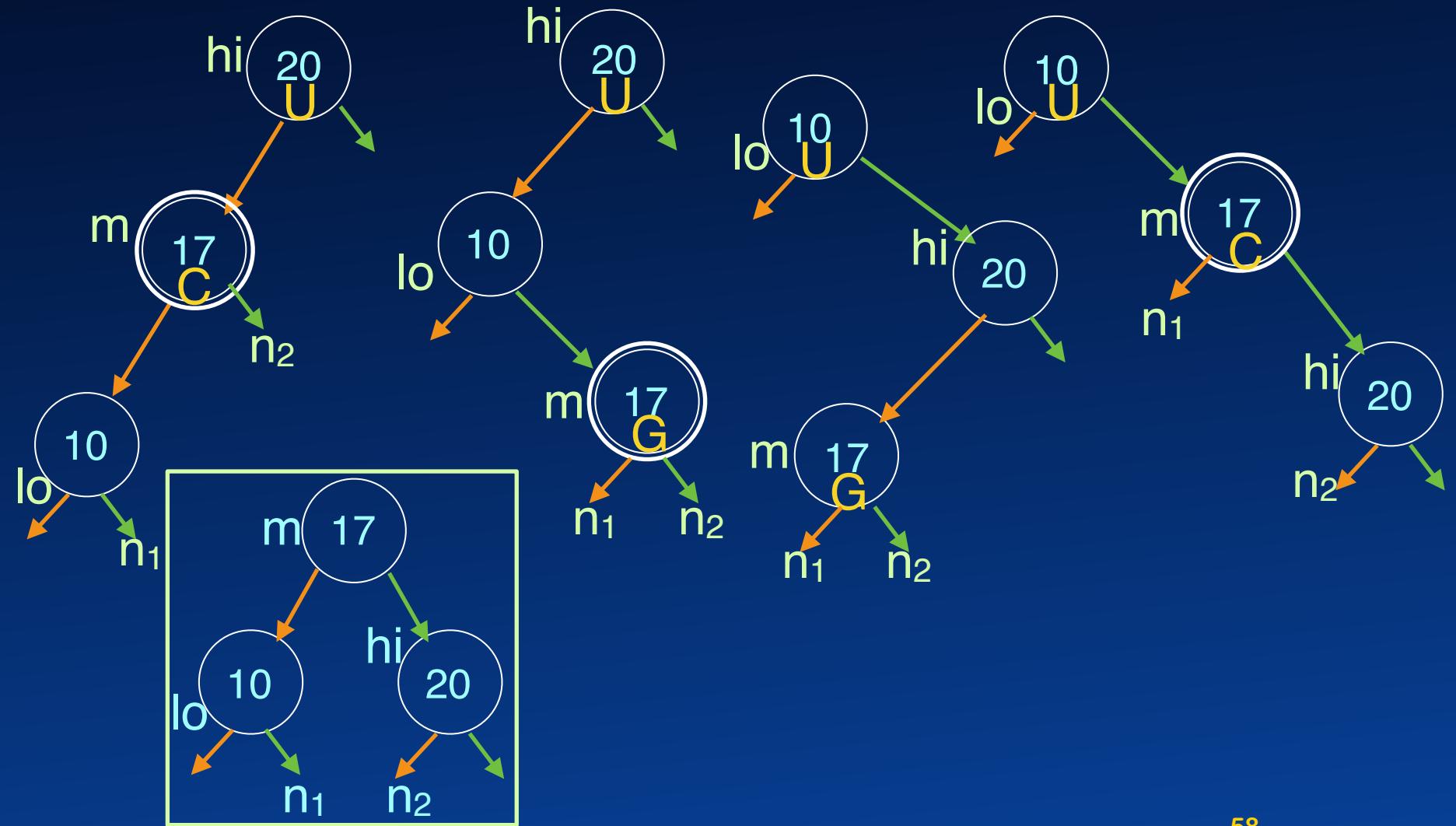


Rebalancing AVL Tree



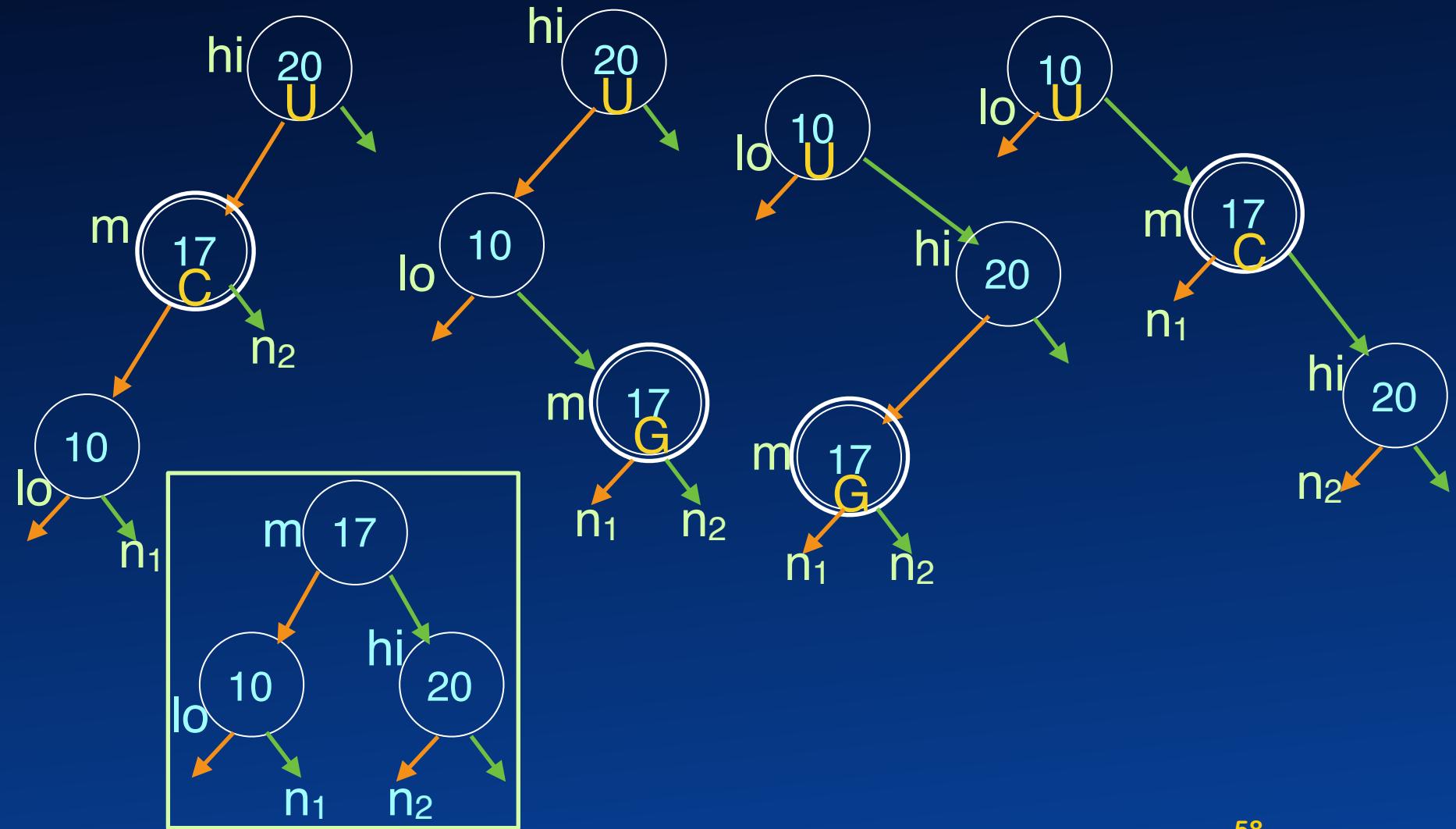


Rebalancing AVL Tree



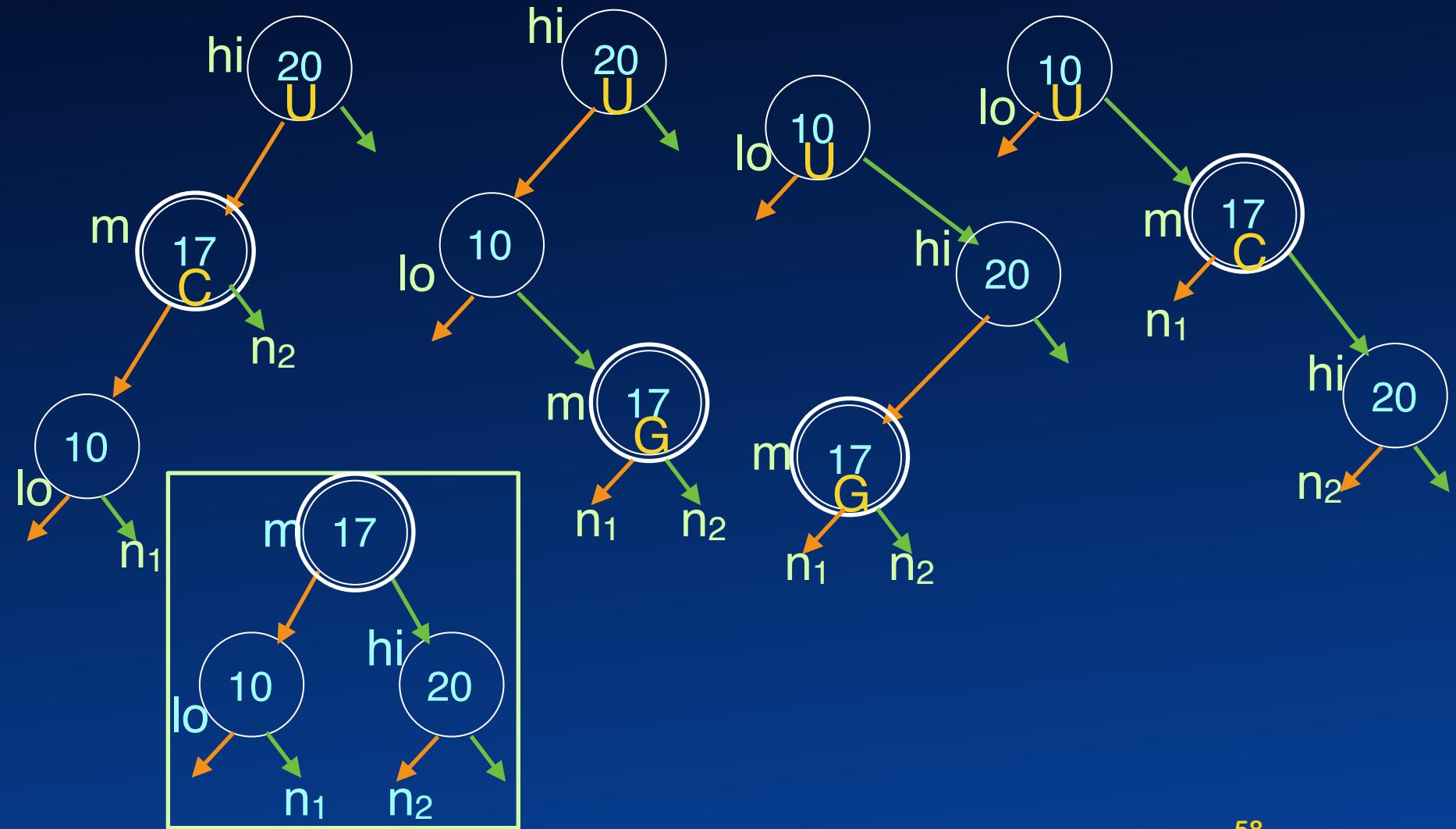


Rebalancing AVL Tree



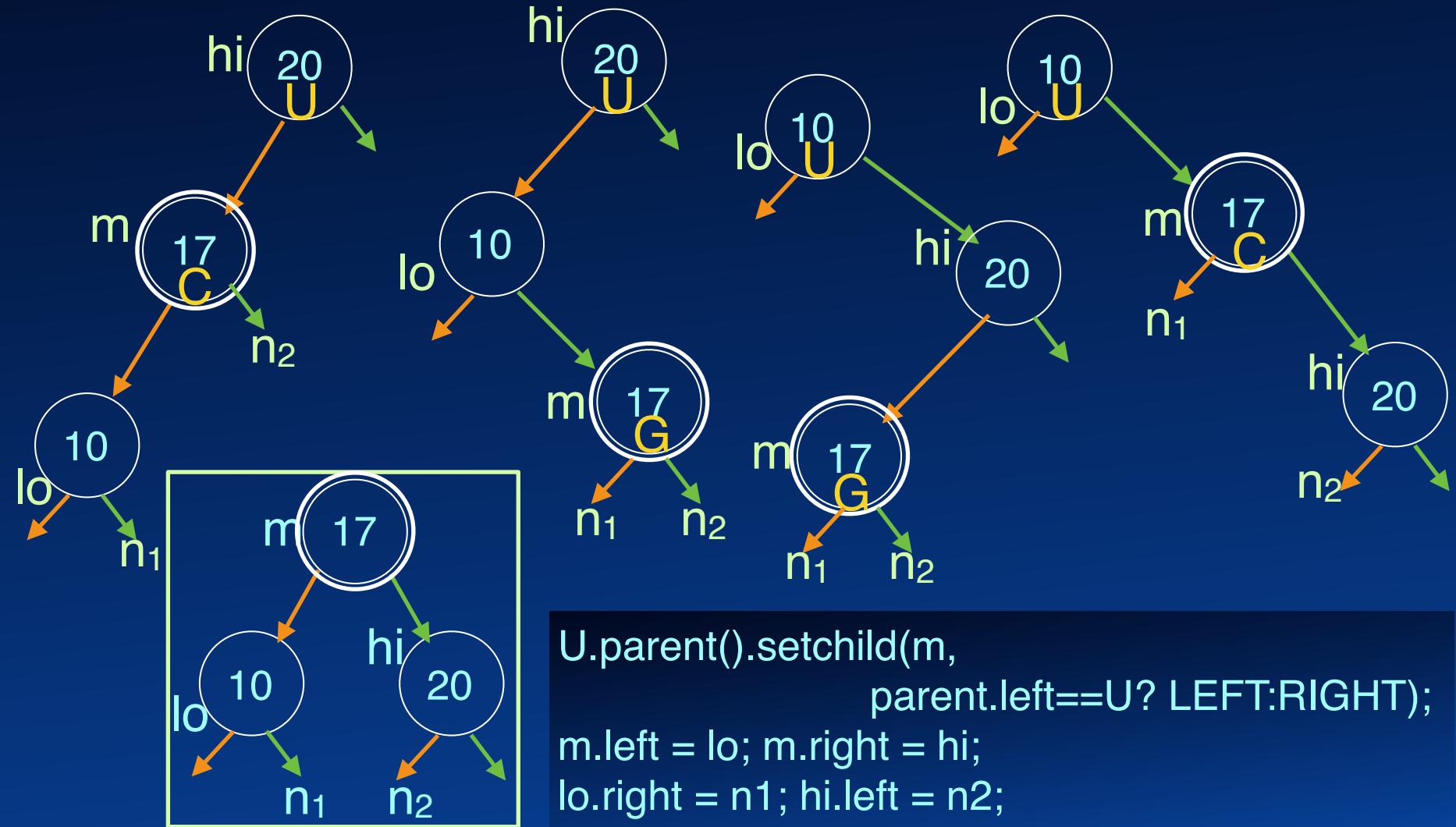


Rebalancing AVL Tree





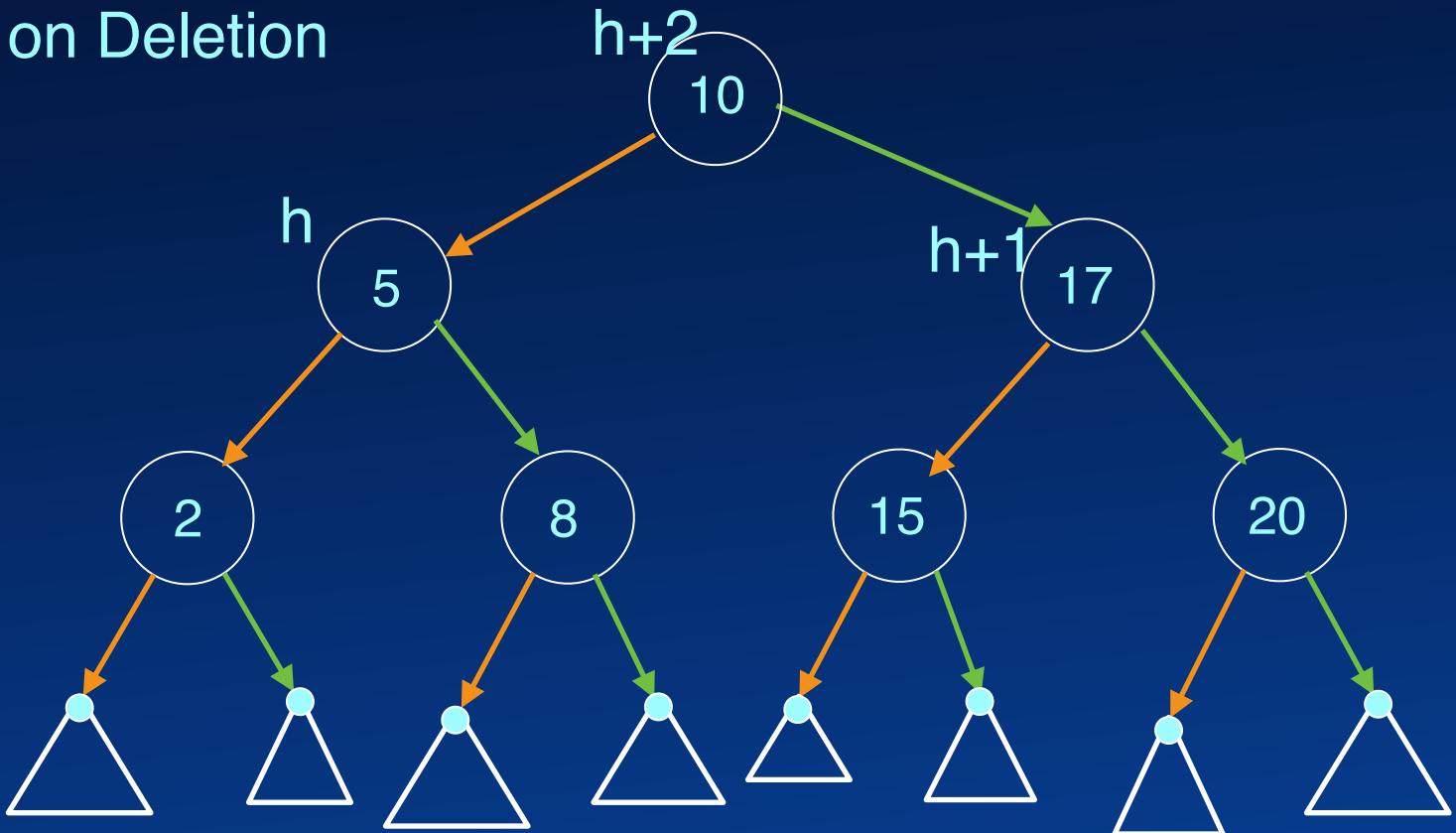
Rebalancing AVL Tree





Rotation

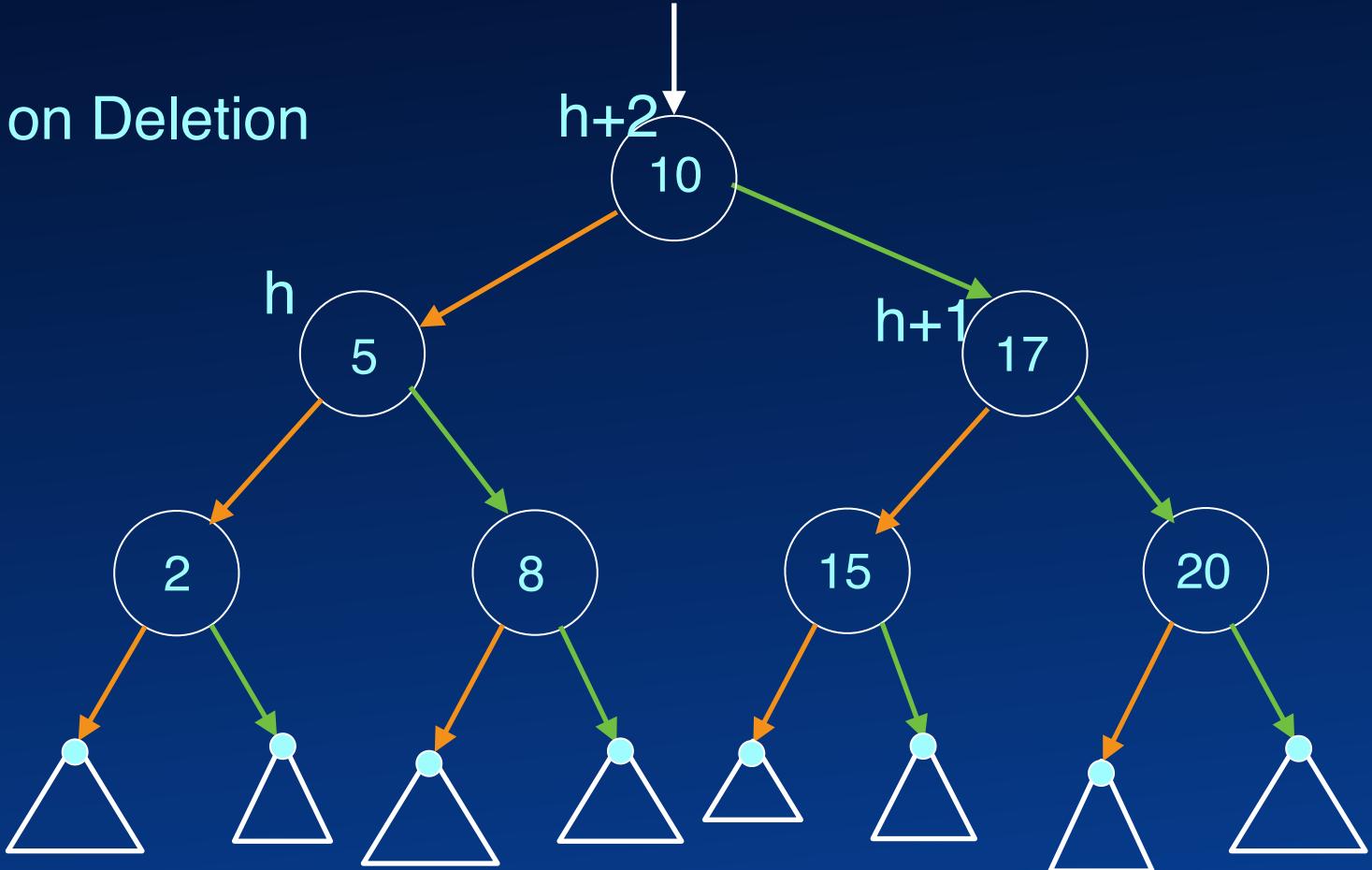
Imbalance on Deletion





Rotation

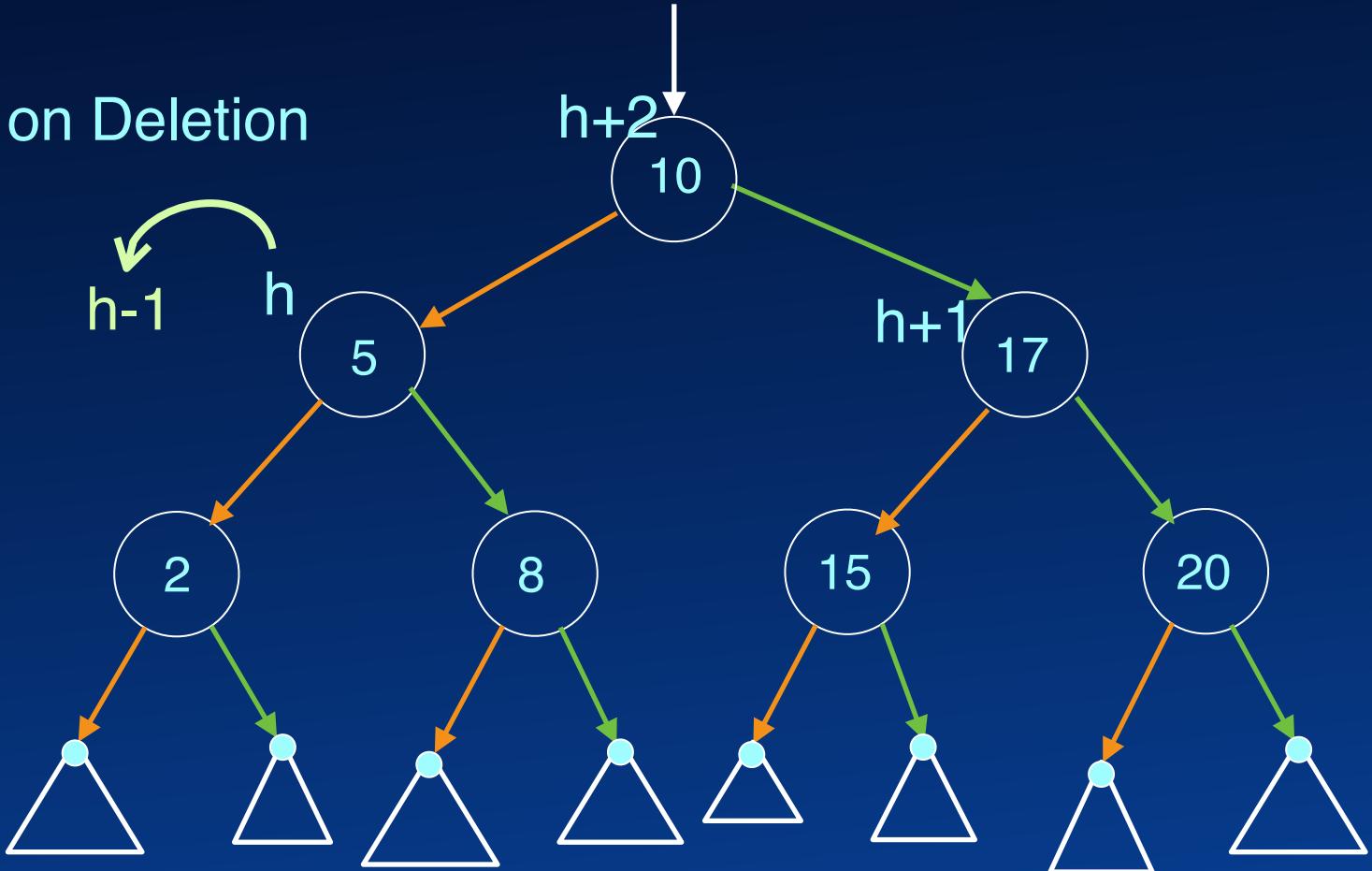
Imbalance on Deletion





Rotation

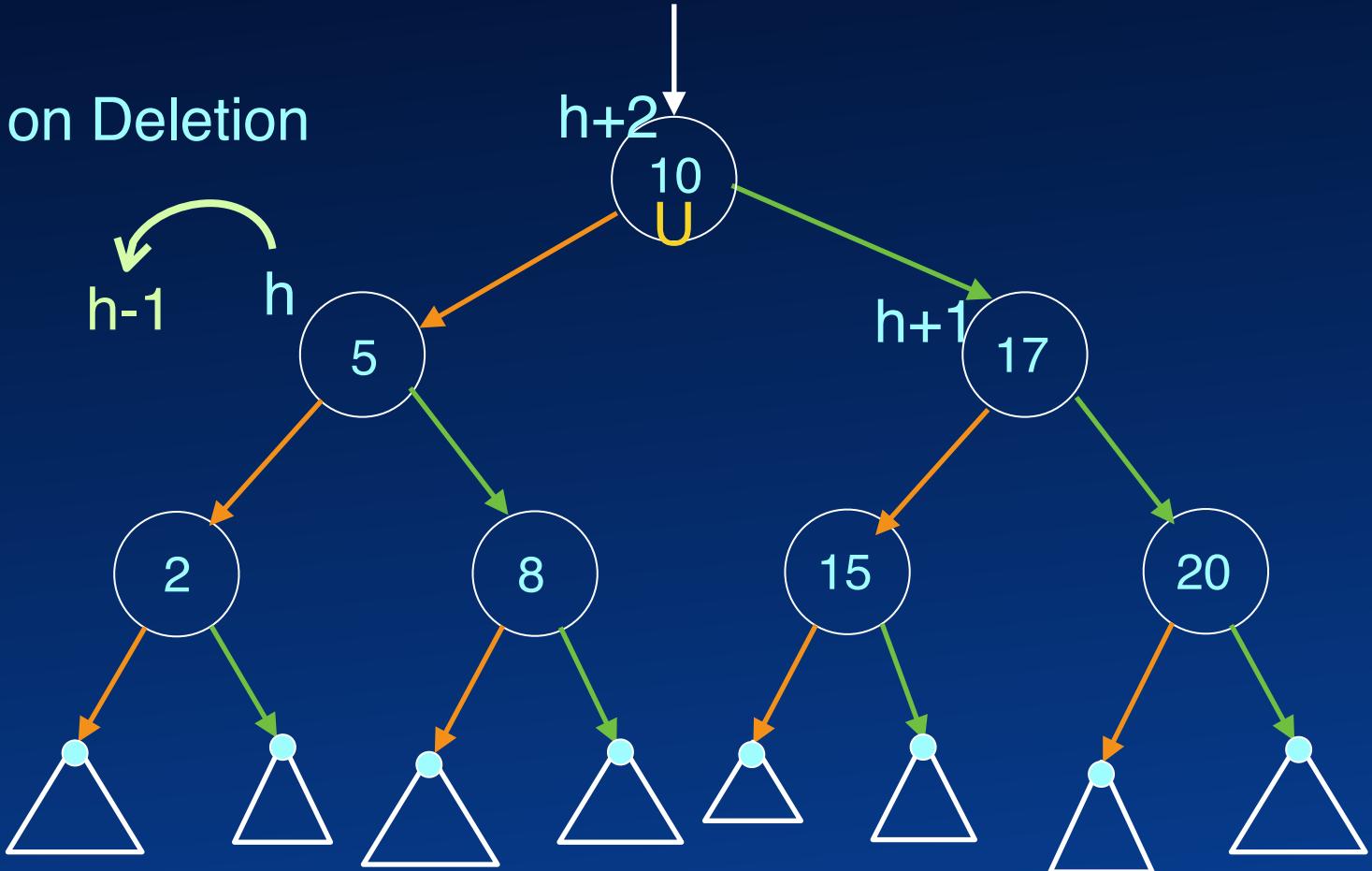
Imbalance on Deletion





Rotation

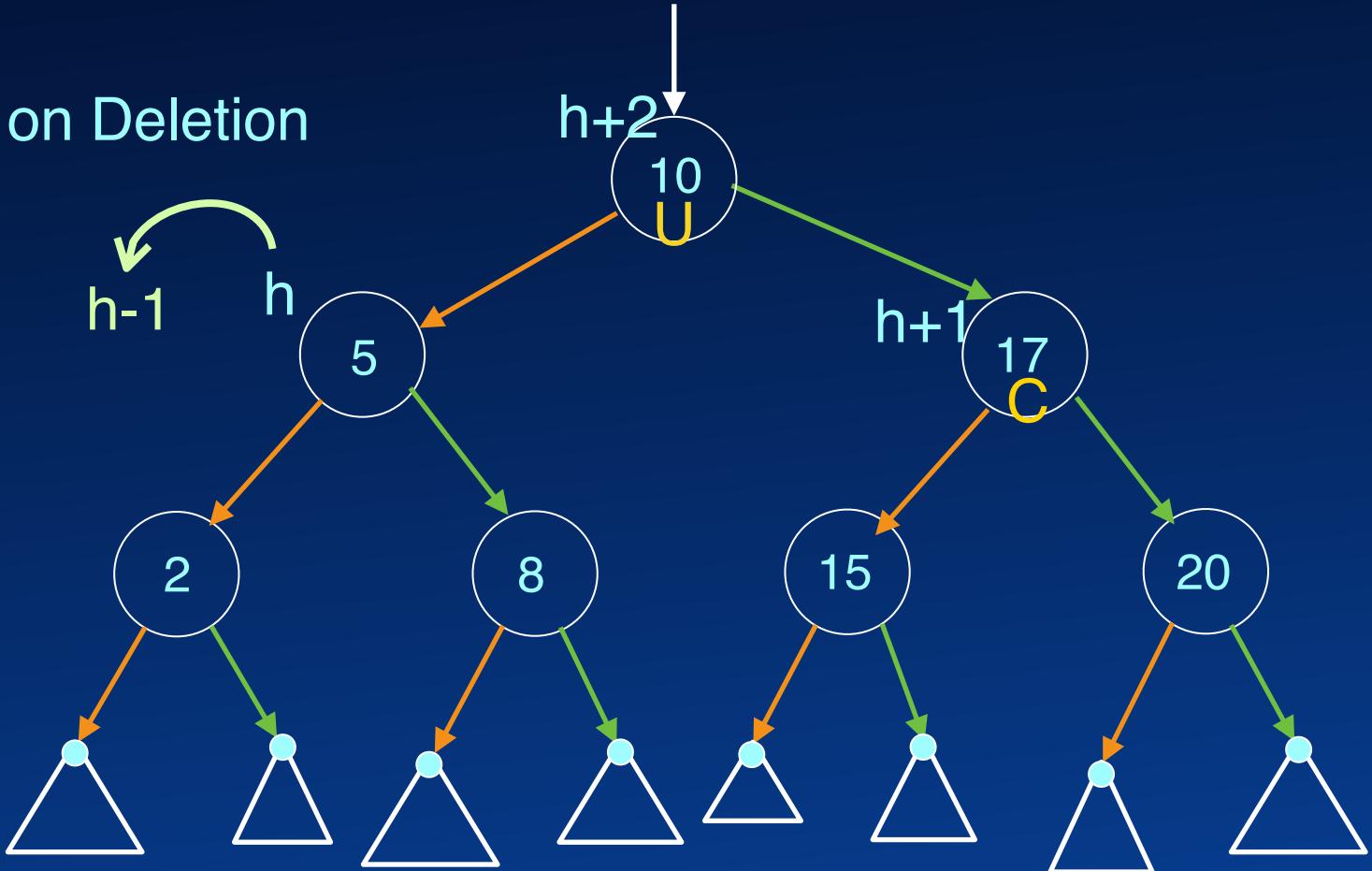
Imbalance on Deletion





Rotation

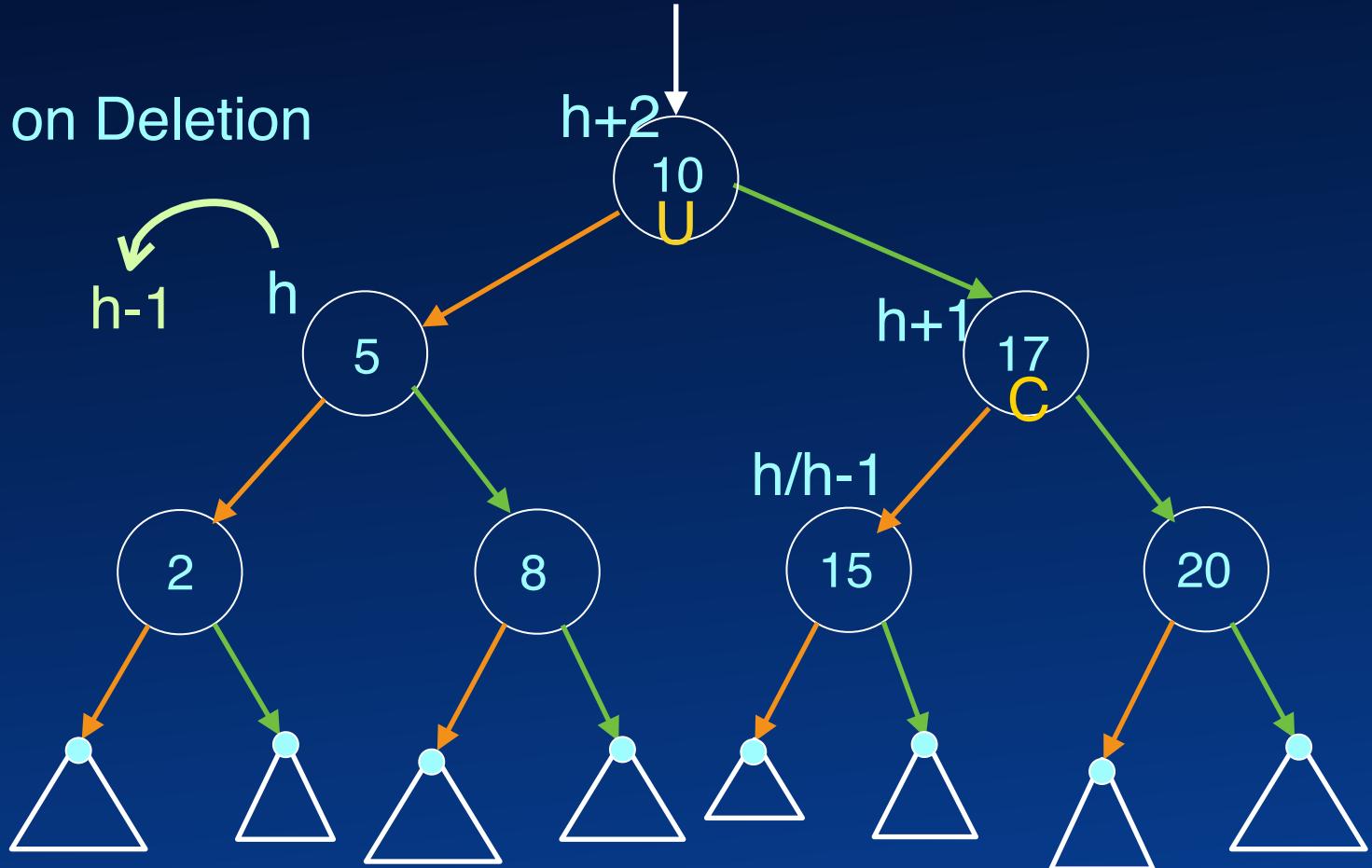
Imbalance on Deletion





Rotation

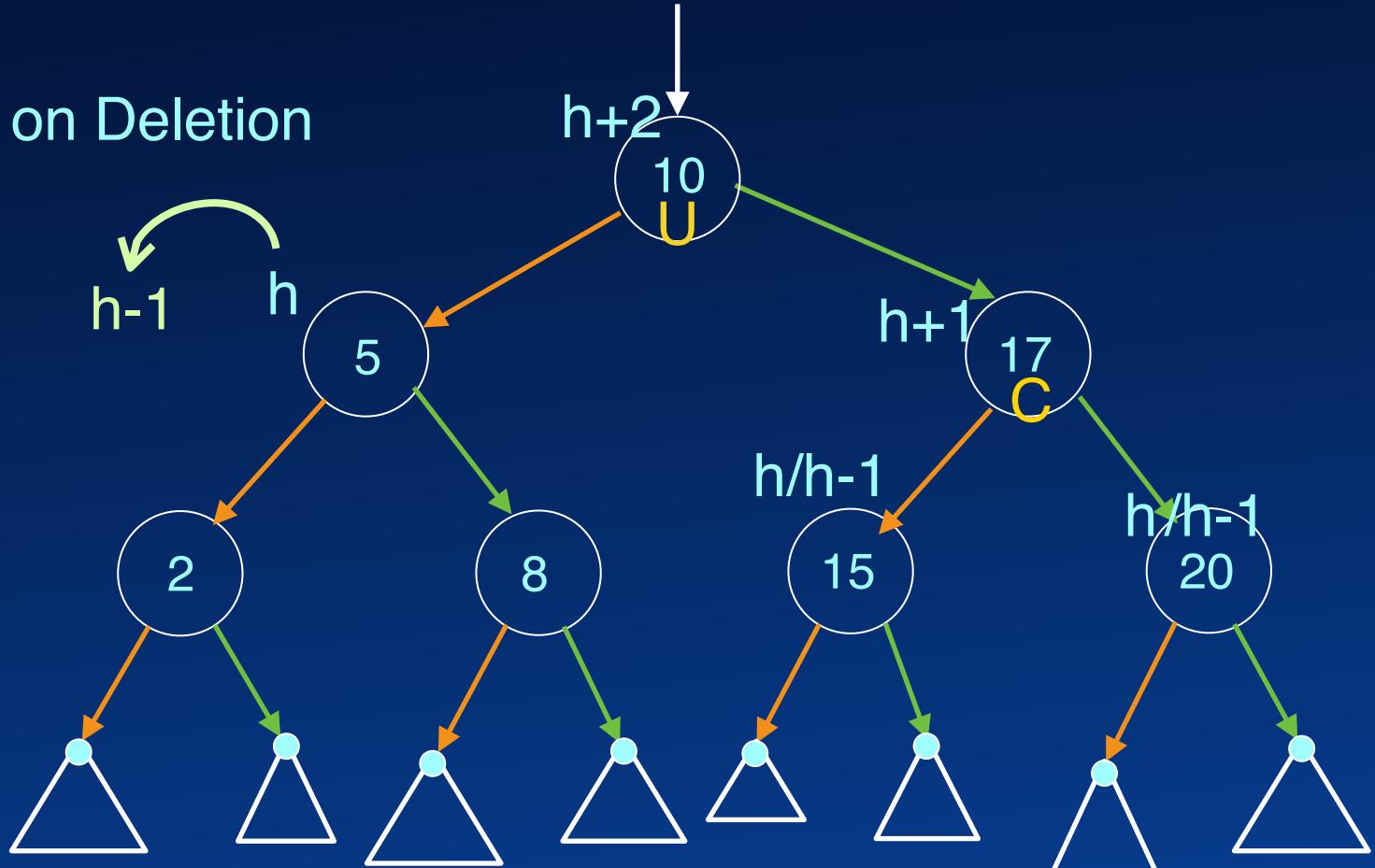
Imbalance on Deletion





Rotation

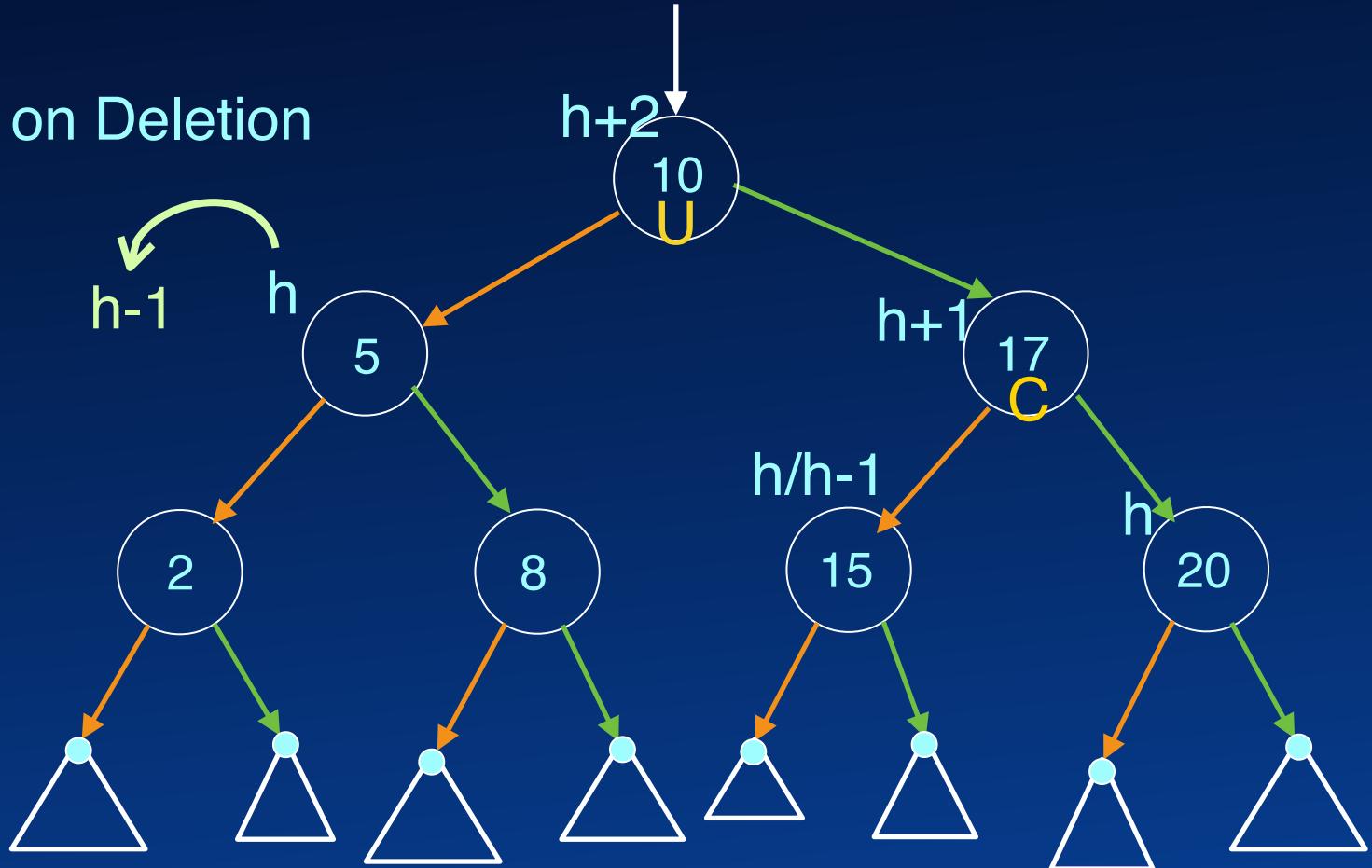
Imbalance on Deletion





Rotation

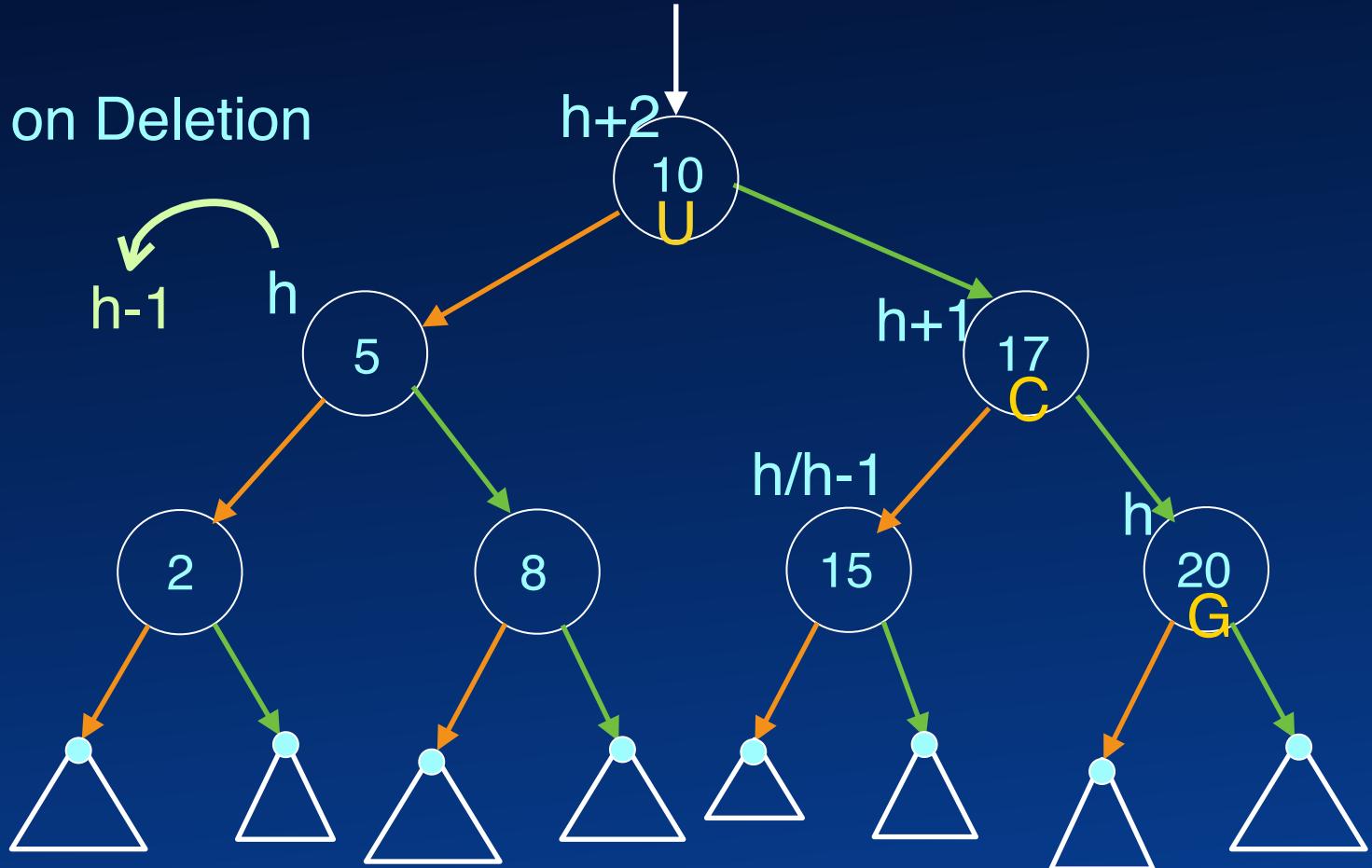
Imbalance on Deletion





Rotation

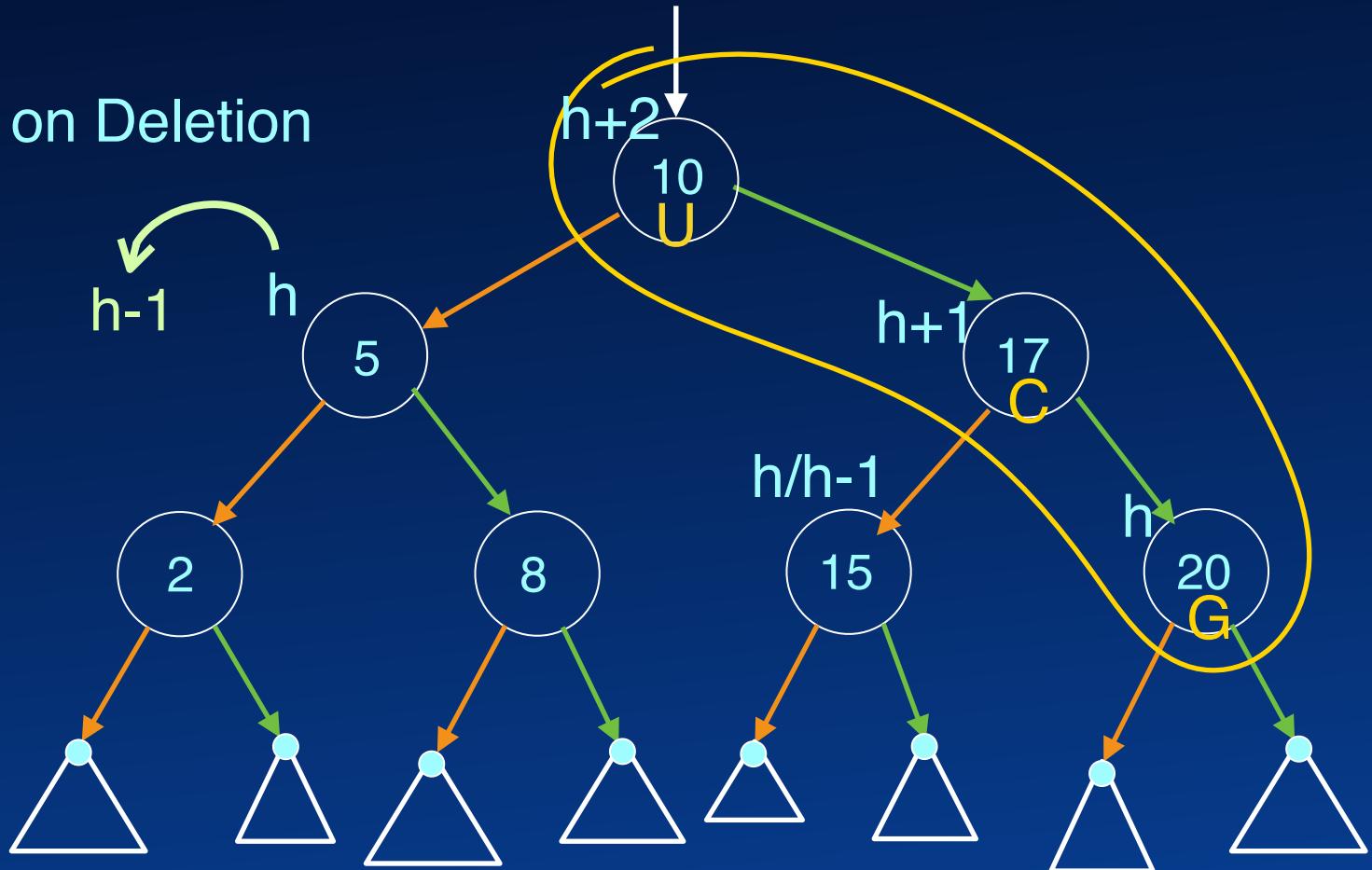
Imbalance on Deletion





Rotation

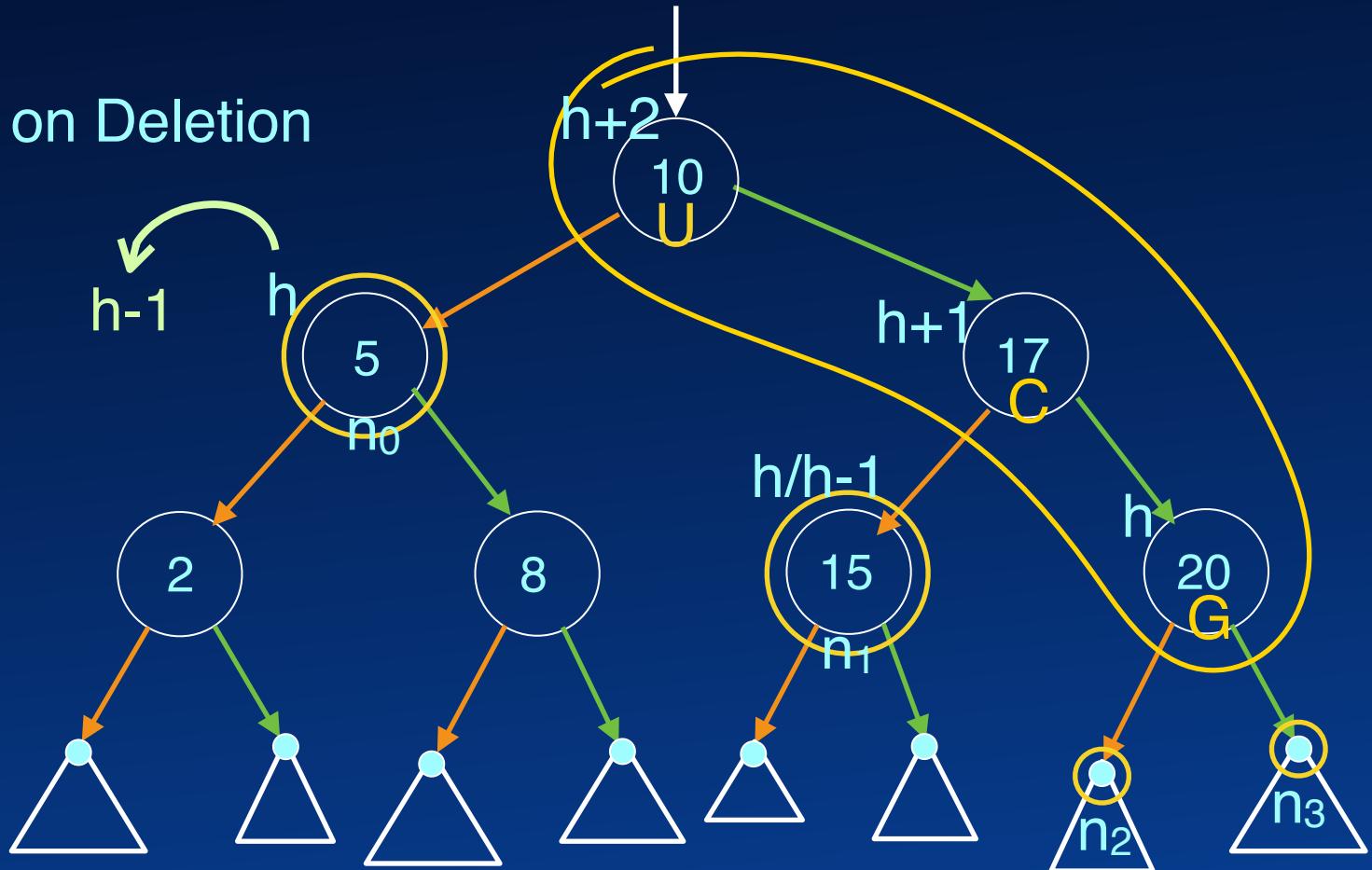
Imbalance on Deletion





Rotation

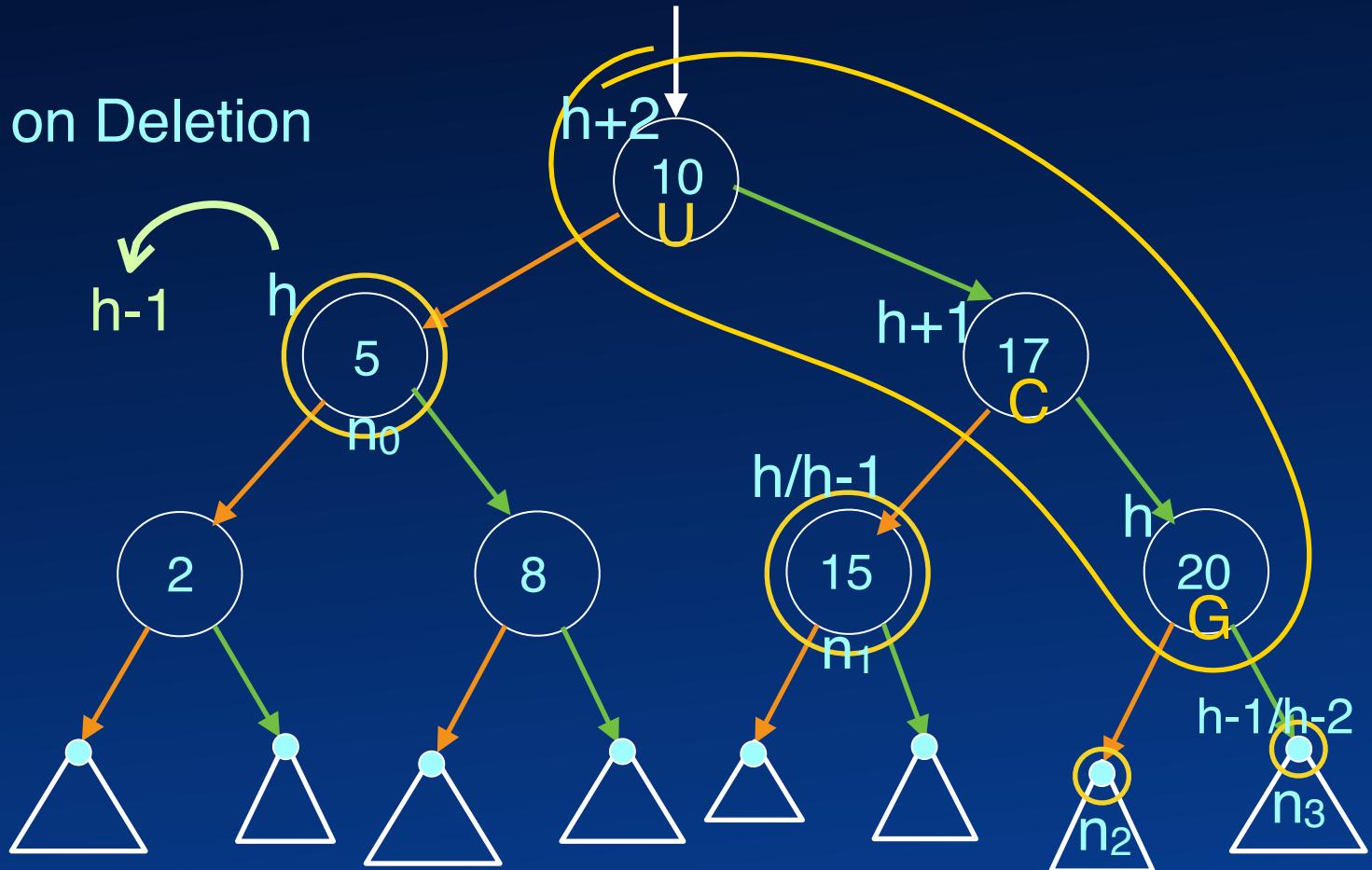
Imbalance on Deletion





Rotation

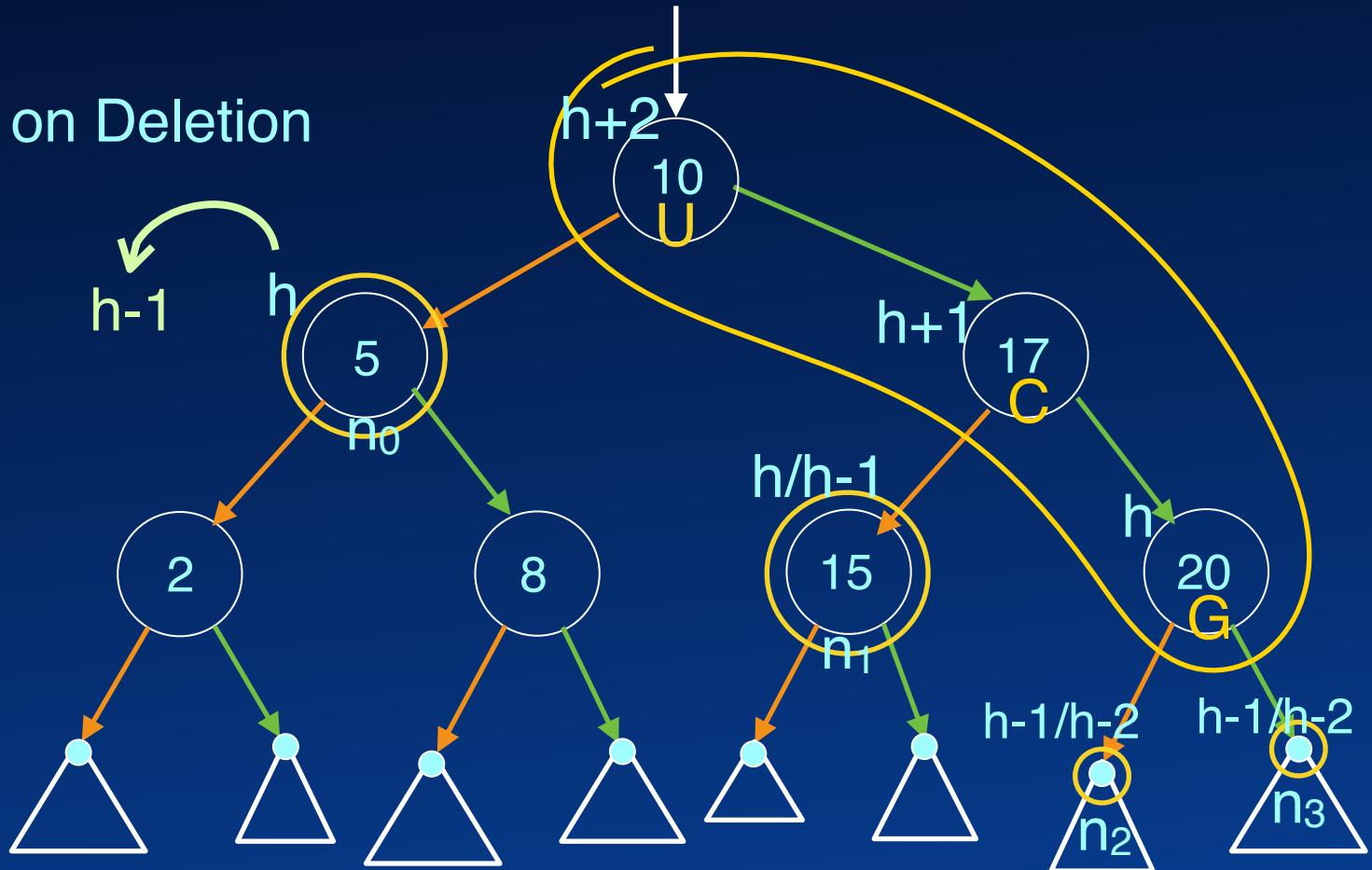
Imbalance on Deletion





Rotation

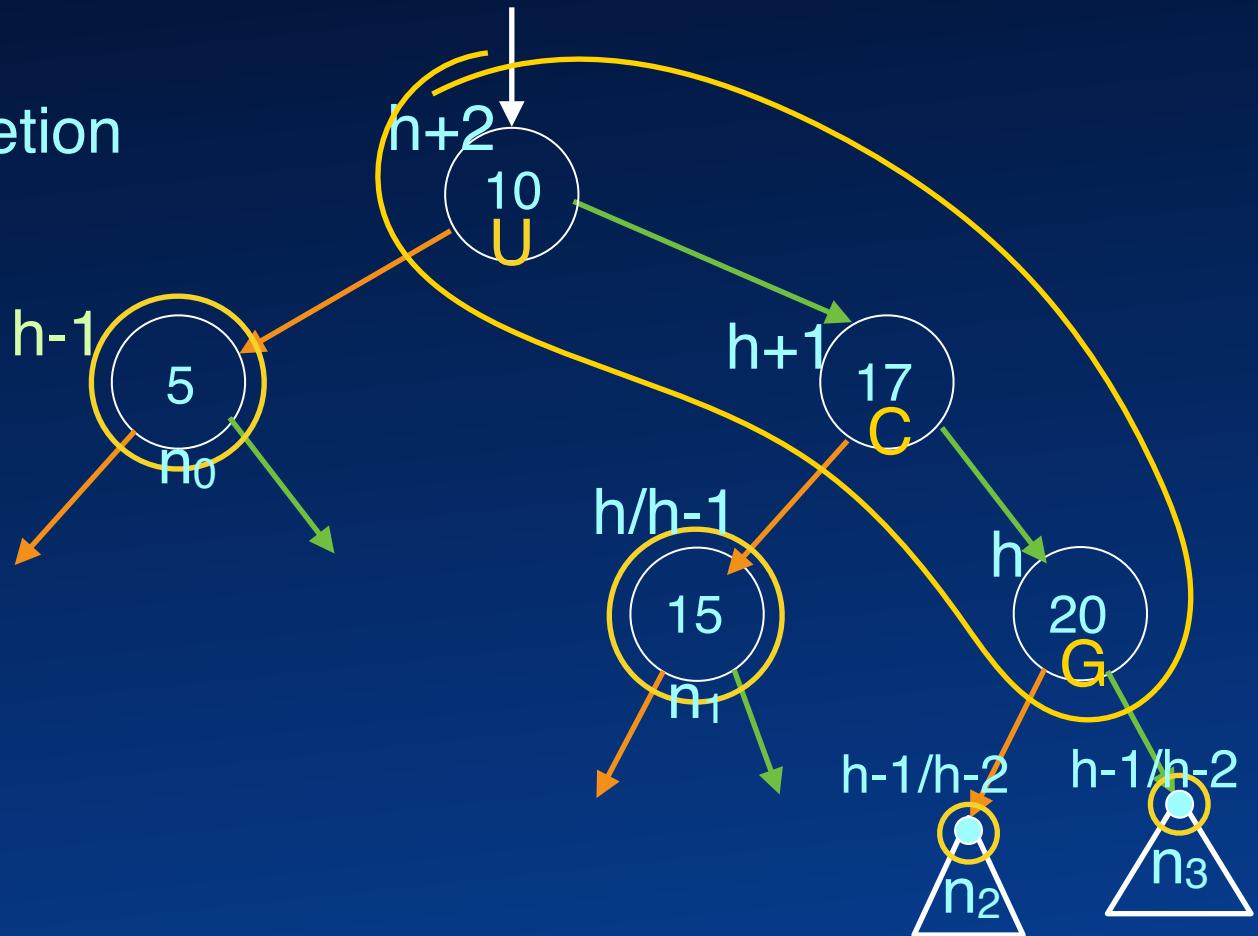
Imbalance on Deletion





Rotation

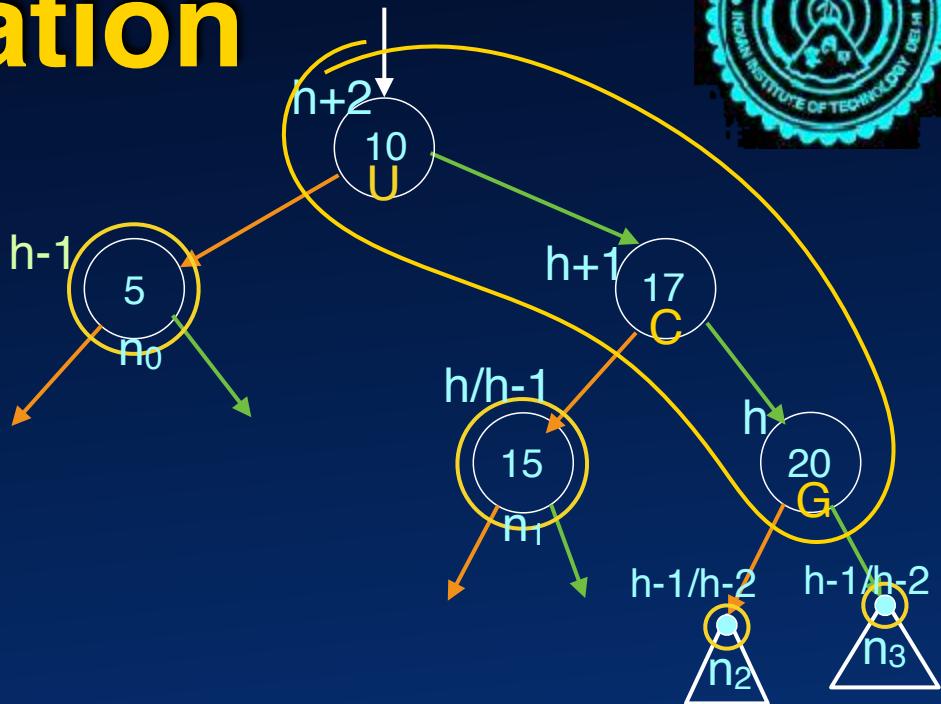
Imbalance on Deletion





Rotation

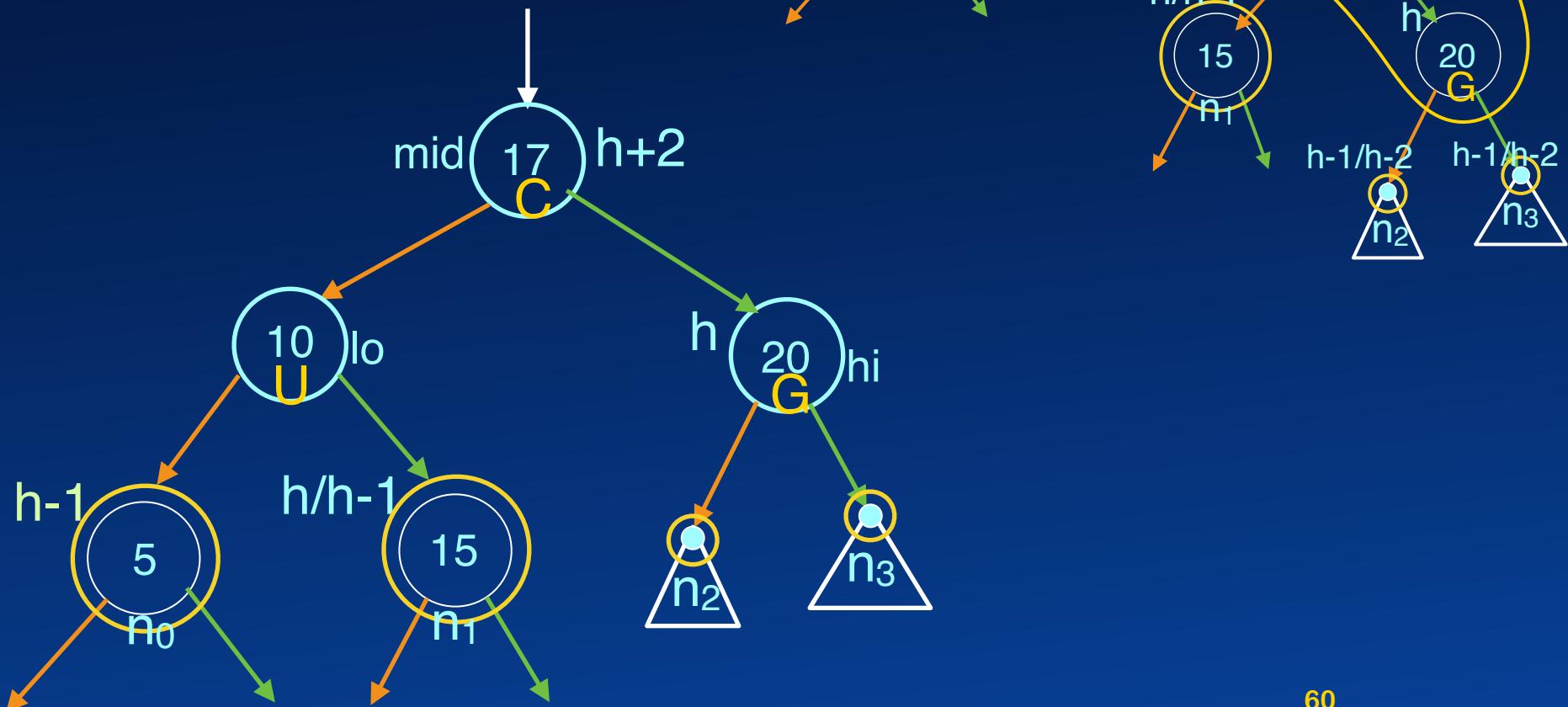
Imbalance on Deletion





Rotation

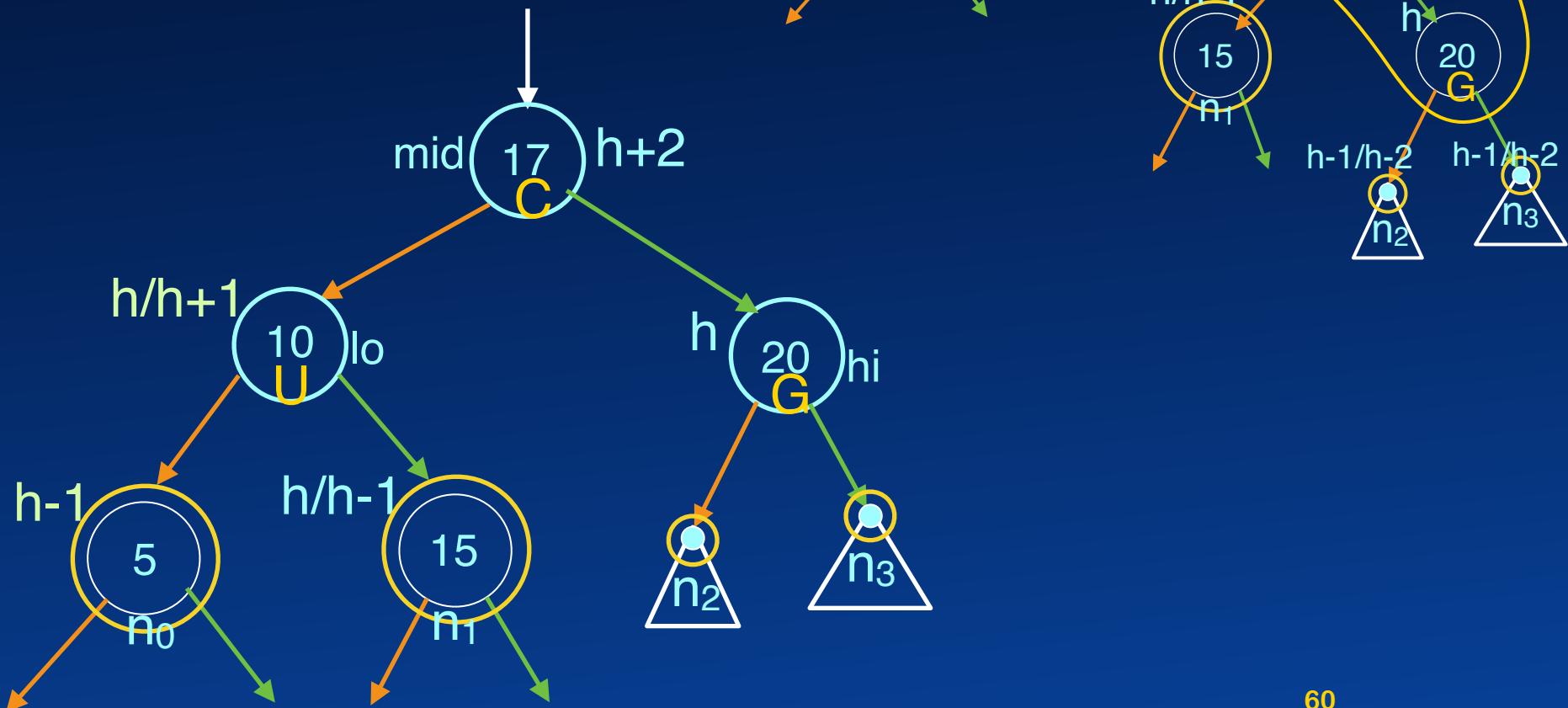
Imbalance on Deletion





Rotation

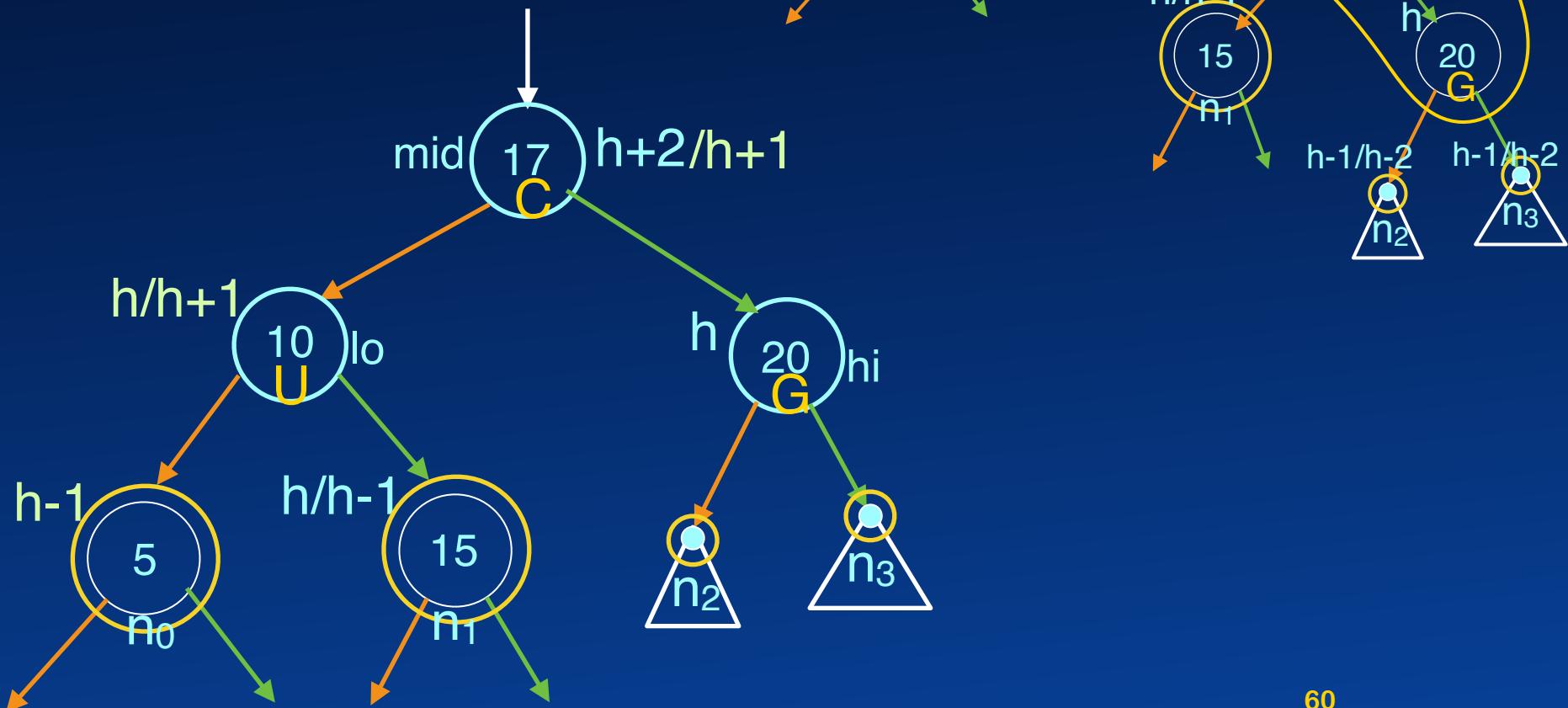
Imbalance on Deletion





Rotation

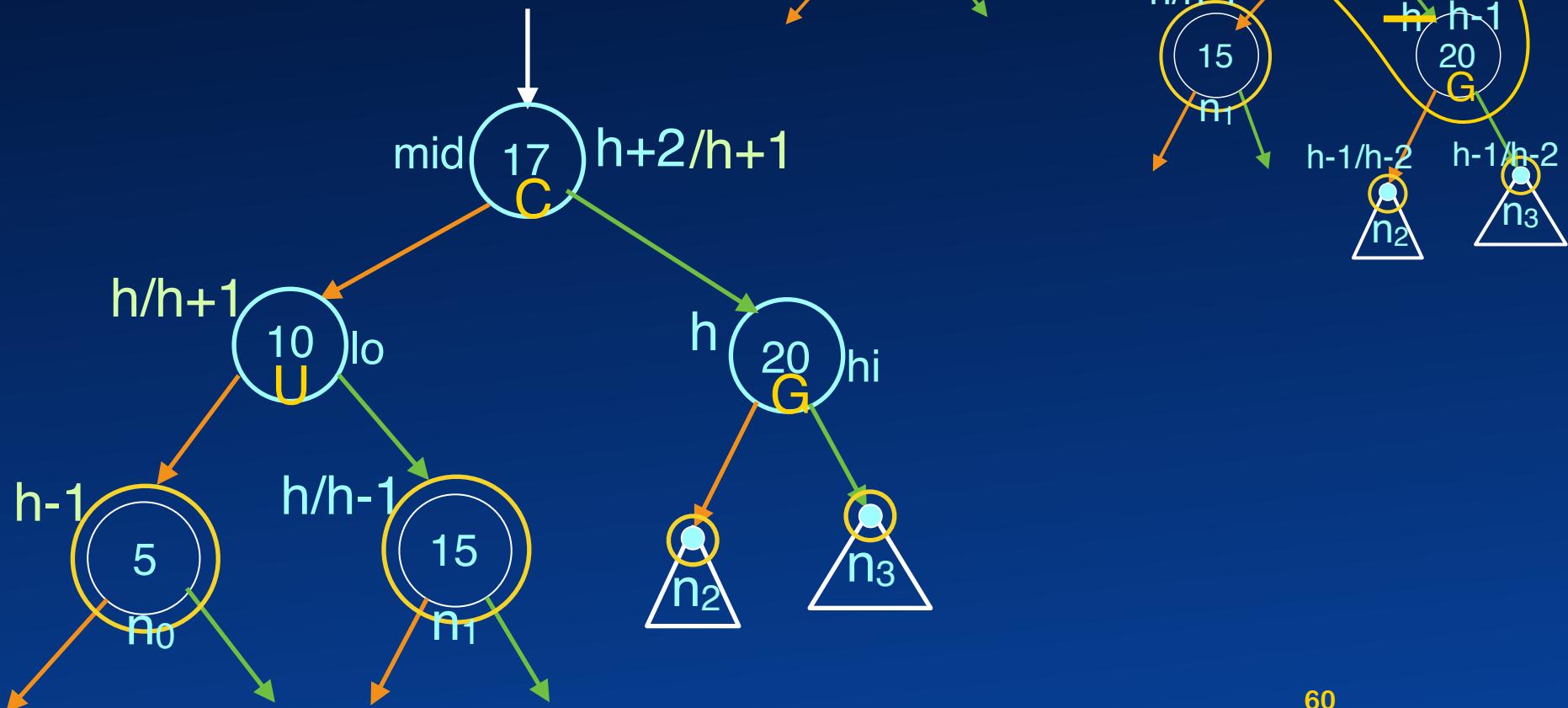
Imbalance on Deletion





Rotation

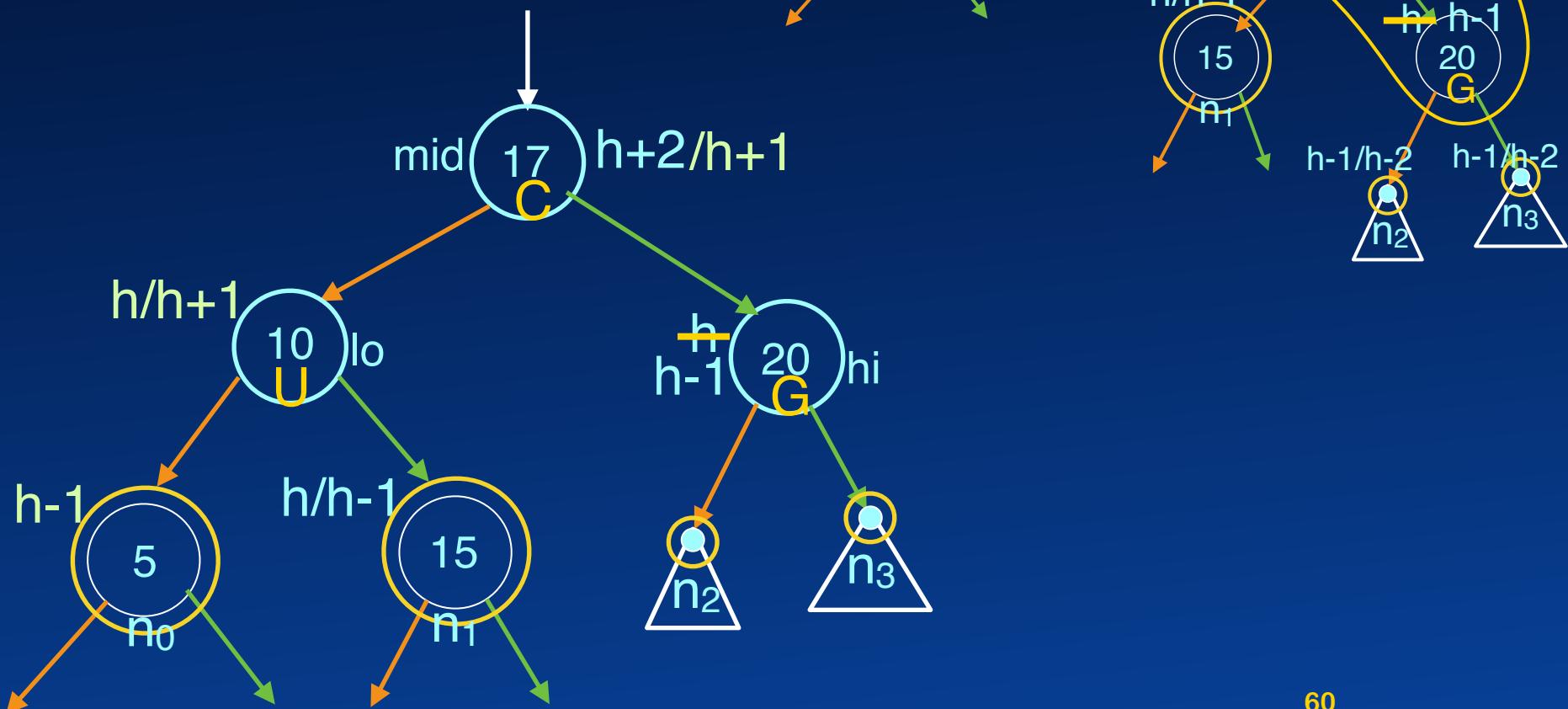
Imbalance on Deletion





Rotation

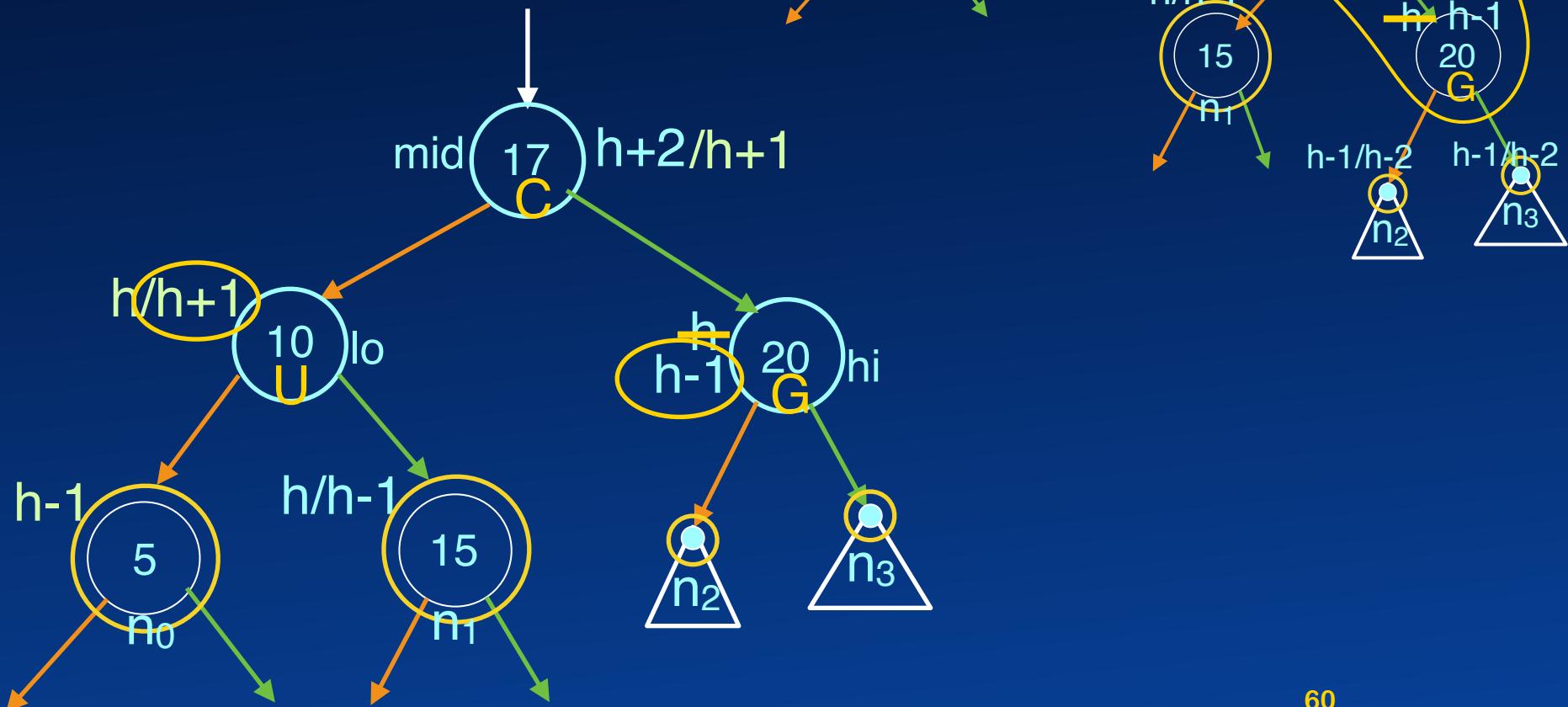
Imbalance on Deletion





Rotation

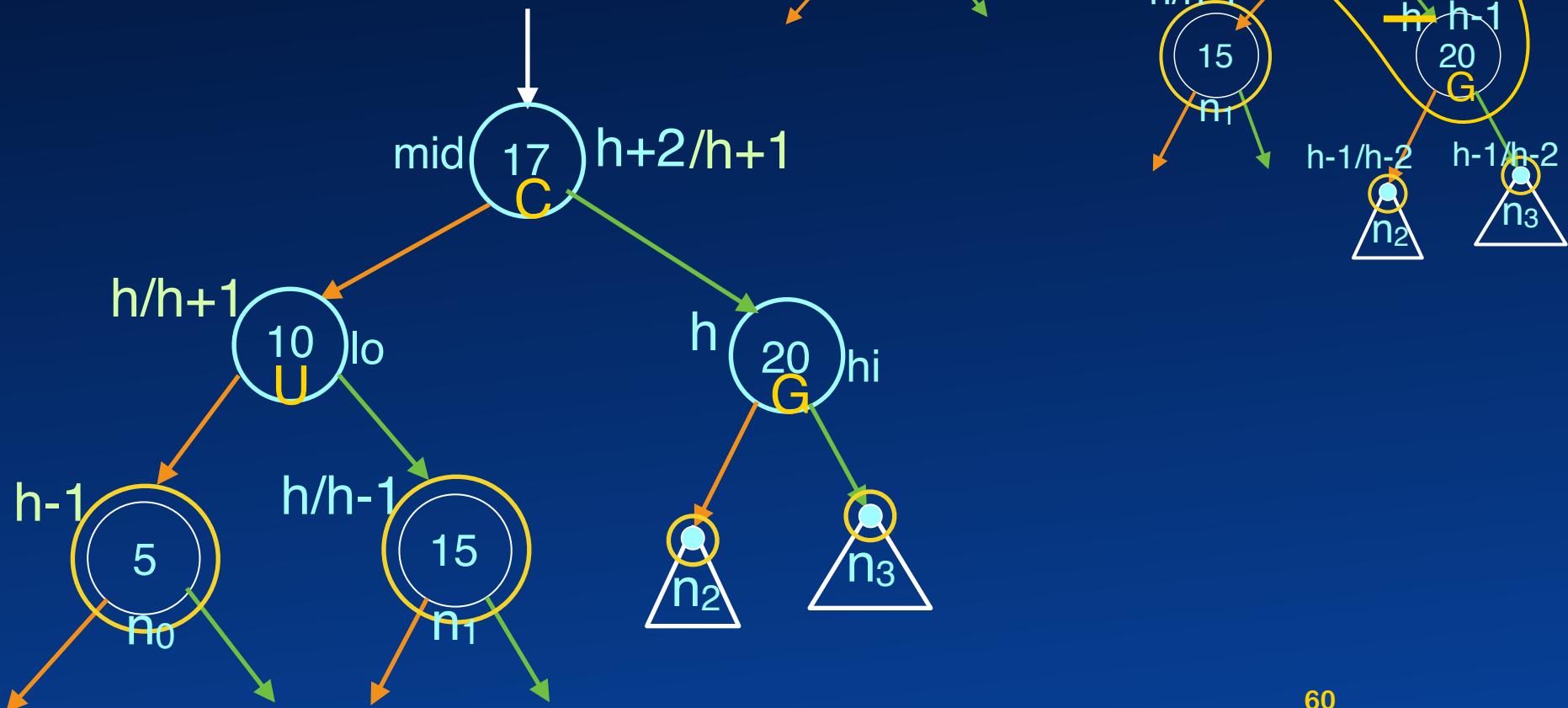
Imbalance on Deletion





Rotation

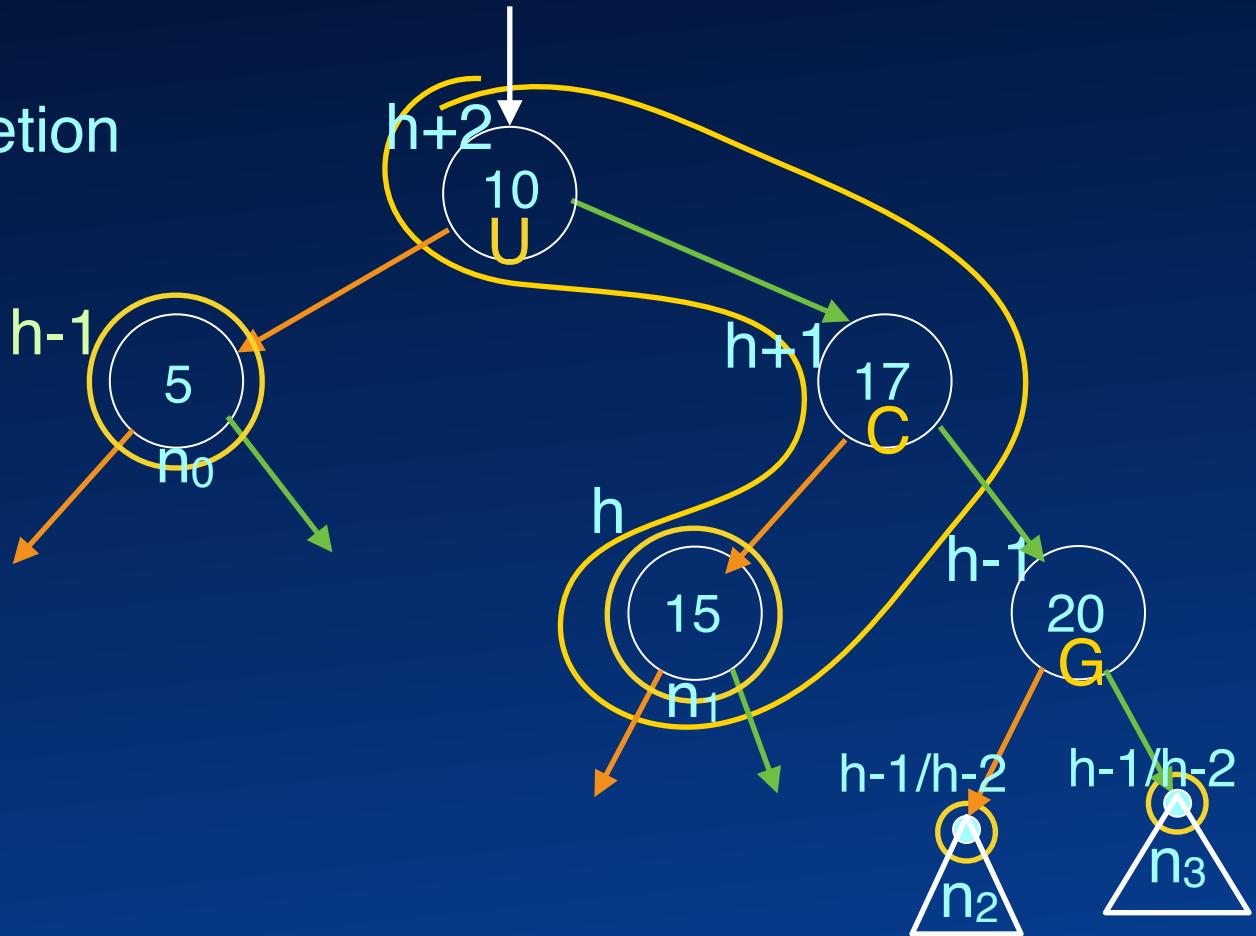
Imbalance on Deletion





Rotation

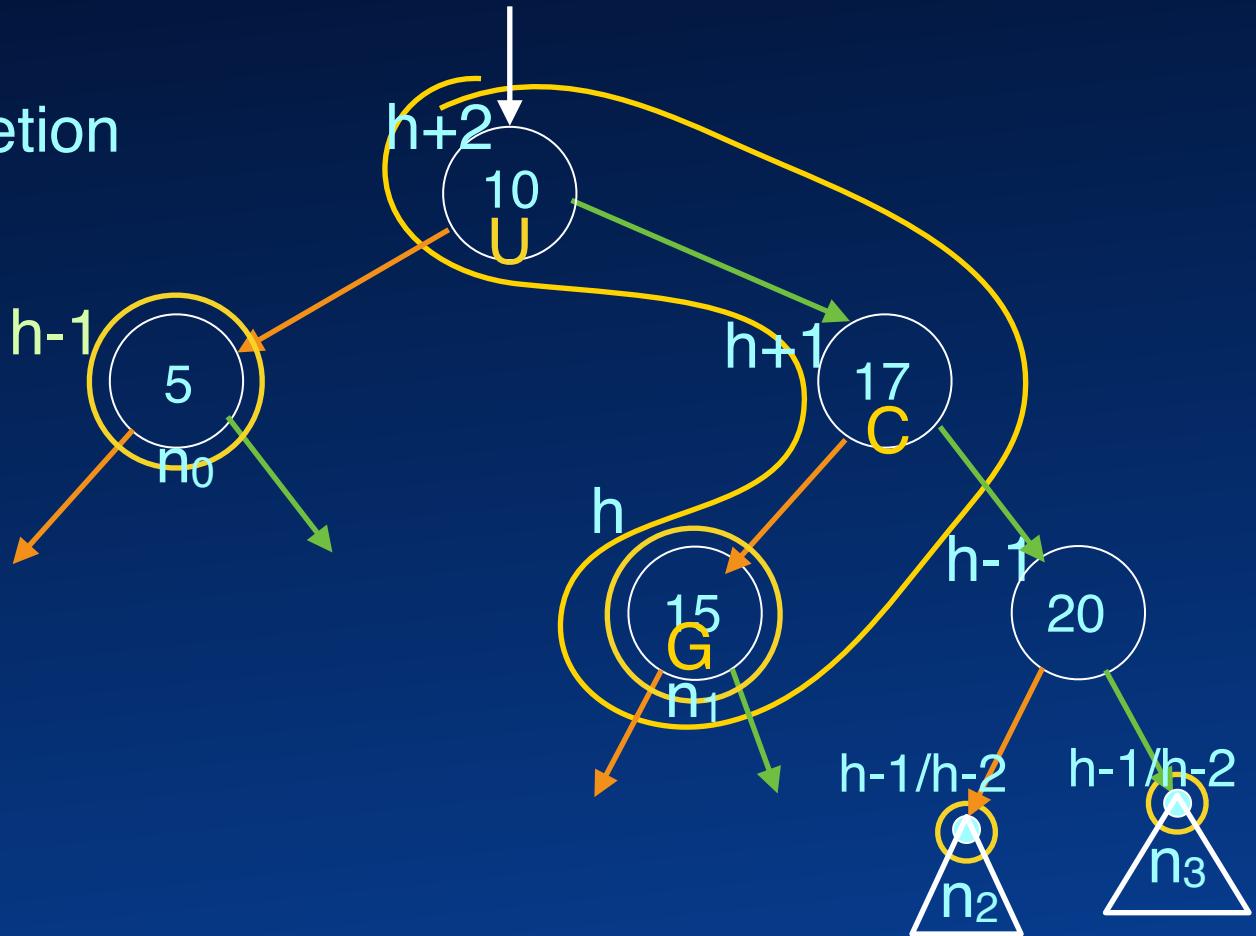
Imbalance on Deletion
Case 2





Rotation

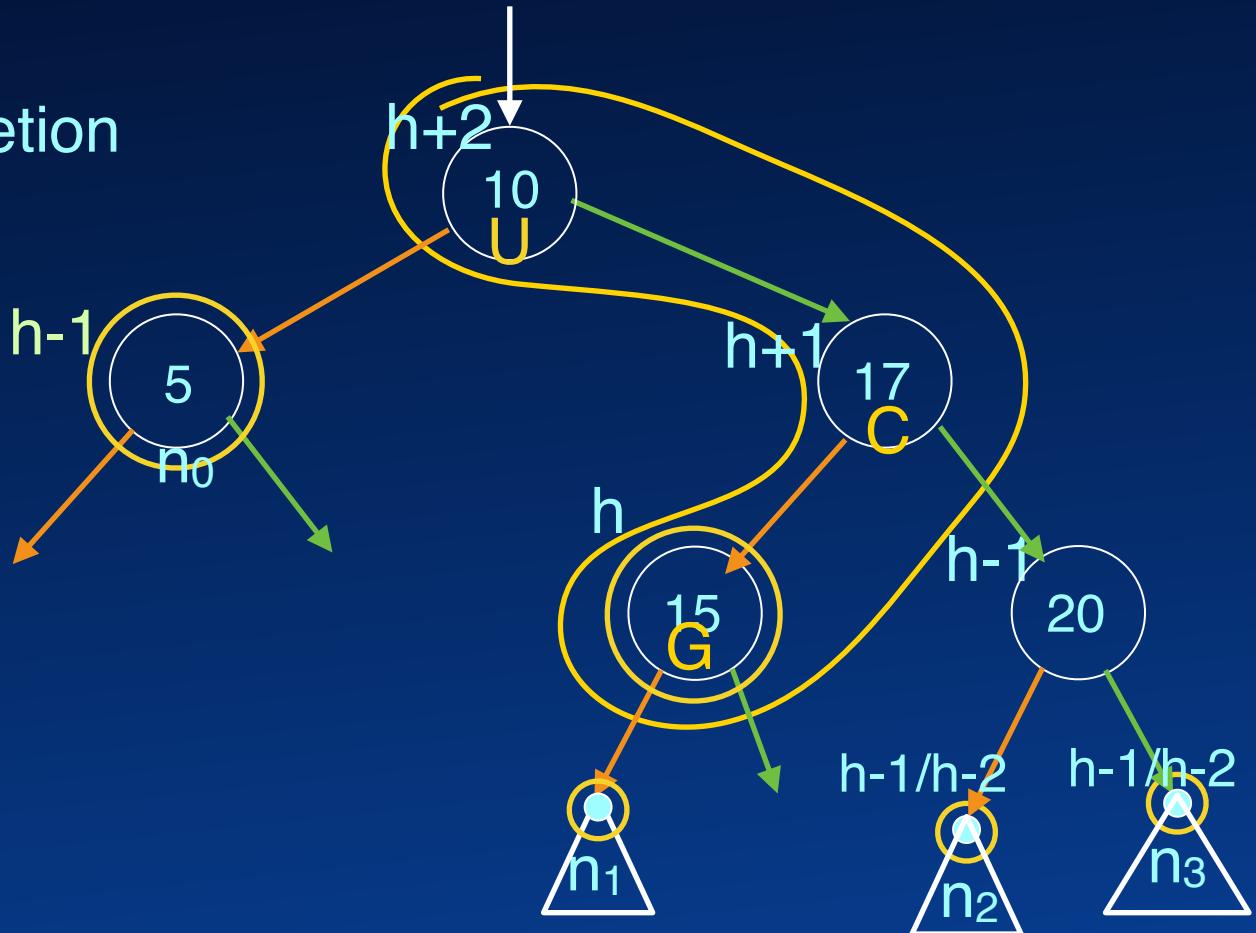
Imbalance on Deletion
Case 2





Rotation

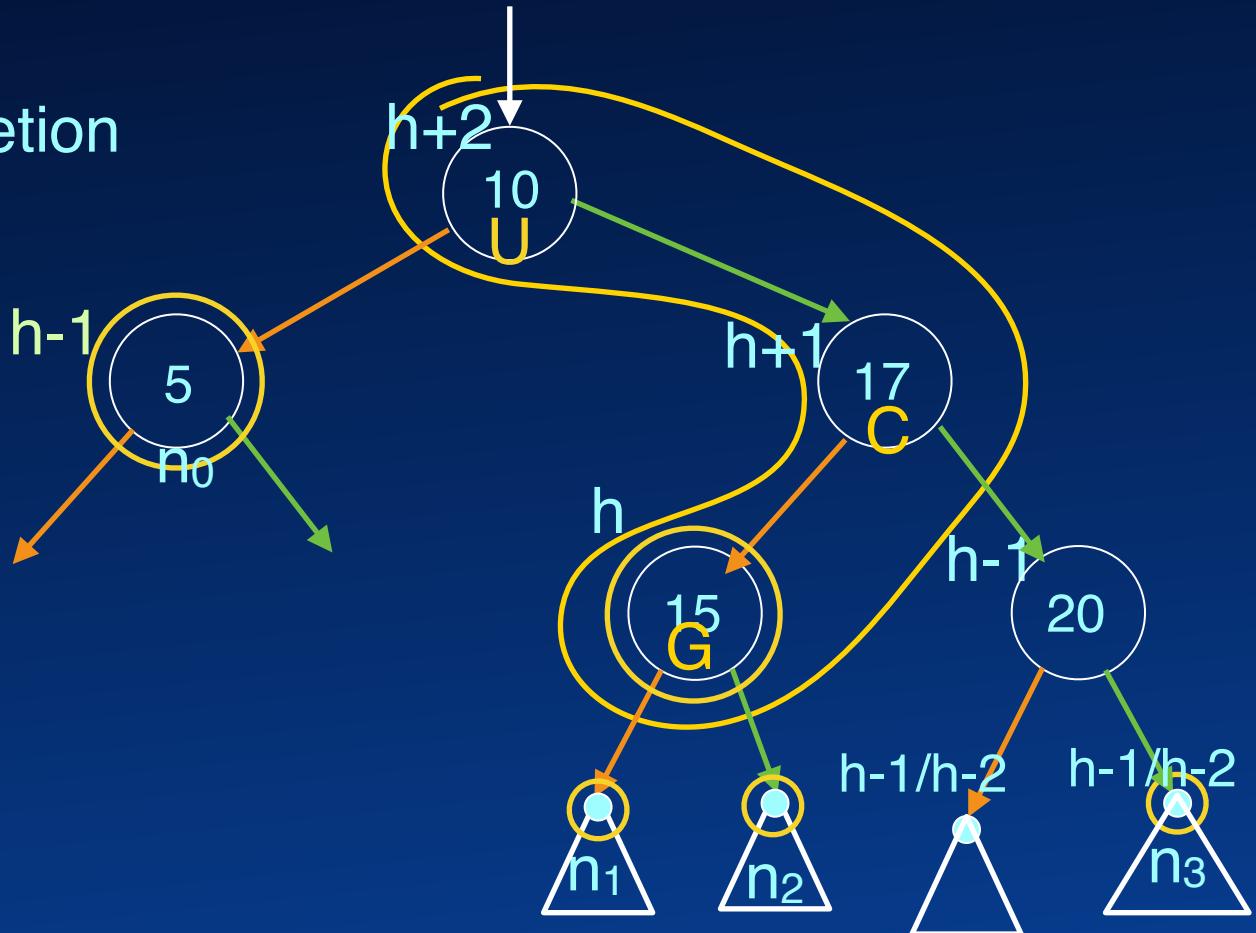
Imbalance on Deletion
Case 2





Rotation

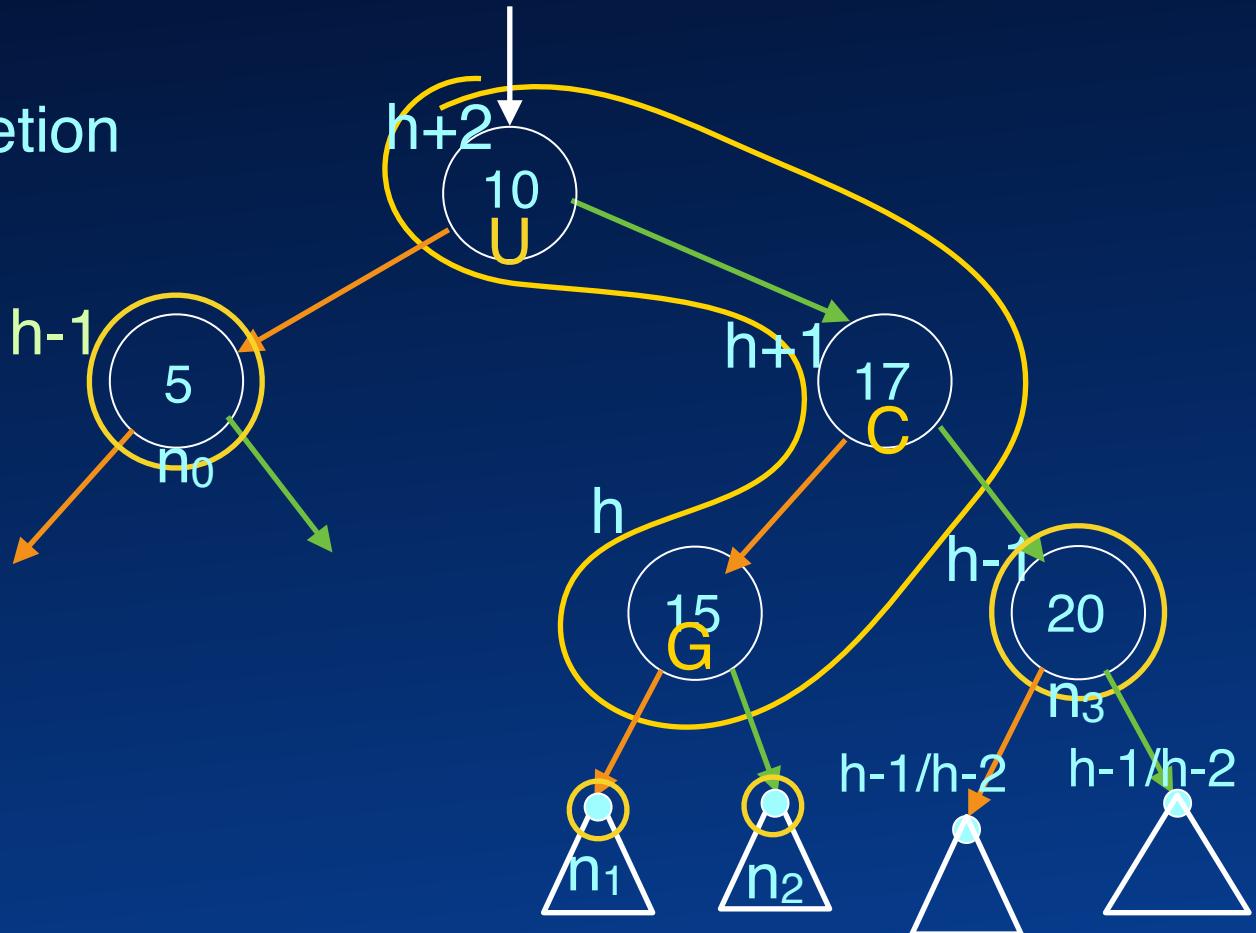
Imbalance on Deletion
Case 2





Rotation

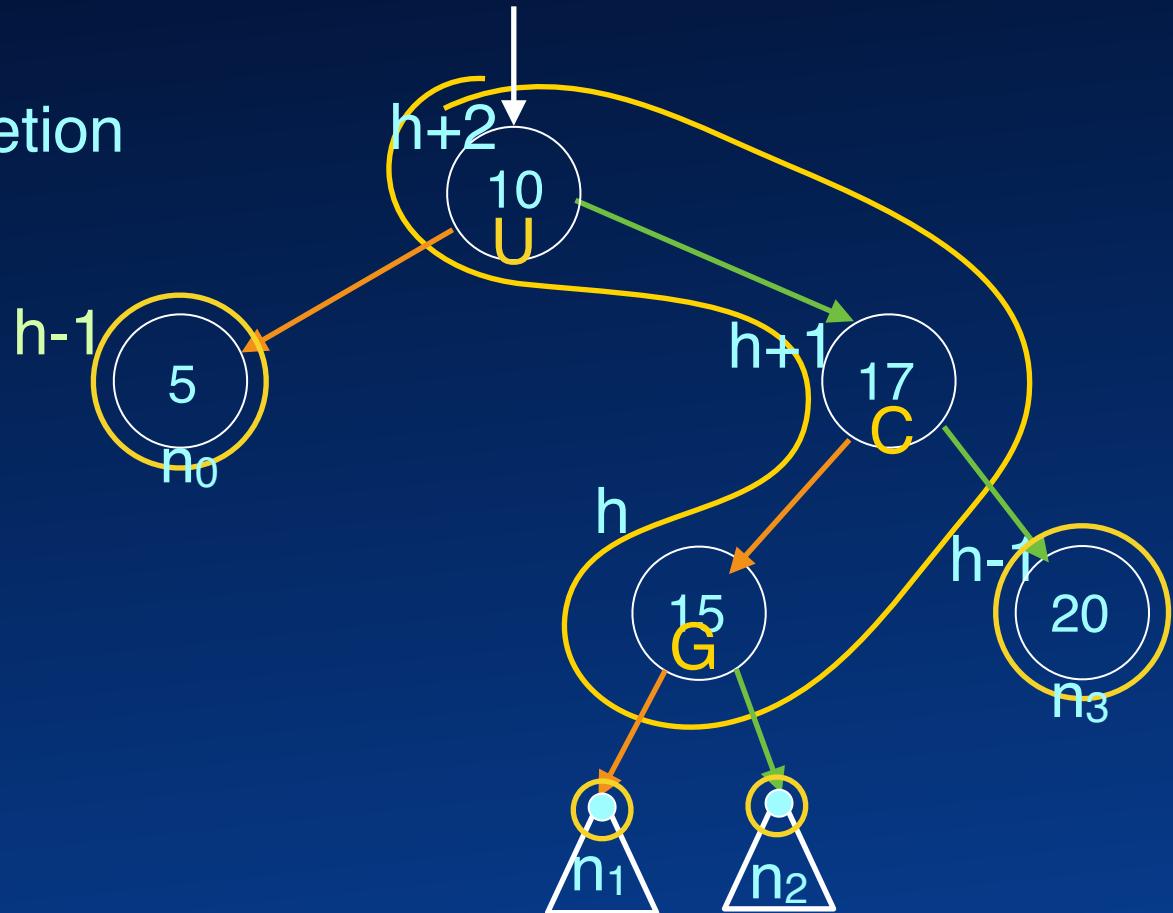
Imbalance on Deletion
Case 2





Rotation

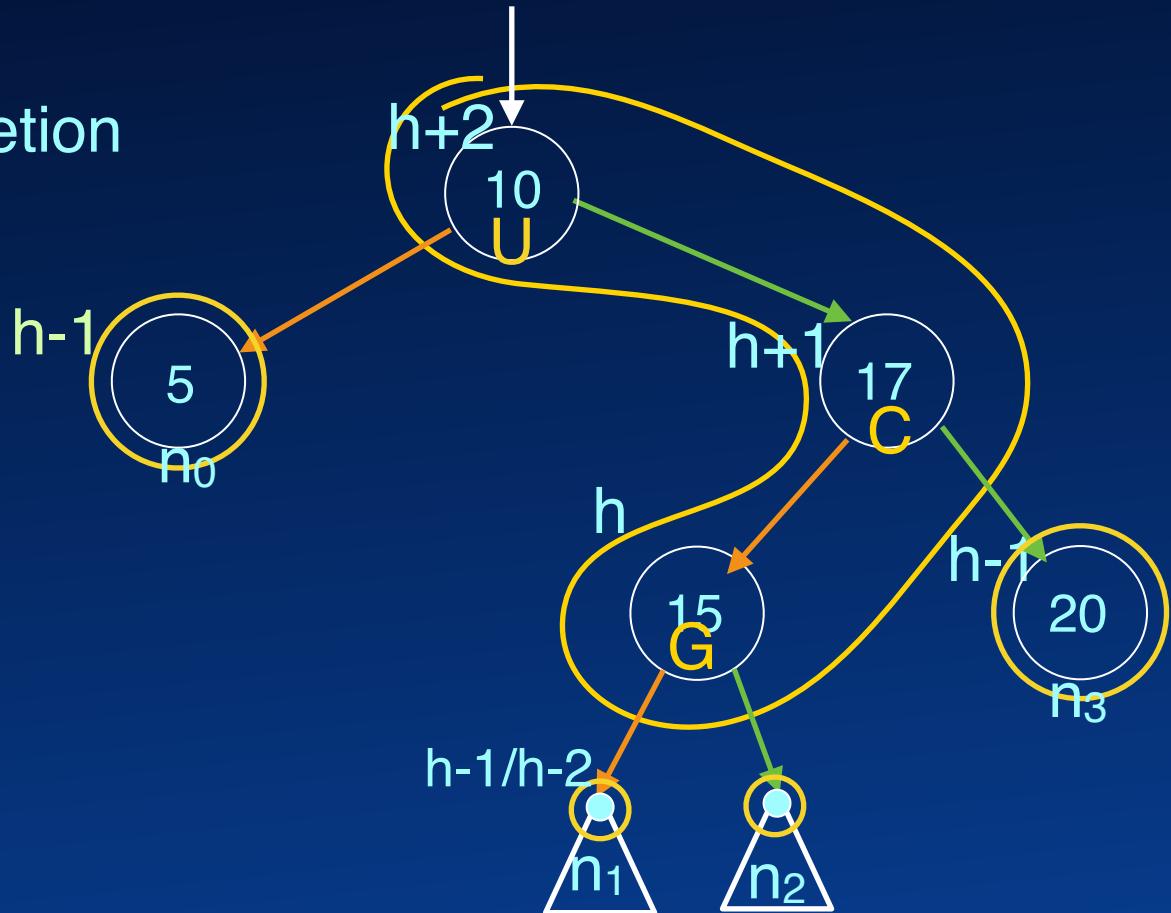
Imbalance on Deletion
Case 2





Rotation

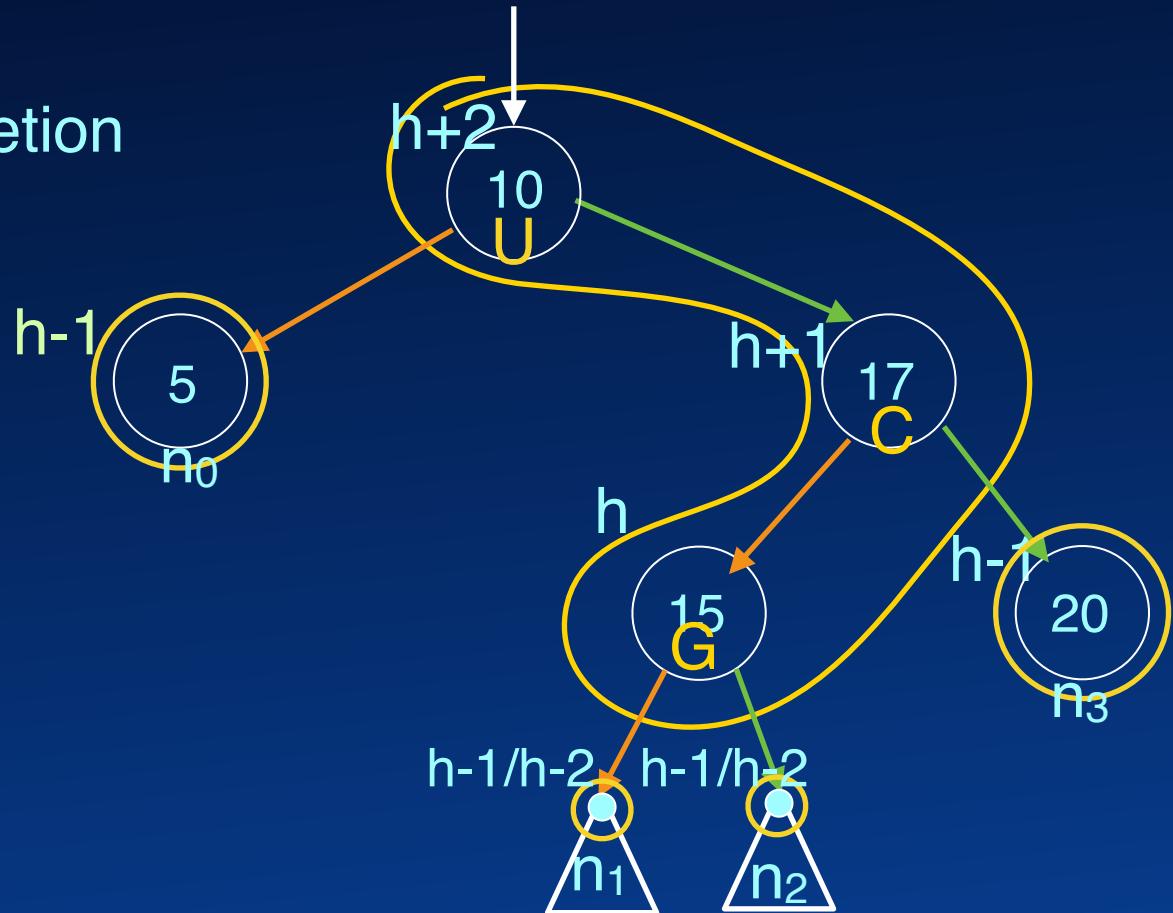
Imbalance on Deletion
Case 2





Rotation

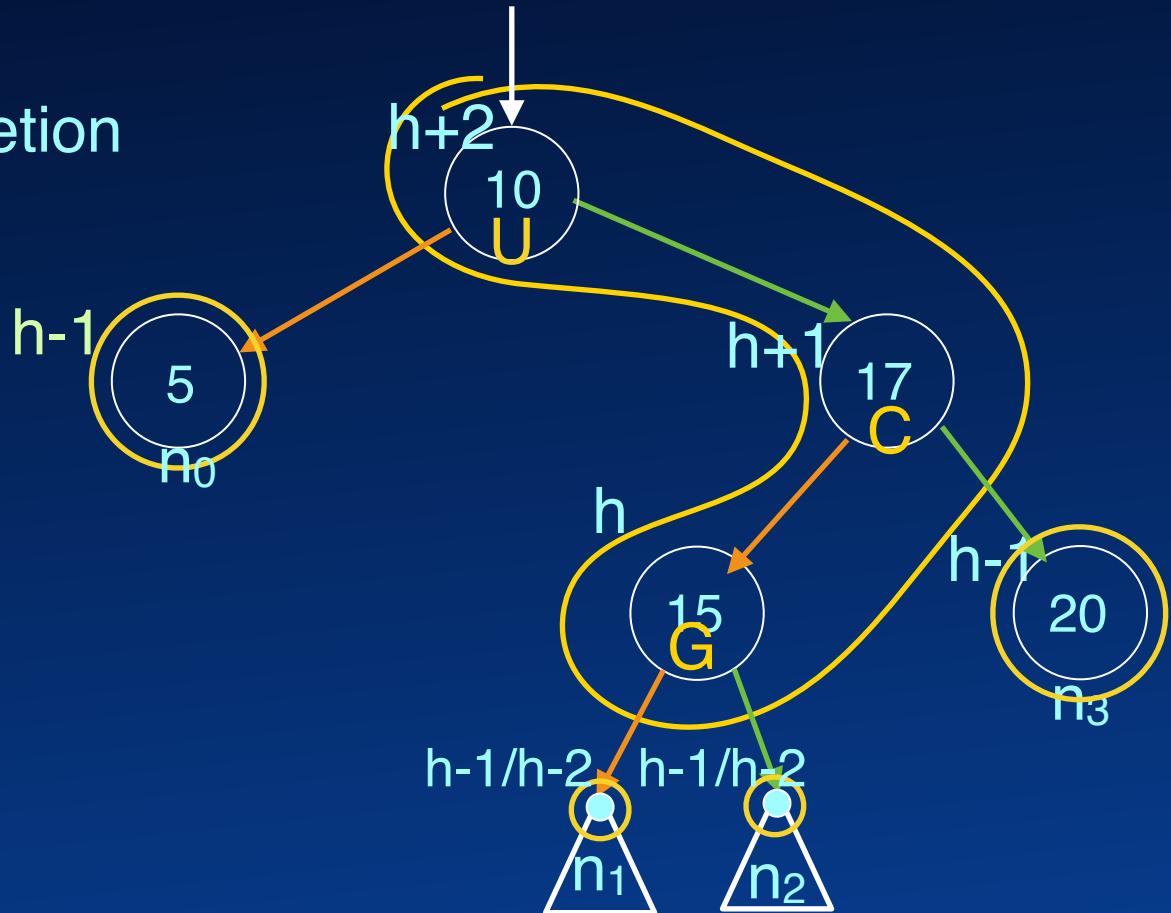
Imbalance on Deletion
Case 2





Rotation

Imbalance on Deletion
Case 2

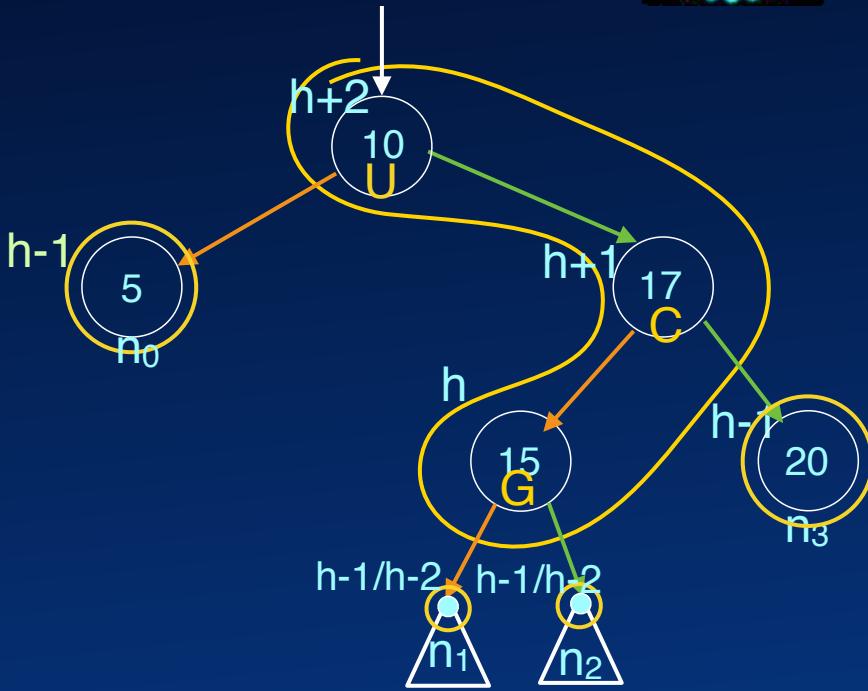
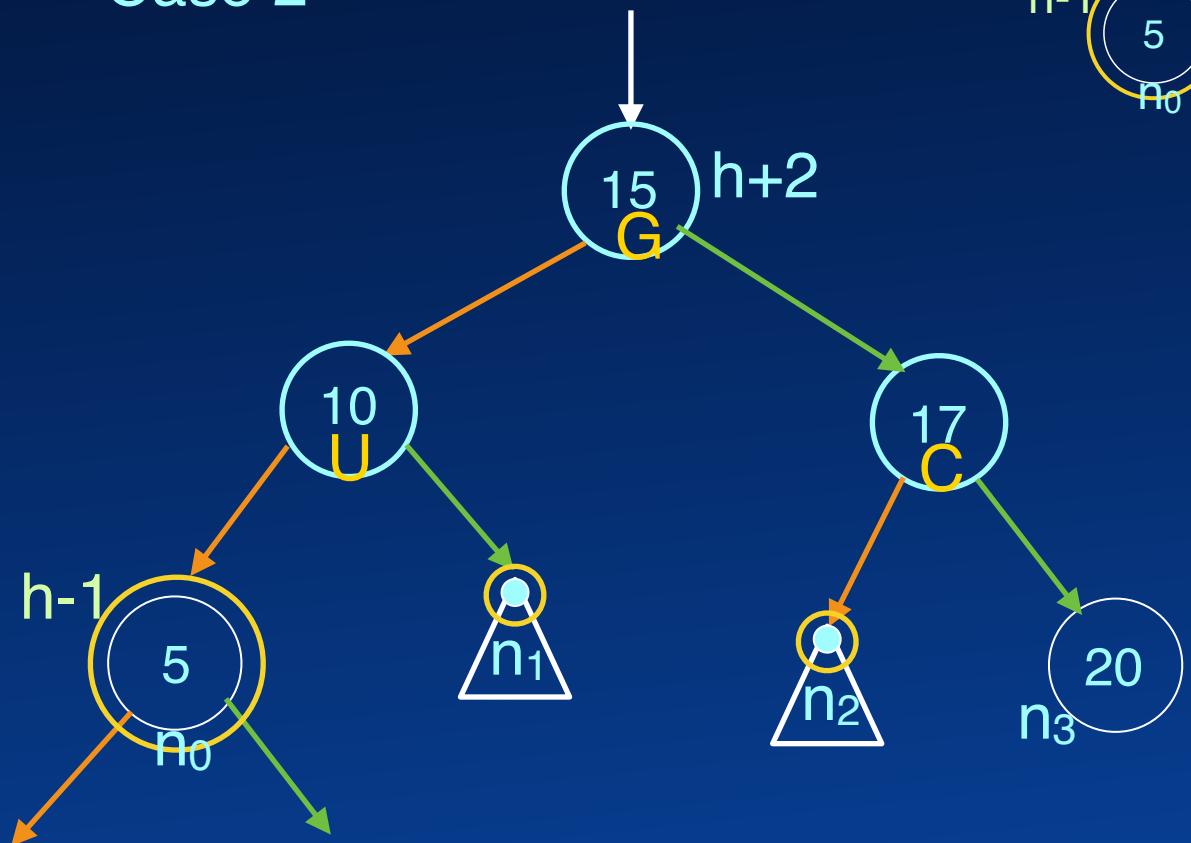


Rotation



Imbalance on Deletion

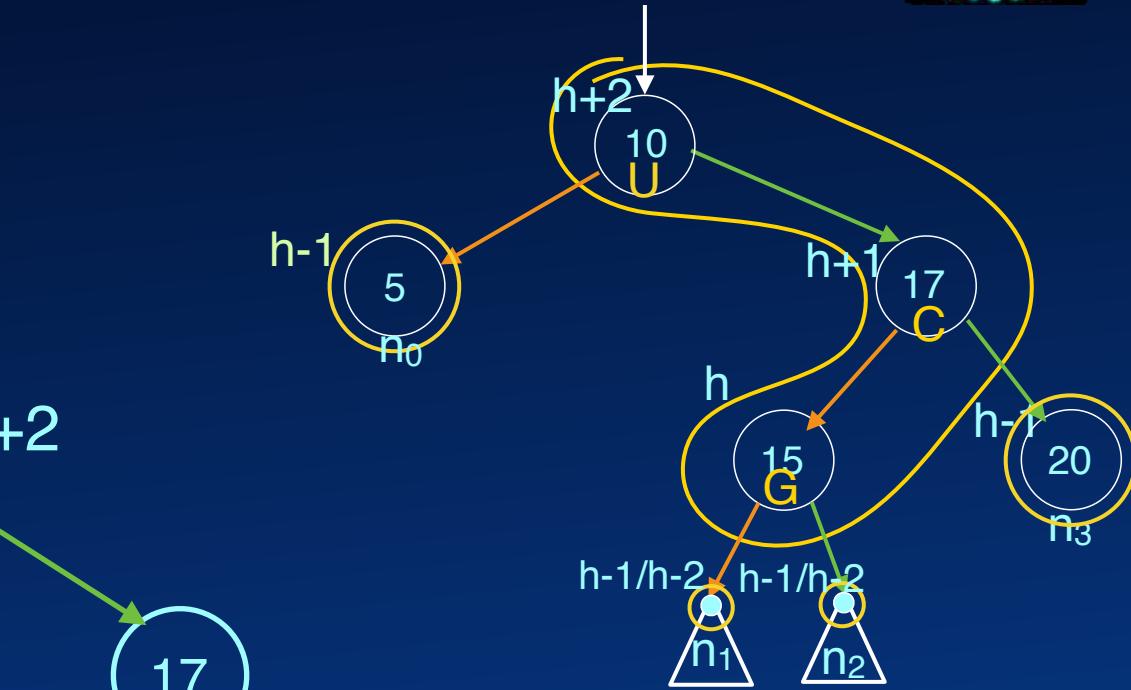
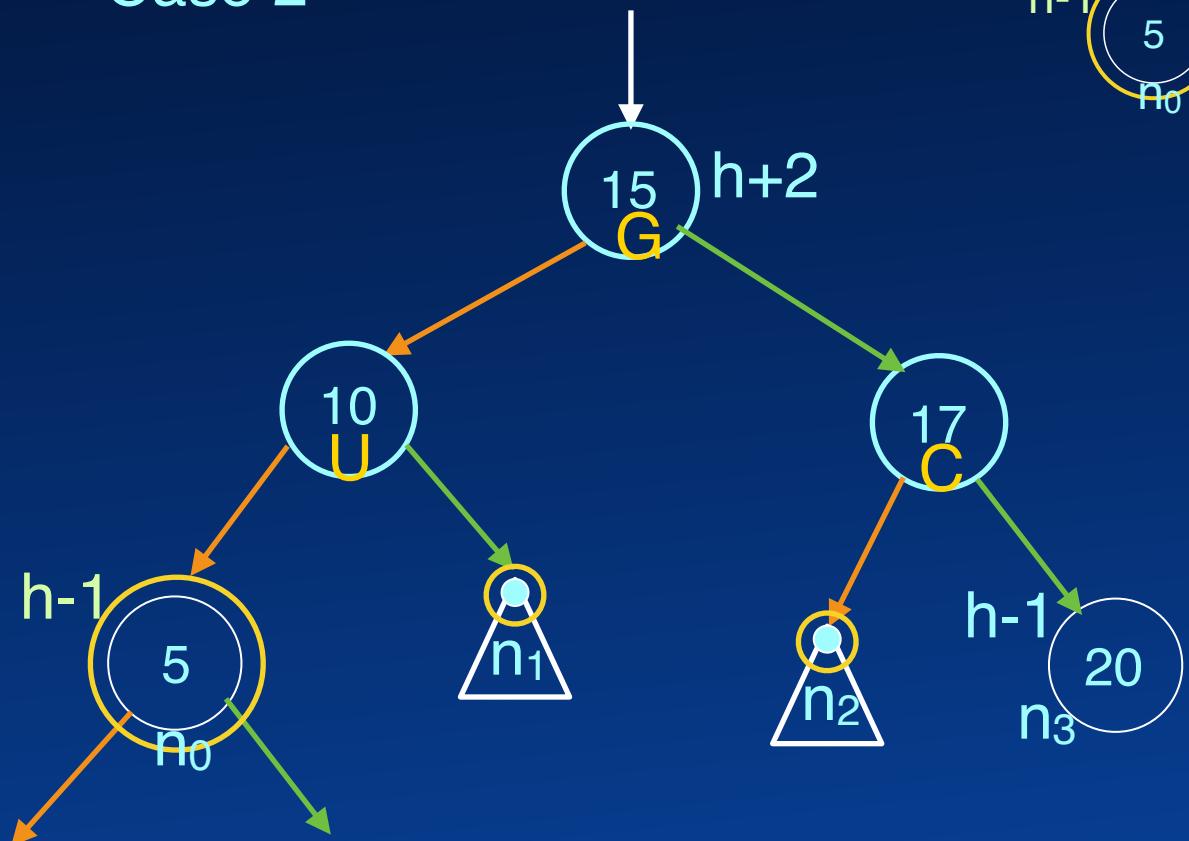
Case 2





Rotation

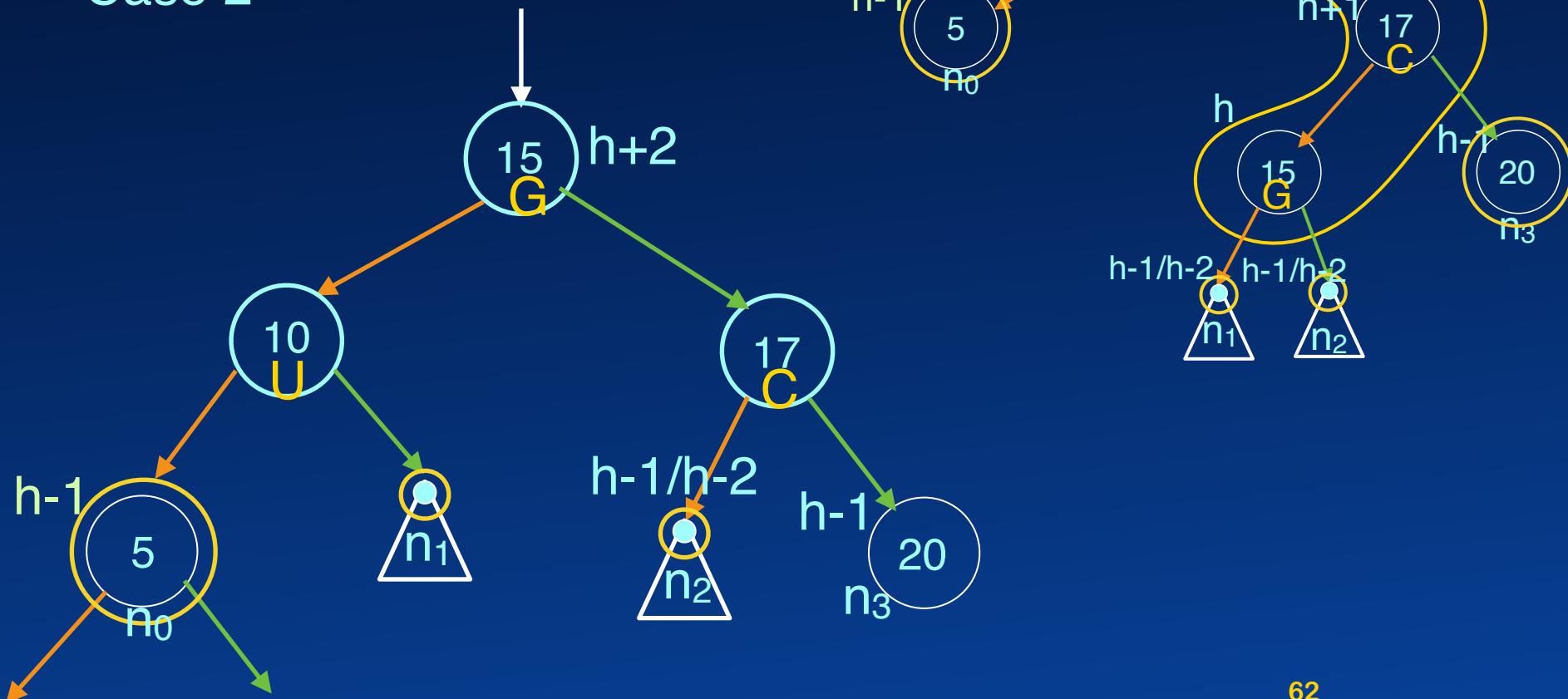
Imbalance on Deletion
Case 2





Rotation

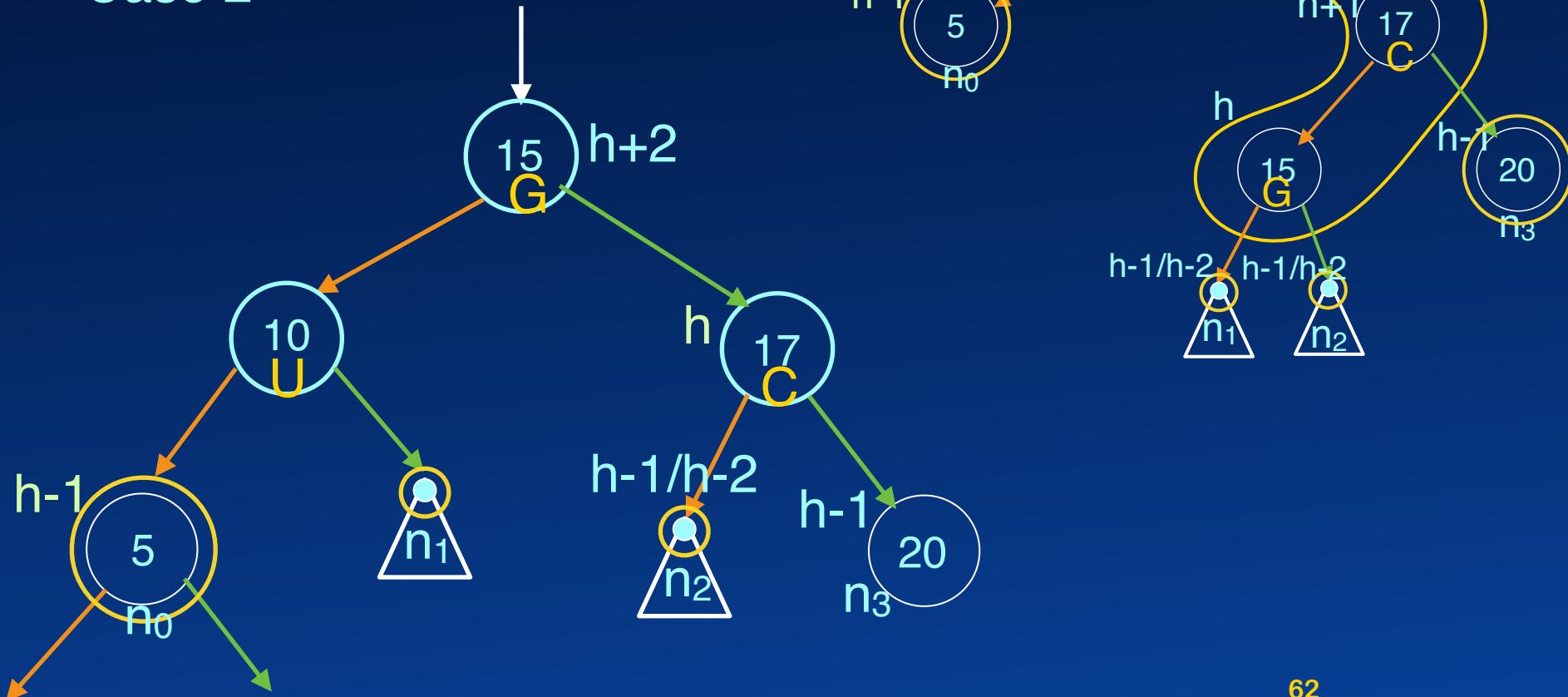
Imbalance on Deletion
Case 2





Rotation

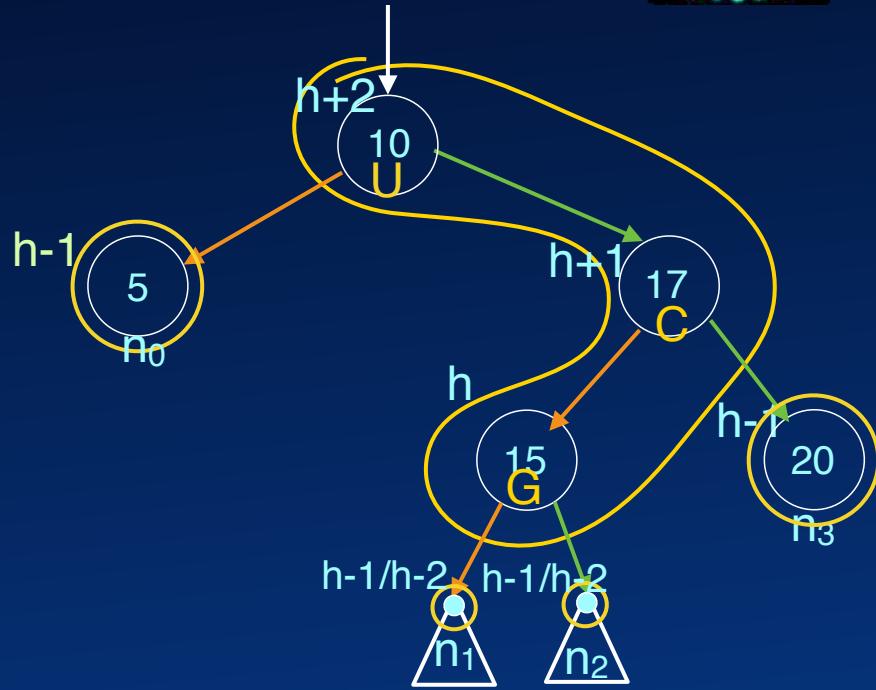
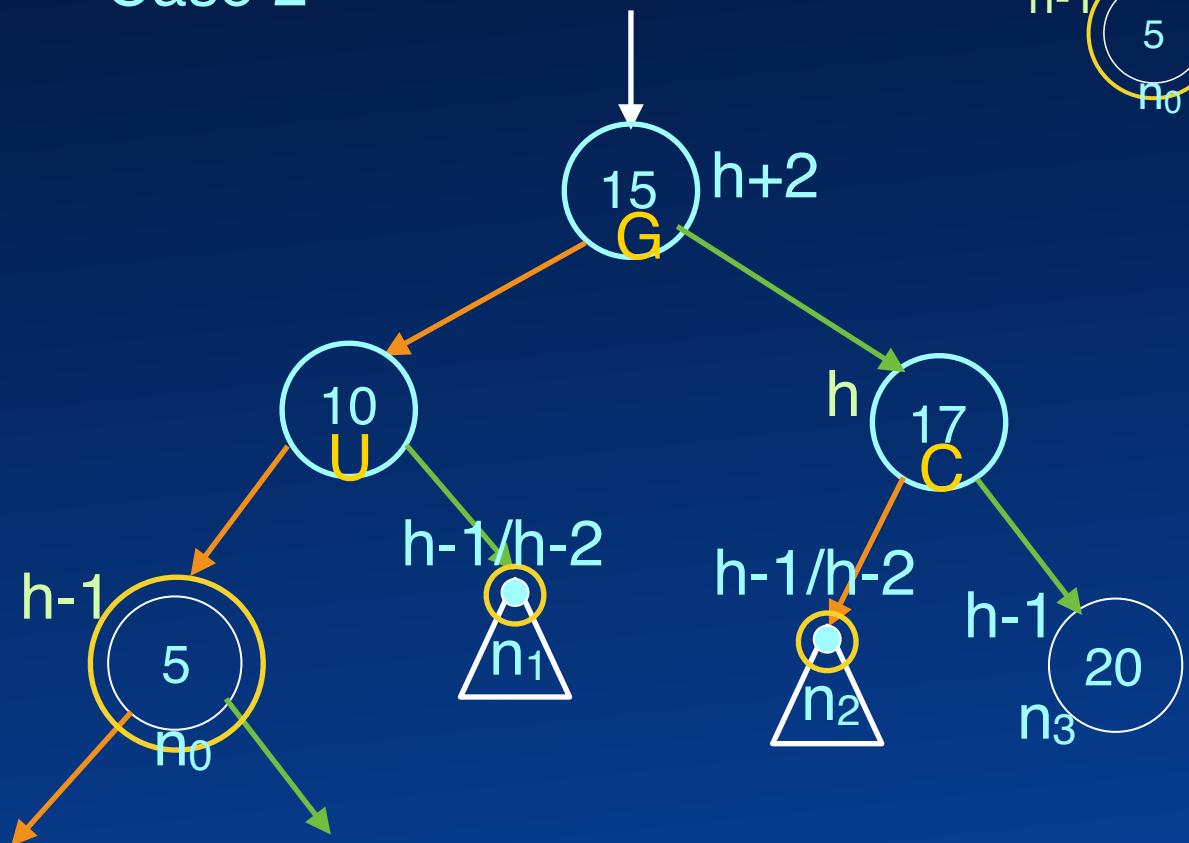
Imbalance on Deletion
Case 2





Rotation

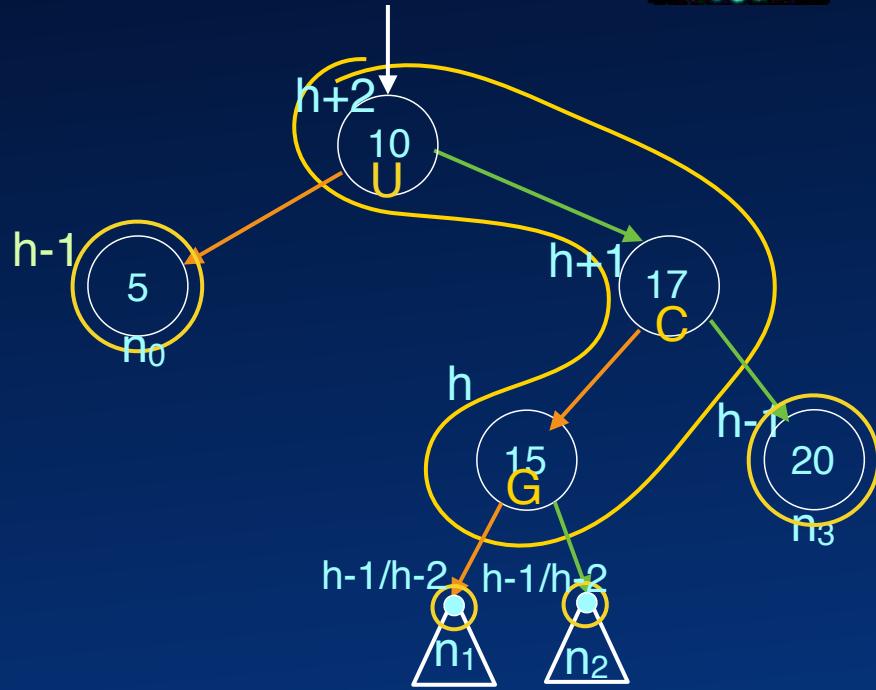
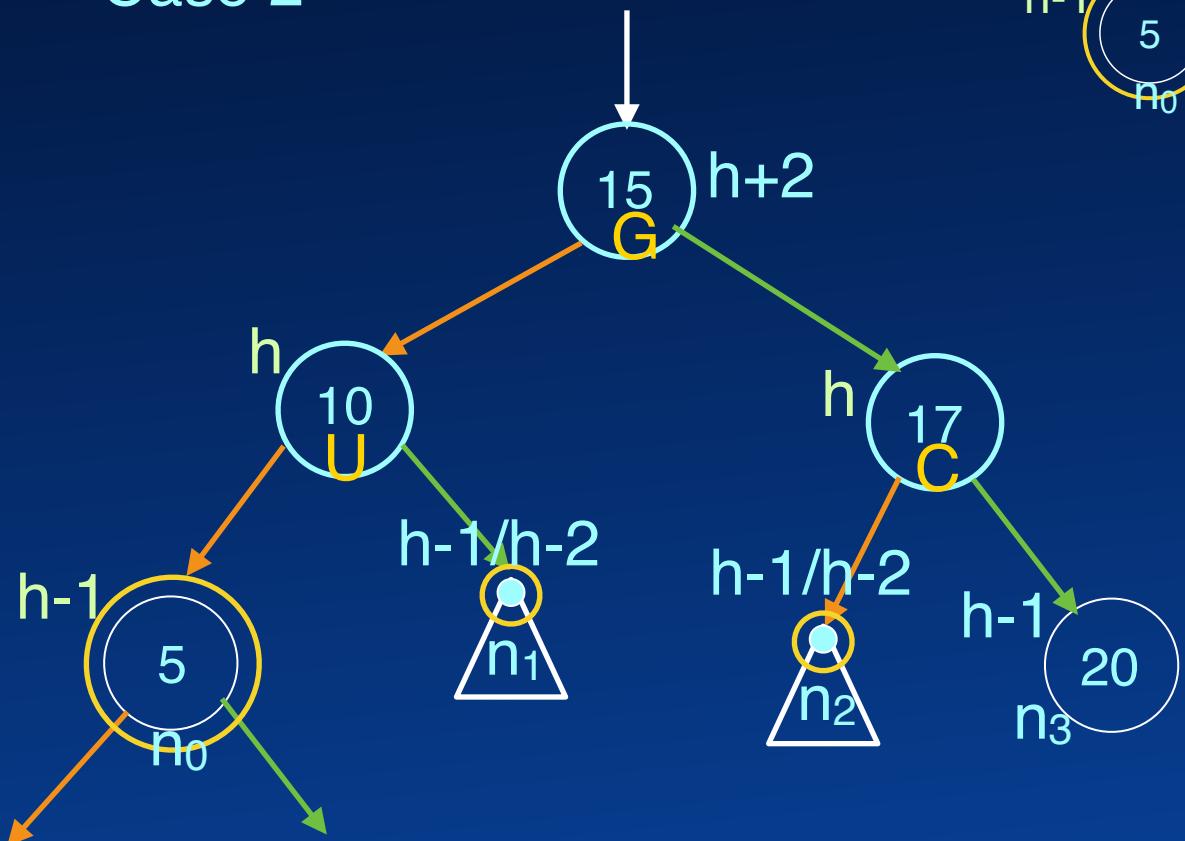
Imbalance on Deletion
Case 2





Rotation

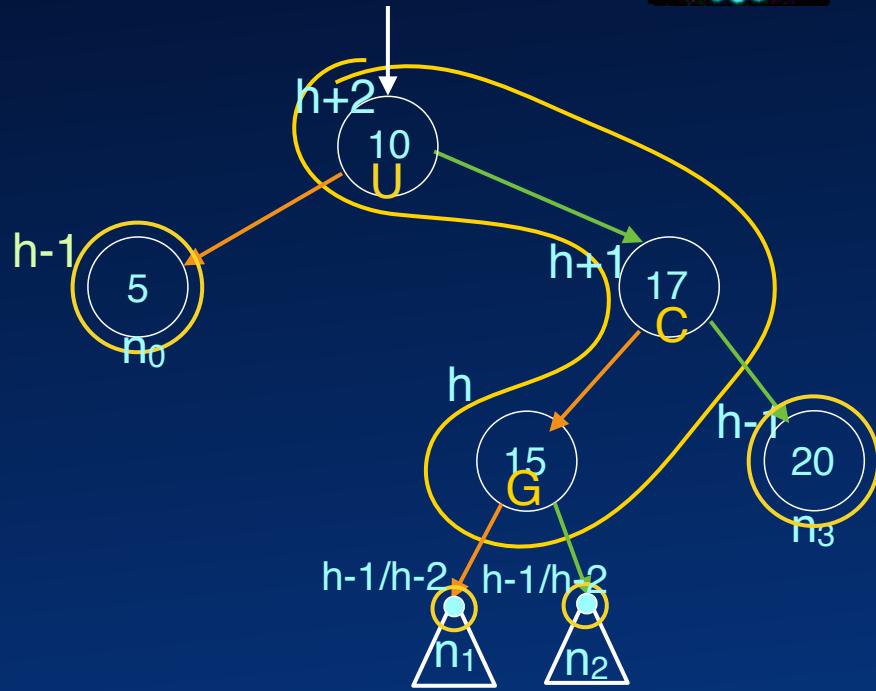
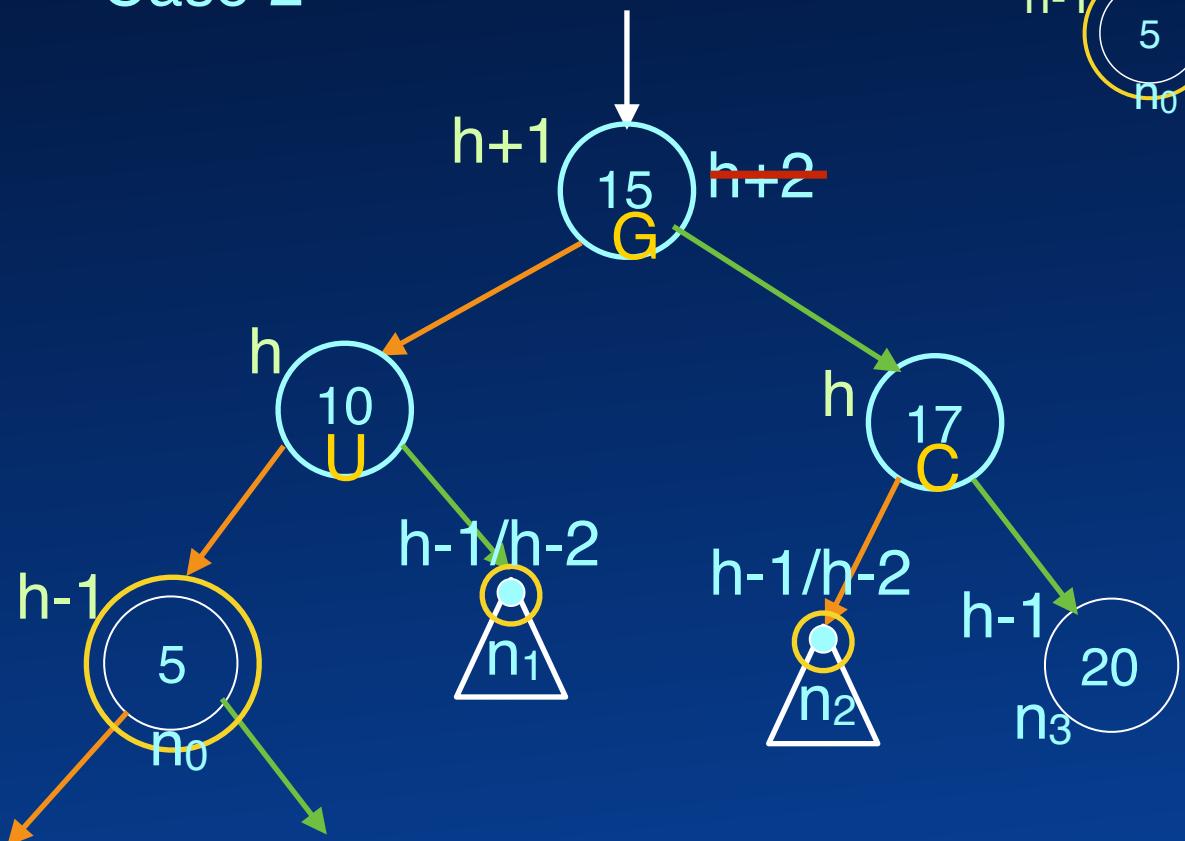
Imbalance on Deletion
Case 2





Rotation

Imbalance on Deletion
Case 2





True or False?

Format: t,t,f

Mail: col106quiz@cse.iitd.ac.in

- **The height of an AVL tree with n nodes is O(log n)**
- **The difference in the heights of two children of an AVL tree node is at most 1**
- **The maximum number of nodes requiring restructuring when deleting a node in AVL tree with n nodes is log n**