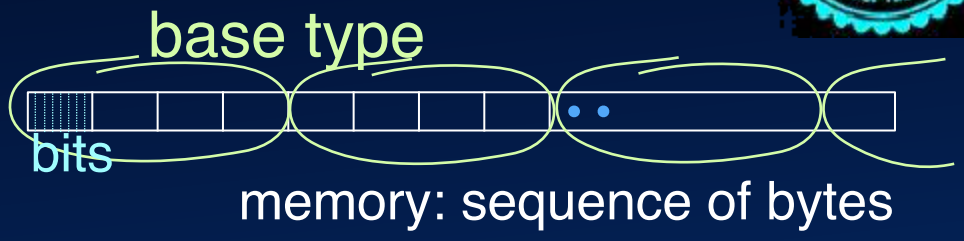
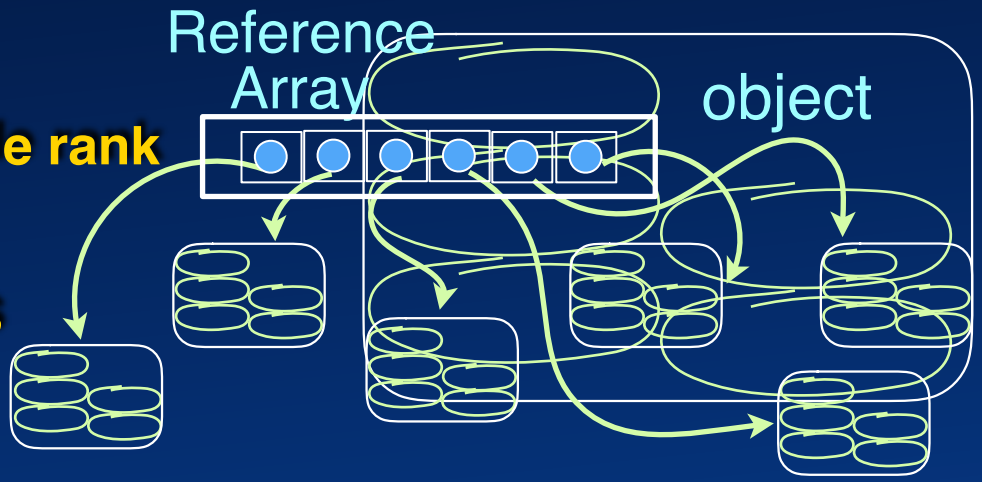
Data Structures & Algorithms

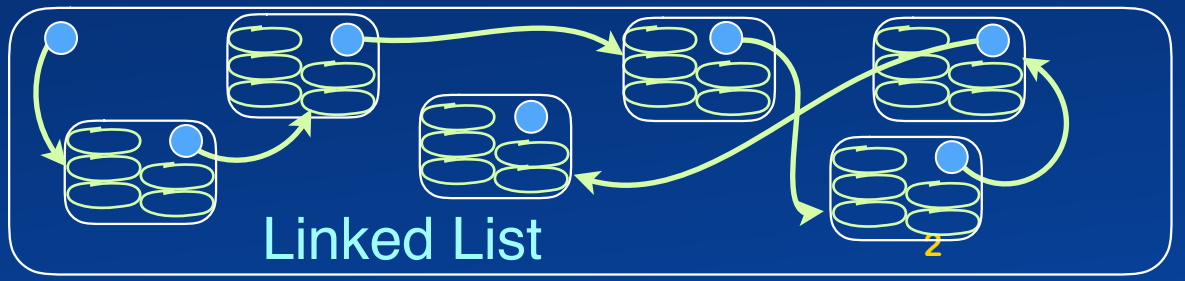
Subodh Kumar
(subodh@iitd.ac.in, Bharti 422)
Dept of Computer Sc. & Engg.

Basic Data Structures

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

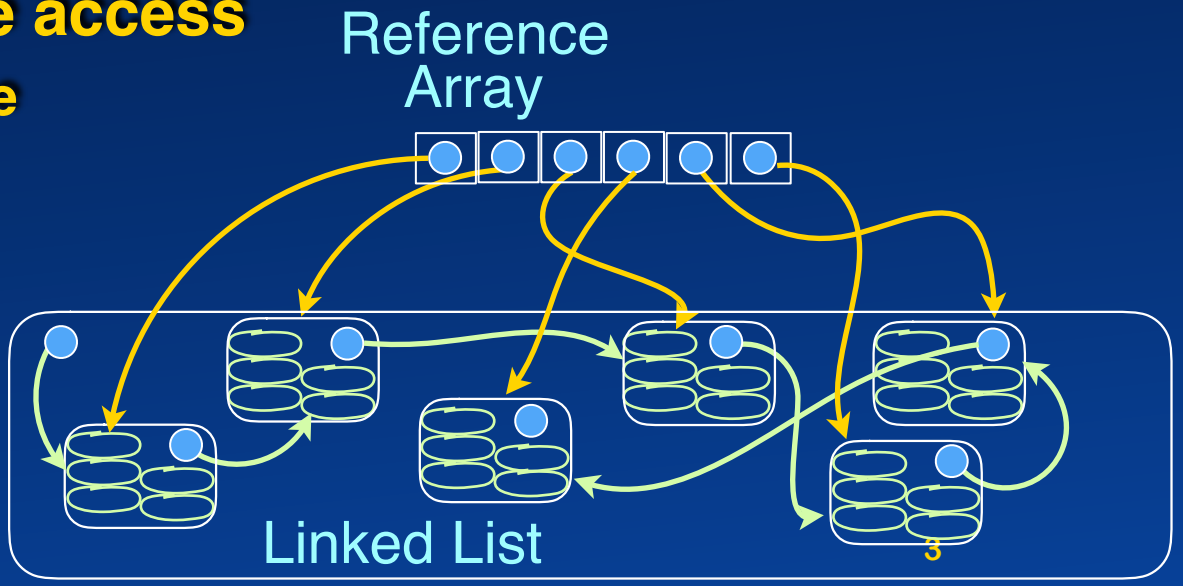






Basic Data Structures

- 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

Restrict usage

col106quiz@cse.iitd.ac.in
Format:t,t,t,f

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

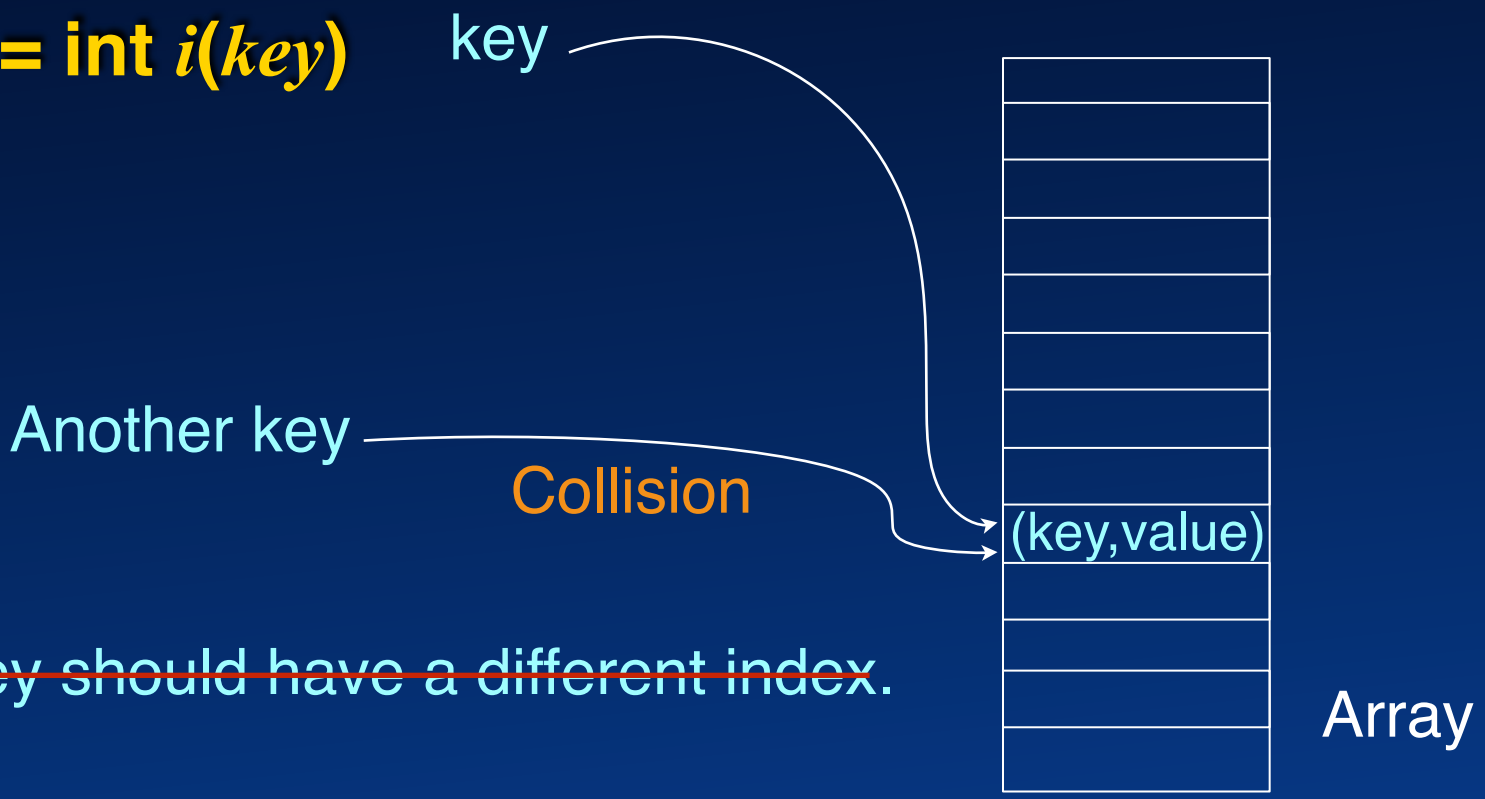
```
public interface Dictionary<K, V> {
    public V get(K key);
    public void put(K key, V value);
    public V remove(K key);
    public iterator<V> allvalues();
    public iterator<K> allkeys();
}
```

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

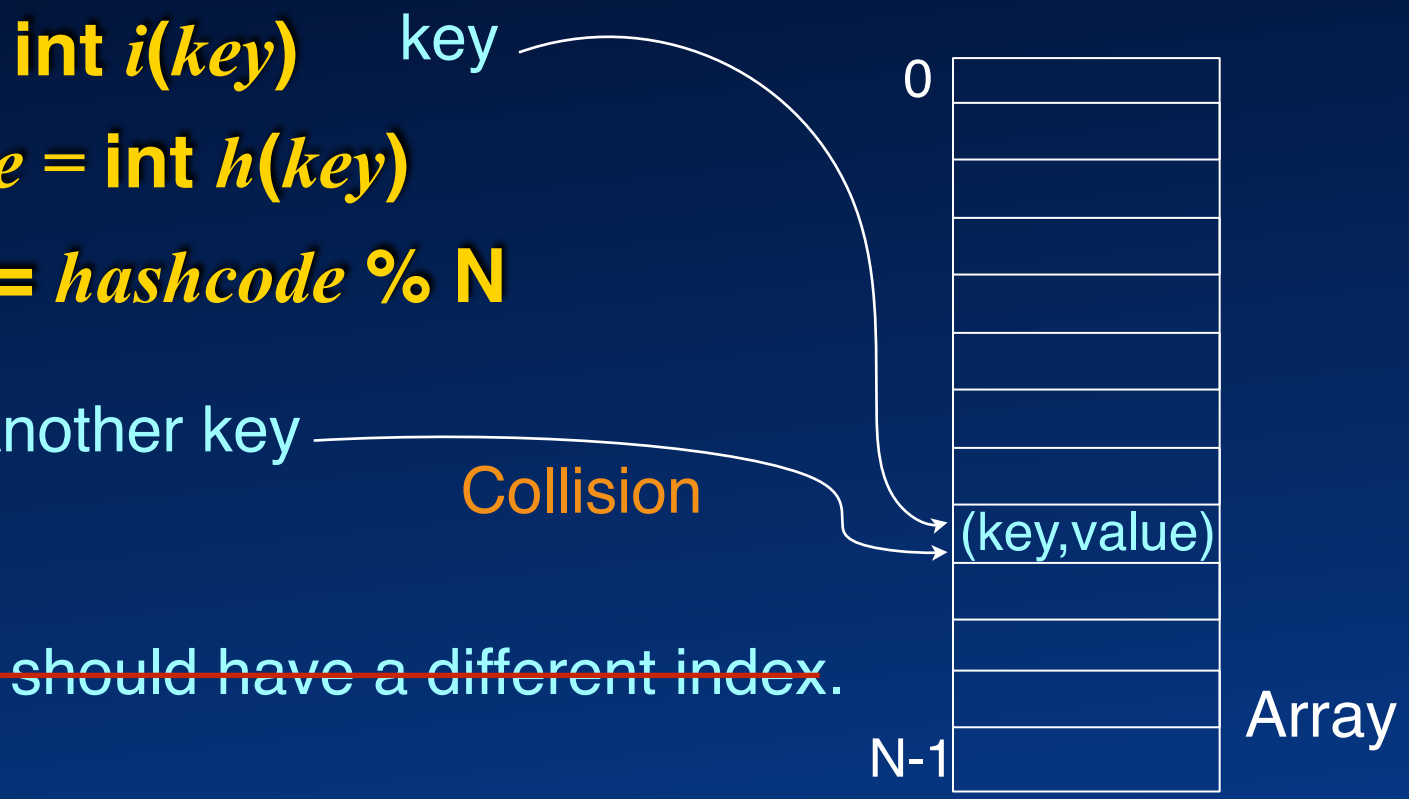
- **index = int *i*(*key*)**



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

Hashing

- index = int *i*(*key*)
- *hashcode* = int *h*(*key*)
- index *i* = *hashcode* % N



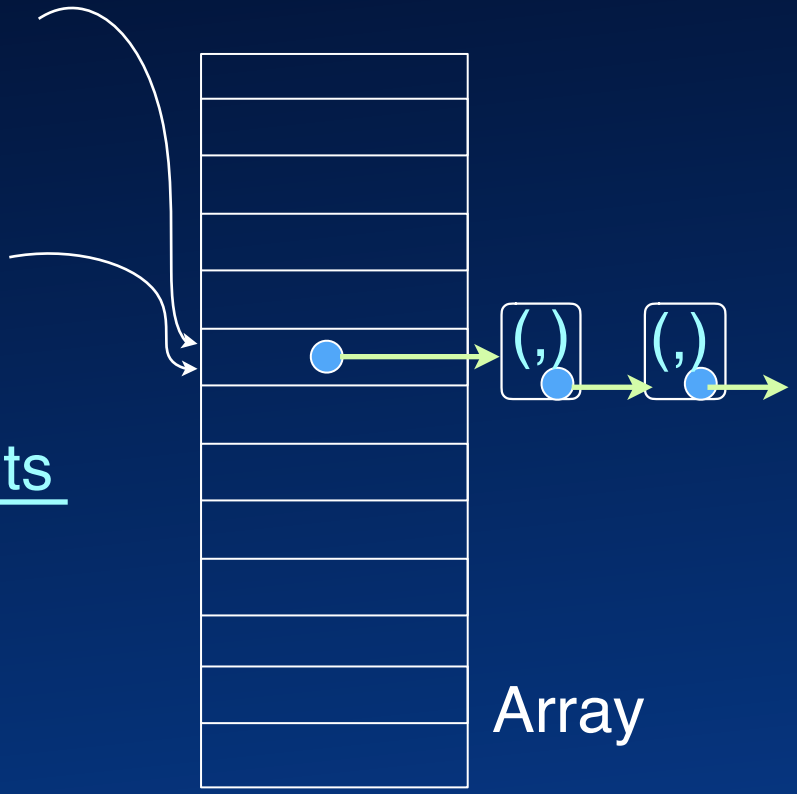
~~Every key should have a different index.~~

~~Every key should have a different hashcode~~

~~and then index also?~~

Collision Resolution

- **Separate chaining**



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

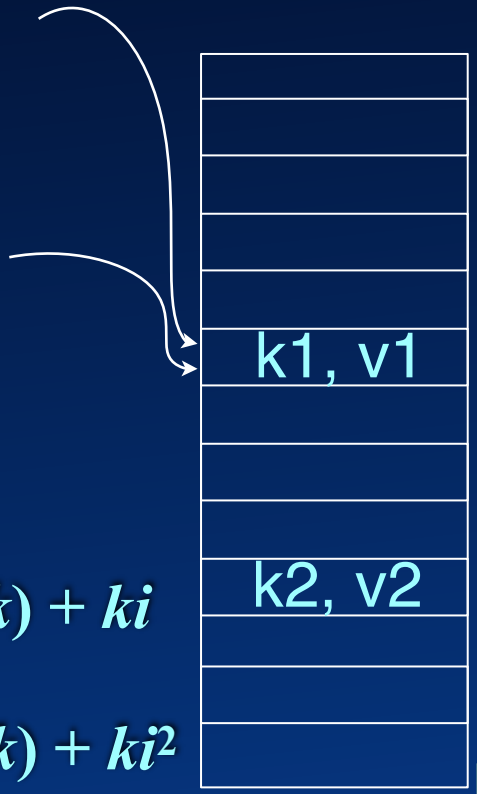
Uniformly random keys ⇒

Probability of collision ≤ min(λ, 1)

= average chain length

Collision Resolution

- Separate chaining



h_1

h_2

h_3

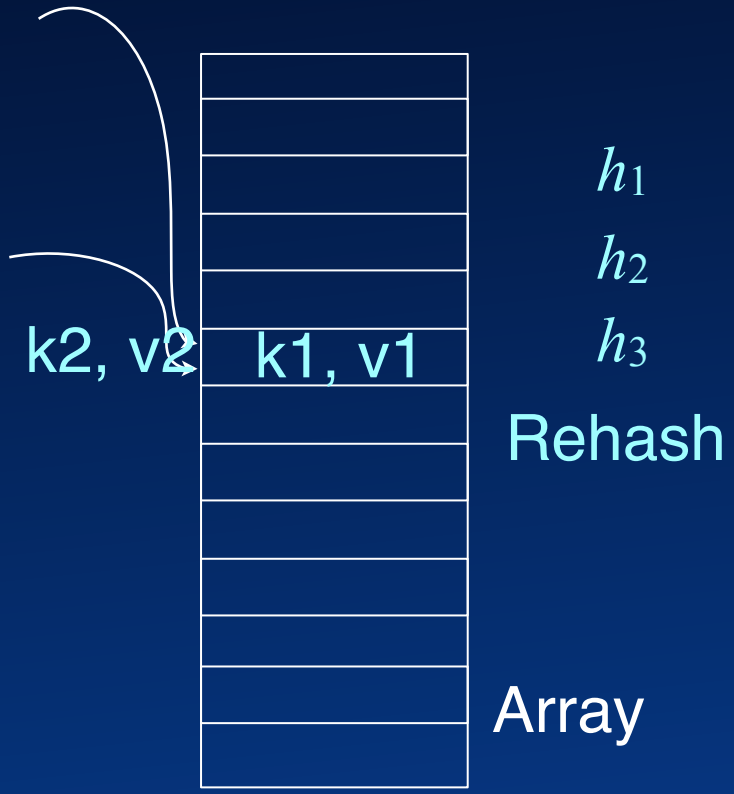
- Open Addressing
  - Linear probing
    $h_i(k) = h(k) + ki$
  - Quadratic probing
    $h_i(k) = h(k) + ki^2$
  - Double Hashing
    $h_i(k) = h(k) + h'(k)i$
  - (Pseudo) Random probing
    $h_i(k) = h(k) + random(h(k), i)$

Rehash
until space located

Array size: N

Index is "%N"
Choose Prime N

Cuckoo Hashing



col106quiz@cse.iitd.ac.in
Format:t,t,t,f

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 $O(1)$

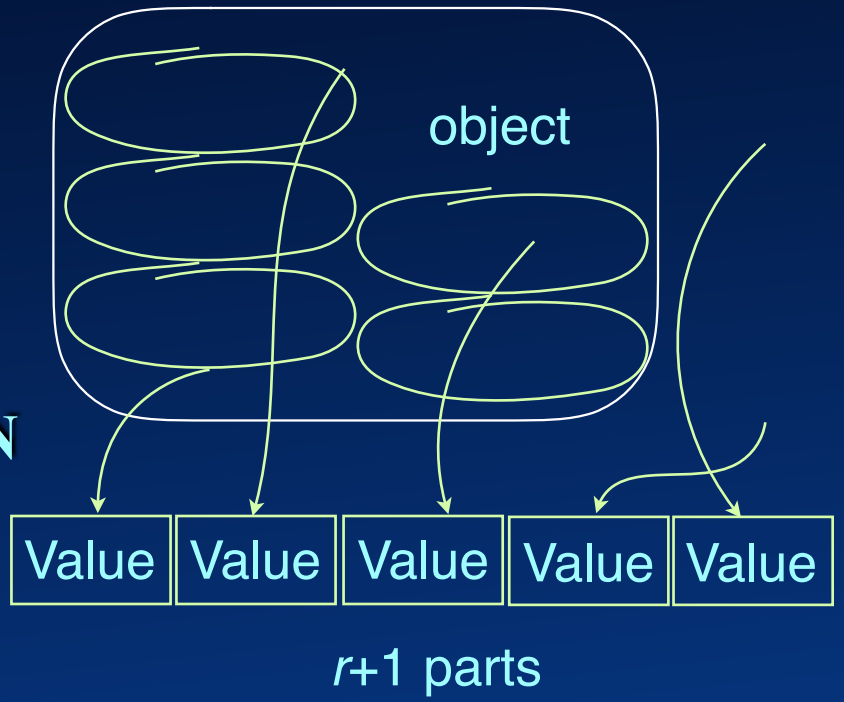
Hash of Integer

- h(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

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



- Cyclic shift and add

$$h = 0$$
$$h \ += \text{Cyclic Shift}(h) + k_i$$

- Universal hashing

Universal Hash Function

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

  $k_0, k_1, .. k_r$ s.t. $\max(k_0) < N$
- Choose all possible hash functions

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

  where $a_0, a_1, .. a_r$, are each in {0 .. 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

$$\text{Probability } (h(k1) = h(k2)) = 1/N$$

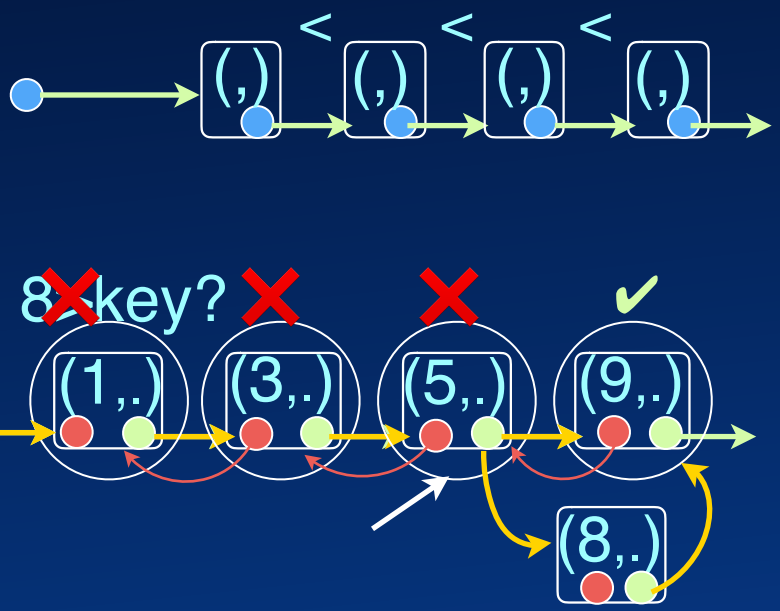
col106quiz@cse.iitd.ac.in
Format:t,t,f

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.

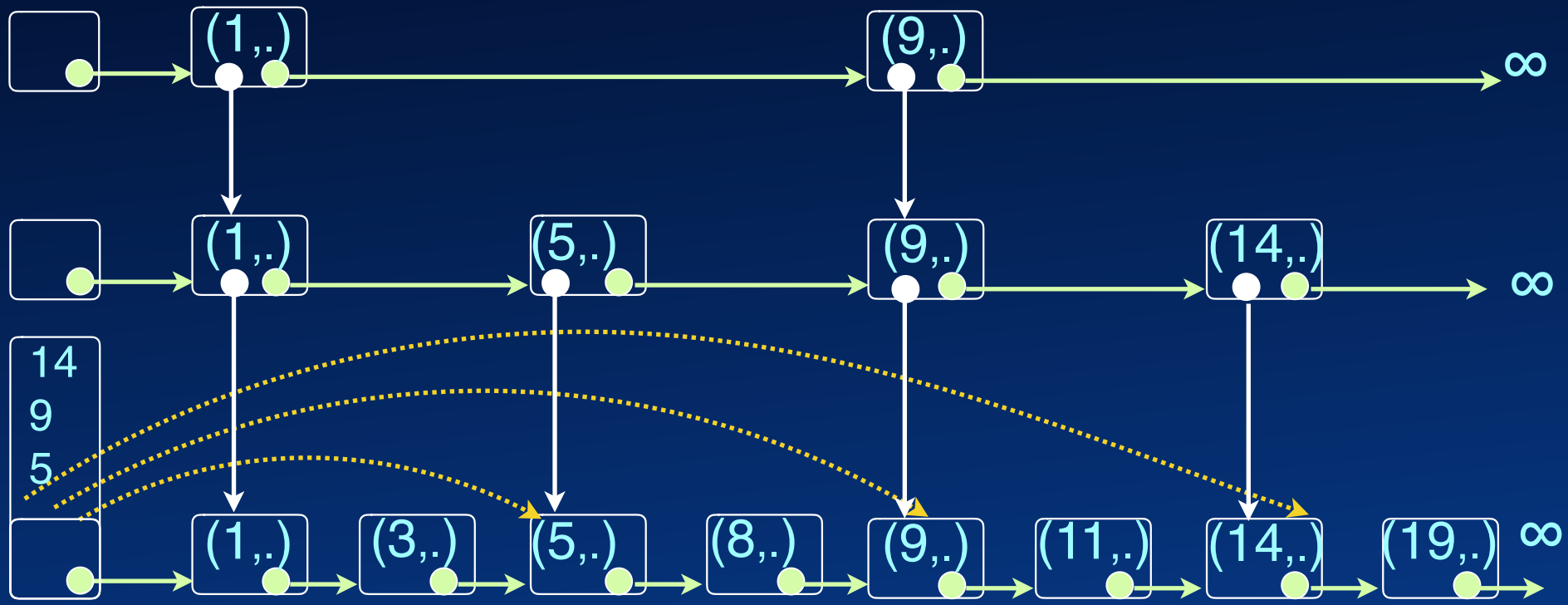
Ordered Keys

- Put it in a sorted list
  - Insertion sort

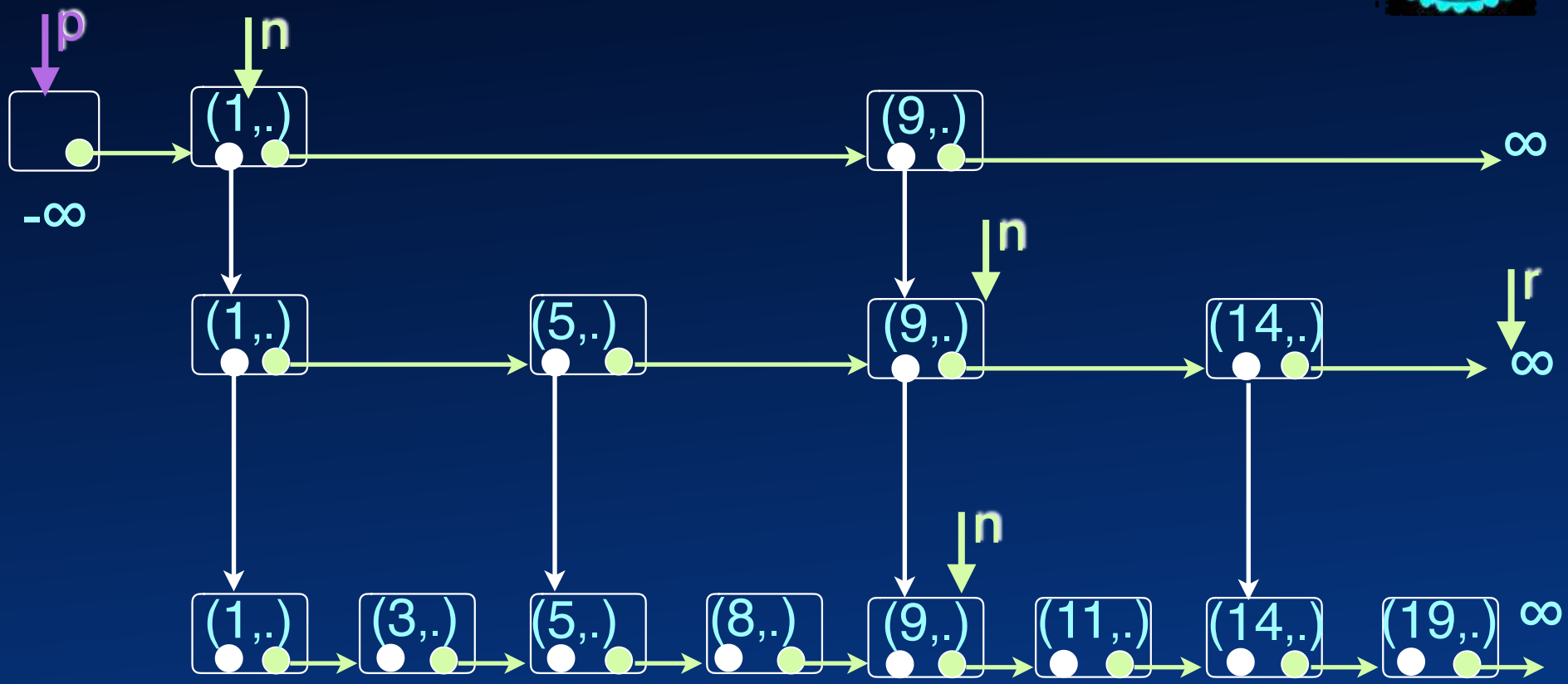



```
node p = sentinel;
node n = sentinel.next;
while ( n.key != null &&
        n.key < k ) {
    p = n; n = n.next;
}
p.insertAfter(p, Pair(k, v))
```

Skip-list

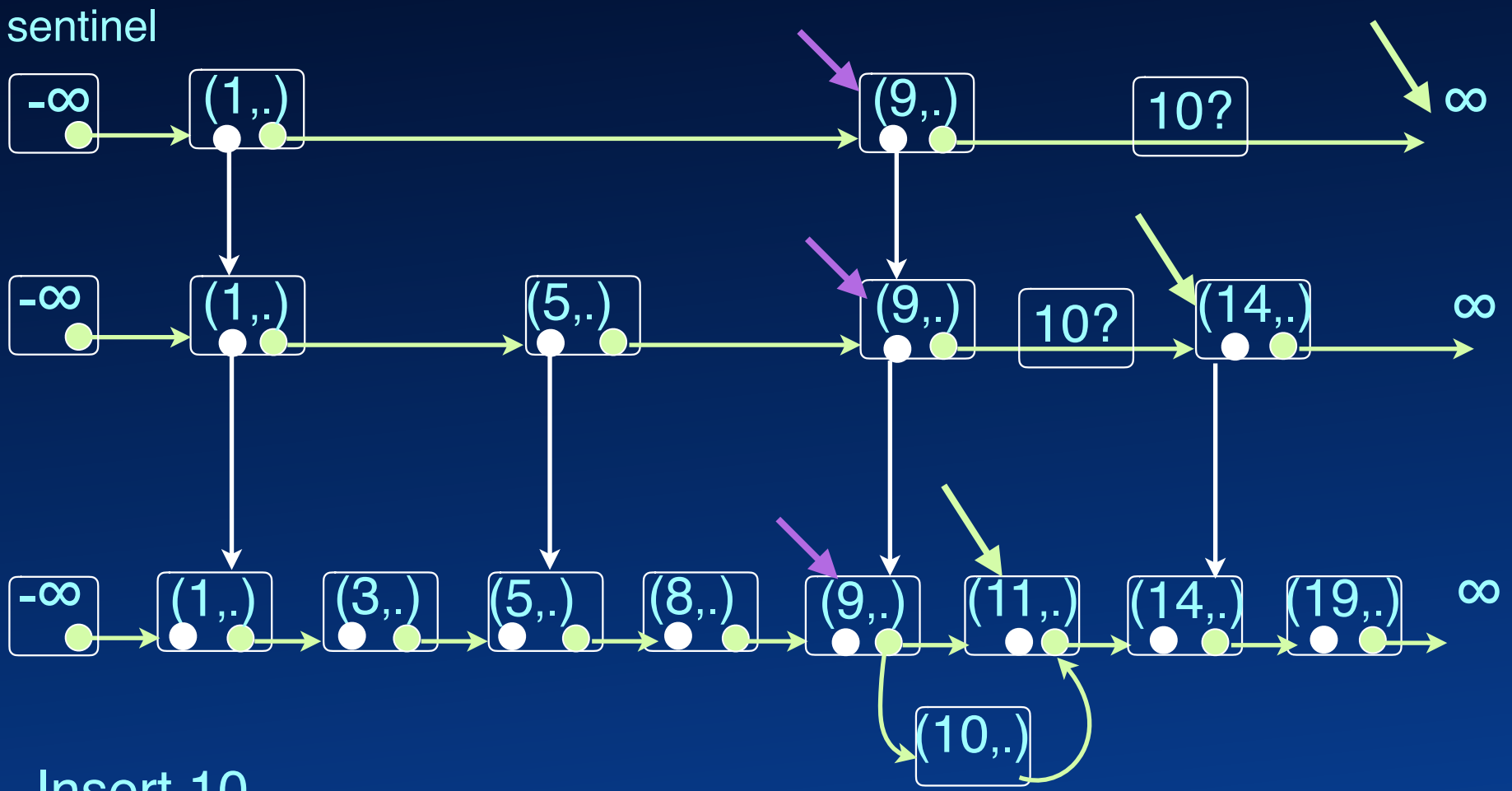


Skip-list



Find 10

Skip-list



Insert 10

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);

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);
    insert(p.below, k, topinsert, level-1)
    add below ref
```

What about the below ref?

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

What if topinsert > height?

Skip-list: Increase height

Insert 10

Skip-list: Analysis

$Prob(\text{level } \boldsymbol{i} \text{ exists}) <= \boldsymbol{n/2^i}$

$\boldsymbol{i}$ successive heads in any of $\boldsymbol{n}$ experiments

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

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

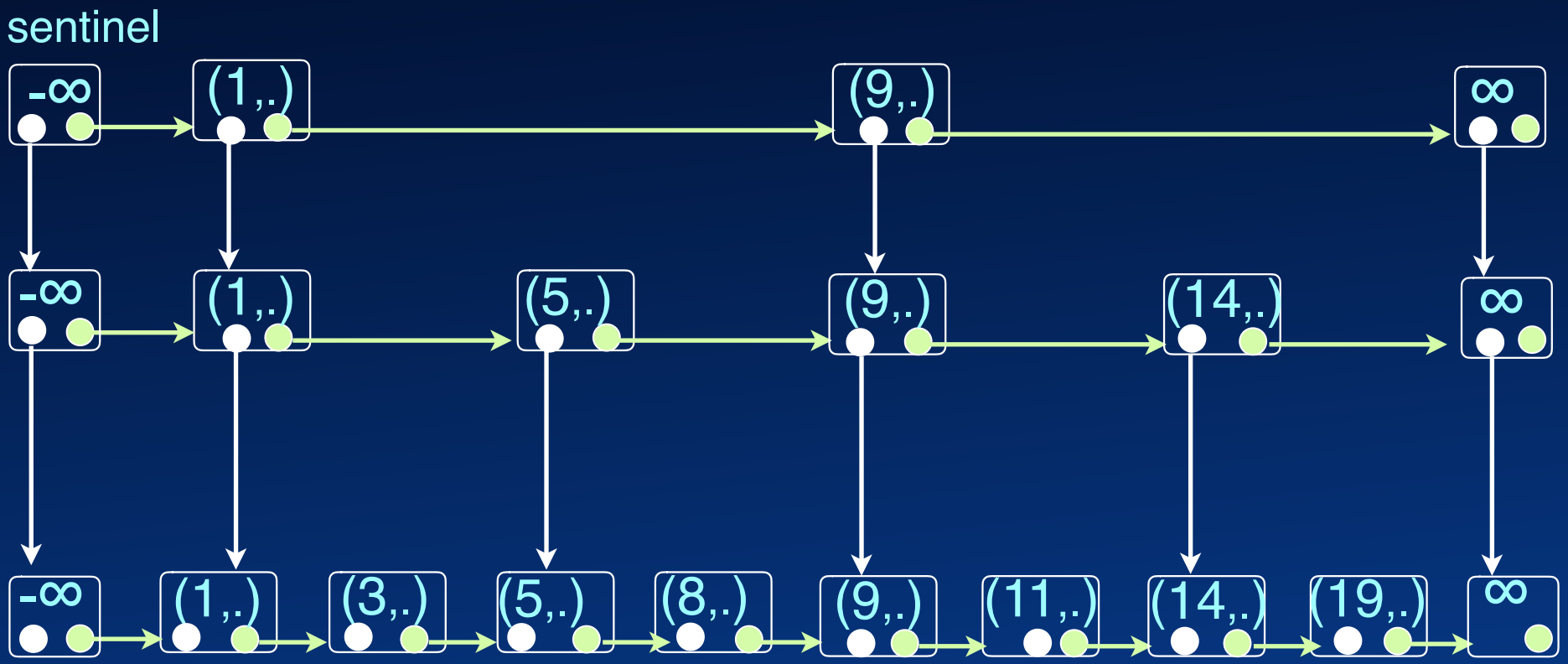
$\Downarrow$

$\text{height} = \boldsymbol{k}\mathbf{\log} \boldsymbol{n}$

Expected no. of nodes visited at any level = 2

= Expected number of tosses before *heads*

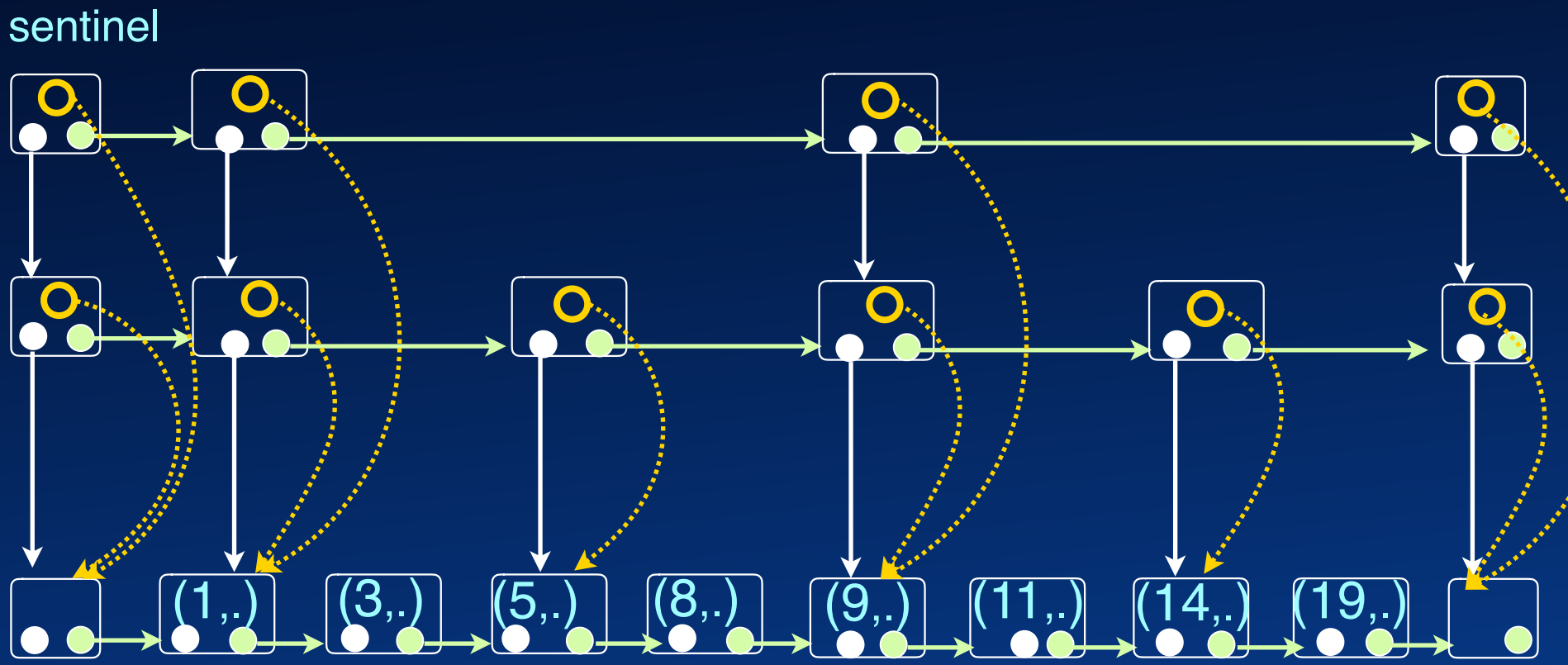
Skip-list



Next to 10

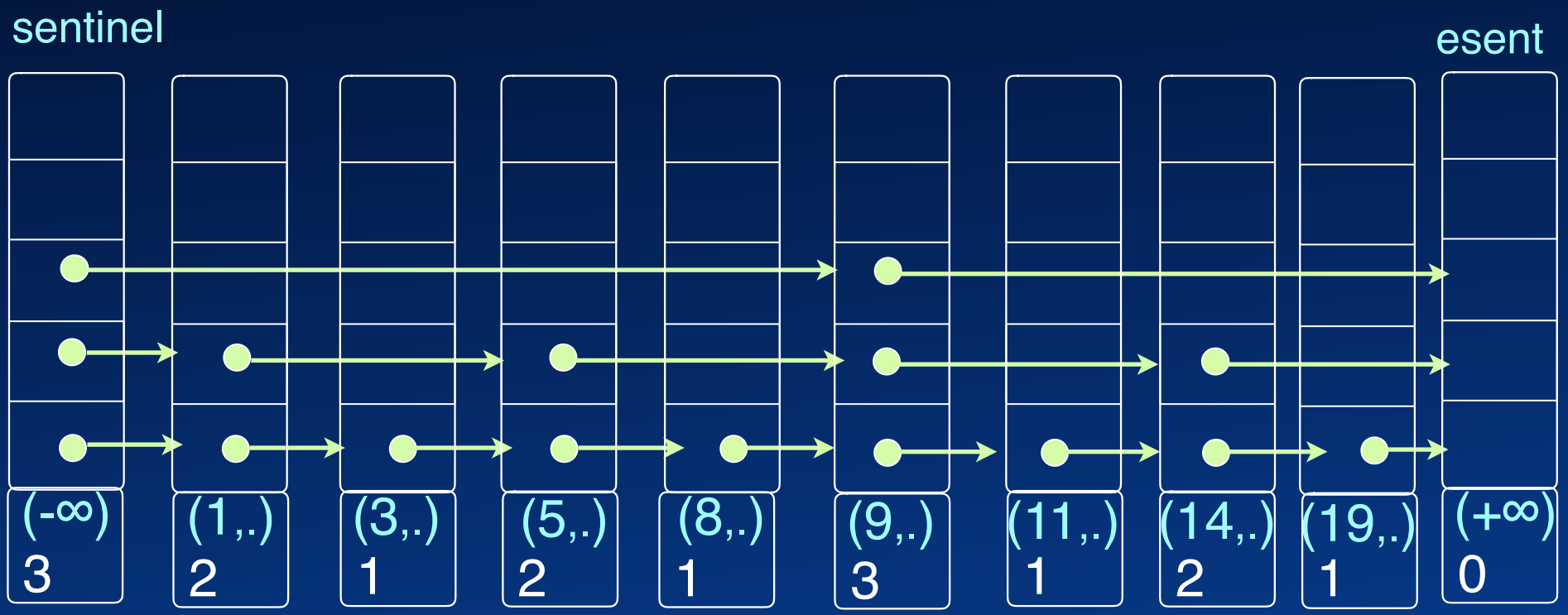
Just before 10?

Skip-list: Save Space



May store only references at the upper levels

Skip-list: Array

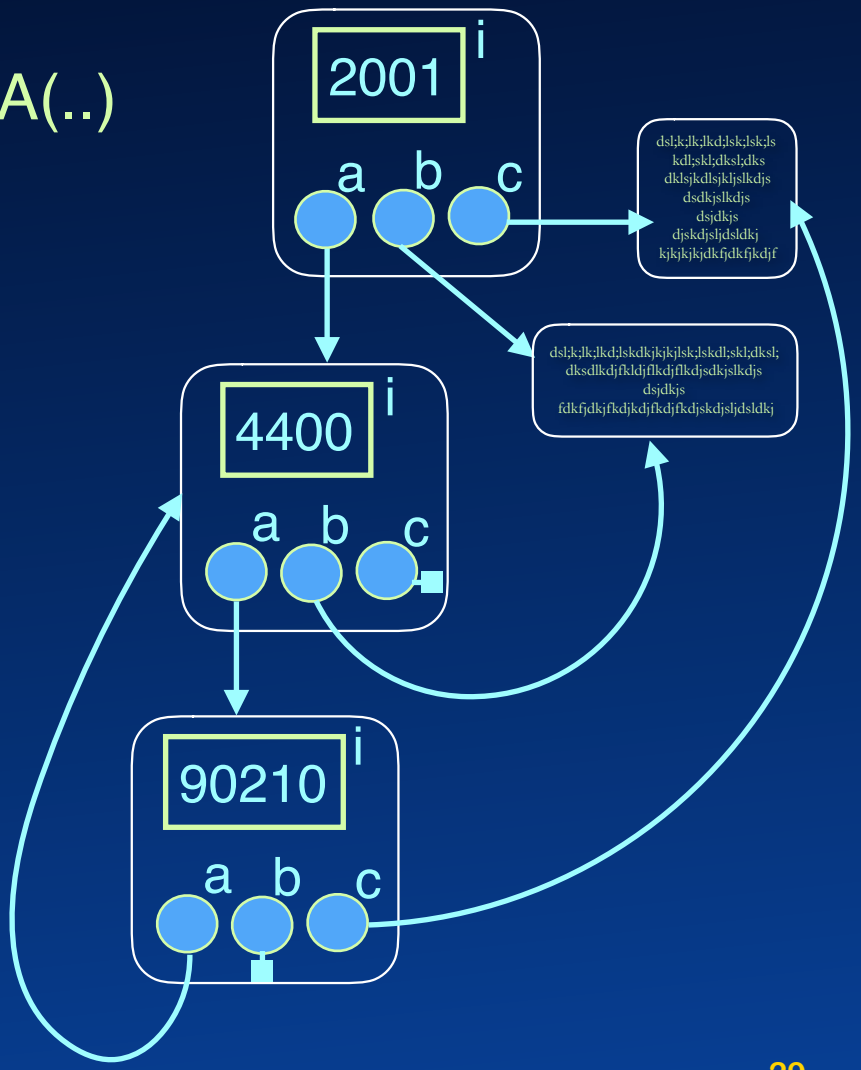


Could even store in arrays

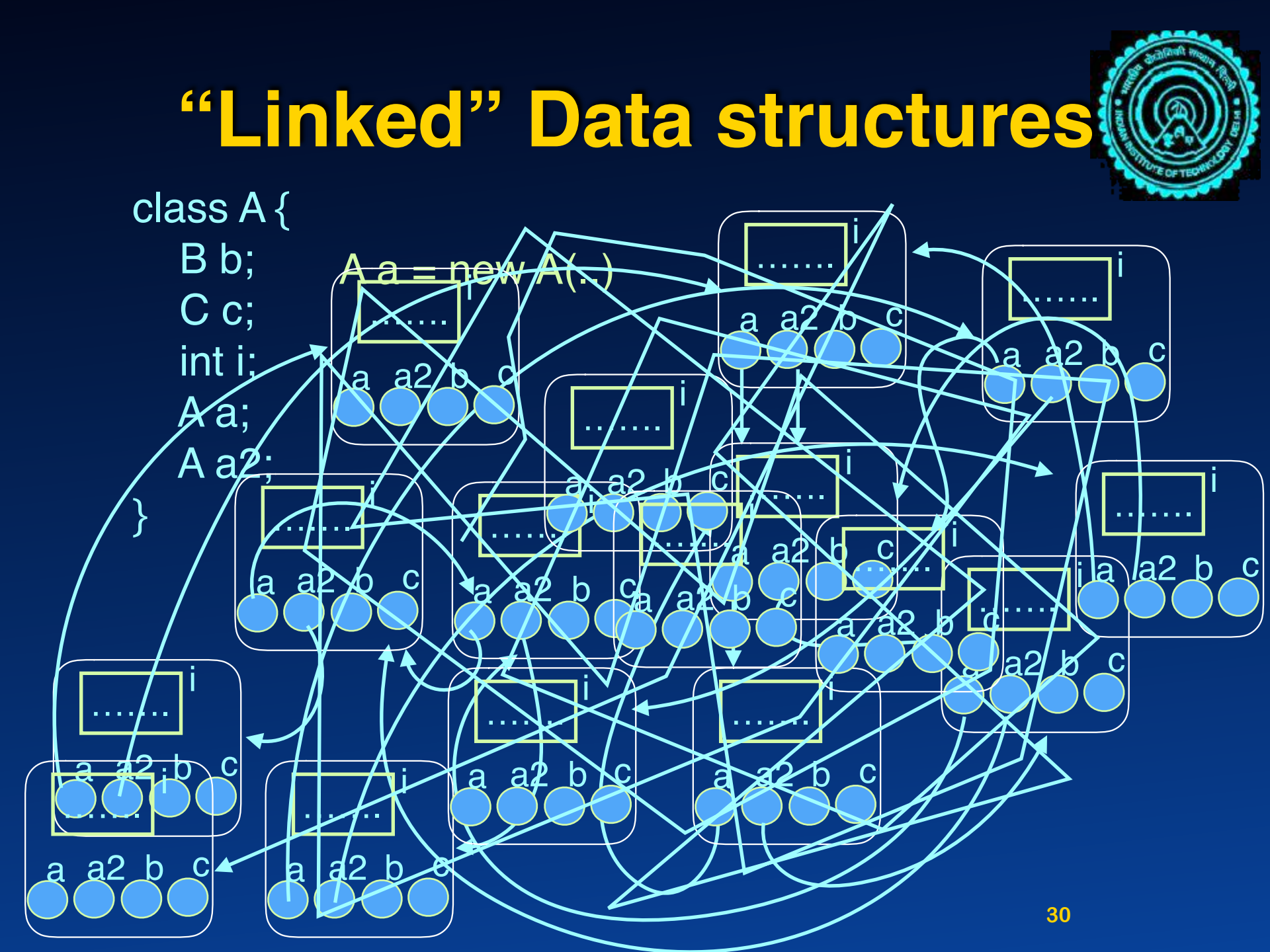
“Linked” Data structures

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

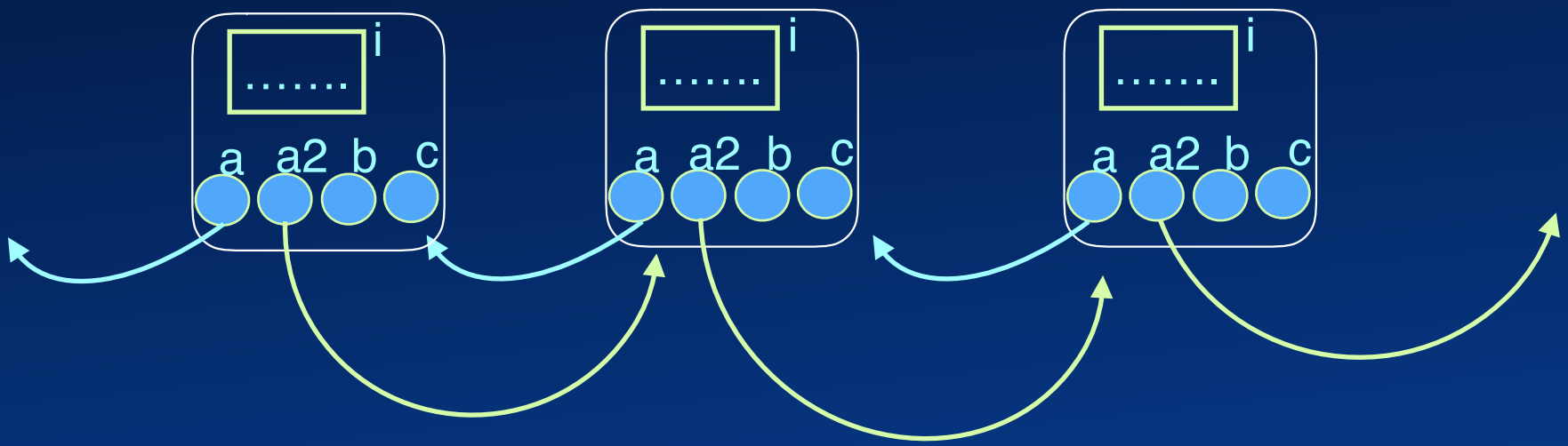
A a = new A(..)



“Linked” Data structures

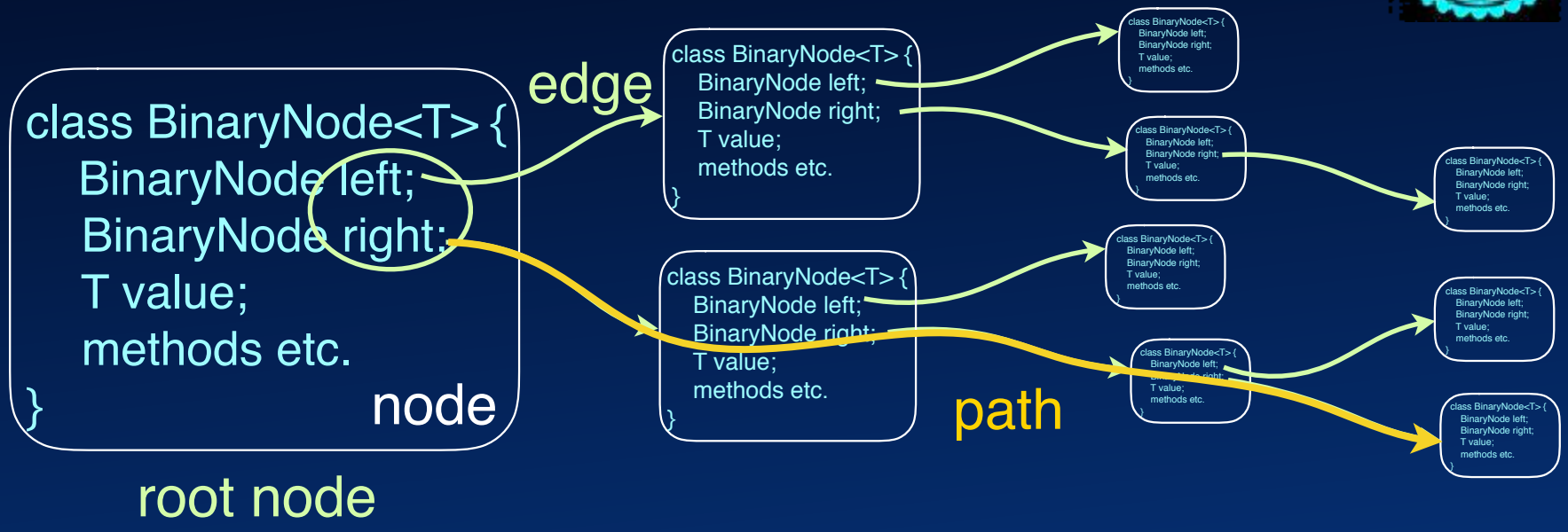


Doubly Linked List



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

Binary Tree



Exactly one other node contains reference to any given node.

One special node has no other node with reference to it.

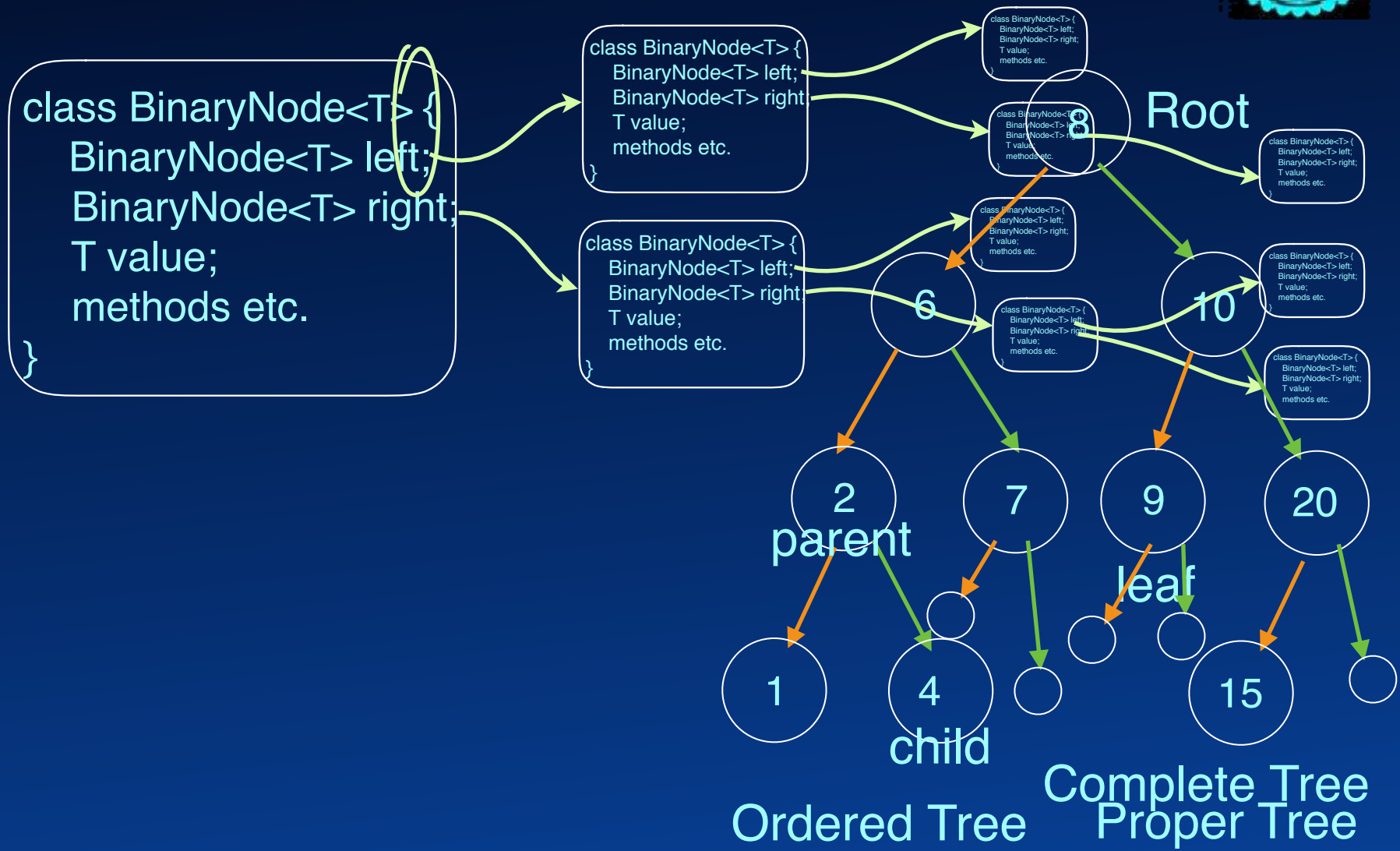
Binary tree has two references per node

Quiz

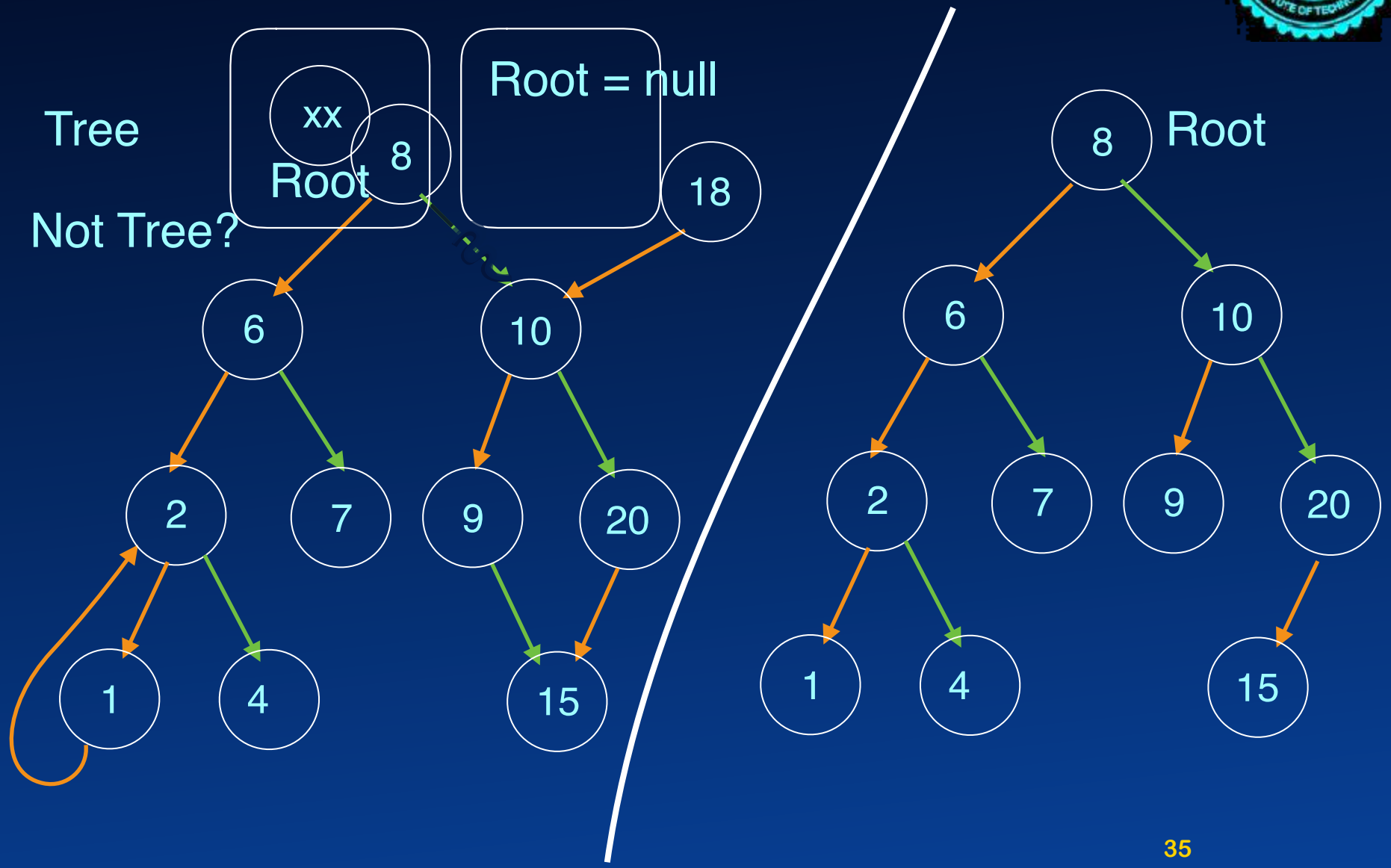
- Keeping skip-list towers in arrays saves space. What else does it save? [1 word]
- How? [maximum 10 words]
- What deficiency does it suffer from? [Maximum 3 words]

col106quiz@cse.iitd.ac.in
Format: time, No separate fetch of below node, fixed count

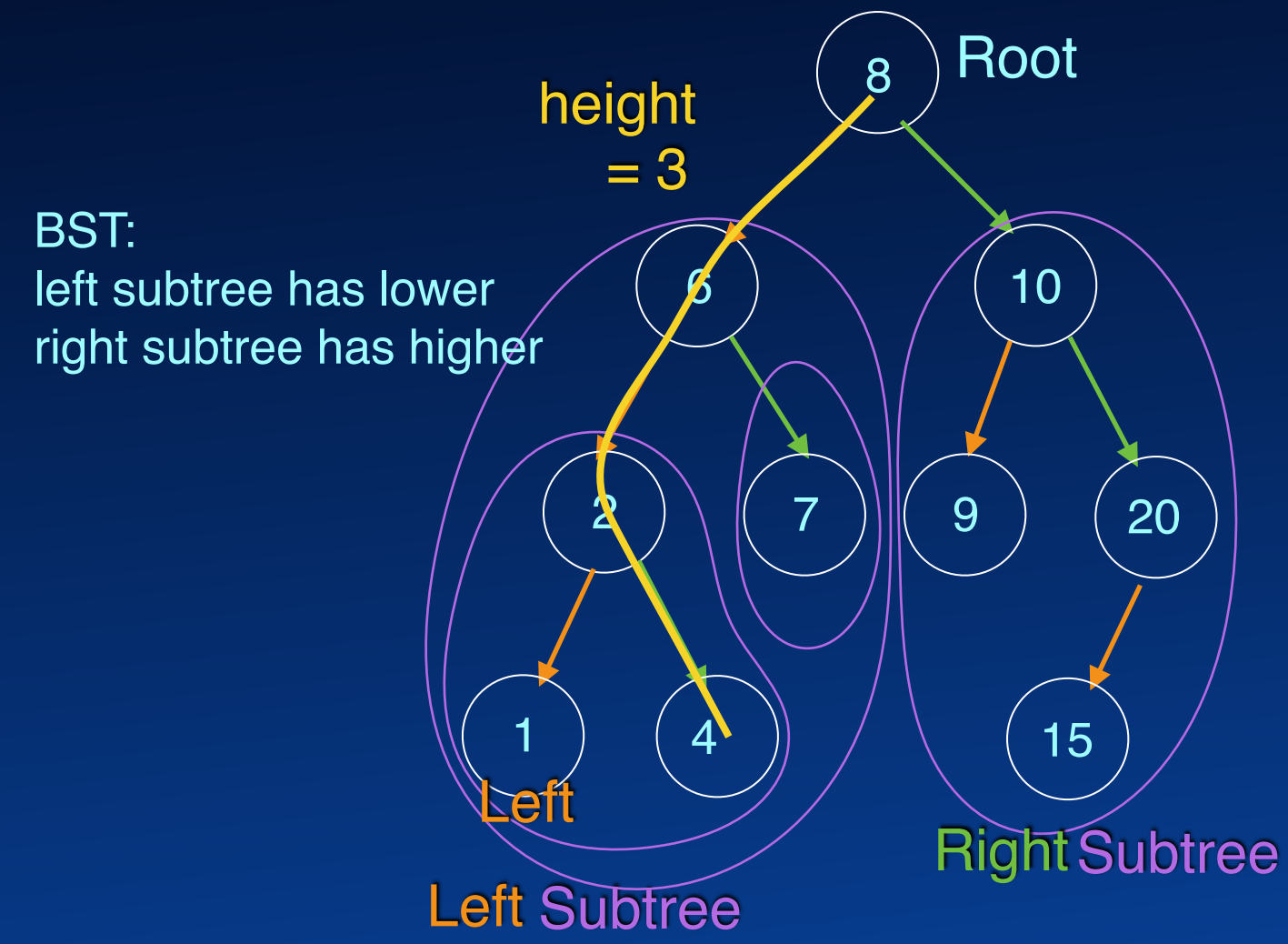
Binary Tree


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.
}
8
Root
6
10
2
parent
7
9
leaf
20
1
4
child
15
Ordered Tree
Complete Tree
Proper Tree

Tree


Tree
Not Tree?
xx
Root
8
Root = null
18
6
10
2
7
9
20
1
4
15
8
Root
6
10
2
7
9
20
1
4
15

Tree


BST:
left subtree has lower
right subtree has higher
height
= 3
8
Root
6
10
2
7
9
20
1
4
15
Left
Left Subtree
Right Subtree

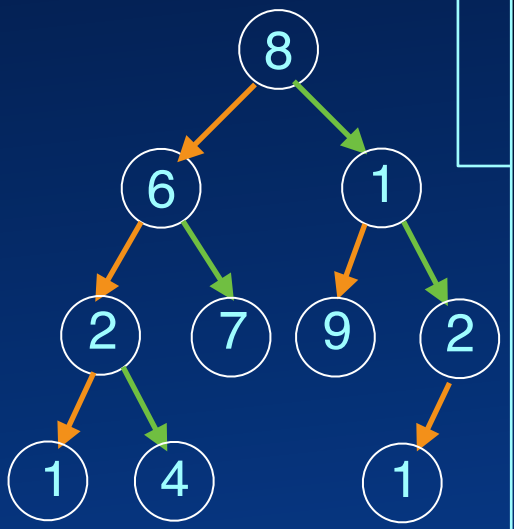
Tree Operations

```
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));
    }
}
```

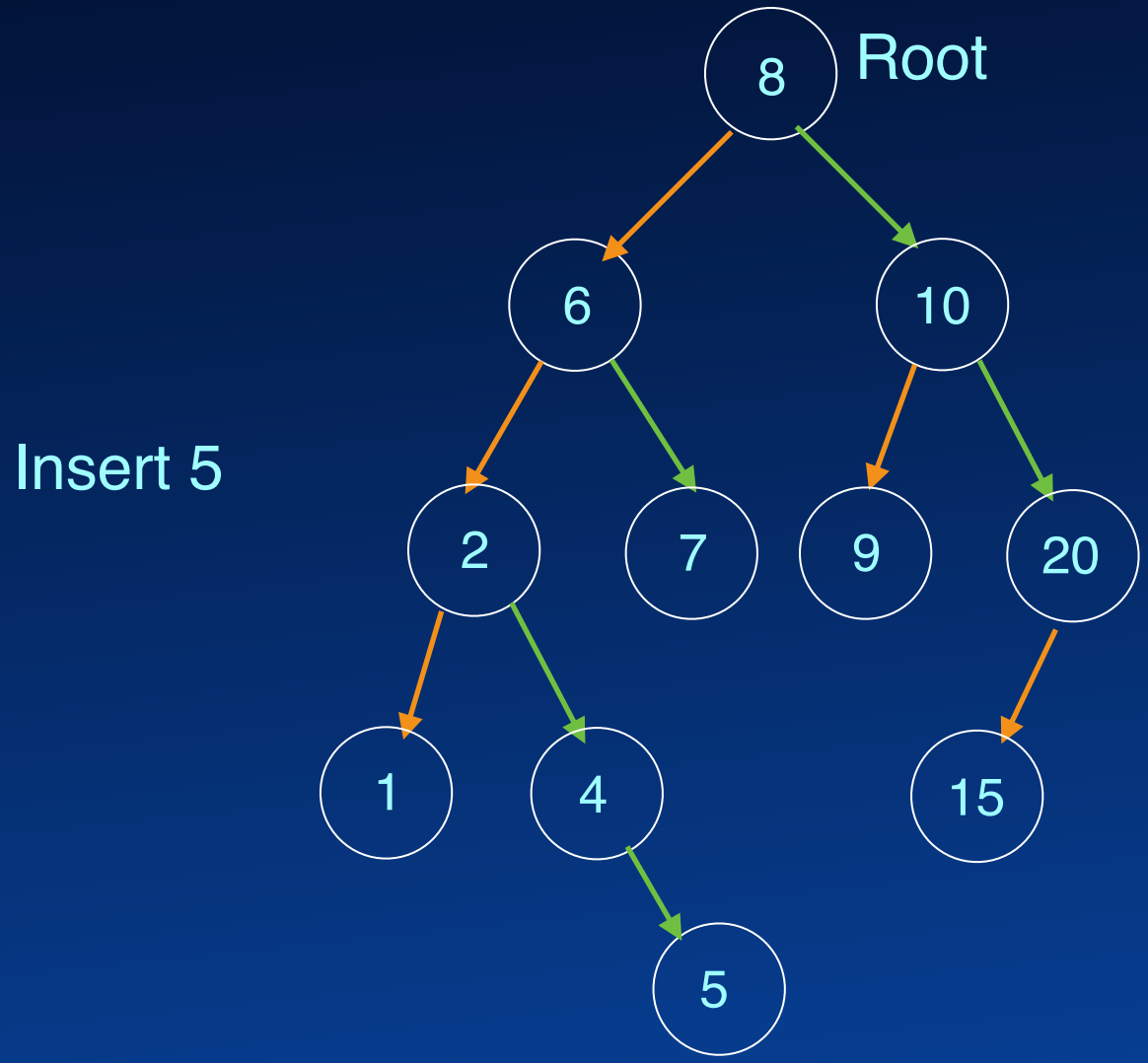
BST

```
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;
```

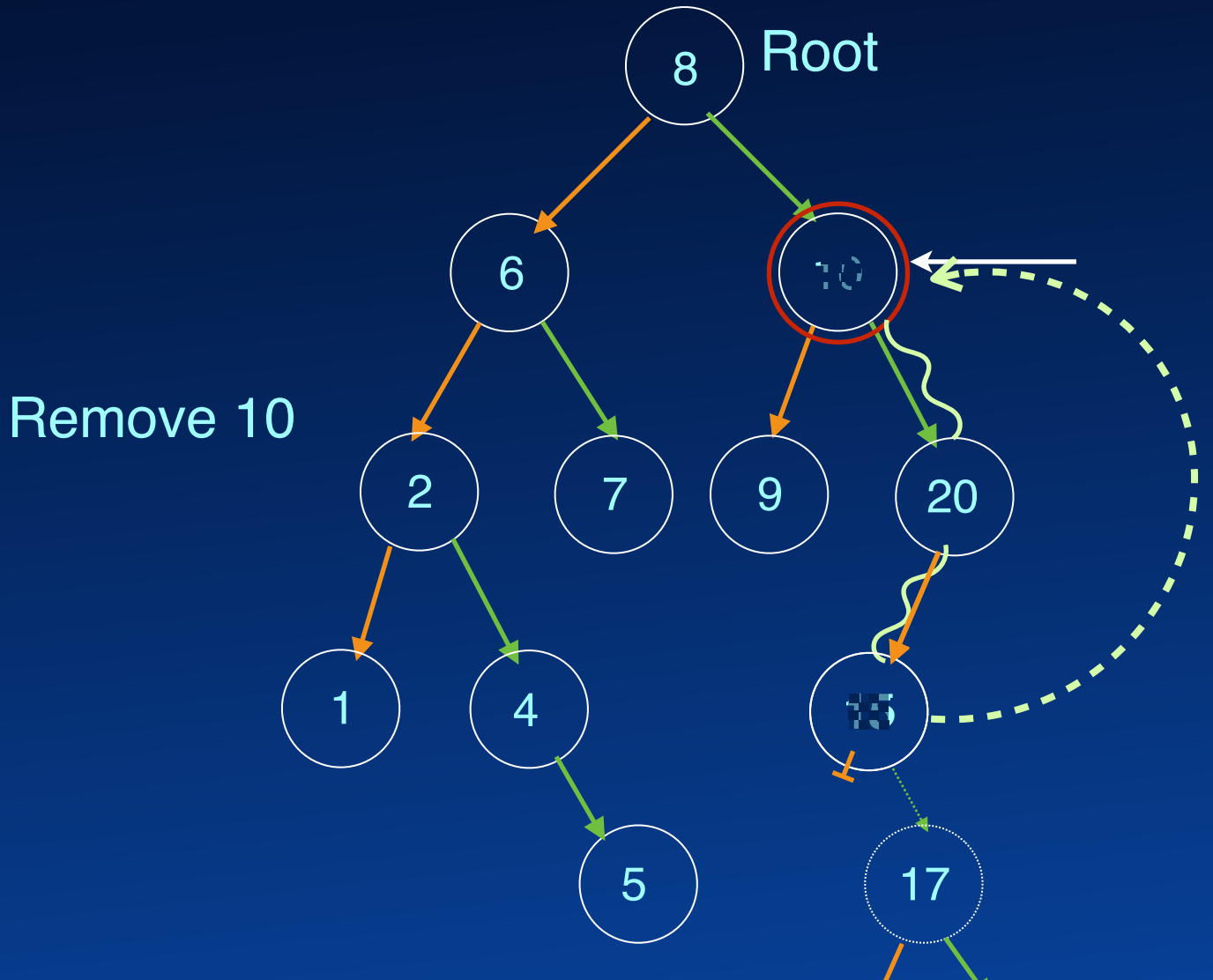



```
class BST<T extends Comparable<T>>  {
    BinaryNode<T> root;
    BinaryNode<T> find(T v)  { return find(root, v); }
    BinaryNode<T> find(BinaryNode<T> node, T v) {
        if(node == null) return null;
        int compared = v.compareTo(node.value);
        if(compared == 0) return node;
    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

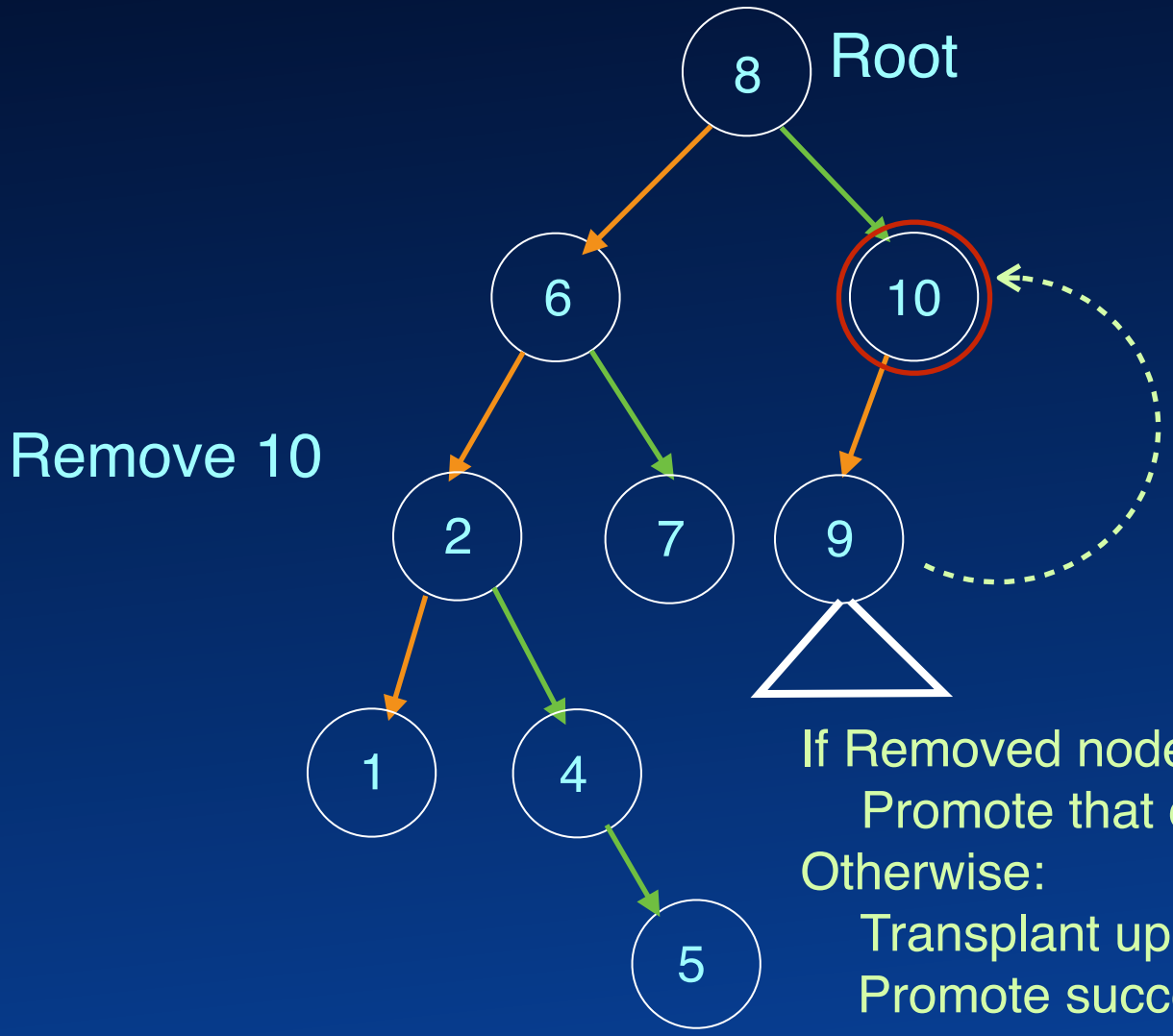
BST Operations



BST Operations



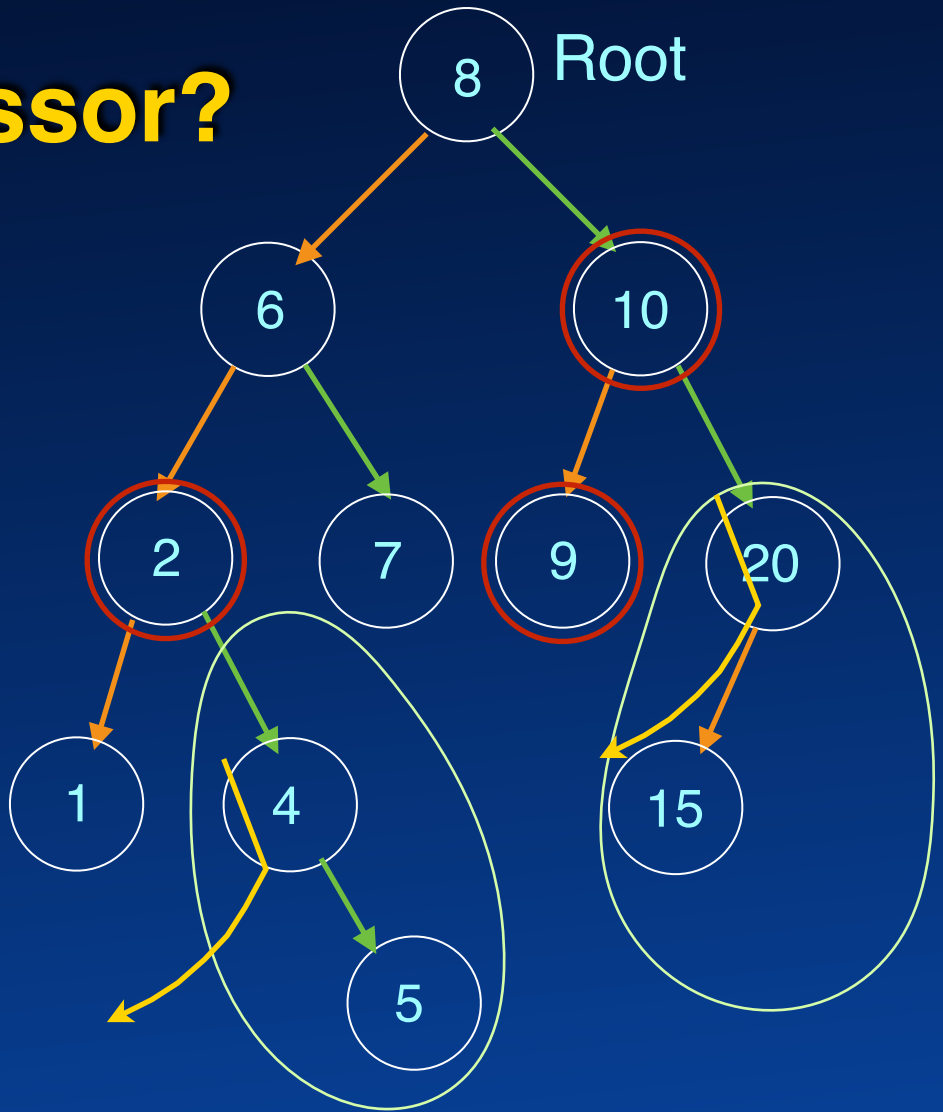
BST Operations



If Removed node has one child:
Promote that child
Otherwise:
Transplant up the successor;
Promote successor's child

Successor

Predecessor?

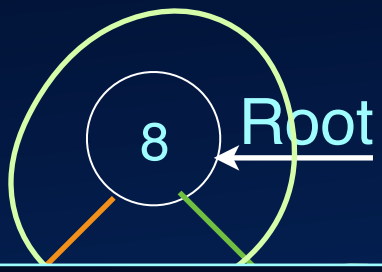


True or False?

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 2logn + 1
- The minimum height of a complete binary tree with n nodes, is ceil(log (n+1))
- The number of non-leaf nodes in an n-node tree is always <= n/2

Traversal



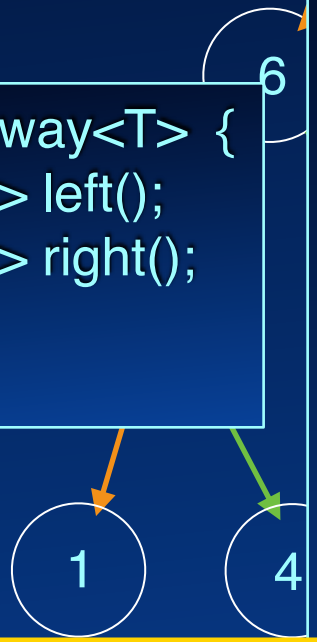

```
class BST<T extends Comparable<T>> {
    BinaryNode<T> root;
    void iterate(Consumer<T> op)  { iterate(root, op); }
    void iterate(BinaryNode<T> node, Consumer<T> op) {
        if(node == null) return;
        op.accept(node.value);
        iterate(node.left, op);
        iterate(node.right,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

Pre-order Traversal

Traversal

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

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




```
ls SortablePair> {
 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

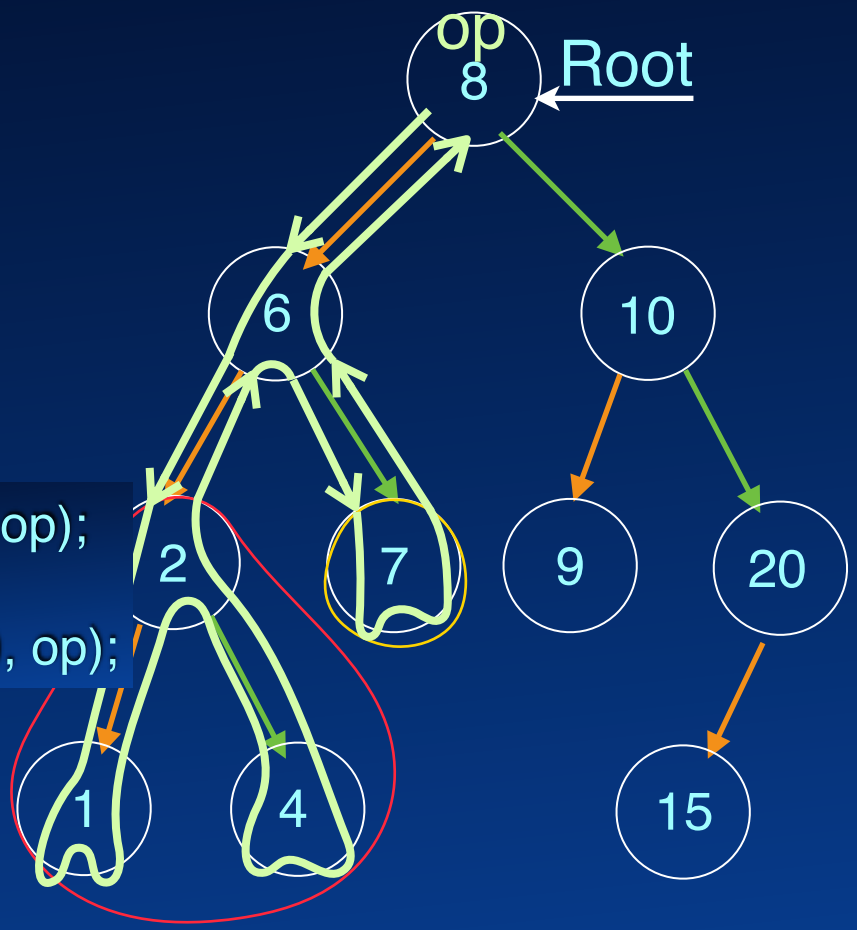
Traversal


```
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) {
        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.left() != null) iterate(node.right(), op);
    }
…
}
```

Euler's Tour

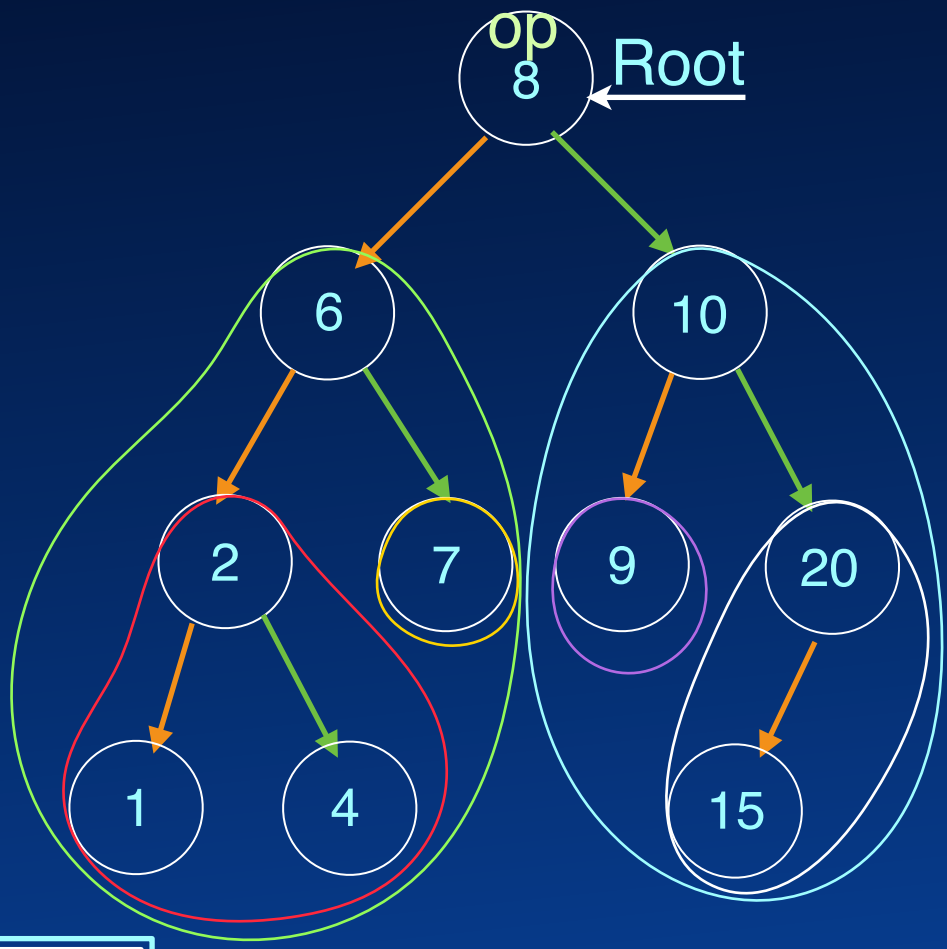


Depth First Traversal

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

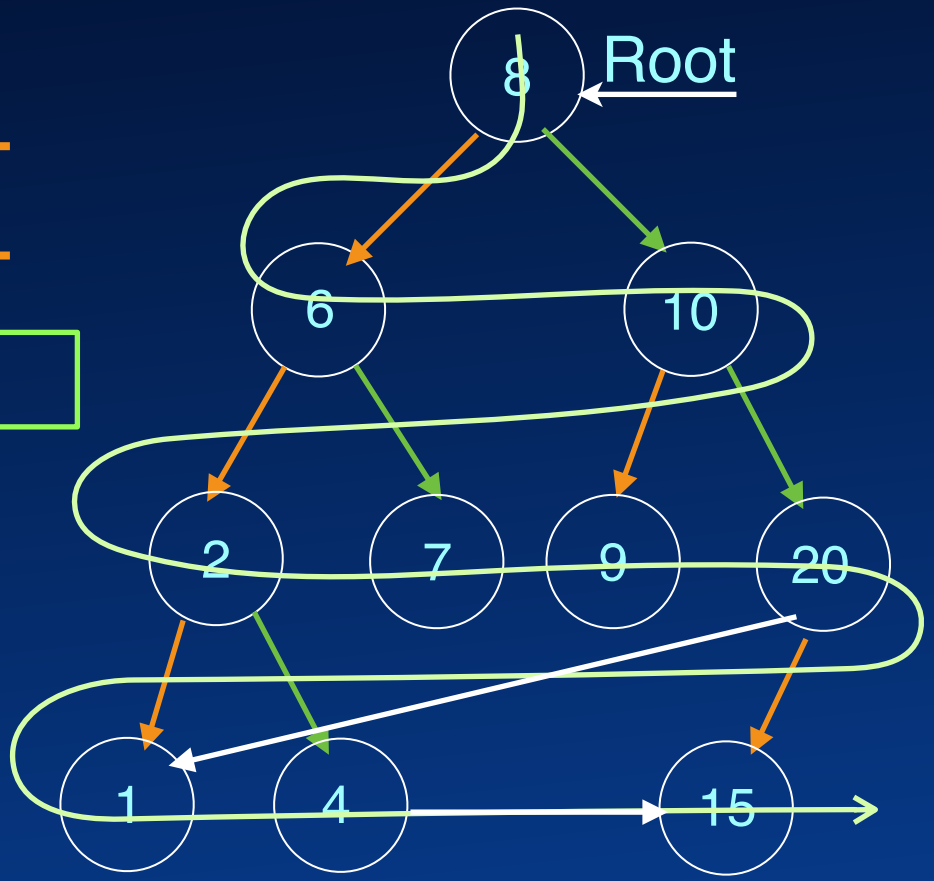


Breadth-first Traversal

Queue

8 6 10 2 7 9 20 1 4 15

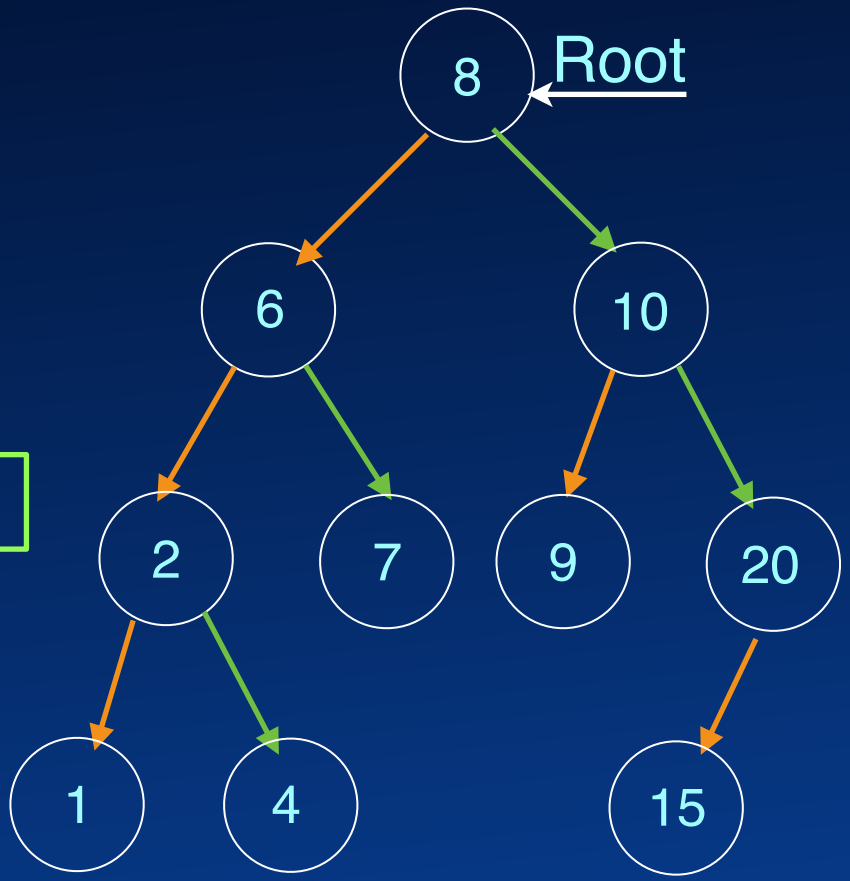
```
q.insert(root)
while(q.hasNext()) {
    x = q.next();
    q.insert(x.left);
    q.insert(x.right);
    op.accept(x);
}
```



Depth-first Traversal

Stack
Queue

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

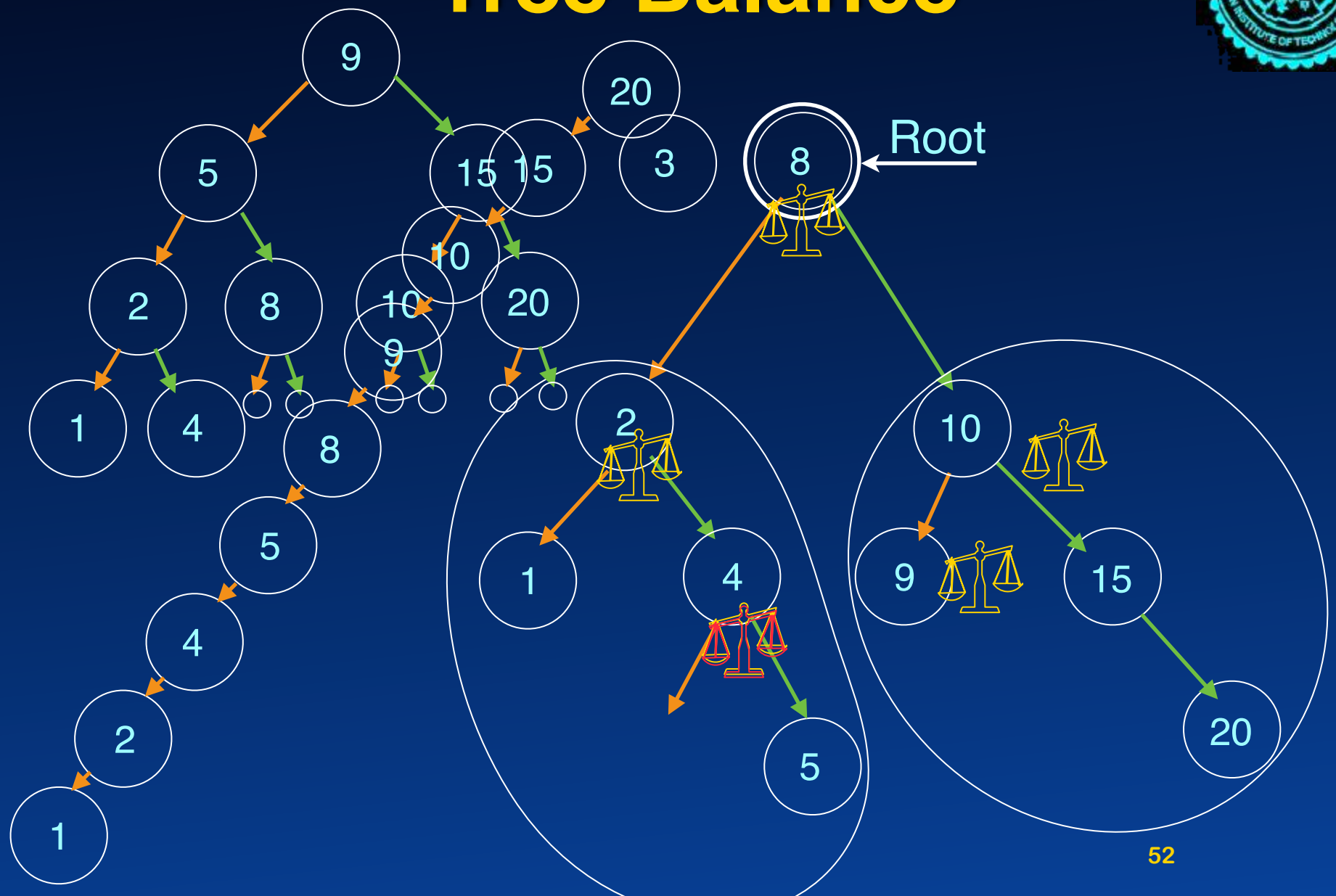


True or False?

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 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



Balanced Tree

Count balanced (Weight balanced):

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

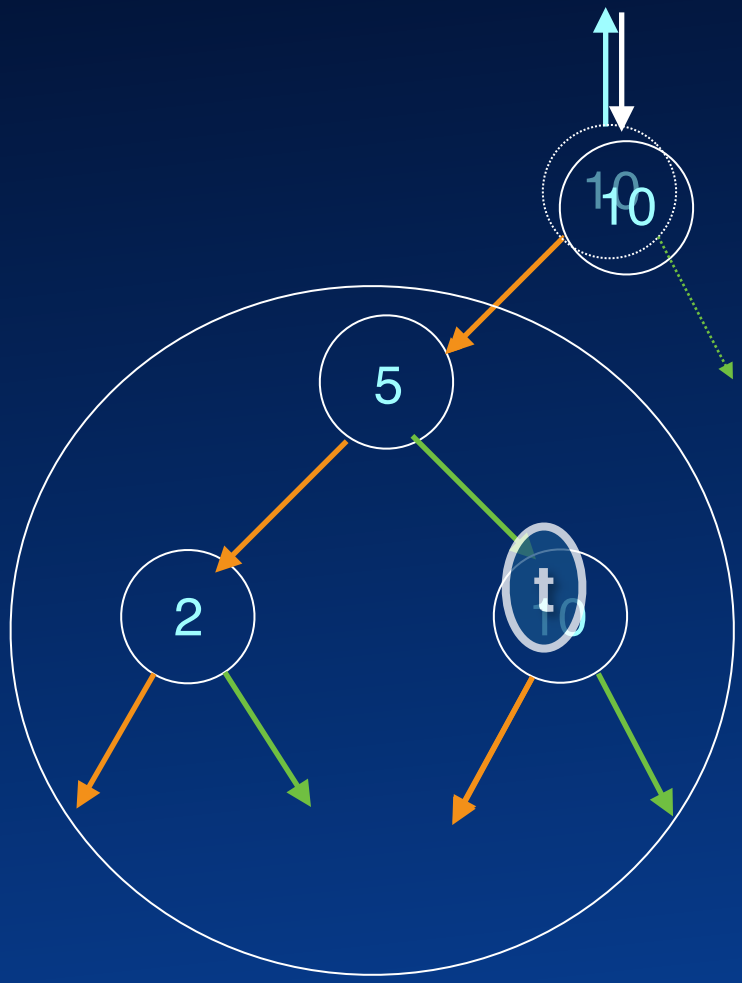
$\alpha < .29$

Height balanced:

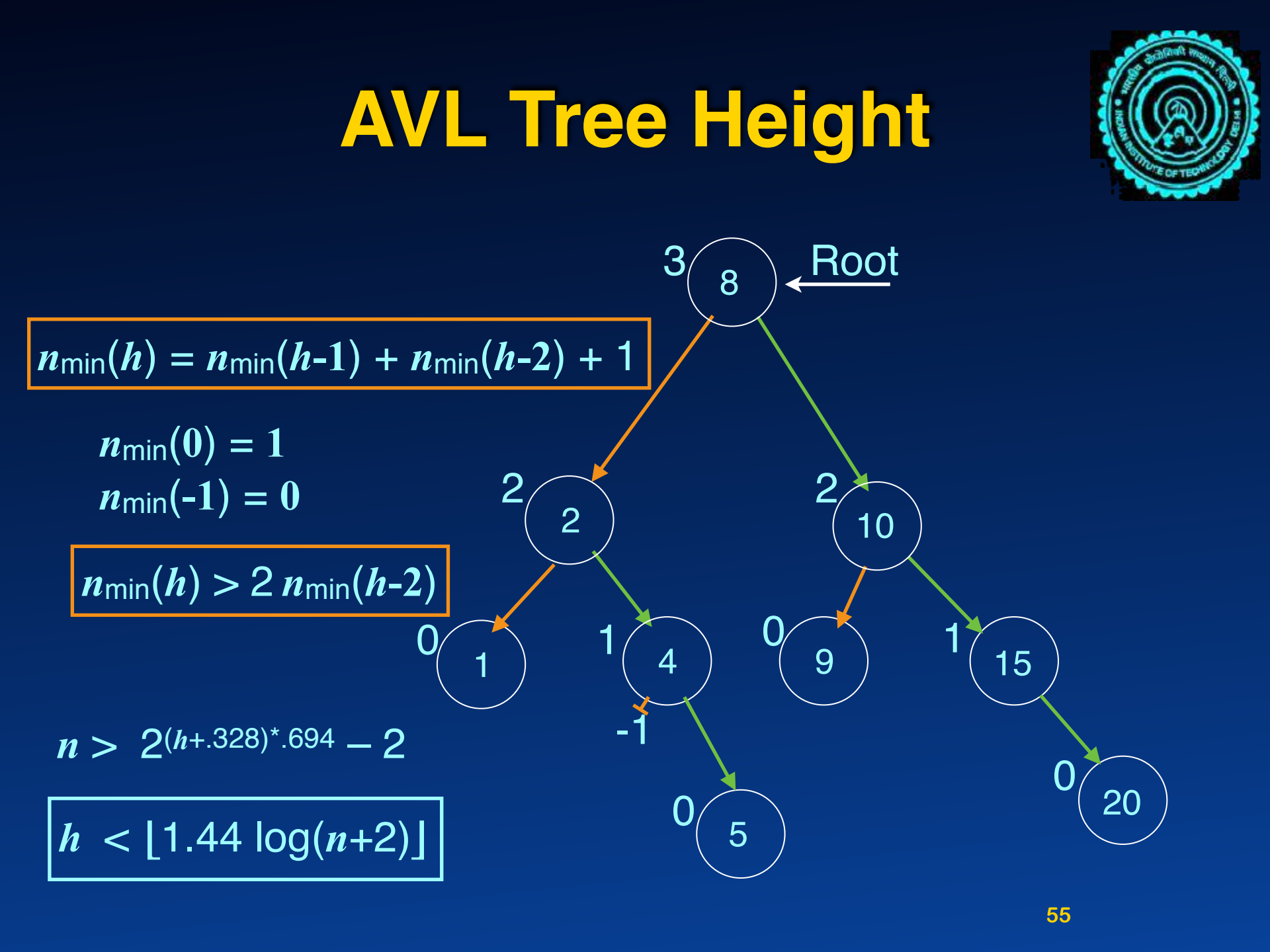
Height of left subtree is within ±1 of that of right subtree

AVL Tree

Rotation

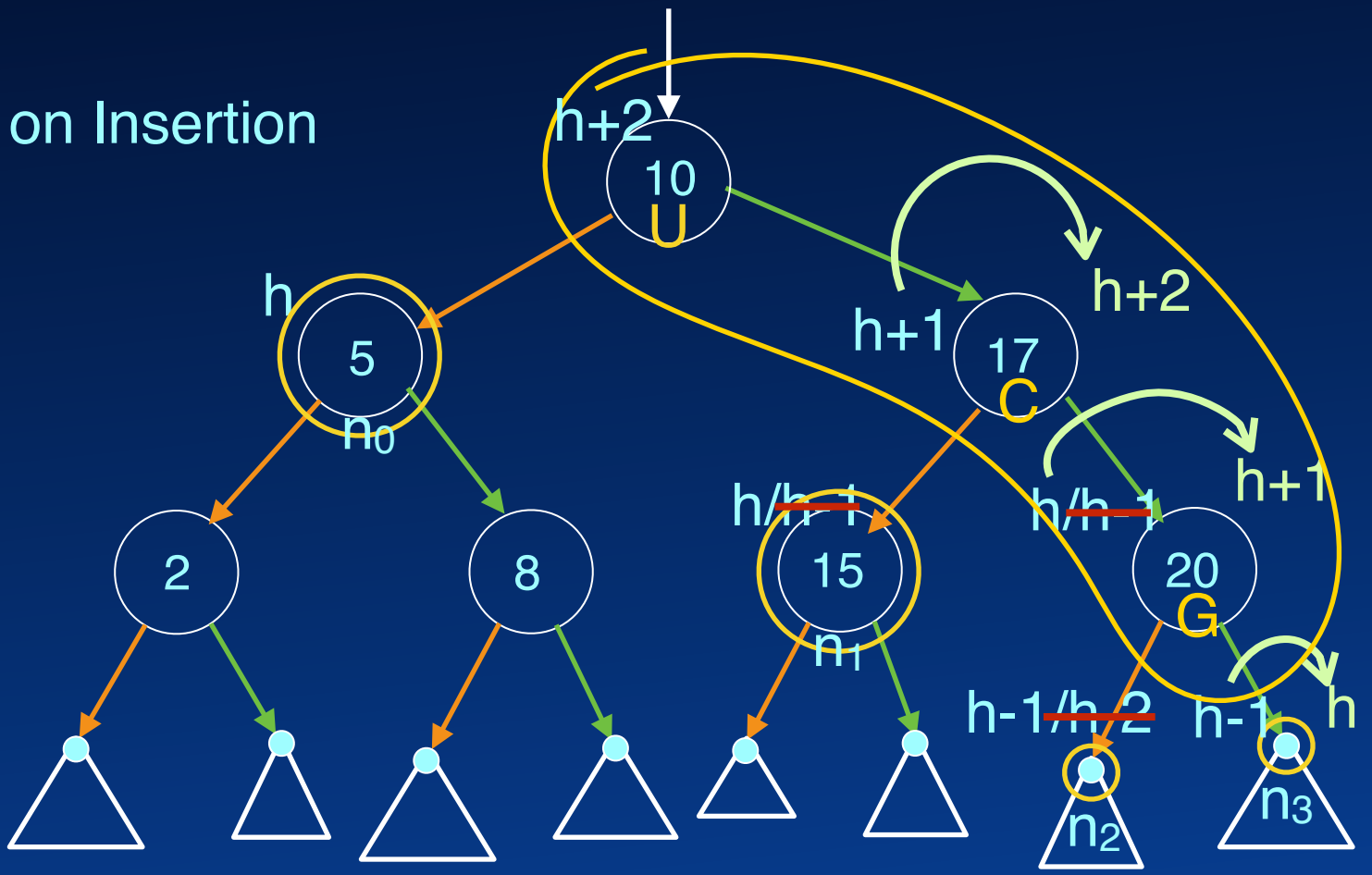


AVL Tree Height

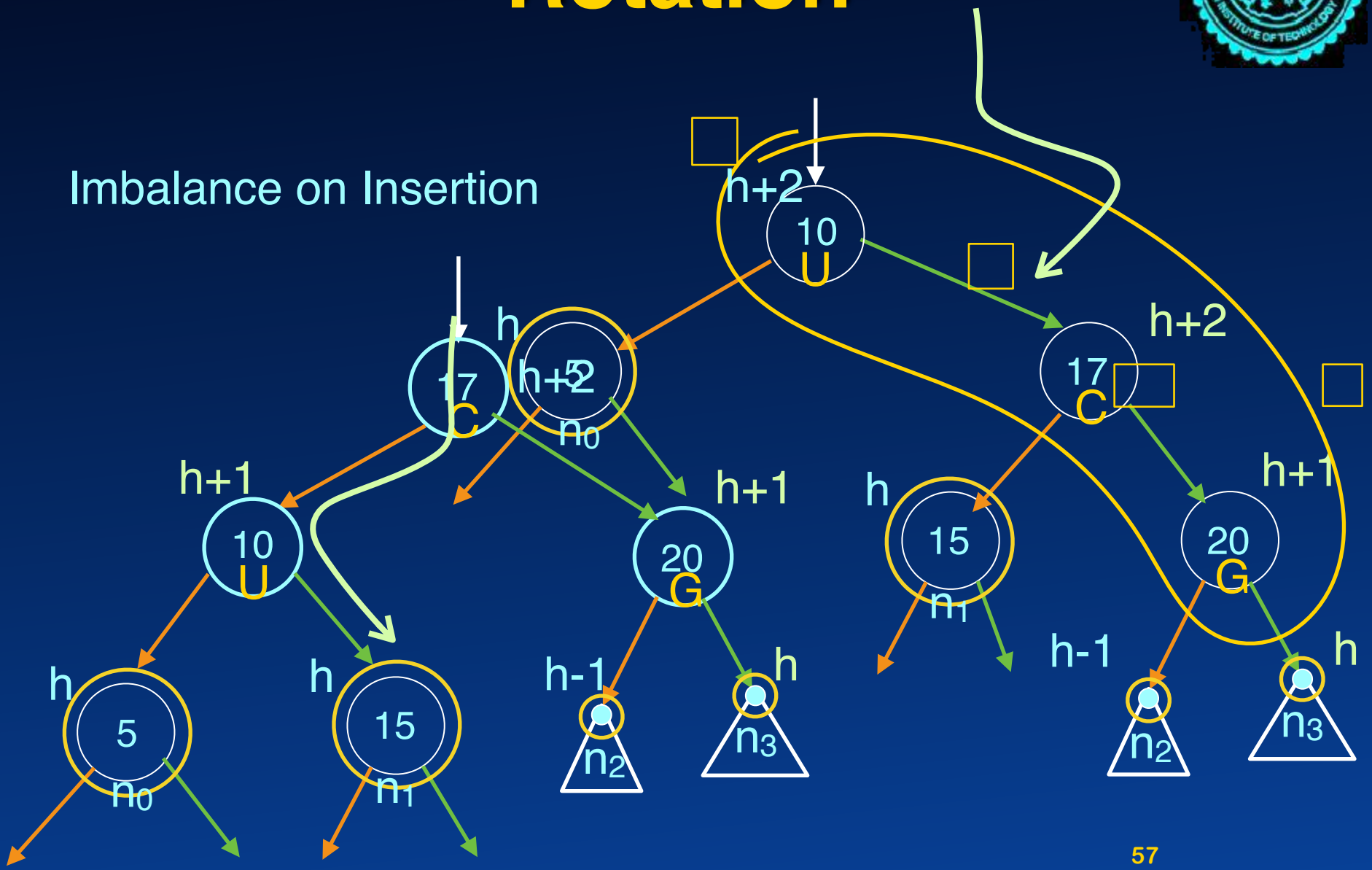


Rotation

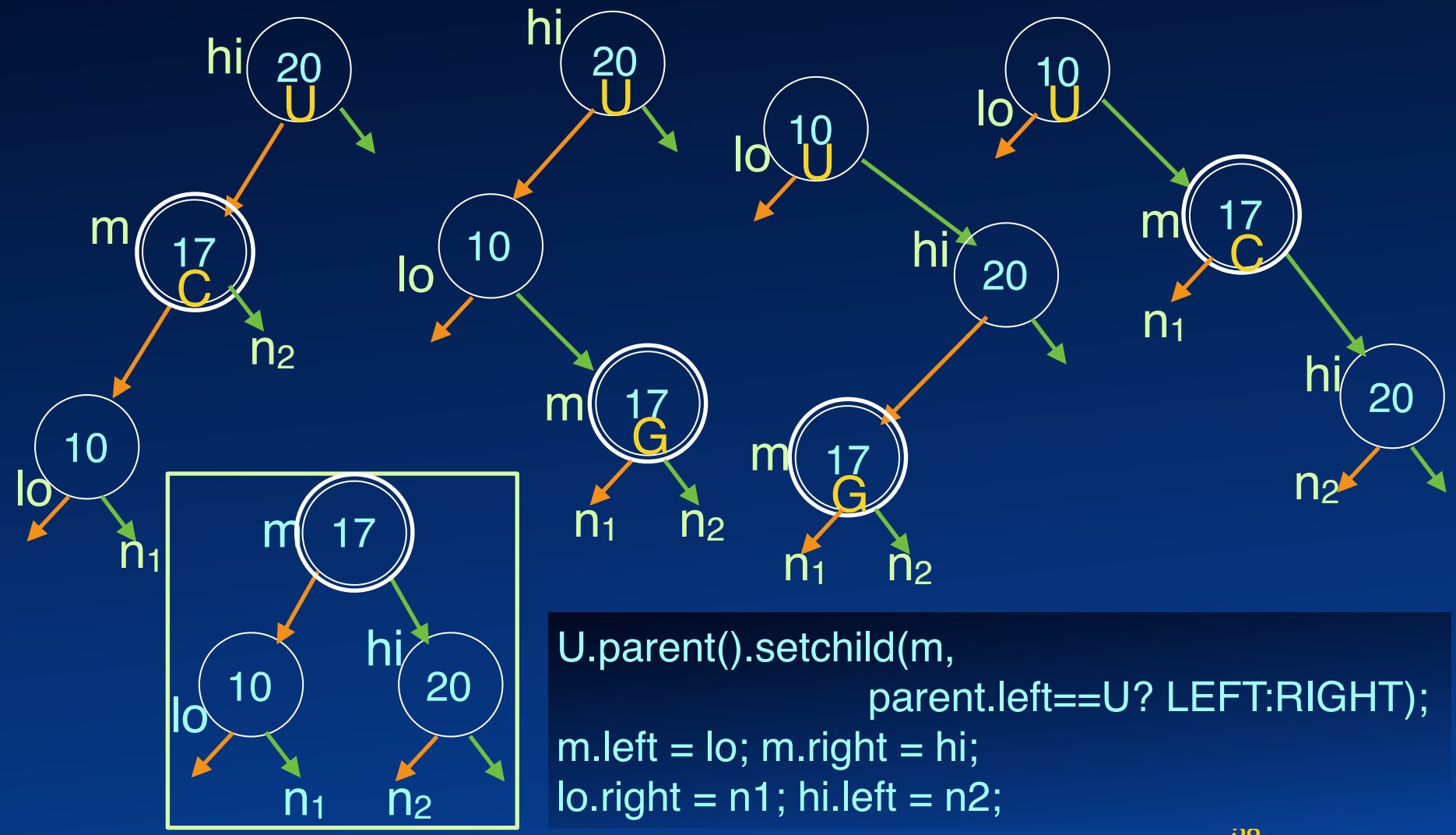
Imbalance on Insertion



Rotation


Imbalance on Insertion
h+2
10
U
h
17
C
h+2
h+2
17
C
h+1
10
U
h+1
20
G
h
15
h+1
20
G
h
5
n_0
h
15
n_1
h-1
n_2
h
n_3
h-1
n_2
h
n_3

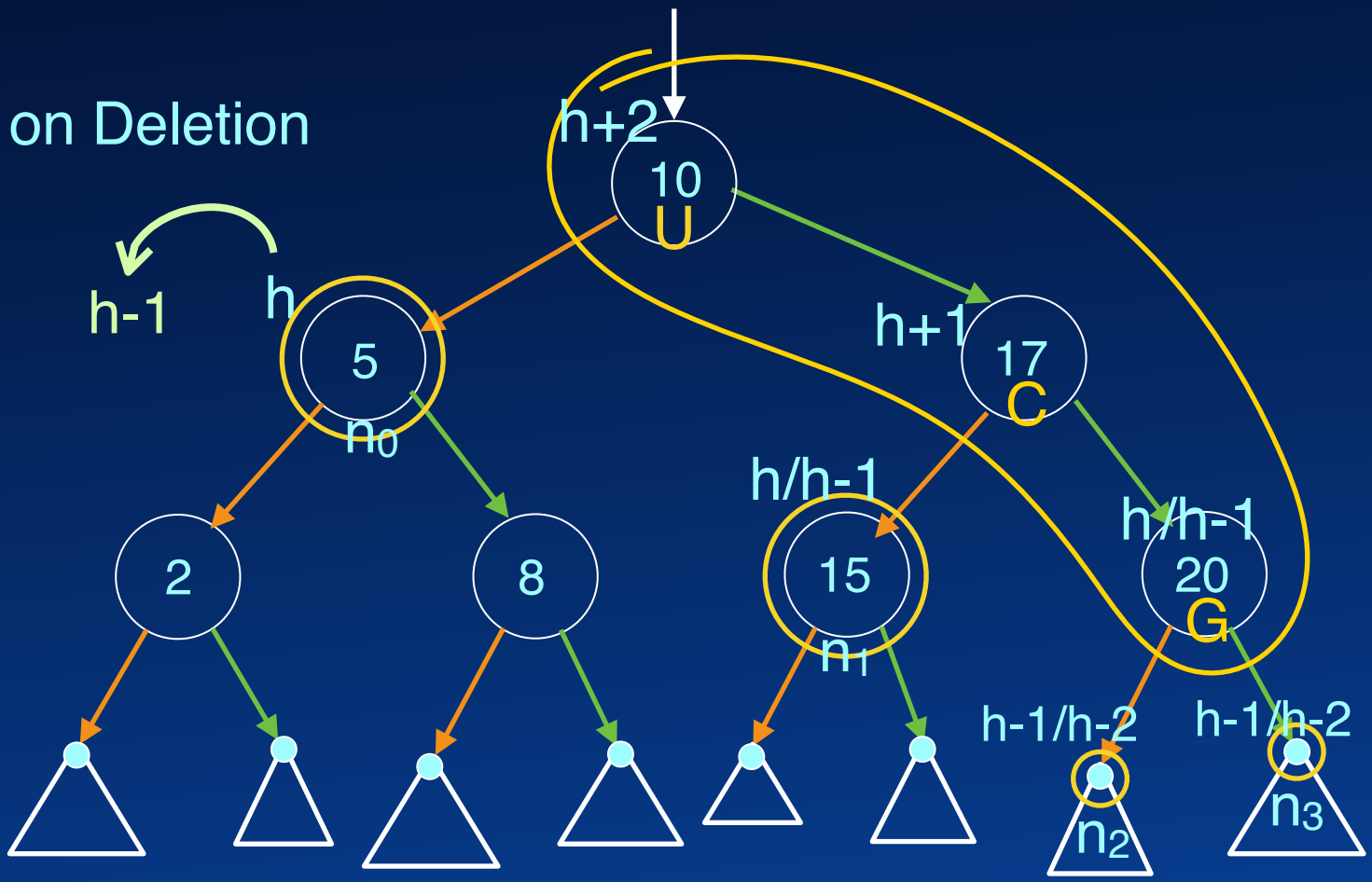
Rebalancing AVL Tree



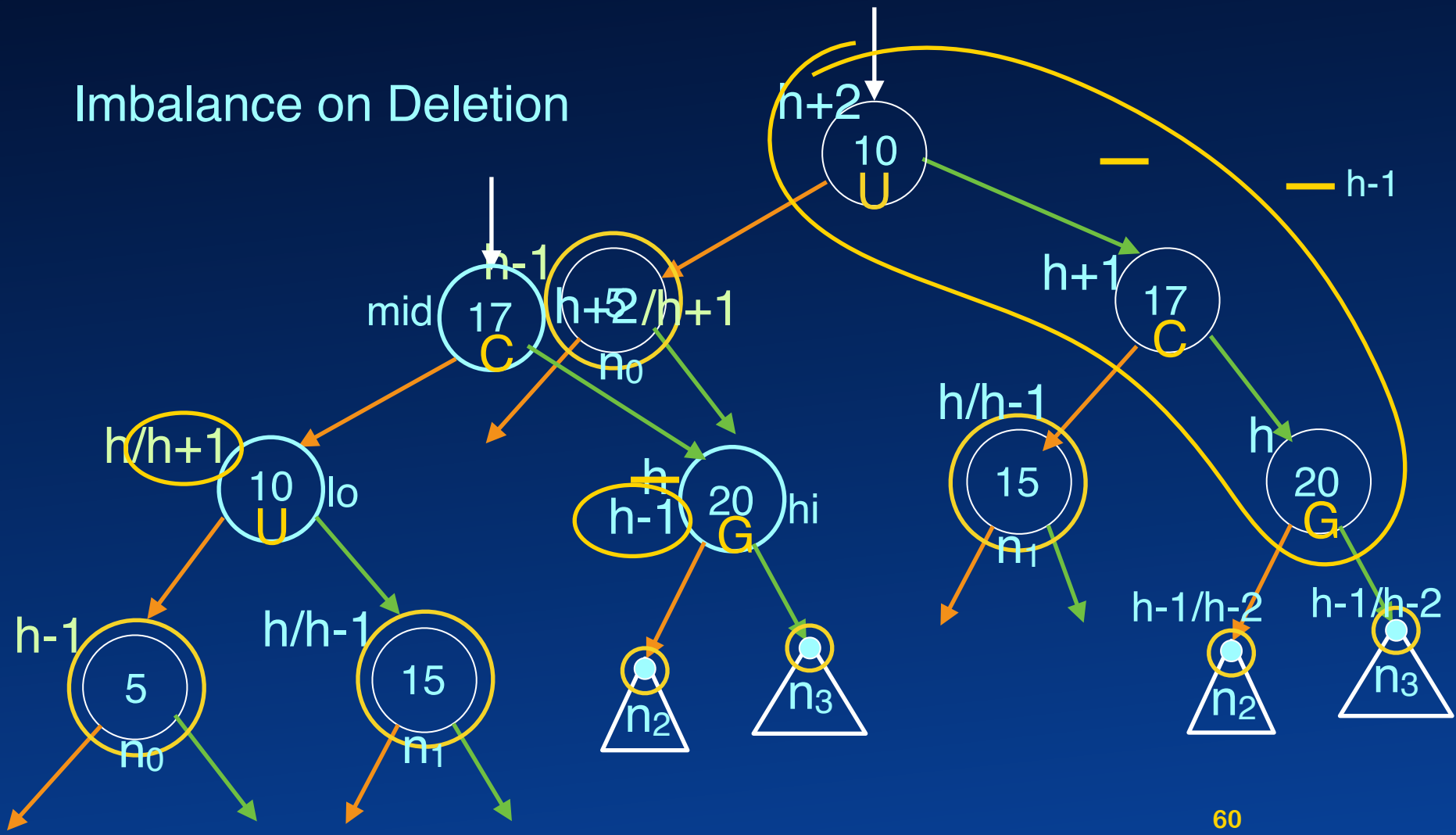

```
U.parent().setchild(m,
                    parent.left==U? LEFT:RIGHT);
m.left = lo; m.right = hi;
lo.right = n1; hi.left = n2;
```

Rotation

Imbalance on Deletion

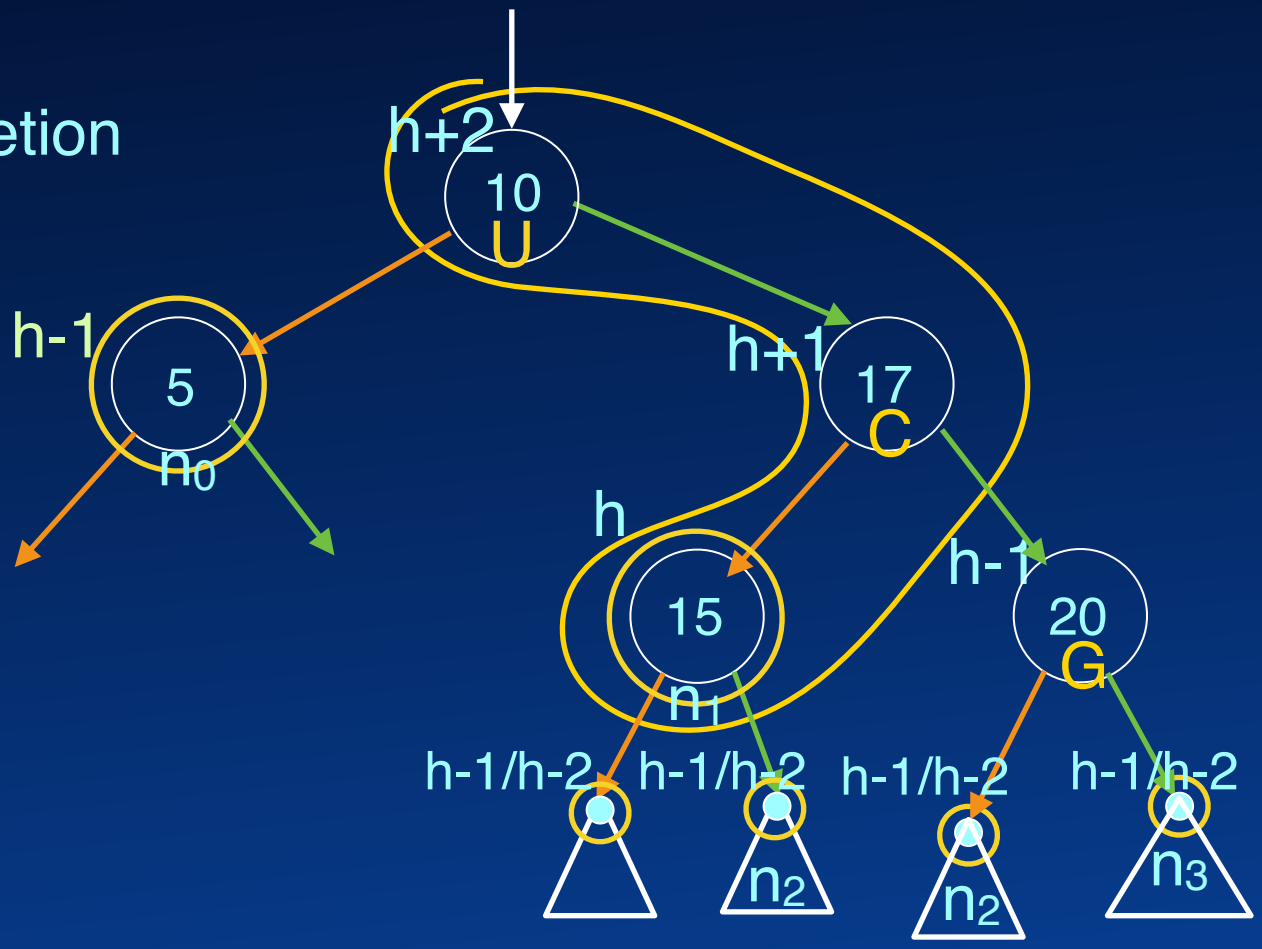


Rotation

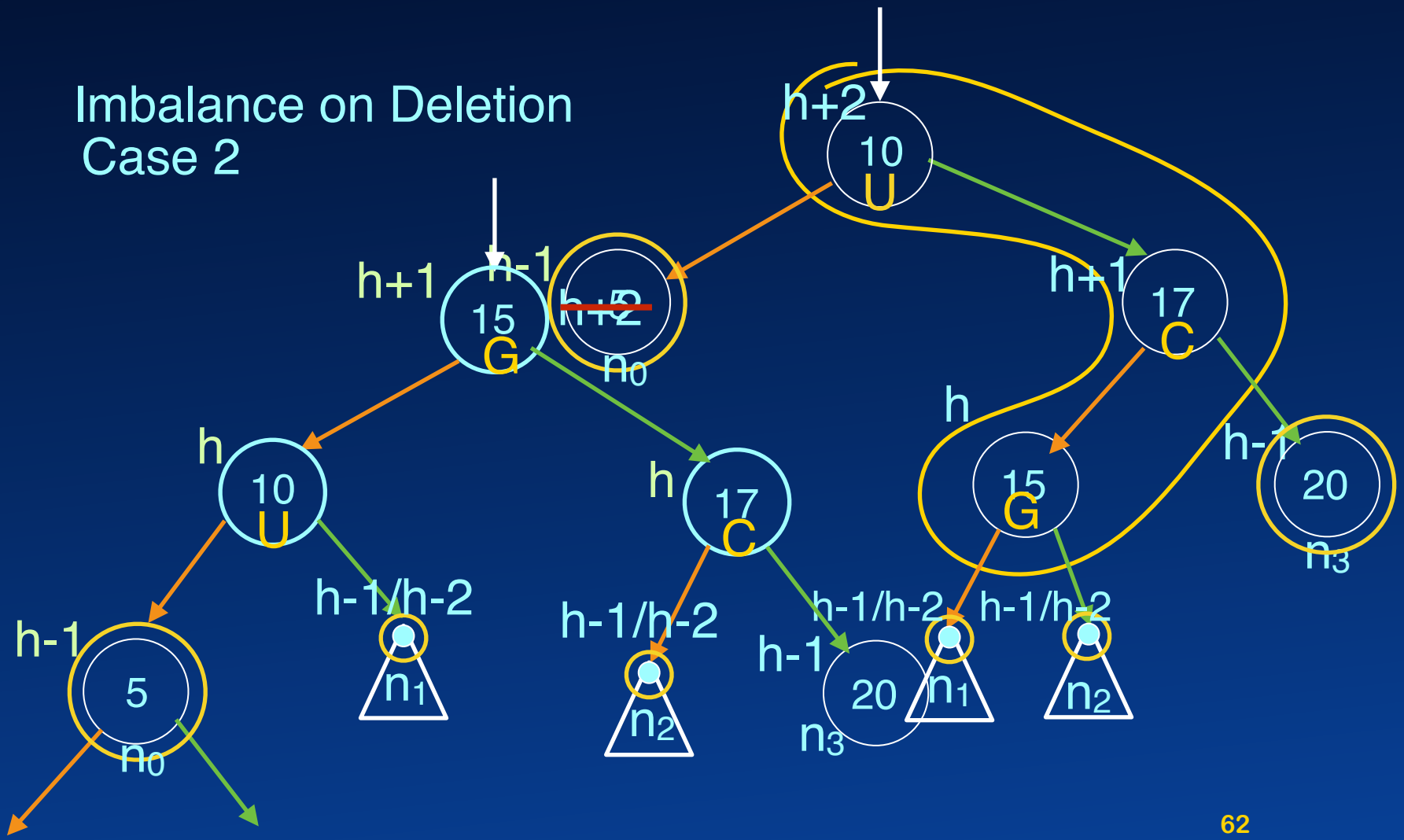

Imbalance on Deletion
h+2
10
U
h-1
h+1
17
C
mid
h-1
h+2/h+1
5
n0
h/h+1
10
U
lo
h
h-1
20
G
hi
h/h-1
15
n1
h
20
G
h-1
5
n0
h/h-1
15
n1
n2
n3
h-1/h-2
n2
h-1/h-2
n3

Rotation

Imbalance on Deletion
Case 2



Rotation


Imbalance on Deletion
Case 2
h+2
10
U
h+1
15
G
h-1
h+2
n0
h+1
17
C
h
10
U
h
17
C
h
15
G
h-1
20
n3
h-1
5
n0
h-1/h-2
n1
h-1/h-2
n2
h-1
20
n3
h-1/h-2
n1
h-1/h-2
n2

True or False?

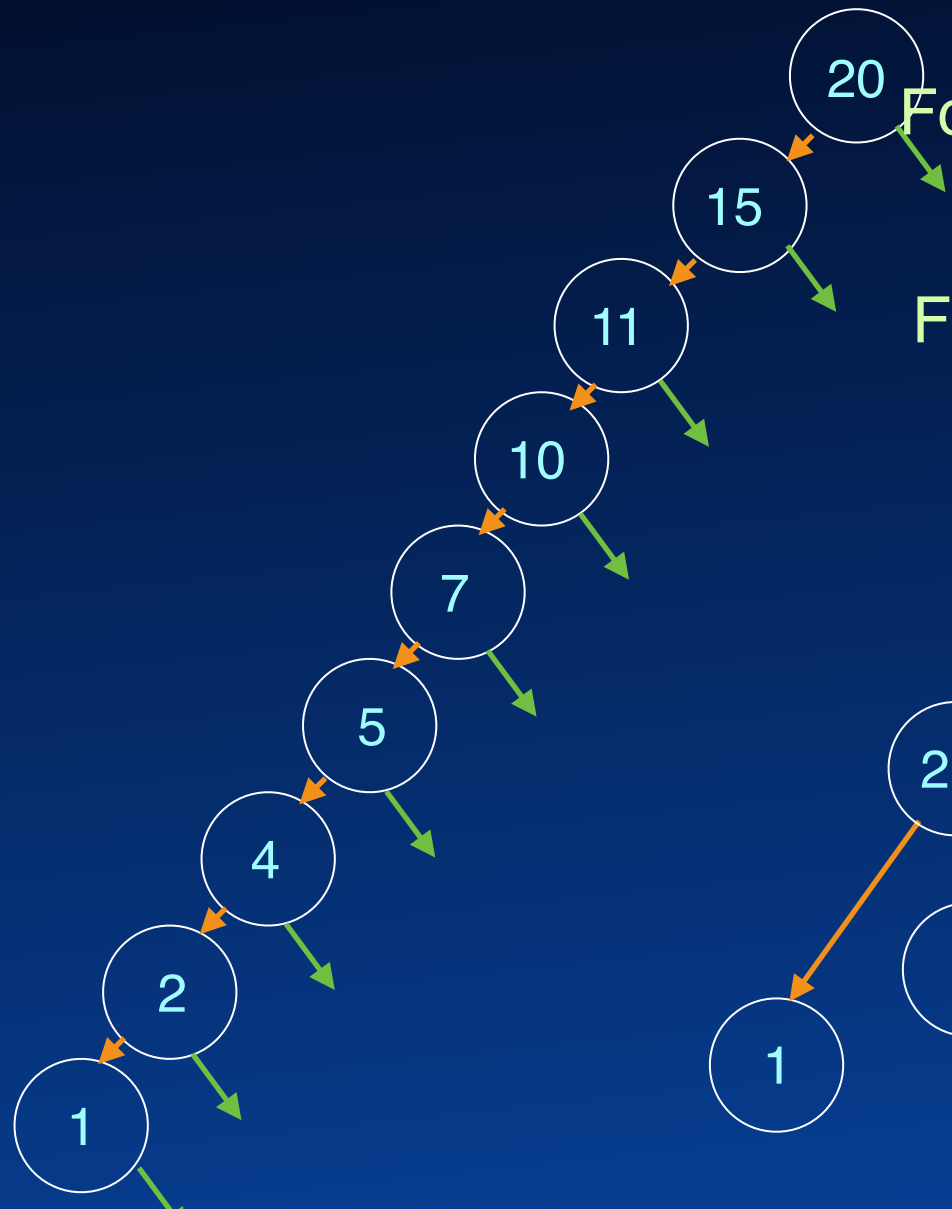
Format: t,t,f
Mail: col106quiz@cse.iitd.ac.in

- 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 re-structuring when deleting a node in AVL tree with n nodes is log n

AVL Updates

- On insertion, re-structure the deepest unbalanced node
  - C and G lie on the side of the insertion
  - Reduces the height of the taller side and fixes the imbalances of all ancestors
- On deletion, re-structure the only unbalanced node
  - C is on the other side of the deletion
  - G is the taller child of C
    - If both children of C have the same height, choose the Left-left or Right-right configuration for U C G
  - Restructure could reduce height causing an imbalance in the parent
    - Traverse higher and repeat, until no imbalance remains

Common Leaf Level Tree

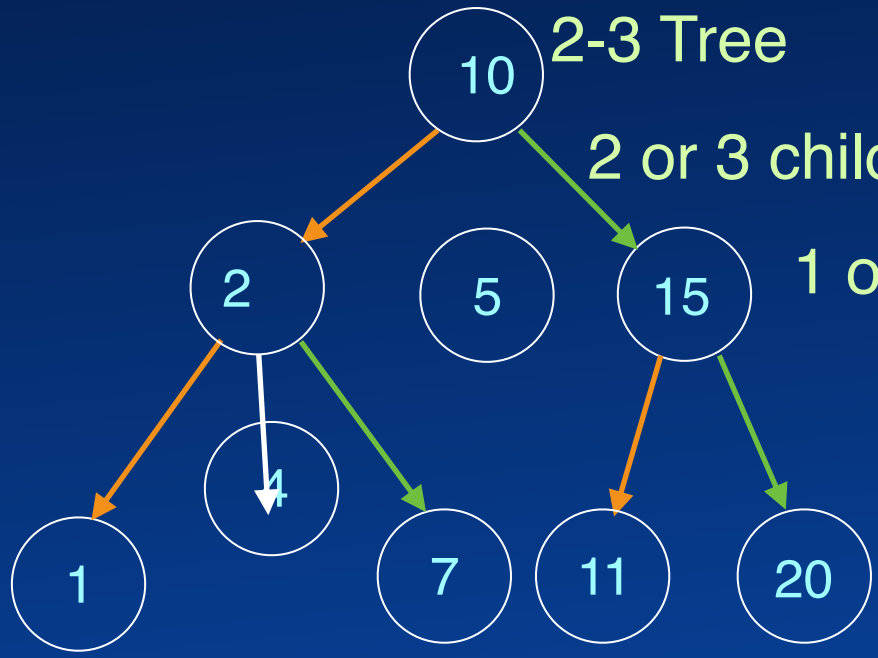


For n nodes:

$\lfloor \log_3(2n) \rfloor \leq \text{height} \leq \lfloor \log_2(n) \rfloor$

For k keys:

$\lfloor \log_3(k) \rfloor \leq \text{height} \leq \lfloor \log_2(k) \rfloor$

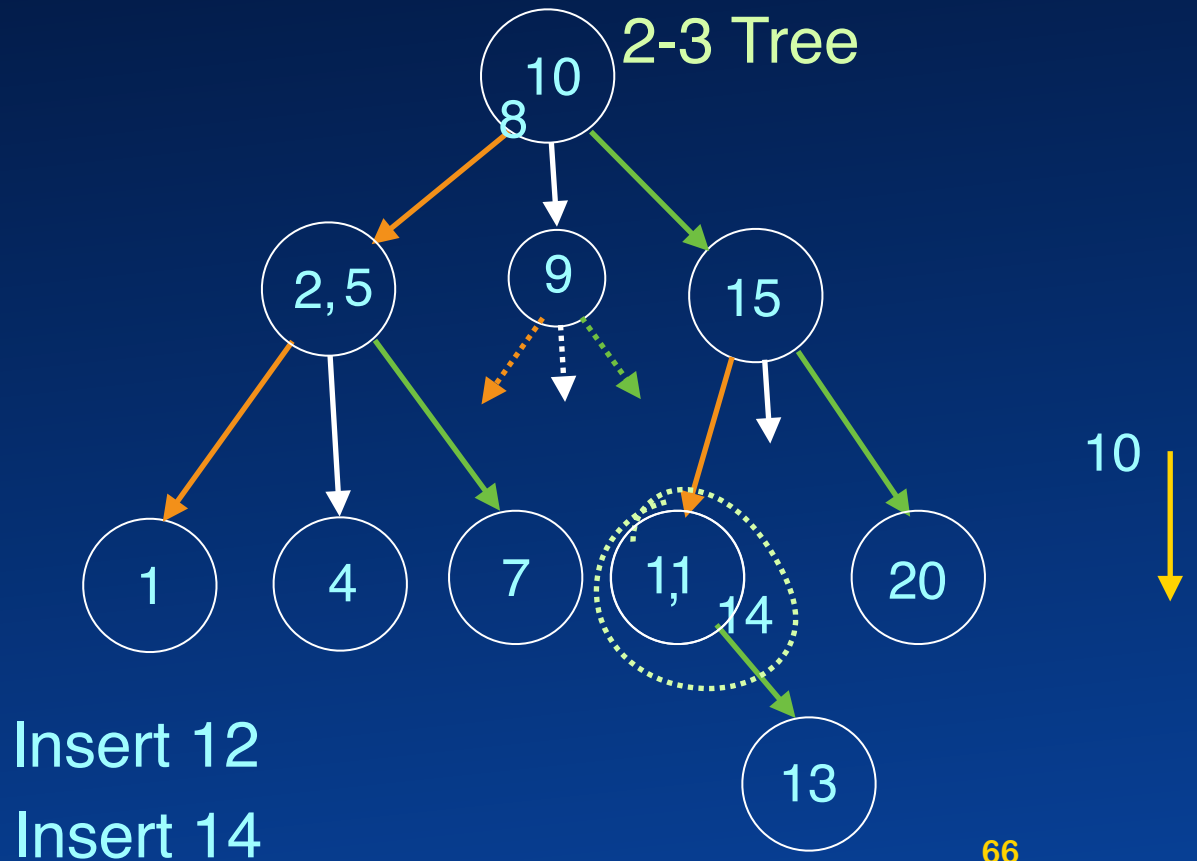


2-3 Tree

2 or 3 children/node

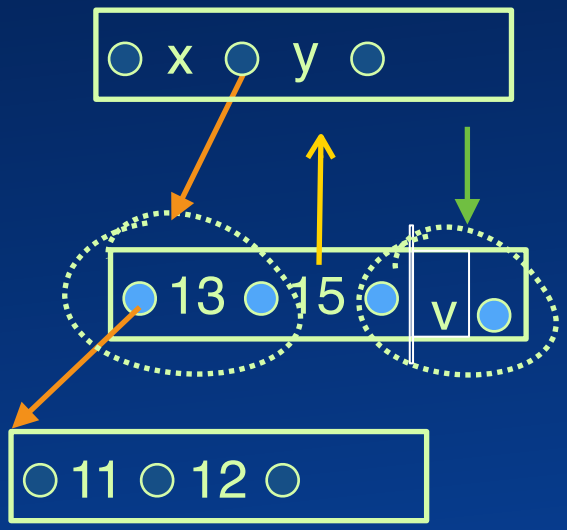
1 or 2 keys

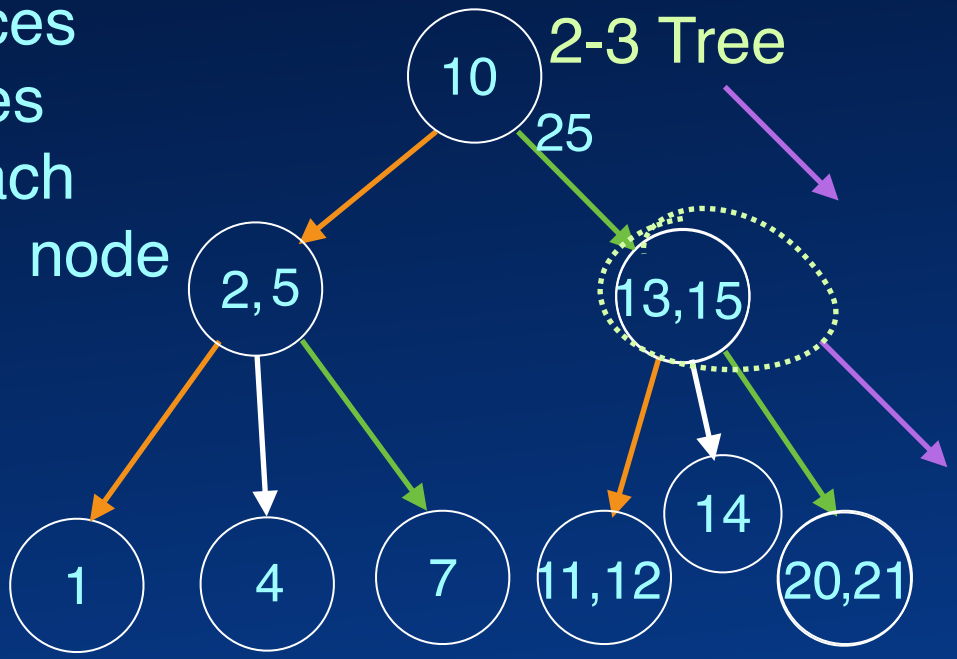
Insert in 2-3 Tree



Insert in 2-3 Tree

- Always insert a child with a key
  - At the leaf, the child may be null
- If the insertion causes overflow
  - ⇒3 Keys, 4 references
  - Split into two nodes
  - One key, 2 refs each
  - Promote 1 key + 1 node

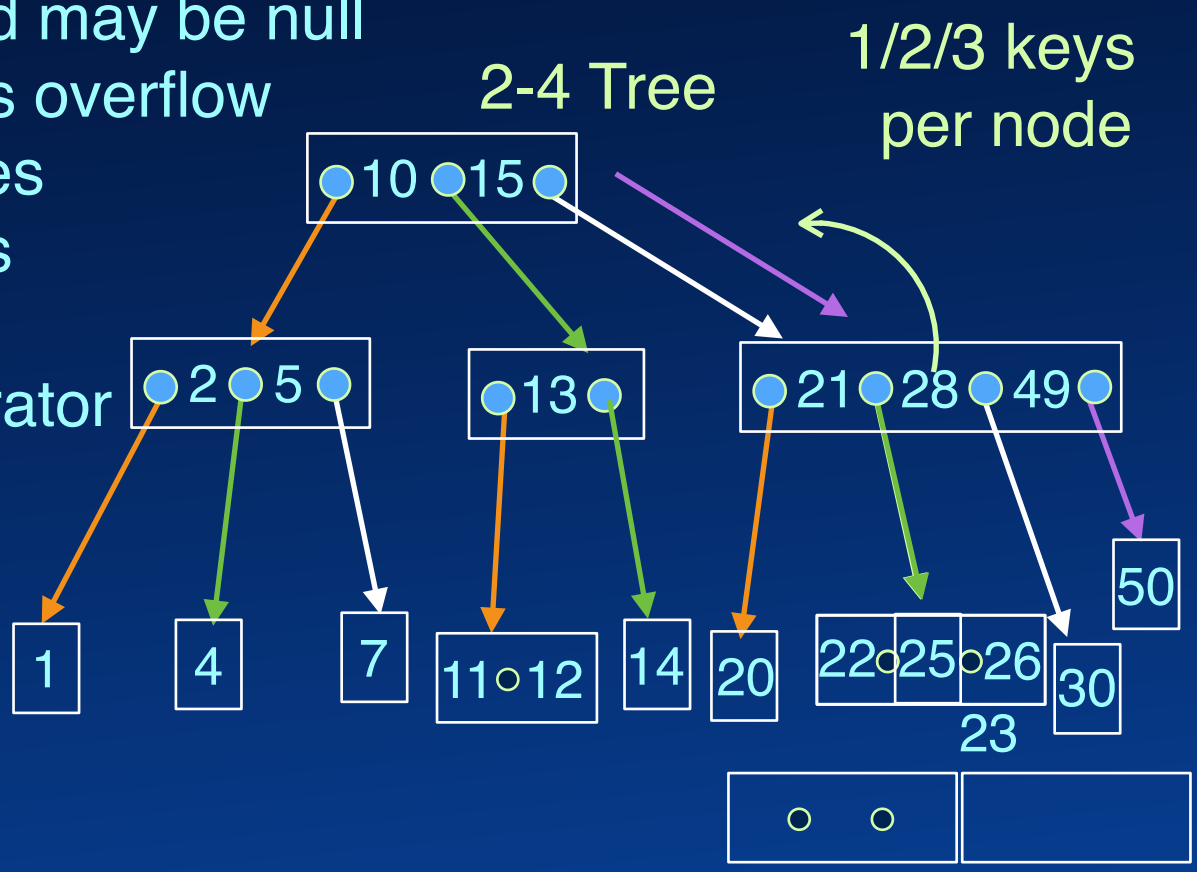




Insert 25

Insert in 2-4 Tree

- Always insert a child with a key
  - At the leaf, the child may be null
- If the insertion causes overflow
  - ⇒4 Keys, 5 references
  - Split into two nodes
  - 1 key, 2 refs each
  - And promote separator



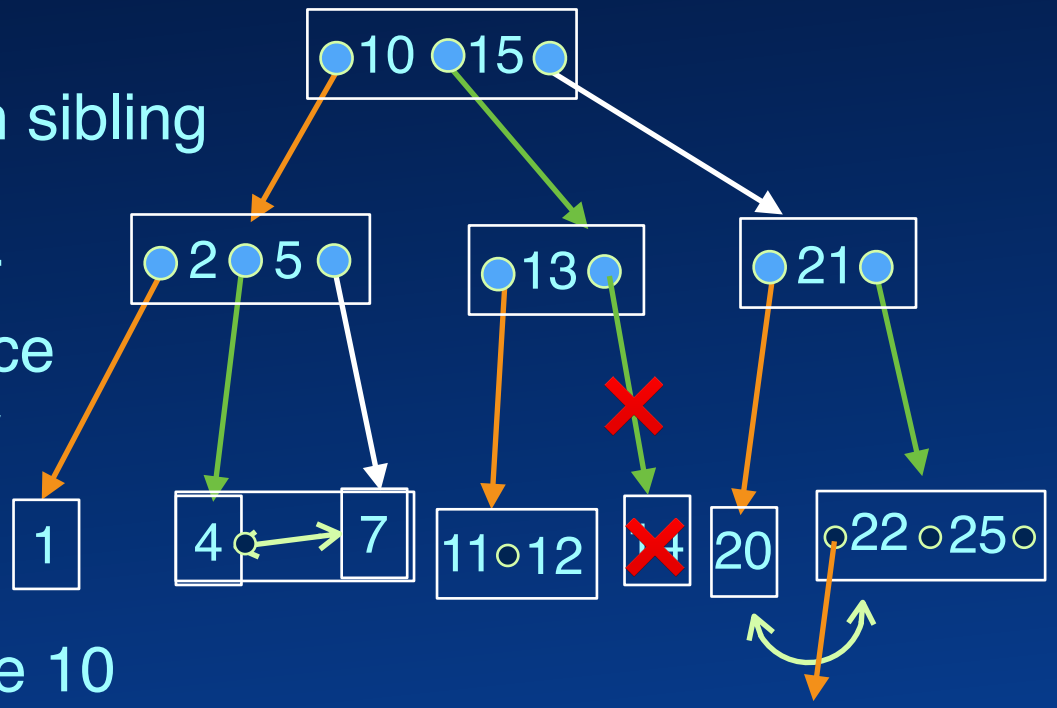
Works for a-b tree

$a > 1$

Delete in 2-3 Tree

In a-b tree:
Merged node must not exceed b children
a-1 children + a children <= b children

- ssor
  - Rotate: promote sibling key, demote parent key
  - Graft child
- Otherwise, merge with sibling
  - For parent:
    - demote separator
    - remove 1 reference
    - Fix any underflow



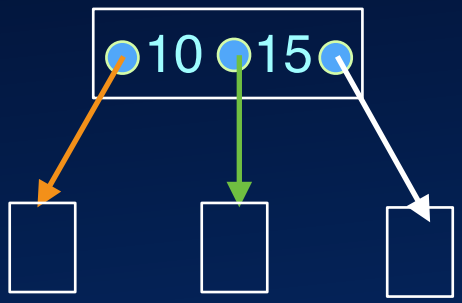
Delete 10
Delete 7
Delete 20

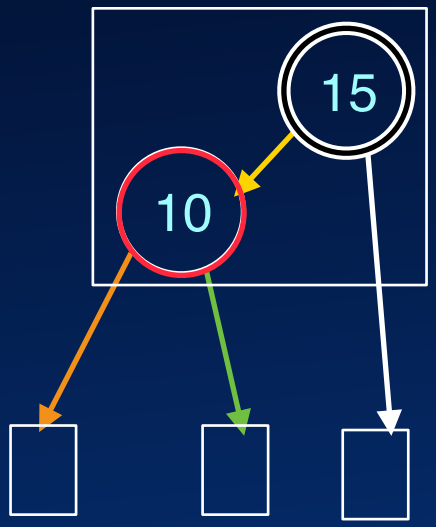
True/False for a-b tree?

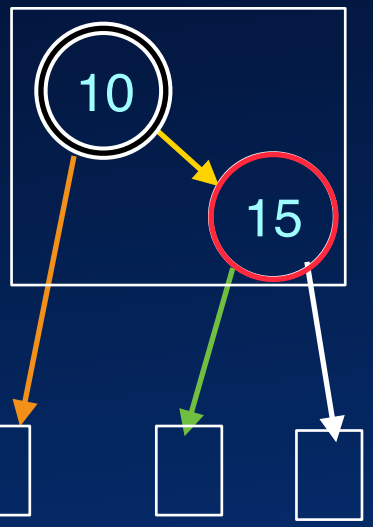
- All leaves are at the same depth
- Each node has at least b and at most b children
- $2a \leq b+1$
- The root is allowed to have fewer than a children, but no fewer than 2
- A node with x children has x-1 keys

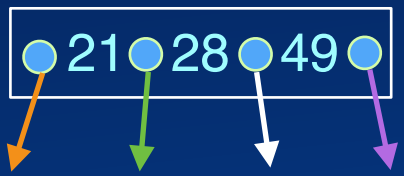
Format: t,f,t,t,t/f
Mail: col106quiz@cse.iitd.ac.in

Binar-izing a 2-4 Tree

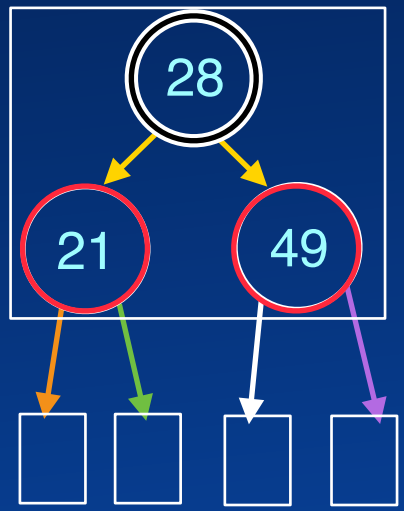


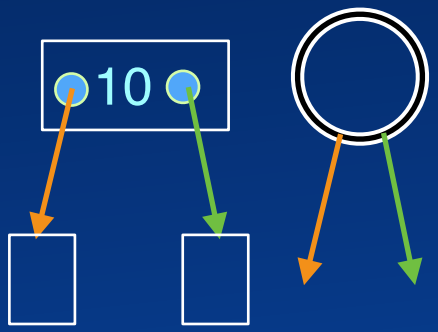






Red Black Tree





Red Black Tree

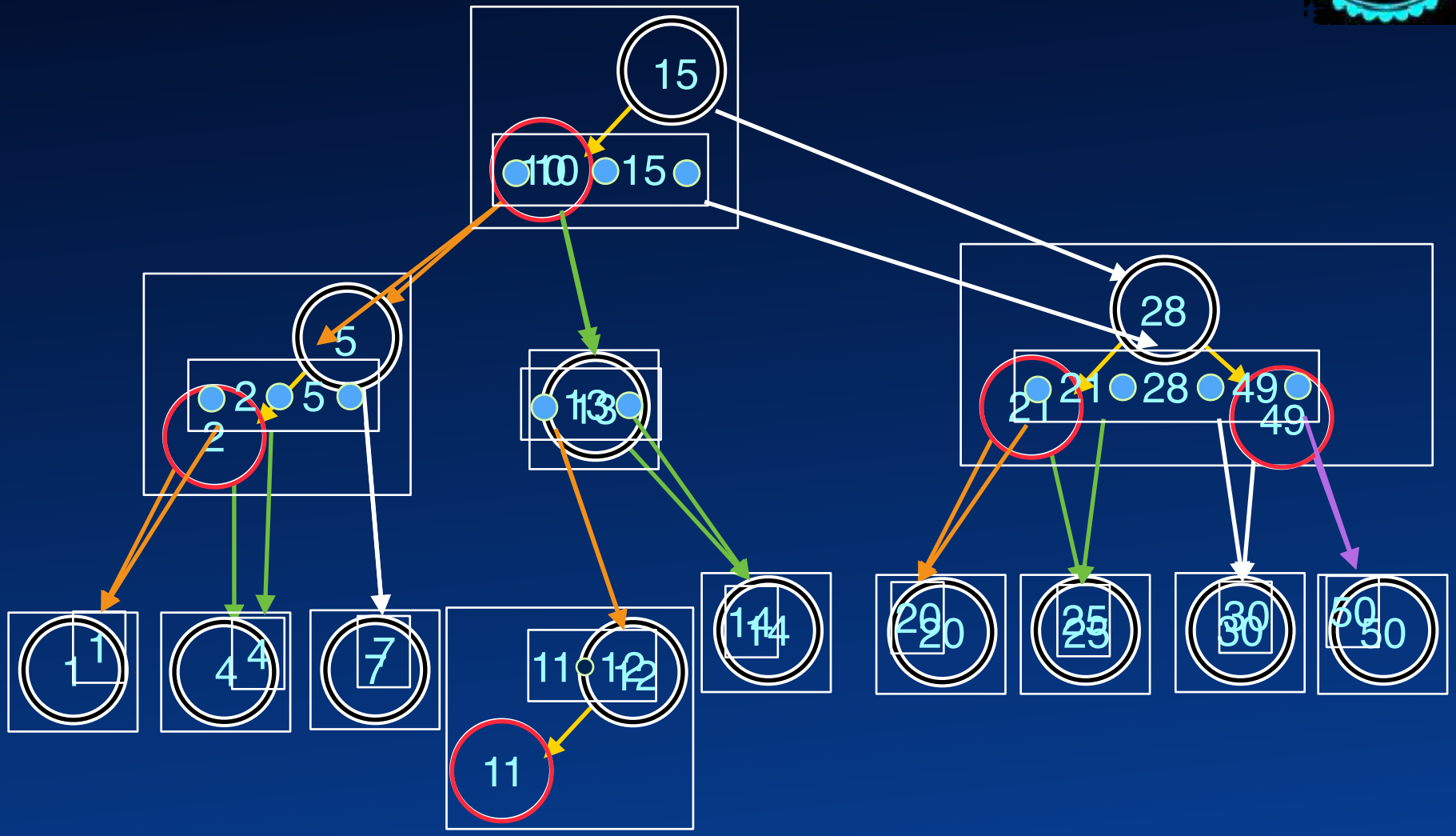
- **No red colored node has a red colored child**
- **The number of black colored node on the path from the root to each null reference is the same**

=> height ≤ 2 log (n)

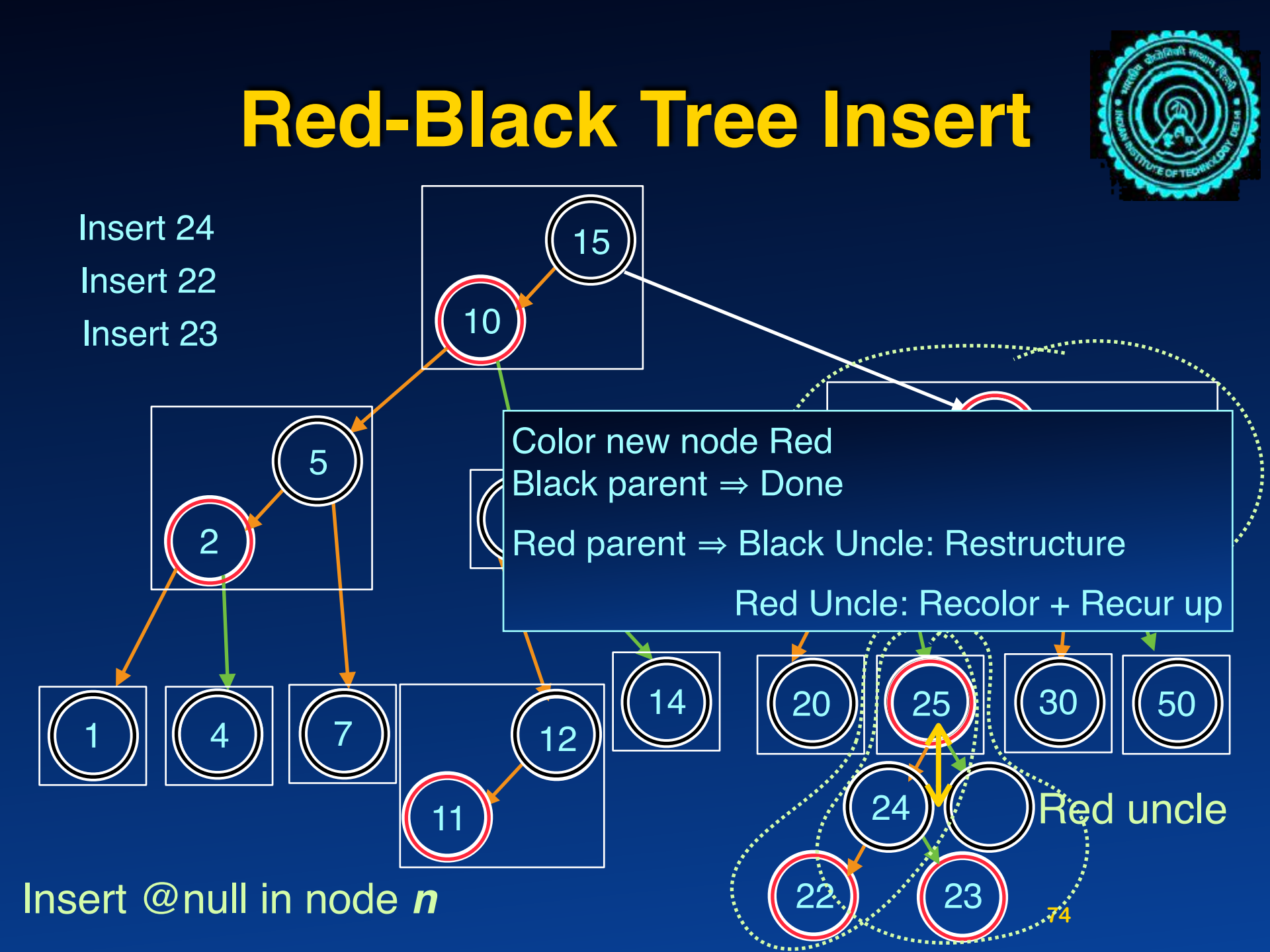
- Null references are assumed black
- Root is always black

2-4 tree height: $\lg_4 n = .5 \lg_2 n$ to $\lg_2 n$ [Up to 2 comparisons /node]
2-3 tre height: $\lg_3 n = .63 \lg_2 n$ to $\lg_2 n$ [Up to 2 comparisons /node]
AVL tree height between $\lg_2 n$ to $1.4\lg_2 n$ [Up to 1 comparison /node]

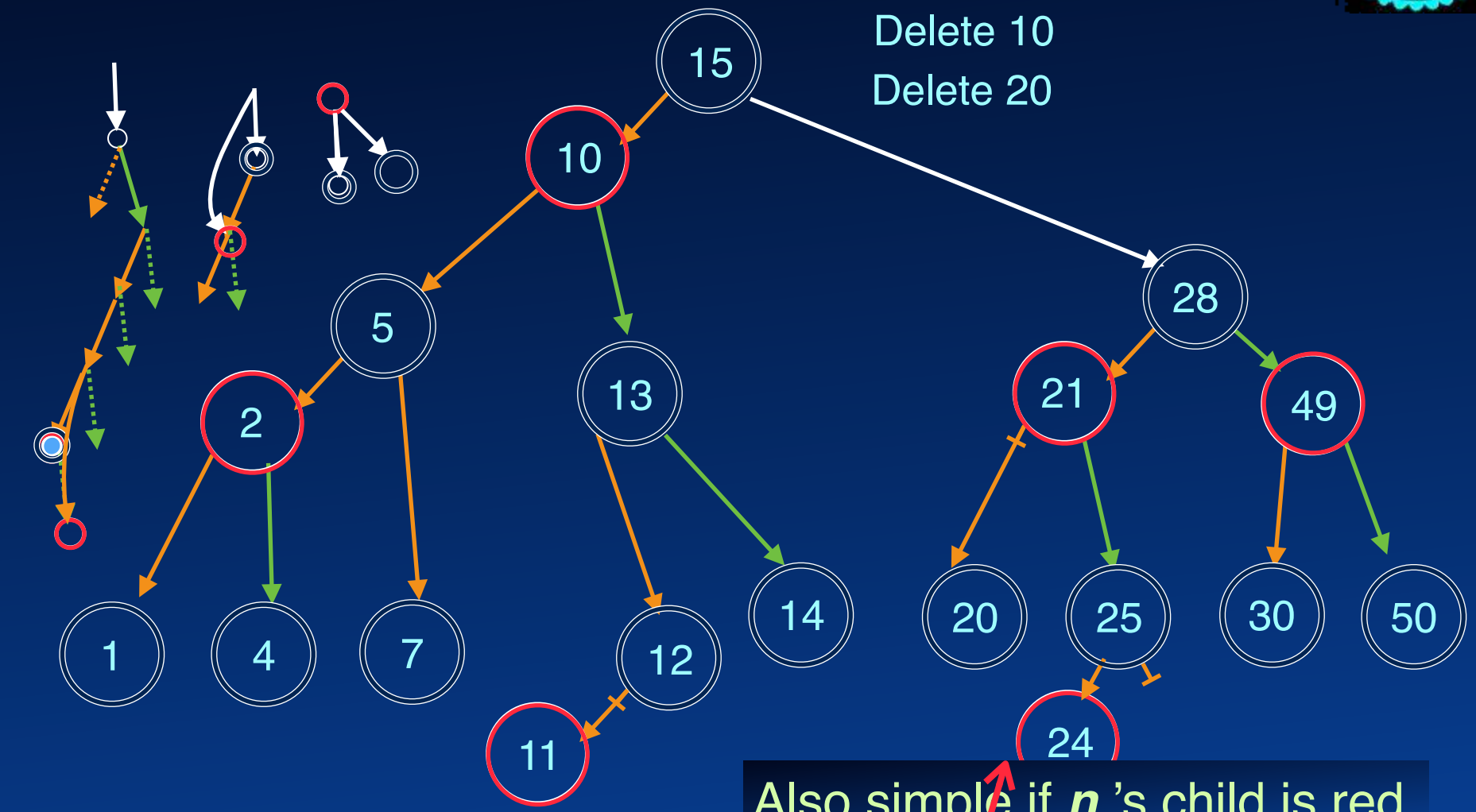
Binar-ized 2-4 Tree



Red-Black Tree Insert



R-B Tree Deletion



Also simple if ***n*** 's child is red.

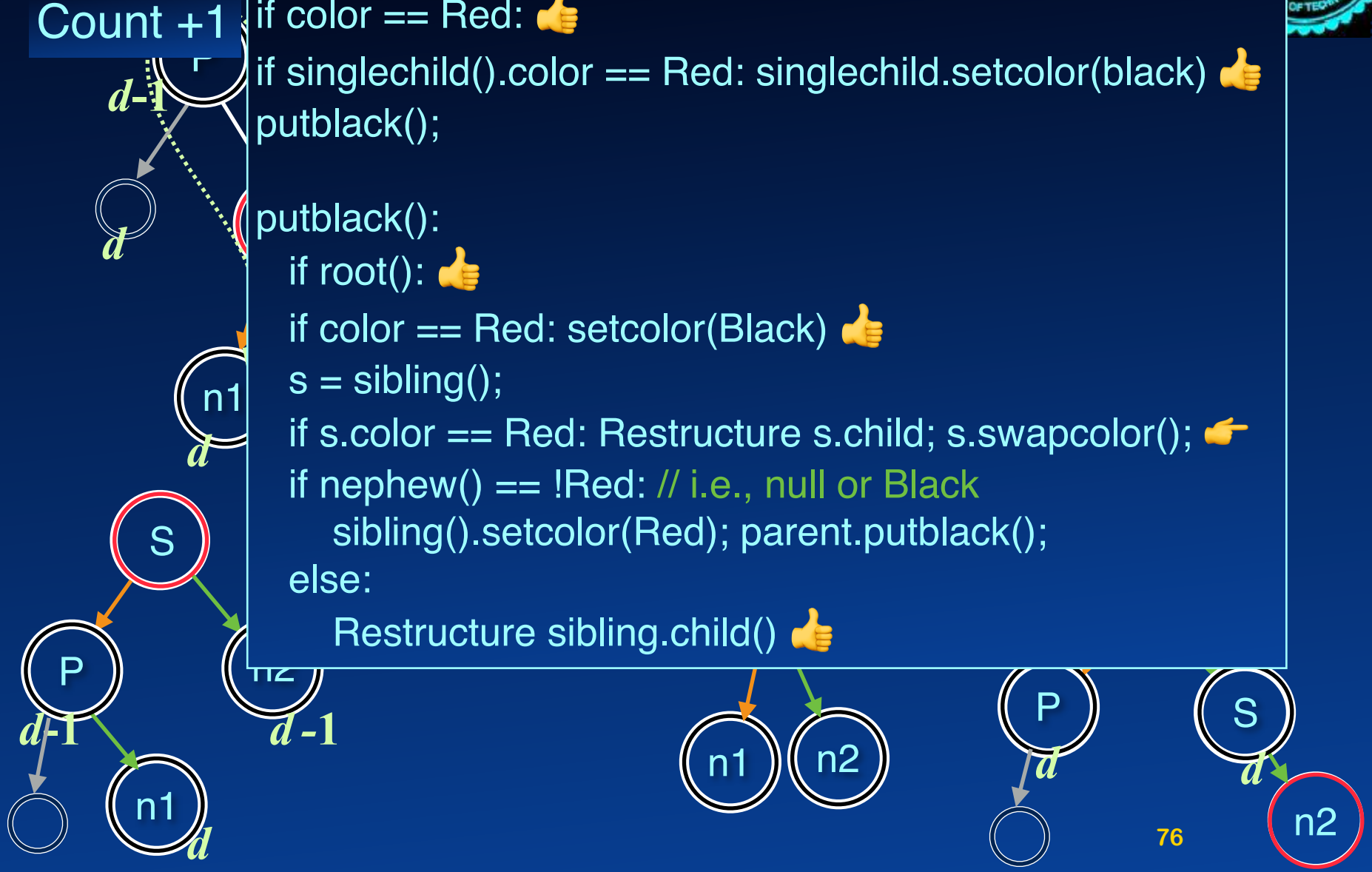
Expunge node ***n*** with @null in child. Simple if ***n*** is red.

Complex Delete Cases

Count +1

```
if color == Red: 👍
if singlechild().color == Red: singlechild.setcolor(black) 👍
putblack();

putblack():
   if root(): 👍
   if color == Red: setcolor(Black) 👍
   s = sibling();
   if s.color == Red: Restructure s.child; s.swapcolor(); 👉
   if nephew() == !Red: // i.e., null or Black
       sibling().setcolor(Red); parent.putblack();
   else:
       Restructure sibling.child() 👍
```



Red-Black Trees

Black Sibling ⇒

No red nephew: Recolor S,P; Fix P.

Red nephew: Restructure P,S,n; Done.

Red Sibling ⇒

Restructure P,S,n

P←Red; S←Black

$d = 3$

Expunge node with @null in node ***n***