Programming with Data Structures

CMPSCI 187

Spring 2016

- **Please find a seat**
  - **Try to sit close to the center (the room will be pretty full!)**
- **Turn off or silence your mobile phone**
- **Turn off your other internet-enabled devices**

Reminders and Topics

- **Midterm 2 is Wed, March 30, 7-9pm**
- **No lecture on Wednesday / Thursday this week**

- Topics of this lecture:
  - **Balancing BST**
  - **Scapegoat Trees**
  - **Storing BST as an Array**

Review from last time

- BST is a binary tree with a special ordering property.
- In-order traversal of BST will visit all nodes in ascending order.
- Searching, insertion, deletion costs are all bounded by `O(h)` where `h` is the tree height.
- Assuming the tree is balanced, the height is approximately `log N`, therefore searching, insertion, deletion all cost `O(log N)` for a balanced BST.

Comparing BST to Linear List

Assume N elements stored in each and the BST is balanced

	BST	Sorted Array	Linked List
`isEmpty`		O(1)	O(1)
`getNext`		O(1)	O(1)
`contains`		$O(\log_2 N)$	$O(N)$
`get`			
Find		$O(\log_2 N)$	
Process		O(1)	
Total		$O(\log_2 N)$	
`add`			
Find		$O(\log_2 N)$	
Process		$O(N)$	
Total		$O(N)$	
`remove`			
Find		$O(\log_2 N)$	
Process		$O(N)$	
Total		$O(N)$	

Comparing BST to Linear List

Assume N elements stored in each and the BST is balanced

	BST	Sorted Array	Linked List
`isEmpty`		O(1)	O(1)
`getNext`		O(1)	O(1)
`contains`		O($\log_2 N$)	O(N)
`get`			
Find		O($\log_2 N$)	O(N)
Process		O(1)	O(1)
Total		O($\log_2 N$)	O(N)
`add`			
Find		O($\log_2 N$)	O(N)
Process		O(N)	O(*1*)
Total		O(N)	O(N)
`remove`			
Find		O($\log_2 N$)	O(N)
Process		O(N)	O(1)
Total		O(N)	O(N)

Comparing BST to Linear List

Assume N elements stored in each and the BST is balanced

	BST	Sorted Array	Linked List
`isEmpty`	O(1)	O(1)	O(1)
`getNext`	O(1)	O(1)	O(1)
`contains`	$O(\log_2 N)$	$O(\log_2 N)$	$O(N)$
`get`			
Find	$O(\log_2 N)$	$O(\log_2 N)$	$O(N)$
Process	O(1)	O(1)	O(1)
Total	$O(\log_2 N)$	$O(\log_2 N)$	$O(N)$
`add`			
Find	$O(\log_2 N)$	$O(\log_2 N)$	$O(N)$
Process	O(1)	$O(N)$	$O(1)$
Total	$O(\log_2 N)$	$O(N)$	$O(N)$
`remove`			
Find	$O(\log_2 N)$	$O(\log_2 N)$	$O(N)$
Process	O(1)	$O(N)$	O(1)
Total	$O(\log_2 N)$	$O(N)$	$O(N)$

Clicker Question #1

What are the costs of **`get, add, remove`** for an **<u>unsorted</u>** array? Assume these methods all take an element (NOT index) as the argument.

(a) O(N), O(N), O(N)

(b) O(1), O(1), O(1)

(c) O(N), O(1), O(1)

(d) O(1), O(1), O(N)

(e) O(N), O(1), O(N)

Answer on next slide

Clicker Question #1

What are the costs of **`get, add, remove`** for an **<u>unsorted</u>** array? Assume these methods all take an element (NOT index) as the argument.

(a) O(N), O(N), O(N)

(b) O(1), O(1), O(1)

(c) O(N), O(1), O(1)

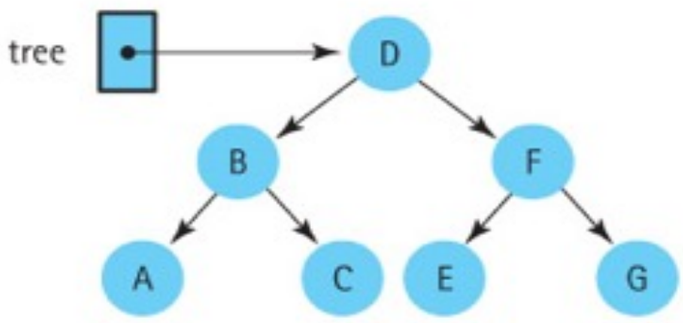
(d) O(1), O(1), O(N)

(e) O(N), O(1), O(N)

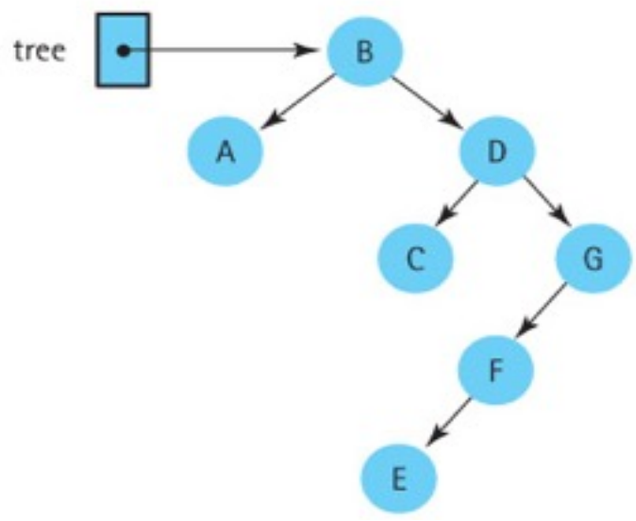
What is Balance?

- There are many ways to define **balance**. For example, standing at any node, we can define balance as the difference between the height of its right subtree vs. the height of the left subtree.

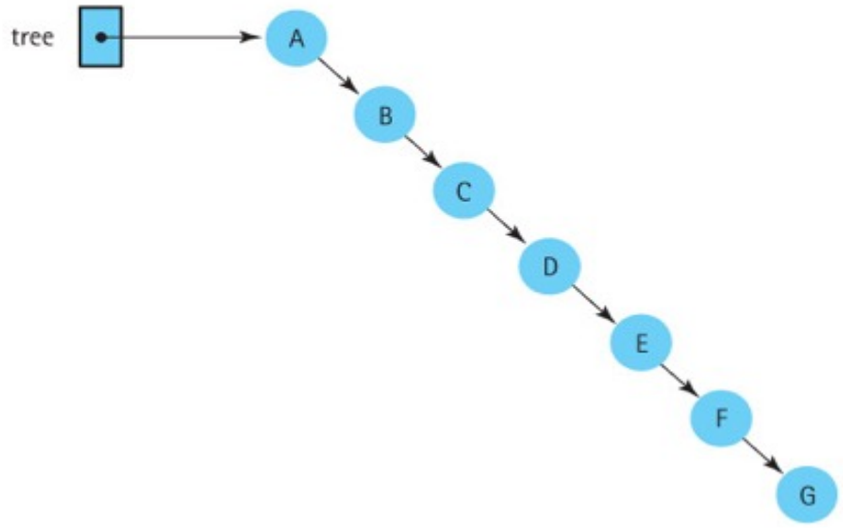
```
balance(n) = height(n.right) - height(n.left)
```



perfectly balanced



Not very balanced



Very unbalanced

What is Balance?

- Another way to think about balance is to look at how the tree height **`h`** is related to the number of nodes **`N`**. Ideally we want to **guarantee** that `h = O(log N)`.

- Because insertion and deletion can cause the BST to become unbalanced, we need to perform extra steps to restore the balance (or preserve the height guarantee).

- Such trees are called **self-balancing trees** (a.k.a. **height-balanced trees**).

Examples of Self-Balancing Trees

- **AVL tree**
  Guarantees that at any node, the height difference between its left and right subtrees is no more than 1.

- **Red-black tree**
  Colors each node with red/black (requires extra bit per node), and enforces color compatibility rules during insertion / deletion to maintain height guarantee.

- **Scapegoat tree**
  When tree becomes unbalanced, find a ‘scapegoat’ and re-balances the sub-tree rooted at the scapegoat.

- **2-3 tree** (not binary tree)
  Each interior node either stores 1 data element and has 2 children, or stores 2 data elements and has 3 children.

DJW's approach to balancing

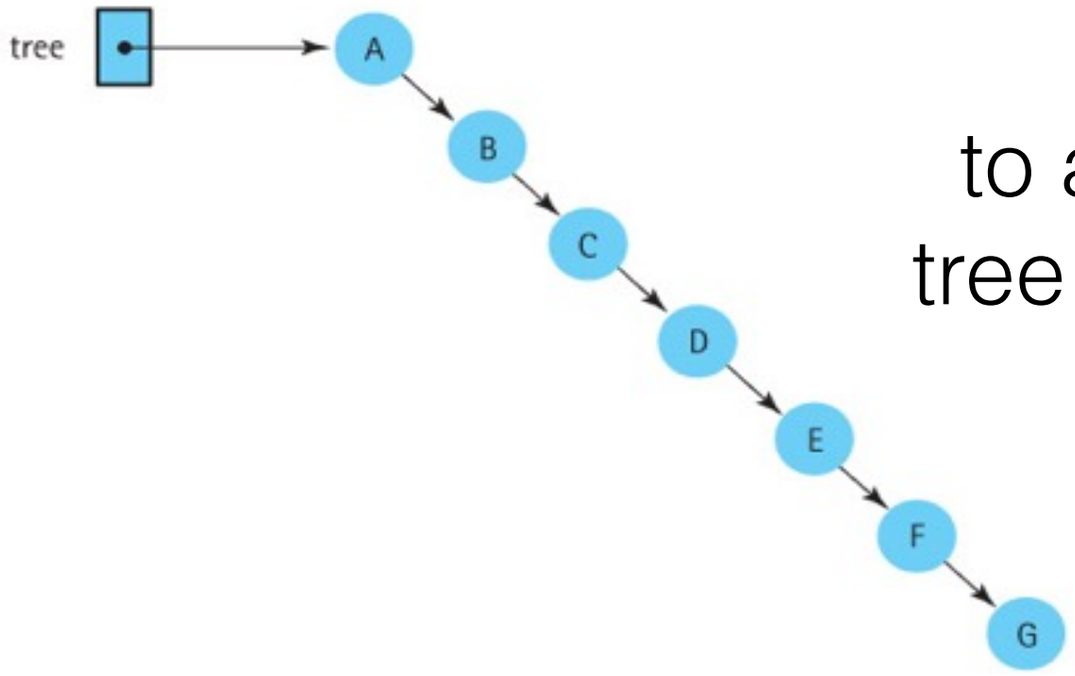
- The user manually calls a `balance()` method from time to time (at the user's discretion) to rebuild the tree into a balanced version.

- This is NOT done automatically so it doesn't qualify as a self-balancing tree.

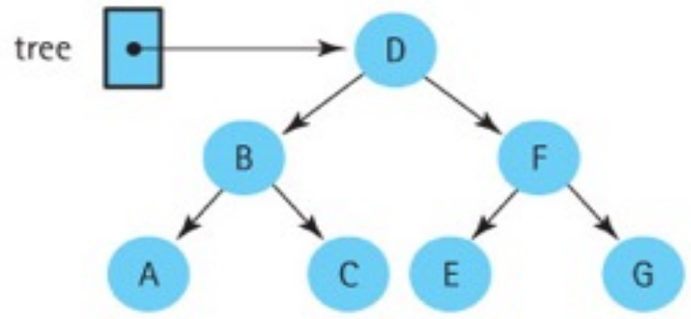
- Still, the `balance()` method is very useful and you will need it for your project 8.

DJW's approach to balancing

- We want to rebuild a tree like this:



to a balanced
tree like this —>



- To do so, we will:
  1. Save the nodes (sorted) into an array
  2. Build a new, balanced tree from the sorted array

DJW's approach to balancing

- How do we 'export' all nodes into an array in sorted (ascending) order?

  Do an in-order traversal and save the result to an array (or a queue if you want)!

  - How do we build a **balanced** tree from a sorted array? For example:

  [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]

It's similar to a binary search: you start from the entire array, pick the middle element to make it a tree node, then recurse on the left and right sub-arrays.

```
// assume nodes stores the sorted elements
// start the recursion by:
//      tree = new BST();
//      insertTree(0, nodes.length-1);

private void insertTree (int lower, int upper) {
    if (lower == upper)
        tree.add(nodes[lower]);
    else if (lower + 1 == upper) {
        tree.add(nodes[lower]);
        tree.add(nodes[upper]);
    } else {
        int mid = (lower + upper)/2;
        tree.add(nodes[mid]);
        insertTree (lower, mid - 1);
        insertTree (mid + 1, upper);
    }
}
```

```
// an improved version that directly creates
// tree nodes without using the tree.add method
// start the recursion by:
//   tree.root = sortedArray2BST(0, nodes.length-1);

BSTNode<T> sortedArray2BST (int lower, int upper) {
     if (lower > upper)
          return null;
     int mid = (low + high)/2;
     BSTNode<T> nn = new BSTNode<T>(nodes[mid]);
     nn.setLeft (sortedArray2BST(lower, mid - 1));
     nn.setRight(sortedArray2BST(mid + 1, upper));
     return nn;
}
```

Clicker Question #2

```
BSTNode<T> sortedArray2BST (int lower, int upper) {
    if (lower > upper)
        return null;
     int mid = (low + high)/2;
     BSTNode<T> nn = new BSTNode<T>(nodes[mid]);
     nn.setLeft (sortedArray2BST(lower, mid - 1));
     nn.setRight(sortedArray2BST(mid + 1, upper));
     return nn;
 }
```

What's the cost of the above method? Assume the sorted array has N elements. Remind you that you start the call by:

`sortedArray2BST(0, N-1);`

(a) O(1)

(b) O(log N)

(c) O(N)

(d) $O((\log N)^2)$

(e) O(N log N)

Answer on next slide

Clicker Question #2

```
BSTNode<T> sortedArray2BST (int lower, int upper) {
    if (lower > upper)
        return null;
     int mid = (low + high)/2;
     BSTNode<T> nn = new BSTNode<T>(nodes[mid]);
     nn.setLeft (sortedArray2BST(lower, mid - 1));
     nn.setRight(sortedArray2BST(mid + 1, upper));
     return nn;
 }
```

What's the cost of the above method? Assume the sorted array has N elements. Remind you that you start the call by:

`sortedArray2BST(0, N-1);`

(a) O(1)

(b) O(log N)

(c) O(N)

(d) $O((\log N)^2)$

(e) O(N log N)

The `balance()` method

- To summarize, the **`balance()`** method first performs an in-order traversal to save all nodes into an array; then it calls the recursive **`insertTree`** (or **`sortedArray2BST`**) to build a balanced tree from the sorted array.

- Instead of balancing the whole tree, you can choose to **balance a sub-tree** (this is a step you will need for scapegoat tree).

Scapegoat Tree

- The name comes from the common wisdom that, when something goes wrong, the first thing people tend to do is find someone to blame (the scapegoat). Once blame is established, we can leave the scapegoat to fix the problem.

Scapegoat Tree

Main ideas:

- Conditions that must be satisfied:

  1. `q/2 <= N <= q` `// upper bound cond.`
  2. `h <=` $\log_{3/2} q$ `// height cond.`

  where `N` is the # of nodes and `q` is called upper bound.

- If insertion causes violation, follow ancestors to find a scapegoat node `w` — rebuild subtree rooted at `w`.

- In removal causes violation, rebuild the entire tree.

Scapegoat Tree

Main ideas:

- Conditions that must be satisfied:

  1. `q/2 <= N <= q` `// upper bound cond.`
  2. $h \le \log_{3/2} q$ `// height cond.`

  where `N` is the # of nodes and `q` is called upper bound.

- Assume the conditions are met at all times, we have

  $h \le (\log_{3/2} q) \le (\log_{3/2} 2N) < (2 + \log_{3/2} N)$

  Thus $h = O(\log N)$ and this is what we wanted.

Example: Insertion

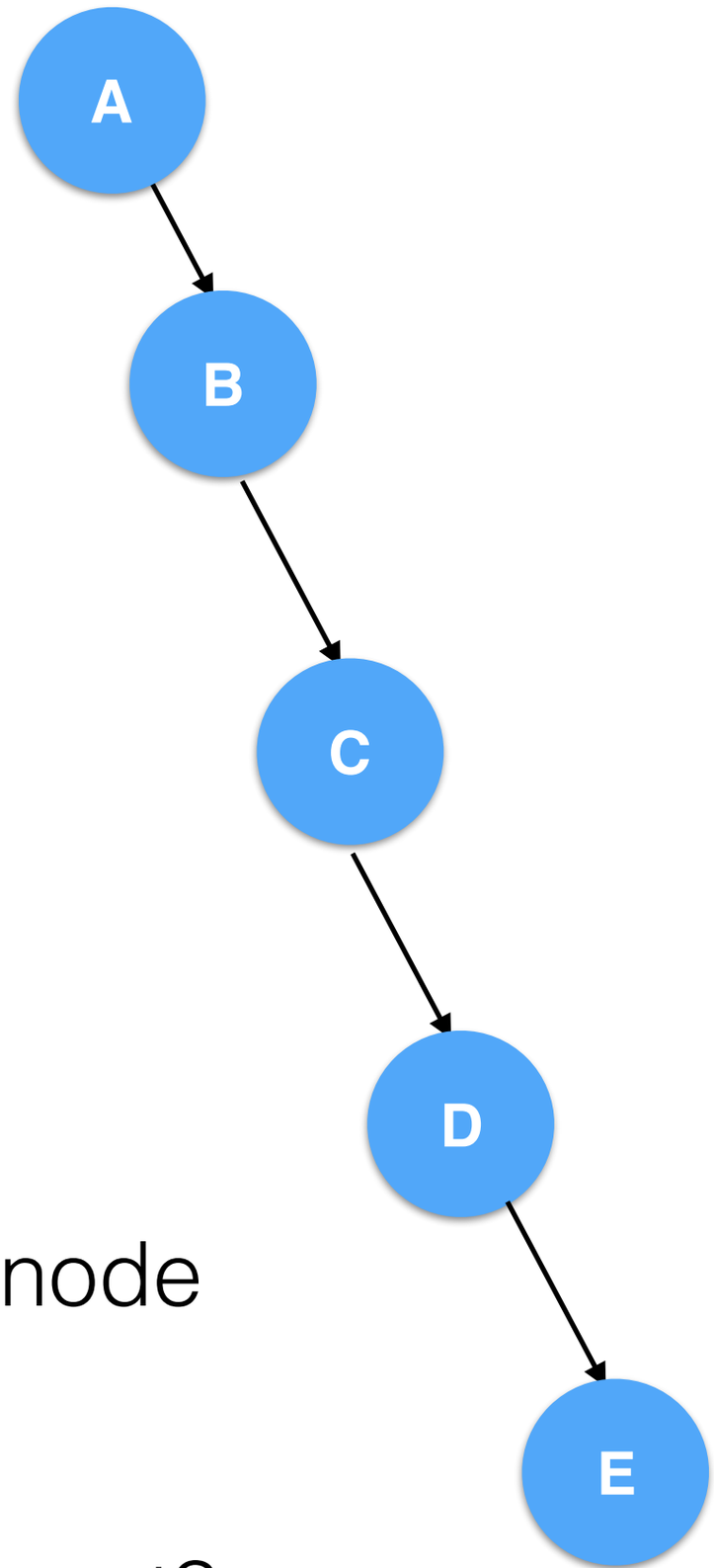
Here we insert 5 elements in ascending order: a worst-case for standard (non-balancing) BST.
During insertion, q increments along with N.

operation	N	q	height	$\log_{3/2}(q)$
insert A	1	1	0	0
insert B	2	2	1	1.79
insert C	3	3	2	2.70
insert D	4	4	3	3.42
insert E	5	5	**4**	**3.97**

- Inserting E violated height condition (`h <= log`$_{3/2}$ `q`)

- Now find scapegoat node `w` (**on the path from the root to the newly added node**) such that:

$$\frac{\texttt{sizeOfSubtree(w.child)}}{\texttt{sizeOfSubtree(w)}} > 2/3$$



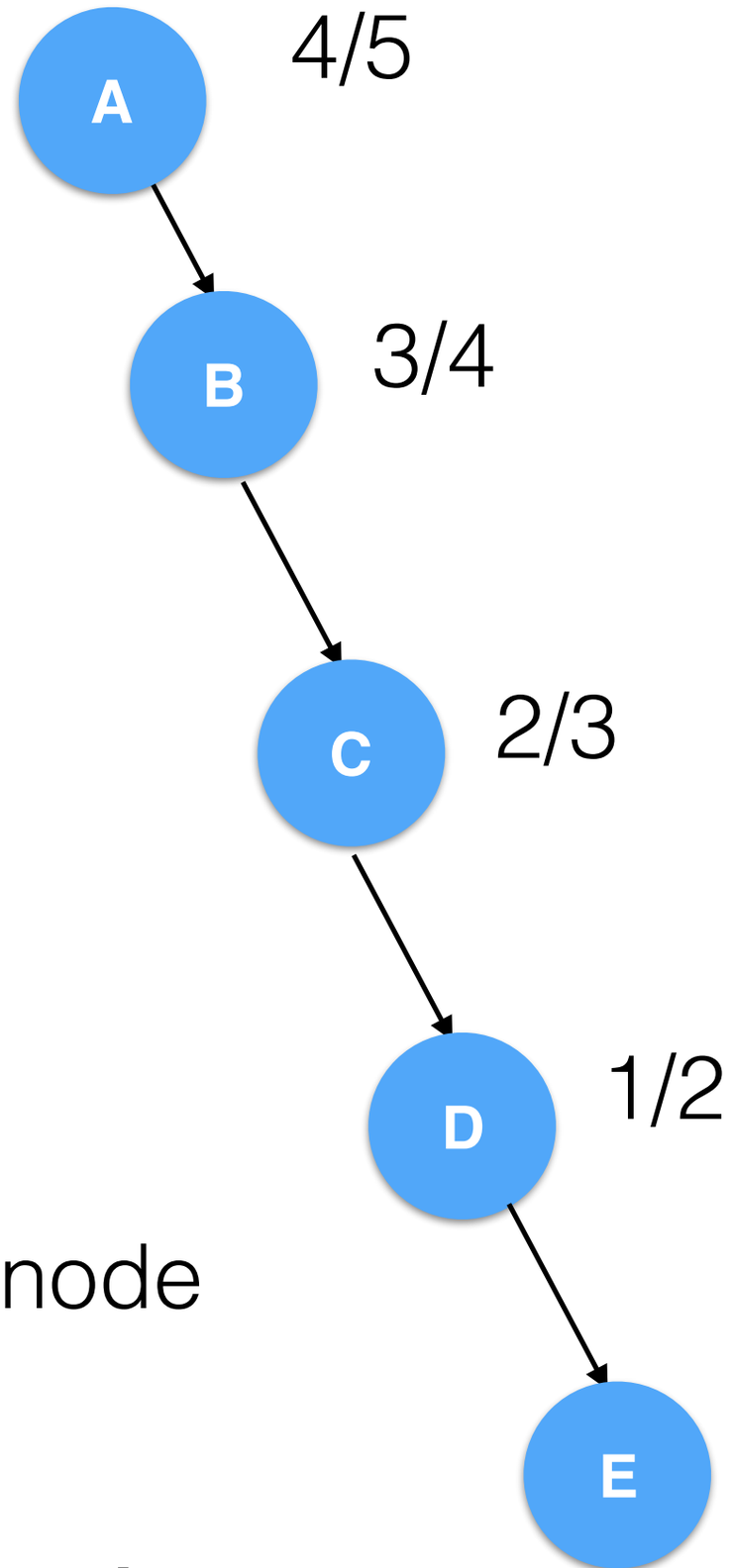
It can be shown that such a scapegoat node must exist (proof using contradiction).

In this example, which one is the scapegoat?

- Inserting E violated height condition (`h <= log3/2 q`)

- Now find scapegoat node `w` (**on the path from the root to the newly added node**) such that:

$$\frac{\texttt{sizeOfSubtree(w.child)}}{\texttt{sizeOfSubtree(w)}} > 2/3$$



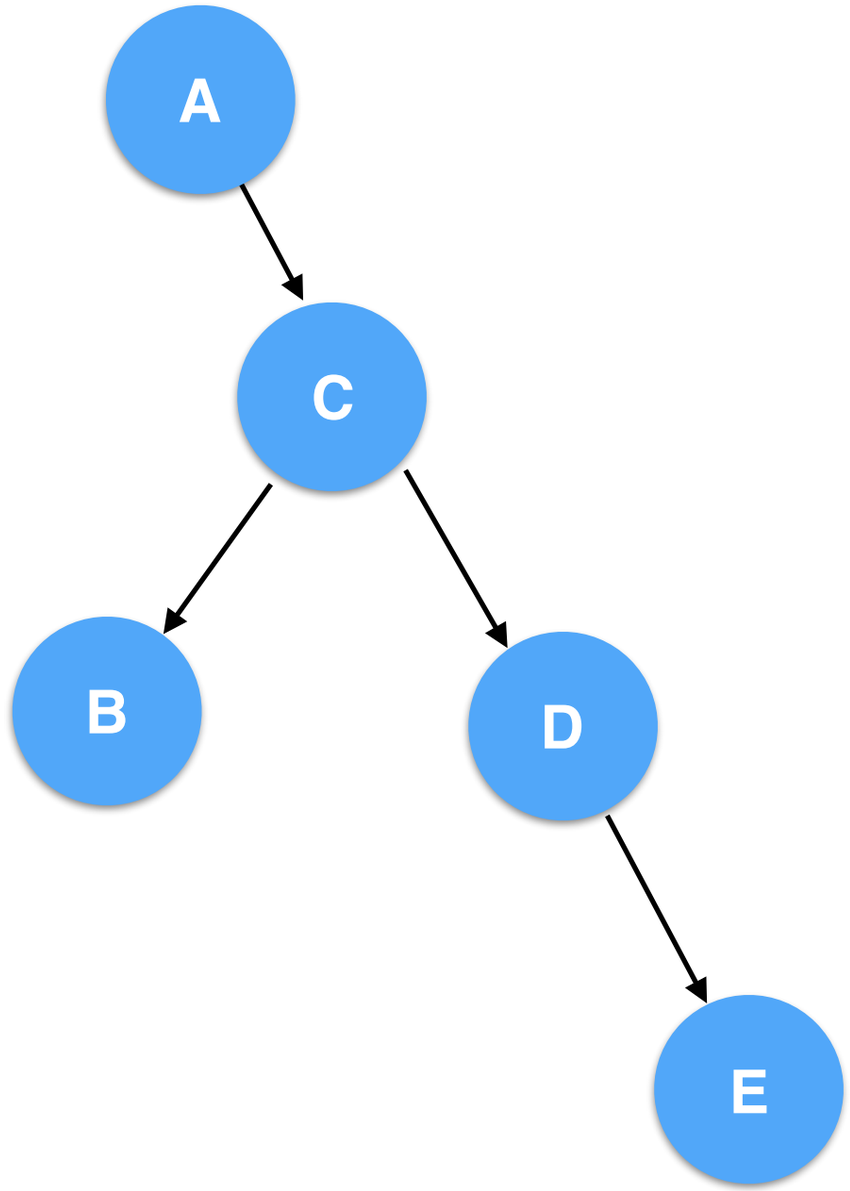
It can be shown that such a scapegoat node must exist (proof using contradiction).

Here B is the scapegoat: since `3/4 > 2/3`

- Once we identified the scapegoat, we rebuild the subtree rooted at the scapegoat, using the same `balance()` method we just covered.

- This is guaranteed to fix the violated condition (proof is omitted here).

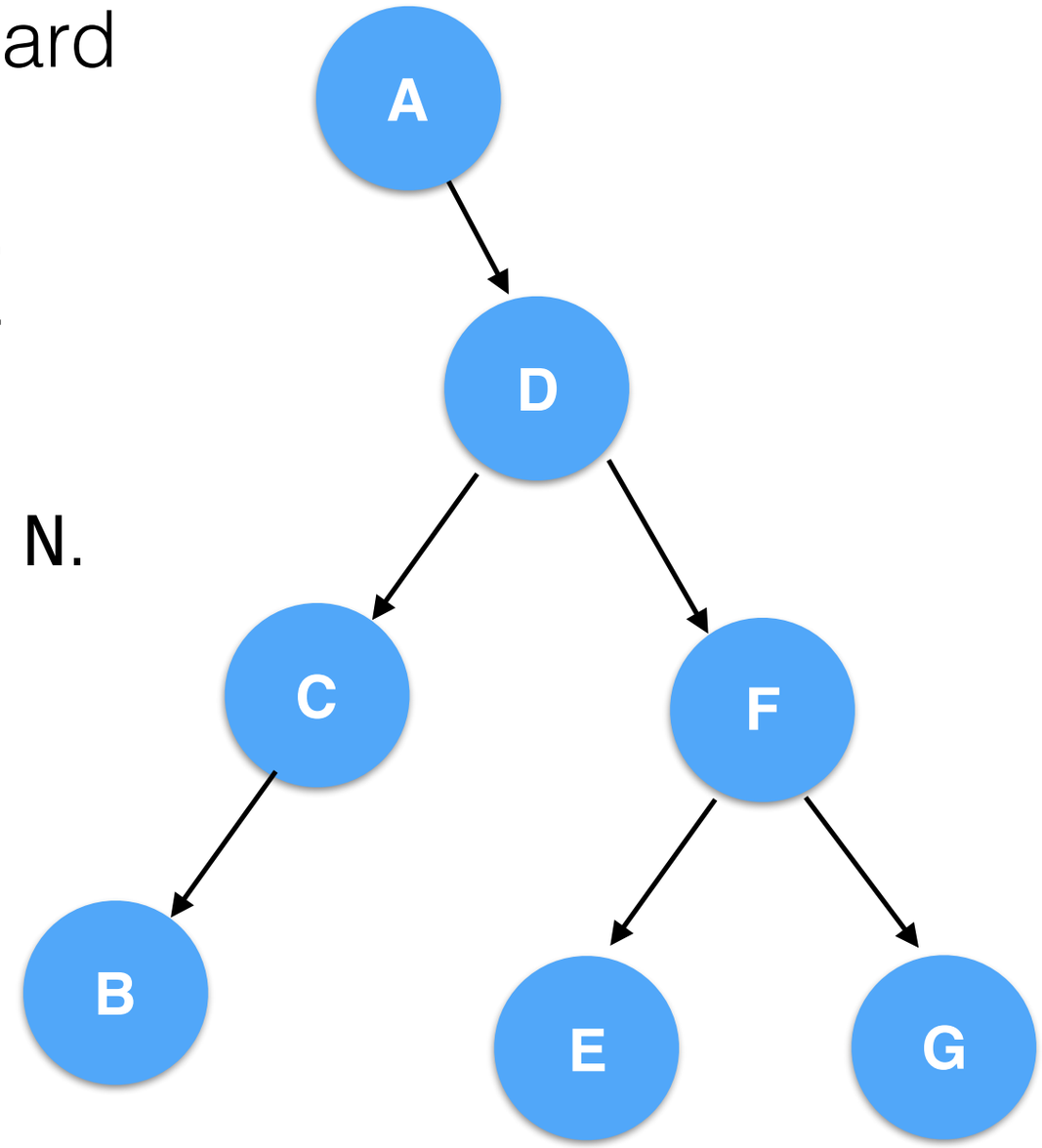
- Let's verify if it's true for this example. After rebuilding, what's the tree height?



Example: Removal

We perform removal by following standard BST removal. At each removal, we decrement `N` while `q` remains the same.

Then check if `q > 2 N`. If so, the upper bound condition is violated, we fix it by rebuilding the **entire** tree, then set `q = N`.



operation	N	q	height	$\log_{3/2}(q)$
remove G	6	7	3	4.8
remove F	5	7	3	4.8
remove E	4	7	3	4.8
remove D	**3**	**7**	2	4.8

Scapegoat Tree

Summary

- Keep track of N (# of nodes) and q (upper bound).
- After inserting a new node, check the height condition. If violated, find scapegoat and rebuild the subtree rooted at the scapegoat.
- After deleting a node, check the upper bound condition, if violated, rebuild the entire tree.
- Although re-building a tree (either sub-tree or entire tree) can take `O(N)` time, it occurs sparingly (not every time), and it can be shown that the amortized cost of insertion and deletion are both `O(log N)`.

Storing a Binary Tree in an Array

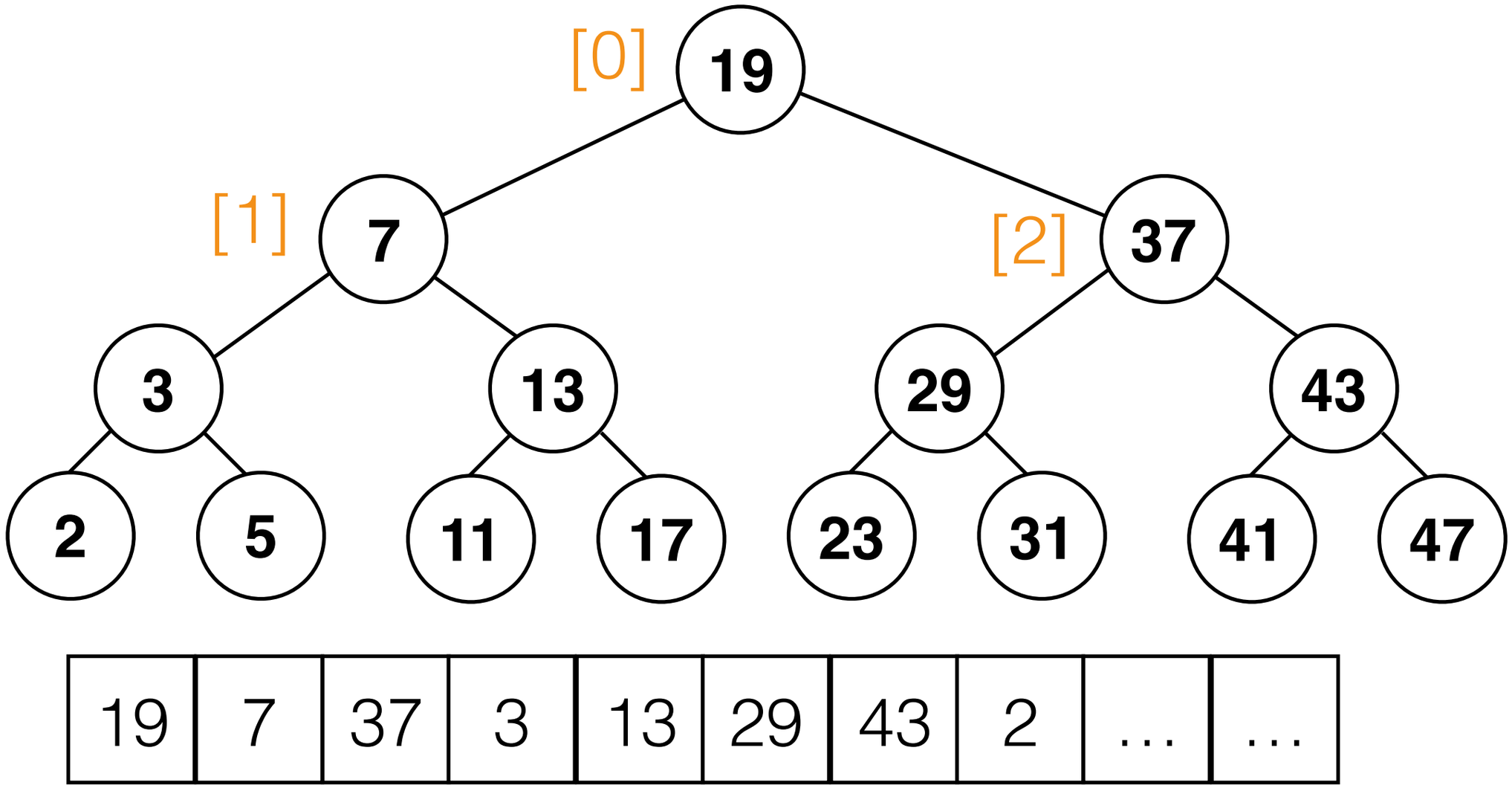
- We can store a List of elements either in a linked structure or an array. Similarly, we can store a binary tree in a linked structure (what we've learned so far), or we can store it in an array!

- As you will see, the array representation has a pre-determined indexing scheme so it eliminates the the left and right child pointers. **This saves precious memory space**!

- The downside is that it's not very flexible and if the tree is very sparse you can waste a lot of memory.

Storing a Binary Tree in an Array

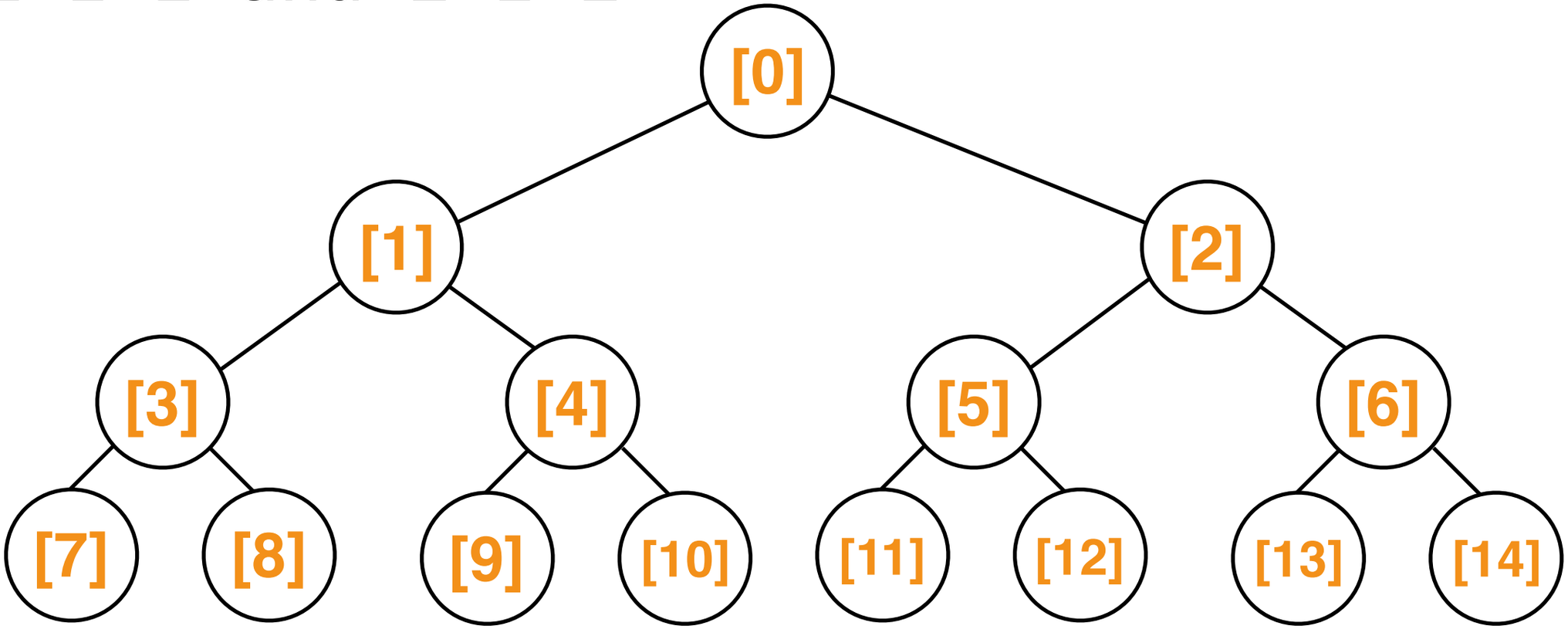
- Imagine we have a full tree as below. We store the nodes into an array in **breath-first order**, where the root node is stored at index [0], its left child at [1], right child at [2], and so on.



Storing a Binary Tree in an Array

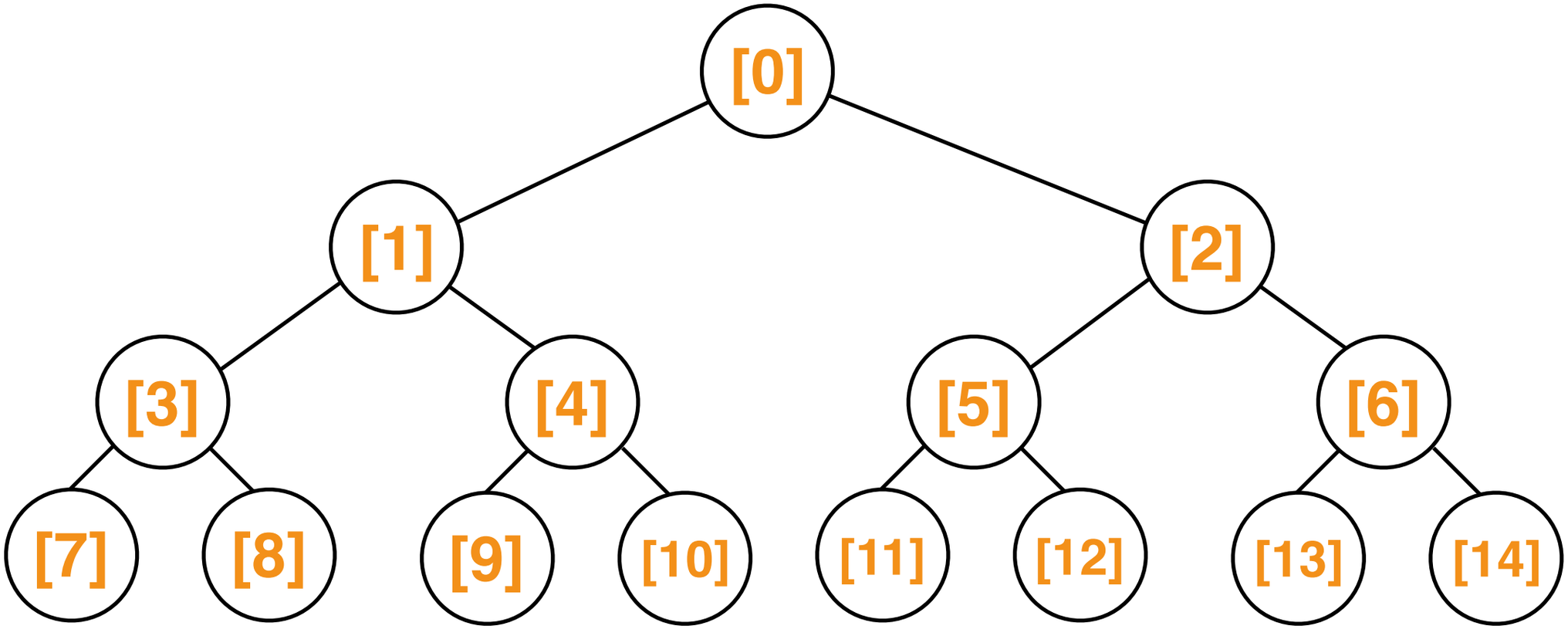
- The index of every node is shown below.
- If a node is stored at index `i`, observe from the patterns below, its two children will be stored at??

`2*i+1 and 2*i+2`

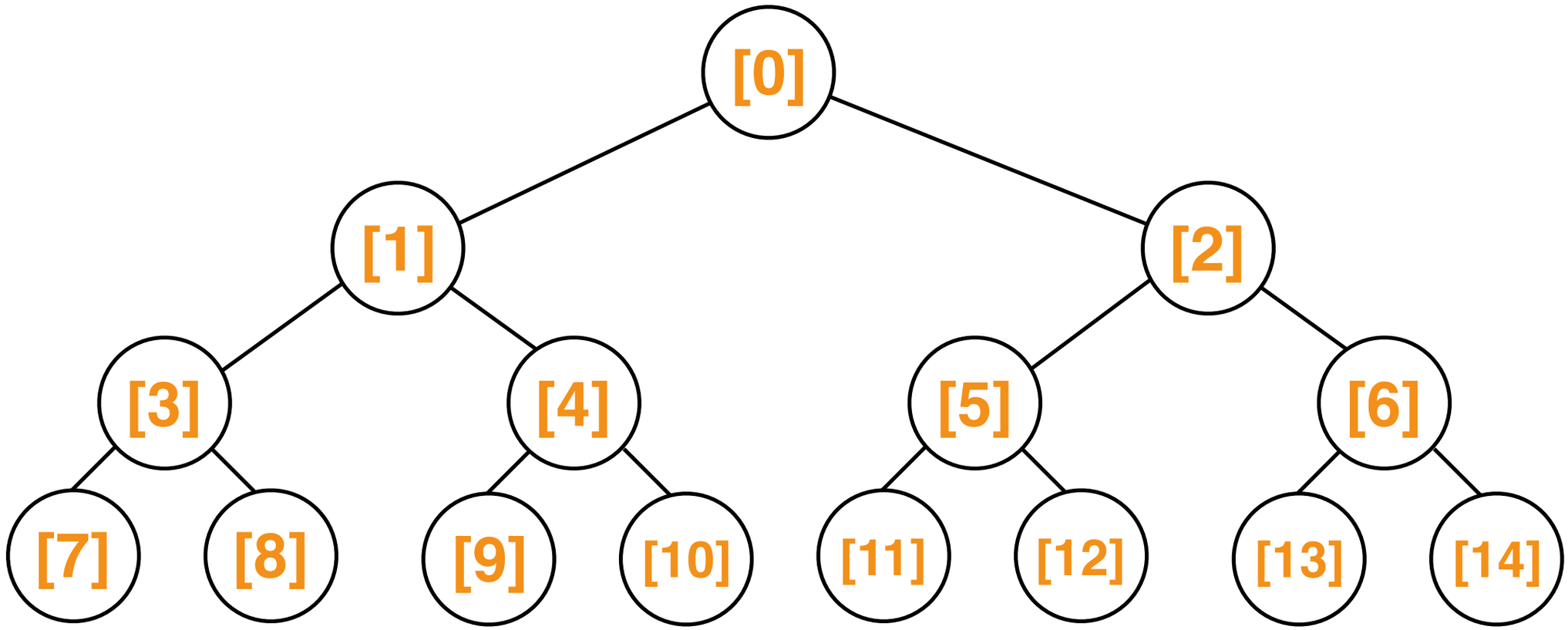


Storing a Binary Tree in an Array

- This indexing scheme is pre-determined, therefore there is no need to store pointers to children, and finding the children of a node involves just some arithmetic calculations.



Clicker Question #3



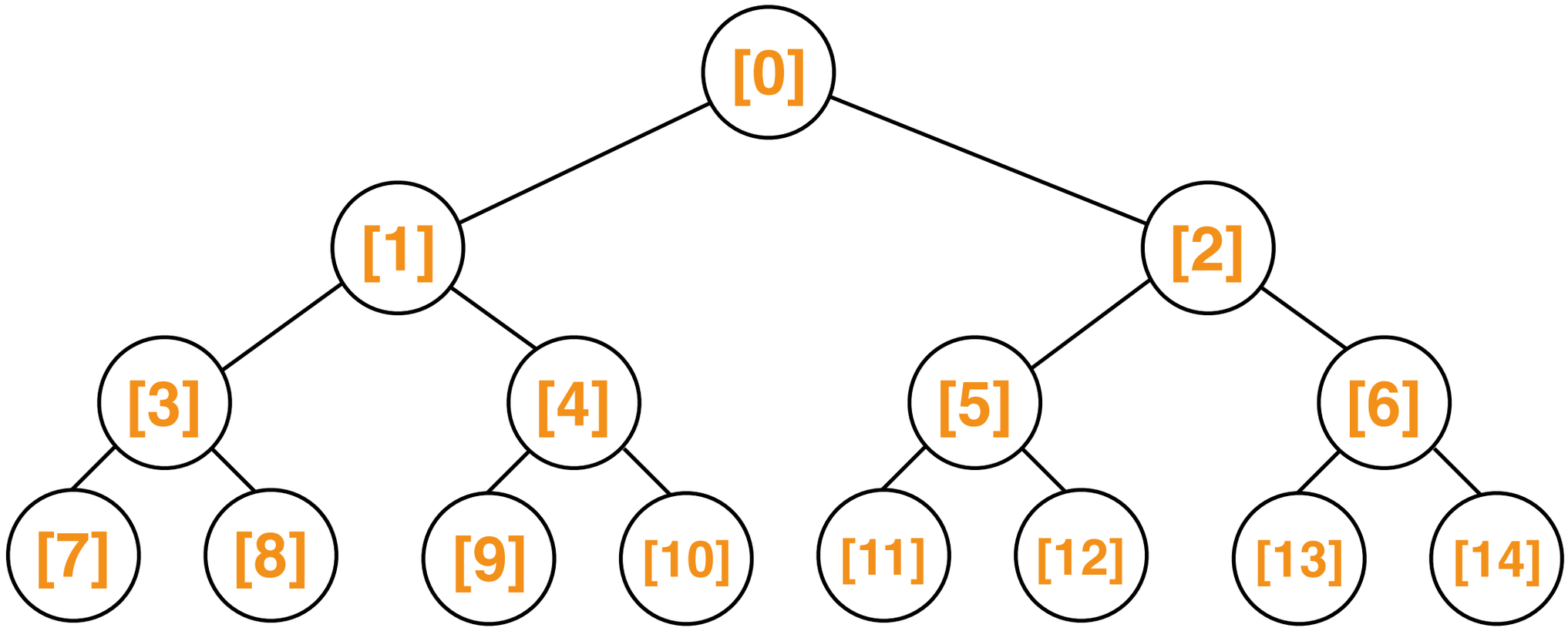
- If a node is stored at index `i`, what would be the index of its parent? (Assume integer division)

```
(a)  i/2        (b) (i+1)/2     (c) (i-1)/2

(d)  (i/2)+1    (e) (i/2)-1
```

Answer on next slide

Clicker Question #3



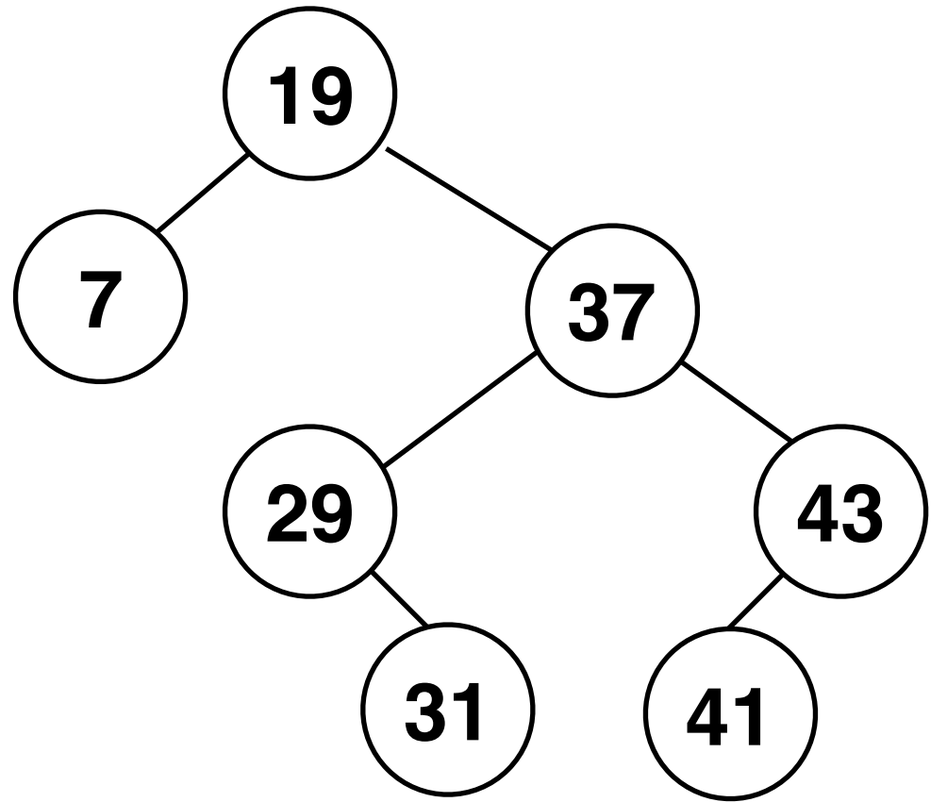
- If a node is stored at index `i`, what would be the index of its parent? (Assume integer division)

(a) `i/2` (b) `(i+1)/2` (c) `(i-1)/2`

(d) `(i/2)+1` (e) `(i/2)-1`

Storing a Binary Tree in an Array

- If a node does not exist, its corresponding array element will be null. If the tree is very sparse, a lot of elements will be null, in this case the array representation can waste a lot of memory storing null.



19	7	37	null	null	29	43	null	null	null	null	null	31	41	null