Lecture 29
TREES

November 18, 2021
Thursday

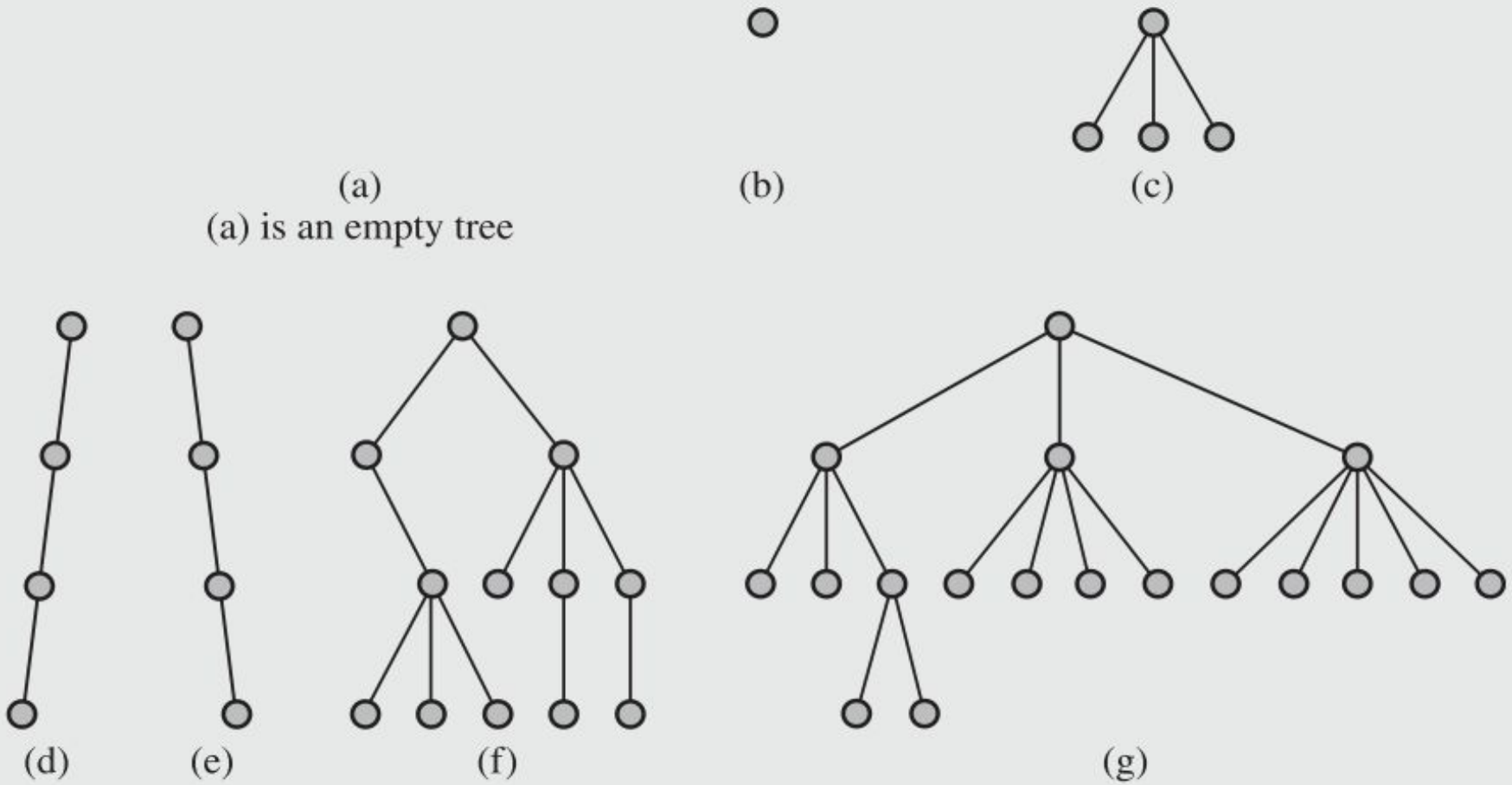
LIMITATION OF LINEAR STRUCTURE

- Linked lists usually provide greater flexibility than arrays, but they are linear structures and so it is difficult to use them to organize a hierarchical representation of objects.

- Stacks and Queues provide some hierarchy but are still flexible to one dimension.

TREE

1. An empty structure is an empty Tree.
2. If $t_1, \ldots, t_k$ are disjointed trees, then the structure whose root has as its children the roots of $t_1, \ldots, t_k$ is also a tree.
3. Only structures generated by rules 1 and 2 are trees.


(a)
(b)
(c)
(a) is an empty tree
(d)
(e)
(f)
(g)

TREES | TERMINOLOGIES

- Unlike natural Trees, these trees are depicted upside down.
- Elements of Tree are called Nodes.
- The top most node is called the “root”.
  - Root has no parents.
  - Can have only Childrens.
- Leaves on the other hand
  - Only have parents.
  - Have no children (their children are empty structure).

TREE | TERMINOLOGIES

- Connection between two nodes, is known as Edge also called Arc.
- Parent Node: the converse notion of a child.
- Siblings: Nodes with the same parent.
- Path: Each node has to be reachable from the root through a unique sequence of edges.
- The number of edges in a path is called the length of the path.

TREE | TERMINOLOGIES

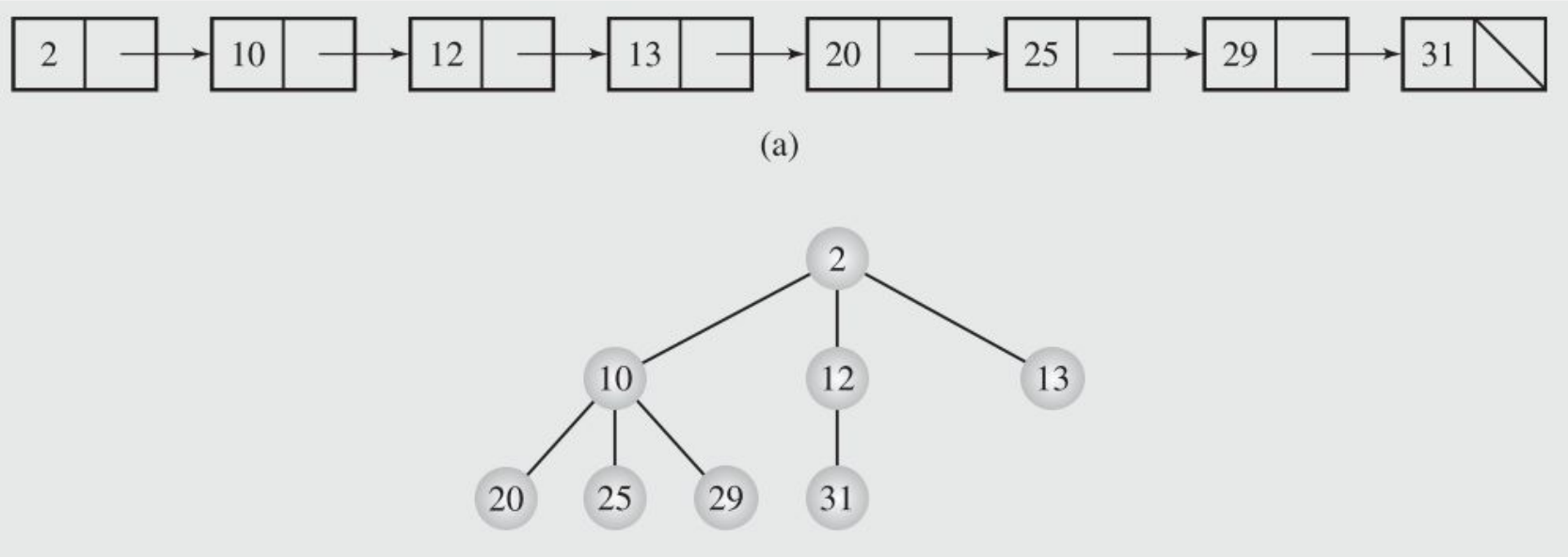
- The level of a node is the length of the path from the root to the node plus 1, which is the number of nodes in the path.
- The height of a non- empty tree is the maximum level of a node in the tree.
  - An empty tree has height of 0.
  - A single node has height of 1.
    i. This is the only case when a node is both **root** and a **leaf.**
- The level of a node must be between 1 (the level of the root) and the height of the tree

TREE | TERMINOLOGIES

The definition of a tree does not impose any condition on the number of children of a given node. This number can vary from 0 to any integer. In hierarchical trees, this is a welcome property.

TREE | TERMINOLOGIES

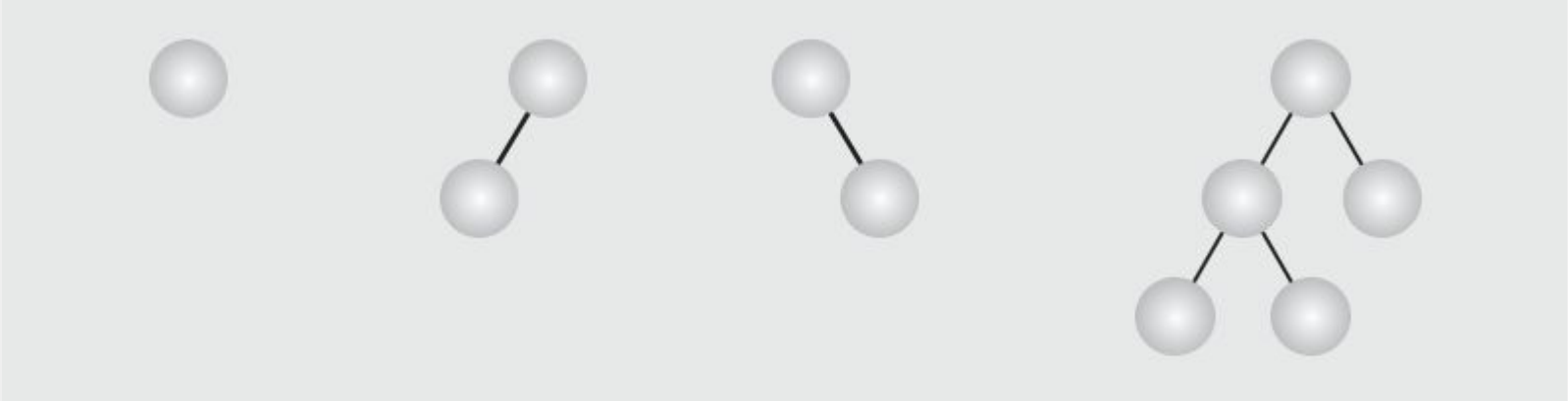
Linked list required Linear Search. Converting a list into Tree offers benefits



TREE | TERMINOLOGIES

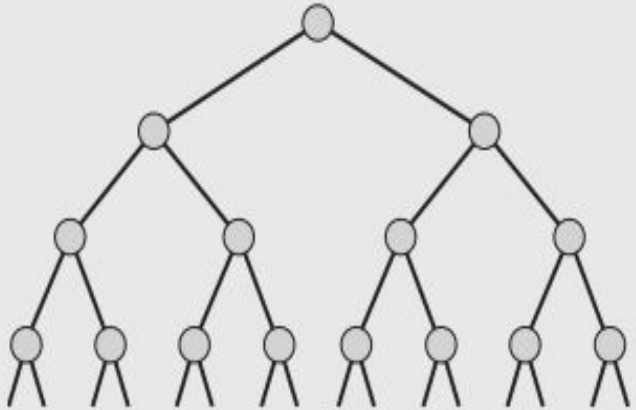
A binary tree is a tree whose nodes have two children (possibly empty), and each child is designated as either

a left child or a right child.



TREE | TERMINOLOGIES

- The root is at level 1.
  - Its children are at level 2.
  - Its grandchildren are at level 3 and so on.
- Generally there are 2^i nodes at level *i + 1.* A tree satisfying this condition is referred to as Complete Tree.
- All nonterminal nodes have both their children, and all leaves are at the same level.



Height	Nodes at One Level	Nodes at All Levels
1	$2^0 = 1$	$1 = 2^1 - 1$
2	$2^1 = 2$	$3 = 2^2 - 1$
3	$2^2 = 4$	$7 = 2^3 - 1$
4	$2^3 = 8$	$15 = 2^4 - 1$
⋮		
11	$2^{10} = 1{,}024$	$2{,}047 = 2^{11} - 1$
⋮		
14	$2^{13} = 8{,}192$	$16{,}383 = 2^{14} - 1$
⋮		
h	2^{h-1}	$n = 2^h - 1$
⋮		

THE BST PROPERTY

A Binary Search Tree (BST) is a binary tree such that:

Every LEFT descendant of a node has key less than that node
Every RIGHT descendant of a node has key larger than that node

Storing multiple copies of the same value in the same tree is avoided.

BINARY SEARCH TREE MOTIVATION

OPERATION	SORTED ARRAY	UNSORTED LINKED LIST
SEARCH	O(log(n))	O(n)
DELETE	O(n)	O(n)
INSERT	O(n)	O(1)

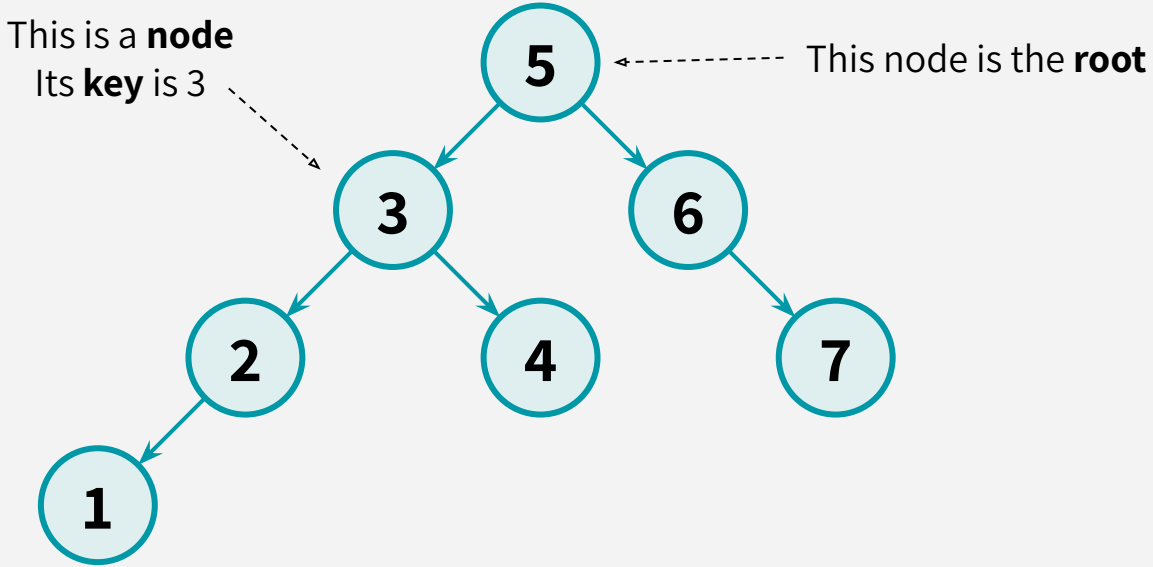
(Balanced) Binary Search Trees can give us the best of both worlds!

BINARY SEARCH TREE MOTIVATION

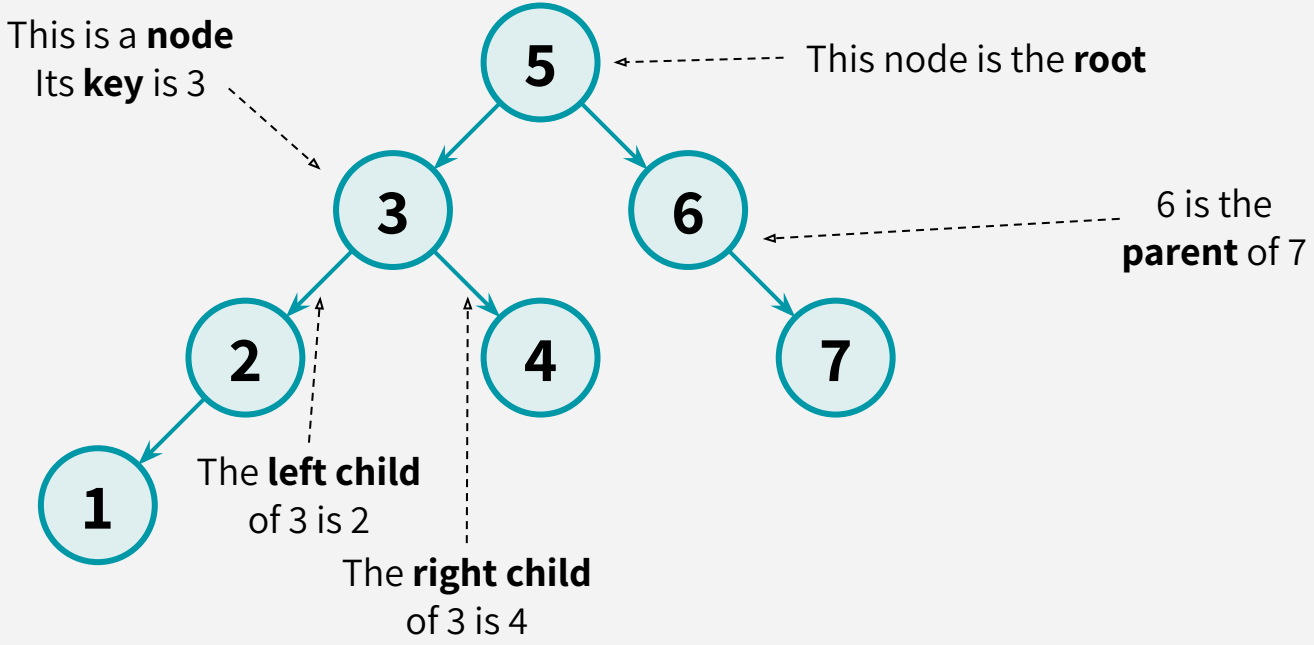
OPERATION	SORTED ARRAY	UNSORTED LINKED LIST	BST (WORST CASE)	BST (BALANCED)
SEARCH	O(log(n))	O(n)	O(n)	O(log(n))
DELETE	O(n)	O(n)	O(n)	O(log(n))
INSERT	O(n)	O(1)	O(n)	O(log(n))

(Balanced) Binary Search Trees can give us the best of both worlds!

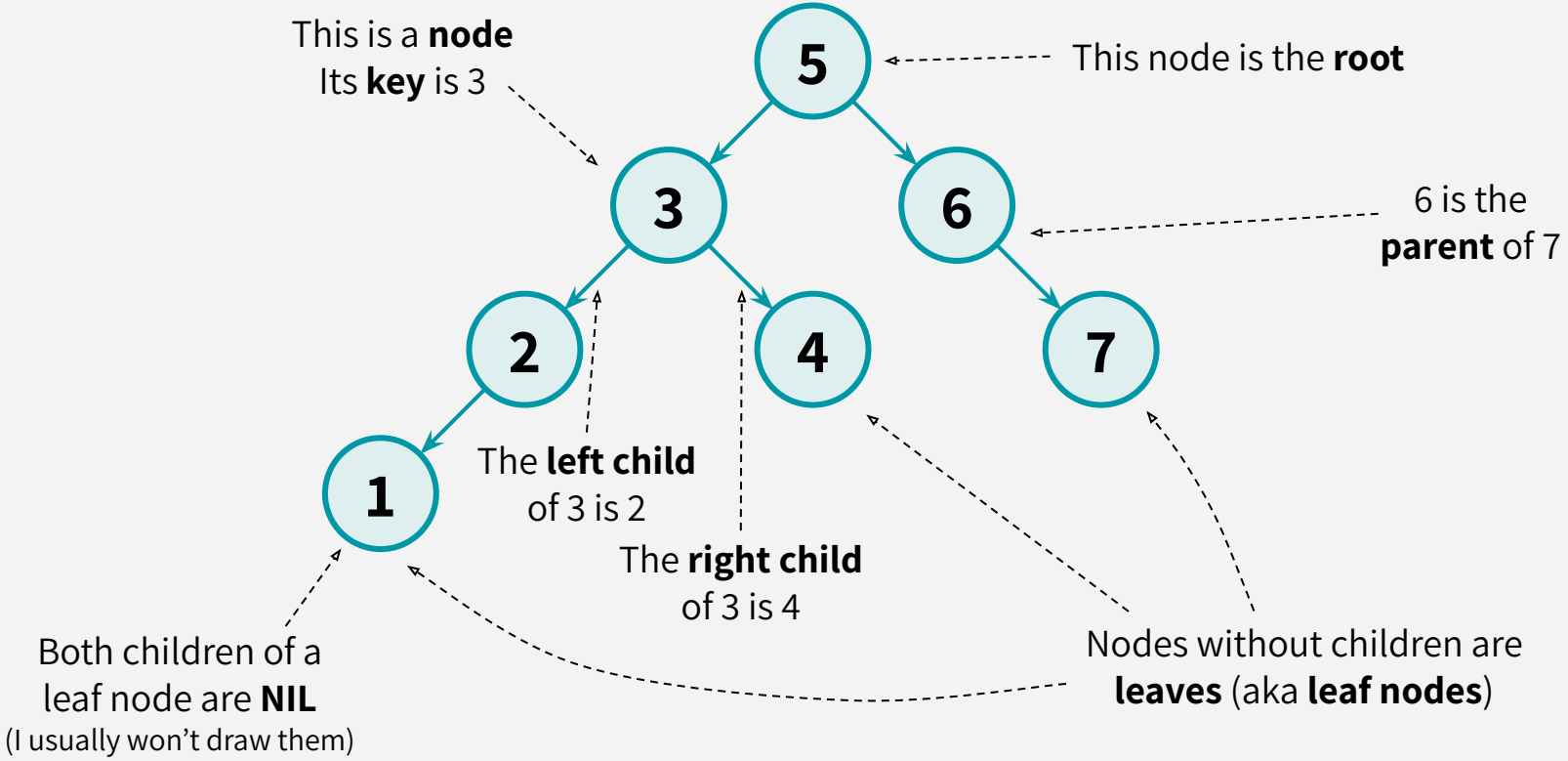
BINARY TREE TERMINOLOGY



BINARY TREE TERMINOLOGY



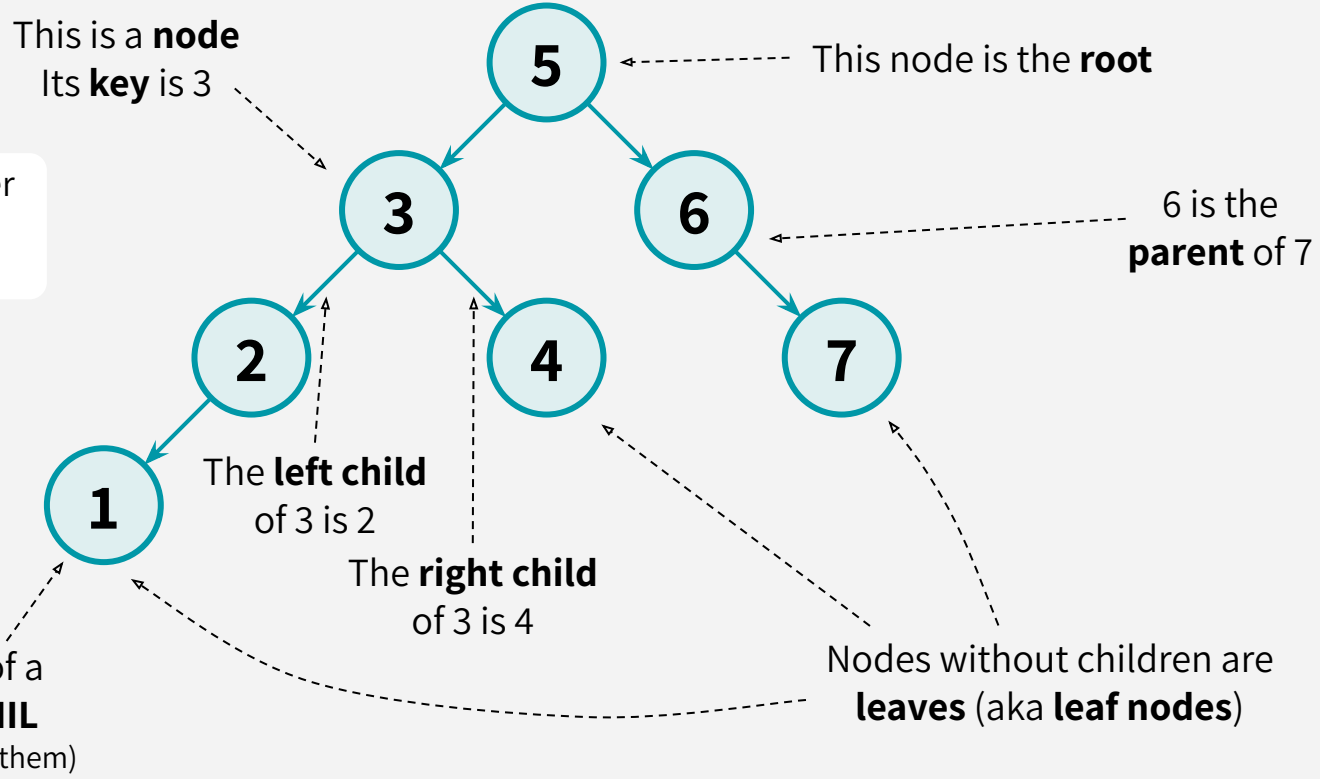
BINARY TREE TERMINOLOGY



BINARY TREE TERMINOLOGY

Each node has two children

Each node has a pointer to its left child, right child, and parent



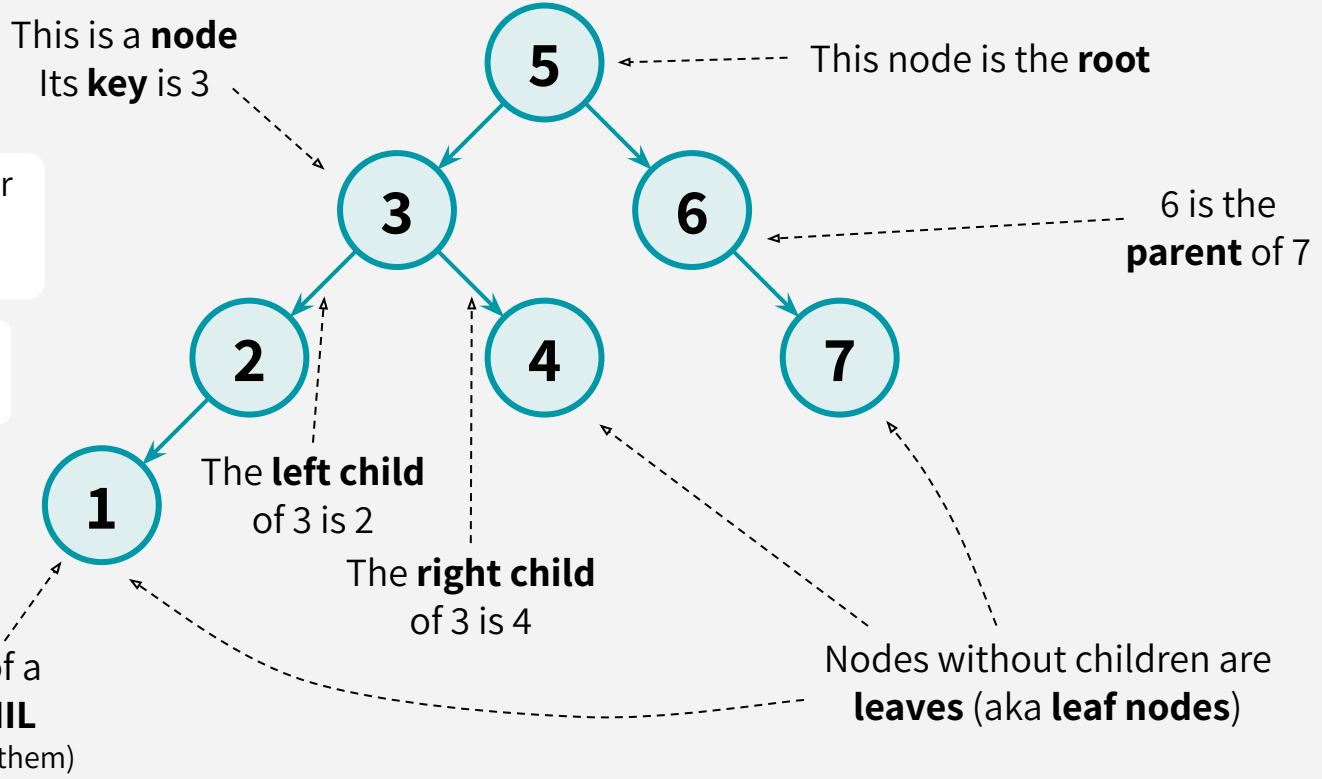
BINARY TREE TERMINOLOGY

Each node has two children

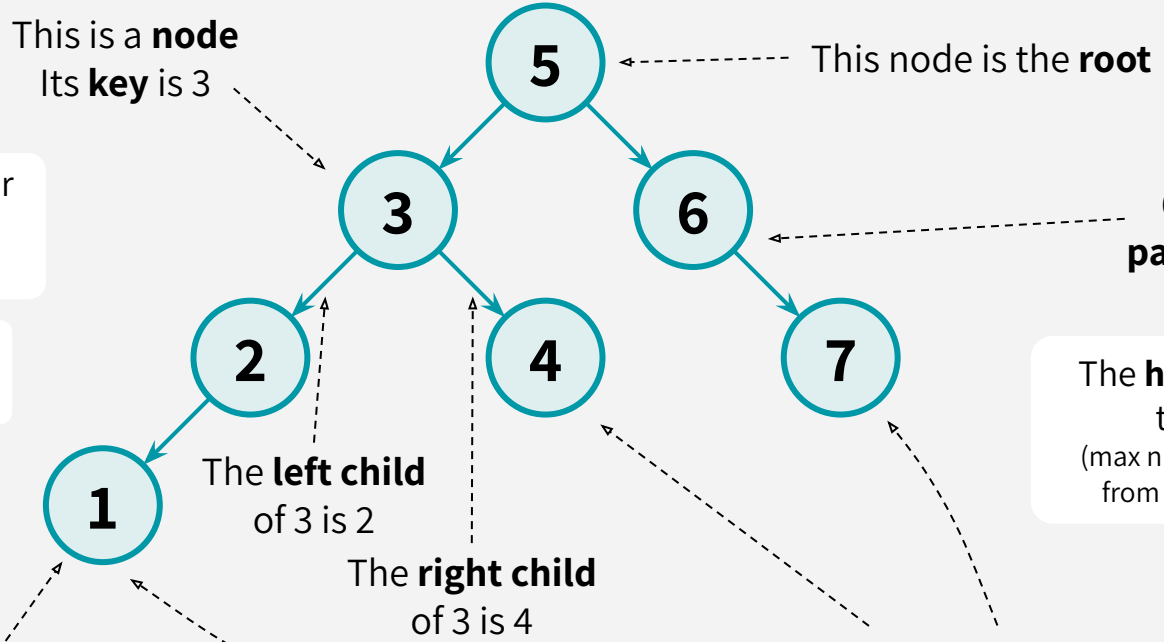
Each node has a pointer to its left child, right child, and parent

The **left descendants** of 5 are 1, 2, 3, and 4

The **ancestors** of 1 are 2, 3, and 5



BINARY TREE TERMINOLOGY



Each node has two children

Each node has a pointer to its left child, right child, and parent

The **left descendants** of 5 are 1, 2, 3, and 4

The **ancestors** of 1 are 2, 3, and 5

This is a **node**
Its **key** is 3

This node is the **root**

6 is the **parent** of 7

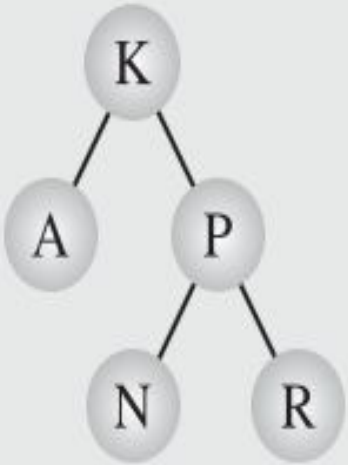
The **height** of this tree is 3
(max number of edges from root to a leaf)

The **left child** of 3 is 2

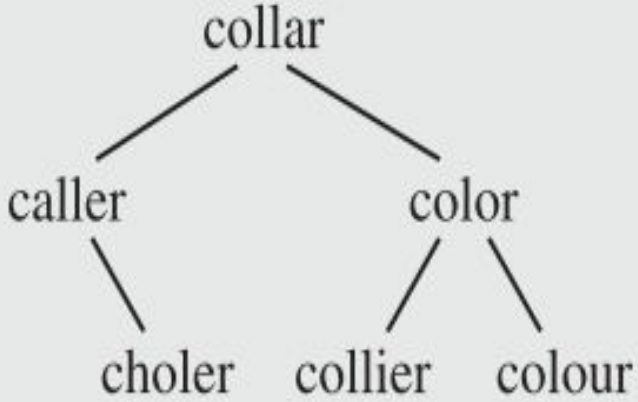
The **right child** of 3 is 4

Both children of a leaf node are **NIL**
(I usually won't draw them)

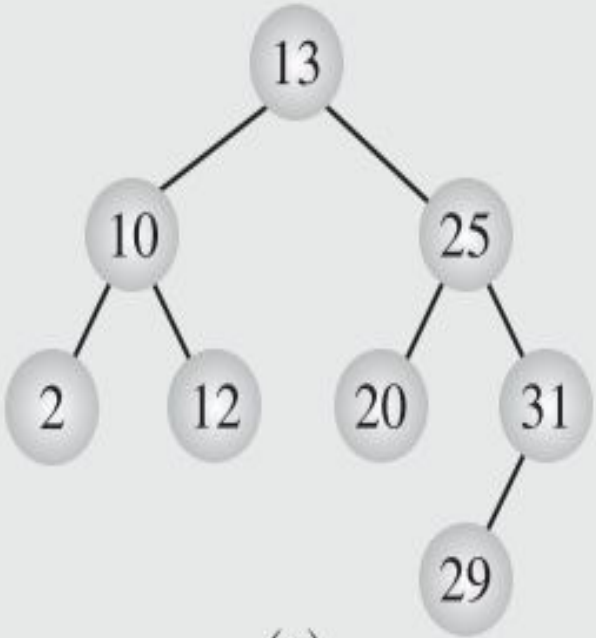
Nodes without children are **leaves** (aka **leaf nodes**)


K
A
P
N
R

(a)


collar
caller
color
choler
collier
colour

(b)

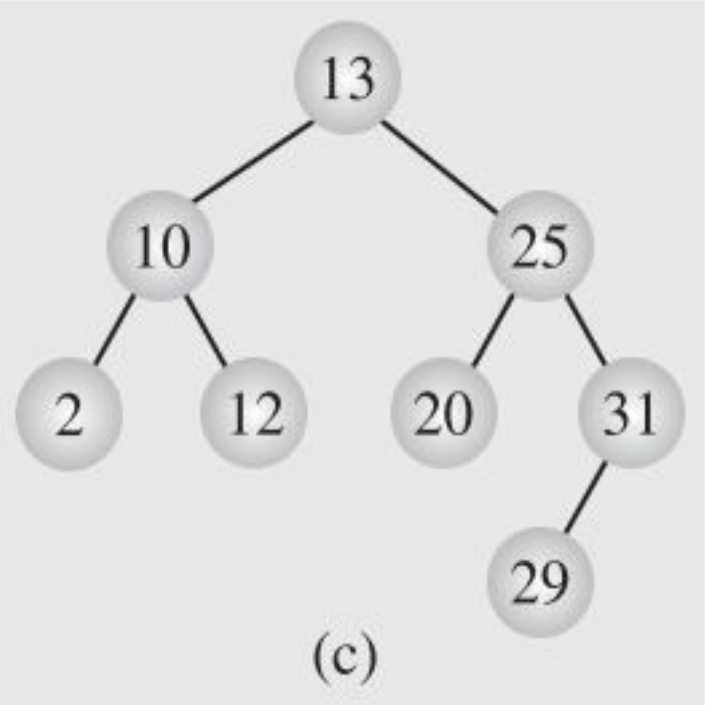

13
10
25
2
12
20
31
29

(c)

BST WITH ARRAYS

- A node is declared as a structure with an information field and two “pointer” fields.
- These pointer fields contain the indexes of the array cells in which the left and right children are stored, if there are any.

Index	Info	Left	Right
0	13	4	2
1	31	6	−1
2	25	7	1
3	12	−1	−1
4	10	5	3
5	2	−1	−1
6	29	−1	−1
7	20	−1	−1



NODE CLASS

```
template<class T> class BSTNode {
public:
    BSTNode( ) { left = right = 0; }
    BSTNode (const T& e, BSTNode<T> *l = 0, BSTNode<T> *r = 0) {
        el = e; left = l; right = r; }
        T el;
        BSTNode<T> *left, *right;
};
```

BST CLASS

- BST ( ) Constructor
- ~BST ( ) Destructor
- void Clear ( ) Clears the Tree
- isEmpty ( ) Returns true for empty tree false otherwise
- void preorder ( ) performs preorder traversal
- void inorder ( ) performs inorder traversal
- void postorder ( ) performs postorder traversal
- T* search (const T& el) search the tree for given key
- void bredthFirst ( ) performs breadth first traersal

BST CLASS

- void iterativePreorder ( ) — performs preorder traversal
- void iterativeInorder ( ) — performs inorder traversal
- void iterativePostorder ( ) — performs postorder traversal
- void deleteByMerging (BSTNode<T> *&)
- void deleteByCopying (BSTNode<T> *&)
- void insert ( ) — Adds new node to the tree

SEARCH IN BST

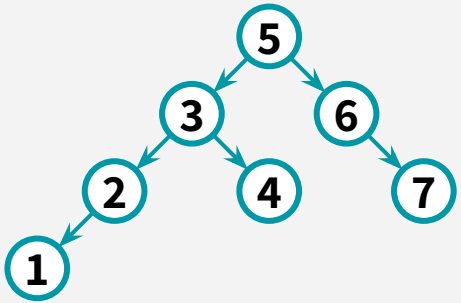
- Compare the element to be located with the value stored in the node currently pointed at.
- If the element is less than the value, go to the left subtree
- If it is greater than that value, try the right subtree.
- And Try Again
- We stop either if the element is found or we reach the end of the tree.

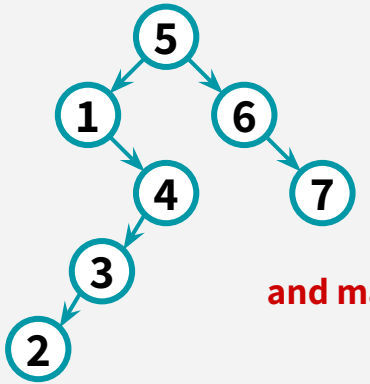
THE BST PROPERTY

A Binary Search Tree (BST) is a binary tree such that:

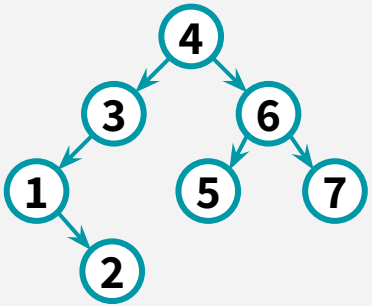
Every LEFT descendant of a node has key less than that node
Every RIGHT descendant of a node has key larger than that node

There exist many valid BSTs that contain these numbers:





and many more...

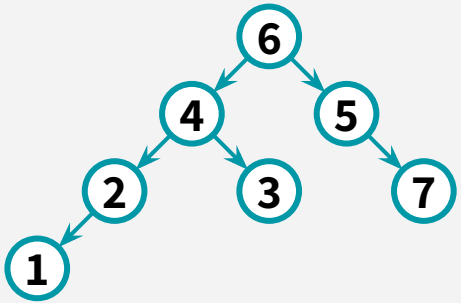


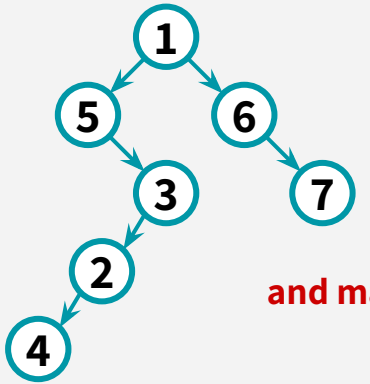
THE BST PROPERTY

A Binary Search Tree (BST) is a binary tree such that:

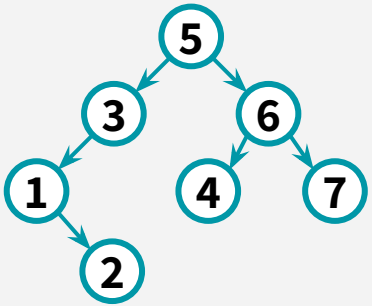
Every LEFT descendant of a node has key less than that node
Every RIGHT descendant of a node has key larger than that node

There also exist many invalid BSTs:





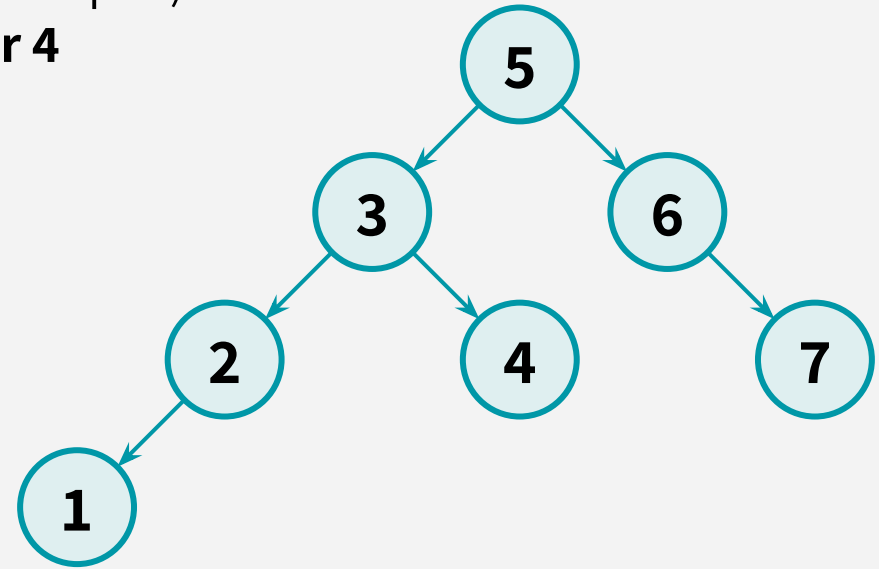
and many more...



SEARCH in BSTs

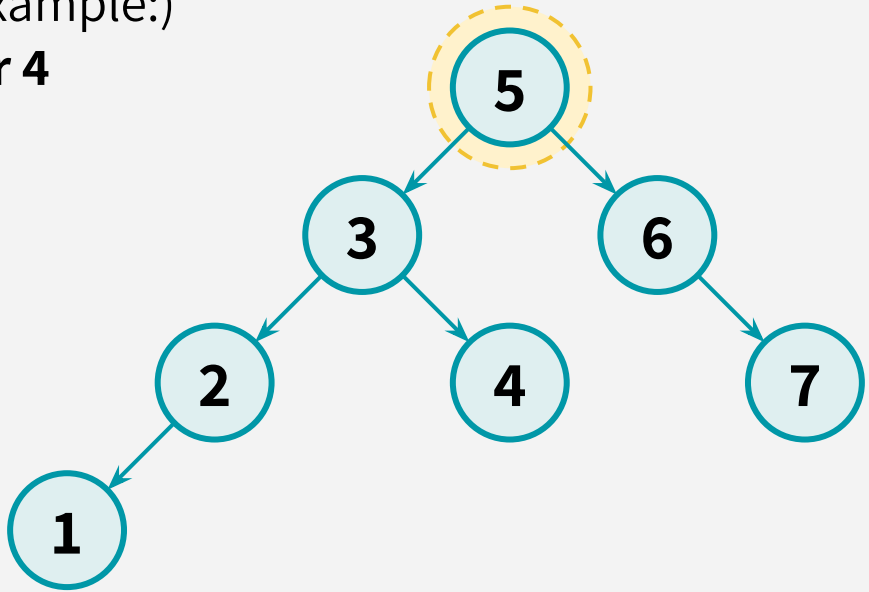
(definition by example:)

search for 4



SEARCH in BSTs

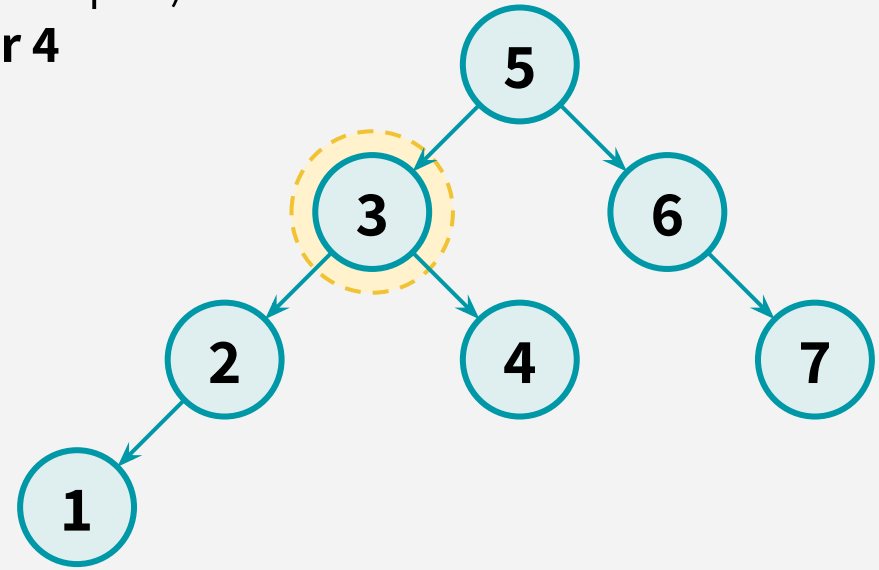
(definition by example:)
search for 4



Compare **4** with **root**:
4 is smaller → go left!

SEARCH in BSTs

(definition by example:)
search for 4

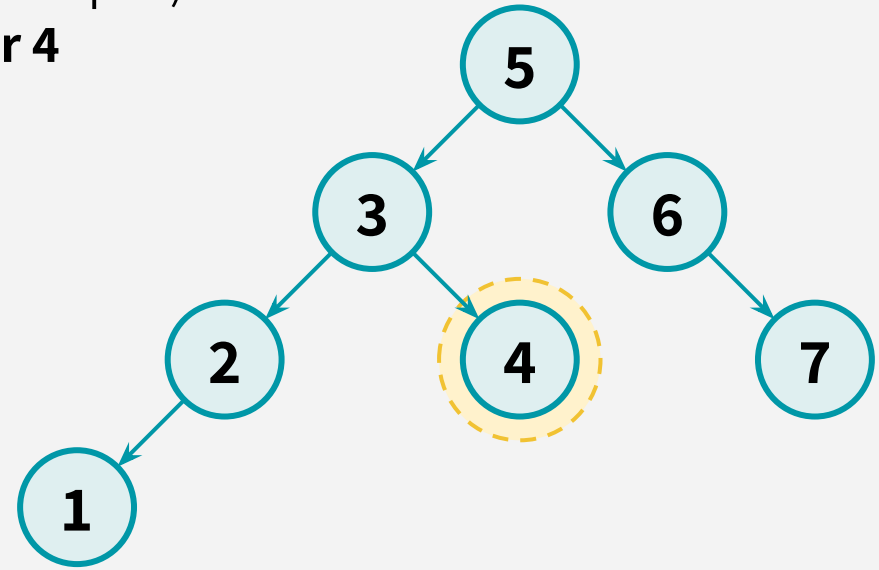


Compare **4** with **root**:
4 is smaller → go left!

Compare **4** with **3**:
4 is larger → go right!

SEARCH in BSTs

(definition by example:)
search for 4



Compare **4** with **root**:
4 is smaller → go left!

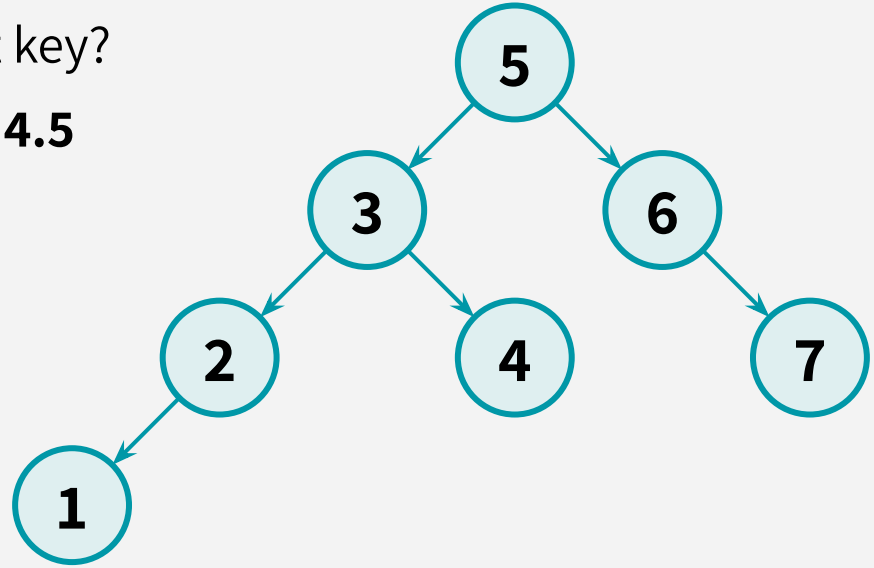
Compare **4** with **3**:
4 is larger → go right!

Compare **4** with **4**:
4 = 4 → We found it!

SEARCH in BSTs

What happens if we search for a non-existent key?

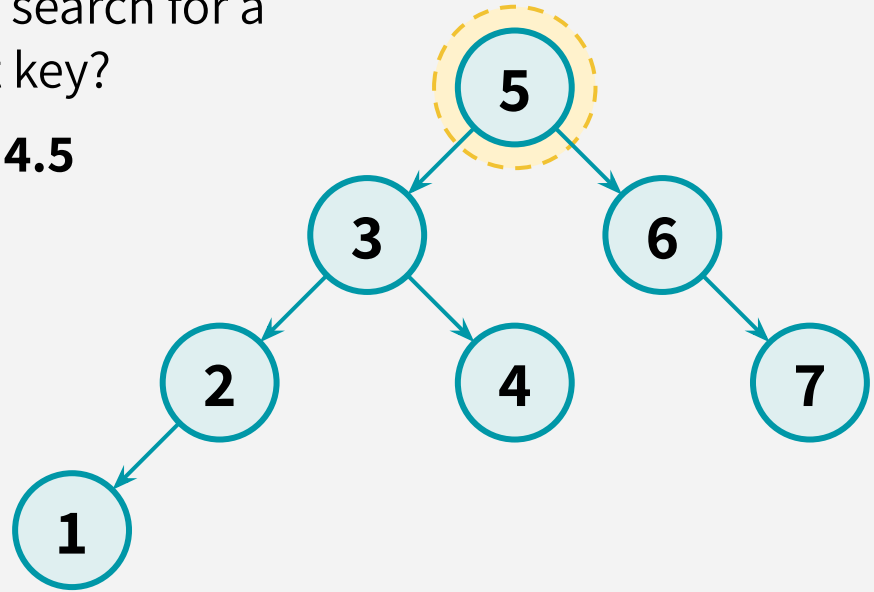
search for 4.5



SEARCH in BSTs

What happens if we search for a non-existent key?

search for 4.5

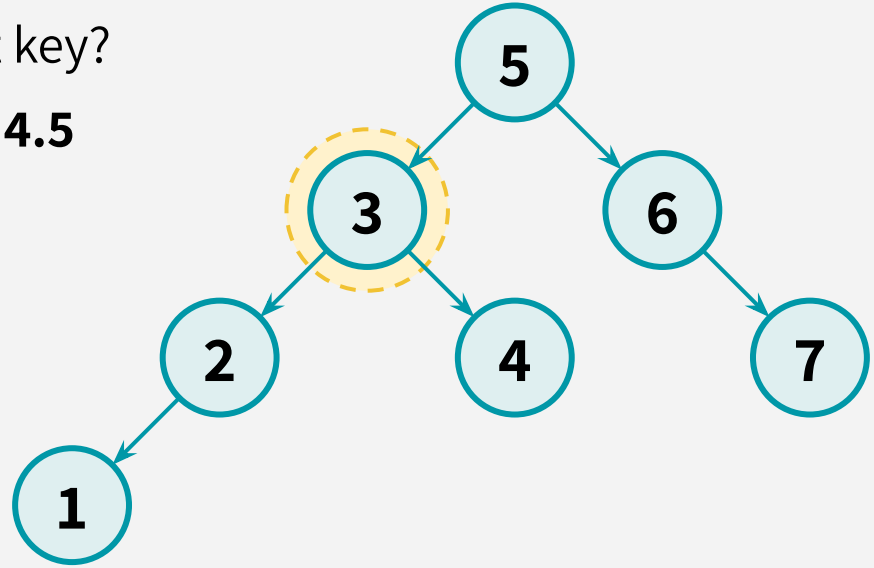


Compare **4.5** with **root**:
4.5 is smaller → go left!

SEARCH in BSTs

What happens if we search for a non-existent key?

search for 4.5



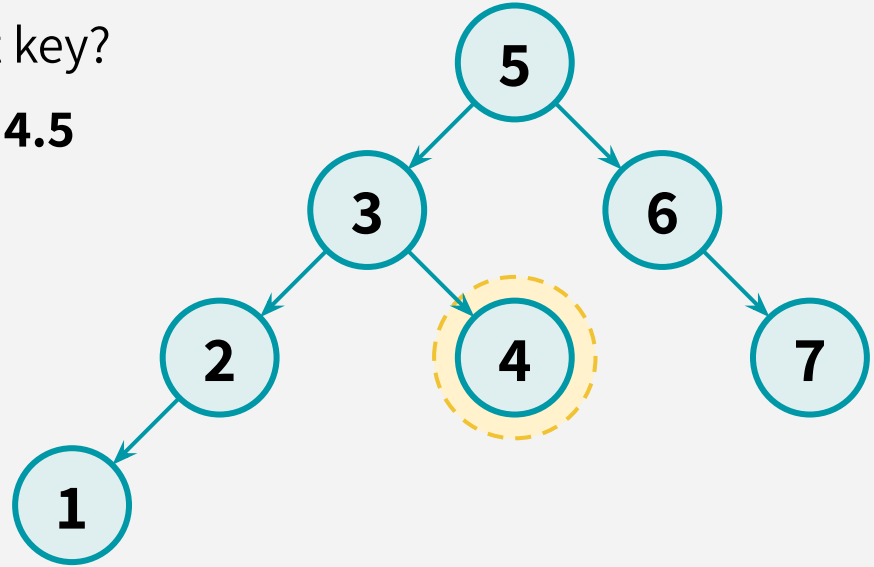
Compare **4.5** with **root**:
4.5 is smaller → go left!

Compare **4.5** with **3**:
4.5 is larger → go right!

SEARCH in BSTs

What happens if we search for a non-existent key?

search for 4.5



Compare **4.5** with **root**:
4.5 is smaller → go left!

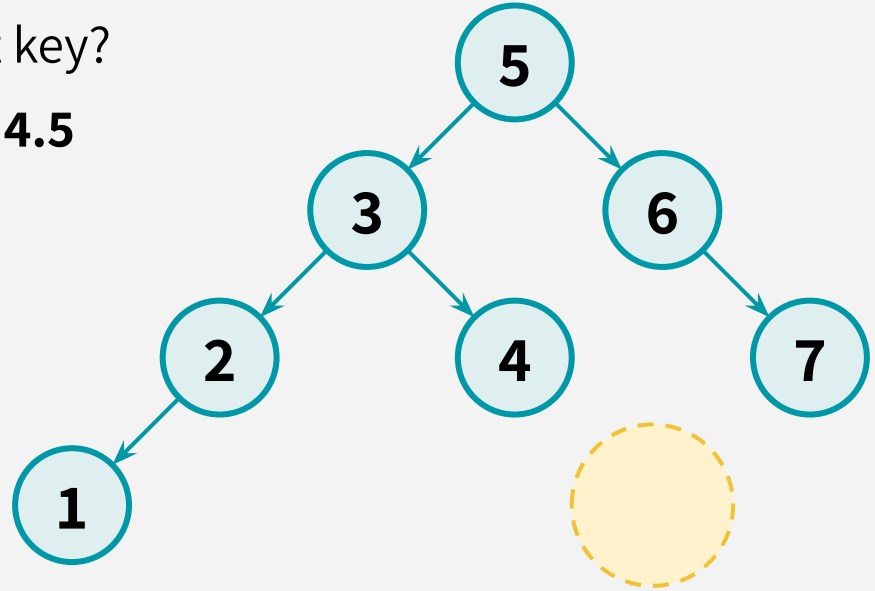
Compare **4.5** with **3**:
4.5 is larger → go right!

Compare **4.5** with **4**:
4.5 is larger → go right!

SEARCH in BSTs

What happens if we search for a non-existent key?

search for 4.5



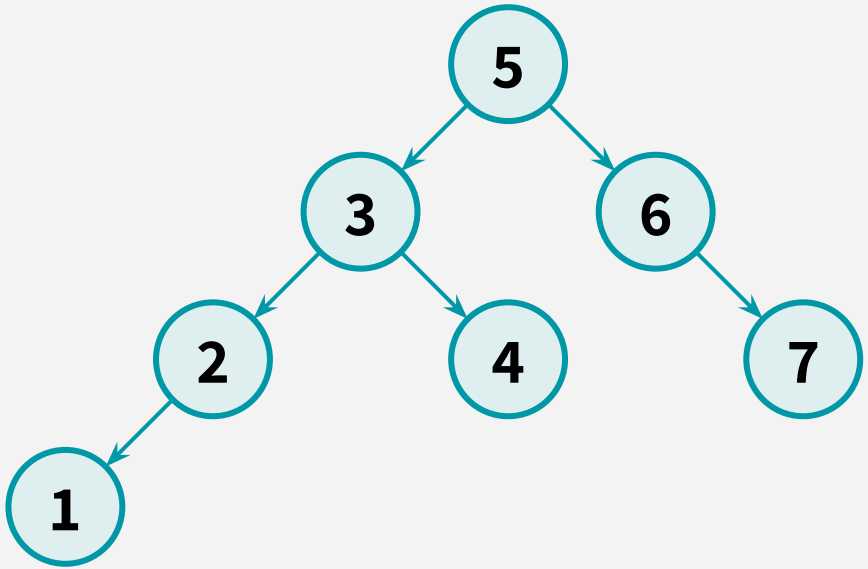
Compare **4.5** with **root**:
4.5 is smaller → go left!

Compare **4.5** with **3**:
4.5 is larger → go right!

Compare **4.5** with **4**:
4.5 is larger → go right!

Oops, we hit **NIL**!
We can just return the last node seen before we fell off the tree (4)

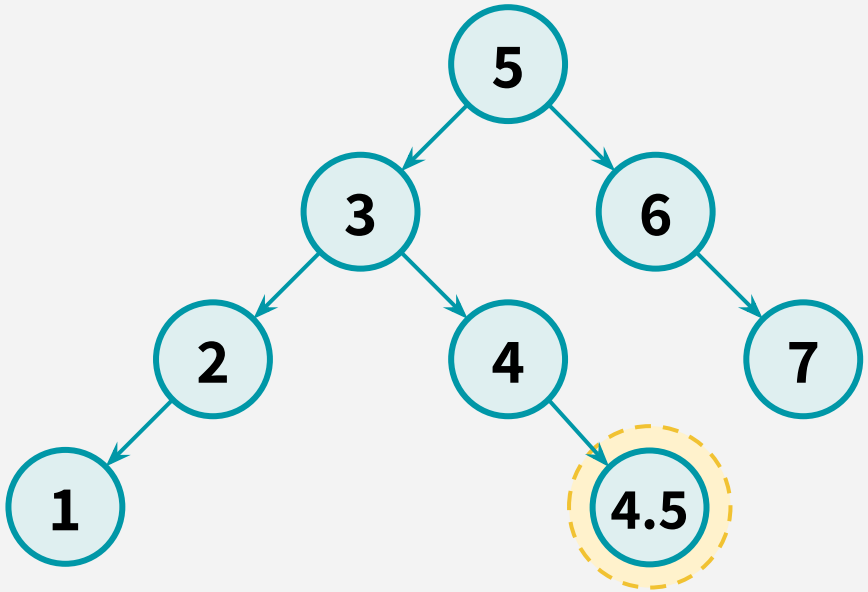
INSERT in BSTs




```
INSERT(root, key):
    x = SEARCH(root, key)
    node = new node with key
    if key < x.key:
        x.left = node
    if key > x.key:
        x.right = node
    if key = x.key:
        return
```

INSERT in BSTs

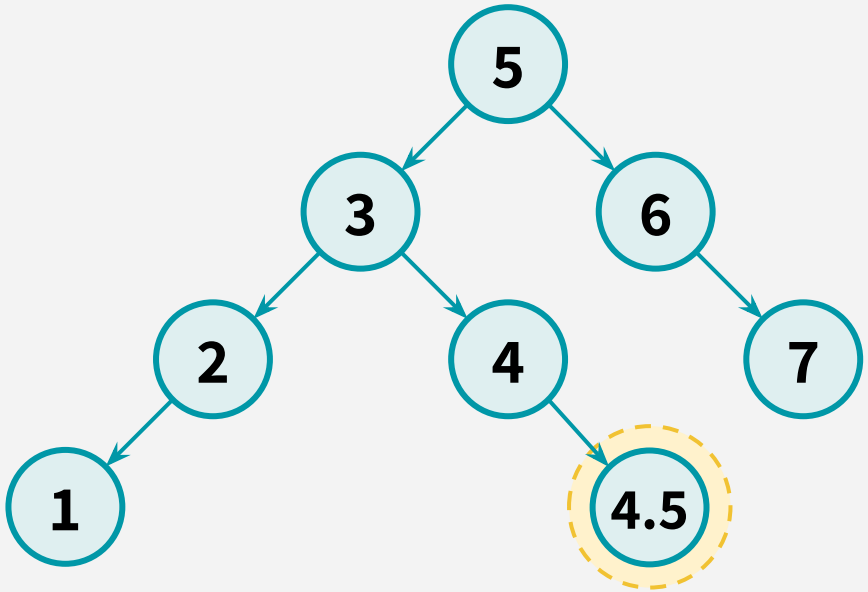
Example: **Insert 4.5**




```
INSERT(root, key):
    x = SEARCH(root, key)
    node = new node with key
    if key < x.key:
        x.left = node
    if key > x.key:
        x.right = node
    if key = x.key:
        return
```

INSERT in BSTs

Example: **Insert 4.5**

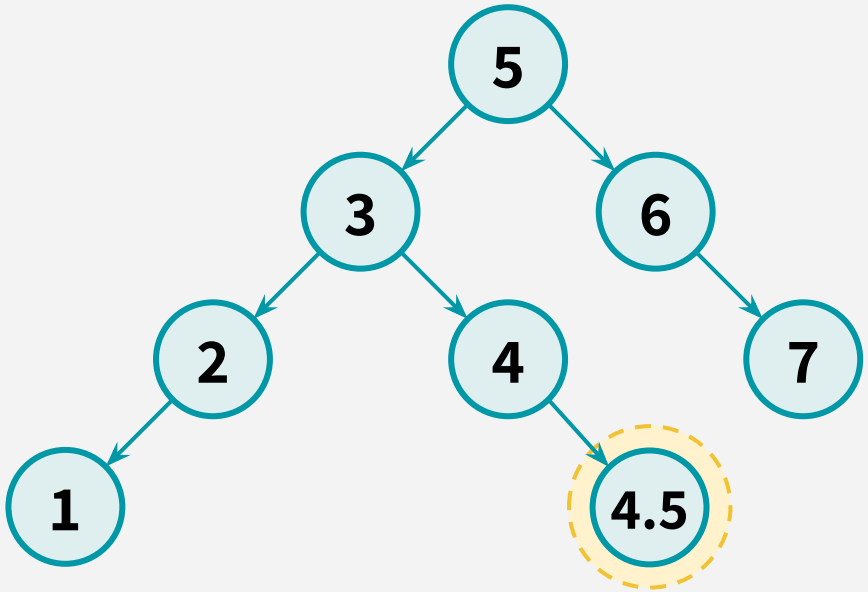



```
INSERT(root, key):
    x = SEARCH(root, key)
    node = new node with key
    if key < x.key:
        x.left = node
    if key > x.key:
        x.right = node
    if key = x.key:
        return
```

What's the runtime?

INSERT in BSTs

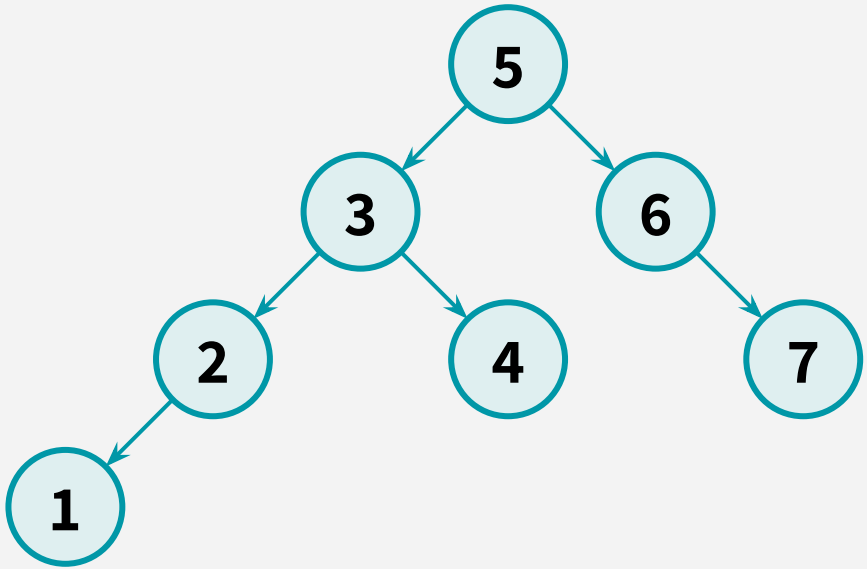
Example: **Insert 4.5**




```
INSERT(root, key):
    x = SEARCH(root, key)
    node = new node with key
    if key < x.key:
        x.left = node
    if key > x.key:
        x.right = node
    if key = x.key:
        return
```

Runtime of **INSERT** = runtime of **SEARCH** = **O(height)**

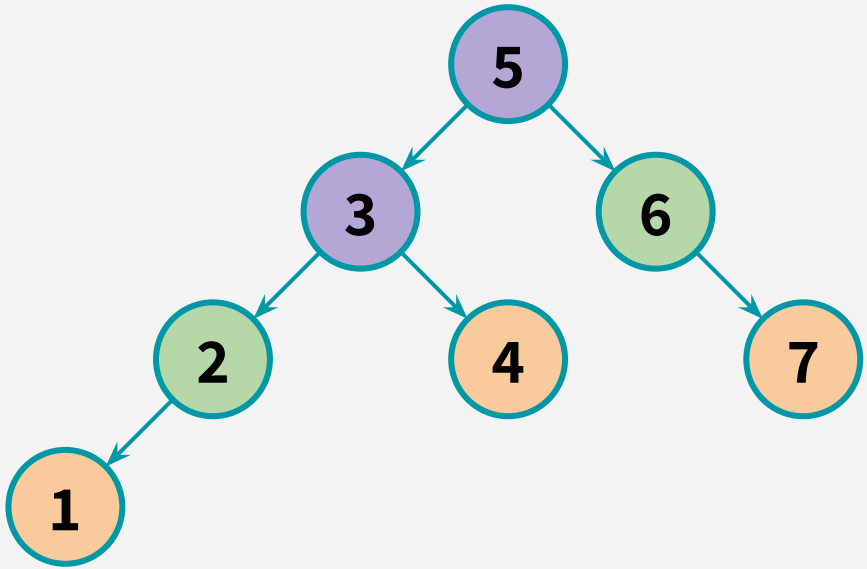
DELETE in BSTs




```
DELETE(root, key):
    x = SEARCH(root, key)
    if key = x.key:
        ...delete x...
```

This is a bit more complicated… we need to consider 3 cases

DELETE in BSTs




```
DELETE(root, key):
    x = SEARCH(root, key)
    if key = x.key:
        CASE 1: x is a leaf
        CASE 2: x has 1 child
        CASE 3: x has 2 children
```

DELETE in BSTs

CASE 1: x is a leaf

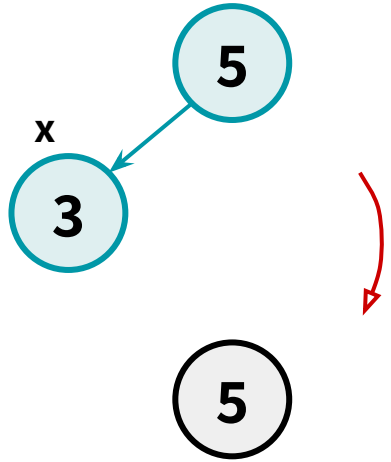
CASE 2: x has 1 child

CASE 3: x has 2 children

DELETE in BSTs

CASE 1: x is a leaf

Just delete x!



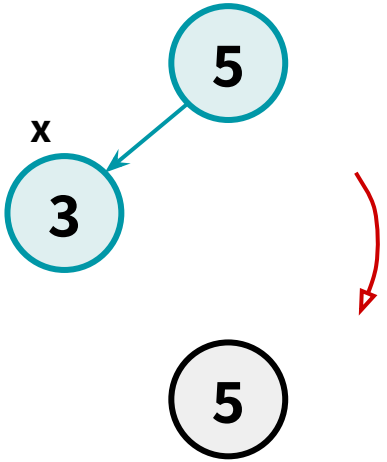
CASE 2: x has 1 child

CASE 3: x has 2 children

DELETE in BSTs

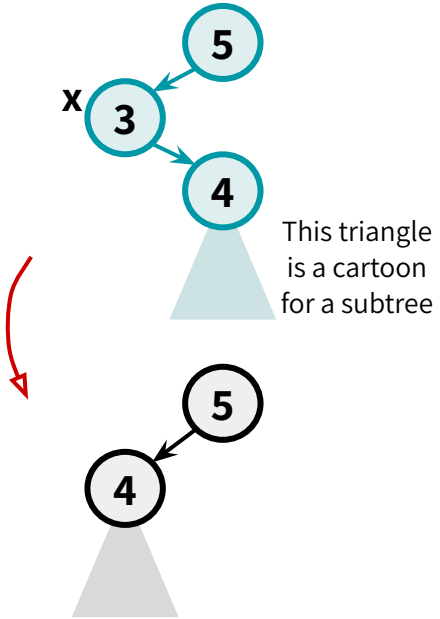
CASE 1: x is a leaf

Just delete x!



CASE 2: x has 1 child

Move its child up!

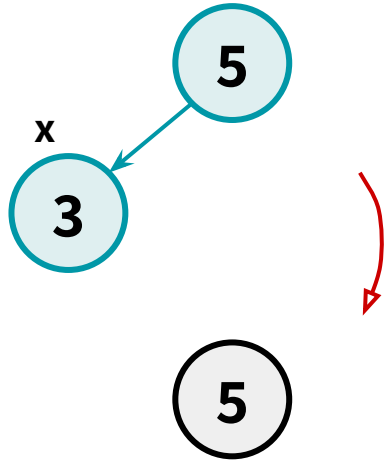


CASE 3: x has 2 children

DELETE in BSTs

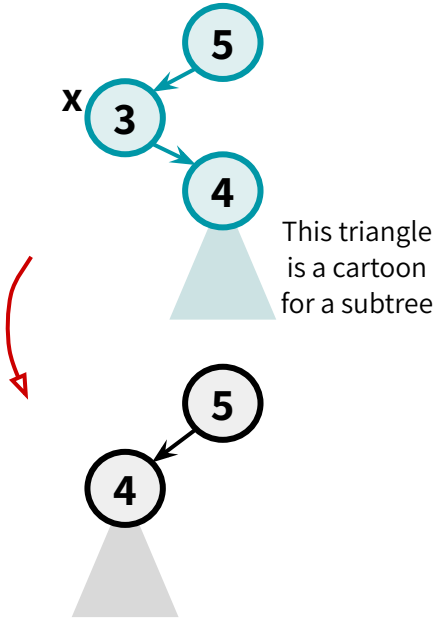
CASE 1: x is a leaf

Just delete x!



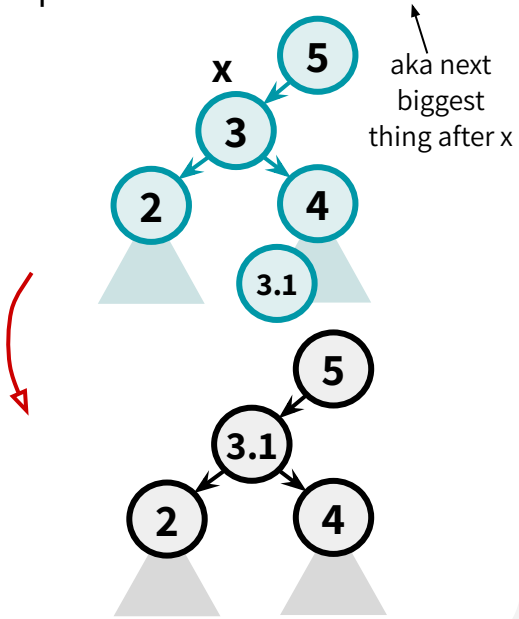
CASE 2: x has 1 child

Move its child up!



CASE 3: x has 2 children

Replace x with its successor



RUNTIME OF SEARCH/INSERT/DELETE

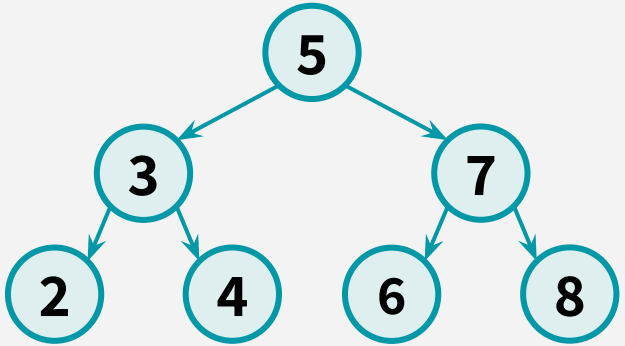
INSERT and **DELETE** both call **SEARCH** (and then do some O(1)-time operation)

Runtime of **SEARCH** = **O(height)**

RUNTIME OF SEARCH/INSERT/DELETE

INSERT and **DELETE** both call **SEARCH** (and then do some O(1)-time operation)

Runtime of **SEARCH** = **O(height)**

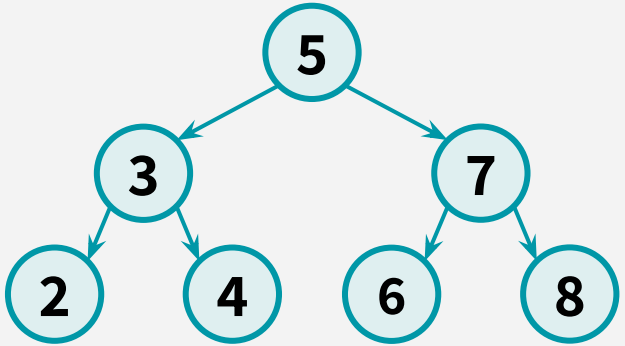


Sometimes SEARCH takes **O(log n)**

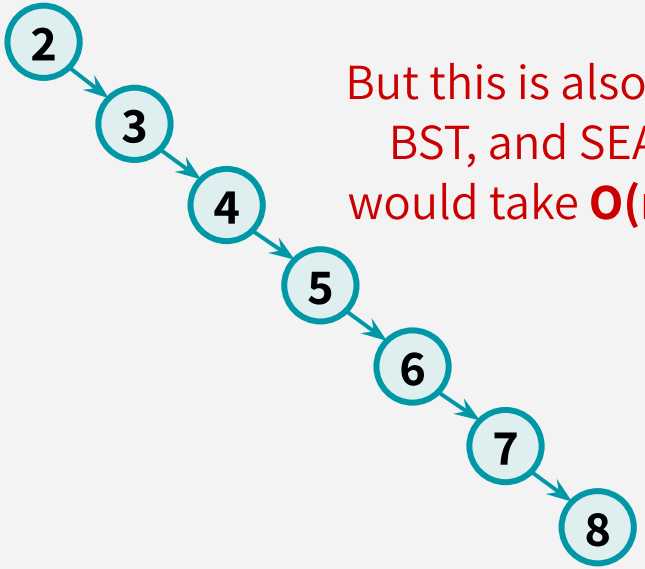
RUNTIME OF SEARCH/INSERT/DELETE

INSERT and **DELETE** both call **SEARCH** (and then do some O(1)-time operation)

Runtime of **SEARCH** = **O(height)**



Sometimes SEARCH takes **O(log n)**



But this is also a valid BST, and SEARCH would take **O(n)** here

RUNTIME OF SEARCH/INSERT/DELETE

INSERT and **DELETE** both call **SEARCH** (and then do some O(1)-time operation)

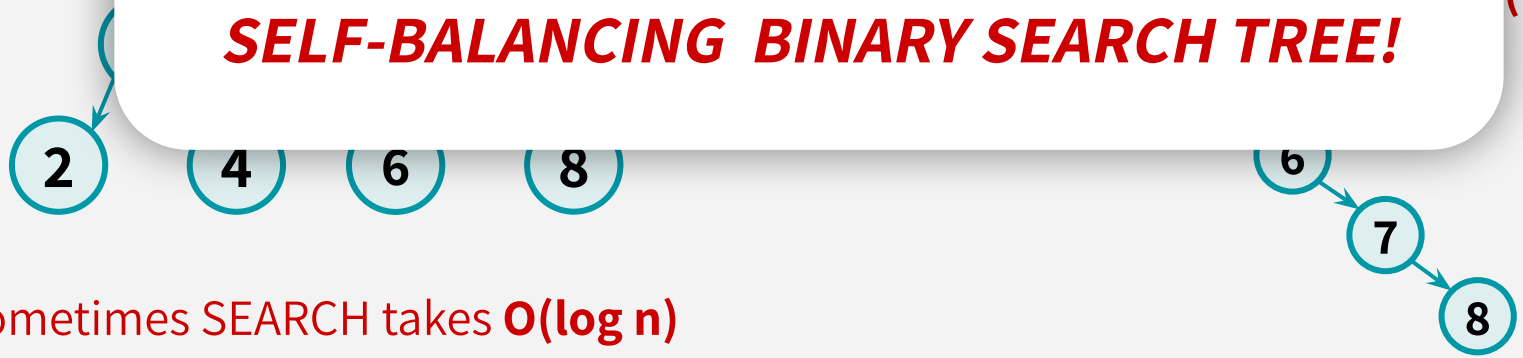
Runtime of **SEARCH** = **O(height)**

What do we do? We want fast SEARCH/INSERT/DELETE but sometimes the height might be big (O(n))!!!

We like balanced trees… introducing

SELF-BALANCING BINARY SEARCH TREE!

o a valid
EARCH
(n) here



Sometimes SEARCH takes **O(log n)**

SEARCH

```
template<class T>
T* BST<T>::search(BSTNode<T>* p, const T& el) const
{
        while (p != 0)
            if (el == p->el) return &p->el;
            else if (el < p->el) p = p->left;
            else p = p->right;
        return 0;
}
```

SEARCH IN BST

- If an element occurs more than once, then two approaches are possible.
- Locates the first occurrence of an element and disregards the others.
    a. In this case, the tree contains redundant nodes that are never used for their own sake; they are accessed only for testing.
- All occurrences of an element may have to be located. Such a search always has to finish with a leaf.

INSERT

- To insert a new node with key el, a tree node with a dead end has to be reached, and the new node has to be attached to it.
- The key el is compared to the key of a node currently being examined during a tree scan. If el is less than that key, the left child (if any) of p is tried; otherwise, the right child (if any) is tested.
- If the child of p to be tested is empty, the scanning is discontinued and the new node becomes this child.

INSERT

```
template<class T> void BST<T>::insert(const T& el) {

BSTNode<T> *p = root, *prev = 0;
        while (p != 0) { // find a place for inserting new node;
                prev = p;
                if (el < p->el) p = p->left;

                else p = p->right;        }

        if (root == 0) // tree is empty;
                root = new BSTNode<T>(el);


        else if (el < prev->el) prev->left = new BSTNode<T>(el);


        else prev->right = new BSTNode<T>(el); }
```