ساختمان داده و الگوریتم ها (CE203)

جلسه سیزدهم:
درخت دودویی جستجو

سجاد شیرعلی شهرضا

پاییز 1401

شنبه، 28 آبان 1401

Based on Stanford CS161 Slides from Summer 2020 by Karey Shi

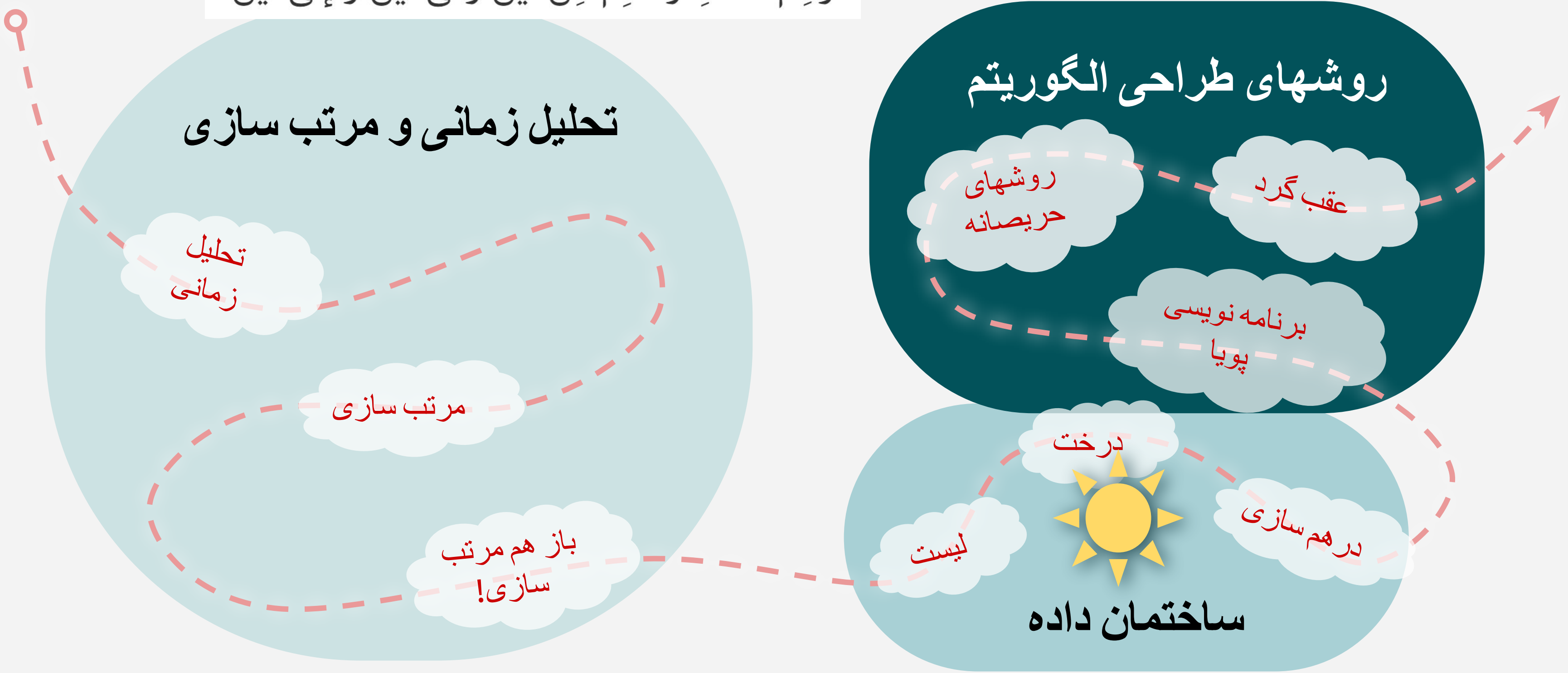
اطلاع رسانی

- بخش مرتبط کتاب برای این جلسه: 12

نقشه راهنما

روزها فکر من این است و همه شب سخنم — که چرا غافل از احوال دل خویشتنم
از کجا آمده‌ام آمدنم بهر چه بود — به کجا می‌روم آخر ننمایی وطنم

«رَحِمَ اللّهُ اِمْرَأً عَلِمَ مِن أینَ وَفی أینَ وَ إلی أینَ»



درخت دودویی جستجو

د.د.ج. چیست و چگونه از آن استفاده میکنیم؟

SOME OTHER DATA STRUCTURES

Here are some data structures that can store objects like $\boxed{5}$ (aka, **nodes** with **keys**)

SOME OTHER DATA STRUCTURES

Here are some data structures that can store objects like $\boxed{5}$ (aka, **nodes** with **keys**)

Sorted Arrays

1	2	3	4	5	7

Linked Lists

HEAD → 3 → 5 → 1 → 4 → 7 → 2

SOME OTHER DATA STRUCTURES

Here are some data structures that can store objects like $\boxed{5}$ (aka, **nodes** with **keys**)

Sorted Arrays

1	2	3	4	5	7

O(n) INSERT/DELETE: first, find the relevant element (via SEARCH) and move a bunch of elements in the array

O(log n) SEARCH: use binary search to see if an element is in A

Linked Lists

HEAD → 3 → 5 → 1 → 4 → 7 → 2

SOME OTHER DATA STRUCTURES

Here are some data structures that can store objects like $\boxed{5}$ (aka, **nodes** with **keys**)

Sorted Arrays

1	2	3	4	5	7

O(n) INSERT/DELETE: first, find the relevant element (via SEARCH) and move a bunch of elements in the array

O(log n) SEARCH: use binary search to see if an element is in A

Linked Lists

HEAD → 3 → 5 → 1 → 4 → 7 → 2

O(1) INSERT: just insert the element at the head of the linked list

O(n) SEARCH/DELETE: since the list is not necessarily sorted, you need to scan the list (delete by manipulating pointers)

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)

BINARY SEARCH TREE MOTIVATION

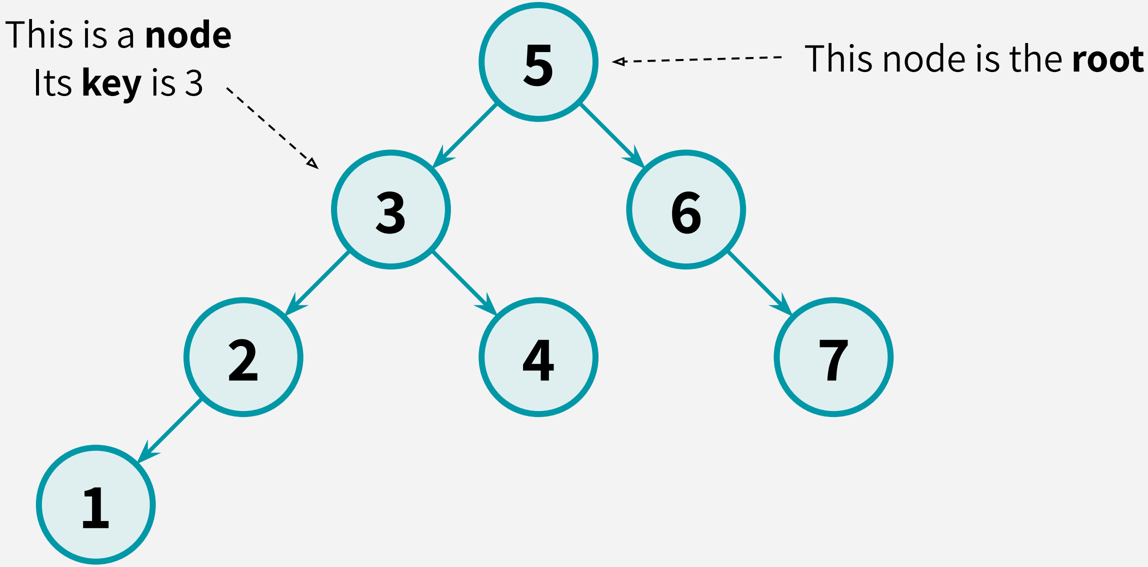
OPERATION	SORTED ARRAY	UNSORTED LINKED LIST	BST (WORST CASE)
SEARCH	O(log(n))	O(n)	O(n)
DELETE	O(n)	O(n)	O(n)
INSERT	O(n)	O(1)	O(n)

BINARY SEARCH TREE MOTIVATION

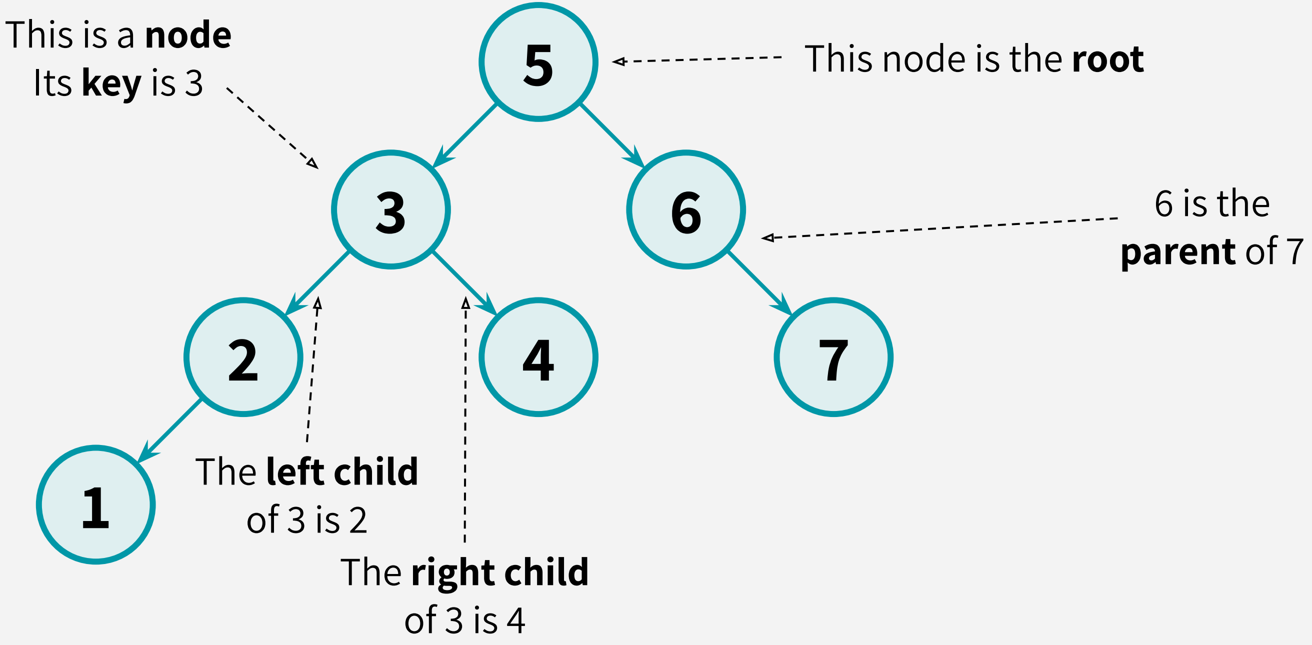
OPERATION	SORTED ARRAY	UNSORTED LINKED LIST	BST (WORST CASE)	BST (BALANCED)
SEARCH	$O(\log(n))$	$O(n)$	$O(n)$	$O(\log(n))$
DELETE	$O(n)$	$O(n)$	$O(n)$	$O(\log(n))$
INSERT	$O(n)$	$O(1)$	$O(n)$	$O(\log(n))$

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

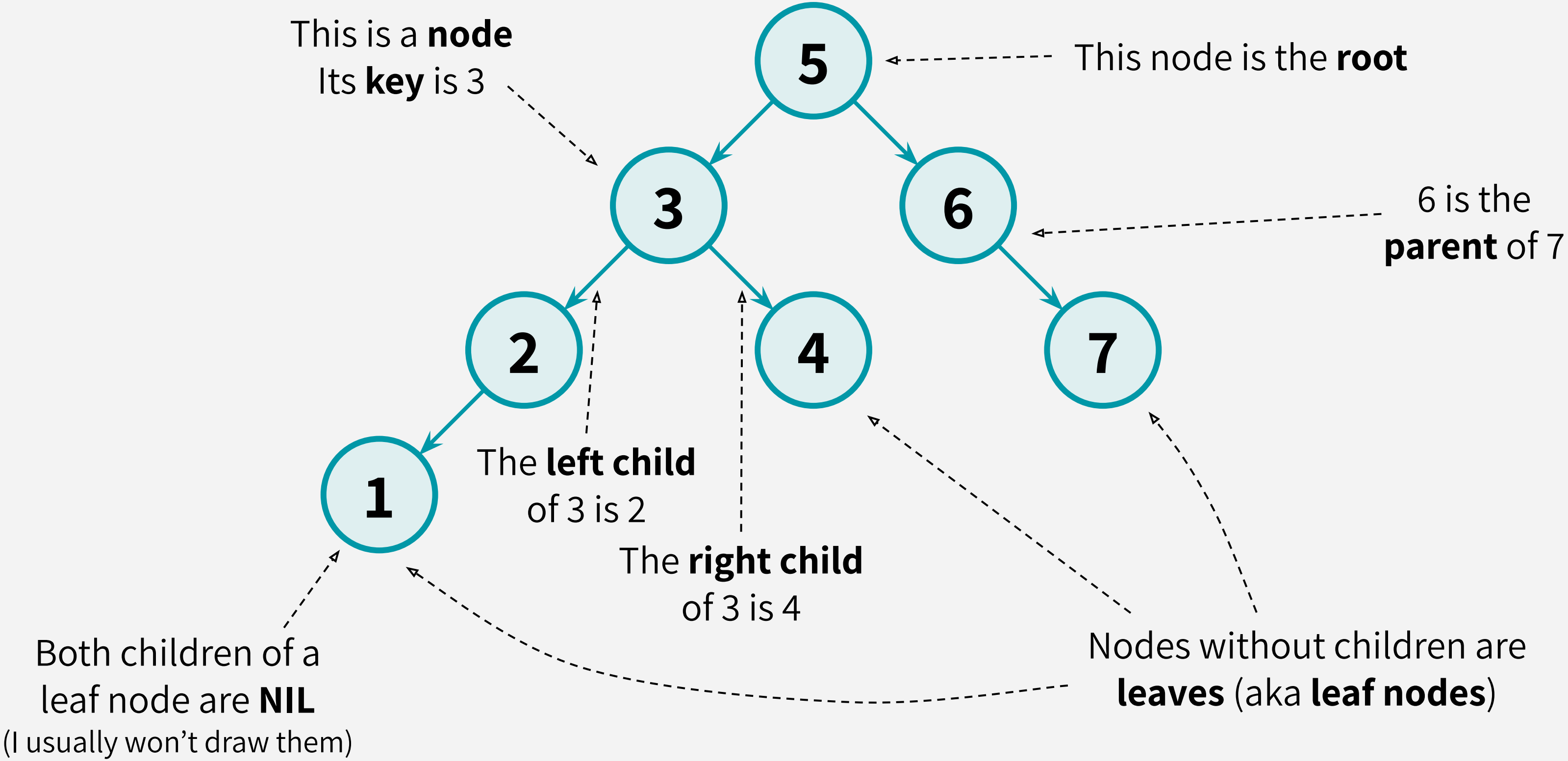
BINARY TREE TERMINOLOGY



BINARY TREE TERMINOLOGY



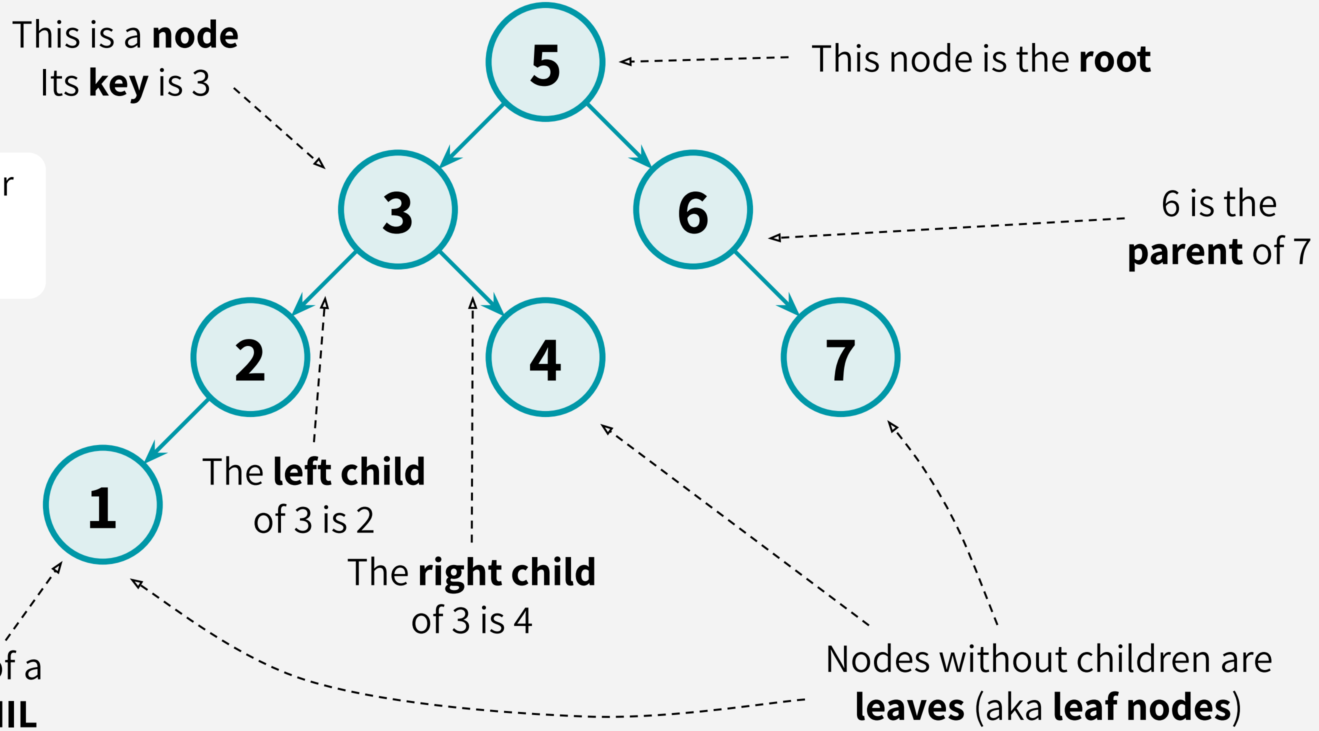
BINARY TREE TERMINOLOGY



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

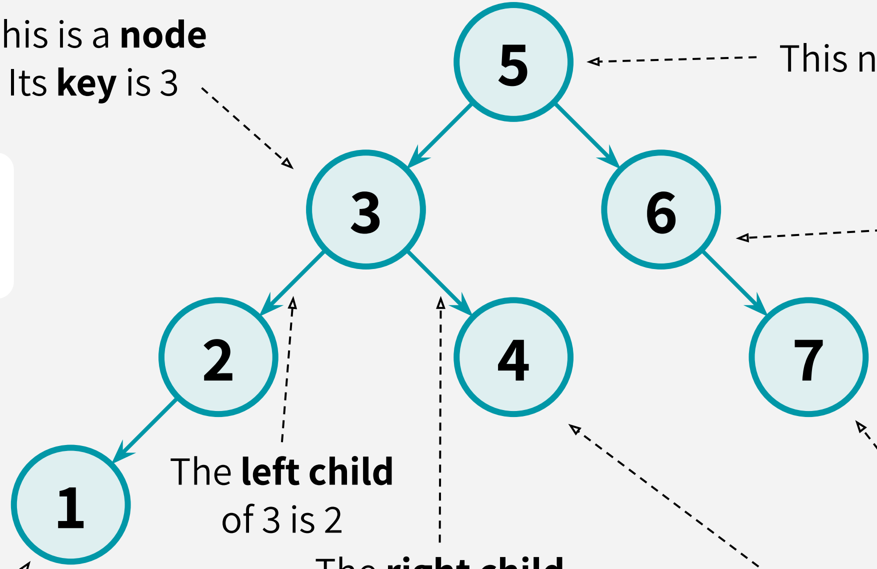
Each node has a pointer to its left child, right child, and parent

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

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

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

This node is the **root**



6 is the **parent** of 7

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**)

BINARY TREE TERMINOLOGY

Each node has two children

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

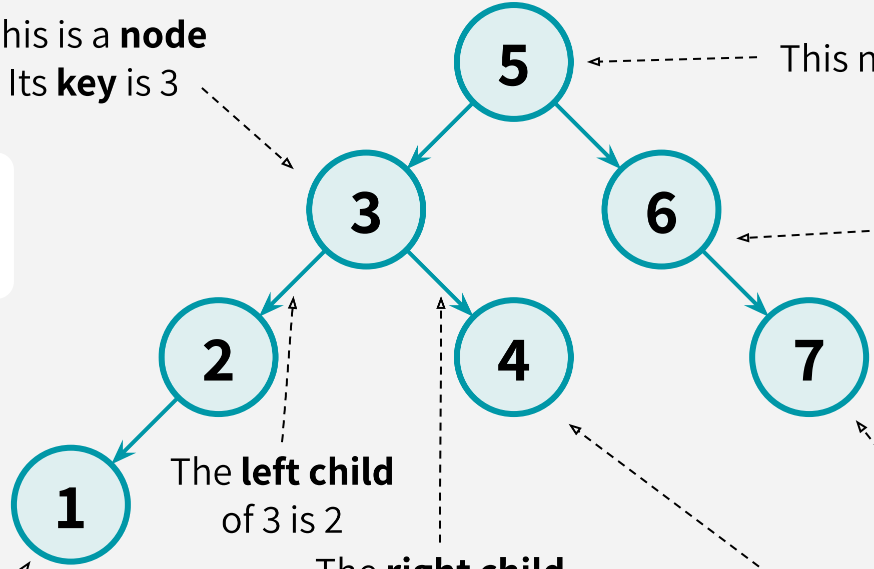
The **left descendants** of 5 are 1, 2, 3, and 4

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**)

سوال؟

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

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:

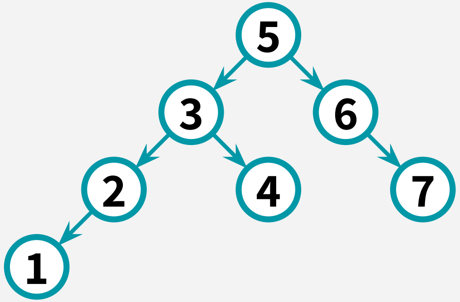
5 3 6 2 1 7 4

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:



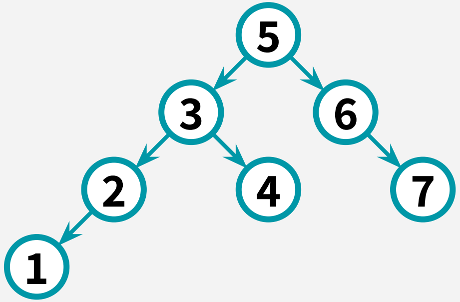
THE BST PROPERTY

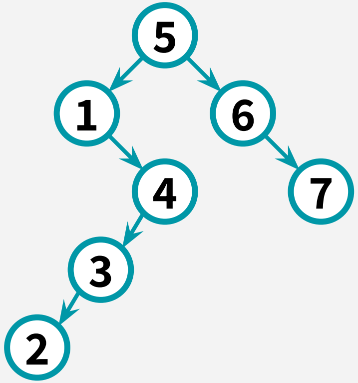
A Binary Search Tree (BST) is a binary tree such that:

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:

5 3 6 2 1 7 4





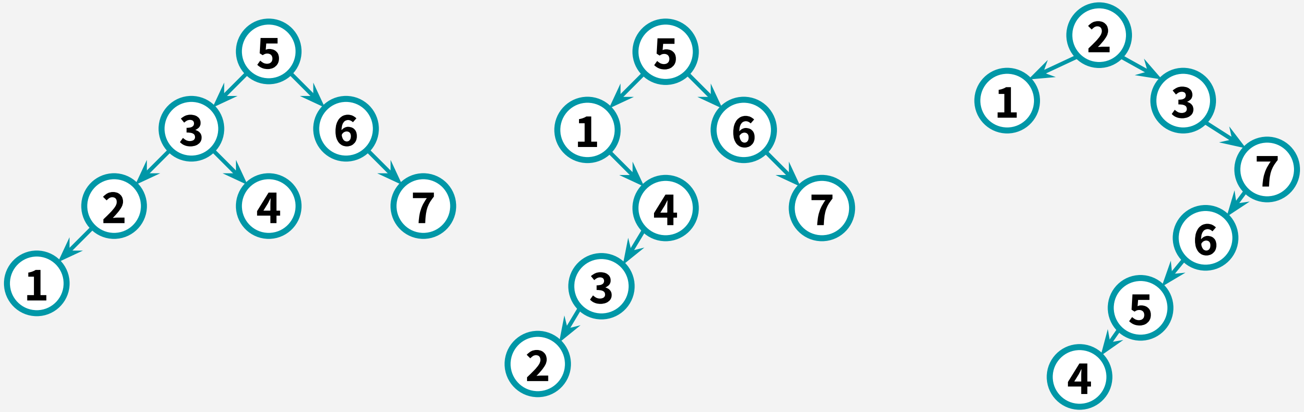
THE BST PROPERTY

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:

5 3 6 2 1 7 4



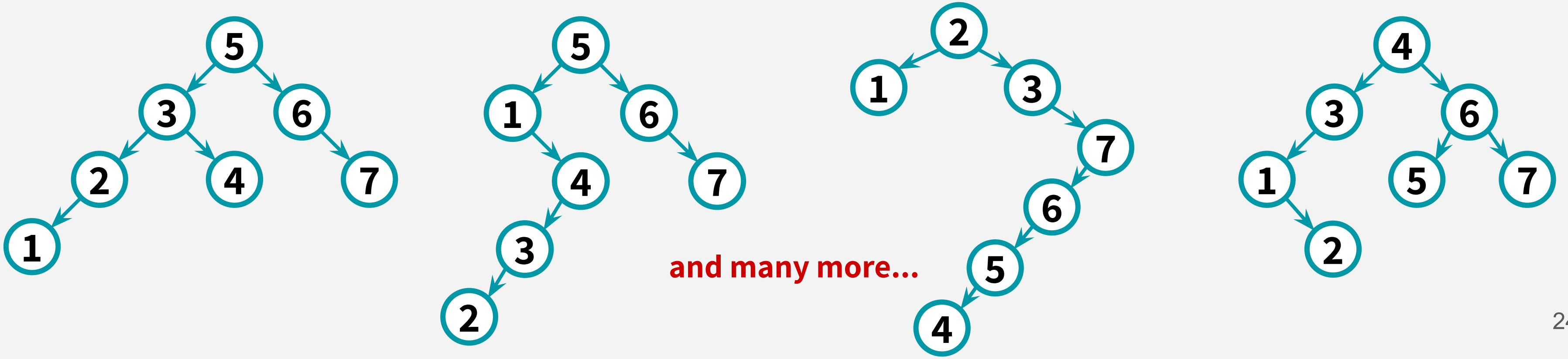
THE BST PROPERTY

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:

5 3 6 2 1 7 4

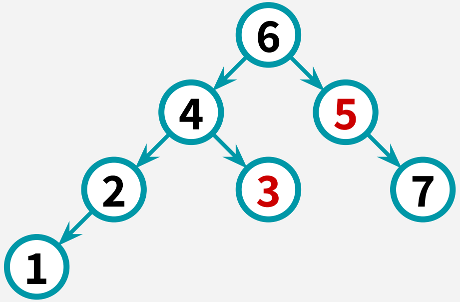


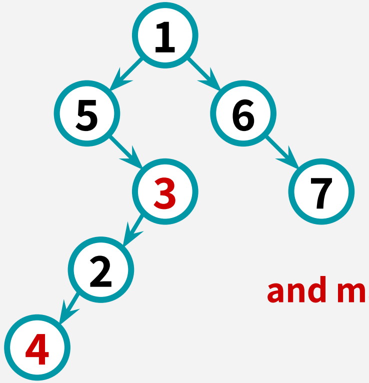
THE BST PROPERTY

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

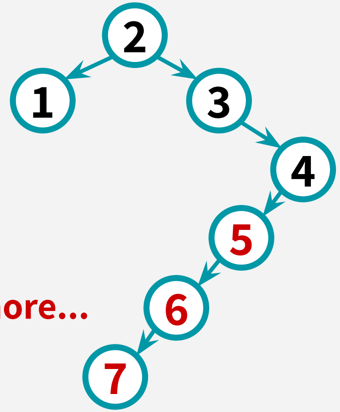
Every LEFT descendant of a node has key less than that node
Every RIGHT descendant of a node has key larger than that node

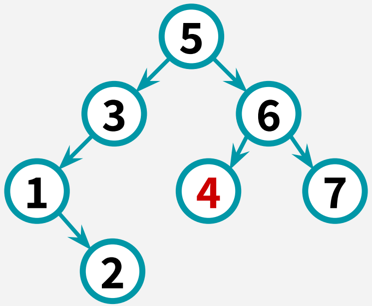
There also exist many **invalid** BSTs:





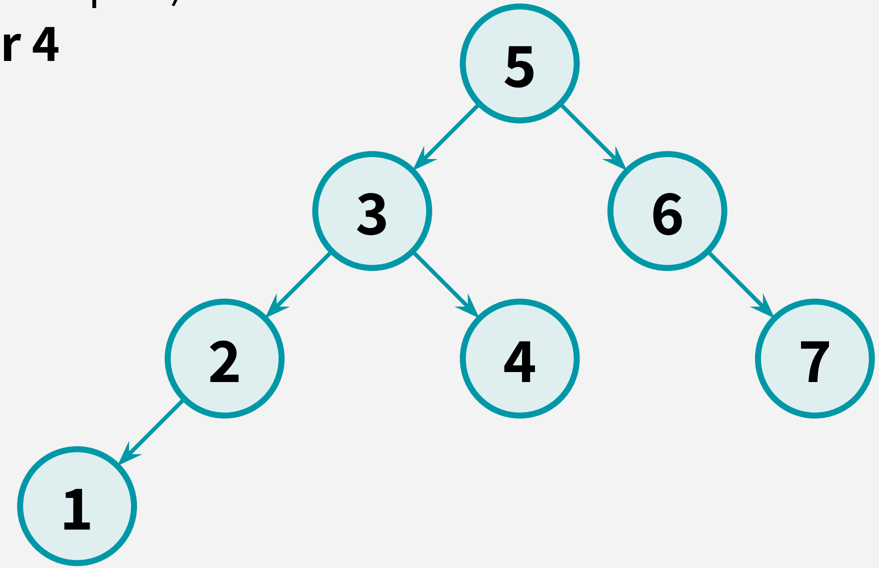
and many more...





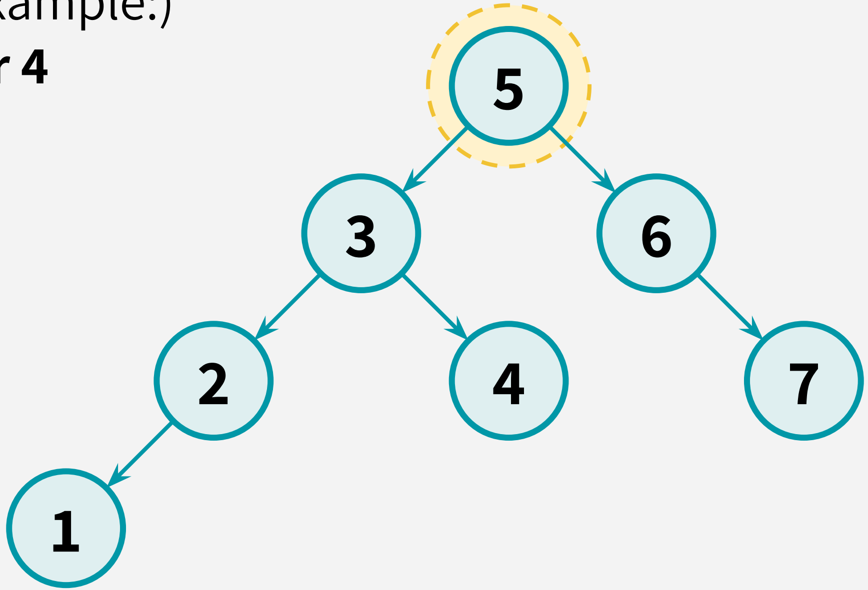
SEARCH in BSTs

(definition by example:)
search for 4



SEARCH in BSTs

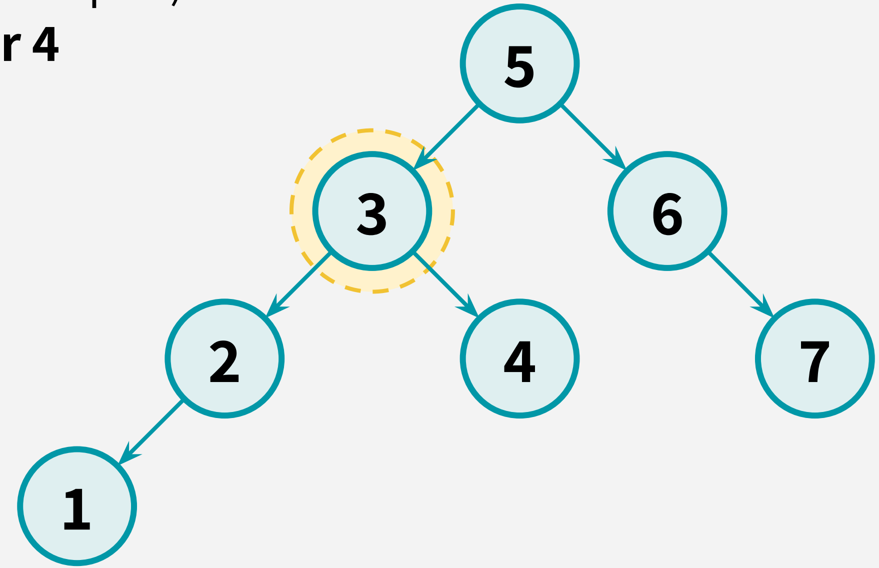
(definition by example:)
search for 4



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

SEARCH in BSTs

(definition by example:)
search for 4

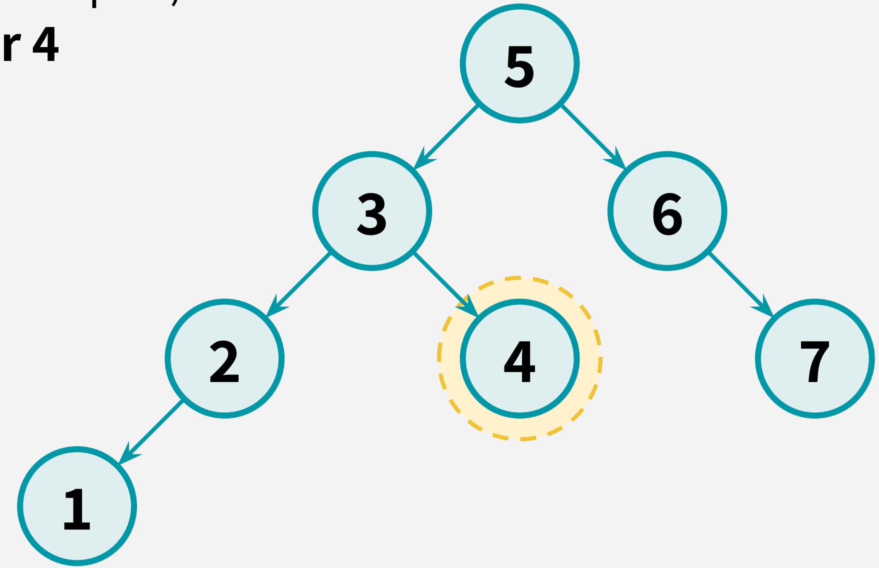


Compare **4** with **root**:
4 is smaller → go left!

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!

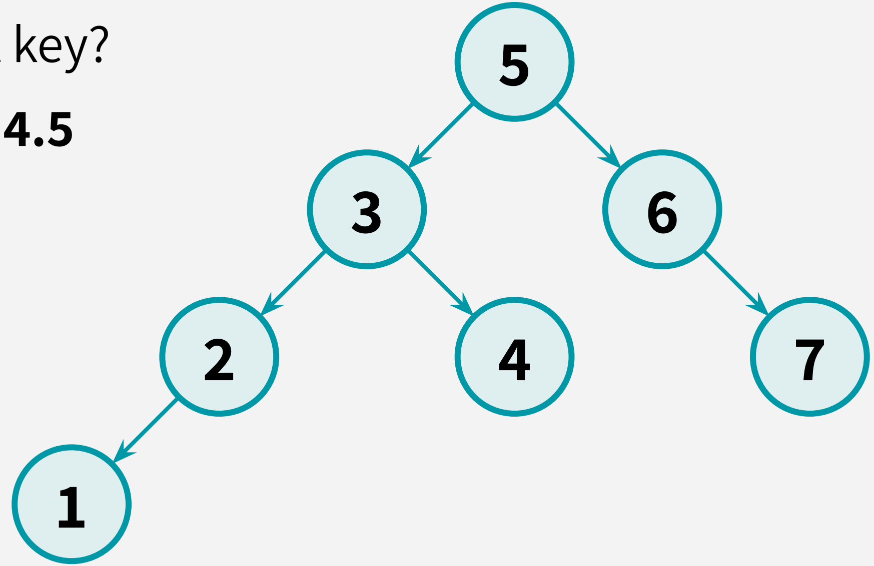
Compare **4** with **3**:
4 is larger ⟶ go right!

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

SEARCH in BSTs

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

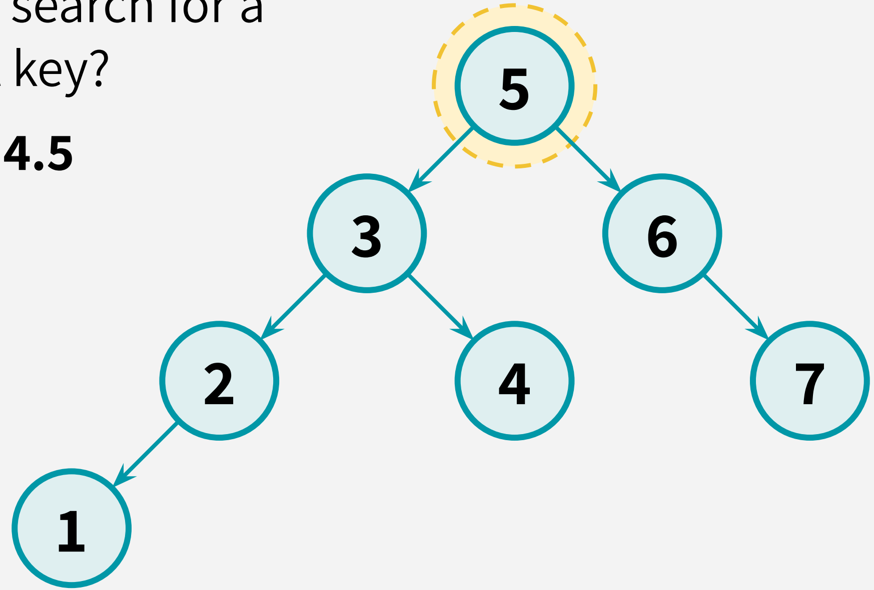
search for 4.5



SEARCH in BSTs

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

search for 4.5

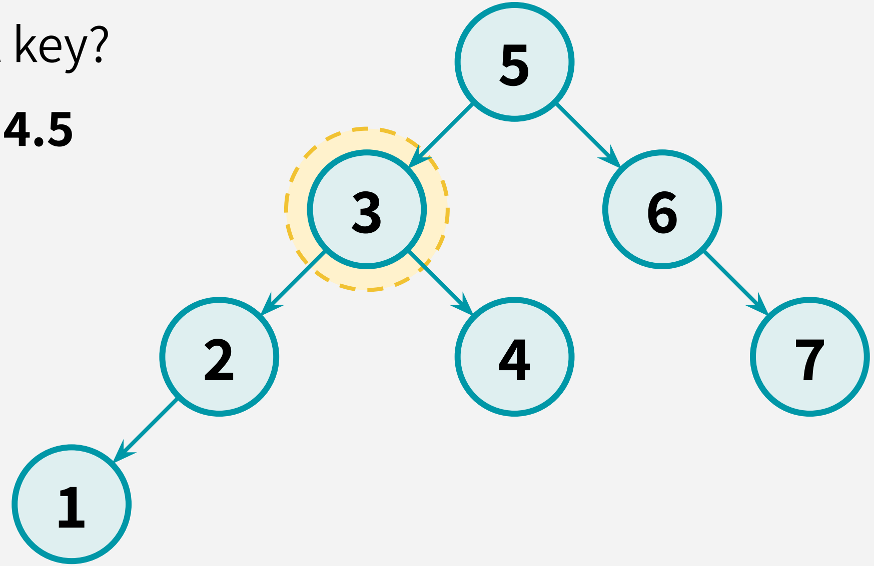


Compare **4.5** with **root**:
4.5 is smaller → go left!

SEARCH in BSTs

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

search for 4.5



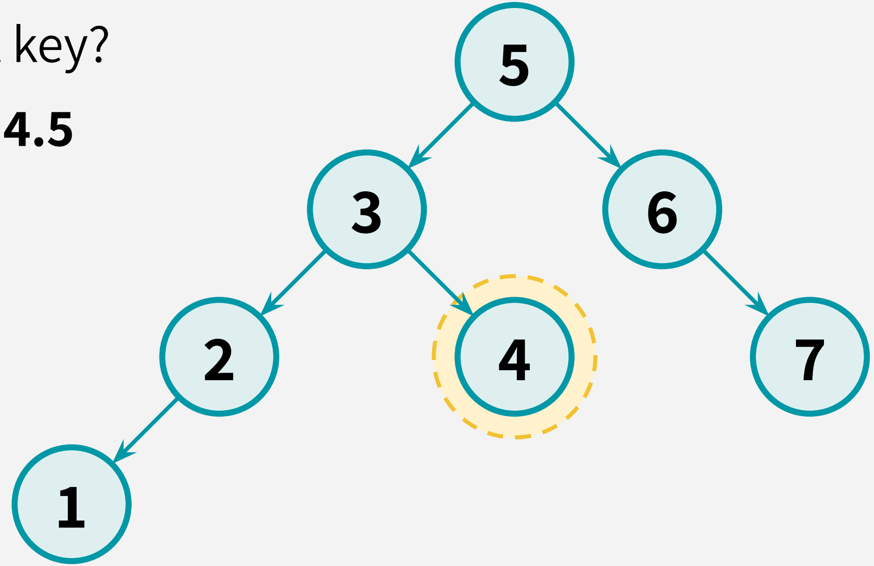
Compare **4.5** with **root**:
4.5 is smaller → go left!

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!

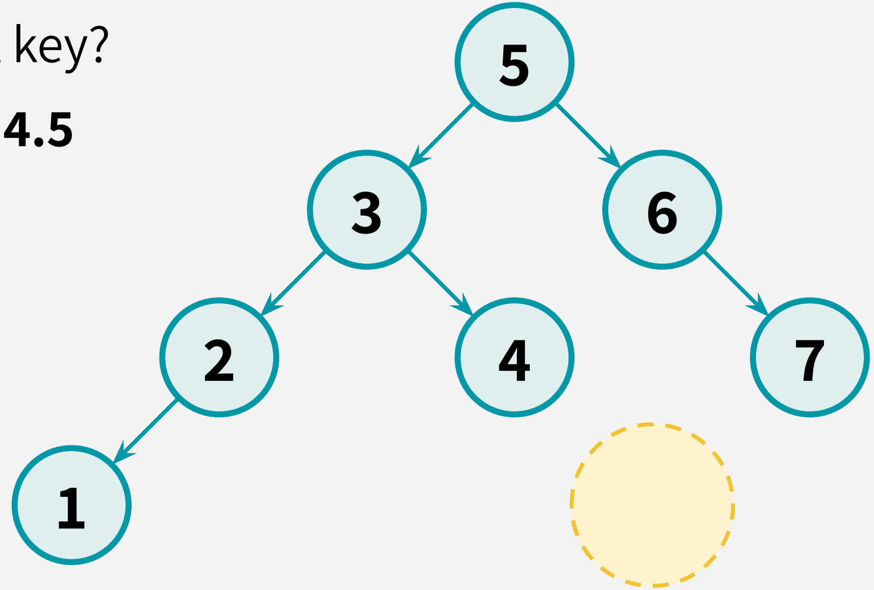
Compare **4.5** with **3**:
4.5 is larger → go right!

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!

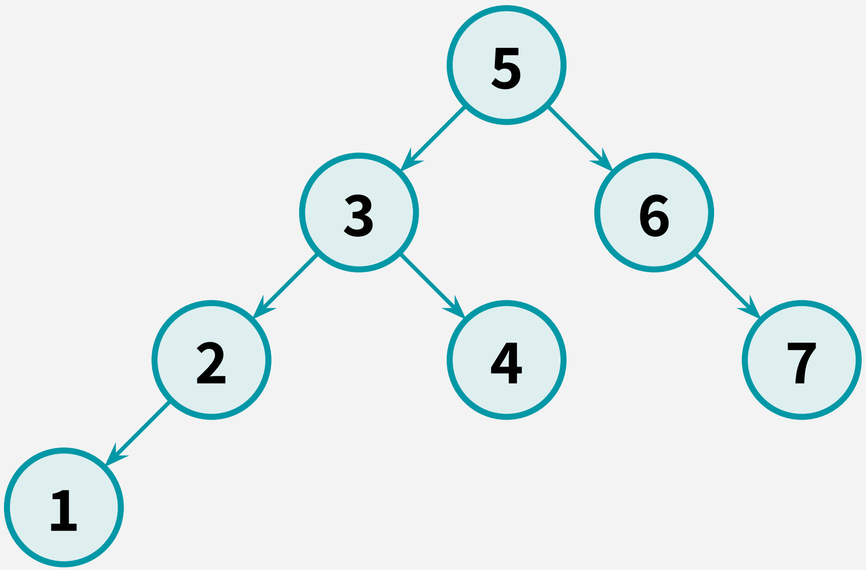
Compare **4.5** with **3**:
4.5 is larger → go right!

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)

سوال؟

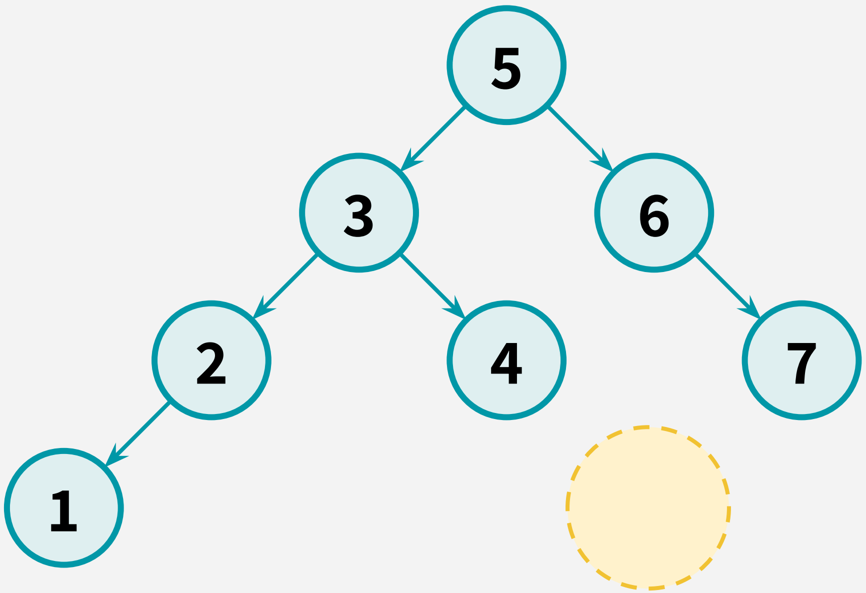
INSERT in BSTs




```
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

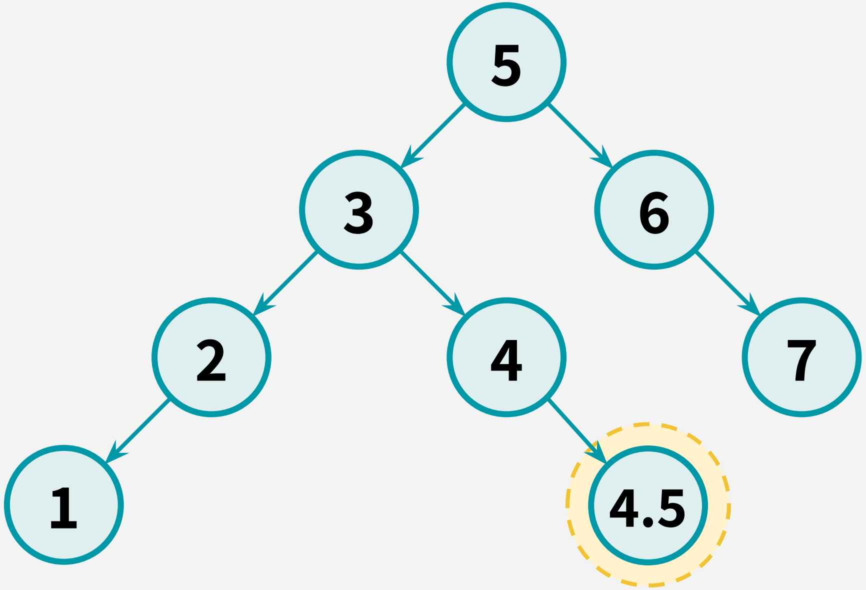
Example: **Insert 4.5**




```
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

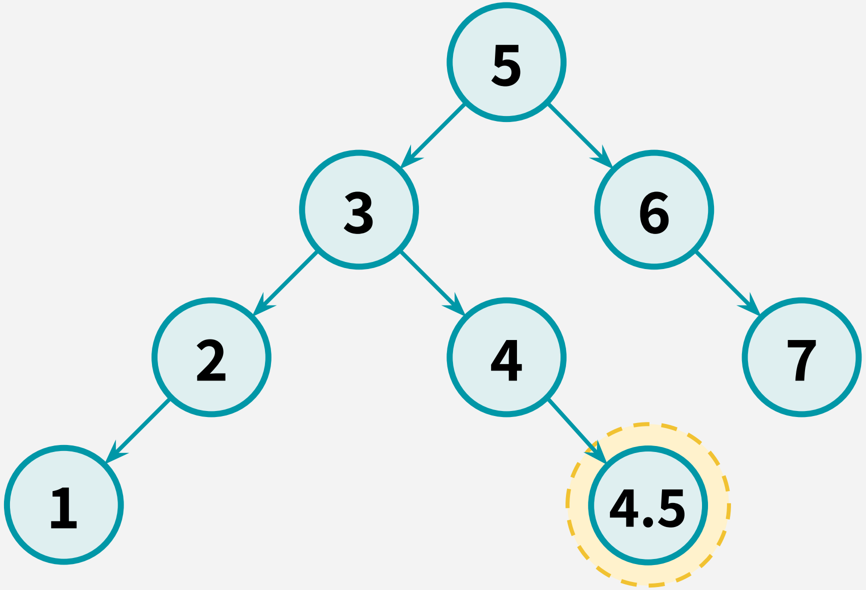
Example: **Insert 4.5**




```
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
```

INSERT in BSTs

Example: **Insert 4.5**

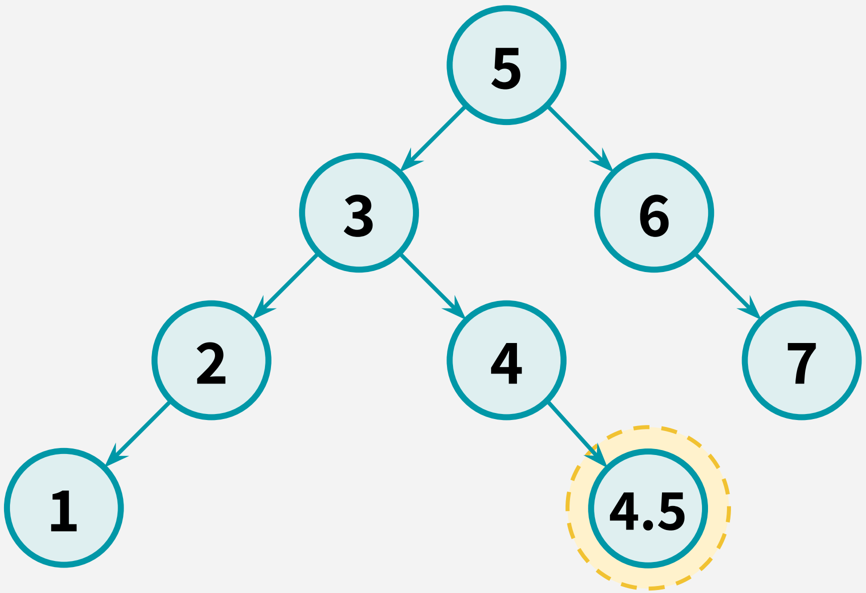



```
INSERT(root, key):
    x = SEARCH(root, key)
    node = new node with key
    if key < x.key:
        x.left = node
    if key > x.key:
        x.right = node
    if key = x.key:
        return
```

What's the runtime?

INSERT in BSTs

Example: **Insert 4.5**

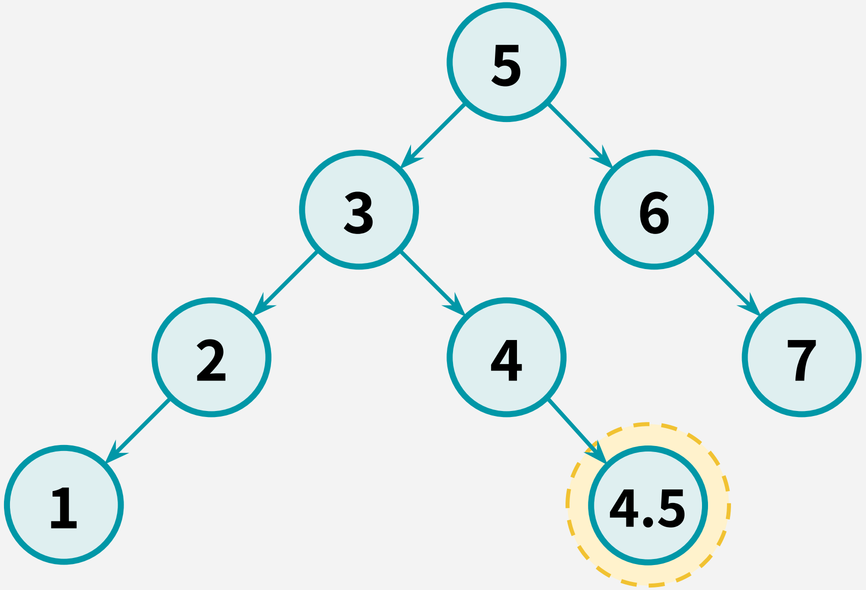



```
INSERT(root, key):
    x = SEARCH(root, key)
    node = new node with key
    if key < x.key:
        x.left = node
    if key > x.key:
        x.right = node
    if key = x.key:
        return
```

Runtime of **INSERT** = runtime of **SEARCH**

INSERT in BSTs

Example: **Insert 4.5**

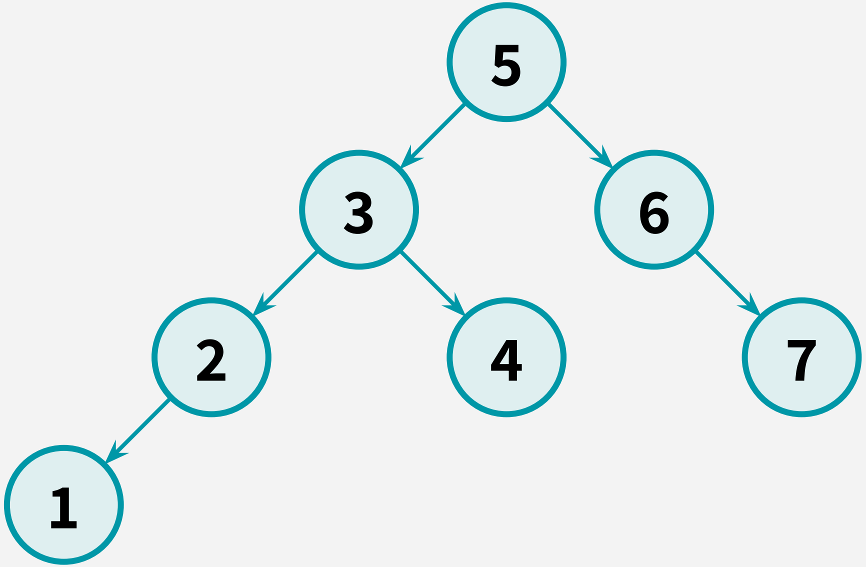



```
INSERT(root, key):
    x = SEARCH(root, key)
    node = new node with key
    if key < x.key:
        x.left = node
    if key > x.key:
        x.right = node
    if key = x.key:
        return
```

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

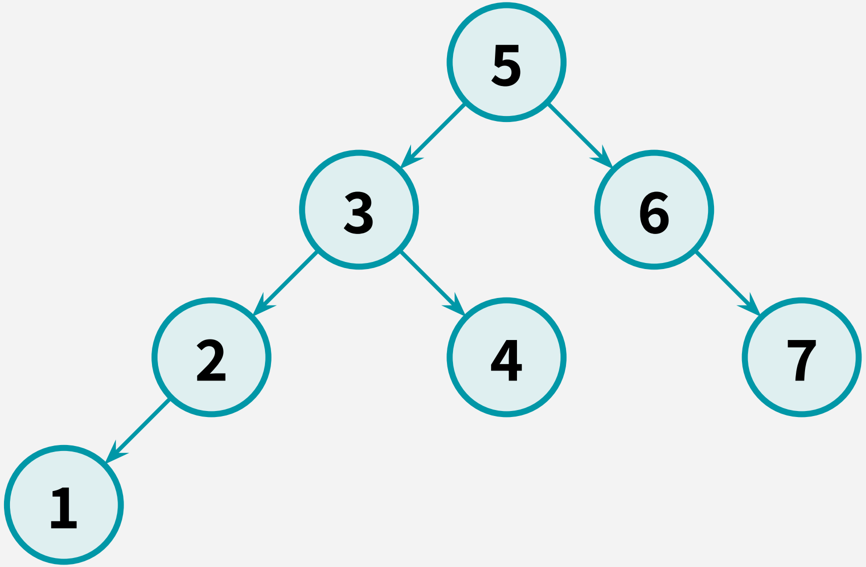
سوال؟

DELETE in BSTs




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

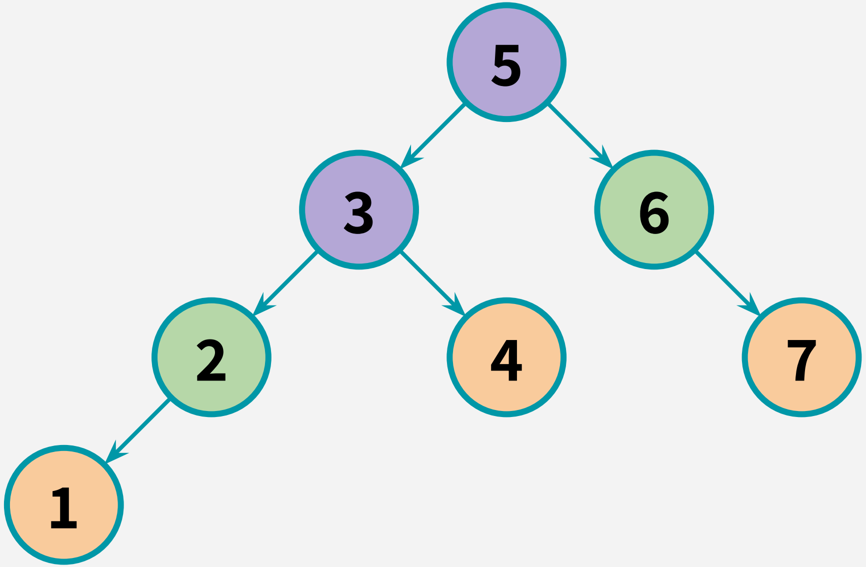
DELETE in BSTs




```
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

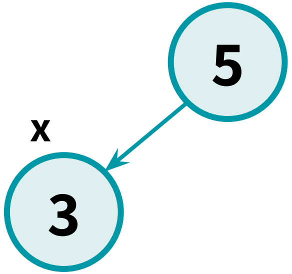
DELETE in BSTs




```
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



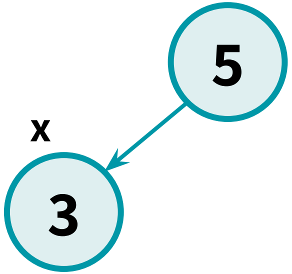
CASE 2: x has 1 child

CASE 3: x has 2 children

DELETE in BSTs

CASE 1: x is a leaf

Just delete x!



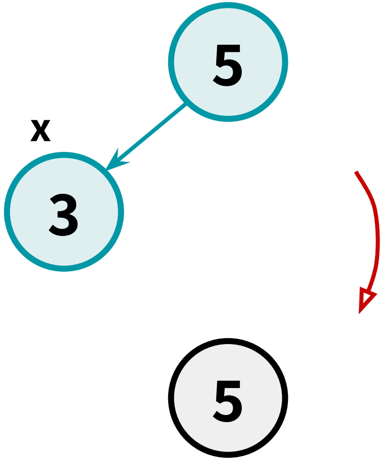
CASE 2: x has 1 child

CASE 3: x has 2 children

DELETE in BSTs

CASE 1: x is a leaf

Just delete x!



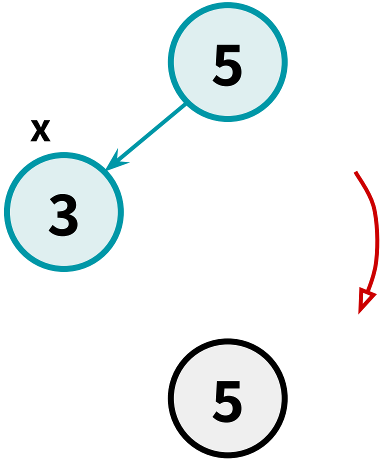
CASE 2: x has 1 child

CASE 3: x has 2 children

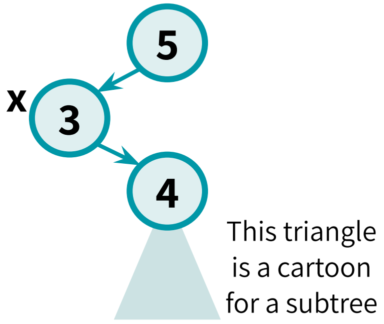
DELETE in BSTs

CASE 1: x is a leaf

Just delete x!



CASE 2: x has 1 child

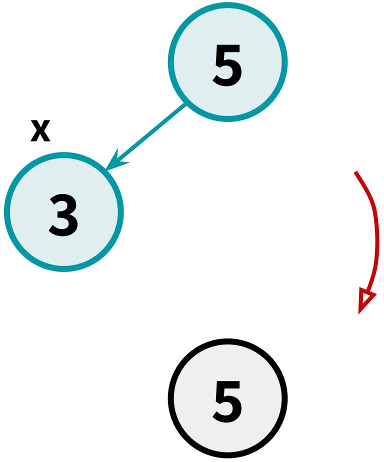


CASE 3: x has 2 children

DELETE in BSTs

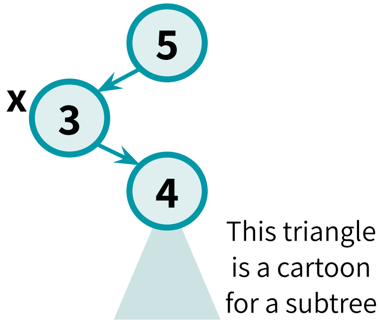
CASE 1: x is a leaf

Just delete x!



CASE 2: x has 1 child

Move its child up!

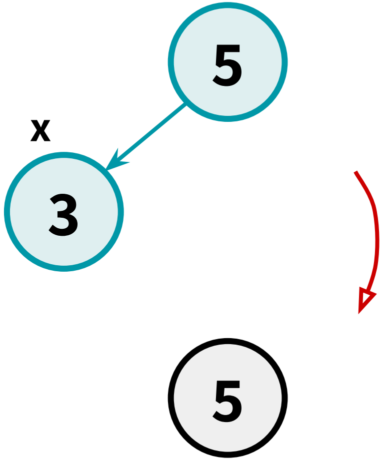


CASE 3: x has 2 children

DELETE in BSTs

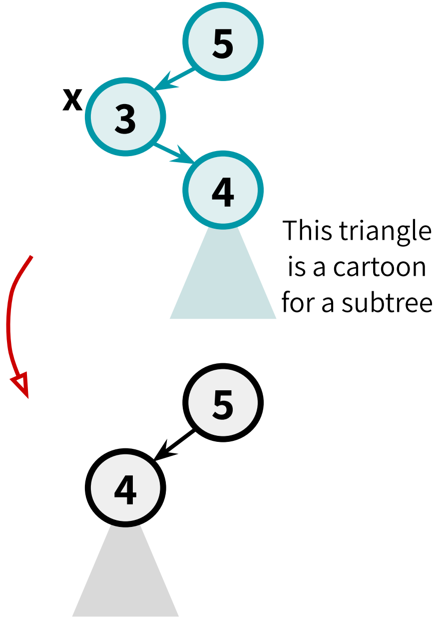
CASE 1: x is a leaf

Just delete x!



CASE 2: x has 1 child

Move its child up!

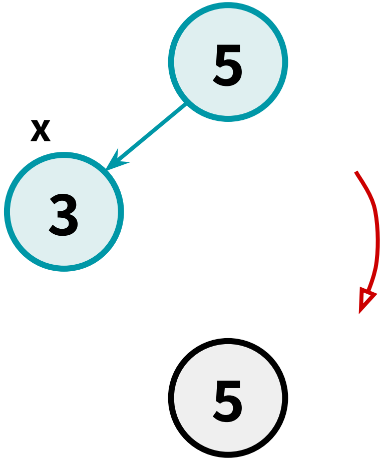


CASE 3: x has 2 children

DELETE in BSTs

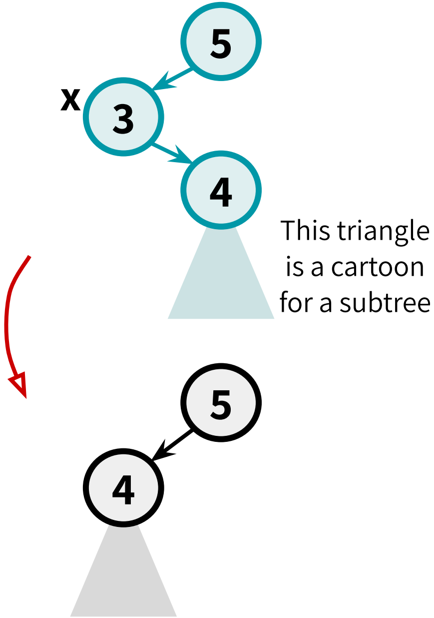
CASE 1: x is a leaf

Just delete x!

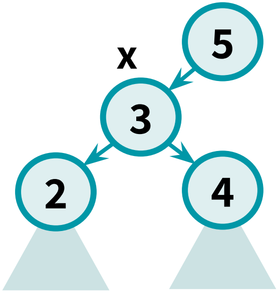


CASE 2: x has 1 child

Move its child up!



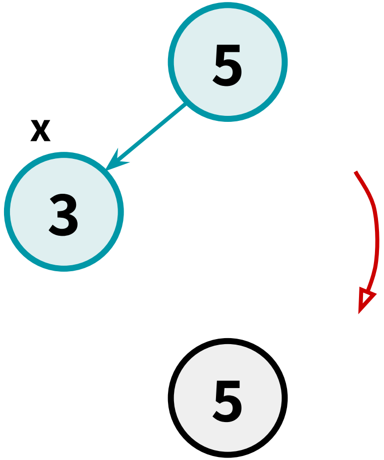
CASE 3: x has 2 children



DELETE in BSTs

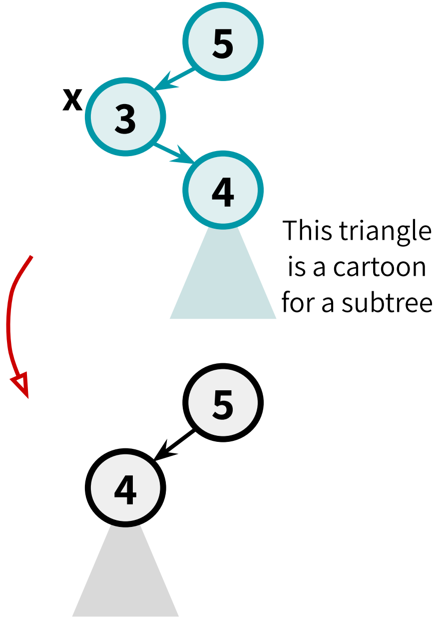
CASE 1: x is a leaf

Just delete x!



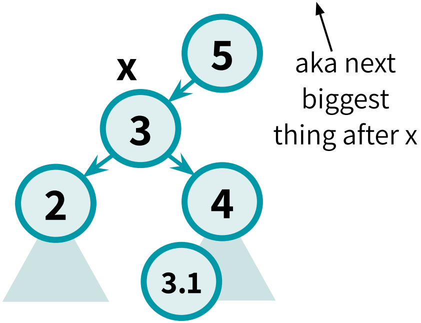
CASE 2: x has 1 child

Move its child up!



CASE 3: x has 2 children

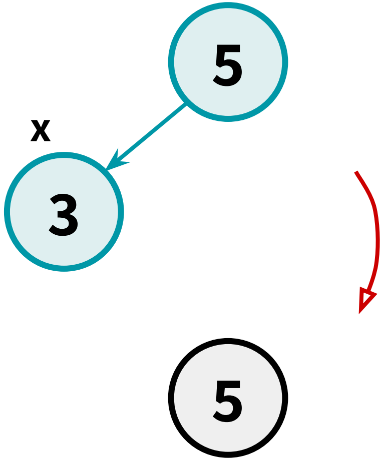
Replace x with its successor



DELETE in BSTs

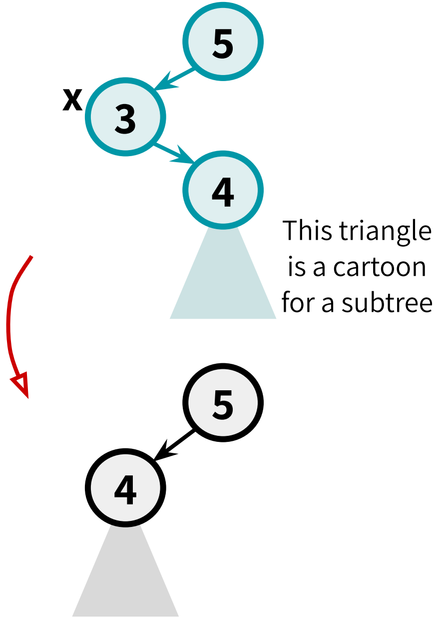
CASE 1: x is a leaf

Just delete x!



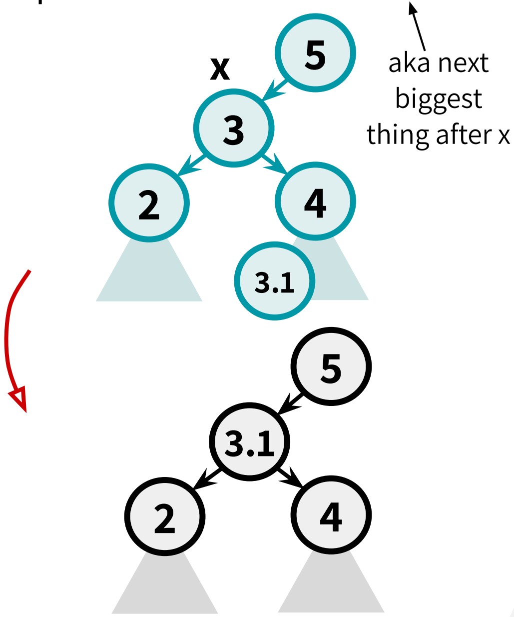
CASE 2: x has 1 child

Move its child up!



CASE 3: x has 2 children

Replace x with its successor



DELETE in BSTs

CASE 1: x is a leaf

CASE 2: x has 1 child

Details for CASE 3:

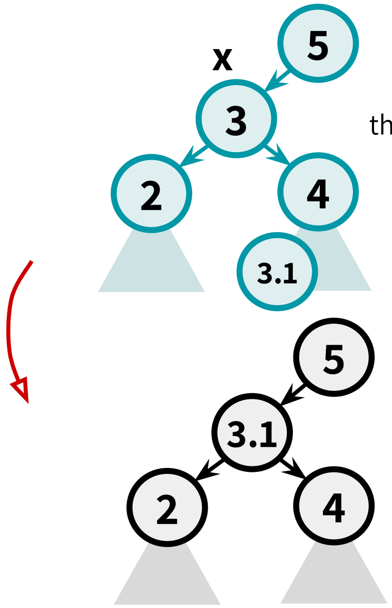
This maintains the BST property!

How do we find the immediate successor?

CASE 3: x has 2 children

Replace x with its successor

aka next biggest thing after x



DELETE in BSTs

CASE 1: x is a leaf

CASE 2: x has 1 child

Details for CASE 3:

This maintains the BST property!

How do we find the immediate successor?

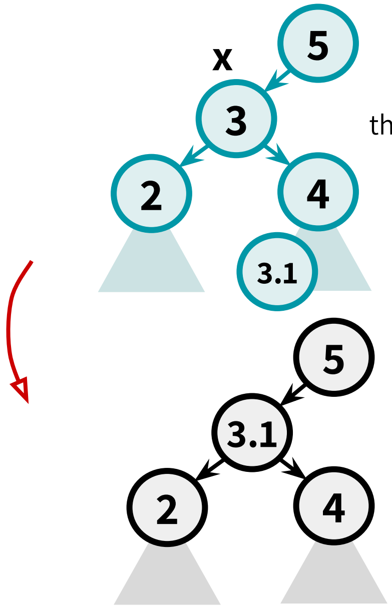
SEARCH for 3 in the subtree under 3.right

e
n
e

CASE 3: x has 2 children

Replace x with its successor

aka next biggest thing after x



DELETE in BSTs

CASE 1: x is a leaf

CASE 2: x has 1 child

e
n
e

Details for CASE 3:

This maintains the BST property!

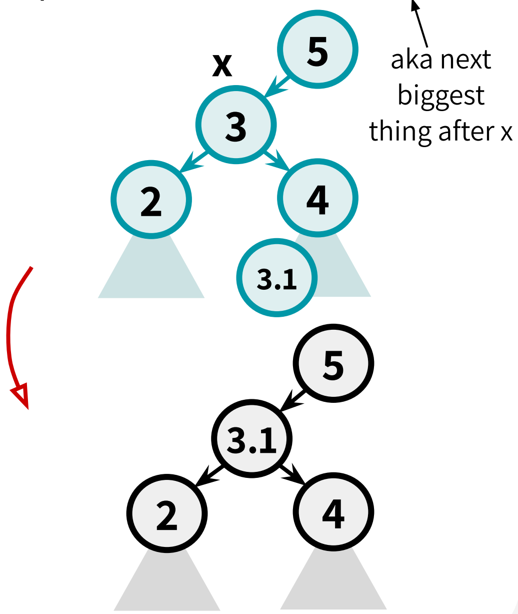
How do we find the immediate successor?

SEARCH for 3 in the subtree under 3.right

How do we remove it when we find it?

CASE 3: x has 2 children

Replace x with its successor



DELETE in BSTs

CASE 1: x is a leaf

CASE 2: x has 1 child

Details for CASE 3:

This maintains the BST property!

How do we find the immediate successor?

SEARCH for 3 in the subtree under 3.right

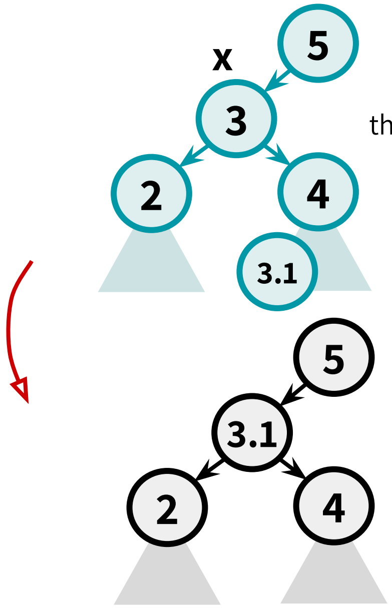
How do we remove it when we find it?

If [3.1] has 0 or 1 children, do CASE 1 or 2.

CASE 3: x has 2 children

Replace x with its successor

aka next biggest thing after x



DELETE in BSTs

CASE 1: x is a leaf

CASE 2: x has 1 child

Details for CASE 3:

This maintains the BST property!

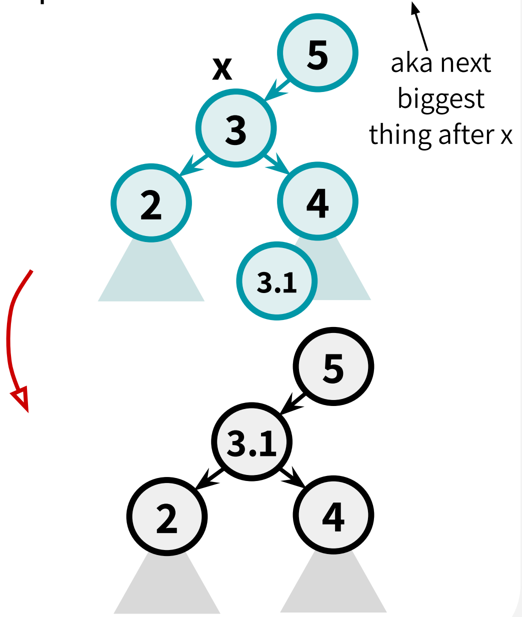
How do we find the immediate successor?
SEARCH for 3 in the subtree under 3.right

How do we remove it when we find it?
If [3.1] has 0 or 1 children, do CASE 1 or 2.

What if [3.1] has two children?

CASE 3: x has 2 children

Replace x with its successor



DELETE in BSTs

CASE 1: x is a leaf

CASE 2: x has 1 child

Details for CASE 3:

This maintains the BST property!

How do we find the immediate successor?

SEARCH for 3 in the subtree under 3.right

How do we remove it when we find it?

If [3.1] has 0 or 1 children, do CASE 1 or 2.

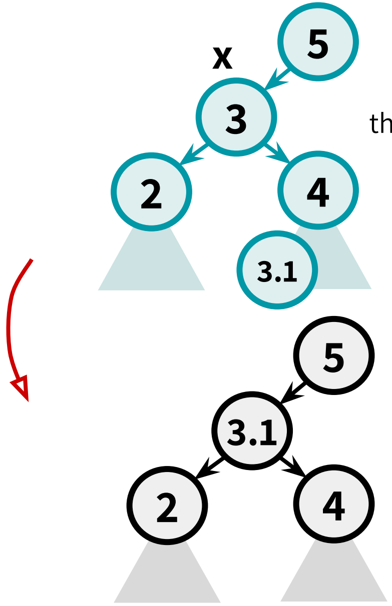
What if [3.1] has two children?

It doesn't! Otherwise it's not the immediate successor.

CASE 3: x has 2 children

Replace x with its successor

aka next biggest thing after x



سوال؟

RUNTIME OF SEARCH/INSERT/DELETE

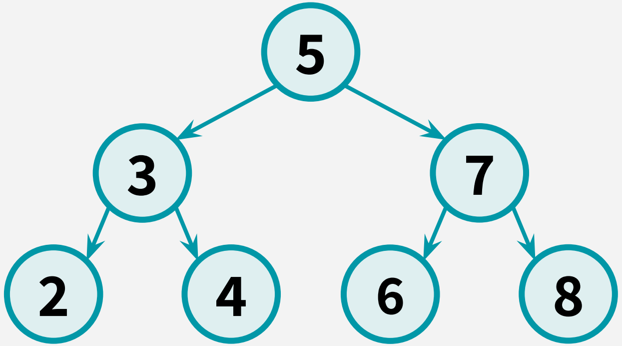
INSERT and **DELETE** both call **SEARCH** (and then do some O(1)-time operation)

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)**

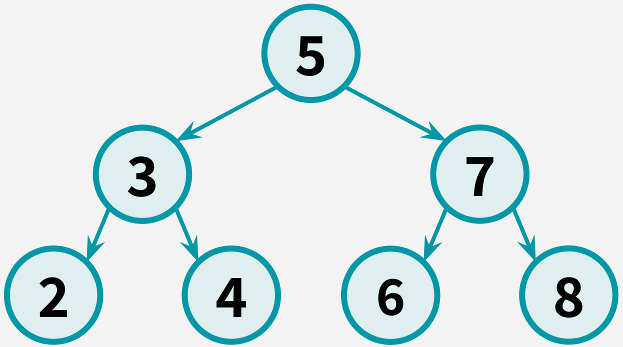


Sometimes SEARCH takes **O(log n)**

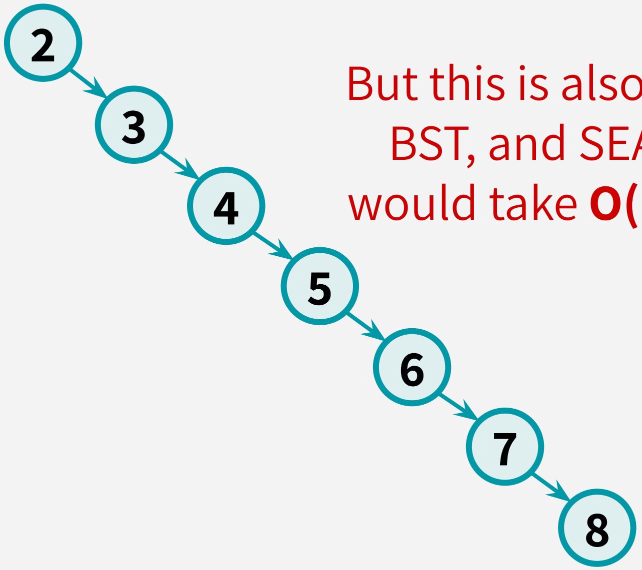
RUNTIME OF SEARCH/INSERT/DELETE

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)

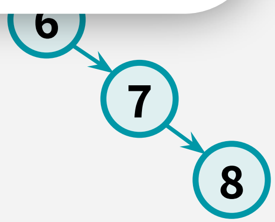
Runtime of **SEARCH = O(height)**

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

We like balanced trees… will introduce

SELF-BALANCING BINARY SEARCH TREE!

o a valid
EARCH
(n) here



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

سوال؟