Brief guide to Binary Search Trees

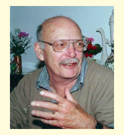
What they are

How they are implemented

Binary Search Trees (BSTs)

Fairly fast search, insertion, deletion

maximum, minimum, successor, predecessor

Creators: Dumey (1952) Wheeler (1957) 
Berners-Lee (1959)  Windley (1960)
Booth & Colin (1960)   Hibbard (1962) 

Complexity (for n data items)

Typically, binary search trees are fairly fast:

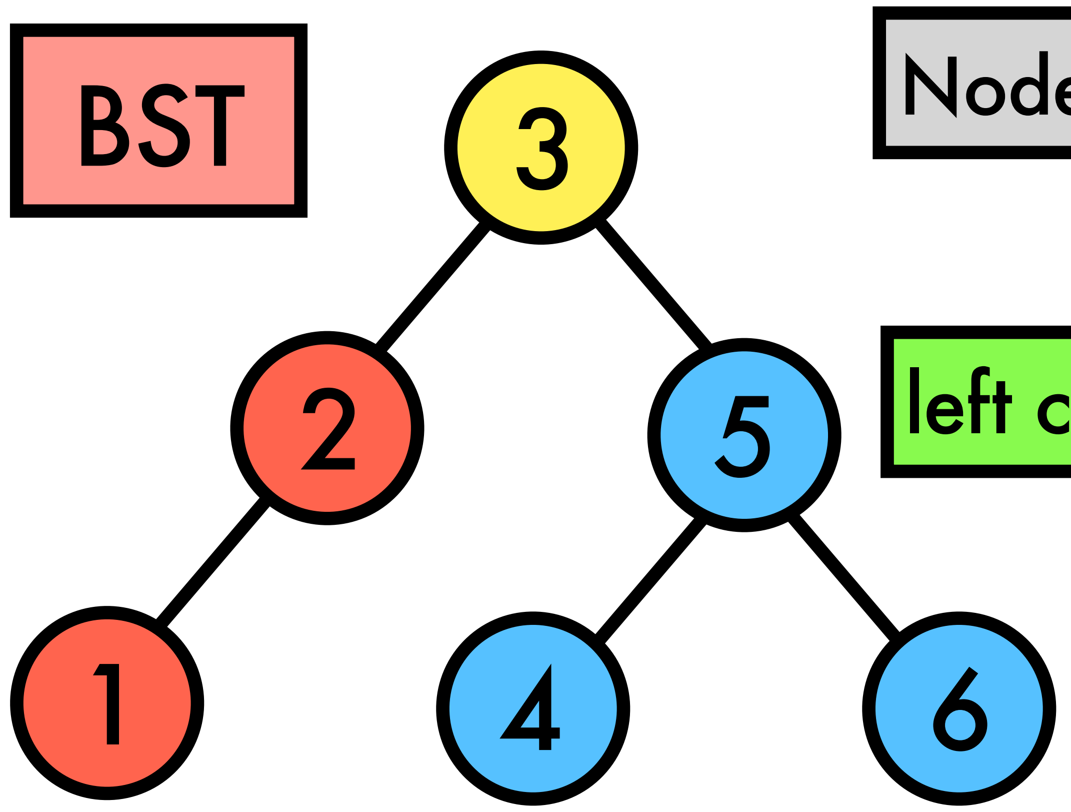
Avg. case: search, insert, delete $\rightarrow O(\log n)$

Worst case: search, insert, delete $\rightarrow \Theta(n)$

search, insert, delete $\rightarrow O(h)$ (h BST height)

Storage of binary search trees: $\Theta(n)$

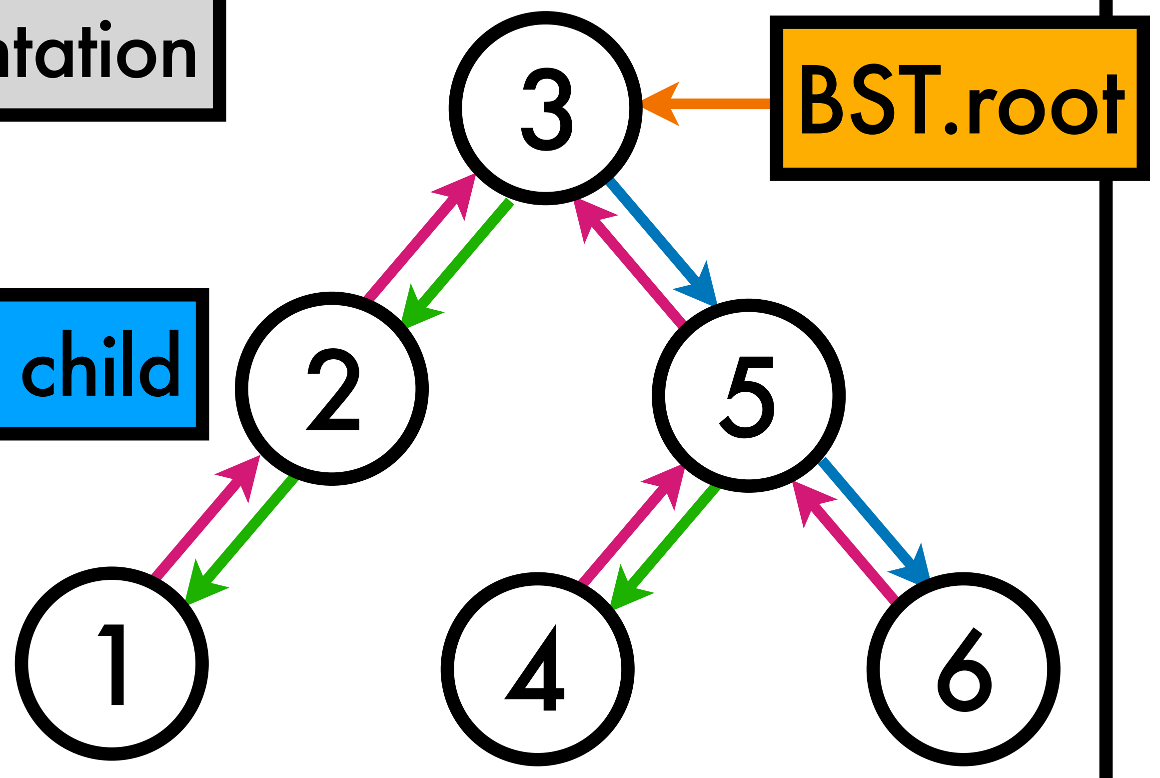
Suits | Abstract Data Types | Set | Map | Priority Queue



Node implementation

parent

left child | right child



Binary Search Tree Property: for each node u, any node l in its left subtree satisfies $l.\mathrm{key} \leq u.\mathrm{key}$, any node r in its right subtree satisfies $r.\mathrm{key} \geq u.\mathrm{key}$

Note: this definition allows duplicate keys

References/Notes/Image credits:
(History of Binary Search Trees) D. E. Knuth, "The art of computer programming, vol. 3: sorting and searching", Chap 6.2.2 (1974)
(Wheeler/Berners-Lee) A. Douglas, "Techniques for the recording of, and reference to data in a computer", The Computer Journal (1959)
(D. Wheeler) https://en.wikipedia.org/wiki/David_Wheeler_(computer_scientist)#/media/File:EDSAC_(14)_(cropped).jpg
(C. Berners-Lee) https://www.bl.uk/voices-of-science/interviewees/conway-berners-lee
A. Booth and A. Colin, "On the efficiency of a new method of dictionary construction", Information and Control (1960)
(Booth) https://www.computerhope.com/people/andrew_booth.htm
(Andrew Colin) https://www.heraldscotland.com/opinion/16998308.obituary–andrew-colin-professor-computer-science/
T. Hibbard, "Some combinatorial properties of certain trees with applications to searching and sorting", JACM (1962)
(Hibbard) http://math.oxford.emory.edu/site/cs171/hibbardDeletion/
(Binary Search Trees) T. Cormen et al., "Introduction to algorithms", Chap 12.1, MIT press, (2022)

Tree traversals

Traversal algorithms

A traversal algorithm aims to "process" each node in the tree exactly once

The simplest traversals use depth-first-search (go deeper first, rather than "breadth first")

inorder | preorder | postorder

Inorder traversal: process each node in-between visiting left subtree and right subtree

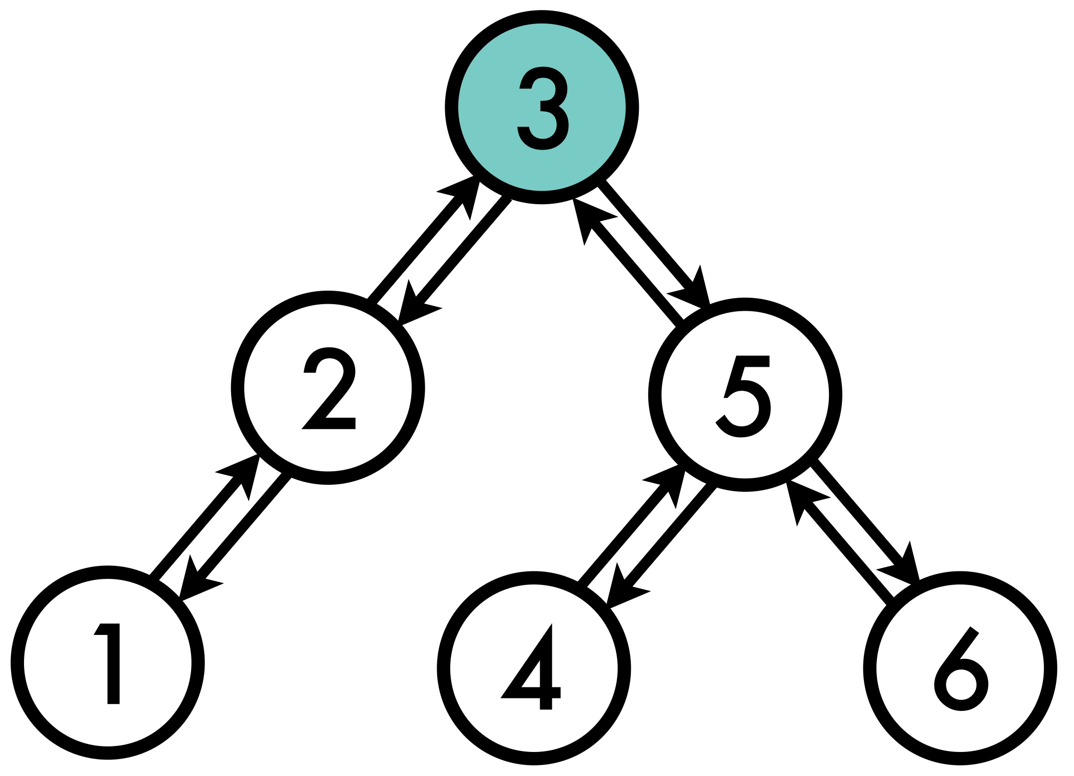
Recursive implementation:

```
def inorder(u):
    if u:
        inorder(u.left)
        print(u.key)
        inorder(u.right)
```

Inorder traversal of BST

print out: 1 2 3 4 5 6

Note that keys were printed in order!




```
def preorder(u):
    if u is not None:
        print(u.key)
        preorder(u.left)
        preorder(u.right)
```

print out: 3 2 1 5 4 6

```
def postorder(u):
    if u is not None:
        postorder(u.left)
        postorder(u.right)
        print(u.key)
```

print out: 1 2 4 6 5 3

Traversals are $\Theta(n)$ - call themselves twice at each node (left child and right child)

References/Notes/Image credits:
(Tree traversals) http://webdocs.cs.ualberta.ca/~holte/T26/tree-traversal.html
(Traversal complexity) T. Cormen et al., "Introduction to algorithms", Chap 12.1, MIT press, (2022)

Binary Search Tree Queries

Minimum/Maximum

```
def minimum(u):
    while u.left:
        u = u.left
    return u
```

E.g. argument: root
returns node with key 1

```
def maximum(u):
    while u.right:
        u = u.right
    return u
```

E.g. argument: root
returns node with key 9

Search

```
def search(u, key):
    while u and key != u.key:
        if key < u.key:
            u = u.left
        else:
            u = u.right
    return u
```

query key: 6
returns node

query key: 7
returns None

Complexity: $O(h)$ where h is tree height

Inorder Predecessor

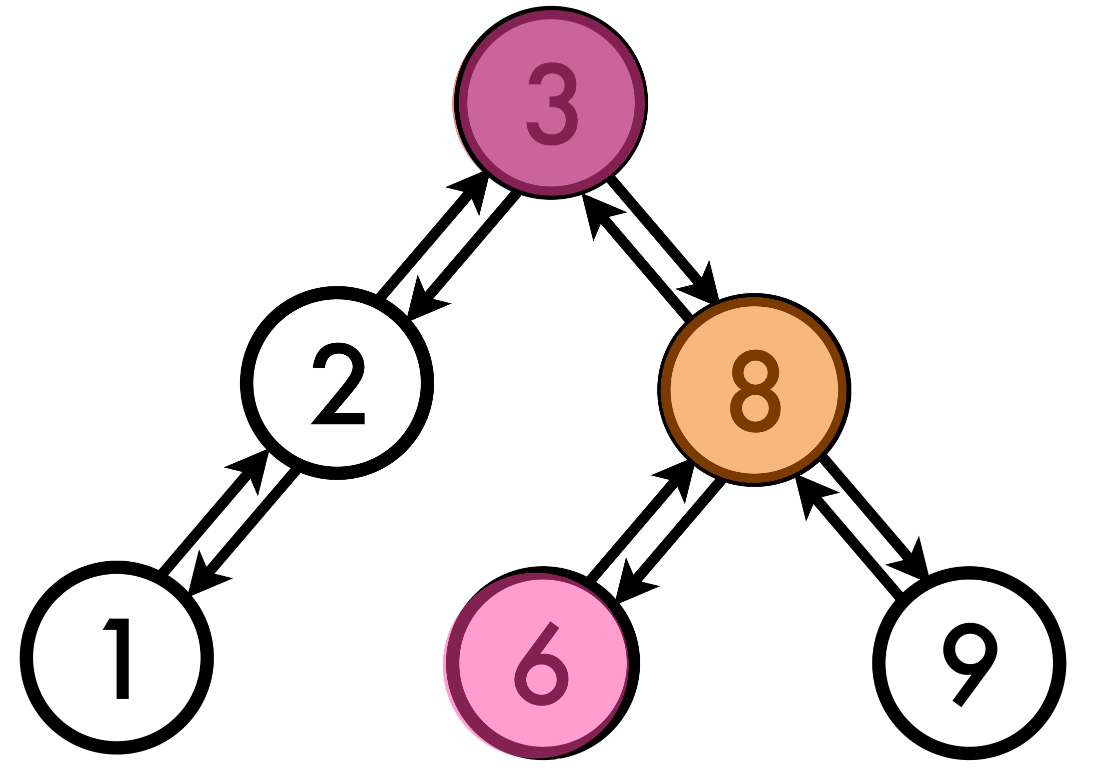
Element immediately before node in the inorder traversal ordering

Note: no key required!

```
def predecessor(u):
    if u.left:
        return maximum(u.left)
    else:
        par = u.parent
        while par and u != par.right:
            u = par
            par = par.parent
        return par
```

Note: inorder successor is symmetric

Complexity: $O(h)$ (h is tree height)



Example Binary Search Tree

Predecessor examples

First case

query node with key 3
returns node with key 2

Second case

query node with key 6
returns node with key 3

References:
(Binary search) https://webdocs.cs.ualberta.ca/~holte/T26/binary-search.html (algorithm pseudocode) https://en.wikipedia.org/wiki/Binary_search_tree#Searching (Querying a BST) T. Cormen et al., "Introduction to algorithms", Chap 12.2, MIT press, (2022)

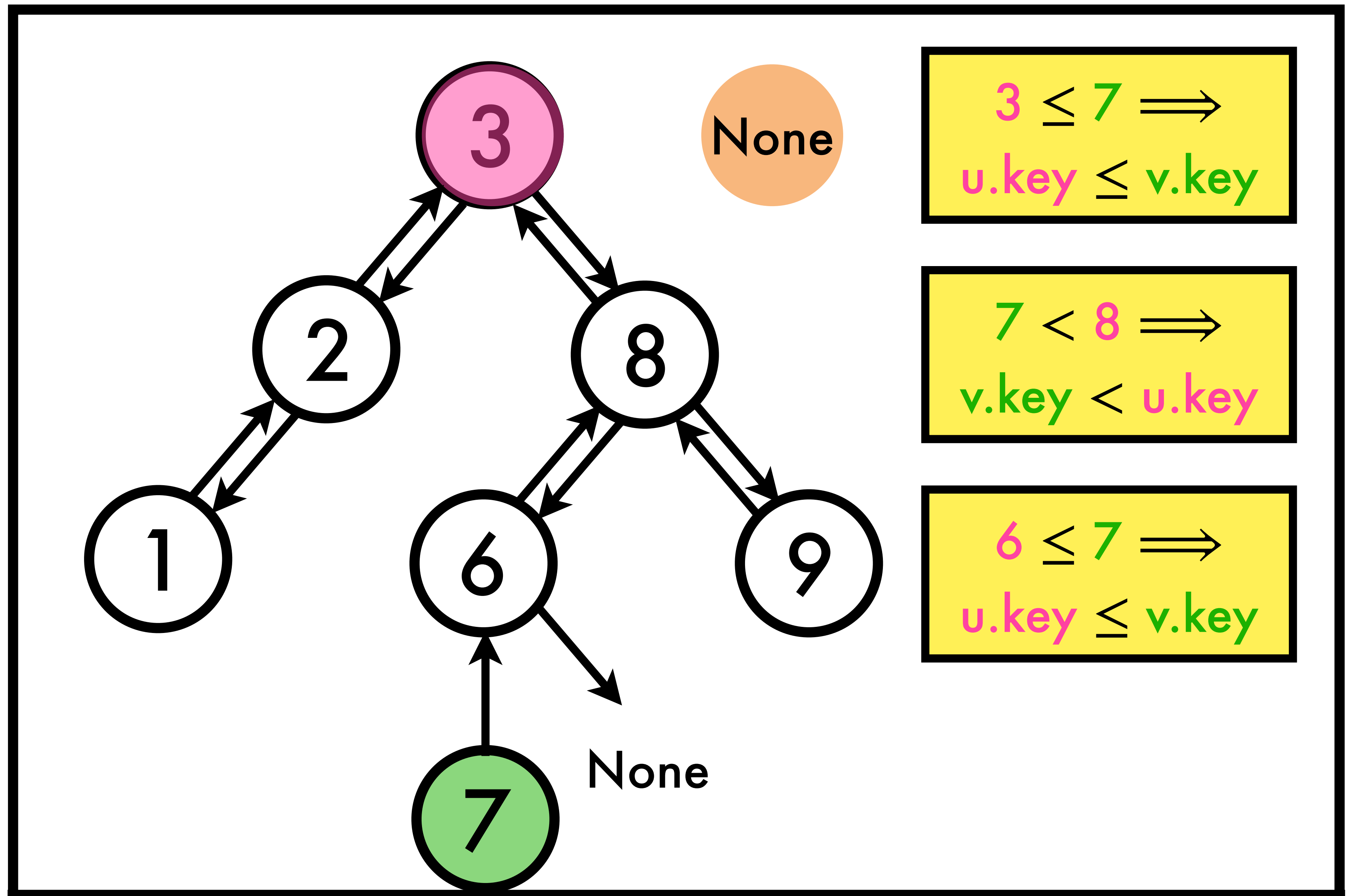
Binary Search Tree Insertion

Insertion

Insert new node `v` into binary search tree `bst`

```
def insert(bst, v):
    u = bst.root
    par = None
    while u:
        par = u
        u = u.left if v.key < u.key else u.right
    v.parent = par
    if not par: # handle case when bst was empty
        bst.root = v
    elif v.key < par.key:
        par.left = v
    else:
        par.right = v
```

Complexity: $O(h)$ where h is tree height



Example Binary Search Tree

Insert example node with key 7

We follow Cormen (BST allows duplicate keys)
If not allowed, insert() must be modified

References:
(insertion pseudocode) https://en.wikipedia.org/wiki/Binary_search_tree#Insertion
(Insertion) T. Cormen et al., "Introduction to algorithms", Chap 12.3, MIT press, (2022) (a.k.a CLRS due to the authors names)

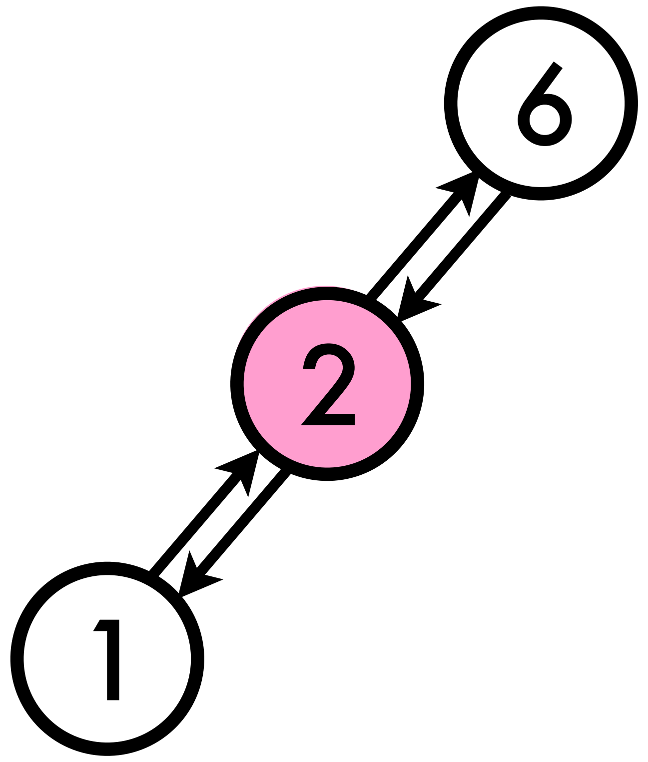
Binary Search Tree Deletion

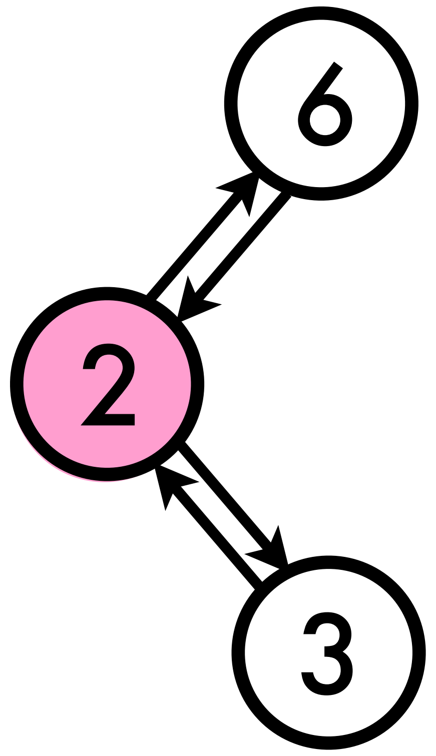
Deletion logic

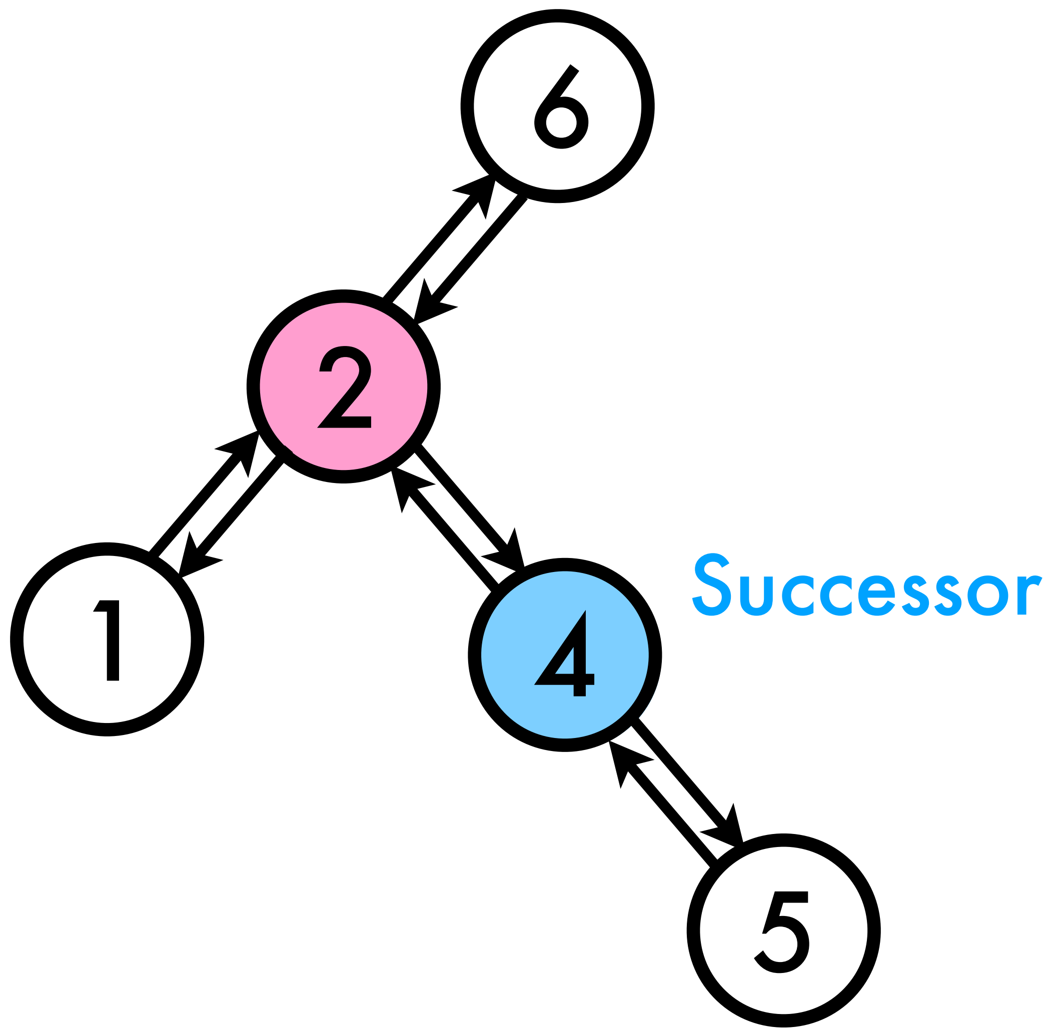
To delete a node from BST, there are four cases to consider

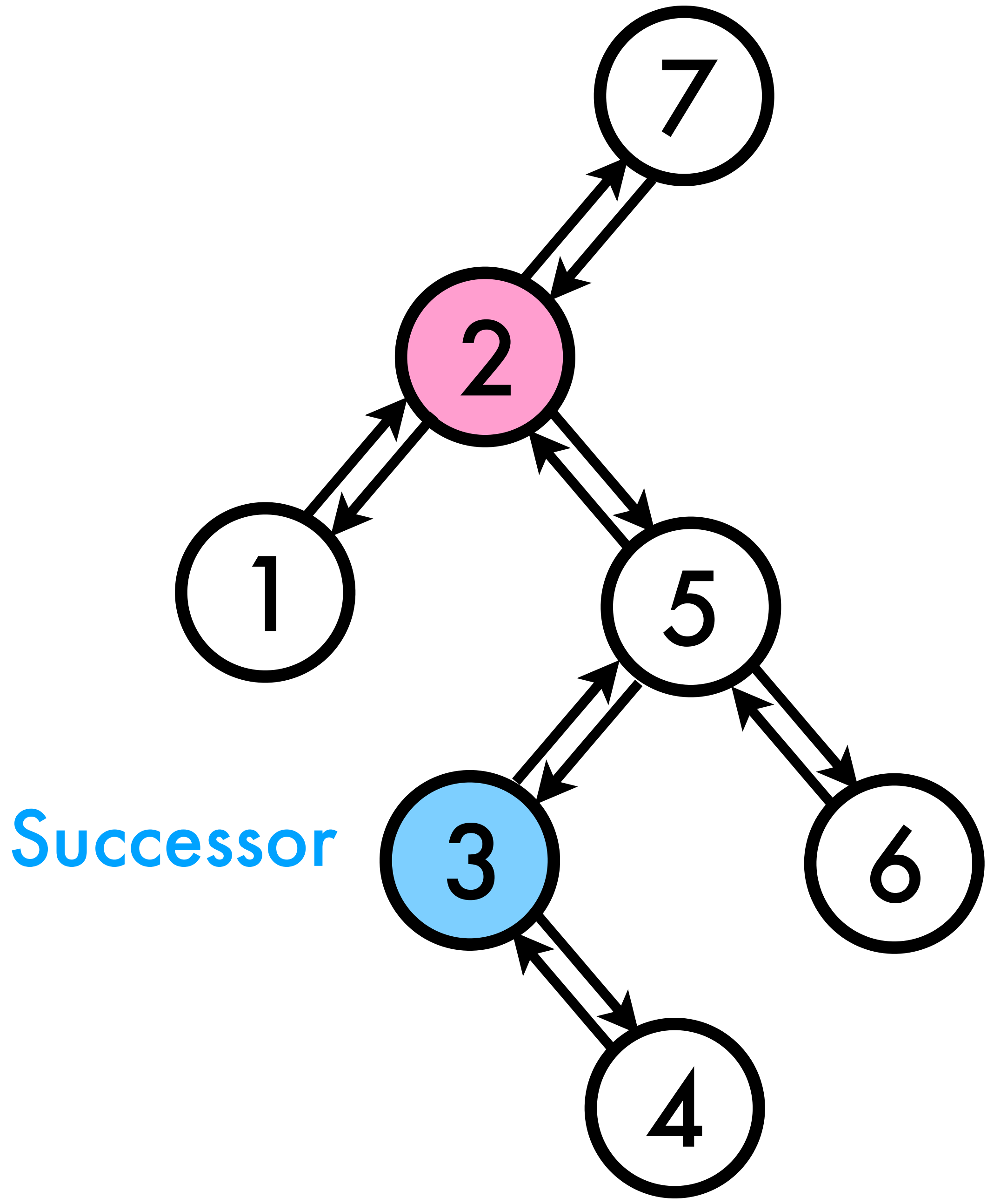
goal: preserve the BST property

In the following, we will focus on deleting the node with key 2









case 1:
no right child

case 2:
no left child

case 3: two children
successor is right child

case 4: two children
successor is **not** right child

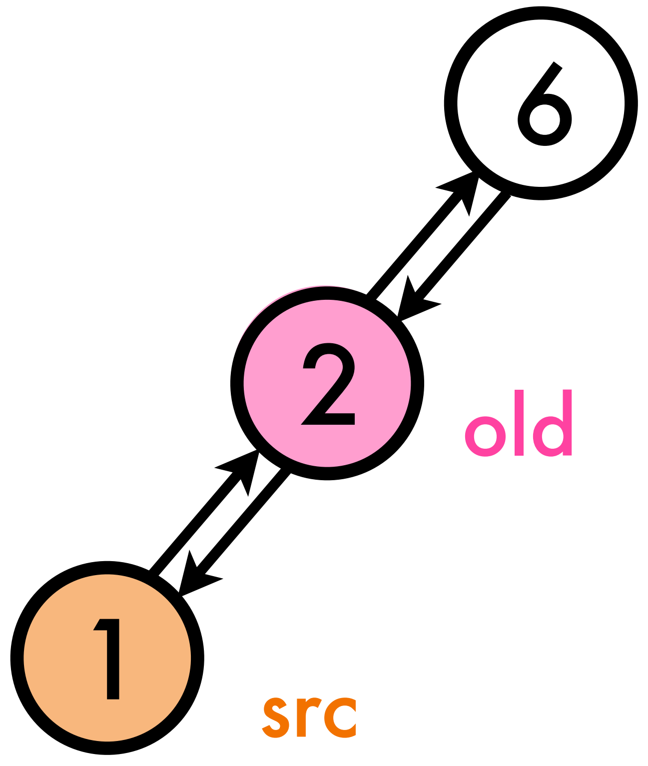
References:
https://en.wikipedia.org/wiki/Binary_search_tree#Deletion
(Deletion) T. Cormen et al., "Introduction to algorithms", Chap 12.3, MIT press, (2022)

Binary Search Tree Deletion

Deletion implementation

Delete node `u` from binary search tree `bst`

```
def delete(bst, u):
    if not u.right:
        shift_nodes(bst, u, u.left)
    elif not u.left:
        shift_nodes(bst, u, u.right)
    else:                                   # u has two children
        u_successor = minimum(u.right)
        if u_successor != u.right:
            shift_nodes(bst, u_successor, u_successor.right)
            u_successor.right = u.right
            u_successor.right.parent = u_successor
        shift_nodes(bst, u, u_successor)
        u_successor.left = u.left
        u_successor.left.parent = u_successor
```




```
def shift_nodes(bst, old, src):
    if not old.parent:
        bst.root = src
    elif old == old.parent.left:
        old.parent.left = src
    else:
        old.parent.right = src
    if src:
        src.parent = old.parent
```

Complexity: $O(h)$ where h is tree height

References:
(Deletion pseudocode - we follow this naming convention) https://en.wikipedia.org/wiki/Binary_search_tree#Deletion
(Deletion) T. Cormen et al., "Introduction to algorithms", Chap 12.3, MIT press, (2022)