CS202 – Data Structures

LECTURE-09

Binary Search Trees

BST Operations

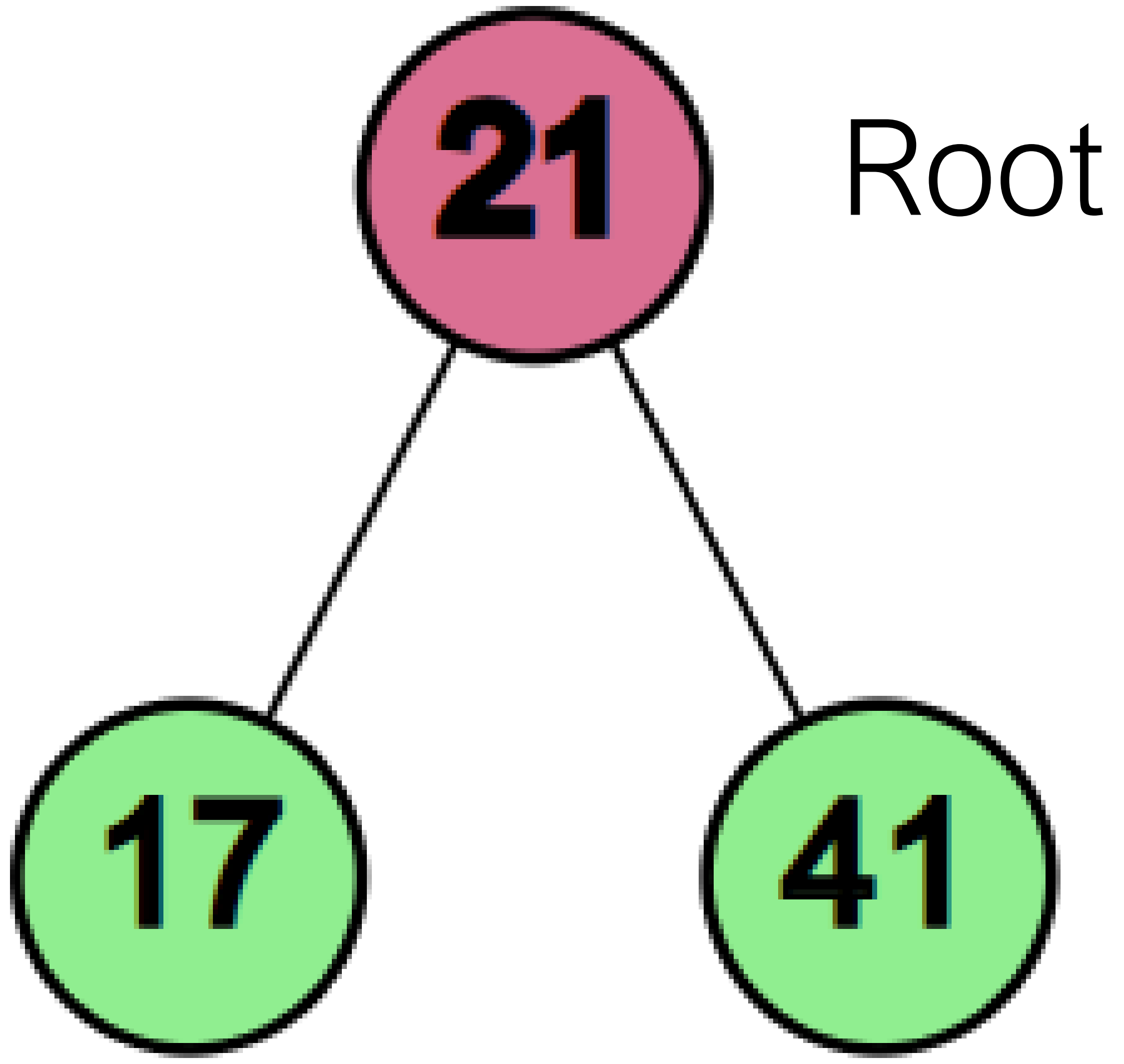
Dr. Maryam Abdul Ghafoor
Assistant Professor
Department of Computer Science, SBASSE

Agenda

- Traversals
- BST Property
  - Next greater element
- Binary Search Tree Operations
  - Search
  - Insertion
  - Deletion

Binary Search Tree (BST)

- A binary search tree (BST) is a binary tree where (for all nodes):
  - Key of the left child is smaller than the parent's key
  - Key of the right child is greater than the parent's key
  - Each node stores a unique key



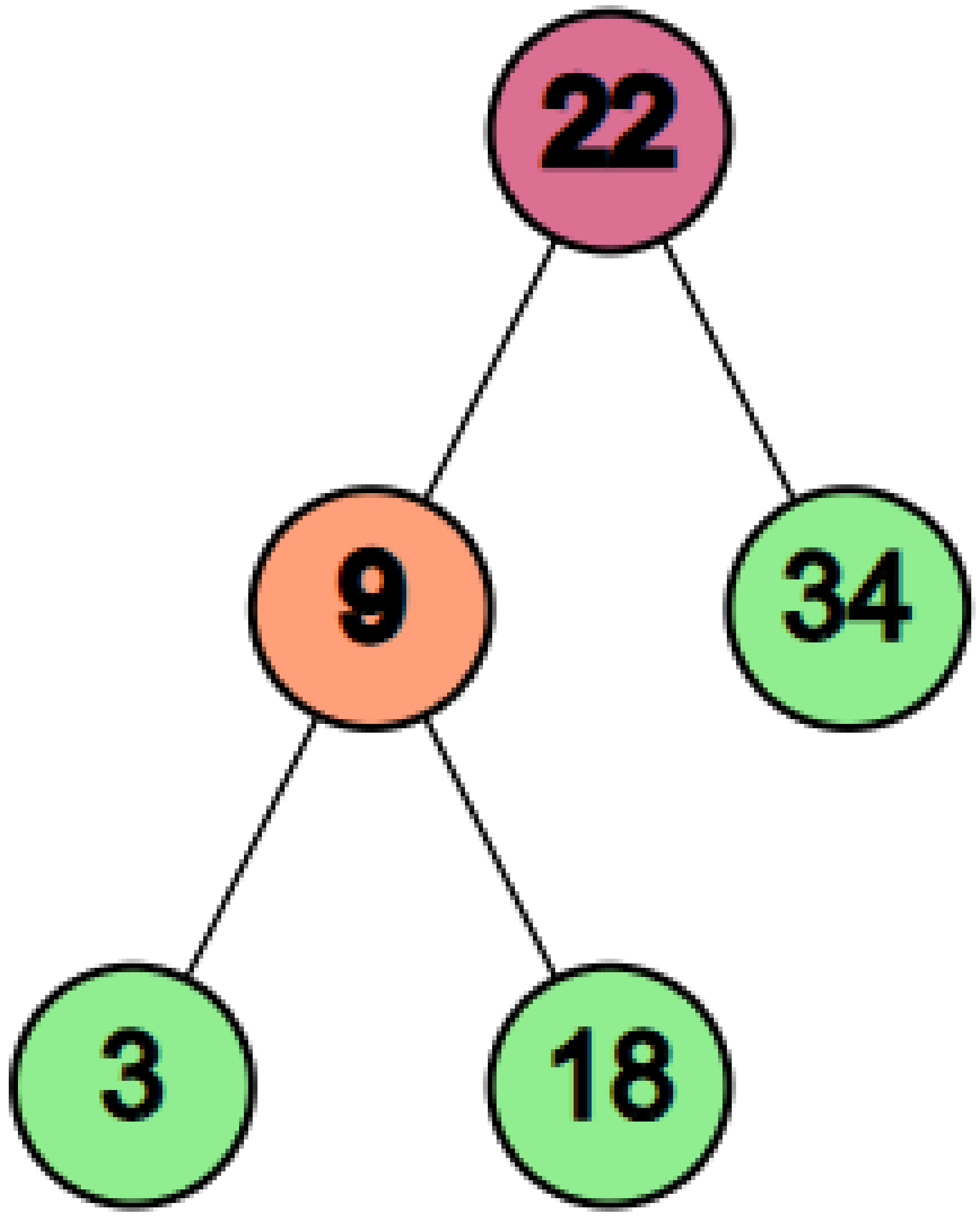
BST Implementation

```
class BST{
public:
   BST();
   ~BST();
   BSTNode* search(int key, BSTNode* t);
   BSTNode* insertNode(int key, BSTNode* t);
   Bool insert(int key);
   BSTNode* removeNode(int key, BSTNode* t);
   Bool remove(int key);
private:
   int size;
   BSTNode* root;
};
```

```
struct BSTNode{
   int key;
   string data;
   BSTNode* parent;
   BSTNode* left;
   BSTNode* right;
};
```

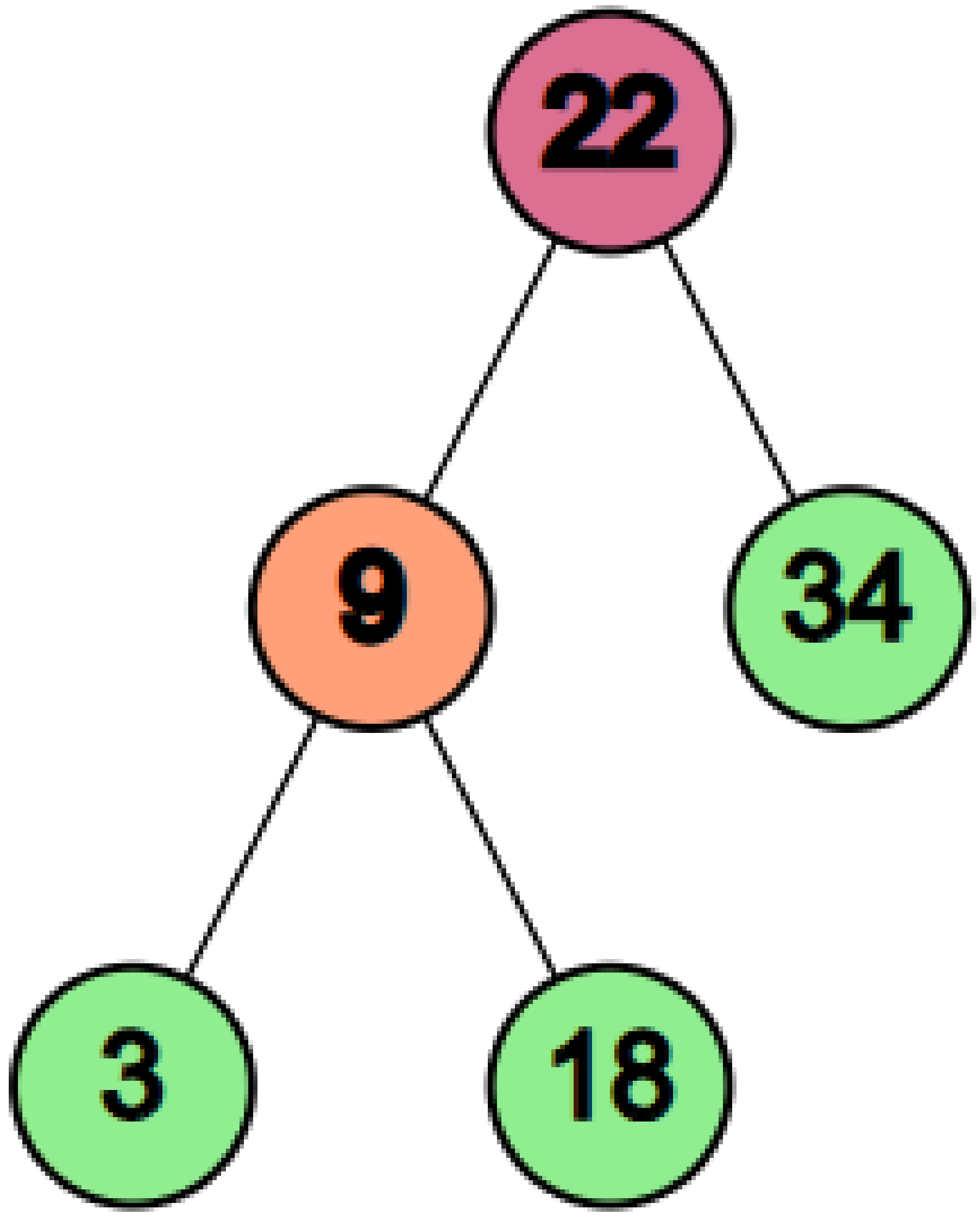
Can we have different BSTs for the same keys?

- Store keys: 22, 9, 34, 18, 3

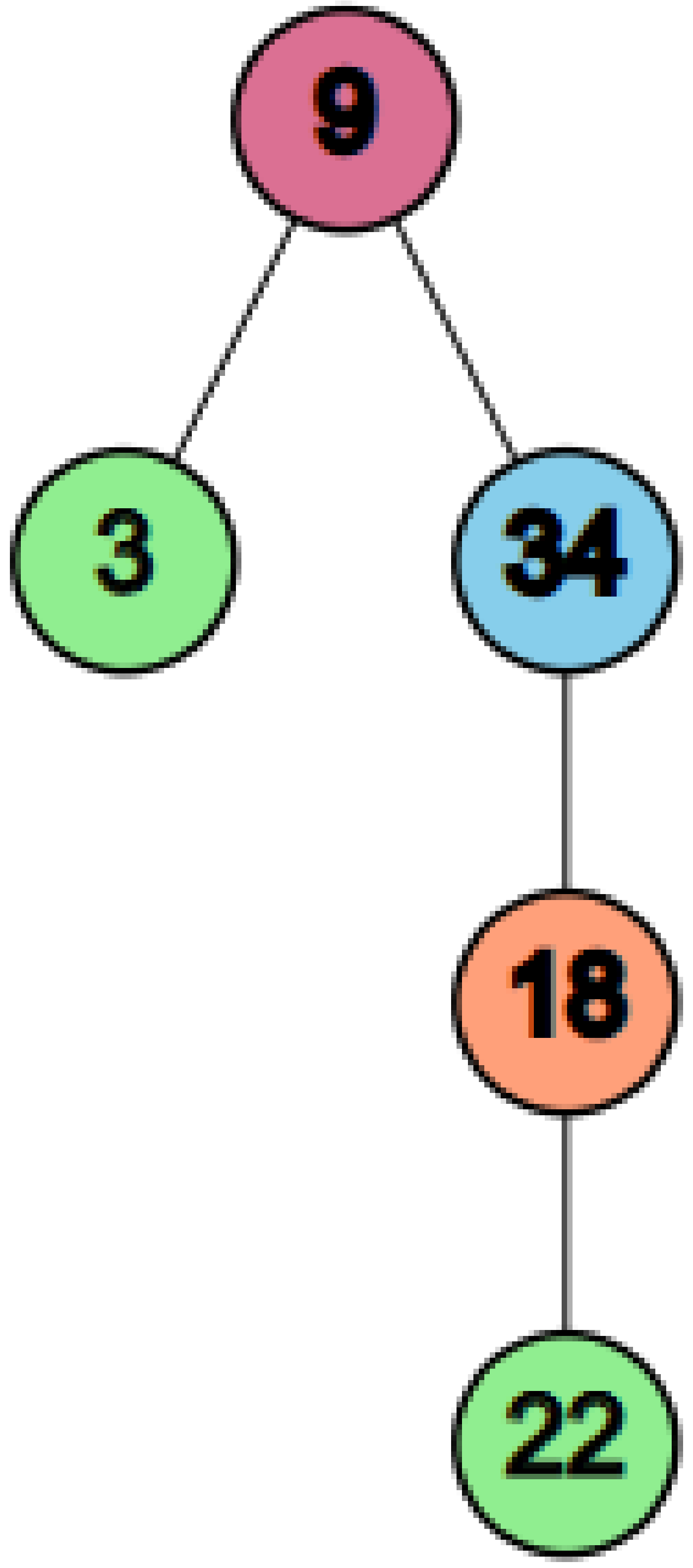


Can we have different BSTs for the same keys?

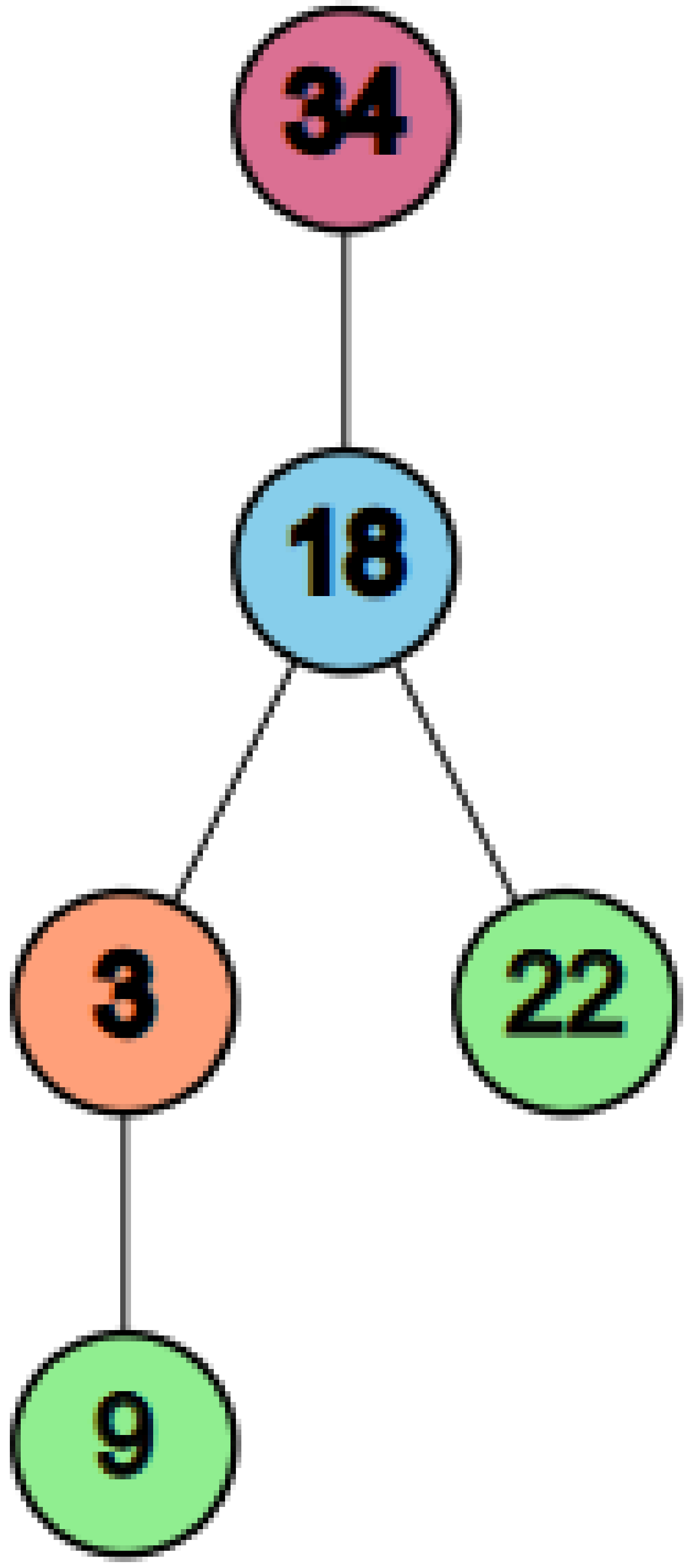
- Store keys: 22, 9, 34, 18, 3



22, 9, 34, 18,3



9, 34, 18, 22,3

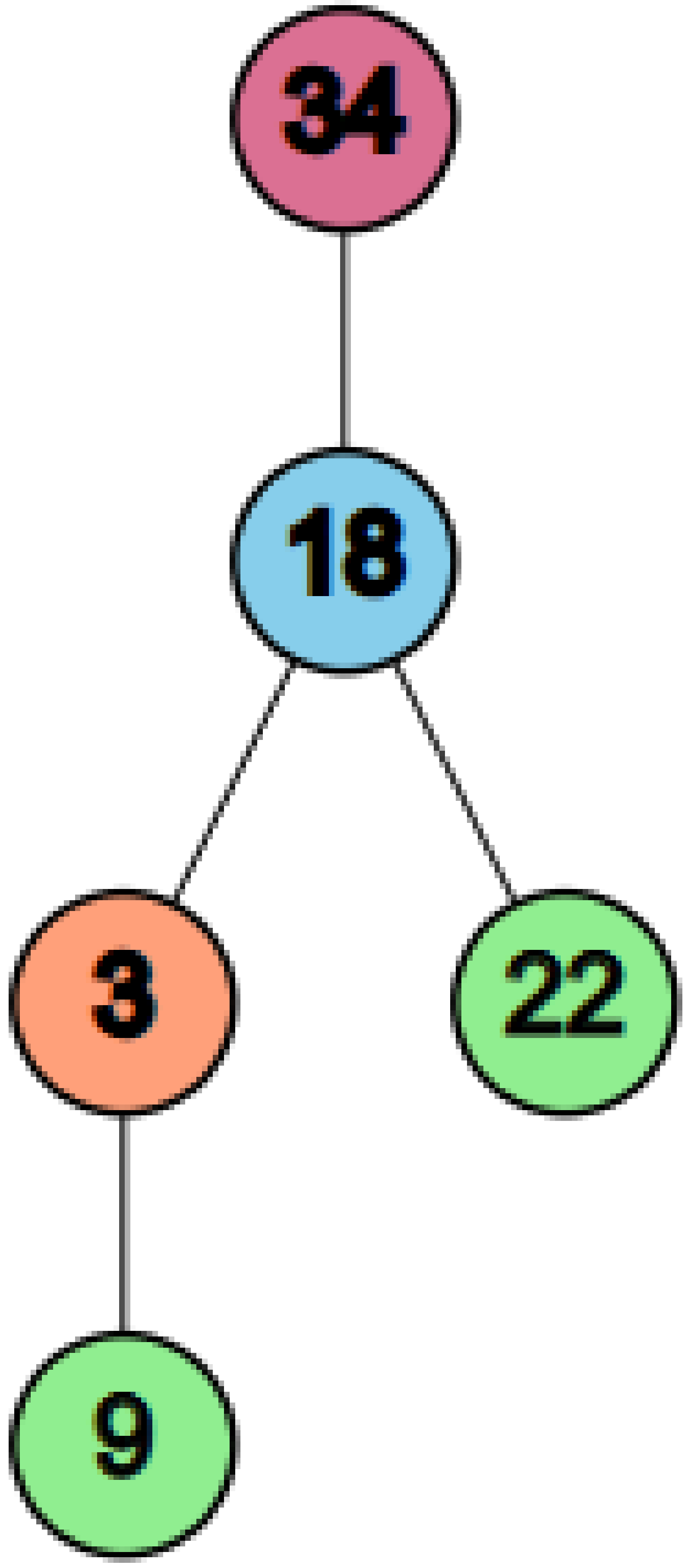


34, 18, 22,3,9

Traversals

- How can we print all keys of the following trees in sorted order?

In-order Traversal



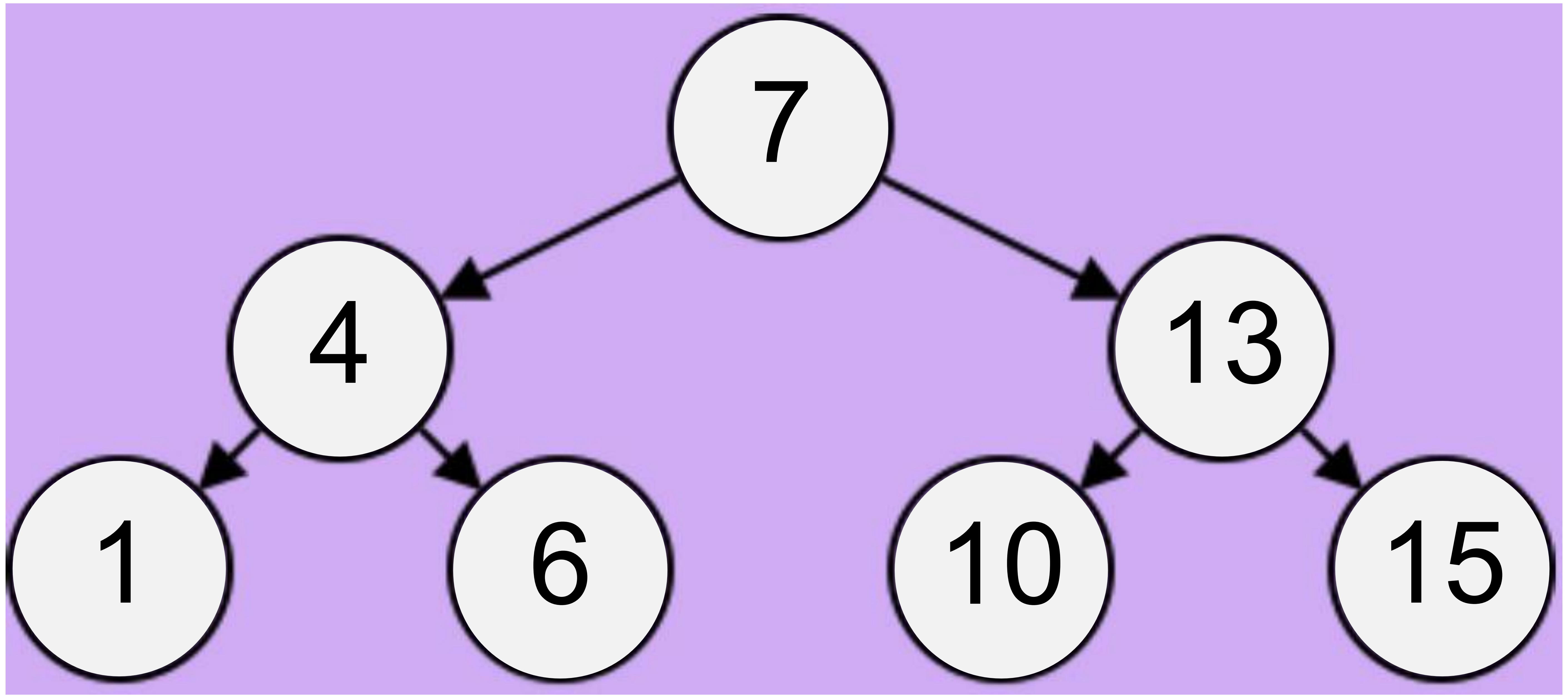
BST Operations

Fast SEARCH/INSERT/DELETE

Can we do these?

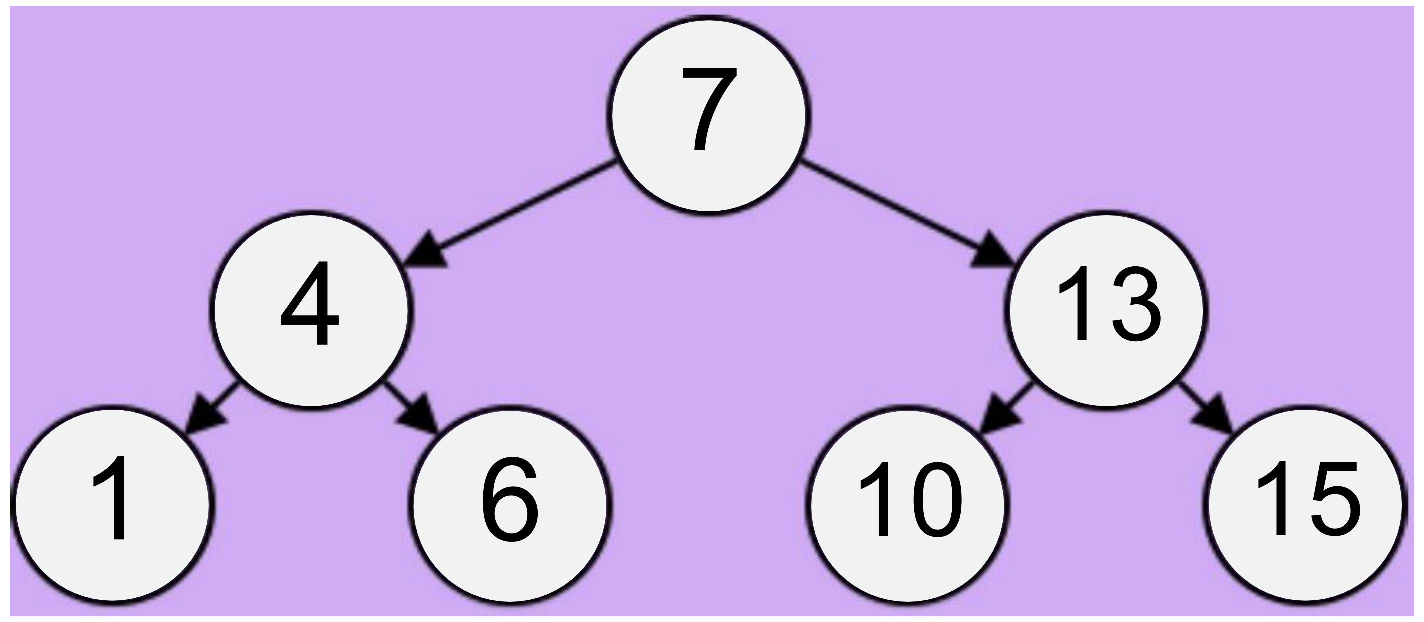
SEARCH in a Binary Search Tree (BST)

How can we do search efficiently?



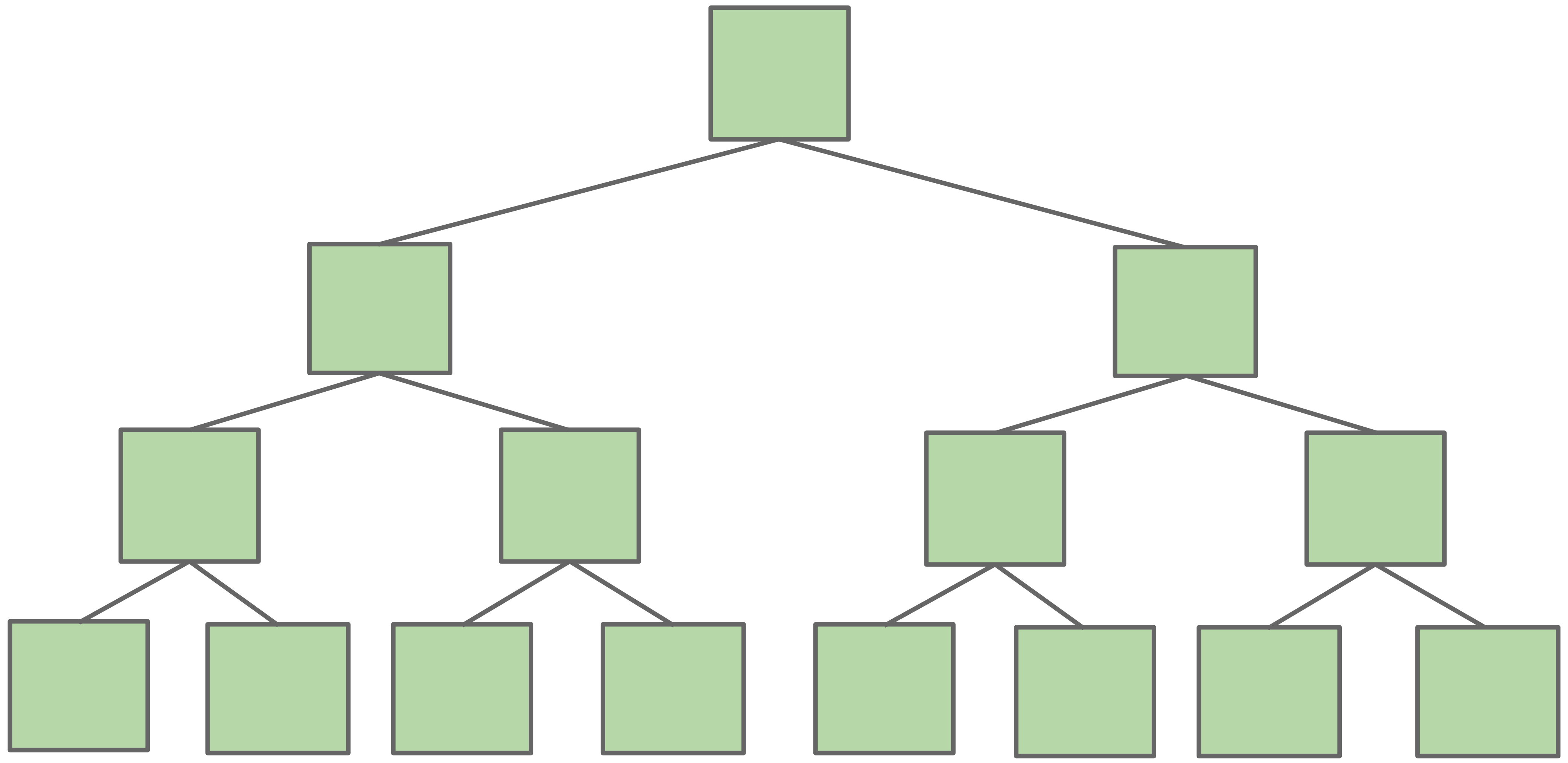
Search in BST

```
BSTNode* BST::search(int key, BSTNode* t){
if(t==NULL)
   return NULL;

else if(t->key == key) //key found
   return t;

else if(key < t->key)
   return search(key, t->left);

else
   return search(key, t->right);
}
```



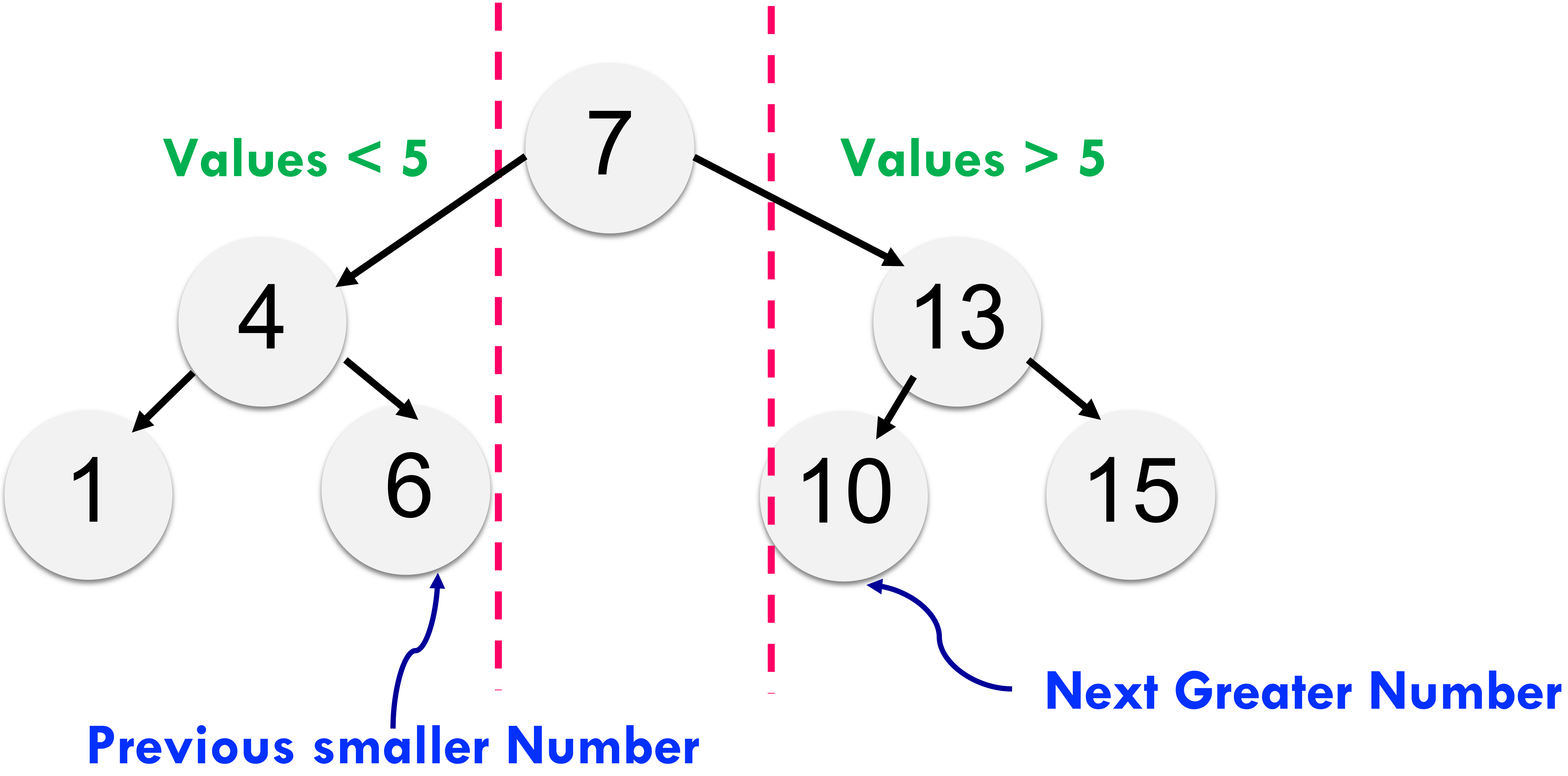
SEARCH in a Binary Search Tree (BST)

- What is the running time complexity for search on a full BST in the worst-case, where N is the number of nodes?



SEARCH in a Binary Search Tree (BST)

How can we find successor(next greater element) of a node?



Next Greater Element(NGE)

- Next greater element of n is

  If right child does not exists, then there is no NGE

  else if n → right → left == NULL

    Return n →right as NGE

  else

    The leftmost descendant(child) of the right child

- How can we find the Previous Smaller Number(PSN)/ predecessor of an element?

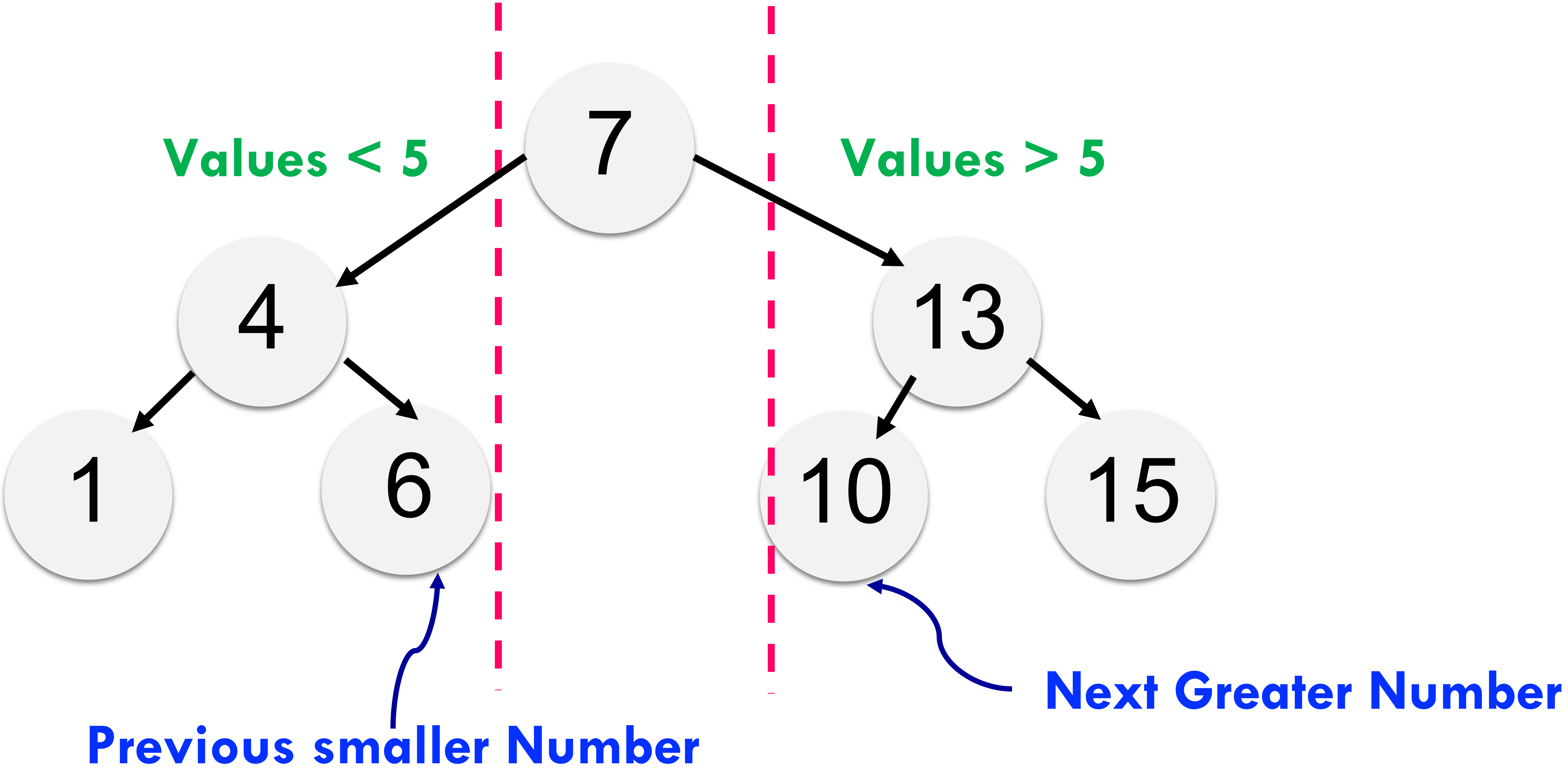
The Power of BSTs: Speed and Efficiency

- Full BSTs are extremely fast
  - At 1 microsecond (or 10^{-6} secs) per operation, we can find something from a tree of size 10^{300000} items in under 1 sec

- Much computation is dedicated towards finding things in response to queries
  - It's a good thing that we can do such queries almost for free

SEARCH in a Binary Search Tree (BST)

How can we find successor(next greater element) of a node?



SEARCH in a Binary Search Tree

EXAMPLE: Search for 88

EXAMPLE: Search for 59

EXAMPLE: Search for 22

EXAMPLE: Search for 37

EXAMPLE: Search for 5

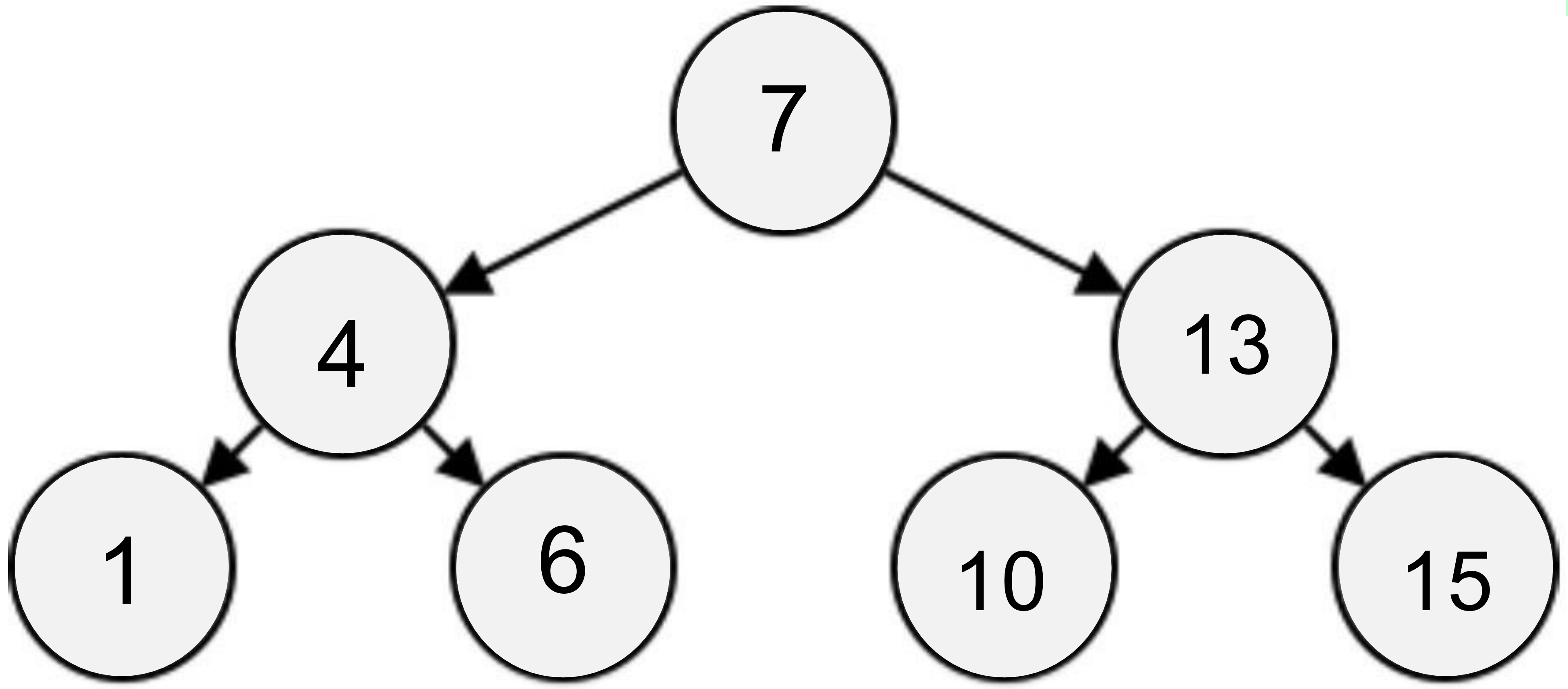
How long does this take?

O(length of longest path) = O(height)

INSERT in a Binary Search Tree (BST)

- INSERT(key, node)
  - X = SEARCH(key)
  - Insert a new node with desired key at x

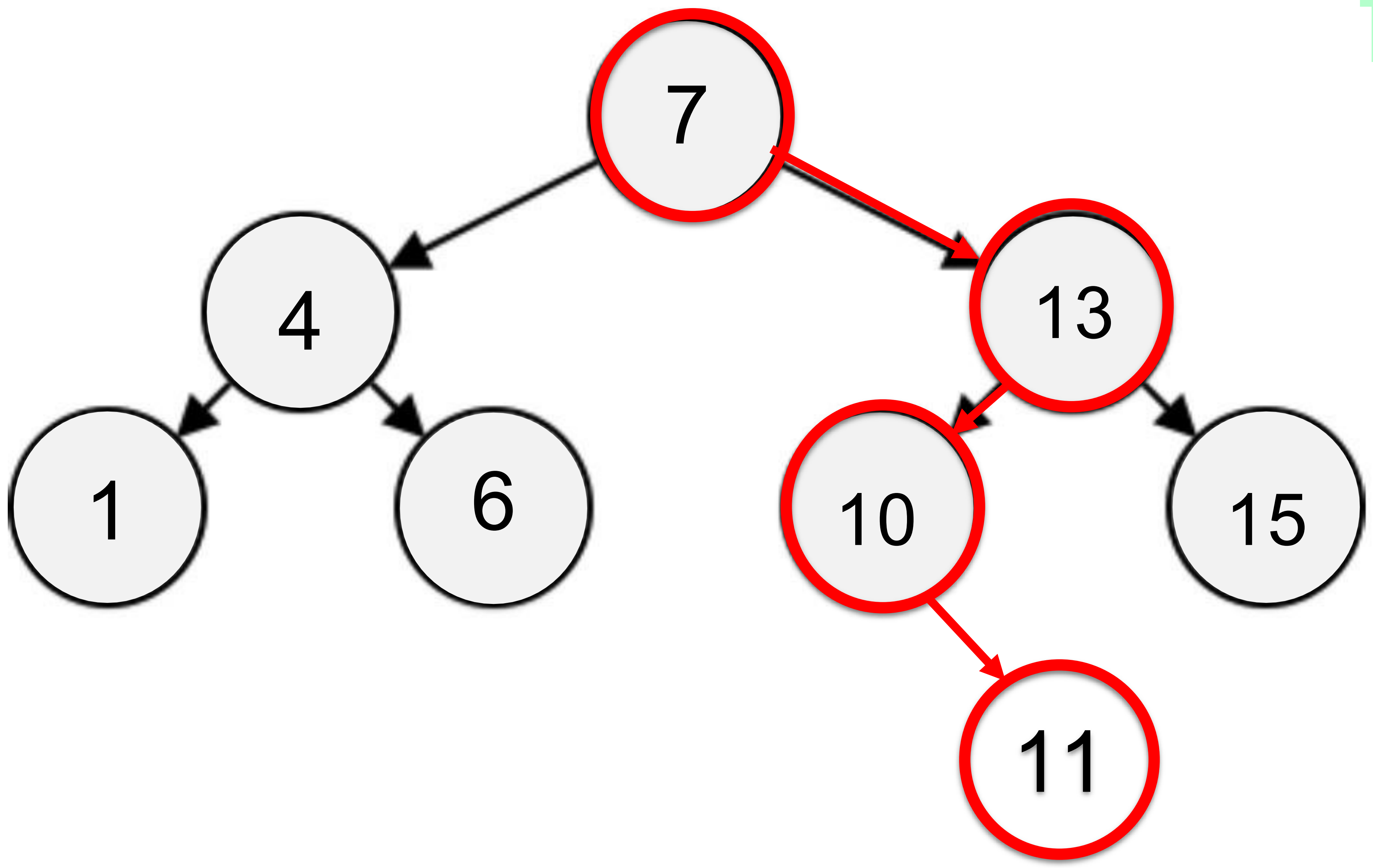
Example: Insert 11



INSERT in a Binary Search Tree (BST)

- INSERT(key, node)
  - X = SEARCH(key)
  - Insert a new node with desired key at x

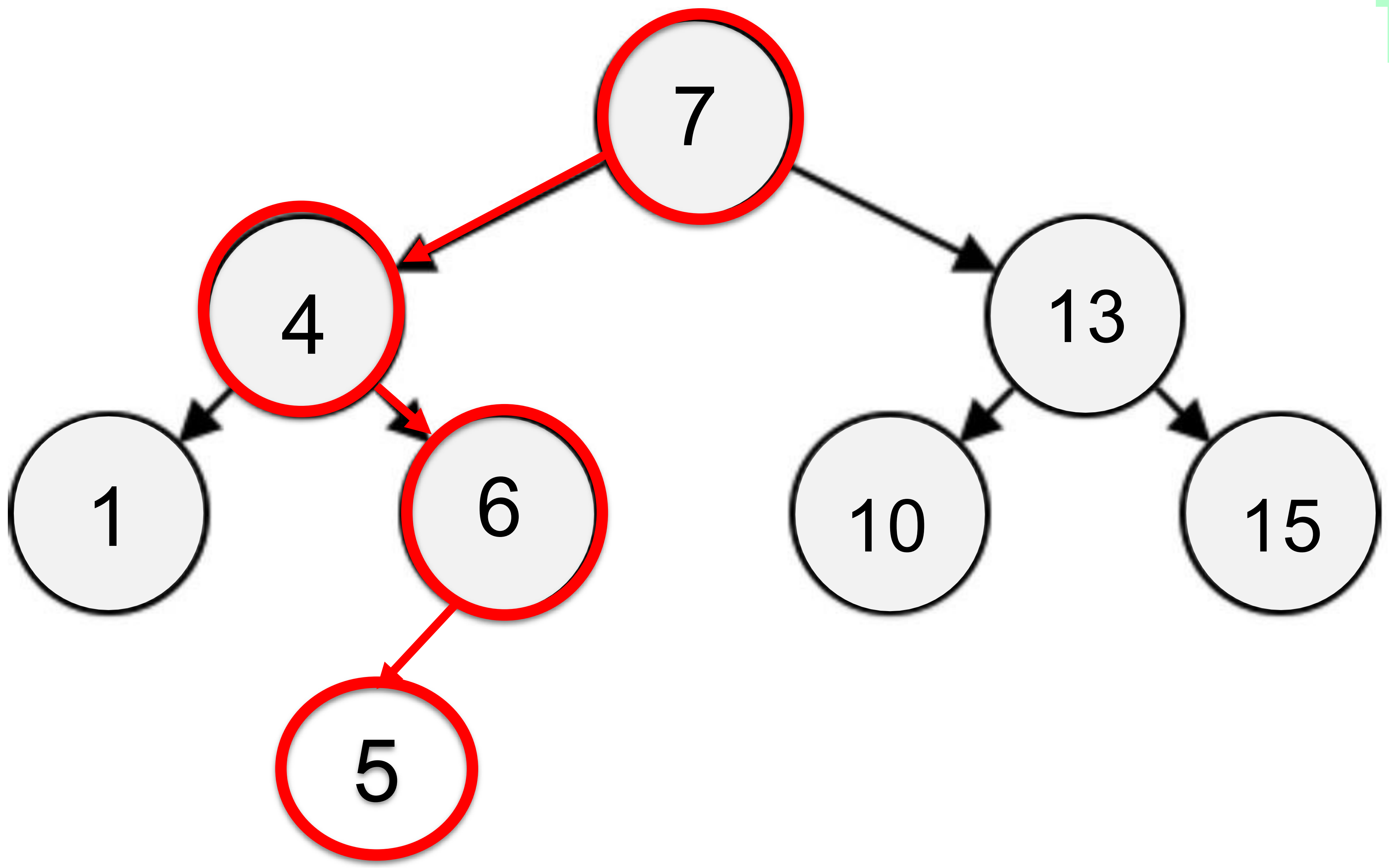
Example: Insert 11



INSERT in a Binary Search Tree (BST)

- INSERT(key, node)
  - X = SEARCH(key)
  - Insert a new node with desired key at x

Example: Insert 5



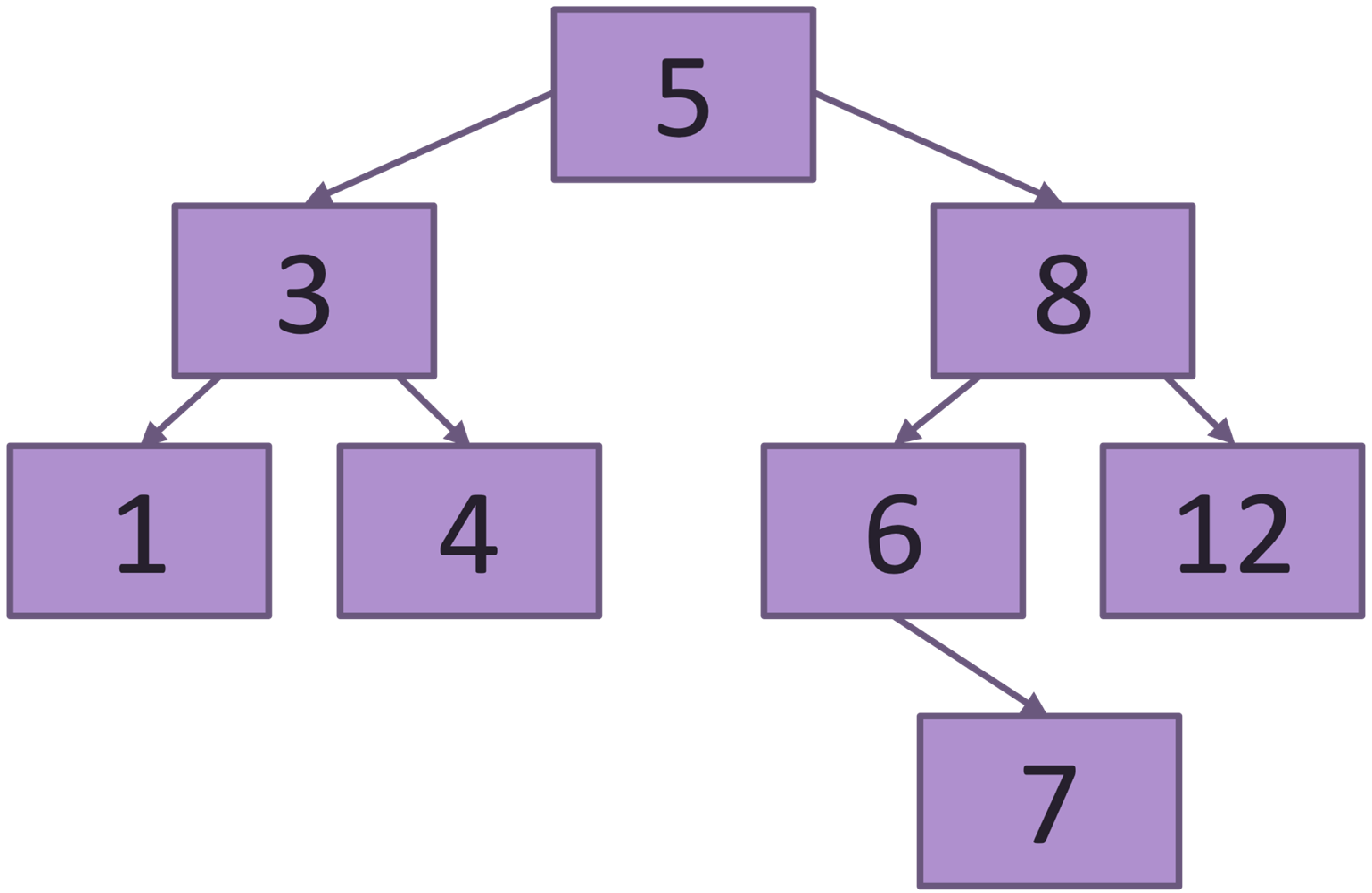
DELETE in a Binary Search Tree (BST)

- DELETE(key, node)

    X = SEARCH(key)

    If x.key == key

      Delete x



DELETE 12

DELETE 6

DELETE 5

Devise an algorithm to delete nodes

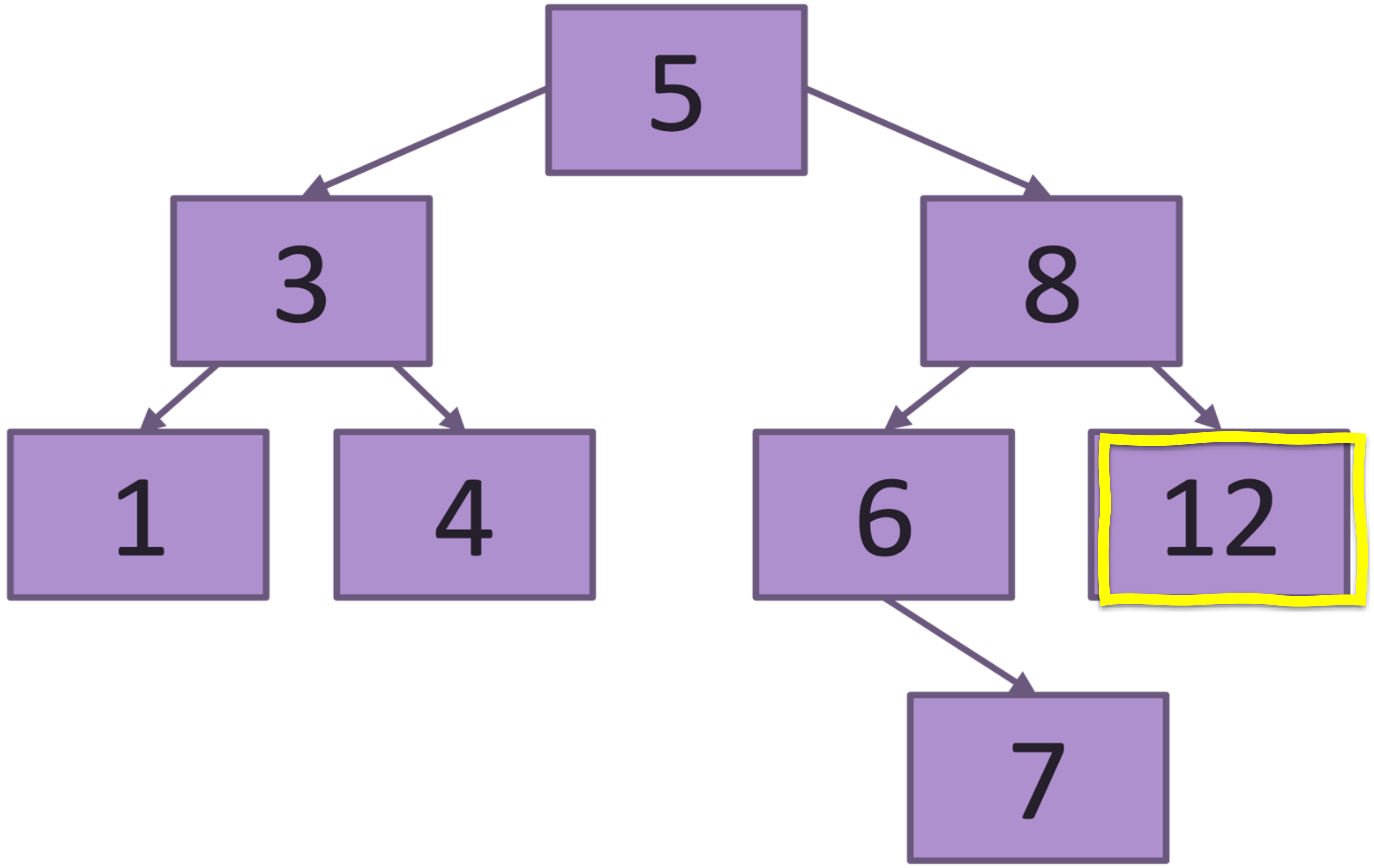
DELETE in a Binary Search Tree (BST)

- 3 Cases
  - Deletion key has no children
  - Deletion key has one child
  - Deletion key has two children

DELETE 12

DELETE 6

DELETE 5



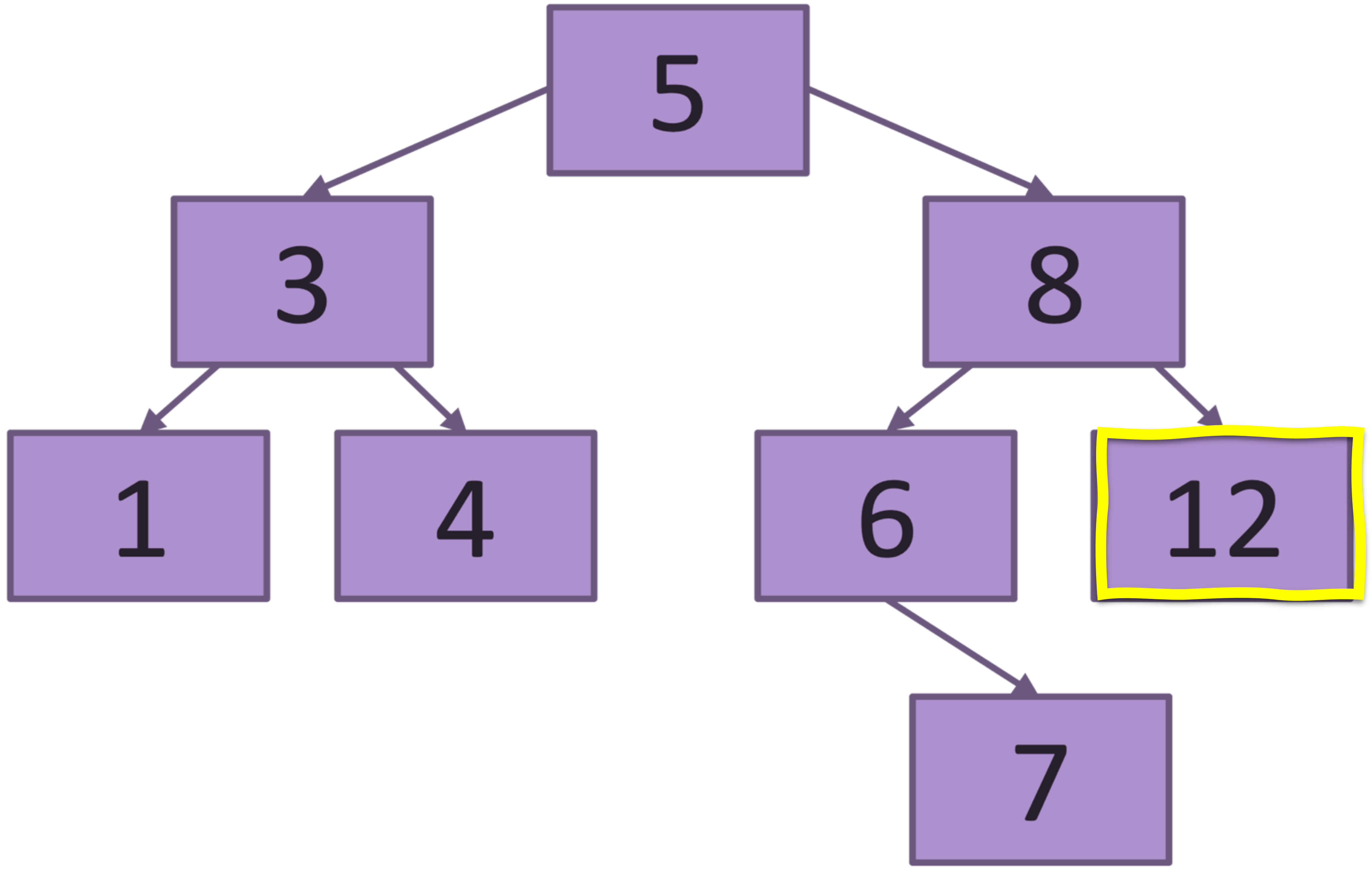
DELETE in a Binary Search Tree (BST)

- 3 Cases
  - Deletion key has no children
  - Deletion key has one child
  - Deletion key has two children

DELETE 12

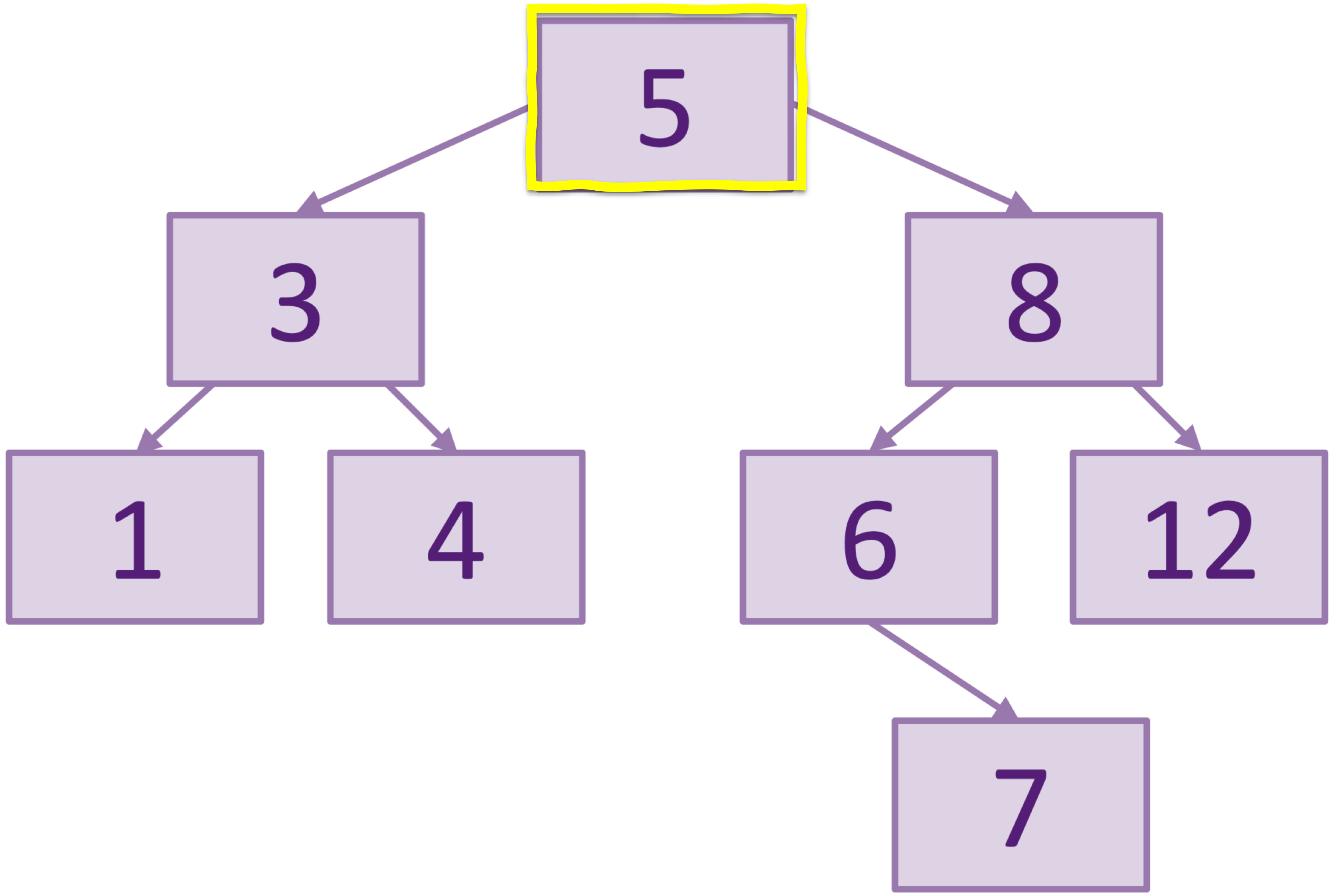
DELETE 6

DELETE 5



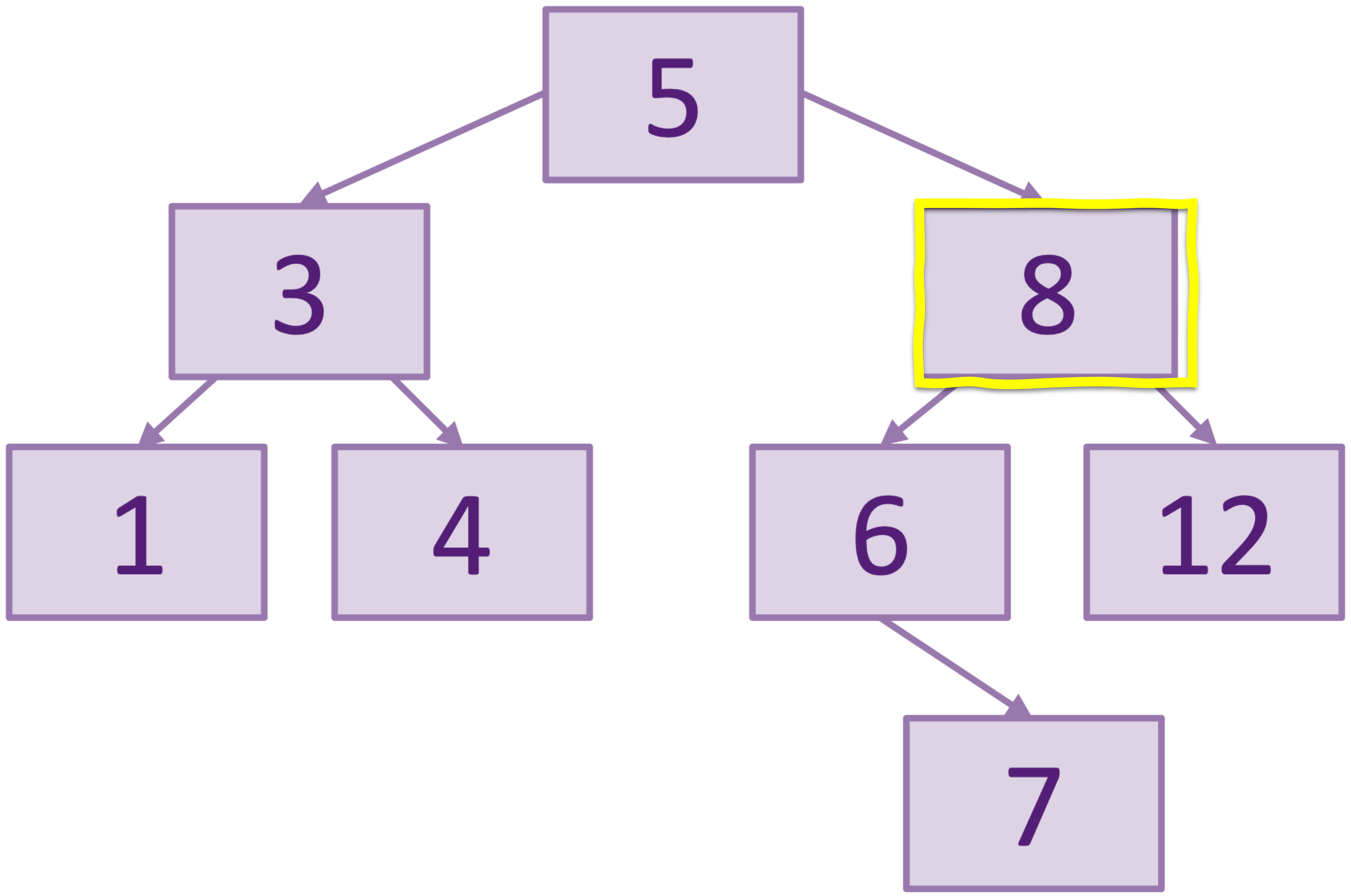
Case 1: Deletion key with no Children

- Example: Delete 12
  - Search and delete the node
  - Set the child pointer to NULL



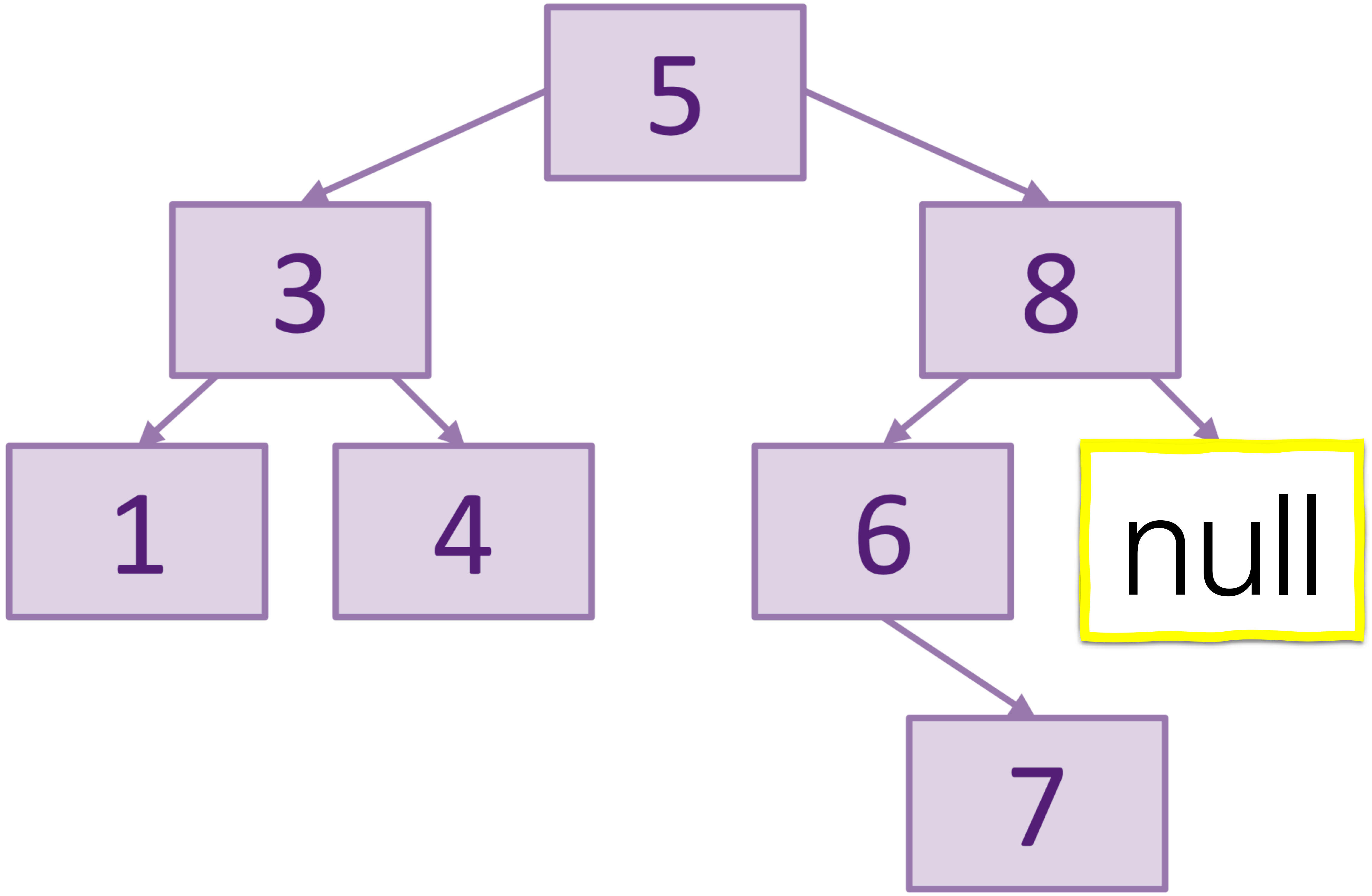
Case 1: Deletion key with no Children

- Example: Delete 12
  - Search and delete the node
  - Set the child pointer to NULL



Case 1: Deletion key with no Children

- Example: Delete 12
  - Search and delete the node
  - Set the child pointer to NULL

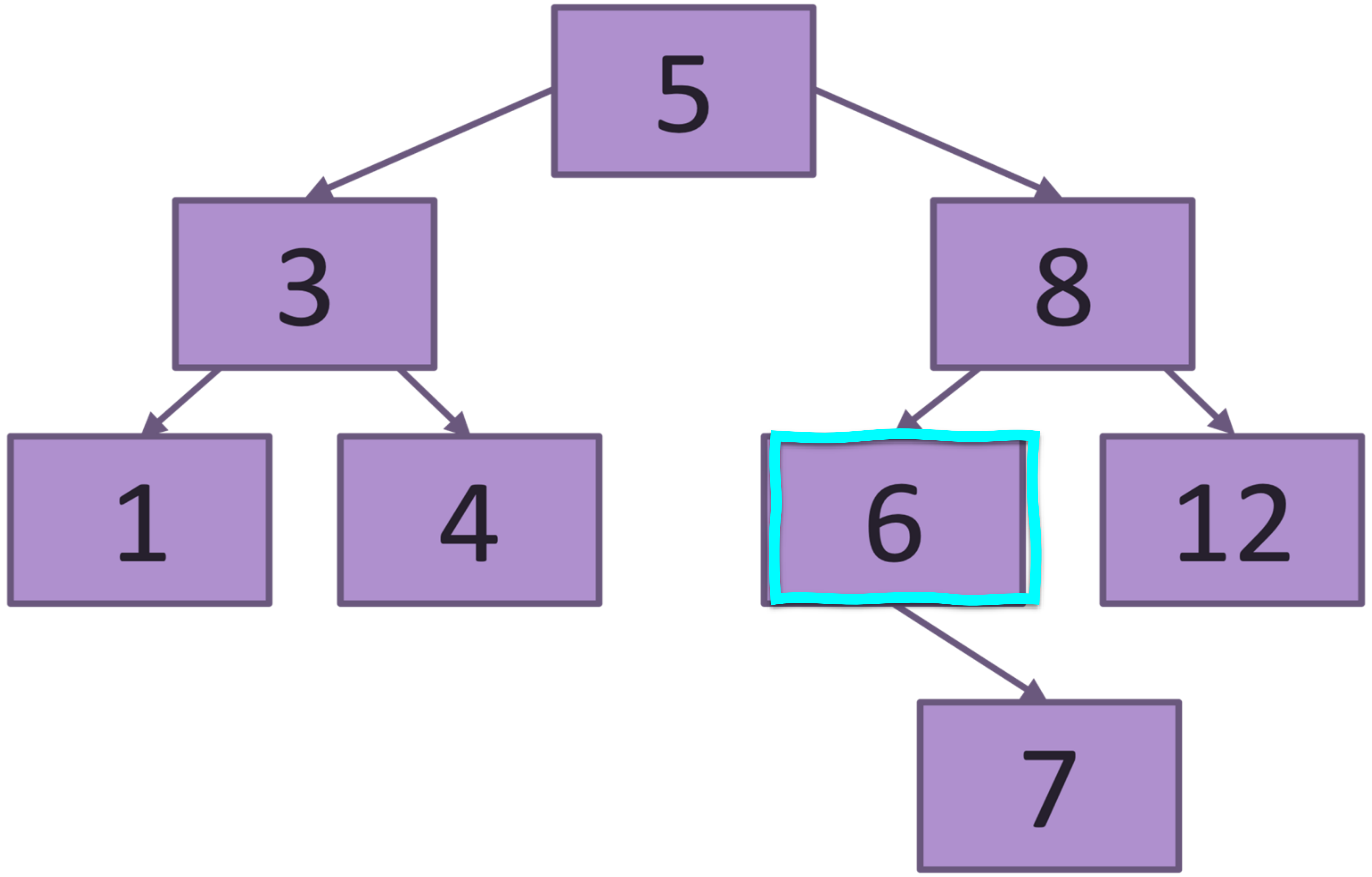


DELETE in a Binary Search Tree (BST)

- 3 Cases
  - Deletion key has no children
  - Deletion key has one child
  - Deletion key has two children

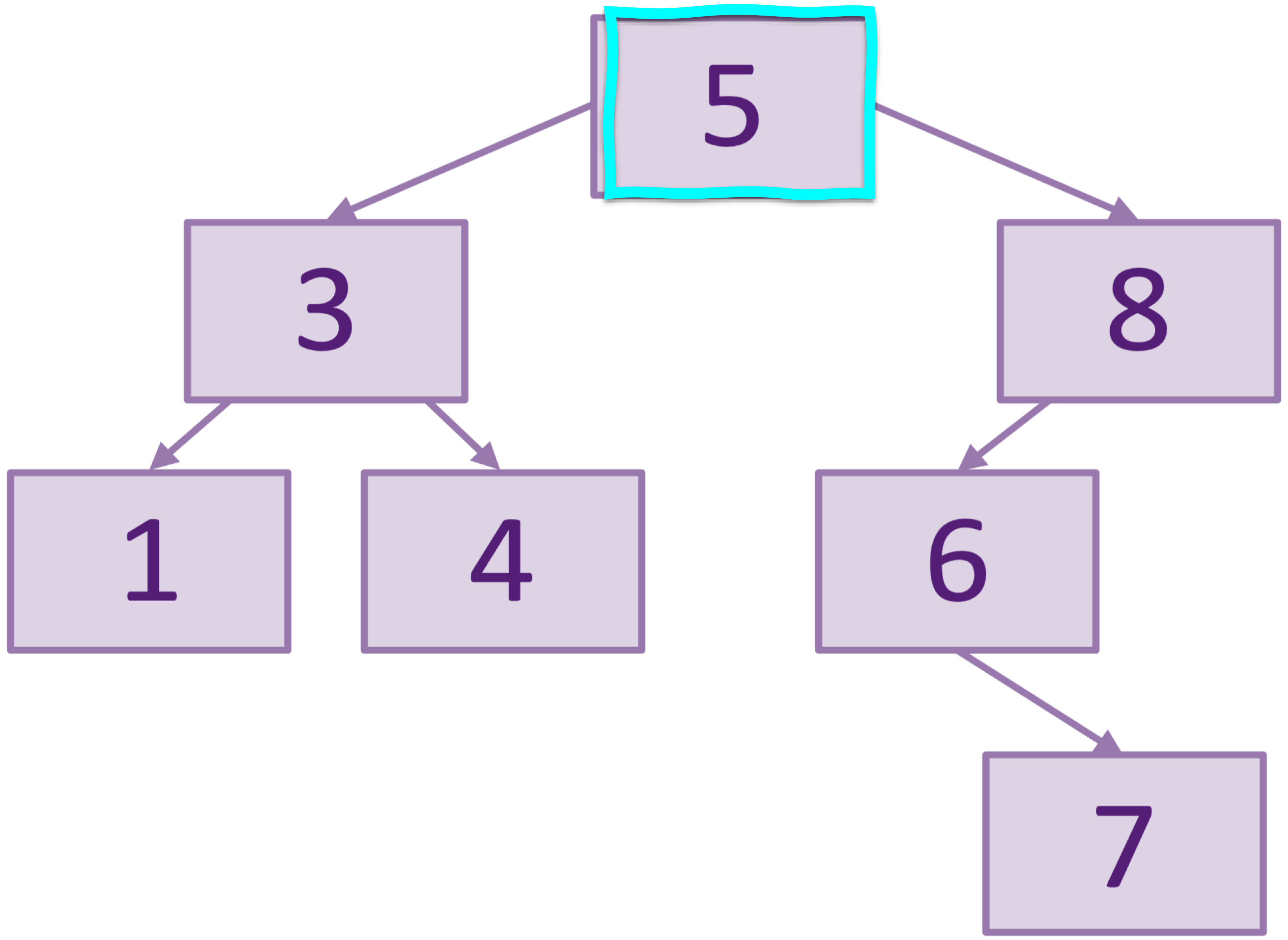
DELETE 6

DELETE 5



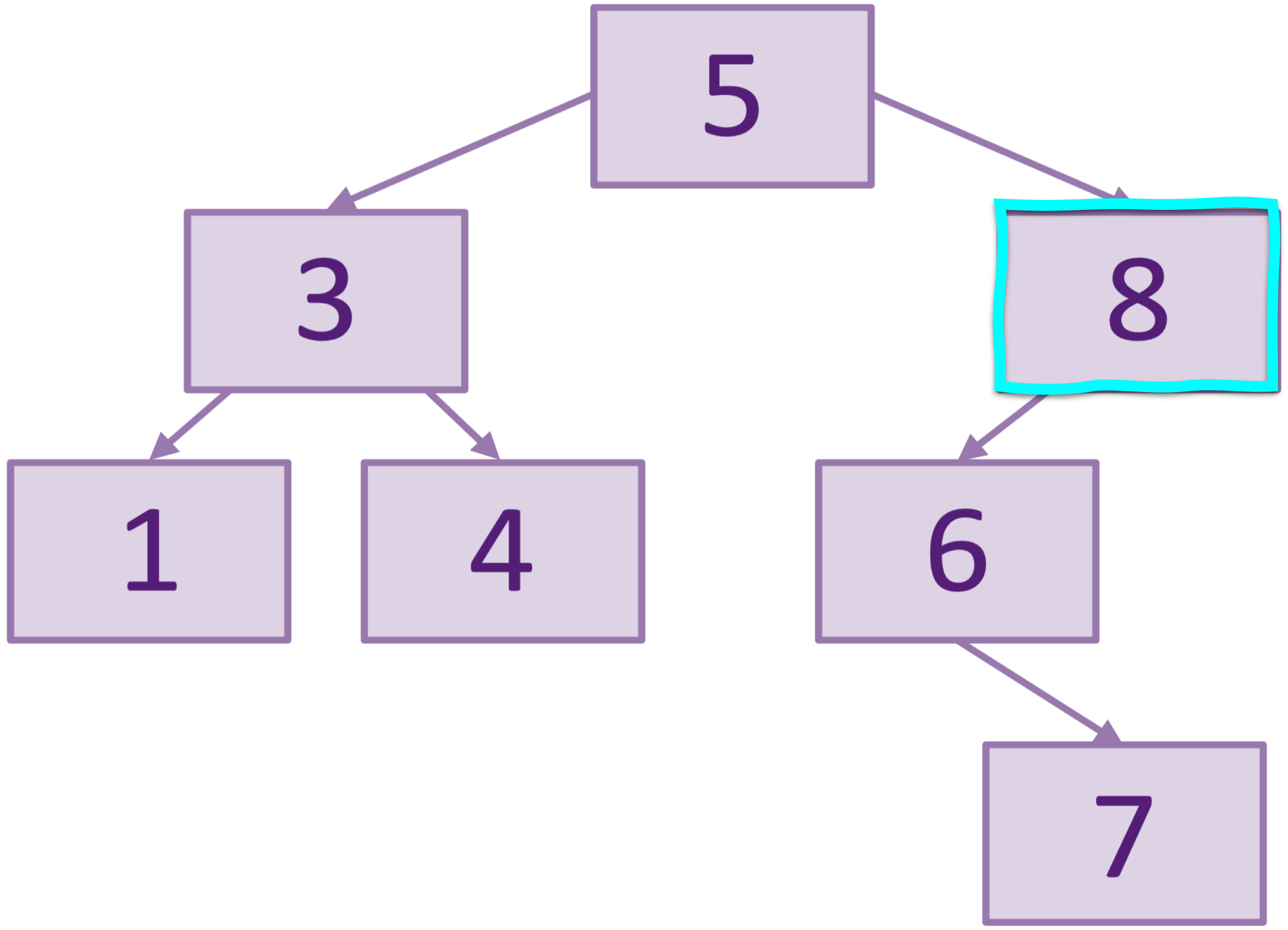
Case 2: Deletion key has one child

- Example: Delete 6
  - Goal: Maintain BST property
  - Observation: 6’s child is definitely less than 6’s parent (8)
  - Safe to move 6’s child into 6’s spot



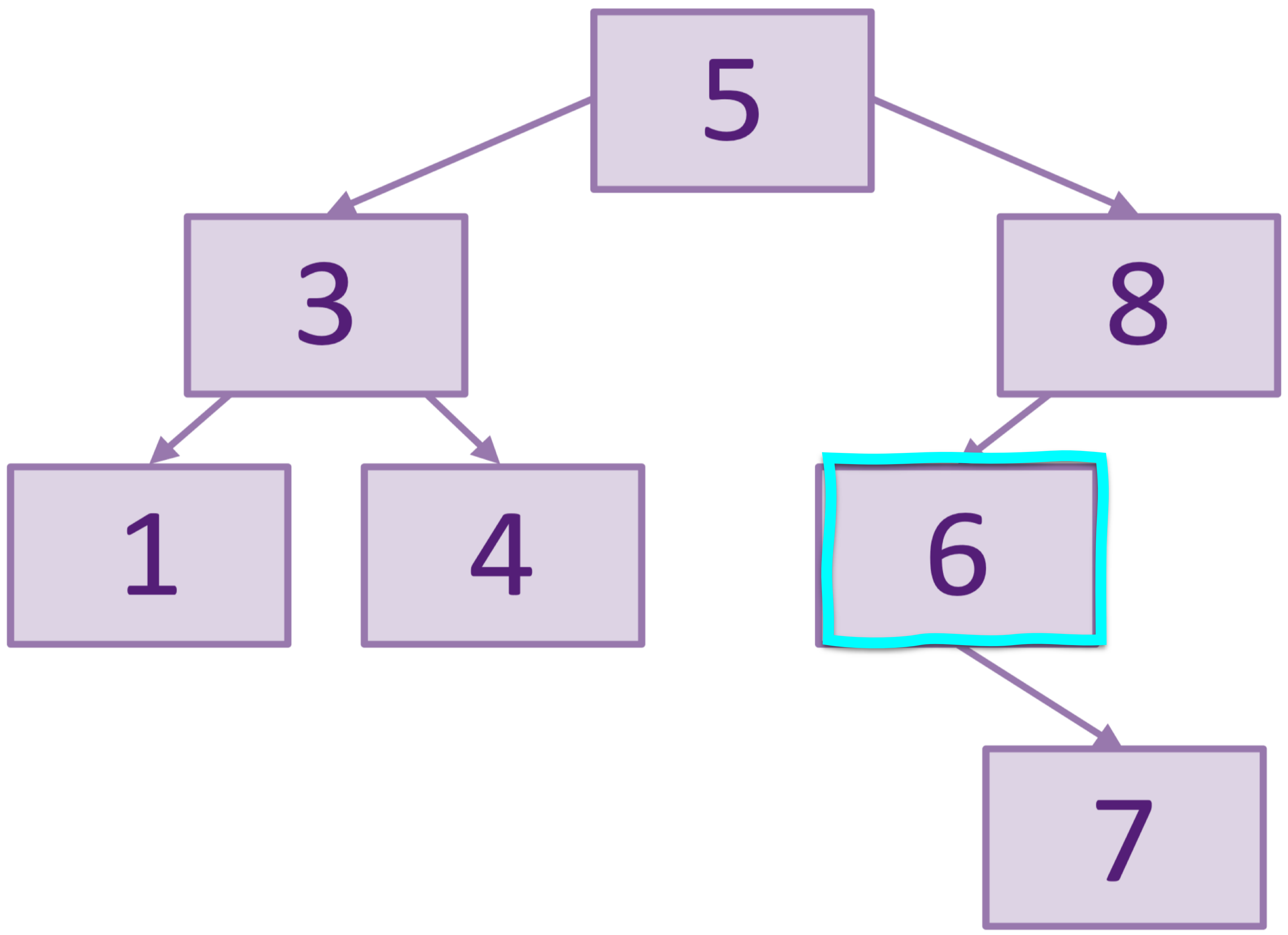
Case 2: Deletion key has one child

- Example: Delete 6
  - Goal: Maintain BST property
  - Observation: 6's child is definitely less than 6's parent (8)
  - Safe to move 6's child into 6's spot



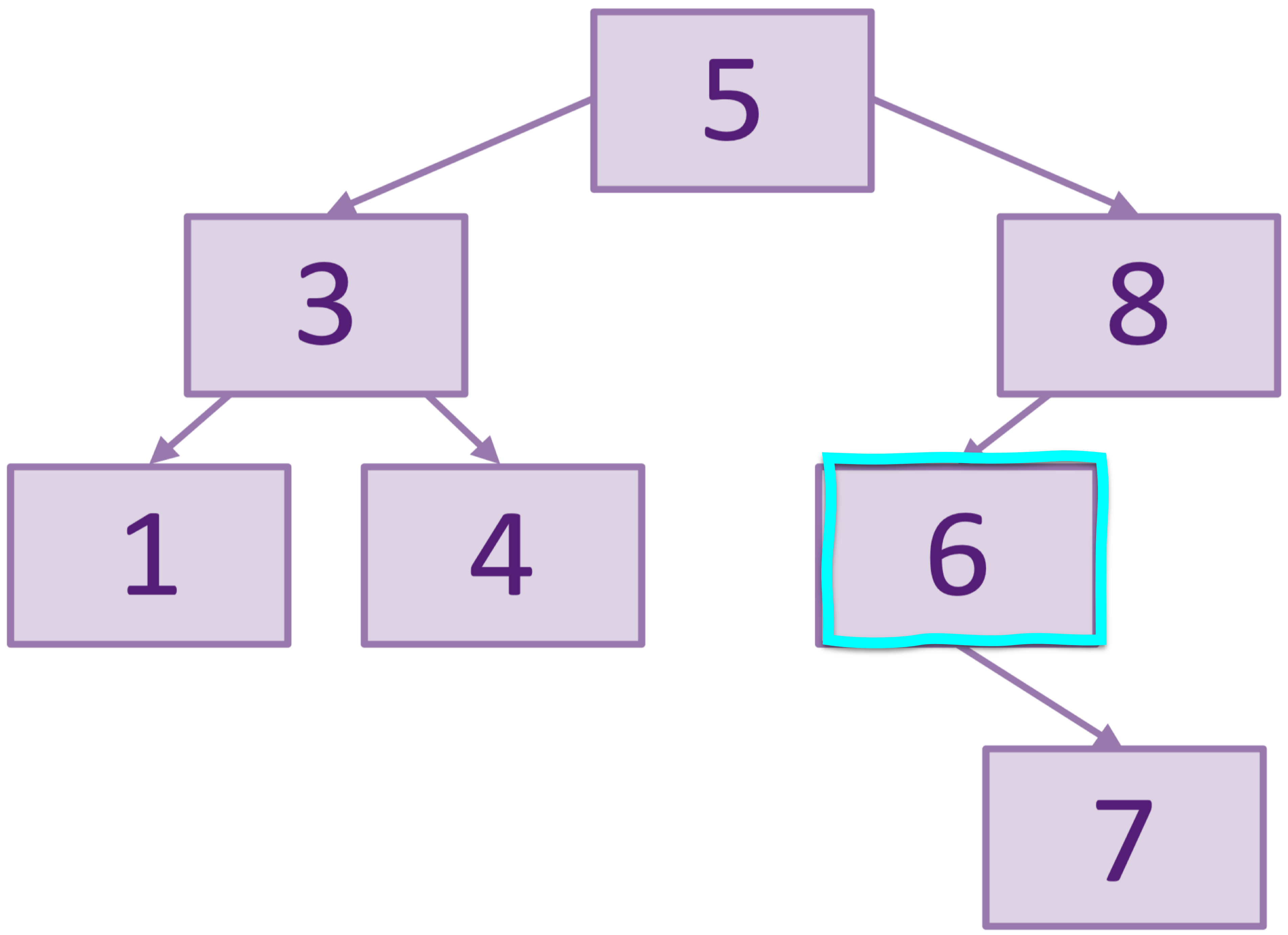
Case 2: Deletion key has one child

- Example: Delete 6
  - Goal: Maintain BST property
  - Observation: 6’s child is definitely less than 6’s parent (8)
  - Safe to move 6’s child into 6’s spot



Case 2: Deletion key has one child

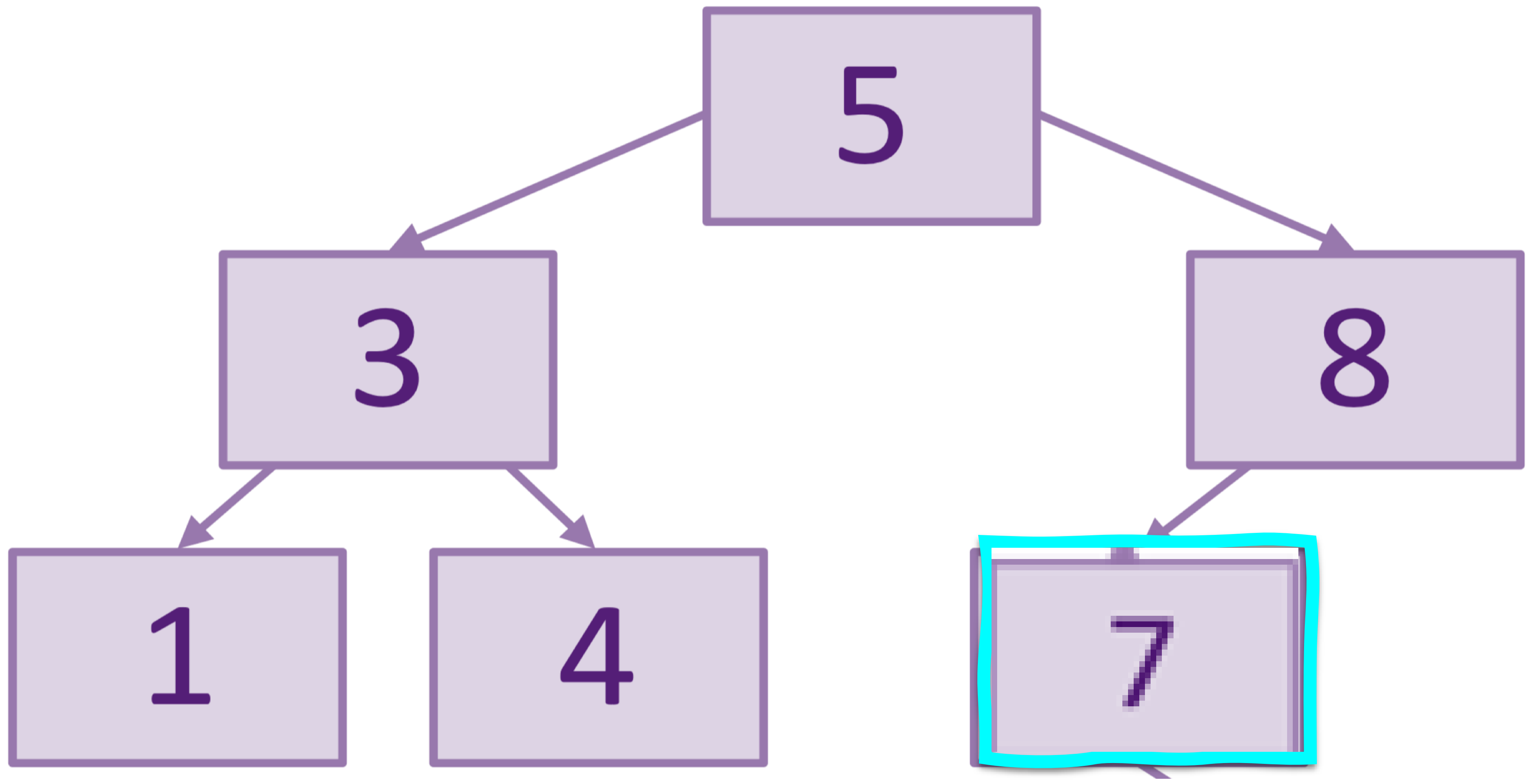
- Example: Delete 6
  - Goal: Maintain BST property
  - Observation: 6's child is definitely less than 6's parent (8)
  - Safe to move 6's child into 6's spot



Make 8's child left pointer to point to 6's child

Case 2: Deletion key has one child

- Example: Delete 6
  - Goal: Maintain BST property
  - Observation: 6’s child is definitely less than 6’s parent (8)
  - Safe to move 6’s child into 6’s spot

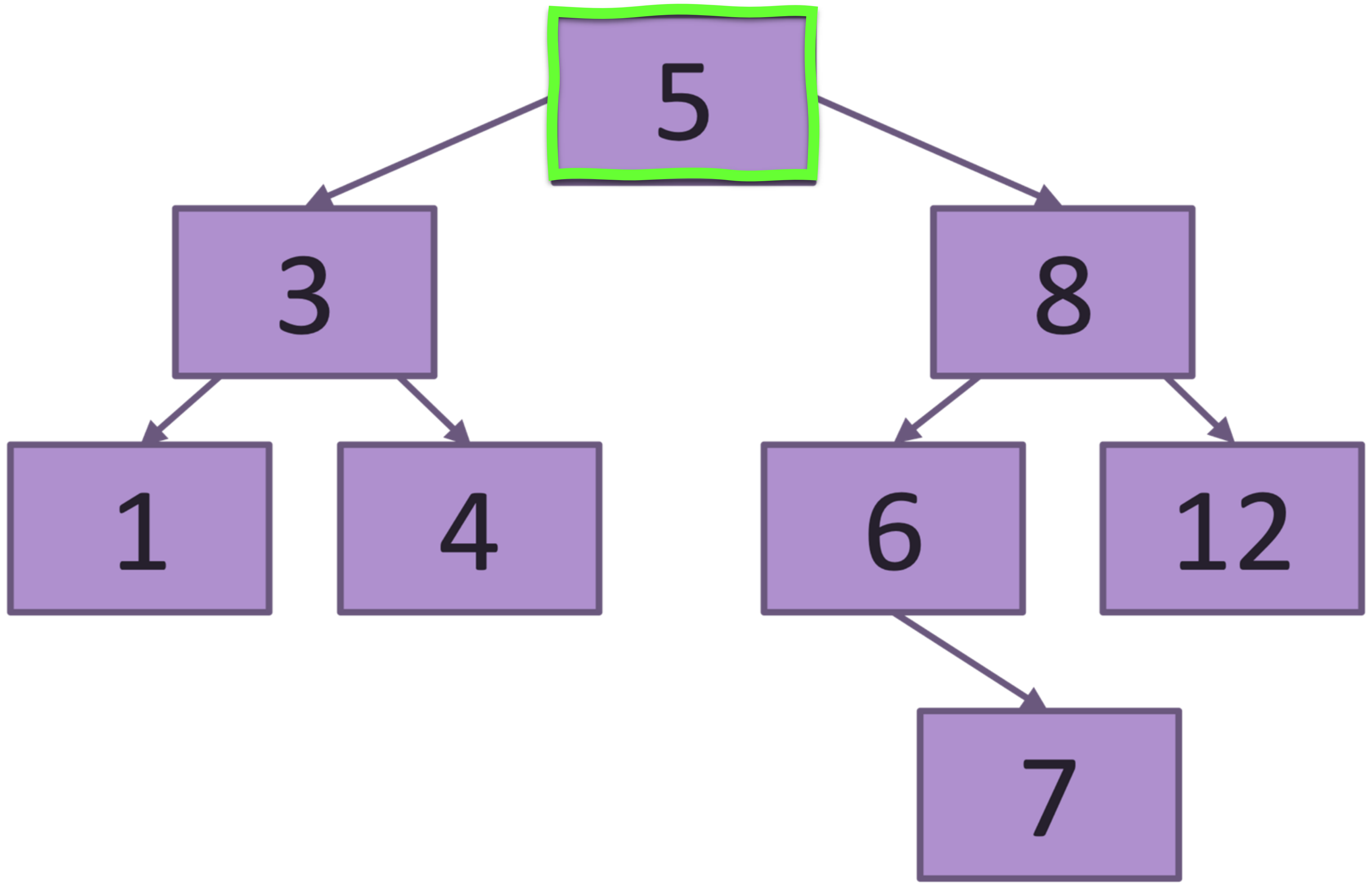


Make 8’s child left pointer to point to 6’s child

DELETE in a Binary Search Tree (BST)

- 3 Cases
  - Deletion key has no children
  - Deletion key has one child
  - Deletion key has two children

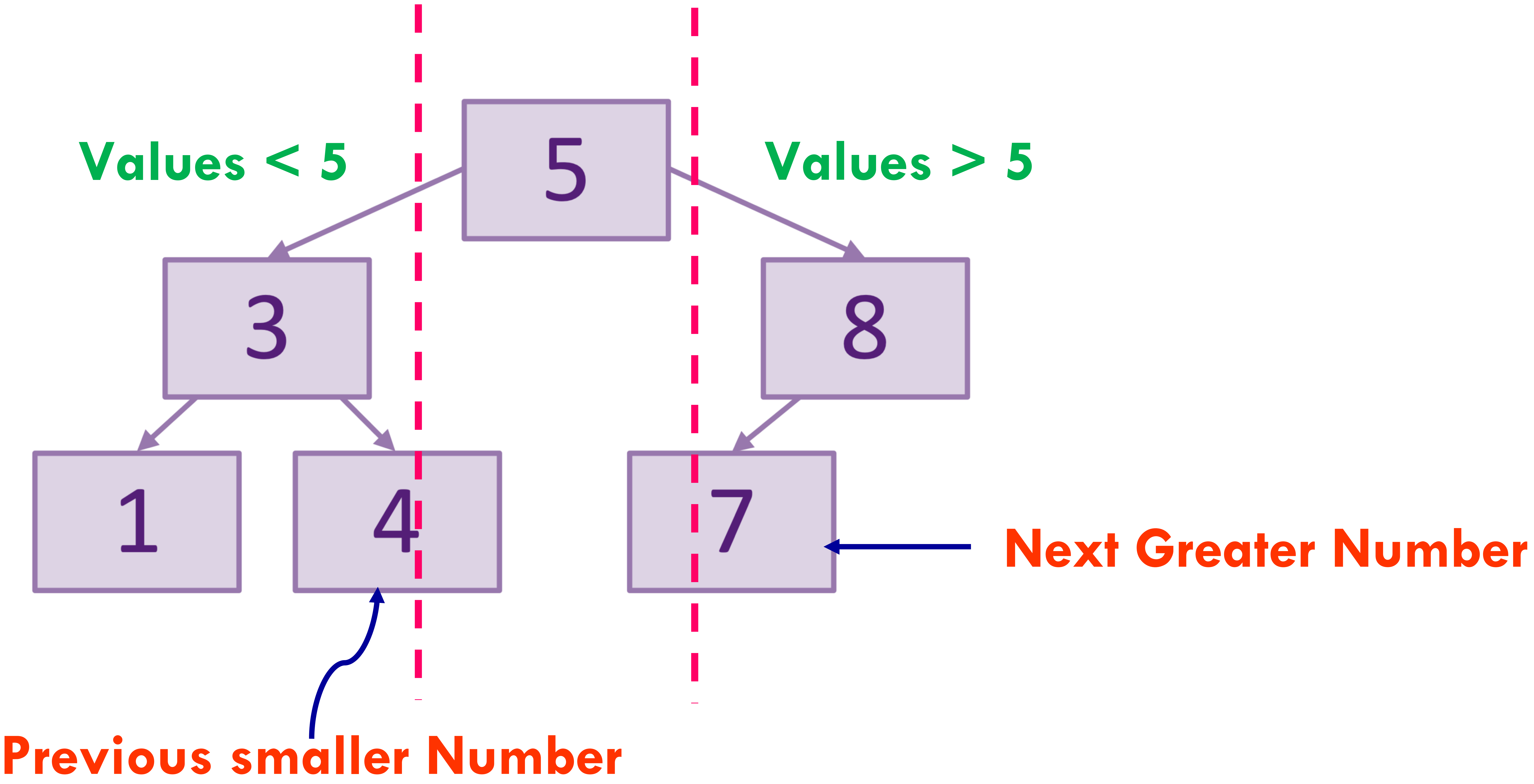
DELETE 5



Harder challenge

- Delete 5

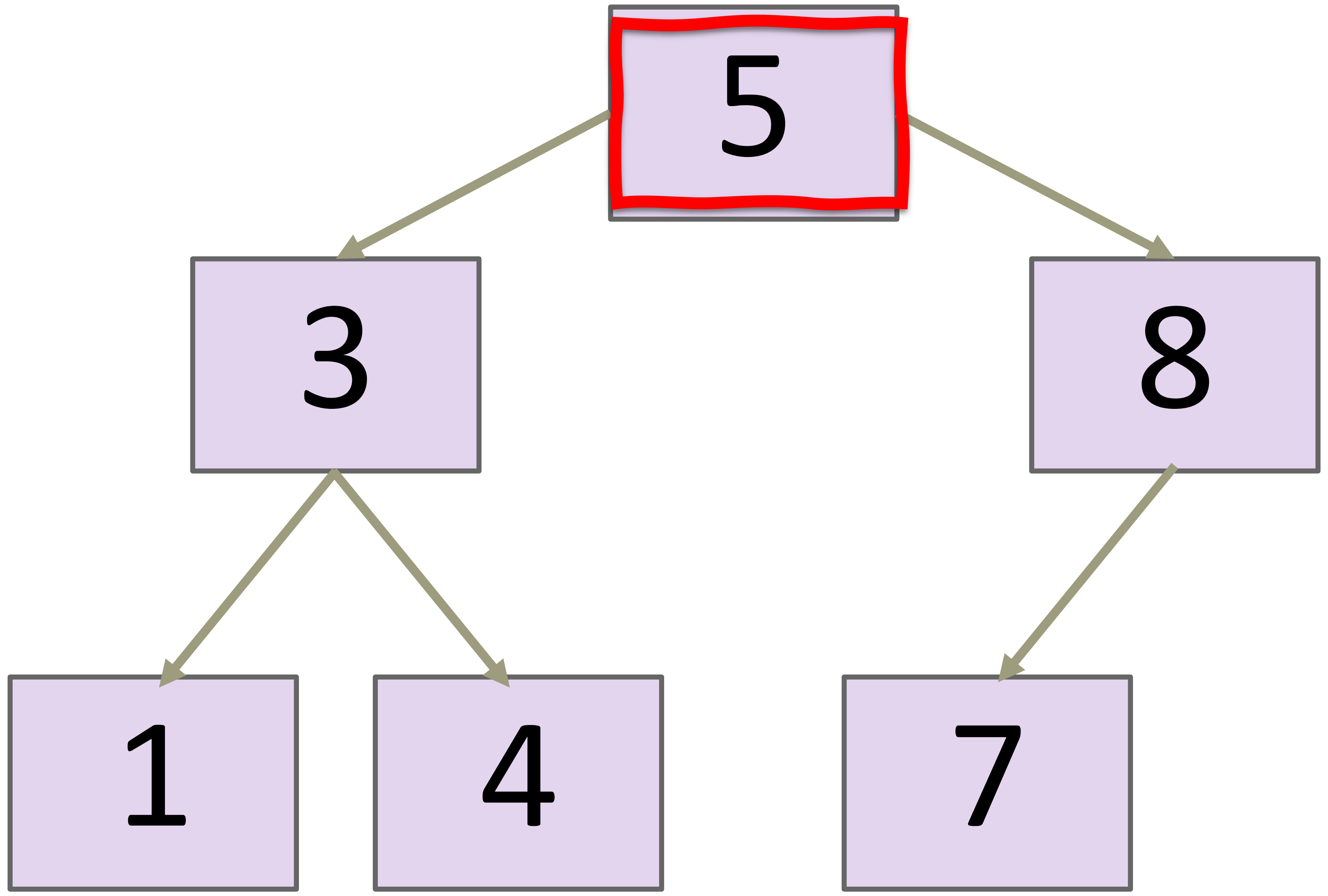
What would allow us to maintain the BST Property?



Case 3: Deletion key has two children

DELETE 5

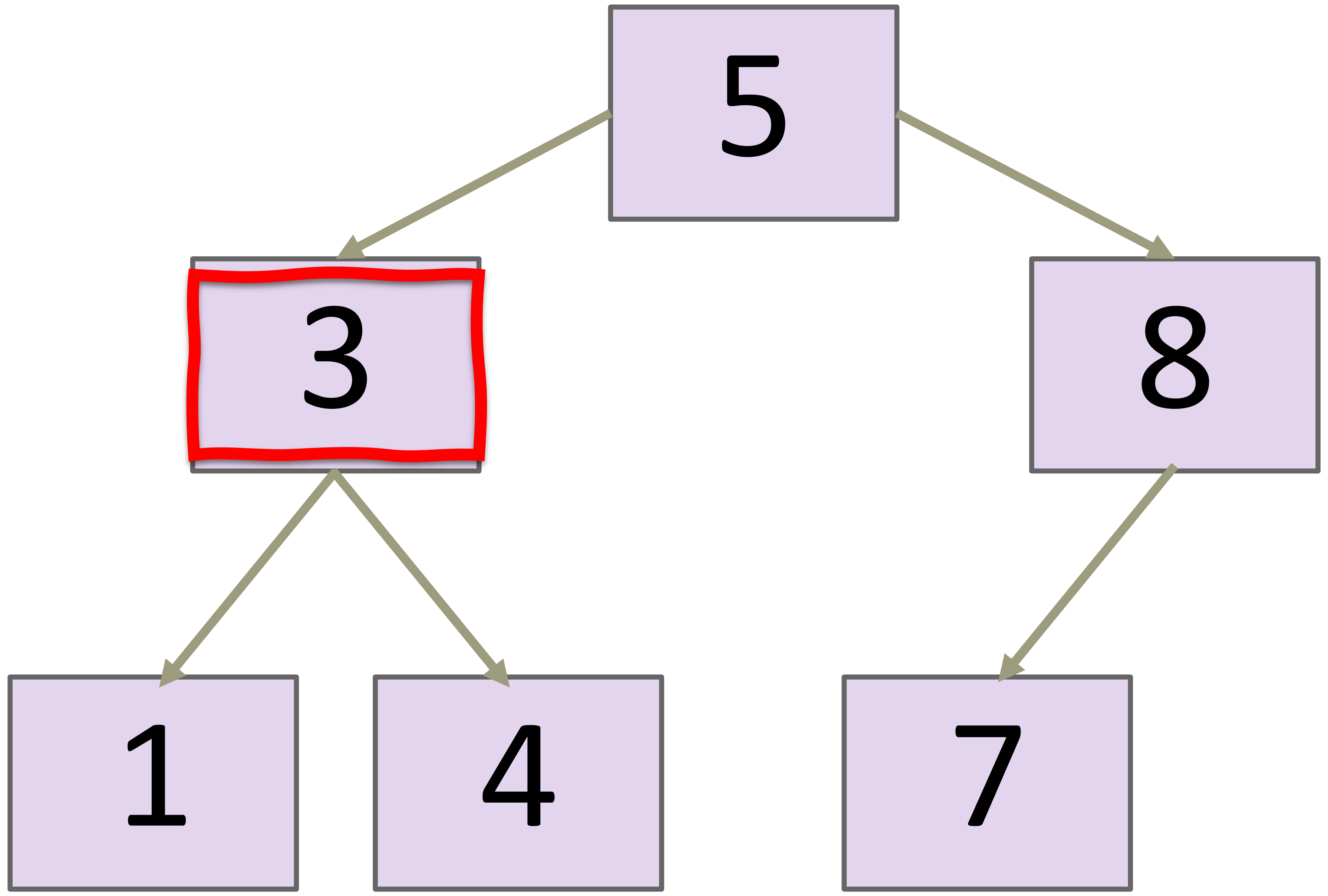
- Two solutions
  - Either promote 4 or 7 to be in the root
  - Call delete (on 4 or 7) in the appropriate sub-tree (Case 1 or Case 2 will be applicable)



Case 3: Deletion key has two children

- Two solutions
  - Either promote 4 or 7 to be in the root
  - Call delete (on 4 or 7) in the appropriate sub-tree (Case 1 or Case 2 will be applicable)

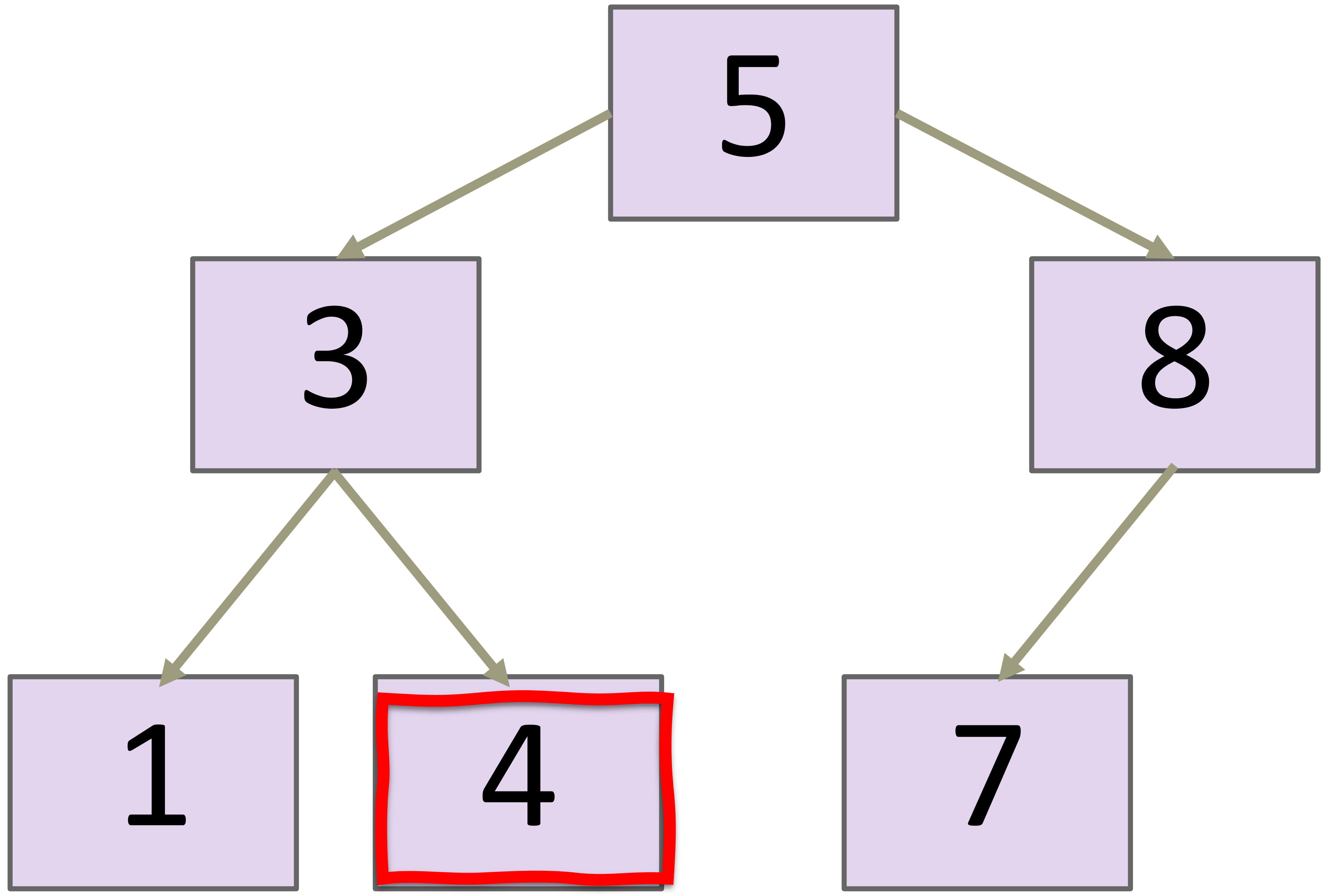
DELETE 5



Case 3: Deletion key has two children

- Two solutions
  - Either promote 4 or 7 to be in the root
  - Call delete (on 4 or 7) in the appropriate sub-tree (Case 1 or Case 2 will be applicable)

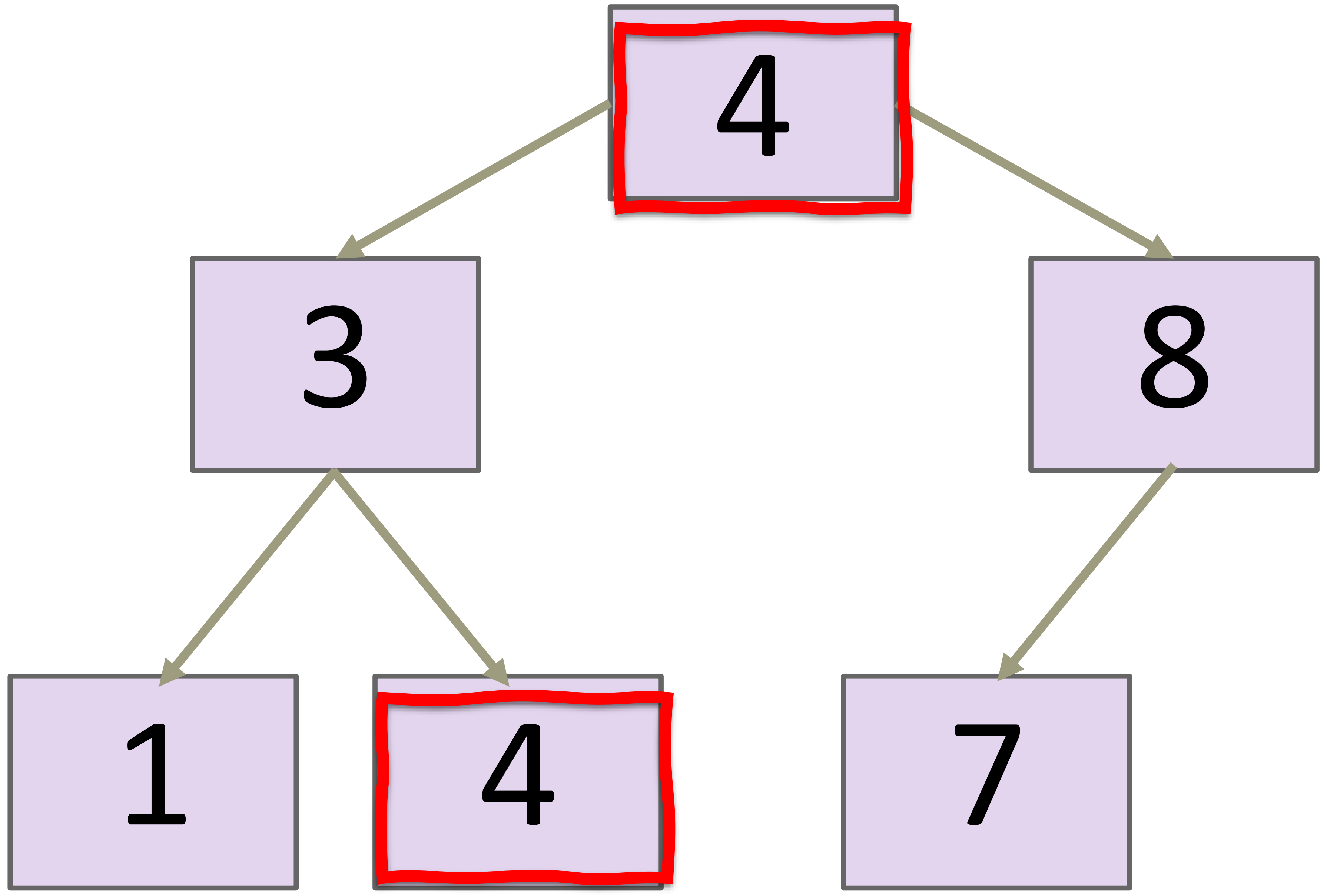
DELETE 5



Case 3: Deletion key has two children

- Two solutions
  - Either promote 4 or 7 to be in the root
  - Call delete (on 4 or 7) in the appropriate sub-tree (Case 1 or Case 2 will be applicable)

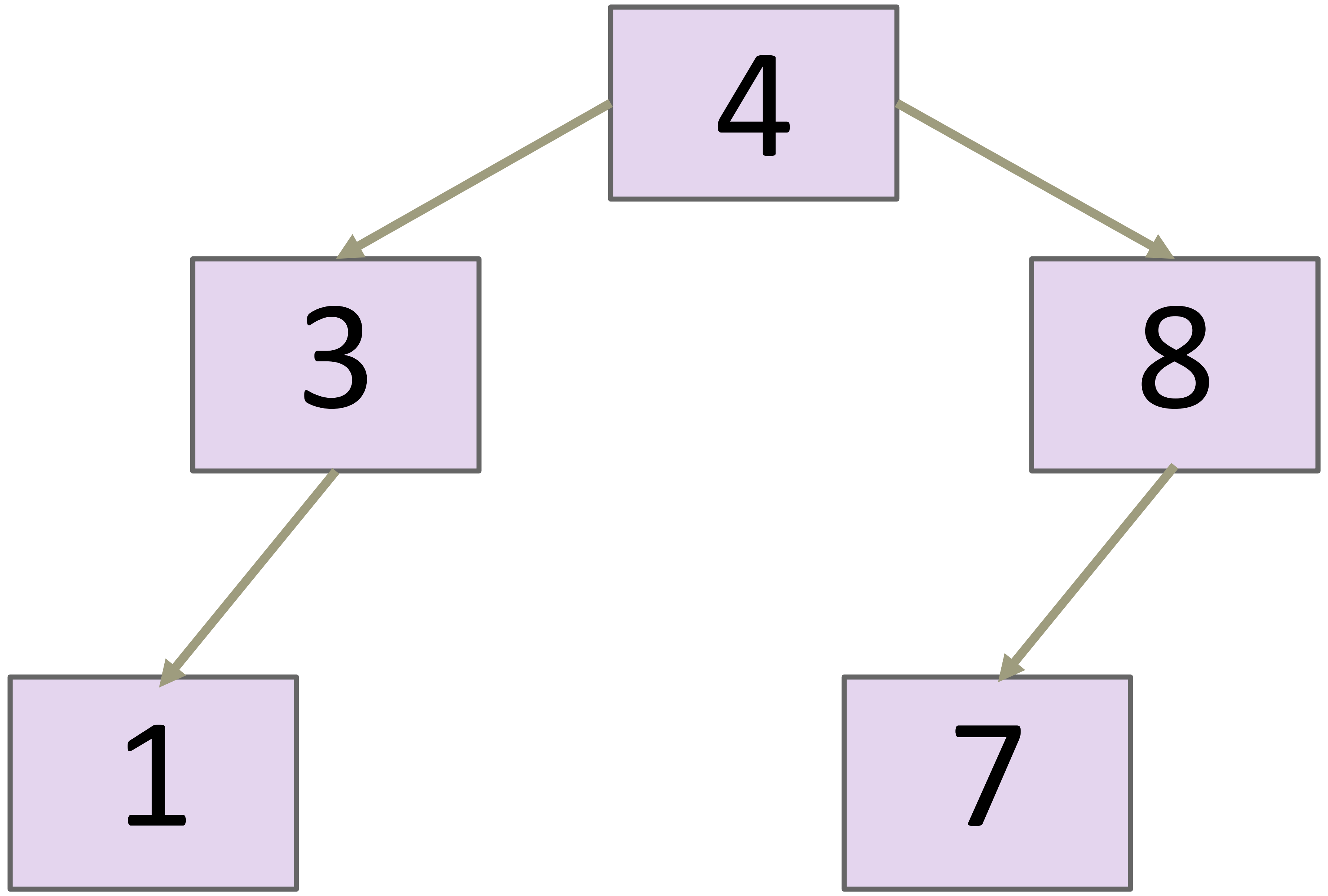
DELETE 5



Case 3: Deletion key has two children

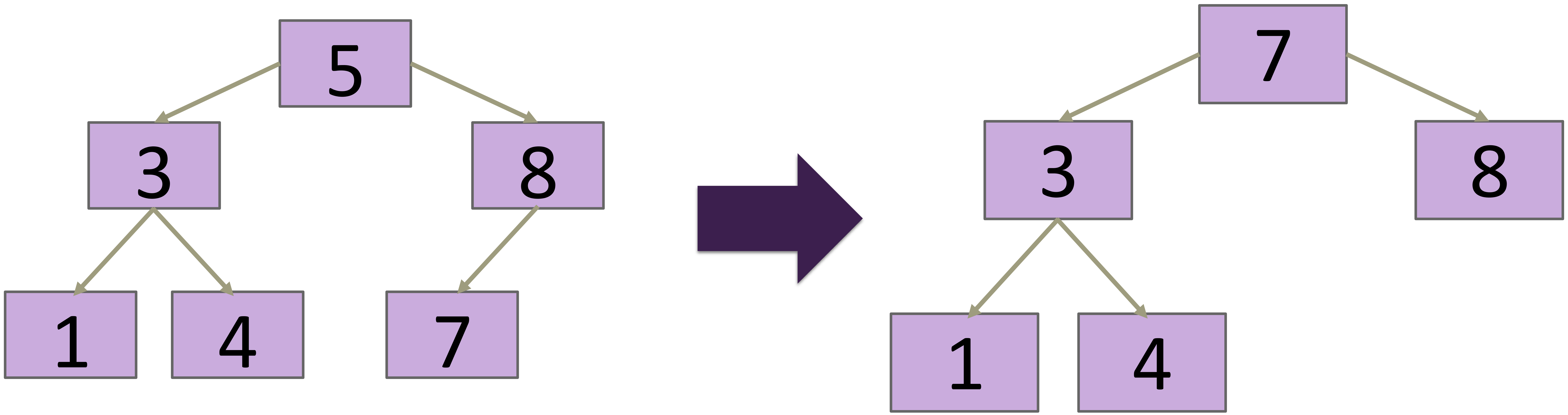
- Two solutions
  - Either promote 4 or 7 to be in the root
  - Call delete (on 4 or 7) in the appropriate sub-tree (Case 1 or Case 2 will be applicable)

DELETE 5



Case 3: Deletion key has two children

- Solution-II: Replace with next greater number
  - Find next greater element and promote it to be in the root.
  - Call delete on next greater element in the appropriate sub-tree



Practice Problem – 1: Next Element

- Given a node N in a BST, find the node with the next largest key

- Eventual goal: Given a node N in a BST, we would like to find adjacent elements

Questions