Tree

- *introduction*
- *binary tree*
- *complete binary tree*
  - *max heap, min heap*
  - *Chapter 7 – heap sorting*
  - *Chapter 9 - priority queues*
- ***binary search tree***

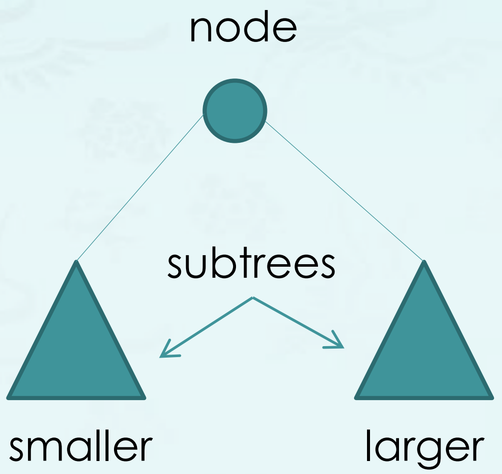
Binary search trees

Definition: A binary search tree is a binary tree in symmetric order.

- A binary tree is either
  - empty
  - a key-value pair and two binary trees [neither of which contain that key]

equal keys ruled out

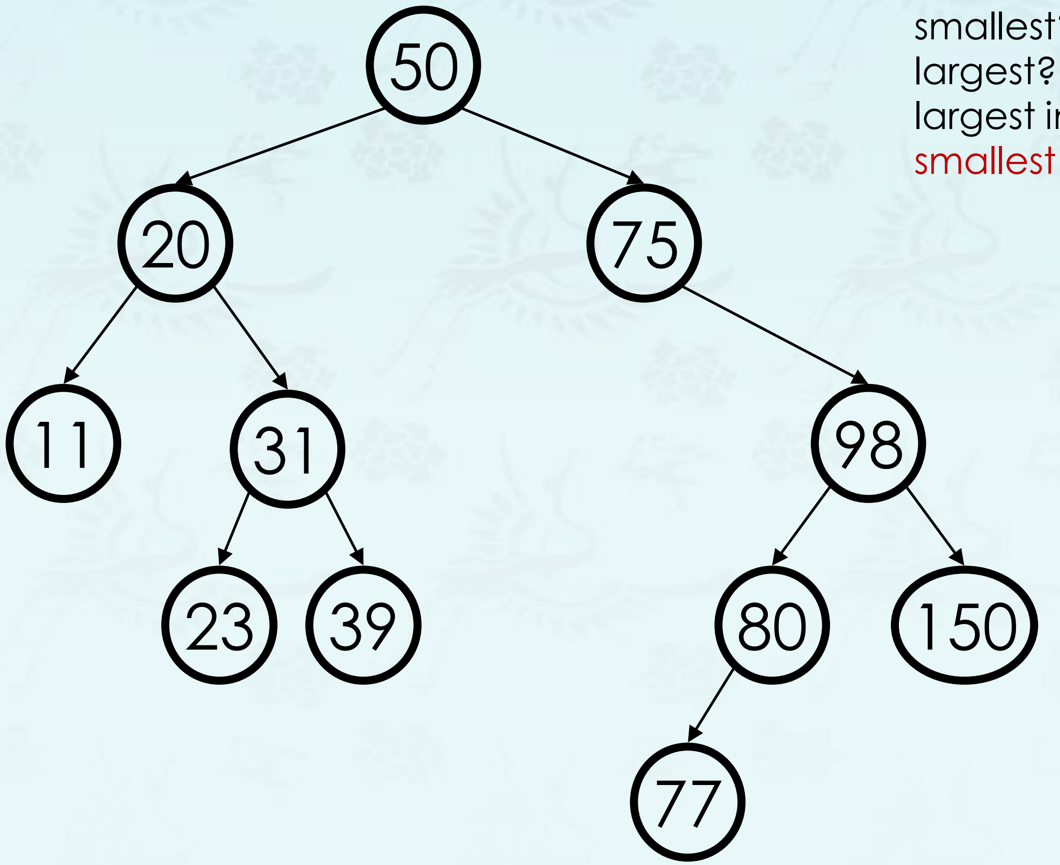
- Symmetric order means that
  - every node has a key
  - every node's key is
    larger than all keys in its left subtree
    smaller than all keys in its right subtree



Operations: grow

- Q: Draw what a binary search tree would look like if the following values were added to an initially empty tree in this order:

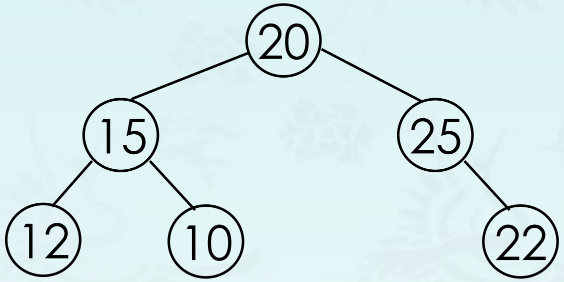
50
20
75
98
80
31
150
39
23
11
77



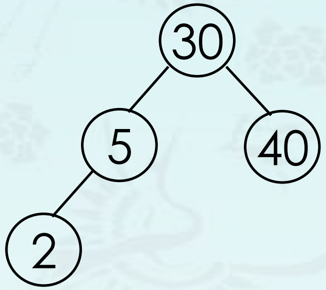
smallest?
largest?
largest in left?
smallest in right?

Binary search trees

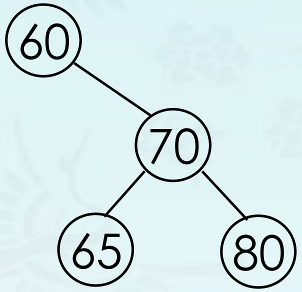
Definition: A binary search tree is a binary tree in symmetric order.
Exercise: Identify non-BST(s) and correct them if not.



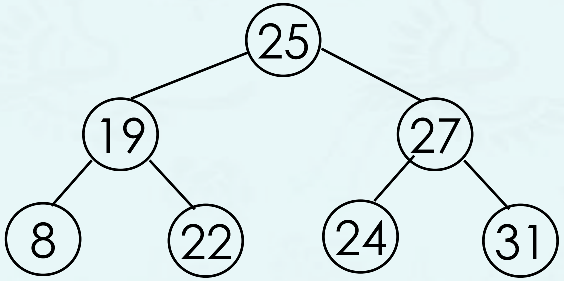
(a)



(b)



(c)

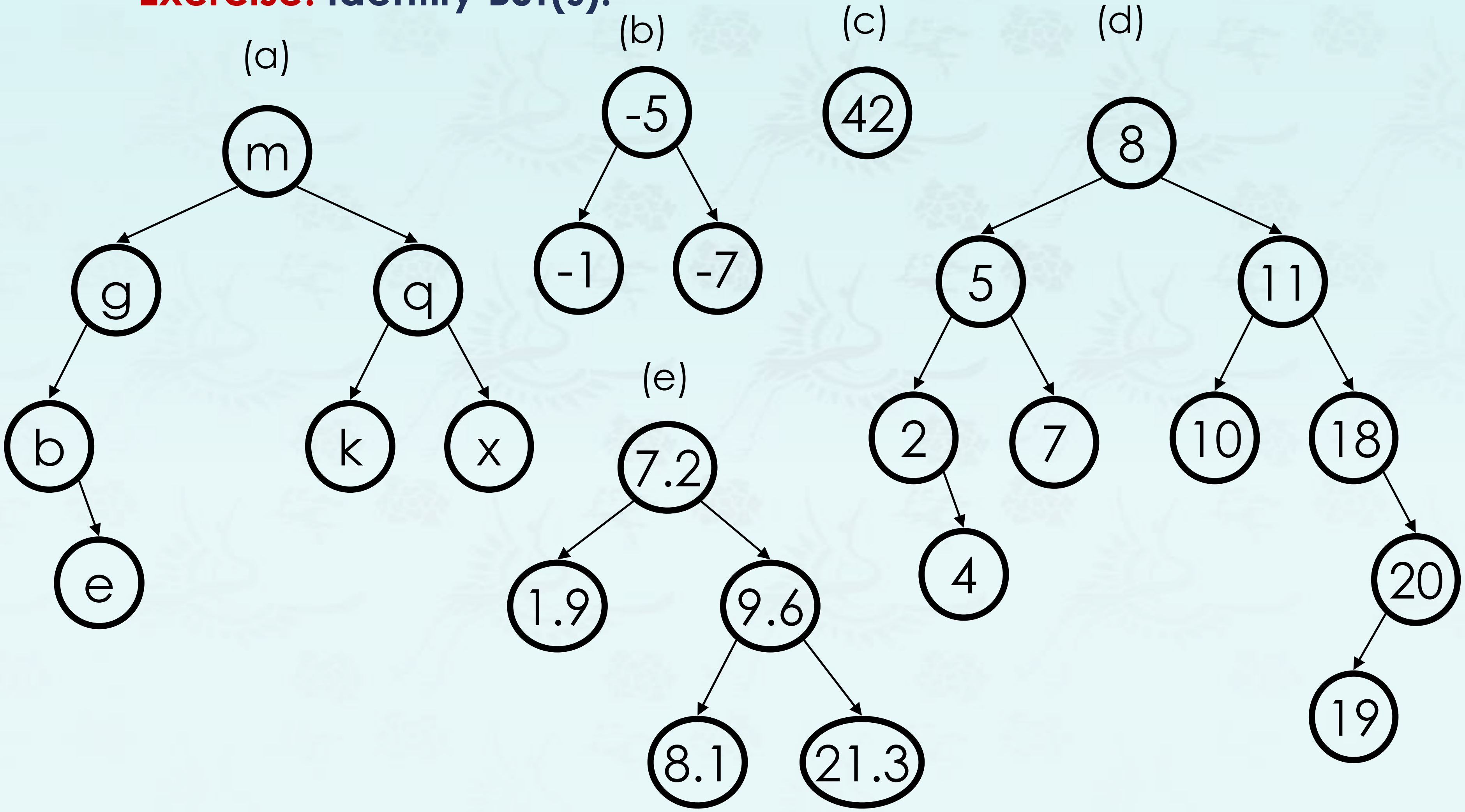


(d)

Binary search trees

Definition: A binary search tree is a binary tree in symmetric order.

Exercise: Identify BST(s).



Binary search trees

Node structure:

<table>
<tr><td colspan="2">key</td></tr>
<tr><td>Left</td><td>Right</td></tr>
</table>

Operations:

- **Query – search, min/max, successor, predecessor**
- **grow – insert**
- **trim – delete**

Binary search trees

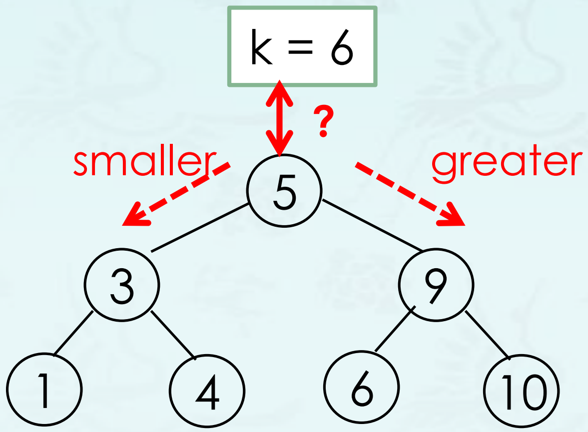
Binary search tree(BST) node structure:

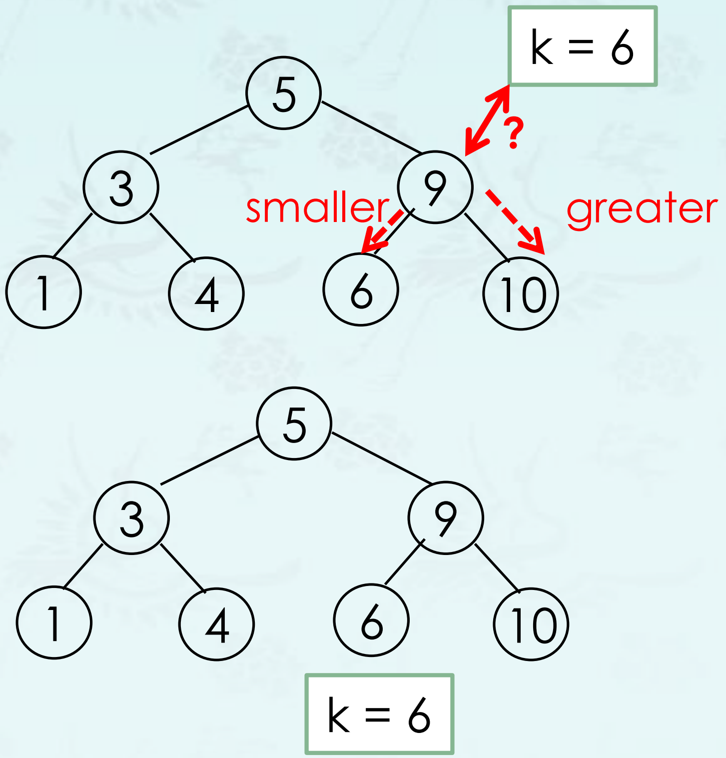
key	
tree left	tree right

```
struct TreeNode {
  int   key;    // sorted by key
  TreeNode*    left;   // left child
  TreeNode*    right;  // right child
};
using tree = TreeNode*;
```

Operations: Search or "**contains**"

Search(T, k) – search the BST, T for a key k





❖ Search operation takes time O(h), where h is the height of a BST.

Operations: Search or "contains"

```
// does there exist a key-value pair with given key?
// search a key in binary search tree iteratively

int containsIteration(tree node, int key)
{
    if (node == nullptr) return false;
    while (node) {
        if (key == node->key) return true;
        if (key  < node->key)
            node = node->left;
        else
            node = node->right;
    }
    return false;
}
```

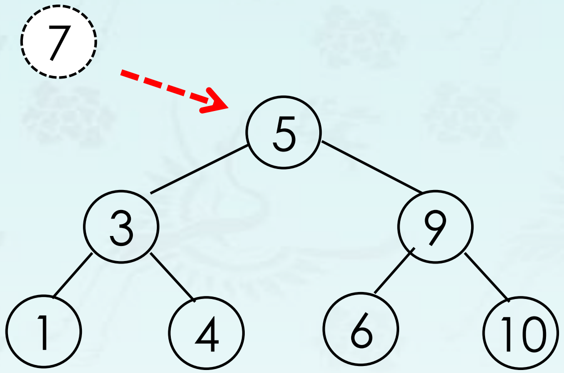
Operations: Search or "contains"

```
// does there exist a key-value pair with given key?
// search a key in binary search tree recursively

int contains(tree node, int key)
{
    if (node == nullptr)         return false;

    if (key  == node->key) return true;

    if (key  <  node->key)
        return contains(node->left, key);

    return contains(node->right, key);
}
```

Binary search trees

Operations: grow

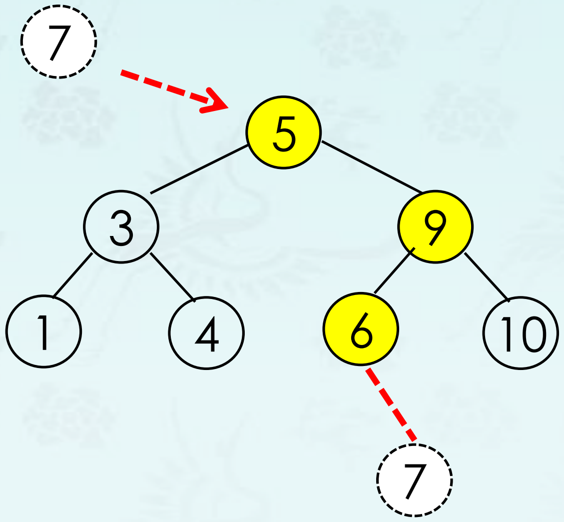
- grow(T, k)
  - Insert a node with Key = k into BST **T**
  - Time complexity? O(h)
- **Step 1**:
  if the tree is empty, then Root(**T**) = k
- **Step 2**:
  Pretending we are searching for k in BST, until we meet a nullptr node
- **Step 3**:
  Insert k



Q: Where is it inserted at?

Operations: grow

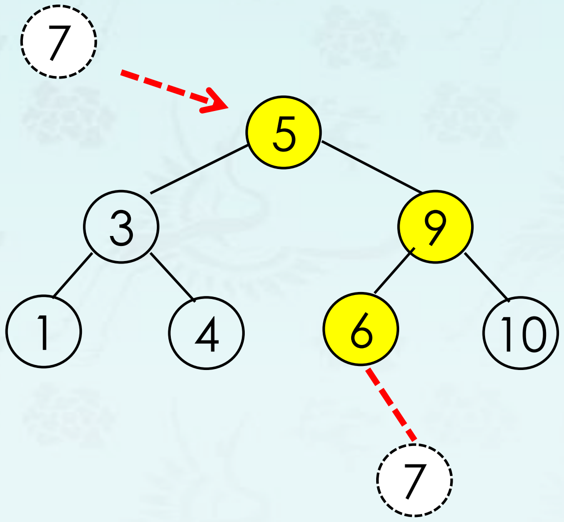
- grow(T, k)
  - Insert a node with Key = k into BST **T**
  - Time complexity? O(h)
- **Step 1**:
  if the tree is empty, then Root(**T**) = k
- **Step 2**:
  Pretending we are searching for k in BST, until we meet a nullptr node
- **Step 3**:
  Insert k



The light nodes are compared with key.

Operations: grow

- grow(T, k)
  - Insert a node with Key = k into BST **T**
  - Time complexity? O(h)
- **Step 1**:
  if the tree is empty, then Root(**T**) = k
- **Step 2**:
  Pretending we are searching for k in BST, until we meet a nullptr node
- **Step 3**:
  Insert k

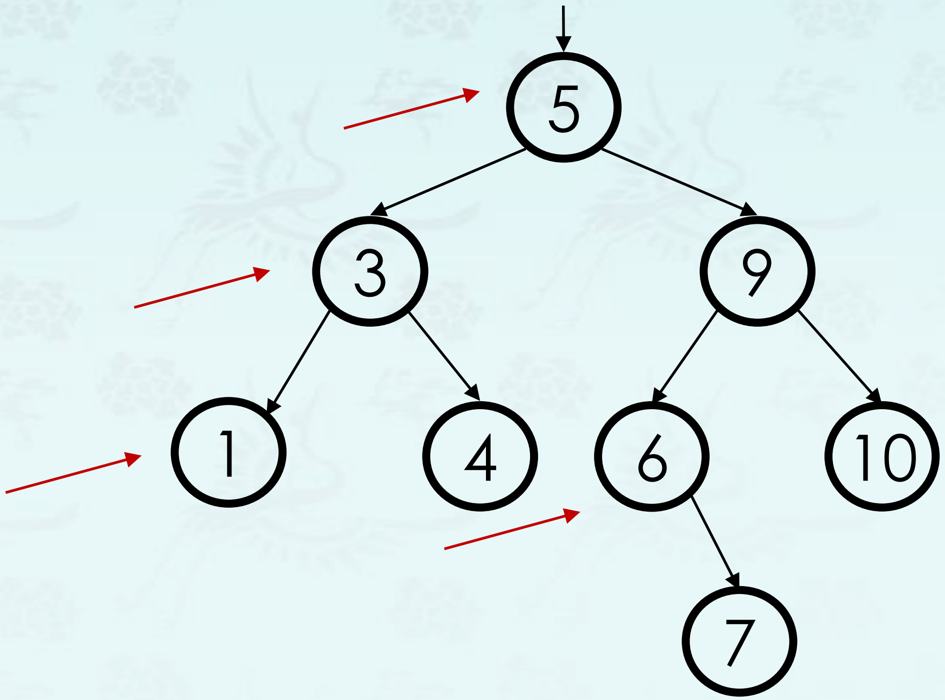


The light nodes are compared with key.

Q: Do you see the difference between the complete binary tree and binary search tree?

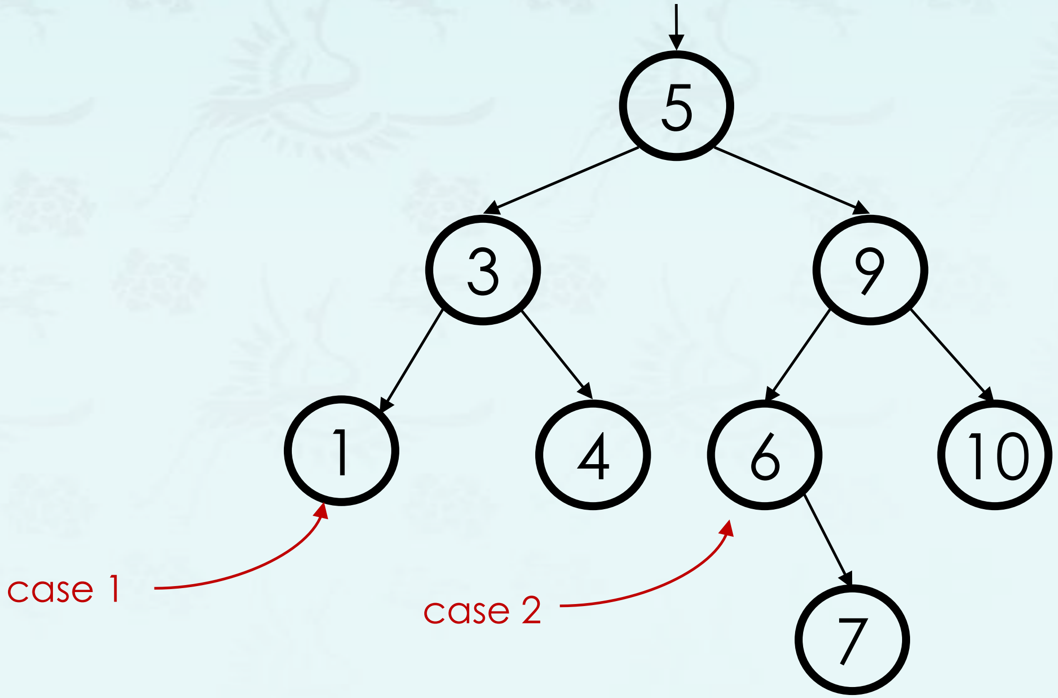
Operations: trim

- How can we **trim** a node from a BST in such a way as to maintain proper BST ordering?
  - trim(1);
  - trim(3);
  - trim(6);
  - trim(5);



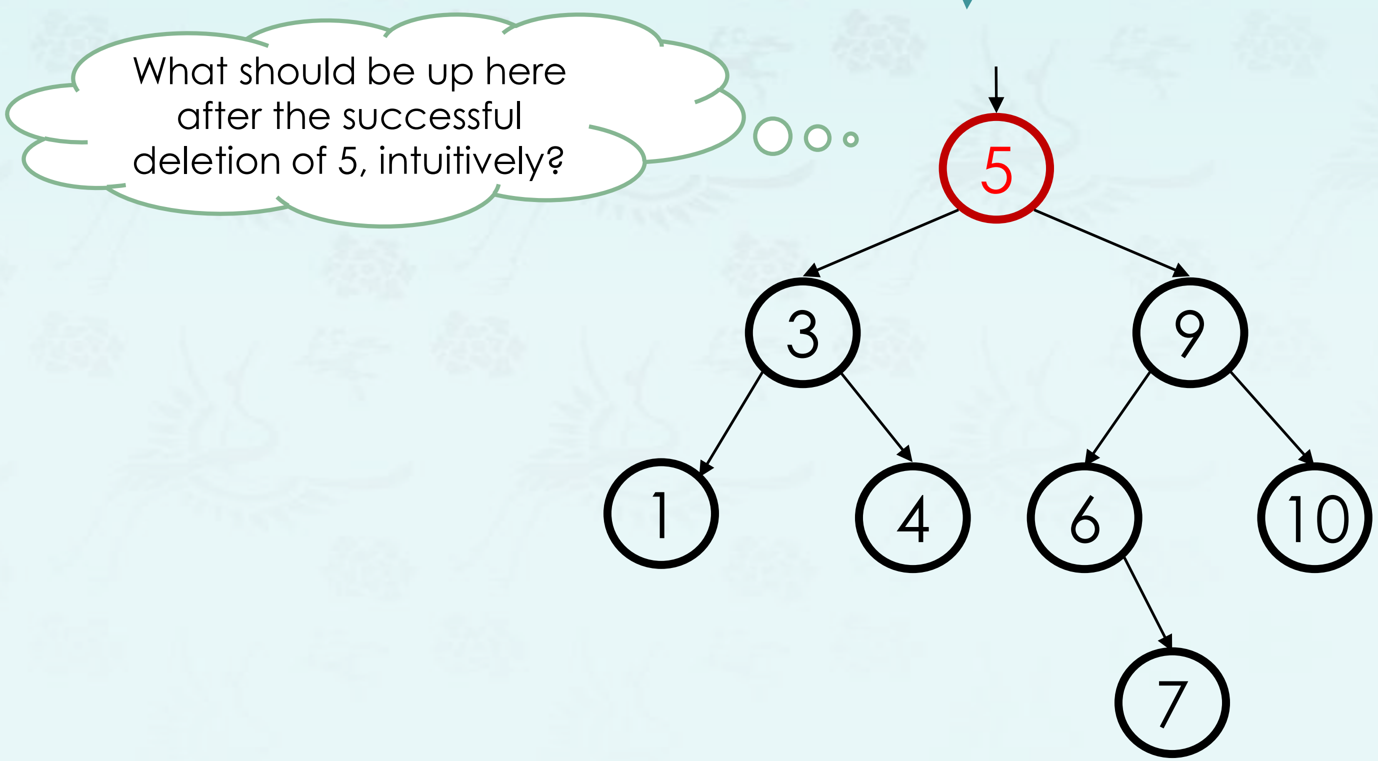
Operations: trim

- **case 1**: leaf
  - a leaf - replace with nullptr
- **case 2**: one child case
  - a node with a left child only - replaced with left child
  - a node with a right child only - replaced with right child
- **case 3**: ?



Operations: trim

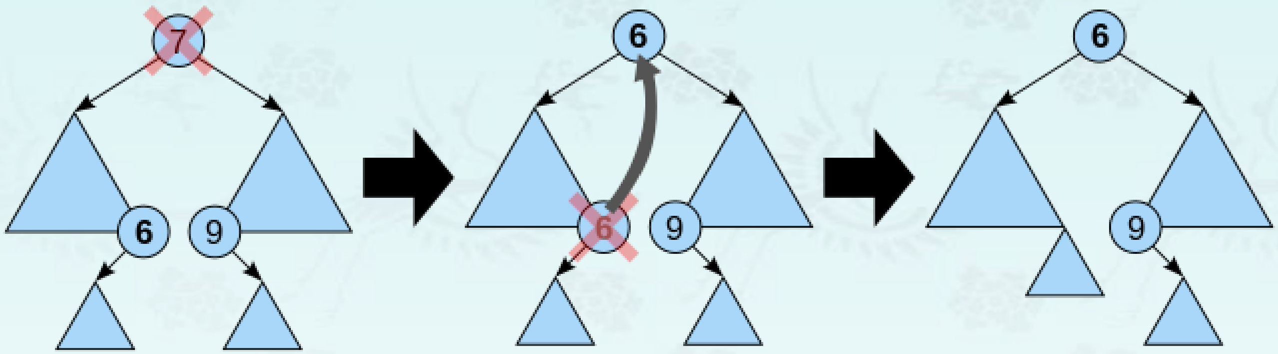
- **case 3**: two children case
  - What can we replace **5** with?



Operations: trim

- **case 3**: two children case

  Where is predecessor or successor of root 7?



1. The rightmost node in the left subtree, the inorder **predecessor 6**, is identified.
2. Its value is copied into the node being trimmed.
3. The inorder **predecessor** can then be trimd because it has at most one child.

NOTE: The same method works symmetrically using the inorder **successor** labelled **9**.

Operations: trim

- **case 3**: two children case

Idea: Replace the trimmed node with a value guaranteed to be between two child subtrees

Options:

- ***predecessor*** from left subtree: **maximum(node->left)**
- ***successor*** from right subtree: **minimum(node->right)**
  - These are the easy cases of predecessor/successor

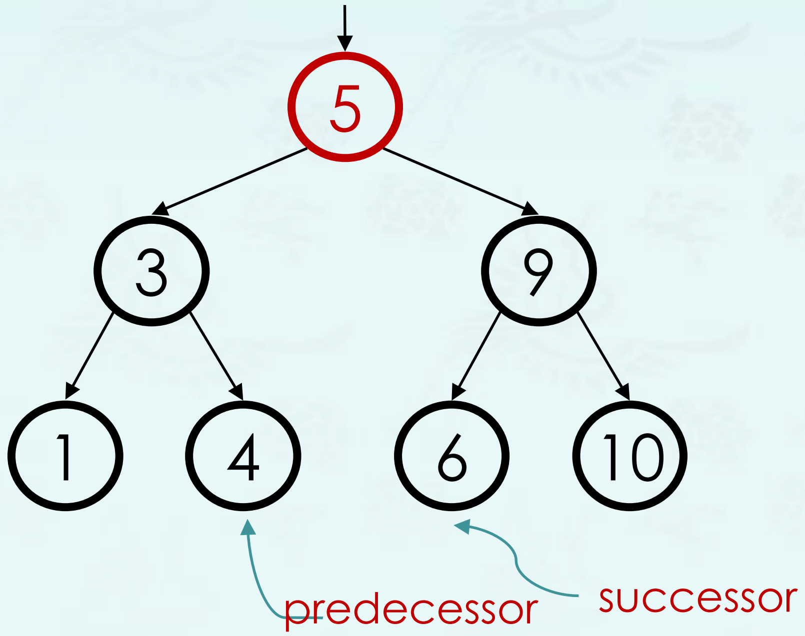
Now trim the original node containing *successor* or *predecessor*

- It becomes leaf or one child case – easy cases of trim!

Operations: trim

- **case 3**: two children case
  - Replace with min from right or max from left
  - **Where is predecessor or successor of root 5?**

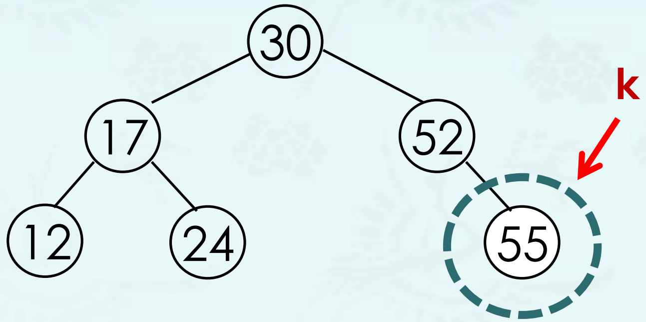
trim(5);

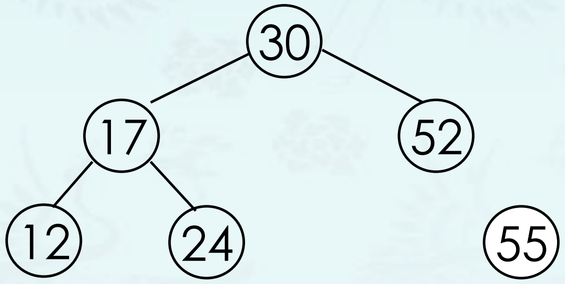


Operations: trim

- trim(**T**, k)
  - trim a node with Key = k into BST **T**
  - Time complexity: O(h)

Case 1: k has no child



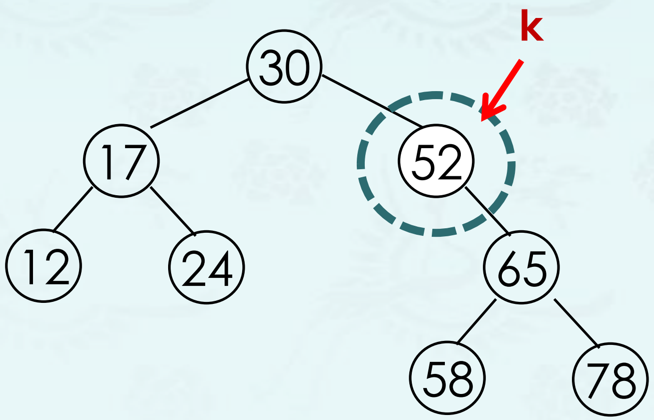


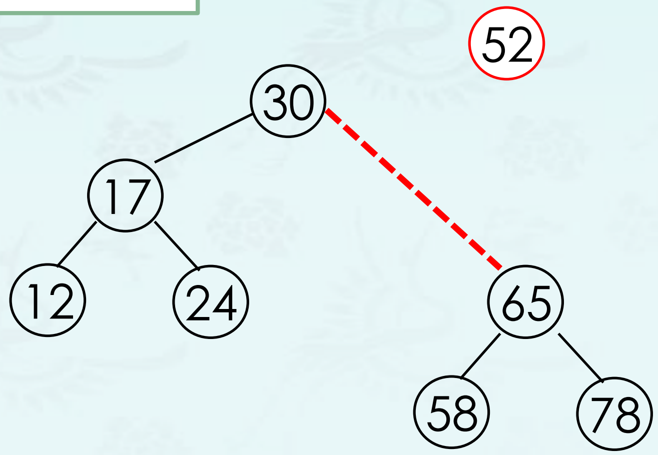
We can simply trim it from the tree

Operations: trim

- trim(**T**, k)
  - trim a node with Key = k into BST **T**
  - Time complexity: O(h)

Case 2: k has one child



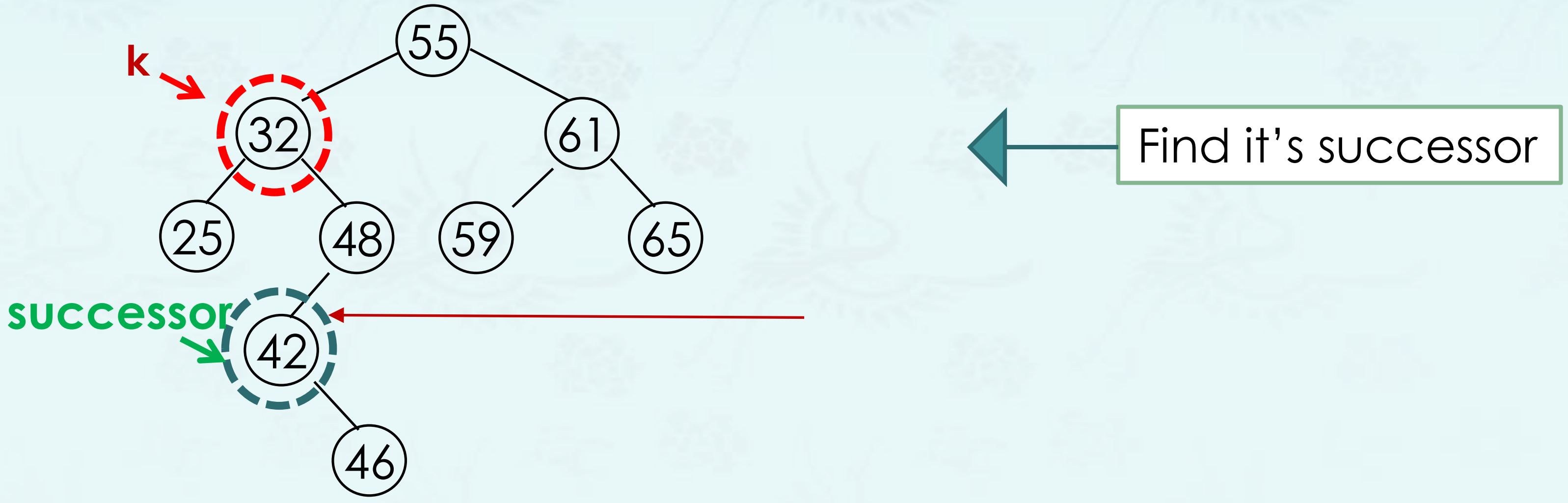


After removing it, connect it's subtree to it's parent node.

Operations: trim

- trim(**T**, k)
  - trim a node with Key = k into BST **T**
  - Time complexity: O(h)

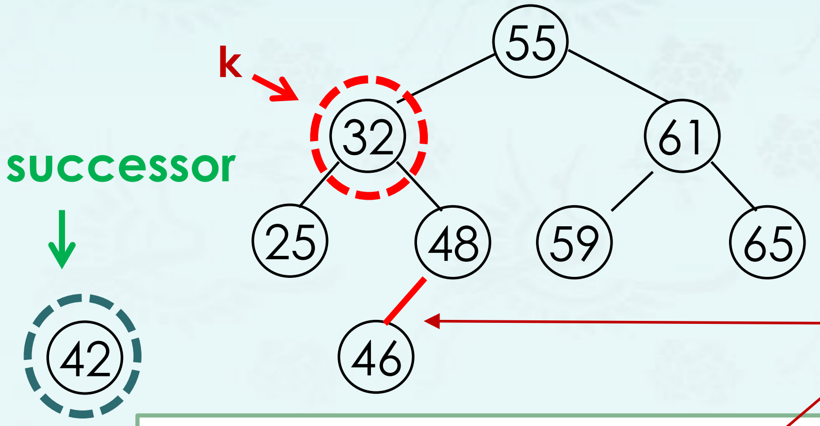
Case 3: k has two children



Operations: trim

- trim(**T**, k)
  - trim a node with Key = k into BST **T**
  - Time complexity: O(h)

Case 3: k has two children



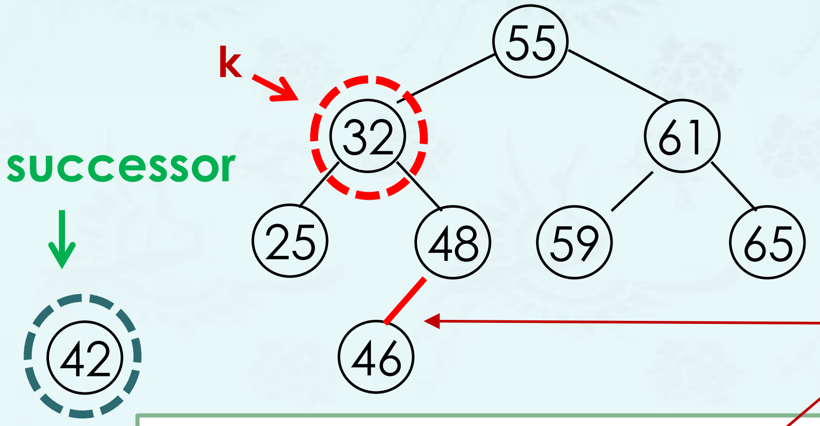
This is done by calling another trim() with succ key, recursively.

Pull out successor,
and connect the tree with it's child

Operations: trim

- trim(**T**, k)
  - trim a node with Key = k into BST **T**
  - Time complexity: O(h)

Case 3: k has two children




```
int succ = value(minimum(root->right));
root->key = succ;
root->right = trim(root->right, succ);
```

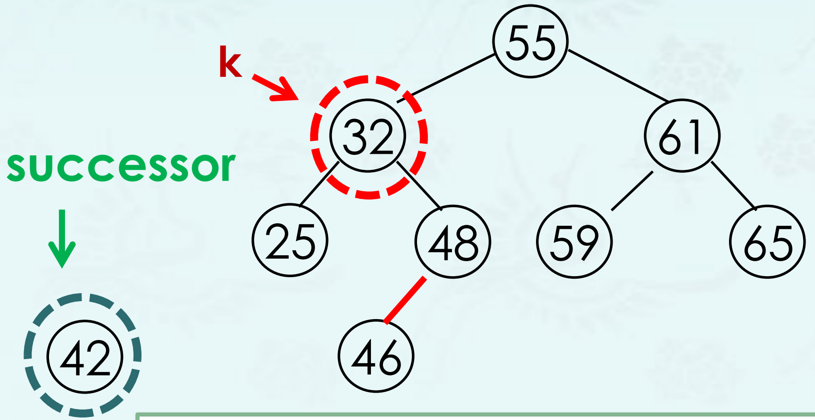
This is done by calling another trim() with succ key, recursively.

Pull out successor,
and connect the tree with it's child

Operations: trim

- trim(**T**, k)
  - trim a node with Key = k into BST **T**
  - Time complexity: O(h)

Case 3: k has two children



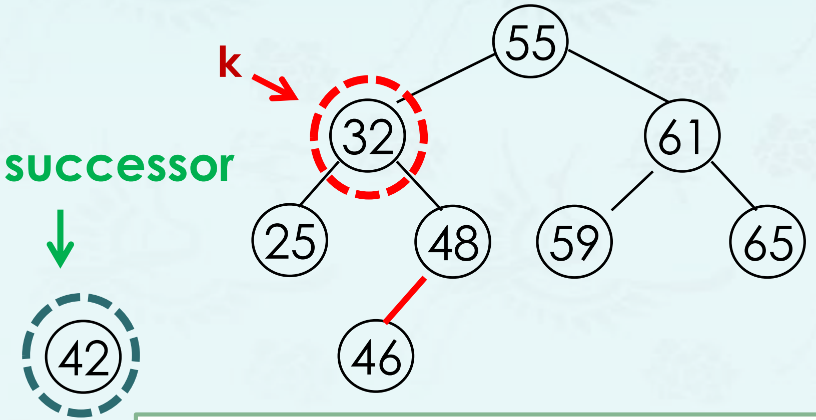
Pull out successor,
and connect the tree with it's child

Q: What if successor has **two** children?

Operations: trim

- trim(**T**, k)
  - trim a node with Key = k into BST **T**
  - Time complexity: O(h)

Case 3: k has two children



A: Not possible !
Because if it has two nodes, at least one of them is less than it, then in the process of finding successor, we won't pick it !

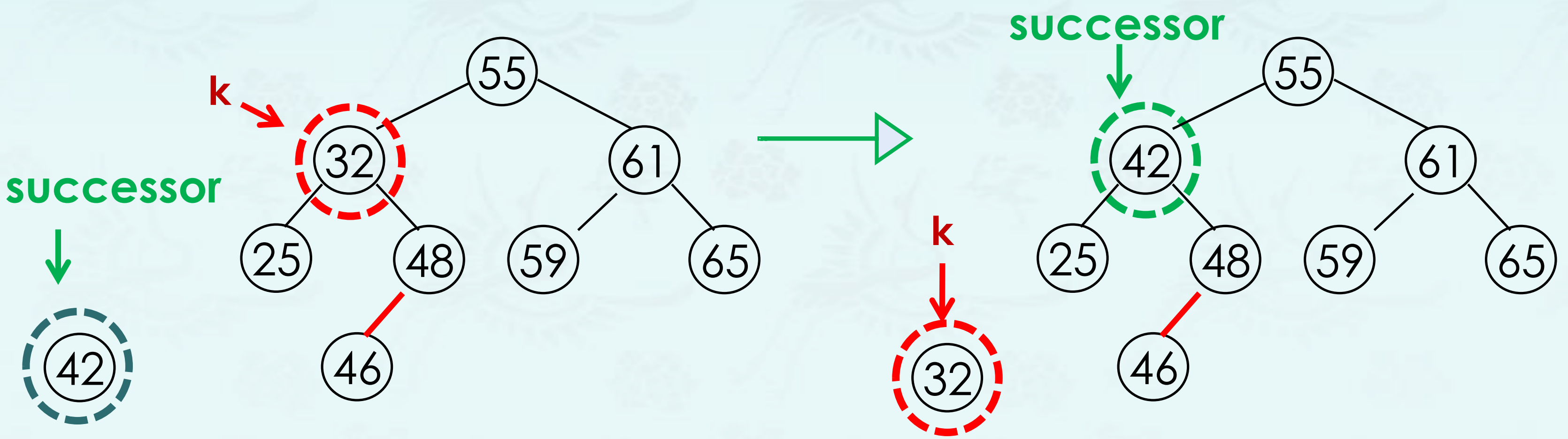
Pull out successor,
and connect the tree with it's child

Q: What if successor has **two** children?

Operations: trim

- trim(**T**, k)
  - trim a node with Key = k into BST **T**
  - Time complexity: O(h)

Case 3: k has two children



Replace the **key** with it's successor

More Operations:

- Query – search, min/max, successor, predecessor

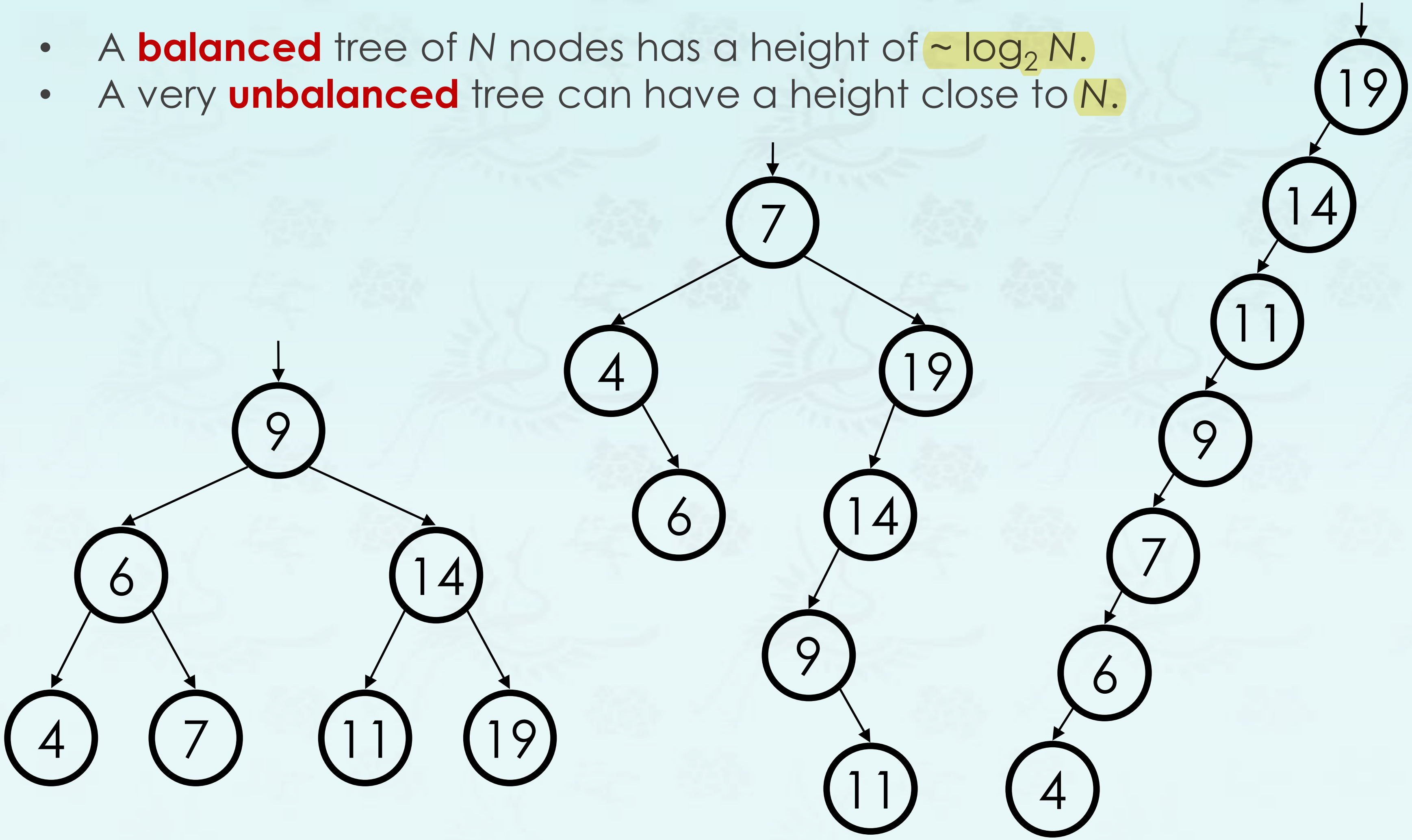
Min/max

- For min, we simply follow the left pointer until we find a nullptr node. Why?
- Similar for Max
- Time complexity: O(h)

❖ Search operation takes time O(h), where h is the height of a BST.

Observations: What do you see in the following BSTs?

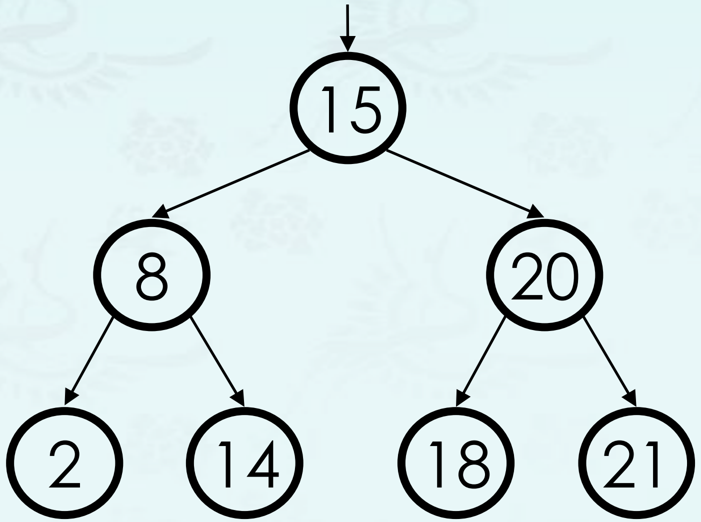
- A **balanced** tree of N nodes has a height of $\sim \log_2 N$.
- A very **unbalanced** tree can have a height close to N.



Observations: What do you see in the following BSTs?

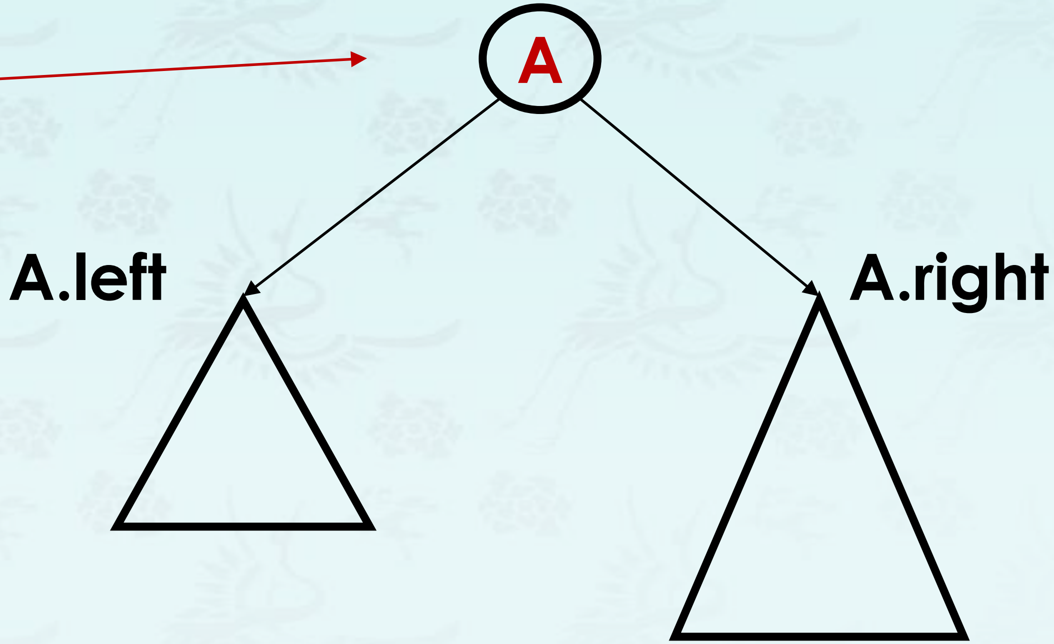
- *Observation:* The shallower the BST the better.
  - Average case height is O(log N)
  - Worst case height is O(N)
  - Simple cases such as adding (1, 2, 3, ..., N), or the opposite order, lead to the worst case scenario: height O(N).

- For binary tree of height h:
  - max # of leaves: 2^{h-1}
  - max # of nodes: $2^h - 1$
  - min # of leaves: 1
  - min # of nodes: h



Q: Calculate tree height.

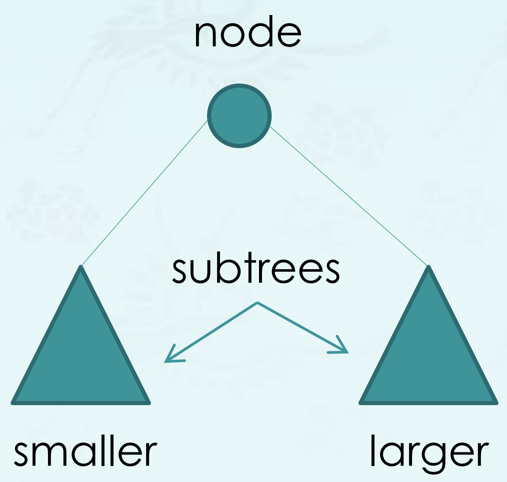
- ***Height*** *is* max number of nodes in path from root to any leaf.
  - height(nullptr) = 0
  - height(a leaf) = ?
  - height(**A**) = ?
- ***Hint***:
  - use recursive.
  - use max(a, b).



- ***A:***
  - height(a leaf) = 1
  - height(A) = 1 + max(

Conclusion:

- If you have a sorted sequence, and we want to design a data structure for it
- **Array or BST? and why?**



Conclusion:

- If you have a sorted sequence, and we want to design a data structure for it
- **Array or BST? and why?**

Time Complexity	
BST	$O(h)$
Array	$O(\log n)$

Conclusion:

Q. When searching, we're traversing a path (since we're always moving to one of the children); since the length of the longest path is the height h of the binary search tree, then finding an element takes $O(h)$.

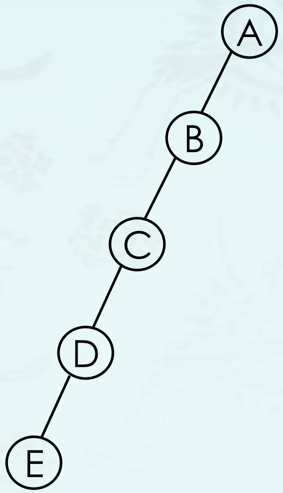
Since $h = \lg n$ (where n is the number of elements), then it's good! – right?

No, of course, it is wrong! Why?

A. The nodes could be arranged in linear sequence in BST, so the $\boldsymbol{height}$ $\boldsymbol{h}$ could be $\boldsymbol{n}$. In worst case, it is $\boldsymbol{O(n)}$ instead of $\boldsymbol{O(h)}$.

Conclusion:

- We already know that $\boldsymbol{n}$ is fixed, but $\boldsymbol{h}$ differs from how we insert those elements!
- So why we still need BST?
  - Easier insertion and deletion
  - And with some optimization, we can avoid the worst case!



$$\boldsymbol{n = h}$$

a skew binary search tree