Tree

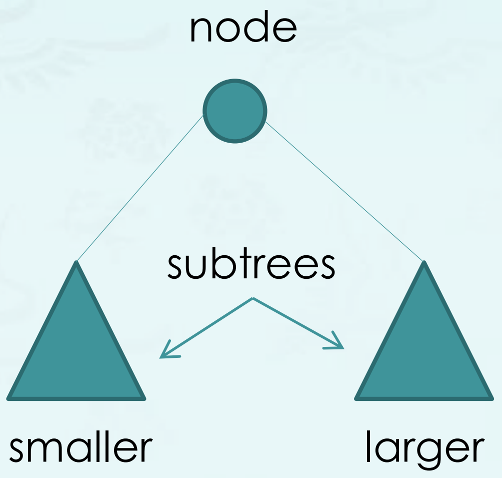
- *introduction*
- *binary tree*
- *complete binary tree*
  - *max heap, min heap*
  - *Chapter 7 – heap sorting*
  - *Chapter 9 - priority queues*
- ***binary search tree***

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

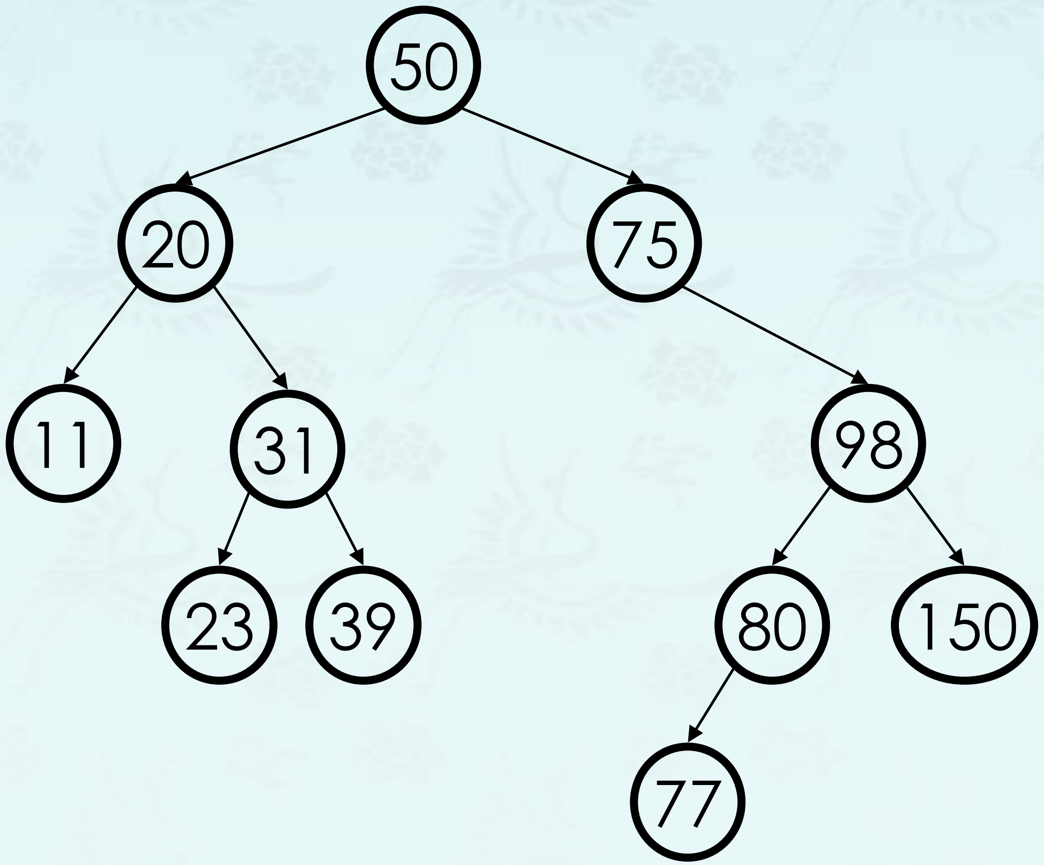
- Symmetric order means that
  - every node has a key
  - every node's key is
    larger than all keys in its left subtree
    smaller than all keys in its right subtree



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:

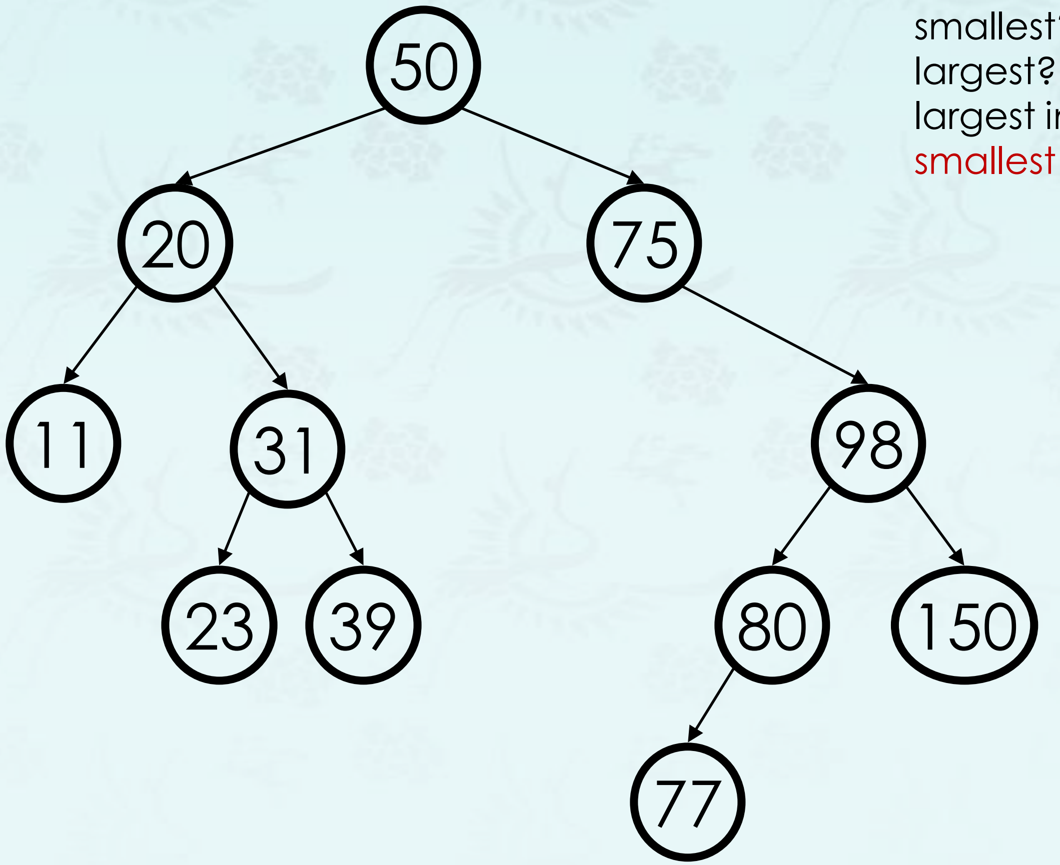
50
20
75
98
80
31
150
39
23
11
77



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:

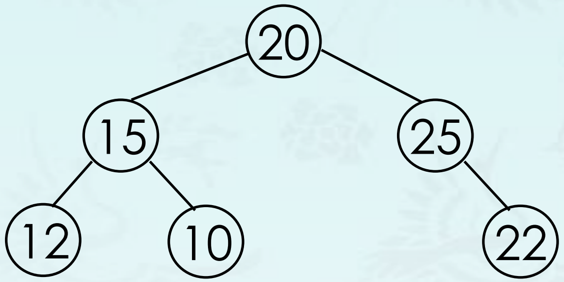
50
20
75
98
80
31
150
39
23
11
77



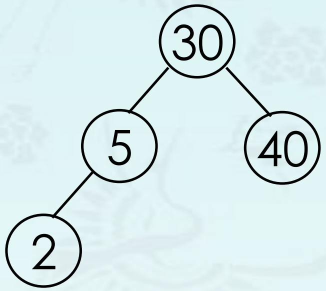
smallest?
largest?
largest in left?
smallest in right?

Binary search trees

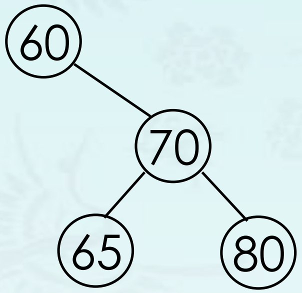
Definition: A binary search tree is a binary tree in symmetric order.
Exercise: Identify non-BST(s) and correct them if not.



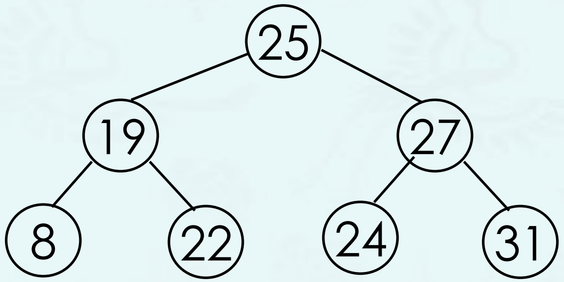
(a)



(b)



(c)

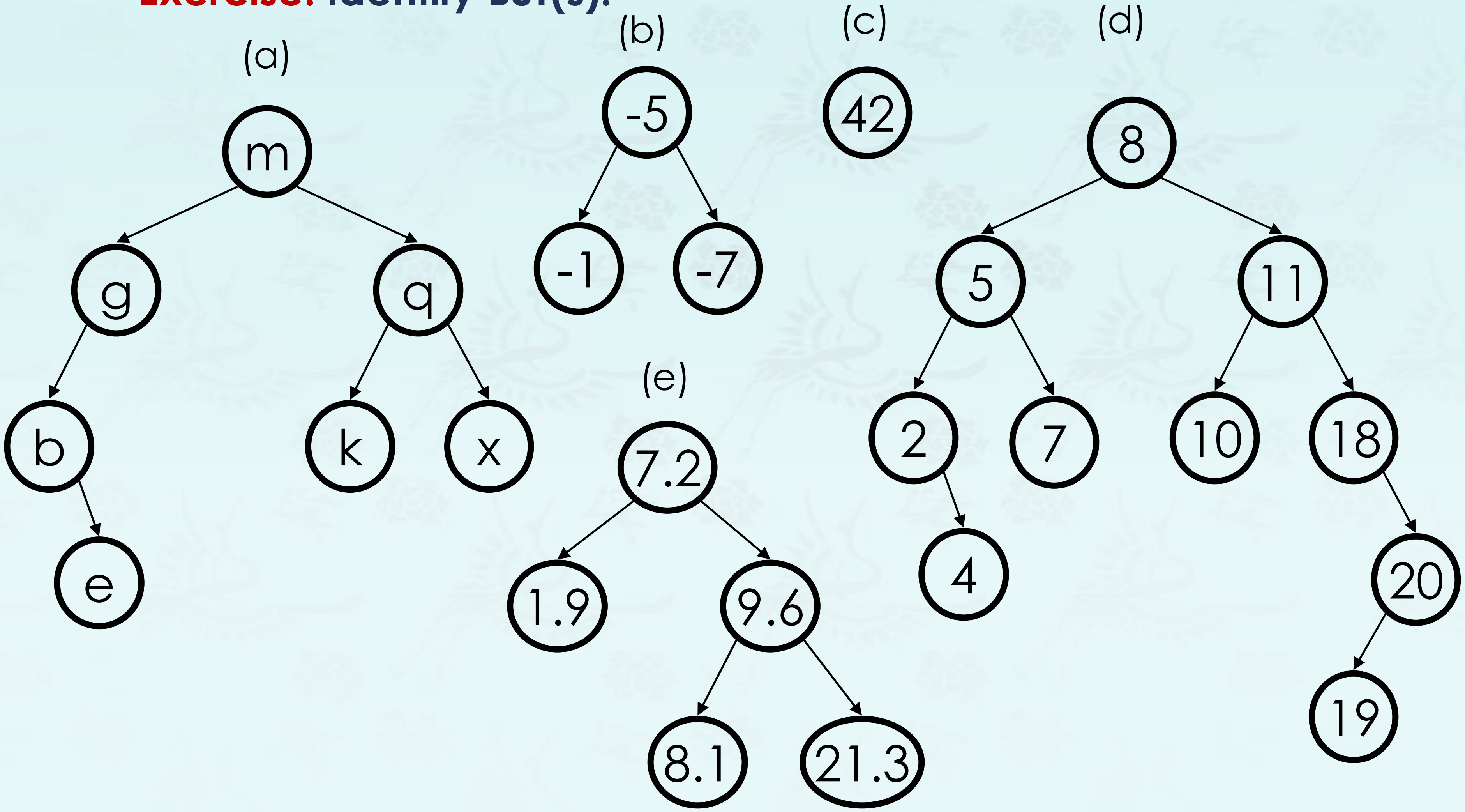


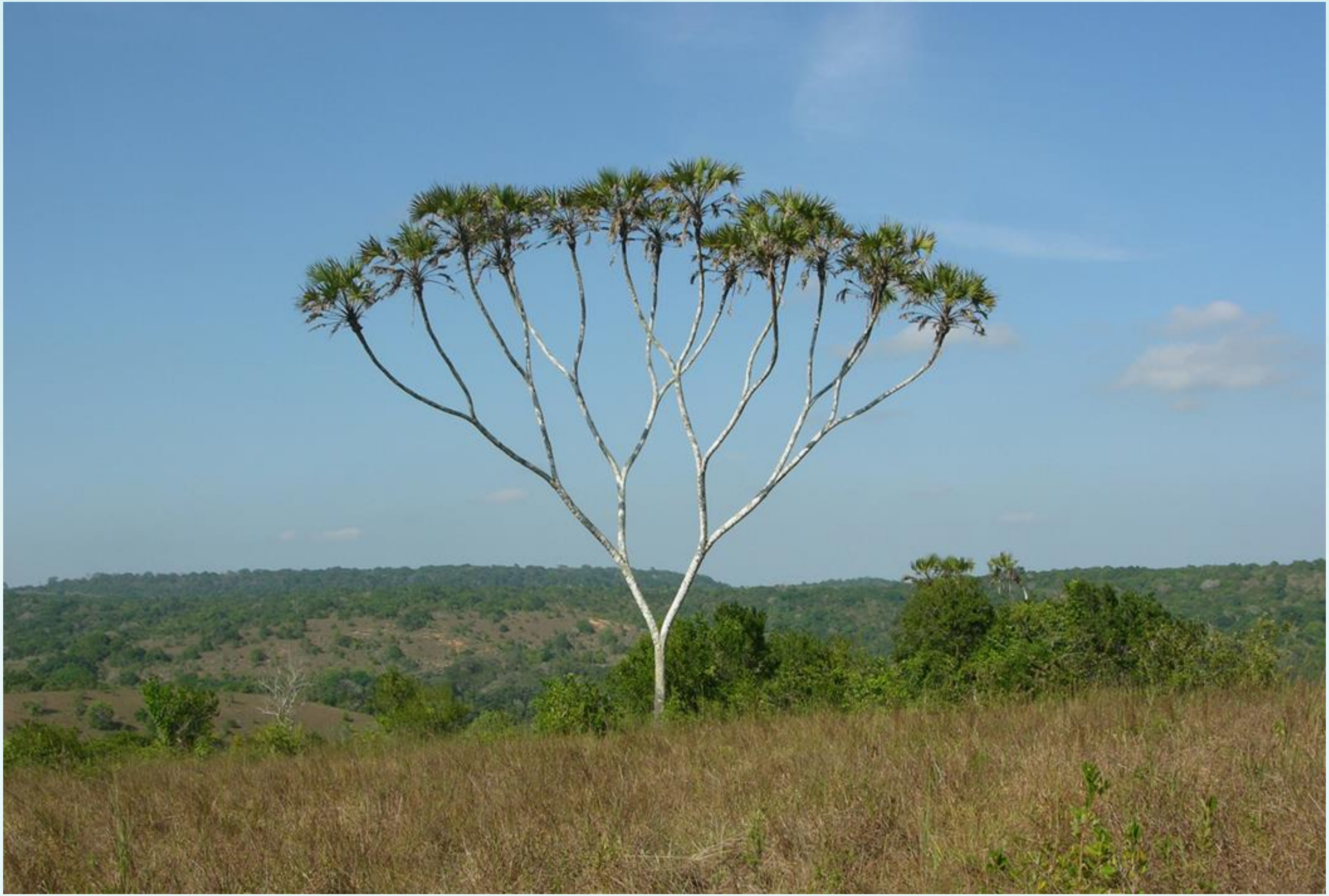
(d)

Binary search trees

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

Exercise: Identify BST(s).





Node structure:

key	
Left	Right

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;    // key
  TreeNode*    left;   // left child
  TreeNode*    right;  // right child
};
using tree = TreeNode*;
```

Binary search trees

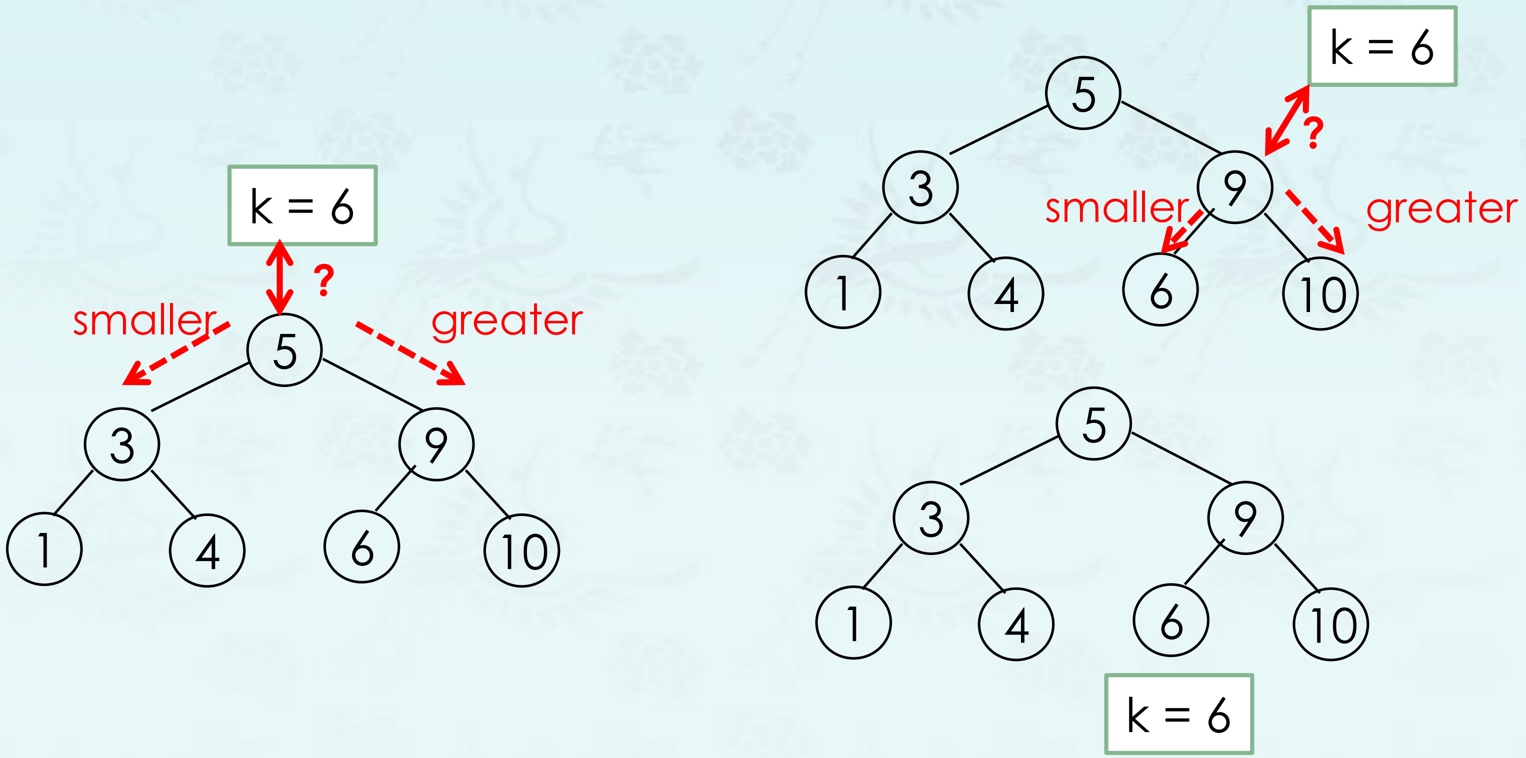
Binary search tree(BST) node structure:

key	
tree left	tree right

```
struct TreeNode{
  int        key;     // sorted by key
  TreeNode* left;     // left child
  TreeNode* right;    // right child
  TreeNode(int k, TreeNode* l, TreeNode* r) {
    key = k; left = l;right = r;
  }
  TreeNode(int k) : key(k), left(nullptr), right(nullptr) {}
  ~TreeNode(){}
};
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"

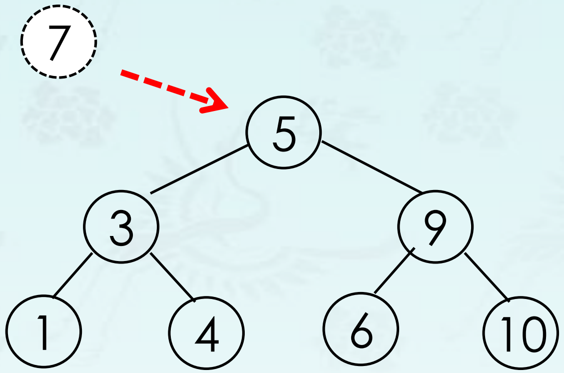
```
// does there exist a key-value pair with given key?
// search a key in binary search tree iteratively

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);
}
```

Operations: grow

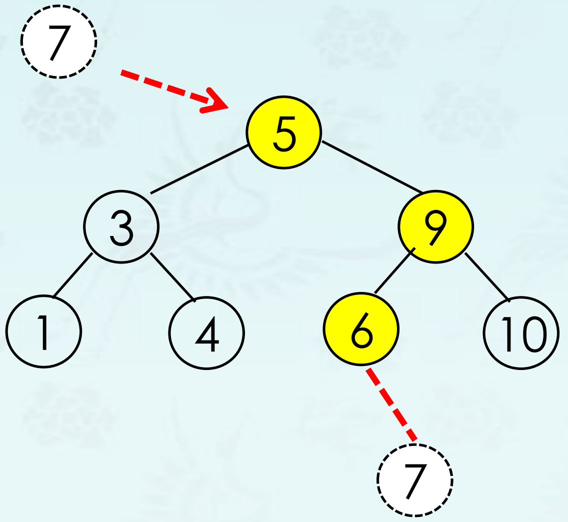
- grow(T, k)
  - Insert a node with Key = k into BST **T**
  - Time complexity? O(h)
- **Step 1**:
  if the tree is empty, then Root(**T**) = k
- **Step 2**:
  Pretending we are searching for k in BST, until we meet a nullptr node
- **Step 3**:
  Insert k



Q: Where is it inserted at?

Operations: grow

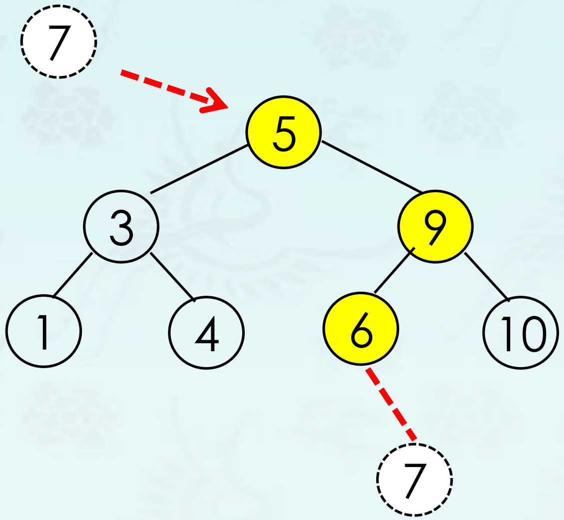
- grow(T, k)
  - Insert a node with Key = k into BST **T**
  - Time complexity? O(h)
- **Step 1**:
  if the tree is empty, then Root(**T**) = k
- **Step 2**:
  Pretending we are searching for k in BST, until we meet a nullptr node
- **Step 3**:
  Insert k



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?

grow: insert a new node with given key in BST

Something wrong?

```
tree grow(tree node, int key) {
  if (node == nullptr)
  if (key < node->key)   // recur down the tree
    grow(node->left, key);
  else
    grow(node->right, key);
  else
    cout << "grow: the same key " << key << " is ignored.\n";

  return node;
}
```

grow: insert a new node with given key in BST

Something wrong?

```
tree grow(tree node, int key) {
  if (node == nullptr) return new Tree(key);
  if (key < node->key)  // recur down the tree
    grow(node->left, key);
  else
    grow(node->right, key);
  else
    cout << "grow: the same key " << key << " is ignored.\n";

  return node;
}
```

grow: insert a new node with given key in BST

Something wrong?

```
tree grow(tree node, int key) {
  if (node == nullptr) return new Tree(key);
  if (key < node->key)  // recur down the tree
    grow(node->left, key);
  else if (key > node->key)
    grow(node->right, key);
  else
    cout << "grow: the same key " << key << " is ignored.\n";

  return node;
}
```

Something wrong?

grow: insert a new node with given key in BST

Something wrong?

```
tree grow(tree node, int key) {
  if (node == nullptr) return new Tree(key);
  if (key < node->key)  // recur down the tree
    grow(node->left, key);
  else if (key > node->key)
    grow(node->right, key);
  else
    ;

  return node;
}
```

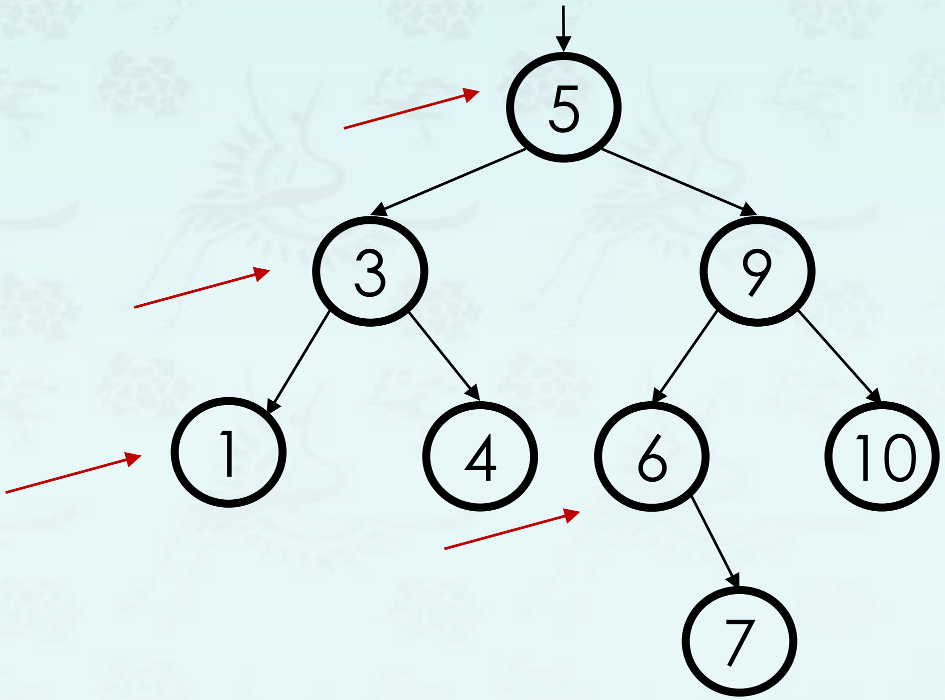
grow: insert a new node with given key in BST

Something wrong?

```
tree grow(tree node, int key) {
  if (node == nullptr) return new Tree(key);
  if (key < node->key)  // recur down the tree
    node->left = grow(node->left, key);
  else if (key > node->key)
    node->right = grow(node->right, key);
  else
    ;

  return node;
}
```

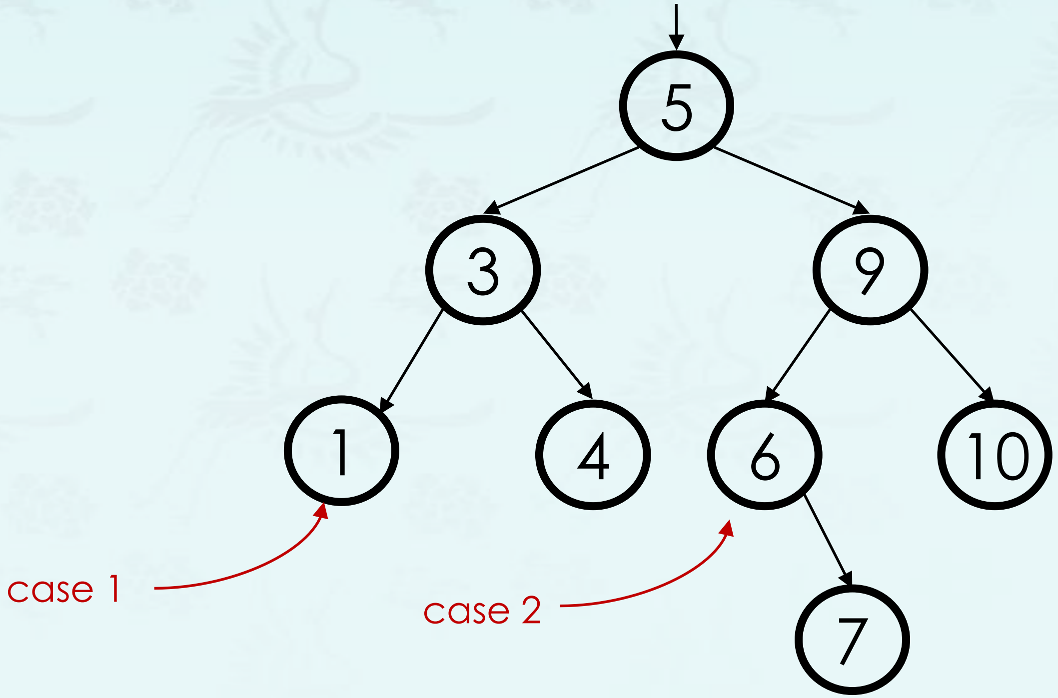
Operations: trim

- 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);



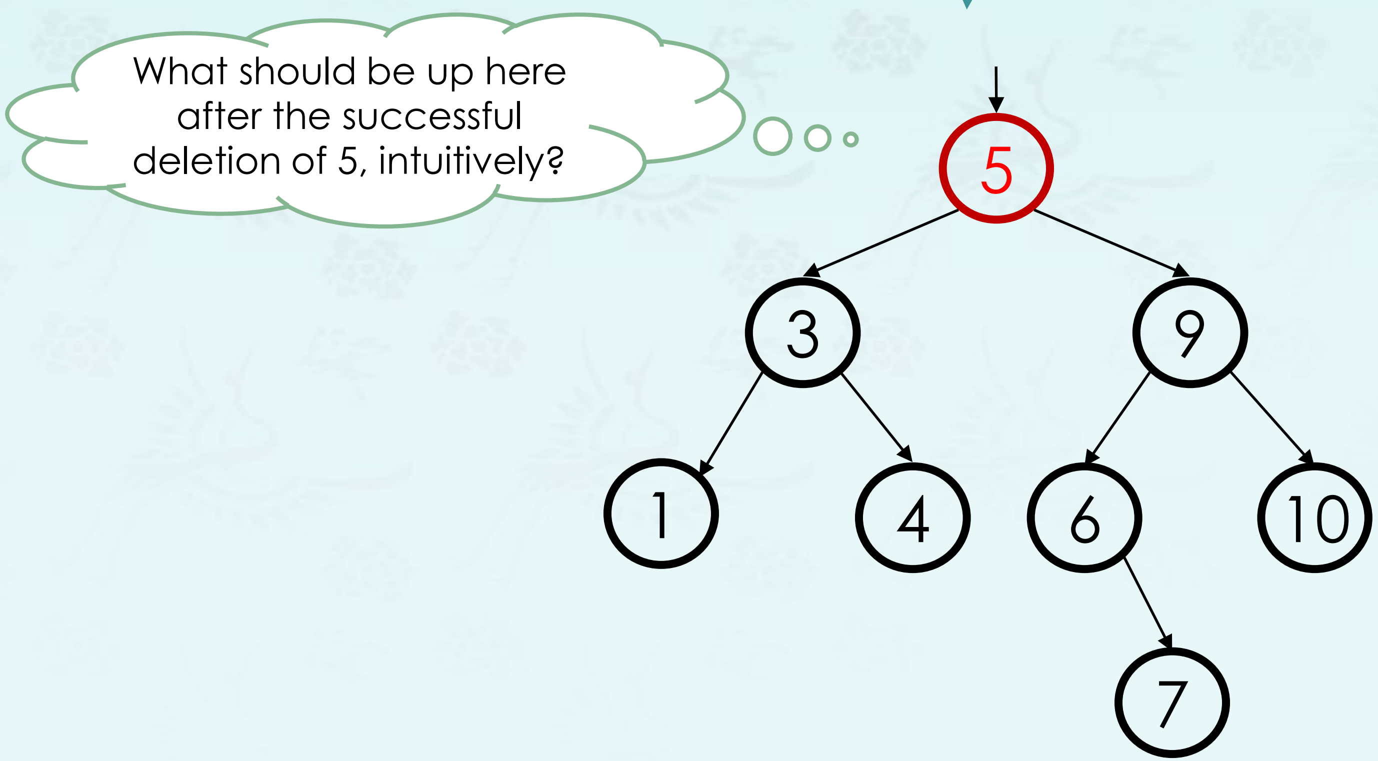
Operations: trim

- **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

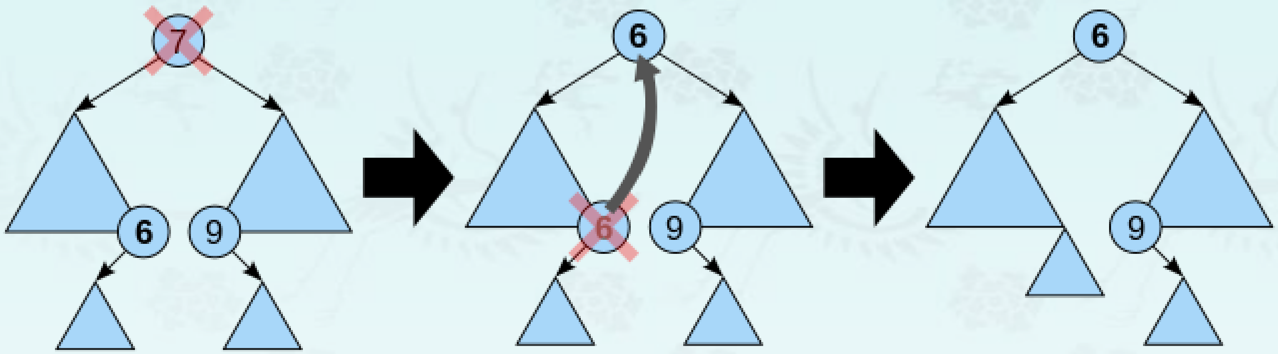
- **case 3**: two children case
  - What can we replace **5** with?



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:

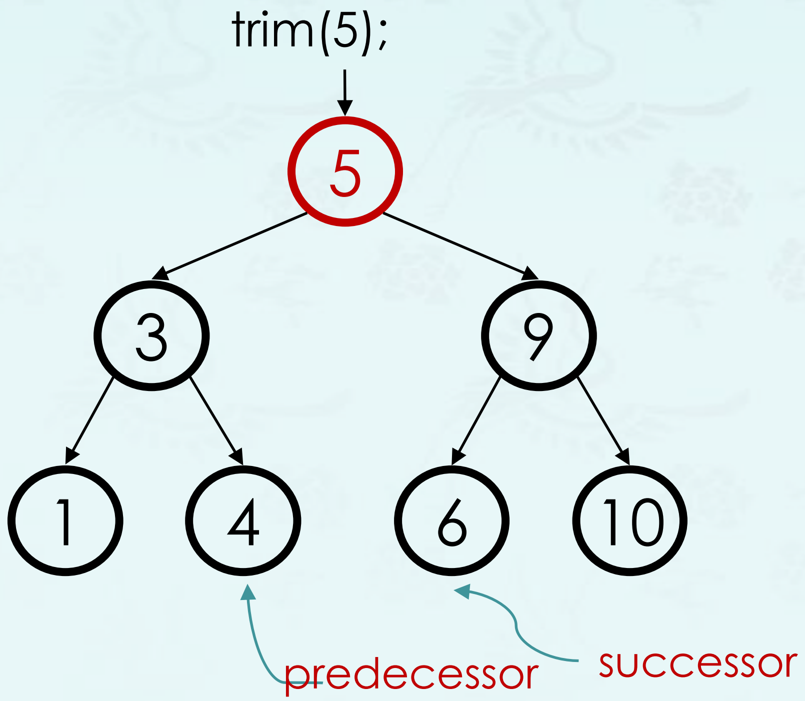
- ***predecessor*** from left subtree: **maximum([redacted])**
- ***successor*** from right subtree: **minimum([redacted])**
    - These are the easy cases of predecessor/successor

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

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

Operations: trim

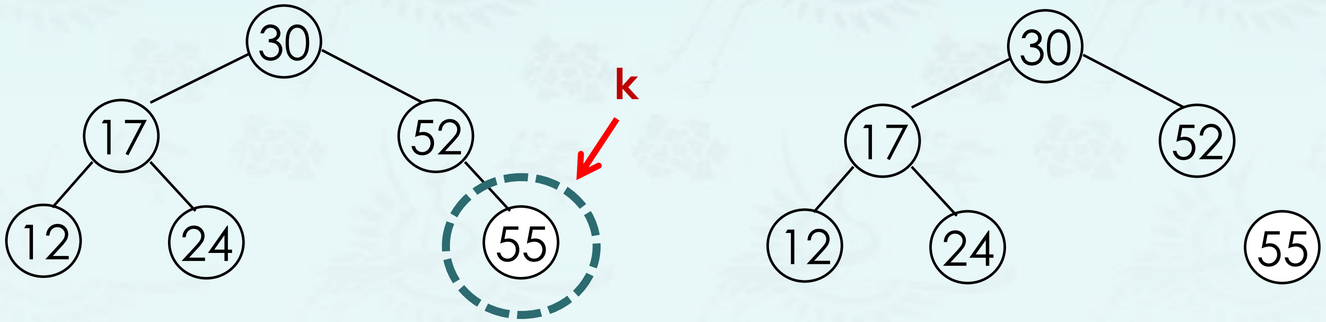
- **case 3**: two children case
  - Replace with min from right or max from left
  - **Where is predecessor or successor of root 5?**



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 **55** from the tree.

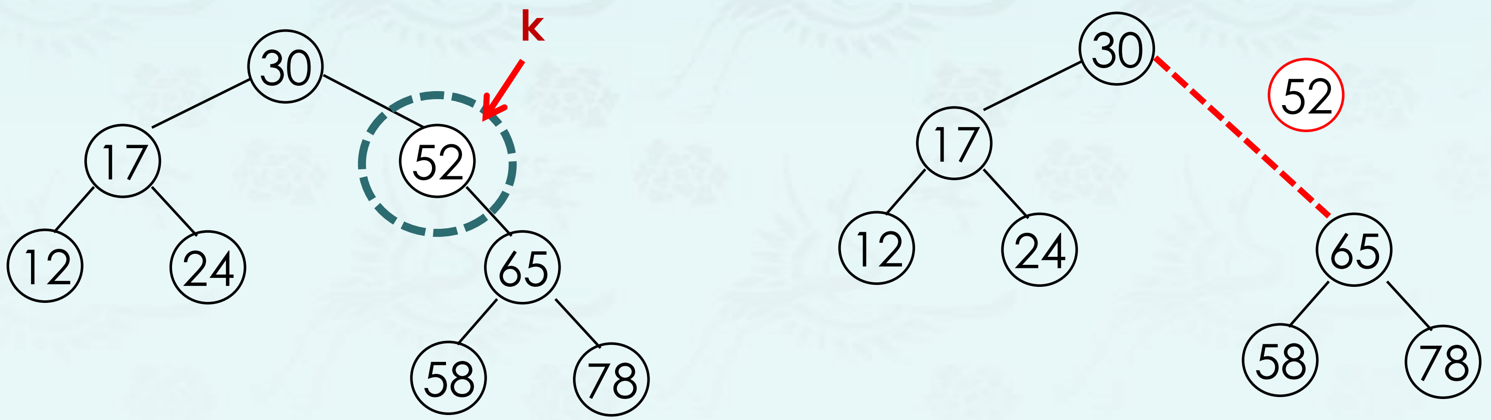
1. delete **55**
2. **52**→right = nulltptr

How?

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.

How?

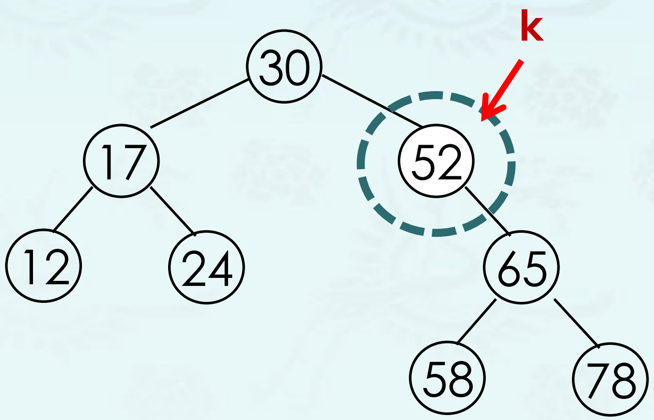
Operations: trim

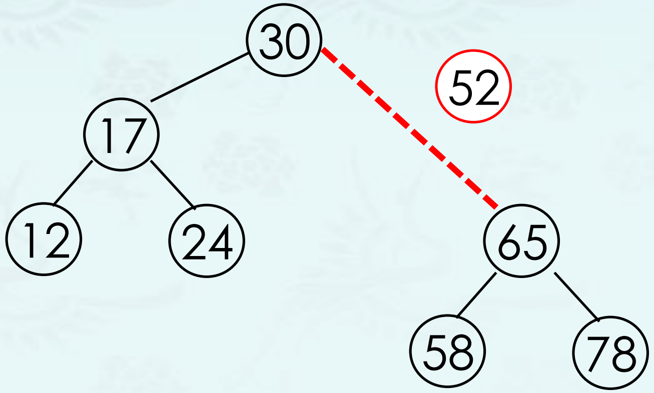
- trim(**T**, k)
  - trim a node with Key = k into BST **T**
  - Time complexity: O(h)

root = trim(root, 52)   // in main()

// in trim(root, key)
root→right = trim(root→right, 52)

Case 2: k has one child





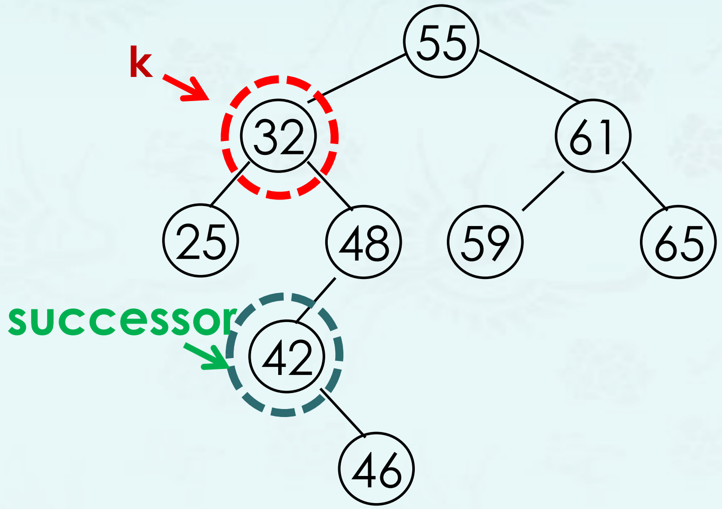
1. delete 52
2. return 52→right

Don't forget to save 52→right before delete 52

Operations: trim

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

Case 3: k has two children

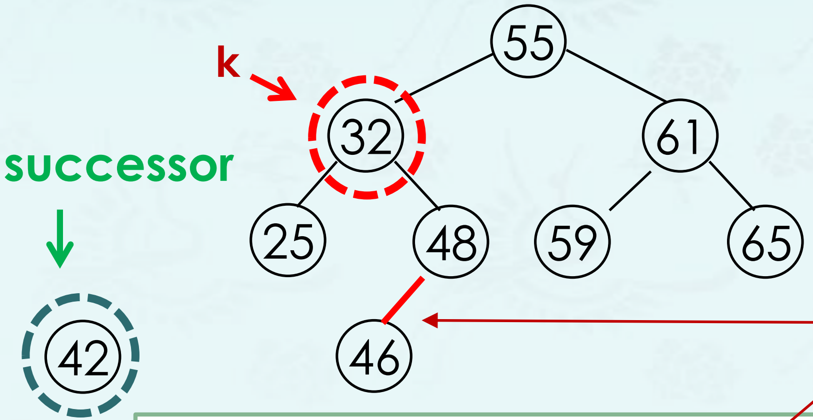



```
1. found the node 32
2. find the successor and its key 42.
3. replace node→32 with 42.
4. node->right = trim(node->right, 42);
```

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);
```

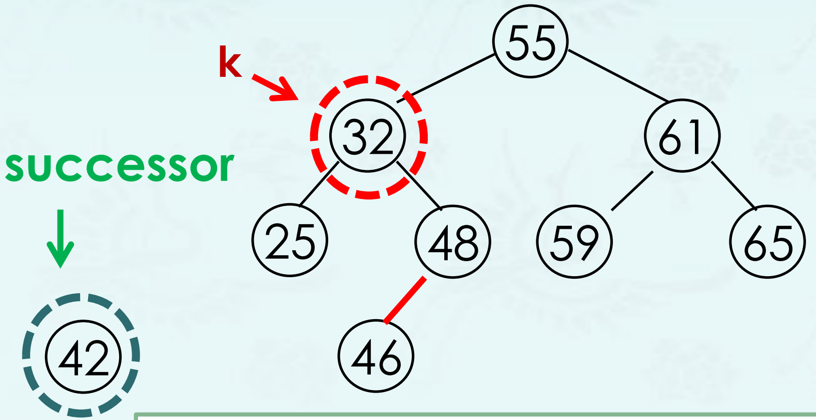
This is done by calling another trim() with succ key, recursively.

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

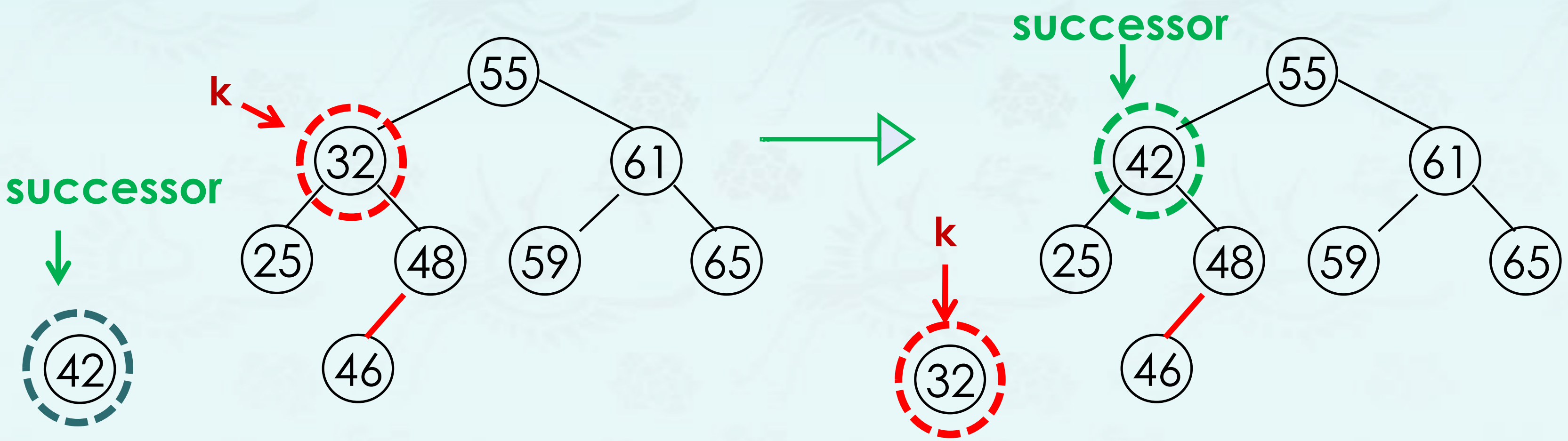
Q: What if successor has **two** children?

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 !

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.

Binary Search Tree

- *Recursion Revisited*
- *binary search tree Implementation*
  - *size*
  - *height*
  - *traversal - inorder, preorder, postorder, levelorder*
  - *minimum, maximum,*
  - *predecessor, successor*

bunnyEars(): counting bunny ears in recursion

```
// each bunny has two ears.
int bunnyEars(int bunnies) {

  return 2 + bunnyEars(bunnies-1);
}
```

funnyEars(): counting funny ears in recursion

```
// even numbered funny has two ears, odd numbered funny three.
int funnyEars(int funnies) {
  if (bunnies == 0) return 0;

  if (funnies % 2 == 0)
    return
  else
    return
}
```

bunnyEars(): counting bunny ears in recursion

```
// each bunny has two ears.
int bunnyEars(int bunnies) {
  if (bunnies == 0) return 0;
  return 2 + bunnyEars(bunnies-1);
}
```

funnyEars(): counting funny ears in recursion

```
// even numbered funny has two ears, odd numbered funny three.
int funnyEars(int funnies) {
  if (bunnies == 0) return 0;

  if (funnies % 2 == 0)
    return 2 + funnyEars(funnies - 1);
  else
    return 3 + funnyEars(funnies - 1);
}
```

size(): in doubly linked list

```
int size(pList p) {
  int count = 0;
  for (pNode c = begin(p); c != end(p); c = c->next)
    count++;
  return count;
}
```

size(): in singly linked list

```
int size(pNode node) {
  if (node->next == nullptr) return 0;
  return 1 + size(node->next);
}
```

Binary search trees

size: Count the number of nodes in the binary tree recursively.

```
int size(tree node) {
  if (node == nullptr) return 0;
  return 1 + size(node->left) + size(node->right);
}
```

size: Count the number of nodes in the binary tree recursively.

```
int size(tree node) {
  if (node == nullptr) return 0;
  cout << " size at: " << node->key << endl;
  return 1 + size(node->left) + size(node->right);
}
```

```
      5
     / \
    /   \
   /     \
  3       7
 / \     / \
2   4   6   8
             \
              10
```

```
size at:
size at:
size at:
size at:
size at:
size at:
size at:
size at:
```

size: Count the number of nodes in the binary tree recursively.

```
int size(tree node) {
  if (node == nullptr) return 0;
  cout << " size at: " << node->key << endl;
  return 1 + size(node->left) + size(node->right);
}
```

```
      5
     / \
    /   \
   /     \
  3       7
 / \     / \
2   4   6   8
             \
             10
```

```
size at: 5
size at: 3
size at: 2
size at: 4
size at: 7
size at: 6
size at: 8
size at: 10
```

Binary search trees

size: Count the number of nodes in the binary tree recursively.

```
int size(tree node) {
  if (node == nullptr) return 0;
  return 1 + size(node->left) + size(node->right);
}
```

height: compute the max height(or depth) of a tree.
// It is the number of nodes along the longest path from the root node
// down to the farthest leaf node.

```
int height(tree node) {


}
```

inorder traversal: do inorder traversal of BST.

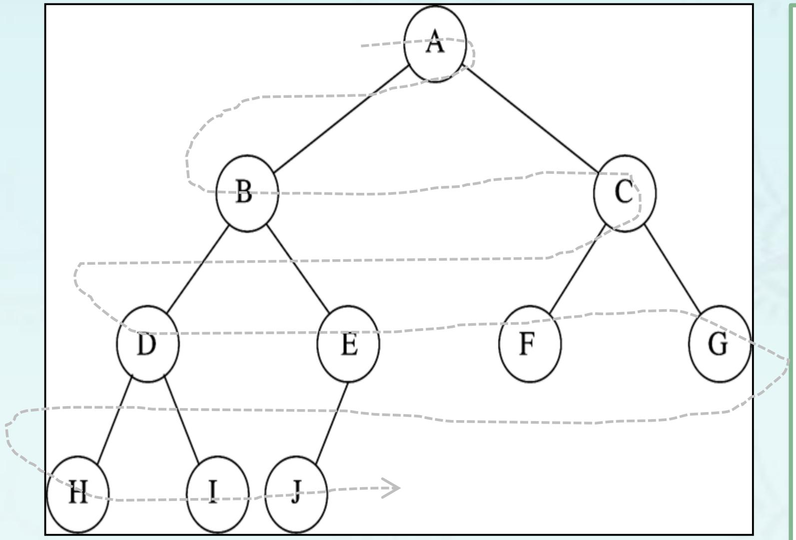
```
void inorder(tree node) {
    if (node == nullptr) return;

    inorder(node->left);
    cout << node->key;
    inorder(node->right);
}
```

```
void inorder(tree node, vector<int>& vec) {
  if (node == nullptr) return;

  inorder(node->left, vec);
          _
  inorder(node->right, vec);
}
```

```
case 'l':
  cout << "\n\tinorder:    ";
  vec.clear();
  inorder(root, vec);
  for (int i : vec)
    cout << i << " ";
```

Level-order traversal

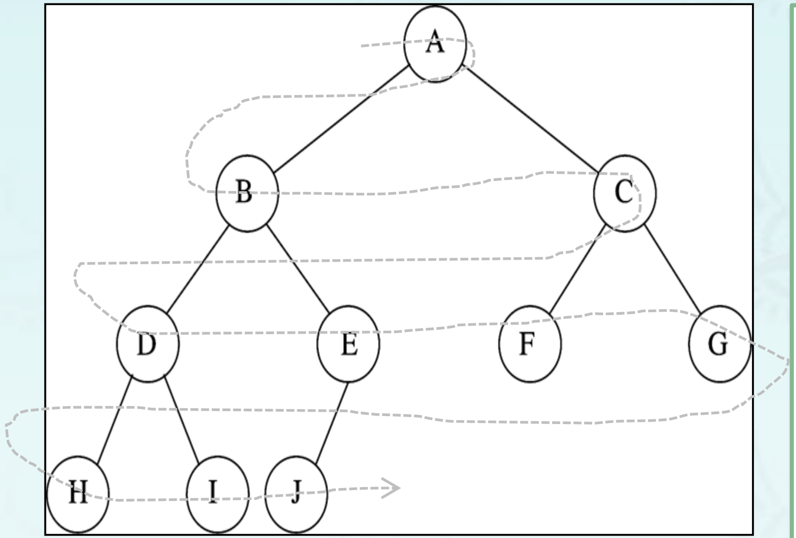
1. **Depth first** search(DFS) – preorder, inorder, postorder traversal
2. **Breadth first** search(BFS) - **level-order** traversal




```
#include <queue>
#include <vector>

void levelorder(tree root, vector<int>& vec)
•  Visit the root. if it is not null, enqueue it.
•  while queue is not empty
    1. que.front() – get the node in the queue
    2. save the key in vec.
    3. if its left child is not null, enqueue it.
    4. if its right child is not null, enqueue it.
    5. que.pop() – remove the node in the queue.
```

Level-order traversal

1. **Depth first** search(DFS) – preorder, inorder, postorder traversal
2. **Breadth first** search(BFS) - **level-order** traversal




```
#include <queue>
#include <vector>

void levelorder(tree root, vector<int>& vec) {
  queue<tree> que;
  if (!root) return;
  que.push(root);
  while ...{


     cout << "your code here\n";


  }
}
```

minimum, maximum:

returns the node with min or max key.
Note that the entire tree does not need to be searched.

```
tree minimum(tree node) {   // returns left-most node key


}
```

```
tree maximum(tree node) {   // returns right-most node key


}
```

pred(), succ() – predecessor, successor:

```
Input: root node, key
output: predecessor node, successor node

1. If root is nullptr, then return

2. if key is found then
    a. If its left subtree is not nullptr
        Then predecessor will be the right most
        child of left subtree or left child itself.
    b. If its right subtree is not nullptr
        The successor will be the left most child
        of right subtree or right child itself.
    return
```

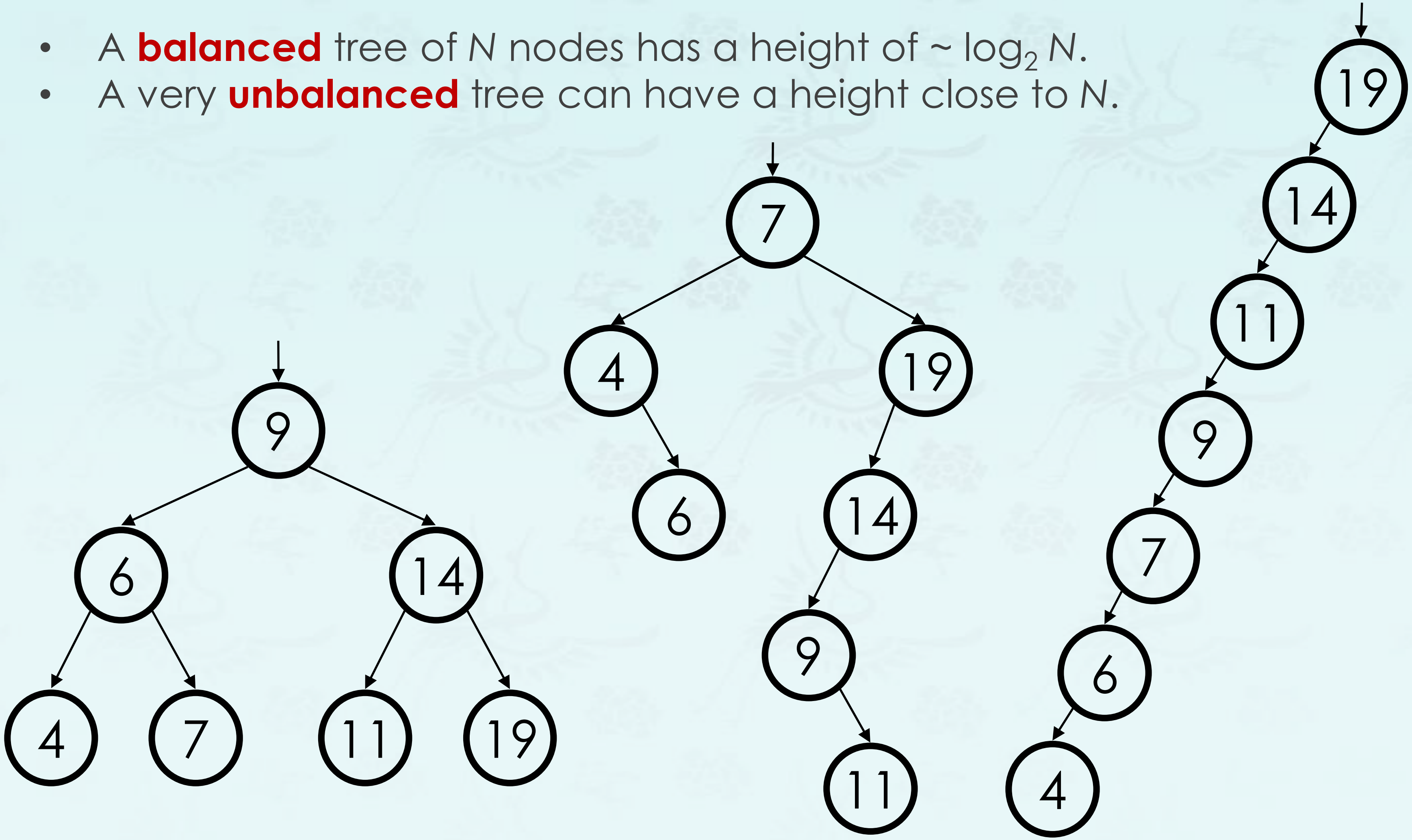
trim**: trim node with the key and return the new root.

```
tree trim(tree node, int key) {
  if (node == nullptr) return node;// base case
  if (key < node->key)
    node->left = trim(node->left, key);
  else if (key > node->key) {
    node->right = trim(node->right, key);
  else {
    if (node->left == nullptr) {
       // your code here - trim the right one, return node
    }
    else if (node->right == nullptr) {
       // your code here - trim the left one, return node
    }
    else {// two children case
      // get the successor: smallest in right subtree
      // copy the successor's content to this "node" node
      // node->right = trim the successor recursively, which has one or no child case
    }
  }
  return node;
}
```

http://visualgo.net/bst

Observations: What do you see in the following BSTs?

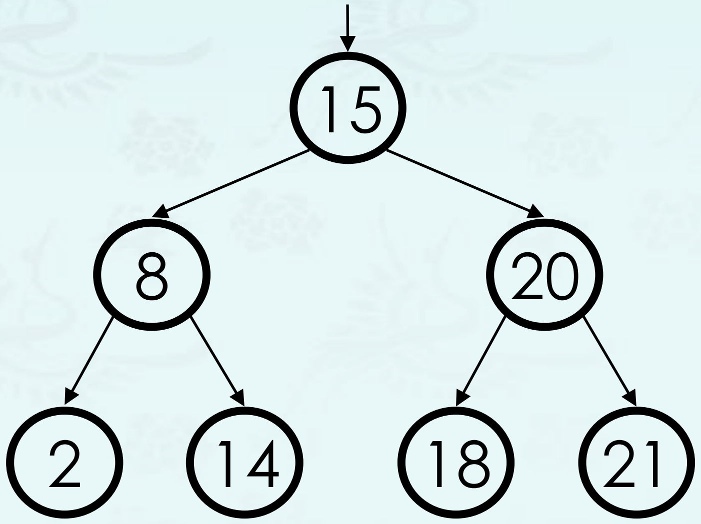
- A **balanced** tree of N nodes has a height of ~ $\log_2 N$.
- A very **unbalanced** tree can have a height close to N.



Observations: What do you see in the following BSTs?

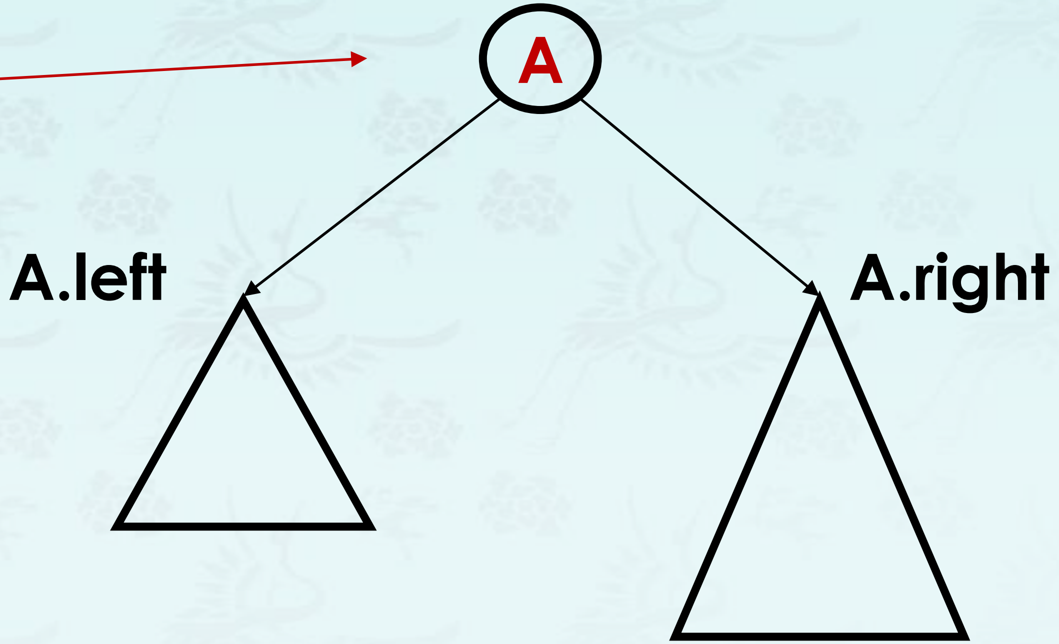
- *Observation:* The shallower the BST the better.
  - Average case height is O(log N)
  - Worst case height is O(N)
  - Simple cases such as adding (1, 2, 3, ..., N), or the opposite order, lead to the worst case scenario: height O(N).

- 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.

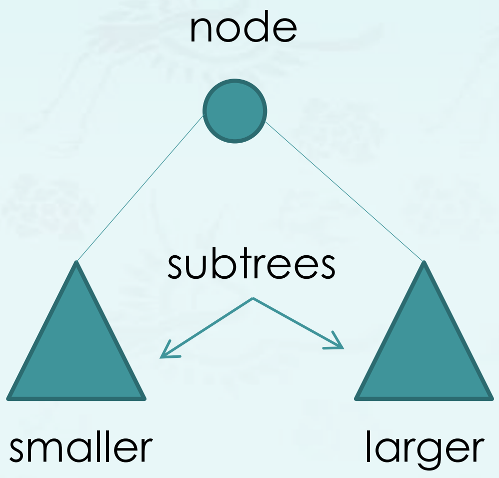
- ***Height*** *is* max number of nodes in path from root to any leaf.
  - height(nullptr) = 0
  - height(a leaf) = ?
  - height(**A**) = ?
- ***Hint***:
  - use recursive.
  - use max(a, b).



- ***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 $O(h)$.

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

Prof. Youngsup Kim, idebtor@gmail.com, Data Structures, CSEE Dept., Handong Global University

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

No, of course, it is wrong! Why?

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 $O(h)$.

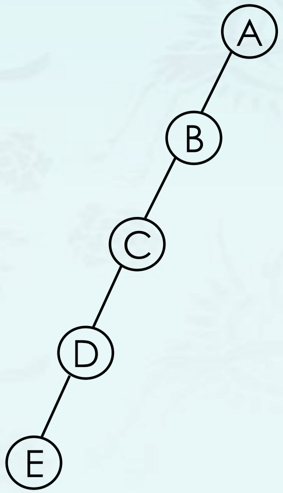
Since $h = \log n$ (where n is the number of elements), then it's good! – right?

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!



$\boldsymbol{n} = \boldsymbol{h}$

a skew binary search tree

Binary search trees

1. trim
   https://www.youtube.com/watch?v=gcULXE7ViZw

2. inorder
   https://www.youtube.com/watch?v=5cPbNCrdotA

3. binary search tree
   https://www.youtube.com/watch?v=pYT9F8_LFTM
   https://www.youtube.com/watch?v=COZK7NATh4k