Binary Search Tree (BST)

Searching Items in Lists

- Searching & removing an item from a list
  - Search if 14 is in the list.
  - Delete 3 from the list.

10	6	5	12	7	9	8	3	13

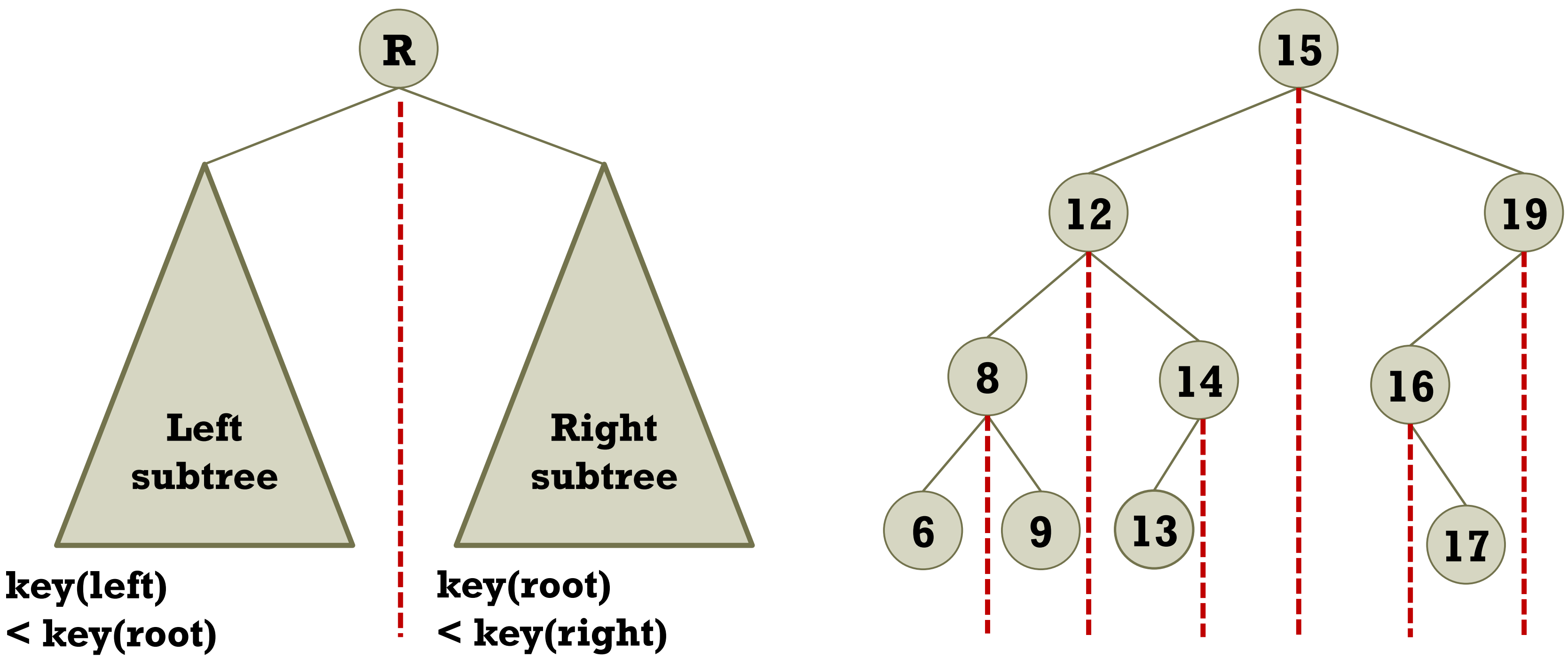
  - The time complexity of searching and deletion is $\boldsymbol{O(n)}$.
    - This is **too slow** if the number of items are huge.

- Q) How to improve time complexity of searching and deleting an element?

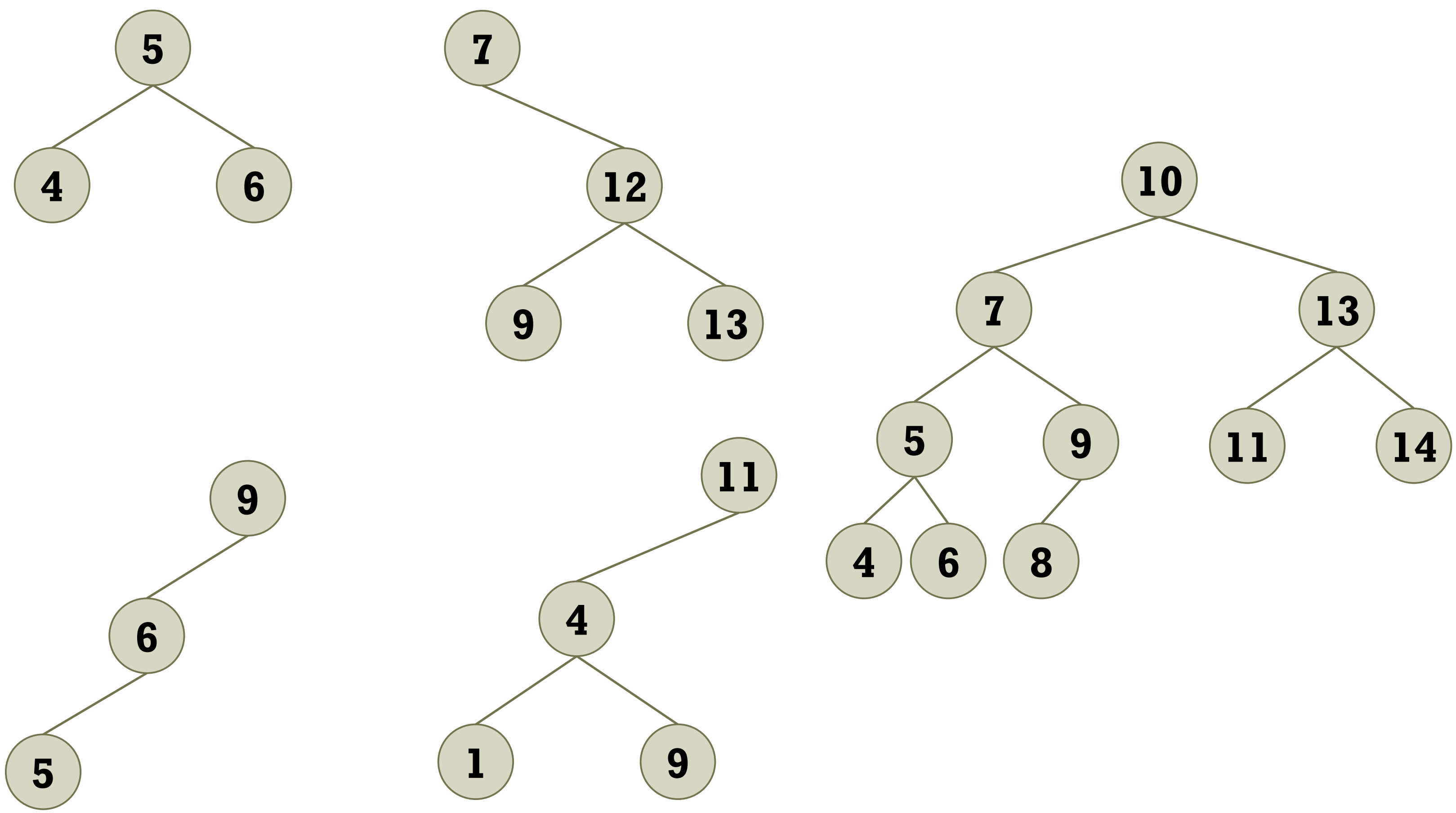
- A) Use a Binary Search Tree (BST).

What is Binary Search Tree (BST)?

- A binary tree that is empty or each node satisfies the following conditions.
  - Every element has a key, and there are **no duplicate keys**.
  - **key(left subtree) < key(root node) < key(right subtree)**

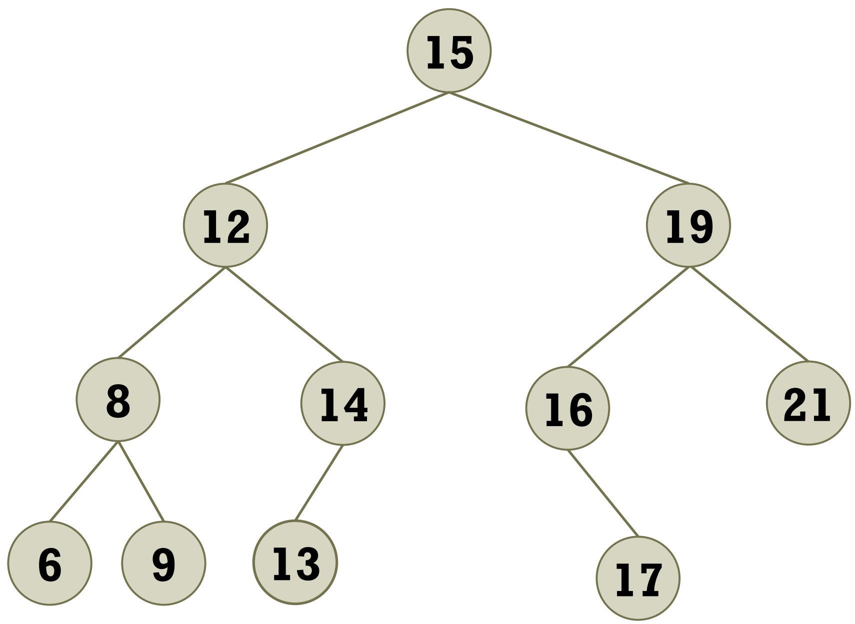


Examples of BST



Properties of BST

- Properties
  - The operations for searching, insertion, and deletion are bound by $\boldsymbol{O(h)}$, where h is the height is the BST.
  - The inorder traversal of BST **can generate a sorted list**.



Inorder traversal: **6 8 9 12 13 15 16 17 19 21**

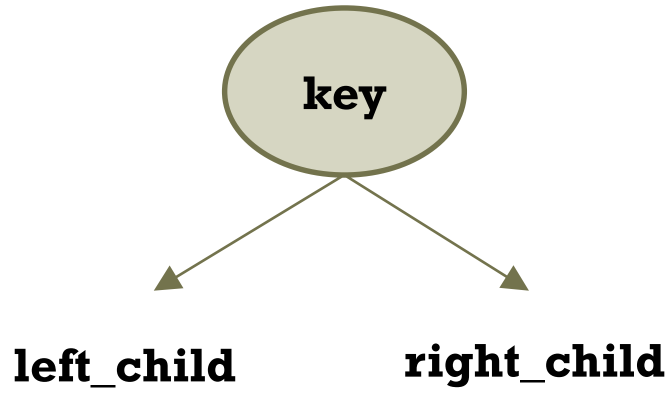
Implementation of BST

- Node representation in BST

left_child	key	right_child

```c
typedef int Key;

typedef struct _BSTNode
{
    Key key;
    struct _BSTNode * left_child;
    struct _BSTNode * right_child;
} BSTNode;
```



Implementation of BST

- Operations

```
// Create a new node.
BSTNode * CreateNode(Key key);
// Destroy a node.
void DestroyNode(BSTNode * node);

// Verify whether the tree is a binary search tree or not.
bool Verify(BSTNode* root);
// Search an item in BST.
BSTNode* Search(BSTNode* root, Key key);
// Insert an item to BST.
void Insert(BSTNode* root, Key key);
// Remove an item from BST.
void Remove(BSTNode* node, Key key);

// Traverse BST (the inorder travsal is a sorted list.)
void Traverse(BSTNode* root);
// Clear a tree.
void ClearTree(BSTNode* root);
```

Verifying BST

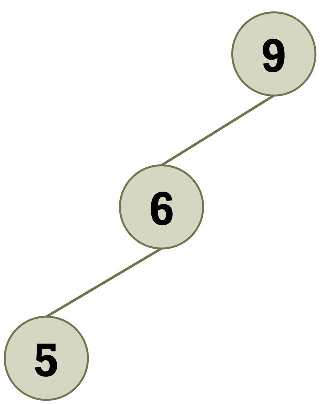
- Description

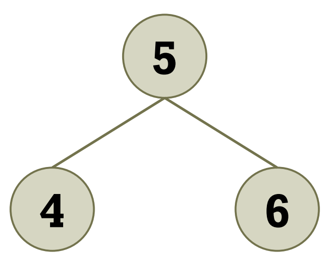
  1. Check the following conditions for every node.

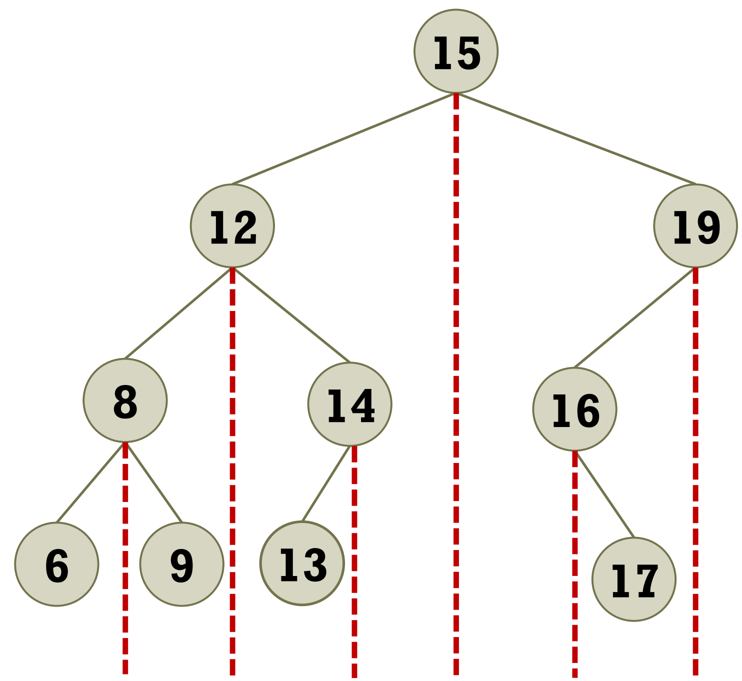
     1.1. The key of the current node should be greater than the keys of all nodes in its left subtree.

     1.2. The key of the current node should be less than the keys of all nodes in its right subtree.

- Are they binary search trees?







Verifying BST

- Algorithm
  - Check the minimum and maximum conditions for each node.

```
// Initialize the minimum and maximum as INT_MIN and INT_MAX
// for 32 bits, -2147483648  ~ +2147483647
bool Verify(BSTNode* root, int min, int max)
{
     if (root != NULL)
     {
         // Return false if this node violates the min/max constraints.
          if (root->key < min || root->key > max)
               return false;
          else
               // Check the subtrees with the min/max constraints.
               return Verify(root->left_child, min, root->key) &&
                       Verify(root->right_child, root->key, max);
     }
     else
          return true;  // an empty tree is BST.
}
```

Searching an Element in BST

- Description

    1. Begin by examining the root node.

        1.1. If the node is NULL, **the element does not exist**.

    2. Compare the key of the root with the element.

        2.1 If the element is equal to the key, the element is found.

        2.2 If the element is less than the key, **search a left subtree**.

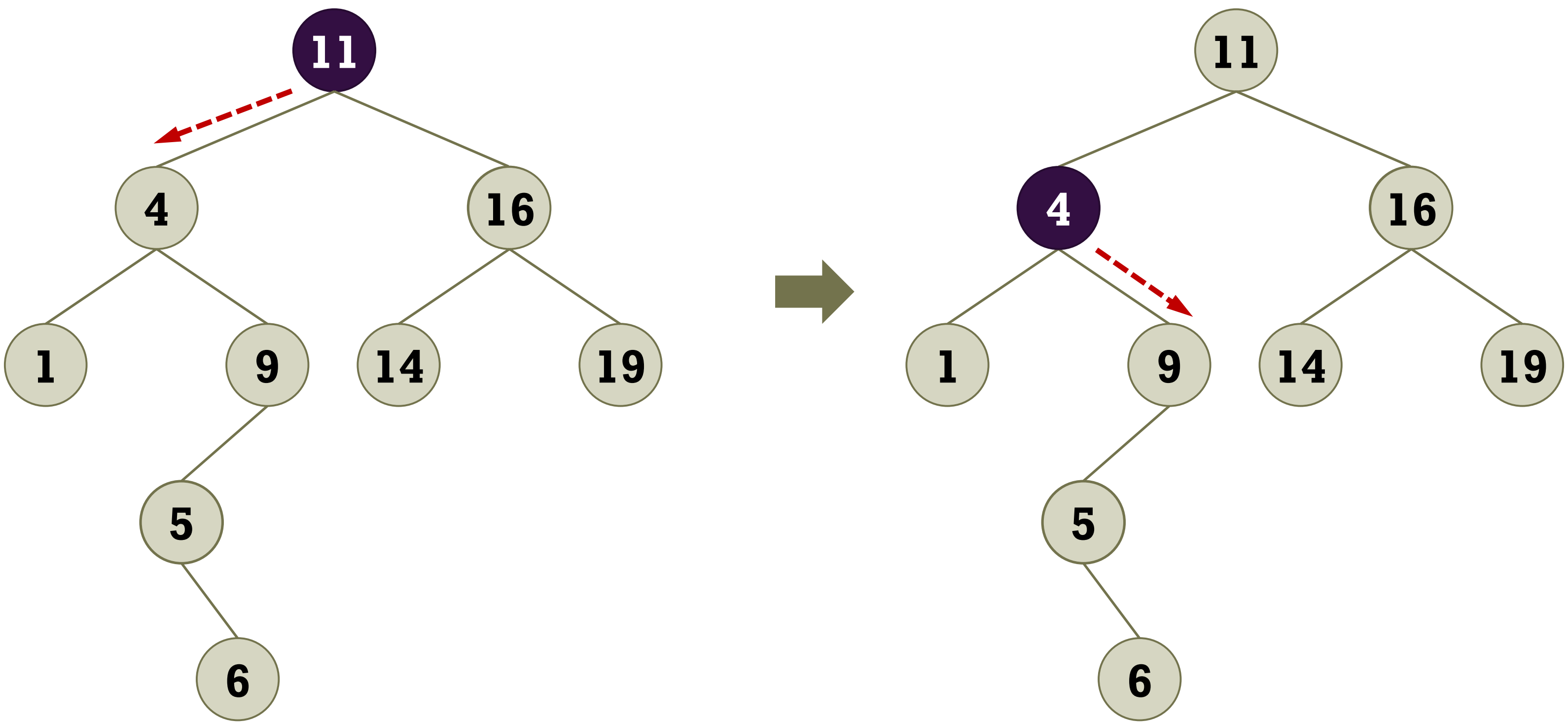
        2.3 If the element is greater than the key, **search a right subtree**.

    3. Repeat steps 1-2 until **the element is found** or **the root is NULL**.

- Animation: https://visualgo.net/en/bst

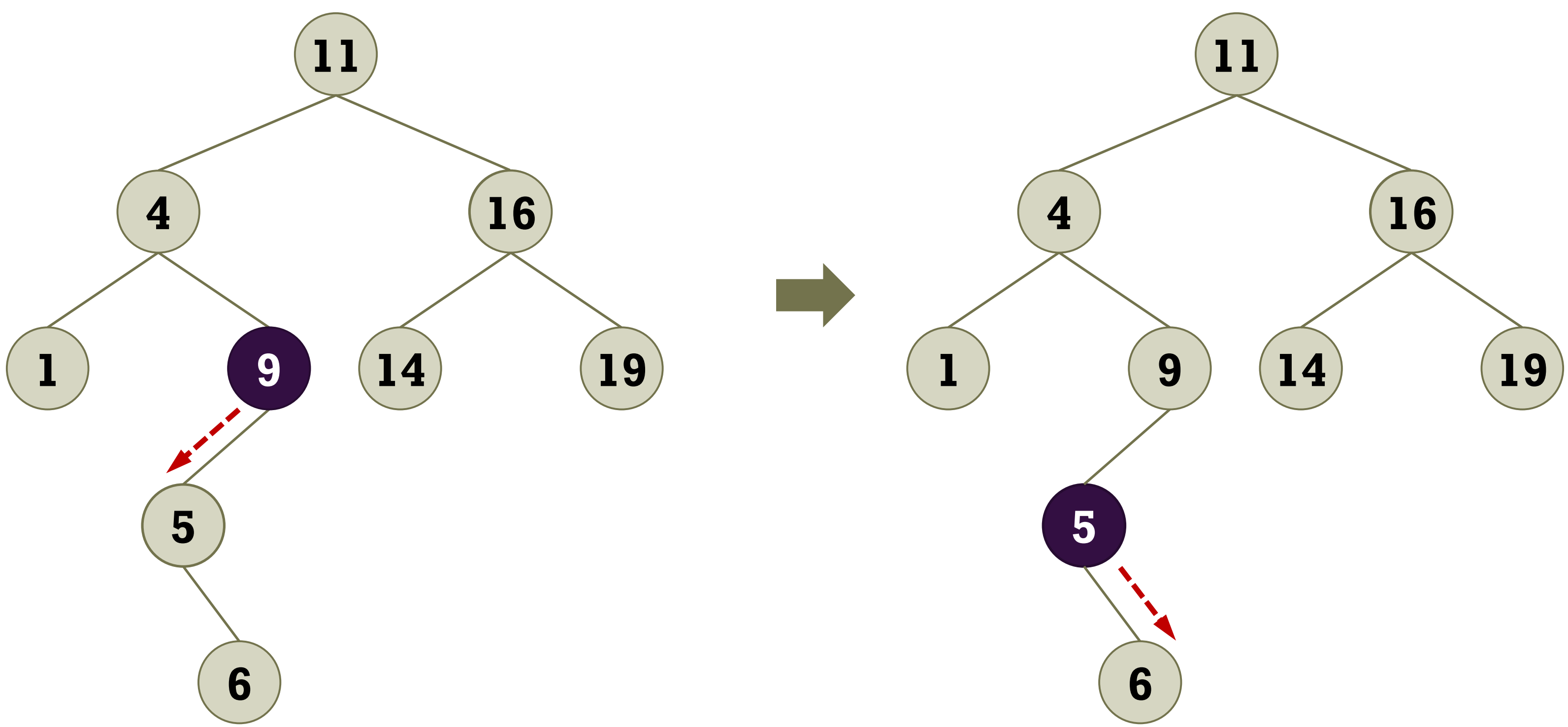
Searching an Element in BST

- Searching 6 in BST
  - Compare 6 with 11. Because 6 < 11, traverse a left subtree.
  - Compare 6 with 4. Because 6 > 4, traverse a right subtree.



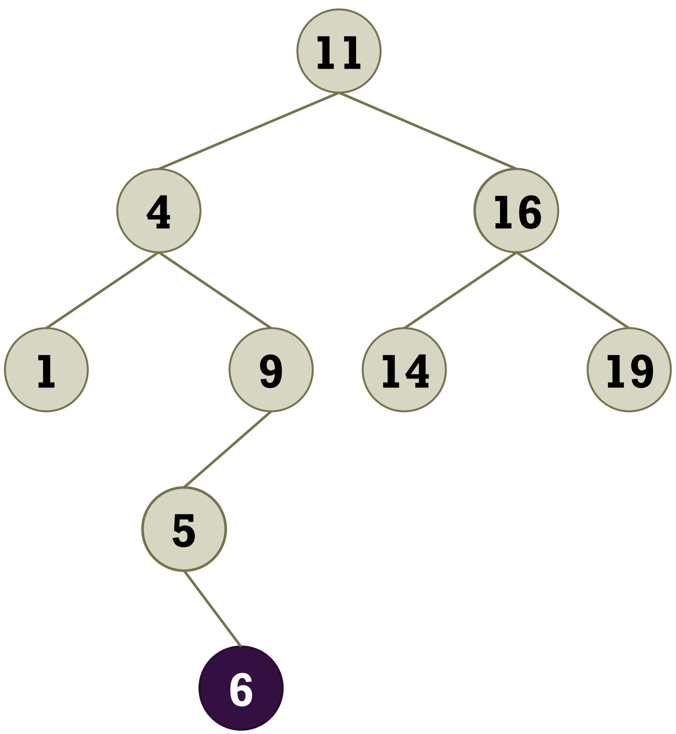
Searching an Element in BST

- Searching 6 in BST
  - Compare 6 with 9. Because 6 < 9, traverse a left subtree.
  - Compare 6 with 5. Because 6 > 5, traverse a right subtree.

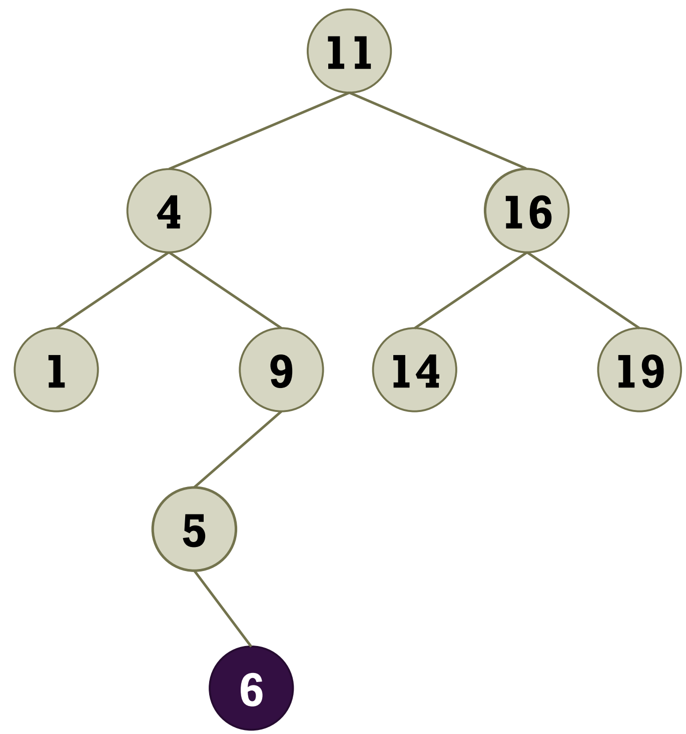


Searching an Element in BST

- Searching 6 in BST
  - Compare 6 with 6. Finally, 6 is found.
  - If searching 7, traverse a right subtree. Because the node is NULL, 7 does not exist in BST.



The element 6 is found!

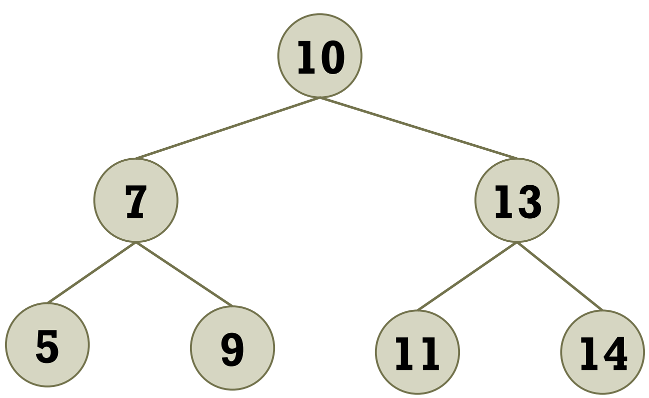


The element 7 does not exist.

Searching an Element in BST

- Algorithm: Recursive version

```
BSTNode* Search(BSTNode* root, Key key)
{
     if (root == NULL || root->key == key) return root;
     else if (root->key > key)
          return Search(root->left_child, key);
     else
          return Search(root->right_child, key);
}
```

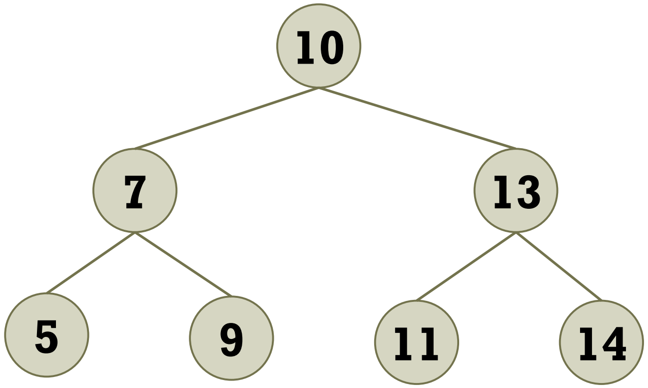


How to search 9?

Searching an Element in BST

- Algorithm: Iterative version

```
BSTNode* Search(BSTNode* root, Key key)
{
	BSTNode* cur = root;
	while (cur != NULL) {
		if (cur->key == key) break;
		else if (cur->key > key)
			cur = cur->left_child;
		else
			cur = cur->right_child;
	}
	return cur;
}
```



How to search 12?

Inserting an Element in BST

- Description
  1. Begin by examining the root node.
     1.1. If the node is NULL, **the element is inserted at this position**.

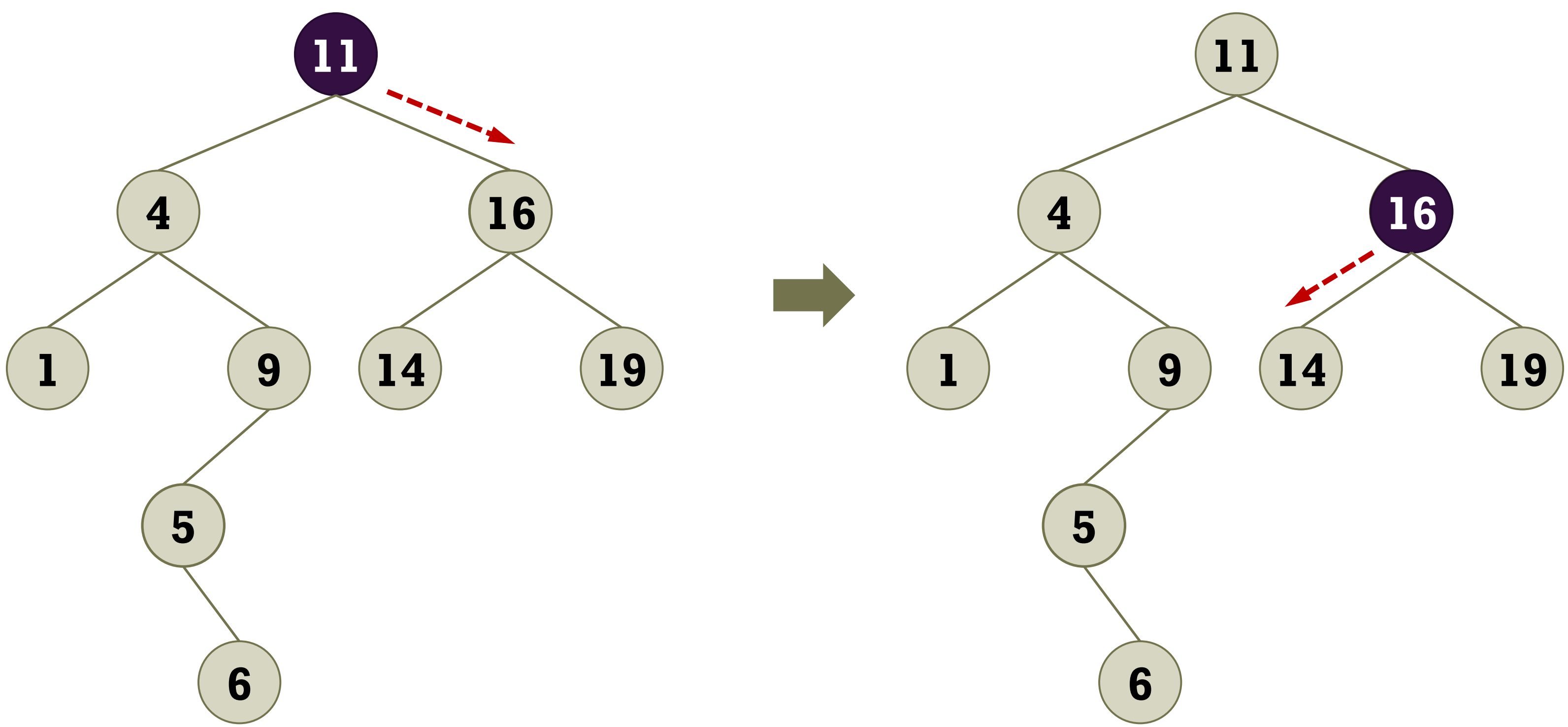
  2. Compare the key of the root with the element.
     2.1 If the element is equal to the key, **return an error**.
     2.2 If the element is less than the key, **search a left subtree**.
     2.3 If the element is greater than the key, **search a right subtree**.

  3. Repeat steps 1-2 until the element is found or the root is NULL.

- Similar to searching an element in the BST, it finds a position to be inserted.

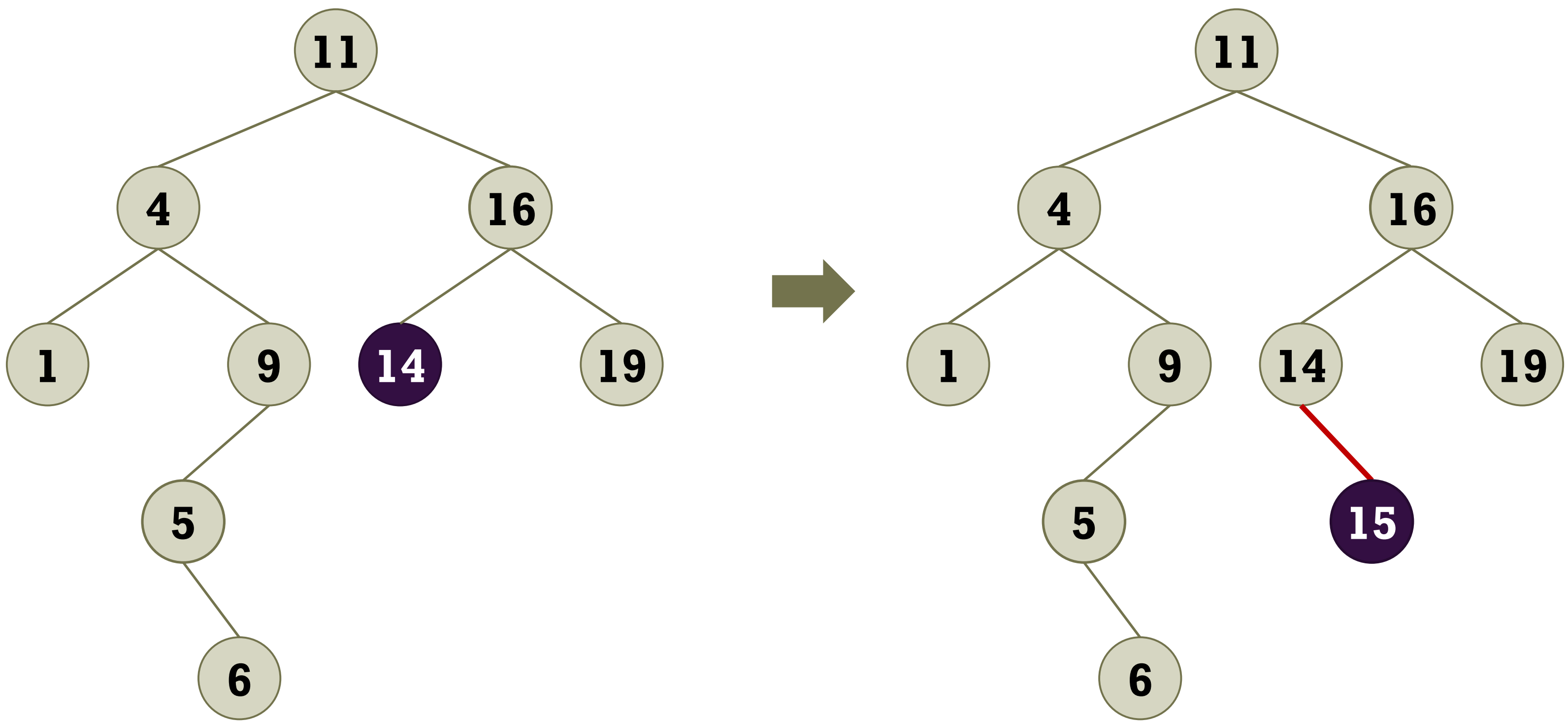
Inserting an Element in BST

- Inserting 15 to BST
  - Compare 15 with 11. Because 15 > 10, traverse a right subtree.
  - Compare 15 with 16. Because 15 < 16, traverse a left subtree.



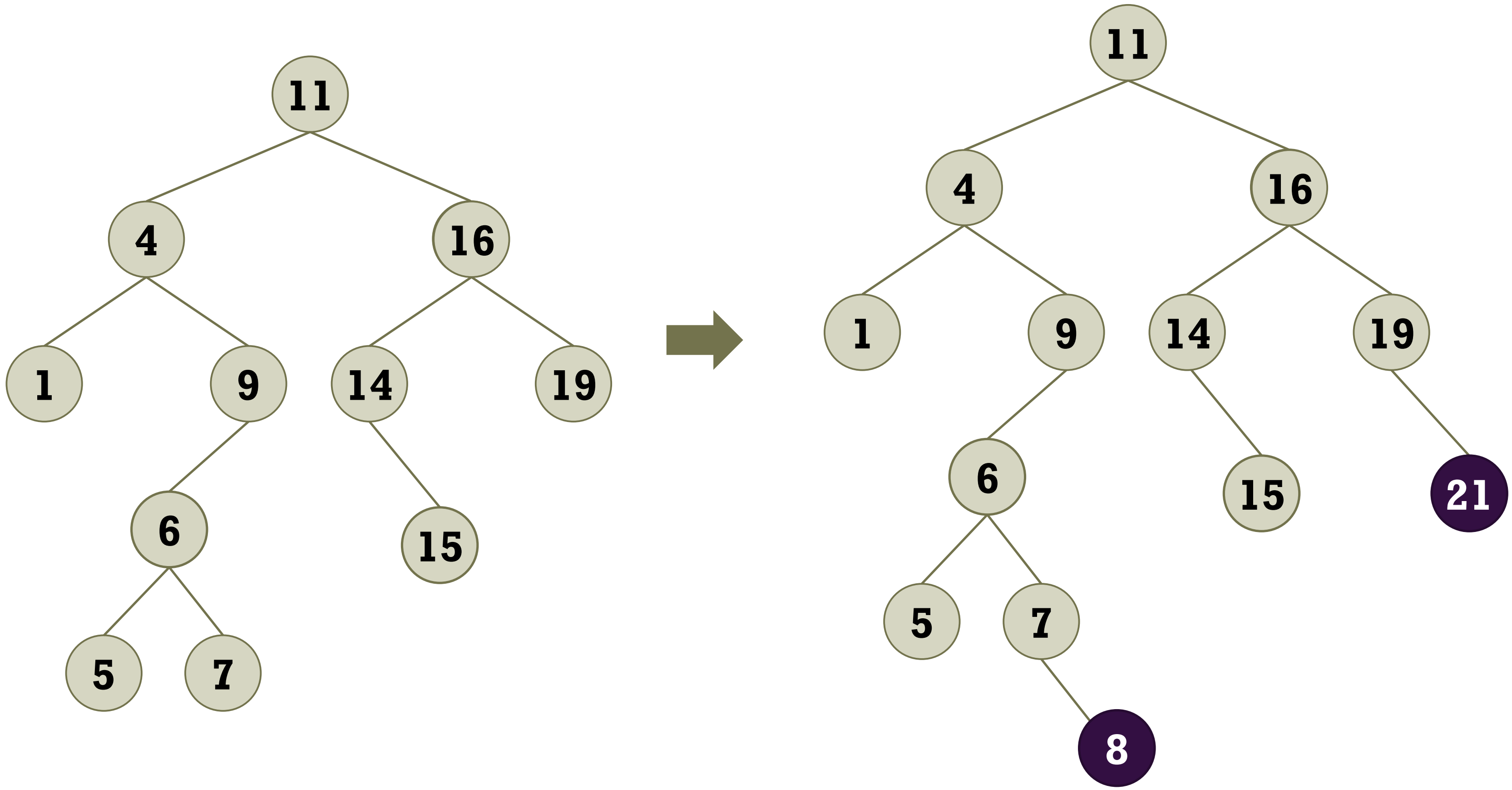
Inserting an Element in BST

- Inserting 15 to BST
  - Compare 15 with 14. Because 15 > 14, traverse a right subtree.
  - Because the node is NULL, 15 is inserted at this position.



Inserting an Element in BST

- Inserting 21 and 8 to BST in sequence



Inserting an Element in BST

- Algorithm: Recursive version

```
void Insert(BSTNode* root, Key key)
{
     if (root->key == key) exit(1);
     else if (root->key > key) {
          // Insert a new node for a left child.
          if (root->left_child == NULL)
               CreateLeftSubtree(root, key);
          else
               Insert(root->left_child, key);
     }
     else {
          // Insert a new node for a right child.
          if (root->right_child == NULL)
               CreateRightSubtree(root, key);
          else
               Insert(root->right_child, key);
     }
}
```

- Animation: https://visualgo.net/en/bst

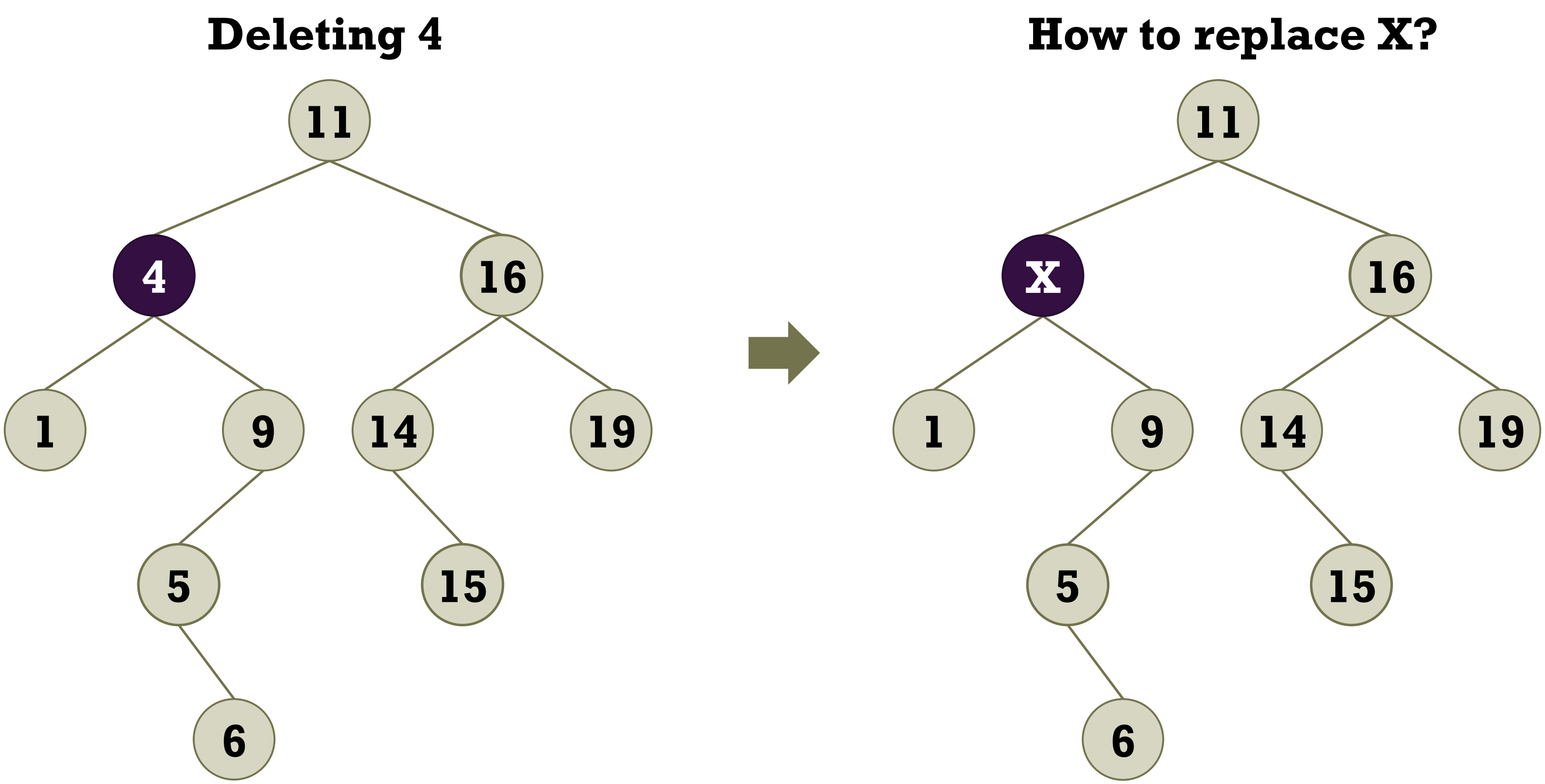
Inserting an Element in BST

- Algorithm: Iterative version

```
void Insert(BSTNode* root, Key key)
{
     BSTNode* cur = root;
     while (cur != NULL) {
          if (cur->key == key) exit(1);
          else if (cur->key > key) {
               // Insert a new node for a left child.
               if (cur->left_child == NULL) {
                    CreateLeftSubtree(cur, key);
                    break;
               } else
                    cur = cur->left_child;
          }
          else {
               // Insert a new node for a right child.
               if (cur->right_child == NULL) {
                    CreateRightSubtree(cur, key);
                    break;
               } else
                    cur = cur->right_child;
          }
     }
}
```

Removing an Element in BST

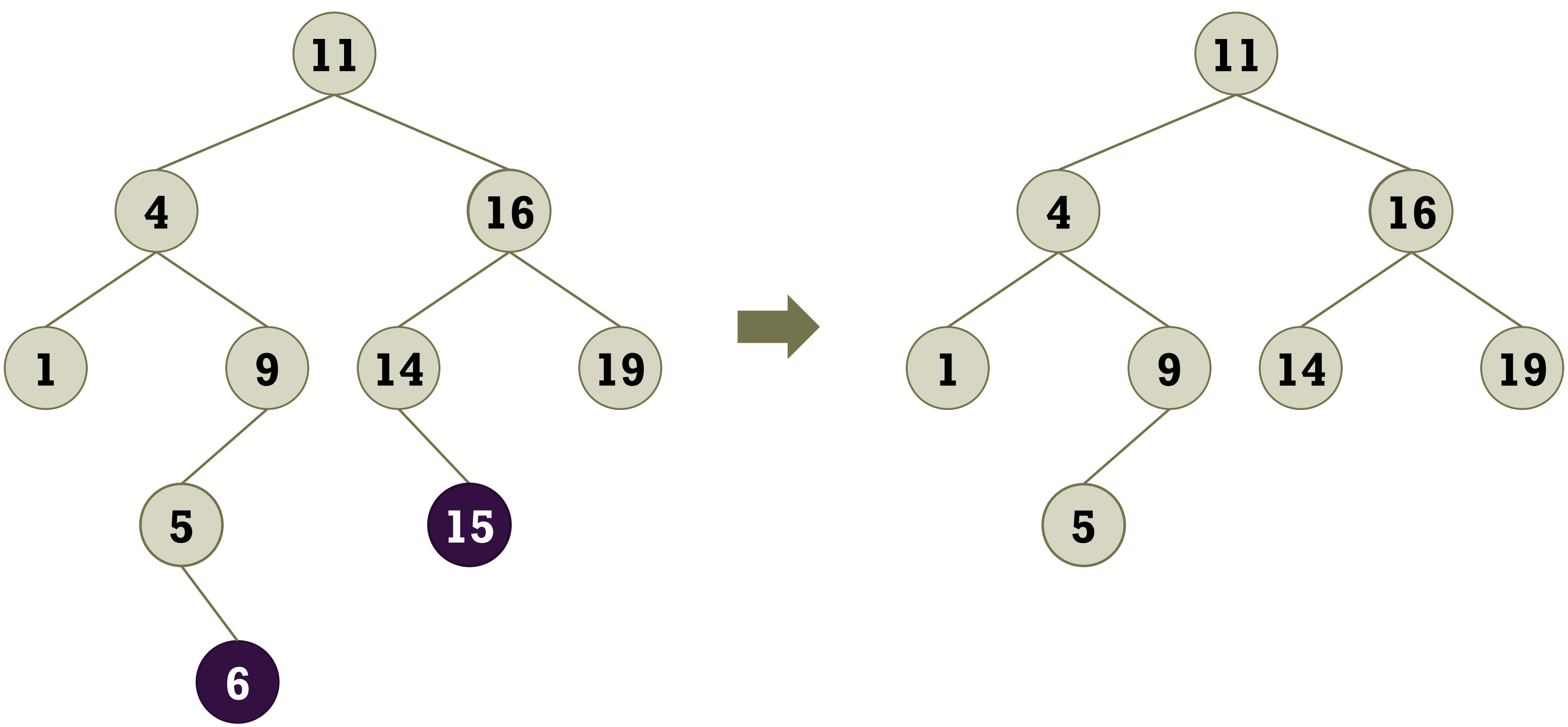
- When removing a node from BST, it is important to maintain the inorder sequence of the nodes.
  - The heights of the subtree need to be changed by at most one.



Removing an Element in BST

- Case 1: Deleting a node with no children
  - Simply remove the node from the tree.

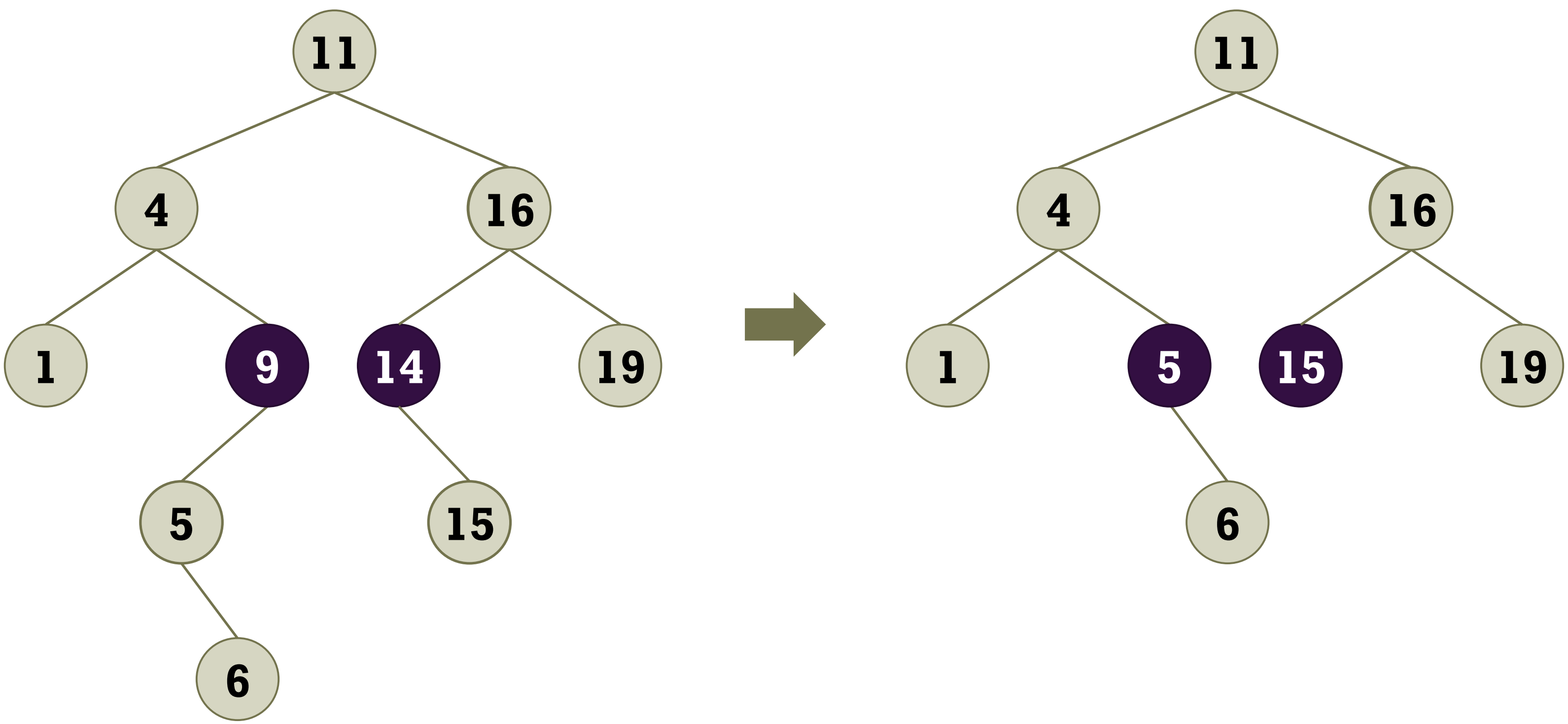
- Deleting 6 and 15 from BST



Removing an Element in BST

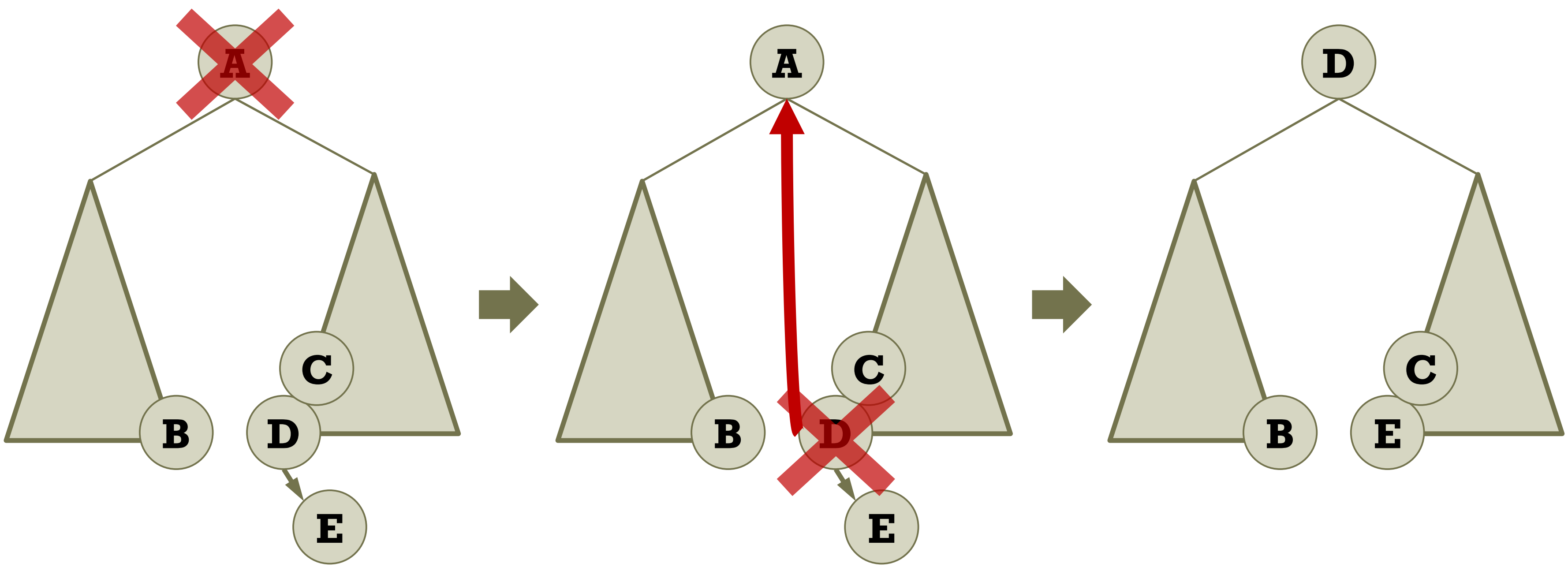
- Case 2: Deleting a node with one child
  - Remove the node and replace it with its child node.

- Deleting 9 and 14 from BST



Removing an Element in BST

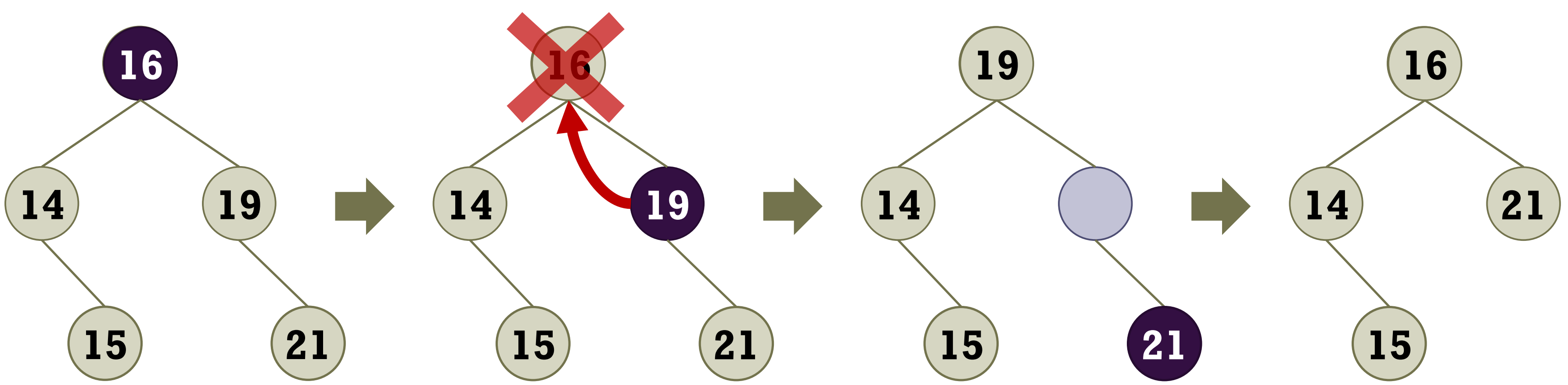
- Case 3: Deleting a node with two children
  - Choose either its **inorder predecessor node** or **successor node** as a replacement node.
    - Inorder predecessor: The largest in the left subtree
    - Inorder successor: The smallest in the right subtree



Removing an Element in BST

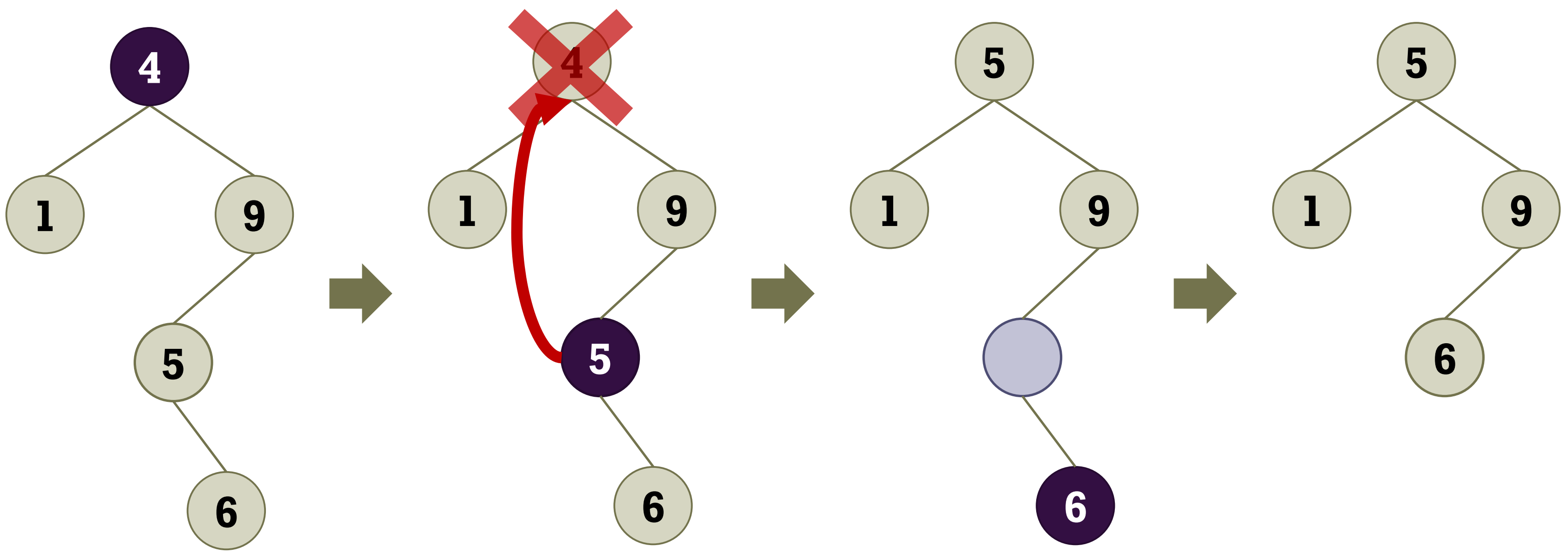
- Case 3: Deleting a node with two children
  - Choose its **inorder successor node** as a replacement node.
    - When a node is removed, the successor is the left-most node of its right child node.

- Deleting 16 from BST
  - The replacement node is updated by **the right child node** of the successor node.



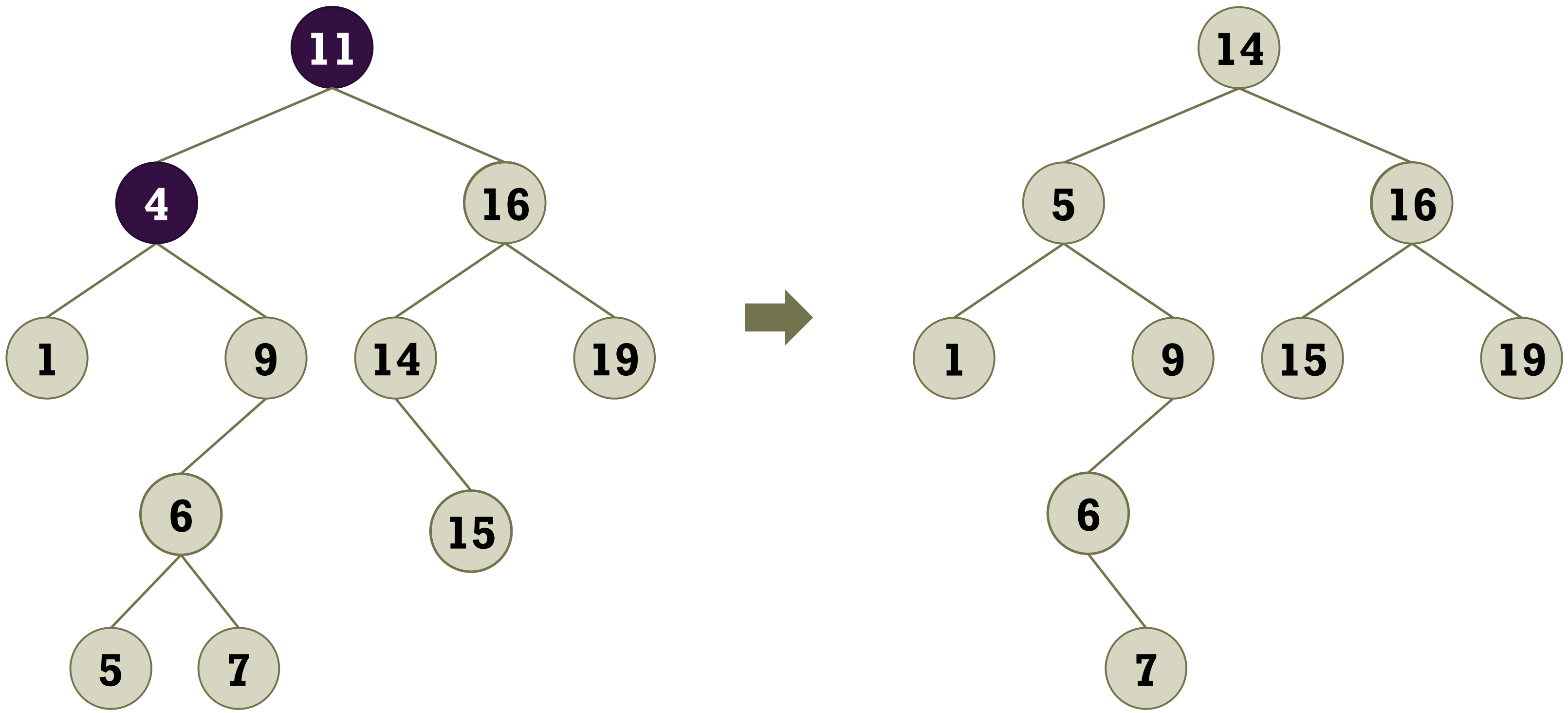
Removing an Element in BST

- Deleting 4 from BST
  - The replacement node is updated by **the left child node** of the successor node.



Removing an Element in BST

- Deleting 4 and 11 from BST in sequence



Removing an Element in BST

- Algorithm
  - Handle three possible cases for removing an element in BST.

```
void Remove(BSTNode* root, Key key)
{
     BSTNode* cur = root, * parent = NULL;

     // Find the current node and its parent node.
     while (cur != NULL && cur->key != key) {
          parent = cur;  // Update the parent node.
          if (cur->key > key)
               cur = cur->left_child;
          else
               cur = cur->right_child;
     }
```

- Animation: https://visualgo.net/en/bst

Removing an Element in BST

- Case 1: Deleting a node with no children
  - Simply remove the node from the tree.

```
if (cur == NULL) exit(1);

if (cur->left_child == NULL && cur->right_child == NULL) {
     if (parent != NULL) {
          // Remove the current node depending on its position.
          if (parent->left_child == cur)
               parent->left_child = NULL;
          else
               parent->right_child = NULL;
     }
     else
          cur = NULL;    // The current node is the root.
}
```

Removing an Element in BST

- Case 2: Deleting a node with one child
  - Remove the node and replace it with its child node.

```
    else if (cur->left_child == NULL || cur->right_child == NULL) {
        BSTNode* child;
        // Replace a node with its child node.
        if (cur->left_child != NULL)
            child = cur->left_child;
        else
            child = cur->right_child;

        // Replace the child node of its parent node.
        if (parent != NULL) {
            if (parent->left_child == cur)
                parent->left_child = child;
            else
                parent->right_child = child;
        }
    }
```

Removing an Element in BST

- Case 3: Deleting a node with two children
  - Choose the successor node as a replacement node.

```
     else {
          BSTNode* succ_parent = cur, *succ = cur->right_child;
          // Find the successor (left-most node of the current node.)
          while (succ->left_child != NULL) {
               succ_parent = succ;
               succ = succ->left_child;
          }

          // If the successor has a child, update its the child node.
          if (succ_parent->right_child == succ)
               succ_parent->right_child = succ->right_child;
          else
               succ_parent->left_child = succ->right_child;

          cur->key = succ->key;
          cur = succ;   // Remove the successor.
     }
     DestroyNode(cur);
}
```

Summary of Binary Search Tree

- Time complexity of BST
  - The operations for searching, insertion, and deletion are bound by $O(h)$, where h is the height of BST.

Algorithm	Average case	Worst case
Searching	$\boldsymbol{O(\log n)}$	$\boldsymbol{O(n)}$
Insertion	$\boldsymbol{O(\log n)}$	$\boldsymbol{O(n)}$
Deletion	$\boldsymbol{O(\log n)}$	$\boldsymbol{O(n)}$

  - where n is the number of nodes in BST.

- How to maintain a balanced tree?