CS14: INTRO TO DATA STRUCTURES AND ALGORITHMS

Ryan Rusich
rusichr@cs.ucr.edu
Department of Computer Science and Engineering
UC Riverside

Memory Allocation

C++ allocates space for variables declared from one of several memory regions.

1. **Static data** - memory reserved for variables that persist throughout the lifetime of the program, e.g. constants.

2. **Heap** - aka *free store* is a pool of memory that can be dynamically allocated at runtime.

3. **Stack data** - C++ allocates a new block of memory called a stack frame for method called to hold method's local variables.

   In classical architectures, the stack and heap grow toward each other to maximize the available space.

Dynamic Memory Allocation

C++ uses the **`new`** operator to allocate memory on the heap.

- To allocate space on the heap for a single value use **`new`** followed type, *e.g.* **`int`**.

```
int *ip = new int;
```

- To allocate space on the heap for an array of values use **`new`** followed array *type* and *size* in [ ].

```
new int[size]
```

- To dynamically allocate an **`int`** array of size **`10000`**

```
int *array = new int[10000];
```

- The **`delete`** operator frees previously allocated heap memory. For arrays. Include empty [ ].

```
delete[] array;
```

Destructors

- C++ class definitions often include a **destructor** that specifies how to **free** the memory used to store an instance of that class.
- A **destructor** function prototype has no return type, must take no arguments and uses the class name preceded by a tilde (~)

  e.g. `List::~List();`
- C++ automatically calls the destructor whenever a variable of a particular class is released.
  - **Stack objects** - when a functions returns the stack will automatically reclaim those objects declared as local variables in that function.
  - **Heap objects** - space for objects allocated with `new` must be explicitly deallocated (*free)* by calling `delete`.
- Calling `delete` automatically invokes the destructor.

Memory related problems

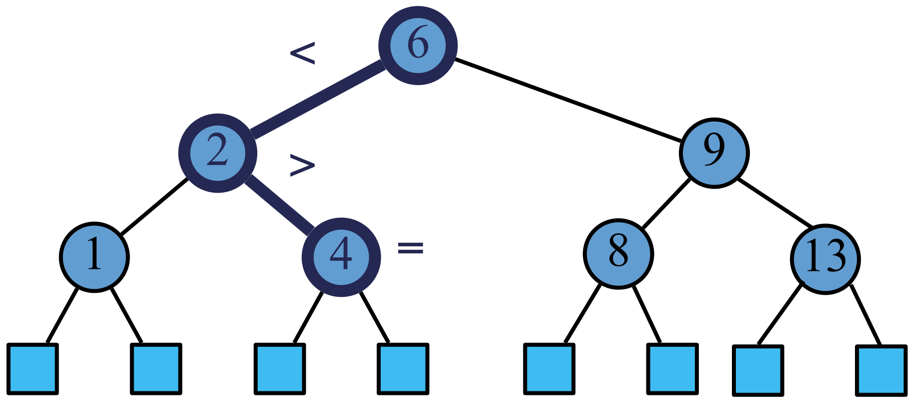
- Use of uninitialized memory
- Reading/writing memory after it has been freed (delete)
- Reading/writing off the end of malloc’d (new) blocks
- Memory leaks
- Mismatched use of malloc/new/new[] vs free/delete/delete[]
- Doubly freed memory

C++ “valid” pointer

A **valid** pointer is one of that:

- is a null pointer (value is nullptr, NULL)
- points to an object
- points to a[1] where &a[0] points to an object, i.e., “one position past an object”

Binary Search Trees BST

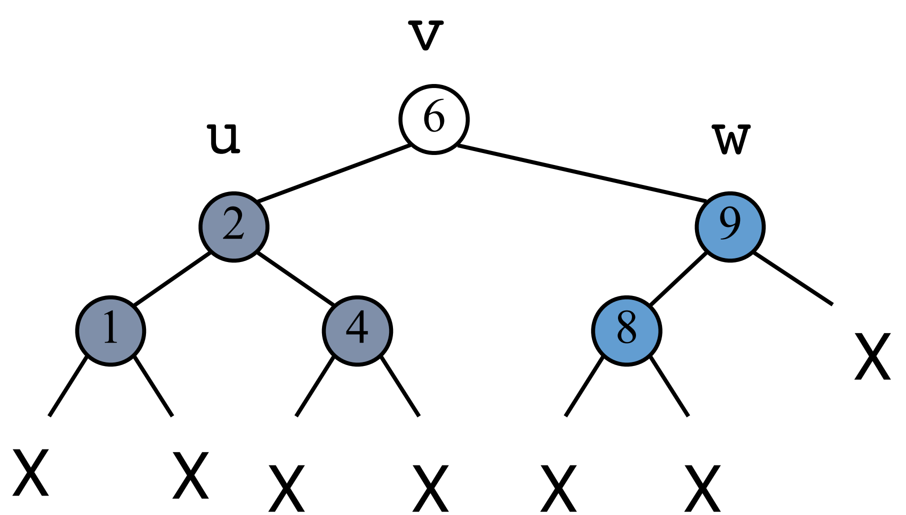


Binary Search Tree

- A binary search tree is a binary tree storing keys (or key-element pairs) satisfying the following property.

- Let **`u`**, **`v`**, and **`w`** be three nodes such that **`u`** is in the left-subtree of *v* and **`w`** is in the right-subtree of **`v`**.

`key(u) < key(v) < key(w)`



BST Operations

- isEmpty - return true if empty, false if not
- search (private) - return pointer to node in which key is found, otherwise return NULL
- search (public) - return true if key is found, otherwise return false
- findMin - return smallest node value
- findMax - return largest node value

BST Operations

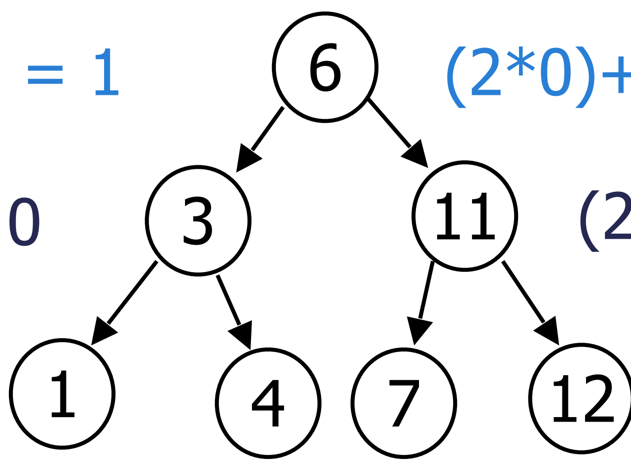
- insert - insert a new node into the tree maintaining BST property. All inserts are done at a leaf
- remove - remove a node from the tree maintaining BST property.
- display - print a tree in an ordered traversal

Array Implementation of a BST

- Binary trees (all nodes have branching factor of <= 2 can be implemented with an array
- BST is a binary tree
- Operations to find parent and children's array index rooted at zero.
- Given a node at index i
  - parent(i) = (i - 1)/2
    - If i == 0, then no parent since root
  - leftChild(i) = 2i+1
    - If 2i+1 <= N, otherwise no child
  - rightChild(i) = 2i+2
    - If 2i+2 <= N, otherwise no child

Array Implementation of a BST

0	1	2	3	4	5	6	7
6	3	11	1	4	7	12	



(2*0)+1 = 1

(2*0)+2 = 2

Find children

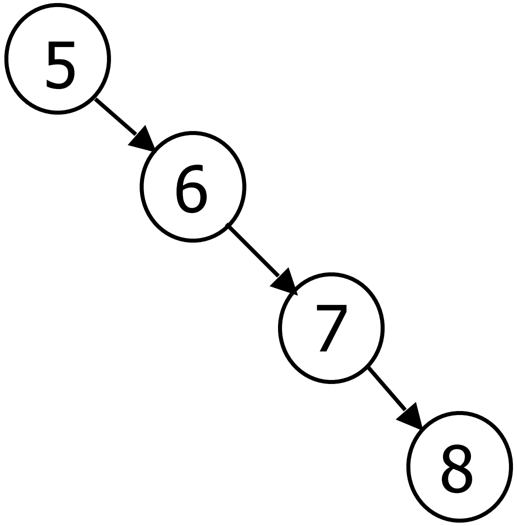
(1-1)/2 = 0

(2-1)/2 = 0

Find parents

Array Implementation of a BST

- In class exercise - show the array for the following tree



Linked Implementation of a BST

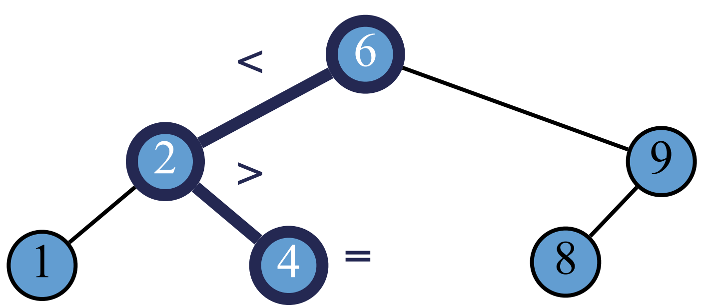
- Linked implementation
  - Similar to linked list – dynamic size can grow and shrink easily during runtime

```
class Node {
        itemtype item
        Node* left
        Node* right
}
```

```
class BST {
private:
        Node root
        // internal functions
public:
        // functions for
        operating on BST
}
```

Search

- To search for a key ***k***, we trace a downward path starting at the root
- The next node visited depends on the outcome of the comparison of ***k*** with the key of the current node
- If we reach a leaf, the key is not found and we return null
- Example: find(4)



Search

Recursive implementation of search (private)

```
Node search ( Node* nodePtr, itemtype key )
        if (nodePtr == NULL)
                return NULL
        else if ( nodePtr->item == key )
                return nodePtr
        else if ( nodePtr->item > key )
                return search(nodePtr->left, key)
        else
                return search(nodePtr->right, key)
```

Inorder Traversal

Recursive implementation of inorder traversal

```
void inorder(Node* nodePtr)
    if ( nodePtr )
        inorder (nodePtr->left)
        print node
        inorder (nodePtr->right)
```

Preorder Traversal

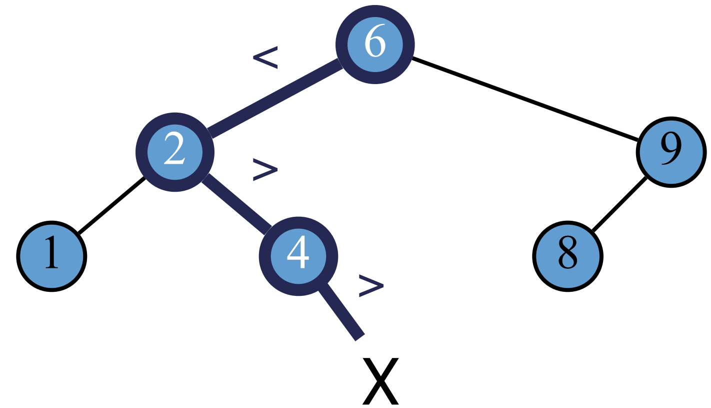
```
void preorder(Node* nodePtr)
    if ( nodePtr )
        print node
        preorder (nodePtr->left)
        preorder (nodePtr->right)
```

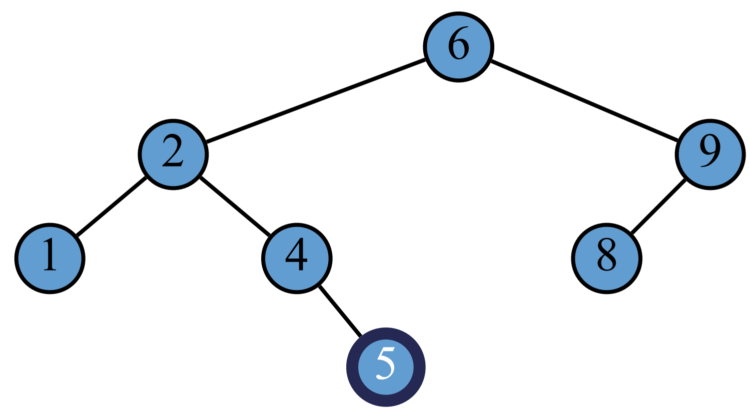
Postorder Traversal

```
void postorder(Node* nodePtr)
    if ( nodePtr )
        postorder (nodePtr->left)
        postorder (nodePtr->right)
        print node
```

Insertion

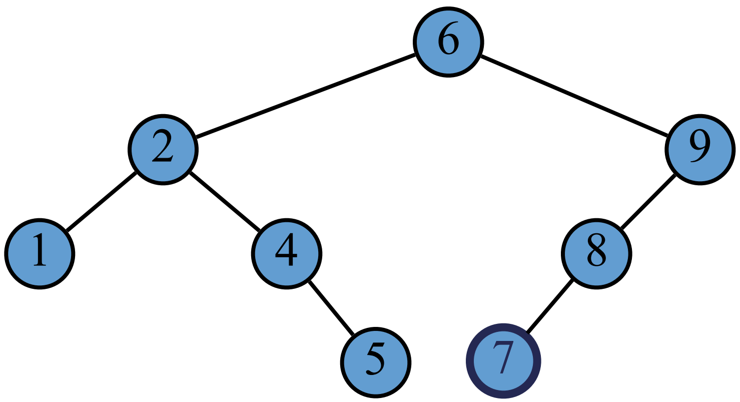
- To perform operation insertItem(k, o), we search for the position k would be in if it were in the tree
- All insertions create a new leaf node
- Example: insert 5



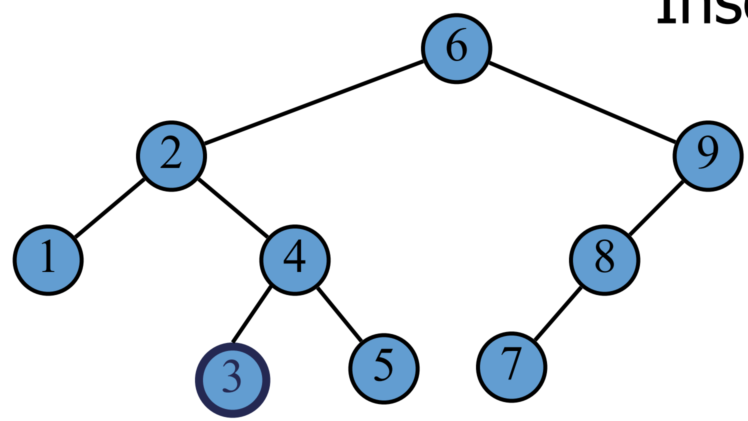


Insertion

Insert 7



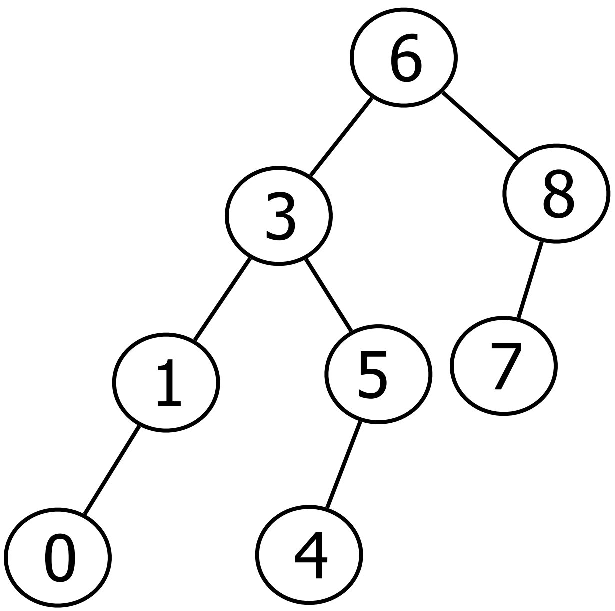
Insert 3



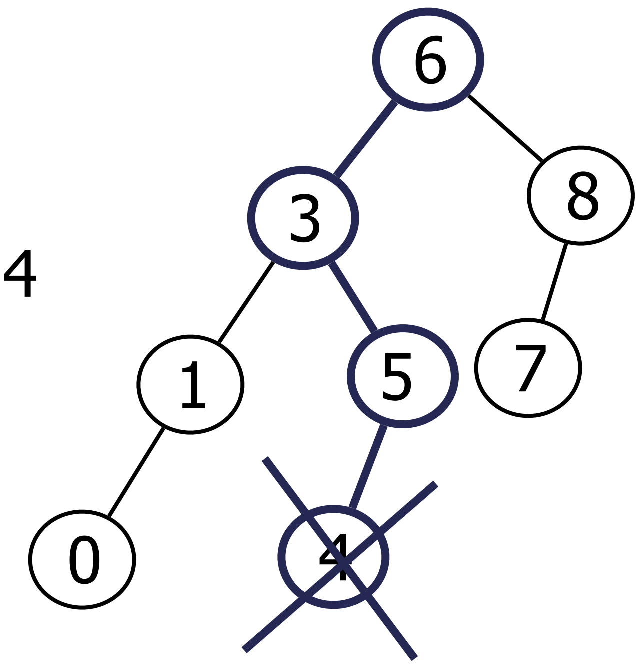
Deletion

- Traverse tree and search for node to remove
  - Five possible situations
    - Item not found
    - Removing a leaf
    - Removing a node with two children
    - Removing a node with one child - right only
    - Removing a node with one child - left only

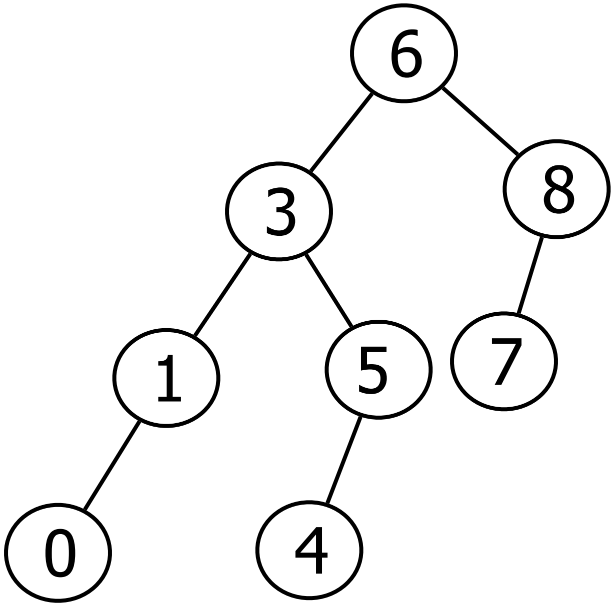
Deletion - Removing a leaf



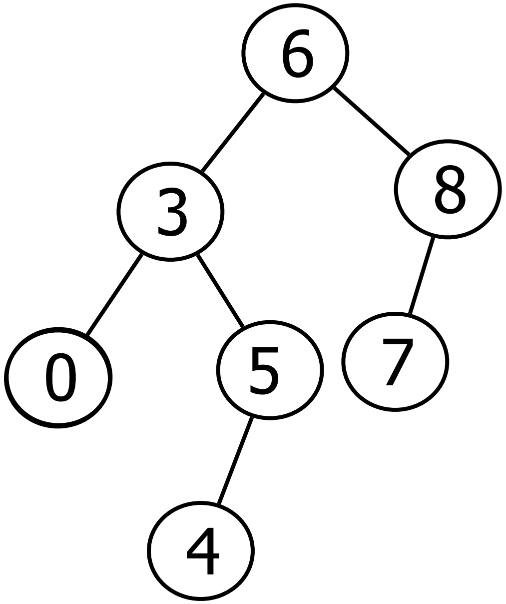
Remove 4



Deletion - Removing a leaf



Remove 1

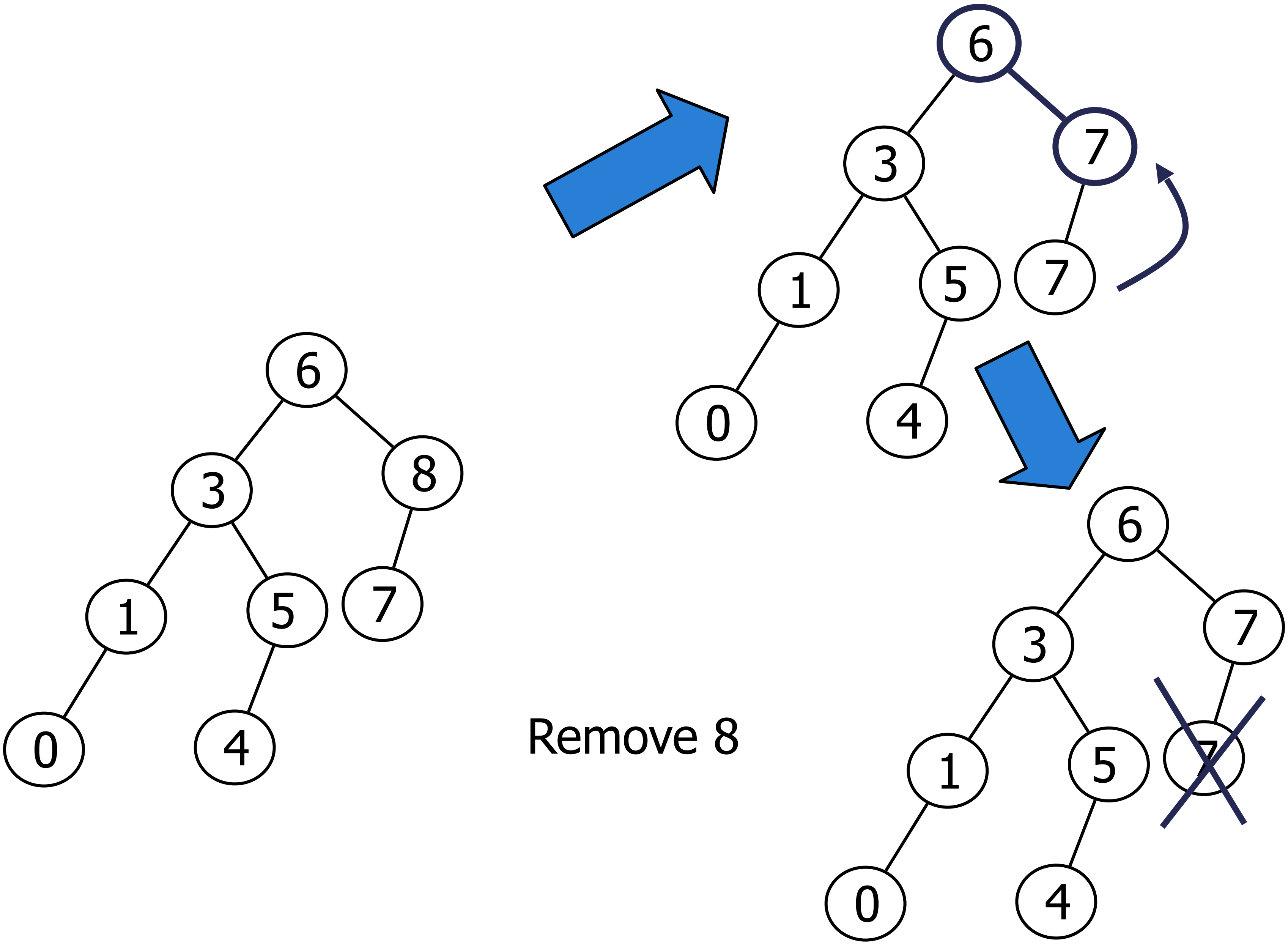


Deletion - Removing a node with children

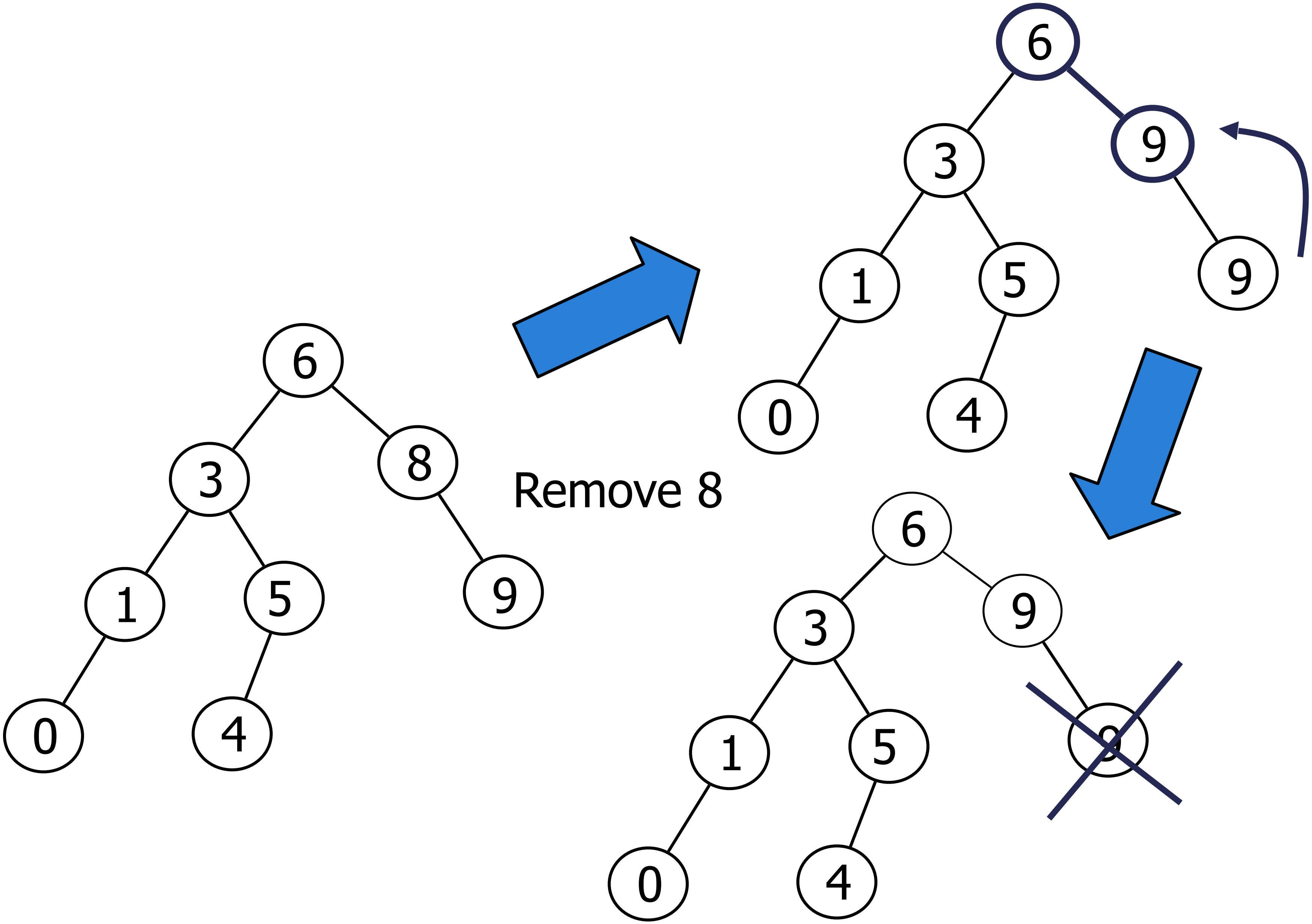
- Otherwise the node has children - find replacement node
  - If the left child exists
    - Replace node information with the ***largest*** value smaller than the value to remove
      - findMax(leftChild)
  - Else there is a right child
    - Replace node information with the ***smallest*** value larger than value to remove
      - findMin(rightChild)

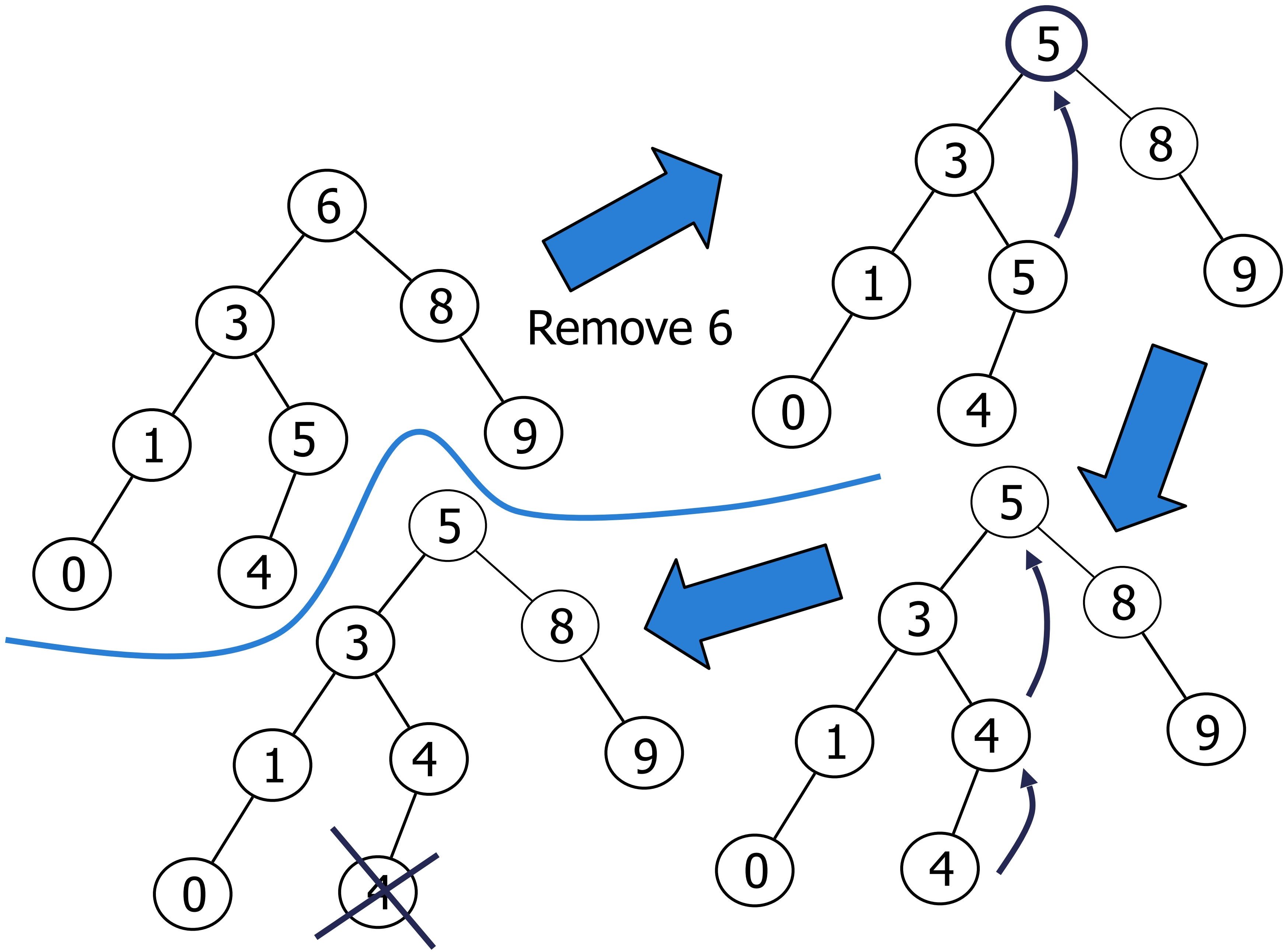
Deletion - Removing a node with children (continued)

- Splice out replacement node (call remove recursively)
- Just copy in info of replacement node over the value to remove (overload = if necessary)
- Delete replacement node if leaf



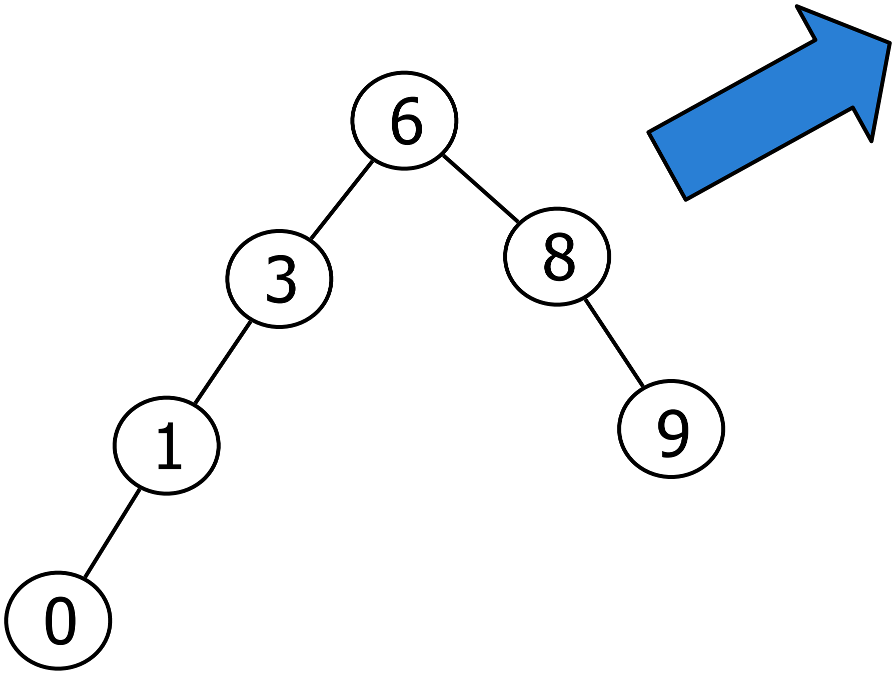
Remove 8


6
3
8
1
5
9
0
4
Remove 8
6
3
9
1
5
9
0
4
6
3
9
1
5
9
0
4

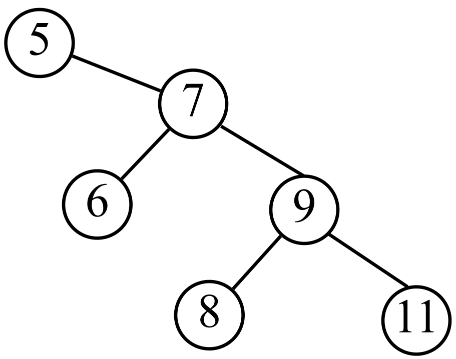

Remove 6

In class exercise

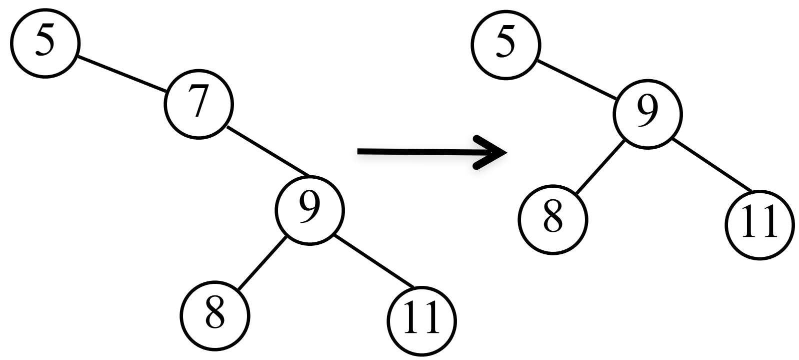
Remove 6



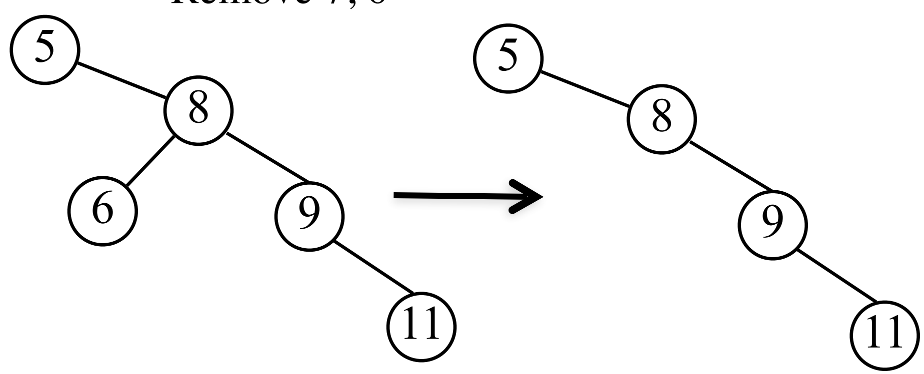
Removal Operation – Commutative?



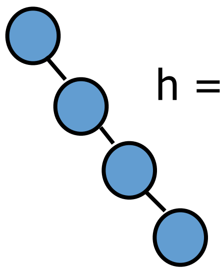
Remove 6, 7

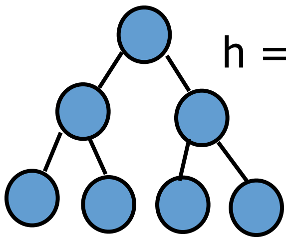


Remove 7, 6



Analysis of BST Operations

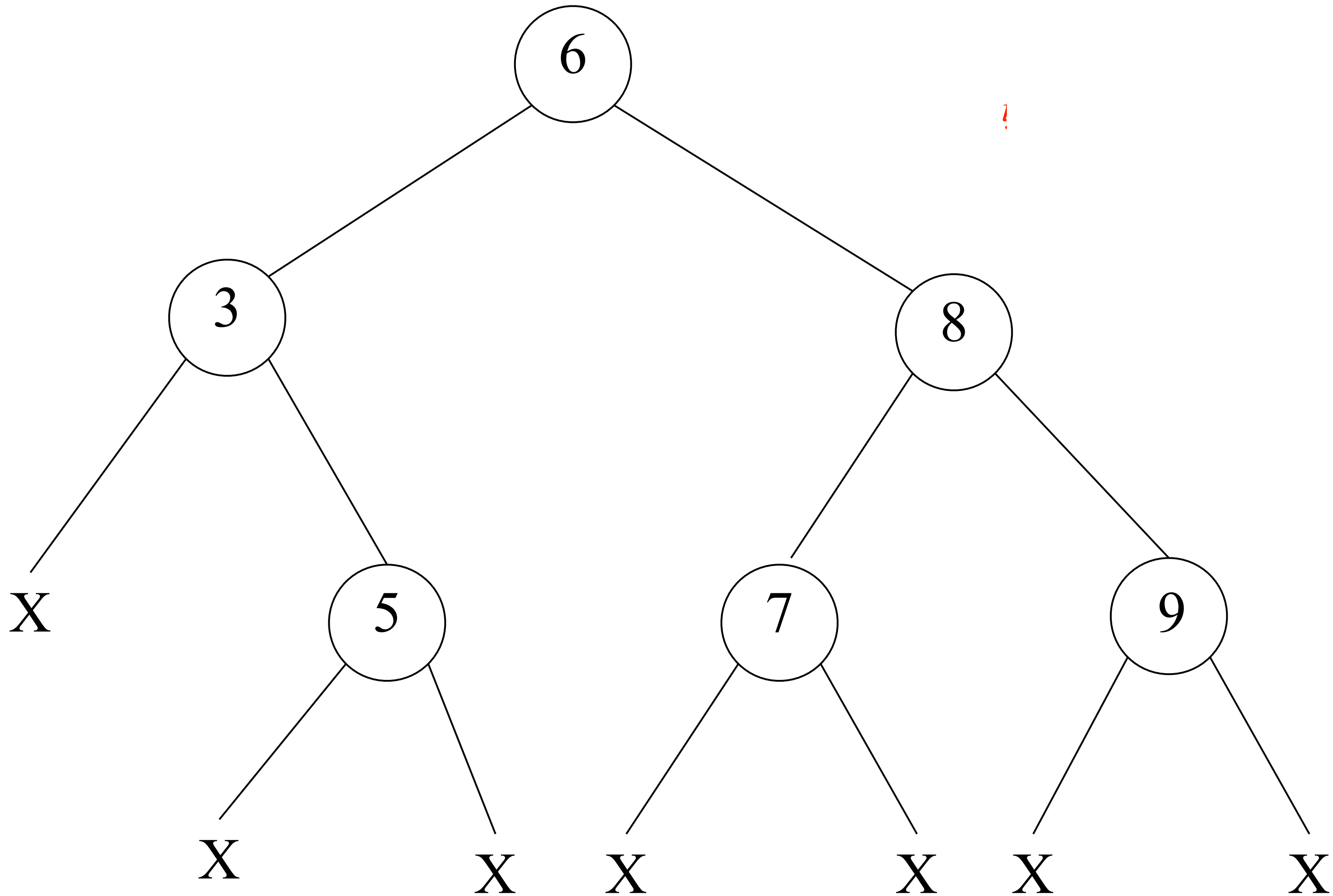




	Worst Case	Average Case
empty	O(1)	O(1)
search	O(N)	O(log N)
findMin	O(N)	O(log N)
findMax	O(N)	O(log N)
insert	O(N)	O(log N)
remove	O(N)	O(log N)
display	O(N)	O(N)

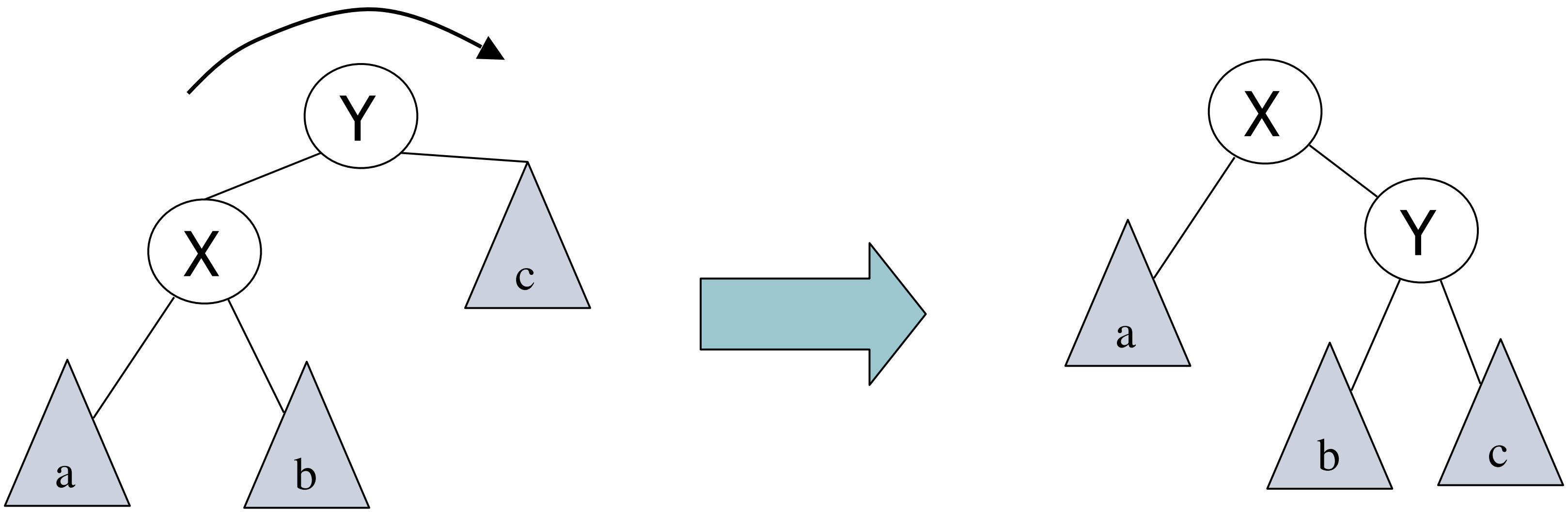
3 5 6 7 8 9

Inorder Traversal

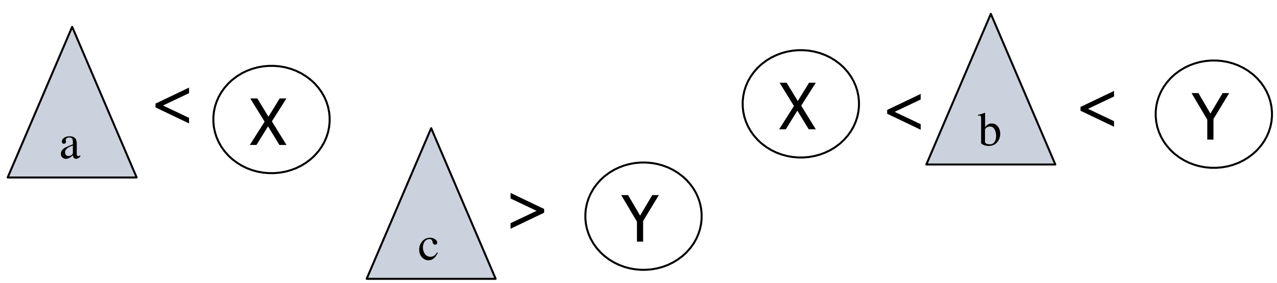


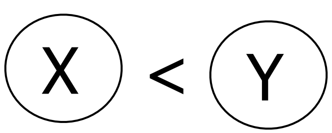
Right Rotate

Right rotate around y



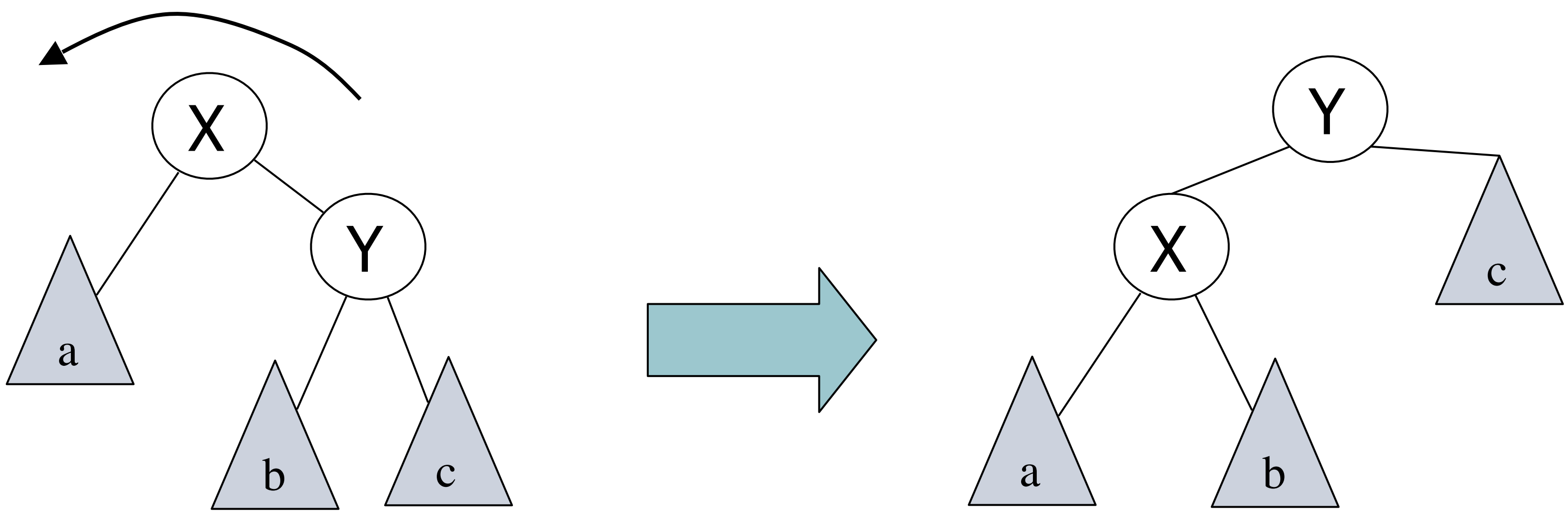
BST ordering properties hold:



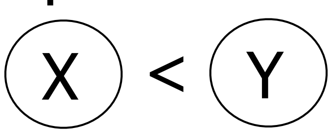


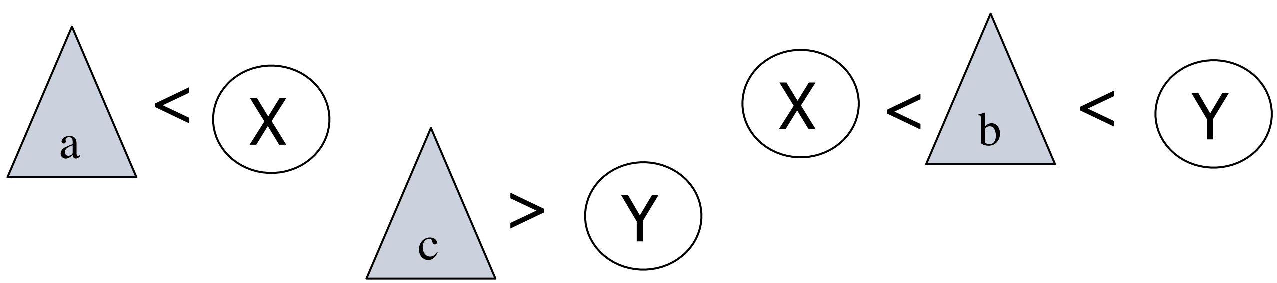
Left Rotate

Left rotate around X



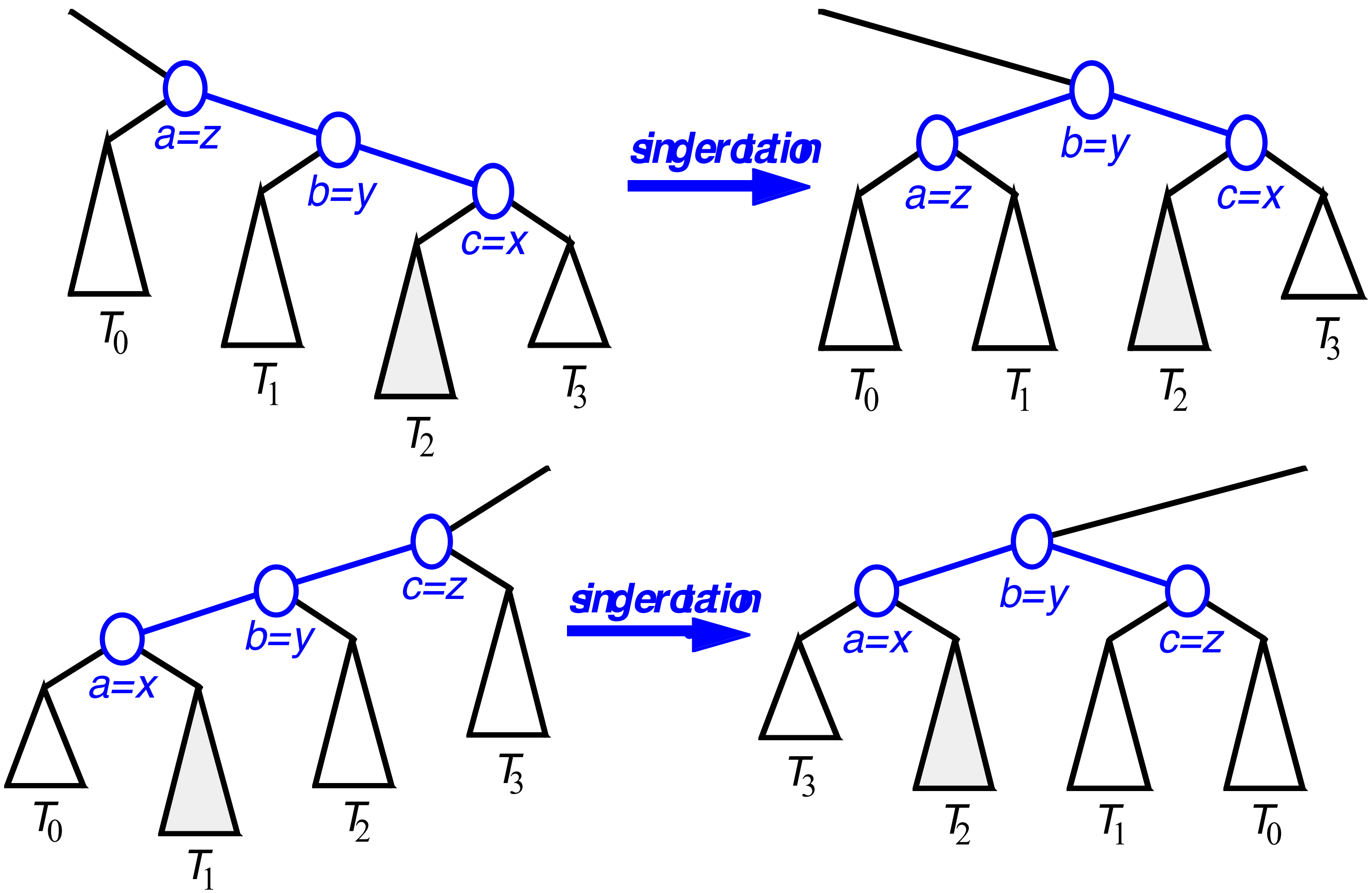
BST ordering properties hold:





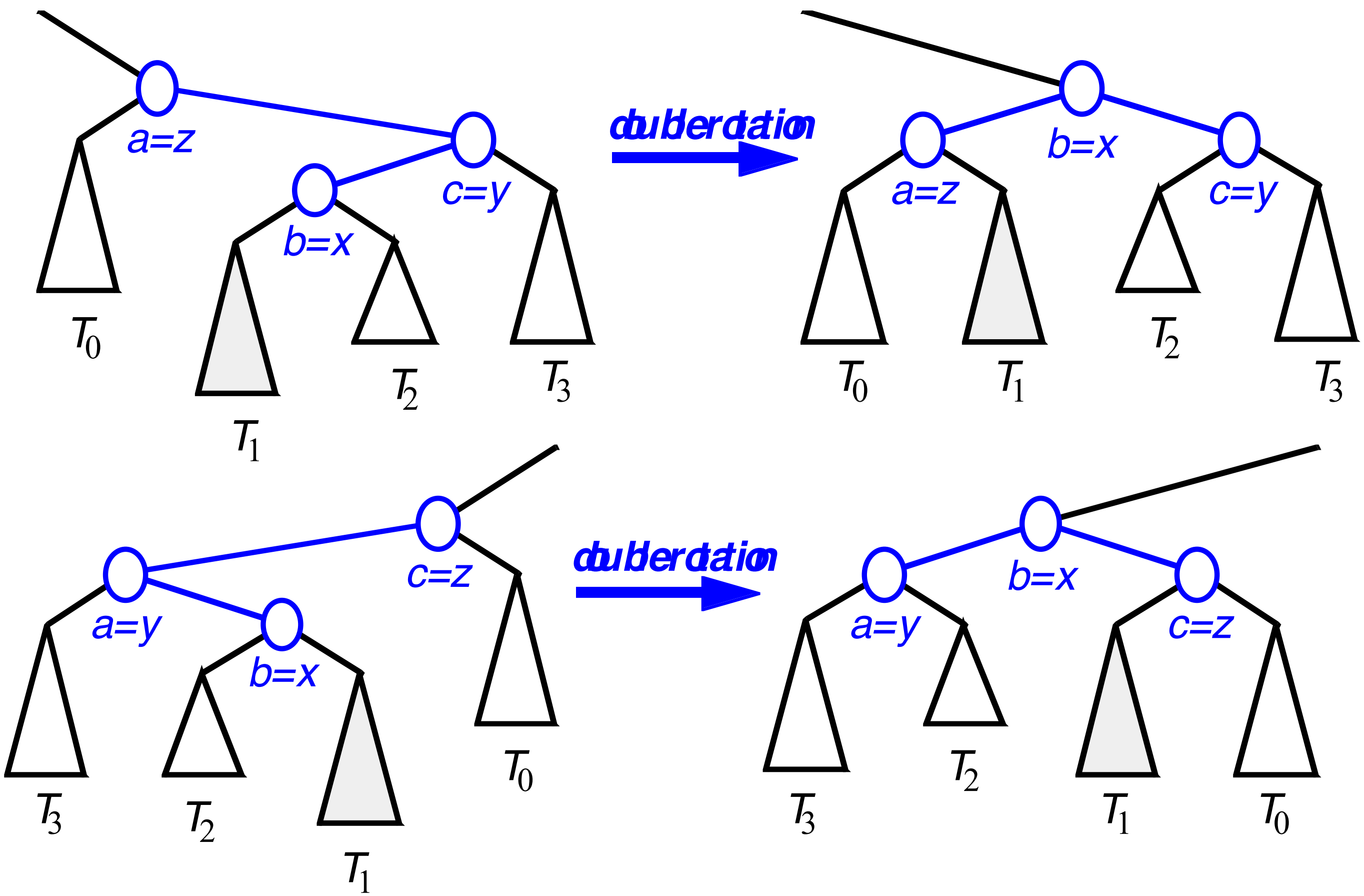
Restructuring (as Single Rotations)

- Single Rotations:



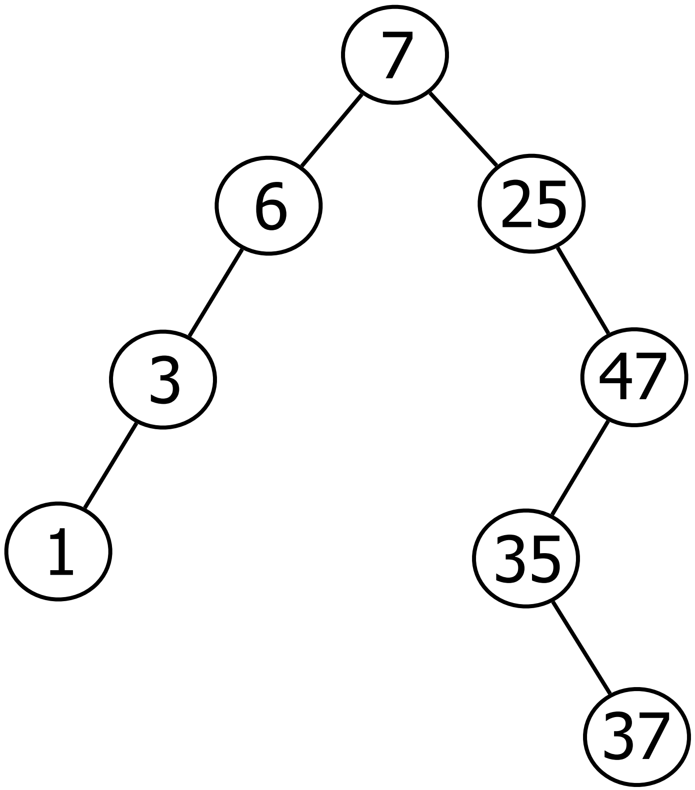
Restructuring (as Double Rotations)

- double rotations:



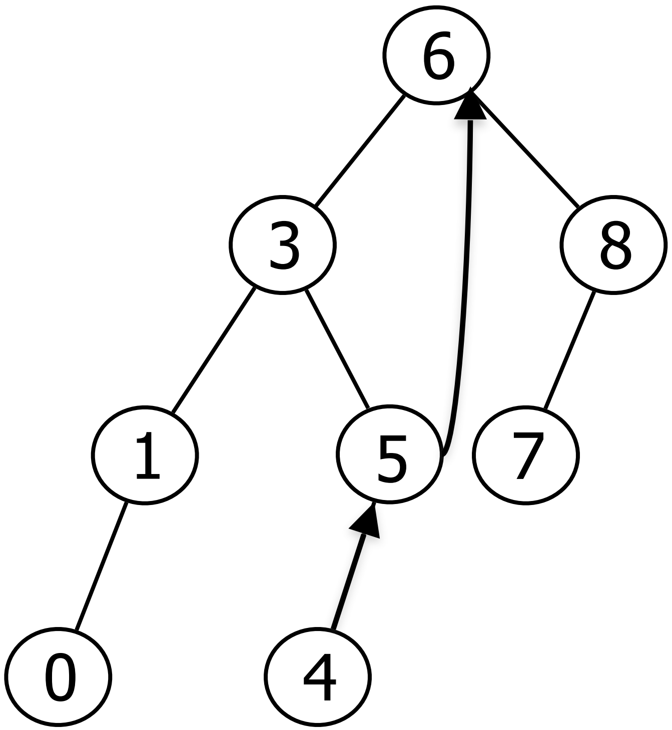
BST Insert

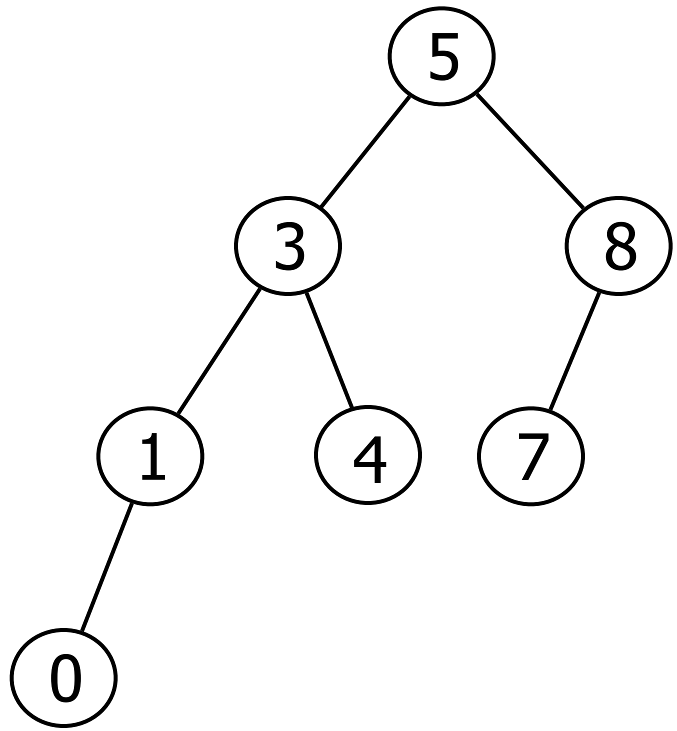
7, 25, 47, 35, 37, 6, 3, 1



Exam 4 Q4:
Summer 17

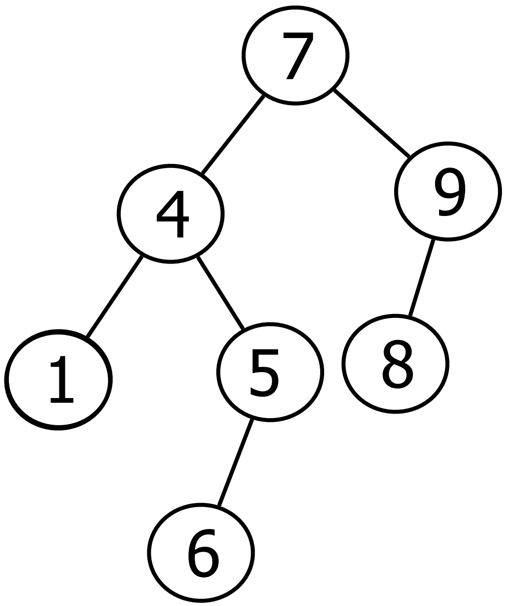
BST Remove





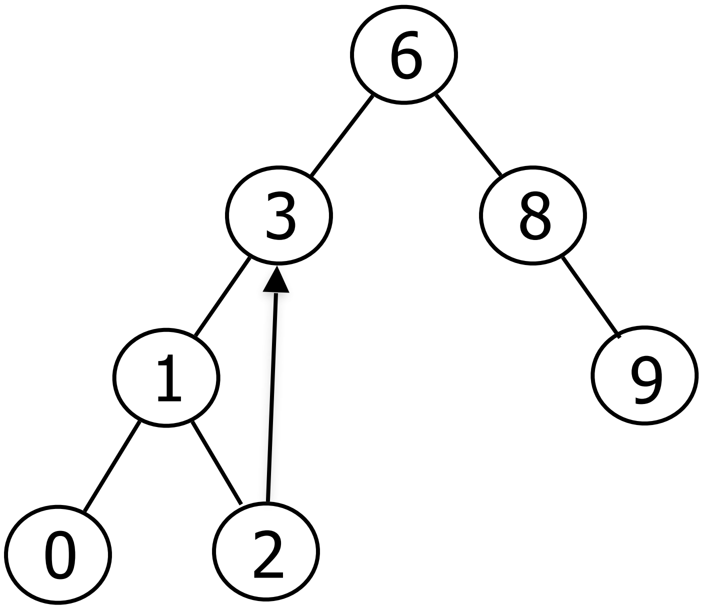
Exam 4 Q7:
Summer 17

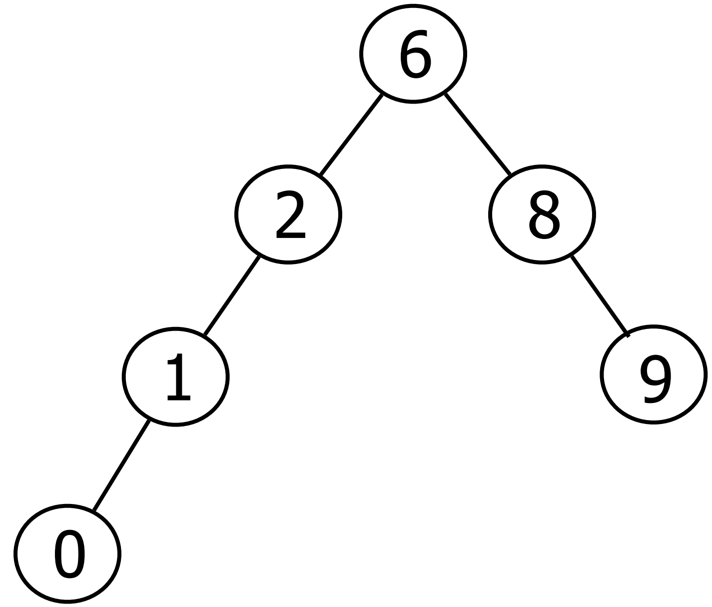
Binary Search Tree?



Exam 3 Q9:
Summer 17

BST Remove

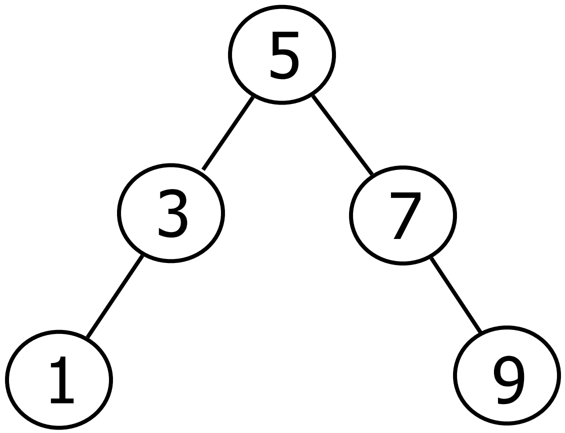




Exam 4 Q8:
Summer 17

Remove 3

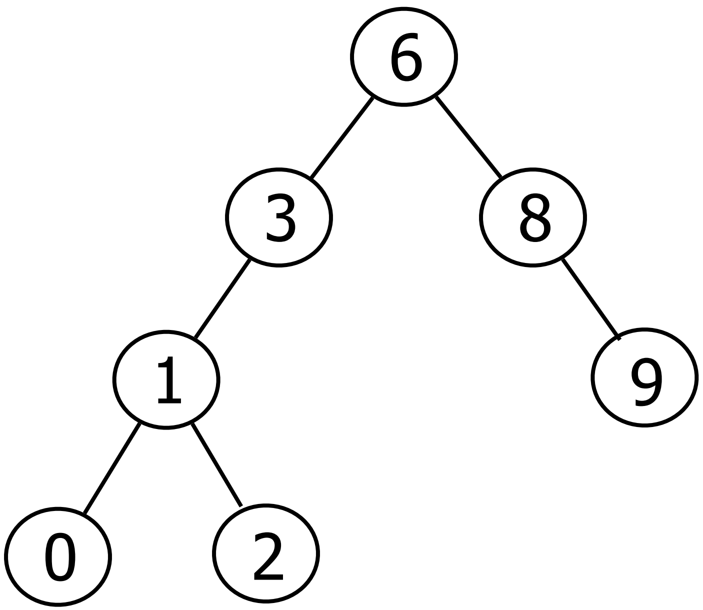
BST as an Array

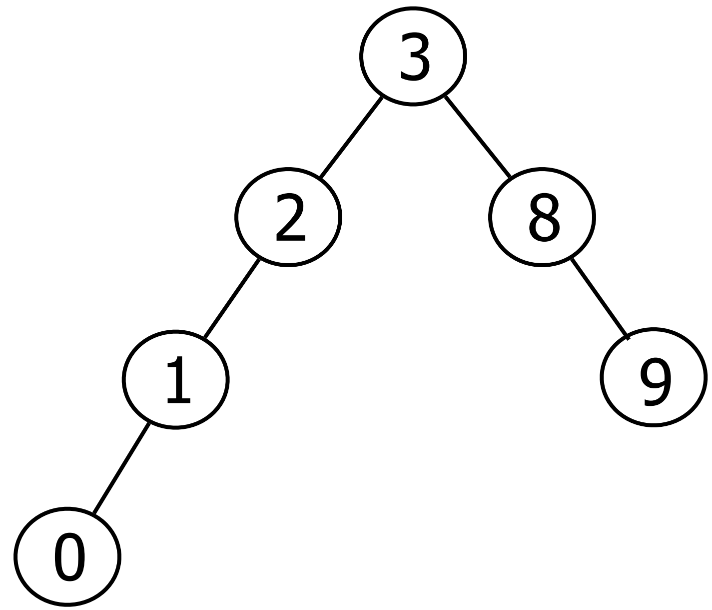


Exam 4 Q9:
Summer 17

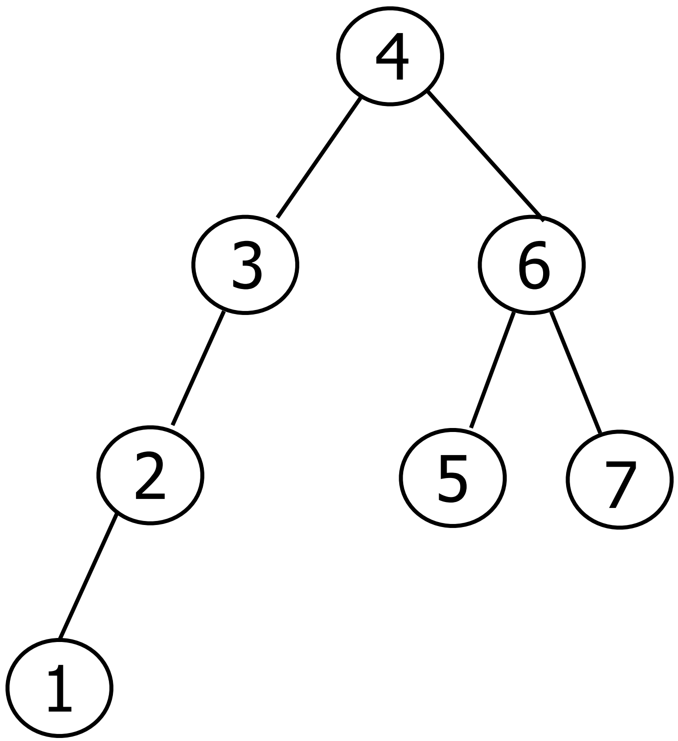
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
5	3	7	1	NULL	NULL	9

BST Traversal



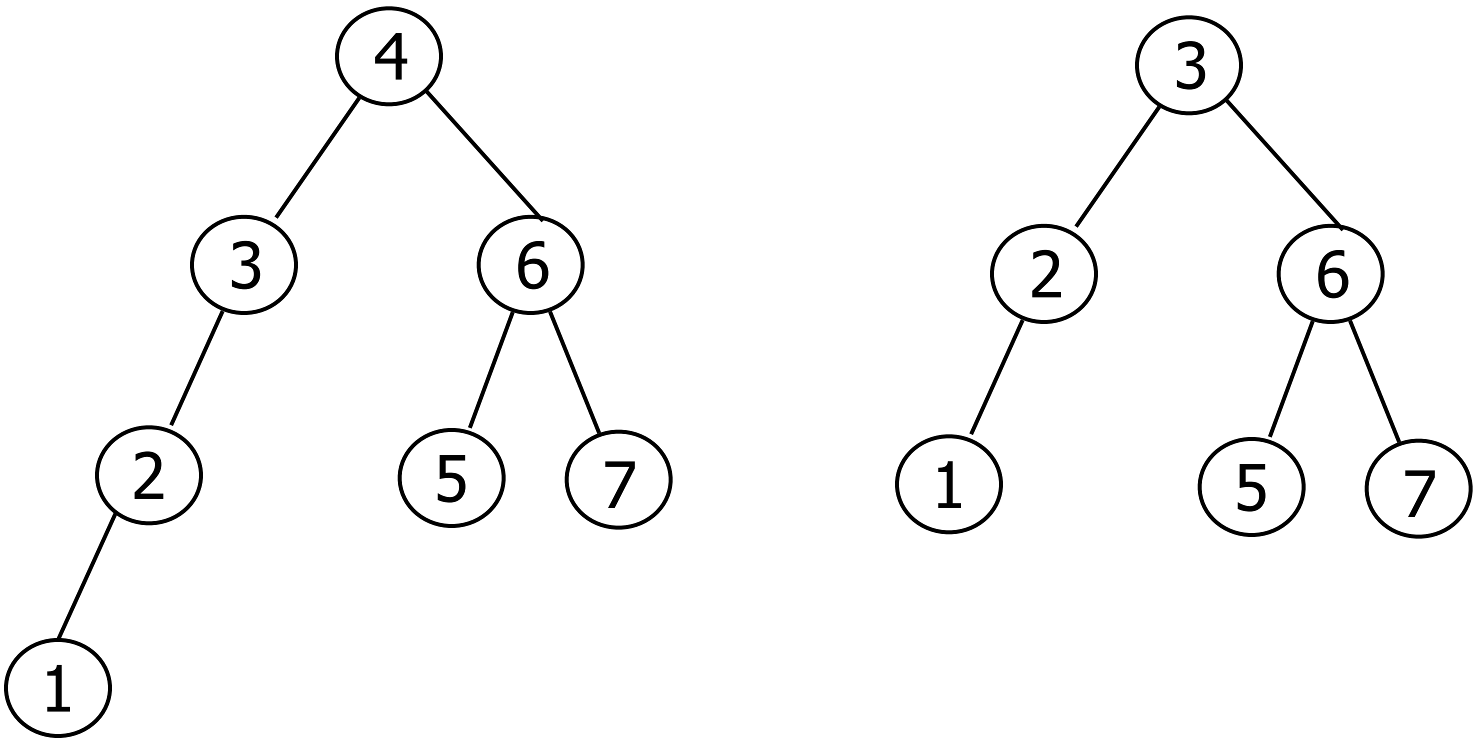


BST Insert



Insert into empty BST:
4, 3, 2, 1, 6, 7, 5

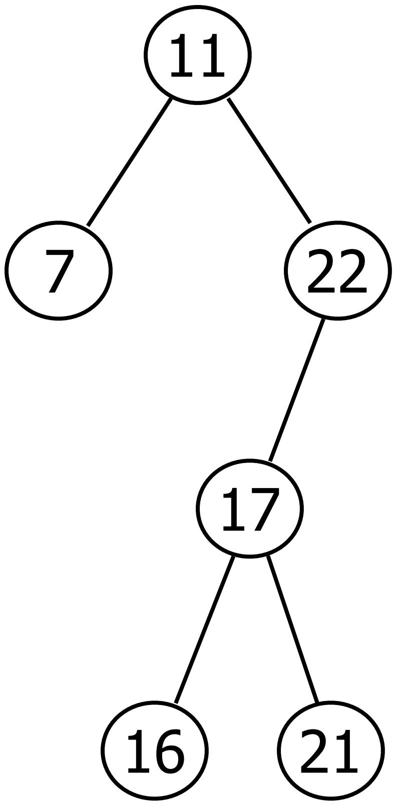
BST Remove



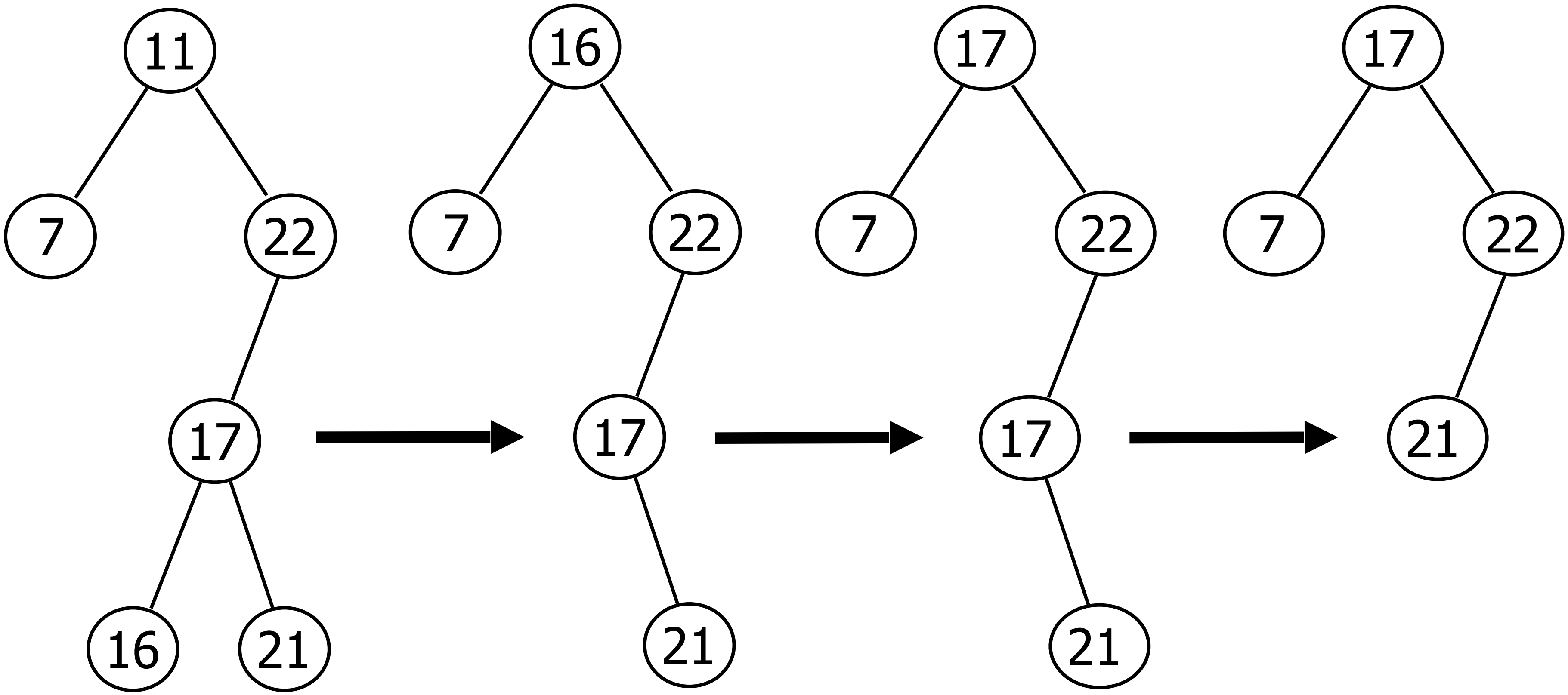
Remove root from BST
Use in-order
predecessor

BST Insert

Insert into empty BST:
11, 22, 17, 21, 16, 7



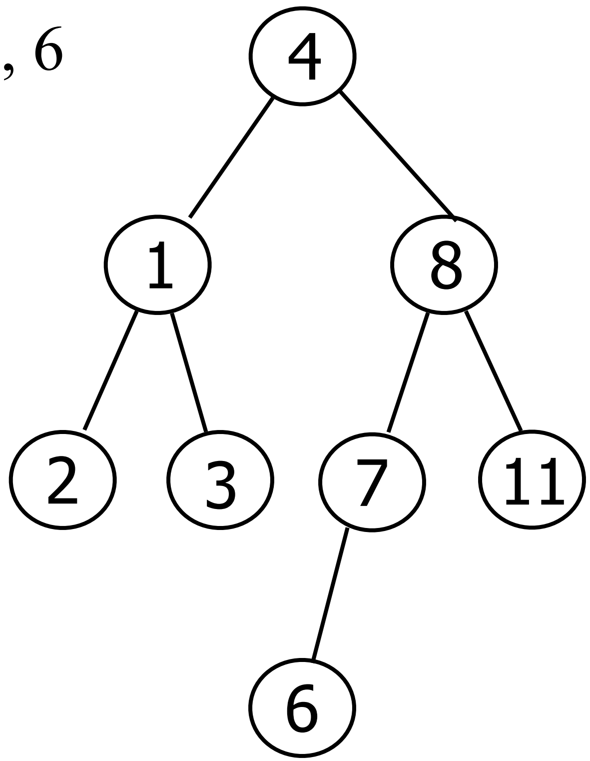
BST Remove



Remove 11
Remove 16

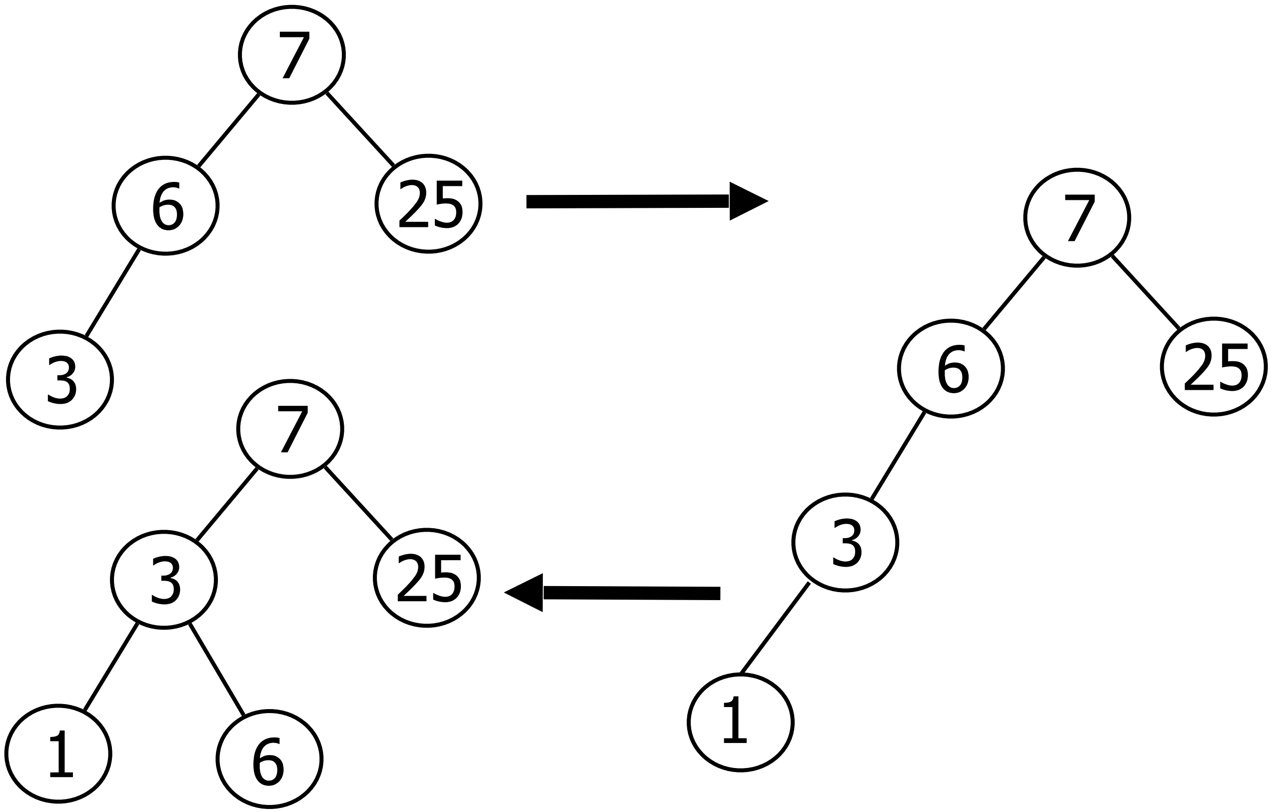
BST Insert

Insert into empty BST:
4, 1, 3, 2, 8, 11, 7, 6



AVL Tree Insert

Insert 1 into AVL tree



AVL Tree Rebalance