EECS 114:
Engineering Data Structures and Algorithms
Lecture 7

Instructor: Ryan Rusich
E-mail: rusichr@uci.edu
Office: EH 2204

The Henry Samueli School of Engineering
Electrical Engineering and Computer Science
University of California, Irvine

Dictionaries

- Allows user to insert key and a corresponding element and then search for the element using the key
- Methods:
  - findElement(k) - returns the element corresponding to key k
  - insertItem(k,o) - inserts the key-element pair <k,o> into the the dictionary
  - removeElement(k) - removes the key-element pair corresponding to k
  - size(), isEmpty()

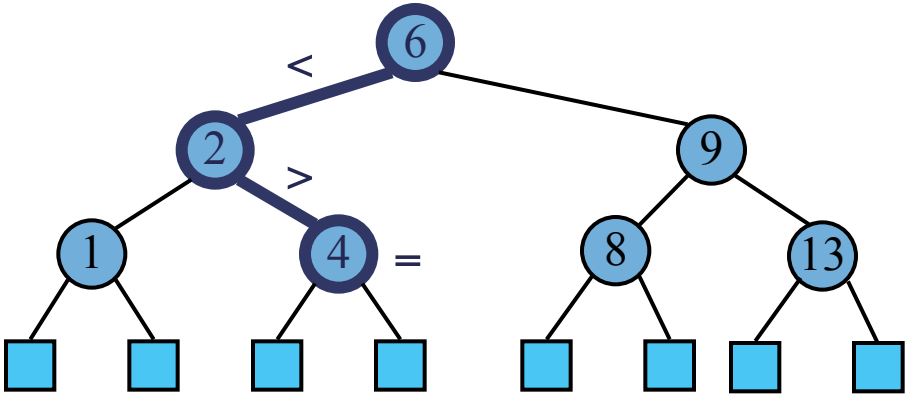
Implementation Strategies

- Unsorted sequence
  - Simply keep a list of key-element pairs
  - Insertion is O(1)
  - Search is O(n)
- Binary Search of an Array or Vector
  - Keep keys sorted in an array
  - To find an item start search at the middle of the array
  - If we're too high, search in the first half
  - If too low, search in the second half
  - Repeat until we find the element
  - O(log n)

Fast Operations

- What if we could _____ in O (log n)?
  - search
  - insert
  - remove
  - $\log_2 1{,}048{,}576 = 20$
  - $\log_2 1{,}073{,}741{,}824 = 30$

Binary Search Trees
BST

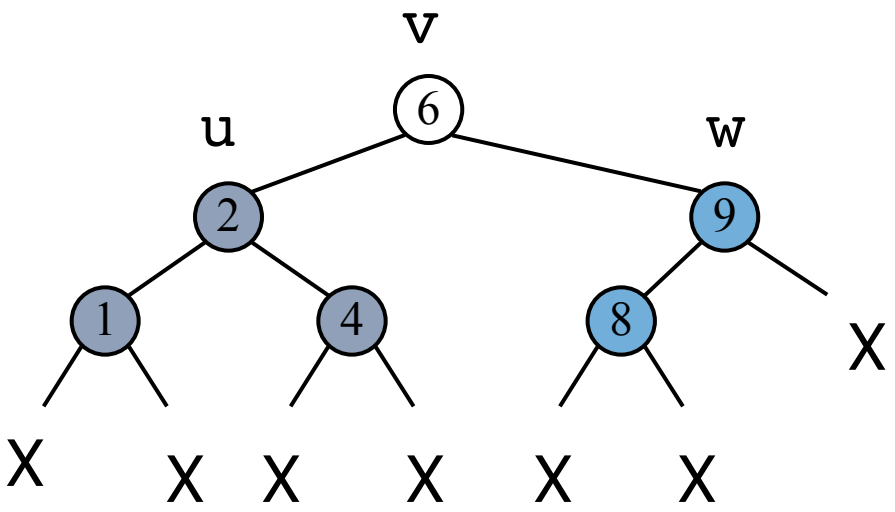


Binary Search Tree

- 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

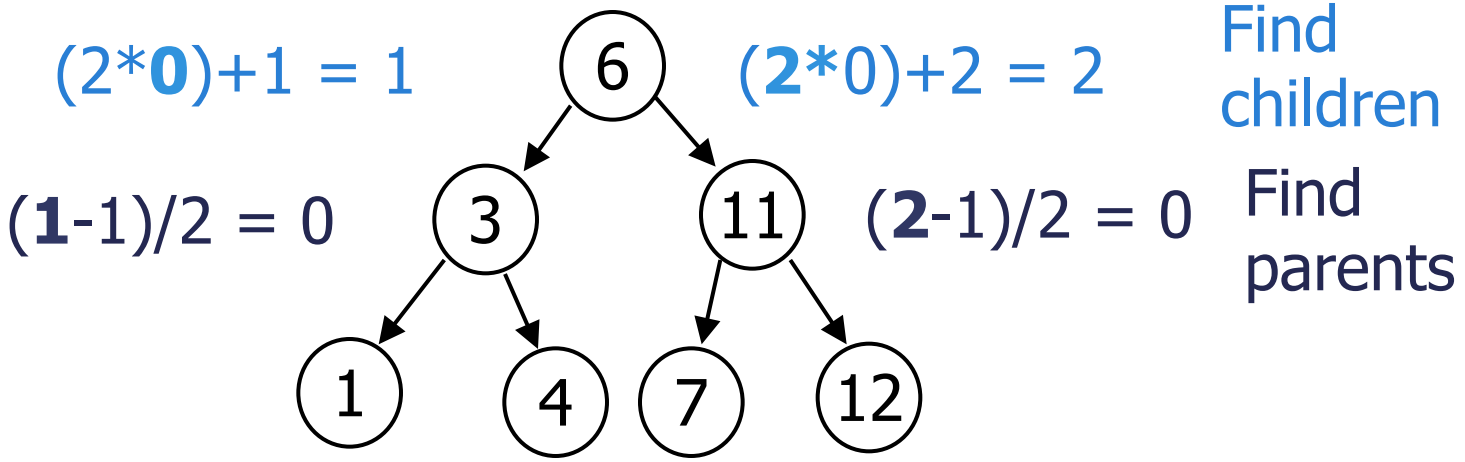
- isEmpty - return true if empty, false if not
- search (private) - return pointer to node in which key is found, otherwise return NULL
- search (public) - return true if key is found, otherwise return false
- findMin - return smallest node value
- findMax - return largest node value

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

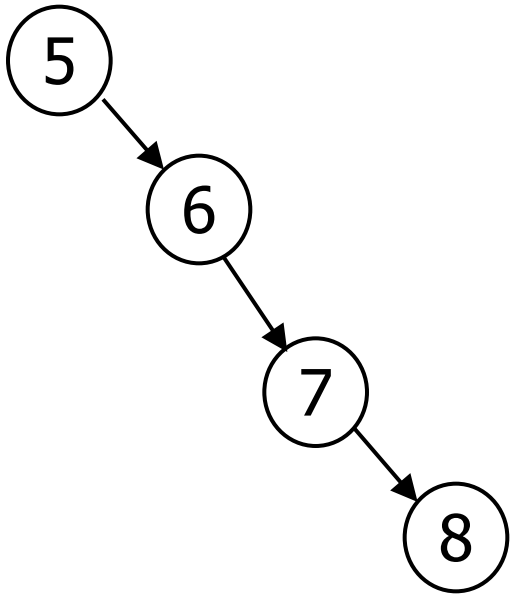
Array Implementation of a BST

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



Array Implementation of a BST

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



Linked Implementation of a BST

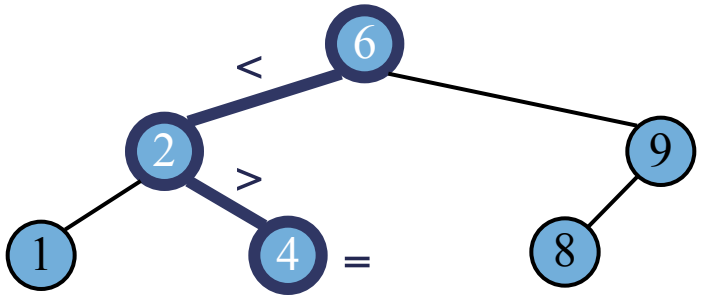
- Linked implementation
  - Similar to linked list – dynamic size can grow and shrink easily during runtime

```
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

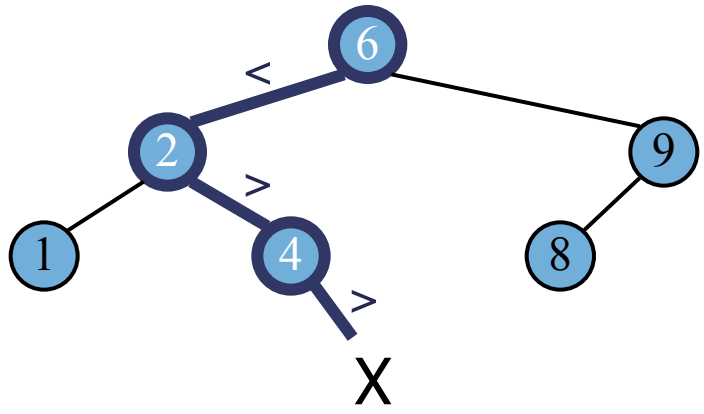
```
void preorder(Node* nodePtr)
        if ( nodePtr )
                print node
                preorder (nodePtr->left)
                preorder (nodePtr->right)
```

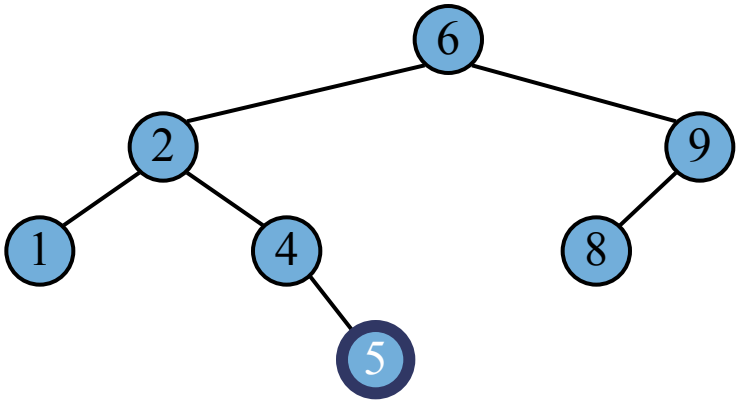
Postorder Traversal

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

Insertion

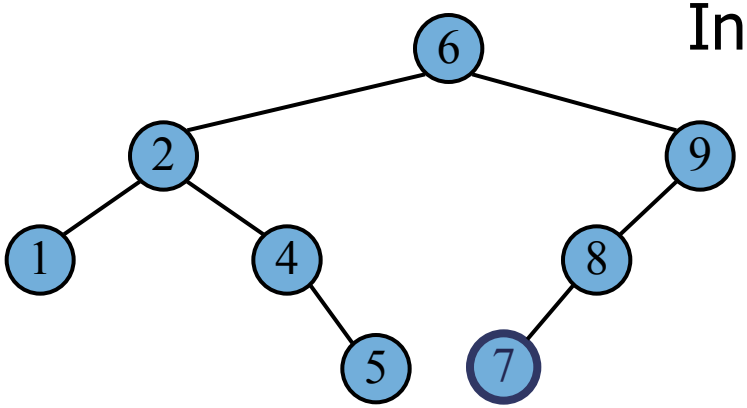
- To perform operation insertItem(k, o), we search for the position k would be in if it were in the tree
- All insertions create a new leaf node
- Example: insert 5



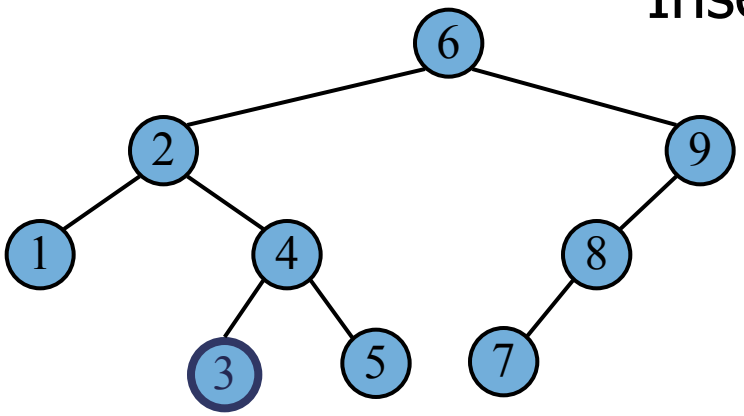


Insertion

Insert 7



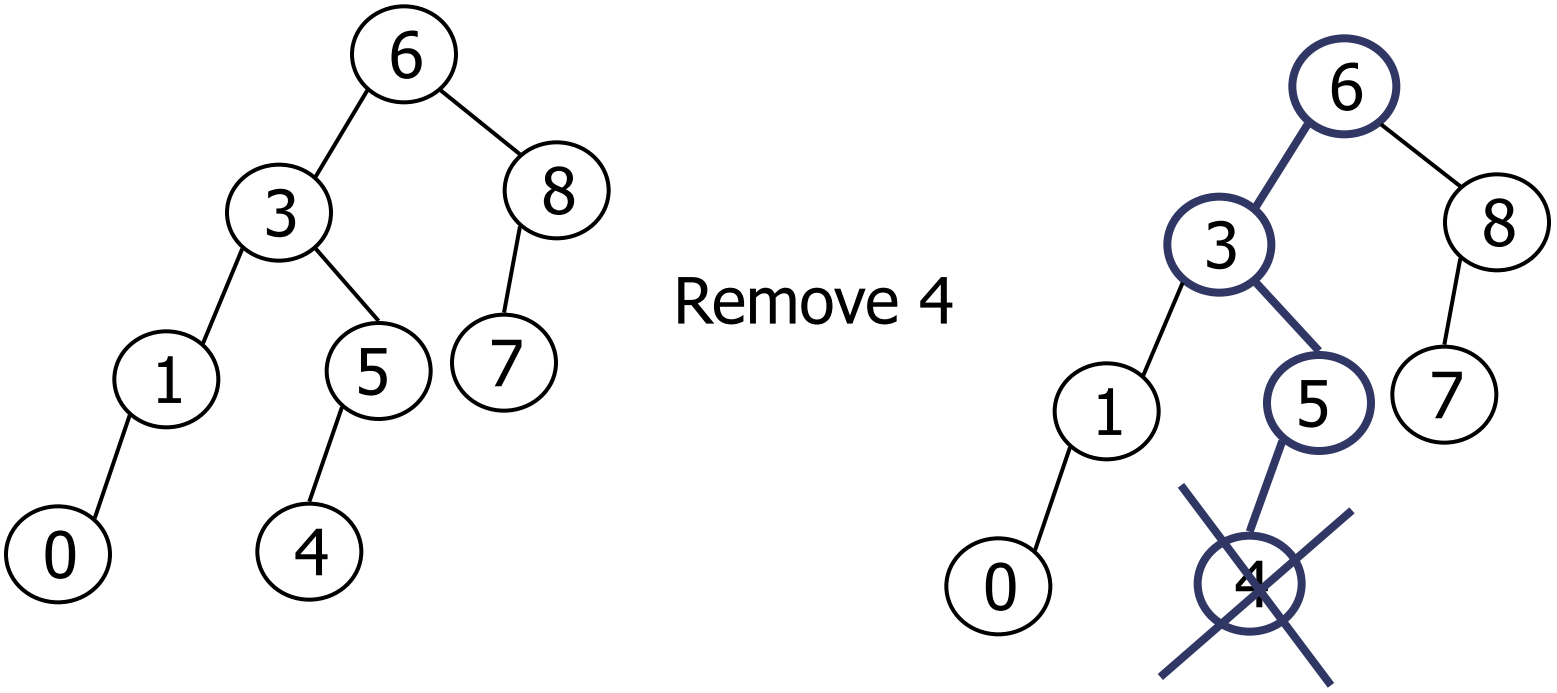
Insert 3



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

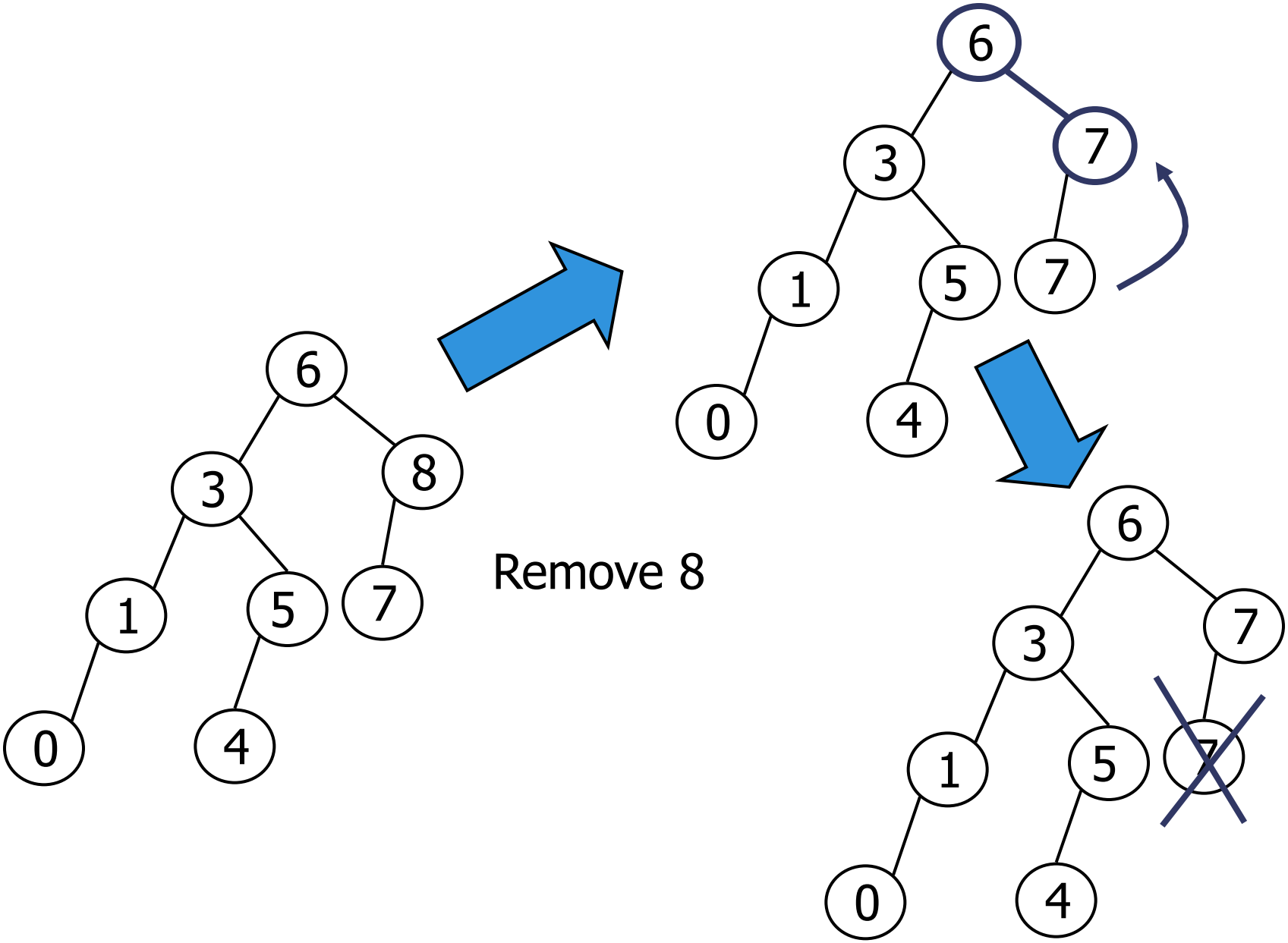


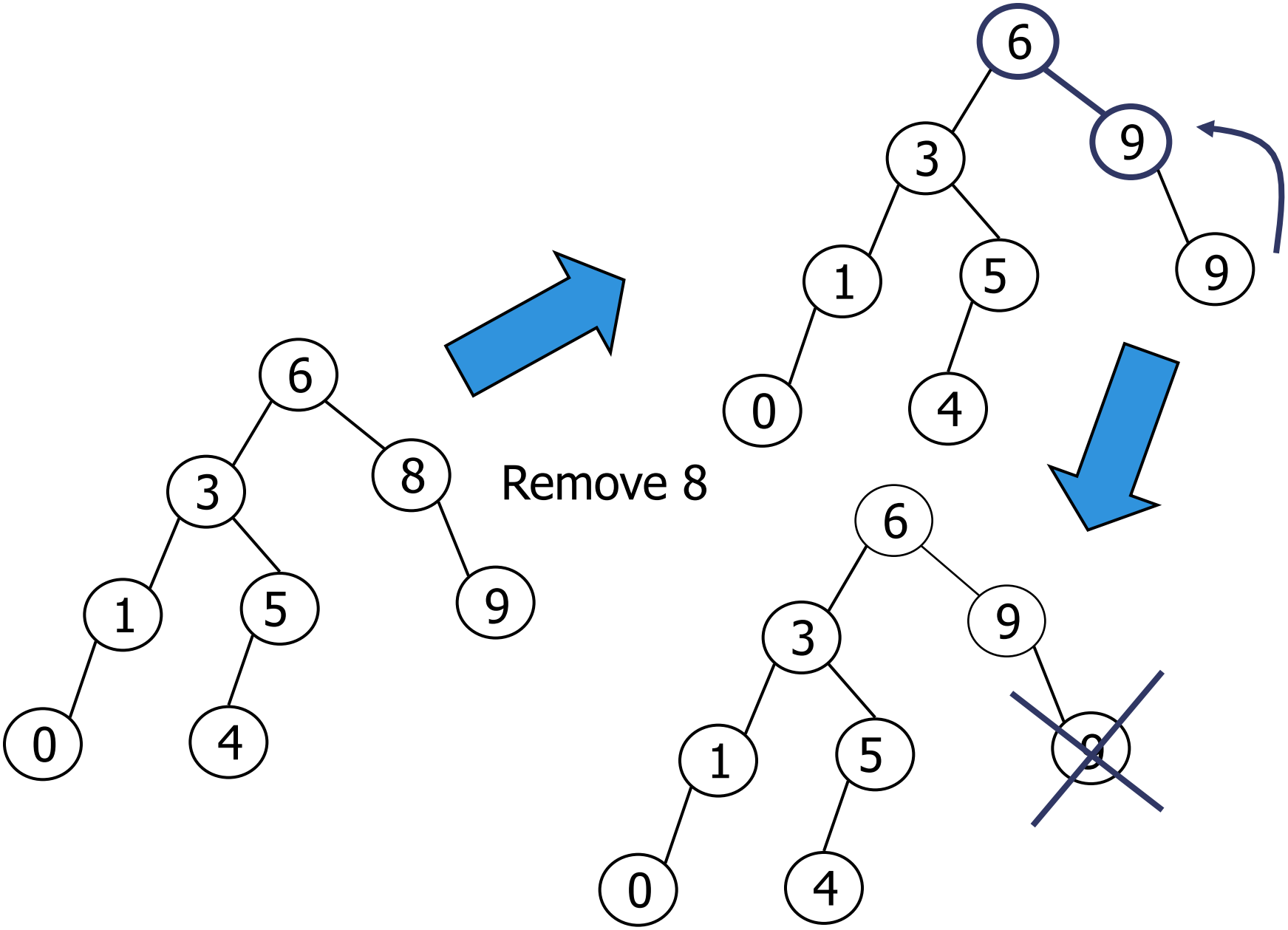
Deletion - Removing a node with children

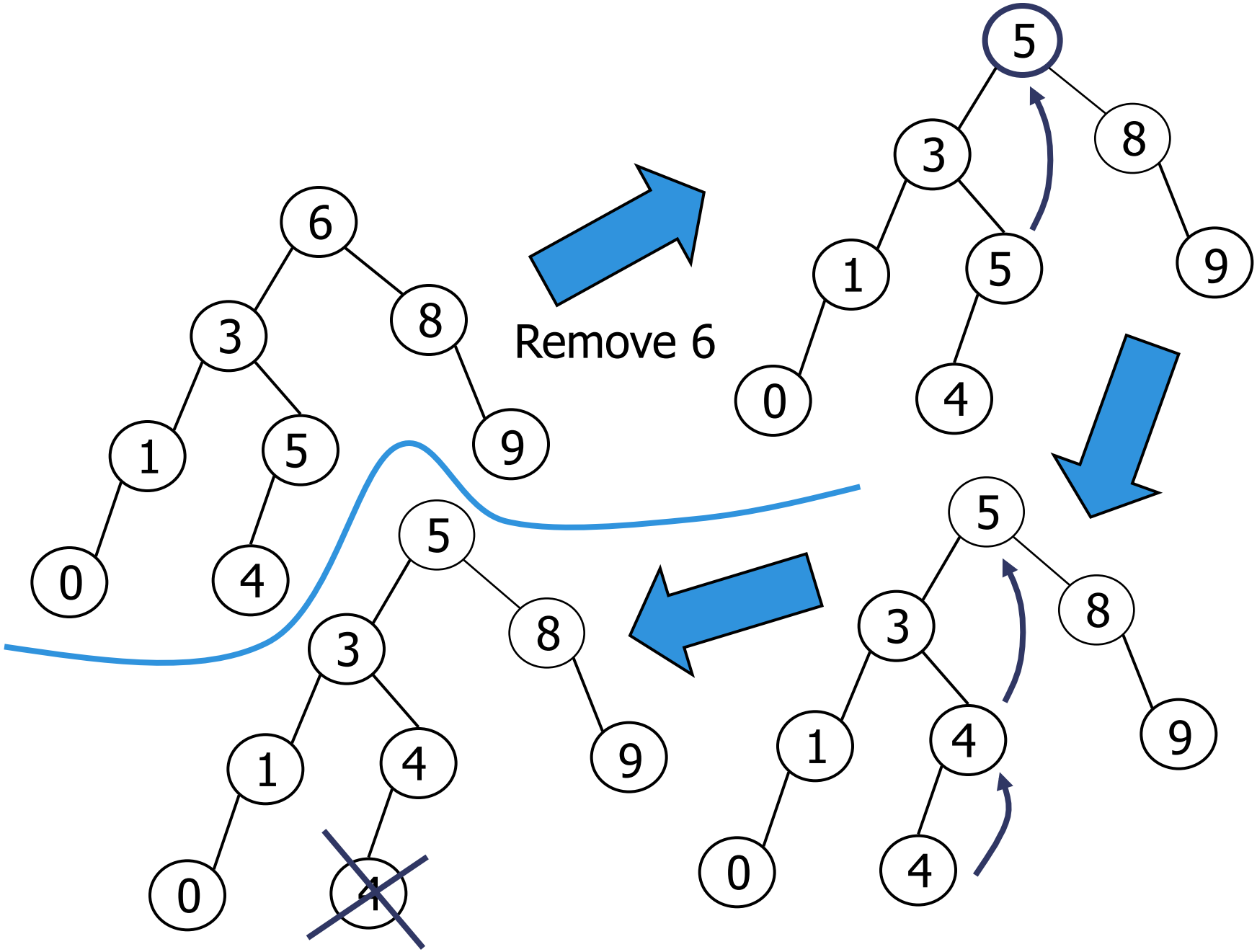
- Otherwise the node has children - find replacement node
  - If the left child exists
    - Replace node information with the ***largest*** value smaller than the value to remove
      - findMax(leftChild)
  - Else there is a right child
    - Replace node information with the ***smallest*** value larger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

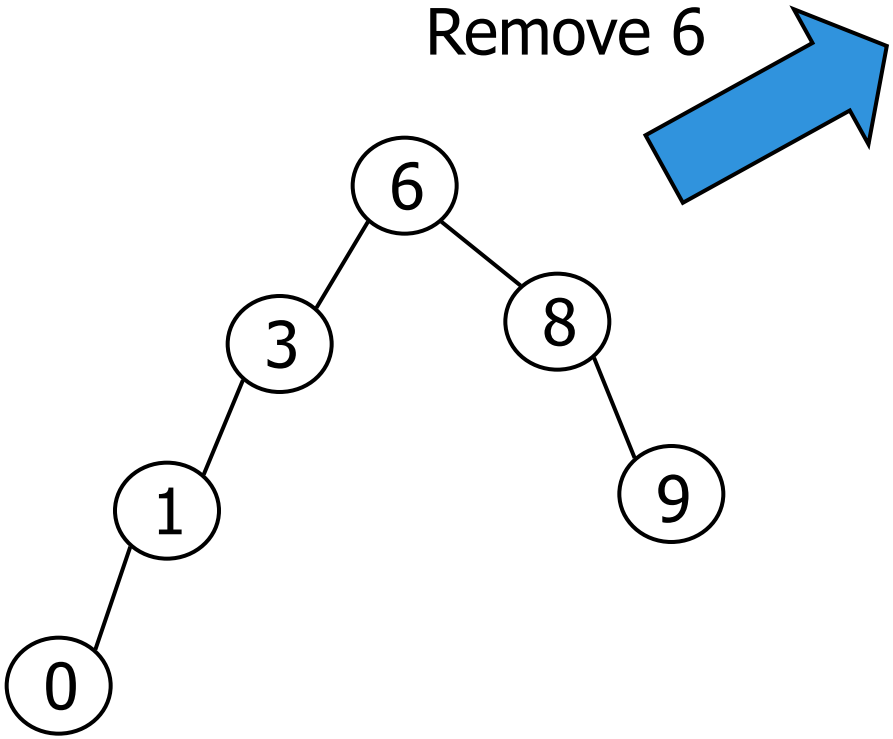

6
3
8
1
5
7
0
4
Remove 8
6
3
7
1
5
7
0
4
6
3
7
1
5
7
0
4


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

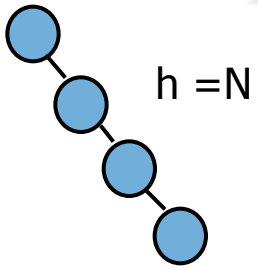

Remove 6
6
3
8
1
5
9
0
4
5
3
8
1
5
9
0
4
5
3
8
1
4
9
0
4
5
3
8
1
4
9
0
4

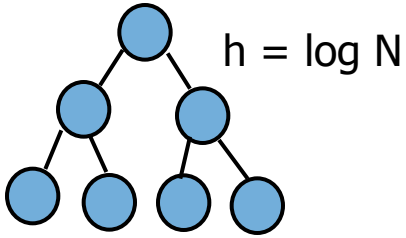
In class exercise

Remove 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)