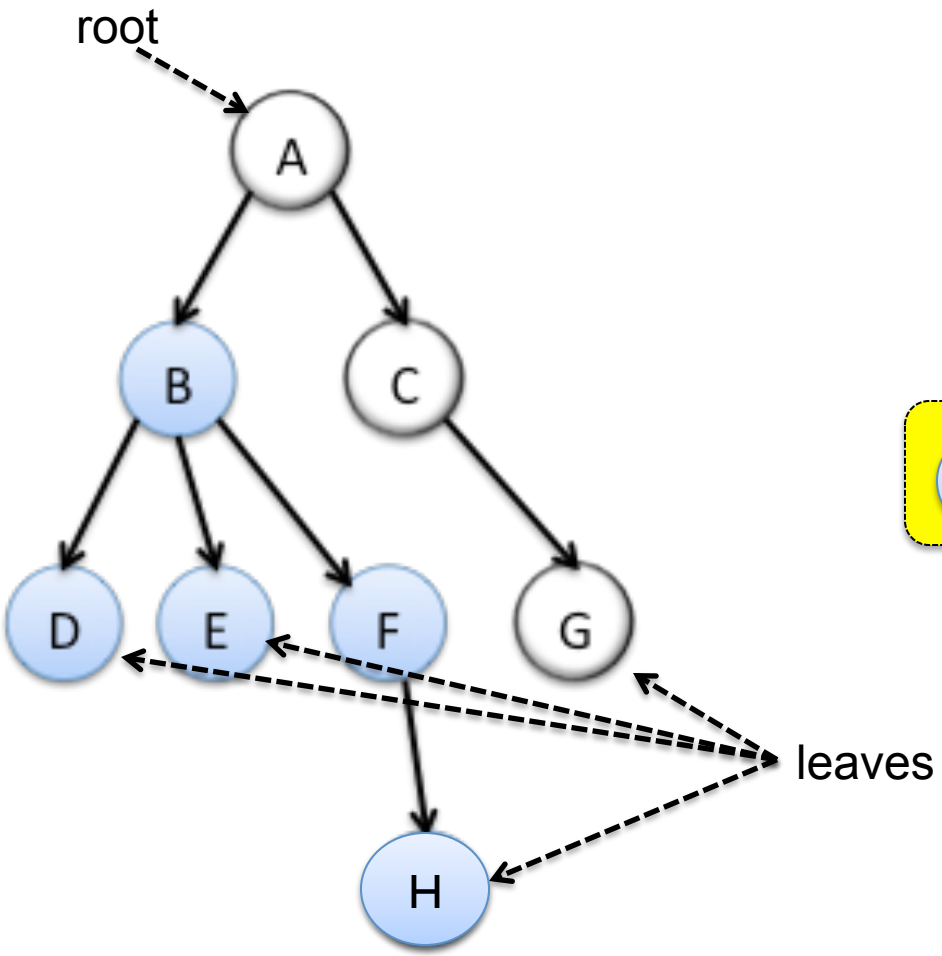
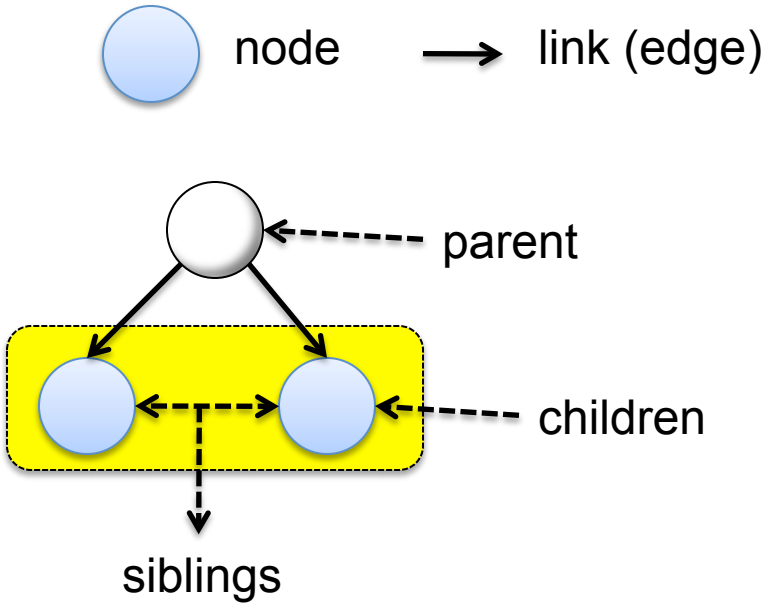
CS32 Discussion
Section 1B
Week 8

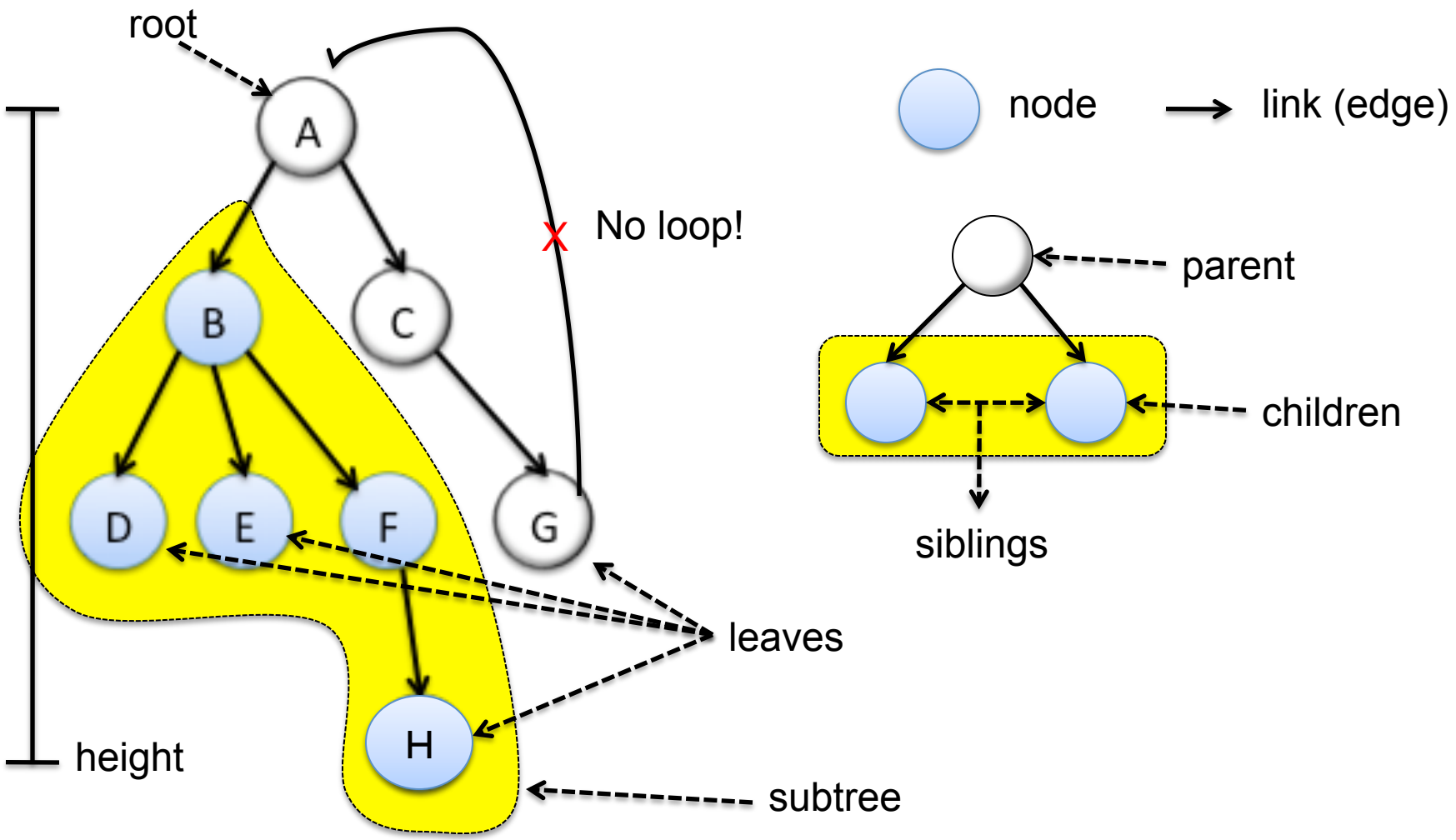
TA: Zhou Ren

Tree: Definitions

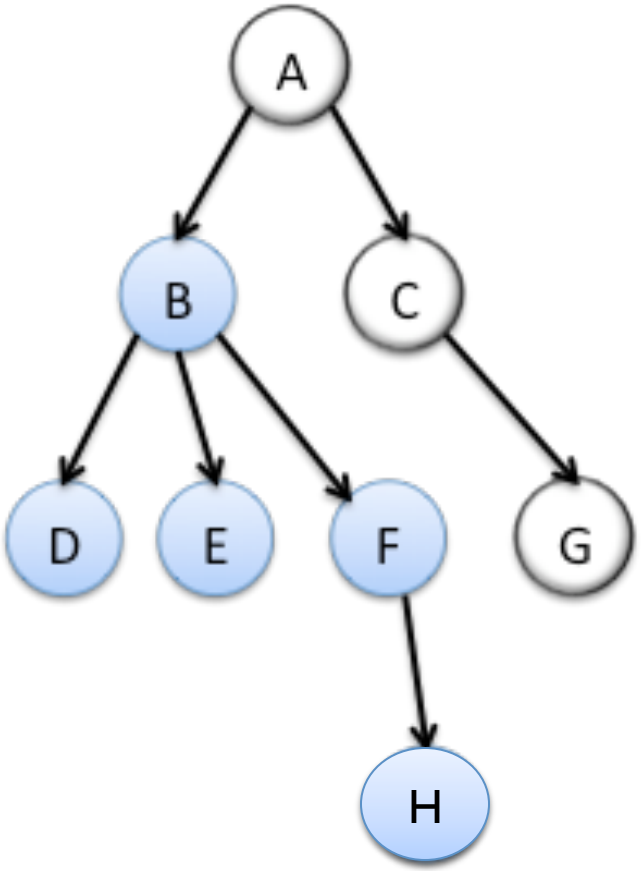




Tree: Definitions

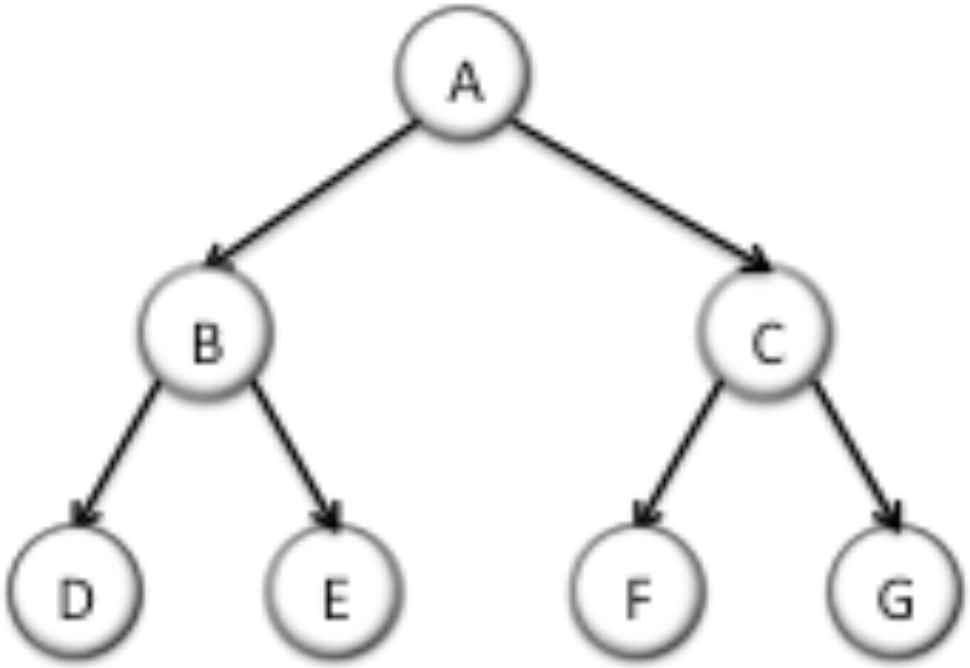


Bound on # of edges



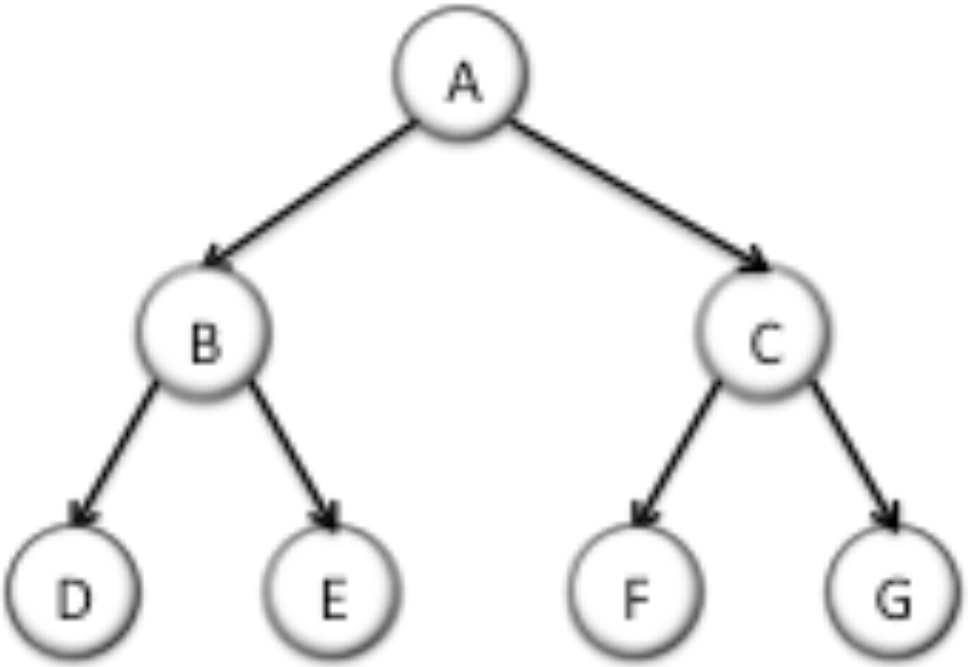
How many edges should there be in a tree of **n** nodes?

Binary Trees



No node has more than 2 children (left child + right child).

Binary Trees



How many nodes can a binary tree of height ***h*** have?
(one with max. # of nodes == full binary tree)

Tree is a data structure!

- For every data structure we need to know:
  - how to insert a node,
  - how to remove a node,
  - search for a node

- and (for tree only)
  - how to traverse the tree

Tree is a data structure!

- For every data structure we need to know:
  - how to insert a node,
  - how to remove a node,
  - search for a node

- and (for tree only)
  - how to traverse the tree

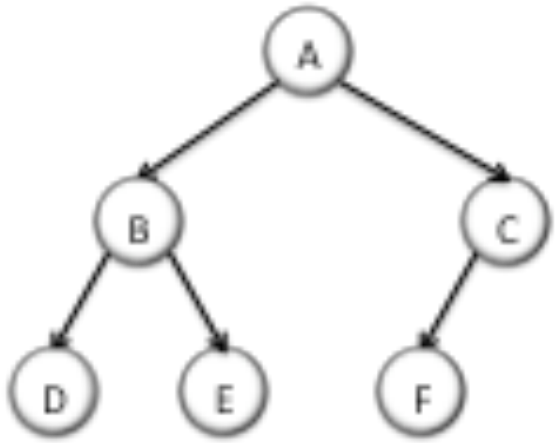
```
struct Node
{
    ItemType val;
    Node* left;
    Node* right;
};
```

Three Methods of Traversal

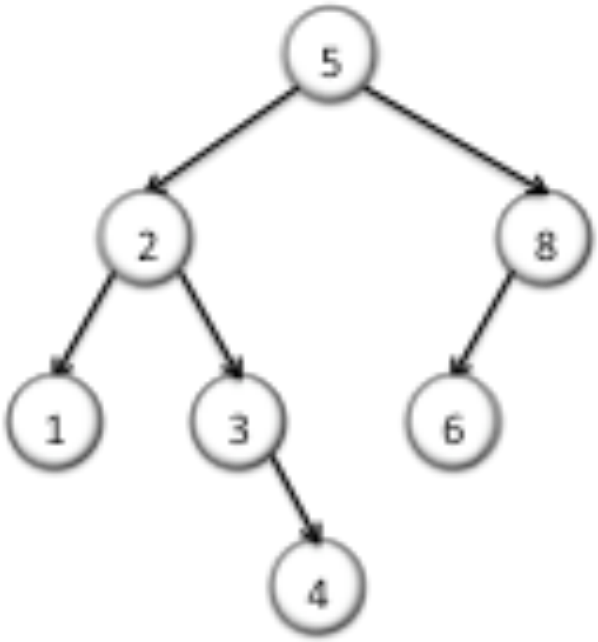
```
void preorder(const Node *node)
{
  if (node == NULL) return;
  cout << node->val << " ";
  preorder(node->left);
  preorder(node->right);
}
```

```
void inorder(const Node *node)
{
  if (node == NULL) return;
  inorder(node->left);
  cout << node->val << " ";
  inorder(node->right);
}
```

```
void postorder(const Node *node)
{
  if (node == NULL) return;
  postorder(node->left);
  postorder(node->right);
  cout << node->val << " ";
}
```

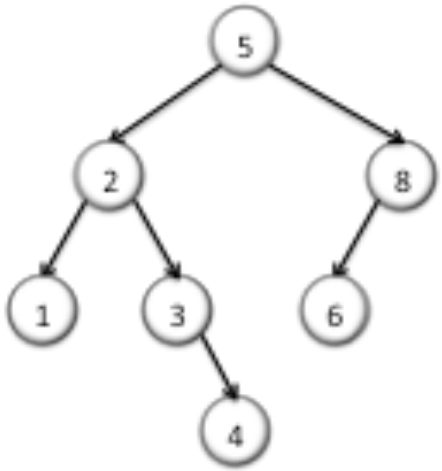


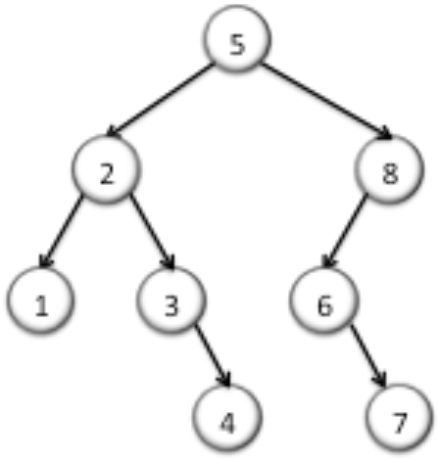
Binary Search Tree



- At all nodes:
  - All nodes in the left subtree have smaller values than the current node's value
  - All nodes in the right subtree have larger values than the current node's value
- Which traversal method should you use to:
  - print values in the increasing order?
  - print values in the decreasing order?

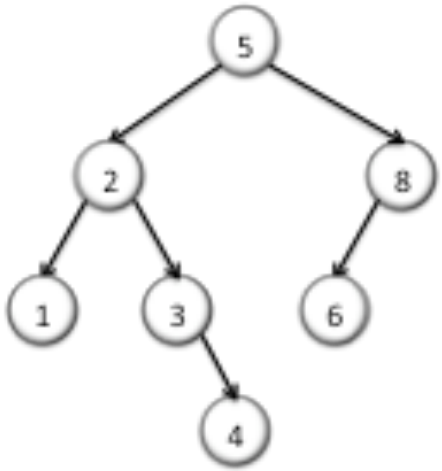
Insert

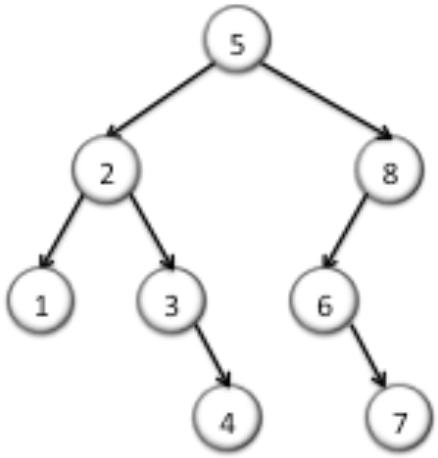





```
void insert(Node* &node, ItemType newVal)
{

}
```

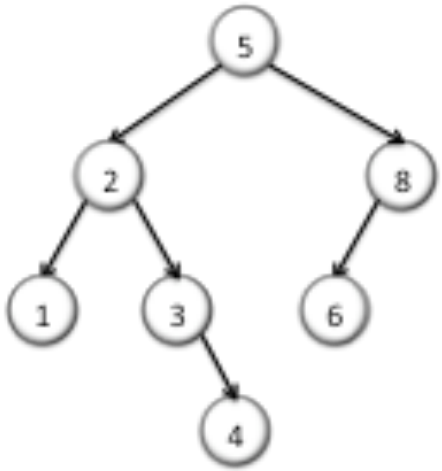
Insert



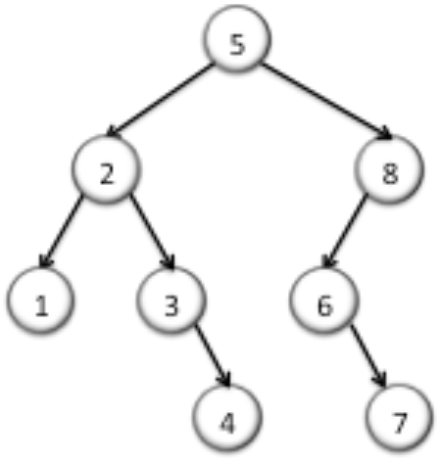



```
void insert(Node* &node, ItemType newVal)
{
    if (node == NULL)
    {
        node = new Node;
        node->val = newVal;
        node->left = node->right = NULL;
    }

    if (node->val > newVal)
        insert(node->left, newVal);
    else
        insert(node->right, newVal);
}
```

Insert

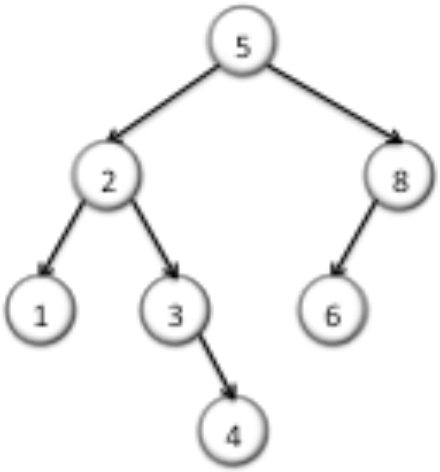


- Worst-case time complexity?
  - as many steps as the height of the tree
  - full tree: $n = 2^{h+1} - 1 \approx 2^{h+1}$ nodes
  - $h \approx \log_2 n - 1$



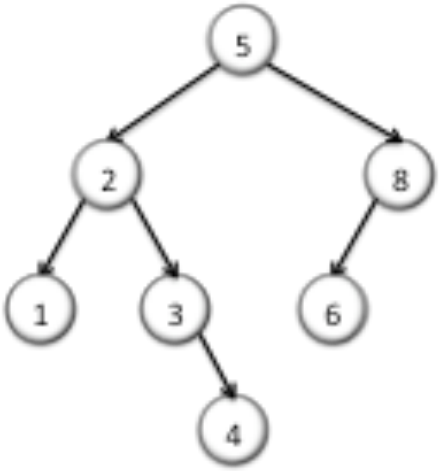
  - Roughly, it takes O(log N).

Search



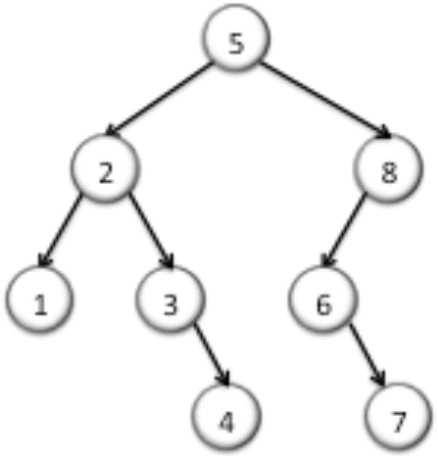

```
Node* search(const Node *node, ItemType value)
{




}
```

Search

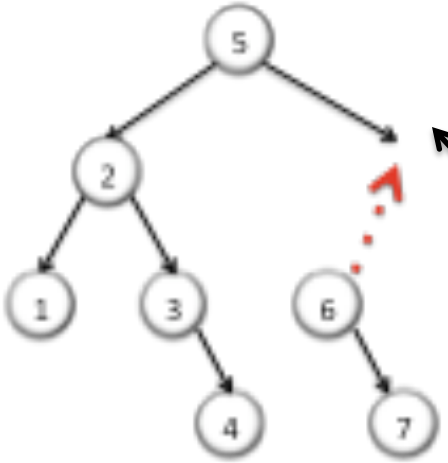



```
Node* search(const Node *node, ItemType value)
{
    if (node == NULL)
        return NULL;

    if (node->val == value)
        return node;
    else if (node->val > value)
        return search(node->left, value);
    else
        return search(node->right, value);
}
```

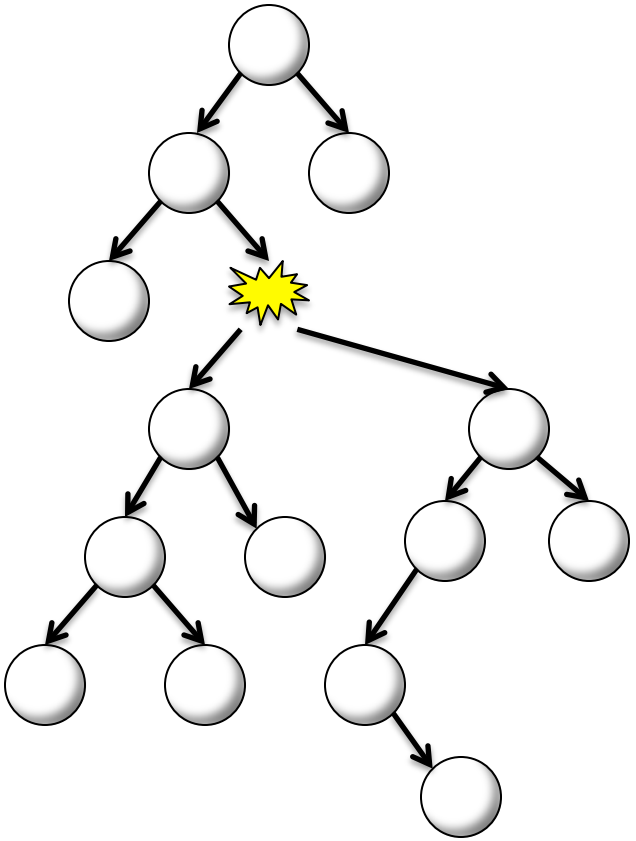
Removal



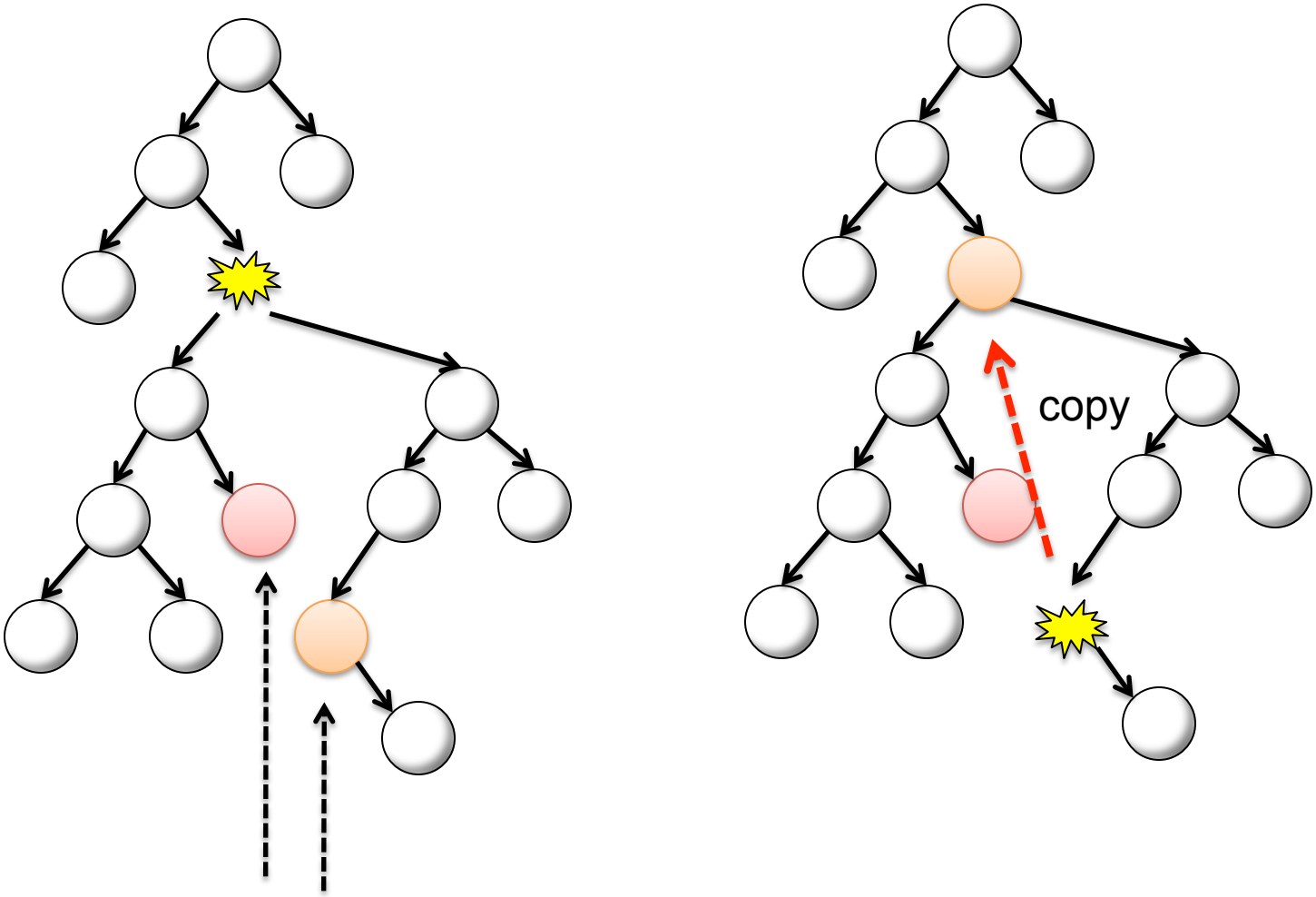
- A little tricky!
- General strategy:
  - Find a replacement.
  - Delete the node.
  - Replace.
- Case-by-case analysis
  - Case 1: the node is a leaf (easy)
  - Case 2: the node has one child
  - Case 3: the node has two children



Case 3



Case 3



Use in-order traversal to identify these nodes

findMax

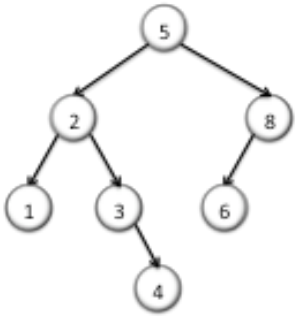
```
ItemType findMax(const Node *node)
{









}
```

findMax

```
ItemType findMax(const Node *node)
{
    if (node->left == NULL && node->right == NULL)
        return node->val;

    int maxVal = node->val;
    int leftMax = findMax(node->left);
    int rightMax = findMax(node->right);

    if (maxVal < leftMax)
        maxVal = leftMax;
    if (maxVal < rightMax)
        maxVal = rightMax;

    return maxVal;
}
```

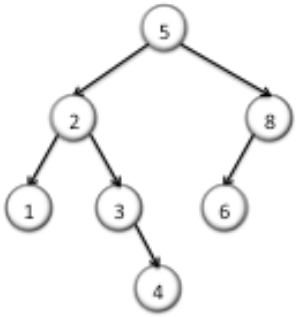
findMin

```
ItemType findMin(const Node *node)
{
    if (node->left == NULL && node->right == NULL)
        return node->val;

    int minVal = node->val;
    int leftMin = findMin(node->left);
    int rightMin = findMin(node->right);

    if (minVal > leftMin)
        minVal = leftMin;
    if (maxMin > rightMin)
        minVal = rightMin;

    return minVal;
}
```



valid

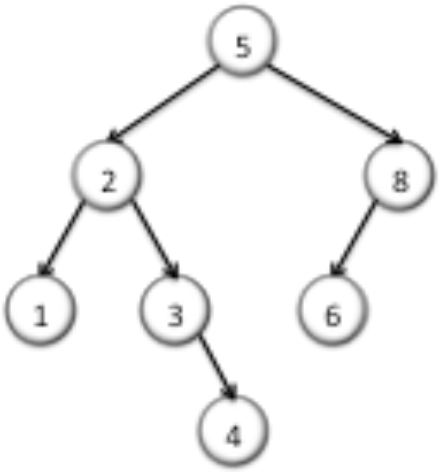
```
bool valid(const Node *node)
{









}
```



valid

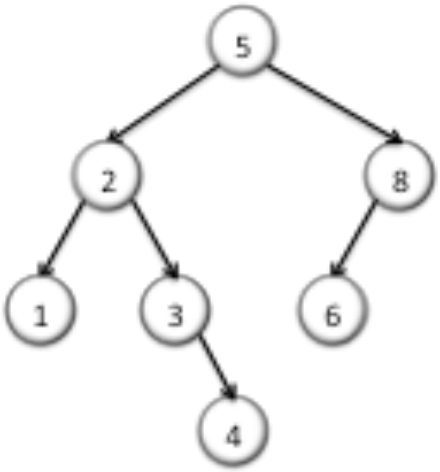
```
bool valid(const Node *node)
{
    if (node == NULL)
        return true;

    if (node->left != NULL && findMax(node->left) > node->val)
        return false;

    if (node->right != NULL && findMin(node->right) < node->val)
        return false;

    return valid(node->left) && valid(node->right);
}
```

treeHeight




```
int treeHeight(const Node *node)
{









}
```

treeHeight




```
int treeHeight(const Node *node)
{
    if (node == NULL)
        return -1;

    int leftHeight = treeHeight(node->left);
    int rightHeight = treeHeight(node->right);

    if (leftHeight > rightHeight)
        return leftHeight + 1;
    else
        return rightHeight + 1;
}
```