Binary Search Tree

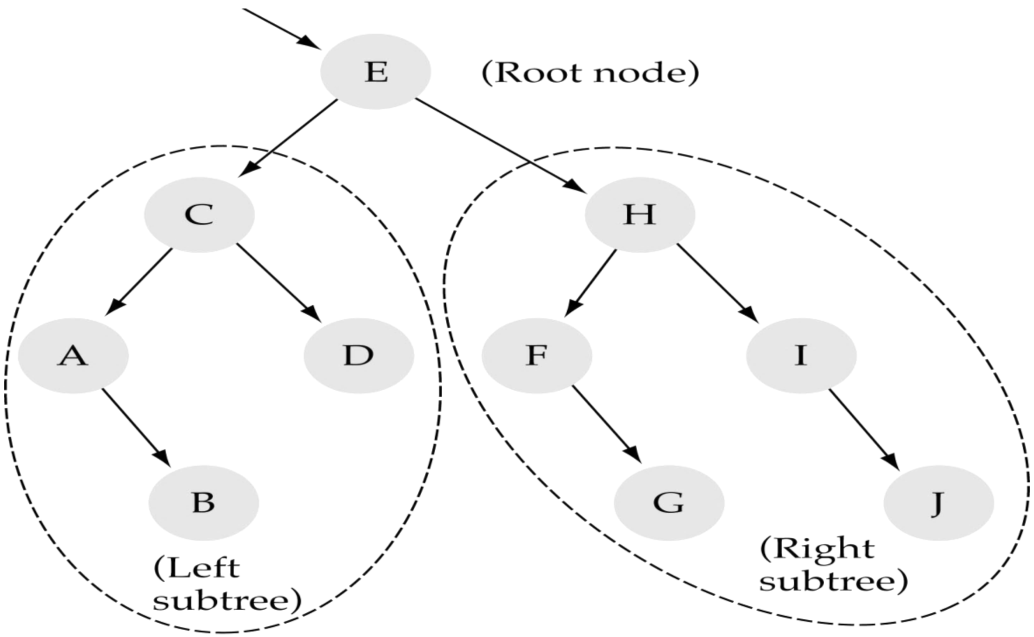
Lecturer: **LeNga**

CONTENTS

- The concept of BST
- Representation
- Operations

The concept of BST

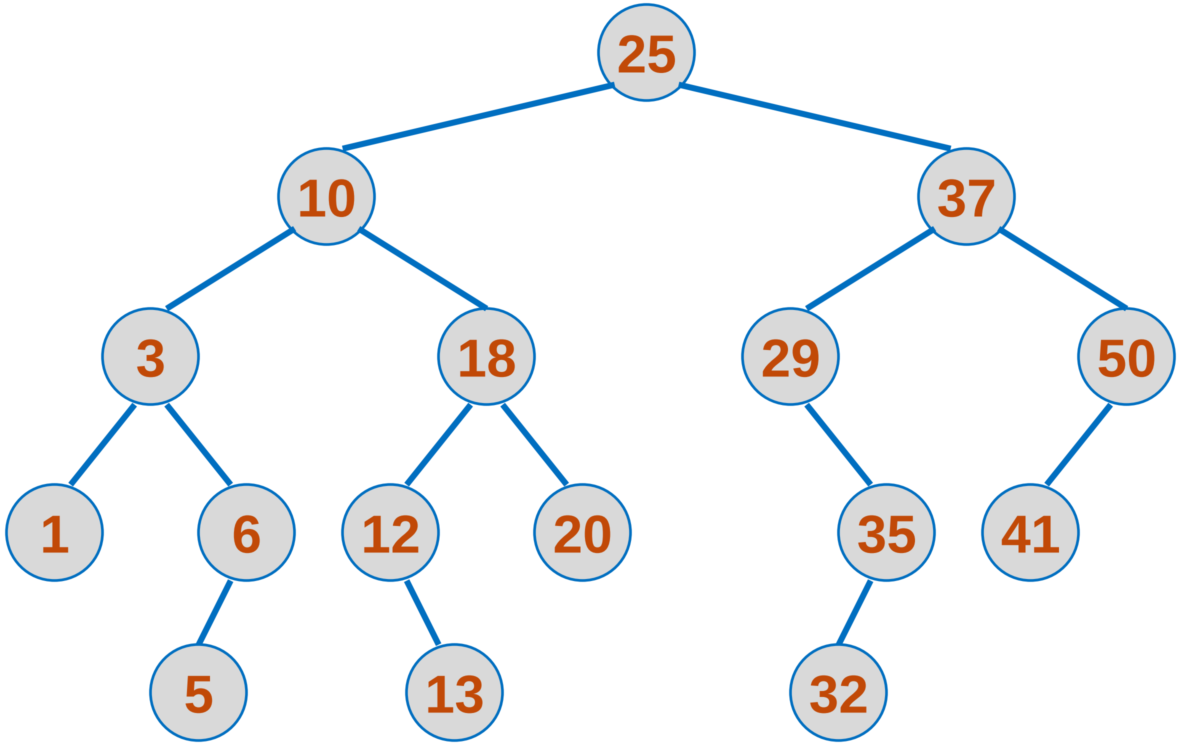
- **A binary search tree**
  - Is a binary tree (may be empty)
  - Every node must contain an identifier.
  - An identifier of any node in the left subtree is less than the identifier of the root.
  - An identifier of any node in the right subtree is greater than the identifier of the root.
  - Both the left subtree and right subtree are binary search trees.



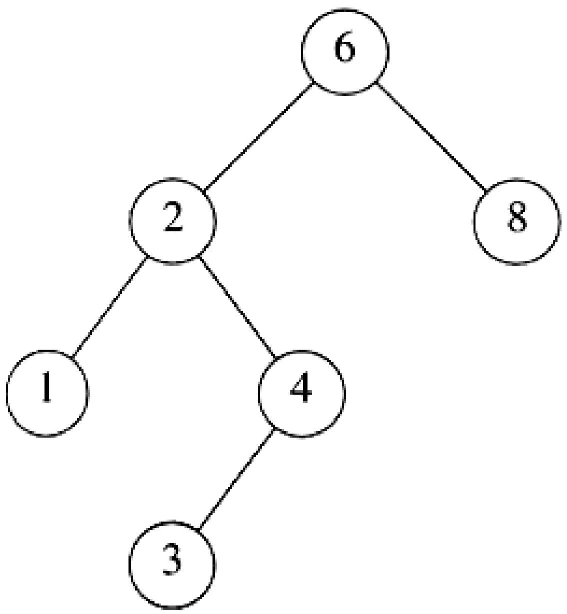
…The concept of BST

25 37 10 18 29 50 3 1 6 5 12 20 35 13 32 41

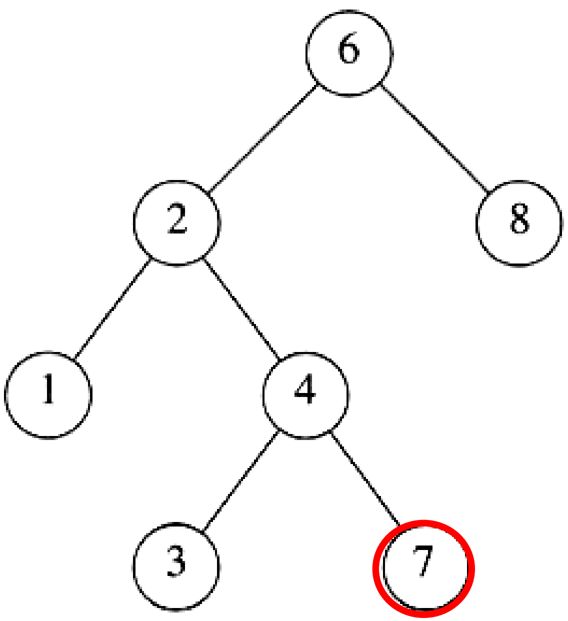
25	37	10	18	29	50	3	1	6	5	12	20	35	13	32	41



...The concept of BST



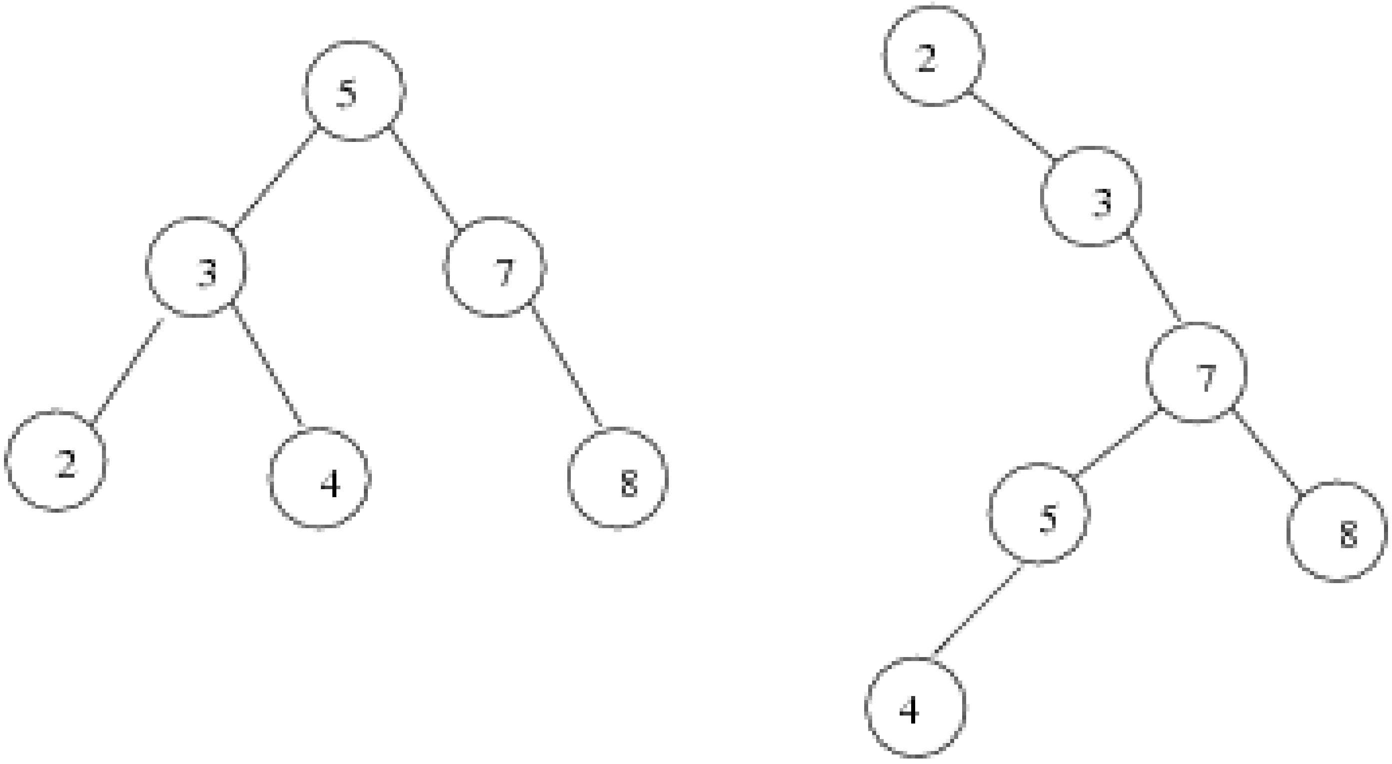
A binary search tree



Not a binary search tree

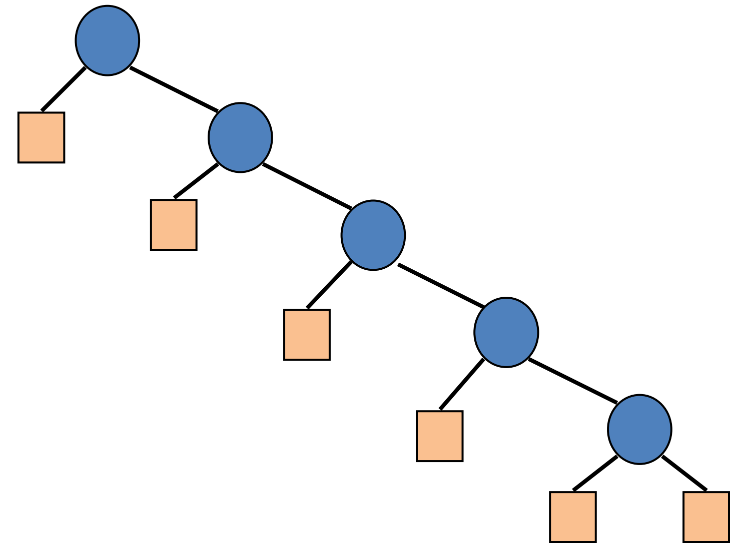
Representation

Two binary search trees representing the same set: Why?

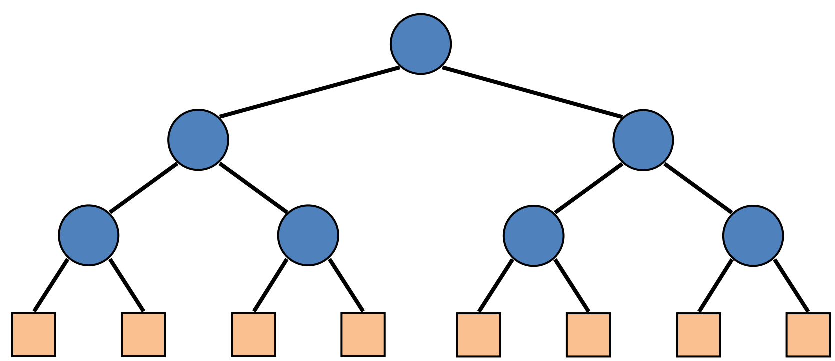


…Representation

- Consider a dictionary with n items implemented by means of a binary search tree of height h
  - the space used is $O(n)$
  - methods find, insert and remove take $O(h)$ time



- $O(n)$ in the worst case
- $O(\log n)$ in the best case



…Representation

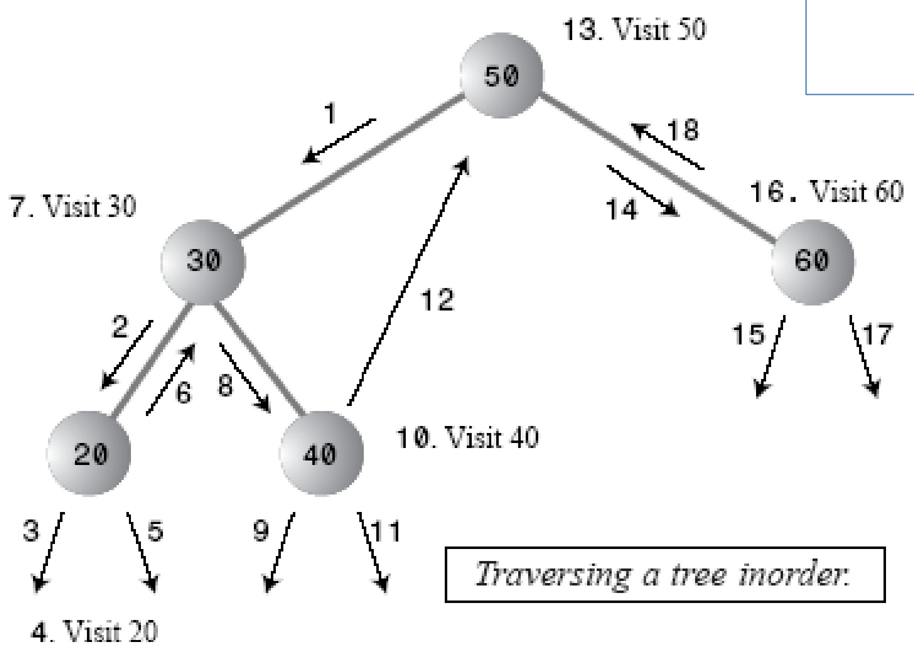
- Why using binary search tree
  - traverse in inorder: sorted list
  - searching becomes faster
- But..
  - Insert, delete: slow
- Important thing: Index in Database system
  - Using the right way of Index property

Operations

- Traverse node
- Search node
- Insert node
- Delete node
- Create Tree
- Delete Tree

Traverse node

```
void  inOrder(TREE  root){
      if (root!=NULL) {
          inOrder(root ->left);
          cout<< root ->data <<" ";
          inOrder(root ->righ);
      }
}
```

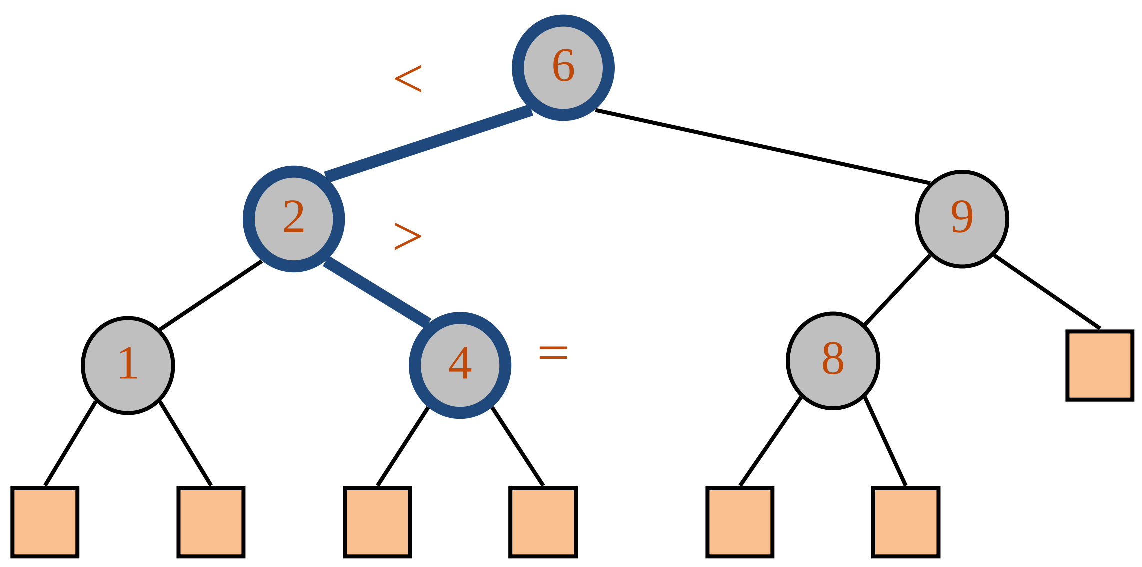


Traversing a tree inorder.

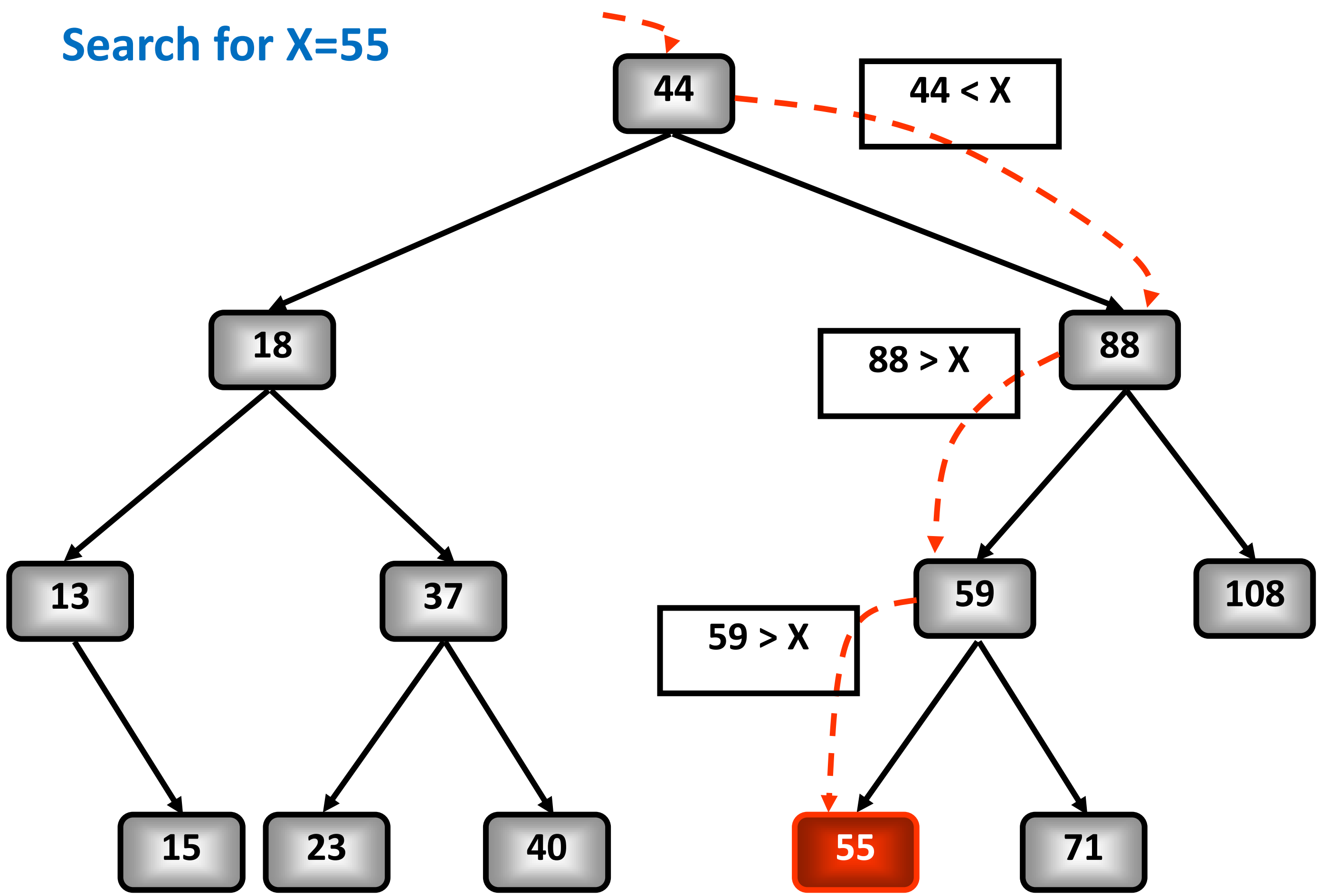
Search node

- 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 value 4
  - Call   Search(4,root)



Search node



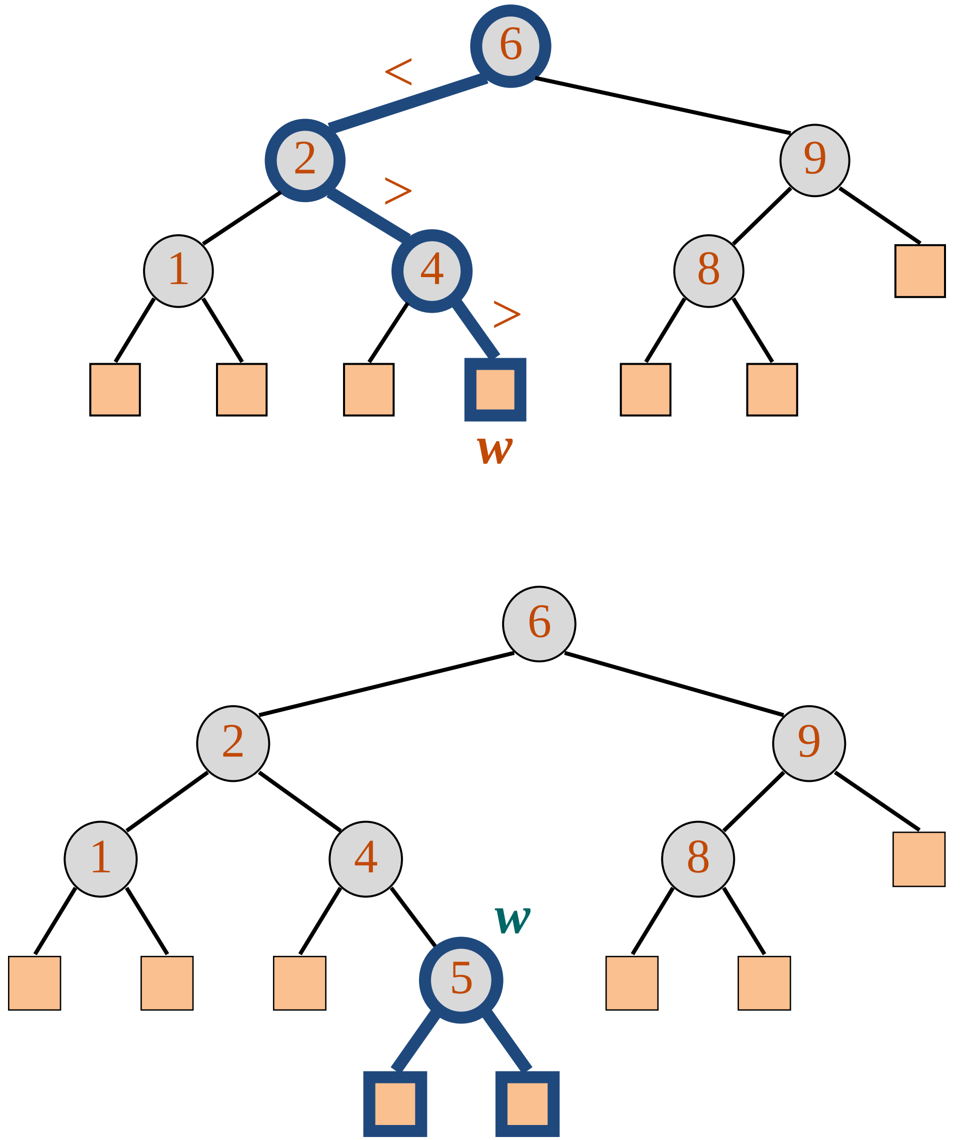
Search node

```
// recursion
node*  searchNode(int  x, TREE  root){
    if ((root!=NULL) || (root ->data==x))       return root;
    else if (x < root ->data)  return searchNode (x, root ->left);
    else if (x > root ->data)  return searchNode (x, root ->right);
}
```

```
// loop
node*  searchNode (int  x, TREE  root){
        node *p = root;
        while ((x != p->data) && (p!=NULL)){
            if (x < p->data)  p = p->left;
            else              p = p->right;
        }
        return p;
}
```

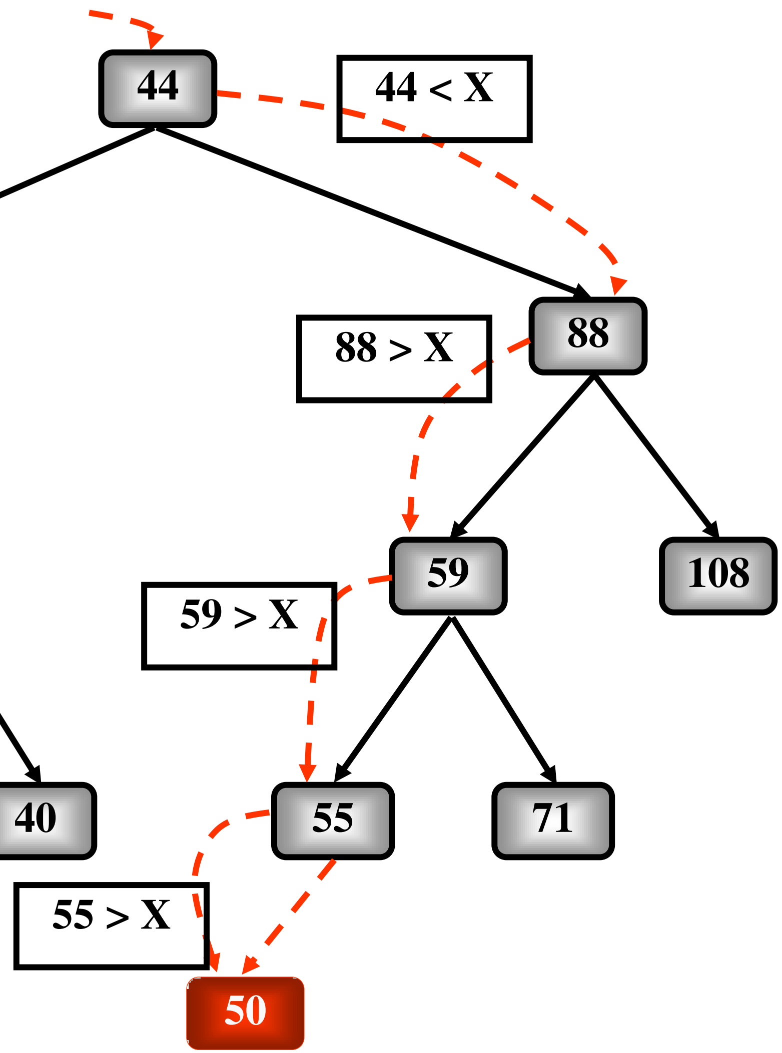
Insert node

- To perform operation insert(k, root), for a key *k*
- Assume k is not already in the tree, and let **w** be the leaf reached by the search
- We insert k at node **w** and expand **w** into an internal node
- Example: Insert (5,root)



Insert node

insert X=50



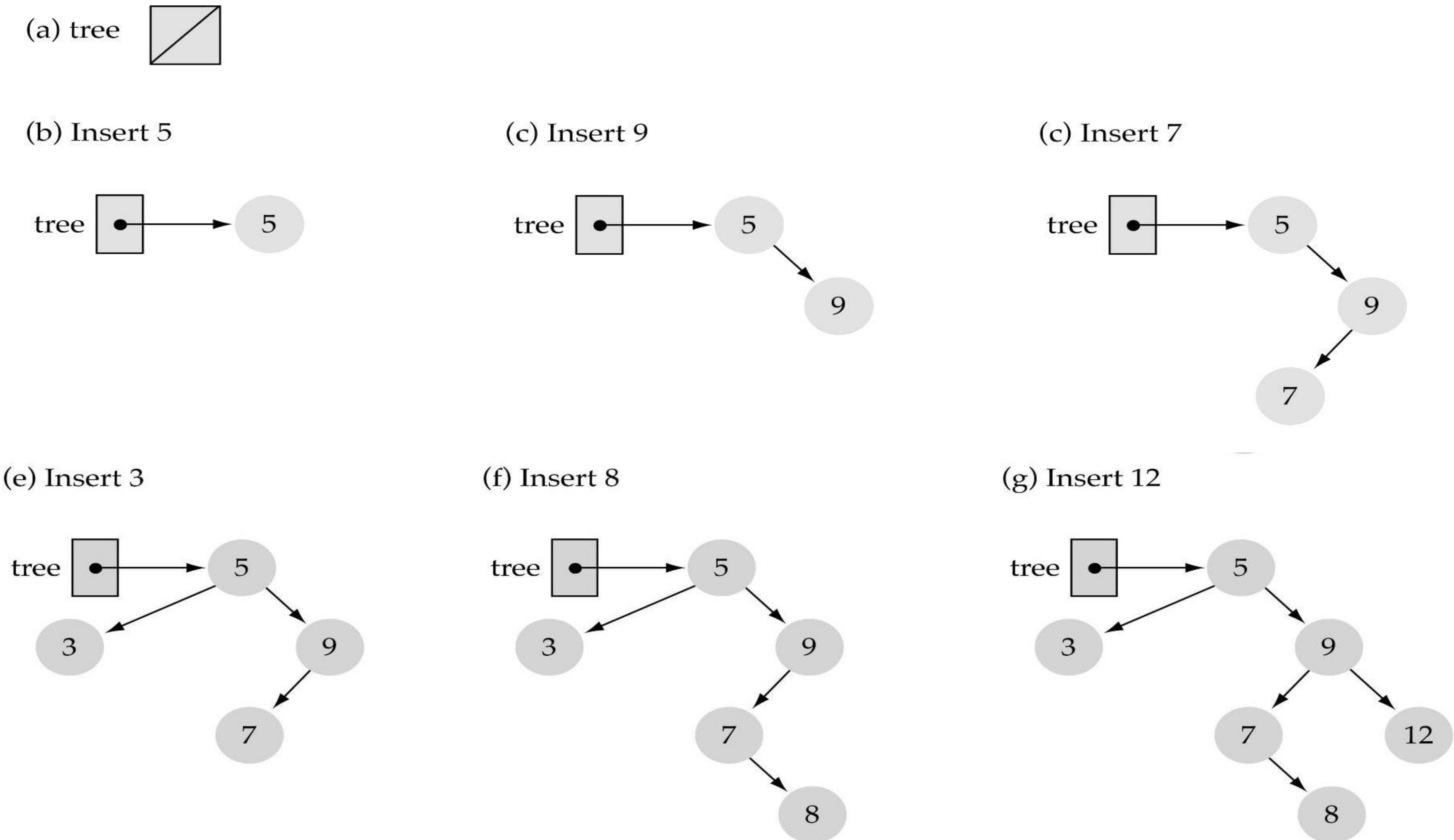
Insert node

```
void InsertNode(TREE  root,   node *p){
   if (root==NULL)  root=p;
   else if  (root->data > p->data)    InsertNode(root->left,  p);
   else if  (root->data < p->data)    InsertNode(root->right, p);
}
```

Insert node

```
int insertNode(TREE root, int X)
{ if (root) {
      if(root->data == X) return 0; // đã có
      if(root->data >  X)
            return insertNode(root->left, X);
      else
            return insertNode(root->right, X);
  }
  root       = new Node;
  if (root == NULL) return -1; // thiếu bộ nhớ
  root->data    = X;
  root->left = root->right = NULL;
  return 1;   // thêm vào thành công
}
```

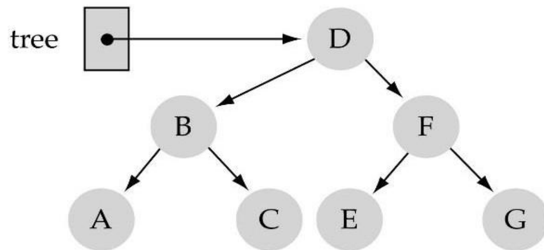
- **Insert node**



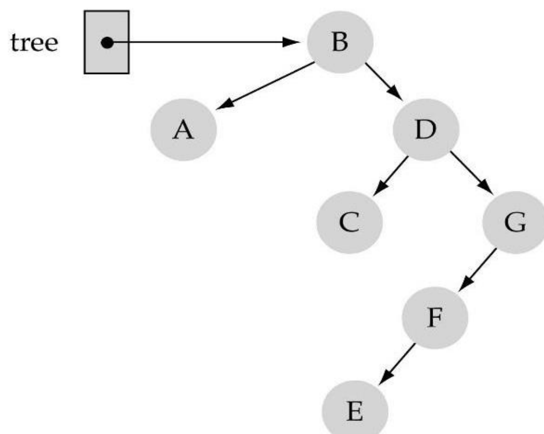
(a) tree
(b) Insert 5
tree
5
(c) Insert 9
tree
5
9
(c) Insert 7
tree
5
9
7
(e) Insert 3
tree
5
3
9
7
(f) Insert 8
tree
5
3
9
7
8
(g) Insert 12
tree
5
3
9
7
12
8

…Operations

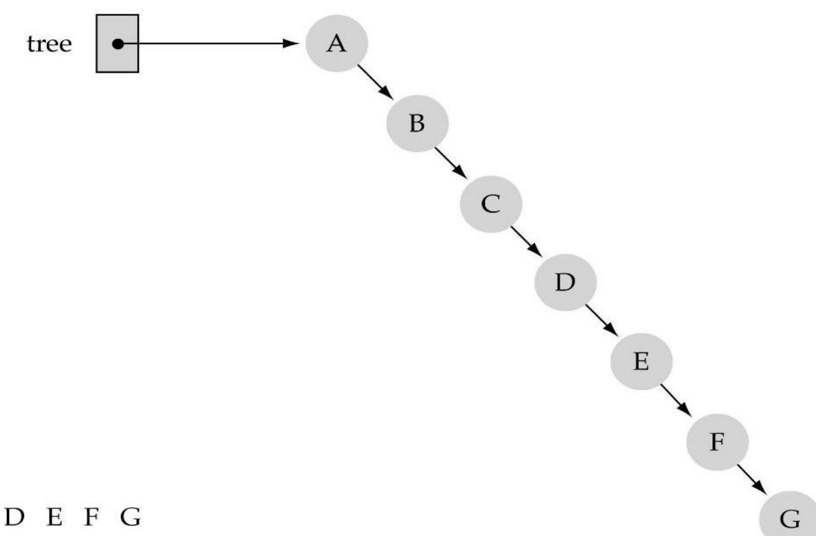
Insert Order



(a) Input: D B F A C E G



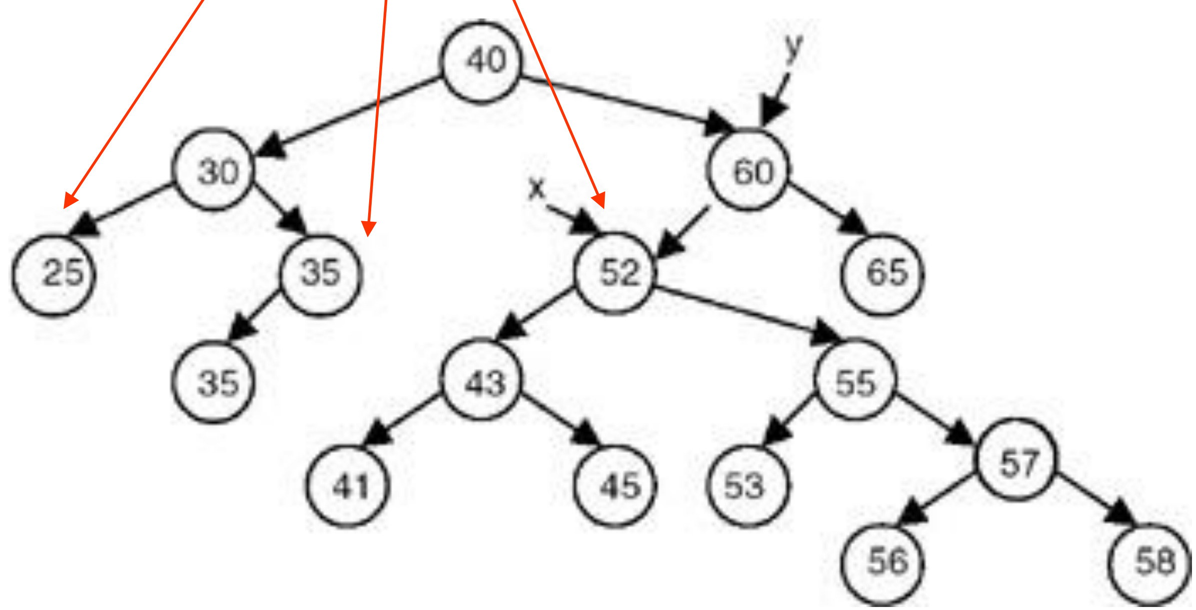
(b) Input: B A D C G F E



(c) Input: A B C D E F G

Delete node

- Divide into 3 cases
  - Delete a node with No Child (node is a leaf)
  - Delete a node with one Child (node has only one child)
  - Delete a node with two Children (node has two children)



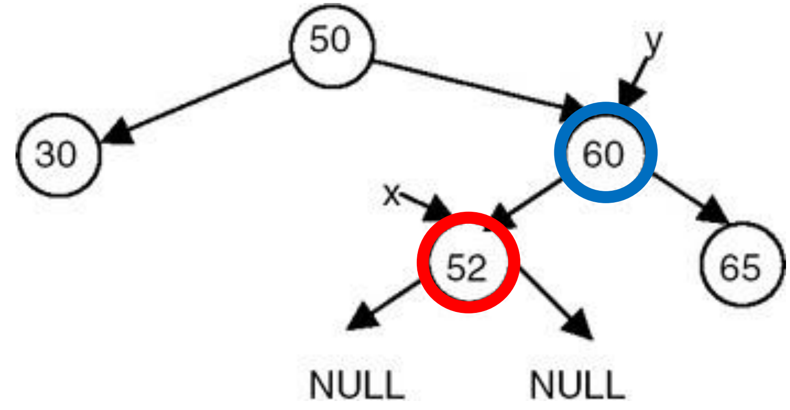
Delete a node with No Child

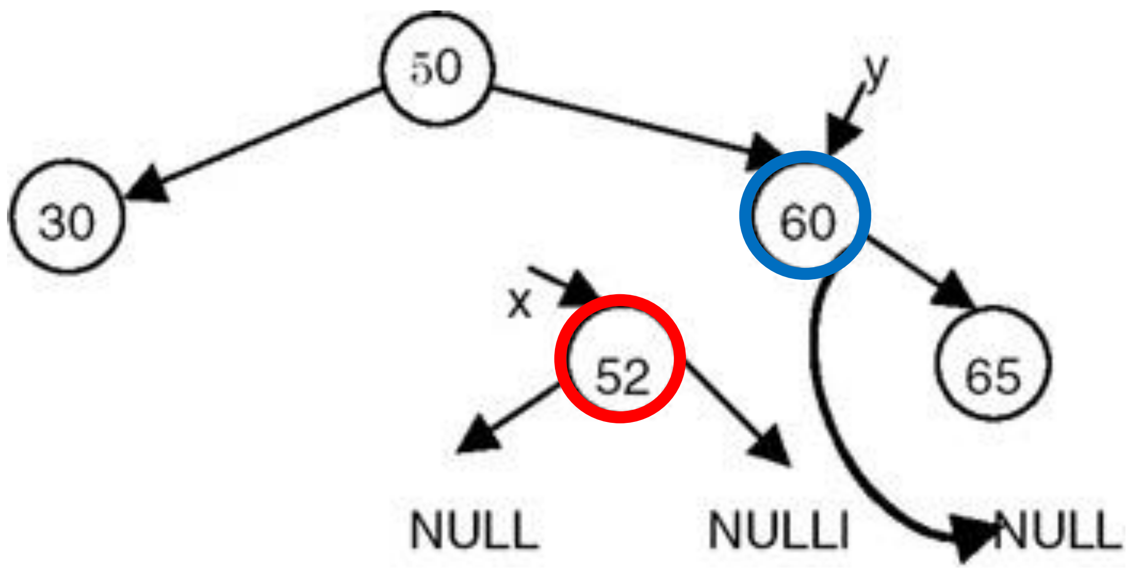
- set the left of **y** to NULL:

  y->left = NUll

- delete the node pointed to by **x** :

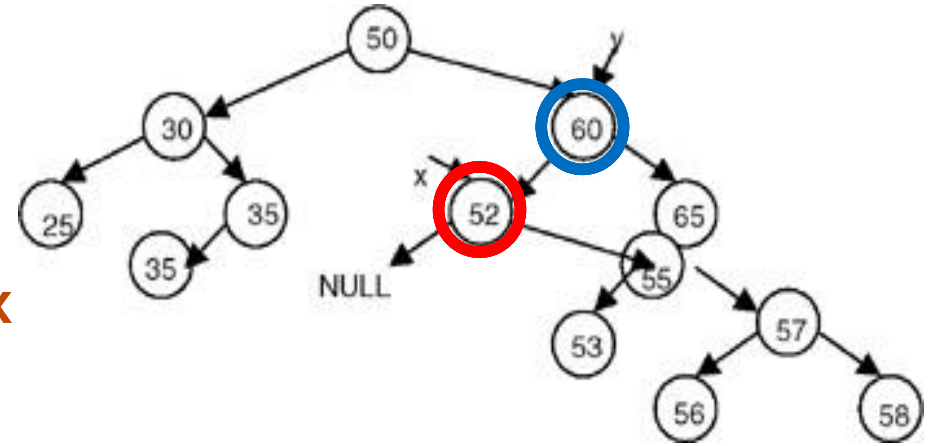
  delete x

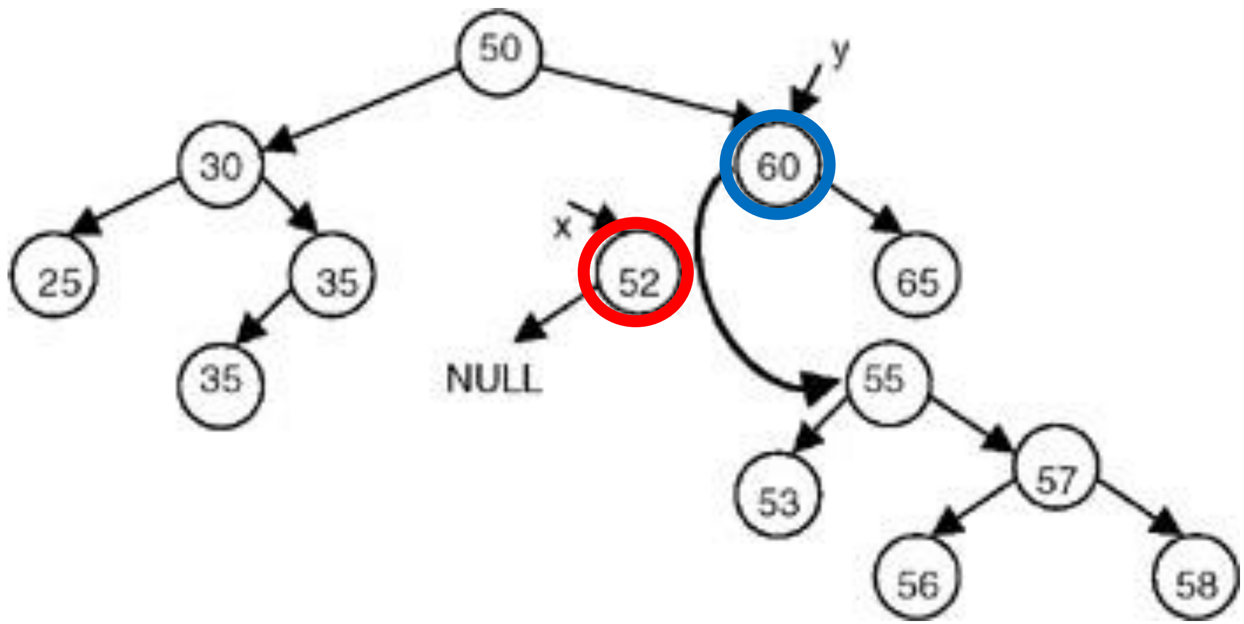




Delete a node with One Child

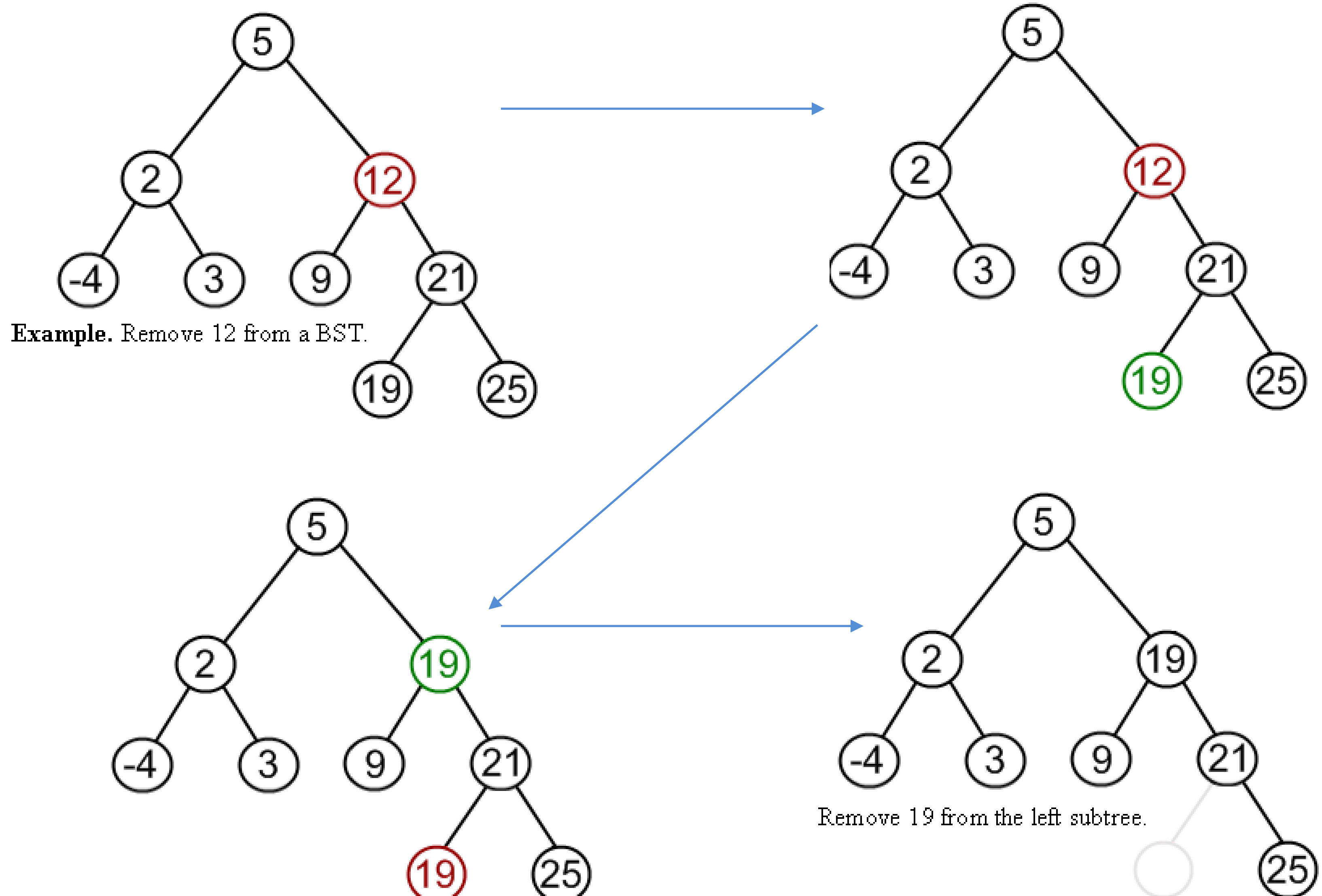
- make **y->left** = **x->right**
- delete the node pointed to **x**





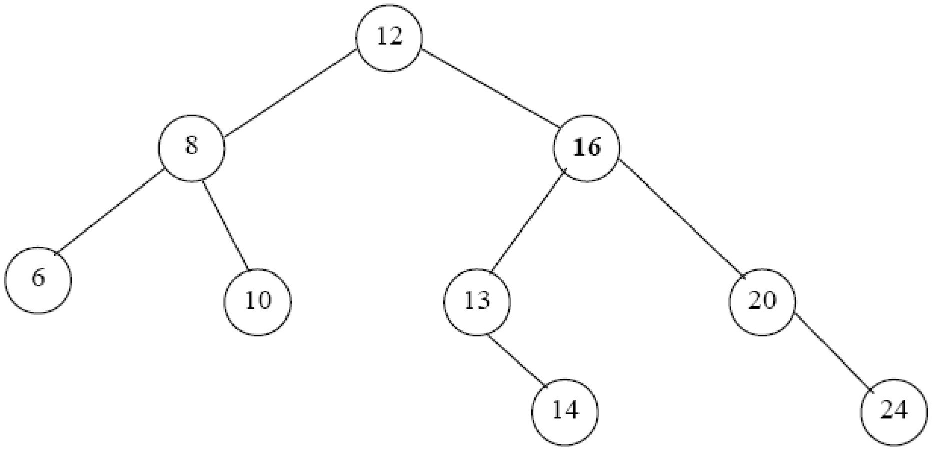
...Operations

Delete a node with Two Children

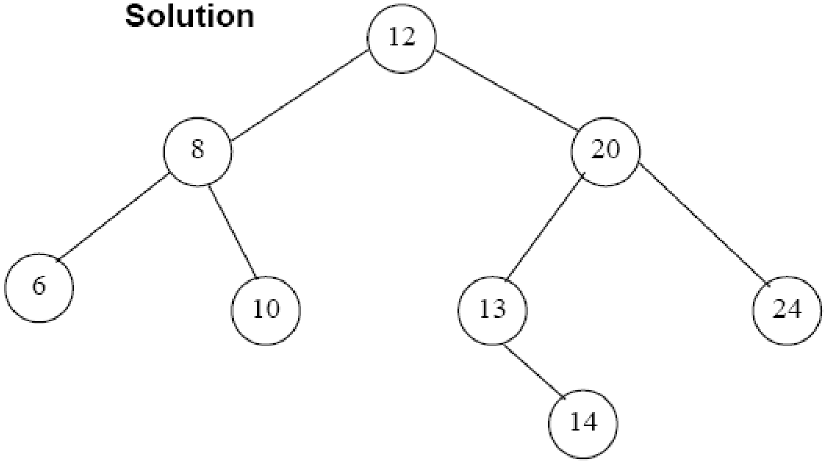


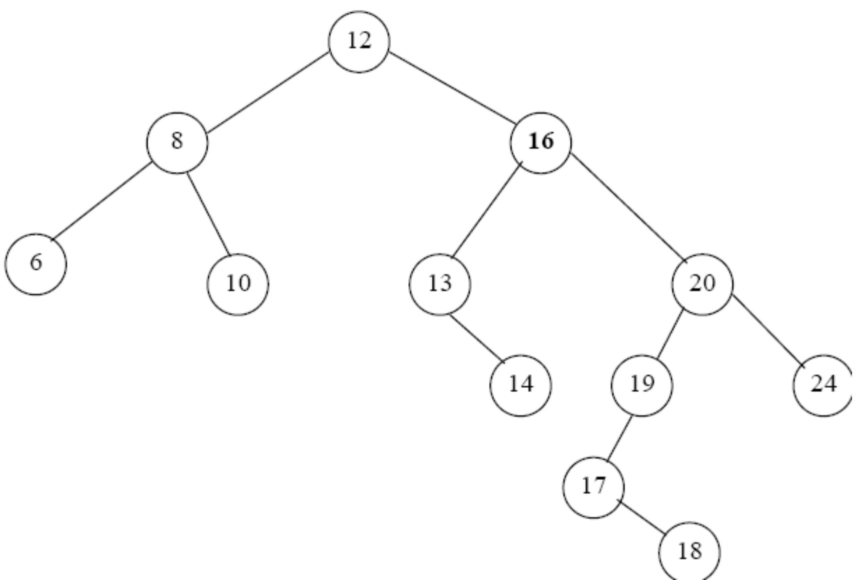
...Operations

Delete a node with Two Children - Ex: remove 16

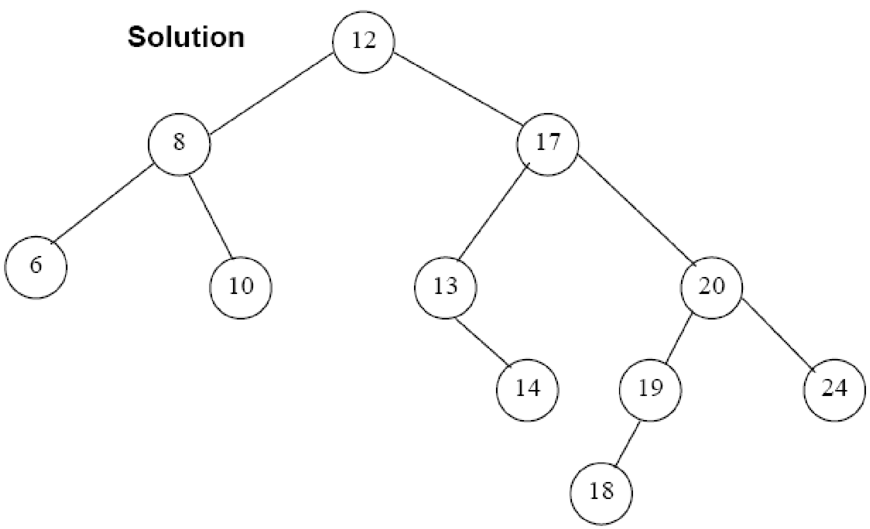


Solution





Solution

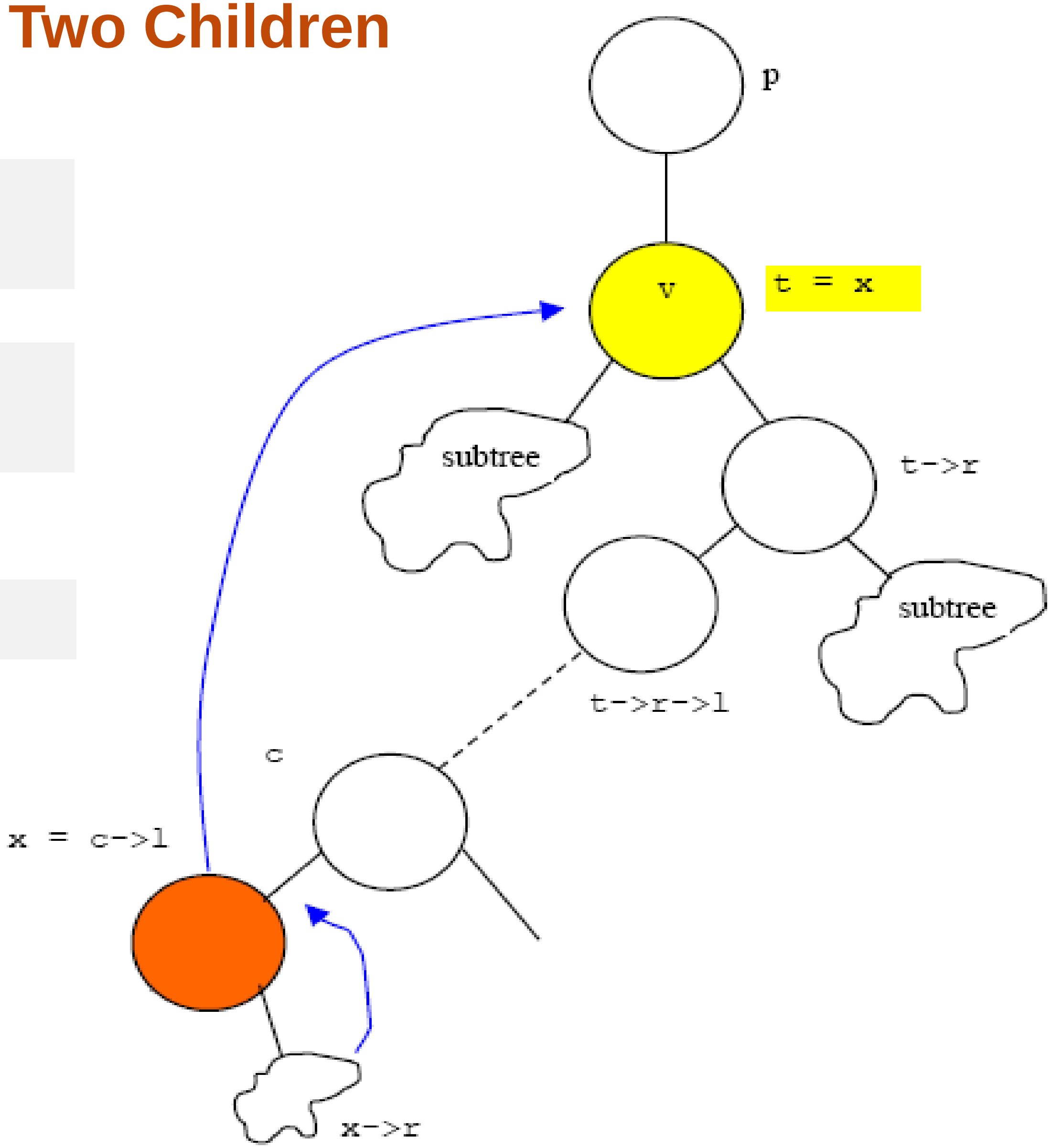


Delete a node with Two Children

rightmost child of the subtree of the left

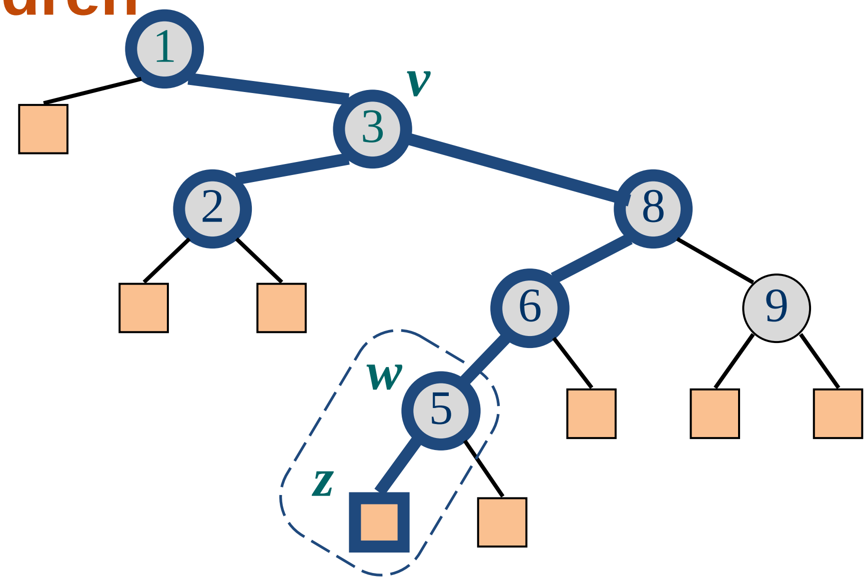
leftmost child of the subtree of the right

WHY???

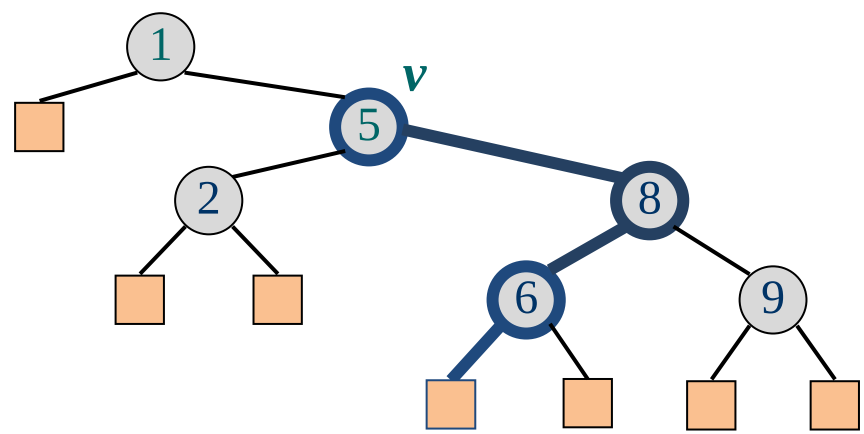


Delete a node with Two Children

- consider the case where the key k to be removed is stored at a node v whose children are both internal
  - find the internal node w that follows v in an inorder traversal
  - copy $\mathbf{key}(w)$ into node v
  - remove node w and its left child z (which must be a leaf) by means of operation Delete(z)

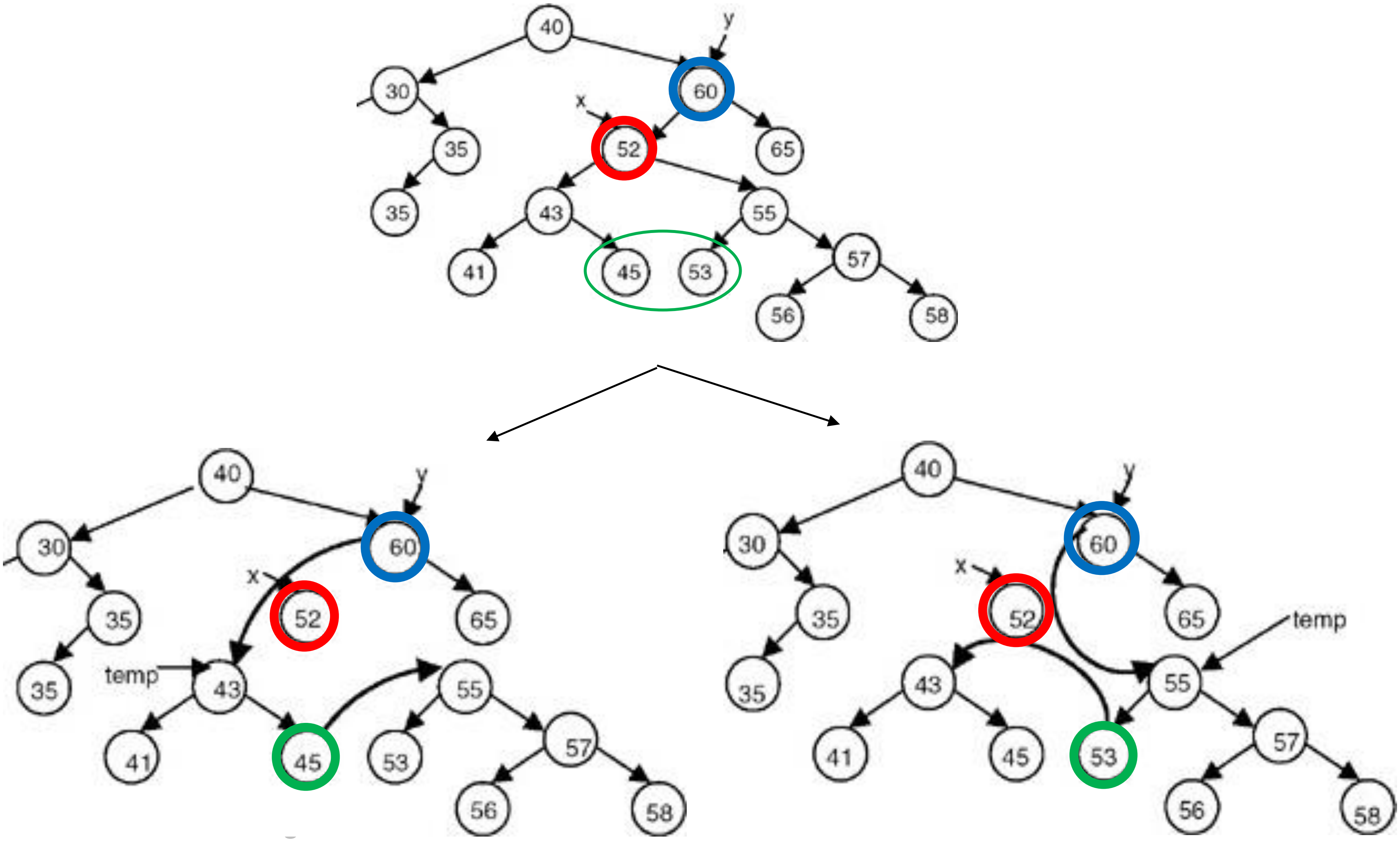


- Example: Delete 3



…Operations

Delete a node with Two Children



Delete node

```
void deleteNode(node* p) {
    if (p->left == NULL) {
            node *temp = p;             p = p->right;       delete temp;   }
   else if (p->right == NULL) {
            node *temp = p;             p = p->left;        delete temp;    }
   else {
            // In-order predecessor (rightmost child of left subtree)
            node *temp = p->left;       node *parent = p;
            // find the rightmost child of the subtree of the left node
            while (temp->right != NULL) {
                    parent = temp;              temp = temp->right;      }
            // copy value from the in-order predecessor to the original node
            p->value = temp->value;
            // now delete the "swapped" node value and unlink
            if(parent->left == temp)    deleteNode (parent->left);
            else                        deleteNode (parent->right);
    }
}
```

Create Tree

```
void createTree(TREE root){
        int x,n;
        scanf("%d",&n);
        for(int i=1; i<=n;i++){
                scanf("%d",&x);
                insertNode(root,x);
        }
}
```

Delete Tree

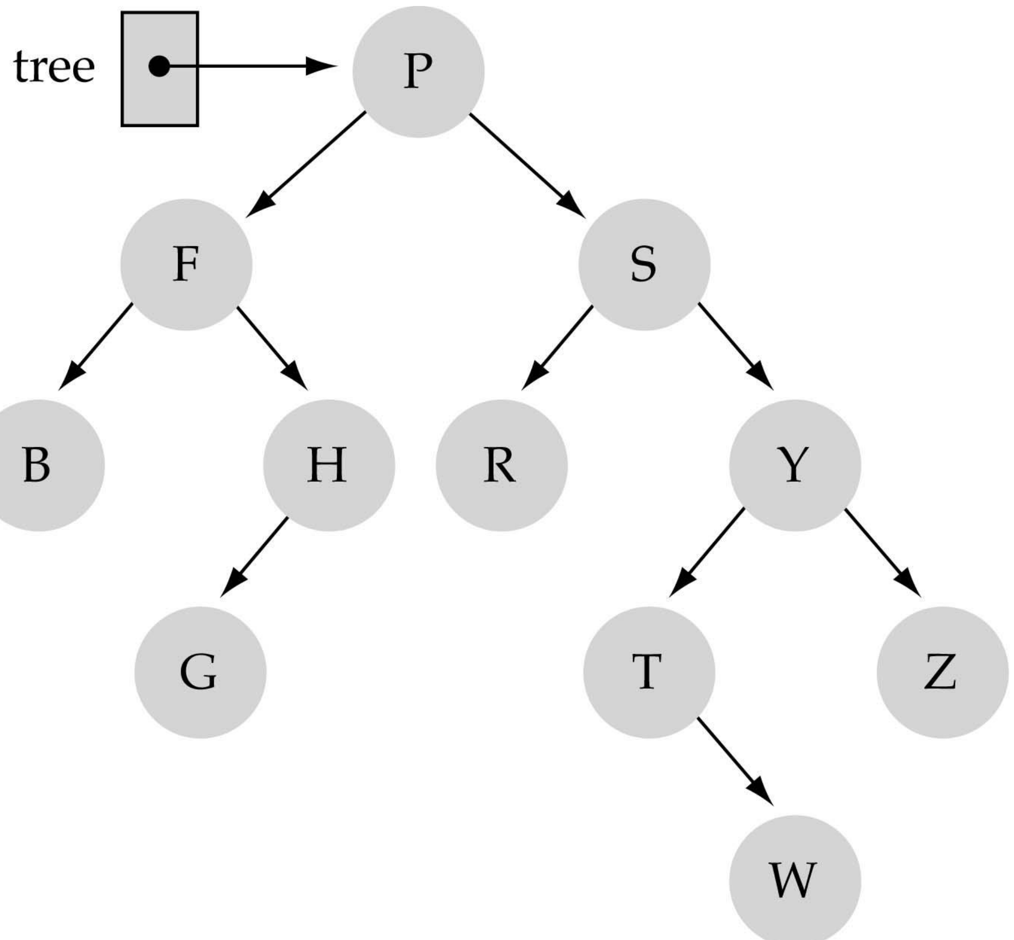
```
void deleteTree(TREE root){
    if  (root)          {
            deleteTree(root->reft);
            deleteTree(root->right);
            delete(root);
    }
}
```

Exercise 1

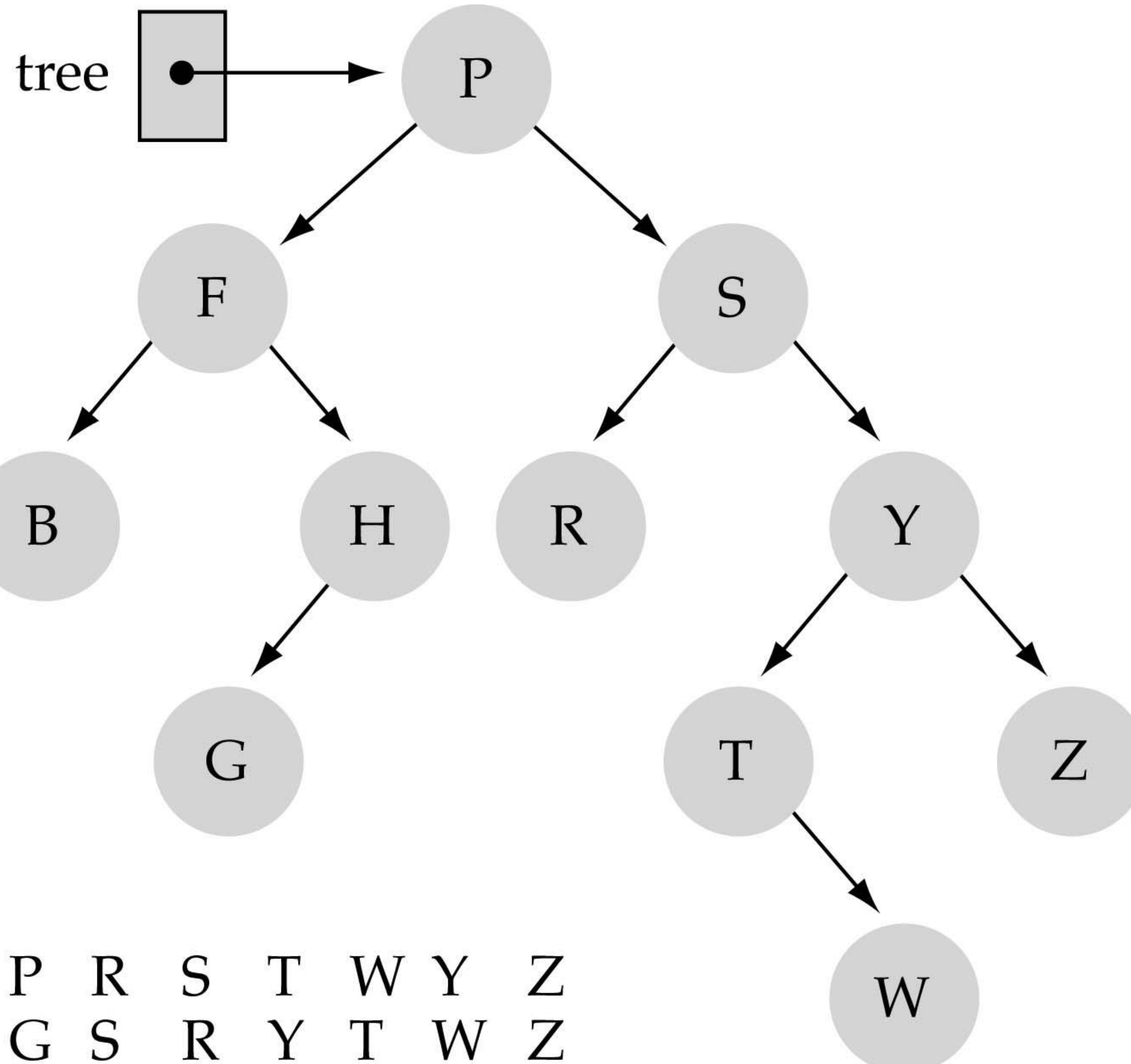
- Build Binary Search Tree from list
  - 10 4 7 12 16 20 30 5 2 26 15
  - 24 12 89 4 32 50 10 6 36 79 5 9 11

Exercise 2

- Order of: inoder, postorder, preorder of



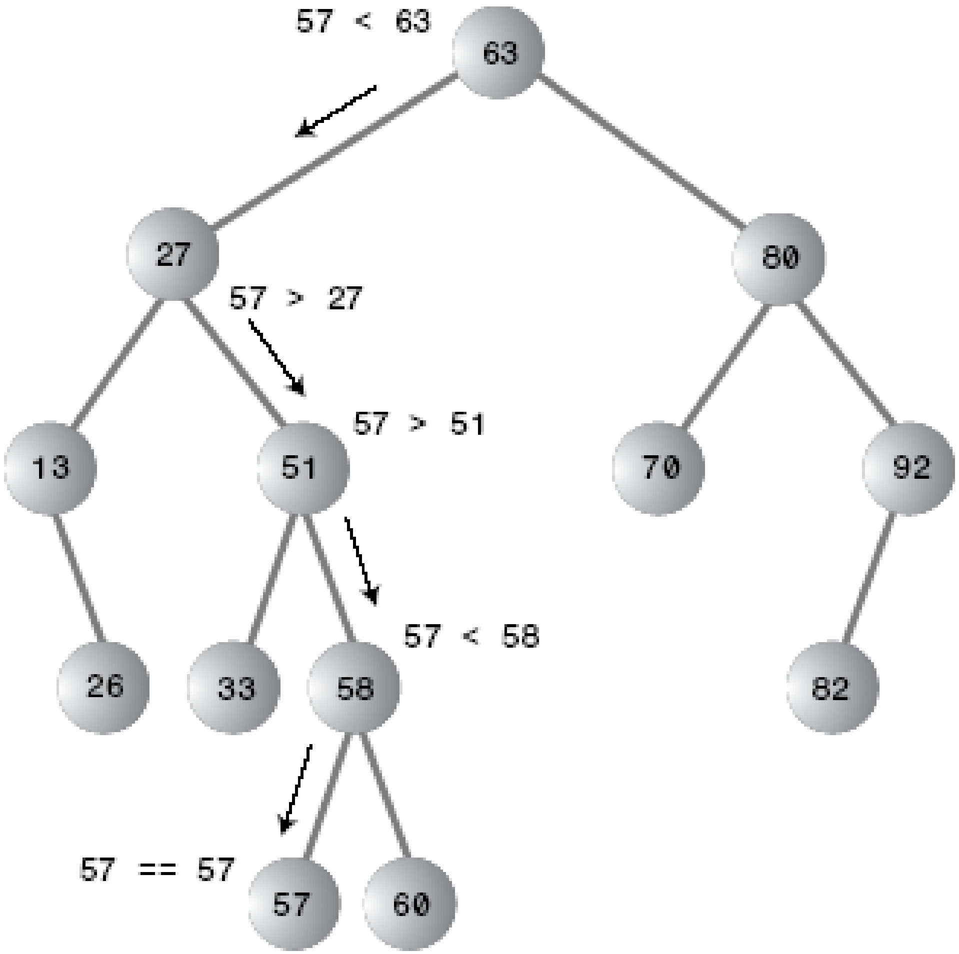
Exercise 2



Inorder:	B	F	G	H	P	R	S	T	W	Y	Z
Preorder:	P	F	B	H	G	S	R	Y	T	W	Z
Postorder:	B	G	H	F	R	W	T	Z	Y	S	P

Exercise 3

- Order of: inoder, postorder, preorder of



Exercise 4

- Count
  - Even/Odd
  - Leaf
- Sum
  - Even/Odd
- Height

Counting the number of nodes

- int count(struct tnode *p) {

```
    if( p == NULL)      return(0);
    else        return(1 + (count(p->lchild) + count(p->rchild)));

}
```

Counting the number of even (odd) nodes

```
int counteven(node* r){
   if (r!=NULL)
        if (r->data%2==0)   return   1+ counteven(r->l)+counteven(r->r);
        else                return      counteven(r->l)+counteven(r->r);
   else       return 0;
}
```

Sum of all nodes

- int sum(node *p) {

```
if( p == NULL)   return(0);
else      return ( p->data+sum(p->l)+sum(p->r) );
}
```

Sum of even (odd) nodes

```
int   counteven(node* r) {
  if (r!=NULL)
      if (r->data%2==0)  ?????????
      else               ?????????
  else            return 0;
}
```

Count number of leaf nodes

```
int countleaf(node* r){
   if (r!=NULL)
            if (r->l==NULL && r->r==NULL)  return   1;
            else      return   countleaf(r->l)+countleaf(r->r);
   else   return 0;
}
```

Count number of node had 1 child

```
int countleaf(node* r){
   if (r!=NULL)
         if (????????????)        return   1;
         else      return   countleaf(r->l)+countleaf(r->r);
   else   return 0;
}
```

Count number of node had 2 child

```
int countleaf(node* r){
   if (r!=NULL)
         if (???????????)        return   1;
         else      return   countleaf(r->l)+countleaf(r->r);
   else   return 0;
}
```

Find height of tree

- ```
  int Height (node* n){
          if(n==NULL)    return 0;
          else     return 1+max(Height (n->l)),
          Height (n->r));
  }
  ```
  ```

Thank You…!