Data Structures and Algorithms

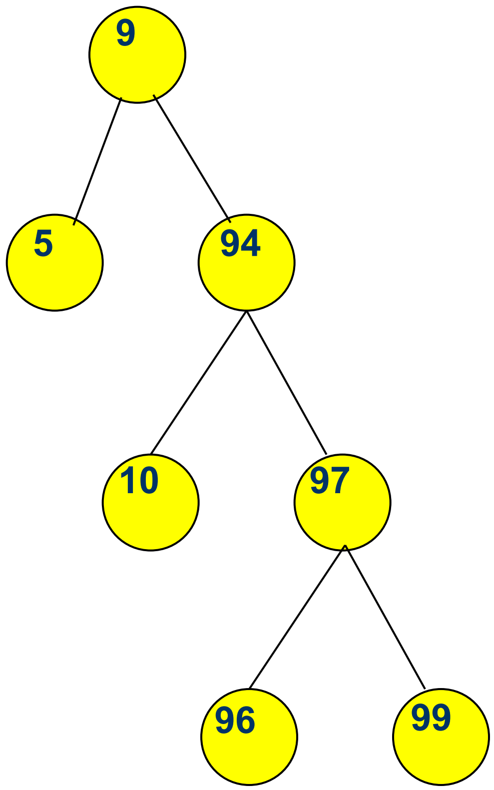
Lecture 9 – Binary Search Tree

Miao Zhang

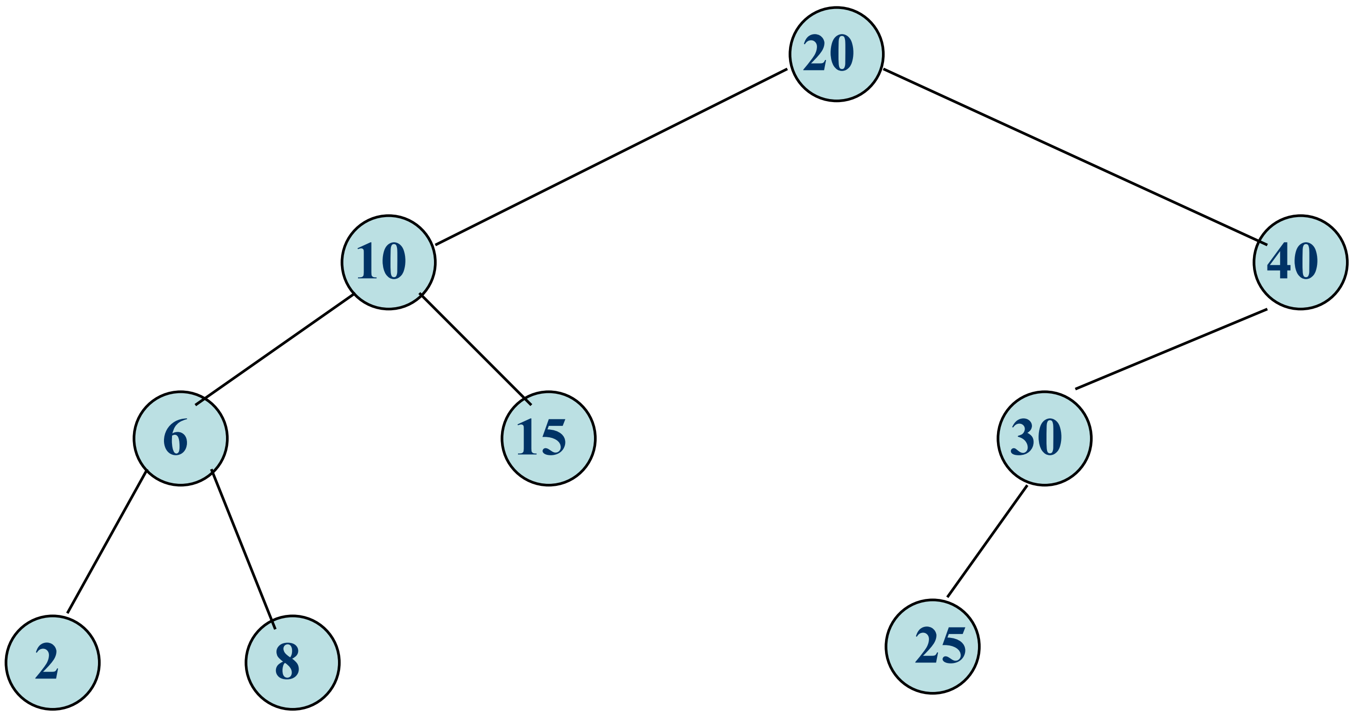
Binary Search Trees

- Binary search trees are binary trees in which
  - all values in the node's left subtree are less than node value
  - all values in the node's right subtree are greater than or equal to node value
- Operations:
  - Find, FindMin, FindMax, Insert, Delete

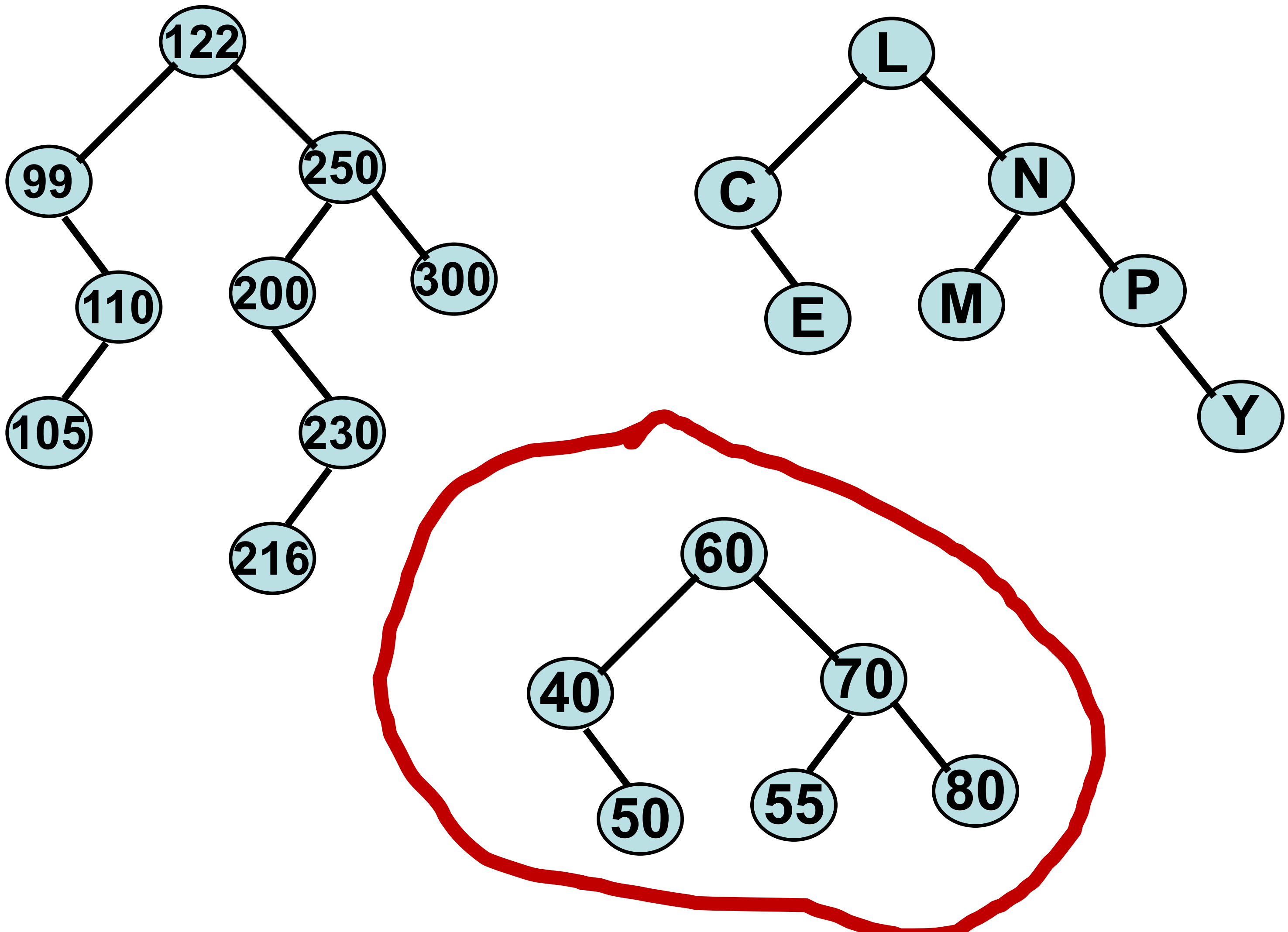
What happens when we traverse the tree in inorder?



Example Binary Search Tree



Are all binary search tree?

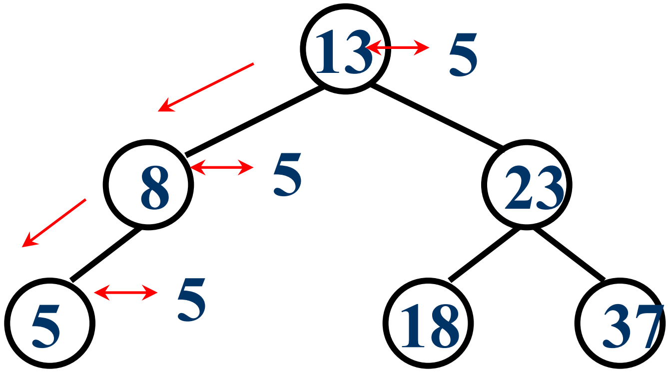


Binary Search Tree

- Search/ Find
- Insert
- Remove

Search

How to find 5 ?



Success !

The BinaryNode class

```
template <class Comparable>
class BinaryNode
{
   Comparable element;   // this is the item stored in the node
   BinaryNode *left;
   BinaryNode *right;

   BinaryNode( const Comparable & theElement, BinaryNode *lt,
     BinaryNode *rt ) : element( theElement ), left( lt ),
      right( rt ) { }
};
```

Find

```
/**
 * Method to find an item in a subtree.
 * x is item to search for.
 * t is the node that roots the tree.
 * Return node containing the matched item.
 */
template <class Comparable>
BinaryNode<Comparable> *
find( const Comparable & x, BinaryNode<Comparable> *t ) const
{
  if( t == NULL )
      return NULL;
  else if( x < t->element )
      return find( x, t->left );
  else if( t->element < x )
      return find( x, t->right );
  else
      return t;    // Match
}
```

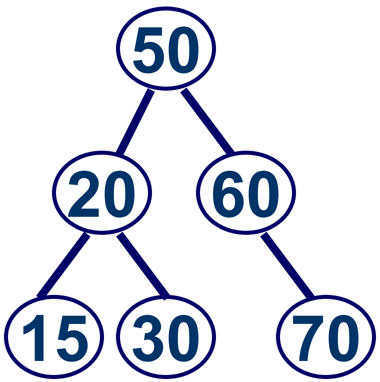
findMin (recursive implementation)

```
/**
 * method to find the smallest item in a subtree t.
 * Return node containing the smallest item.
 */
template <class Comparable>
BinaryNode<Comparable> *
findMin( BinaryNode<Comparable> *t ) const
{
   if( t == NULL )
       return NULL;
   if( t->left == NULL )
       return t;
   return findMin( t->left );
}
```

findMax (nonrecursive implementation)

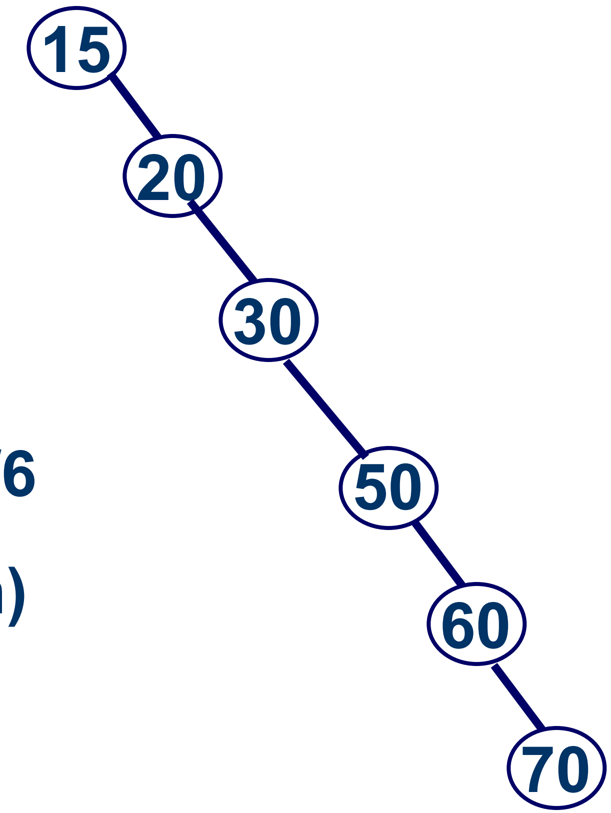
```
/**
 *method to find the largest item in a subtree t.
 *Return node containing the largest item.
 */
template <class Comparable>
BinaryNode<Comparable> *
findMax( BinaryNode<Comparable> *t ) const
{
  if( t != NULL )
    while( t->right != NULL )
       t = t->right;
  return t;
}
```

ASL Examples



ASL=(1+2+2+3+3+3)/6=14/6

(average searching length)



ASL=(1+2+3+4+5+6)/6=21/6

Insert

- First find the parent node of the new node.
- Call this node R.
- We then add a new node containing the new record as a child of R(left or right).
- If the binary tree is empty. The new node is the root.

Note that the new node is always inserted as a leaf node.

Insert operation

Algorithm for inserting X into tree T:

- **Proceed down the tree as you would with a find operation.**
- **if X is found**

  do nothing, (or "update" something)

  else

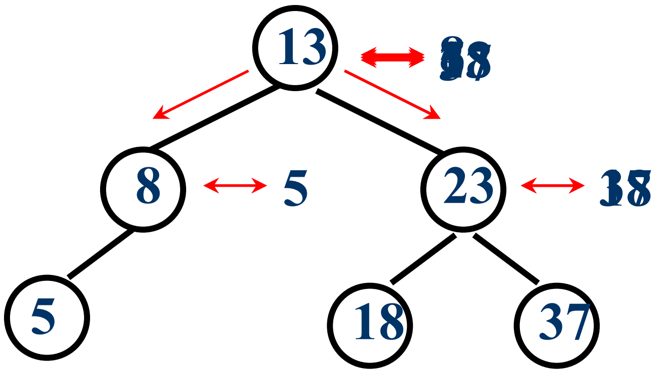
  insert X at the last spot on the path traversed.

Insert

We can use insert operation to construct a BST from the scratch

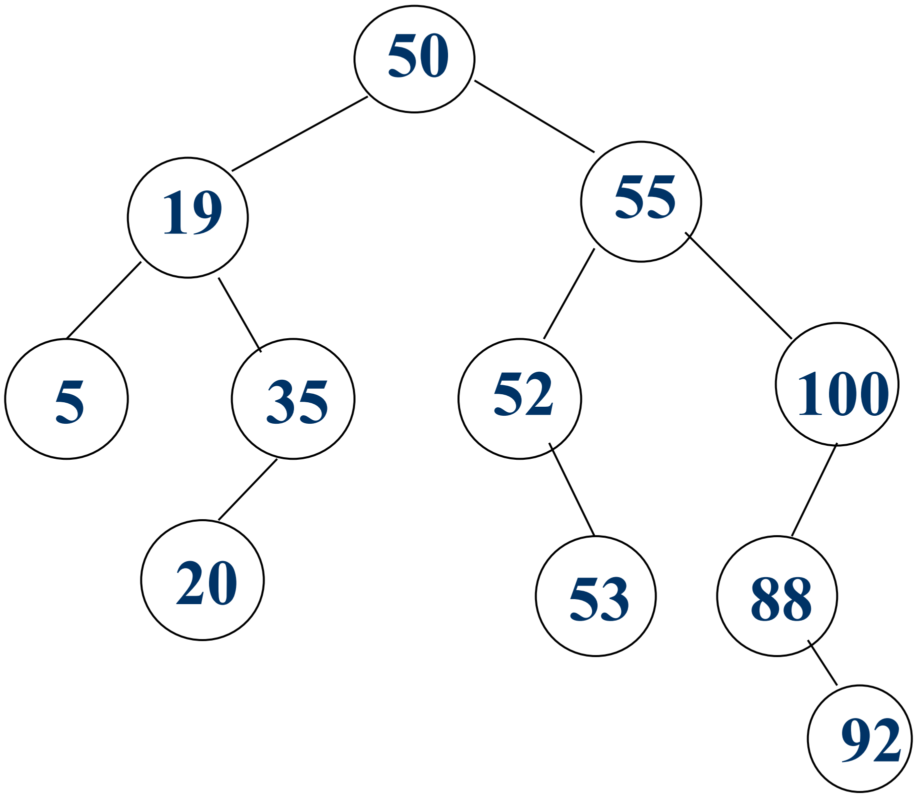
Input key values:

13、8、23、5、18、37



Another example

K = {50， 19， 35， 55， 20， 5， 100， 52， 88， 53， 92}



Insertion into a BST

```
/* method to insert into a subtree.
 * x is the item to insert.
 * t is the node that roots the tree.
 * Set the new root.
 */
template <class Comparable>
void insert( const Comparable & x,
             BinaryNode<Comparable> * & t ) const
{
   if( t == NULL )
      t = new BinaryNode<Comparable>( x, NULL, NULL );
   else if( x < t->element )
      insert( x, t->left );
   else if( t->element <= x )
      insert( x, t->right );
}
return t;
```

Deletion operation

There are three cases to consider:

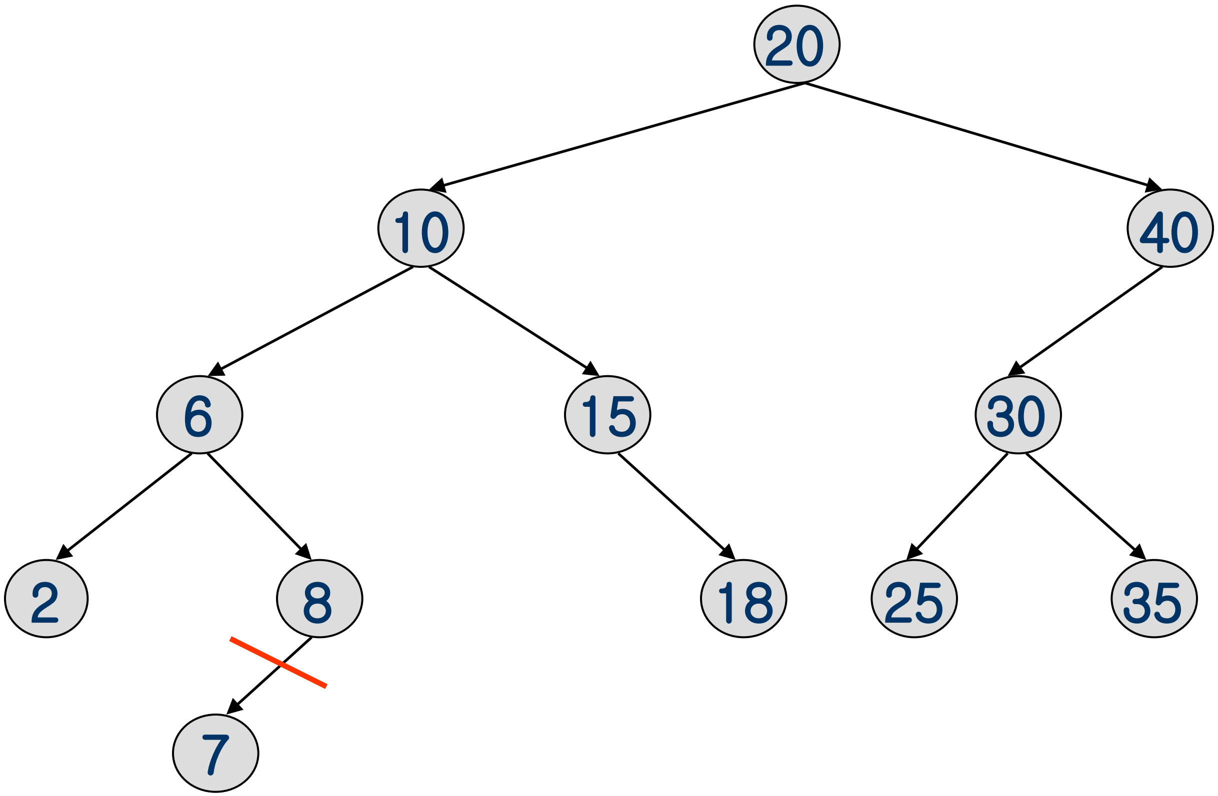
1. **Deleting a leaf node**
   - **Replace the link to the deleted node by NULL.**

2. **Deleting a node with one child:**
   - **The node can be deleted after its parent adjusts a link to bypass the node.**

3. **Deleting a node with two children:**
   - **The deleted value must be replaced by an existing value that is either one of the following:**
     - The largest value in the deleted node's left subtree
     - The smallest value in the deleted node's right subtree.

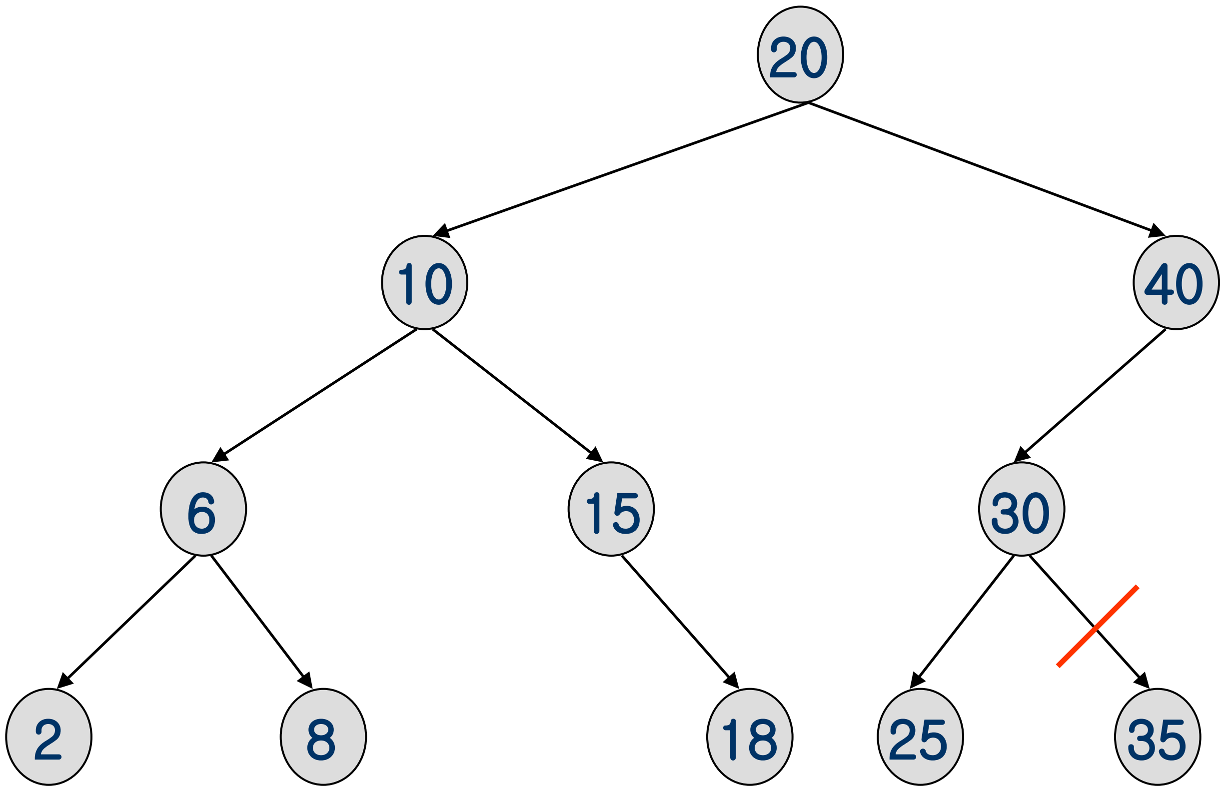
Case 1: A Leaf Node

- For case 1, we can simply discard the leaf node.
- Example, delete a leaf element. key=7



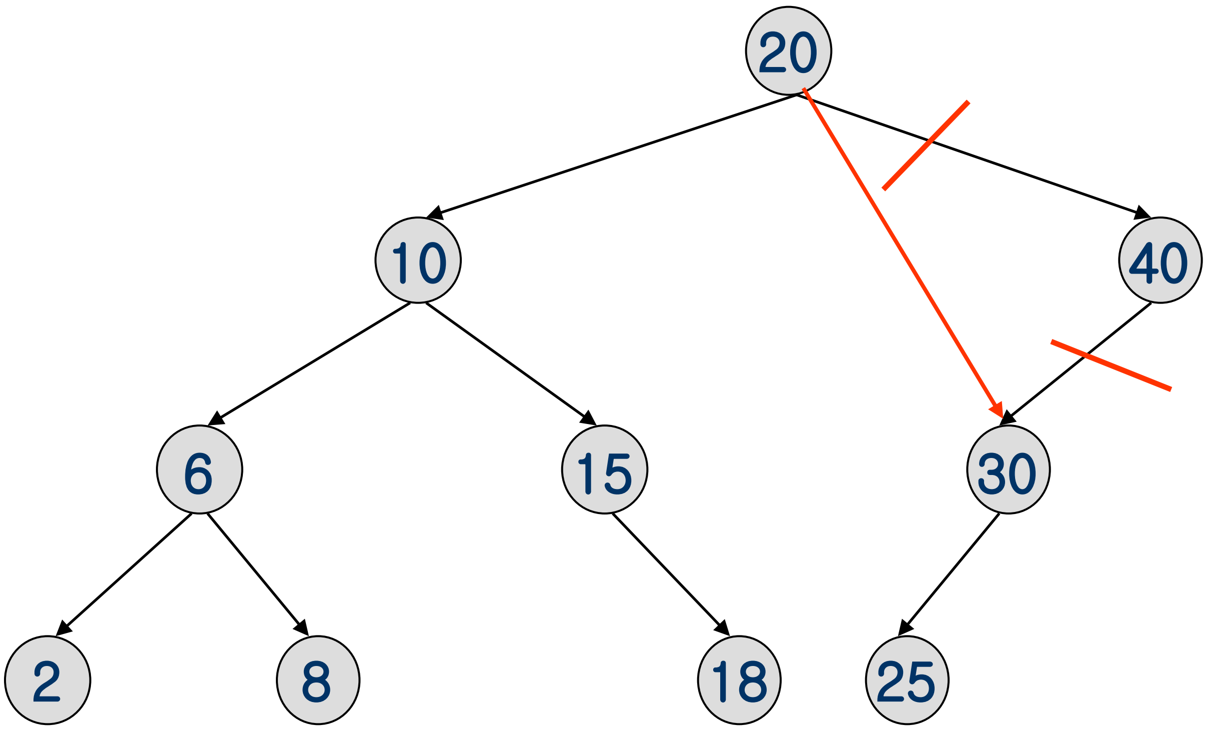
Case 1: A Leaf Node

Delete a leaf element. key=35



Case 2: A Node with Degree 1

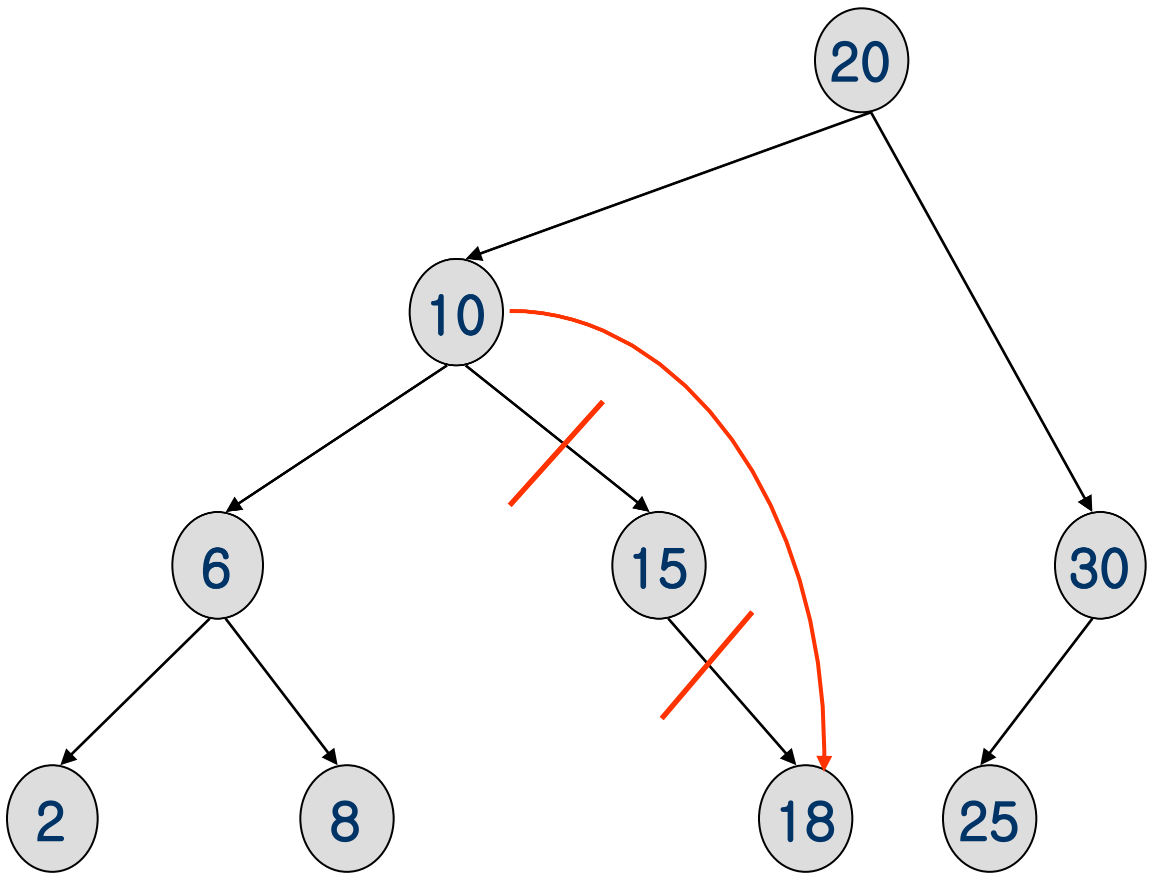
Delete 40 (A node with only a left child)



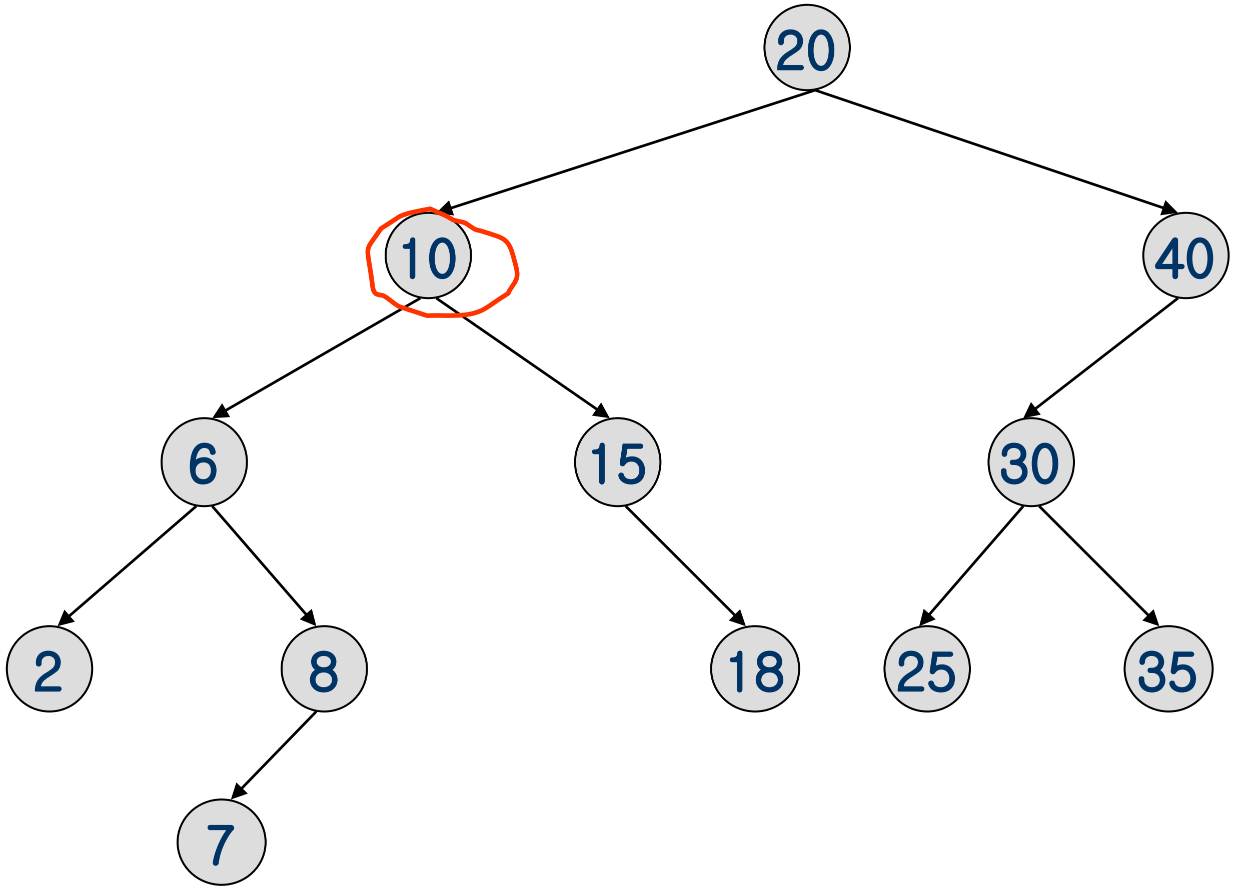
- **Which nodes have a degree 1?**
- **Example: Delete key=40**

Case 2: A Node with Degree 1

Delete 15 (A node with only a right child)

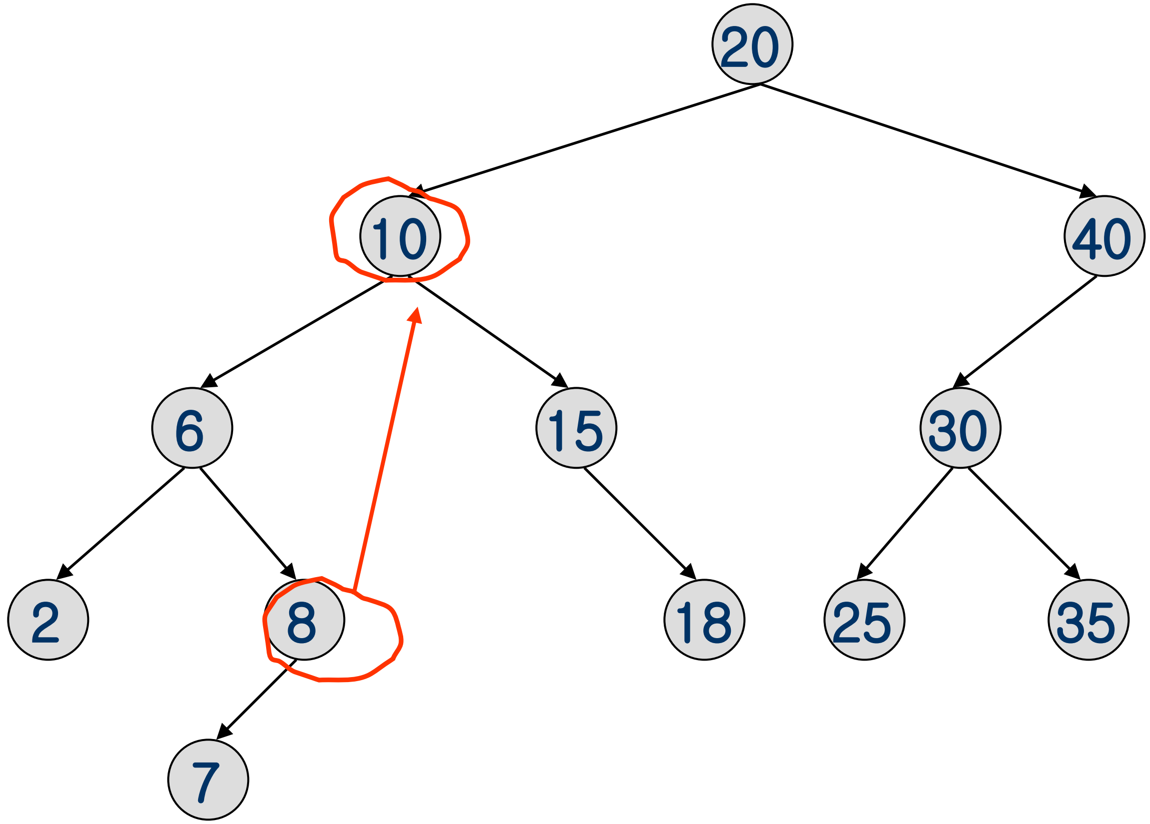


Case 3: A Node with Degree 2



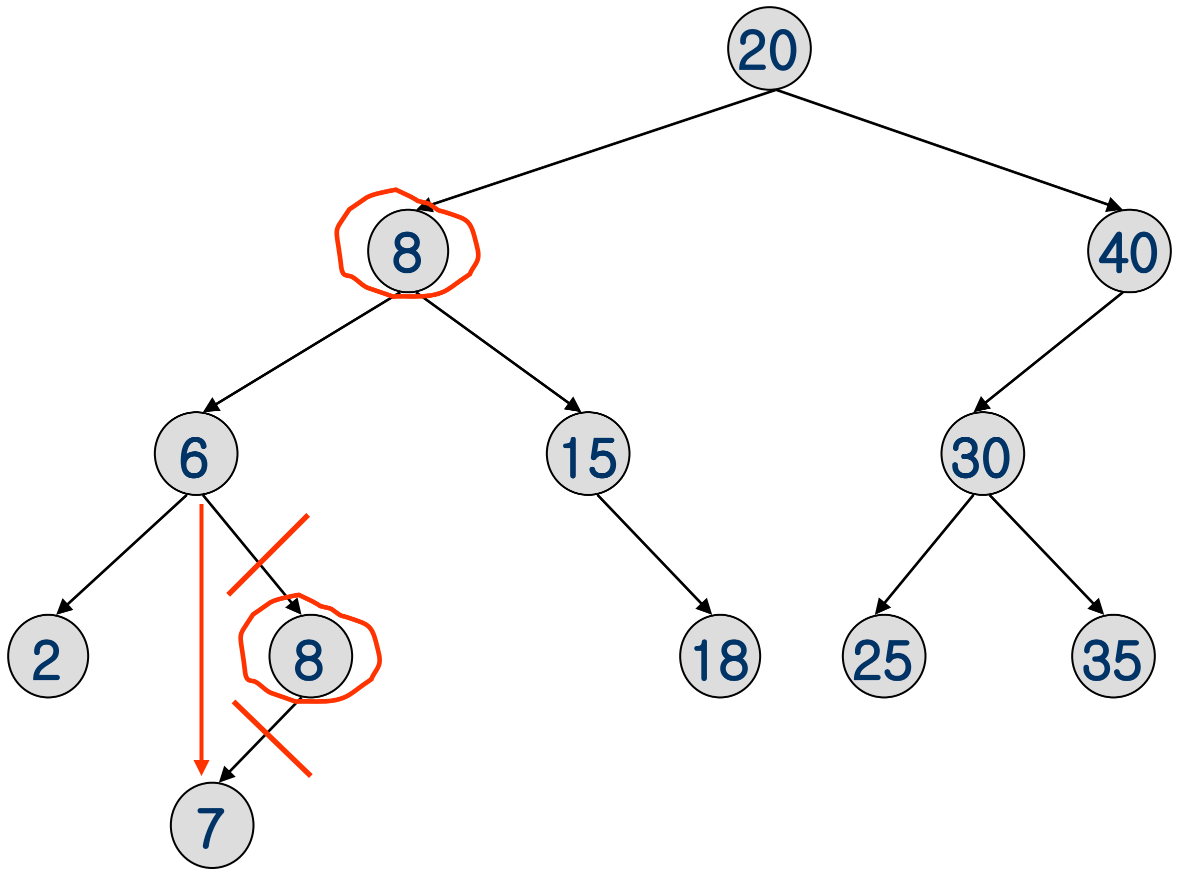
- Which nodes have a degree 2?
- Example: Delete key=10

Case 3: A Node with Degree 2



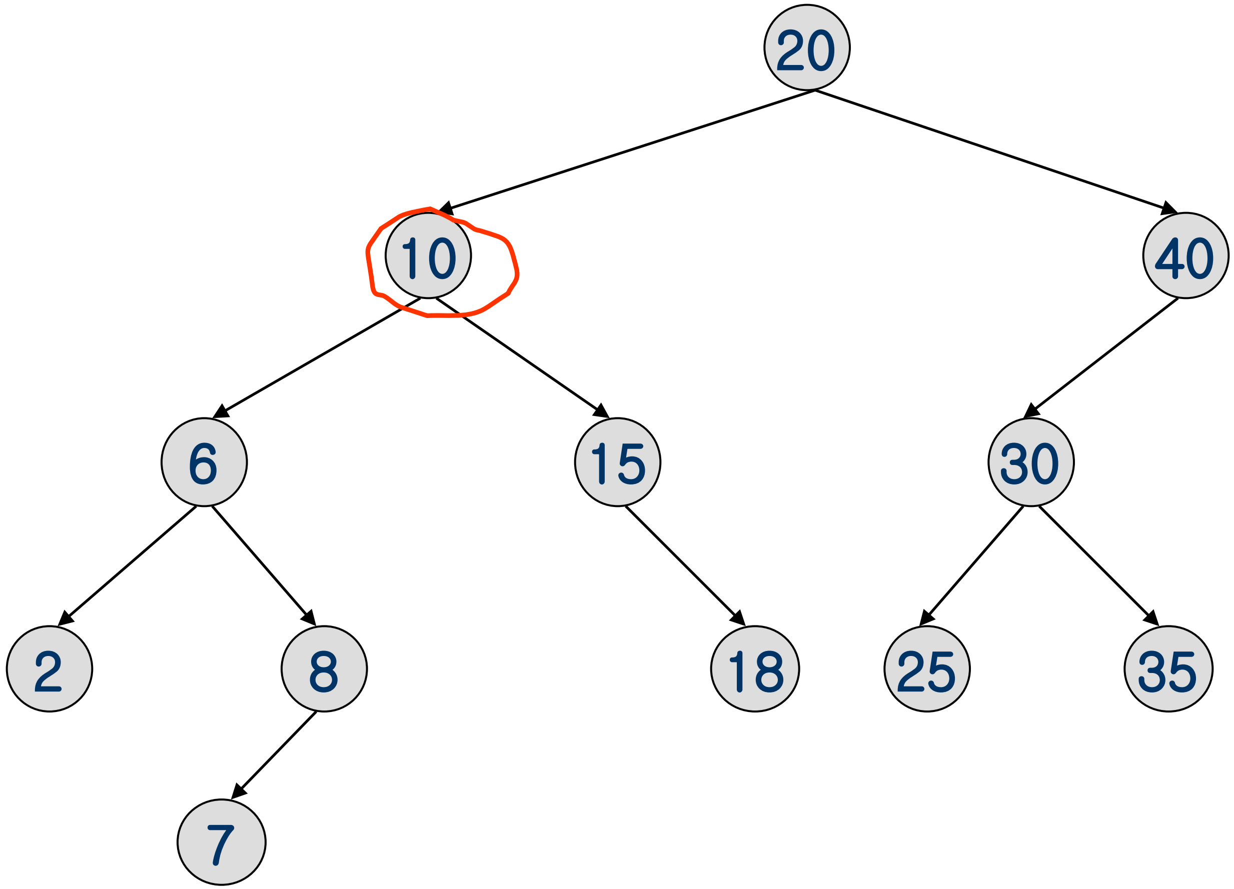
- Replace with the largest key in the left subtree (or the smallest in the right subtree)
- Which node is the largest key in the left subtree?

Case 3: A Node with Degree 2



The largest key must be in a leaf or degree 1 node.

Case 3: A Node with Degree 2



- **Which nodes have a degree 2?**
- **Example: Delete key=10**

Case 3: A Node with Degree 2

- Note that the node with largest key in the left subtree (as well as that with smallest in the right subtree) is guaranteed to be in a node with either zero or one nonempty subtree

- How can we find the node with largest key in the left subtree of a node?

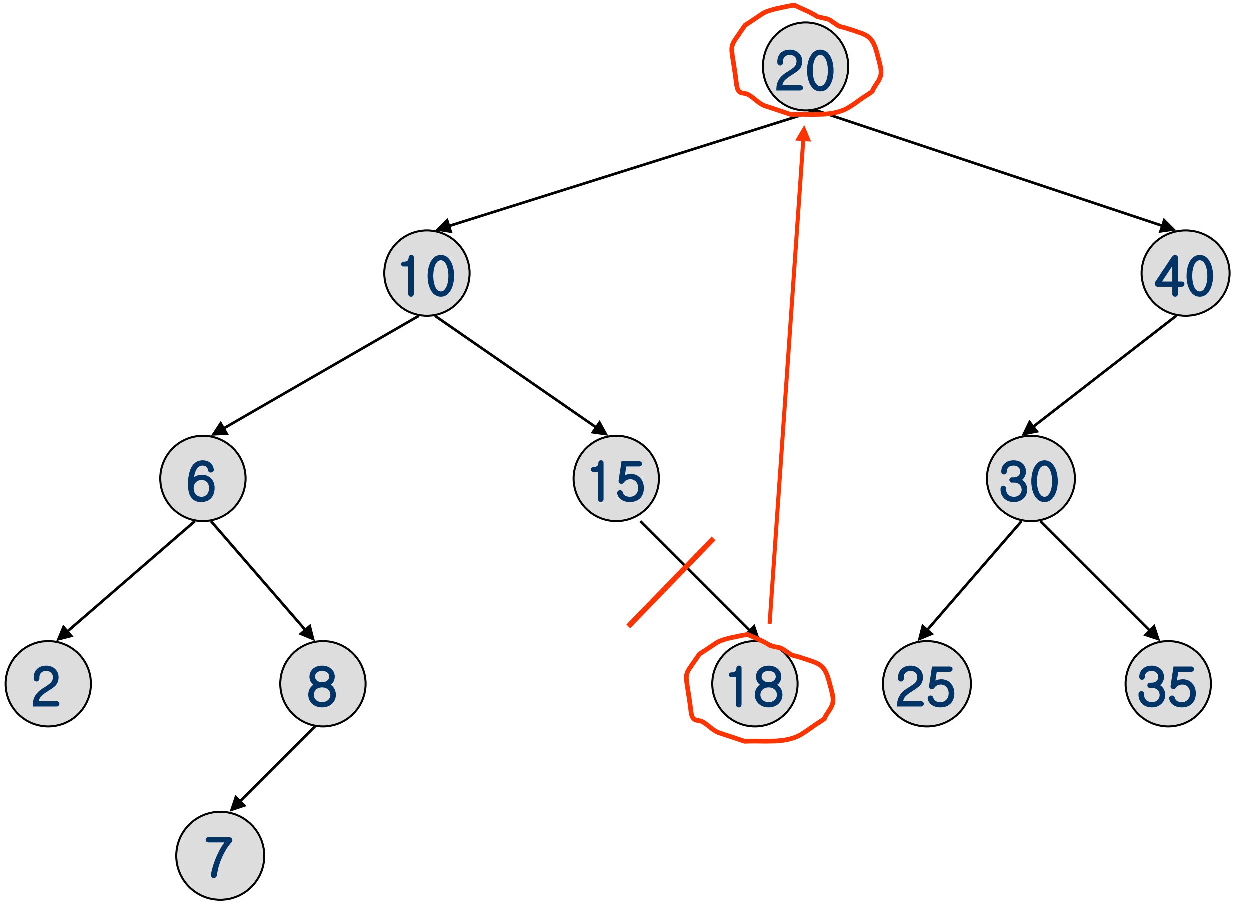
  ➔ by moving to the root of that subtree and then following a sequence of right-child pointers until we reach a node whose right-child pointer is NULL

- How can we find the node with smallest key in the right subtree of a node?

  ➔ by moving to the root of that subtree and then following a sequence of left-child pointers until we reach a node whose left-child pointer is NULL

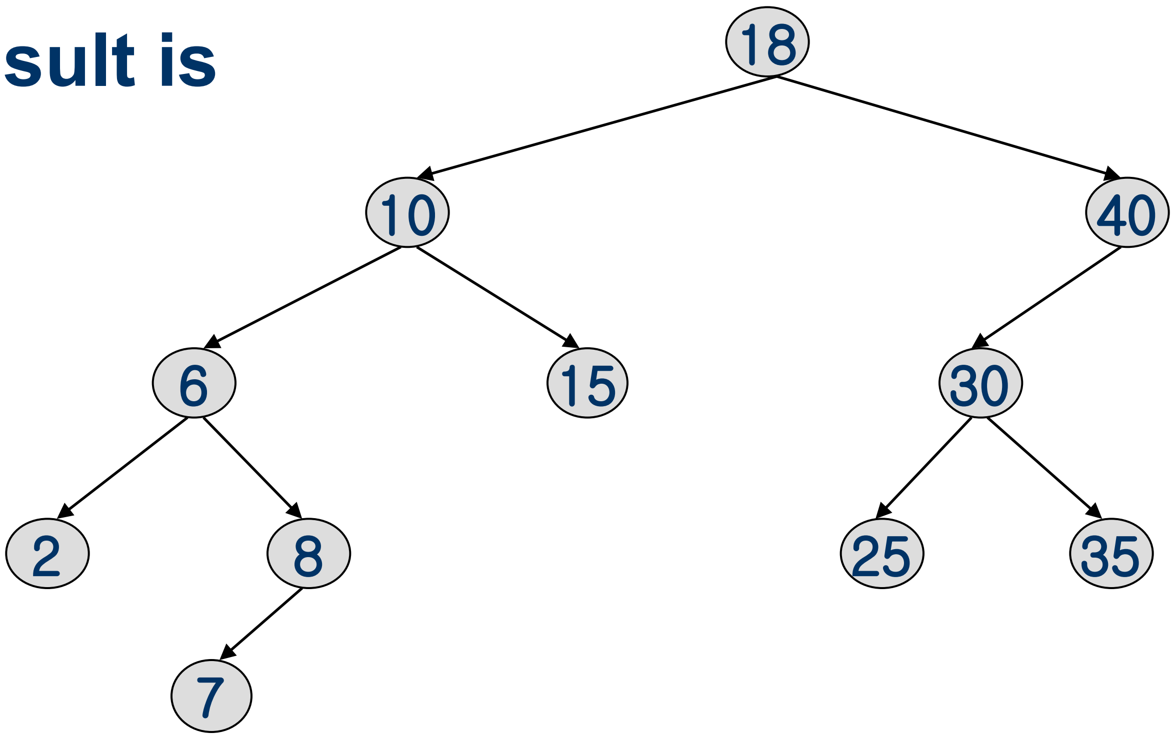
Another Delete from a Degree 2 Node

- Delete from a degree 2 node. key=20
- Replace with the largest in the left subtree.



Another Delete from a Degree 2 Node

The result is



- **The time complexity of delete is O(*height*).**

```
/**
 * Internal method to remove from a subtree.
 * x is the item to remove.
 * t is the node that roots the subtree.
 * Set the new root of the subtree.
 */
void remove( const Comparable & x, BinaryNode * & t )
{
    if( t == nullptr )
        return;   // Item not found; do nothing
    if( x < t->element )
        remove( x, t->left );
    else if( t->element < x )
        remove( x, t->right );
    else if( t->left != nullptr && t->right != nullptr ) // Two children
    {
        t->element = findMin( t->right )->element;
        remove( t->element, t->right );
    }
    else
    {
        BinaryNode *oldNode = t;
        t = ( t->left != nullptr ) ? t->left : t->right;
        delete oldNode;
    }
}
```

Analysis of BST Operations

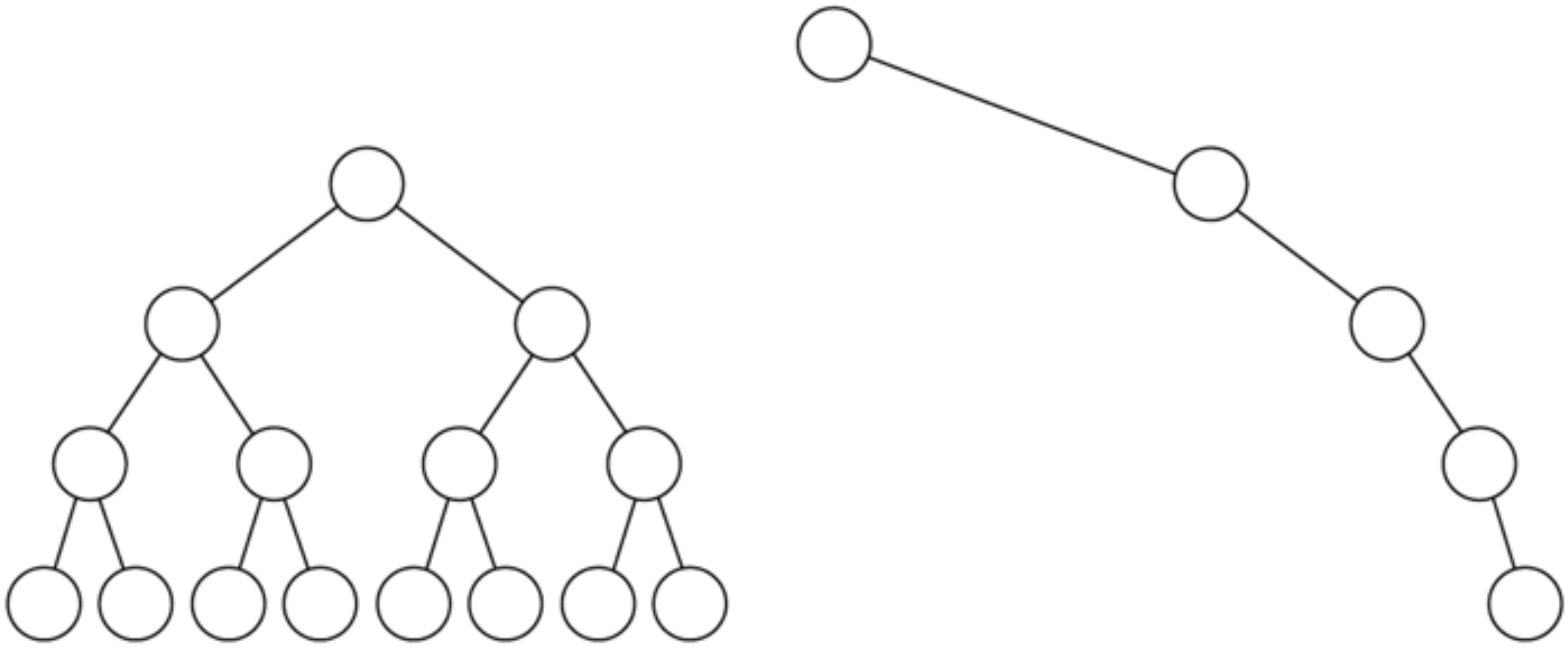
- The cost of an operation is proportional to the depth of the last accessed node.

- The cost is logarithmic for a well-balanced tree, but it could be as bad as linear for a degenerate/skewed tree.

- In the best case we have logarithmic access cost, and in the worst case we have linear access cost.

Examp l e

(a)The balanced tree has a depth of log *N*;
(b) the unbalanced tree has a depth of *N* – 1.



(a) (b)

Maximum and Minimum Heights of a Binary Tree

- **The efficiency of most of the binary tree (and BST) operations depends on the height of the tree.**
- **The maximum number of key comparisons for retrieval, deletion, and insertion operations for BSTs is the height of the tree.**

- **The <u>maximum</u> of height of a binary tree with n nodes is n-1.**
- **Each level of a <u>minimum</u> height tree, except the last level, must contain as many nodes as possible.**

Order of Operations on BSTs

Operation	Average case	Worst case
Retrieval	O(log n)	O(n)
Insertion	O(log n)	O(n)
Deletion	O(log n)	O(n)
Traversal	O(n)	O(n)

Binary search tree

- Exercise
- Input:50, 72, 43, 85, 75, 20, 35, 45, 65, 30
- Delete 72

Homework

- **Please refer to Icourse，Huawei Cloud.**

- **Due date for quiz: 23:30　2022/4/26**
- **Due date for homework: 23:30 2022/5/1**
- **Due data for online lab assignment: 2022/5/1　23：30**
- **Due data for offline lab assignment: 2022/5/8　18:00**