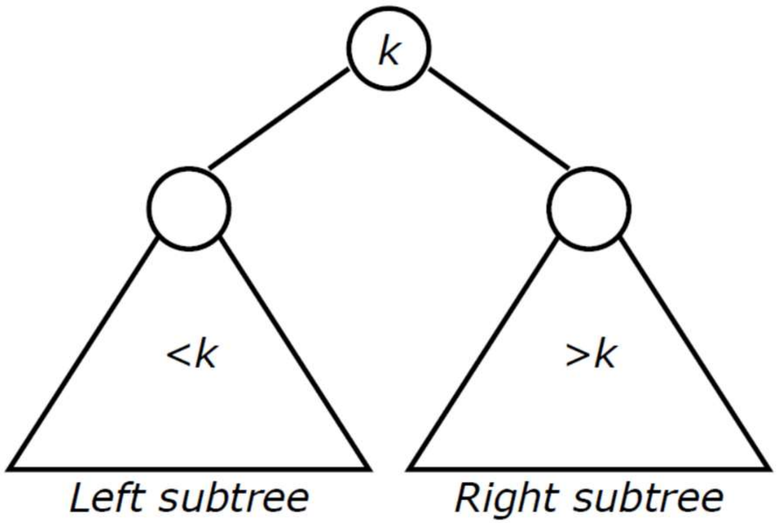
Binary Search Trees

Prof. Ki-Hoon Lee
Dept. of Computer Engineering
Kwangwoon University

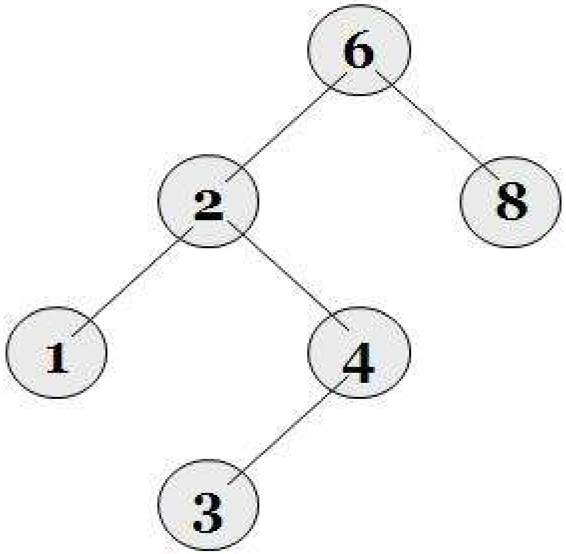
Definition of a Binary Search Tree (BST)

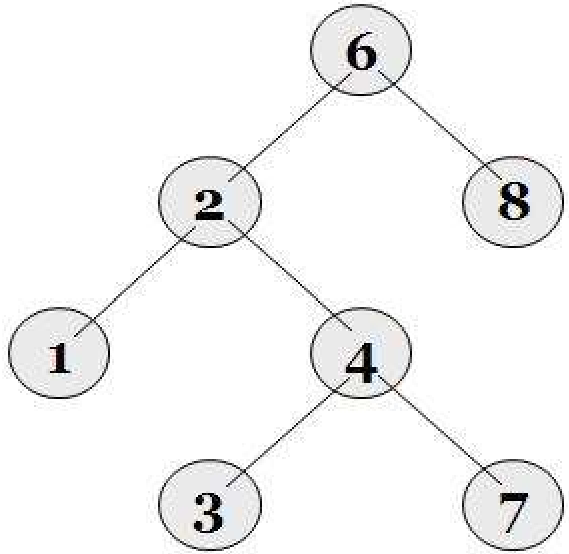
- A binary tree
- Each node has a (*key*, *element*) pair
  - element: value or data
- For every node x, all keys in the left subtree of x are smaller than that in x
- For every node x, all keys in the right subtree of x are greater than that in x
- The left and right subtrees are also binary search trees



Example BST

***A** binary search tree*





Not a ***binary search tree,***
but a ***binary tree***

Only keys are shown.

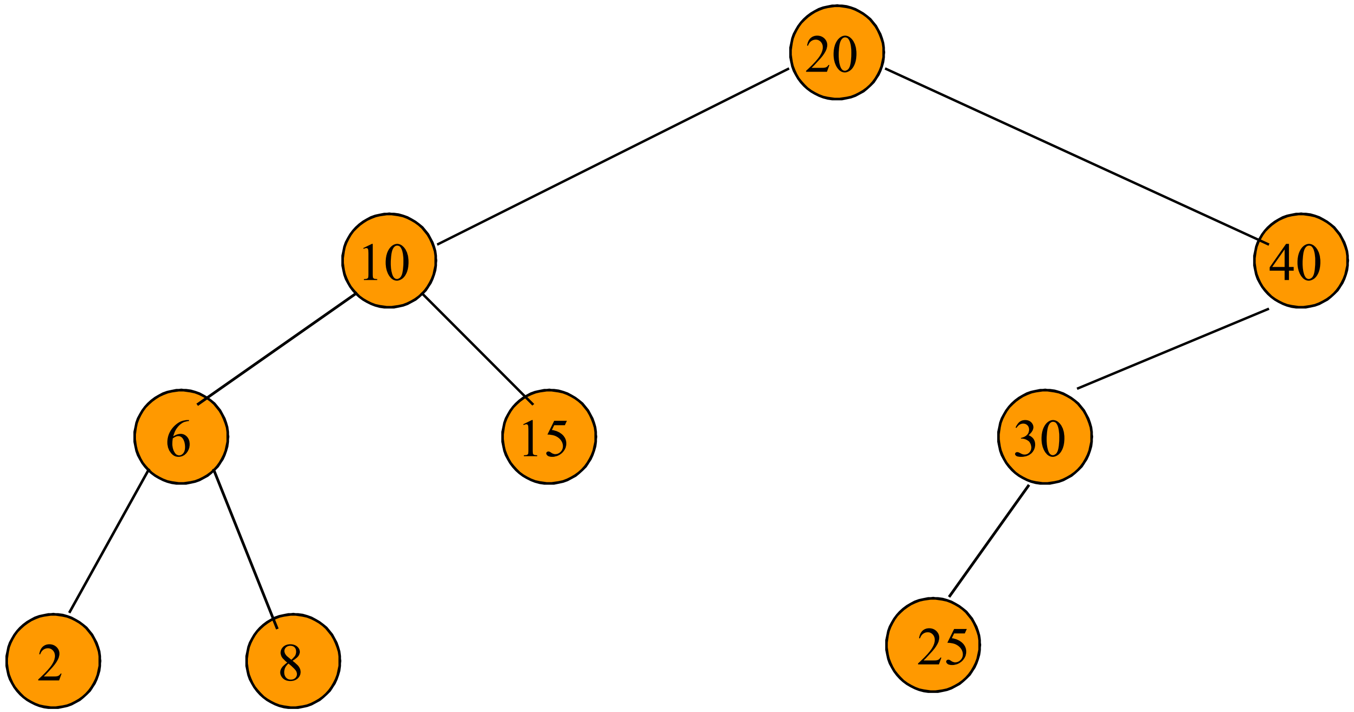
A Dictionary

- A *dictionary* is a collection of pairs, each pair has a key and an associated element
  - It can be implemented using a BST

Making a member **function const** means that it cannot call any non-**const** member **functions**, nor can it change any member variables.

```
template <class K, class E>
class Dictionary {
public:
        virtual void Ascend(void) const = 0;
          // print the dictionary in ascending order by key

        virtual pair<K, E>*Get(const K&) const = 0;
          // return pointer to the pair with specified key; return NULL if no such pair

        virtual void Insert(const pair<K, E>&) = 0;
          // insert the given pair; if key is a duplicate, update the associated element

        virtual void Delete(const K&) = 0;
          // delete pair with specified key
};
```

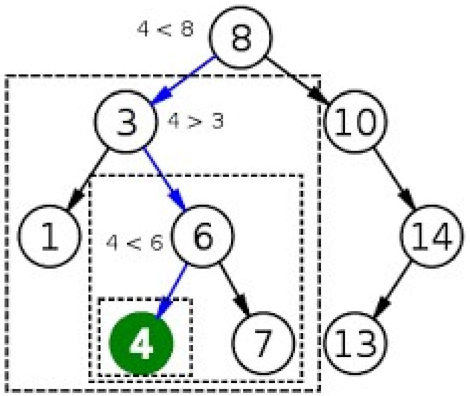
The Operation Ascend()



Do an inorder traversal. O(n) time.

Searching a BST

- Searching for a node with key k
- We begin at the root
- If the root is NULL, the tree is empty and the search is unsuccessful
- Otherwise, we compare k with the key k_{root} in the root
  - If $k < k_{root}$, then only the *left* subtree needs to be searched
  - If $k > k_{root}$, then only the *right* subtree needs to be searched
  - Otherwise, $k == k_{root}$ and the search terminates successfully
- Complexity: O(height)

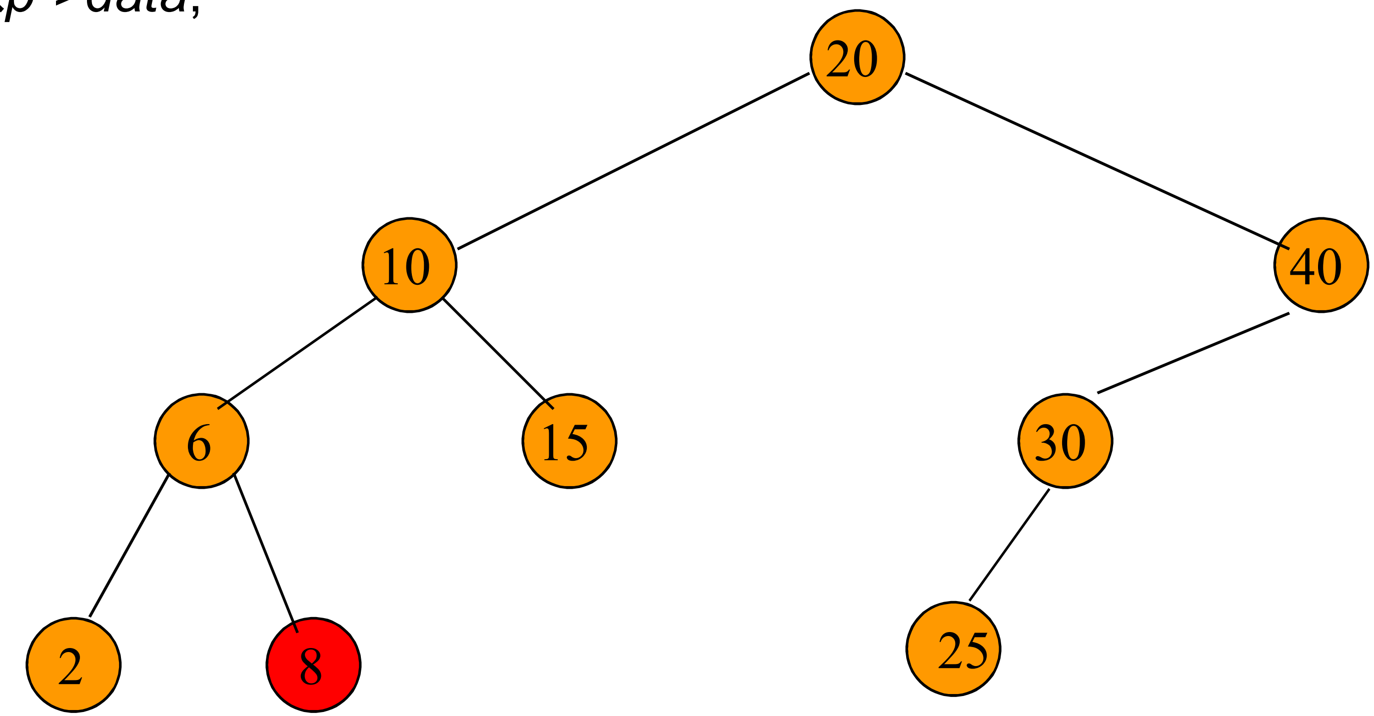


Recursive Search of a BST

```
template <class K, class E> // Driver
pair<K, E>* BST<K, E>::Get(const K& k)
{
// Search the binary search tree (*this) for a pair with key k.
// If such a pair is found, return a pointer to this pair; otherwise, return NULL.
  return Get(root, k);
}

template <class K, class E> // Workhorse
pair<K, E>* BST<K, E>::Get(TreeNode<pair<K, E> >*p, const K& k)
{
  if(p == NULL) return NULL;
  if(k < p->data.first) return Get(p->leftChild, k);
  if(k > p->data.first) return Get(p->rightChild, k);
  return &p->data;
}
```

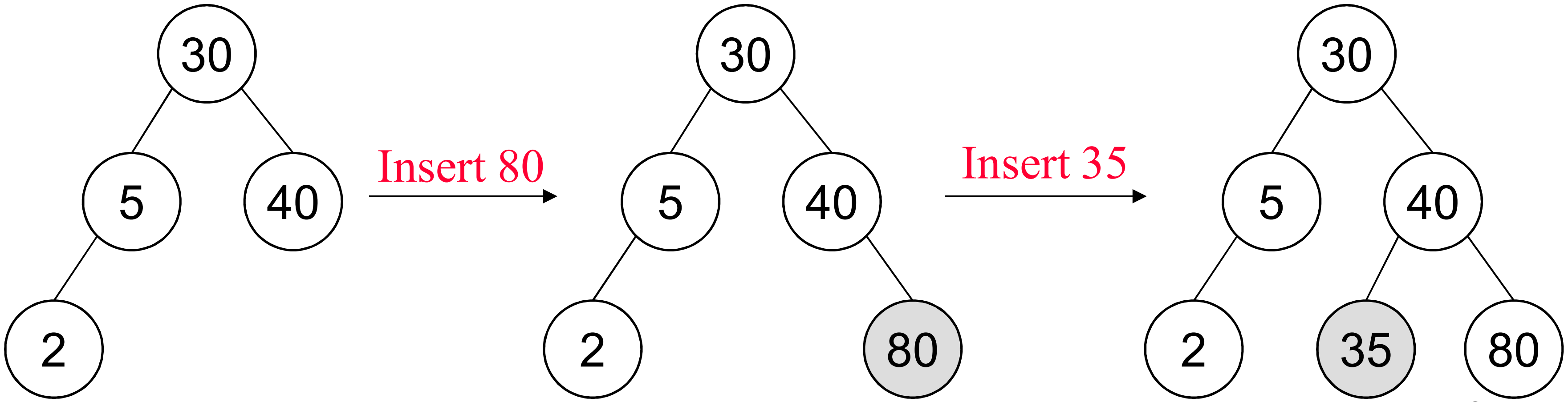
Example ($k = 8$)

```
template <class K, class E> // Workhorse
pair<K, E>* BST<K, E>::Get(TreeNode<pair<K, E> >*p, const K& k)
{
  if(p == NULL) return NULL;
  if(k < p->data.first) return Get(p->leftChild, k);
  if(k > p->data.first) return Get(p->rightChild, k);
  return &p->data;
}
```



Insertion into a BST

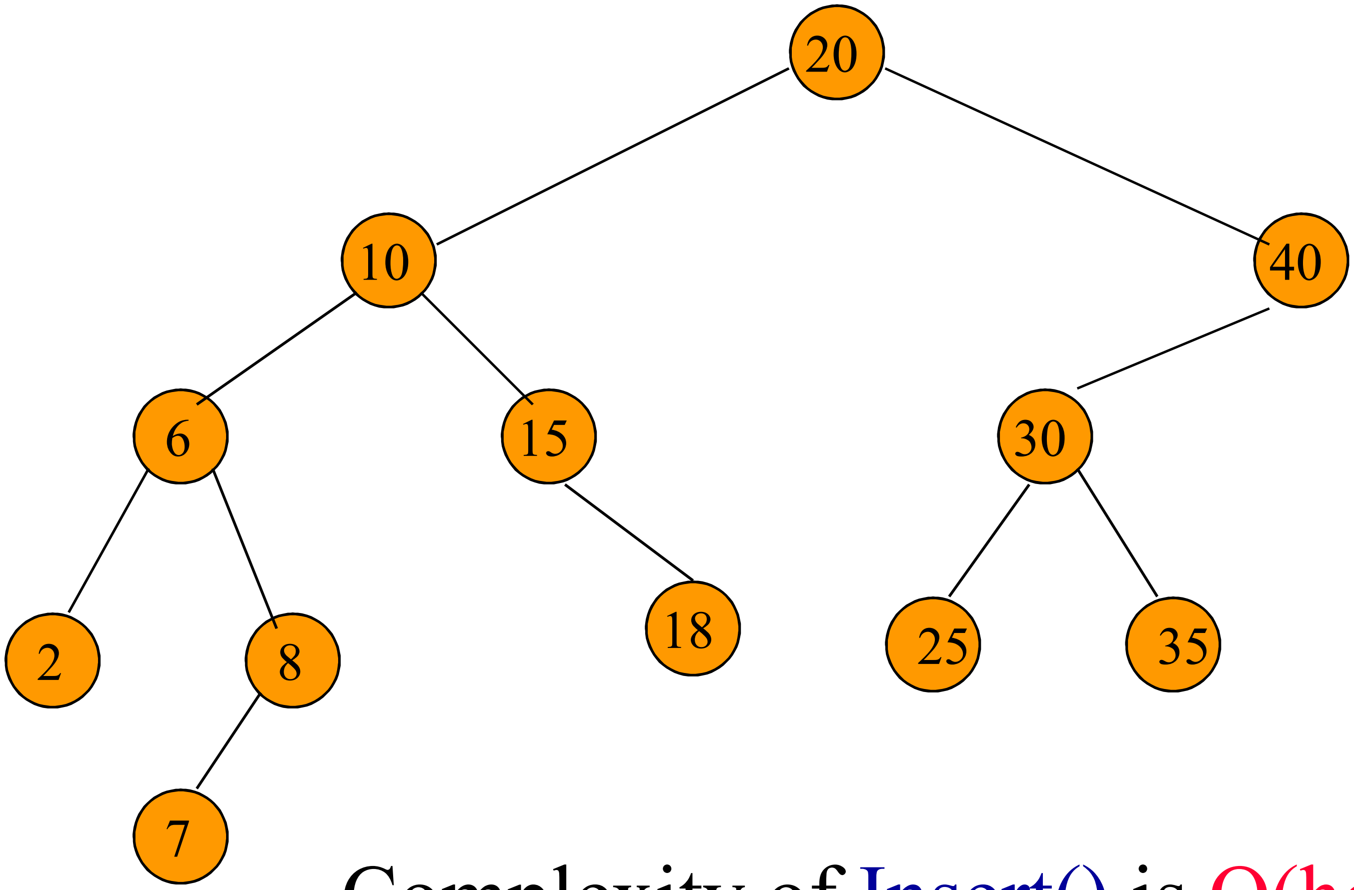
- To insert a pair (k, e), we first search the tree to verify that its key is different from those of existing nodes
  - By the definition of BST, no two nodes have the same key
- If the search is successful (i.e., key is a duplicate), the associated element is updated
- If the search is unsuccessful, the node is inserted at the point the search terminated



Insertion into a BST (cont.)

```
=====================================================================
template <class K, class E>
void BST<K,E>::Insert(const pair<K,E>& thePair)
{// Insert thePair into the binary search tree
 // Search for thePair.first
 // pp is the parent of p
   TreeNode<pair<K,E> > *p = root, *pp = NULL;
   while (p) {
      pp = p;
      if (thePair.first < p->data.first) p = p->leftChild;
      else if (thePair.first > p->data.first) p = p->rightChild;
      else // duplicated, update the associated element
        {p->data.second = thePair.second; return;}
   }

   // Perform insertion
   p = new TreeNode<pair<K,E> > (thePair);
   if (root != NULL)   // tree not empty
      if (thePair.first < pp->data.first) pp->leftChild = p;
      else pp->rightChild = p;
   else root = p;
}
=====================================================================
```

The Operation Insert()



Complexity of Insert() is O(height).

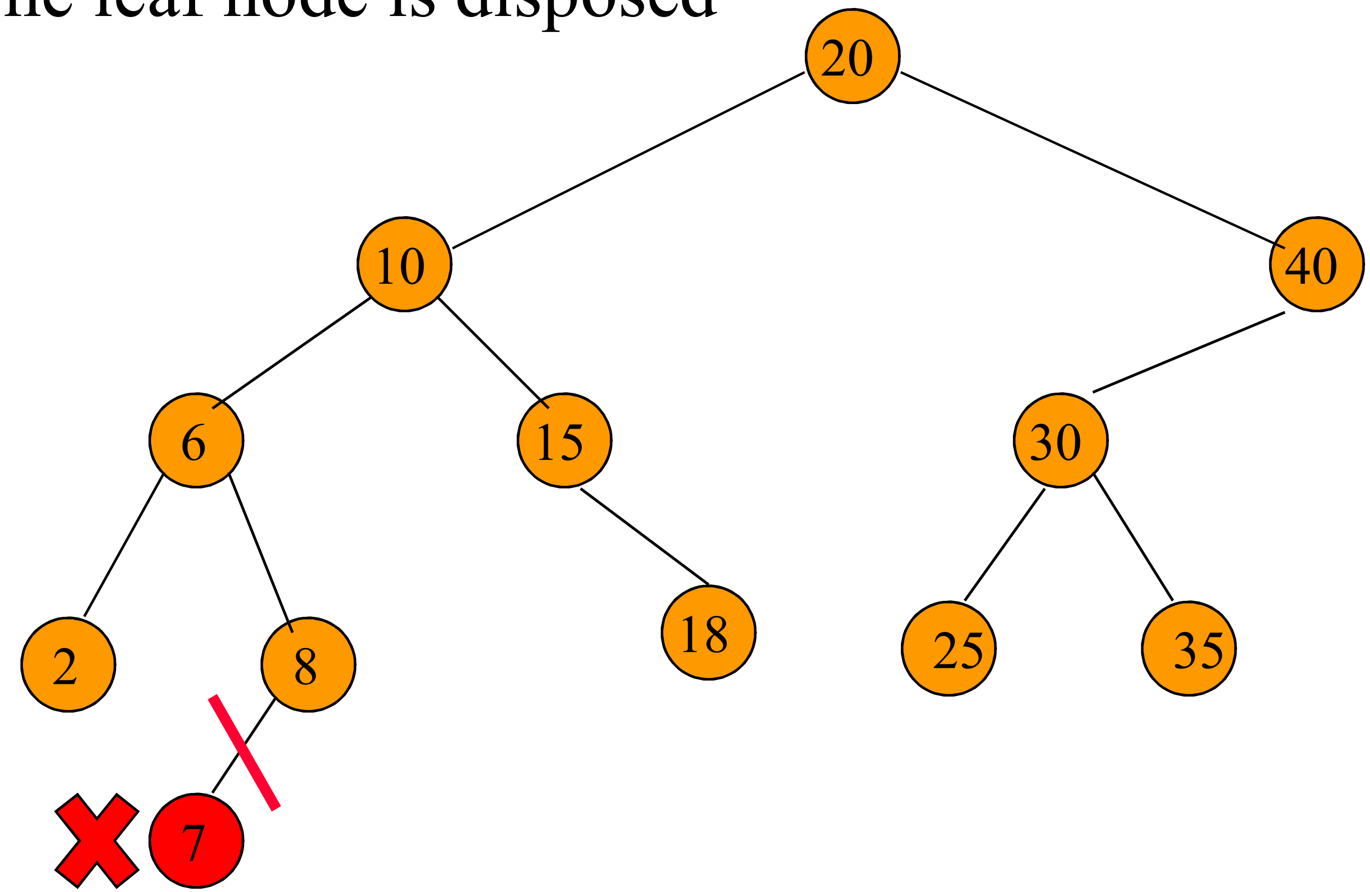
The Operation Delete()

Four cases:

- No node with delete key
- A degree 0 node (leaf node)
- A degree 1 node (internal node)
- A degree 2 node (internal node)

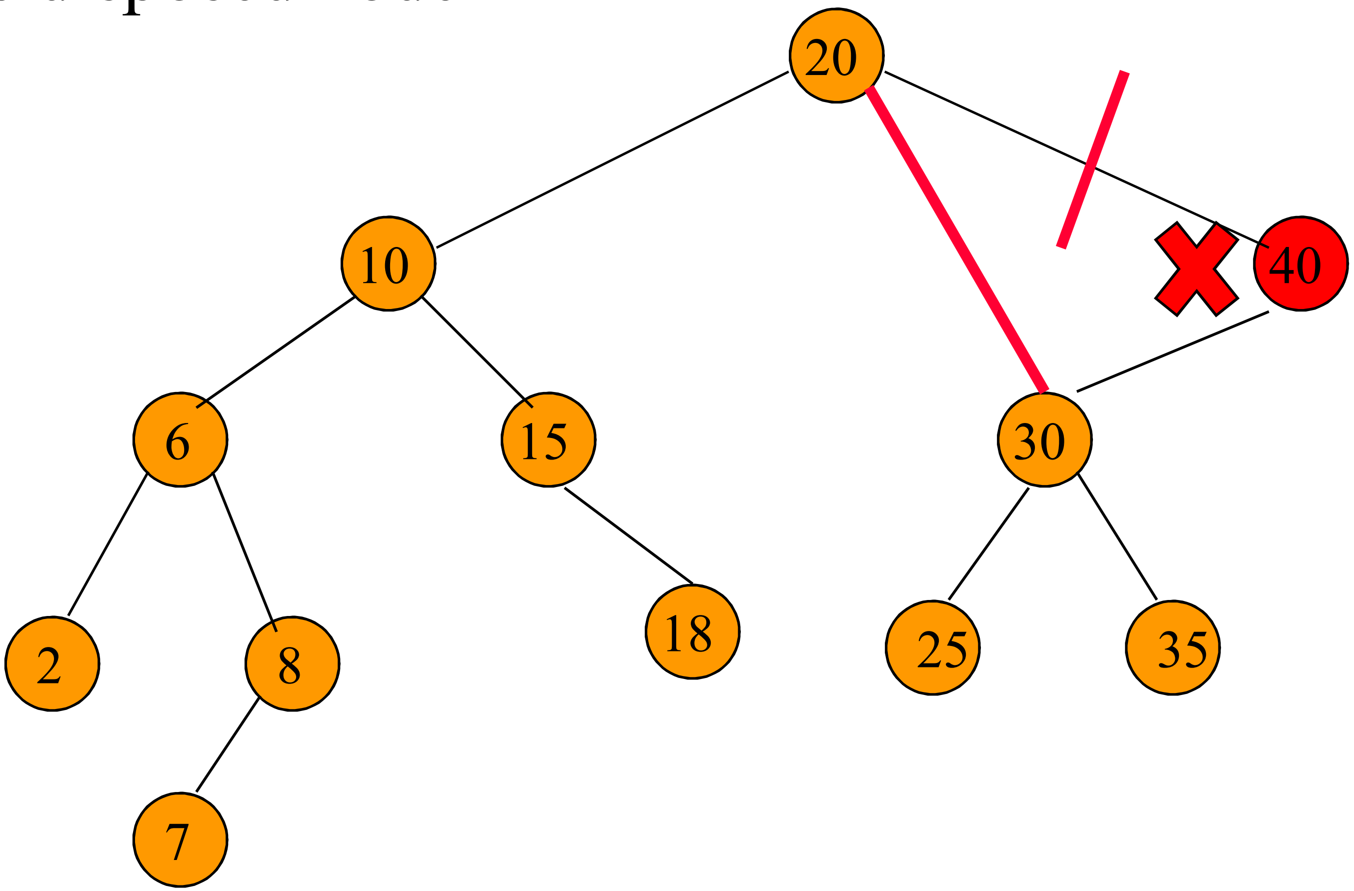
Delete a Leaf Node

- The corresponding child field of its parent is set to NULL
- The leaf node is disposed

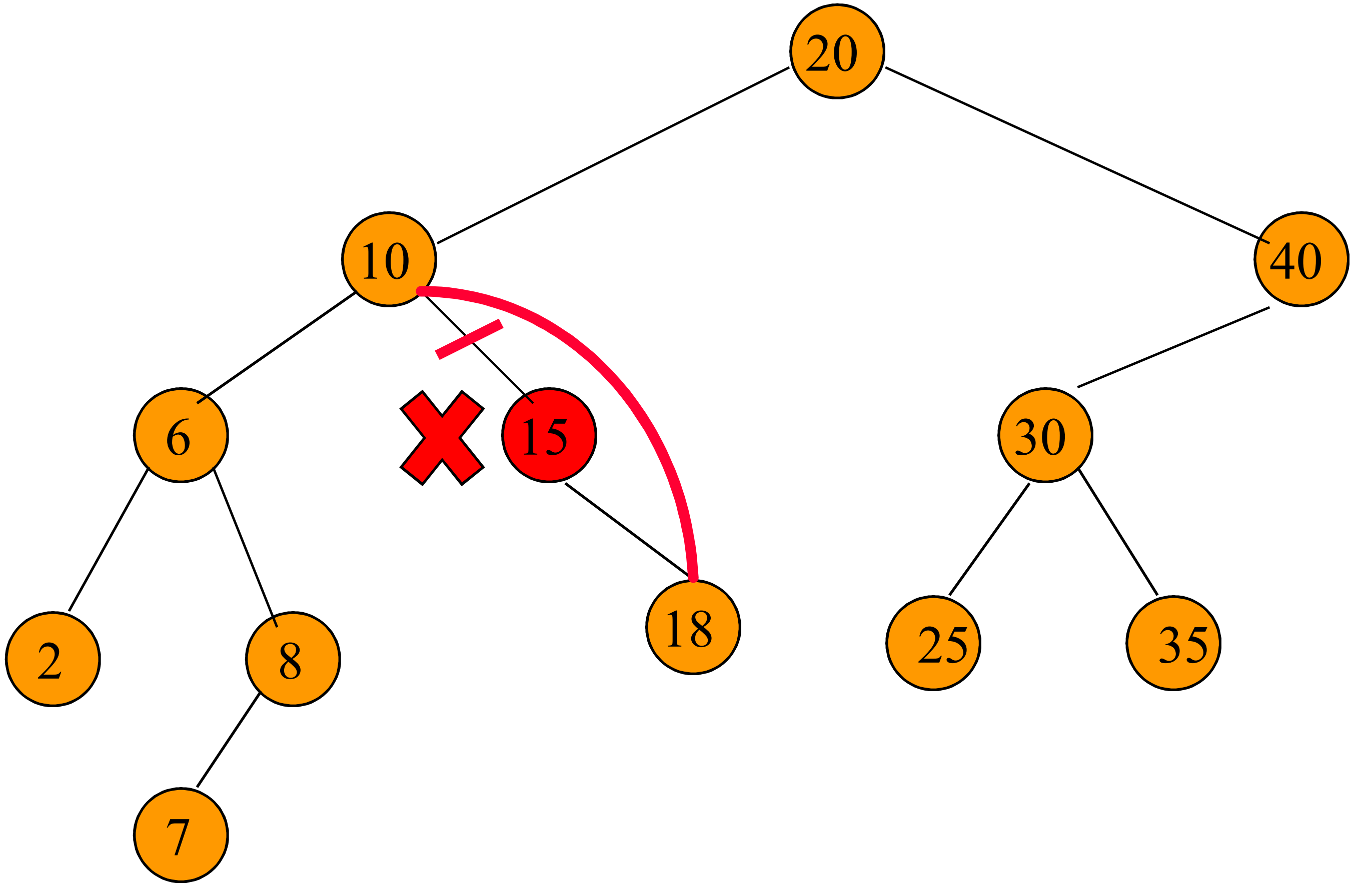


Delete a Degree 1 Node

- The node is disposed
- The single-child of the disposed node takes place of the disposed node



Delete a Degree 1 Node (cont.)

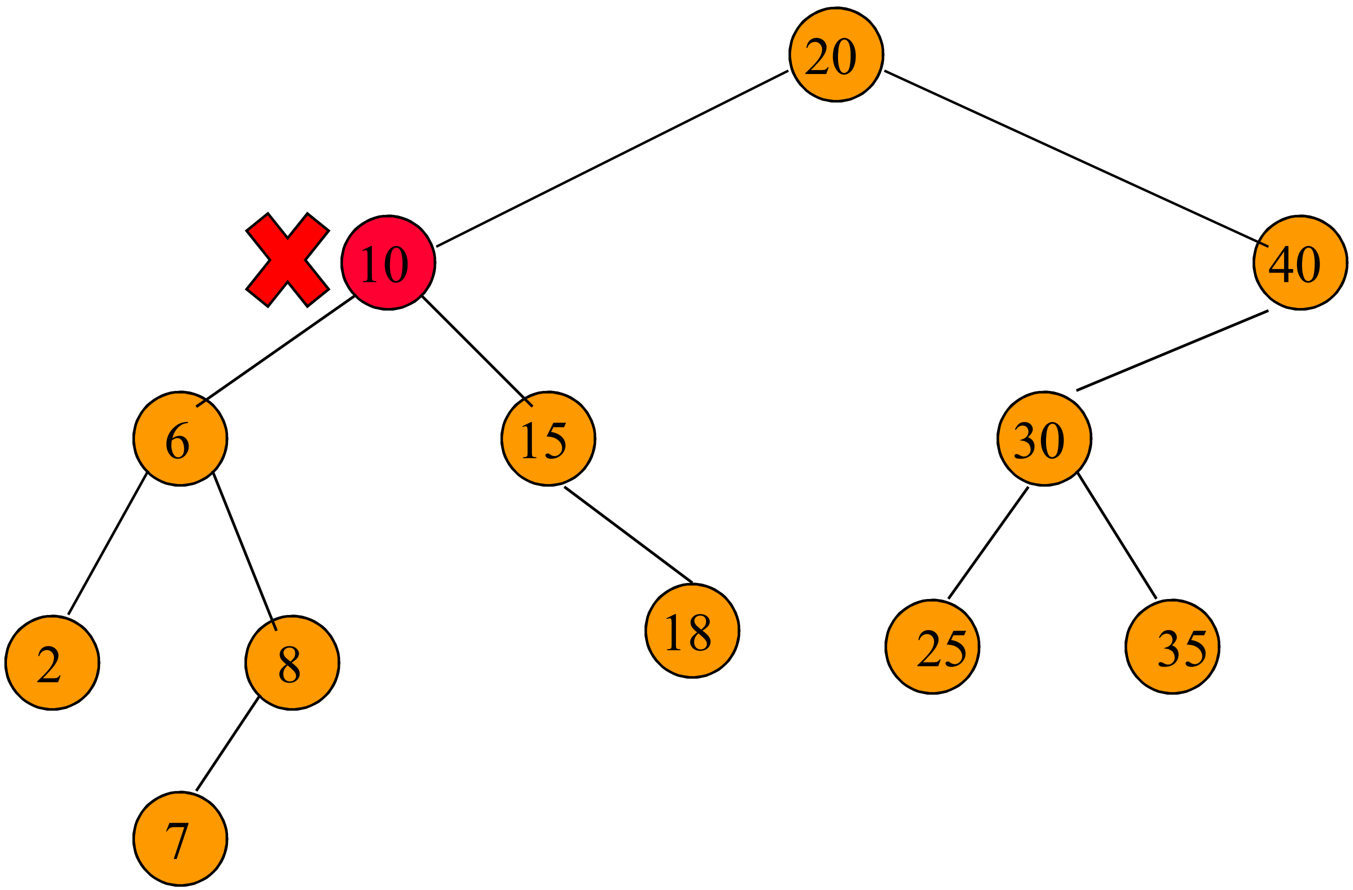


Delete a Degree 2 Node

- The node is replace by either
  - the largest node in its left subtree
  - the smallest node in its right subtree
- Delete this replacing node from the subtree from which it was taken

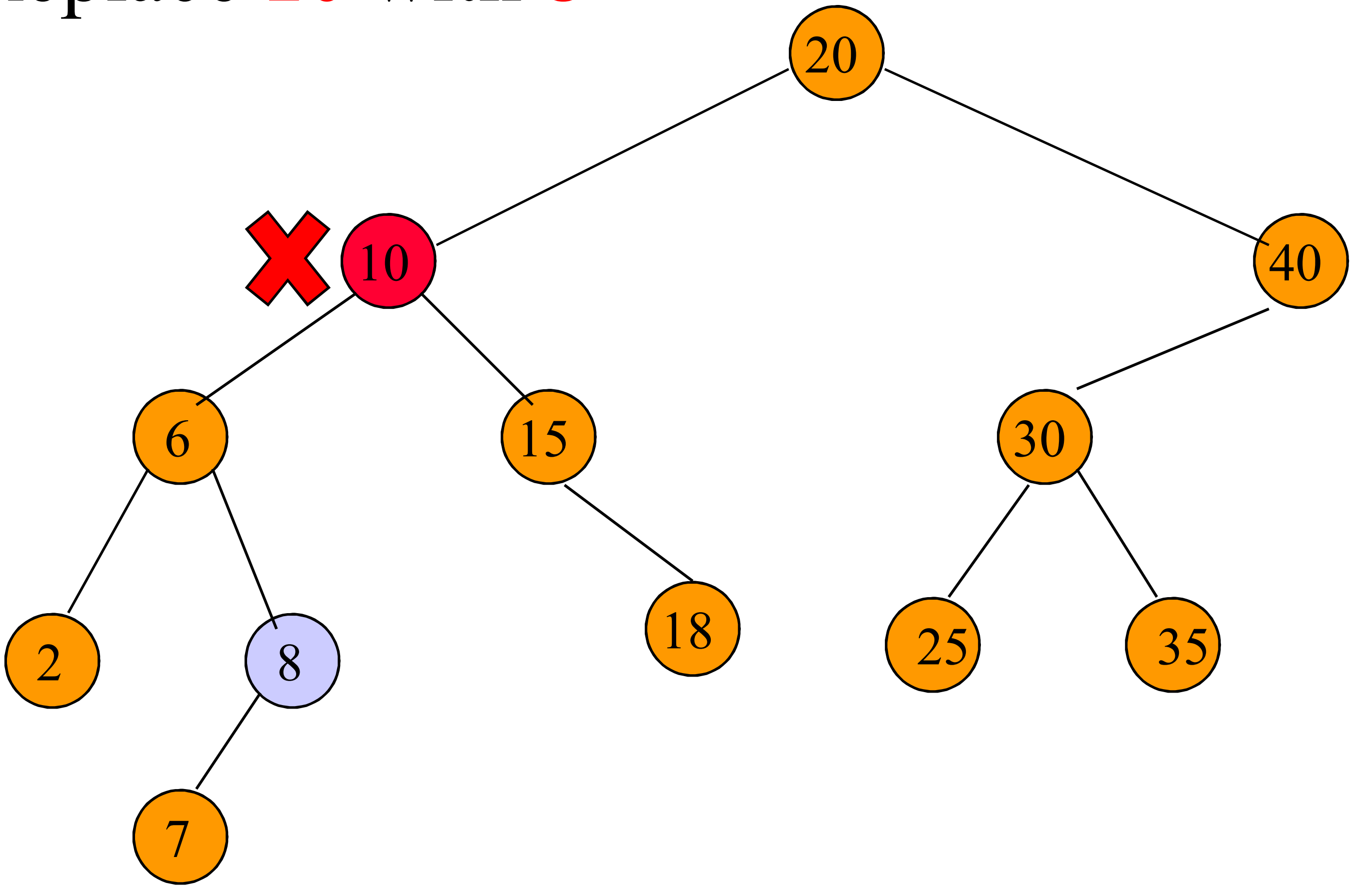
Example

- Delete **10**
- Find the largest key in the left subtree (or the smallest key in the right subtree).



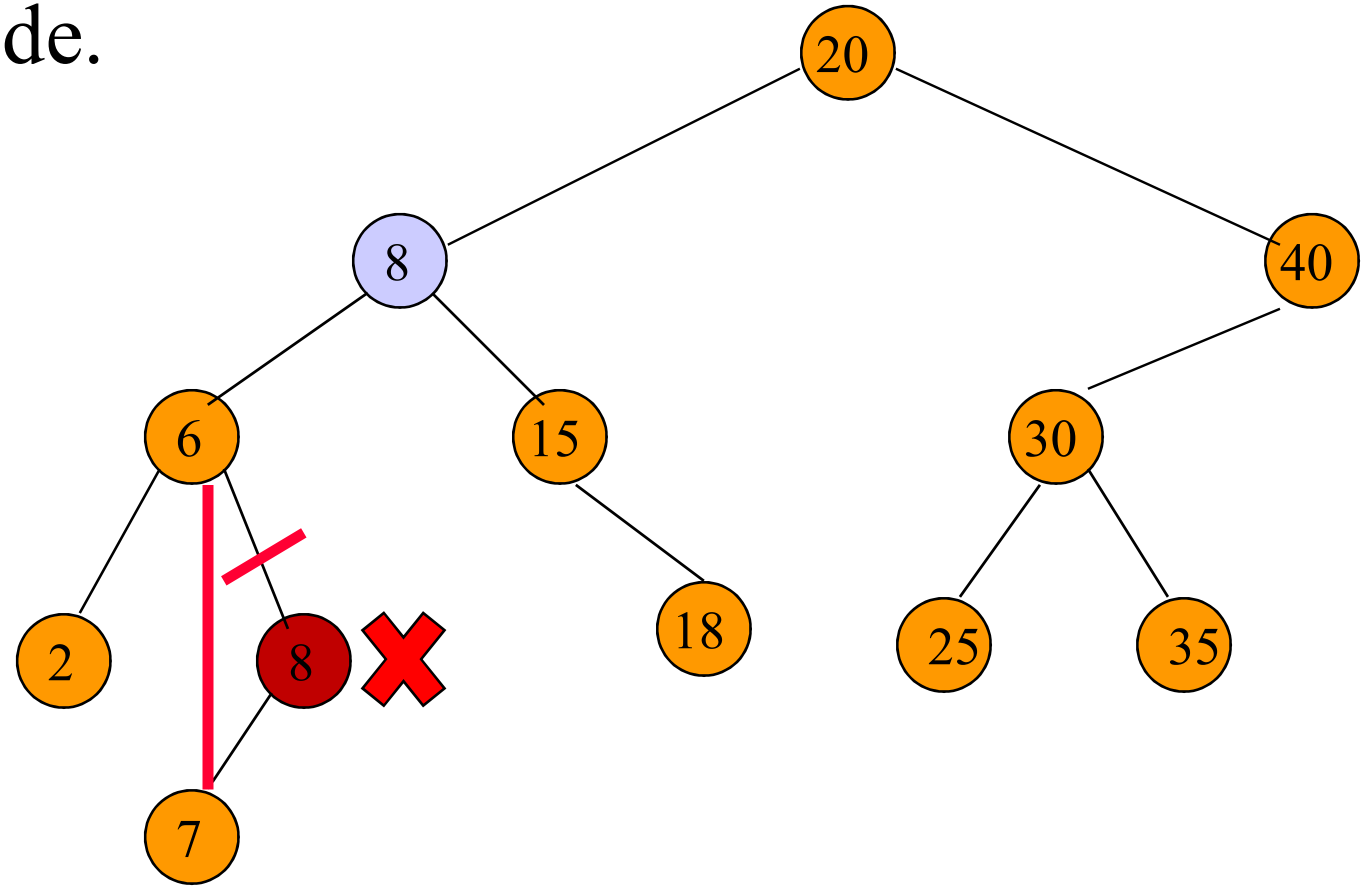
Example (cont.)

- **8** is the largest key in the left subtree
- Replace **10** with **8**

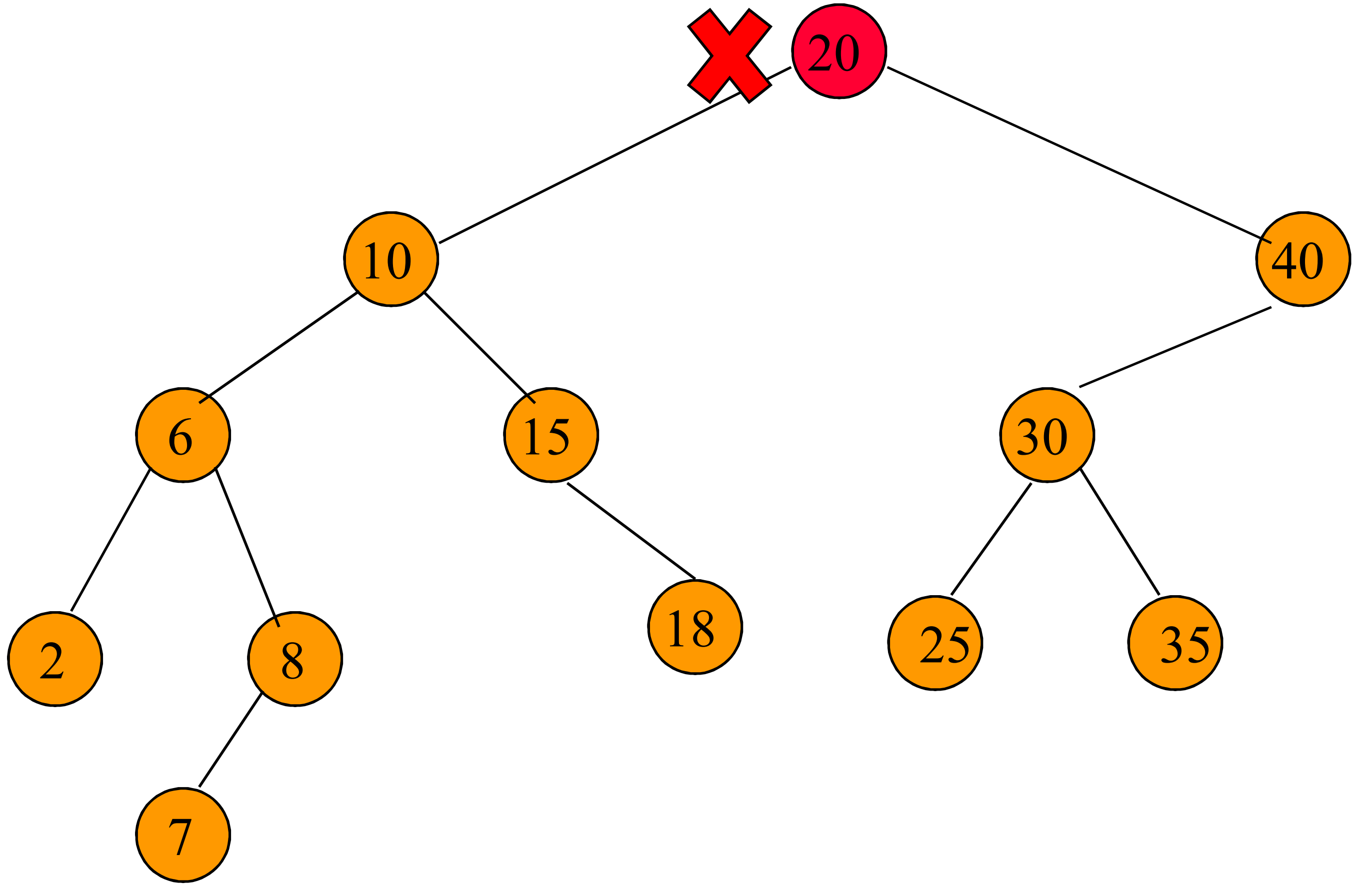


Example (cont.)

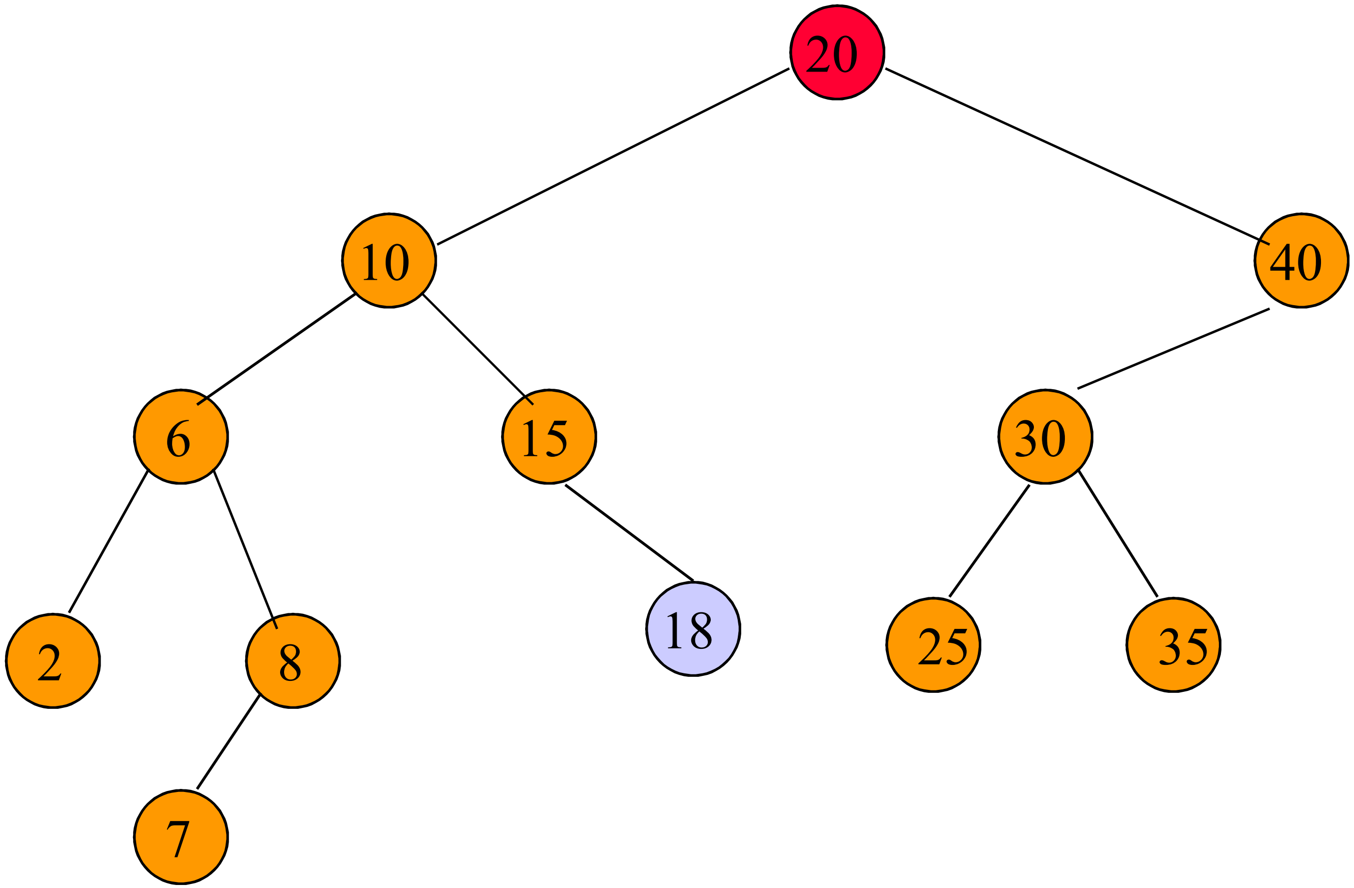
- Delete the replacing node **8**
- The largest key must be in a leaf or degree 1 node.



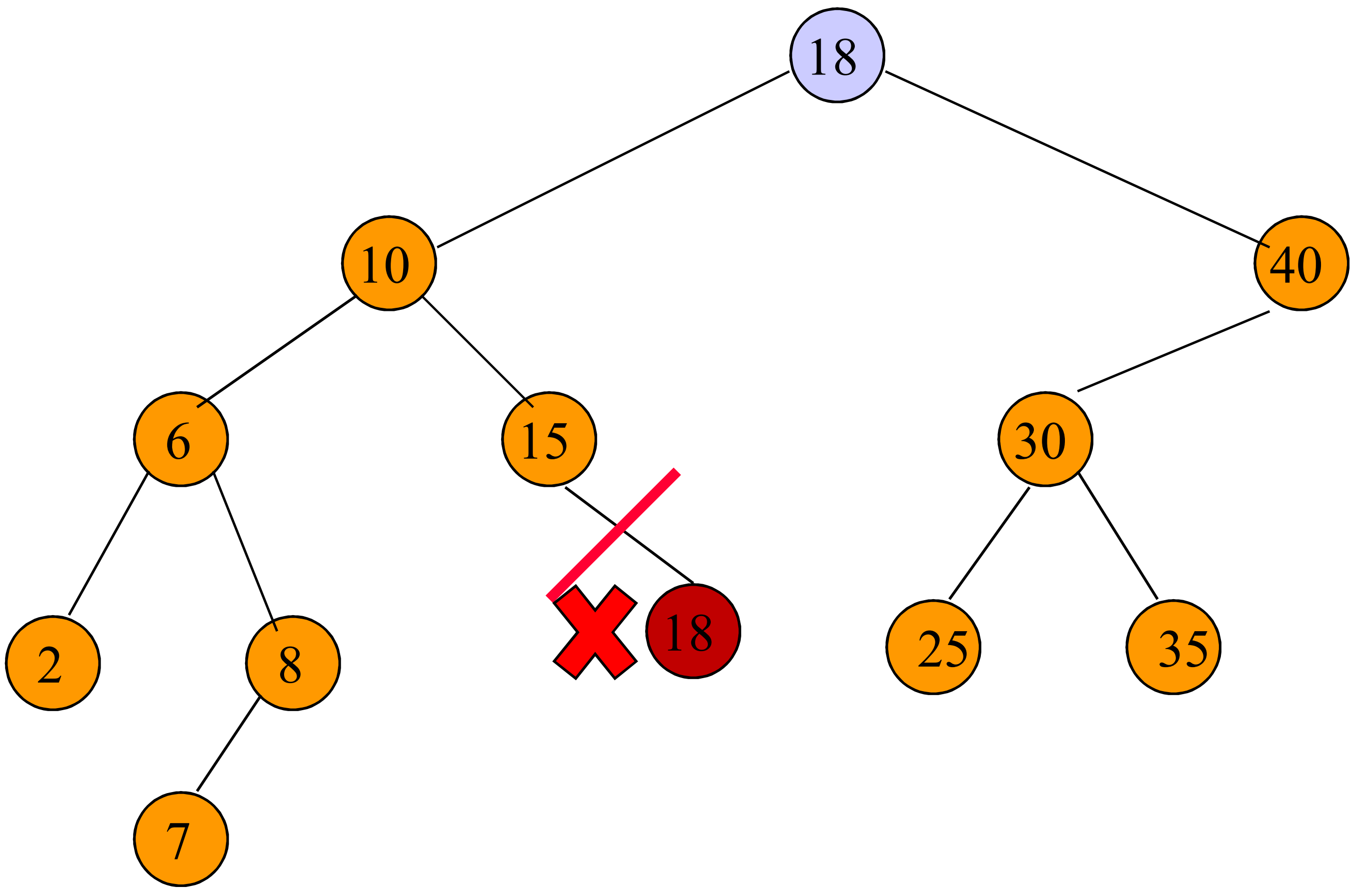
Another Example



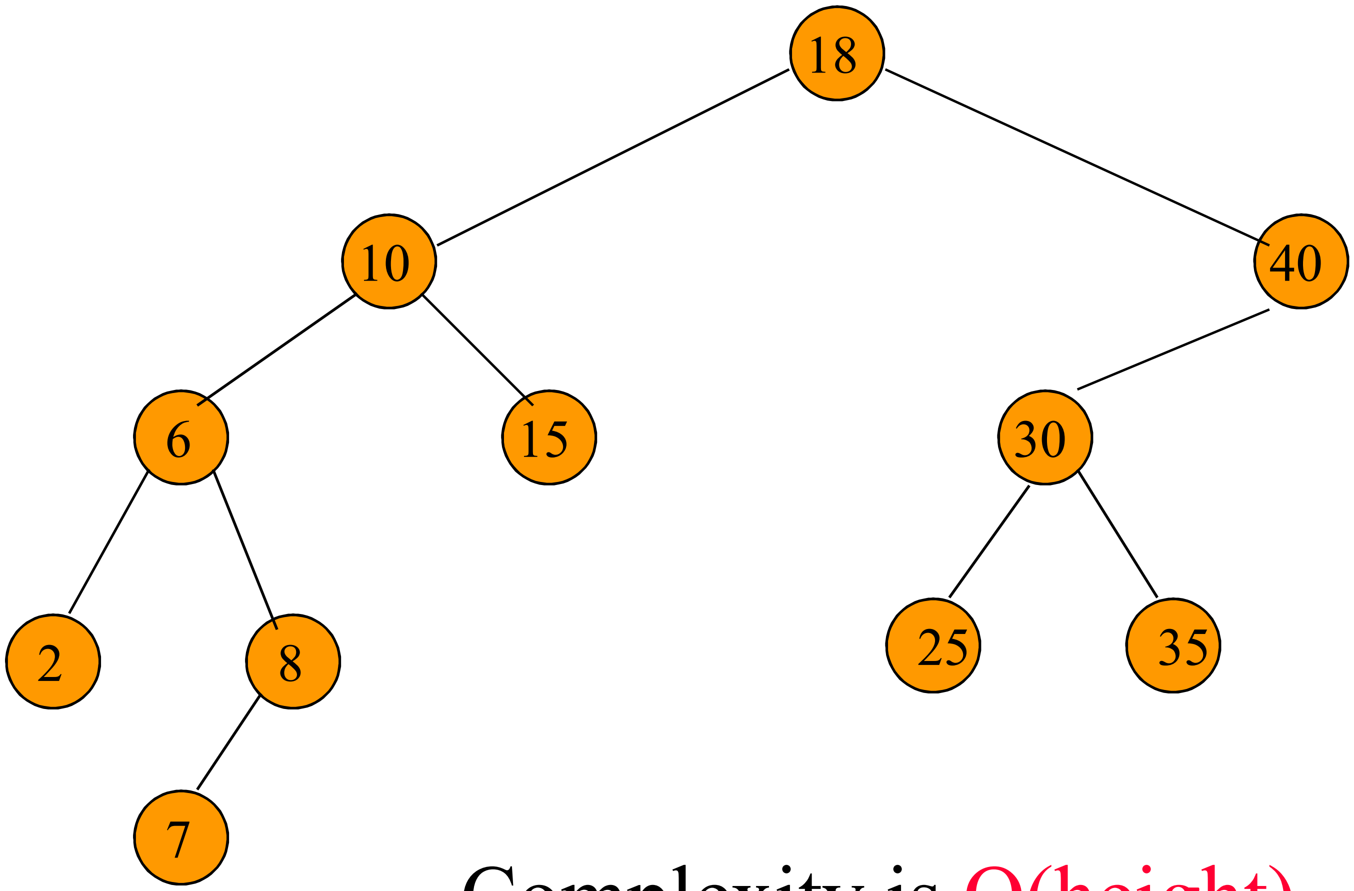
Another Example (cont.)



Another Example (cont.)



Delete a Degree 2 Node



Complexity is O(height).

Implementation

q is the parent of *p*

```
template <class K, class E>
void BST<K,E>::Delete(K k) {
  TreeNode<pair<K,E> > *p = root, *q = 0;
  while (p  && k ≠ p→data.first) {
    q = p;
    if (k < p→data.first) p = p→LeftChild;
    else p = p→RightChild;
  }
  if (p == 0) return; // not found
```

```
if (p→LeftChild == 0 && p→RightChild == 0) // p is leaf
{
  if (q == 0) root = 0;
  else if (q→LeftChild == p) q→LeftChild = 0;
  else q→RightChild = 0;
  delete p;
}

if (p→LeftChild == 0)  // p only has right child
{
  if (q == 0) root = p→RightChild;
  else if (q→LeftChild == p) q→LeftChild = p→RightChild;
  else q→RightChild = p→RightChild;
  delete p;
}
```

```
if (p→RightChild == 0)   // p only has left child
{
   if (q == 0) root = p→LeftChild;
   else if (q→LeftChild == p) q→LeftChild = p→LeftChild;
   else q→RightChild = p→LeftChild;
   delete p;
}
```

prevprev

prev

curr

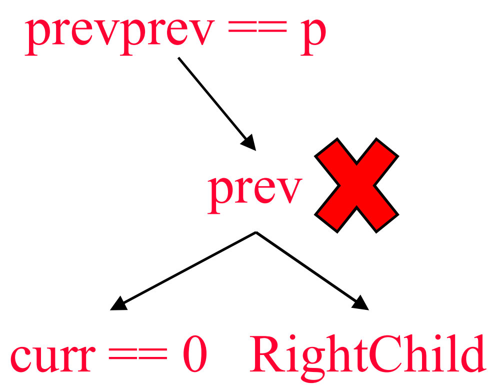
find the smallest node in the right subtree

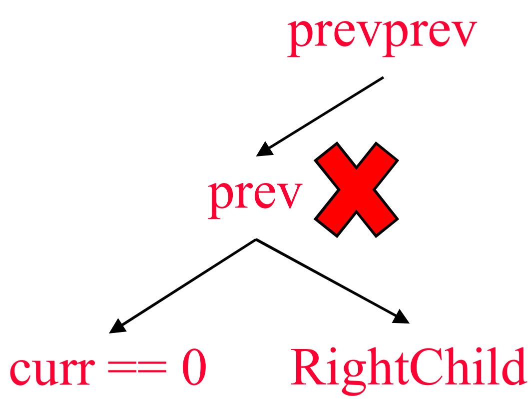
```
// p has left and right child.
TreeNode<pair<K,E> > *prevprev = p, *prev = p→RightChild,
      *curr = p→RightChild→LeftChild;

while (curr) {
   prevprev = prev;
   prev = curr;
   curr = curr→LeftChild;
}
```

prevprev

prev

curr






```
// curr is 0, prev is the node to be deleted, prevprev is prev's
// parent, prev->LeftChild is 0.

  p→data = prev→data;
  if (prevprev == p) prevprev→RightChild = prev→RightChild;
  else prevprev→LeftChild = prev→RightChild;
  delete prev;
}
```

Operations' Efficiency on B.S.T.

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)

Homework #2

Due: 9월 17일 자정까지

Implement and test

- Programs 5.18, 5.19, 5.21
- Exercise 5.7.1 (the delete function)