CSE221

Lecture 11:
Binary Search Trees

Fall 2021

Young-ri Choi

Acknowledgment: The content of this file is based on the slides of the textbook as well as the slides provided in former lectures at UNIST.

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

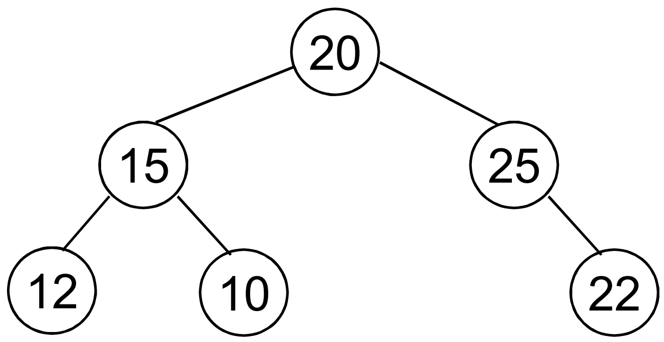
Outline

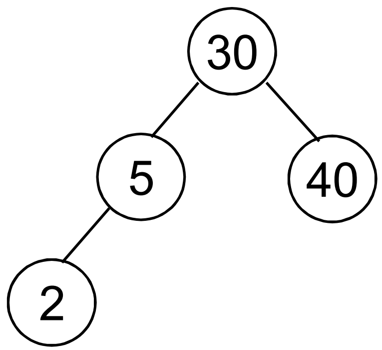
- Binary search trees
- Selection trees

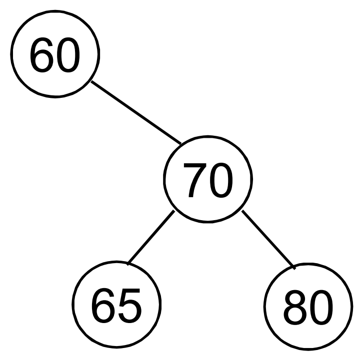
Binary Search Tree

- Definition
  - Every element has a key, and all keys are different
  - Keys in the left subtree are smaller than root
  - Keys in the right subtrees are larger than root
  - Left and right subtrees are also binary search tree

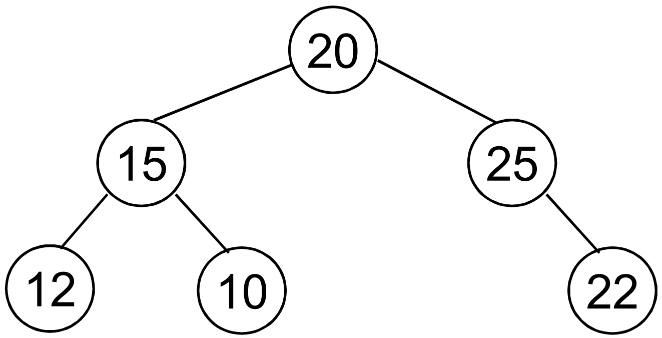
Binary Search Tree?



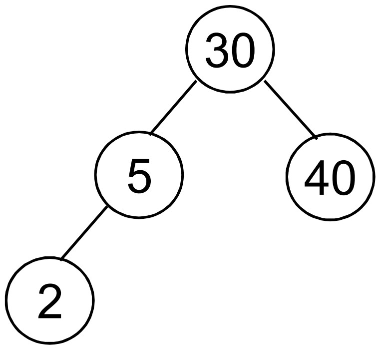




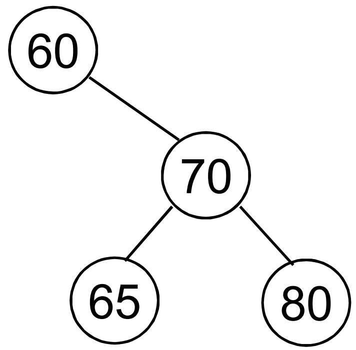
Binary Search Tree?



X



O



O

Searching by the Key Value

- Recursive

```
template<class Type>  // Driver
BstNode<Type> *BST<Type>::Search(const Element<Type> &x)
// Search the binary tree (*this) for a pair with key x
// return 0 if not found
{
    return search(root, x);
}

template<class Type>  // Workhorse
BstNode<Type> *BST<Type>::Search(BstNode<Type> *b,
                                 const Element<Type> &x)
{
   if (!b) return 0;
   if (x.key == b->data.key) return b;
   if (x.key < b->data.key)  return Search(b->LeftChild, x);
   return Search(b->RightChild, x);
}
```

O(h), where h is height of the tree

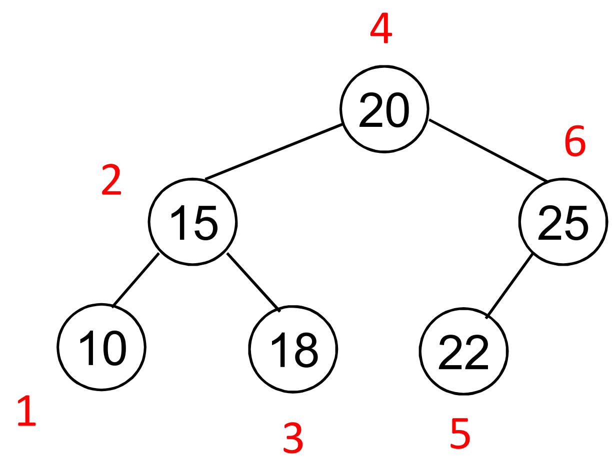
Searching by the Key Value

- Iterative

```
template <class Type>
BstNode<Type> *BST<Type>::IterSearch(const Element<Type> &x)
// Search x in (*this) binary search tree
{
   BstNode<Type> *t = root;
   while(t)
   {
     if (x.key == t->data.key) return t;
     if (x.key < t->data.key) t = t->LeftChild;
     else  t = t->RightChild;
   }
   return 0;
}
```

Searching by the Rank

- Rank
  - Node position in inorder traversial
- leftsize
  - 1 + # of nodes in left subtree



Rank = numerical order of each node's key

Searching by the Rank

```
template <class Type>
BstNode<Type> *BST<Type>::Search(int k)
// Find k-th smallest pair
{
   BstNode<Type> *t = root;
   while (t) {
     if (k == t->LeftSize) return t;
     if (k < t->LeftSize) t = t->LeftChild;
     else {
        k -= t->LeftSize;
        t = t->RightChild;
     }
   }
   return 0;
}
```

- O(h), where h is height of the tree

- Keep leftsize correct during insert/delete
- If leftsize is not maintained, just perform inorder traversal

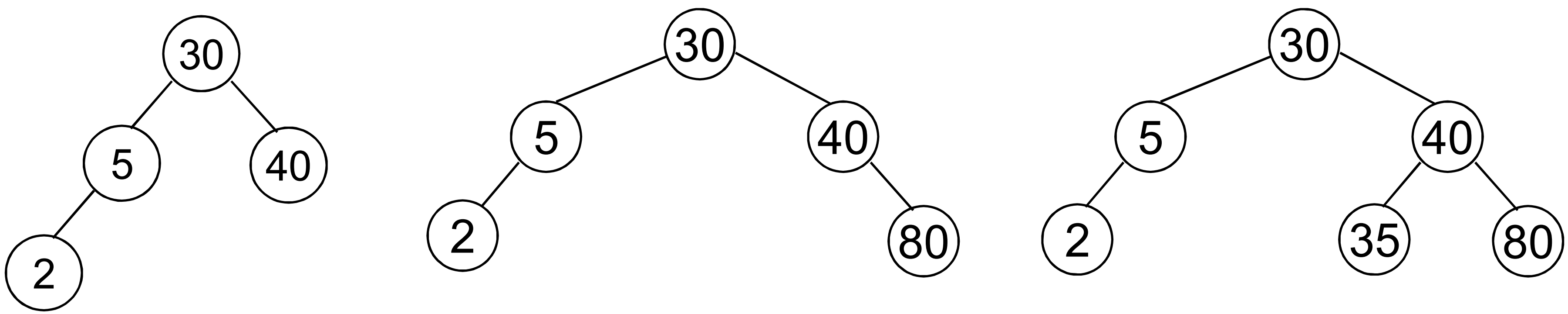
Use Case of the Rank

- Percentile: the value below which a given percentage of values are observed
- E.g., 50th-percentile: the value below which 50% of values are observed (i.e., median)
- What is the rank for the 90th-percentile on the binary search tree with N values?
  - Answer: 0.9 x N

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Insertion into a Binary Search Tree

- Note that every key has to be different!
- Search k
  - Failed : insert k where search terminated
  - Success : update element



(a) Insert 80 (b) Insert 35

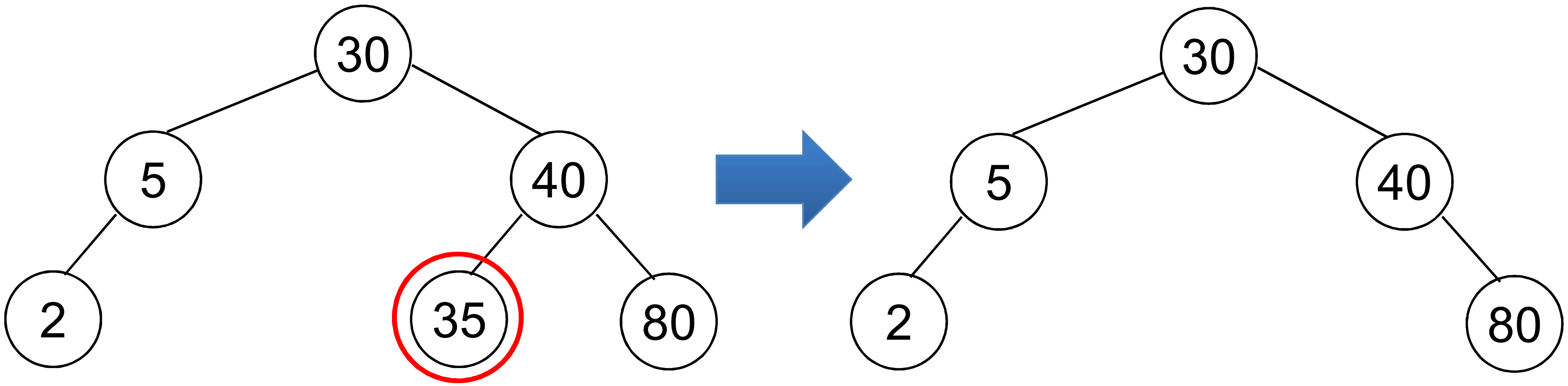
O(h), where h is height of the tree

Deletion from a Binary Search Tree

- Possible status of a node to delete
  - Leaf node
  - One child
  - Two children

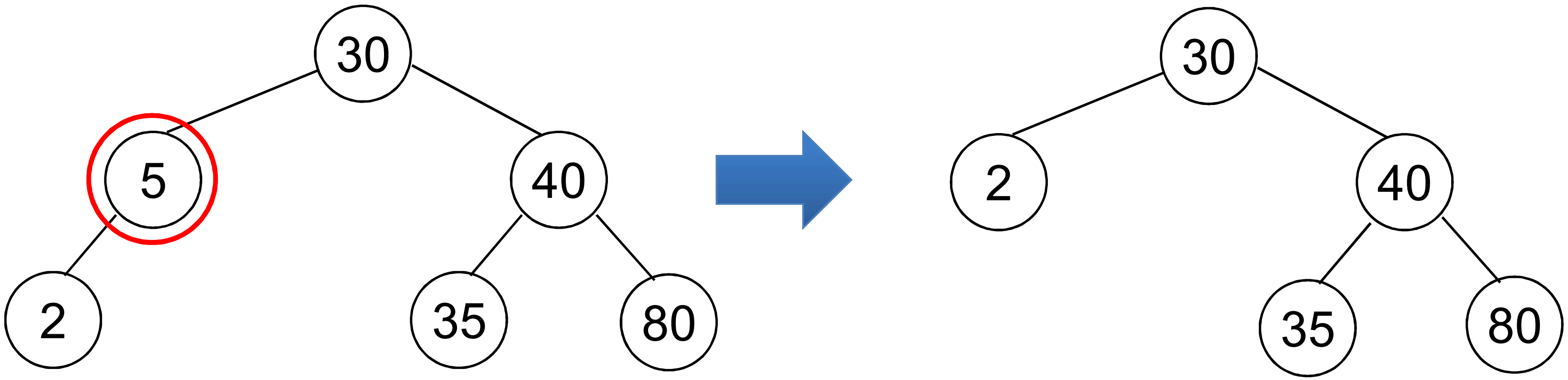
Deletion from a Binary Search Tree

- Leaf node
  - Simply remove the node from its parent
  - e.g., delete 35

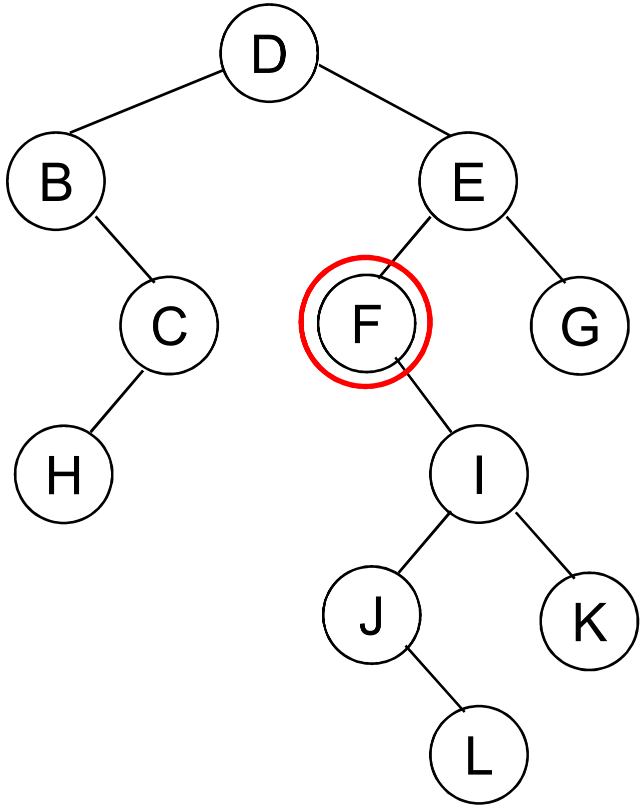


Deletion from a Binary Search Tree

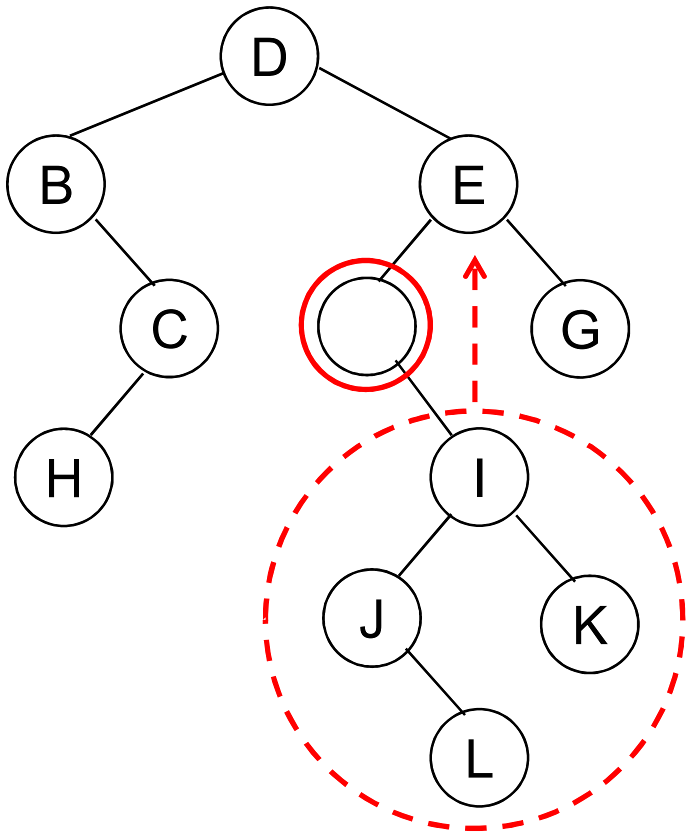
- Non-leaf node with one child
  - Child takes place of the deleted node
  - e.g., delete 5



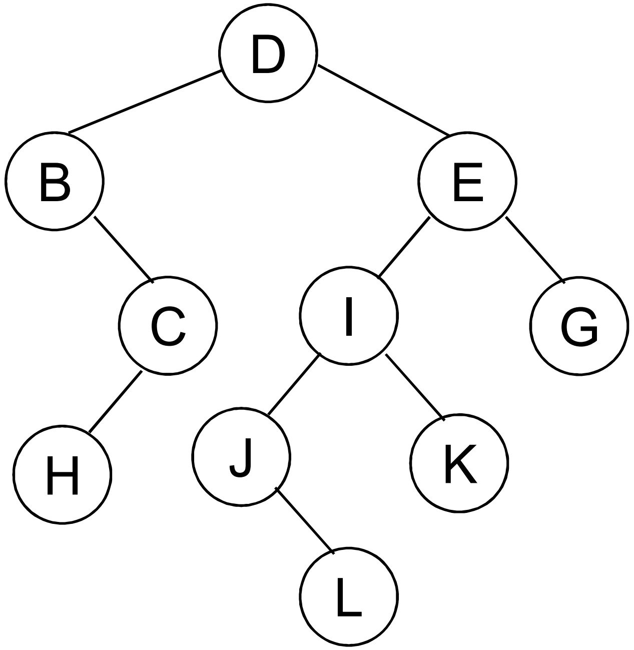
Deletion from a Binary Search Tree



Deletion from a Binary Search Tree

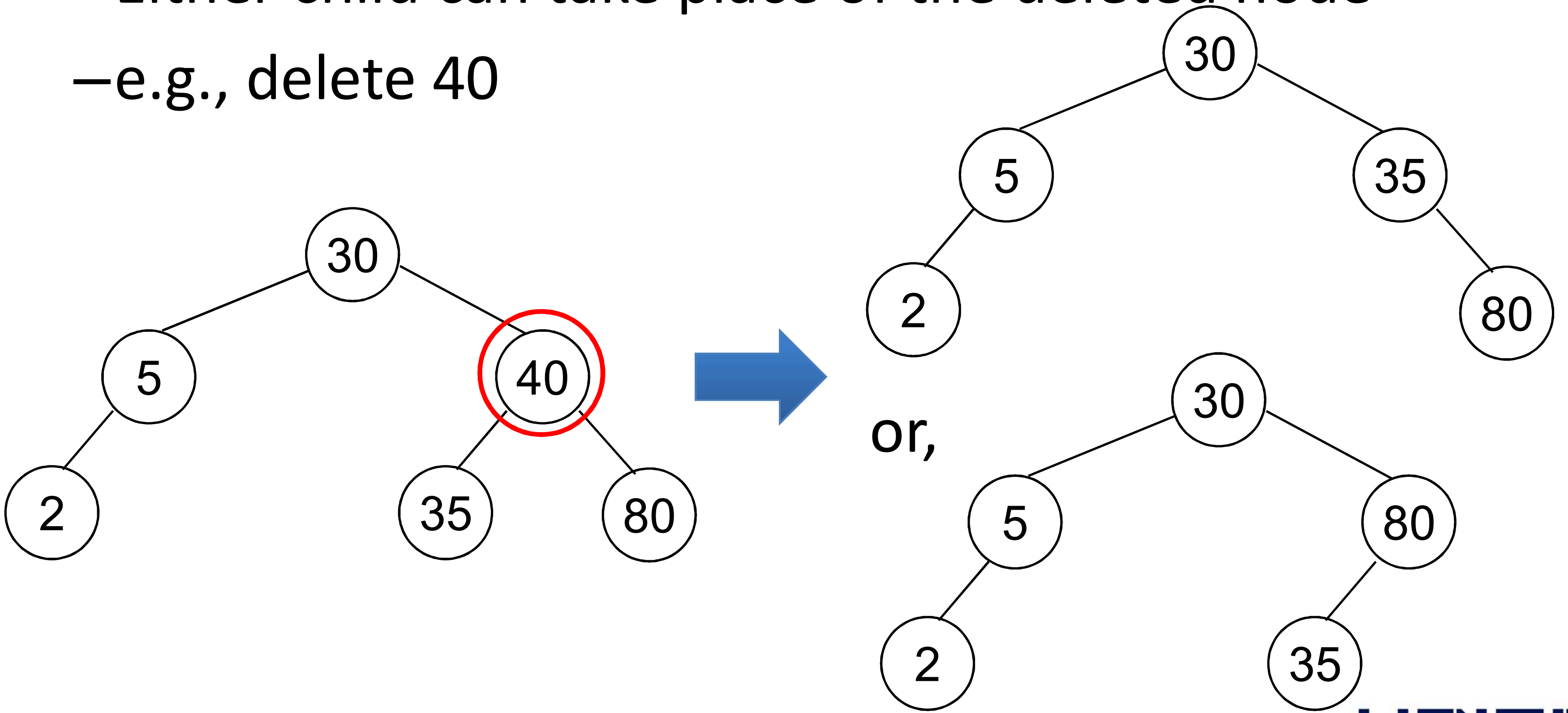


Deletion from a Binary Search Tree



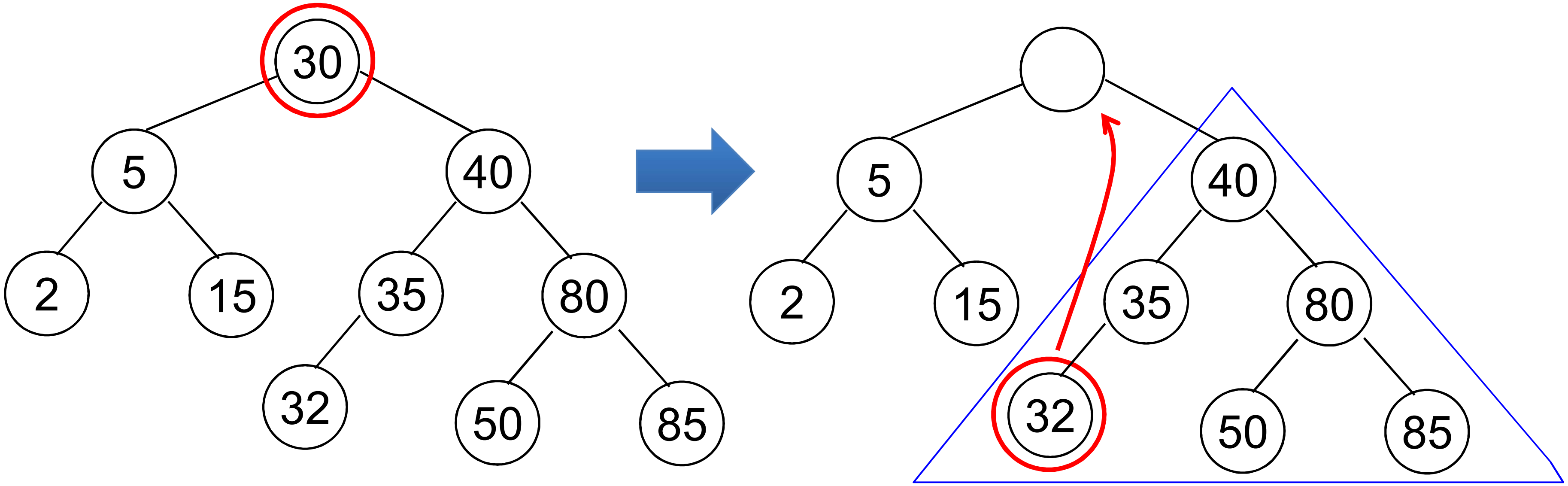
Deletion from a Binary Search Tree

- Non-leaf node with two children
  - Either child can take place of the deleted node
  - e.g., delete 40



Deletion from a Binary Search Tree

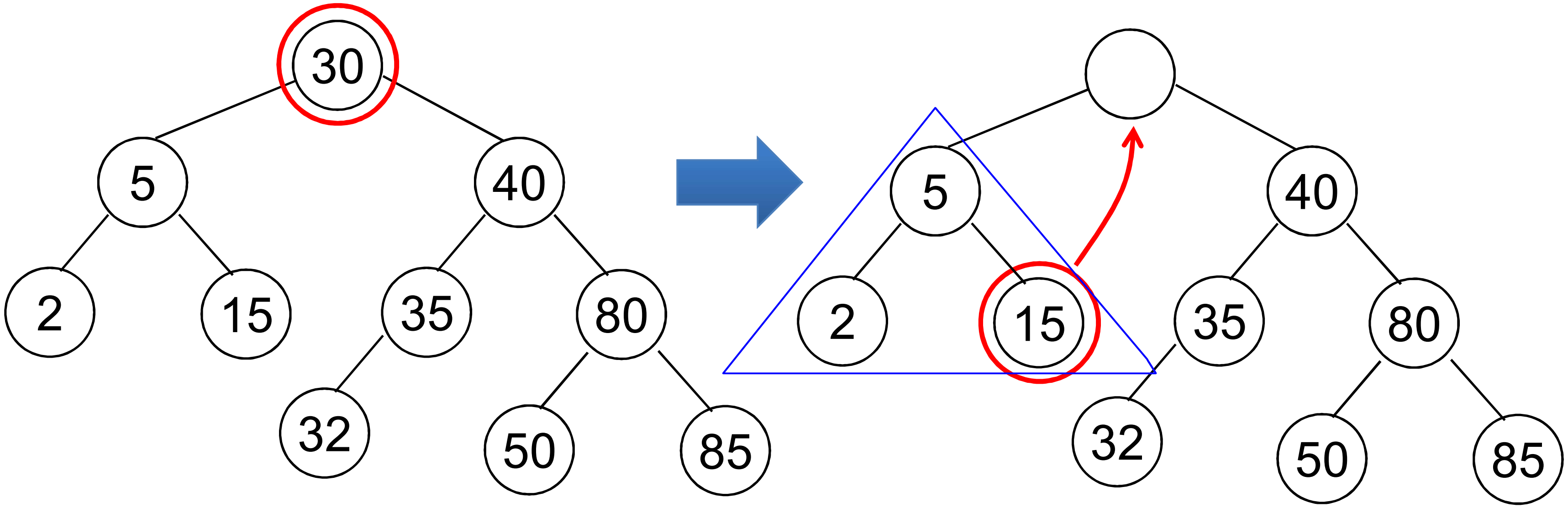
- Non-leaf node with two subtrees
  - Replace with max/min node



delete min from right subtree

Deletion from a Binary Search Tree

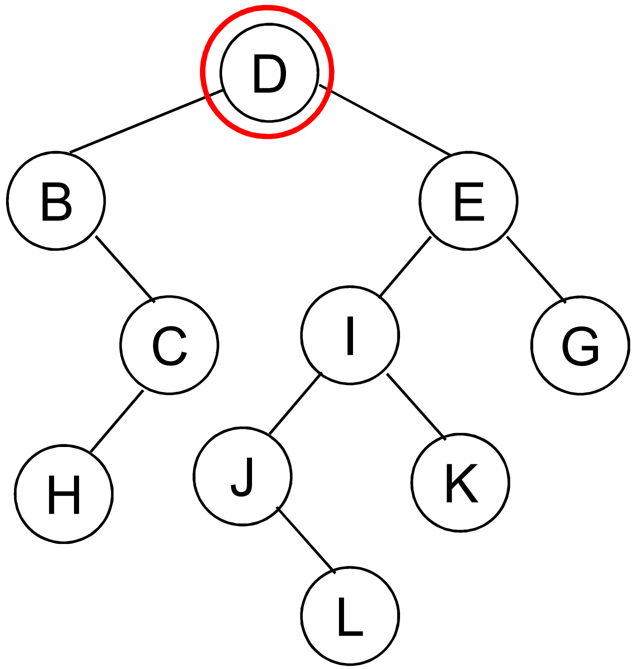
- Non-leaf node with two subtrees
  - Replace with max/min node



delete max from left subtree

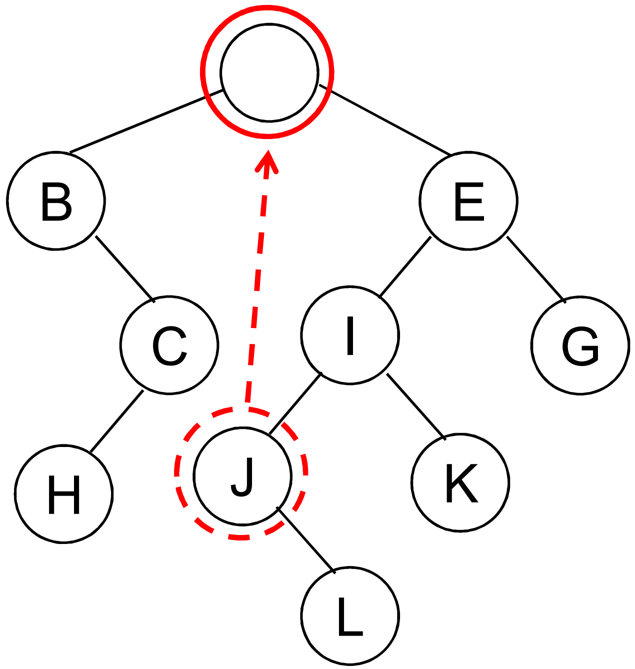
Deletion from a Binary Search Tree

- Non-leaf node with two subtrees
  - If min/max node is not a leaf, apply general rule again



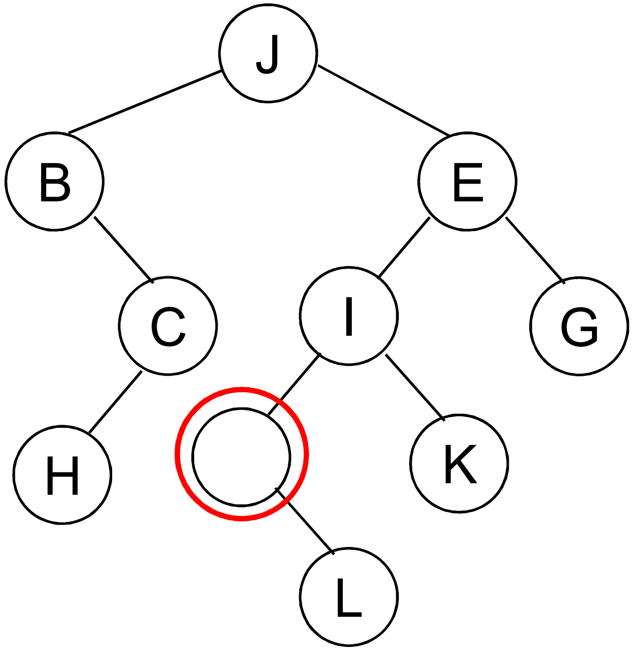
Deletion from a Binary Search Tree

- Non-leaf node with two subtrees
  - If min/max node is not a leaf, apply general rule again



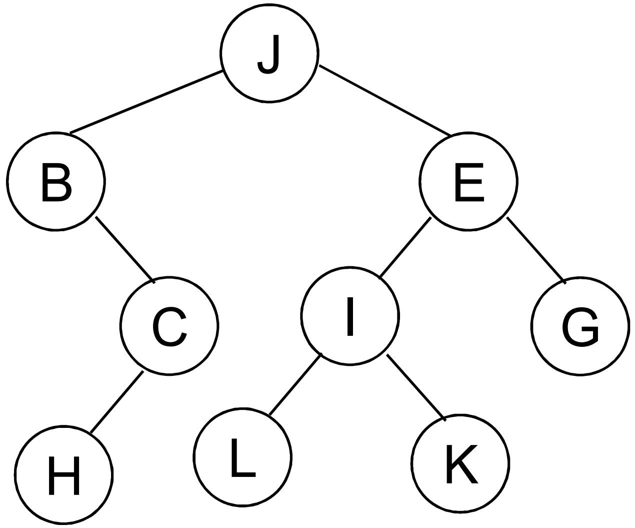
Deletion from a Binary Search Tree

- Non-leaf node with two subtrees
  - If min/max node is not a leaf, apply general rule again



Deletion from a Binary Search Tree

- Non-leaf node with two subtrees
  - –If min/max node is not a leaf, apply general rule again

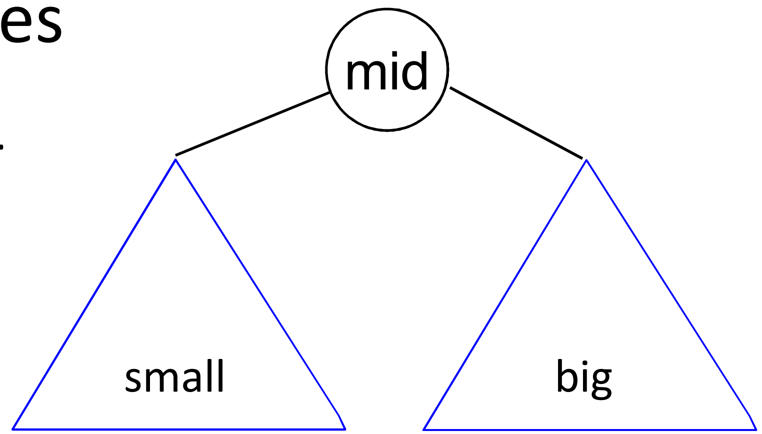


Joining and Splitting Binary Trees

- Three-way join
  - Two binary search trees and a mid value are merged as a single binary search tree
- Two-way join
  - Two binary search trees are merged to a single binary search tree
- Split
  - A single binary search tree is split into two binary search trees with respect to a given mid value

Three-way Join

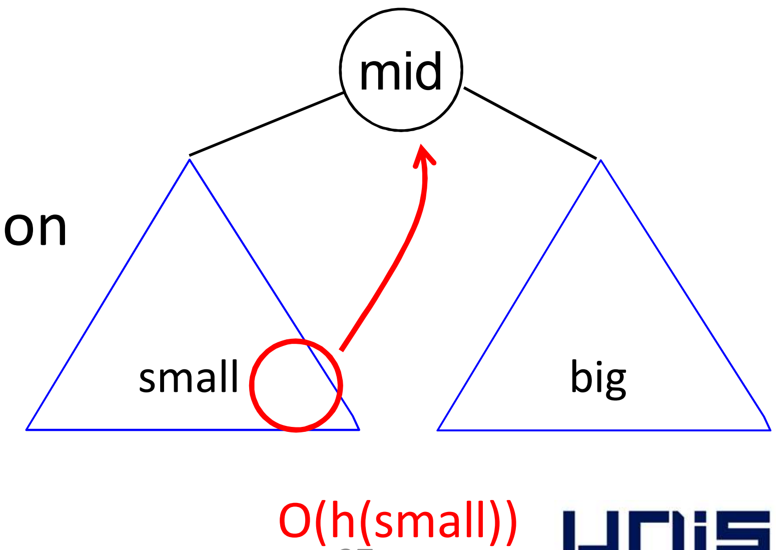
- Input
  - Two binary search trees (small, big) and mid value
- Algorithm
  - Create a mid node and attach small / big binary trees as left / right subtrees
  - h= max(h(small),h(big))+1



O(1)

Two-way join

- Input
  - Two binary search tree (small, big)
- Algorithm
  - Find mid by searching largest (smallest) key in small (big) and three way join
  - h= max(h'(small),h(big))+1
    - h': height of small after deletion

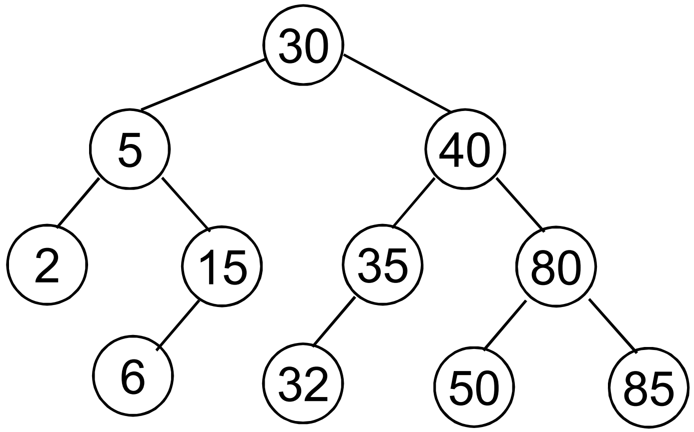


Split

- Input
  - A binary search tree and the mid value k
- Output
  - Two binary search trees (small and big), mid node if k exists

Split

k = 35



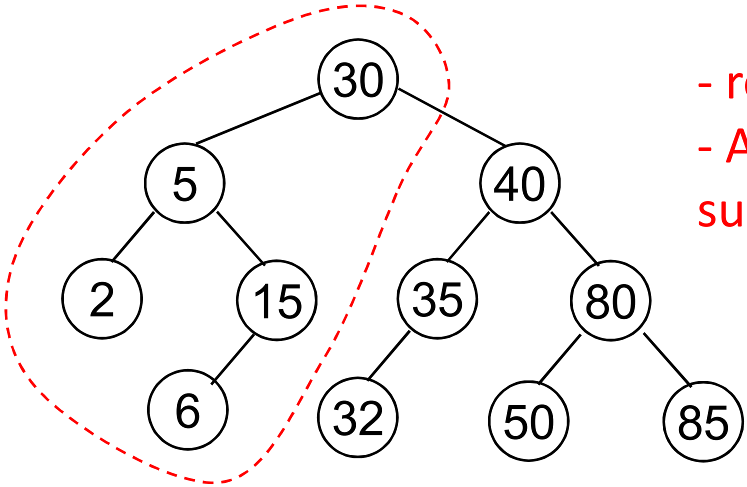
small

big

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Split

k = 35



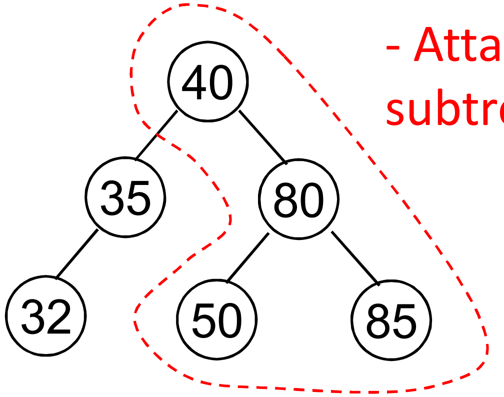
- root = 30 < k
- Attach root & left subtree to small

small

big

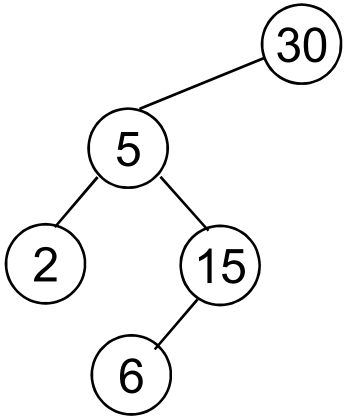
Split

k = 35



- root = 40 > k
- Attach root & right subtree to big

small

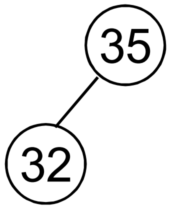


big

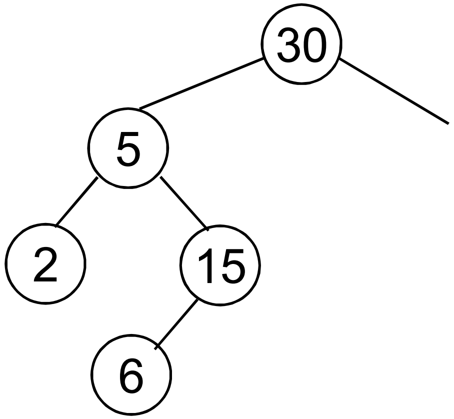
Split

k = 35

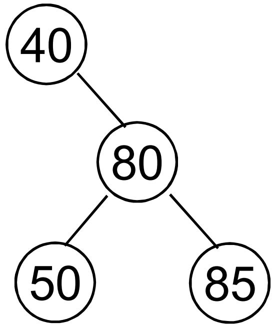
- root = 35 = k
- Attach left subtree to small and right subtree to big



small



big

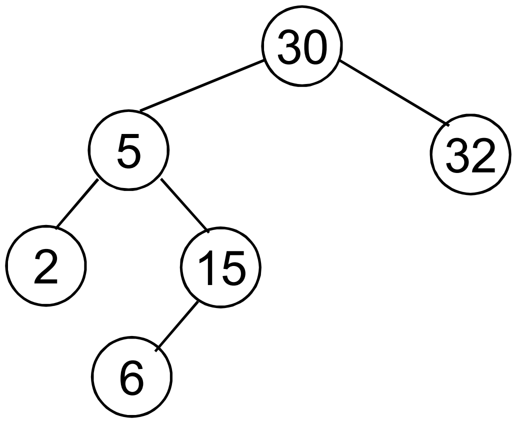


Split

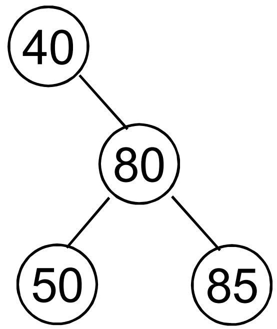
k = 35

- root = 35 = k
- Return 35 as mid

small



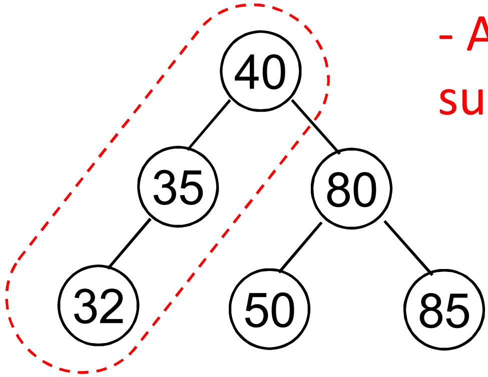
big



Split

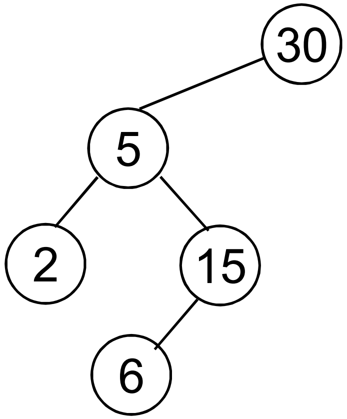
What if k = 80?

- root = 40 < k
- Attach root & left subtree to small



small

big



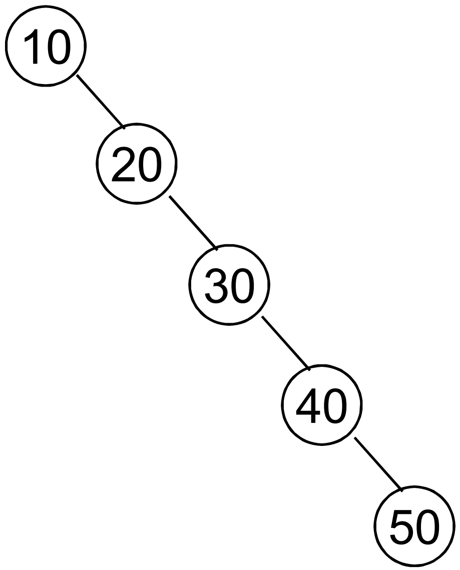
Split Algorithm

- Start with empty small and big binary search trees
- Repeat the following process:
  - If k > root, attach root + left subtree to small
    - Perform two-way join with current small: incoming small is bigger
  - If k < root, attach root + right subtree to big
    - Perform two-way join with current big: incoming big is smaller
  - If k = root, attach left subtree to small and right subtree to big, and return a node containing k as mid

Split

- Worst case
  - Each split, only one node (root) will be attached to small

k = 50



ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Split

- Complexity
  - O(h(input))
  - h(small), h(big) <= h(input)

Height of a Binary Search Tree

- Worst case
  - O(n)
  - e.g., 1,2,3,4,5,..,n (right skewed tree)
- Best case
  - O(log n)
  - e.g., complete/full binary tree
- Balanced search tree
  - Search tree with worst-case height O(log n)