CSE221

Lecture 13:
Binary Search Trees

Hyungon Moon

Acknowledgment: The content of this file is based on the slides of the textbook as well as the slides provided by Prof. Won-Ki Jeong.

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Outline

- Binary search trees
- Selection trees

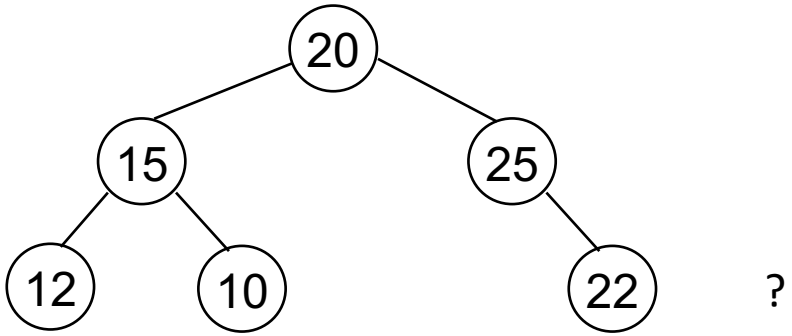
Outline

- Binary search trees
- Selection trees

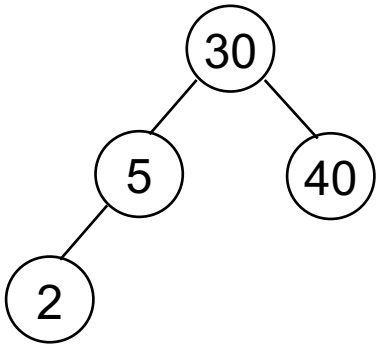
Binary Search Tree

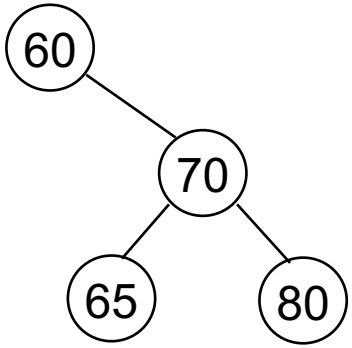
- Definition
  - Every element has a key, and all keys are different
  - Keys in the left subtree are smaller than root
  - Keys in the right subtrees are larger than root
  - Left and right subtrees are also binary search tree

Binary Search Tree

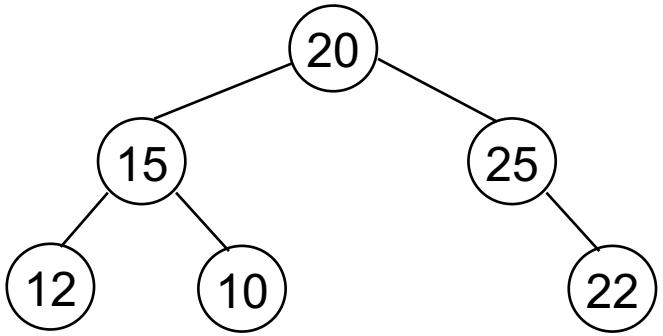


?

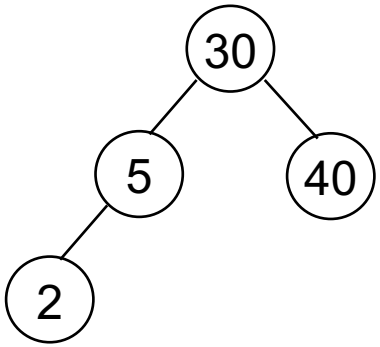




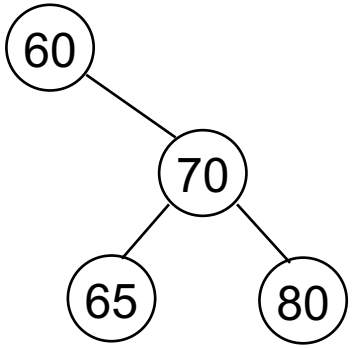
Binary Search Tree



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(?)

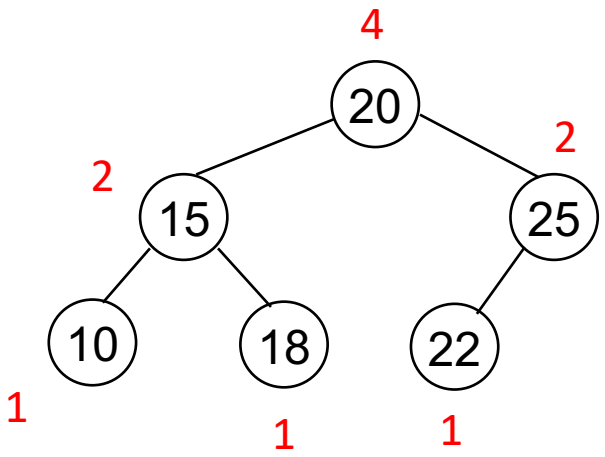
Searching by the Key Value

- 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
- leftsize
  - 1 + # of nodes in left subtree



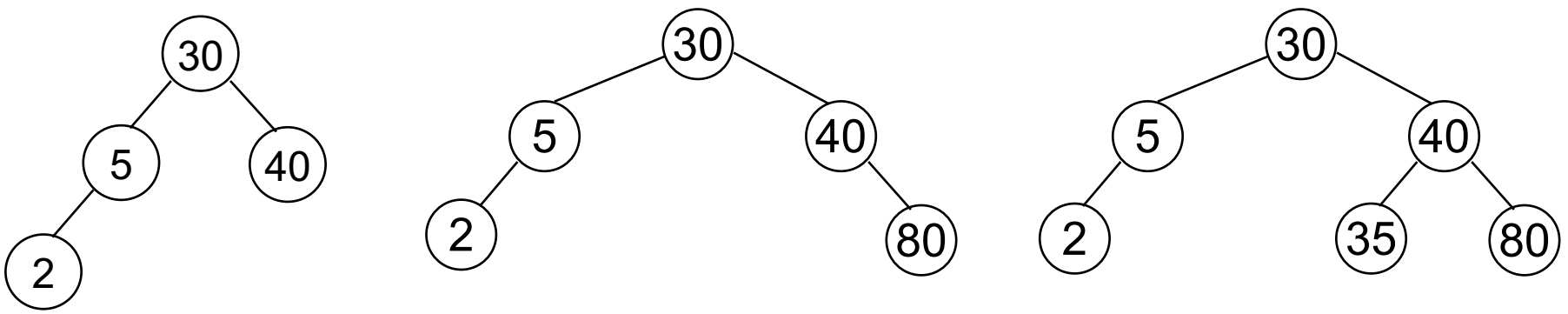
Searching by the Rank

```
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(?)

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
  - O(*h*), *h* : height of the tree

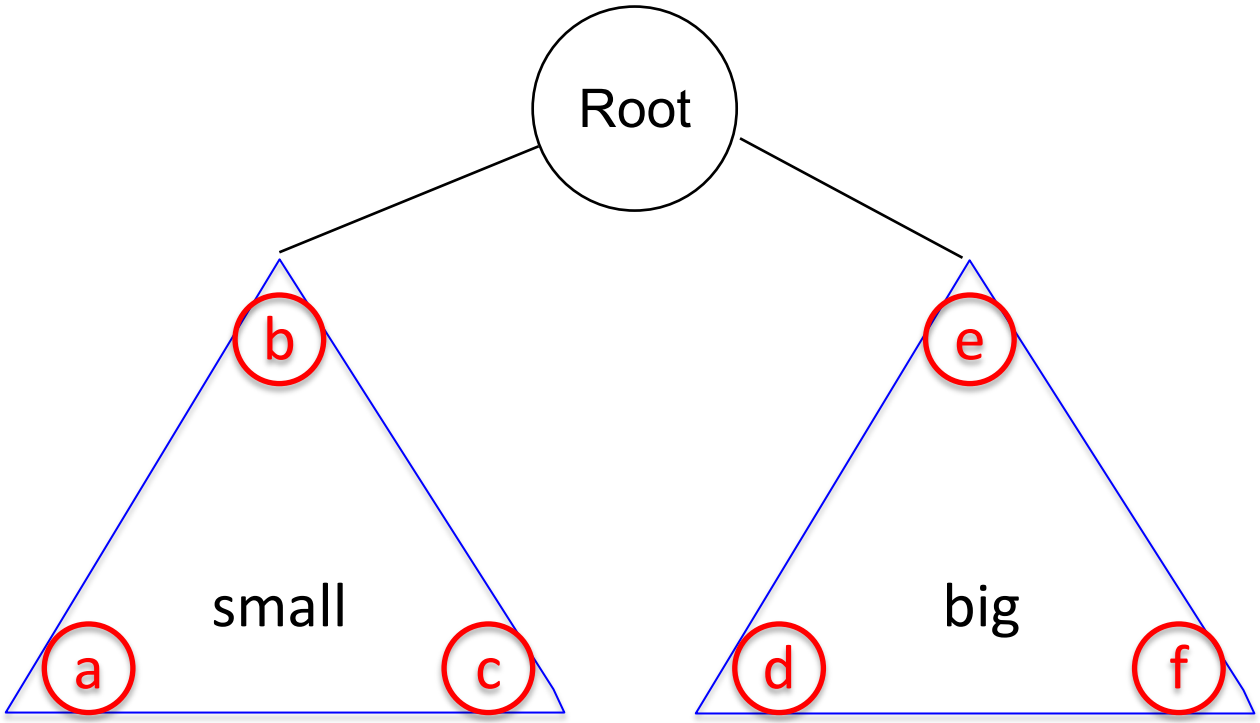


(a) Insert 80

(b) Insert 35

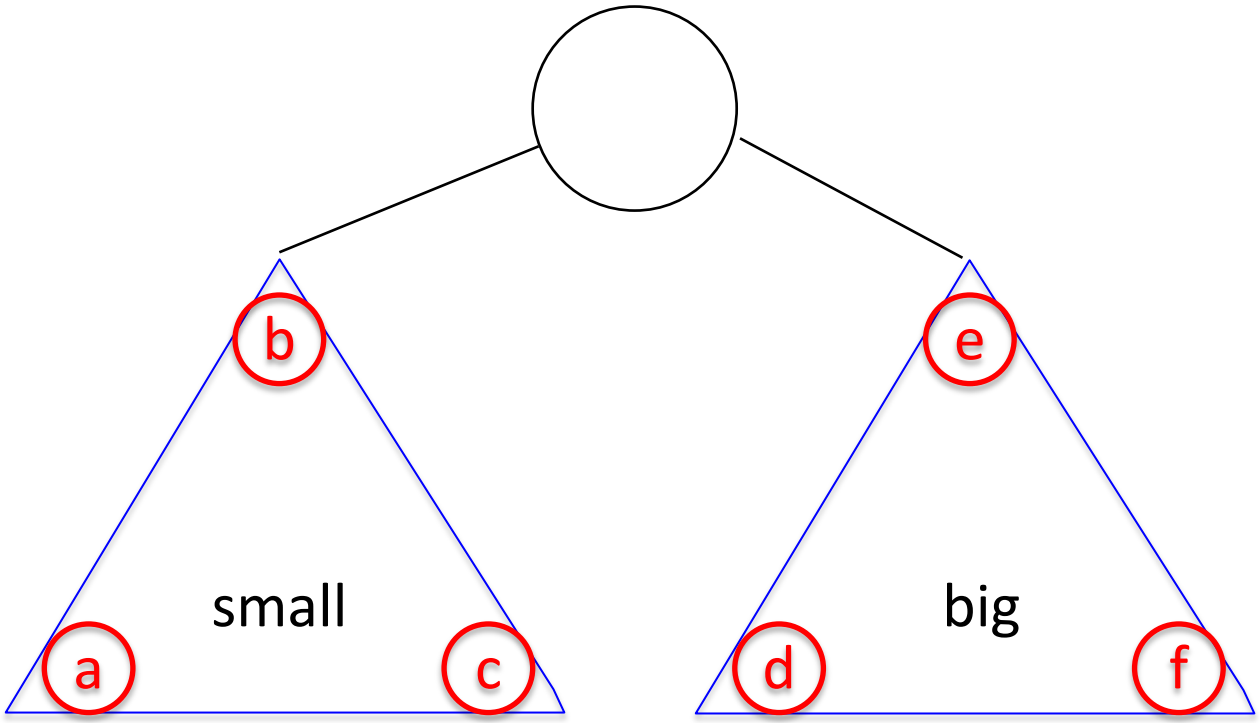
Deletion from a Binary Search Tree

- Left subtree < root < right subtree
  - a < b < c < root < d < e < f



Deletion from a Binary Search Tree

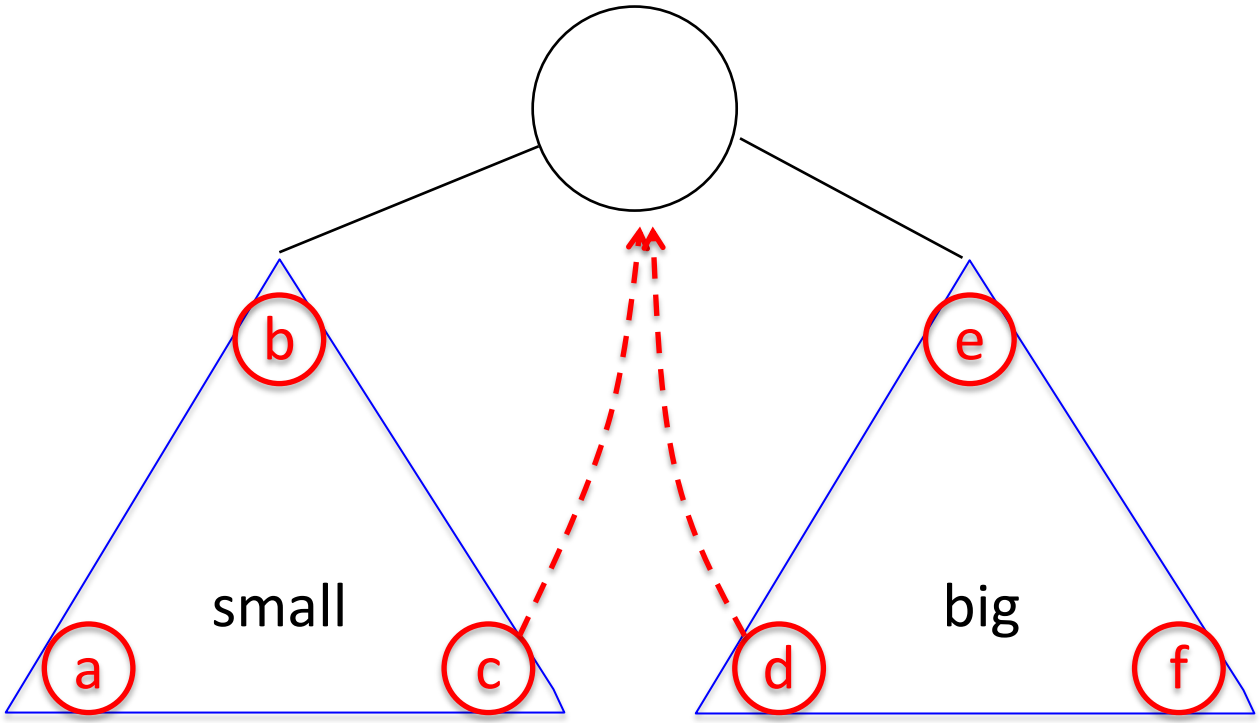
- a < b < c < ~~root~~ < d < e < f



Delete a node that has two children

Deletion from a Binary Search Tree

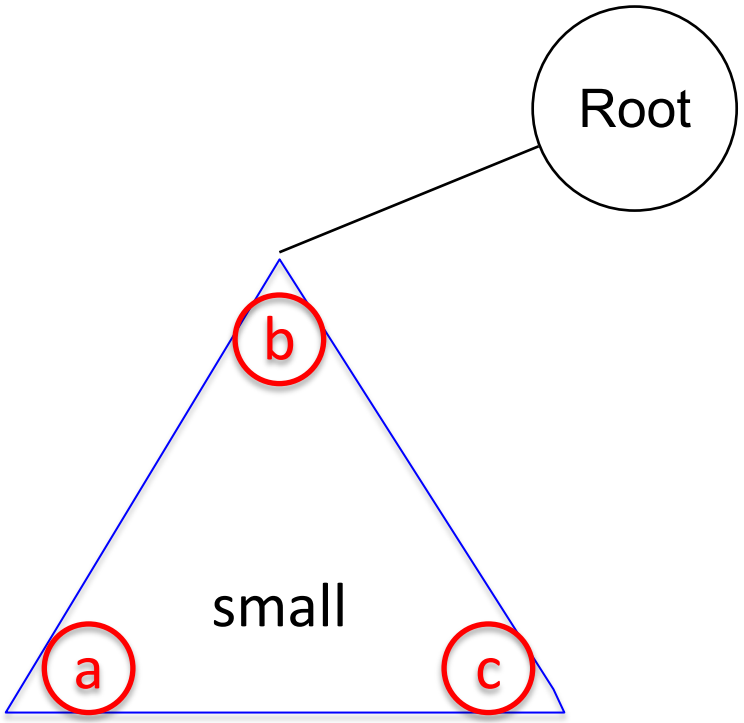
- a < b < c < d < e < f
  - Move c or d to root



Delete a node that has two children

Deletion from a Binary Search Tree

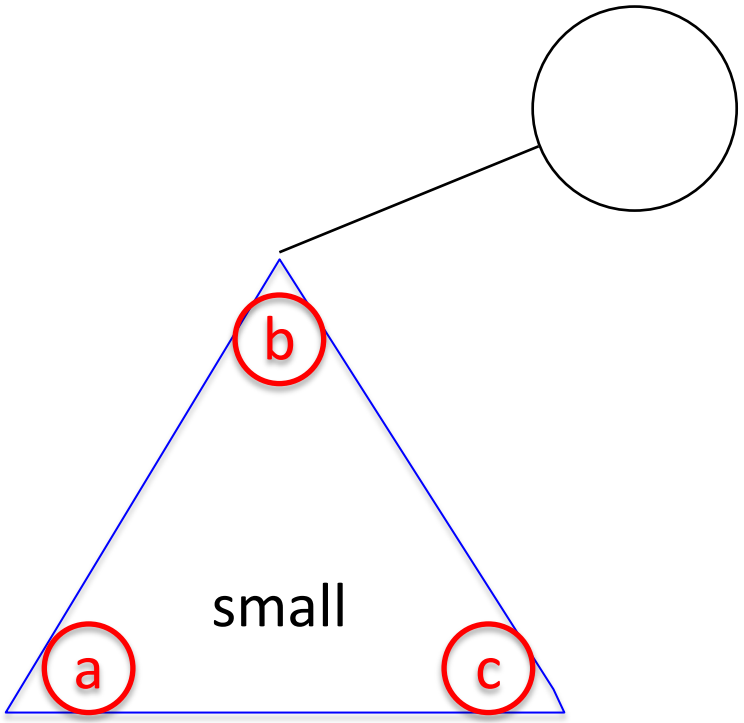
- Left subtree < root
  - a < b < c < root



Delete a node that has one child

Deletion from a Binary Search Tree

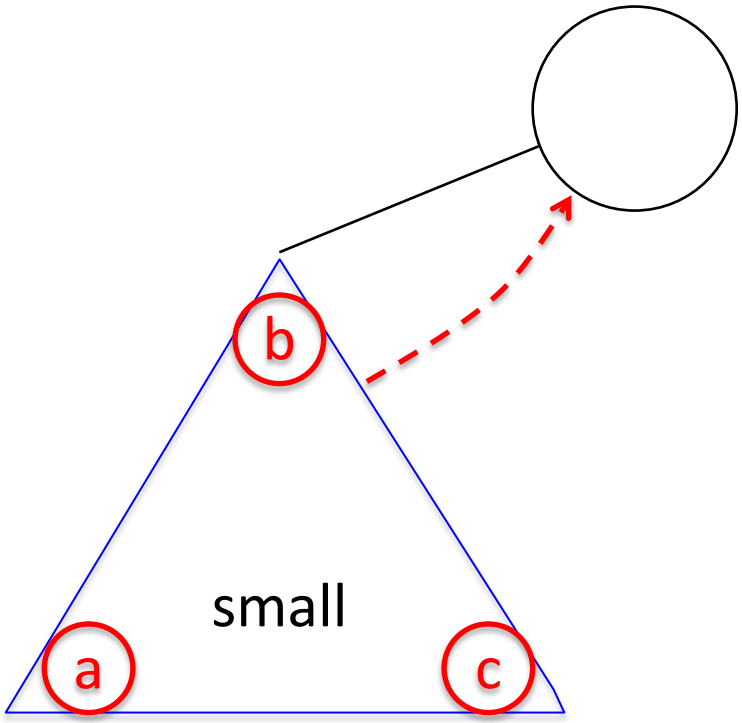
- a < b < c < ~~root~~



Delete a node that has one child

Deletion from a Binary Search Tree

- a < b < c < ~~root~~
  - b is a new root



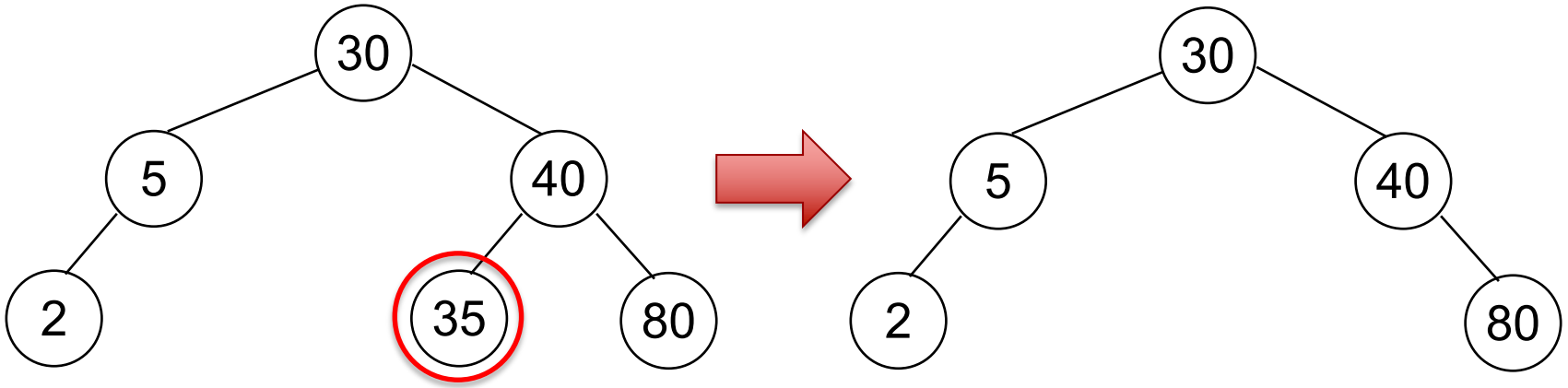
Delete a node that has one child

Deletion from a Binary Search Tree

- General rule
  - Leaf node
    - Delete it
  - One child
    - Replace with its child
  - Two children
    - Replace with min/max node of a subtree
    - If min/max node is not a leaf, apply general rule again

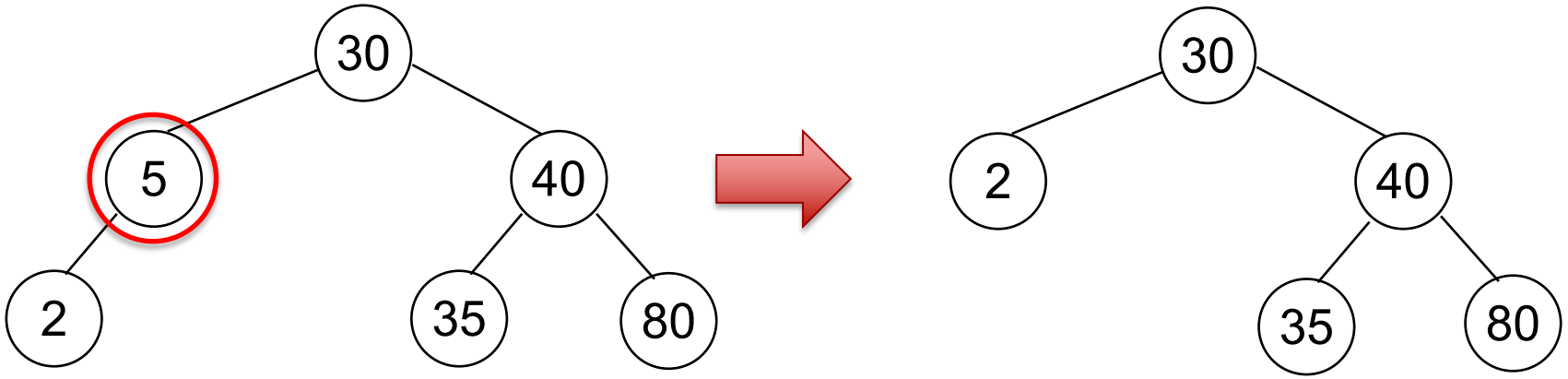
Deletion from a Binary Search Tree

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



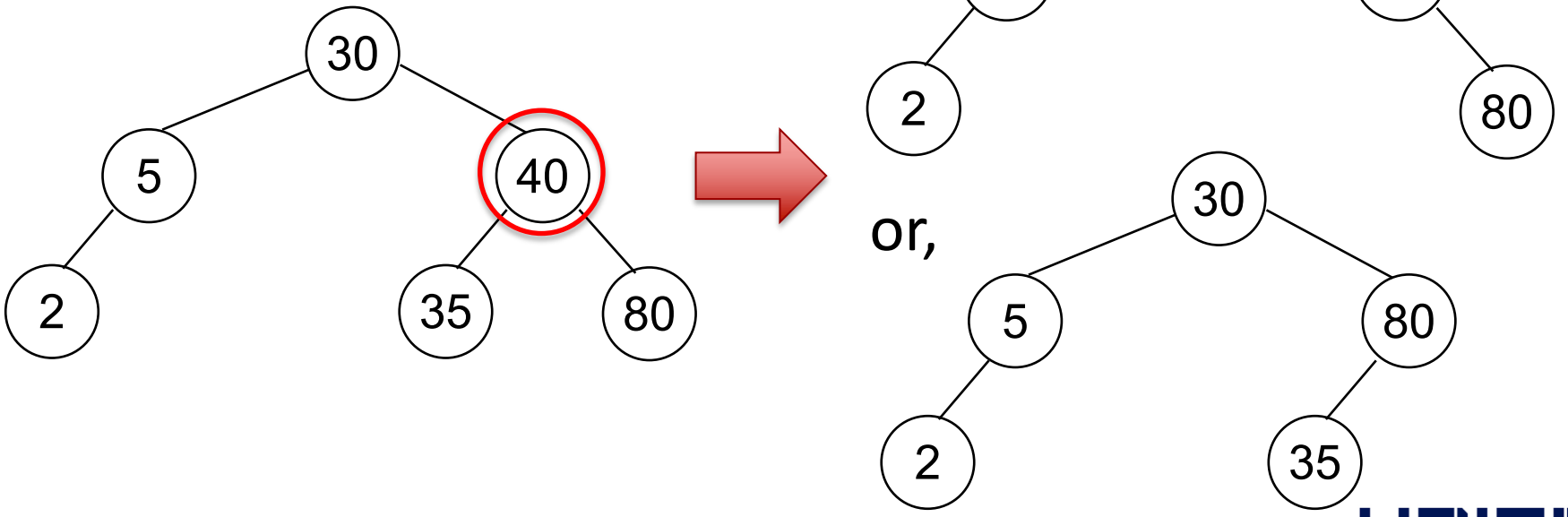
Deletion from a Binary Search Tree

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



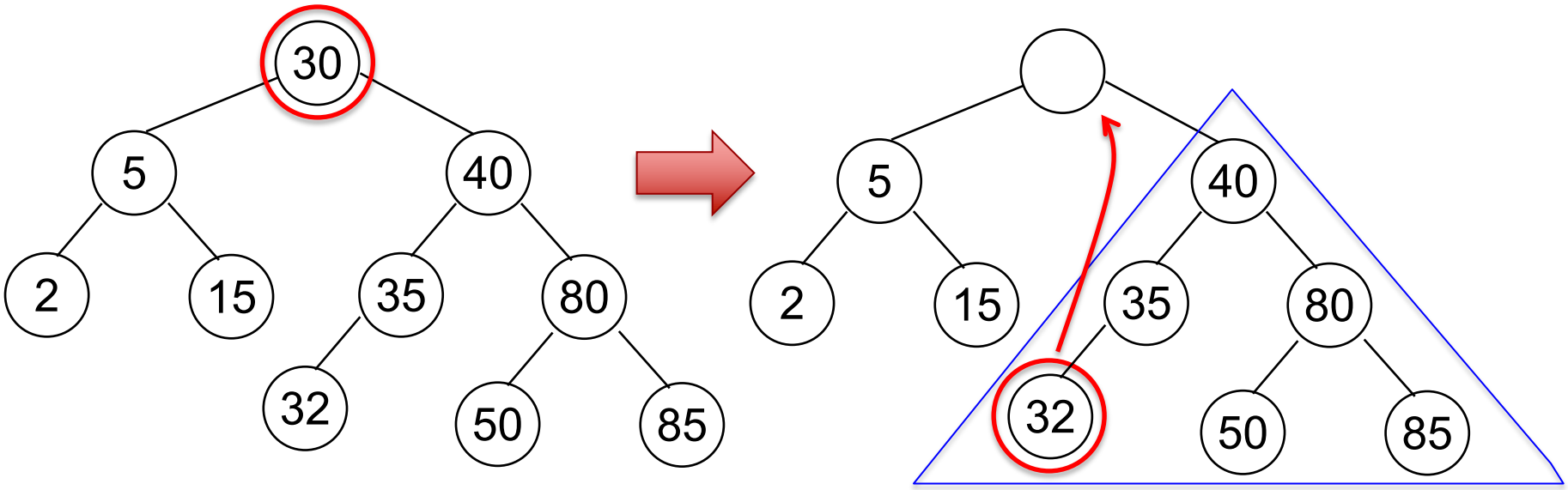
Deletion from a Binary Search Tree

- 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

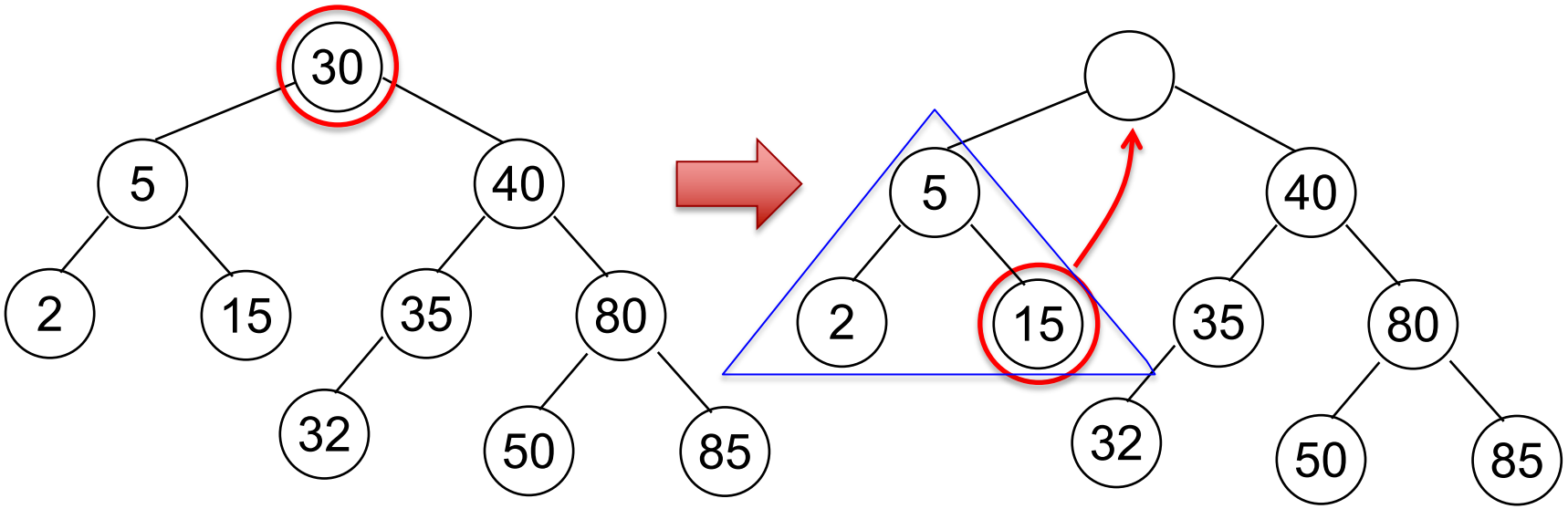
- Non leaf node with two subtrees
  - Replace with max/min node



delete min from right subtree

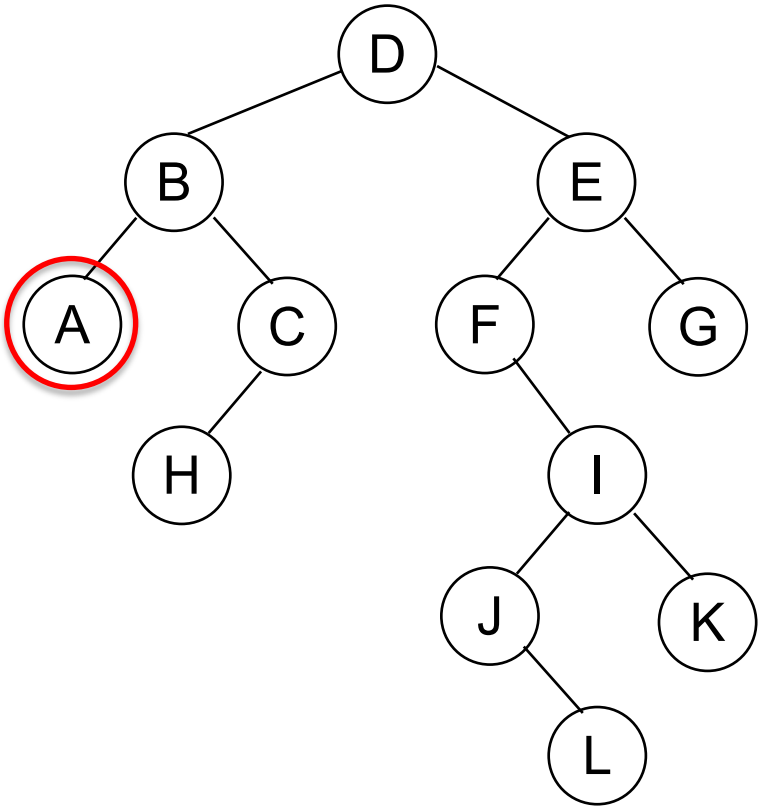
Deletion from a Binary Search Tree

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



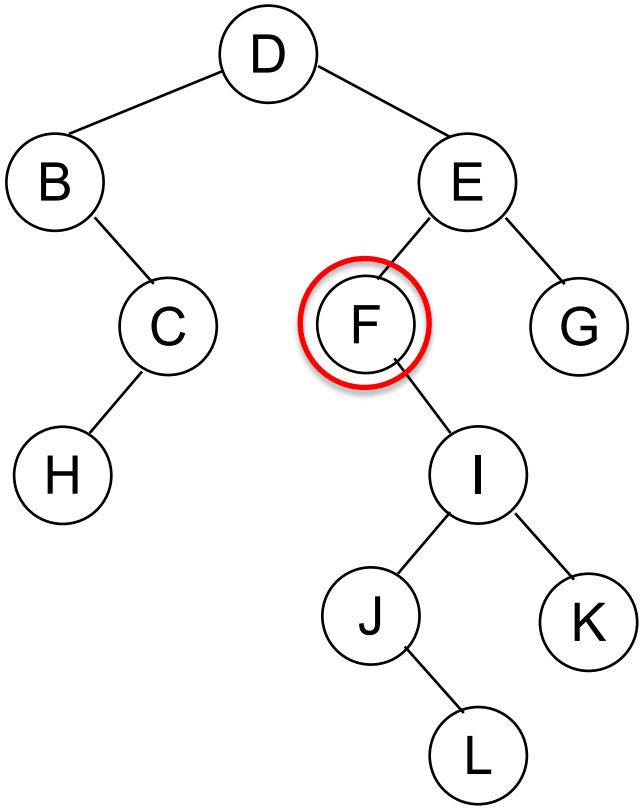
delete max from left subtree

Deletion from a Binary Search Tree

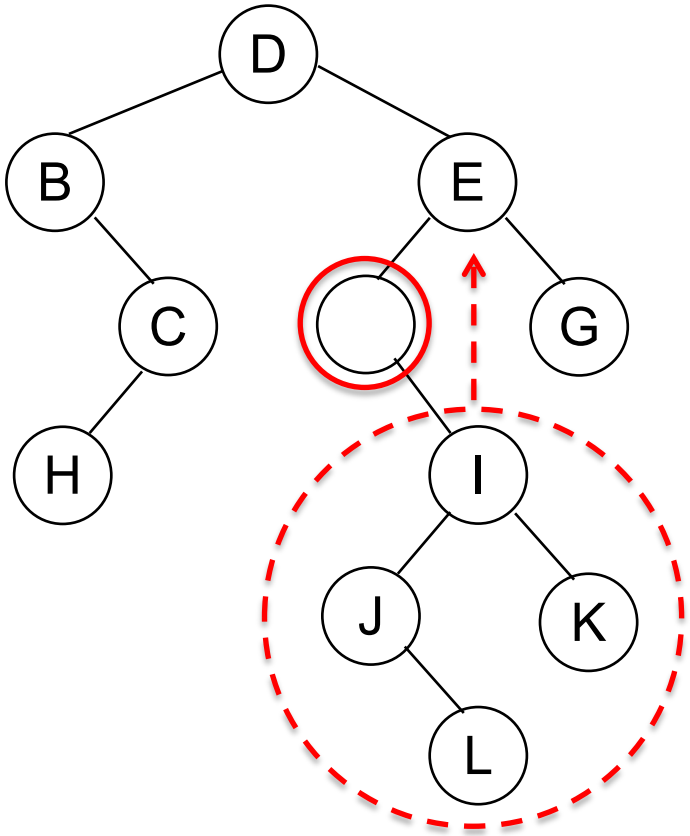


A, B, C, … are not the order of the node

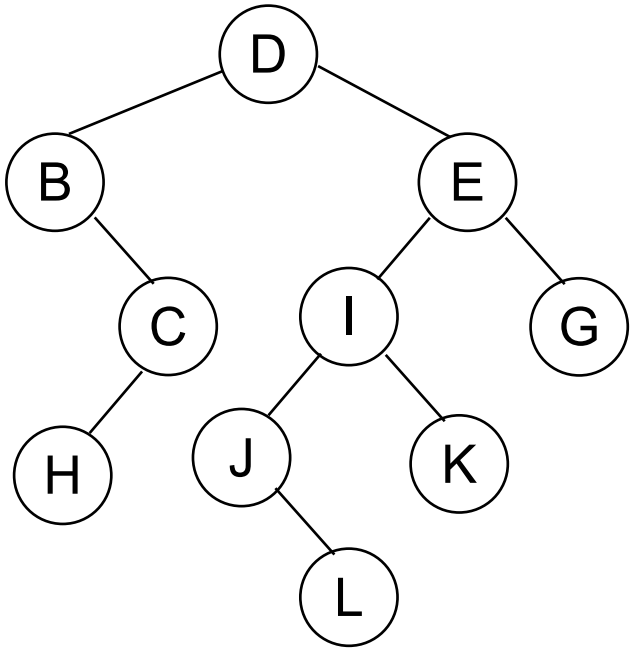
Deletion from a Binary Search Tree



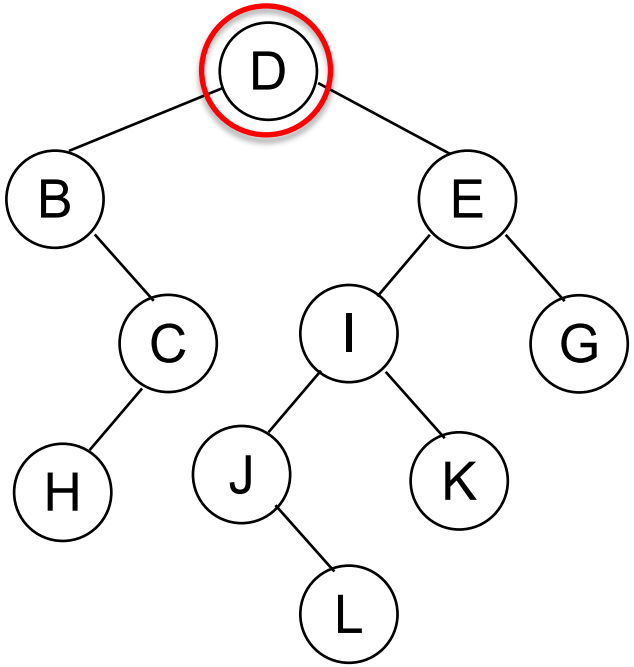
Deletion from a Binary Search Tree



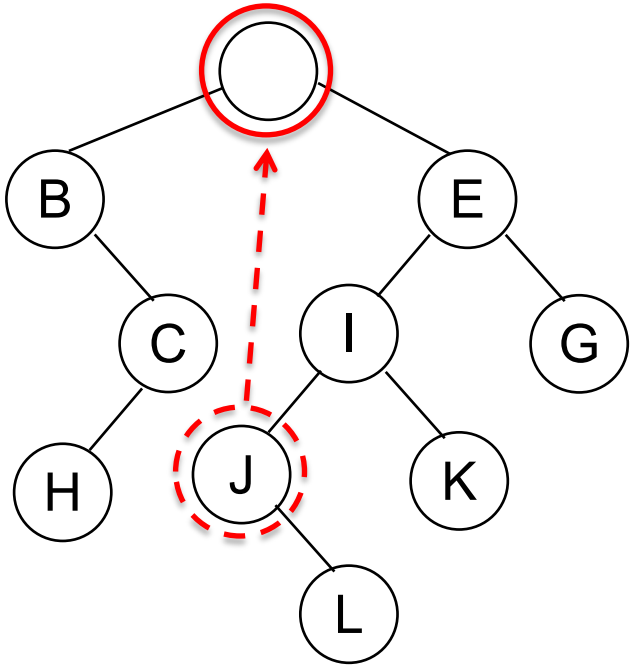
Deletion from a Binary Search Tree



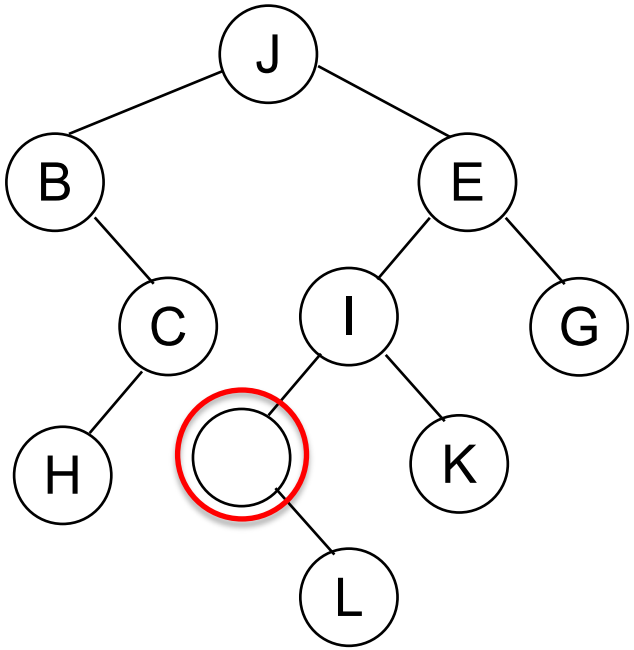
Deletion from a Binary Search Tree



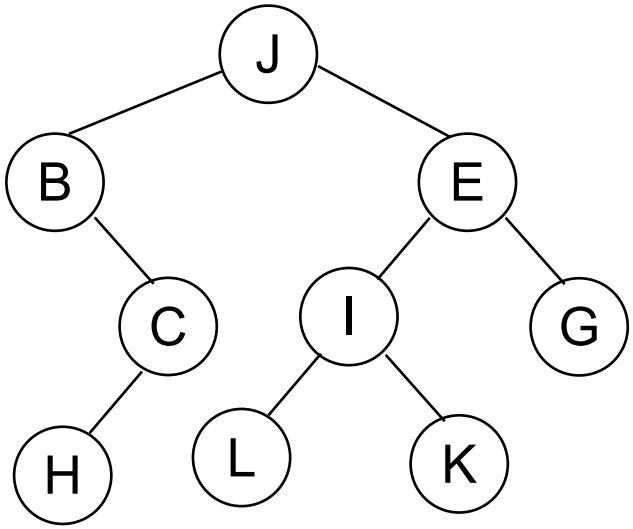
Deletion from a Binary Search Tree



Deletion from a Binary Search Tree



Deletion from a Binary Search Tree

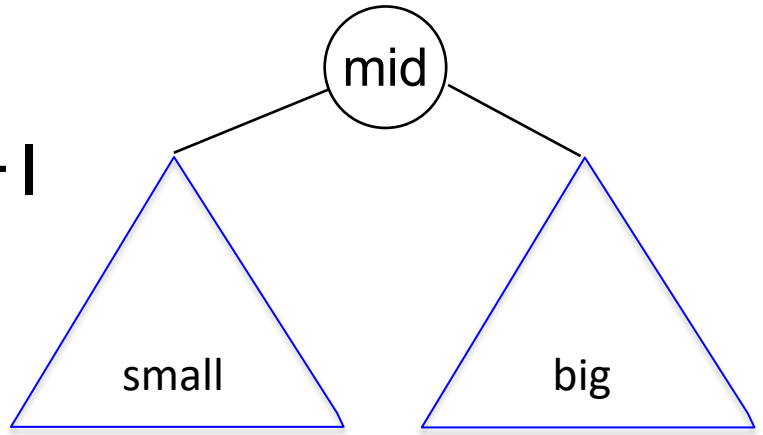


Joining and Splitting Binary Trees

- 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

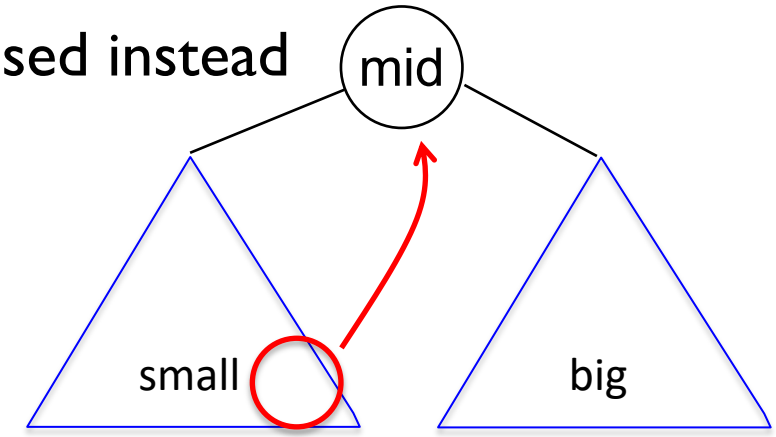
Three-way join

- 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
  - O(1)
  - h= max(h(small),h(big))+1



Two-way join

- Input
  - Two binary search tree (small, big)
- Algorithm
  - Find mid by searching largest key in small and three way join
    - Smallest key in big can be used instead
  - O(h(small))
  - h= max(h'(small),h(big))+1



Split

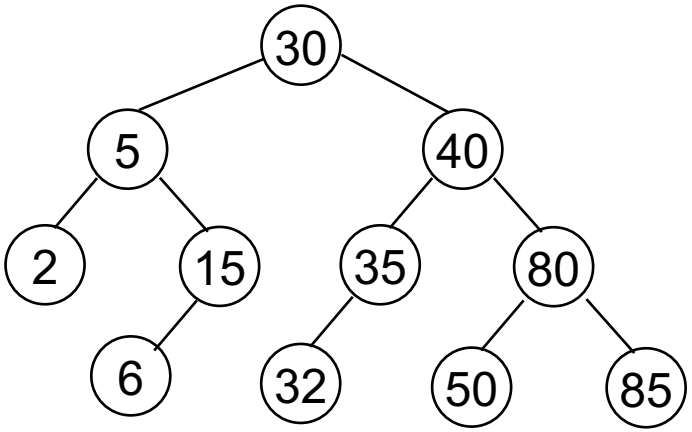
- Input
  - A binary search tree and the mid value k
- Output
  - Two binary search trees (small and big), mid node if k exists in the input binary search tree

Split Algorithm

- Start with two empty binary search trees for small and big
- If k > root, attach root + left subtree to small
  - Two way join with current small, but incoming small is bigger than the current small
- else if k < root, attach root + right subtree to big
  - Two way join with current big, but incoming big is smaller than the current big
- else attach left subtree to small and right subtree to big, and return a node containing k as mid
- Repeat with remaining binary search tree

Split

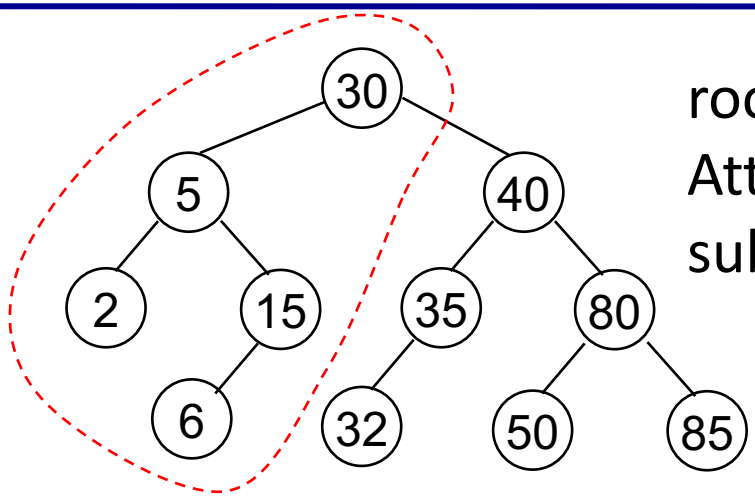
k = 35



small big

Split

k = 35



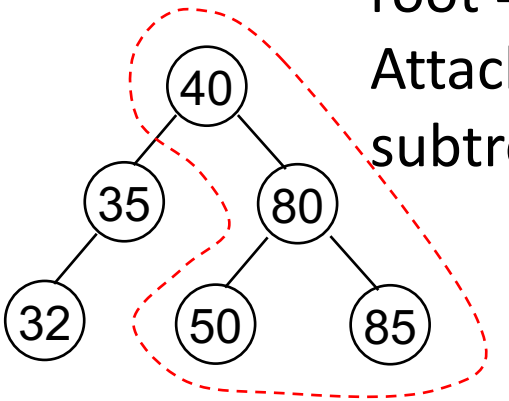
root = 30 < k = 35
Attach root & left subtree to small

small

big

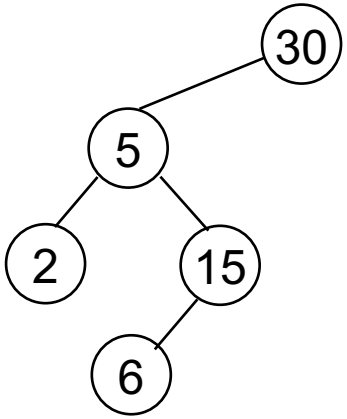
Split

k = 35



root = 40 > k = 35
Attach root & right subtree to big

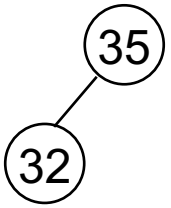
small



big

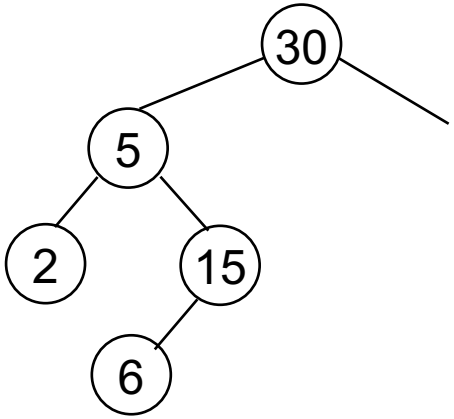
Split

k = 35

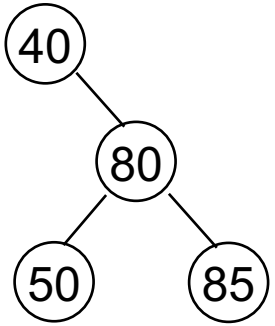


root = 35 = k = 35
Attach left subtree to small and right subtree to big

small



big

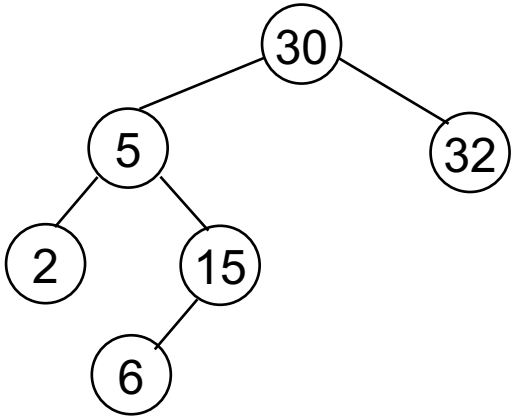


Split

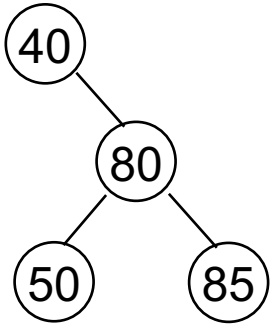
k = 35

root = 35 = k = 35
Return 35 as mid

small

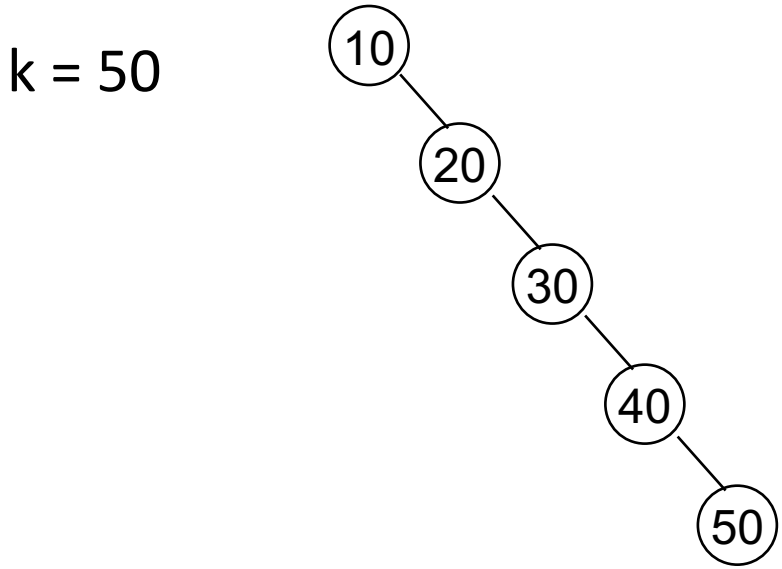


big



Split

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



Split

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

Height of a Binary Search Tree

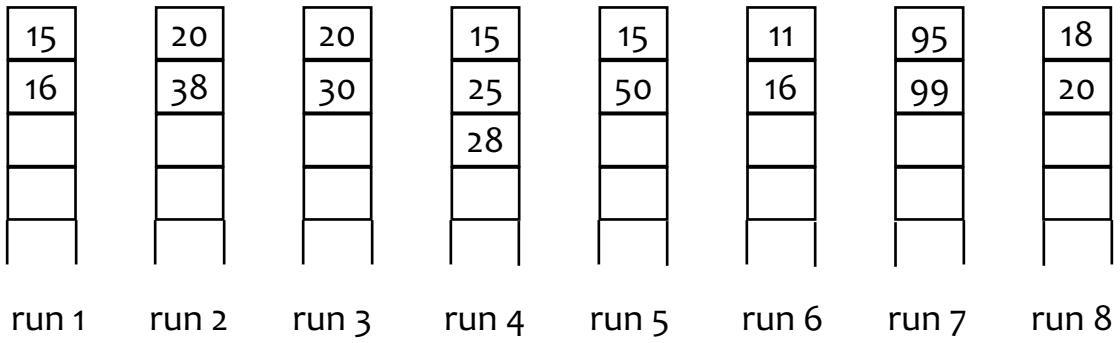
- Worst case
  - O(n)
  - e.g., 1,2,3,4,5,..,n (right skewed tree)
- Best case
  - O(log n)
  - e.g., complete/full binary tree
- Balanced search tree
  - Search tree with worst-case height O(log n)

Outline

- Binary search trees
- Selection trees

Selection Trees

- Merging k ordered list (run)
  - Merged list must be ordered
  - Need to find the smallest among k runs
    - k-1 comparison per element
    - O(nk) for n elements and k runs
    - Expensive if k is large

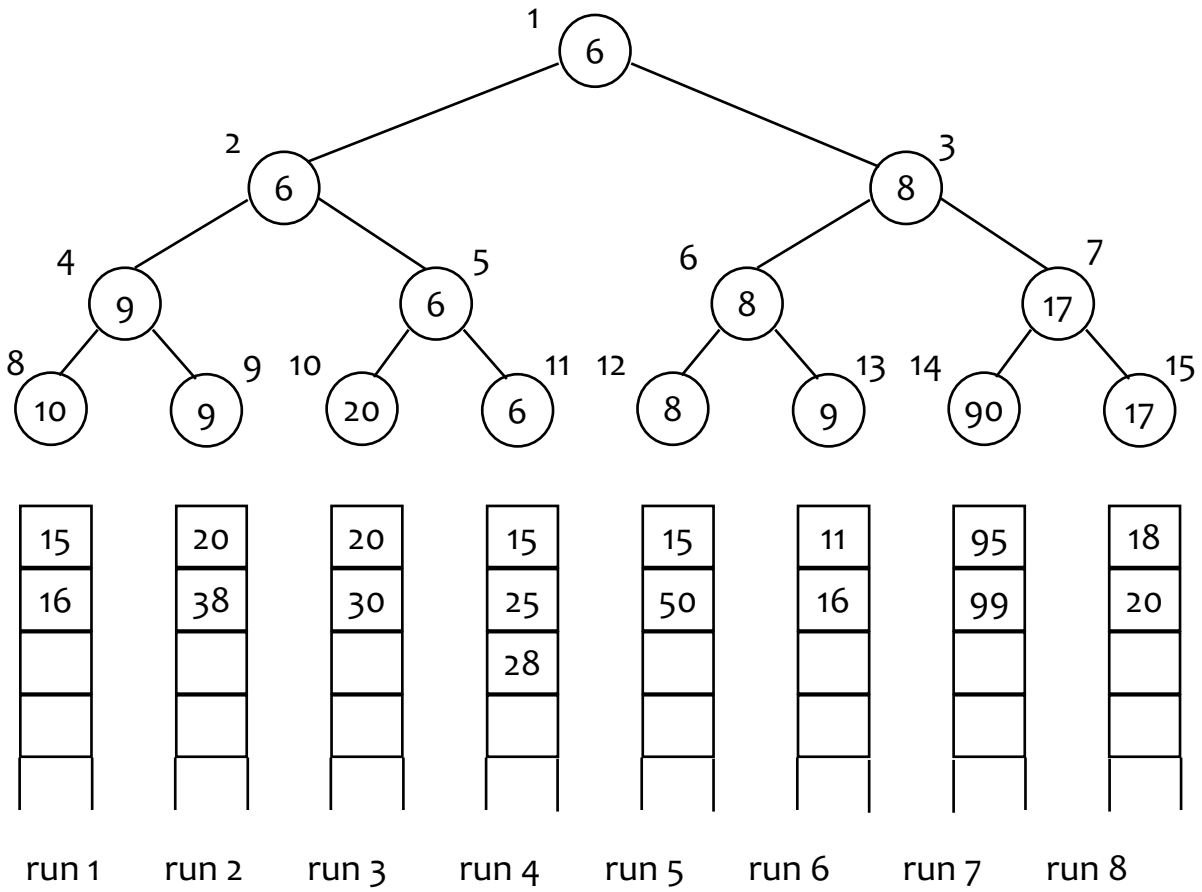


Winner Trees

- Ideas
  - Compare O(log k) instead O(k)
- Complete binary tree
  - Leaf : smallest from each run
  - Non-leaf : winner among two children

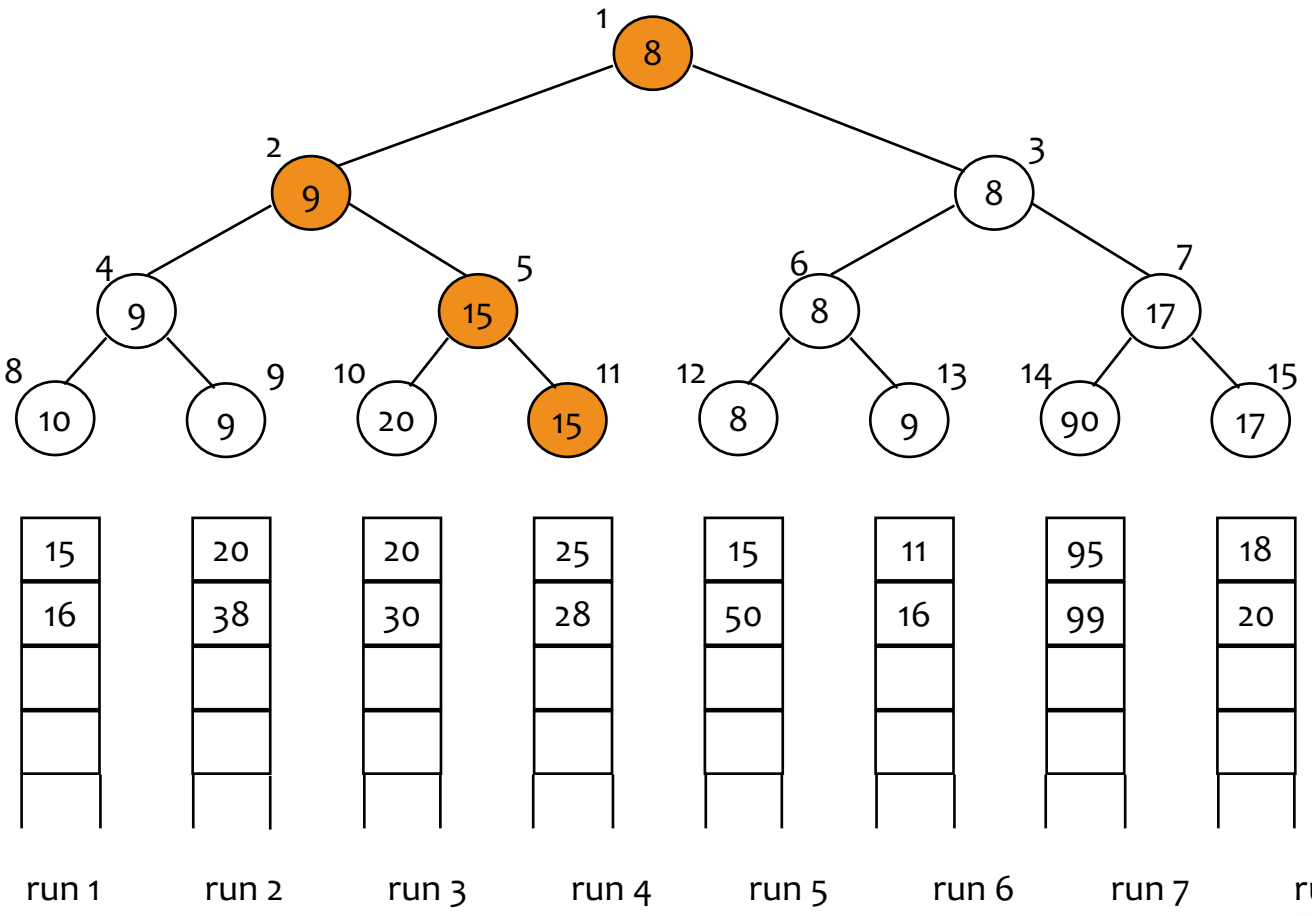
Example of Winner Tree

- 6 is the smallest among 8 values



Example of Winner Tree

- Add next element from run 4 to the tree

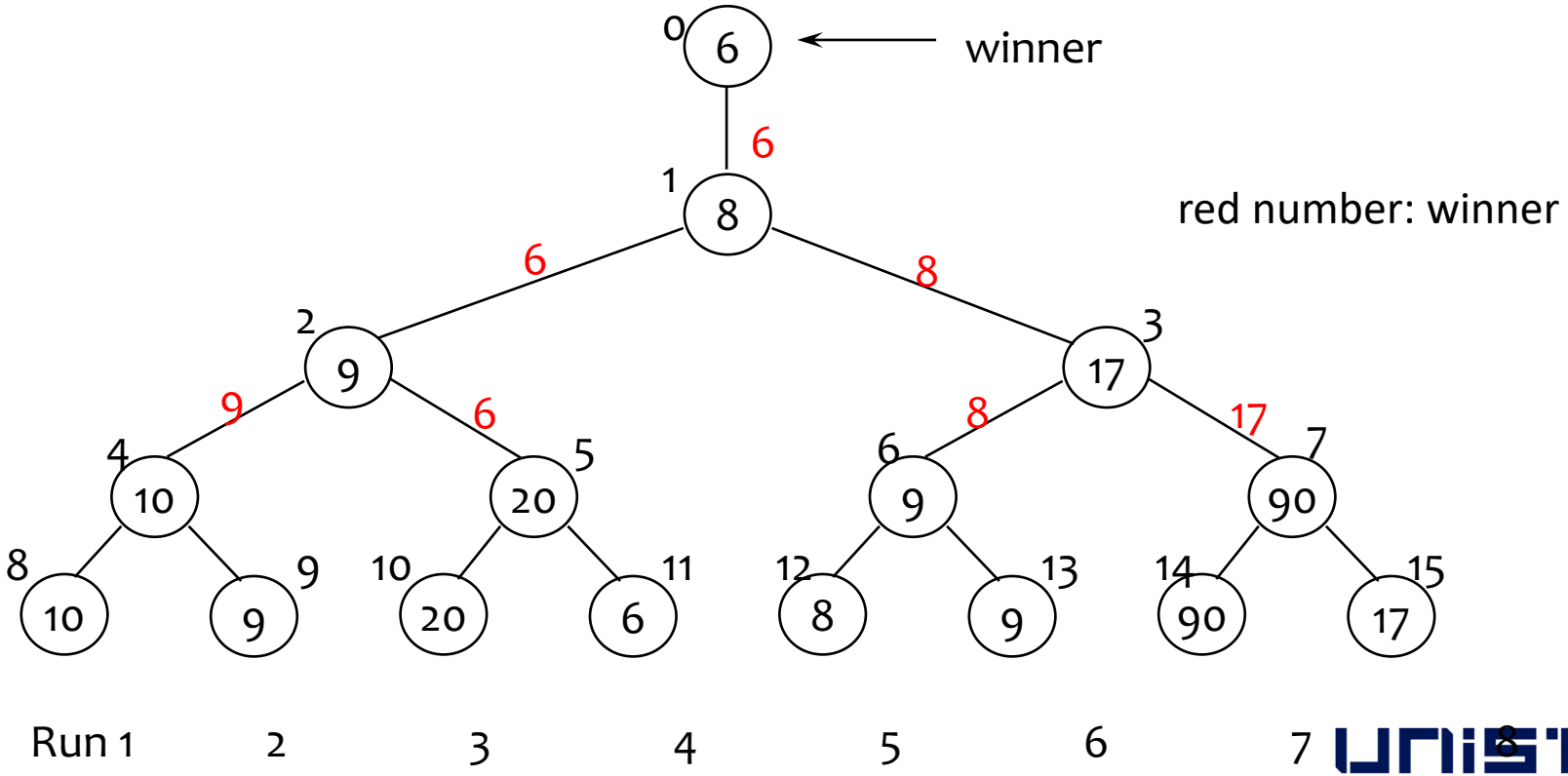


Winner Tree

- # of levels : ceil($\log_2(k+1)$)
- Tree restructuring time (when a new element is inserted) : $O(\log_2 k)$
- Initial tree set up : $O(k)$
- Total time : $O(n\log_2 k)$
  - Restructuring per each element

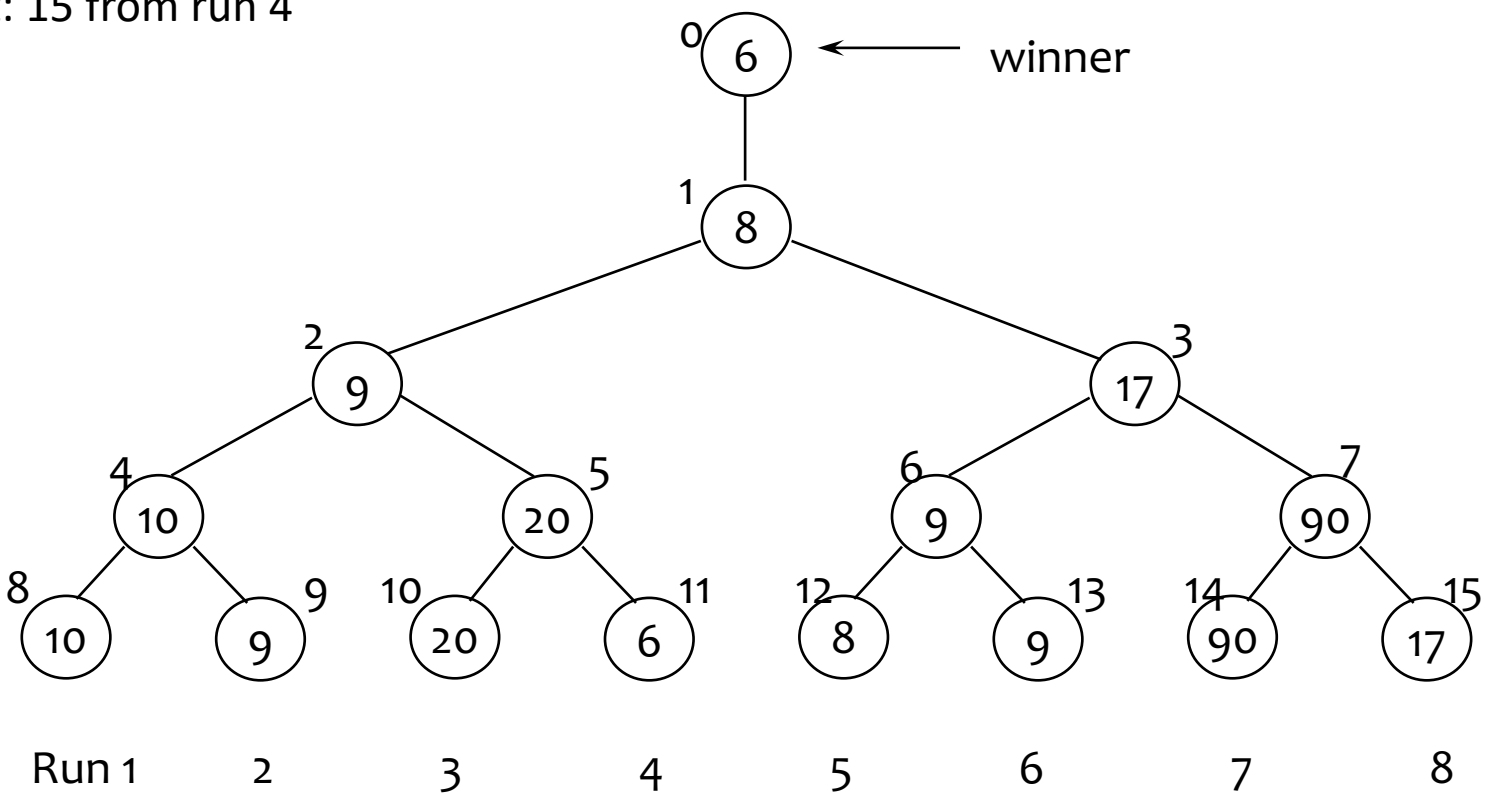
Loser Tree

- Nonleaf is loser, overall winner at the top
- No need to compare with siblings



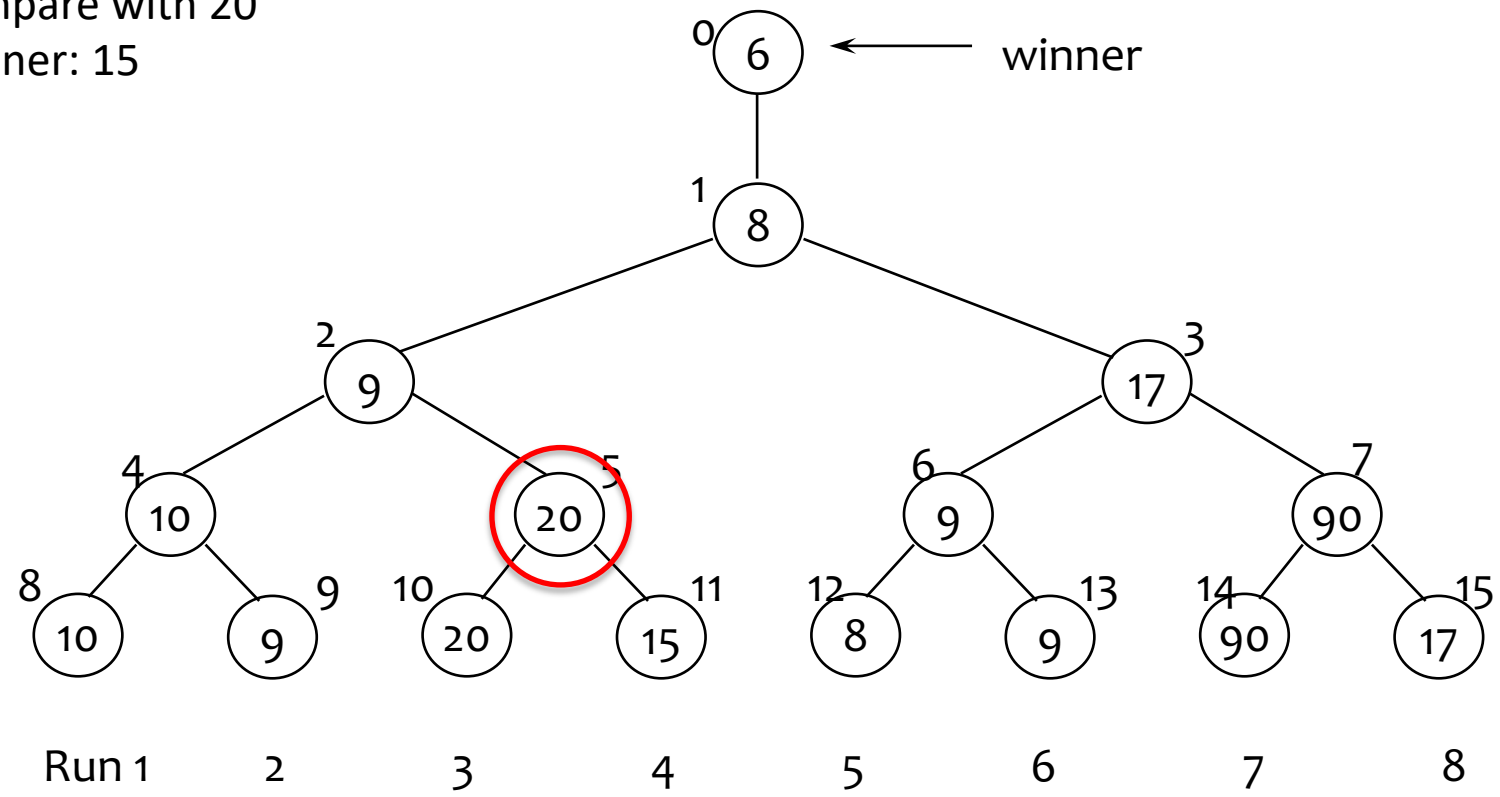
Loser Tree

Next: 15 from run 4



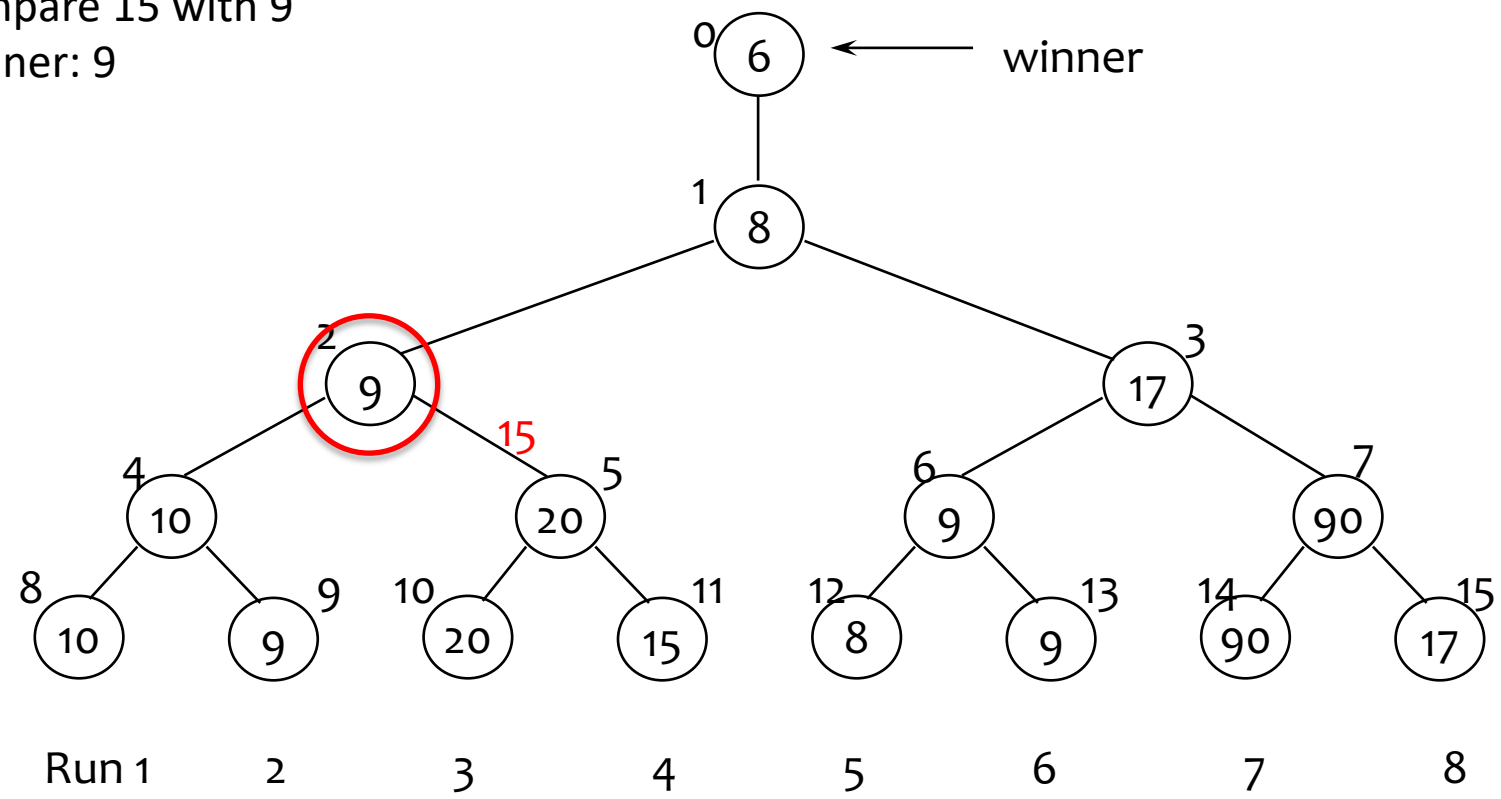
Loser Tree

Compare with 20
Winner: 15



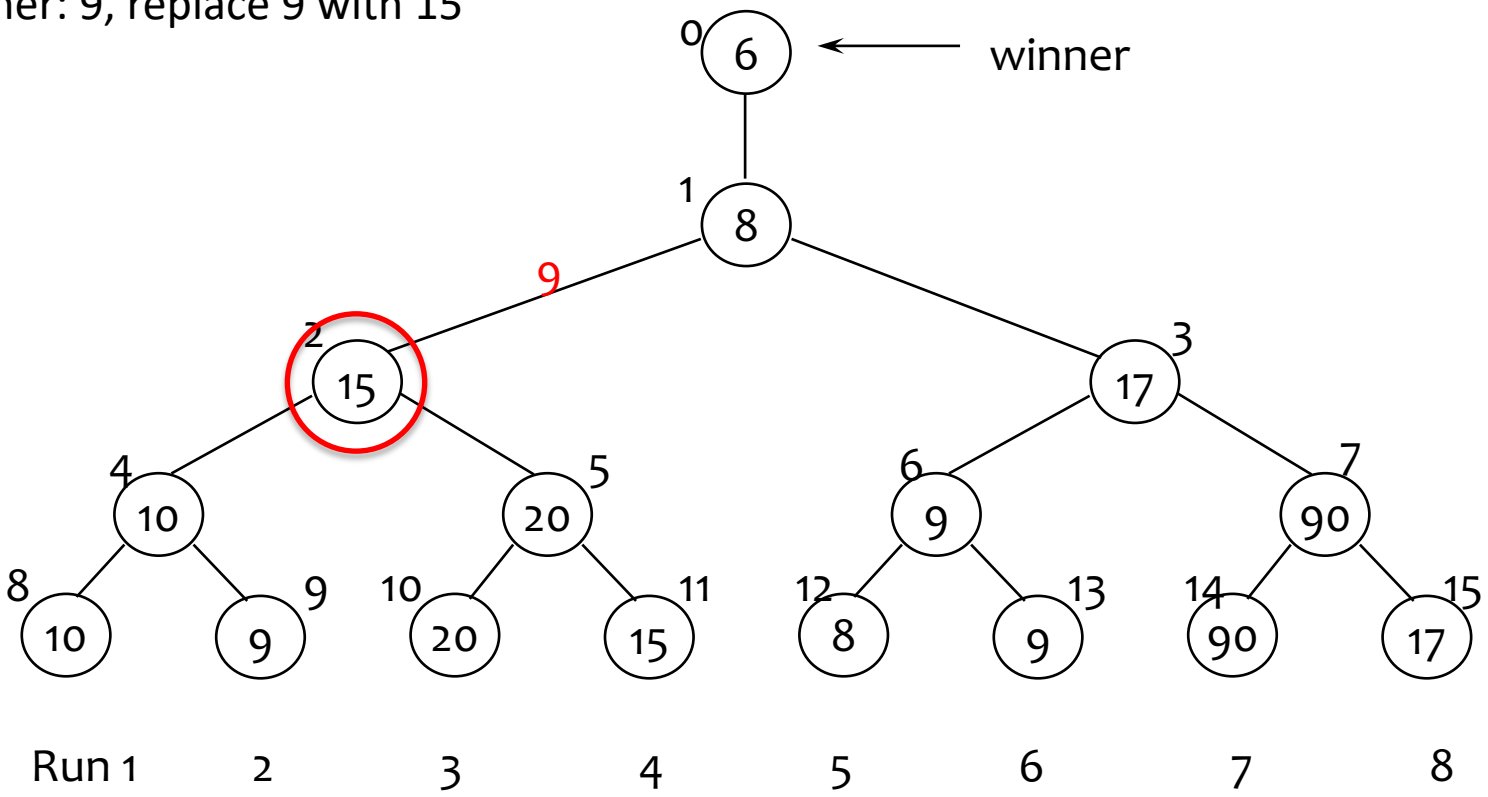
Loser Tree

Compare 15 with 9
Winner: 9



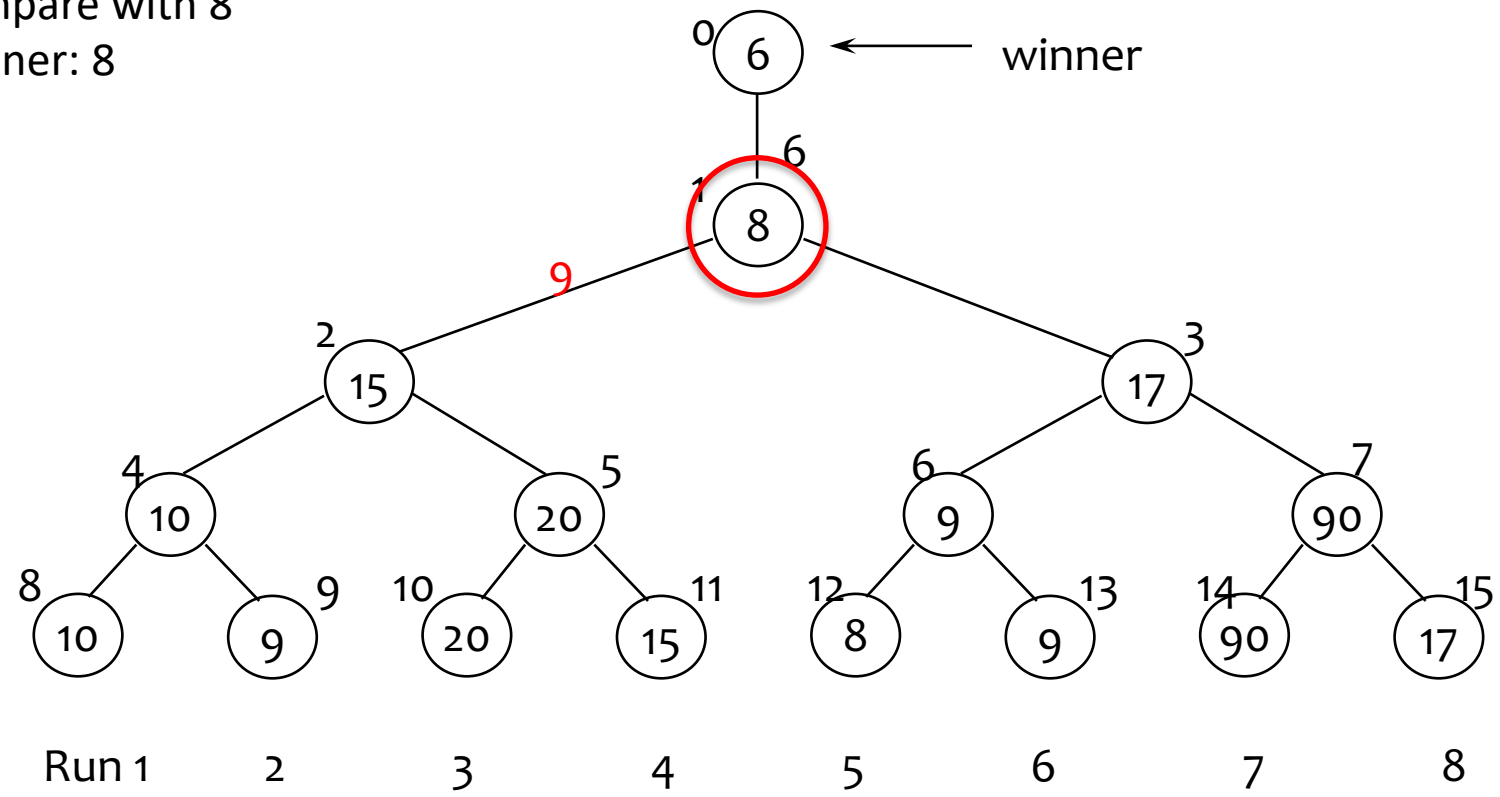
Loser Tree

Winner: 9, replace 9 with 15



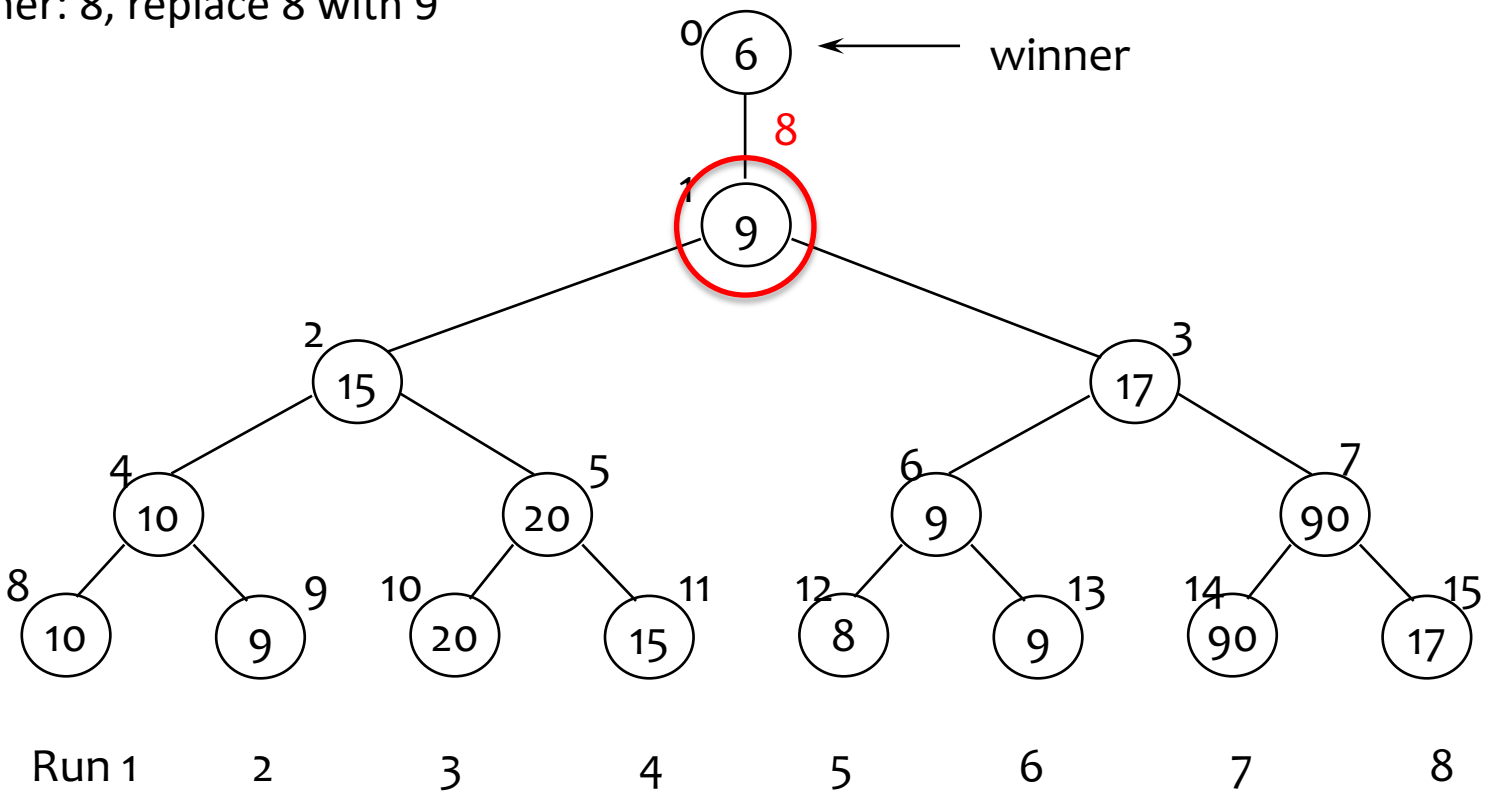
Loser Tree

Compare with 8
Winner: 8



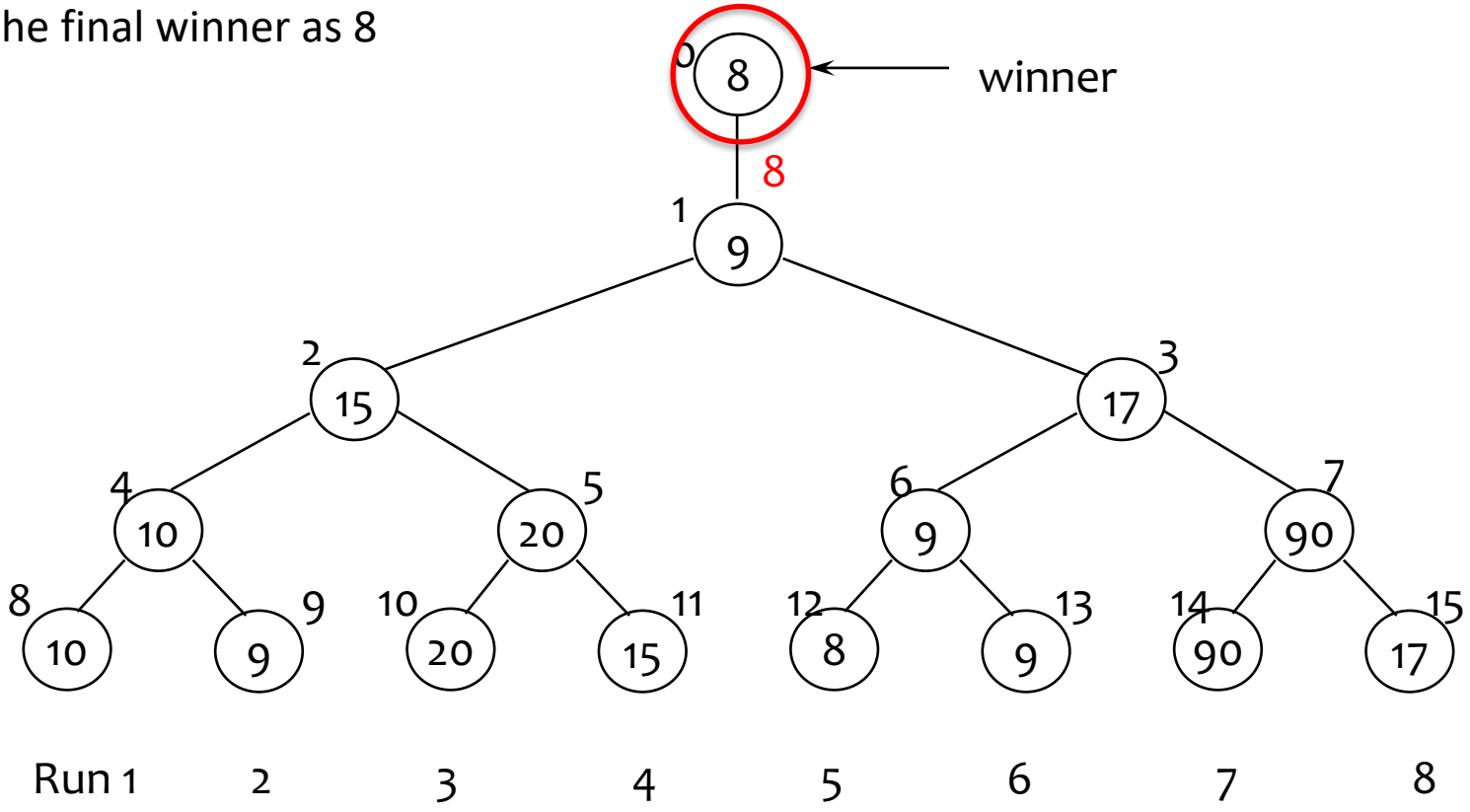
Loser Tree

Winner: 8, replace 8 with 9



Loser Tree

Set the final winner as 8



Questions?