BM267 - Introduction to Data Structures

12. Binary Search Tree

Ankara University

Computer Engineering Department

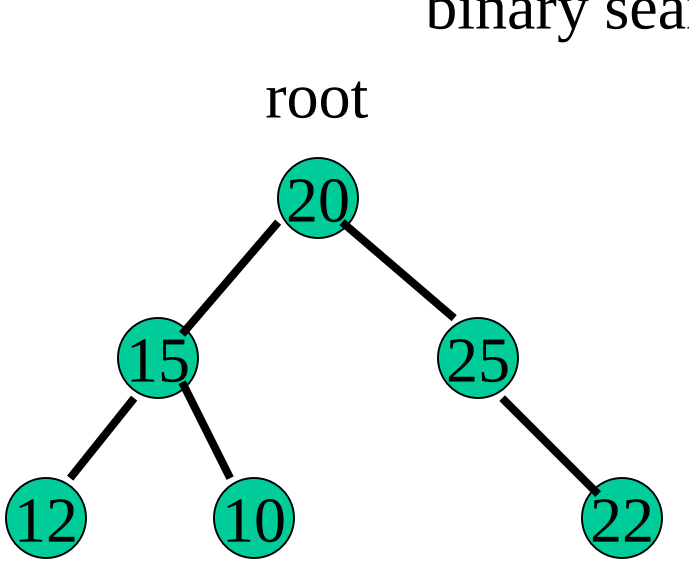
Bulent Tugrul

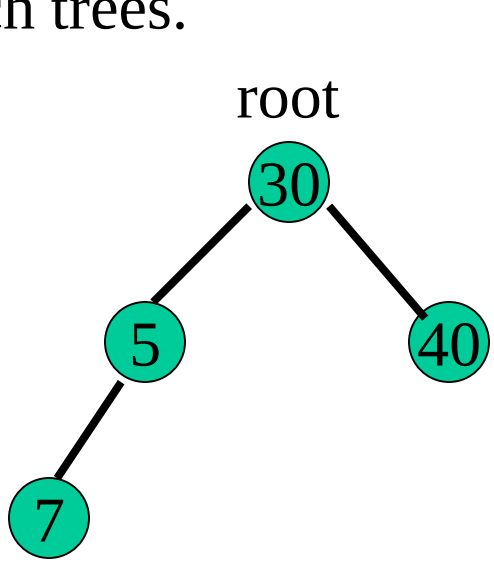
BST

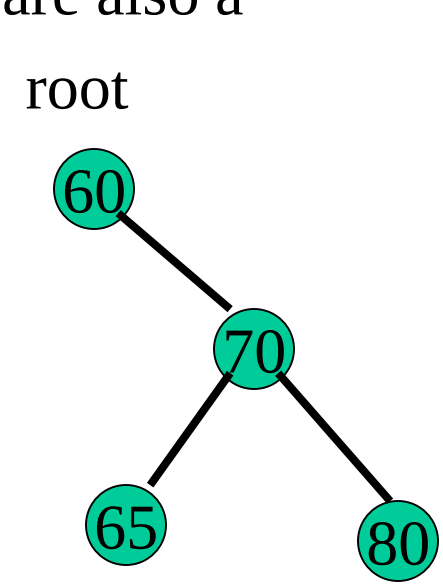
- The basic operations (search, insert, delete) can be performed in O (logn) time when a balanced search tree is used.
- Definition: A binary search tree is a binary tree. A nonempty binary search tree satisfies the following properties:
  1. Every element has a key(or value) and no two elements have the same key; therefore all keys are distinct.
  2. The keys (if any) in the left subtree of the root are smaller than the key in the root.

BST

3. The keys (if any) in the right subtree of the root are larger than the key in the root.
4. The left and right subtrees of the root are also a binary search trees.







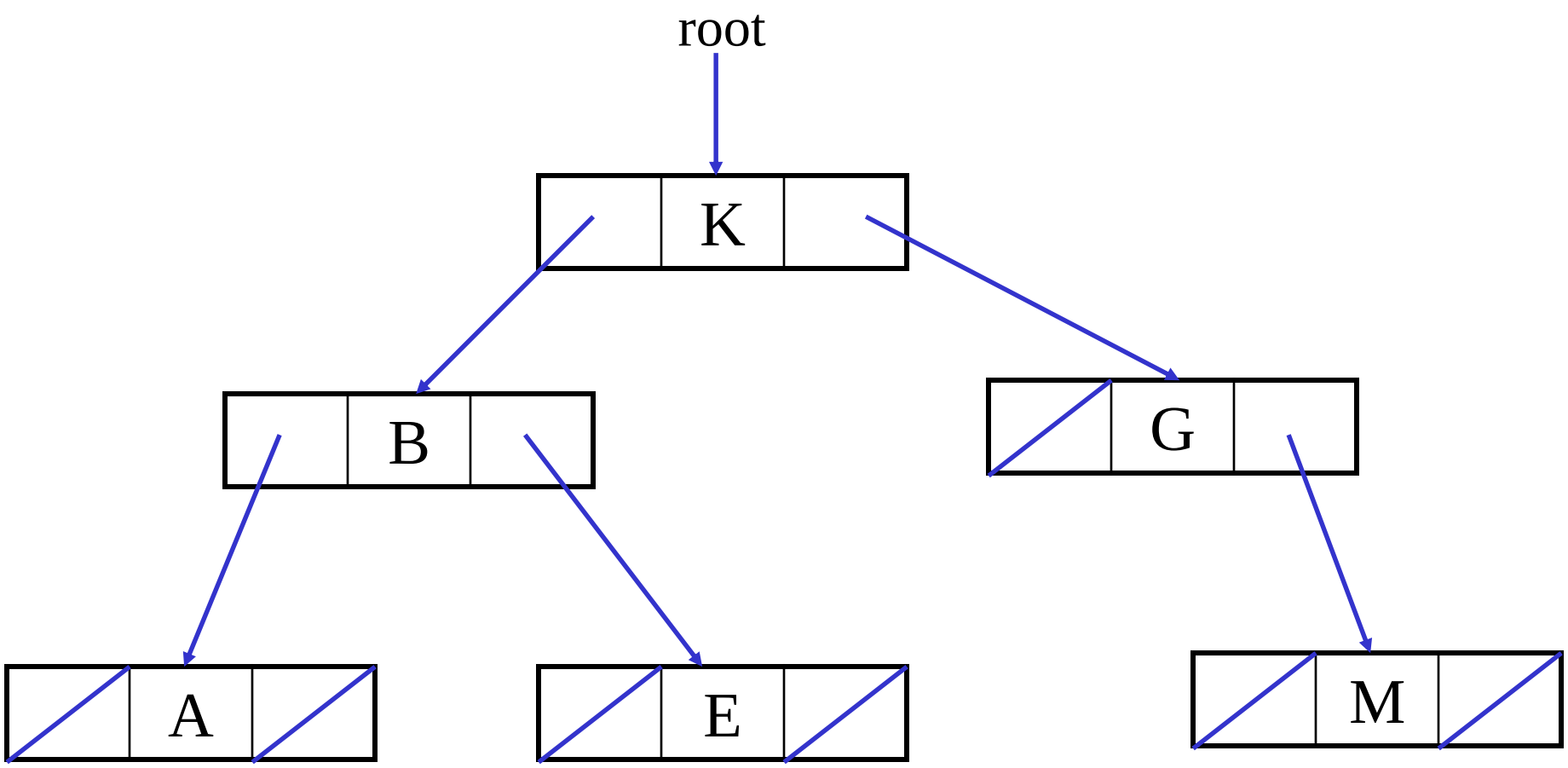
BST

- We can remove the requirement that all elements in a binary search tree need be distinct keys. Now we replace smaller in property 2 by smaller than or equal to.
- The resulting tree is called a binary search tree with duplicates.

```
struct TreeNode
{
        int data;
        struct TreeNode *left;
        struct TreeNode *right;
}
```

BST

Linked representations of binary search trees.



BST- Searching

- Suppose we wish to search for an element with key k. We begin at the root. If the root is NULL, the search tree contains no elements and the search is unsuccessful.
- Otherwise, we compare k with the key in the root. If key is less than the root, then no element in the right subtree can have key value k and only left subtree is to be searched.
- If the key is larger than the key in the root, only right subtree needs to be searched.
- If k equals the key in the root, then the search terminates successfully.
- The subtrees can be searched similarly.
- The time complexity is O(h) where h is the height of the tree.

BST- Searching

```
int tree_search ( int key ){
      struct treenode *p = root;
      while( p !=NULL ){
            if( key < p->data)
                  p = p->left;
            else if ( key > p->data)
                  p = p->right;
            else
                  return 1;   }

      return 0;}
```

BST- Searching

```
int search_rec(struct treenode *p, int key ){
      if( p == NULL )
            return 0;
      if( key == p->data )
            return 1;
      if ( key < p->data )
            return search_rec( p->left, key);
      else
            return search_rec( p->right, key);
}
```

BST- Insertion

- To insert a new element ‘e’ into a binary search tree, we must first verify that its key is different from those of existing elements by performing a search in the tree.
- If the search is unsuccessful, then the element is inserted at the point the search terminated.
- For instance, to insert an element with key 80 into tree, we must search for 80.
- This search terminates unsuccessfully, and the last node examined the one with key 40.
- The new element is inserted as the right child of this node.

BST- Insertion

```
void tree_insert( int key ){
      struct treenode *p, *pp, *r;
      p = root;  // search pointer
      pp = NULL; //parent of p
      while( p ){
            pp = p;
            if( key < p->data)
                  p = p->left;
            else if ( key > p->data)
                  p = p->right;
            else
printf(" The key is already in the tree.\n");   }
```

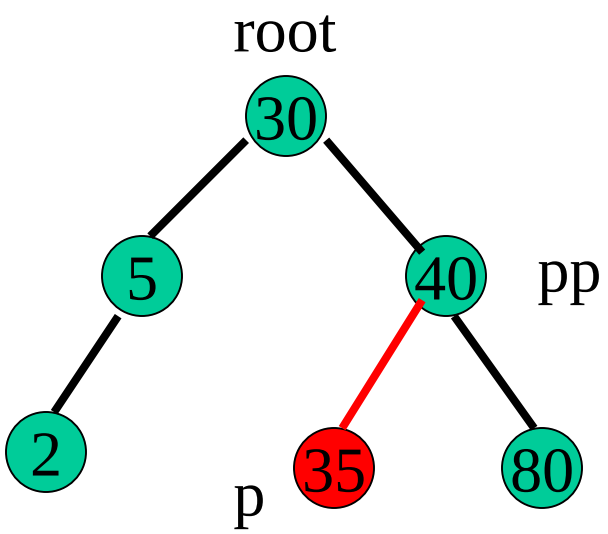
BST- Insertion

```
r = malloc( sizeof(struct treenode * ));
      r->data = key;
      r->left = NULL;
      r->right = NULL;
      if(root)   // tree is not empty      {
            if( key < pp->data)
                  pp->left = r;
            else
                  pp->right = r;     }
      else   // insertion into an empty tree
            root = r; }
```

BST- Deletion

- For deletion we consider three possibilities for the node p that contains the element to be deleted:
  - p is a leaf
  - p has one nonempty subtree
  - p has two nonempty subtree
- Case 1 is handled by discarding the leaf node.
- To delete 35 from the tree, the left-child field of its parent is set to NULL and node is discarded.

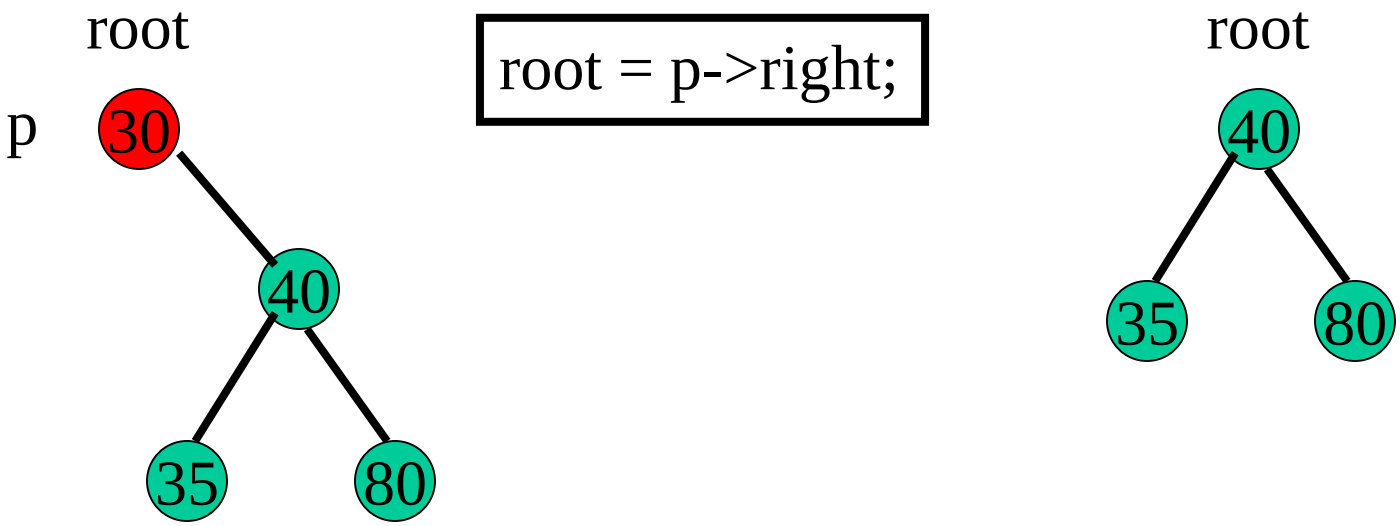
BST- Deletion




```
pp->left  =  NULL
```

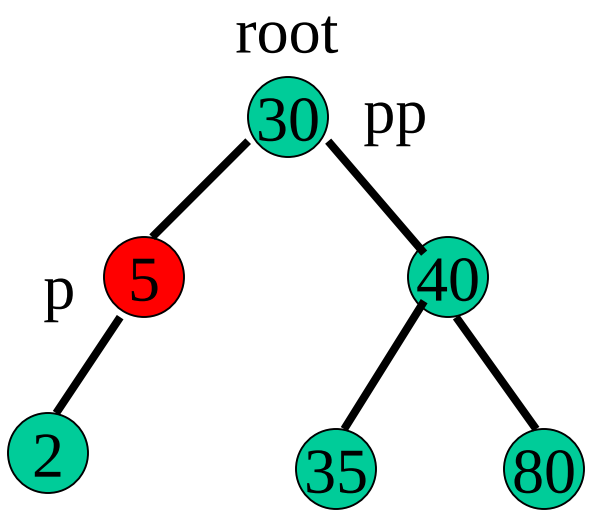
BST- Deletion

- Next consider the case when p has only one nonempty subtree.
- if p has no parent (i.e. it is the root), node p is discarded and the root of its single subtree becomes the new search tree root.



BST- Deletion

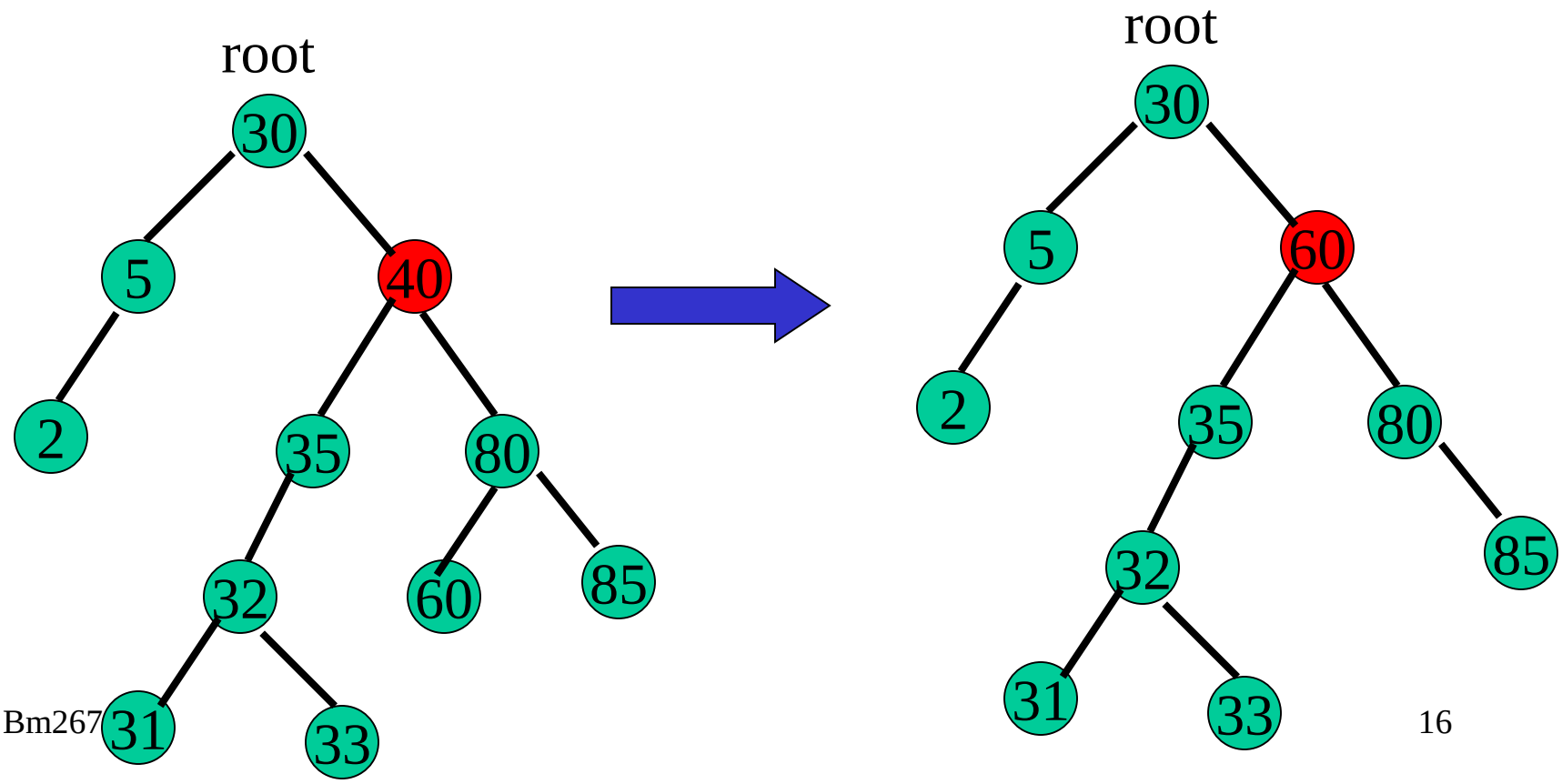
- if p has a parent pp, then we change the pointer from pp so that it points to p's only child and then delete the node p.
- For instance if we want to delete 5, we change the left-child field of its parent(the node containing 30) to point the node containing the 2.



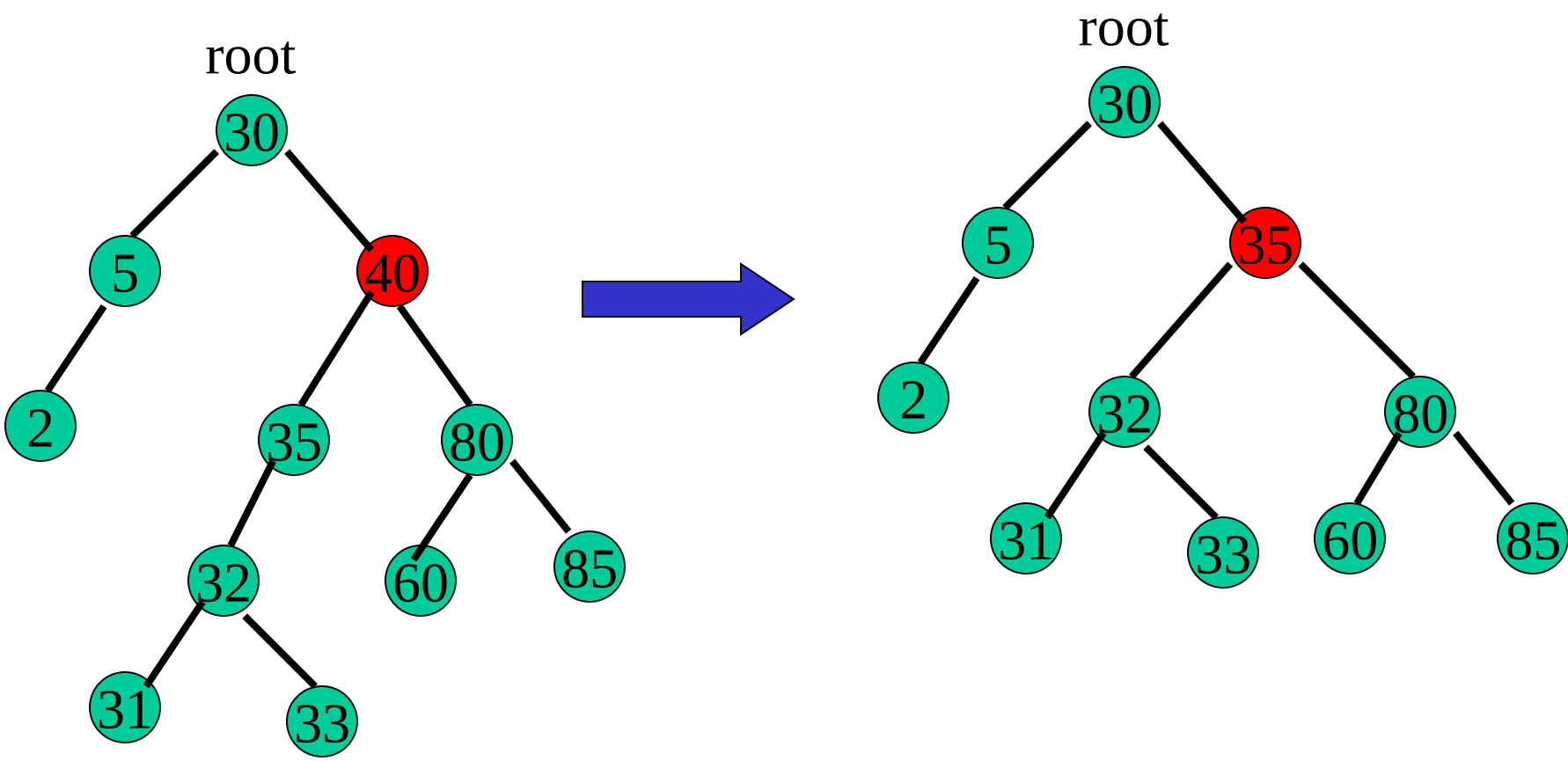

```
pp->left  =  p->left;
```

BST- Deletion

- Finally, to delete an element that has two nonempty subtrees, we replace this element with either the largest element in its left subtree or the smallest element in its right subtree.

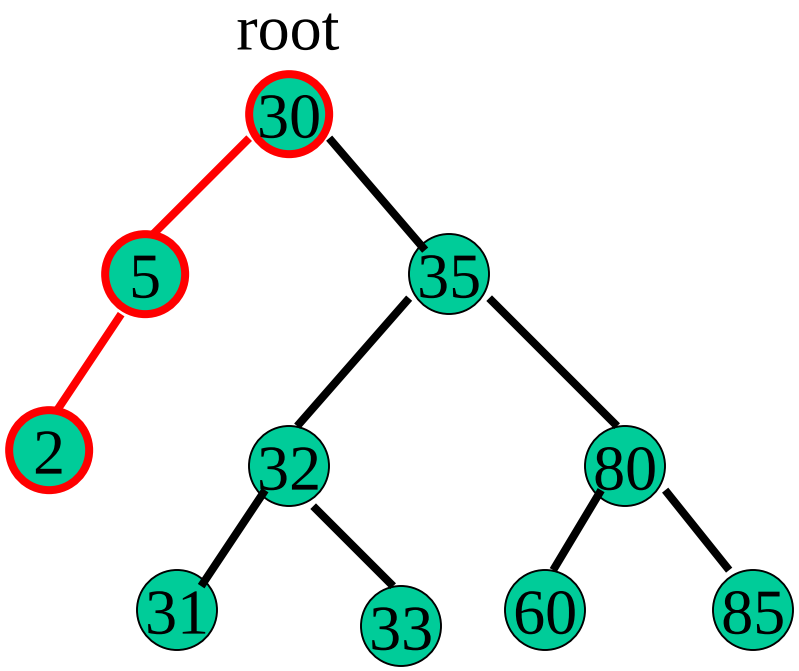


BST- Deletion



BST- Tree Min

- An element in binary search tree whose key is the minimum can always be found by following left child from the root until a NULL is encountered.

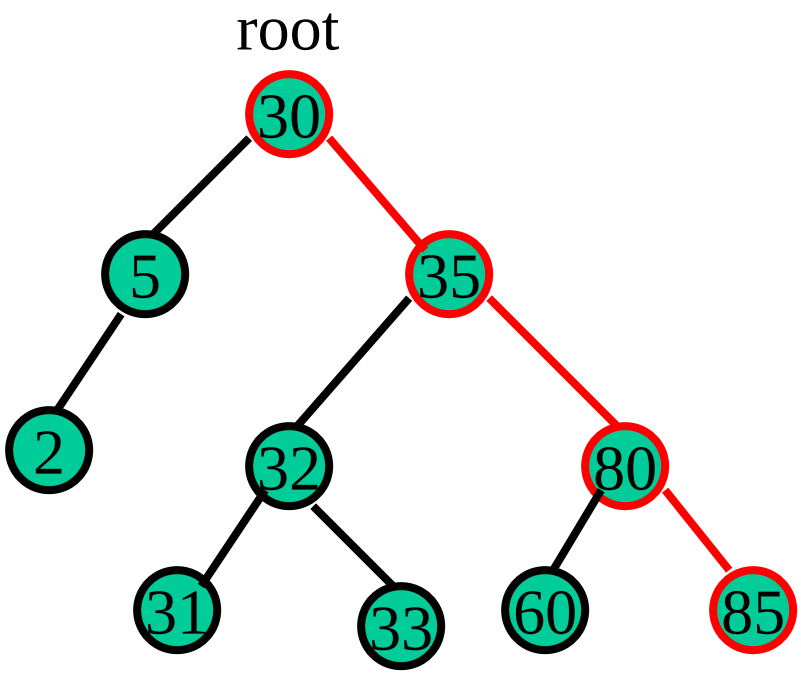


BST- Tree Min

```
struct node * tree_min(struct node * n )
{
      while( n->left != NULL )
            n = n->left;
      return n;
}
```

BST- Tree Max

- An element in binary search tree whose key is the maximum can always be found by following right child from the root until a NULL is encountered.

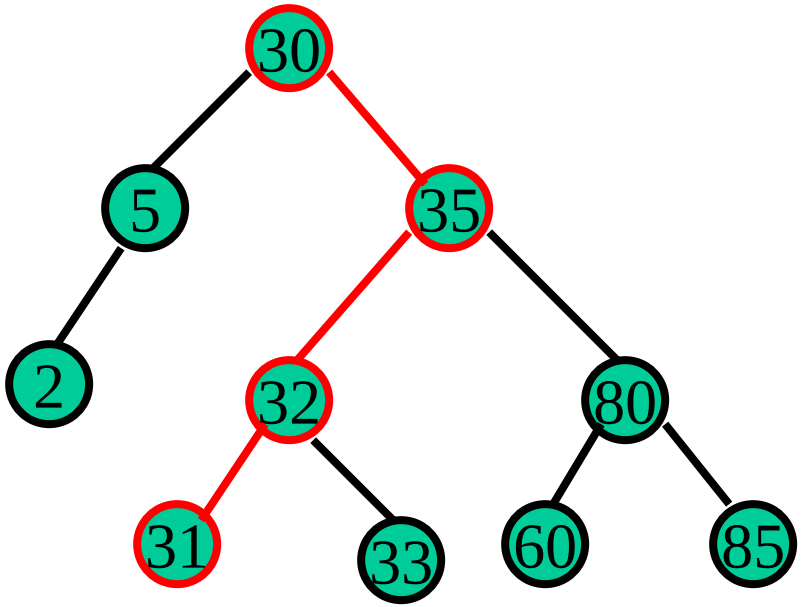


BST- Tree Max

```
struct node * tree_max(struct node * n )
{
      while( n->right != NULL )
            n = n->right;
      return n;
}
```

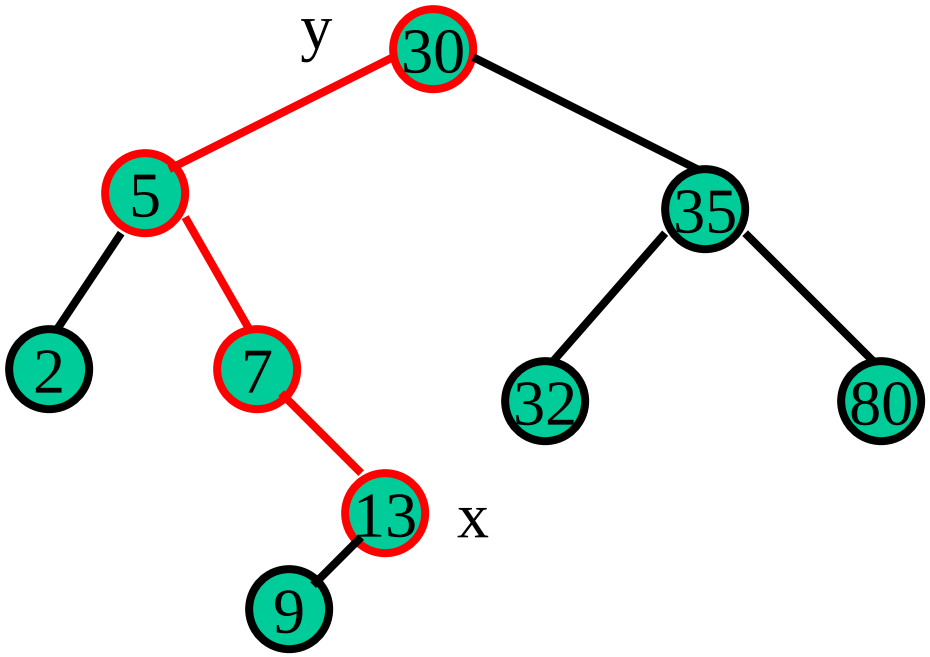
BST- Successor

- The successor of a node ‘x’ is the node with the smallest key greater than ‘x’.
- If the right subtree of node x is nonempty, then the successor of x is the leftmost node in the right subtree.



BST- Successor

- If the right subtree of node x is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x.



BST- Successor

```
struct node * tree_succ(struct node * n ){
      struct node *y;
      if( n->right != NULL )
            return tree_min(n->right );
      y = n->parent;
      while( y != NULL && n == y->right)
      {
            n = y;
            y = y->parent;
      }
      return y;}
```

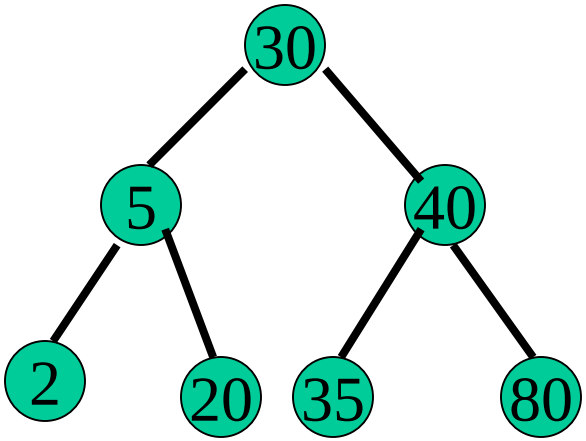
BST- Analysis

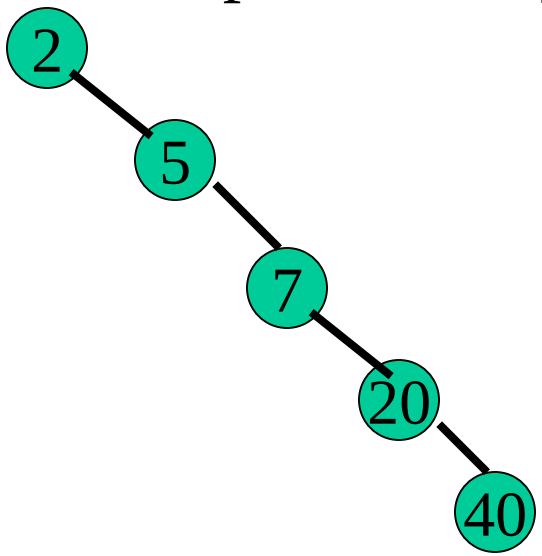
What is the run-time of search, insert, and delete?

- Proportional to the height of the tree

What is the height of a tree with *n* nodes?

- Worst-Case: O(n) in a linear tree
- Best-Case: O(log n) in a complete binary tree





- References: Jeffrey S. Childs
  Clarion University of PA