Dictionary ADT, continued.

Dictionary = "Get elements by orderable Key" container

Besides search, insert, remove the application may need

- next_Key_higher
- next_Key_lower
- maximum
- minimum
- process keys in order of key value

Such an ADT is also called a Dynamic Set

Binary Search Trees

– an implementation of Dictionary, or Dynamic Set

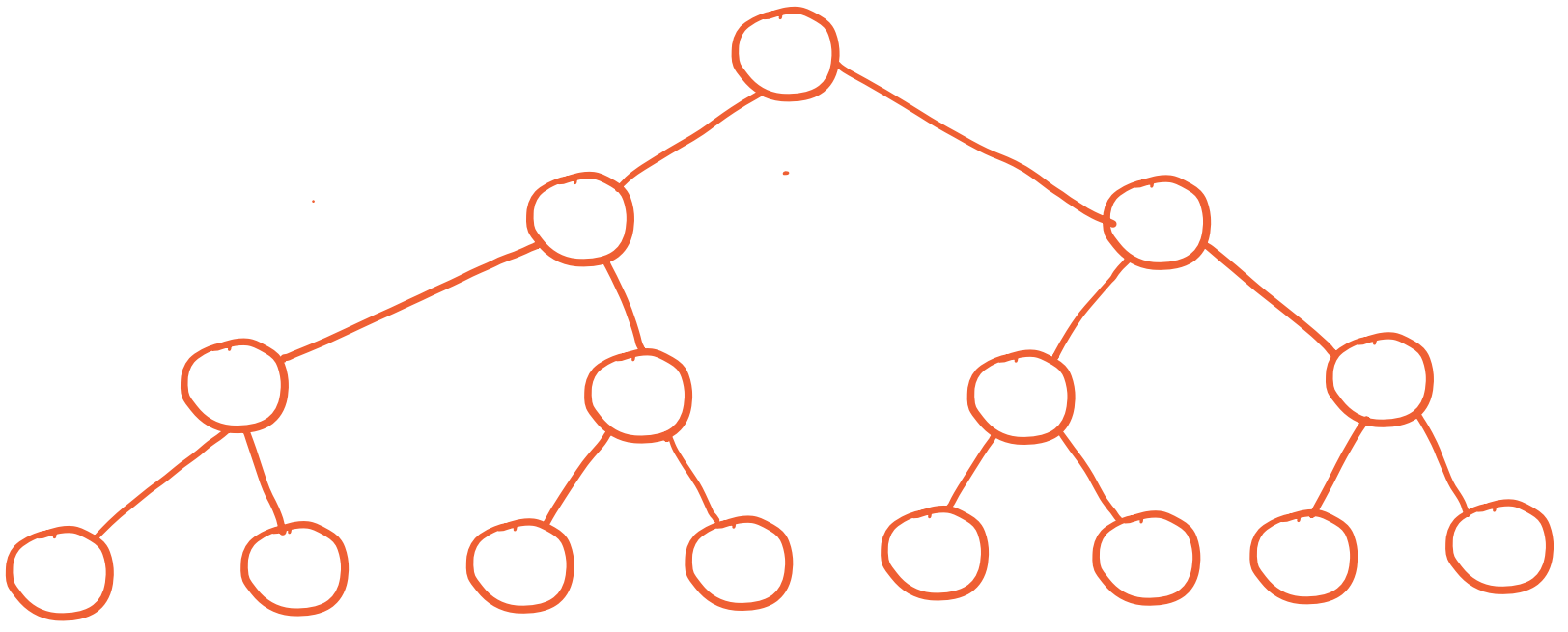
The Data Structure consists of rooted trees. Each node has
- a left child, which may be NULL
- a right child, which may be NULL
- a parent, which is NULL if the node is a root

- Children and parents are maintained using pointers.
- elements (or pointers to them) are stored at the node
- elements have Keys

The nodes in the Binary Search Tree (BST) are in BST-order:

For any node v in the tree:
- Keys in the subtree $v \to left$ are $\leq v \to key$
- Keys in the subtree $v \to right$ are $\geq v \to key$

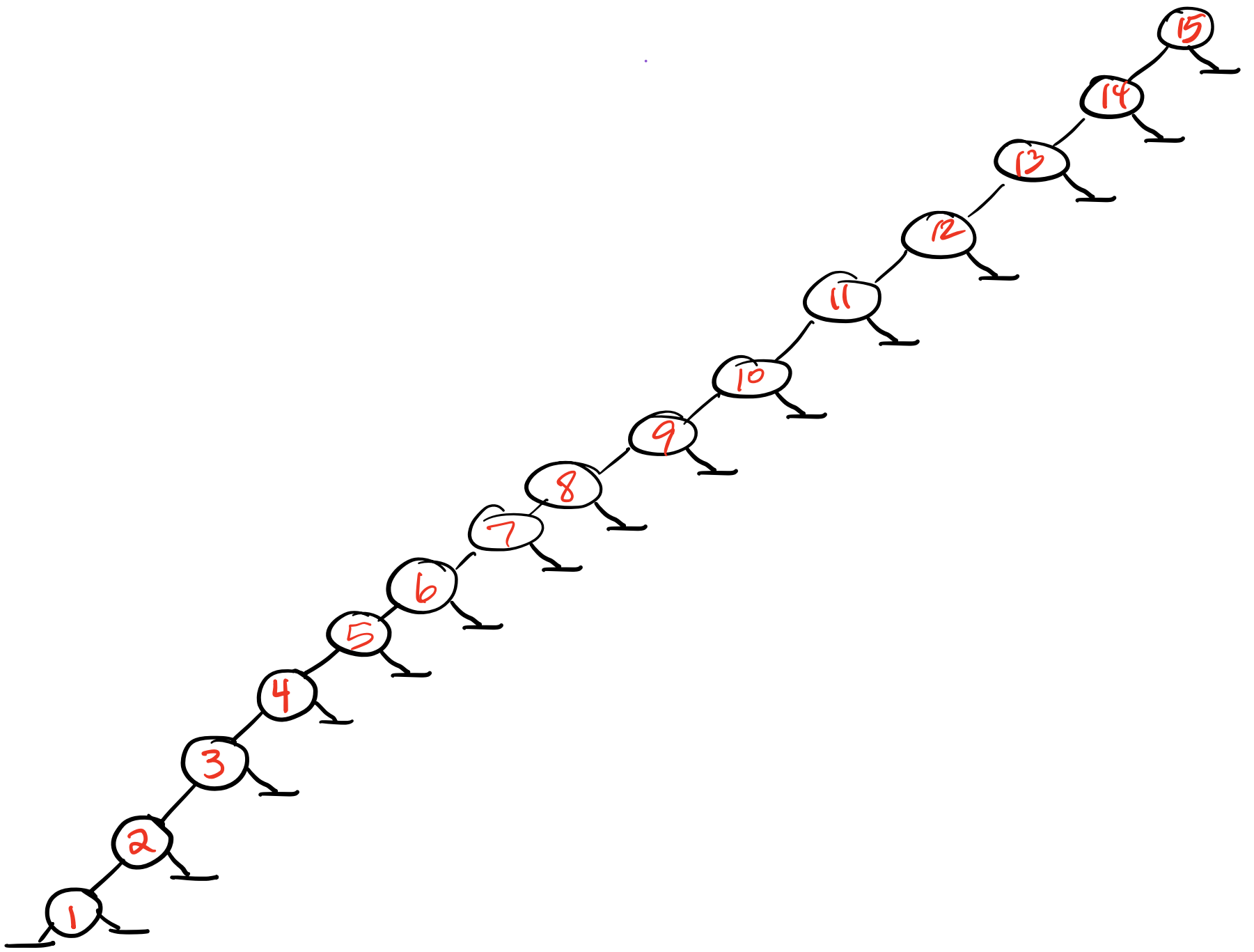
Eg. Fill the following tree with the values $\{1, \ldots, 15\}$ so that it is in BST-order



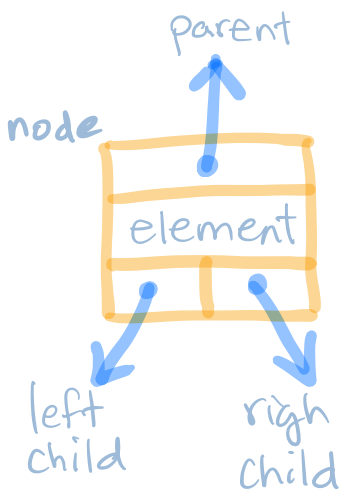
What is the tree traversal method that lists keys in order?

```
InOrderTraversal (r)

    if  r != NULL
        InOrderTraversal (r → left)
        print  r → key
        InOrderTraversal (r → right)
```

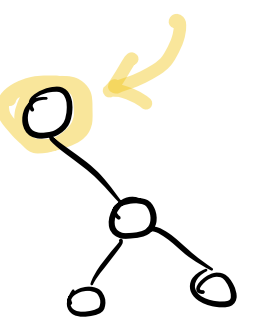


```
node* BST_Search ( r , k )   // return ptr to node with key k
                             // or NULL if no such node
    if   r == NULL   or   k == r→key
        return r
    if   k < r→Key
        return BST_Search ( r→left, k)
    else
        return BST_Search ( r→right, k)
```



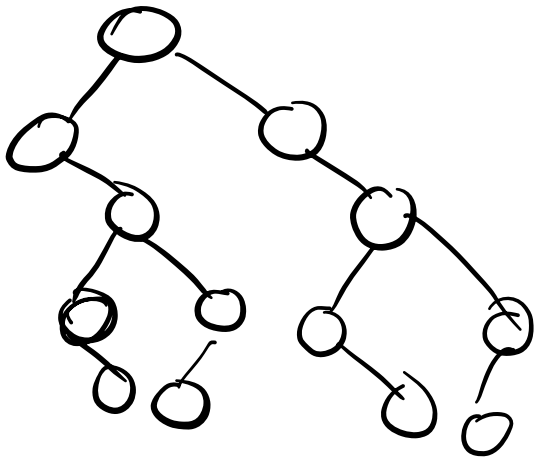
BST_Search has running-time $\Theta(h)$, where h is height of the tree.

```
node* BST_Minimum ( r )   // return ptr to the node in
                          // tree with min key val,
                          // NULL if
    if ( r == Null  or                )
        return r
    else return
```



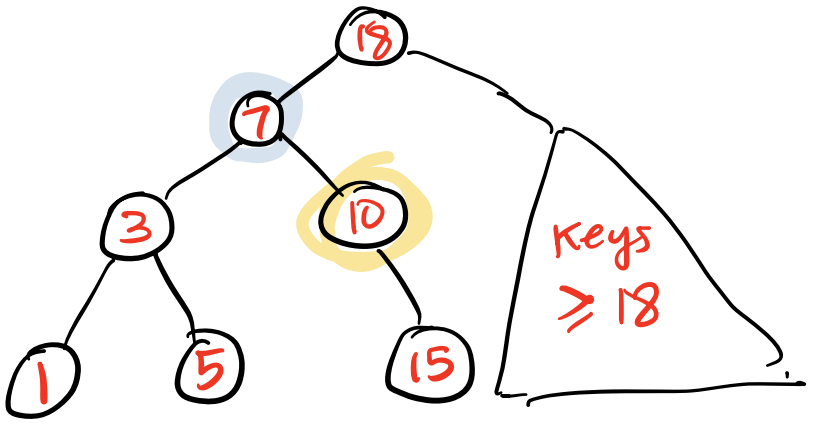
```
node *  BST_Maximum ( r )   // return ptr to the node in tree with max key val, NULL if
```

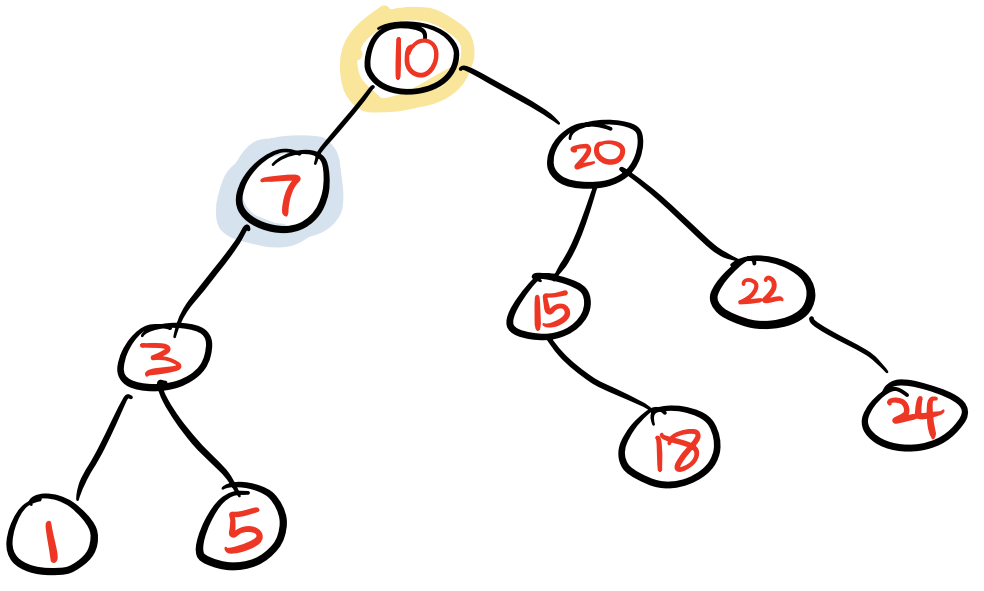
Exercise for the student



```
node *  BST_Successor ( x )   // return ptr with smallest key that is ≥ x→key

        if   x→right != NULL
             [
        else
             y = x→parent
             while ( y != NULL and x = y→right
```





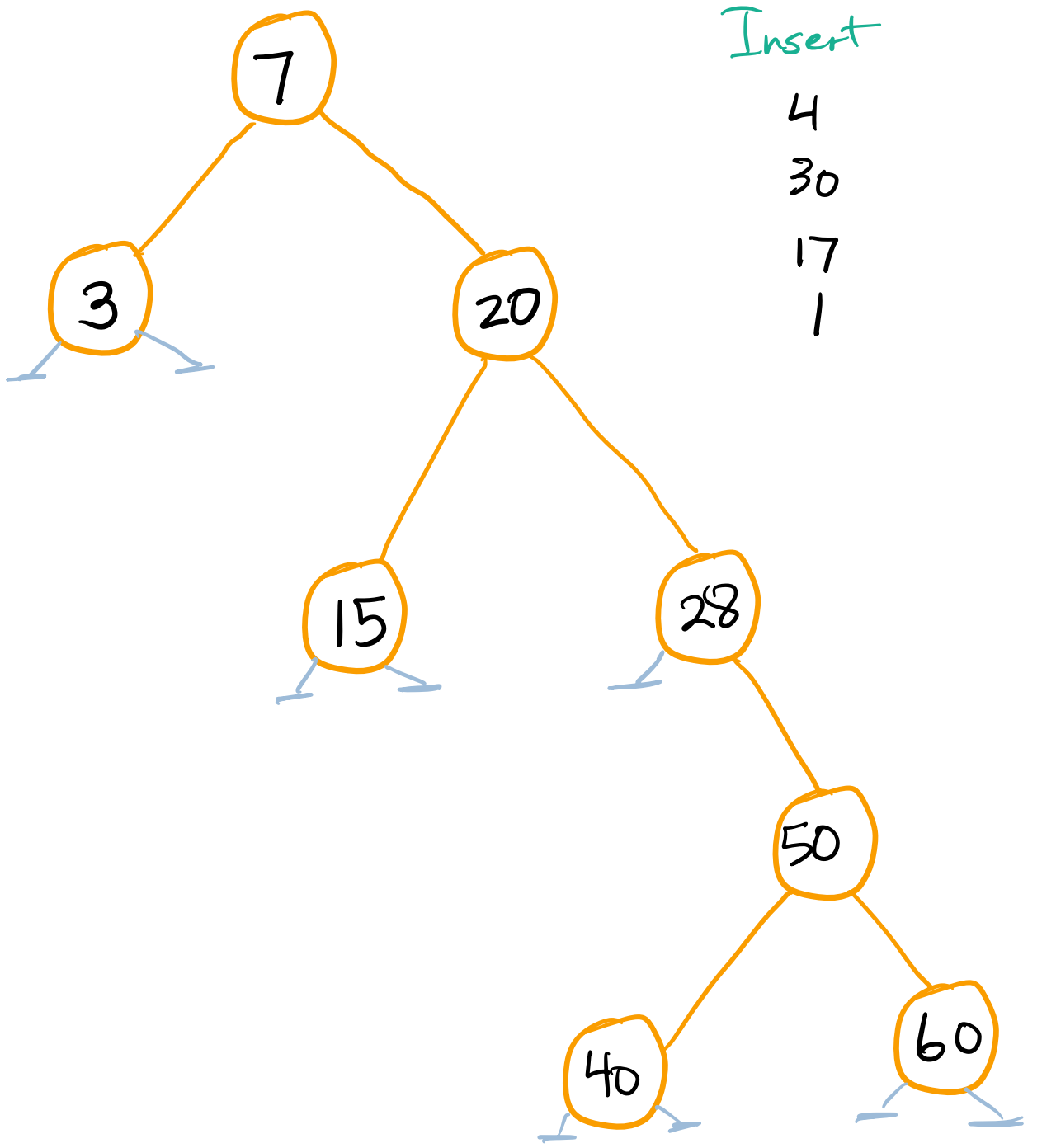
Theorem: The Dictionary operations

Search, Minimum, Maximum, Predecessor, Successor

can be implemented in $\Theta(h)$ time, where

h is height of the tree

```
void *  BST_Insert (&r, e, K)   // insert element e with key K
                                // into subtree rooted at r
if (r == NULL)
    r = new treenode(e, K)
else if (r-> Key < K )
    BST_insert(                 )
else
    BST_insert(                 )
```



Insert

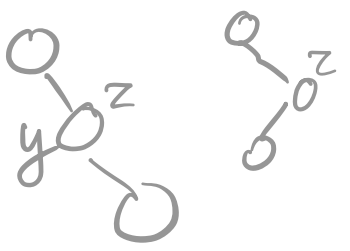
4

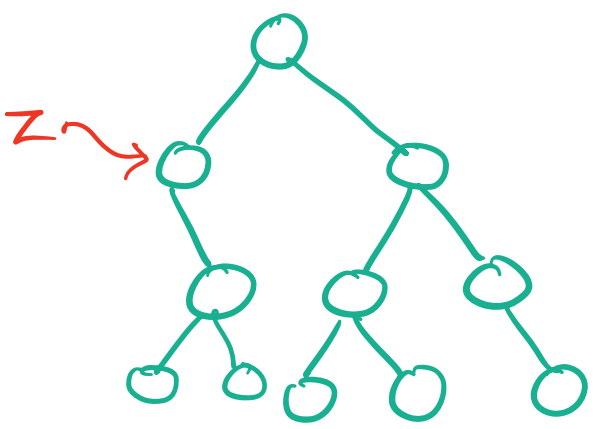
30

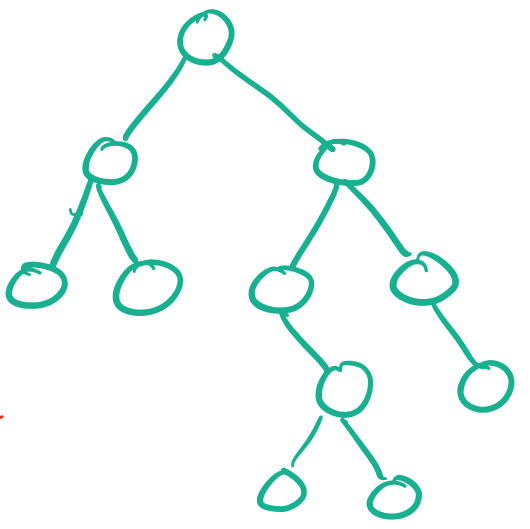
17

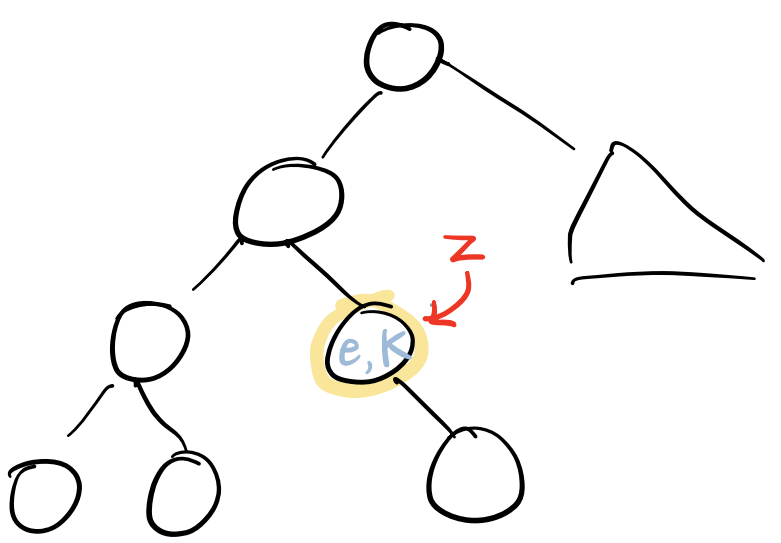
1

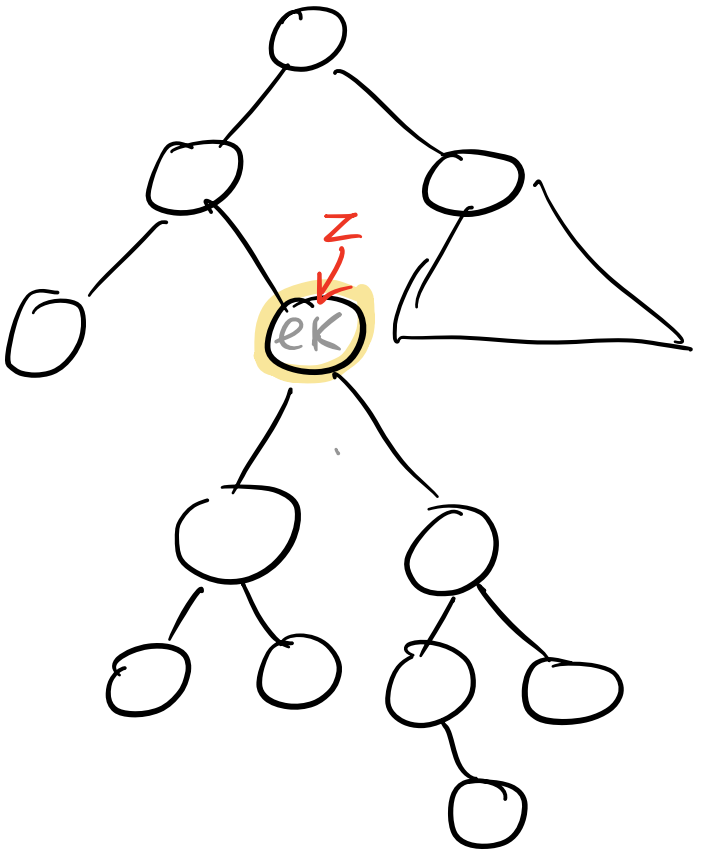
```
void BST_delete(&r, z)        // z is a pointer to a node in the tree rooted at r.
    if (z->left == NULL or z->right == NULL)
        y = z                 // will be the "victim" that will move to z's location
    else y = BST_successor(z)
/* At this point, we know y is missing at least 1 child.
    if y->left != NULL
        x = y->left
    else x = y->right
    if x != NULL
        x->parent = y->parent
    if y->parent == NULL
        r = y
    else if y == y->parent->left
        y->parent->left = x
    else y->parent->right = x
    if y != z
        z->element = y->element
        z->Key = y->Key
    return y
```

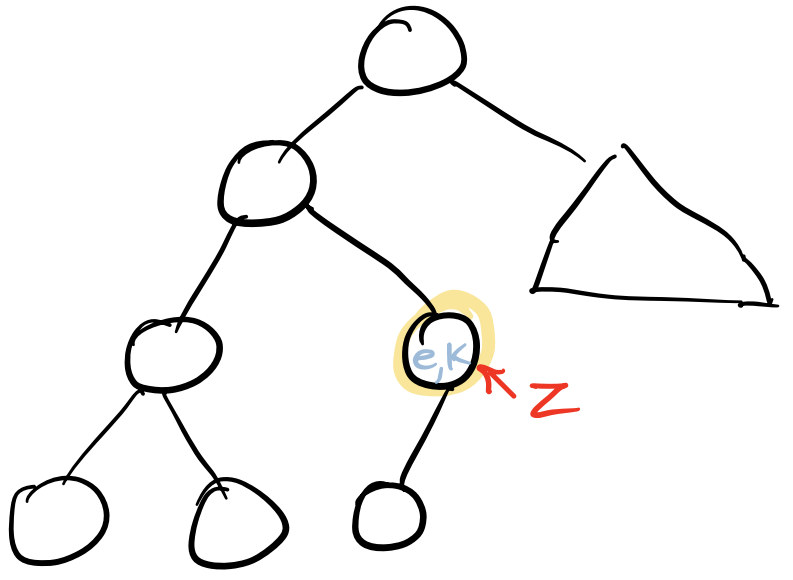







z
e,K


z
eK


e,K
z

Theorem: BST_Insert and BST_Delete can be implemented to run in $\Theta(h)$ time, where h is height of tree.

Theorem: Expected height of a BST built on a keyset, insertions are uniform random distribution, is $\Theta(\log n)$

n = # of elements (keys, nodes) in tree

Theorem: Worst-case BST has height $\Theta(n)$.

Note: uniform random distribution is not a good assumption. Distribution comes from application; ∄ a hash function to make it more "random".

Binary Search Trees

Main takeaway: A **binary search tree (BST)** implements a dictionary or dynamic set ordered by key. Search, insertion, deletion, and order-related operations take time proportional to the tree height, h. A randomly built BST has expected height $\Theta(\log n)$, but its worst-case height is $\Theta(n)$.

Dictionary ADT and BST Structure

- A dictionary stores elements associated with **orderable keys**.
- In addition to searching and inserting, applications may need to:
  - Find the **minimum** or **maximum** key.
  - Process keys in sorted order.
  - Find a key's **successor** or **predecessor**.
- A dictionary with these ordered-key operations is also called a **dynamic set**.
- A BST is a rooted tree in which each node contains an element (or a pointer to one) and its key. Nodes may have:
  - A left child.
  - A right child.
  - A parent; the root's parent is `NULL`.
- **BST ordering:** For every node with key k:
  - Keys in its left subtree are less than k.
  - Keys in its right subtree are greater than k.

Traversal

In-order traversal

Visits the BST's keys in sorted order:

```
InOrder(r):
    if r == NULL:
```

```
        return
    InOrder(r.left)
    print r.key
    InOrder(r.right)
```

Search and Order Operations

Search

Search follows one subtree at each step, based on comparison with the current key.

```
BST-Search(r, k):
    if r == NULL or k == r.key:
        return r
```