Dictionary ADT, continued.

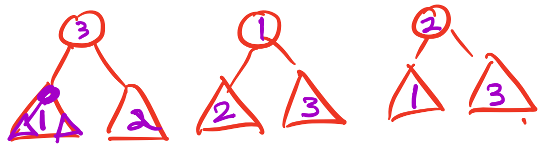
- Dictionary ("get elements by orderable key" container) may also require: Successor Predecessor Minimum Maximum and ability to process keys in order. Also called Dynamic Set

Binary Search Trees

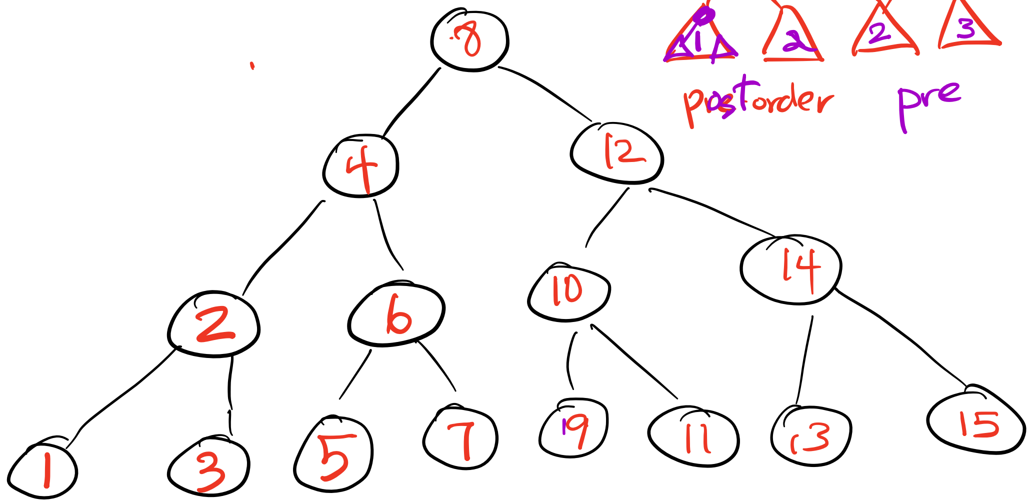
- rooted trees, each node having
  - a left child, which may be NULL
  - a right child, which may be NULL
  - a parent, which is NULL if the node is the root.

- elements (or pointers to them) are stored at the nodes; elements have keys

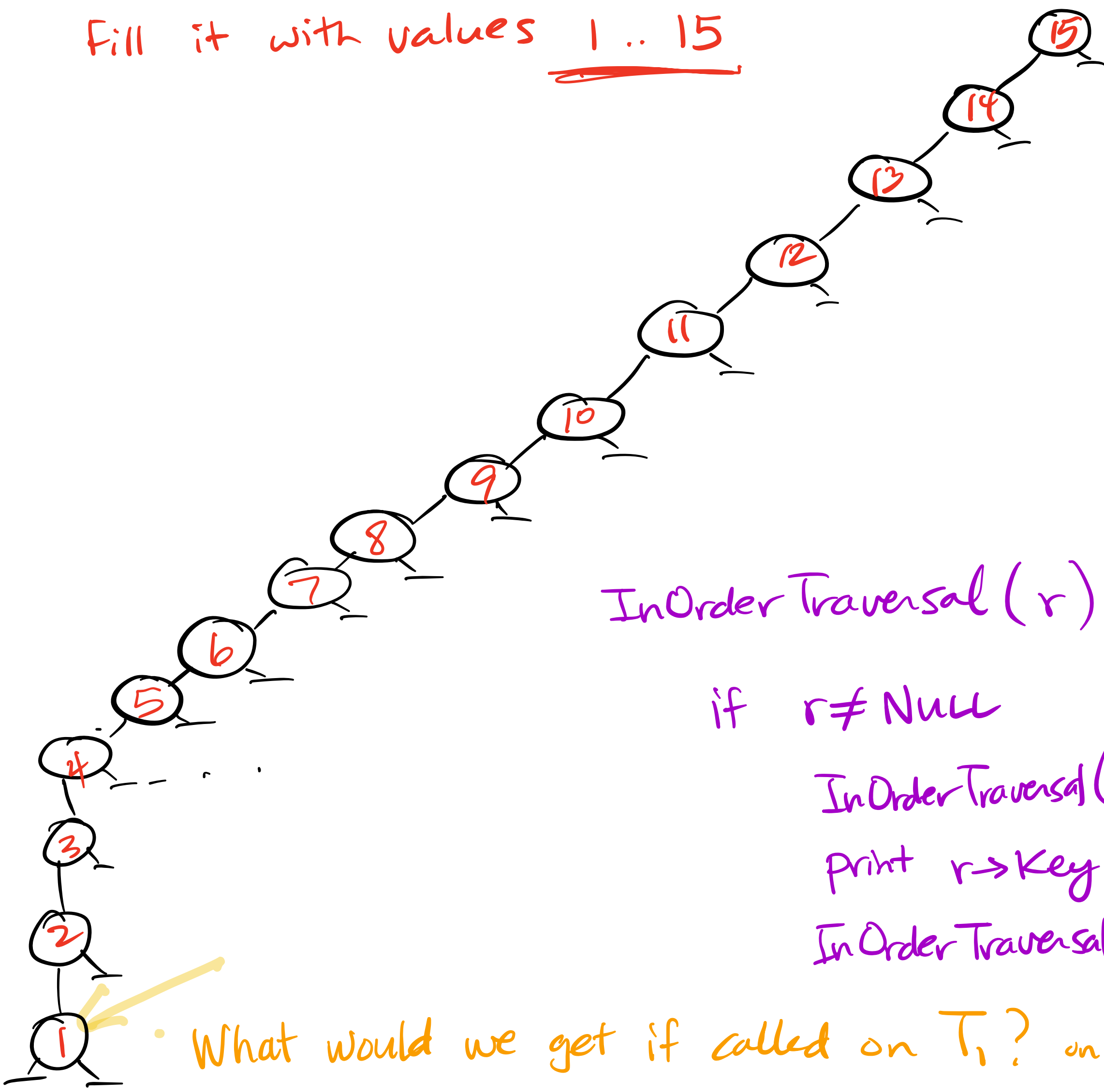
- The nodes are in BST-order:
  - for any node v
    keys in subtree $v \to left$ are $\leq v \to key$
    keys in subtree $v \to right$ are $\geq v \to key$



postorder pre



Fill it with values 1 .. 15



InOrderTraversal (r)

if r ≠ NULL

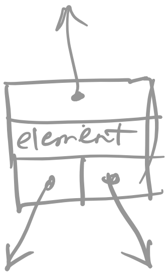
InOrderTraversal(r→left)

print r→key

InOrderTraversal(r→right)

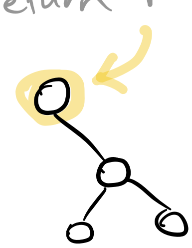
What would we get if called on T_1? on T_2?

```
BST_Search (r, k)    /* returns a node with key k
                     /* if exists, NULL o.w.        */
    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 the height of the tree.

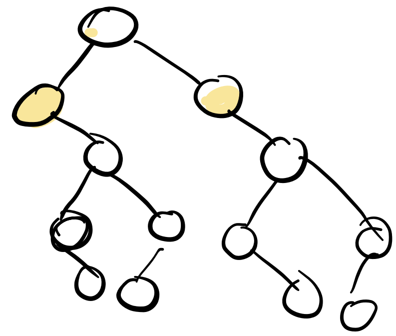
```
node* BST_Minimum (r   )
  /* return ptr to node with smallest key in tree*/
      if (r == NULL or r->left == NULL) return r
      else return BST_Minimum (r->left)
```



BST_Maximum (r)

/* return pointer to node with largest key in tree */

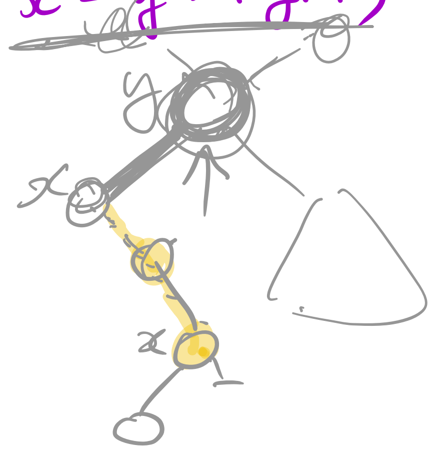
Exercise for student

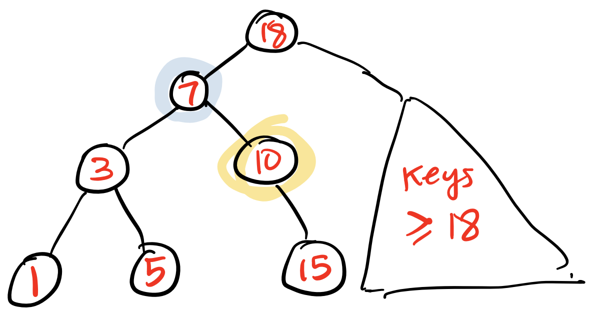


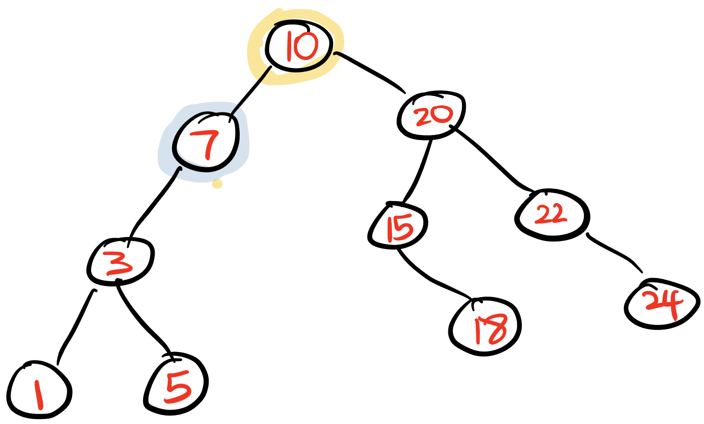
BST_Successor(x)

/* return pointer to node with smallest key > K */

```
if (x->right != NULL)
    return BST_Minimum(x->right)

y = x->parent
while (y != NULL and x = y->right)
    x = y
    y = y->parent

return y
```







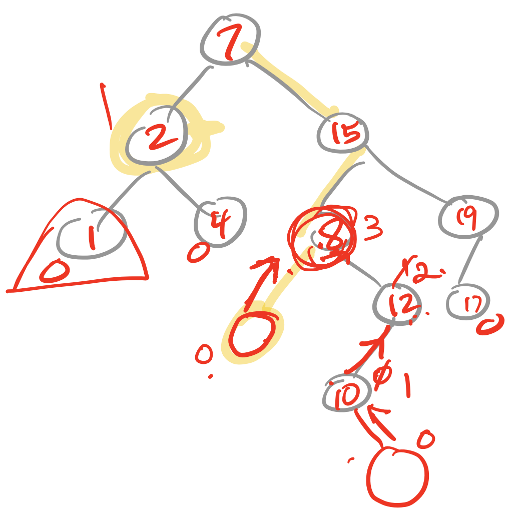
Theorem: The Dictionary operations

Search, Minimum, Maximum, Successor, Predecessor

can be implemented in $\Theta(h)$ time using a

BST of height h.

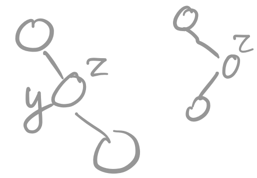
```
void BST_Insert(&r, e, k)
  /* Insert element e with key k into
  /* subtree rooted at r */
    if (r==NULL)
        r = new treenode(e, k)
    else if (r->key < k)
        BST_Insert(r->right, e, k)
    else BST_Insert(r->left, e, k).
```



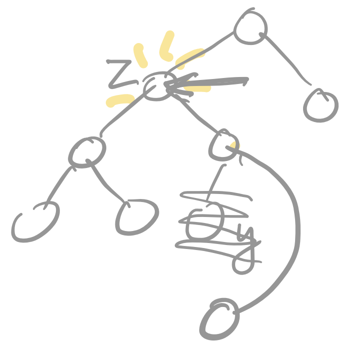
```
BST_Delete ( &r, z )
```

z is a pointer to a node in the tree rooted at r.

```
if (z->left == NULL or z->right == NULL)
        y = z.
else    y = BST_Successor(z)
```



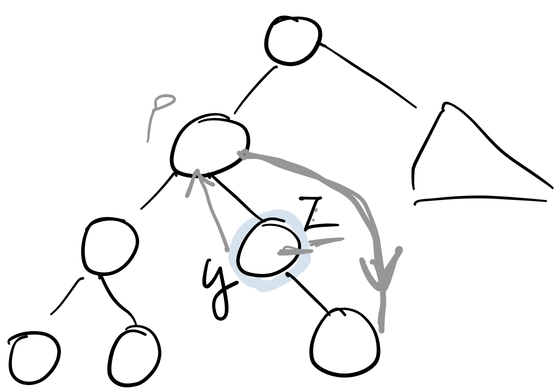

```
/* y is missing at least one child */
if      y->left != NULL
            x = y->left
else        x = y->right

if   x != NULL
         x->parent = y->parent
if   y->parent == NULL
         r = x
else if    y == y->parent->left
         y->parent->left = x
         else   y->parent->right = x.

if   y != z
```

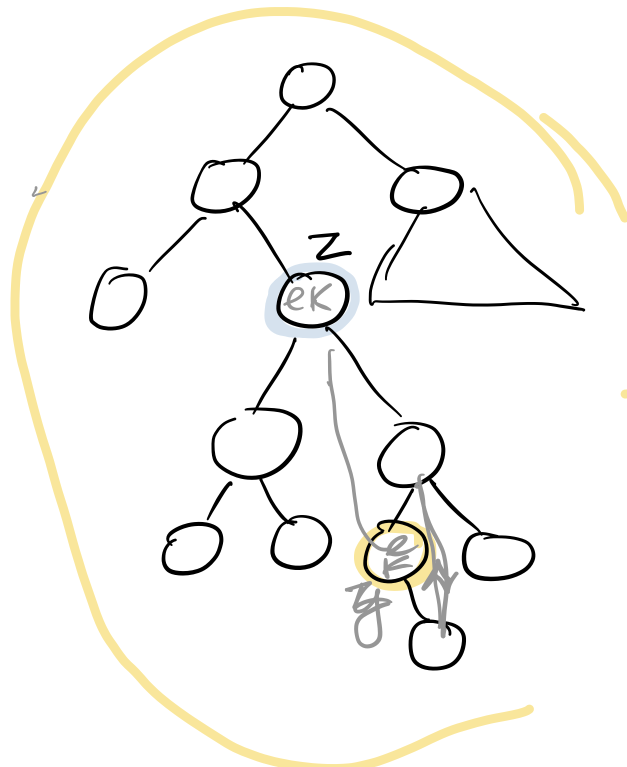


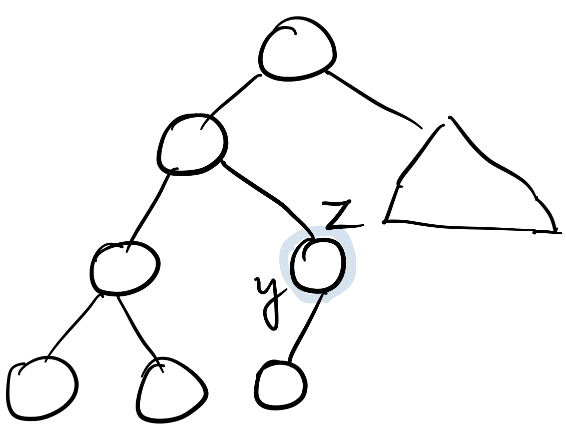
$z \to element = y \to element$

$z \to key = y \to key.$

return y.







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 $\Theta(\log n)$

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