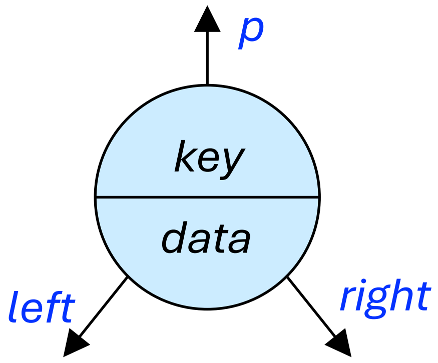
Binary Search Tree

Dynamic Set

A binary tree's node contains fields:

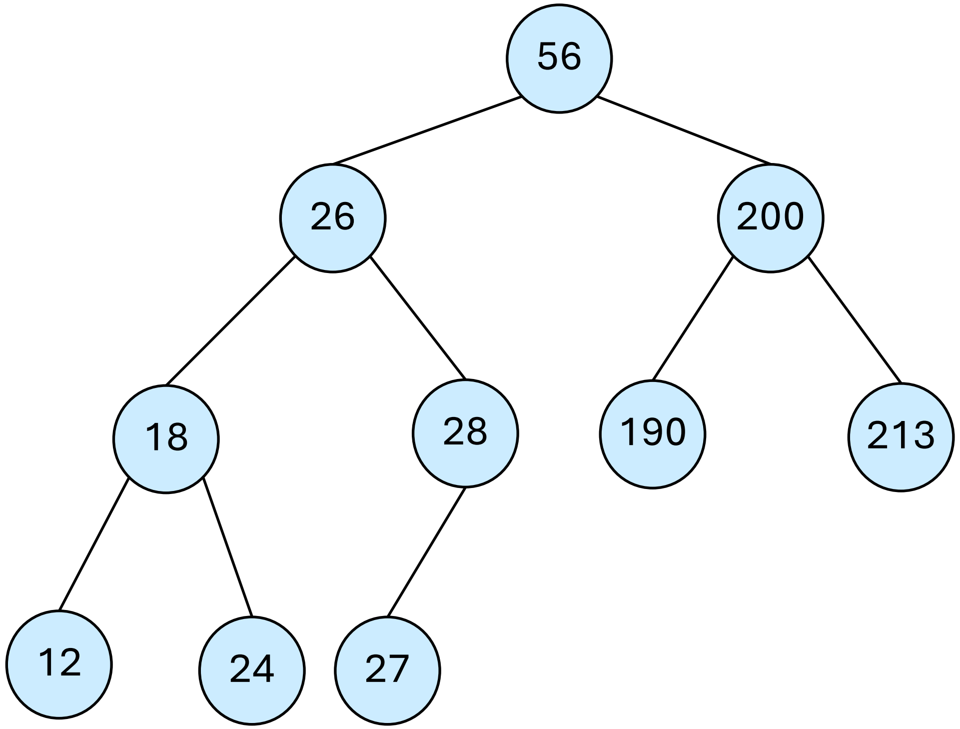
- *key*
- *left* – pointer to left child: root of left subtree.
- *right* – pointer to right child : root of right subtree.
- *p* – pointer to parent. *T.root.p* = NIL (optional)
- *satellite data*



Linked Structure

A binary tree is either

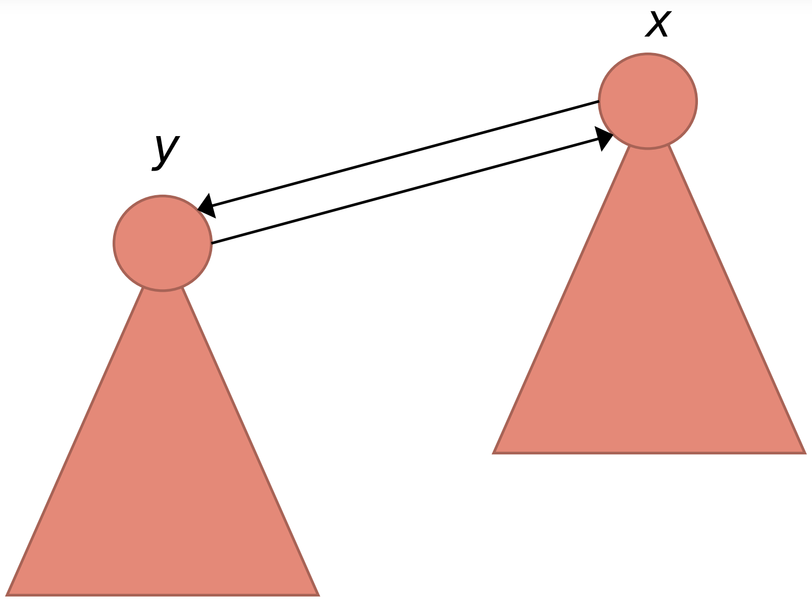
1. an empty tree
2. root node + left subtree + right subtree



Linking subtrees

x.left = y

y.parent = x



Binary Search Trees

A data structures that can support dynamic set operations.

- Search
- Minimum
- Maximum
- Predecessor
- Successor
- Insert
- Delete

Can be used to implement

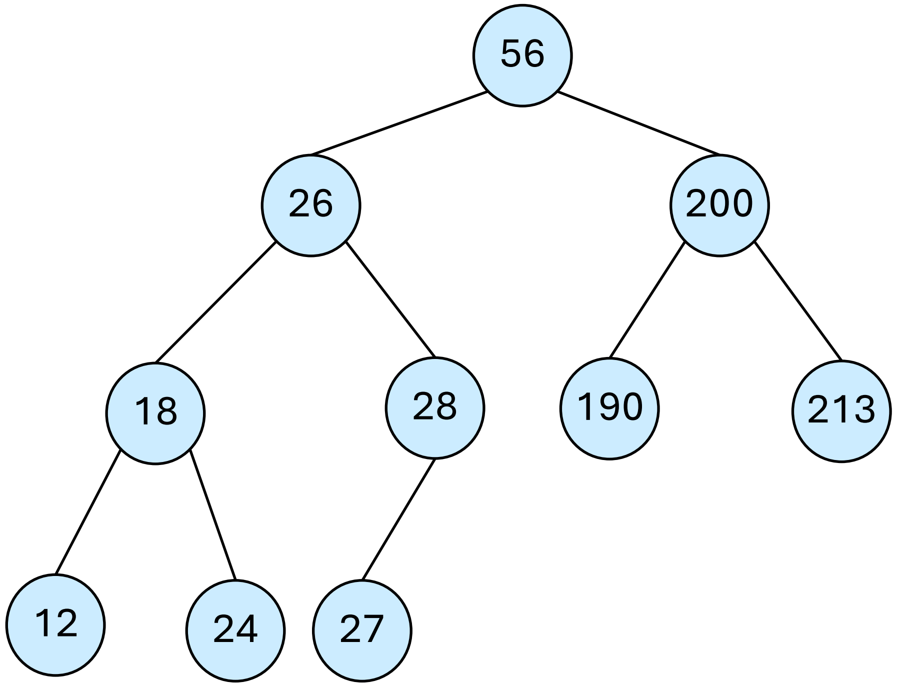
- Dictionaries.
- Priority Queues

The prescribed operations take time proportional to the height of the tree – $O(h)$.

Binary Search Tree Property

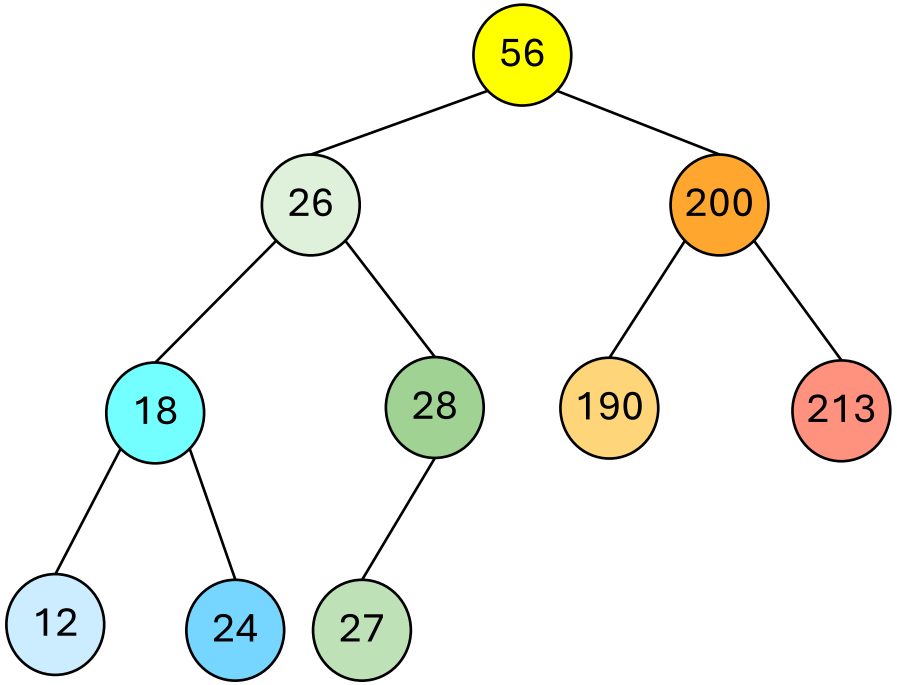
Stored keys must satisfy the *binary search tree* property.

- *key*[*y*] ≤ *key*[*x*], ∀ *y* in left subtree of x
- *key*[*y*] ≥ *key*[*x*], ∀ *y* in right subtree of *x*
- *In practice, key*[*y*] == *key*[*x*] *must be chosen to either left or right consistently.*



Inorder Traversal

```
INORDER-TREE-WALK(x)
if x ≠ NIL
    INORDER-TREE-WALK(x.left)
    print x.key
    INORDER-TREE-WALK(x.right)
```



Tree-Search

recursive

```
TREE-SEARCH(x, k)
if x == NIL or k == x.key
        return x
if k < x.key
        return TREE-SEARCH(x.left, k)
else return TREE-SEARCH(x.right, k)
```

iterative

```
ITERATIVE-TREE-SEARCH(x, k)
while x ≠ NIL and k ≠ x.key
        if k < x.key
                x = x.left
        else x = x.right
return x
```

Running time: $O(h)$

Minimum and Maximum

TREE-MINIMUM(x)

```
while x.left ≠ NIL
    x = x.left
return x
```

TREE-MAXIMUM(x)

```
while x.right ≠ NIL
    x = x.right
return x
```

Running time: $O(h)$

Successor and Predecessor

Successor(x) :

the smallest key that is $\geq x$

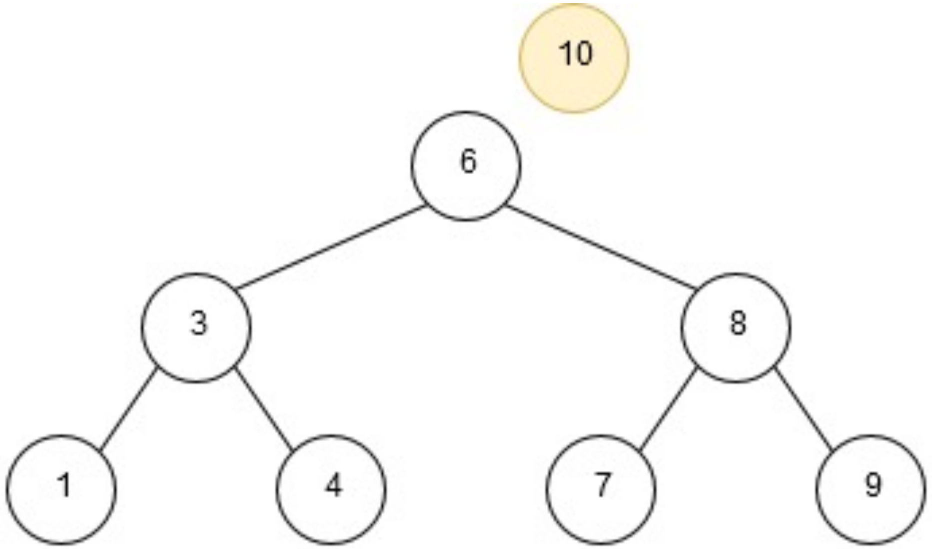
Predecessor(x) :

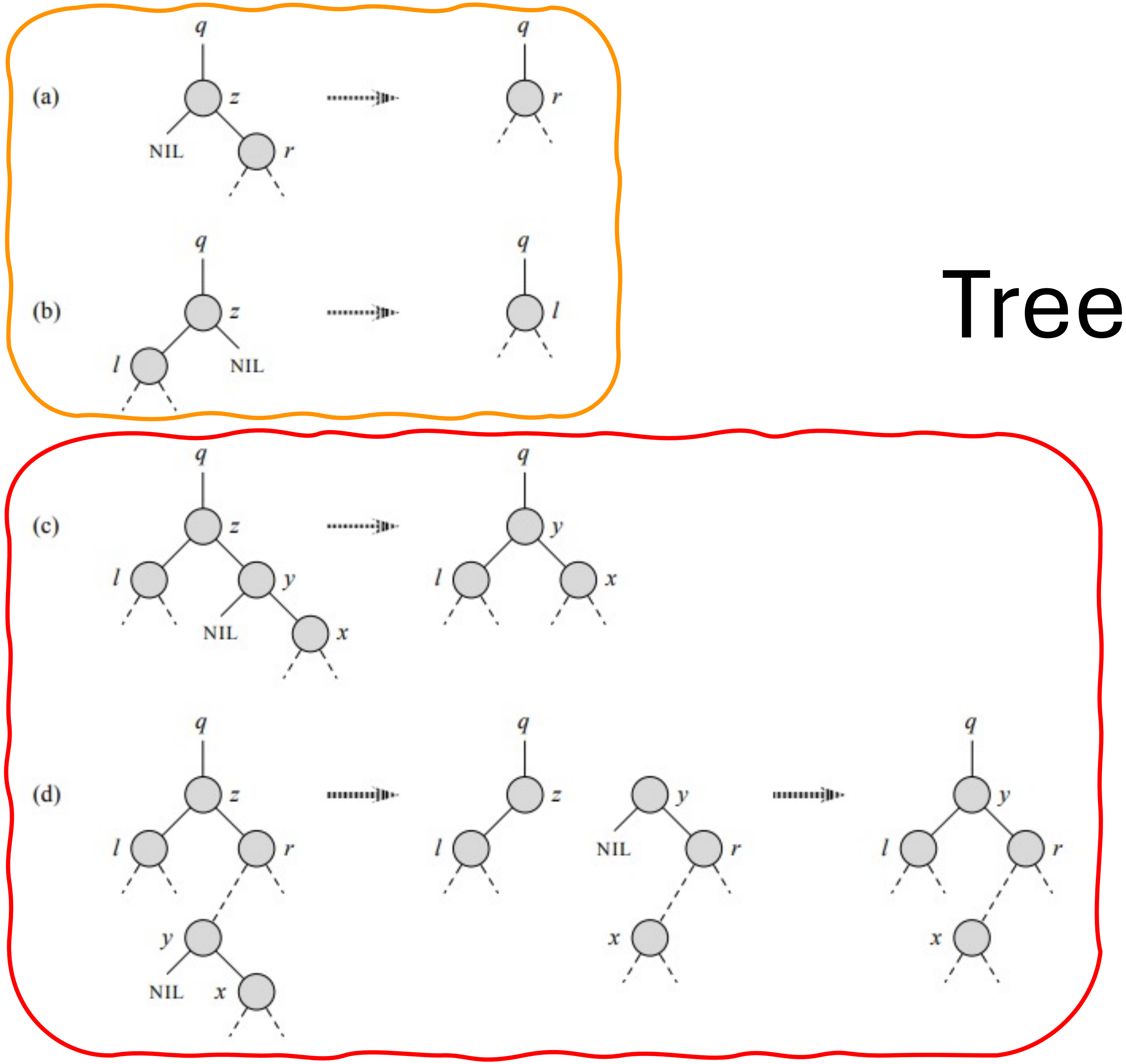
the largest key that is $\leq x$

```
TREE-SUCCESSOR(x)
if x.right ≠ NIL
    return TREE-MINIMUM(x.right)
y = x.p
while y ≠ NIL and x == y.right
    x = y
    y = y.p
return y
```

Insertion

```
TREE-INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z          // tree T was empty
elseif z.key < y.key
    y.left = z
else y.right = z
```





Tree-Delete(T, z)

Tree-Delete(T, z)

z has no child:	Just remove z, by having its parent point to NIL.
z has one child:	Switch things so the parent points to z’s child, instead of pointing to z.
z has two children:	Replace z with its successor s (which must be somewhere in the right subtree)

```
TRANSPLANT(T, u, v)
if u.p == NIL
    T.root = v
elseif u == u.p.left
    u.p.left = v
else u.p.right = v
if v ≠ NIL
    v.p = u.p
```

```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSPLANT(T, z, z.right)
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    if y.p ≠ z
        TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
```