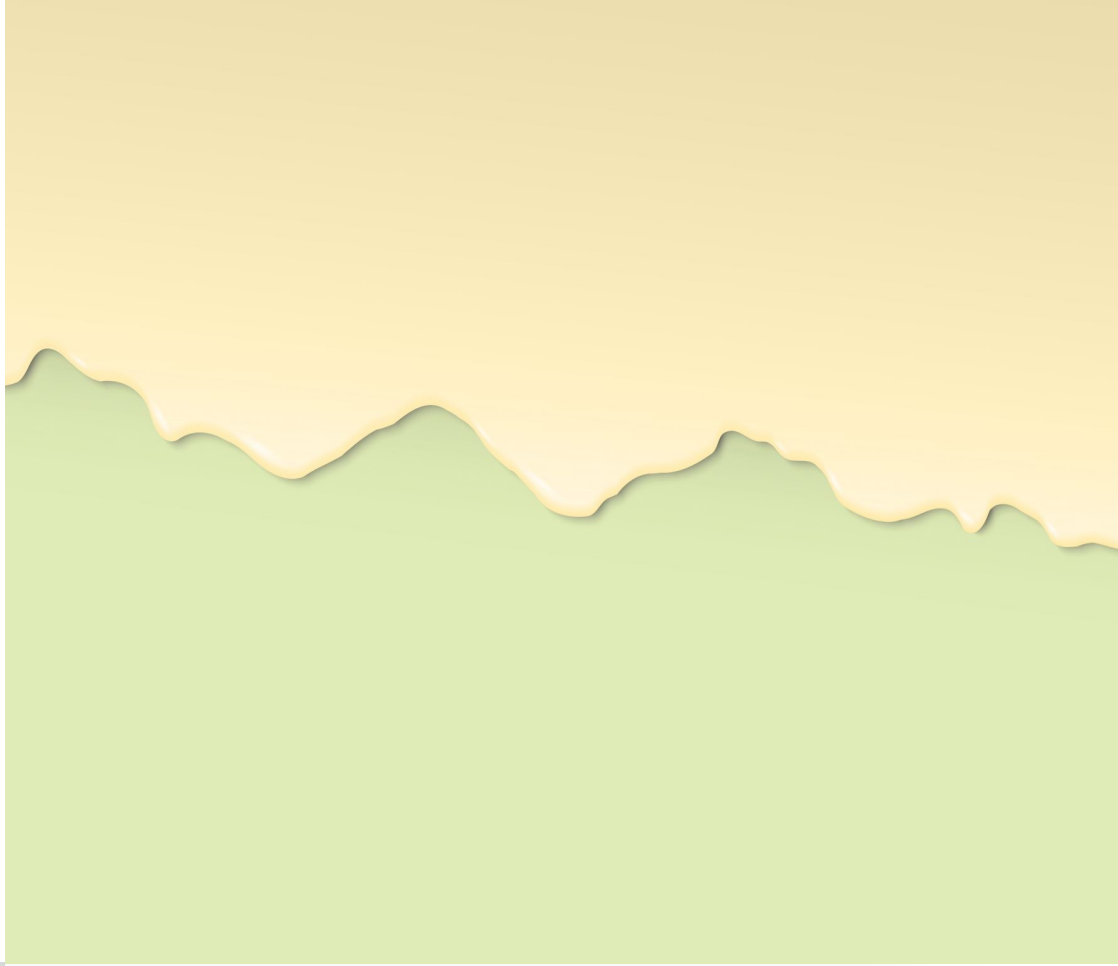
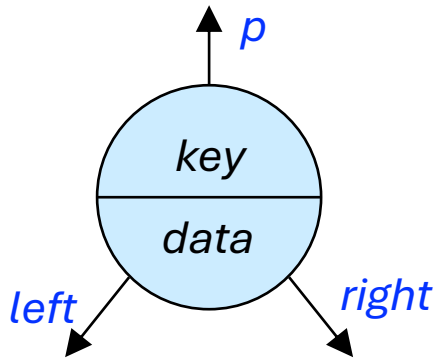
Binary Search Tree



Dynamic Set

A binary tree's node contains fields:

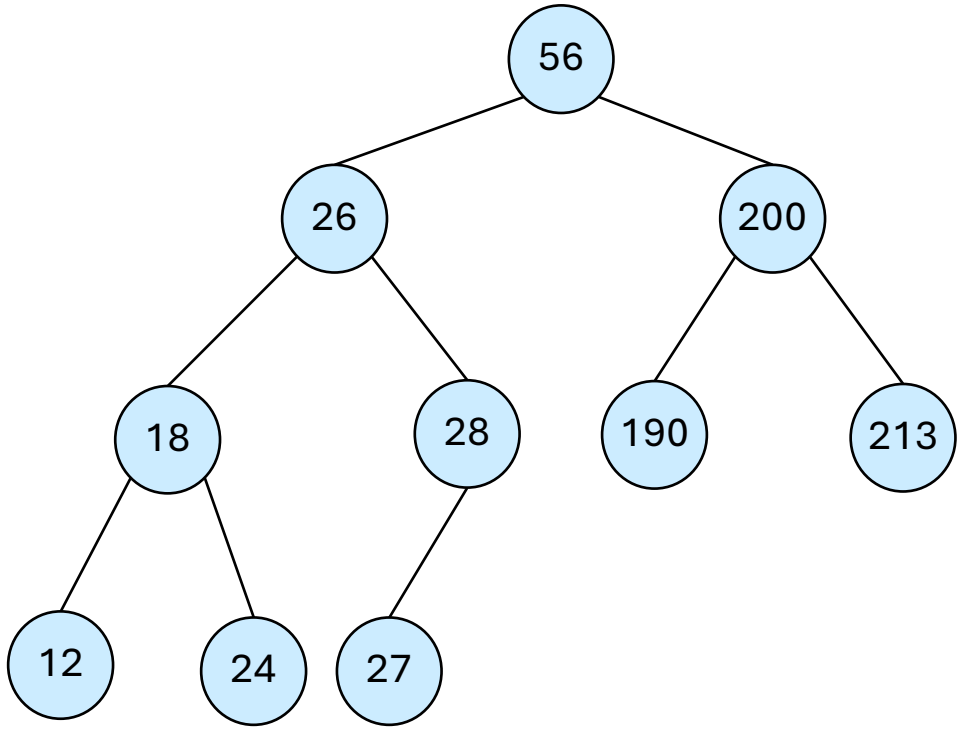
- *key*
- *left* – pointer to left child: root of left subtree.
- *right* – pointer to right child : root of right subtree.
- *p* – pointer to parent. *T.root.p* = NIL (optional)
- *satellite data*



Linked Structure

A binary tree is either

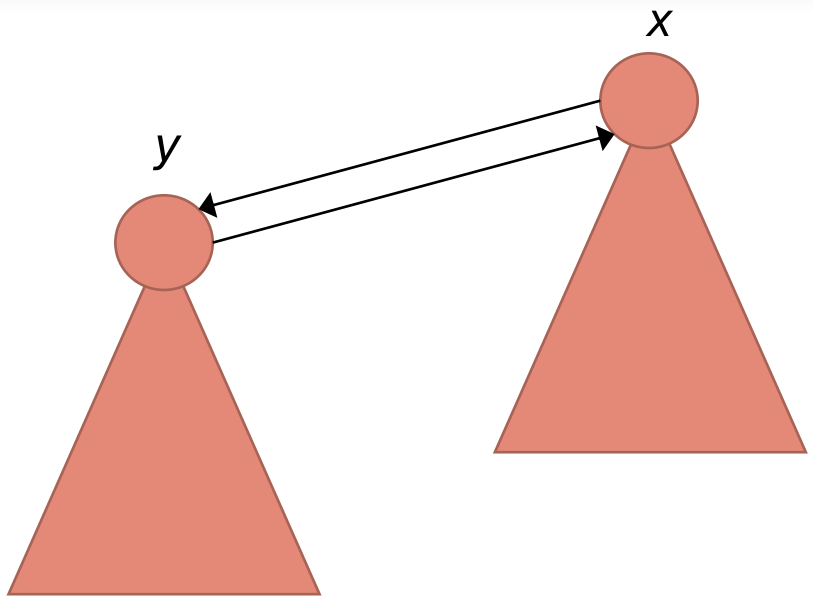
1. an empty tree
2. root node + left subtree + right subtree



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

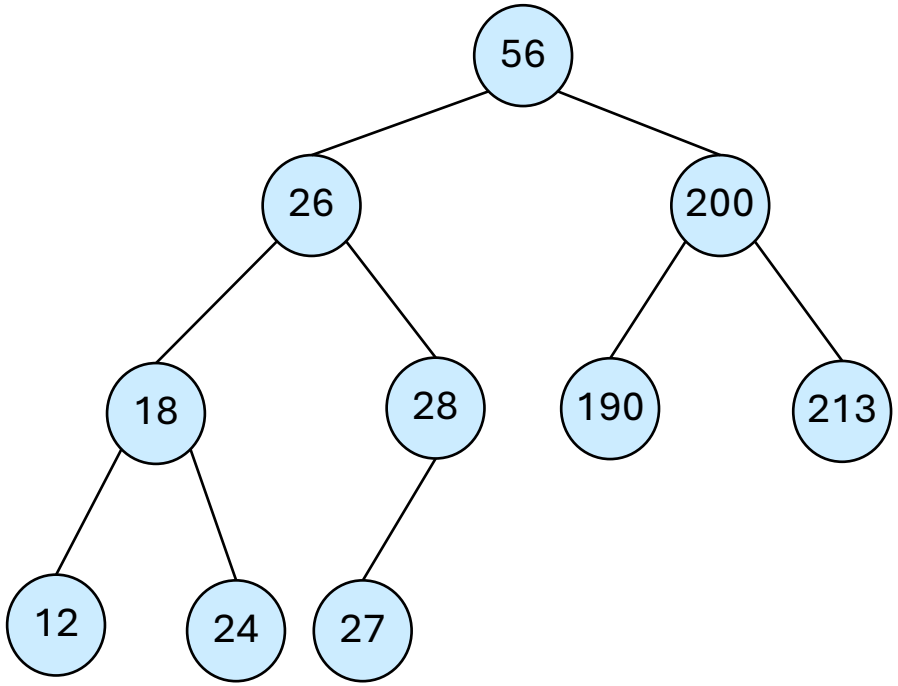
- Dictionaries.
- Priority Queues

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

Binary Search Tree Property

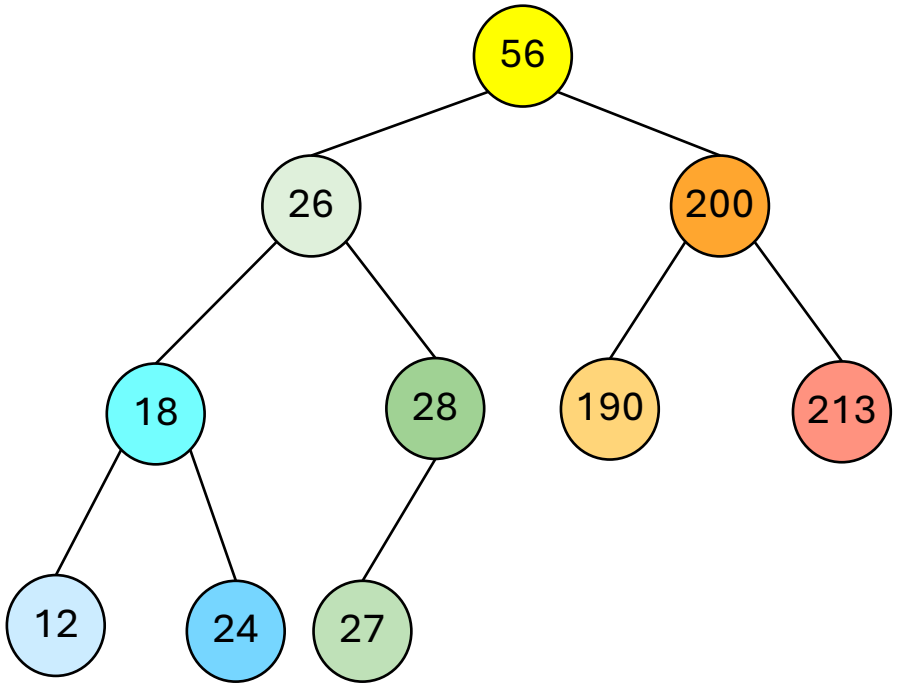
Stored keys must satisfy the *binary search tree* property.

- $key[y] \leq key[x]$, $\forall$ y in left subtree of x
- $key[y] \geq key[x]$, $\forall$ 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)
1  if x ≠ NIL
2      INORDER-TREE-WALK(x.left)
3      print x.key
4      INORDER-TREE-WALK(x.right)
```



Tree-Search

recursive

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

iterative

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

Running time: $O(h)$

Minimum and Maximum

```
TREE-MINIMUM(x)
1  while x.left ≠ NIL
2      x = x.left
3  return x
```

```
TREE-MAXIMUM(x)
1  while x.right ≠ NIL
2      x = x.right
3  return x
```

Running time: $O(h)$

Successor and Predecessor

Successor(x) :

the smallest key that is $\geq x$

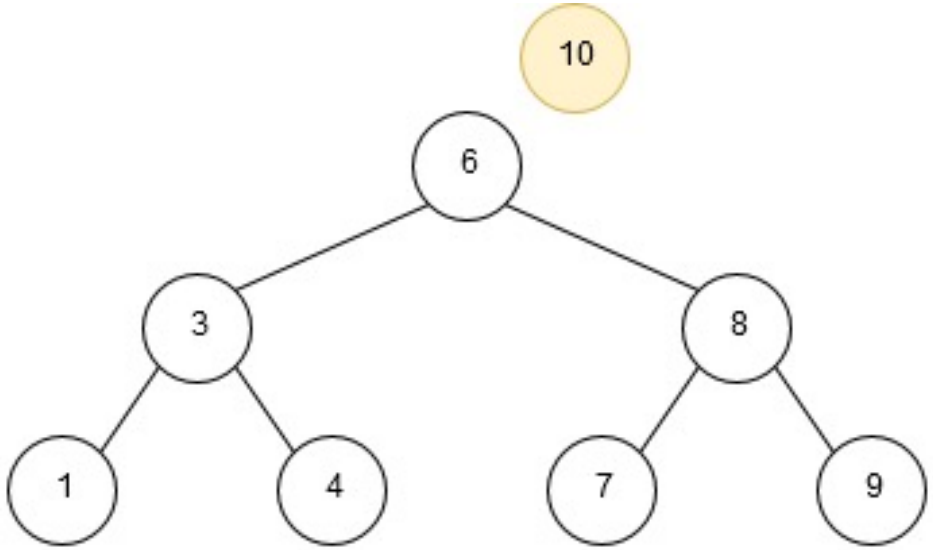
Predecessor(x) :

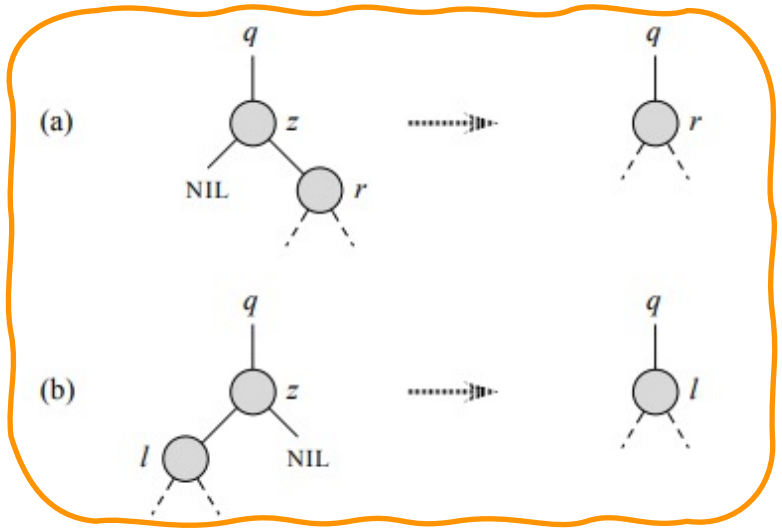
the largest key that is $\leq x$

```
TREE-SUCCESSOR(x)
1  if x.right ≠ NIL
2       return TREE-MINIMUM(x.right)
3  y = x.p
4  while y ≠ NIL and x == y.right
5       x = y
6       y = y.p
7  return y
```

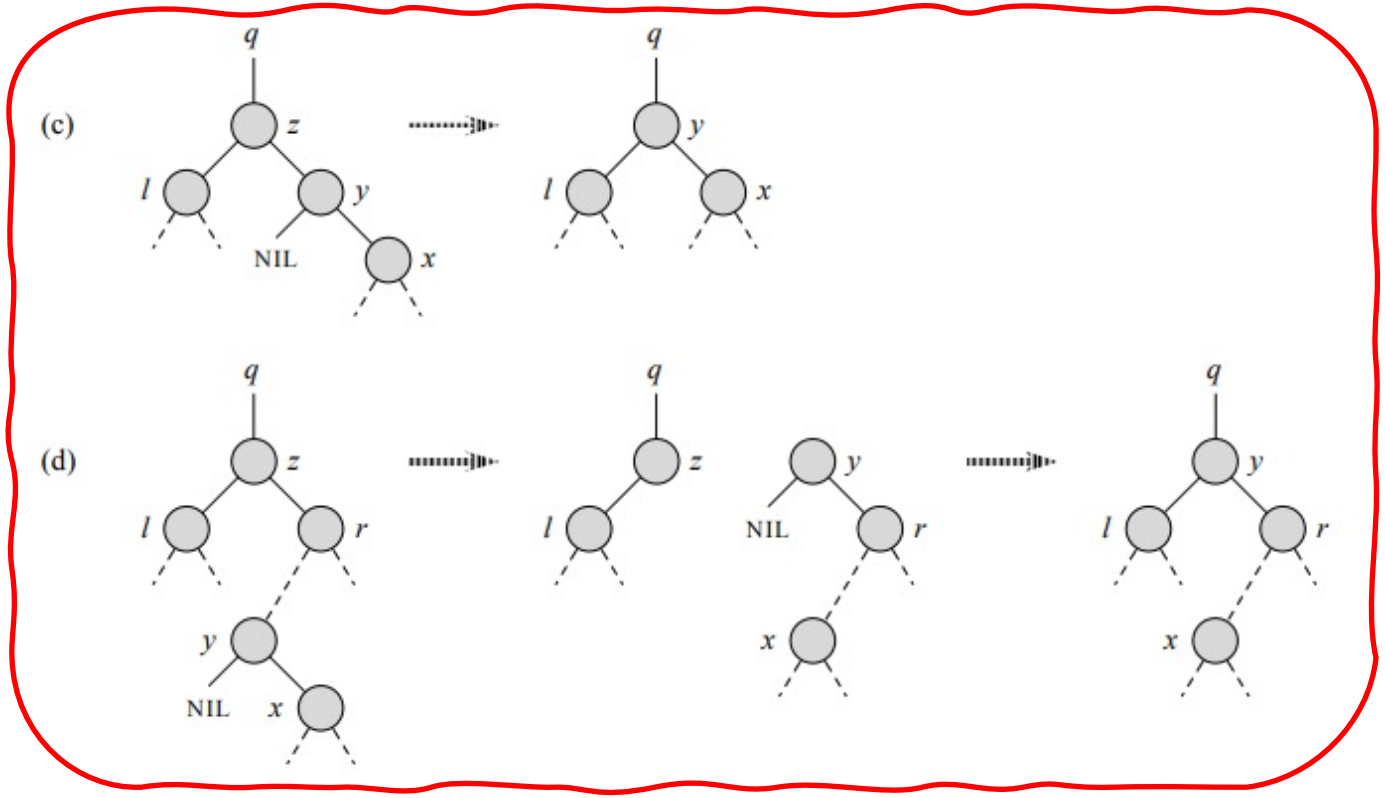
Insertion

```
TREE-INSERT(T, z)
 1   y = NIL
 2   x = T.root
 3   while x ≠ NIL
 4         y = x
 5         if z.key < x.key
 6               x = x.left
 7         else x = x.right
 8   z.p = y
 9   if y == NIL
10         T.root = z          // tree T was empty
11   elseif z.key < y.key
12         y.left = z
13   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)
1  if u.p == NIL
2      T.root = v
3  elseif u == u.p.left
4      u.p.left = v
5  else u.p.right = v
6  if v ≠ NIL
7      v.p = u.p
```

```
TREE-DELETE(T, z)
 1  if z.left == NIL
 2      TRANSPLANT(T, z, z.right)
 3  elseif z.right == NIL
 4      TRANSPLANT(T, z, z.left)
 5  else y = TREE-MINIMUM(z.right)
 6      if y.p ≠ z
 7          TRANSPLANT(T, y, y.right)
 8          y.right = z.right
 9          y.right.p = y
10      TRANSPLANT(T, z, y)
11      y.left = z.left
12      y.left.p = y
```