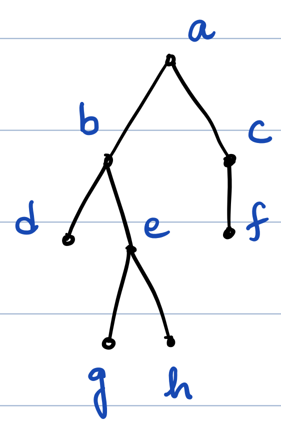
Binary Search tree

Binary tree :

It is a tree in which every node|vertex has at most two children.

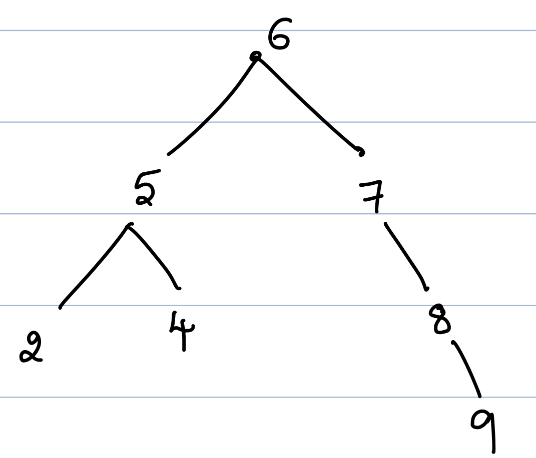


- Binary search tree is a binary tree with the following Property.

The keys in a BST are always stored in such a way as to satisfy the binary-search-tree property

Let x be a node in a BST. If y is a node in the left subtree of x, then $y.key \le x.key$. If y is a node in the right subtree of x, then $y.key \ge x.key$

Eg:



We can represent BST by linked datastructure in which each node is an object.

In addition to a key and Satellite data, each node Contains attributes left, right and p that Point to the nodes Corresponding to its left child, its right child and its Parent respectively.

If a Child or a Parent is missing the appropriate attribute Contains the Value NIL.

The root node is the only node in the tree whose Parent is NIL.

Representation of a binary tree

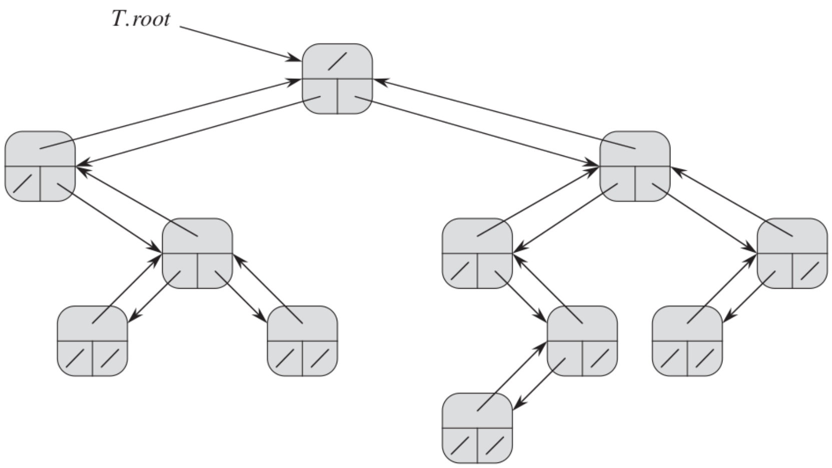


Figure 10.9 The representation of a binary tree T. Each node x has the attributes $x.p$ (top), $x.left$ (lower left), and $x.right$ (lower right). The *key* attributes are not shown.

The BST Property allows us to Print out all the keys in a BST in sorted order by Performing a inorder tree walk.

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

Exercise:

① For the set of $\{1, 4, 5, 10, 16, 17, 21\}$ of keys, draw binary search trees of heights 2, 3, 4, 5 and 6.

Searching in a BST

Given a Pointer to the root of the tree and a key k, TREE-SEARCH returns a Pointer to a node with key k if One exists; otherwise, it returns NIL.

```
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)
```

Running time: $O(h)$, where h is the height of the tree.

Iterative algorithm

```
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
```

Minimum and Maximum

The following algorithm returns a Pointer to the minimum element in the subtree rooted at a given node x, ($\neq$ NIL)

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

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

Successor and Predecessor

If all keys are distinct, the successor of a node x is the node with the smallest key greater than $x.key$.

The structure of a BST allows us to determine the successor of a node without ever comparing keys.

Lemma: Let T be a BST. If a node z in T has two children then z's successor has no left child and z's predecessor has no right child.

Proof Spse z has two children, then we know that its successor y is the minimum element of the BST rooted at z.right. If y had a left child then it wouldn't be the minimum element. So y must not have a left child.

Similarly, the predecessor has no right child.

Lemma: Let T be a binary tree whose all keys are distinct. Let x be a node in T such that the right subtree of x is empty.
Show that if x has a successor y then y is the lowest ancestor of x whose left child is also an ancestor of x.

Proof: Claim: y is an ancestor of x

Spse y is not an ancestor of x, then y & x have a common ancestor z.

By BST Property we have $x < z < y$,

$\therefore$ y cannot be successor of x.

Observe that $y.left$ must be an ancestor of x otherwise $y.right$ would be an ancestor of x implying $x > y$.

Spse y is not the lowest ancestor of x whose left child is also an ancestor of x. Let z denote the lowest ancestor, then z must be in the left subtree of y, which implies $z < y$, contradicting that fact that y is the successor of 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
```

TREE-SUCCESSOR has two cases.

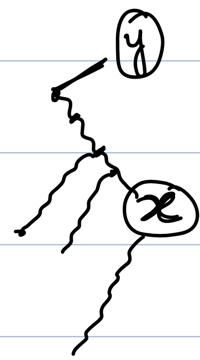
(a) The right subtree of node x is non-empty.

In this case the successor of x is just the leftmost node in x's right subtree, which can be found using TREE-MINIMUM(x.right).

(b) The right subtree of node x is empty.

If x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x.

To Find y, we simply go up the tree from x until we encounter a node that is the left child of its Parent.



Running time is $O(h)$

- The Procedure TREE-PREDECESSOR, is symmetric to TREE-SUCCESSOR.

TREE_PREDECESSOR (x)

if x.left $\neq$ NIL then

return TREE-MAXIMUM (x.left)

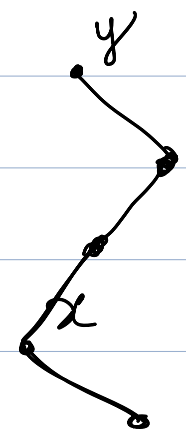
$y = x.p$

while $y \neq$ NIL and $x == y$.left

$x = y$

$y = y.p$

return y.



Insertion:

To insert a new value v into a BST T, we use the procedure TREE-INSERT.

The procedure takes a node z for which $z.key = v$, $z.left = NIL$ and $z.right = NIL$.

It modifies T and some of the attributes of z in such a way that it inserts z into an appropriate position in the tree.

```
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
```

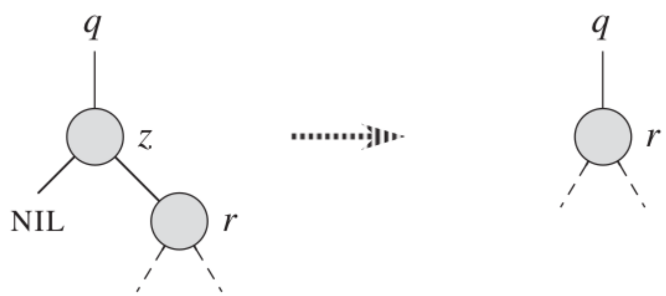
Deletion

The overall strategy for deleting a node z from a binary search tree T has three basic cases but, as we shall see, one of the cases is a bit tricky.

- If z has no children, then we simply remove it by modifying its parent to replace z with NIL as its child.
- If z has just one child, then we elevate that child to take z's position in the tree by modifying z's parent to replace z by z's child.
- If z has two children, then we find z's successor y—which must be in z's right subtree—and have y take z's position in the tree. The rest of z's original right subtree becomes y's new right subtree, and z's left subtree becomes y's new left subtree. This case is the tricky one because, as we shall see, it matters whether y is z's right child.

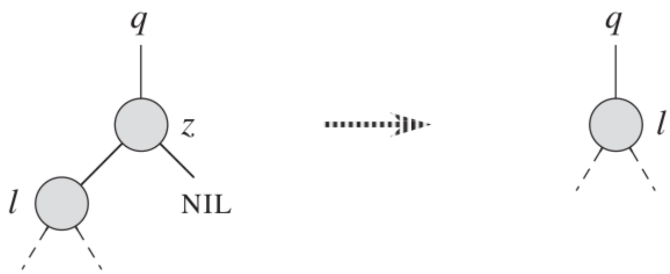
Case 1:

If z has no left child (part (a) of the figure), then we replace z by its right child, which may or may not be NIL. When z's right child is NIL, this case deals with the situation in which z has no children. When z's right child is non-NIL, this case handles the situation in which z has just one child, which is its right child.



Case 2 :

If z has just one child, which is its left child (part (b) of the figure), then we replace z by its left child.



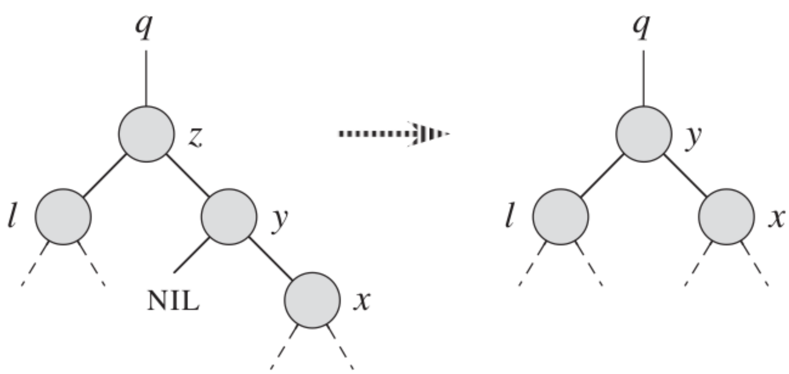
Case 3 :

z has both a left and a right child. We find z's successor y, which lies in z's right subtree and has no left child (see Exercise 12.2-5). We want to splice y out of its current location and have it replace z in the tree.

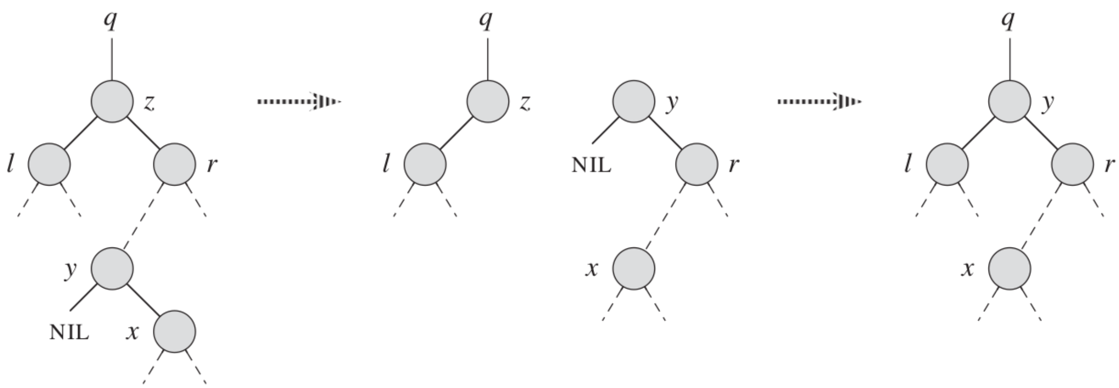
(a) • If y is z's right child (part (c)), then we replace z by y, leaving y's right child alone.

(b) • Otherwise, y lies within z's right subtree but is not z's right child (part (d)). In this case, we first replace y by its own right child, and then we replace z by y.

(a)



b



In order to move subtrees around within the binary search tree, we define a subroutine TRANSPLANT, which replaces one subtree as a child of its parent with another subtree. When TRANSPLANT replaces the subtree rooted at node u with the subtree rooted at node v, node u's parent becomes node v's parent, and u's parent ends up having v as its appropriate child.

```
TRANSPLANT(T, u, v)
if u.p == NIL
    T.root = v
elseif u == u.p.left        // checks whether u is a left child of its parent
    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)        → Finds successor of z.
    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
```

Running time $O(h)$

Exercise :

① Suppose the keys of BST are not distinct then how to perform INSERT, DELETE, SUCCESSOR operations.