BST = BT + some property ← Organize data within BT

Data Structures and Algorithms

Lecture 22: Binary Search Trees

search data faster

Nopadon Juneam
Department of Computer Science
Kasetsart university

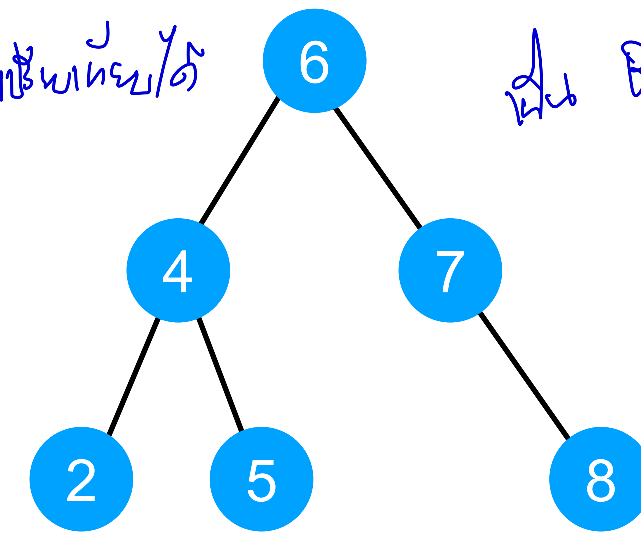
Outlines

- Basics of binary search trees
  - Binary-search-tree property
  - Traversals of binary search trees
- Common search operations on binary search trees
- Update operations on binary search trees

Binary Search Trees

- A *binary search tree* is a data structure organized in a *binary tree*

หยิบ 2 สมาชิกใดๆ ใน set มาเปรียบเทียบได้



เป็น BST

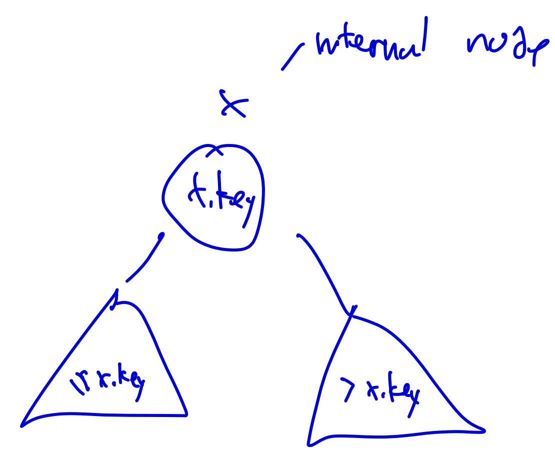
- Let *S* be a set of keys with a total order on S. For example, *S* is a set of integers. A ***binary search tree*** *whose nodes are associated with keys in S* is a *binary tree T* such that:

Node เทียบได้โดยการเทียบ key

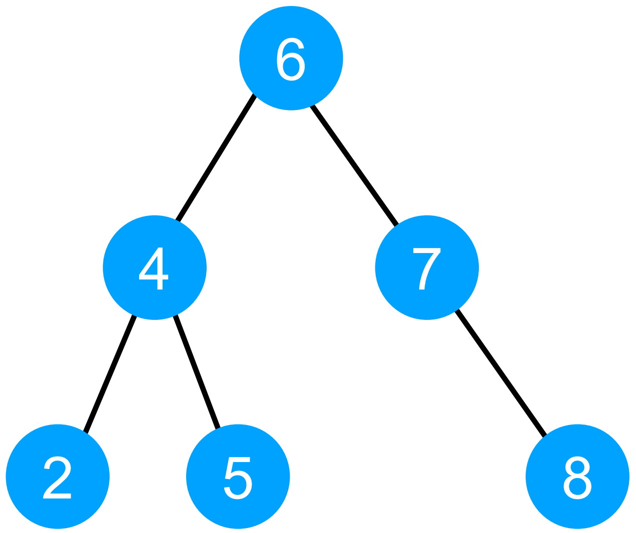
  - Each node *x* of *T* stores an element of *S*, denoted with *x.key*

สำคัญสุดใน BST

  - For each internal node *x* of *T*, the elements stored in the left subtree of *x* are less than or equal to *x.key* and the elements stored in the right subtree of *x* are greater than to *x.key*



Binary-Search-Tree Property



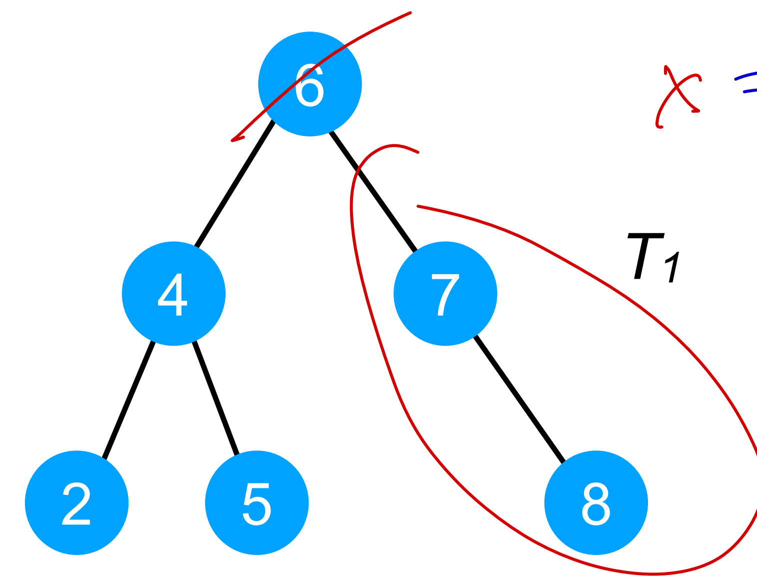
- The keys (elements) in a binary search tree are always stored in such a way as to satisfy the ***binary-search-tree property***:

"Let x be a node in a binary search tree. If y is a node in the left subtree of x, then y.key ≤ x.key. If y is a node in the right subtree of x, then y.key > x.key."

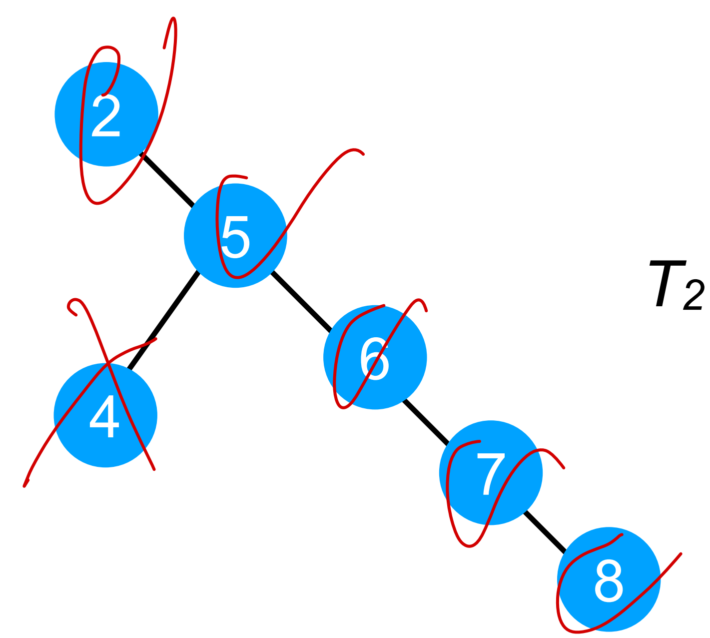
ข้อมูลที่ซ้ำกัน = ไป ที่ subtree ได้

Examples of Binary Search Trees

- For any node *x*, the keys in the left subtree of *x* are at most *x.key*, and the keys in the right subtree of *x* are greater than *x.key*
- Different binary search trees can represent the same set of key values
- The worst-case running time of most binary-search-tree operations is proportional to the height of the tree.
  - T_1 is a binary search of height 2
  - T_2 is considered as a less efficient binary search tree as the height of T_2 is 4



x = 11

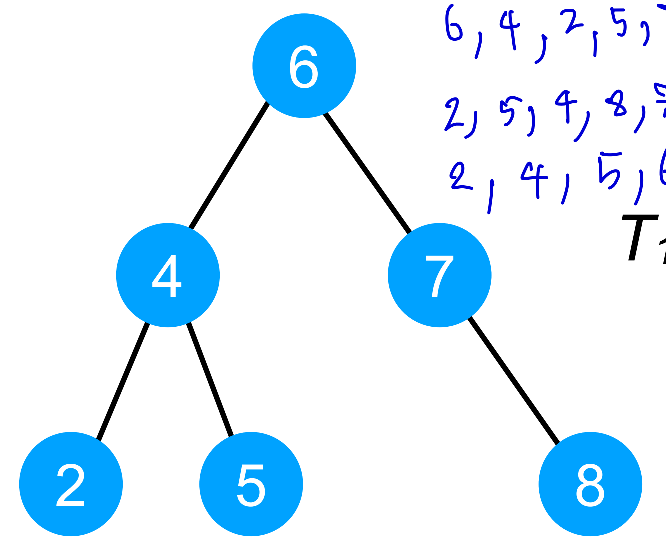


ค้นเยอะเพราะ
height T สูงเยอะ

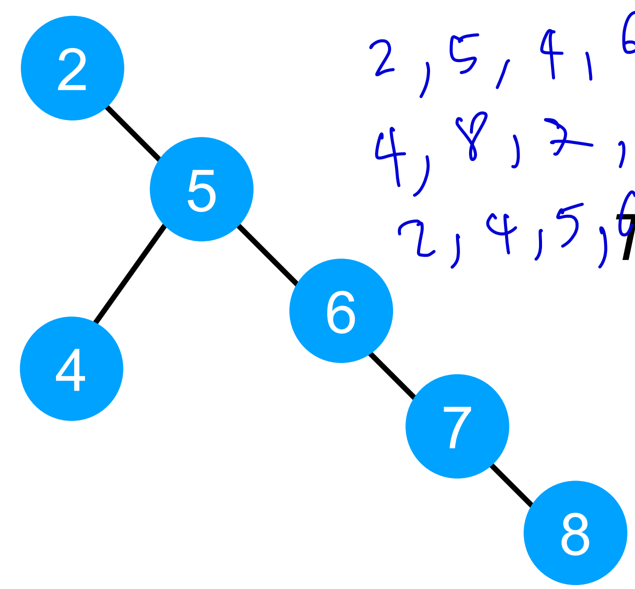
Traversals of Binary Search Trees

- Preorder traversal:
  - T_1: 6, 4, 2, 5, 7, 8
  - T_2: 2, 5, 4, 6, 7, 8
- Postorder traversal:
  - T_1: 2, 5, 4, 8, 7, 6
  - T_2: 4, 8, 7, 6, 5, 2
- **Inorder traversal: √ ธงเดิน ?
  - T_1: 2, 4, 5, 6, 7, 8
  - T_2: 2, 4, 5, 6, 7, 8
- *Remark*: By the binary-search-tree property, an *inorder traversal* of a binary search tree T always visits the elements in *sorted order* of keys.

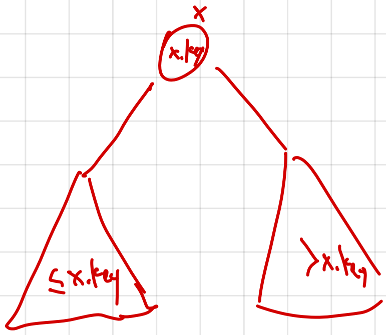
Q: ทำไม inorder traversal
ถึง visit Node ตามลำดับ
การเพิ่มขึ้นของ key



6, 4, 2, 5, 7, 8
2, 5, 4, 8, 7, 6
2, 4, 5, 6, 7, 8

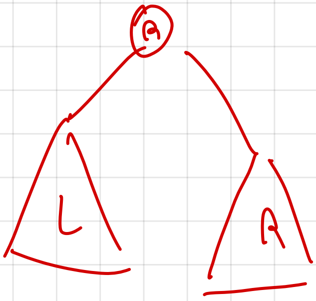


2, 5, 4, 6, 7, 8
4, 8, 7, 6, 5, 2
2, 4, 5, 6, 7, 8



Inorder:

1. Recursivly traverse left subtree
2. visit x
3. Recursivly traverse right subtree



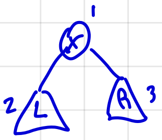
$(L \leq r \leq R)$

$\Rightarrow$ L r R

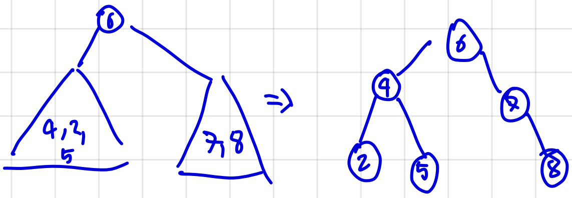
$L' \leq r_1 \leq R'$

Pre-order result ↓

6, 4, 2, 5, 7, 8



$\Rightarrow$



Reconstruct BST from preorder result.

Common Search Operations on Binary Search Trees

- Common operations performed on a binary search tree *T*:
  - search(*k*, *T*): return an object of a node whose key value is *k* in *T*
  - minimum(*T*): return an object of a node with the smallest key value in *T*
  - maximum(*T*): return an object of a node with the largest key value in *T*
  - successor(*x*, *T*): return an object of the successor of node *x* (the node with the smallest key value greater than *x.key*)
  - predecessor(*x*, *T*): return an object of the predecessor of node *x* (the node with the largest key smaller than *x.key*)

Operation: Search

- search(k, T): return an object of a node whose key value is k in T

```
struct node
{
    int key;
    struct node* parent;
    struct node* left;
    struct node* right;
};
```

as Root

```
struct node* search(int key, struct node* node)
{
    if ((node == NULL) || (key == node->key))
        return node;
    if (key < node->key)
        return search(key, node->left); // recurse on left subtree
    else
        return search(key, node->right);
}
```

Operation: Minimum

หา Node ซึ่งมีค่า key ต่ำที่สุด

- minimum(*T*): return an object of a node with the smallest key value in *T*

ดูซ้ายอย่างเดียว

```
struct node
{
   int key;
   struct node* parent;
   struct node* left;
   struct node* right;
};
```

```
struct node* minimum(struct node* node)
{
   while(node->left != NULL)
      node = node->left;
   return node;
}
```

Operation: Maximum

- maximum(*T*): return an object of a node with the largest key value in *T*

```
struct node
{
    int key;
    struct node* parent;
    struct node* left;
    struct node* right;
};
```

ไปขวาอย่างเดียว

```
struct node* maximum(struct node* node)
{
    while(node->right != NULL)
        node = node->right;
    return node;
}
```

Operation: Successor

- successor(*x*, *T*): return an object of the successor of node *x* (the node with the smallest key value greater than x.key)

```
struct node* successor(struct node* node)
{
   if(node->right != NULL)
      return minimum(node->right);
   struct node* ancestor = node->parent;
   while((ancestor != NULL) && (node == ancestor->right))
   {
      node = ancestor;
      ancestor = node->parent;
   }
   return ancestor;
}
```

successor := ผู้ที่ตามมา , สิ่งที่ตามมา

Predecessor ผู้มาก่อน

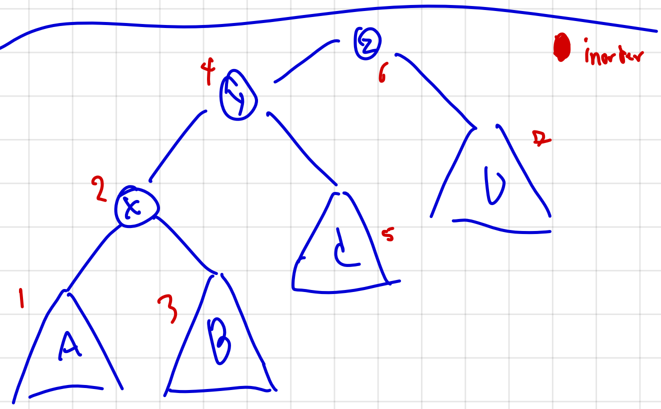
successor(x) :=

↑ Obj of Node x

→ มองหาโนดที่มีค่ามากกว่า x.key และมีค่า key ต่ำที่สุด

$S = \{2, 4, 6, 0, 7\}$

$0 \le \underset{x}{2} \le \underset{\text{successor}(x)}{4} \le 6 \le 7$



successor(x), หรือ q,

Case 1: กรณีที่ B ไม่ใช่ empty โดยที่ B คือ right subtree ของ x

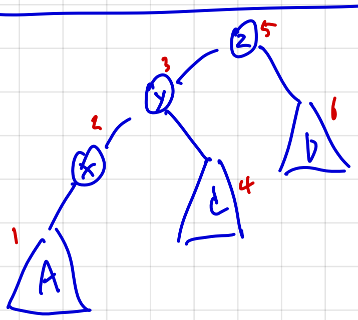
=> จาก inorder traversal เราสรุปได้ว่า q ต้องอยู่ใน B

มากไปกว่านั้น q ต้องเป็น minimum ใน B

case 2: กรณีที่ B เป็น empty

=> จาก Inorder traversal เราสรุปได้ว่า q ต้องเป็น left ancestor ของ x

left ancestor : ancestor ของ x ซึ่งมี ancestor ของ x อีกตัว เป็นลูกทางด้านซ้าย



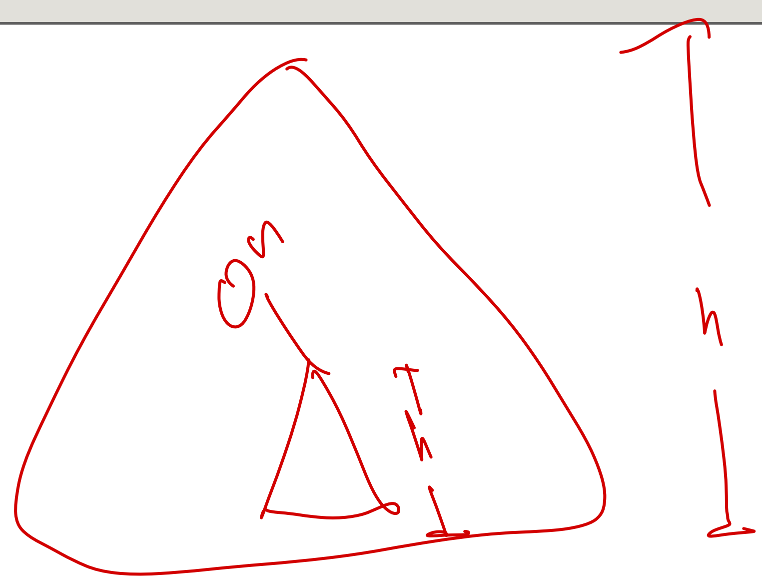
Operation: Predecessor

- predecessor(*x*, *T*): return an object of the predecessor of node *x* (the node with the largest key smaller than *x.key*)

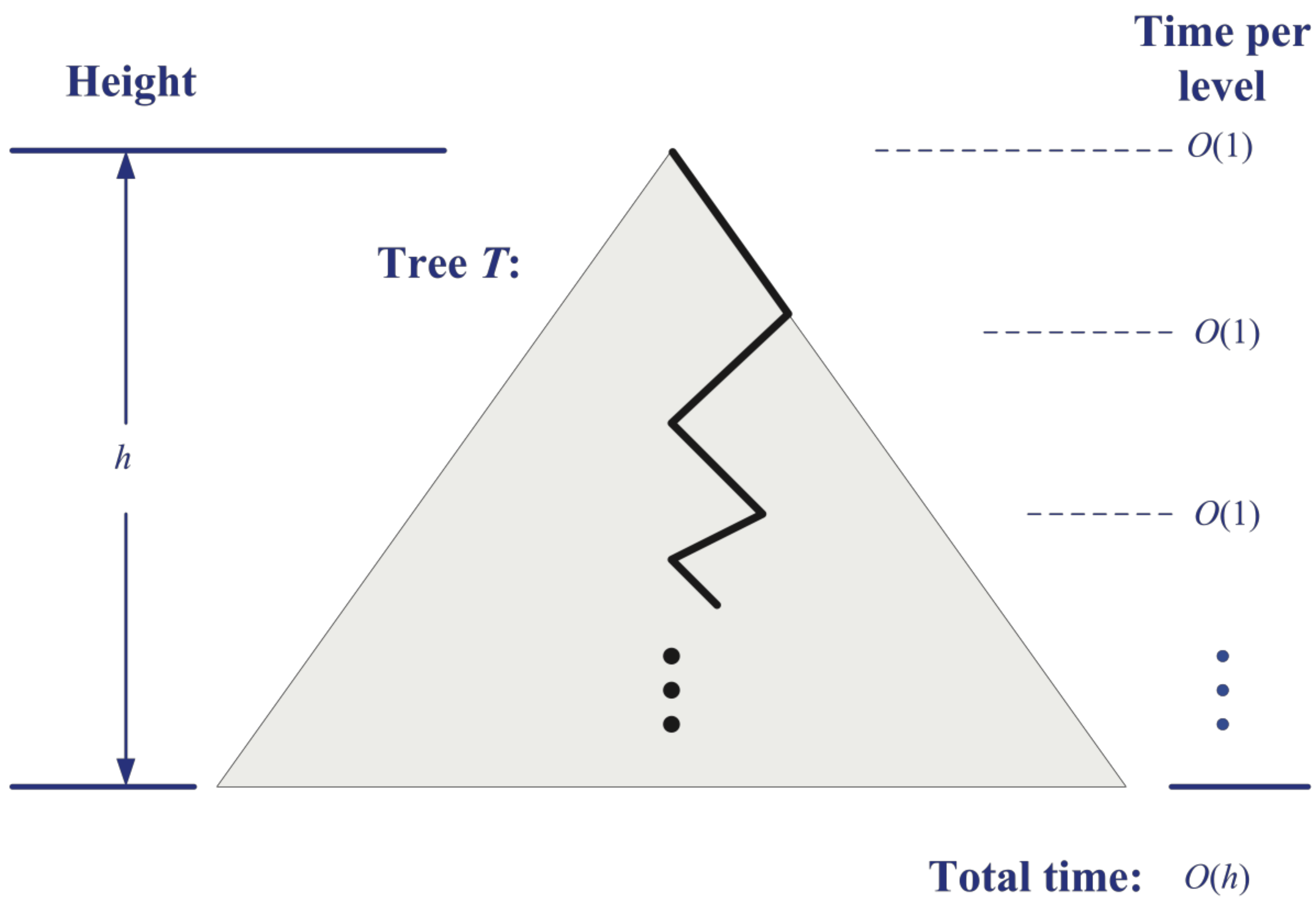
```
struct node* predecessor(struct node* node)
{
   if(node->left != NULL)
      return maximum(node->left);
   struct node* ancestor = node->parent;
   while((ancestor != NULL) && (node == ancestor->left))
   {
      node = ancestor;
      ancestor = node->parent;
   }
   return ancestor;
}
```

Complexity of Search Operations on Binary Search Trees (1)

Operations	Complexity
Search	$O(h)$
Minimum	$O(h)$
Maximum	$O(h)$
Successor	$O(h)$
Predecessor	$O(h)$
	<u>Remark:</u> h is the hight of a binary tree - At worst, h can be n-1 - At best, h can be $\log_2(n+1)-1$



Complexity of Search Operations on Binary Search Trees (2)



Update Operations on Binary Search Trees

- Update operations performed on a binary search tree *T*:
  - Tree-Insert(*k*, *T*): insert a new node object with key *k* into an appropriate position in *T*
  - Tree-Delete(z, *T*): delete an existing node object *z* from *T*

Operation: Tree-Insert (1)

- Tree-Insert(*k*, *T*): insert a new node object with key *k* into an appropriate position in *T*

k= 13

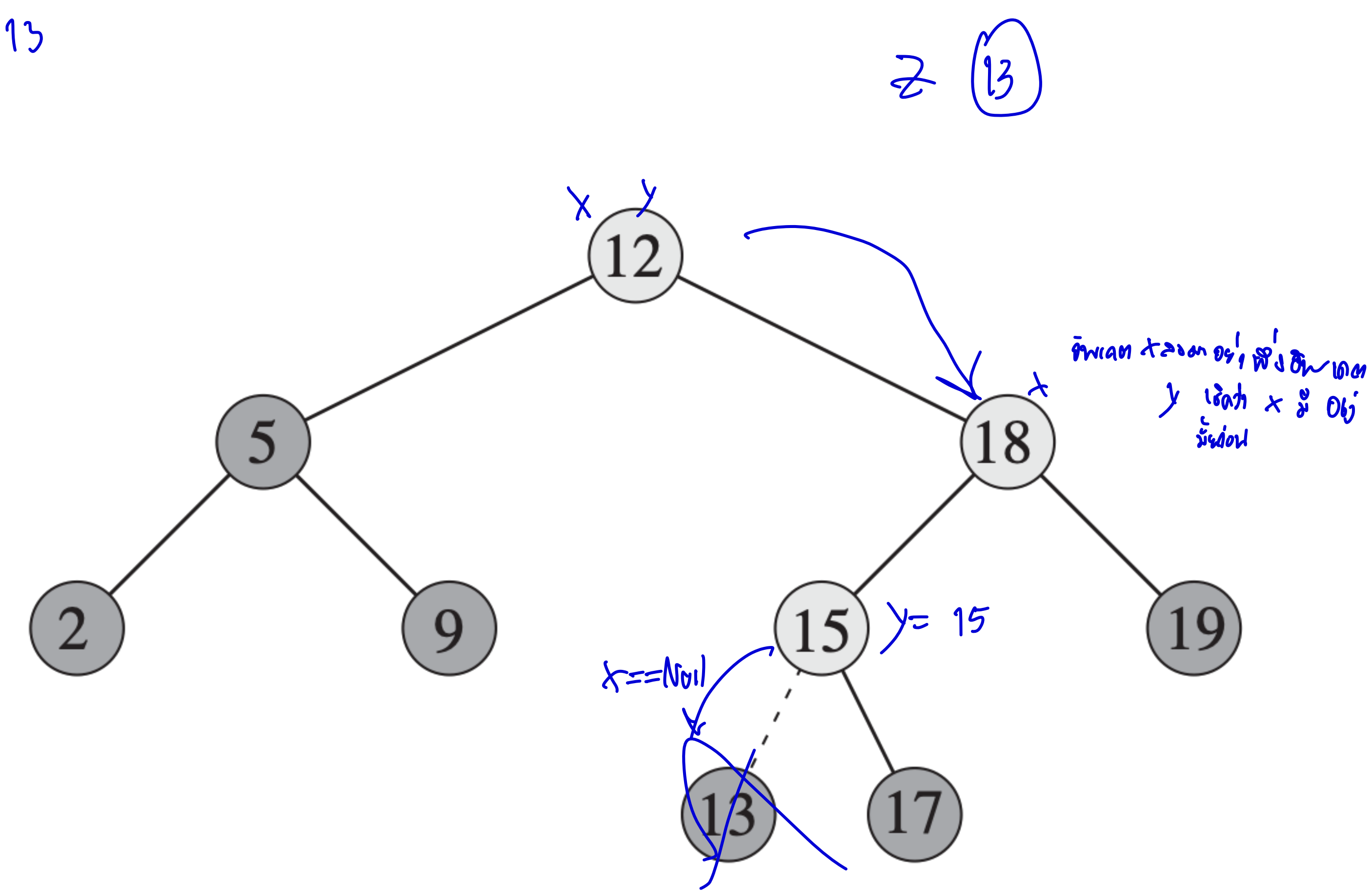
```
Tree-Insert(k, T):
        z = createNode(k)
        y = NULL
        x = T.root
        while (x != NULL):
                y = x
                if z.key < x.key:
                        x = x.left
                else:
                        x = x.right
        z.parent = y
        if y != NULL:
                if z.key < y.key:
                        y.left = z
                else:
                        y.right = z
```

trailing pointer → y = NULL → keep future parent

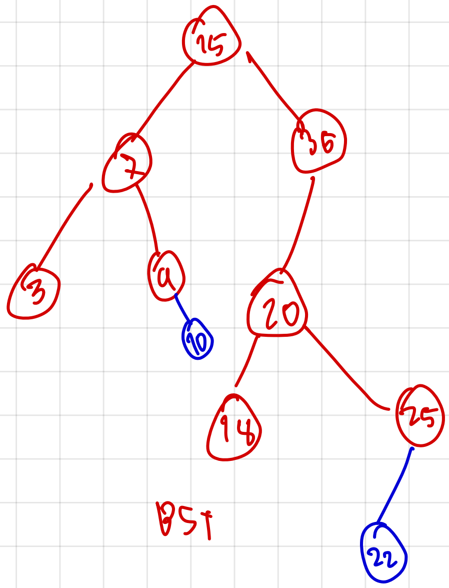
main pointer → x = T.root → เริ่มจาก root ของ T มาเทียบ

z.parent = y → set parent

if y != NULL: หาว่าต้องใส่เป็นลูกทางซ้ายหรือขวา



Insert(10)
Insert(22)



BST

Operation: Tree-Insert (2)

- After the new node z is created, we begin at the root of the tree. The pointer *x* traces a simple path downward looking for a NULL to replace it with *z*
  - We maintain the trailing pointer *y* as the parent of *x*
  - After initialization, the while loop causes the pointers x, y moving down along the tree, going left or right depending on the comparison of *z.key* with *x.key*, until *x* becomes NULL.
  - This NULL occupies the position where we wish to place the new node *z*
  - We need the trailing pointer *y*, because by the time we find the NULL, the search has proceeded one step beyond the position that needs to be replaced

```
Tree-Insert(k, T):
    z = createNode(k)
    y = NULL
    x = T.root
    while (x != NULL):
        y = x
        if z.key < x.key:
            x = x.left
        else:
            x = x.right
    z.parent = y
    if y != NULL:
        if z.key < y.key:
            y.left = z
        else:
            y.right = z
```

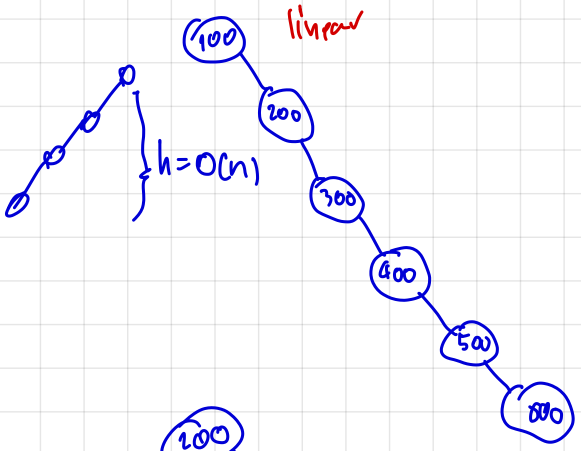
Operation: Tree-Insert (3)

```
Tree-Insert(k, T):
    z = createNode(k)
    y = NULL
    x = T.root
    while (x != NULL):
        y = x
        if z.key < x.key:
            x = x.left
        else:
            x = x.right
    z.parent = y
    if y != NULL:
        if z.key < y.key:
            y.left = z
        else:
            y.right = z
```

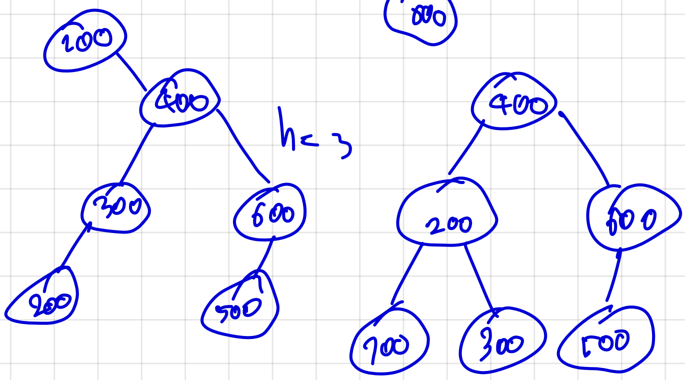
```c
struct node* treeInsert(int key, struct node* root)
{
    struct node* new_node = createNode(key);
    struct node* trail_pt = NULL;
    struct node* node = root;

    while(node!=NULL) {
        trail_pt = node;
        if(new_node->key <= node->key)
            node = node->left;
        else
            node = node->right;
    }
    new_node->parent = trail_pt;
    if(trail_pt != NULL){
         if(new_node->key <= trail_pt->key)
            trail_pt->left = new_node;
        else
            trail_pt->right = new_node;
    }
    return new_node;
}
```

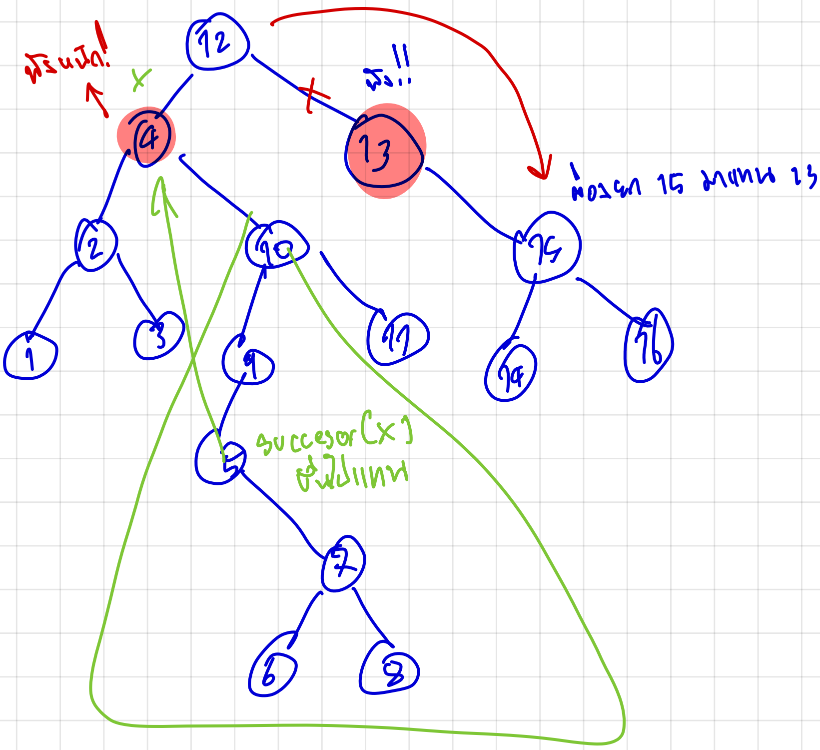
Insert 200, 300, 400, 500, 600

4 2 1 5 3



Random BST





Quiz N :

กิตติคุณ บุญต่อยุทธ 6510405334

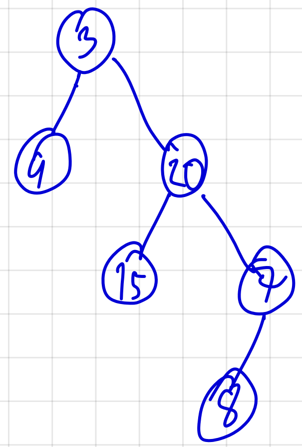
1. จงวาดรูปแสดงโครงสร้างของ BT ที่เป็นไปได้จากผลลัพธ์การ traverse ต่อไปนี้

Pre-order 3, 9, 20, 15, 7, 8 Post-order 9, 15, 8, 7, 20, 3

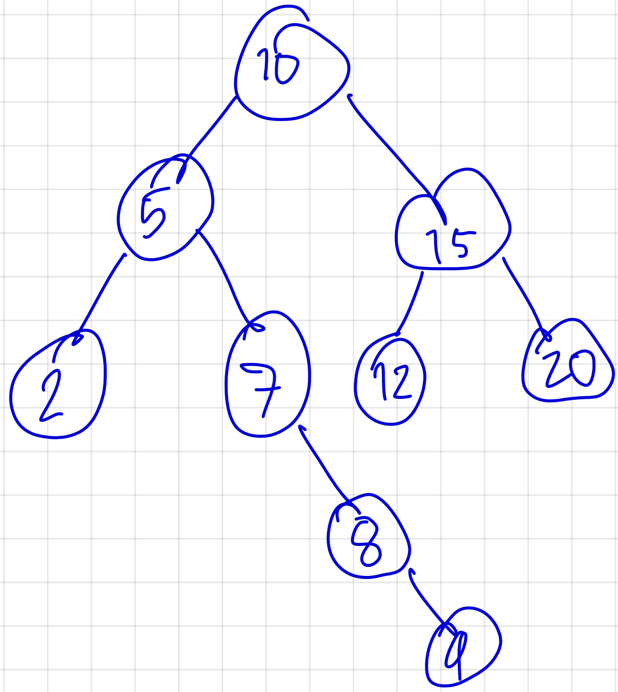
2. จงวาดรูปแสดงโครงสร้างของ BST ที่เป็นไปได้จากผลลัพธ์การ traverse ต่อไปนี้

Pre-order : 10, 5, 2, 7, 8, 9, 15, 12, 20

1.



2.



Operation: Tree-Delete (1)

- Tree-Delete(*z*, *T*): delete an existing node object *z* from *T*.
  - The strategy for deleting z is divided into three basic cases:
    - **<u>Case A.1</u>**: If *z* has no children, then we simply remove *z* by placing NULL in the position of z
    - **<u>Case A.2</u>**: If *z* has just one child, then we elevate that child to take the position of *z.*
    - **<u>Case A.3</u>**: If *z* has two children, we first find *z*'s successor, called *y* (*y* must be the leftmost ~~ancestor~~ descendant in *z*'s right subtree) and let *y* take *z*'s position. Then, 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.

Operation: Tree-Delete (2)

- In actual implementation, we organize the cases a bit differently from what outlined previously:
  - **Case B.1**: If *z* has no left child, then we replace *z* by its right child, which may or may not be NULL
    - If *z*'s right child is NULL, this case deals with the situation in which z has no children at all
    - If *z*'s right child is not NULL, this case handles the situation in which *z* has just right child
  - **Case B.2**: If *z* has just one child, which is its left child, then we replace *z* by its left child
  - **Case B.3**: If *z* has both left and right children, we find the successor *y of z.* Then, we splice *y* out of its current location and let y take the position of z
    - **Case B.3.1**: If *y* is *z*'s right child, then we replace *z* with *y*
    - **Case B.3.2**: If *y* is not *z*'s right child, we first replace *y* with its own right child. Then, we replace *z* with *y*

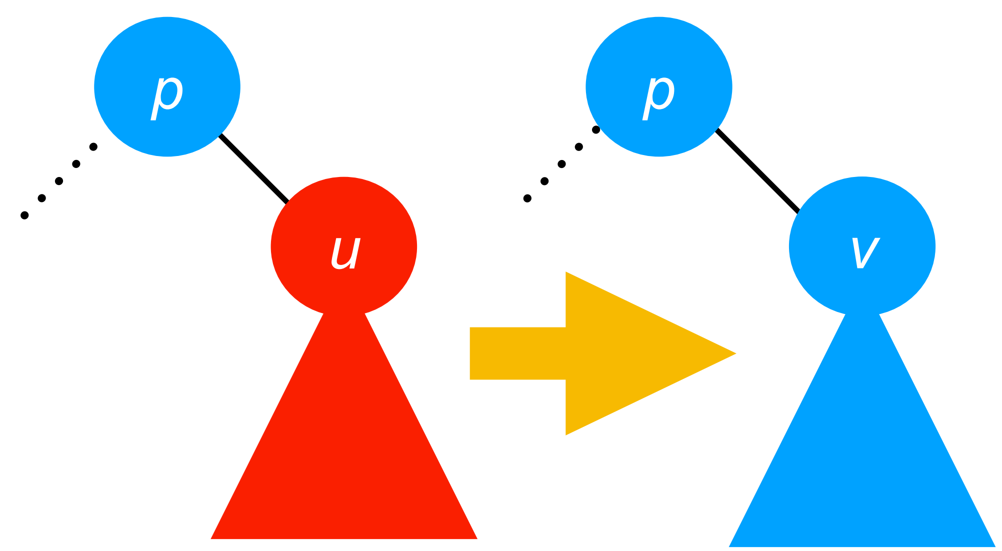
Operation: Tree-Delete (3)

- To be able to move subtrees around within a binary search tree, we define a subroutine Transplant which can be used to replaces one subtree with another subtree

ปลูกถ่าย

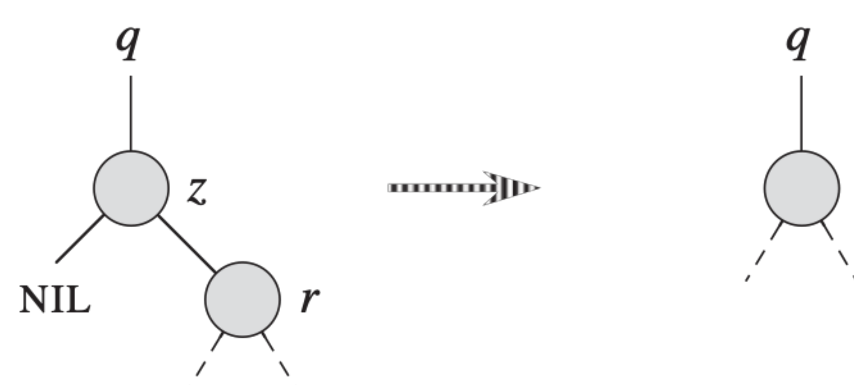
- transplant(*u, v, T*): replace the subtree rooted at node *u* with the subtree rooted at node *v*. So, node *u*'s parent becomes node *v*'s parent, and *u*'s parent ends up having *v* as its appropriate child

```
transplant(u, v, T):
        p = u.parent
        if (p != NULL):
                if u == p.left:
                        p.left = v
                else:
                        p.right = v
        if (v != NULL):
                v.parent = p
```



Operation: Tree-Delete (4)

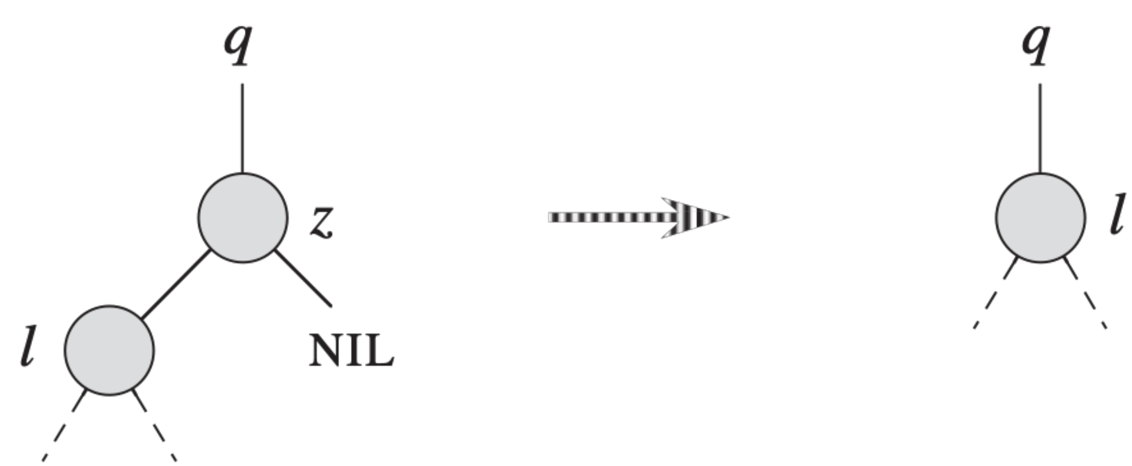
- **Case B.1**: If *z* has no left child, then we replace *z* by its right child, which may or may not be NULL
  - If *z*'s right child is NULL, this case deals with the situation in which z has no children at all
  - If *z*'s right child is not NULL, this case handles the situation in which *z* has just right child




```
//Case B.1
if z.left == NULL:
        transplant(z, z.right, T)
```

Operation: Tree-Delete (5)

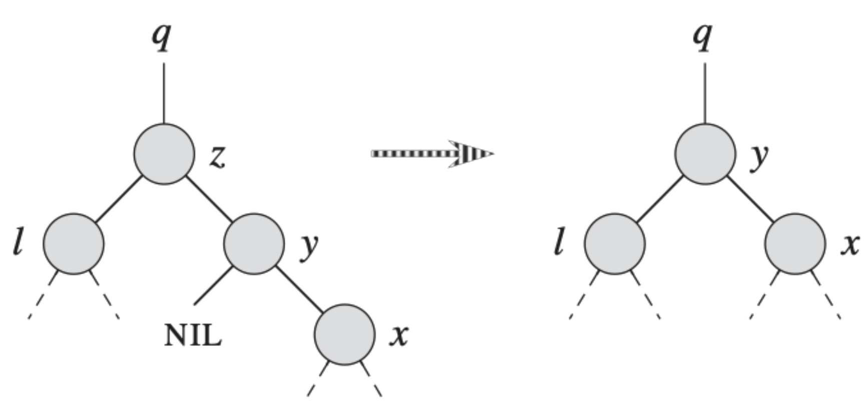
- **Case B.2**: If *z* has just one child, which is its left child, then we replace *z* by its left child



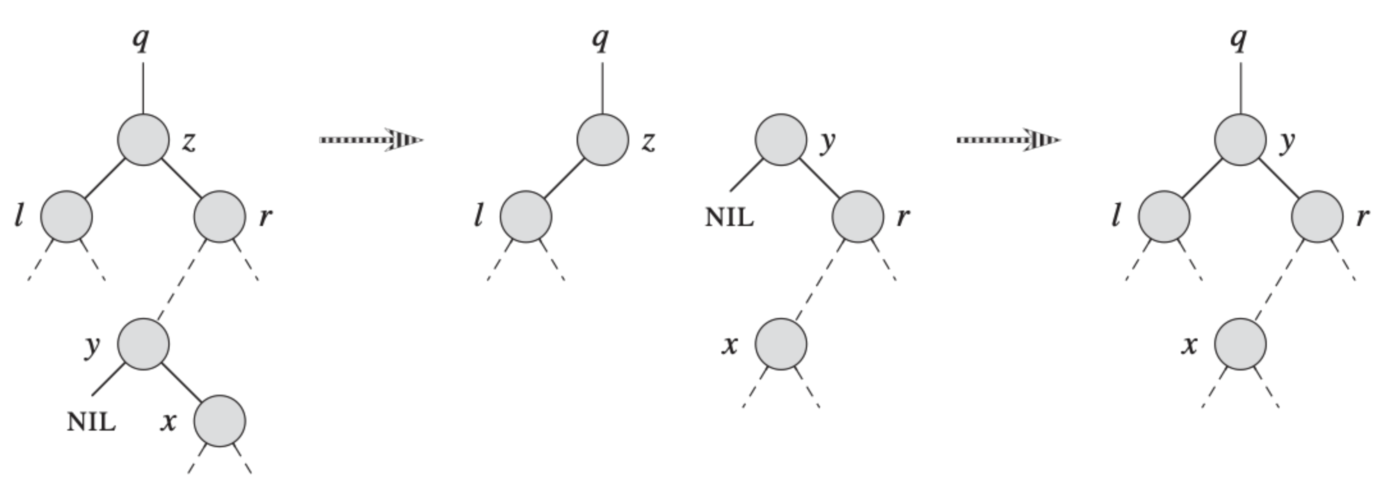

```
//Case B.2
if z.right == NULL:
        transplant(z, z.left, T)
```

Operation: Tree-Delete (6)

- **Case B.3**: If *z* has both left and right children, we find the successor *y of z.* Then, we splice *y* out of its current location and let y take the position of z
  - **Case B.3.1**: If *y* is *z*'s right child, then we replace *z* with *y*



  - **Case B.3.2**: If y is not z's right child, we first replace y with its own right child. Then, we replace z with y




```
//Case B.3
if z.left != NULL and z.right != NULL:
    y = minimum(z.right)
    if(y.parent == z):          //Case B.3.1
        transplant(z, y, T)
        transplant(y.left, l=z.left, T)
    else:                       //Case B.3.2
        transplant(y, x=y.right, T)
        transplant(y.right, r=z.right , T)
        transplant(z, y, T)
        transplant(y.left, l=z.left, T)
```

Operation: Tree-Delete (7)

- Tree-Delete(z, *T*): delete an existing node object *z* from *T*

```
Tree-Delete(z, T):
    //Case B.1
    if z.left == NULL:
        transplant(z, z.right, T)
    //Case B.2
    if z.right == NULL:
        transplant(z, z.left, T)
    //Case B.3
    if z.left != NULL and z.right != NULL:
        y = minimum(z.right)
        if(y.parent == z):          //Case B.3.1
            transplant(z, y, T)
            transplant(y.left, l=z.left, T)
        else:                       //Case B.3.2
            transplant(y, x=y.right, T)
            transplant(y.right, r=z.right , T)
            transplant(z, y, T)
            transplant(y.left, l=z.left, T)
```

Operation: Tree-Delete (8)

```
struct node* treeDelete(struct node* node)
{
    if(node->left==NULL)    //Case B.1
        transplant(node, node->right);
    if(node->right==NULL)   //Case B.2
        transplant(node, node->left);
    if((node->left!= NULL)&&(node->right!=NULL)) //Case B.3
    {
        struct node* y = minimum(node->right);
        if(y->parent == node) {         // Case B.3.1
            transplant(node, y);
            transplant(y, node->left);
        } else {                        // Case B.3.1
            transplant(y, y->right);
            transplant(y->right, node->right);
            transplant(node, y);
            transplant(y, node->left);
        }
    }
}
```

```
void transplant(struct node* u,
struct node* v)
{
    struct node* p = u->parent;
    if(p!=NULL) {
        if(u == p->left)
            p->left = v;
        else
            p->right = v;
    }
    if(v!=NULL)
        v->parent = p;
}
```

Complexity of Operations on Binary Search Trees

Operations	Complexity
Search	$O(h)$
Minimum	$O(h)$
Maximum	$O(h)$
Successor	$O(h)$
Predecessor	$O(h)$
Tree-Insert	$O(h)$
Tree-Delete	$O(h)$
	Remark: h is the hight of a binary tree. - At worst, h can be n-1. - At best, h can be $\log_2(n+1)-1$.