


Binary Search Tree



The logo of Politecnico di Torino is circular, featuring a central figure holding a torch and a book, surrounded by the text "POLITECNICO DI TORINO" and the years "1859" and "1906".

Introduzione

Binary Search Tree, o BST implement
dictionaries or ordered queues –
elements stored are ordered according to a
key
Efficient ($\log n$) operations SEARCH, MINIMUM,
MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, DELETE.

2



Definition

Binary Search Tree:

- **Tree**: hierarchical structure made of nodes with father-son relations. One node has no father (root) and is unique. The path from the root to any node is unique
- **Binary**: each node has at most 2 sons (*left* e *right*) and only one father (p) – except the root that has no father
- **Search**: nodes have a *key* field and are ordered according to it.

3



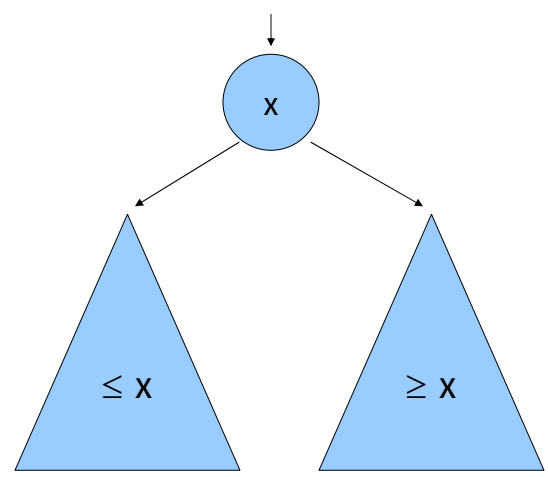
Ordering relations (I)

For each node x :

- For all nodes y in left subtree of x,
 $\text{key}[y] \leq \text{key}[x]$
- For all nodes y in right subtree of x,
 $\text{key}[y] \geq \text{key}[x]$

4

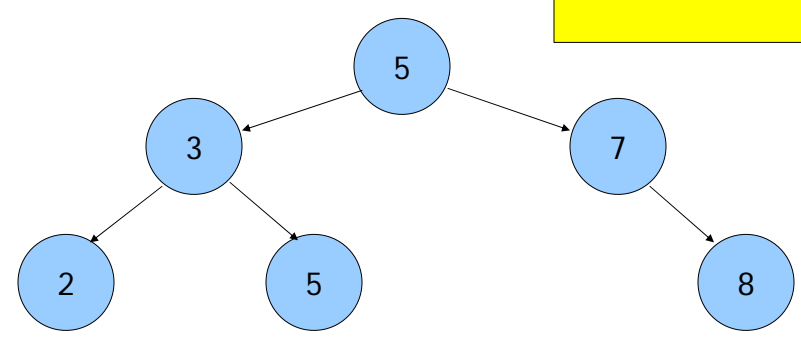
Ordering relations (II)



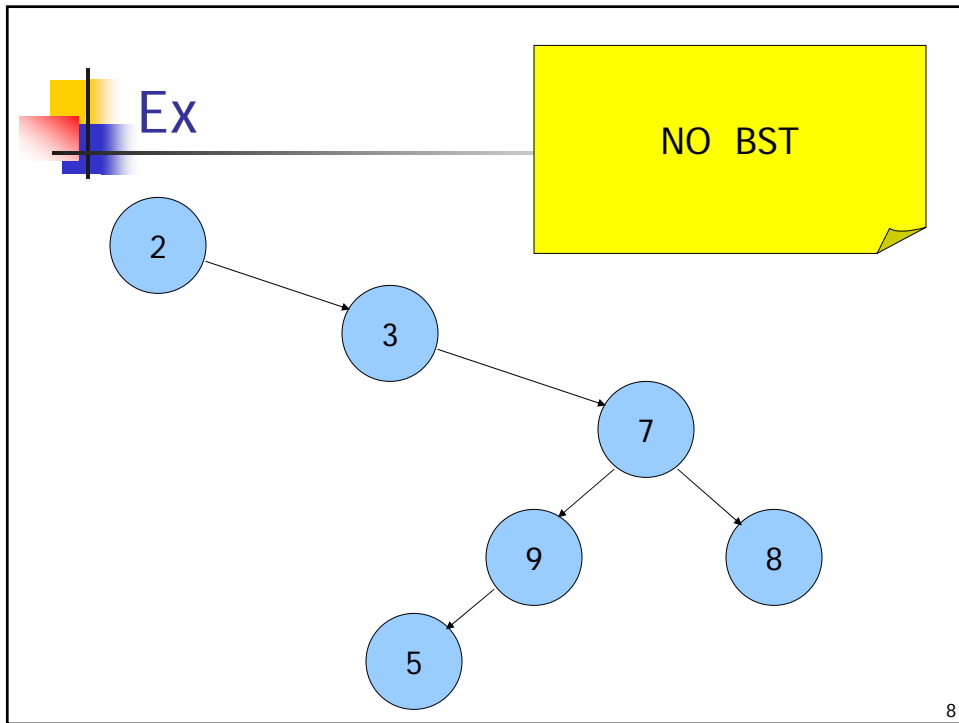
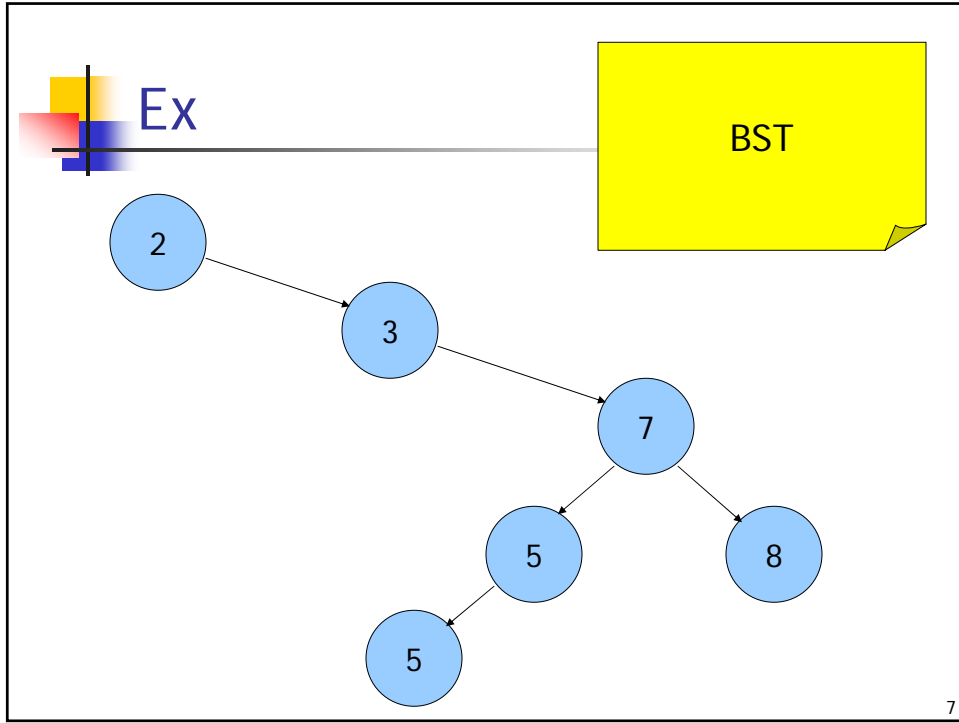
5

Ex 1

This tree is a BST



6

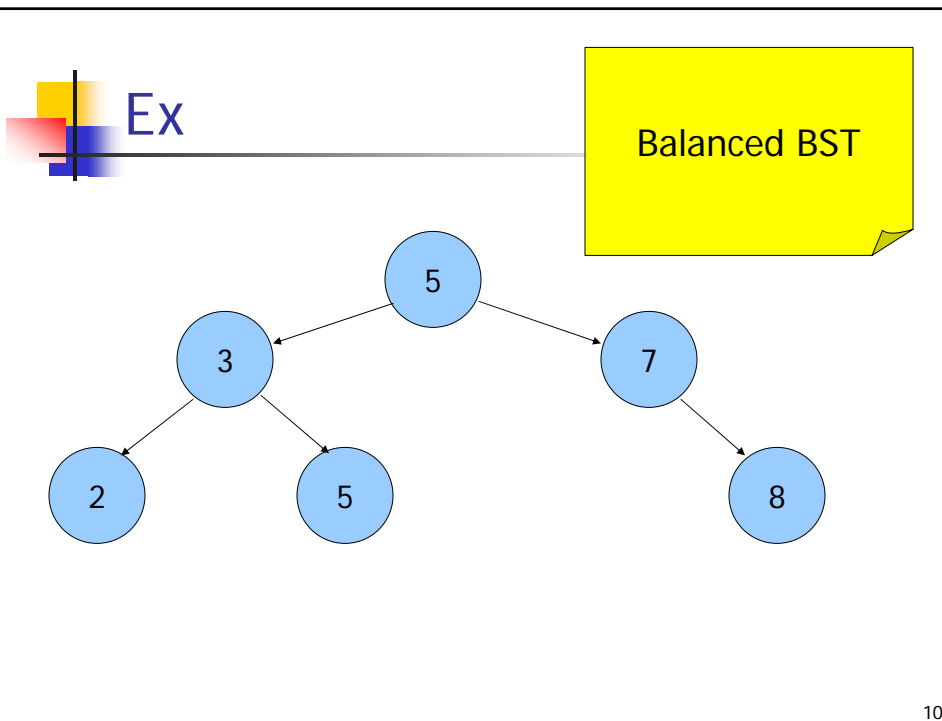




Balanced BST

A BST is balanced if, for each node,
the number of nodes in its left subtree
=
the number of nodes in its right subtree ± 1

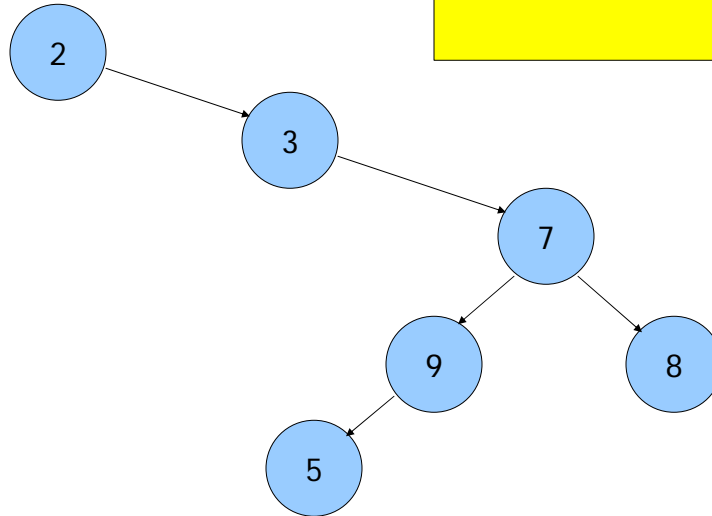
9



10

Ex II

Not Balanced BST



11

Complexity

Operations have complexity proportional to the depth h of the tree

For a balanced tree (n nodes) complexity = $\Theta(\log n)$

For a completely unbalanced tree, the worst case, complexity = $O(n)$.

Average case $\Theta(\log n)$

12



Traversals

Going through all nodes can be done in 3 ways

- **Preorder**: node, subtree left, subtree right
- **Inorder**: subtree left, node, subtree right
- **Postorder**: subtree left, subtree right, node

13



Preorder

Preorder-Tree-Walk(x)

- 1 **if** $x \neq \text{NIL}$
- 2 **then** print key[x]
- 3 Preorder-Tree-Walk(left[x])
- 4 Preorder-Tree-Walk(right[x])

14



Inorder

Inorder-Tree-Walk(x)

```
1  if x ≠ NIL
2      then Inorder-Tree-Walk(left[x])
3      print key[x]
4      Inorder-Tree-Walk(right[x])
```

15



Postorder

Postorder-Tree-Walk(x)

```
1  if x ≠ NIL
2      then Postorder-Tree-Walk(left[x])
3      Postorder-Tree-Walk(right[x])
4      print key[x]
```

16



Notes

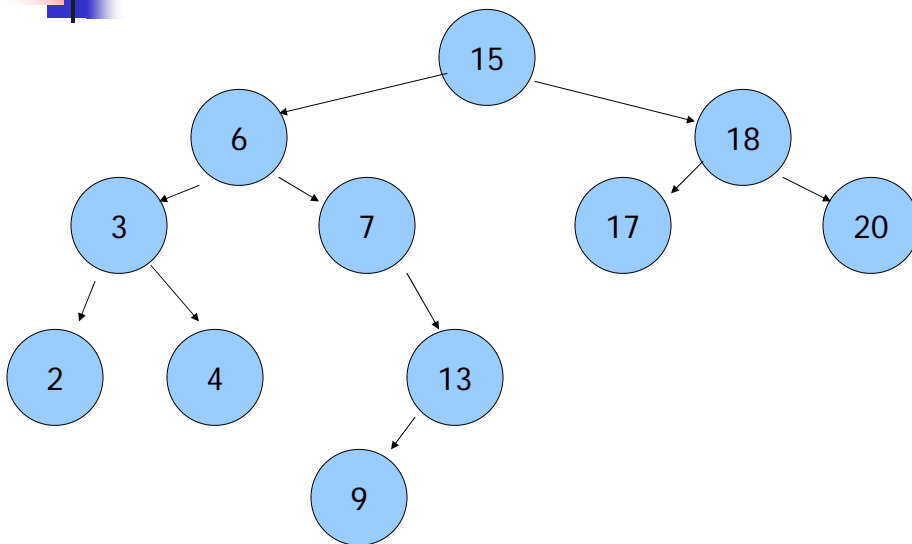
Inorder traversal (see Inorder-Tree-Walk($\text{root}[T]$)) returns the elements in growing order of key

All traversal are $\Theta(n)$ (all nodes are passed once)

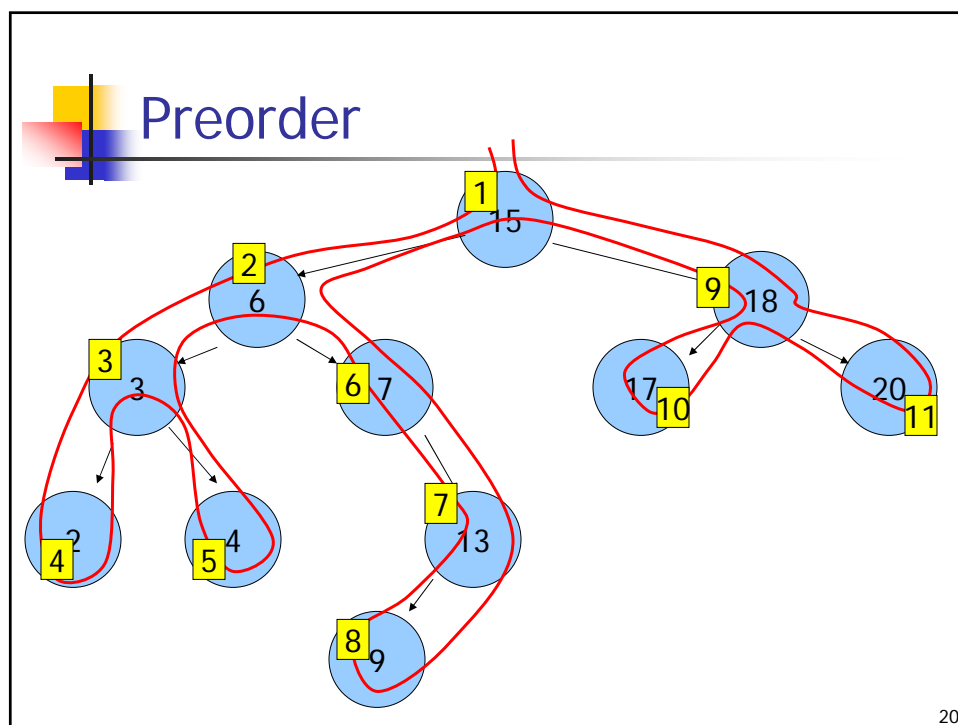
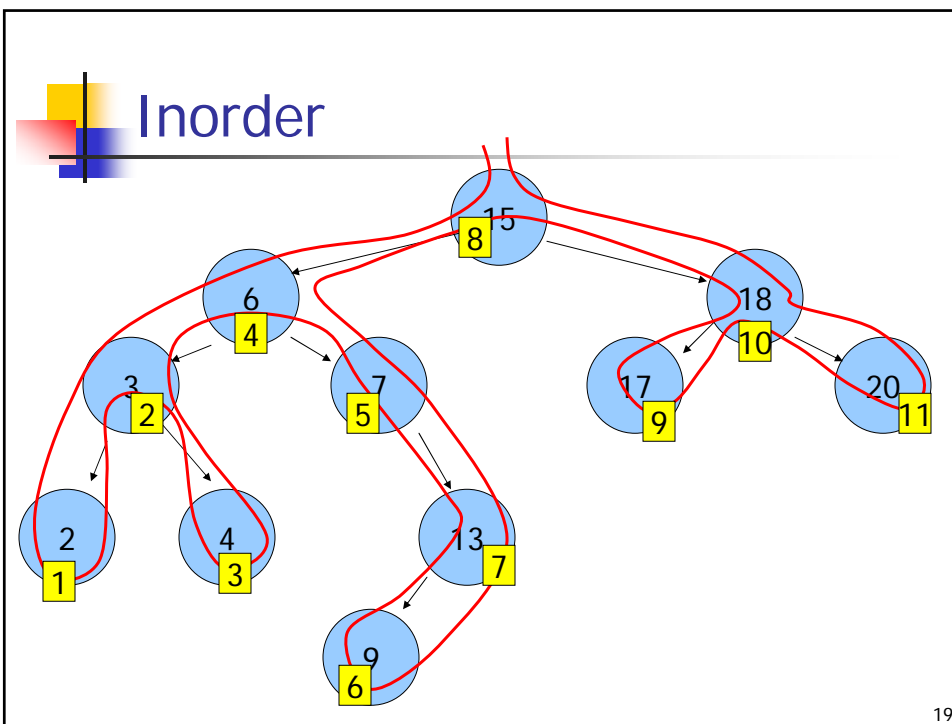
17

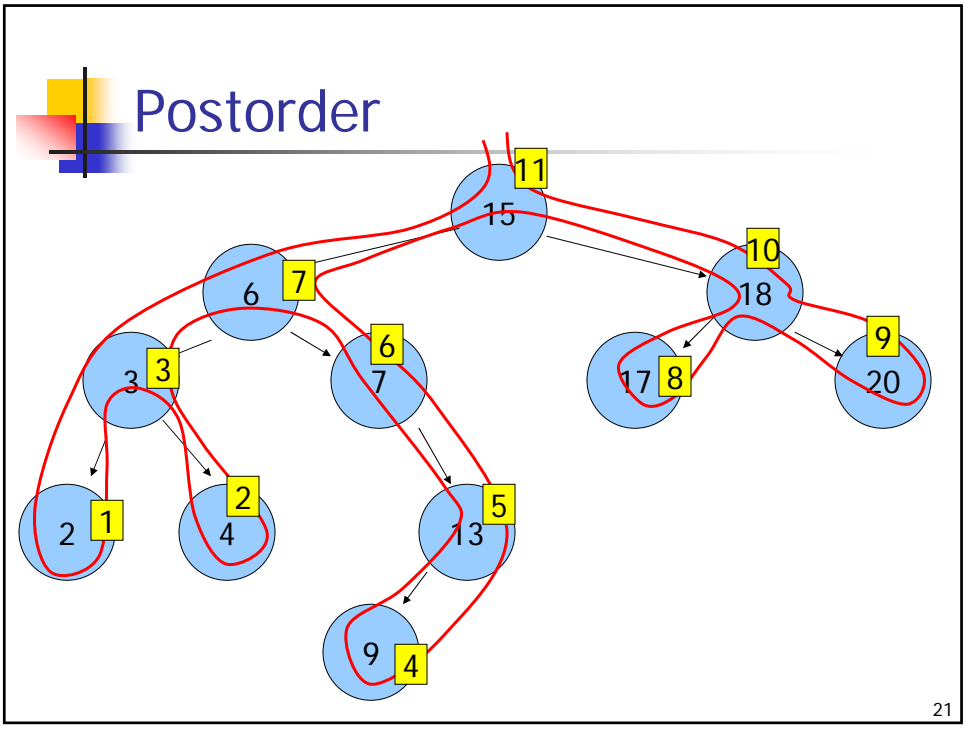


Ex. Traversals?



18





Search operations

Search, Minimum/Maximum,
Predecessor/Successor.
Are all $O(h)$

22

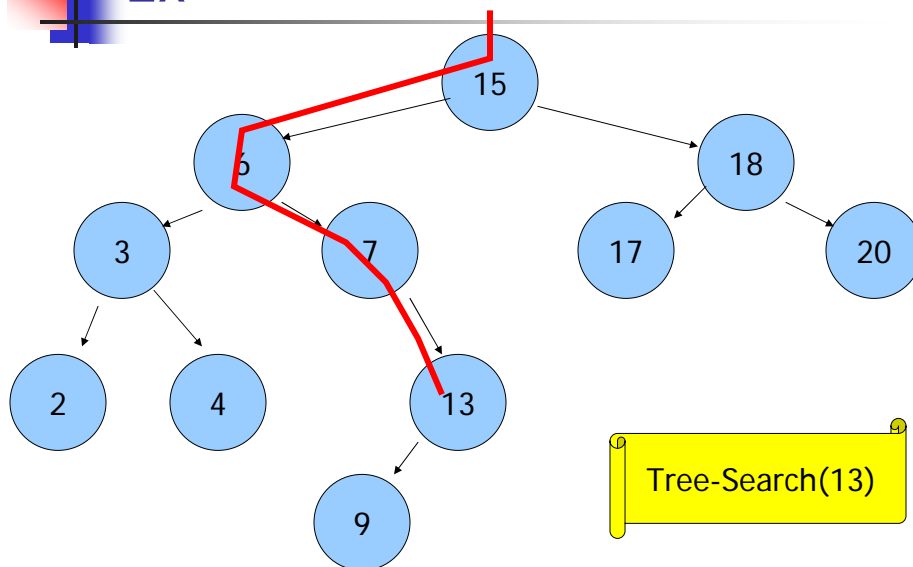
Tree-Search

Tree-Search(x, k)

```
1  if  $x = \text{NIL}$  or  $k = \text{key}[x]$ 
2      then return  $x$ 
3  if  $k < \text{key}[x]$ 
4      then return Tree-Search(left[ $x$ ],  $k$ )
5      else return Tree-Search(right[ $x$ ],  $k$ )
```

23

Ex



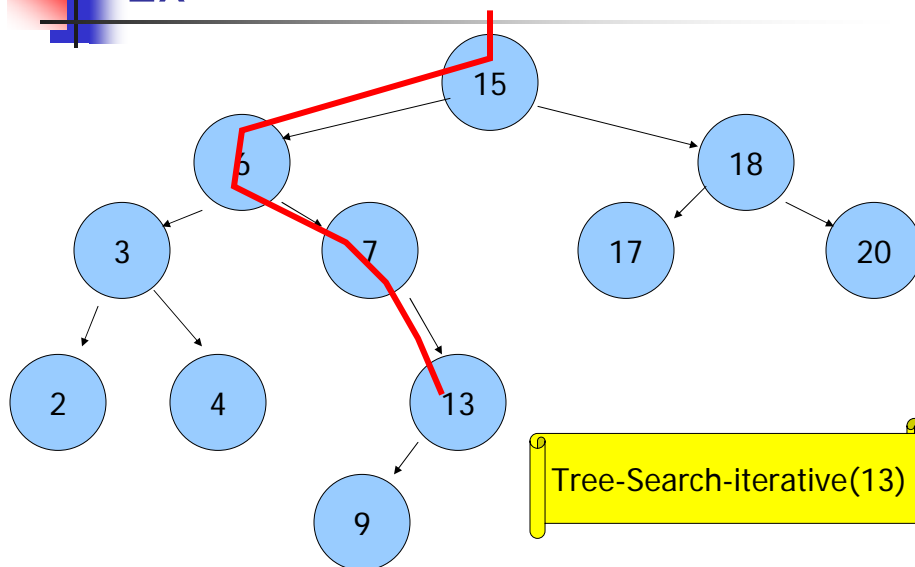
24

Tree-Search (iterative)

```
Tree-Search-iterative(x, k)
1   while x ≠ NIL and k ≠ key[x]
2       do if k < key[x]
3           then x ← left[x]
4           else x ← right[x]
5   return x
```

25

Ex



26

Min / Max (iterative)

Tree-Minimum(x)

```

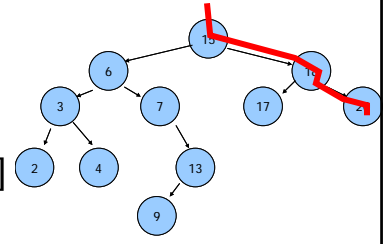
1  while left[x] ≠ NIL
2      do x ← left[x]
3  return x
    
```



Tree-Maximum(x)

```

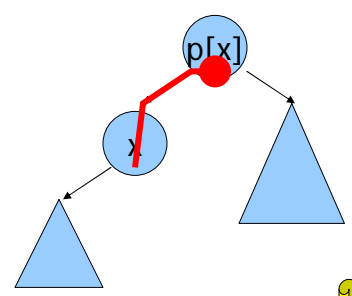
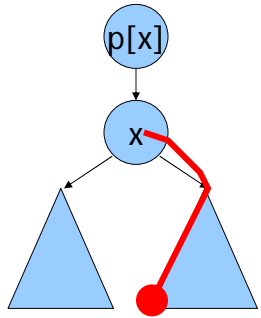
1  while right[x] ≠ NIL
2      do x ← right[x]
3  return x
    
```



Successor

Given a node, find the closest - 2 cases:

Min of right subtree



First ancestor, of which the node is left heir

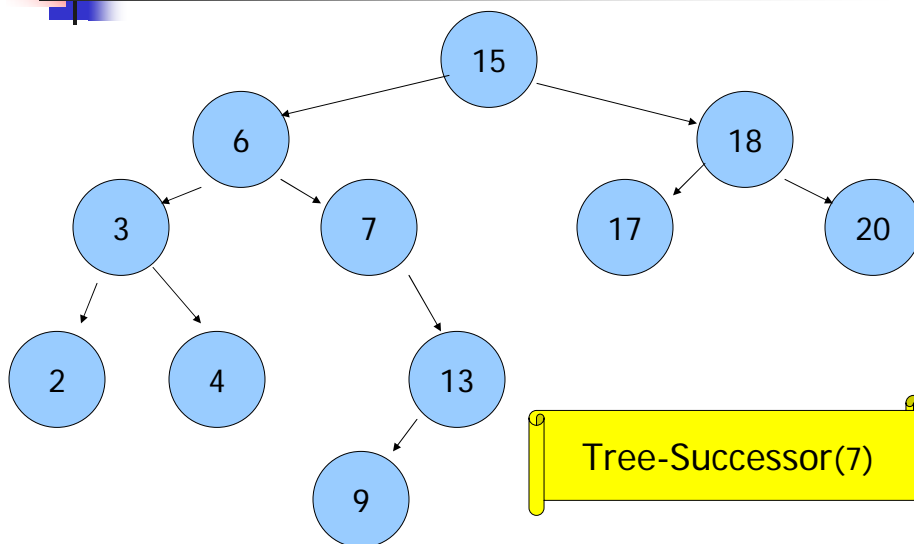
Successor

Tree-Successor(x)

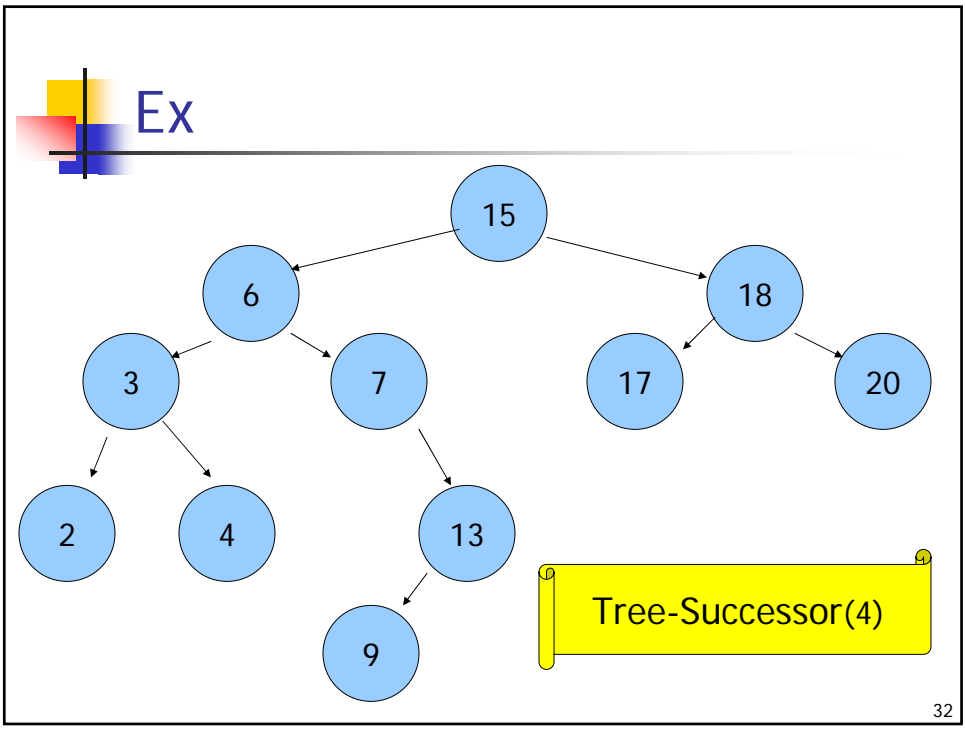
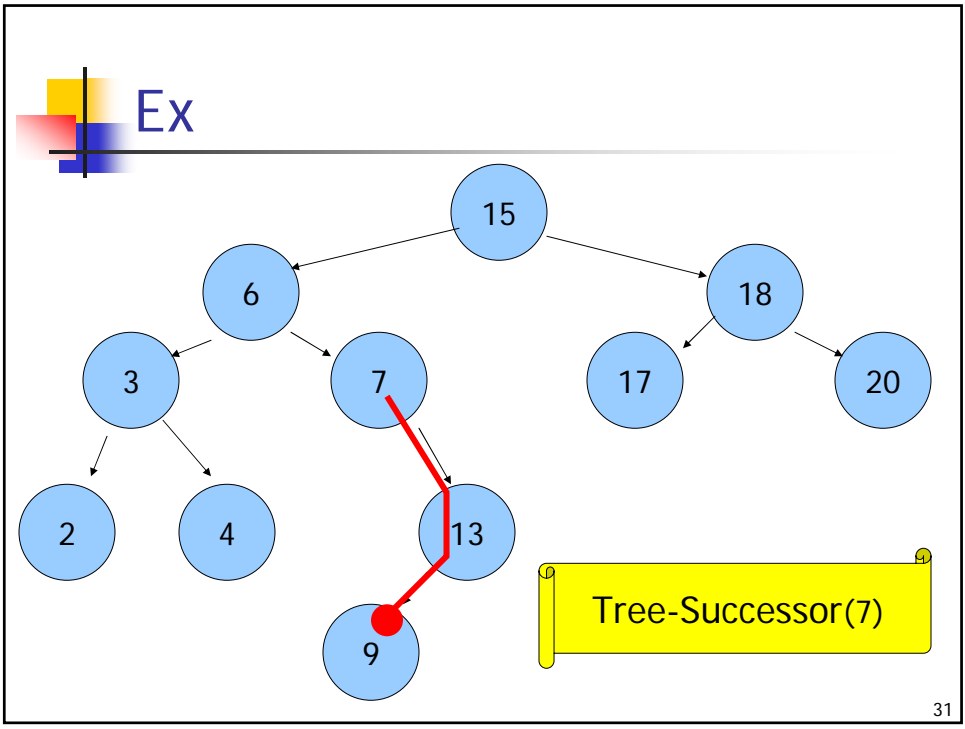
```
1  if right[x] ≠ NIL
2      then return Tree-Minimum(right[x])
3  y ← p[x]
4  while y ≠ NIL and x = right[y]
5      do x ← y
6      y ← p[y]
7  return y
```

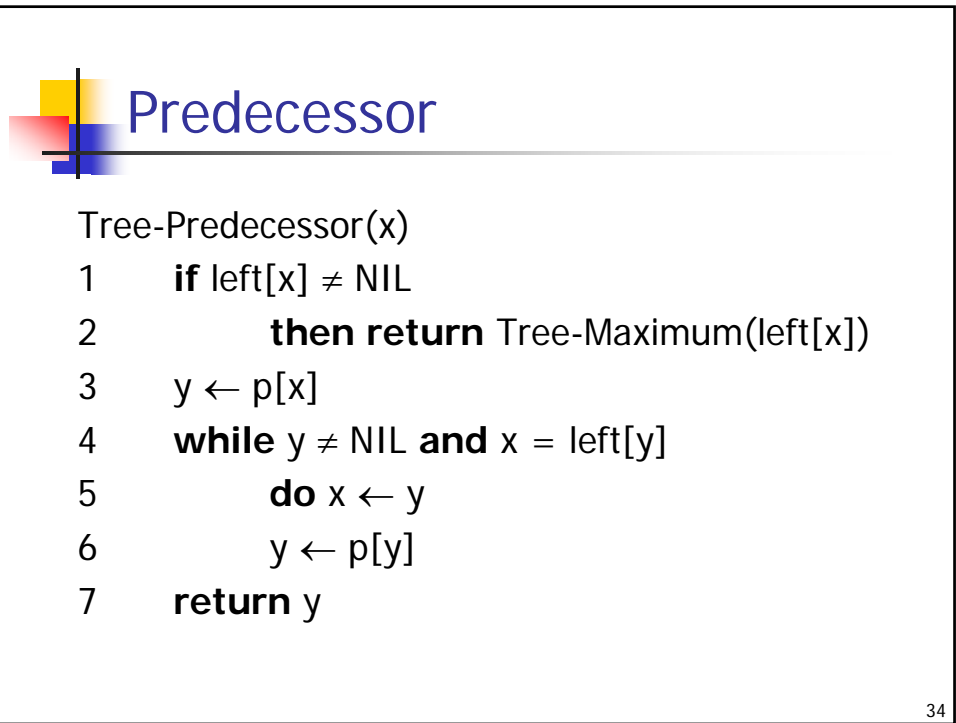
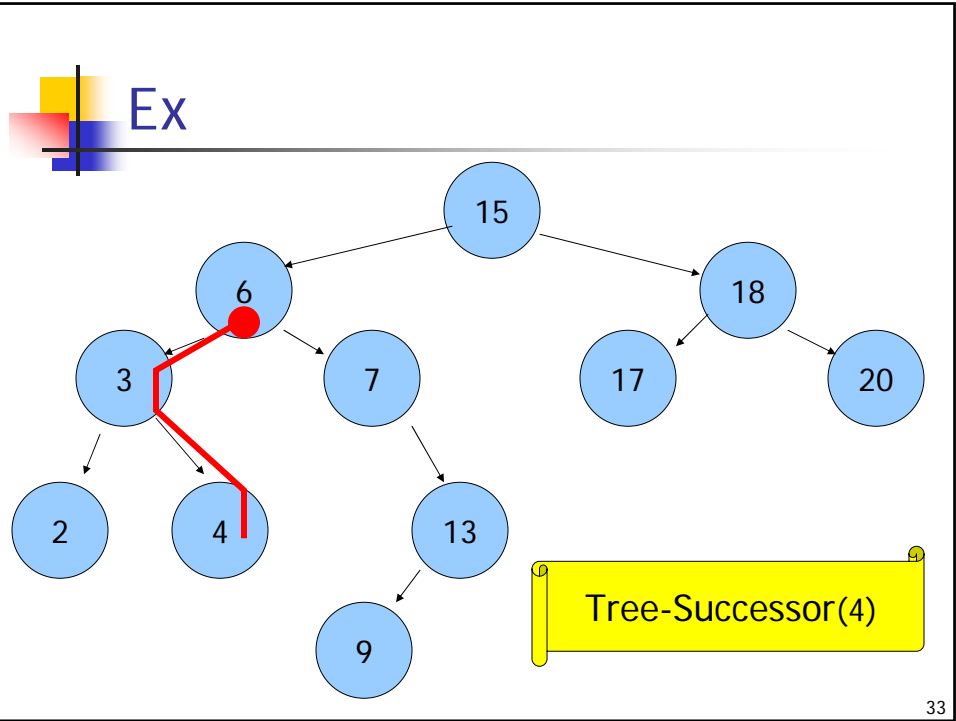
29

Ex



30







Insert Delete

Must maintain the BST properties.

35



Insert

To insert node z with key v :

- Create node z with $\text{left}[z]=\text{right}[z]=\text{NIL}$
- Search $\text{key}[z]$
- Set pointers

The new node is always a leaf

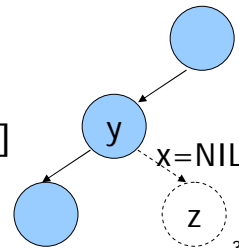
36

Tree-Insert (I)

Tree-Insert(T, z)

```
1   $y \leftarrow \text{NIL}$ 
2   $x \leftarrow \text{root}[T]$ 
3  while  $x \neq \text{NIL}$ 
4      do  $y \leftarrow x$ 
5      if  $\text{key}[z] < \text{key}[x]$ 
6          then  $x \leftarrow \text{left}[x]$ 
7          else  $x \leftarrow \text{right}[x]$ 
```

search key[z]

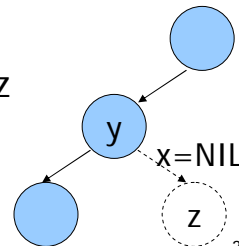


37

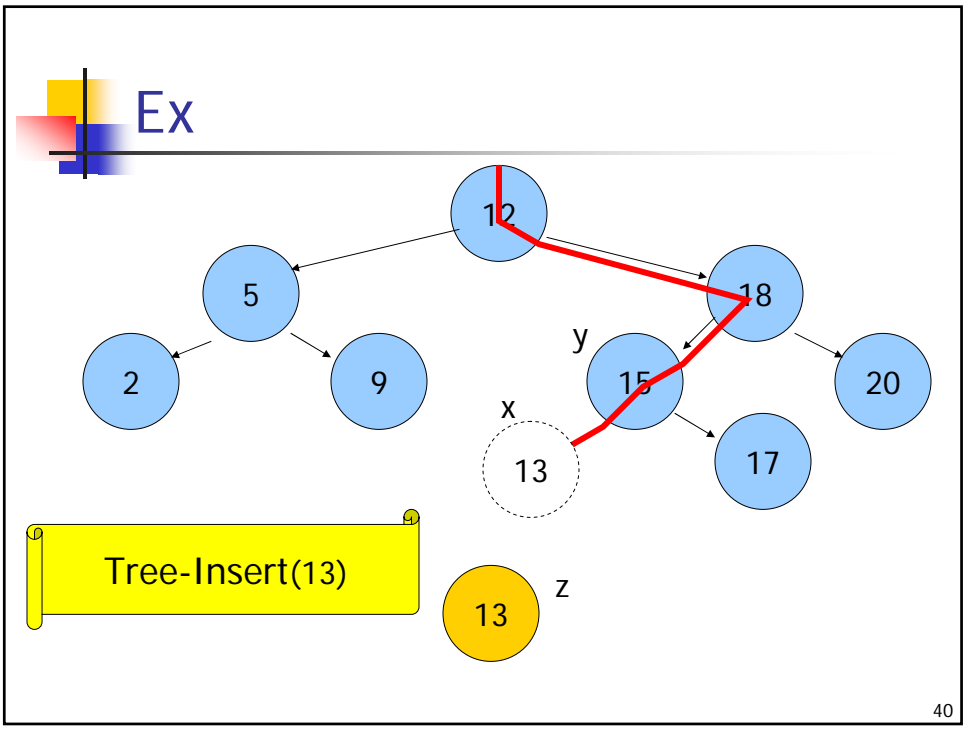
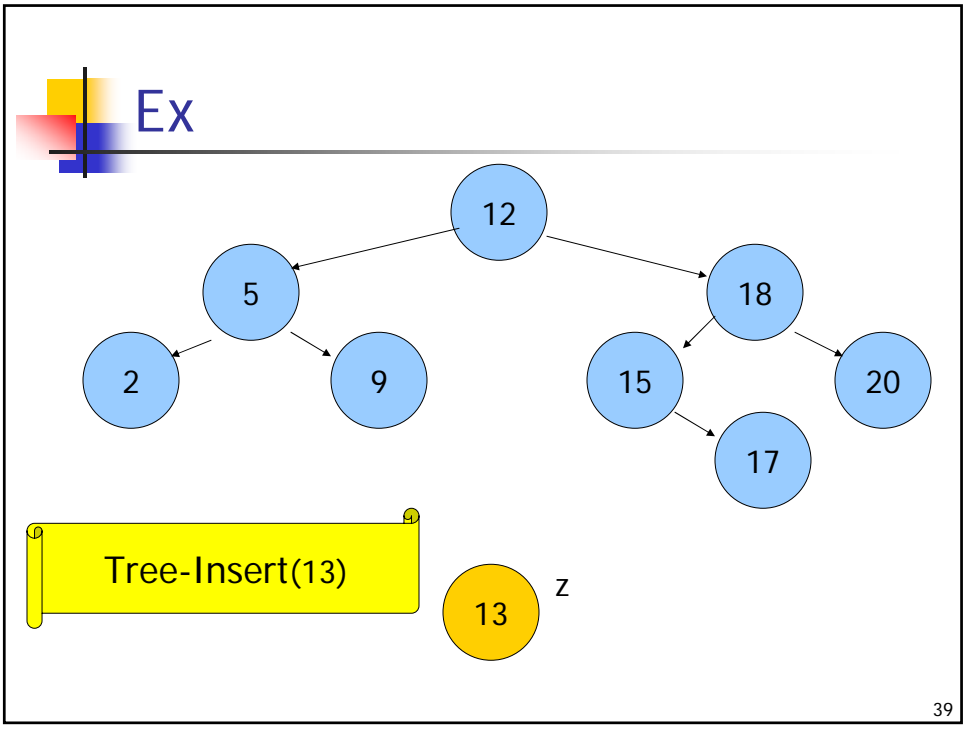
Tree-Insert (II)

```
8   $p[z] \leftarrow y$ 
9  if  $y = \text{NIL}$ 
10     then  $\text{root}[T] \leftarrow z$ 
11     else if  $\text{key}[z] < \text{key}[y]$ 
12         then  $\text{left}[y] \leftarrow z$ 
13         else  $\text{right}[y] \leftarrow z$ 
```

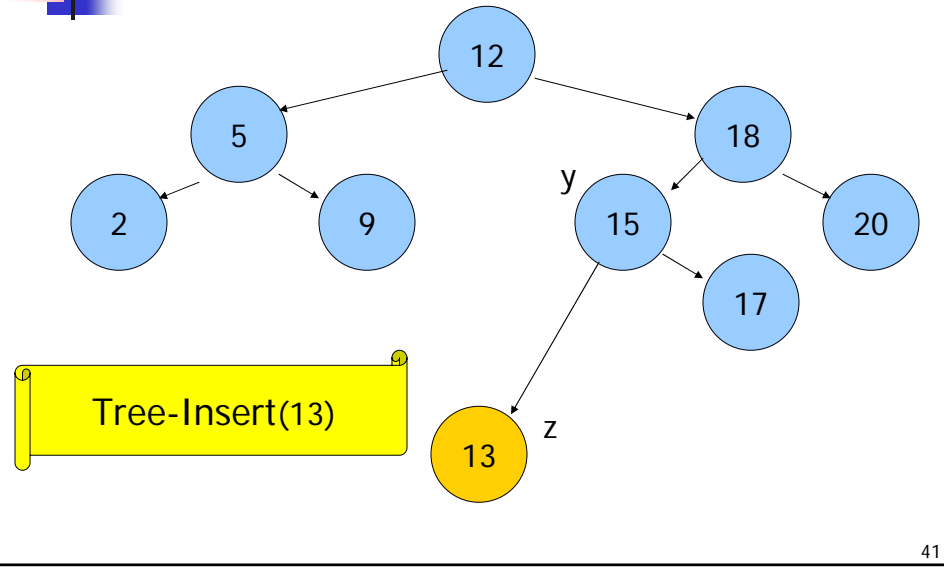
insert z as son
of y



38

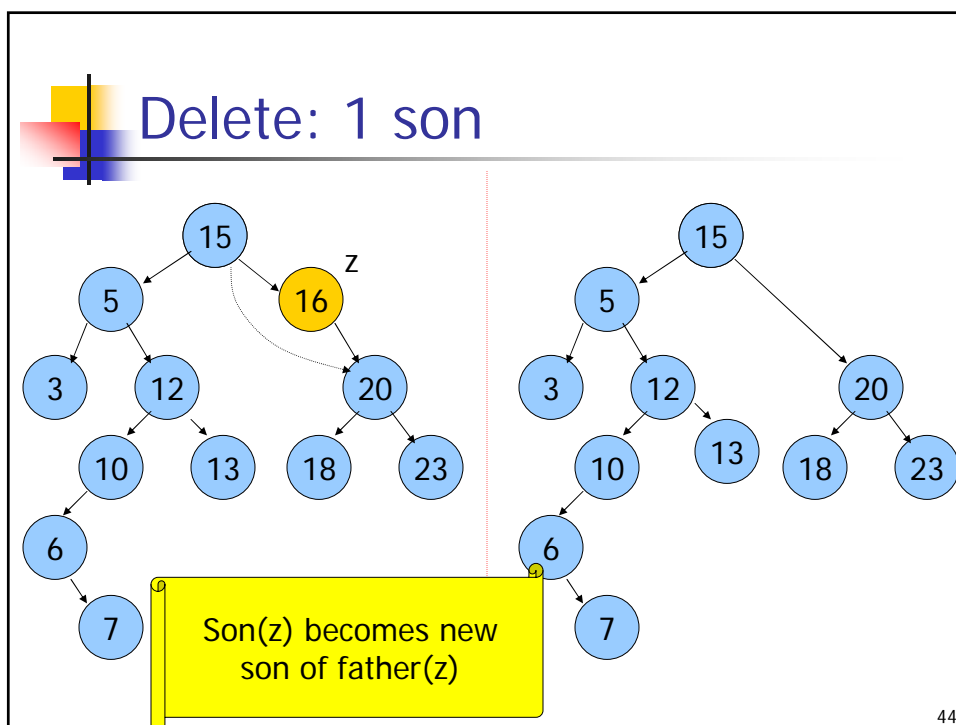
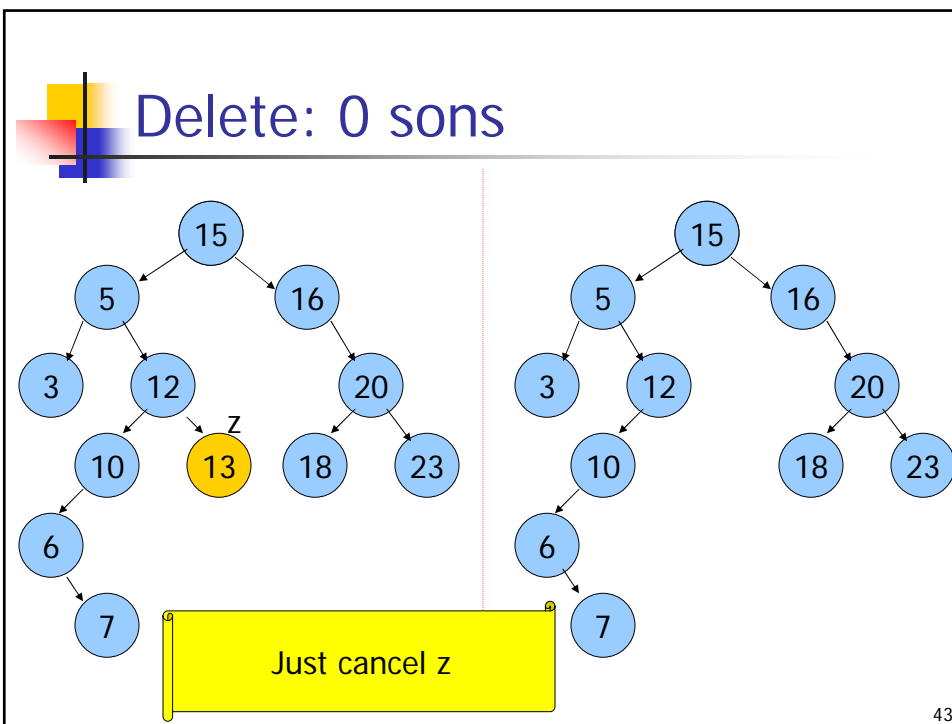


Ex

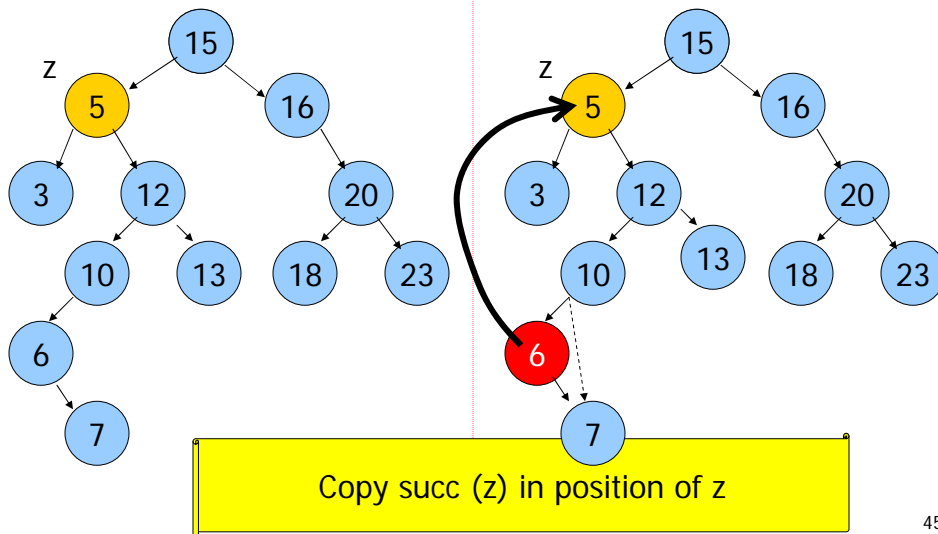


Delete

Three cases, zero, 1, 2 sons

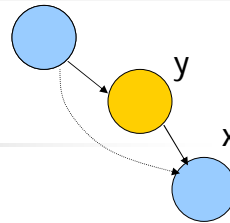


Delete: 2 sons



45

Tree-Delete (I)



Tree-Delete(T, z)

1 **if** left[z]=NIL or right[z]=NIL

2 **then** $y \leftarrow z$

3 **else** $y \leftarrow \text{Tree-Successor}(z)$

4 **if** left[y] \neq NIL

y : to be canceled

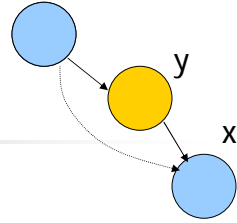
5 **then** $x \leftarrow \text{left}[y]$

6 **else** $x \leftarrow \text{right}[y]$

x : unique son of y

46

Tree-Delete (II)



```
7   if x ≠ NIL
8       then p[x] ← p[y]
9   if p[y] = NIL
10      then root[T] = x
11      else if y = left[p[y]]
12              then left[p[y]] ← x
13              else right[p[y]] ← x
                                otherwise, link x to
                                father(y)
```

47

Tree-Delete (III)

```
14  if y ≠ z
15      then key[z] ← key[y]
16           fields[z] ← fields[y]
17  return y
                                If needed, copy info of
                                successor in node to
                                be canceled
```

48



Complexity

For insert and cancel: $O(h)$.

49



Balancing

The insert/cancel proposed do not guarantee to maintain the tree balanced

50

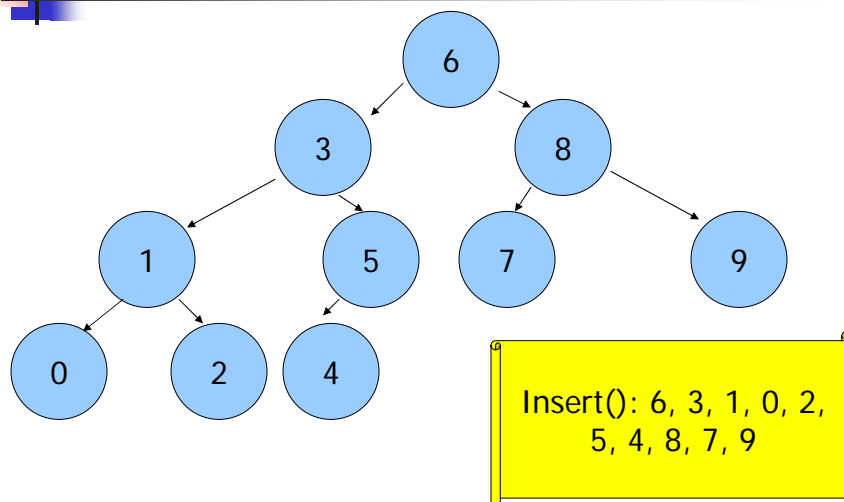
Ex

Build a BST with numbers 0 to 9

- Define an insert sequence that maintains the tree balanced
- Define an insert sequence that maintains the tree unbalanced

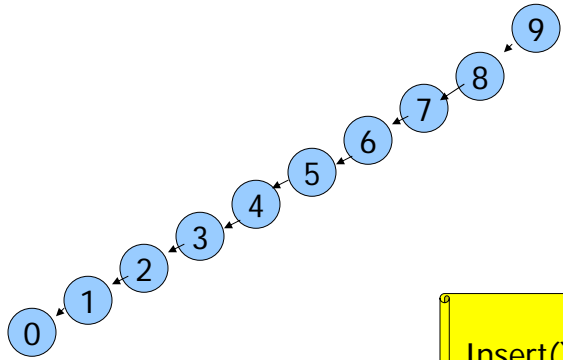
51

Solution (I)



52

Solution (II)



Insert() : 9, 8, 7, 6, 5,
4, 3, 2, 1, 0

C: node

```
typedef struct btnode *P_NODE;  
struct btnode{  
    int key;  
    P_NODE left,  
           right;  
}NODE;
```



inorder

```
void inorder( P_NODE head)
{
    if( head == NULL)
        return;
    inorder( head->left);
    printf( "%d ", head->key);
    inorder( head->right);
}
```

55



search

```
P_NODE search( int key, P_NODE head)
{
    if((head == NULL)!!(head->key == key))
        return( head);
    else
        if( head->key < key)
            return head->right;
        else
            return head->left;
}
```

56



insert (I)

```
int insert( int key, P_NODE *phead)
{ if( *phead == NULL)
  { if((*phead=(P_NODE)malloc
        (sizeof(NODE)))== NULL)
    return( 0);
  else
  { (*phead)->key = key;
    (*phead)->left = NULL;
    (*phead)->right = NULL;
    return( 1);
  }
}
```

57



insert (II)

```
else
  if( (*phead)->key == key)
  { printf("Elemento gia` presente \n");
    return( 0);
  }
else
  if( (*phead)->key < key)
  return( insert( key,
                 &((*phead)->right)));
else
  return( insert( key,
                 &((*phead)->left)));
}
```

58



Main

```
P_NODE    head=NULL;
...
if( !insert( key, &head))
    printf( "Errore in inserimento \n");
...
if( search( key, head))
    printf( "Trovato\n");
else
    printf( "Non trovato\n");
...
inorder( head);
```