Data Structures and Algorithms [1]

BITS-Pilani K. K. Birla Goa Campus

[1] Material for the presentation taken from Cormen, Leiserson, Rivest and Stein, *Introduction to Algorithms, Third Edition*;

Ch. 12: Binary Search Trees

- Dynamic set

- Dynamic set
- Binary Search Tree is a data structure that supports many dynamic set operations:

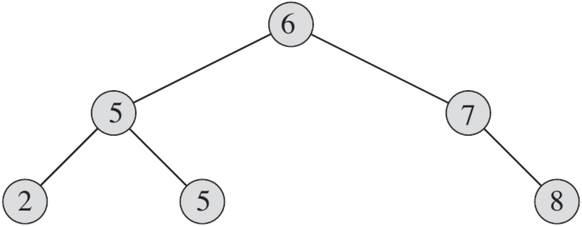
- Dynamic set
- Binary Search Tree is a data structure that supports many dynamic set operations:

  SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT and DELETE.

- Dynamic set
- Binary Search Tree is a data structure that supports many dynamic set operations:

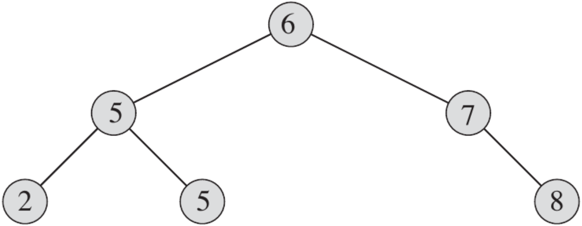
  SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT and DELETE.
- The basic operations take time proportional to the height of the tree (i.e. $O(h)$ time).

Binary search tree



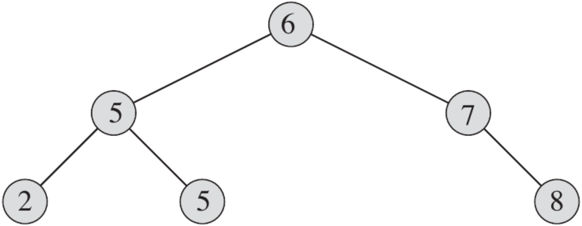
- **Binary-search-tree property**

Binary search tree



- **Binary-search-tree property**

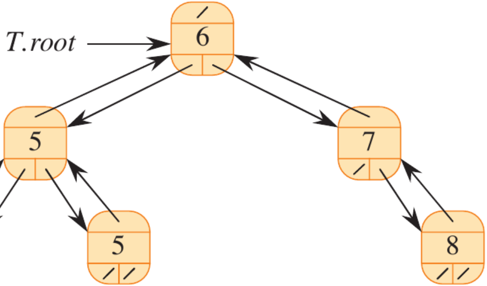
  Let x be a node in a binary search tree. If y is a node in the left subtree of x, then $y.key \leq x.key$. If y is a node in the right subtree of x, then $y.key \geq x.key$.



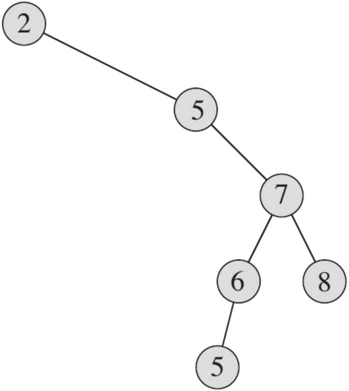
- **Binary-search-tree property**

  Let x be a node in a binary search tree. If y is a node in the left subtree of x, then $y.key \leq x.key$. If y is a node in the right subtree of x, then $y.key \geq x.key$.
- Each node contains a key, satellite data; and attributes *left*, *right* and *p*.

Linked data structure

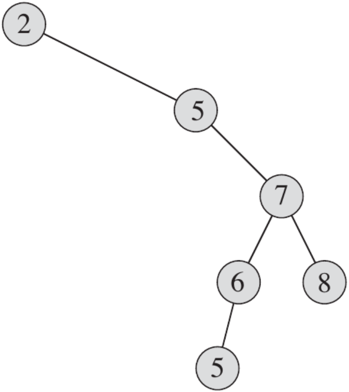


Binary search tree



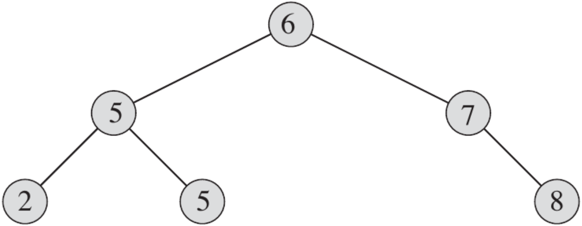
- Does the above binary tree satisfy the Binary-search-tree property?

Binary search tree

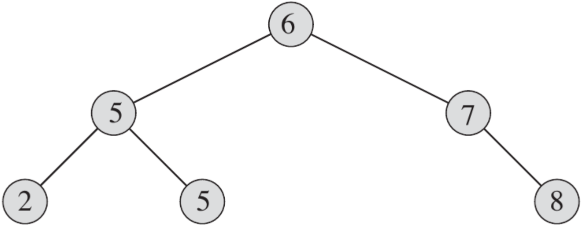


- Does the above binary tree satisfy the Binary-search-tree property?
- The worst-case running time of the search tree operations will be less efficient, because the height of the search tree is more.

Inorder tree walk

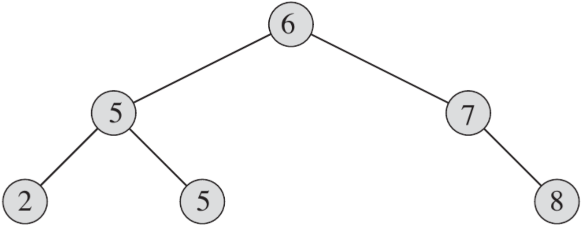


Inorder tree walk



function INORDER-TREE-WALK(x)
    if x $\neq$ NIL **then**
        INORDER-TREE-WALK(x.left)
        print x.key
        INORDER-TREE-WALK(x.right)

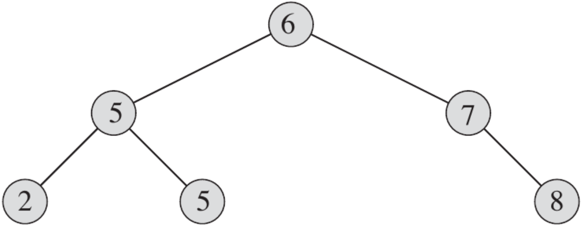
Inorder tree walk




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

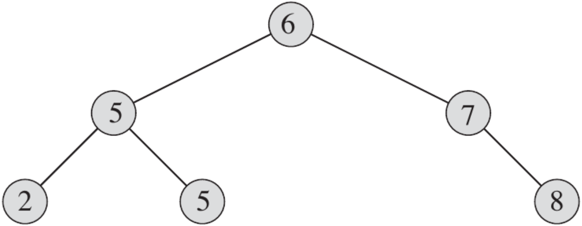
- INORDER-TREE-WALK($T.root$)

Inorder tree walk




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

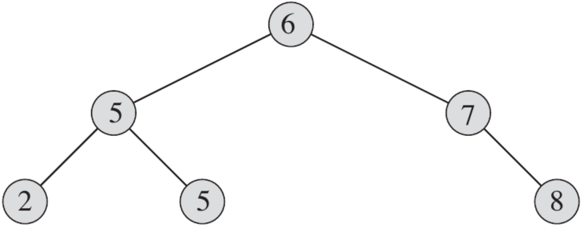
- INORDER-TREE-WALK($T.root$)
- Inorder tree walk prints the keys in a sorted order for a binary search tree.



function INORDER-TREE-WALK(x)
    if x $\neq$ NIL **then**
        INORDER-TREE-WALK(x.left)
        print x.key
        INORDER-TREE-WALK(x.right)

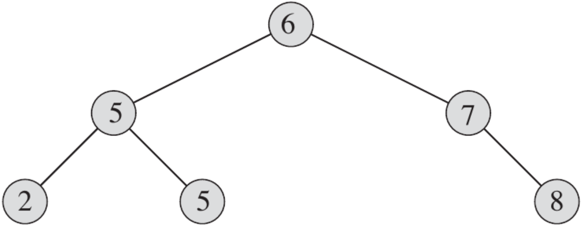
- What is the running time of INORDER-TREE-WALK($T.root$)?

Querying a binary search tree



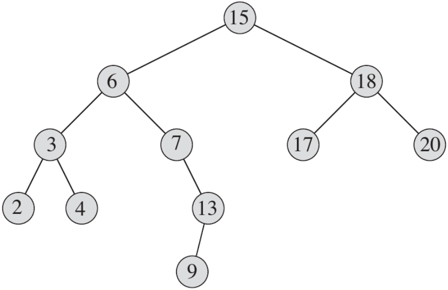
- Operations: SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT and DELETE

Querying a binary search tree

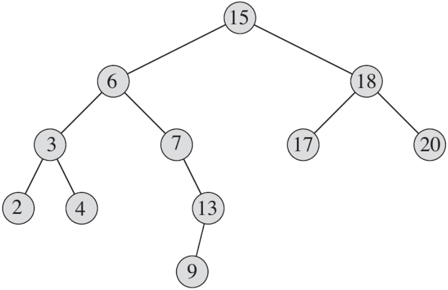


- Operations: SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT and DELETE
- All the operations can be performed in $O(h)$ time.

TREE-SEARCH(x, k)

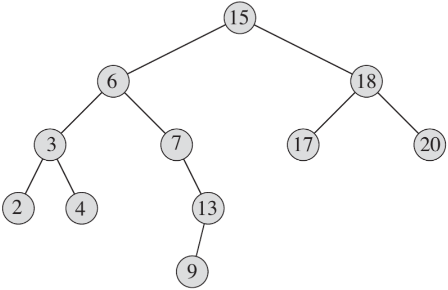


TREE-SEARCH(x, k)



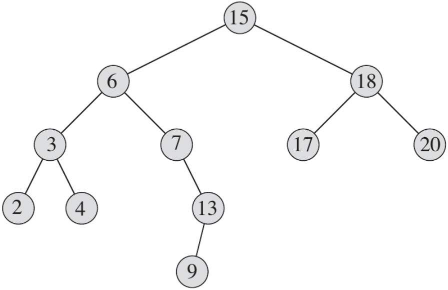
- How can we search node **13**?

TREE-SEARCH(x, k)



- How can we search node **13**?
- How can we search node **12**?

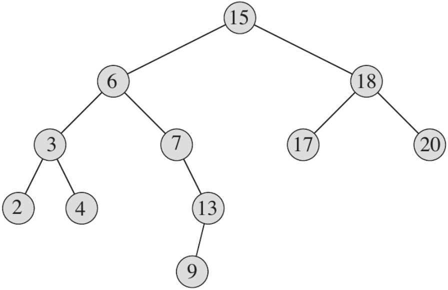
TREE-SEARCH(x, k)



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$(x.right, k)$

TREE-SEARCH(x, k)

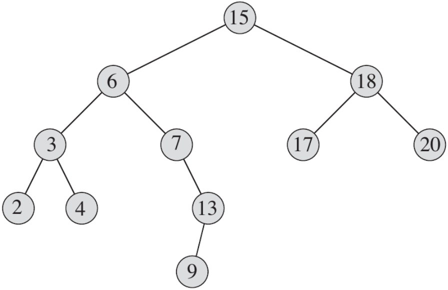


TREE-SEARCH(x, k)

1 **if** $x == \text{NIL}$ or $k == x.key$
2   **return** x
3 **if** $k < x.key$
4   **return** TREE-SEARCH$(x.left, k)$
5 **else return** TREE-SEARCH$(x.right, k)$

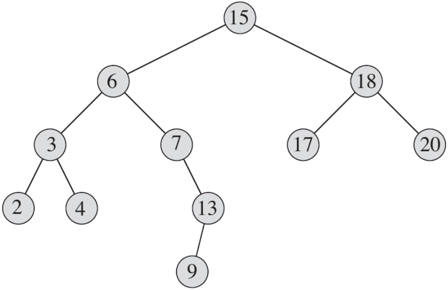
Running time of TREE-SEARCH procedure?

Tree-minimum



- How can we find the minimum element?

Tree-minimum

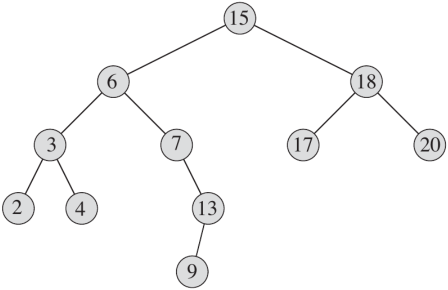


- How can we find the minimum element?

Tree-Minimum(x)

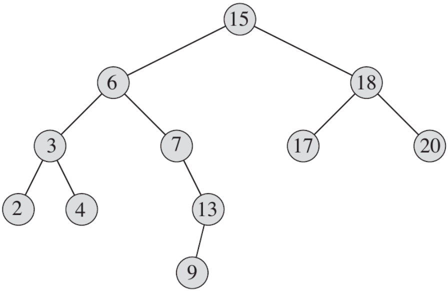
while $x.left \neq$ NIL
    $x = x.left$
return x

Tree-maximum



- How about the maximum element?

TREE-MAXIMUM

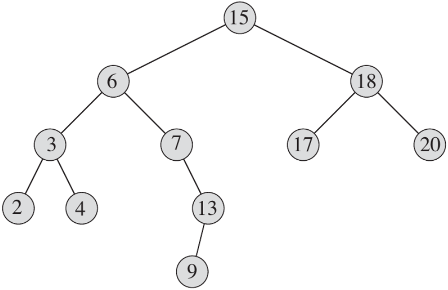


- How about the maximum element?

TREE-MAXIMUM(x)

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

Tree-maximum



- How about the maximum element?

TREE-MAXIMUM(x)

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

- TREE-MINIMUM and TREE-MAXIMUM run in $O(h)$ time.

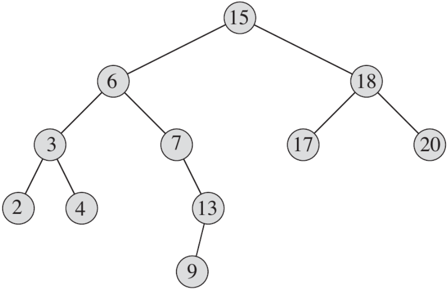
Using Dijkstra's algorithm

- Let $G(V, E)$ be a graph having negative edge weights. Can we use Dijkstra's algorithm?

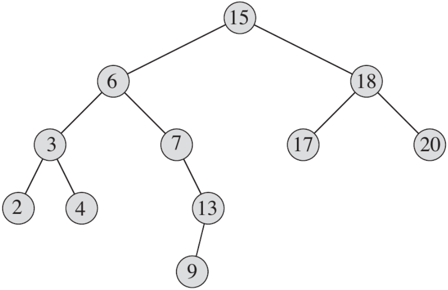
Successor and Predecessor

- Successor of a node x in the BST is the node that comes after node x during an inorder tree walk.

Tree-successor

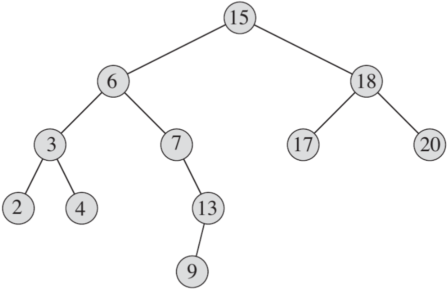


Tree-successor



- How will we find the successor of **15**?

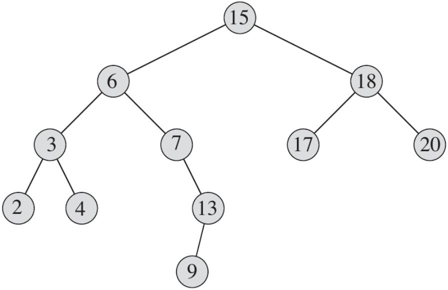
TREE-SUCCESSOR



- How will we find the successor of **15**?

  TREE-MINIMUM(**15**.*right*)

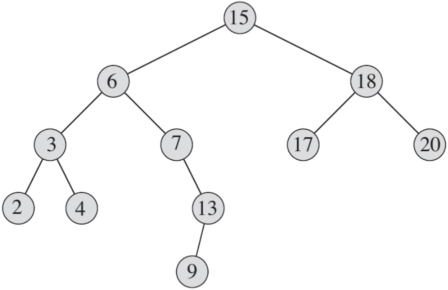
TREE-SUCCESSOR



- How will we find the successor of **15**?

  TREE-MINIMUM(**15**.*right*)
- What if there is no right child?

TREE-SUCCESSOR

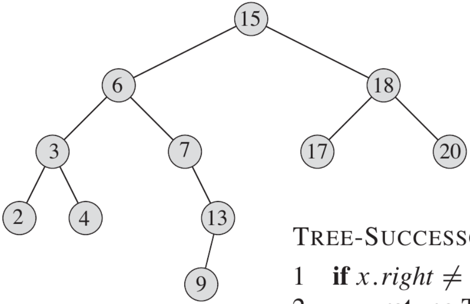


- How will we find the successor of **15**?

  TREE-MINIMUM(**15**.*right*)
- What if there is no right child?

  E.g. How will we find the successor of **13**?

TREE-SUCCESSOR




```
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

- Tree-successor runs in $O(h)$ time.

TREE-SUCCESSOR

- TREE-SUCCESSOR runs in $O(h)$ time.
- The procedure TREE-PREDECESSOR is symmetric to TREE-SUCCESSOR.

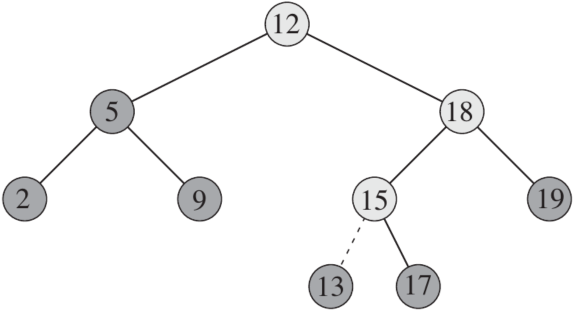
Insert and Delete

INSERT and DELETE

- After we INSERT or DELETE a node from the BST, the binary-search-tree property must continue to hold.

INSERT and DELETE

- After we INSERT or DELETE a node from the BST, the binary-search-tree property must continue to hold.



Tree-Insert

```
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

- For deleting a node, we need to consider multiple cases.

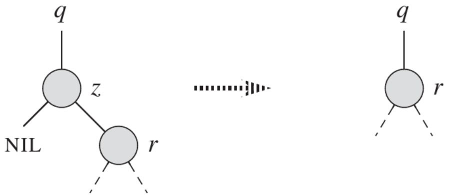
Tree-Delete

- For deleting a node, we need to consider multiple cases.
- We want to delete node z, where z has at most one child node.

Tree-Delete

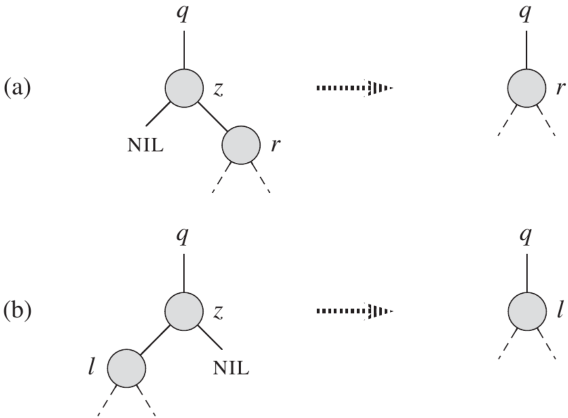
- For deleting a node, we need to consider multiple cases.
- We want to delete node z, where z has at most one child node.

(a)



Tree-Delete

- For deleting a node, we need to consider multiple cases.
- We want to delete node z, where z has at most one child node.



TREE-DELETE procedure

```
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
```

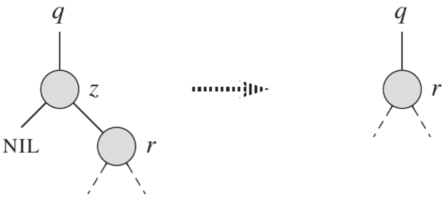
TRANSPLANT procedure

- TRANSPLANT procedure moves subtrees within a binary search tree.

TRANSPLANT procedure

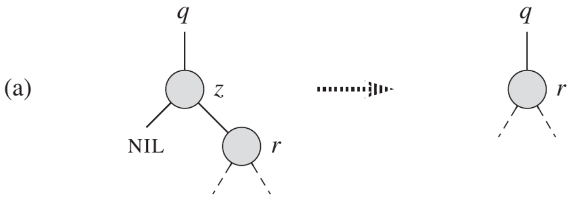
- TRANSPLANT procedure moves subtrees within a binary search tree.

(a)



TRANSPLANT procedure

- TRANSPLANT procedure moves subtrees within a binary search tree.

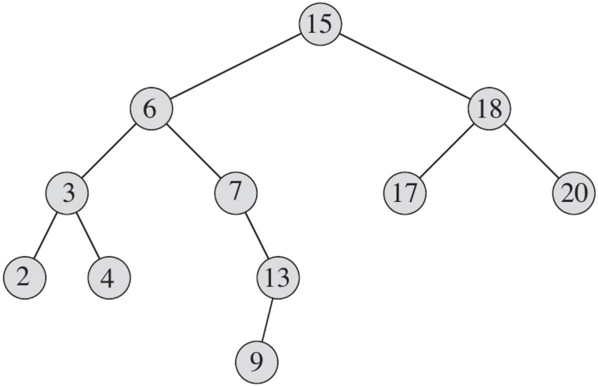


- TRANSPLANT procedure simplifies the TREE-DELETE procedure.

TRANSPLANT procedure

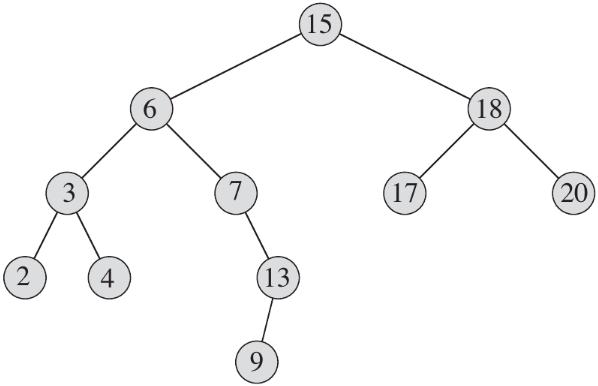
```
TRANSPLANT(T, u, v)
if u.p == NIL
    T.root = v
elseif u == u.p.left
    u.p.left = v
else u.p.right = v
if v ≠ NIL
    v.p = u.p
```

TREE-DELETE : Node z has two child nodes



- When node z has two child nodes, we replace z with its successor node.

Tree-Delete : Node z has two child nodes



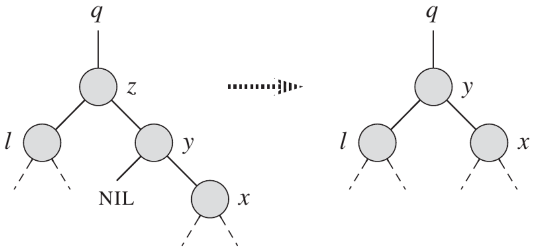
- When node z has two child nodes, we replace z with its successor node.
  (c) Successor node is the right child.
  (d) Successor node is not the right child.

TREE-DELETE : Node z has two child nodes

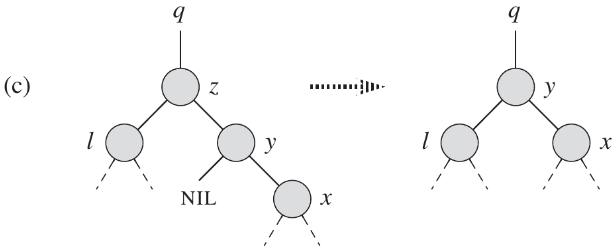
(c) Successor of node z is its right child.

TREE-DELETE : Node z has two child nodes

(c) Successor of node z is its right child.

(c) 

(c) Successor of node z is its right child.



- Suppose node y is the right child and the successor of node z. Is it necessary that the left child of node y be NIL?

TREE-DELETE procedure

```
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
```

TRANSPLANT procedure

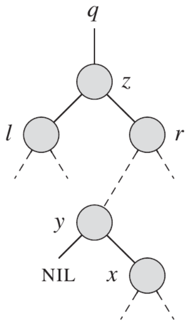
```
TRANSPLANT(T, u, v)
if u.p == NIL
    T.root = v
elseif u == u.p.left
    u.p.left = v
else u.p.right = v
if v ≠ NIL
    v.p = u.p
```

(d) Successor of node z is not its right child

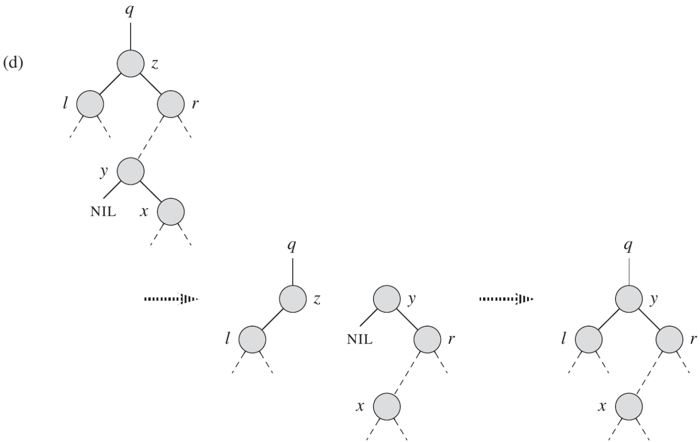
Tree-Delete : Node z has two child nodes

(d) Successor of node z is not its right child

(d)



(d) Successor of node z is not its right child



TREE-DELETE procedure

```
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
```

TRANSPLANT procedure

```
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 procedure

- TREE-DELETE procedure takes $O(h)$ time.

Building binary search trees

- What will the BST look like if we insert the keys in the following order?

  5, 1, 3, 6, 2, 4

Building binary search trees

- What will the BST look like if we insert the keys in the following order?

  5, 1, 3, 6, 2, 4

- What if the order was the following?

  1, 2, 3, 4, 5, 6

Building binary search trees

- What will the BST look like if we insert the keys in the following order?

  5, 1, 3, 6, 2, 4
- What if the order was the following?

  1, 2, 3, 4, 5, 6
- We can perform the following operations in $O(h)$ time :

  Search, Minimum, Maximum, Predecessor, Successor, Insert and Delete

Building binary search trees

- What will the BST look like if we insert the keys in the following order?

  5, 1, 3, 6, 2, 4
- What if the order was the following?

  1, 2, 3, 4, 5, 6
- We can perform the following operations in $O(h)$ time :

  Search, Minimum, Maximum, Predecessor, Successor, Insert and Delete
- However, the height h depends on the order in which the keys are inserted.

Building binary search trees

- **Theorem**: The expected height of a randomly built binary search tree on n distinct keys is $O(\lg n)$.

Building binary search trees

- **Theorem**: The expected height of a randomly built binary search tree on n distinct keys is $O(\lg n)$.
- Proof given in section 12.4. (Proof not part of the syllabus.)