Algorithms and Data Structures 2

14 – Binary search trees

Dr Michele Sevegnani

School of Computing Science
University of Glasgow

michele.sevegnani@glasgow.ac.uk

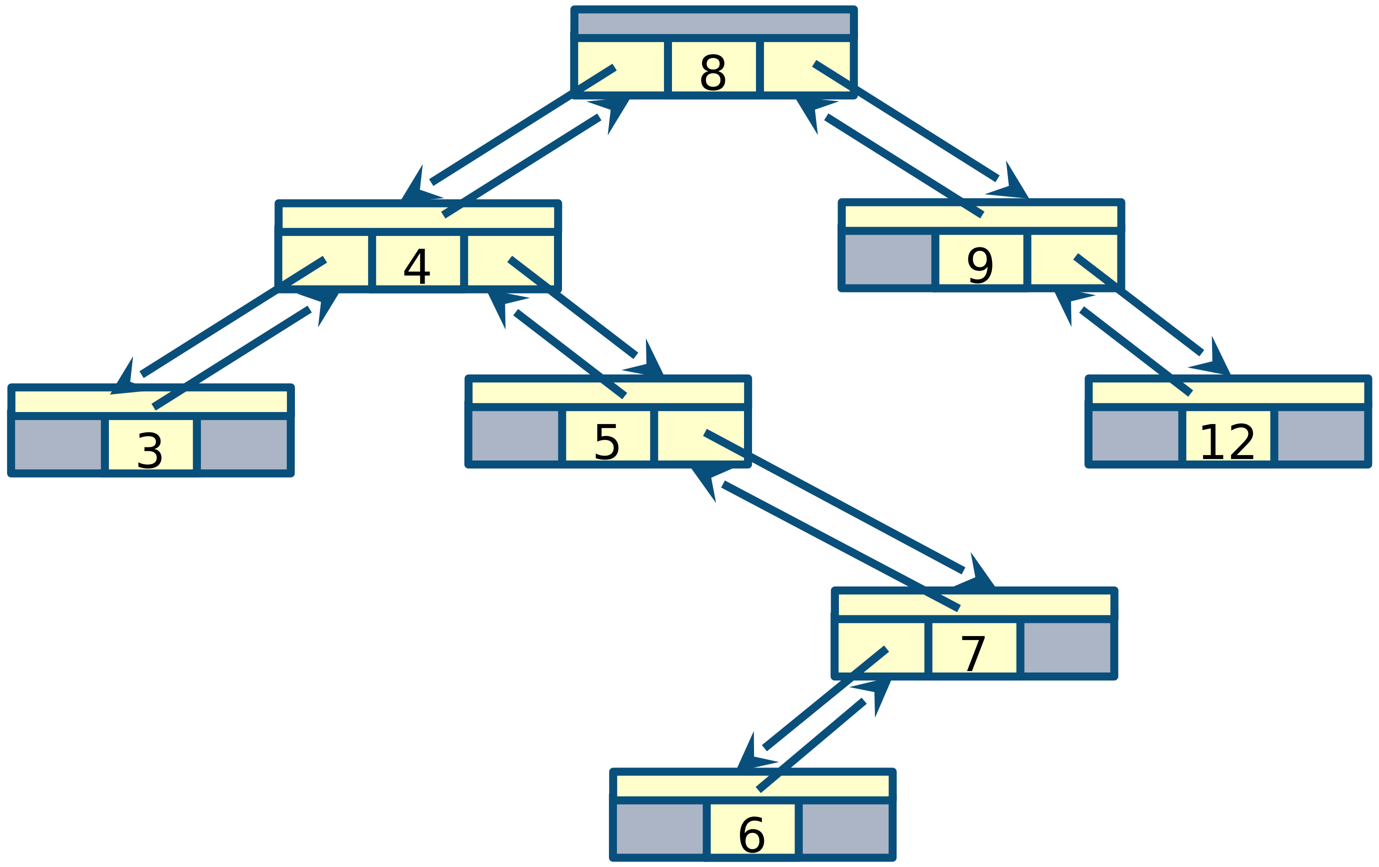
Outline

- **Binary search trees (BSTs)**
- **Querying a tree**
- **Computation of tree parameters**
- **Operations**
  - Insertion
  - Deletion
- **Randomly build BSTs**
- **BSTs with equal keys**

Binary search trees

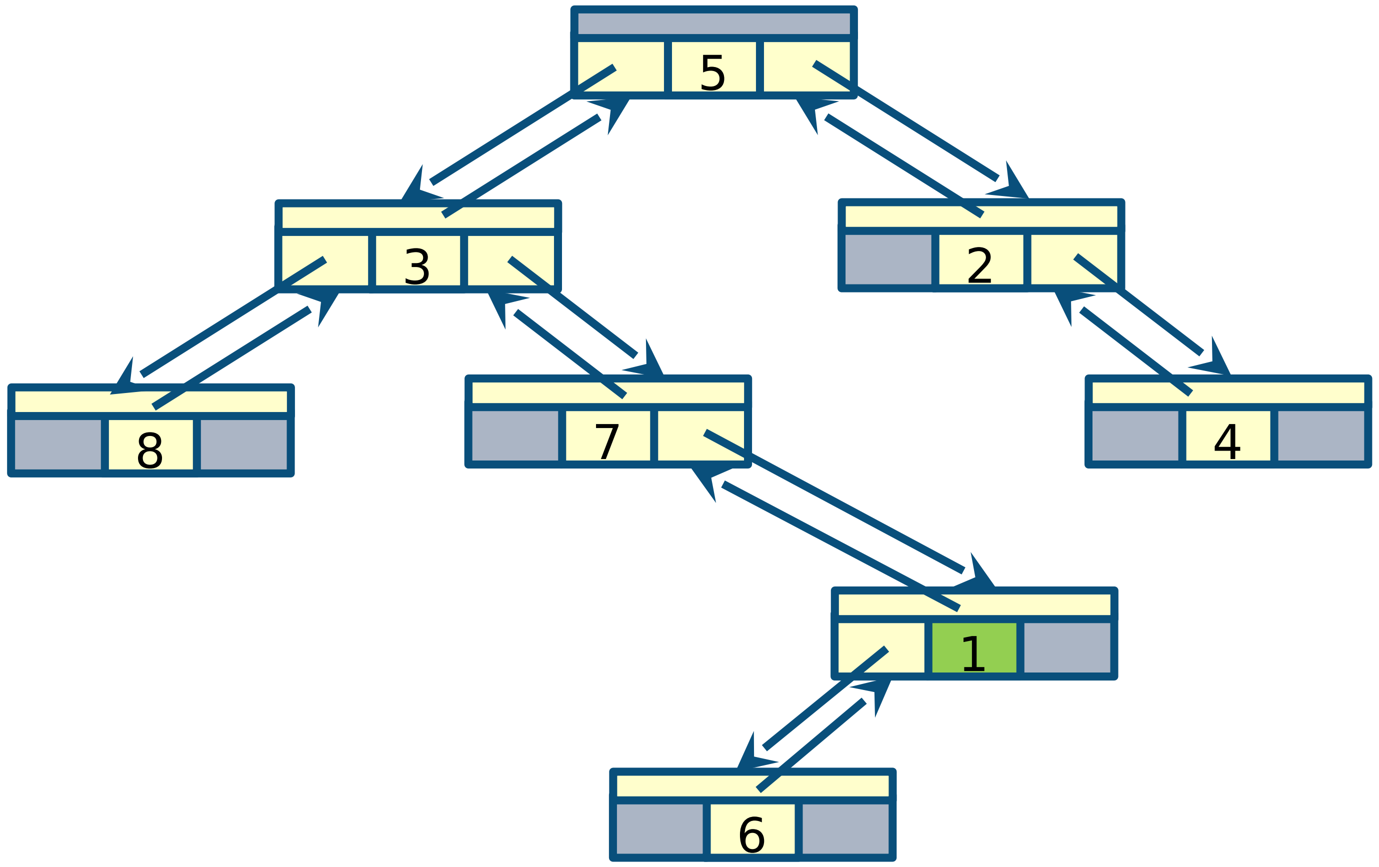
- **A binary search tree (BST) is a binary tree satisfying the binary-search-tree property**
  - Let x be a node. If y is a node in the left subtree of x, then y.key ≤ x.key. If y is a node in the right subtree of x, then y.key ≥ x.key

- **Inorder traversal of a BST is an ordered sequence**
  - 3,4,5,6,7,8,9,12



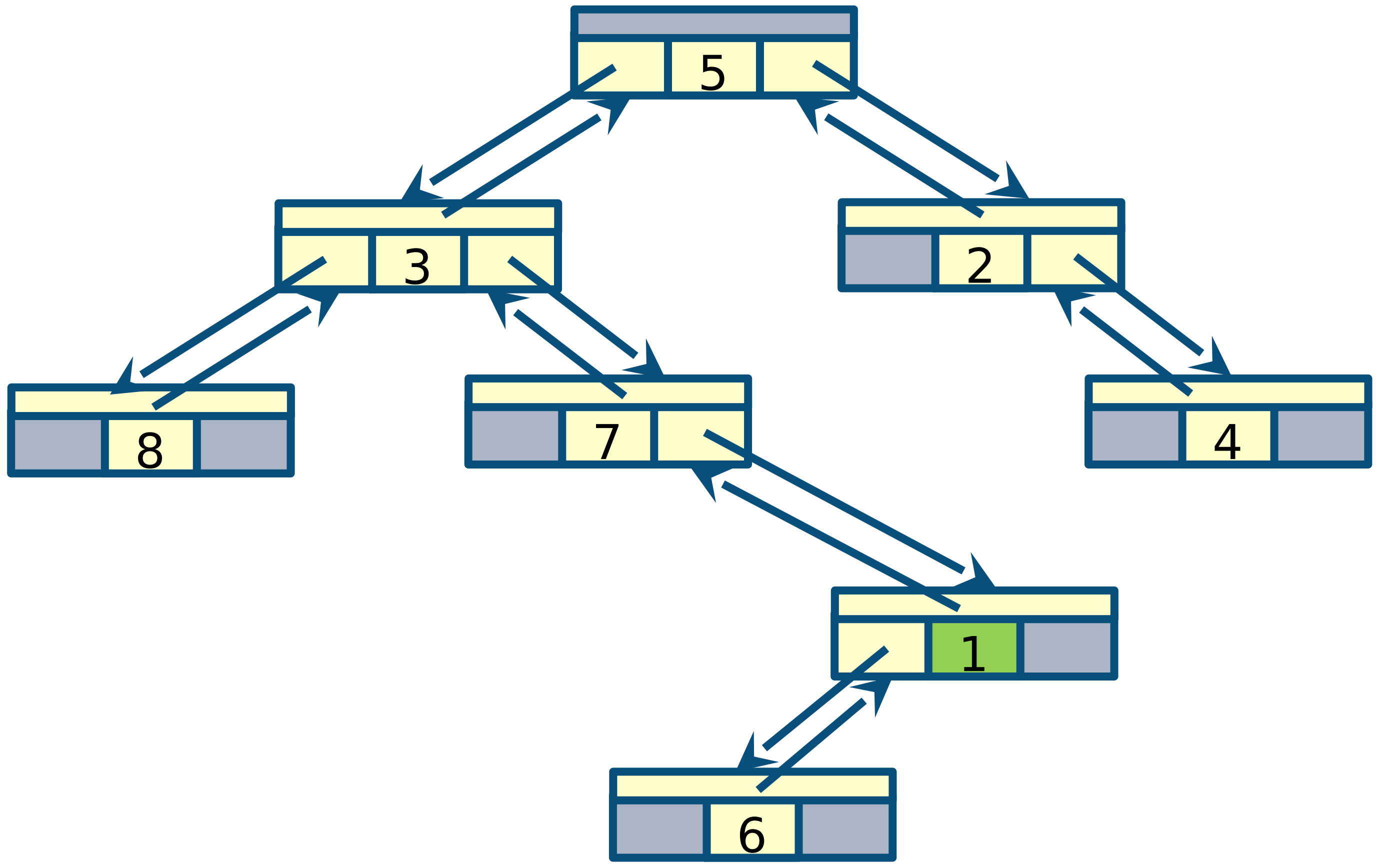
Importance of the BST property

- **Try to find the minimum key in the example binary tree on the right**
  - Note the BST property is not satisfied



Importance of the BST property

- **Try to find the minimum key in the example binary tree on the right**
  - Note the BST property is not satisfied
- **Each node must be visited as the structure of a binary tree does not provide any information on the keys**
  - At each step we need to visit both the left and the right subtree
- **The runtime is O(n)**



Querying a BST

- **Operations to search for a key stored in a BST**
  - SEARCH
  - MINIMUM
  - MAXIMUM
  - SUCCESSOR
  - PREDECESSOR

- **The running time of each of these operations is O(h) on any BST of height h**
  - At each step we can disregard one subtree depending on the key values
  - More efficient than querying ordinary binary trees

SEARCH

- **Search for a node with a given key in a BST**
  - Given a pointer to the root of the tree x and a key k, return a pointer to a node with key k if one exists; return NIL otherwise

- **Recursive definition**
  - If k is smaller than x.key, continue the search in the left subtree of x
  - The search continues in the right subtree otherwise

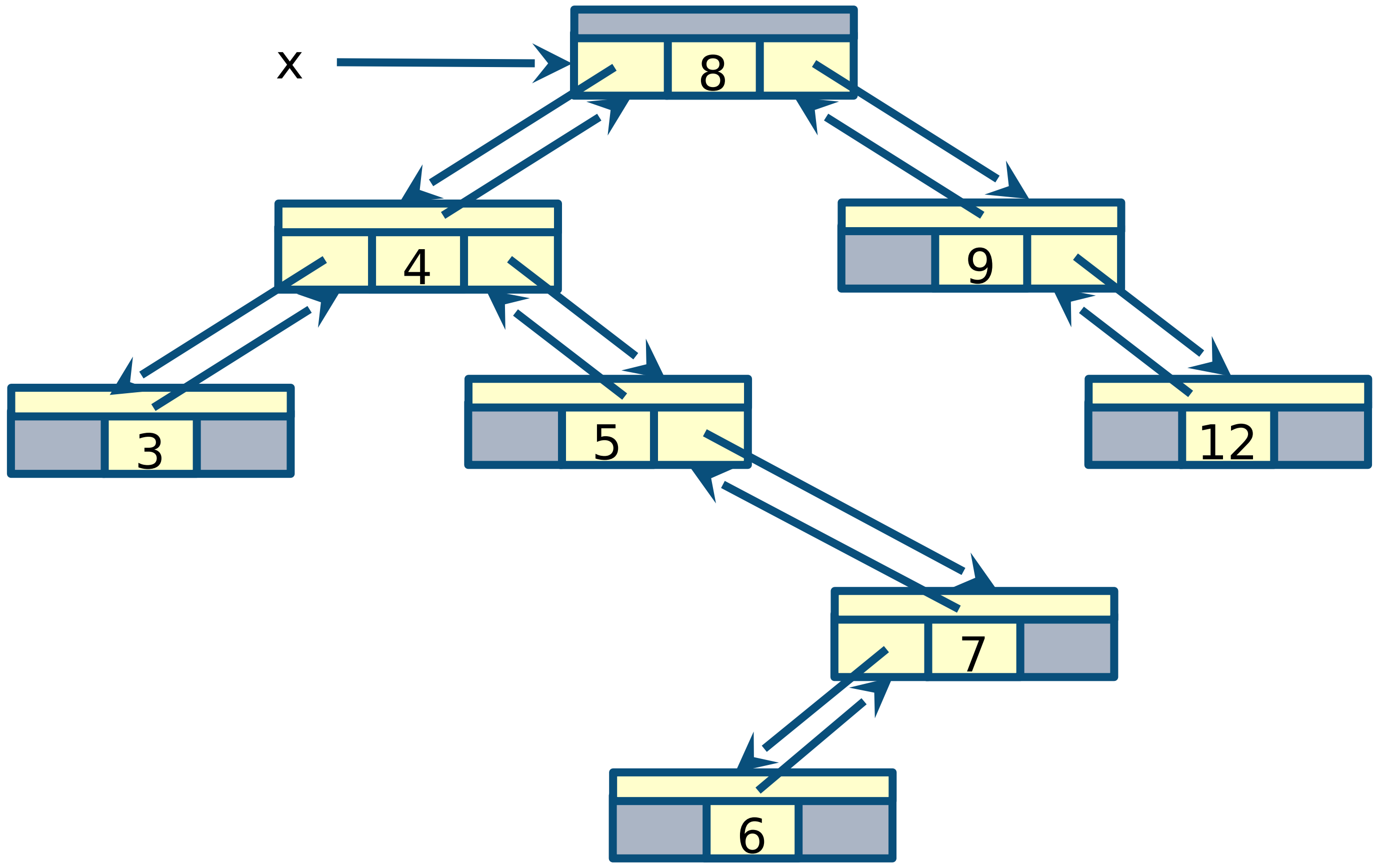
- **The correctness of the procedure follows from the binary-search-tree property**

```
SEARCH(x,k)
  if x = NIL or k = x.key
    return x
  if k < x.key
    return SEARCH(x.left,k)
  else
    return SEARCH(x.right,k)
```

Example

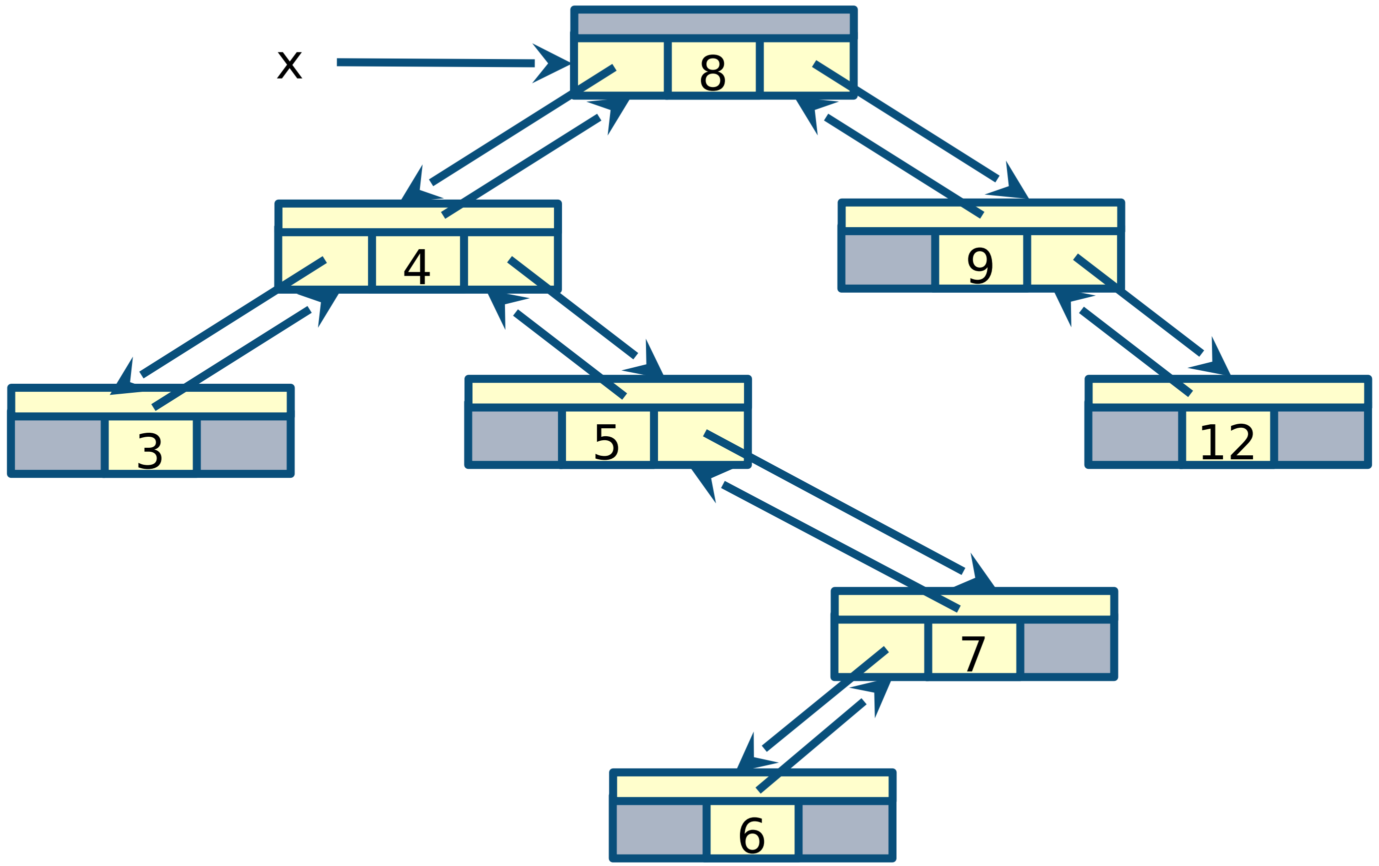
```
SEARCH(x,k)
  if x = NIL or k = x.key
    return x
  if k < x.key
    return SEARCH(x.left,k)
  else
    return SEARCH(x.right,k)
```

- SEARCH(x,7)



Example

```
SEARCH(x,k)
  if x = NIL or k = x.key
    return x
  if k < x.key
    return SEARCH(x.left,k)
  else
    return SEARCH(x.right,k)
```

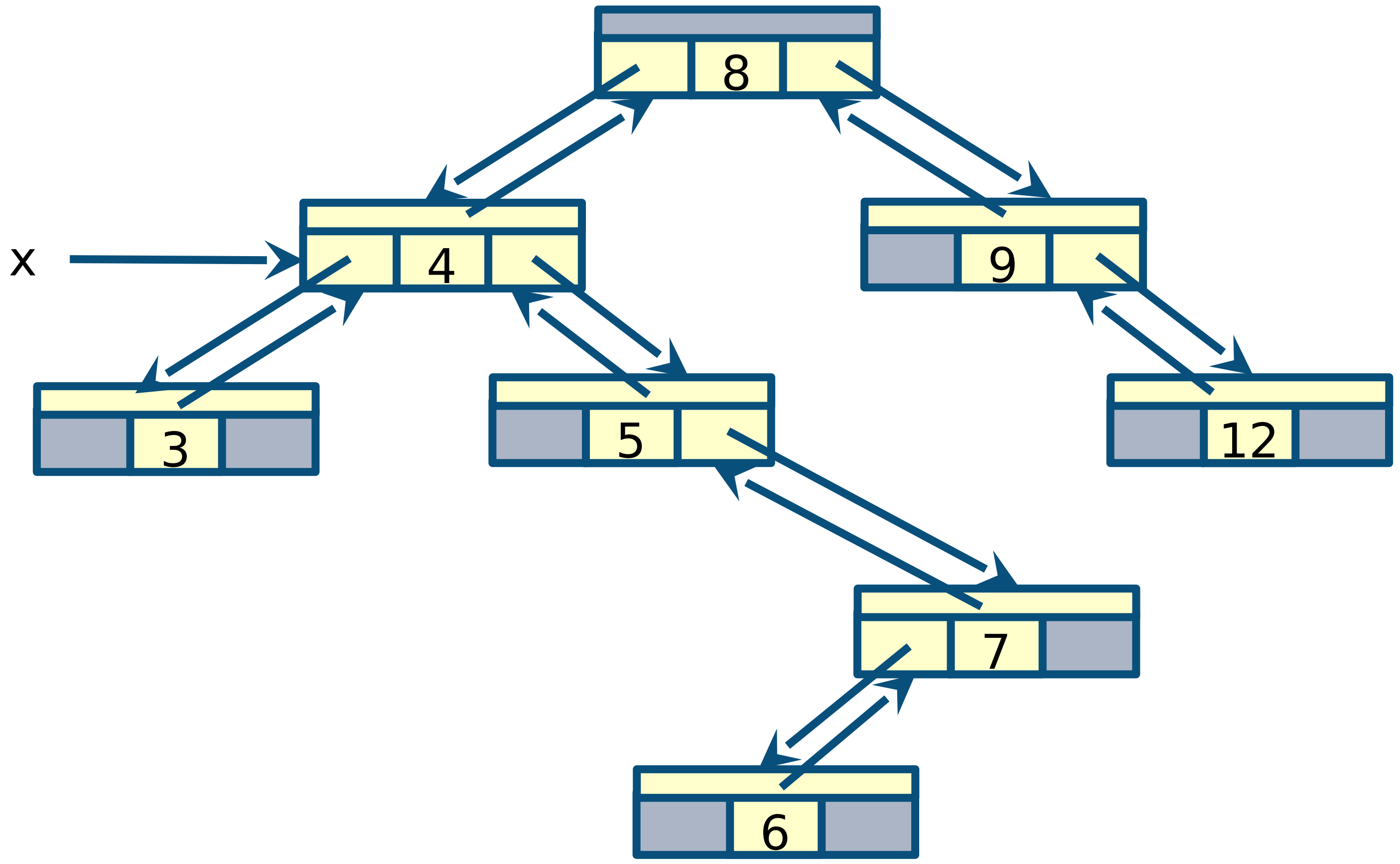


- SEARCH(x,7)

- 7 < 8
- Call SEARCH(x.left,7)

Example

```
SEARCH(x,k)
  if x = NIL or k = x.key
    return x
  if k < x.key
    return SEARCH(x.left,k)
  else
    return SEARCH(x.right,k)
```



- SEARCH(x,7)

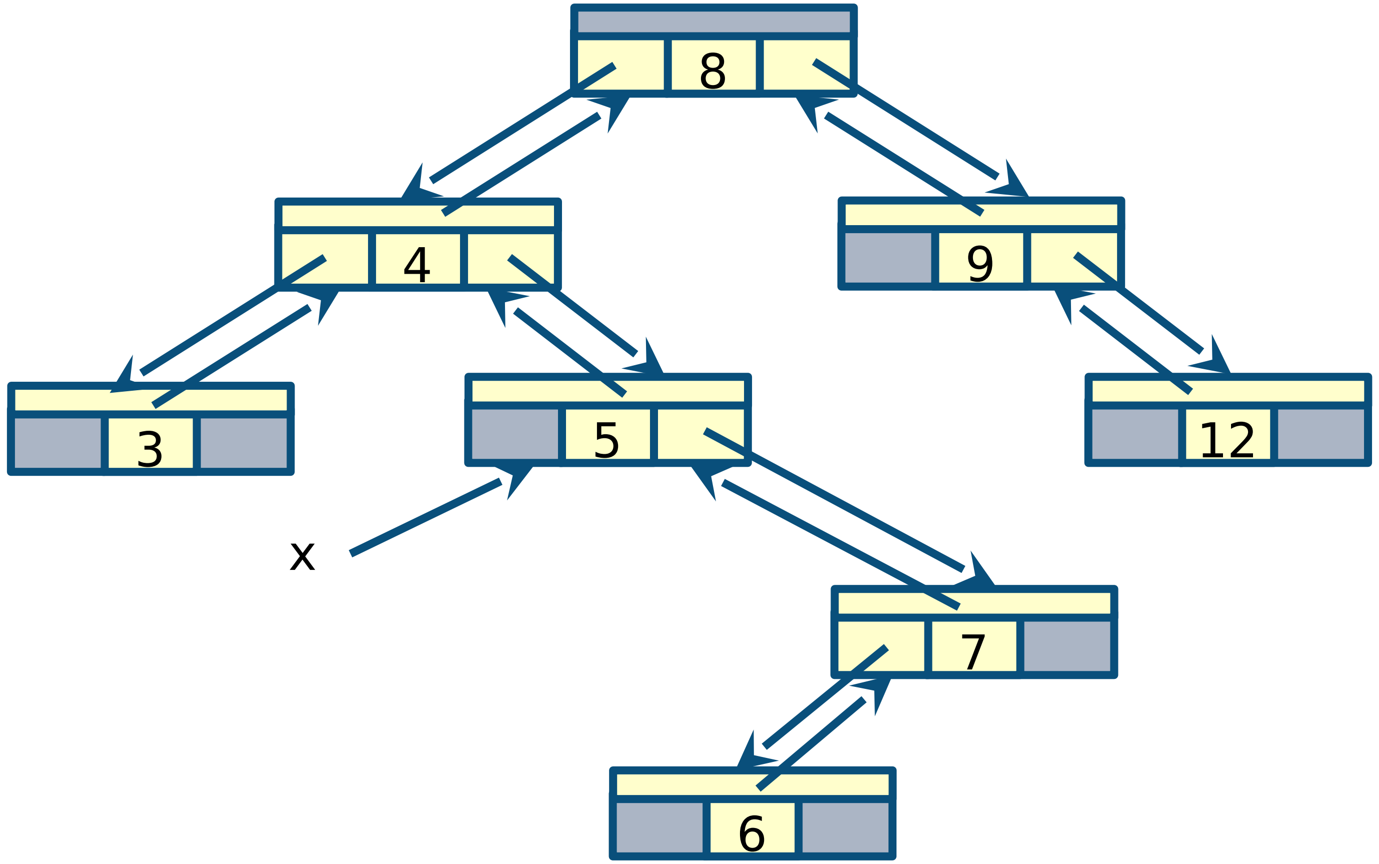
- 7 > 4
- Call SEARCH(x.right,7)

Example

```
SEARCH(x,k)
  if x = NIL or k = x.key
    return x
  if k < x.key
    return SEARCH(x.left,k)
  else
    return SEARCH(x.right,k)
```

- SEARCH(x,7)

- 7 > 5
- Call SEARCH(x.right,7)

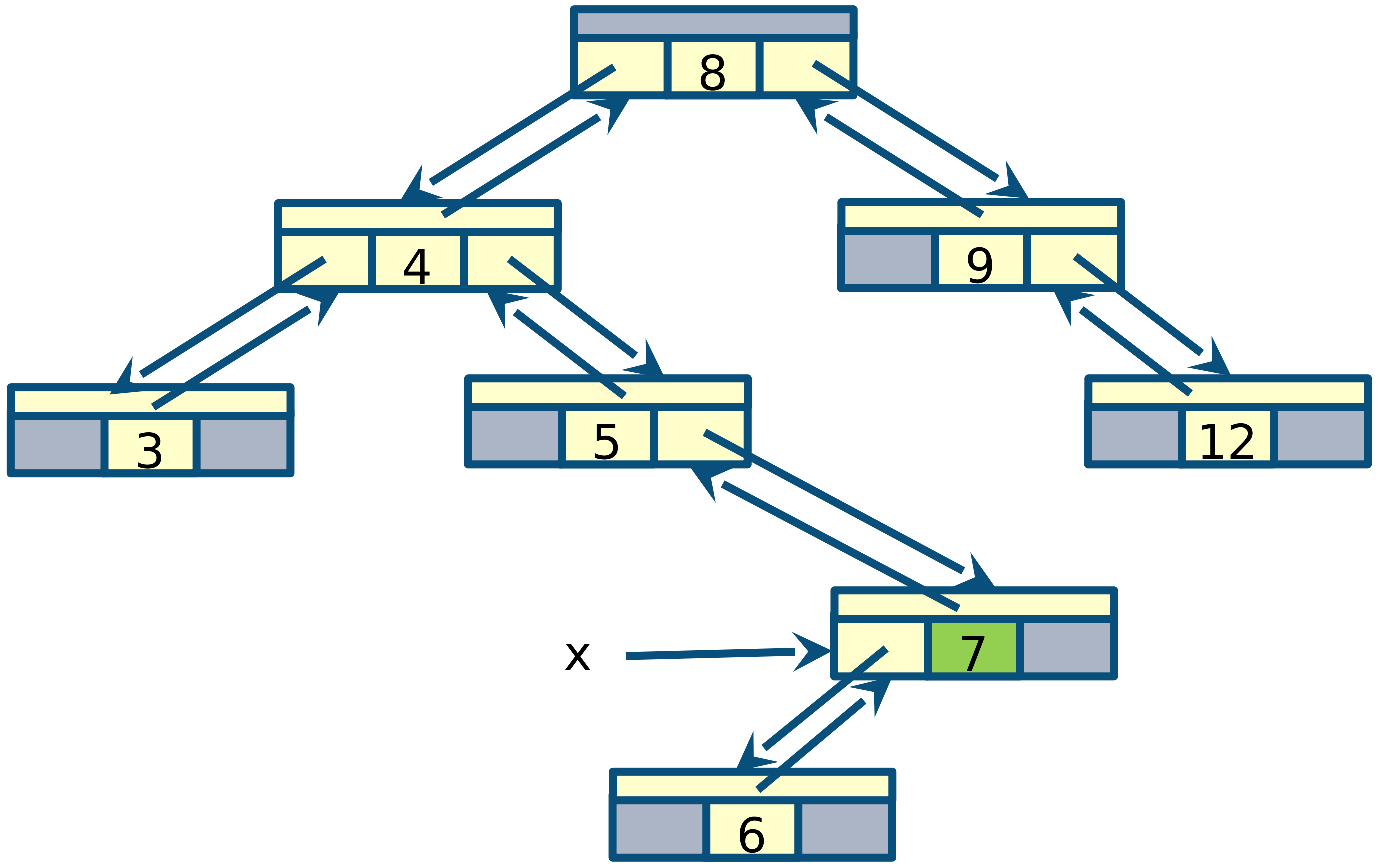


Example

```
SEARCH(x,k)
  if x = NIL or k = x.key
    return x
  if k < x.key
    return SEARCH(x.left,k)
  else
    return SEARCH(x.right,k)
```

- SEARCH(x,7)

- 7 = 7
- Return x



Iterative SEARCH

- **The same operation can be implemented in an iterative fashion**
  - Unroll the recursion in a while loop

- **Usually more efficient**

```
SEARCH-ITER(x,k)
  while x != NIL and k != x.key
    if k < x.key
      x := x.left
    else
      x := x.right
  return x
```

MINUMUM and MAXIMUM

- **Search for the minimum (maximum) key in a BST**

- **MINIMUM**
  - Follow left pointers from the root until we encounter a NIL

- **MAXIMUM**
  - Follow right pointers from the root until we encounter a NIL

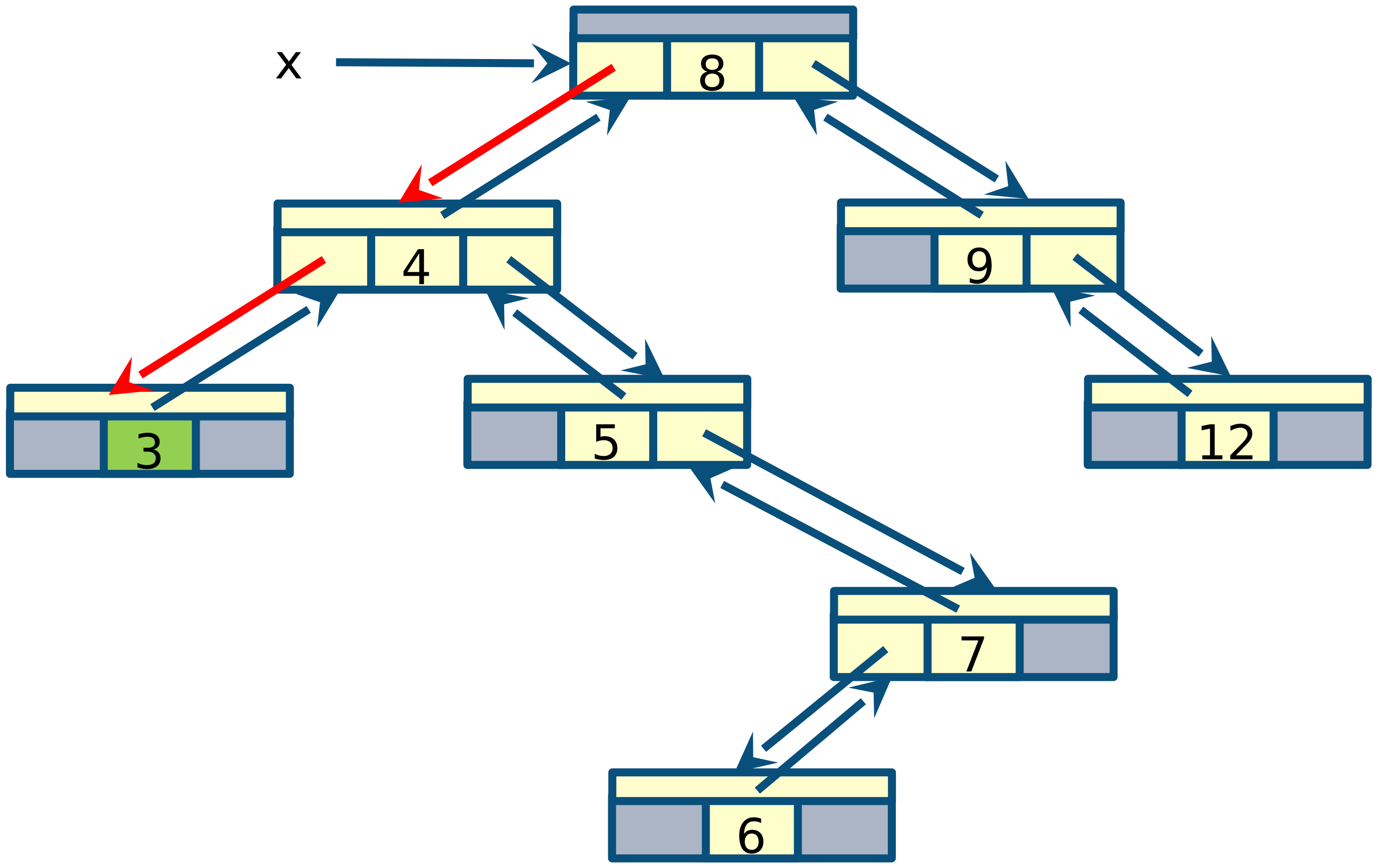
```
MINIMUM(x)
  while x.left != NIL
    x := x.left
  return x
```

```
MAXIMUM(x)
  while x.right != NIL
    x := x.right
  return x
```

- **The binary-search-tree property guarantees that MINIMUM and MAXIMUM are correct**

Example

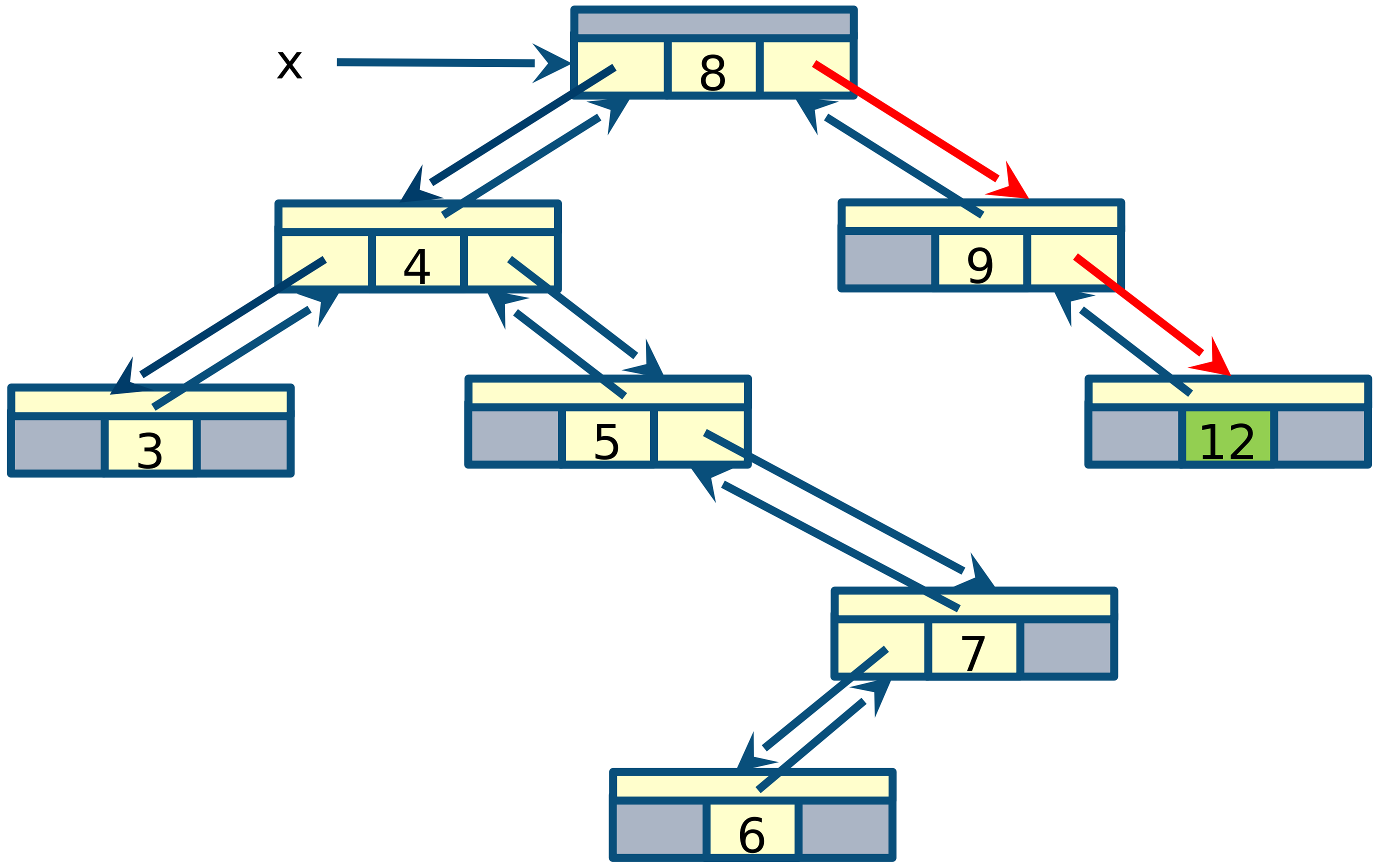
```
MINIMUM(x)
  while x.left != NIL
    x := x.left
  return x
```



- Follow the red pointers until a NIL is found
- Return 3

Example

```
MAXIMUM(x)
  while x.right != NIL
    x := x.right
  return x
```



- Follow the red pointers until a NIL is found
- Return 12

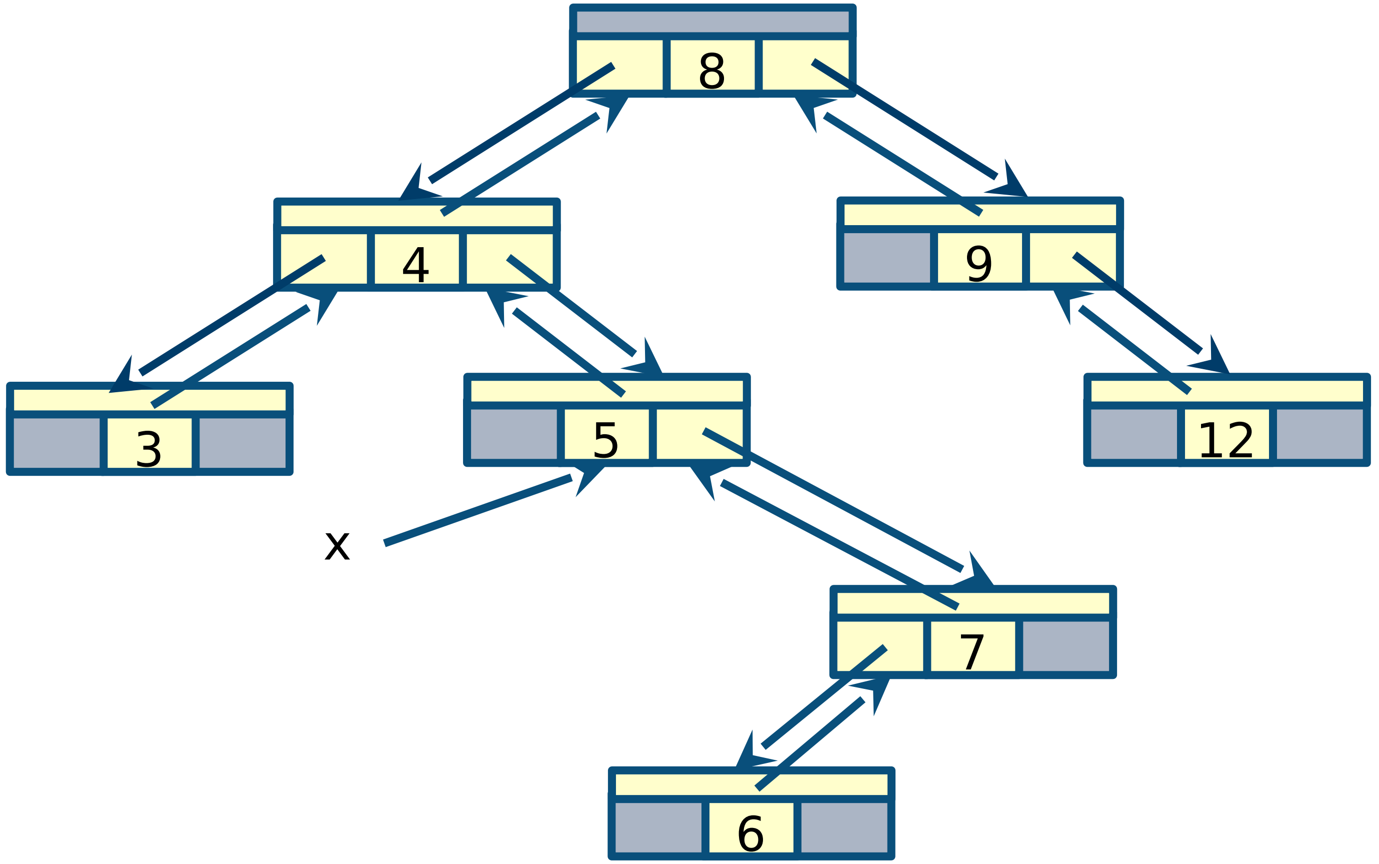
SUCCESSOR

- **Given a node, find its successor in the sorted order determined by an inorder traversal**
- **If all keys are distinct, the successor of a node x is the node with the smallest key greater than x.key**
- **No comparison of keys performed**
  - Exploit the structure of the BST
- **Return NIL if x is the maximum**

```
SUCCESSOR(x)
  if x.right != NIL
    return MINIMUM(x.right)
  y := x.p
  while y != NIL and x = y.right
    x := y
    y := y.p
  return y
```

Example

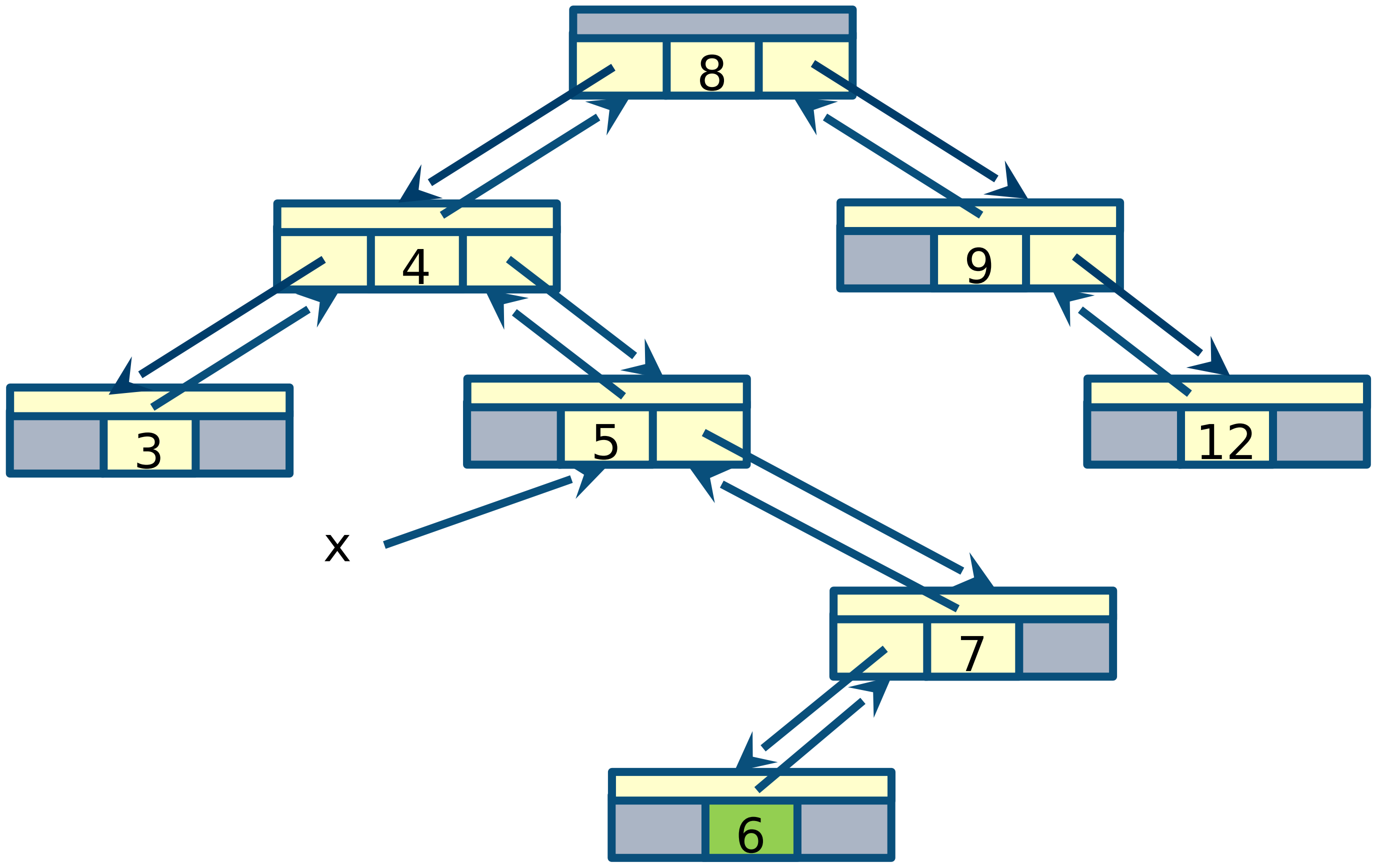
```
SUCCESSOR(x)
  if x.right != NIL
    return MINIMUM(x.right)
  y := x.p
  while y != NIL and x = y.right
    x := y
    y := y.p
  return y
```



– SUCCESSOR(x) x.key = 5

Example

```
SUCCESSOR(x)
  if x.right != NIL
    return MINIMUM(x.right)
  y := x.p
  while y != NIL and x = y.right
    x := y
    y := y.p
  return y
```

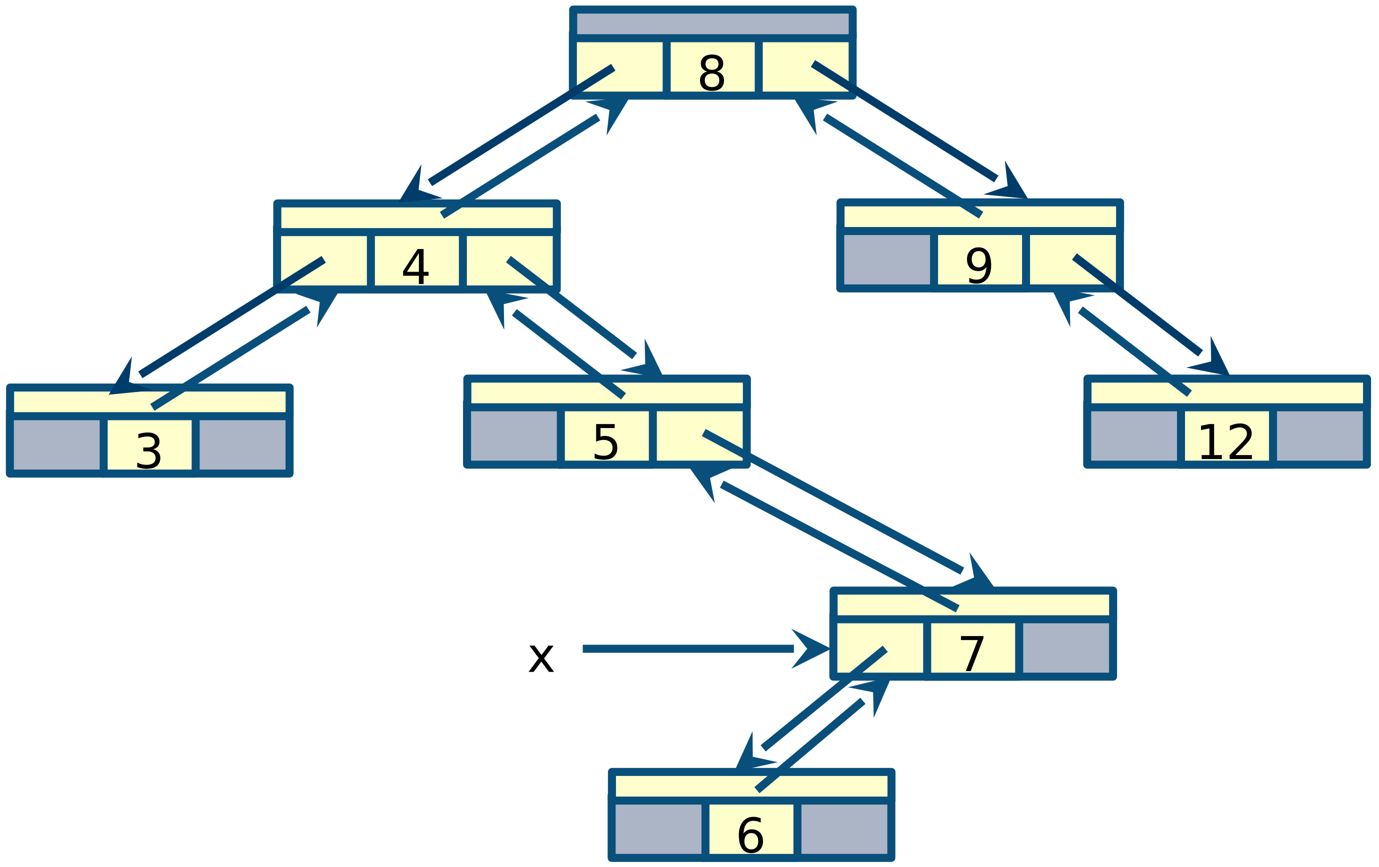


- SUCCESSOR(x) x.key = 5
- x.right != NIL
- Return the minimum of the tree rooted at 7
- 6

Example

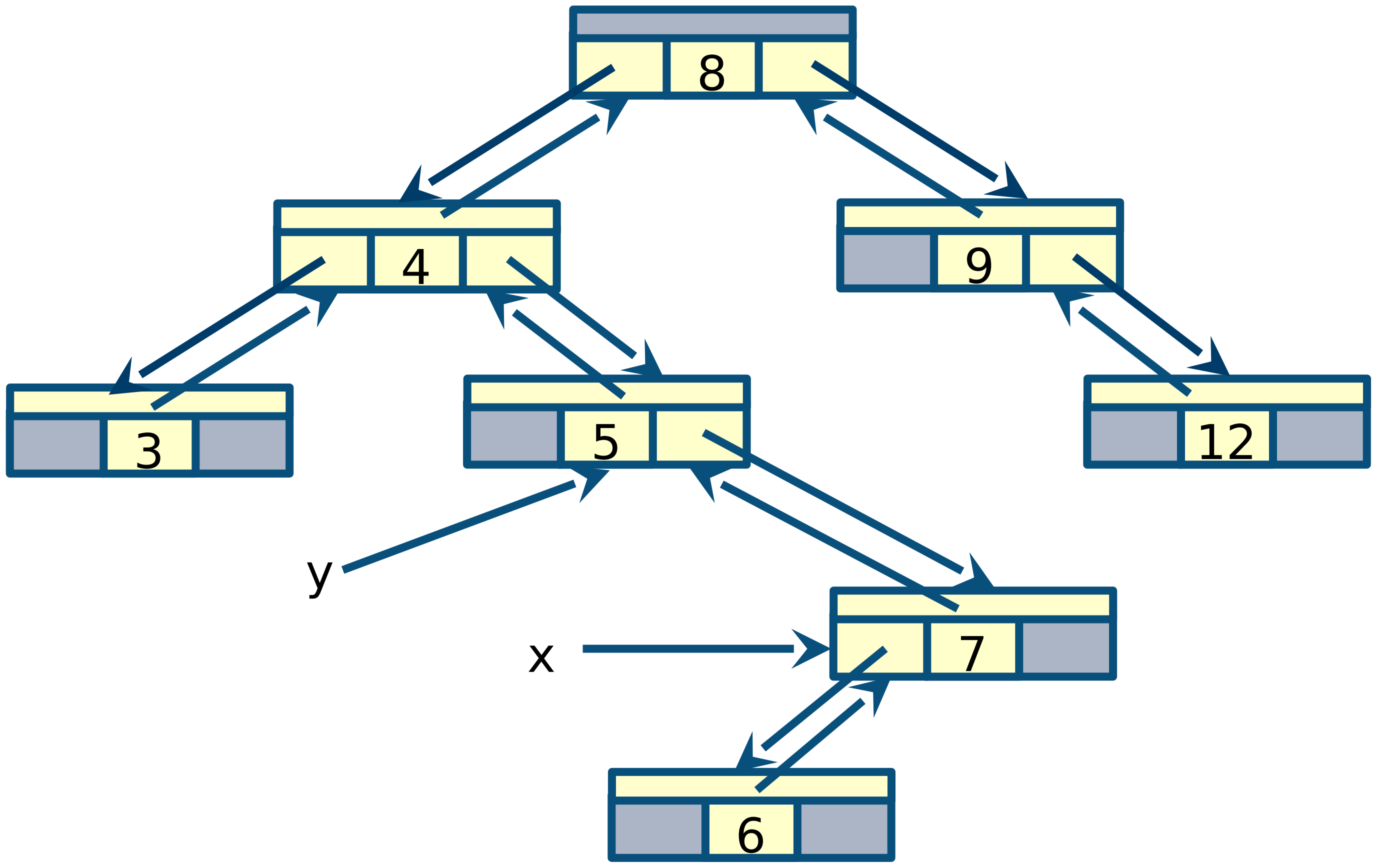
```
SUCCESSOR(x)
  if x.right != NIL
    return MINIMUM(x.right)
  y := x.p
  while y != NIL and x = y.right
    x := y
    y := y.p
  return y
```

– SUCCESSOR(x) x.key = 7



Example

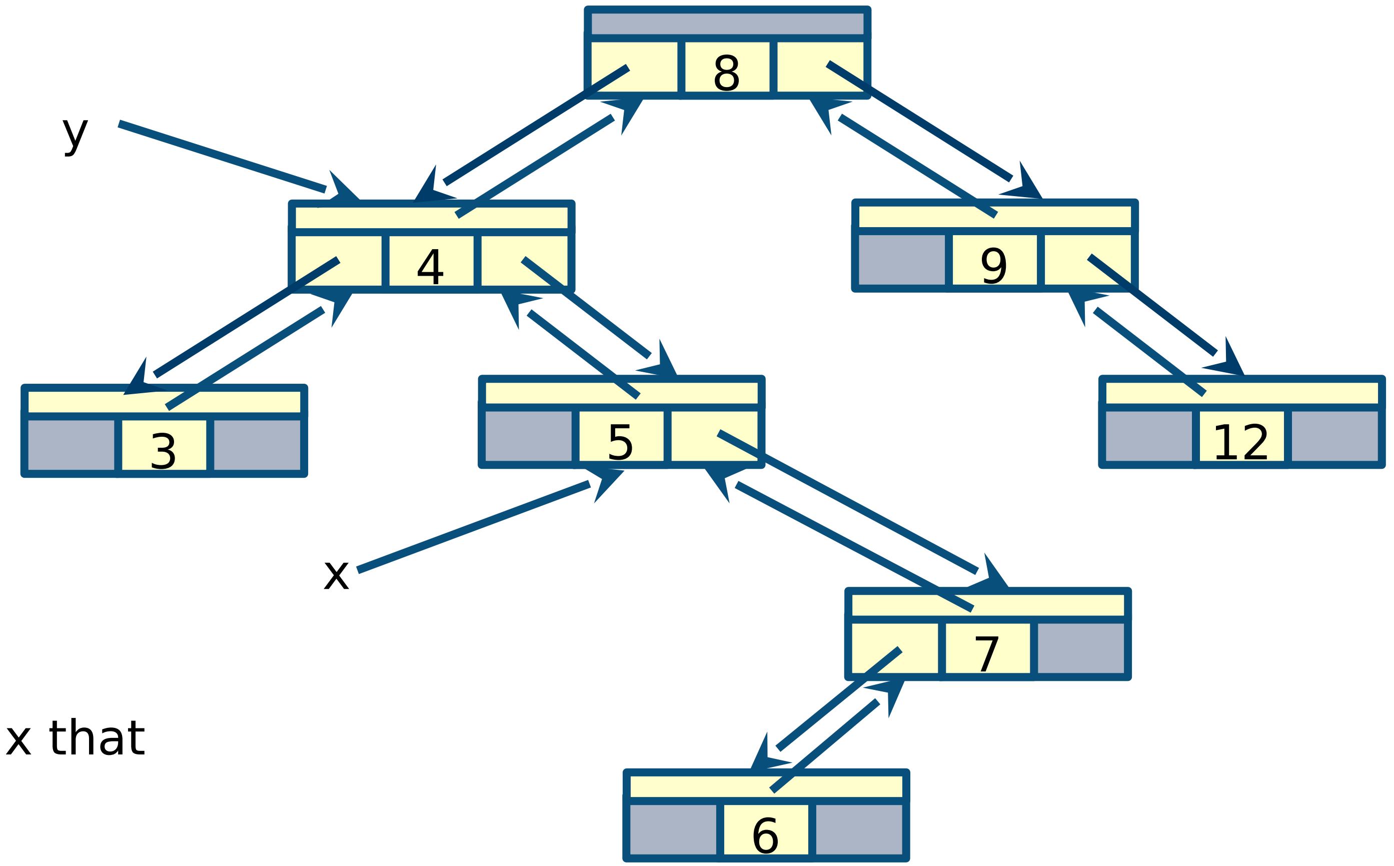
```
SUCCESSOR(x)
  if x.right != NIL
    return MINIMUM(x.right)
  y := x.p
  while y != NIL and x = y.right
    x := y
    y := y.p
  return y
```



- SUCCESSOR(x) x.key = 7
- x.right = NIL
- y points to 5 and enter while loop

Example

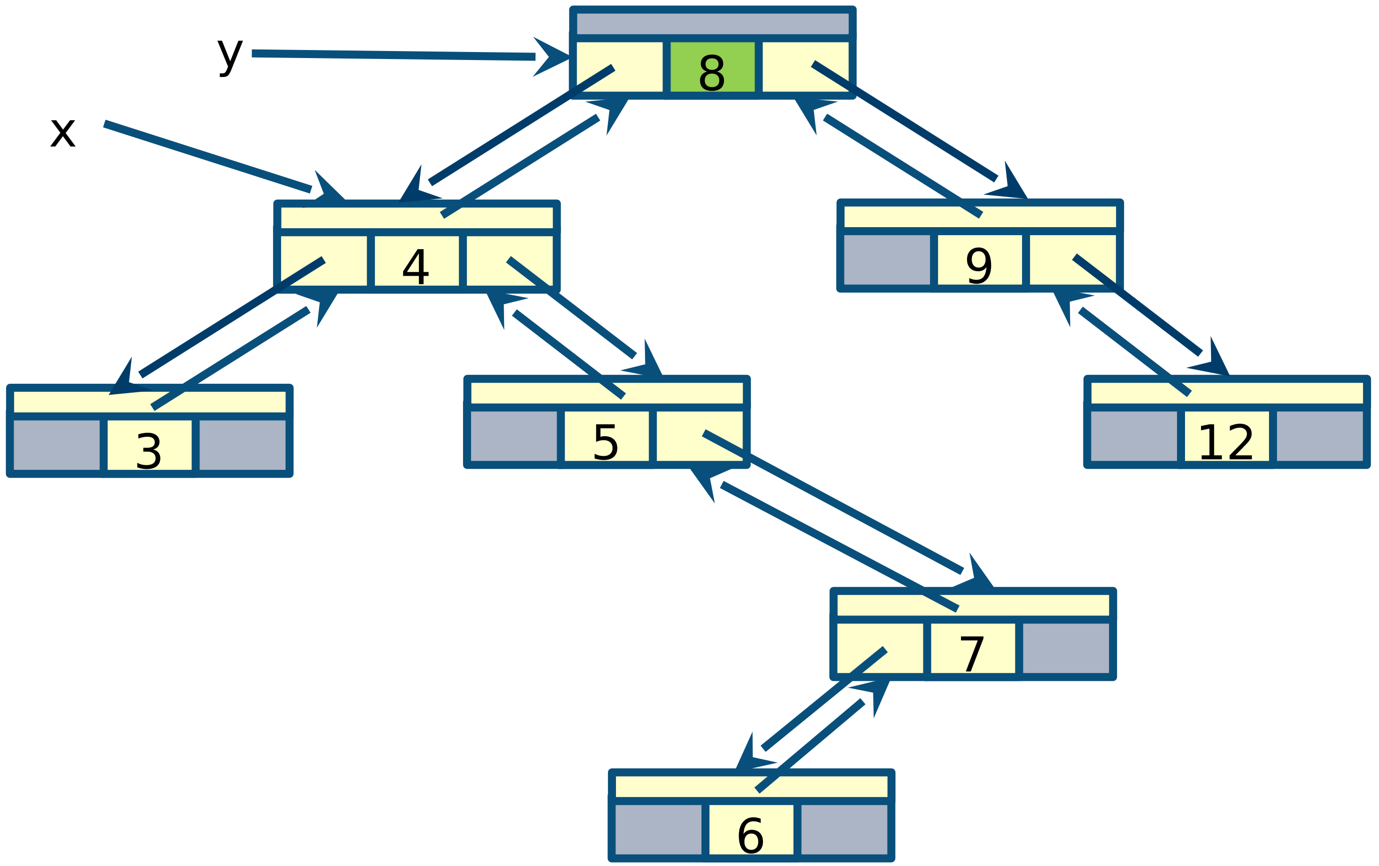
```
SUCCESSOR(x)
  if x.right != NIL
    return MINIMUM(x.right)
  y := x.p
  while y != NIL and x = y.right
    x := y
    y := y.p
  return y
```



- SUCCESSOR(x) x.key = 7
- Go up the tree until we encounter a node x that is the left child of its parent y

Example

```
SUCCESSOR(x)
  if x.right != NIL
    return MINIMUM(x.right)
  y := x.p
  while y != NIL and x = y.right
    x := y
    y := y.p
  return y
```



- SUCCESSOR(x) x.key = 7
- x != y.right return y
- 8

PREDECESSOR

- **Given a node, find its predecessor in the sorted order determined by an inorder traversal**

- **Definition is symmetric to SUCCESSOR**

```
PREDECESSOR(x)
  if x.left != NIL
    return MAXIMUM(x.left)
  y := x.p
  while y != NIL and x = y.left
    x := y
    y := y.p
  return y
```

Computation of tree parameters

```
SIZE(x)
  if x = NIL
    return 0
  return SIZE(x.left) + SIZE(x.right) + 1
```

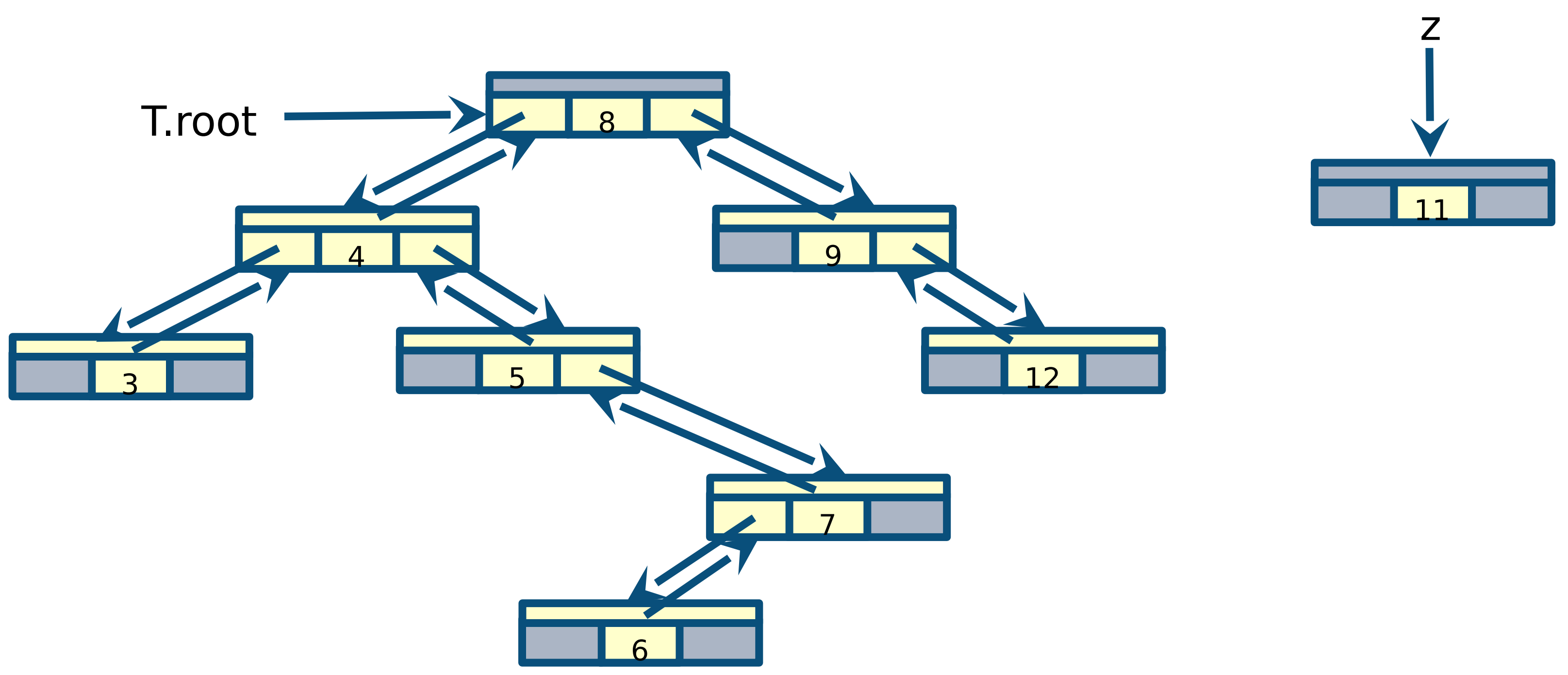
```
HEIGHT(x)
  if x = NIL
    return 0
  if x.left = NIL and x.right = NIL
    return 0
  return MAX(HEIGHT(x.left),HEIGHT(x.right)) + 1
```

Insertion

- **Insert a new node z into an appropriate position in tree T**
  - The binary-search-tree property must be preserved
  - Start at the root and go downwards until you find a NIL to be replaced by z
  - Go left or right depending on the comparison of z.key with x.key
  - Maintain a trailing pointer y as the parent of x
- **Runs in O(h) on a tree of height h**

```
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
  elseif z.key < y.key
    y.left := z
  else y.right := z
```

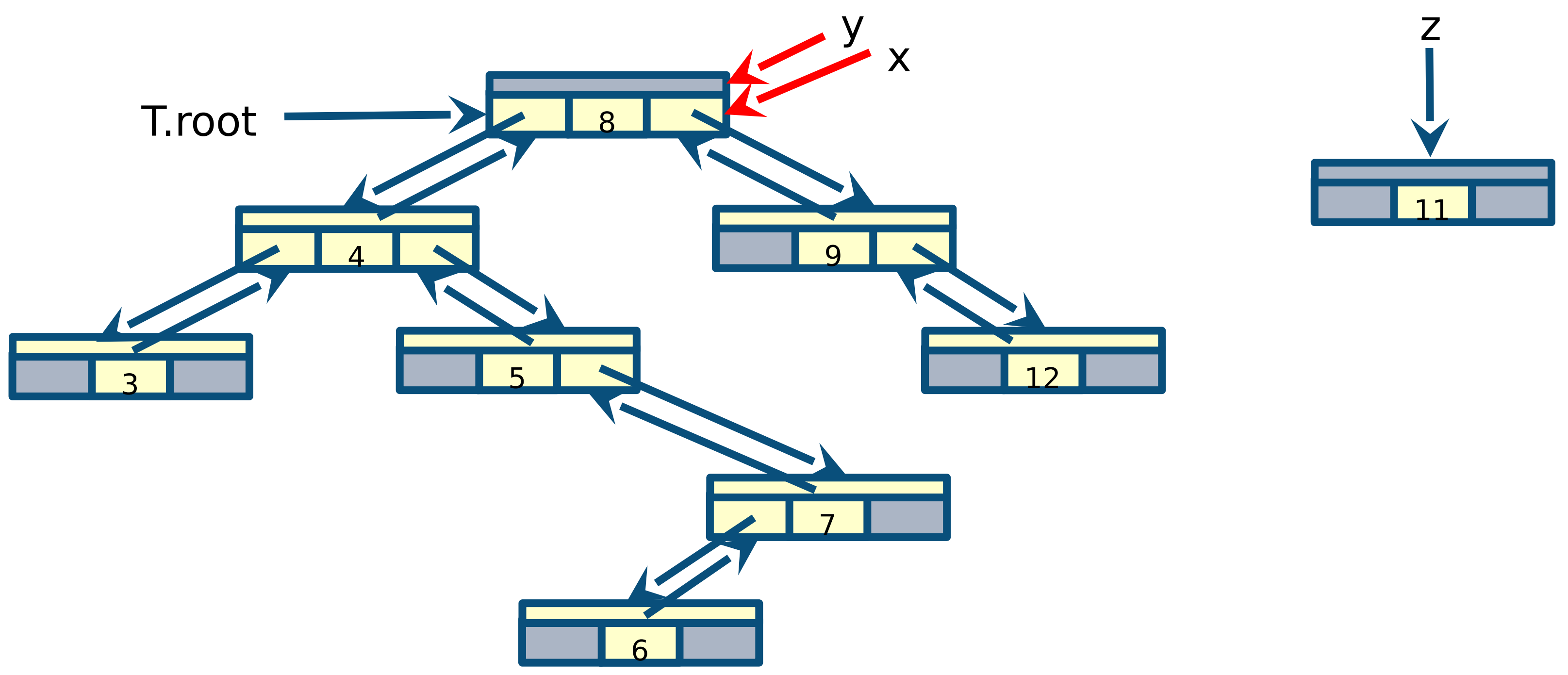
Example



```
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
  elseif z.key < y.key
    y.left := z
  else y.right := z
```

– INSERT(T,z) z.key = 11

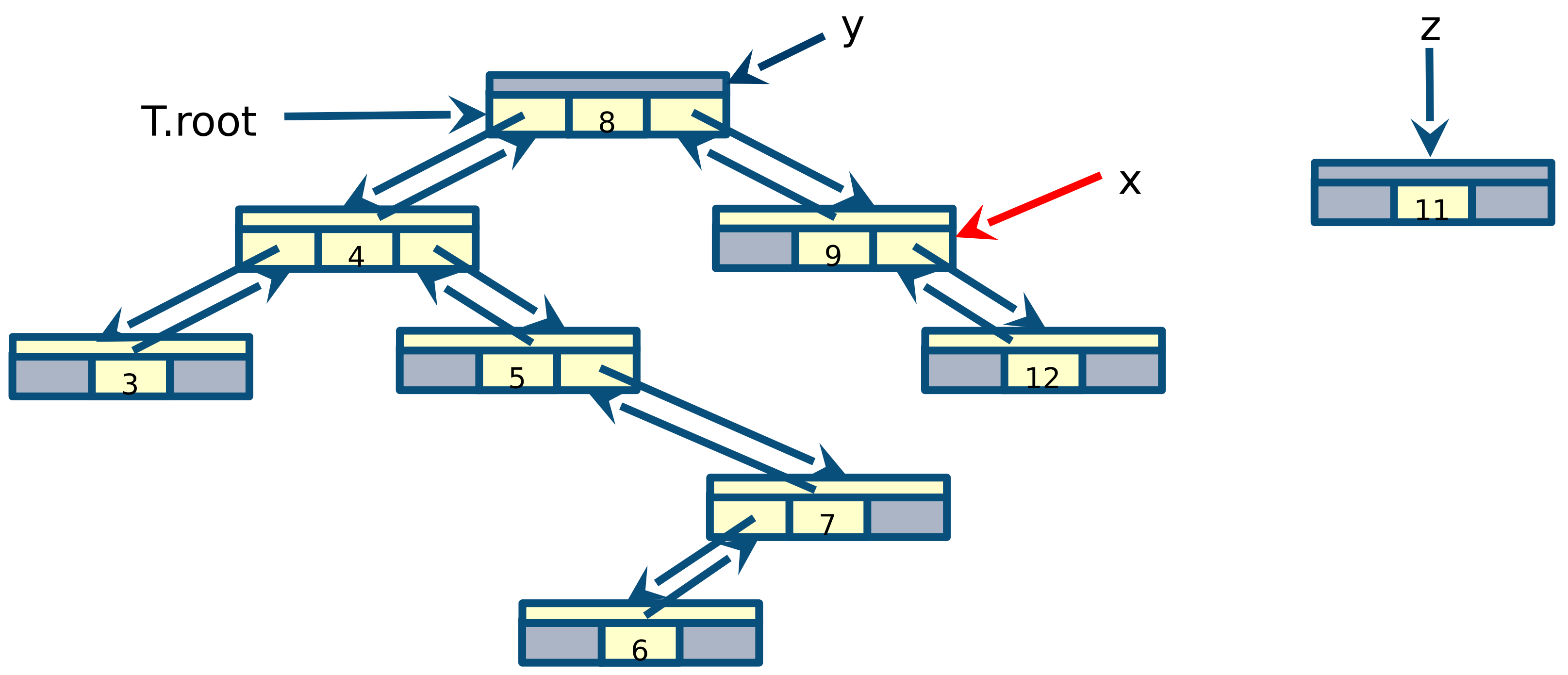
Example



```
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
  elseif z.key < y.key
    y.left := z
  else y.right := z
```

- INSERT(T,z) z.key = 11
- Initialise x and y and enter while loop

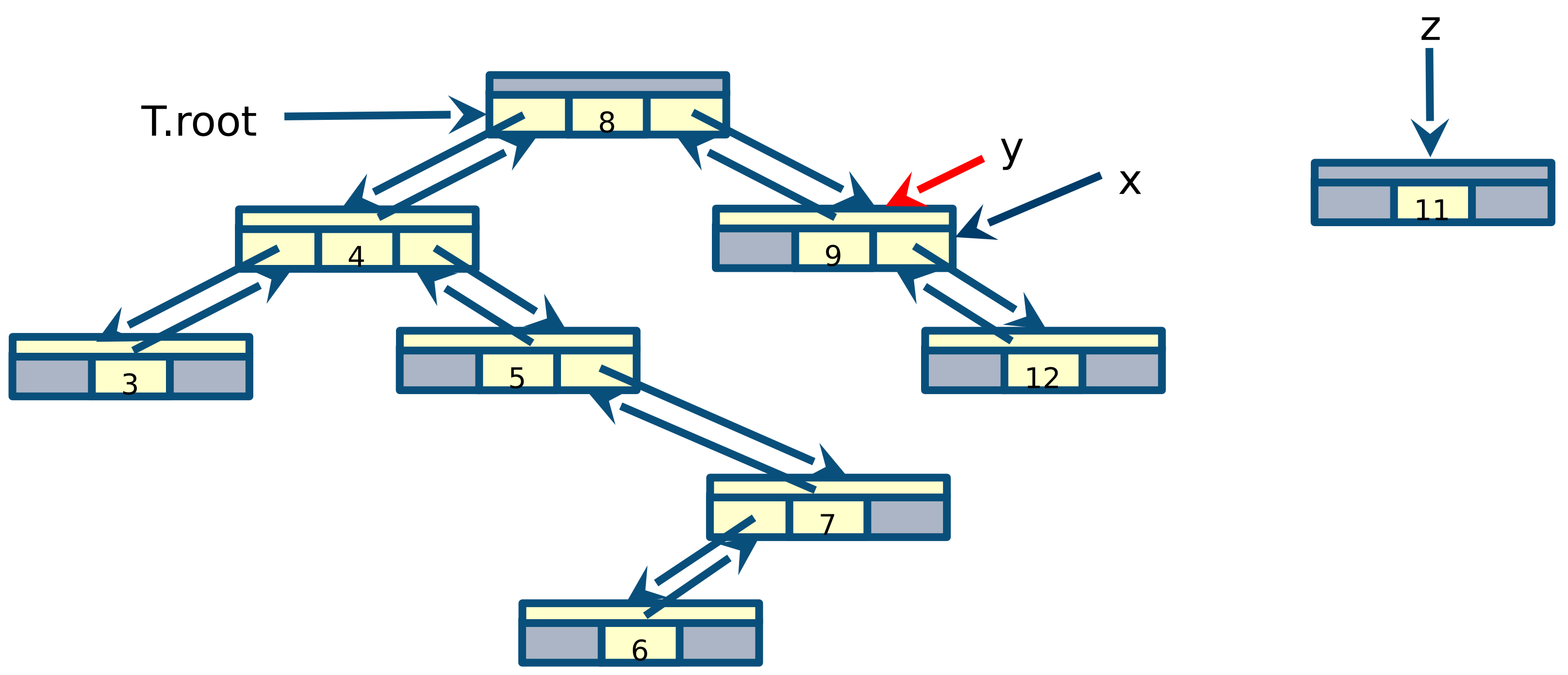
Example



```
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
  elseif z.key < y.key
    y.left := z
  else y.right := z
```

- INSERT(T,z) z.key = 11
- 11 > 8 go right and update x

Example

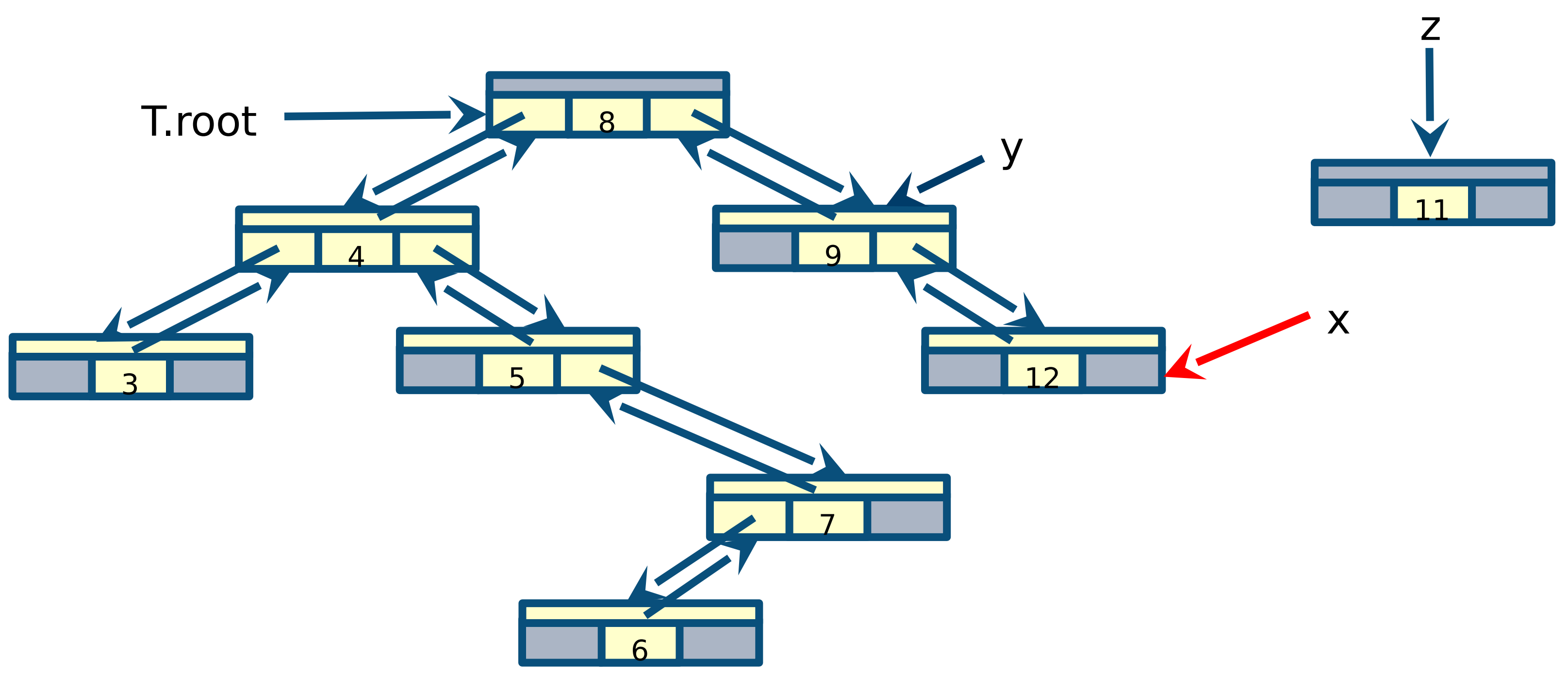


```
INSERT(T,z)
  y := NIL
  x := T.root
  while x != NIL
    y := x
    if z.key < x.key
      x := x.left
    else x := x.right
  z.p := y
  if y = NIL
    T.root := z
  elseif z.key < y.key
    y.left := z
  else y.right := z
```

– INSERT(T,z) z.key = 11

– Repeat while loop and update y

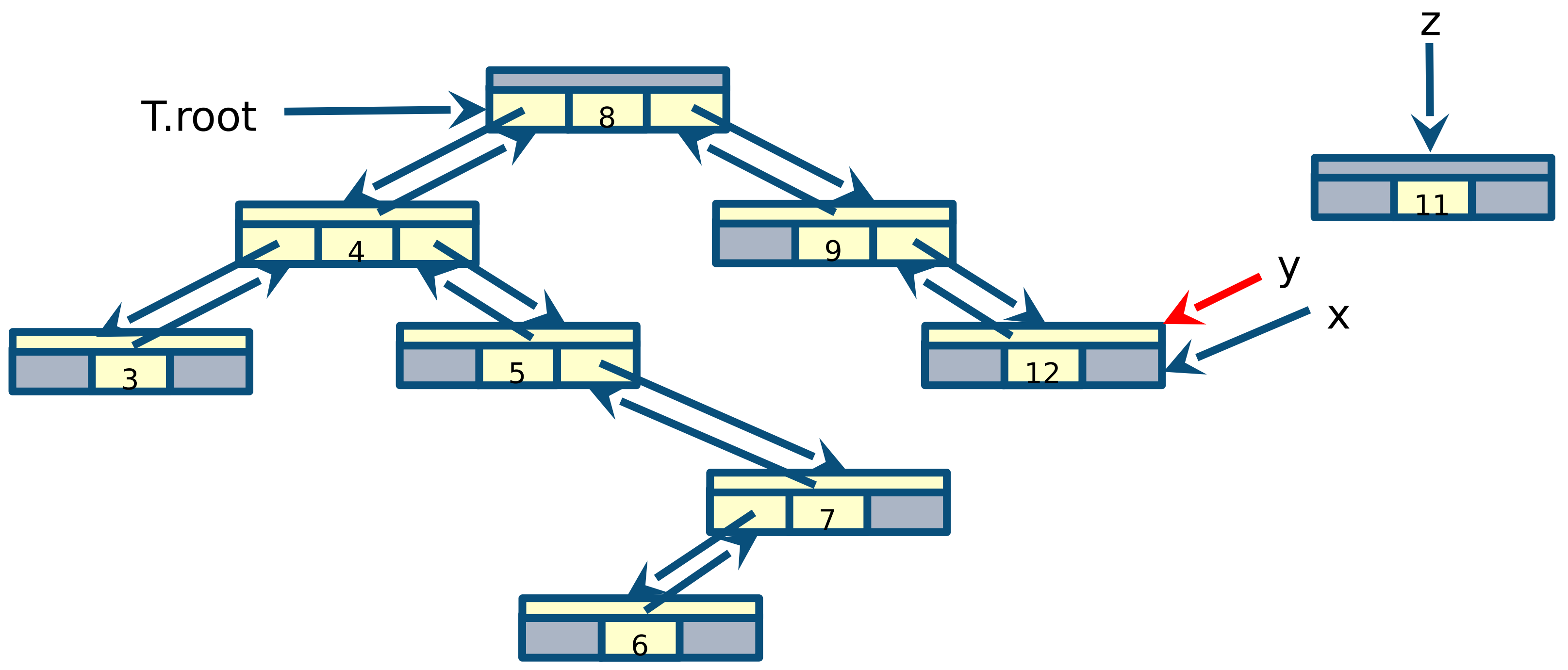
Example



```
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
  elseif z.key < y.key
    y.left := z
  else y.right := z
```

- INSERT(T,z) z.key = 11
- 11 > 9 go right and update x

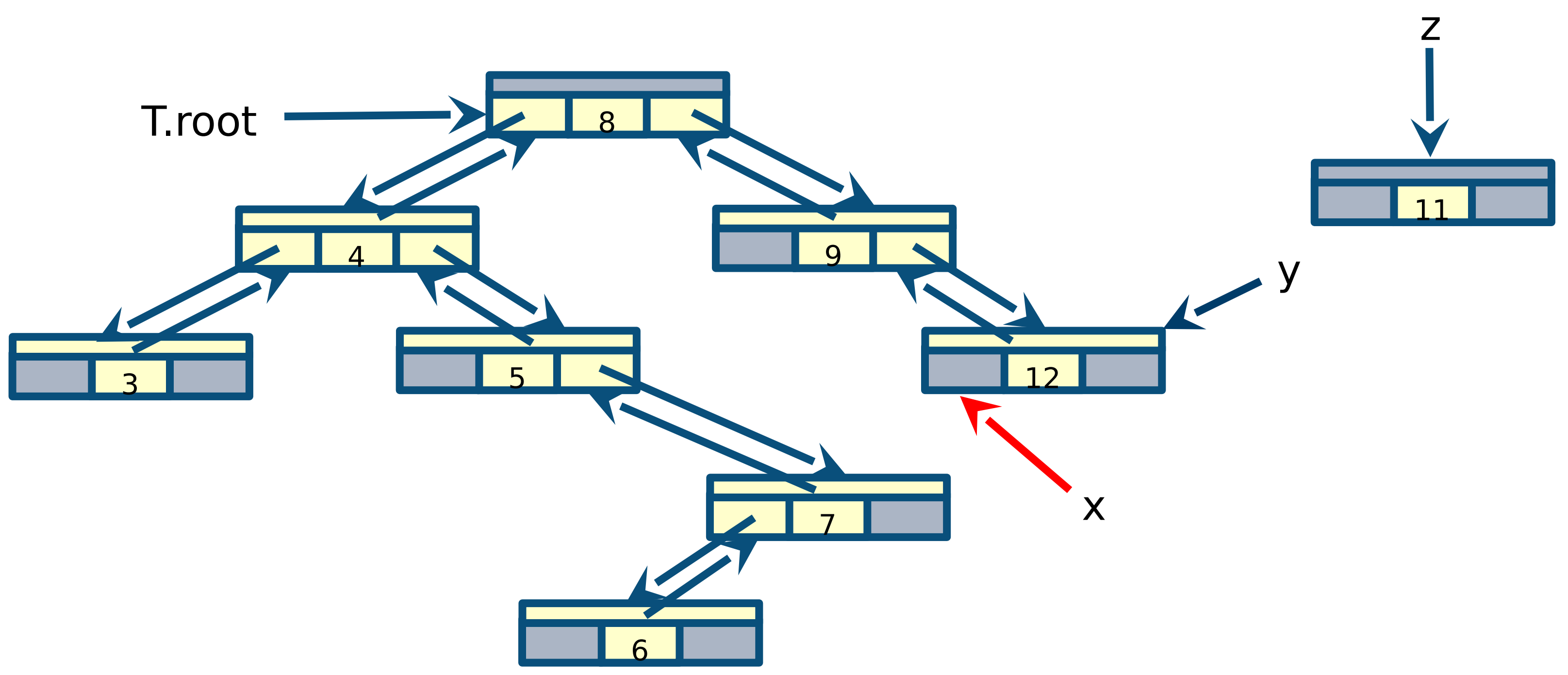
Example



```
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
  elseif z.key < y.key
    y.left := z
  else y.right := z
```

- INSERT(T,z) z.key = 11
- Repeat while loop and update y

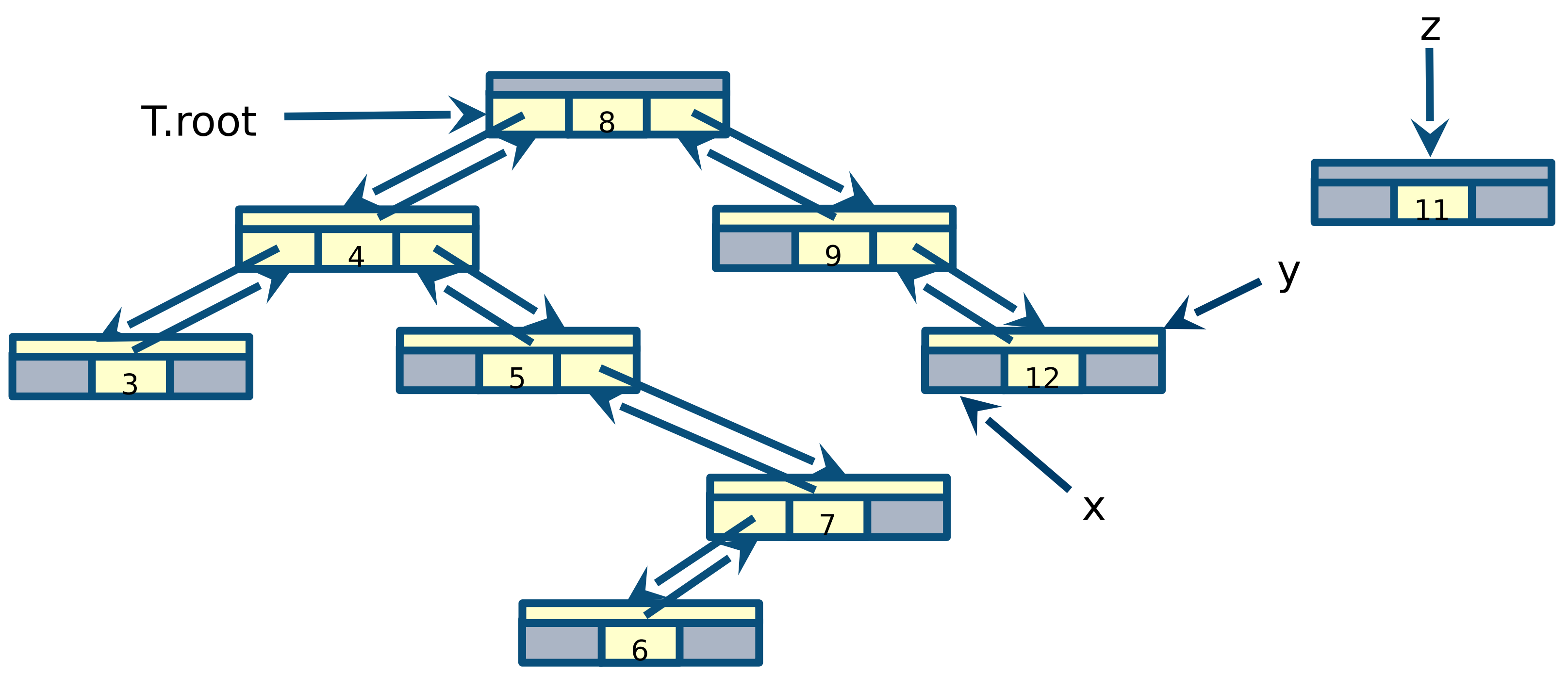
Example



```
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
  elseif z.key < y.key
    y.left := z
  else y.right := z
```

- INSERT(T,z) z.key = 11
- 11 < 12 go left and update x

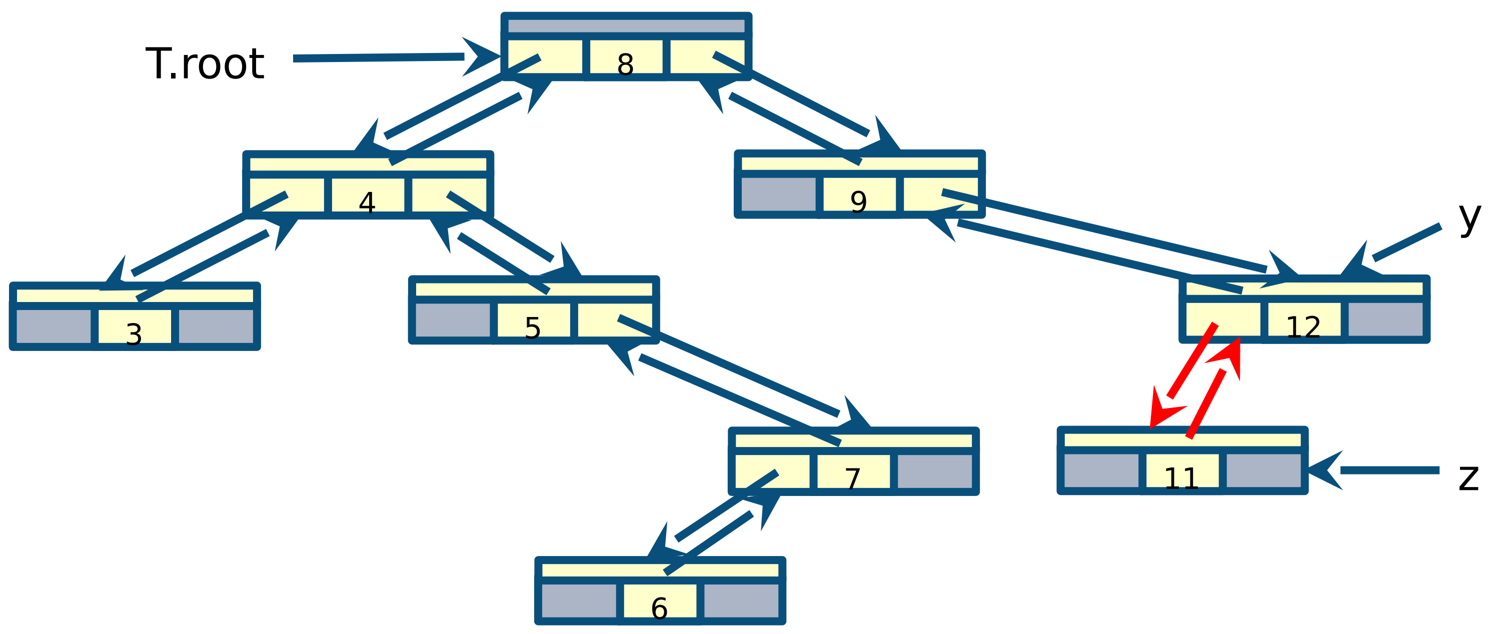
Example



```
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
  elseif z.key < y.key
    y.left := z
  else y.right := z
```

- INSERT(T,z) z.key = 11
- x = NIL end loop

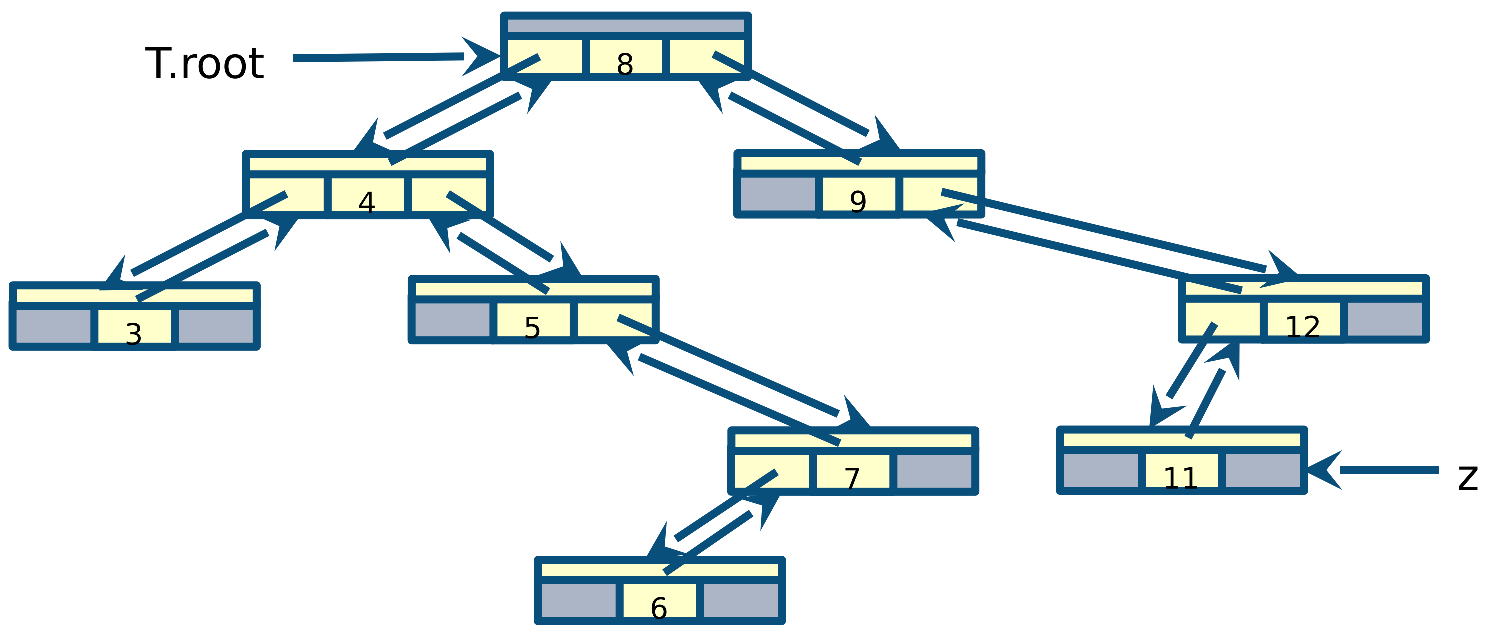
Example



```
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
  elseif z.key < y.key
    y.left := z
  else y.right := z
```

- INSERT(T,z) z.key = 11
- Update pointer attributes in nodes z and y

Example



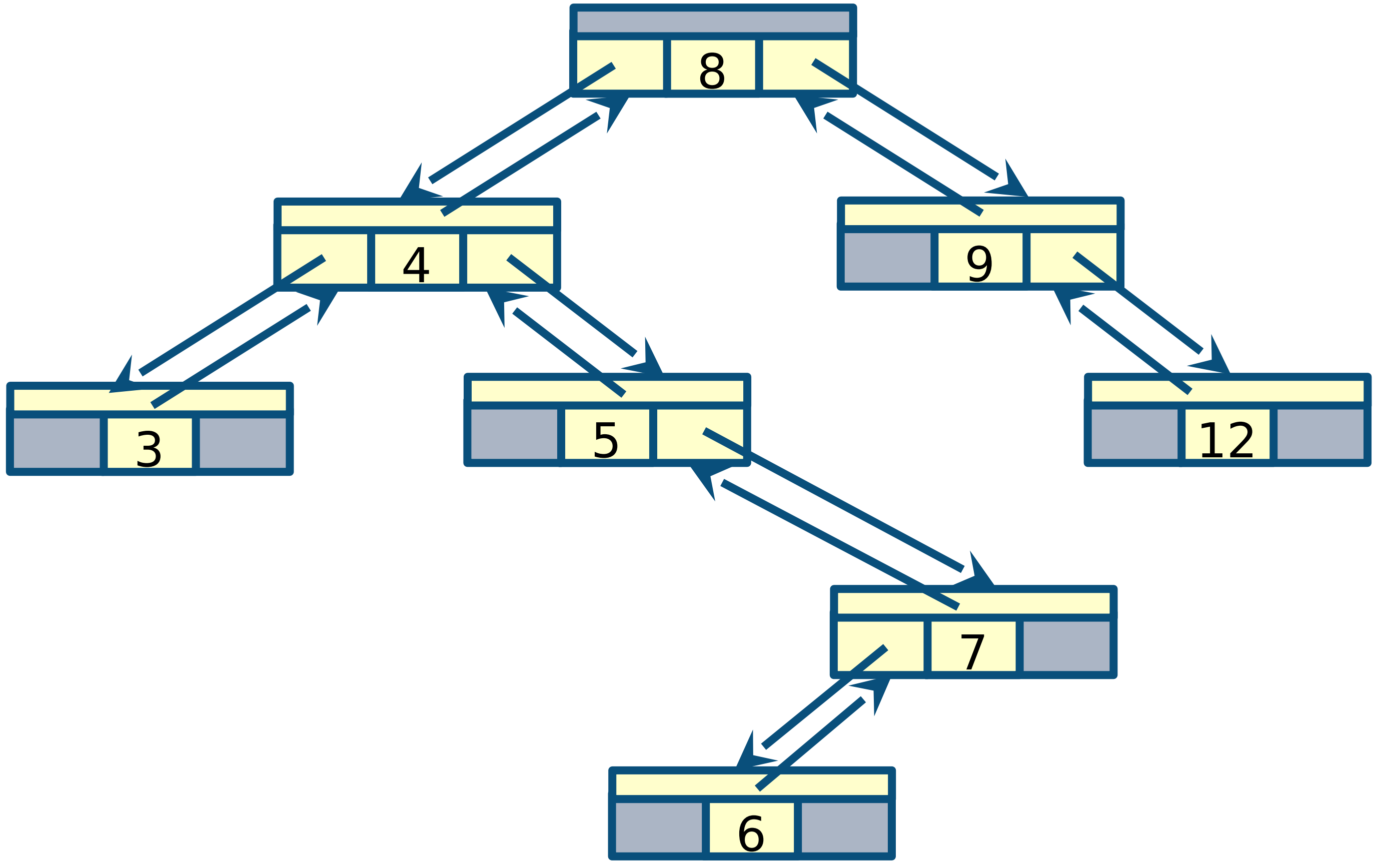
```
INSERT(T,z)
  y := NIL
  x := T.root
  while x != NIL
    y := x
    if z.key < x.key
      x := x.left
    else x := x.right
  z.p := y
  if y = NIL
    T.root := z
  elseif z.key < y.key
    y.left := z
  else y.right := z
```

- INSERT(T,z) z.key = 11
- Termination

Deletion

- Remove node z from BST T

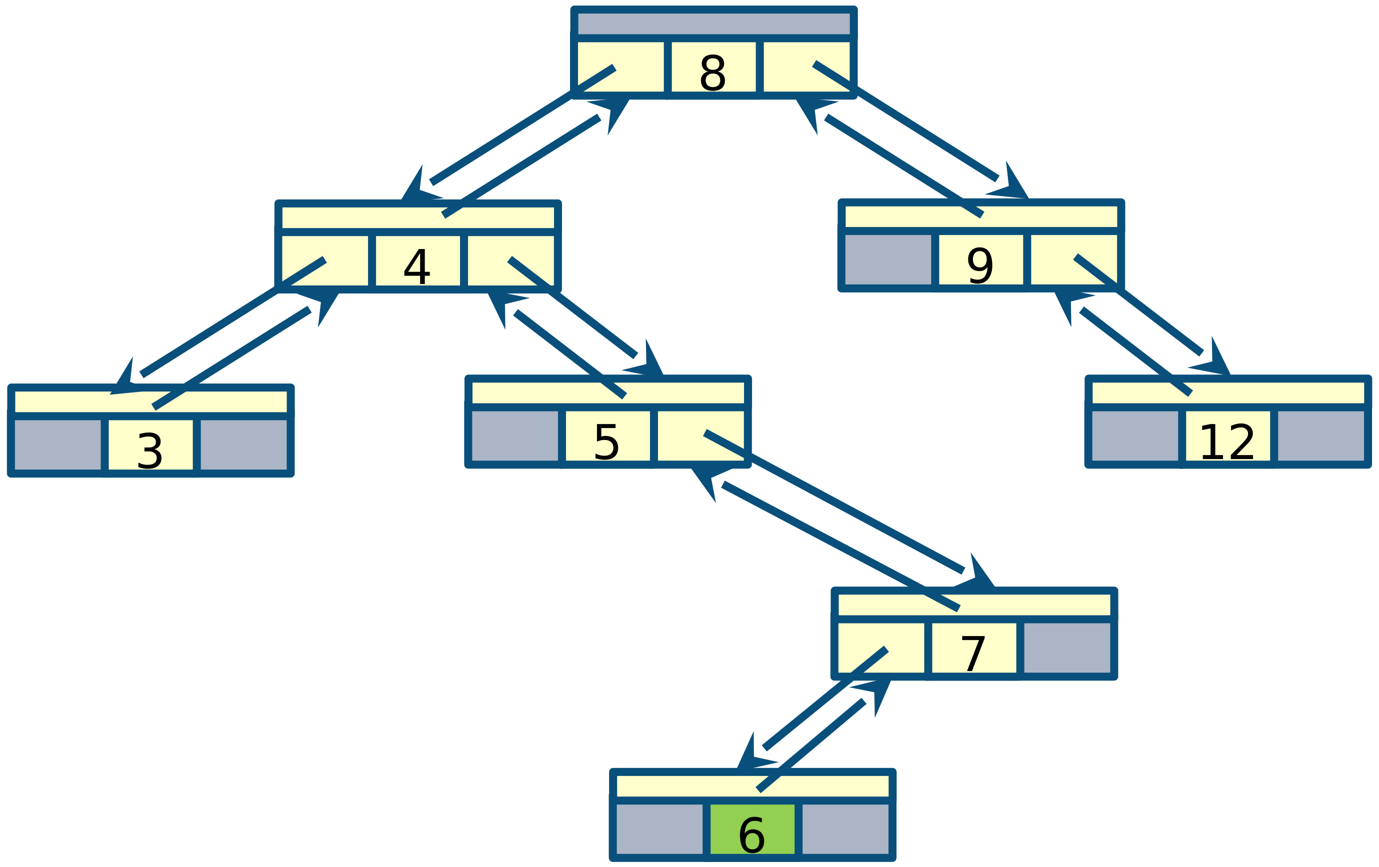
- More complicated than insertion as we need to consider several cases



Deletion

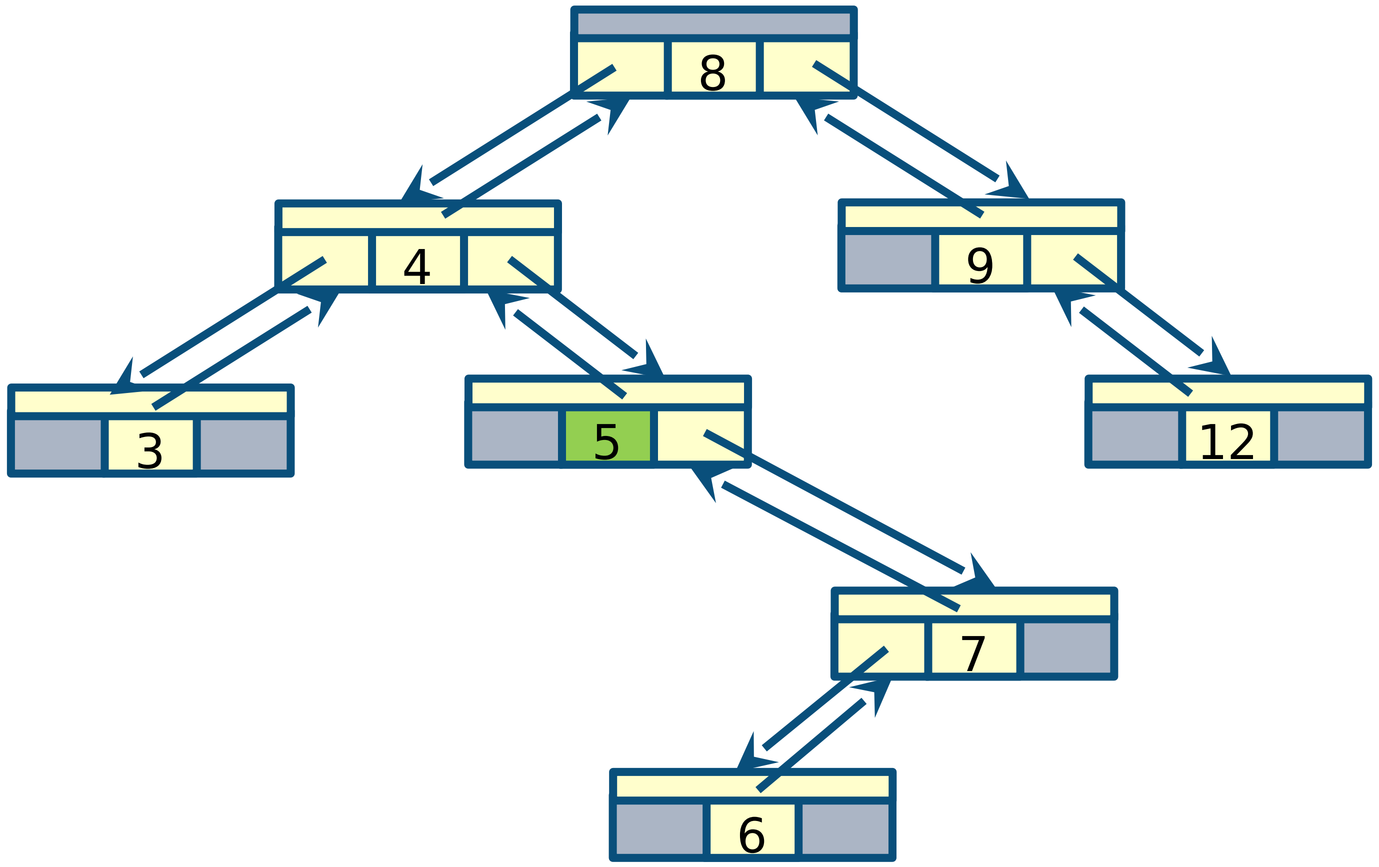
- **Remove node z from BST T**

- **More complicated than insertion as we need to consider several cases**
  - z is a leaf (easy)



Deletion

- **Remove node z from BST T**
- **More complicated than insertion as we need to consider several cases**
  - z is a leaf
  - z has one child (easy)

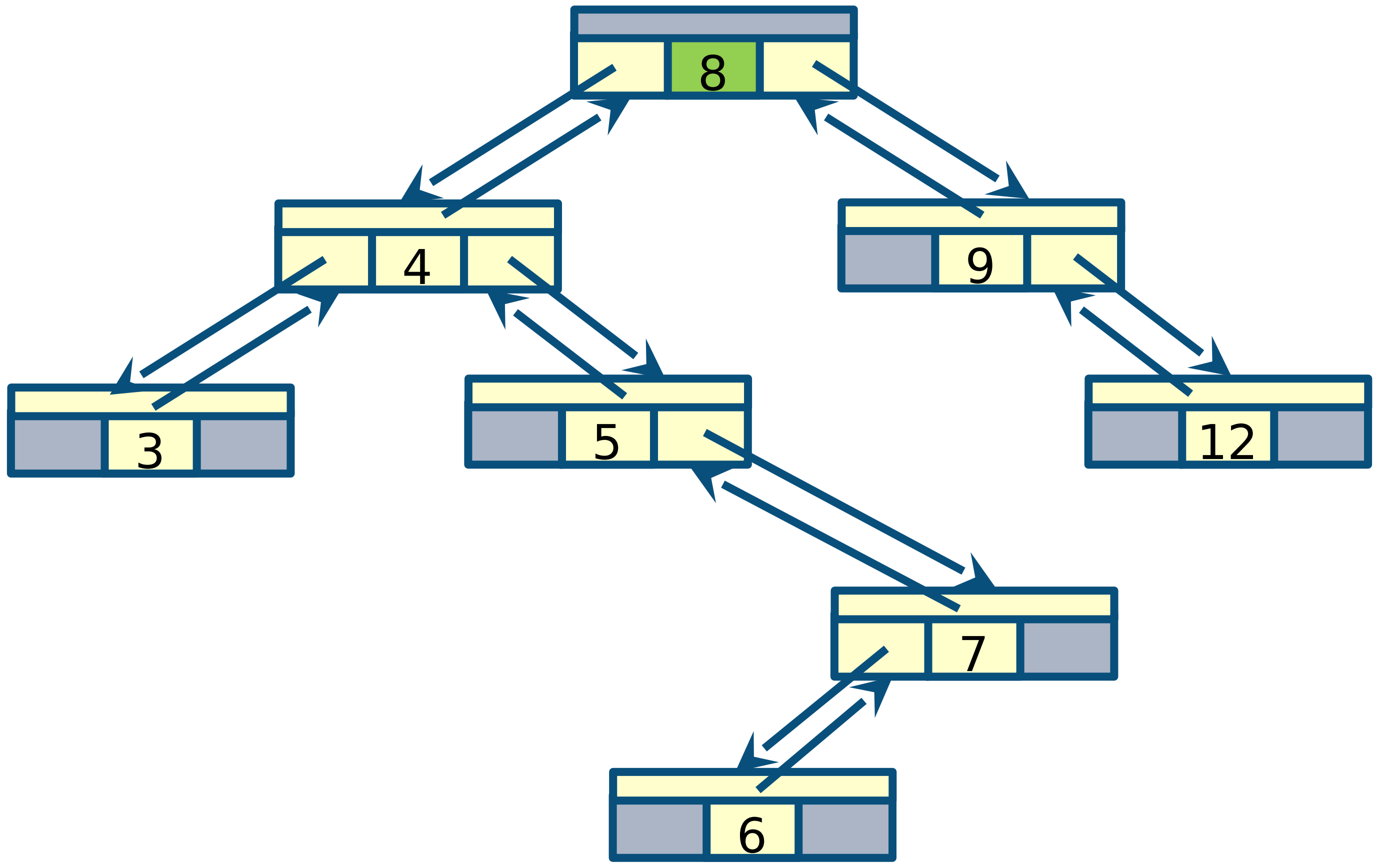


Deletion

- **Remove node z from BST T**

- **More complicated than insertion as we need to consider several cases**
  - z is a leaf
  - z has one child
  - z has two children (difficult)

- **It can be implemented in several different ways**



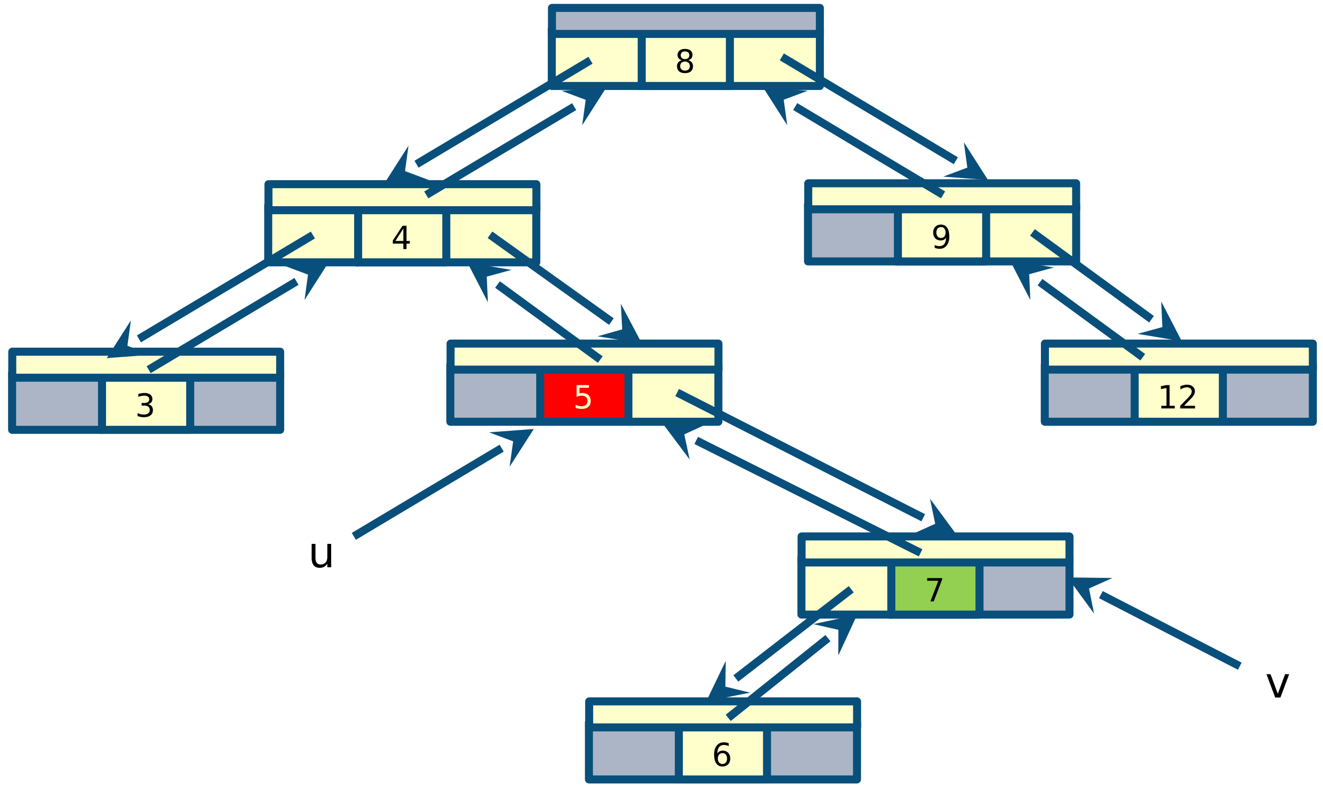
TRANSPLANT

- **We define an auxiliary function to move subtrees around**
  - Replace the subtree rooted at u with the subtree rooted at v
  - u's parent becomes v's parent

```
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
```

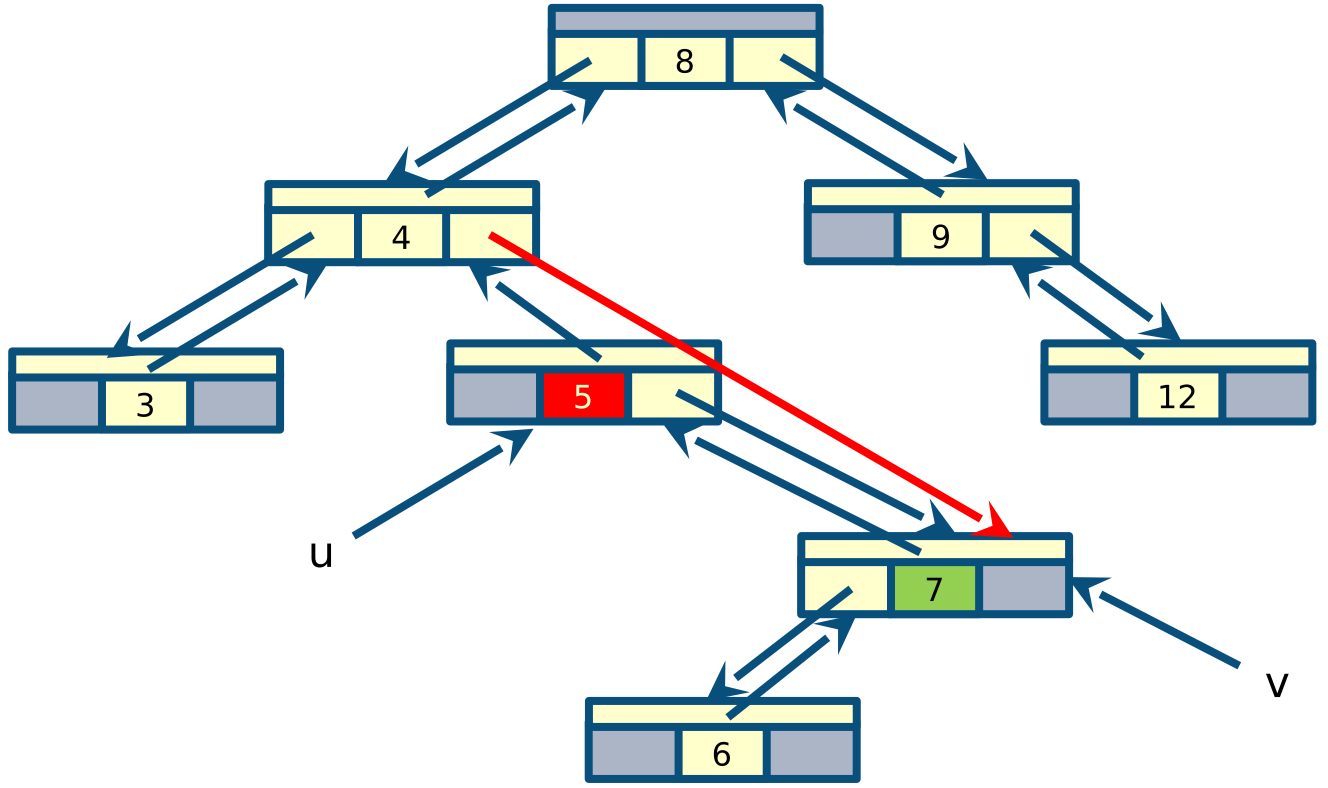
Example

```
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
```



Example

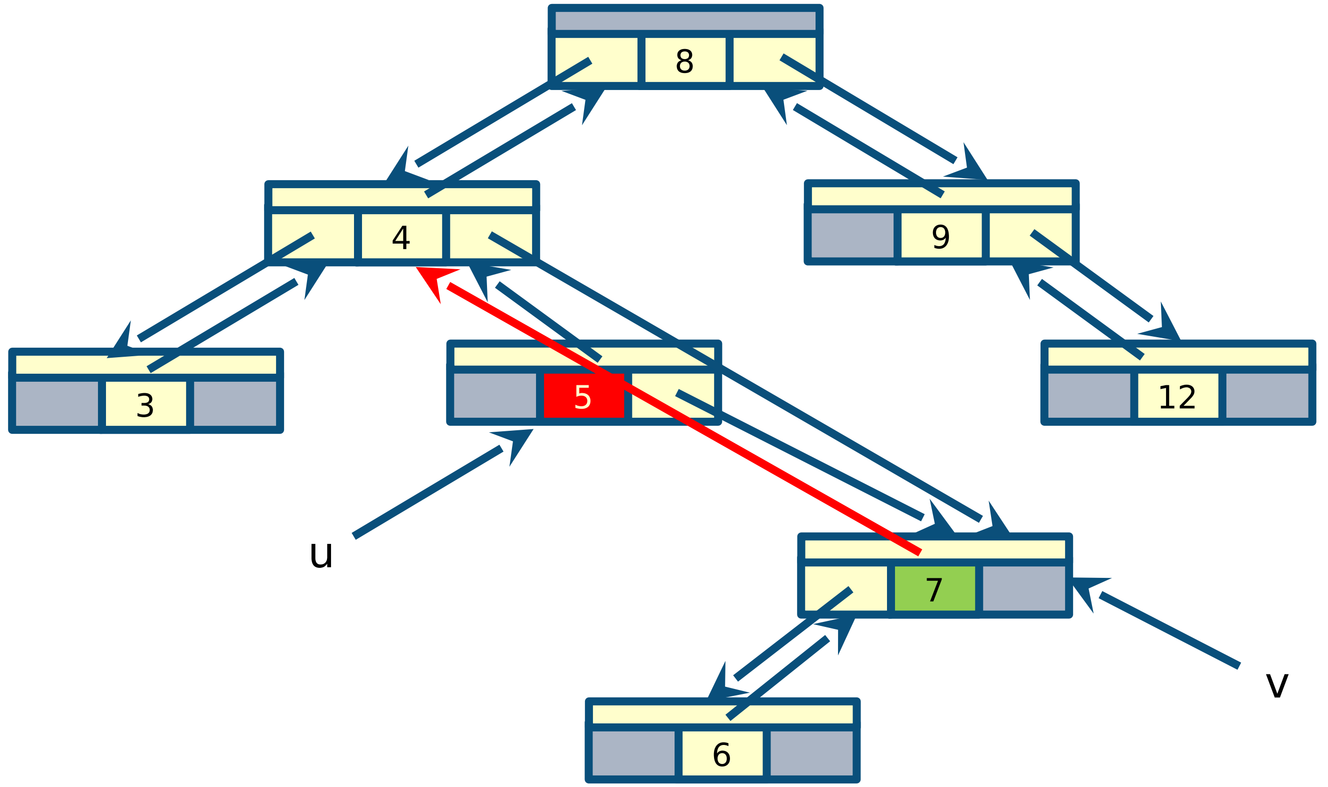
```
TRANSPLANT(T,u,v)
  if u.p = NIL
    T.root := v
  elseif u = u.p.left
    u.p.left := v
  else u.p.right := v
  if v != NIL
    v.p := u.p
```



– Update u.p.right

Example

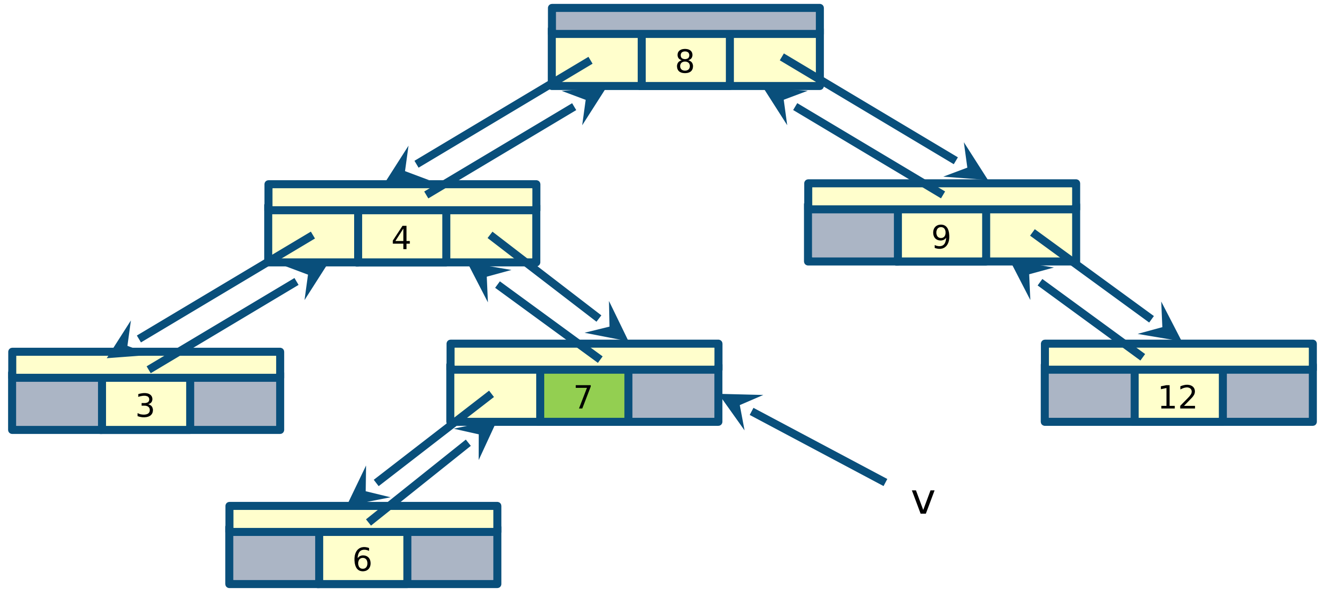
```
TRANSPLANT(T,u,v)
  if u.p = NIL
    T.root := v
  elseif u = u.p.left
    u.p.left := v
  else u.p.right := v
  if v != NIL
    v.p := u.p
```



- Update v.p

Example

```
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
```



– Termination

Deletion

- **Delete node z from tree T**

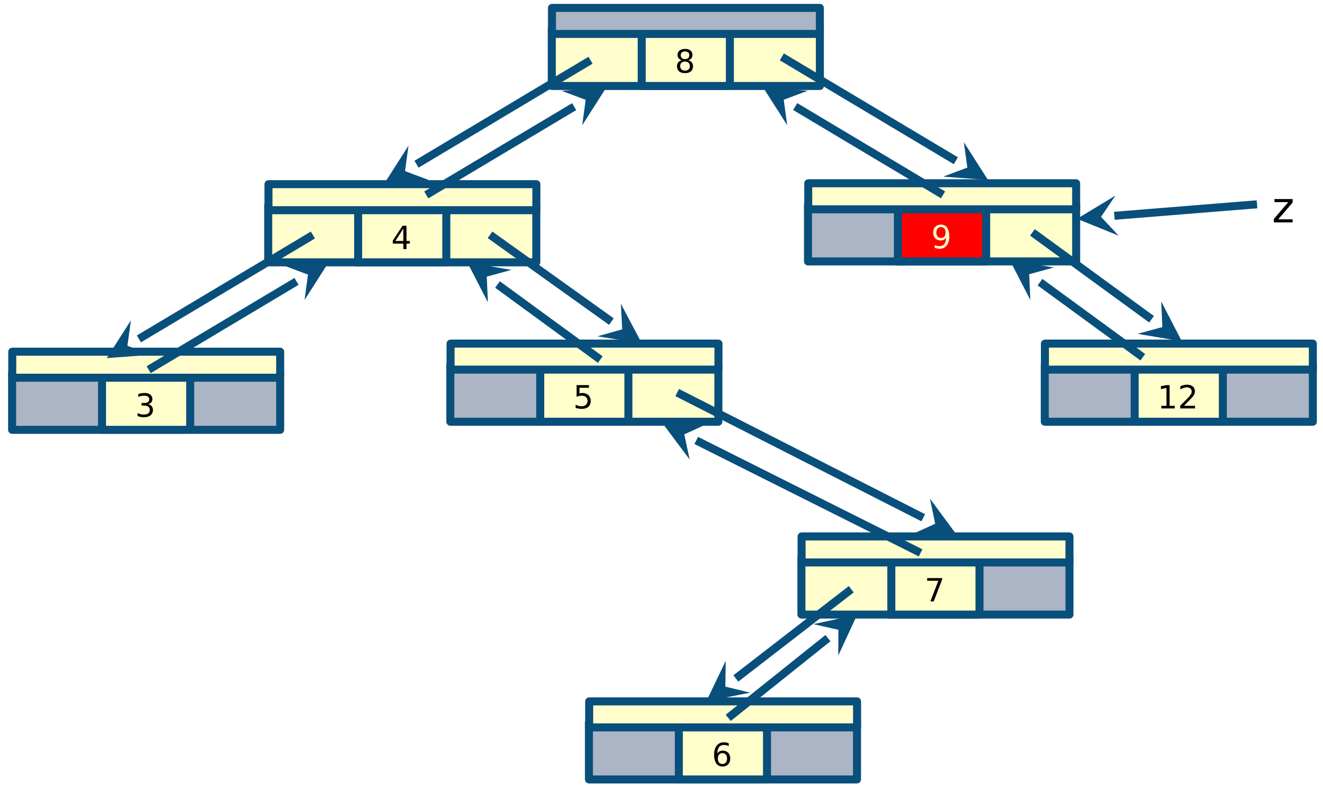
- **We specialise the three cases further**
  1. z has no left child (can be a leaf)
  2. z has a left child but no right child
  3. z has two children and its successor y is its right child
  4. z has two children and its successor y is not its right child

- **Runs in O(h) on a tree of height h**
  - Cost of calling MINIMUM to find y

```
DELETE(T,z)
  if z.left = NIL
    TRANSPLANT(T,z,z.right)
  elseif z.right = NIL
    TRANSPLANT(T,z,z.left)
  else y = 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
```

Example (case 1)

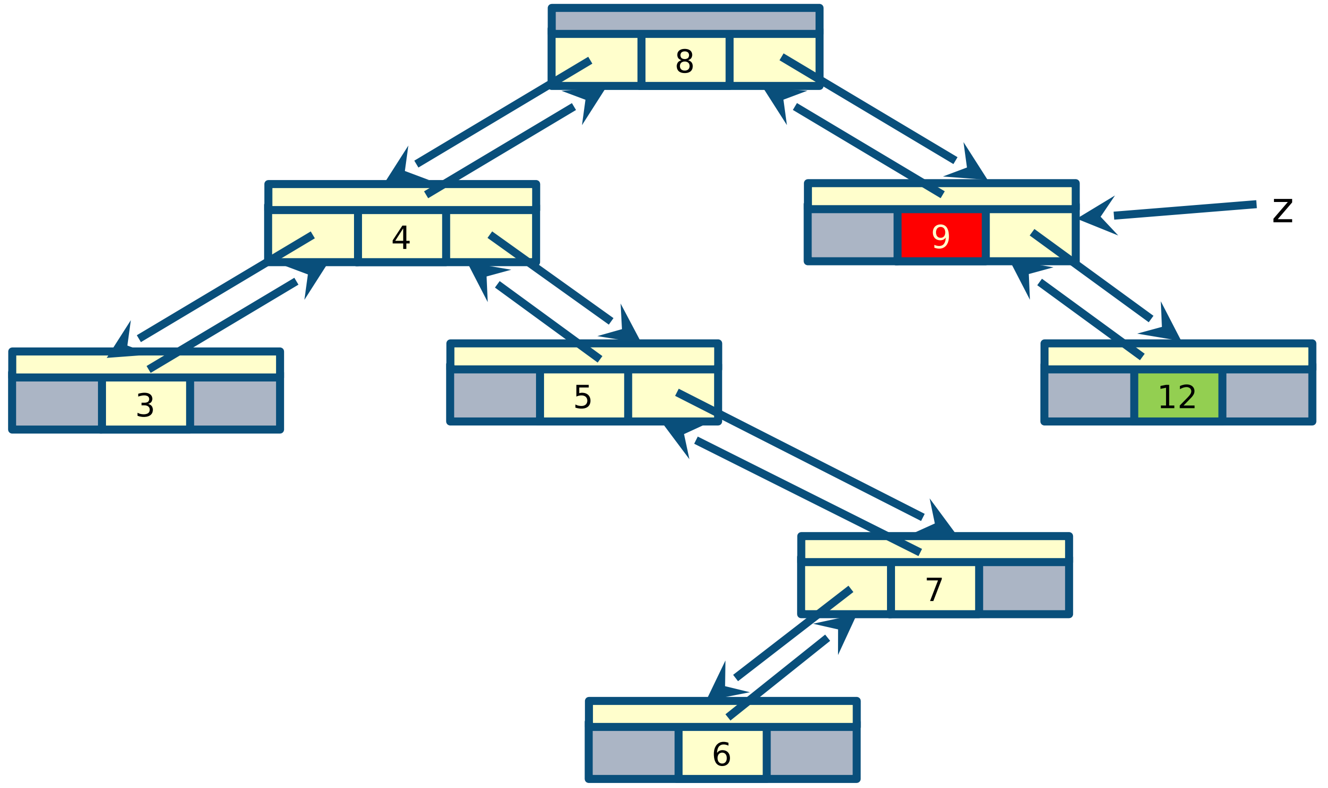
```
DELETE(T,z)
  if z.left = NIL
    TRANSPLANT(T,z,z.right)
  elseif z.right = NIL
    TRANSPLANT(T,z,z.left)
  else y = MINIMUM(z.right)
    if y.p != z
      TRANSPLANT(T,y,y.right)
      y.right := z.right
      y.right.p := y
    TRANSPLANT(T,z,y)
    y.left := z.left
    y.left.p := y
```



- DELETE(T,z) z.key = 9

Example (case 1)

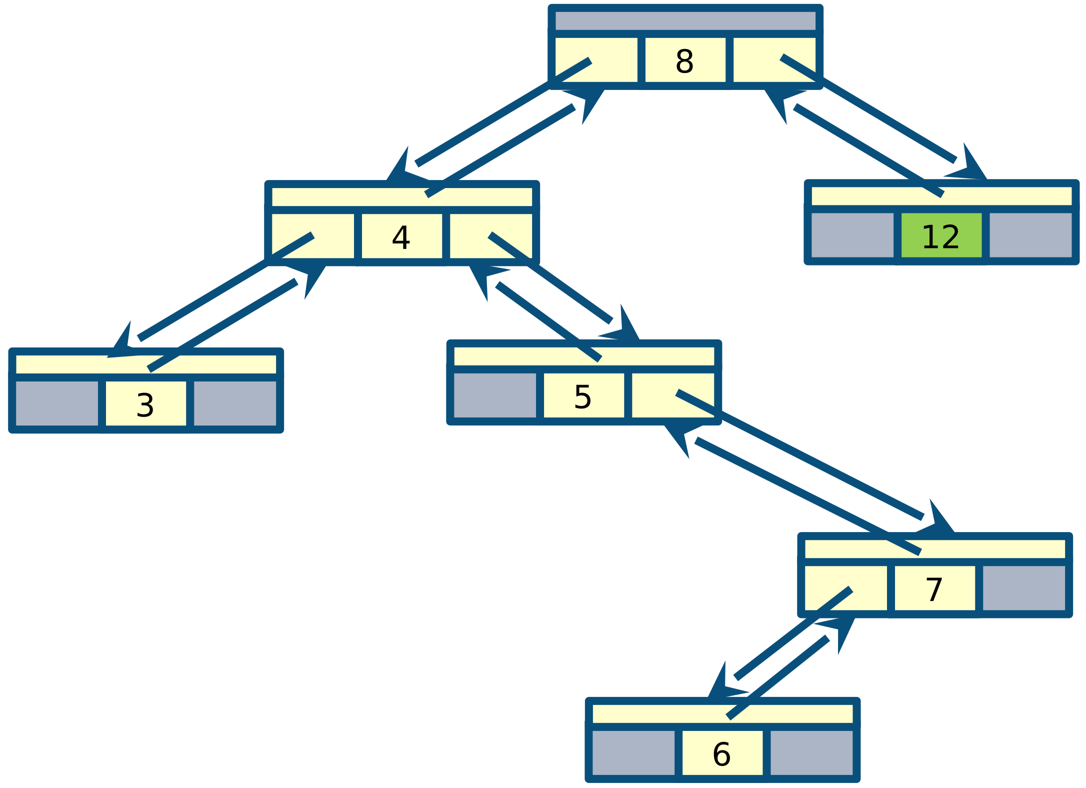
```
DELETE(T,z)
  if z.left = NIL
    TRANSPLANT(T,z,z.right)
  elseif z.right = NIL
    TRANSPLANT(T,z,z.left)
  else y = MINIMUM(z.right)
    if y.p != z
      TRANSPLANT(T,y,y.right)
      y.right := z.right
      y.right.p := y
    TRANSPLANT(T,z,y)
    y.left := z.left
    y.left.p := y
```



- DELETE(T,z) z.key = 9
- Call TRANSPLANT on z.right

Example (case 1)

```
DELETE(T,z)
  if z.left = NIL
    TRANSPLANT(T,z,z.right)
  elseif z.right = NIL
    TRANSPLANT(T,z,z.left)
  else y = 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
```

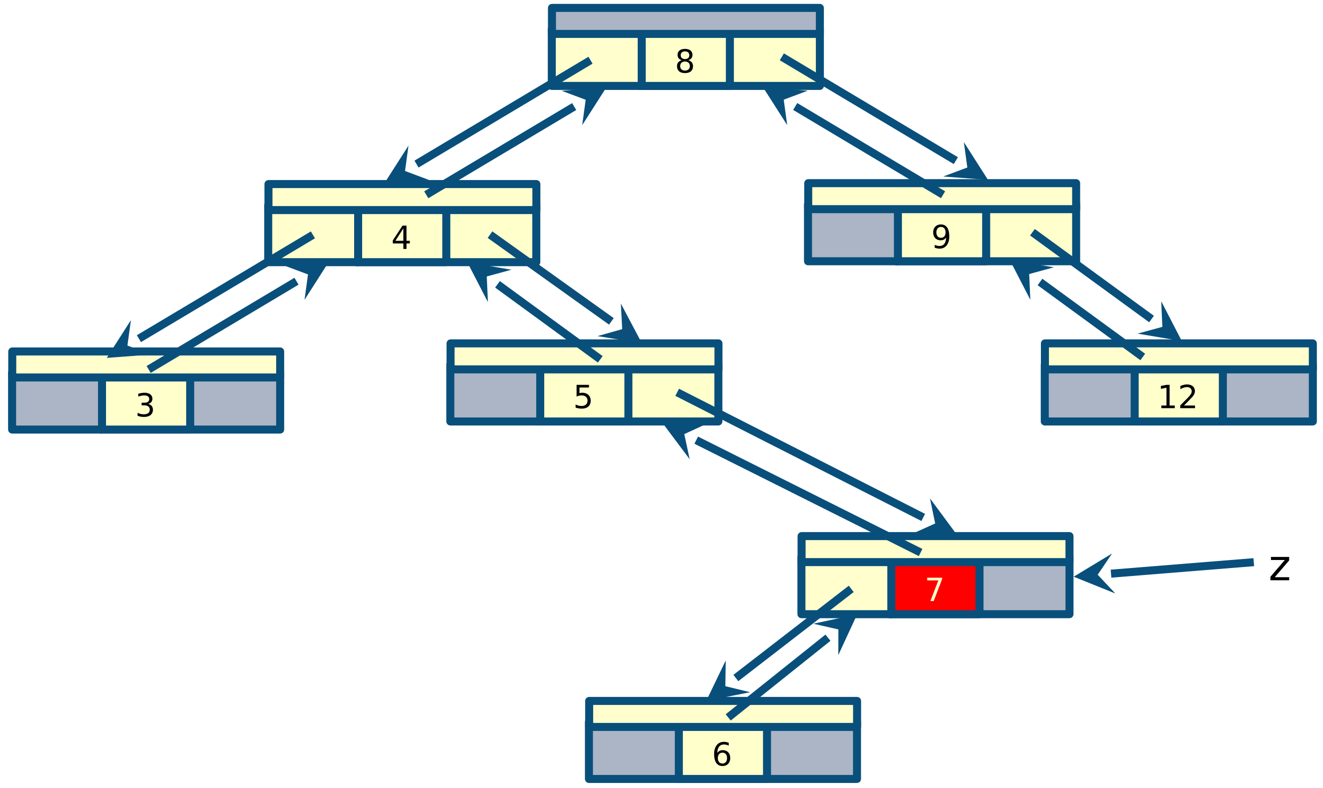


- DELETE(T,z) z.key = 9
- Termination

Example (case 2)

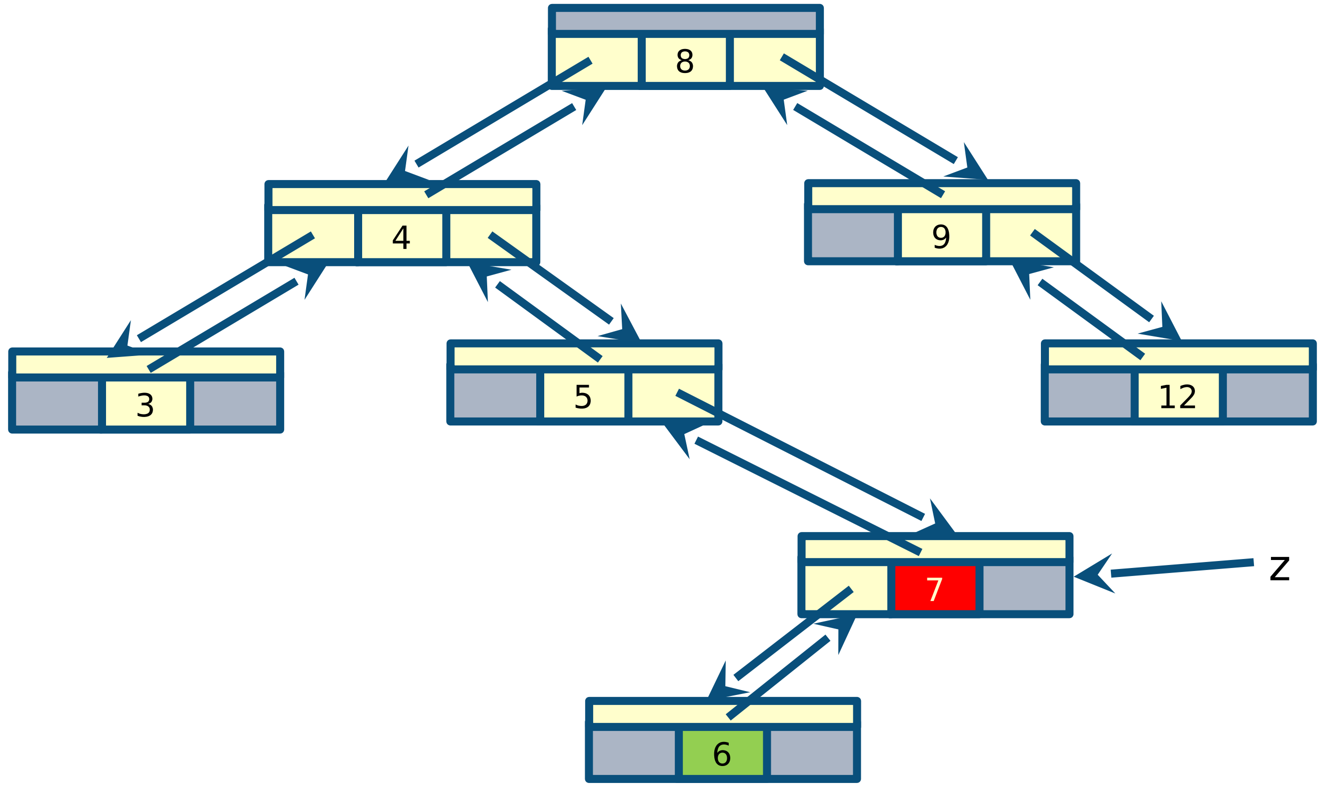
```
DELETE(T,z)
  if z.left = NIL
    TRANSPLANT(T,z,z.right)
  elseif z.right = NIL
    TRANSPLANT(T,z,z.left)
  else y = MINIMUM(z.right)
    if y.p != z
      TRANSPLANT(T,y,y.right)
      y.right := z.right
      y.right.p := y
    TRANSPLANT(T,z,y)
    y.left := z.left
    y.left.p := y
```

– DELETE(T,z) z.key = 7



Example (case 2)

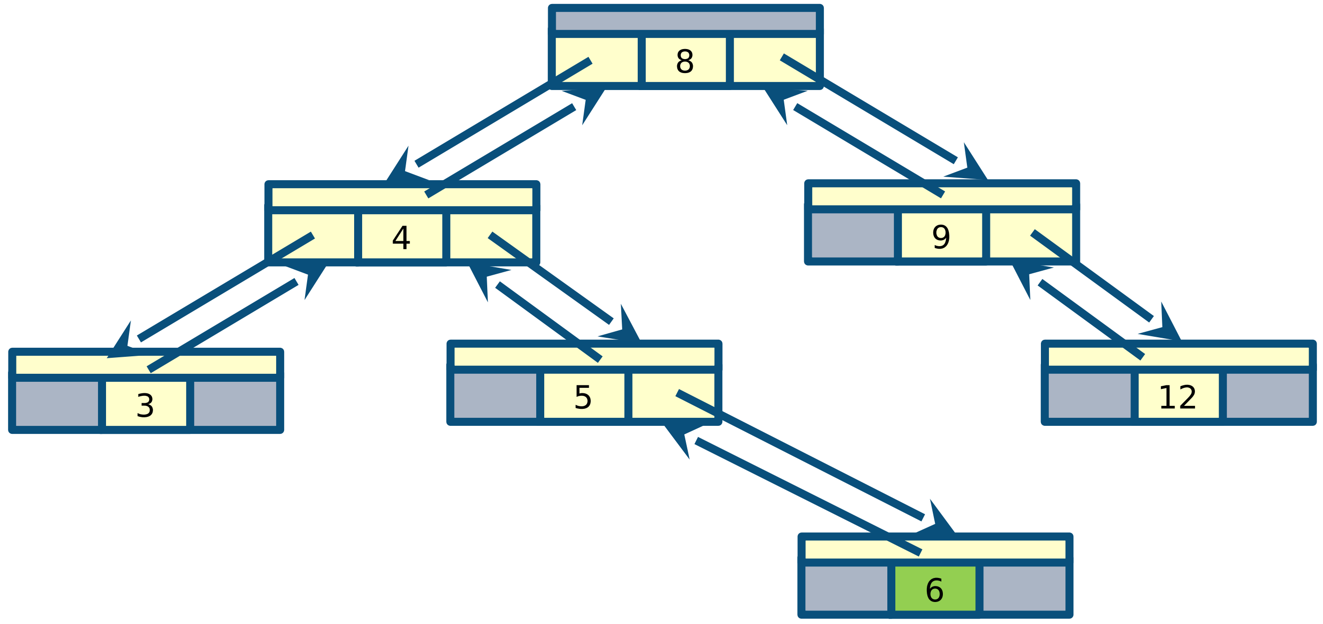
```
DELETE(T,z)
  if z.left = NIL
    TRANSPLANT(T,z,z.right)
  elseif z.right = NIL
    TRANSPLANT(T,z,z.left)
  else y = MINIMUM(z.right)
    if y.p != z
      TRANSPLANT(T,y,y.right)
      y.right := z.right
      y.right.p := y
    TRANSPLANT(T,z,y)
    y.left := z.left
    y.left.p := y
```



- DELETE(T,z) z.key = 7
- Call TRANSPLANT on z.left

Example (case 2)

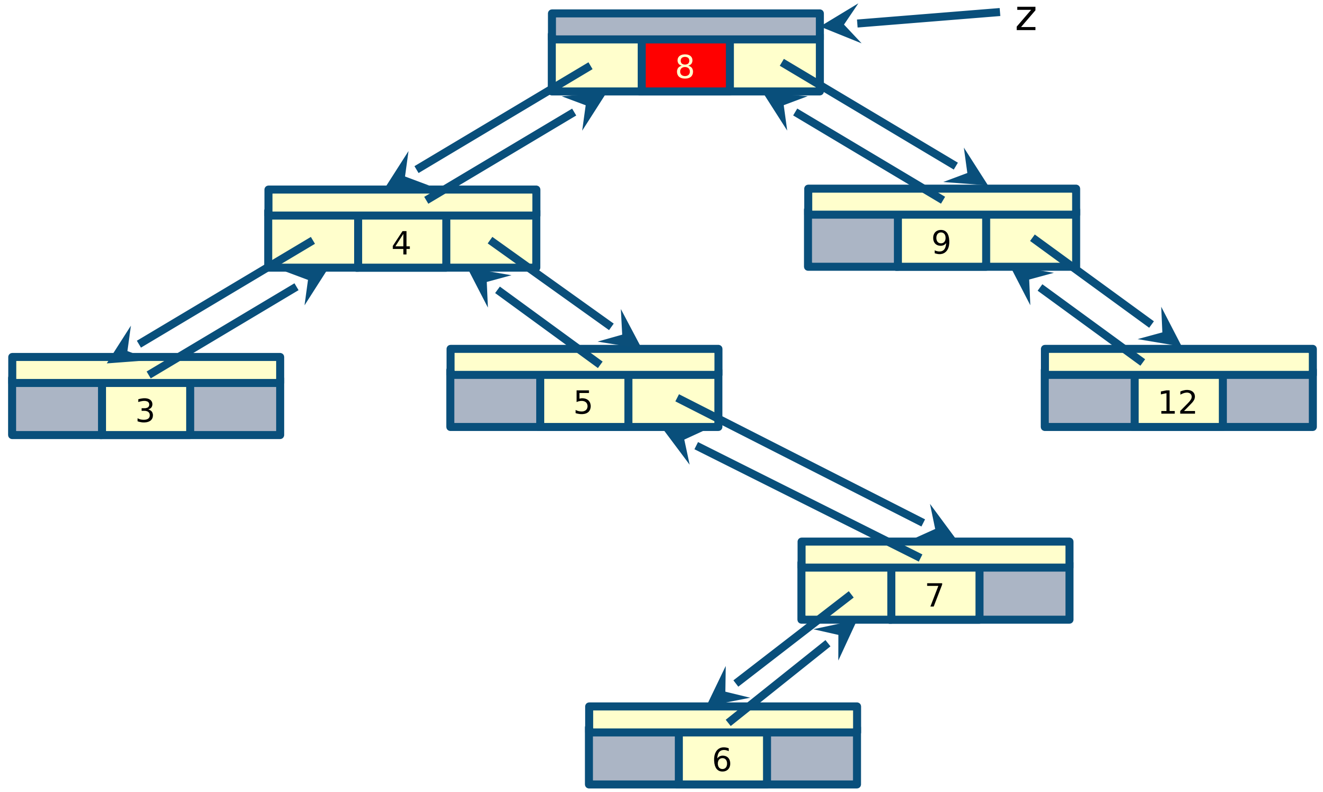
```
DELETE(T,z)
  if z.left = NIL
    TRANSPLANT(T,z,z.right)
  elseif z.right = NIL
    TRANSPLANT(T,z,z.left)
  else y = MINIMUM(z.right)
    if y.p != z
      TRANSPLANT(T,y,y.right)
      y.right := z.right
      y.right.p := y
    TRANSPLANT(T,z,y)
    y.left := z.left
    y.left.p := y
```



- DELETE(T,z) z.key = 7
- Termination

Example (case 3)

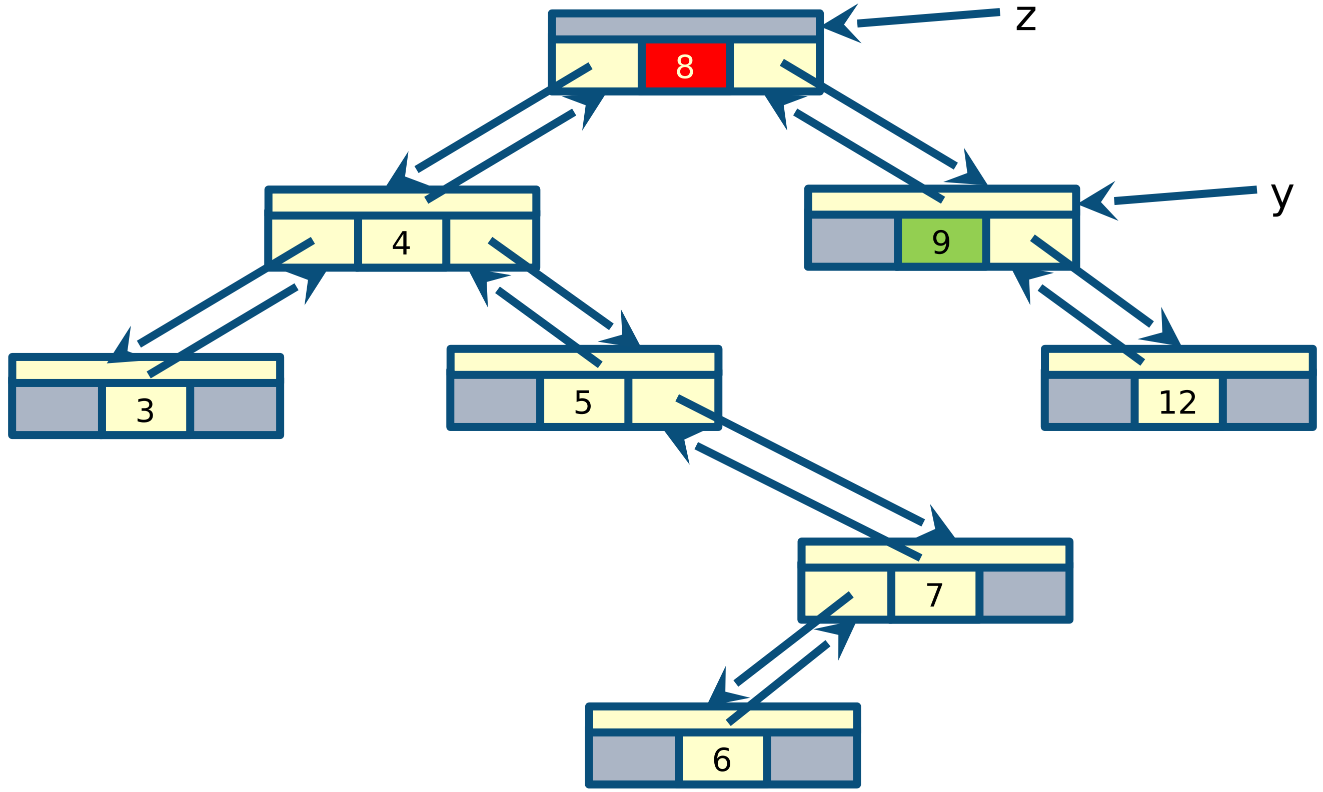
```
DELETE(T,z)
  if z.left = NIL
    TRANSPLANT(T,z,z.right)
  elseif z.right = NIL
    TRANSPLANT(T,z,z.left)
  else y = MINIMUM(z.right)
    if y.p != z
      TRANSPLANT(T,y,y.right)
      y.right := z.right
      y.right.p := y
    TRANSPLANT(T,z,y)
    y.left := z.left
    y.left.p := y
```



– DELETE(T,z) z.key = 8

Example (case 3)

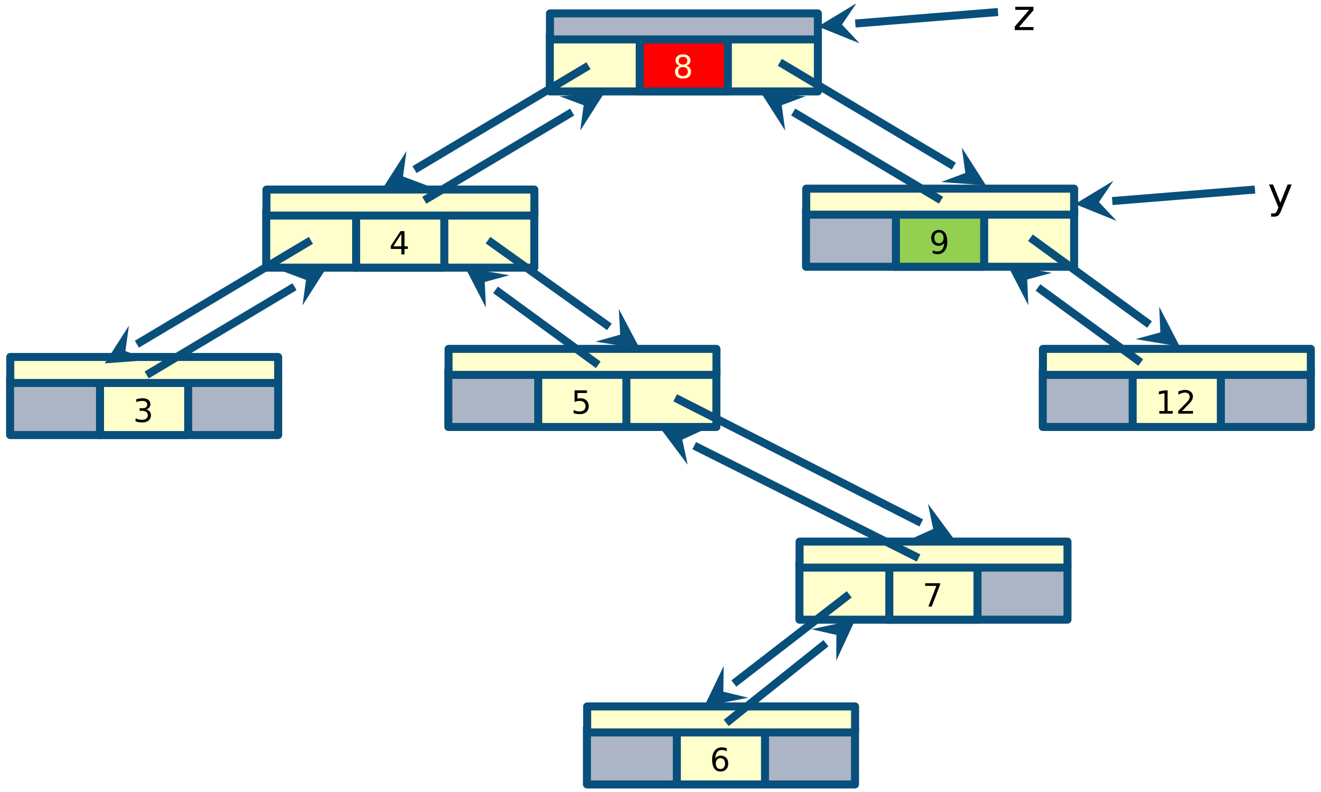
```
DELETE(T,z)
  if z.left = NIL
    TRANSPLANT(T,z,z.right)
  elseif z.right = NIL
    TRANSPLANT(T,z,z.left)
  else y = MINIMUM(z.right)
    if y.p != z
      TRANSPLANT(T,y,y.right)
      y.right := z.right
      y.right.p := y
    TRANSPLANT(T,z,y)
    y.left := z.left
    y.left.p := y
```



- DELETE(T,z) z.key = 8
- Call MINIMUM to find y (successor of z)

Example (case 3)

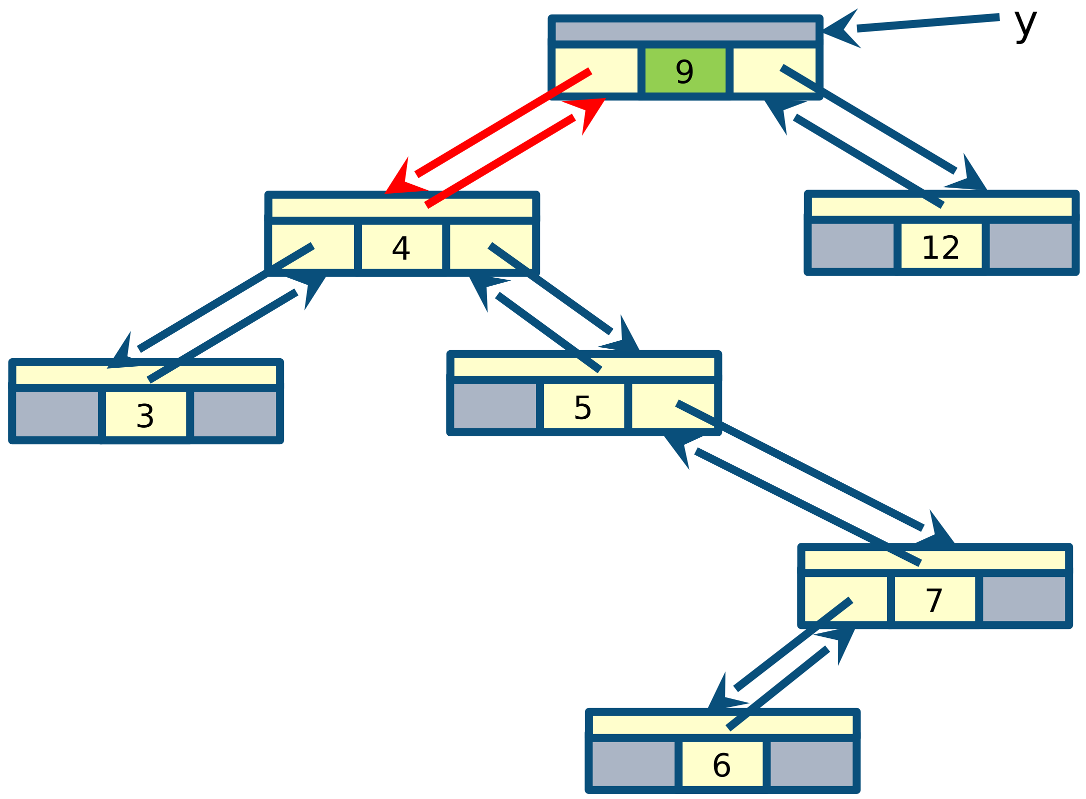
```
DELETE(T,z)
  if z.left = NIL
    TRANSPLANT(T,z,z.right)
  elseif z.right = NIL
    TRANSPLANT(T,z,z.left)
  else y = MINIMUM(z.right)
    if y.p != z
      TRANSPLANT(T,y,y.right)
      y.right := z.right
      y.right.p := y
    TRANSPLANT(T,z,y)
    y.left := z.left
    y.left.p := y
```



- DELETE(T,z) z.key = 8
- z is the parent of y so call TRANPLANT on y

Example (case 3)

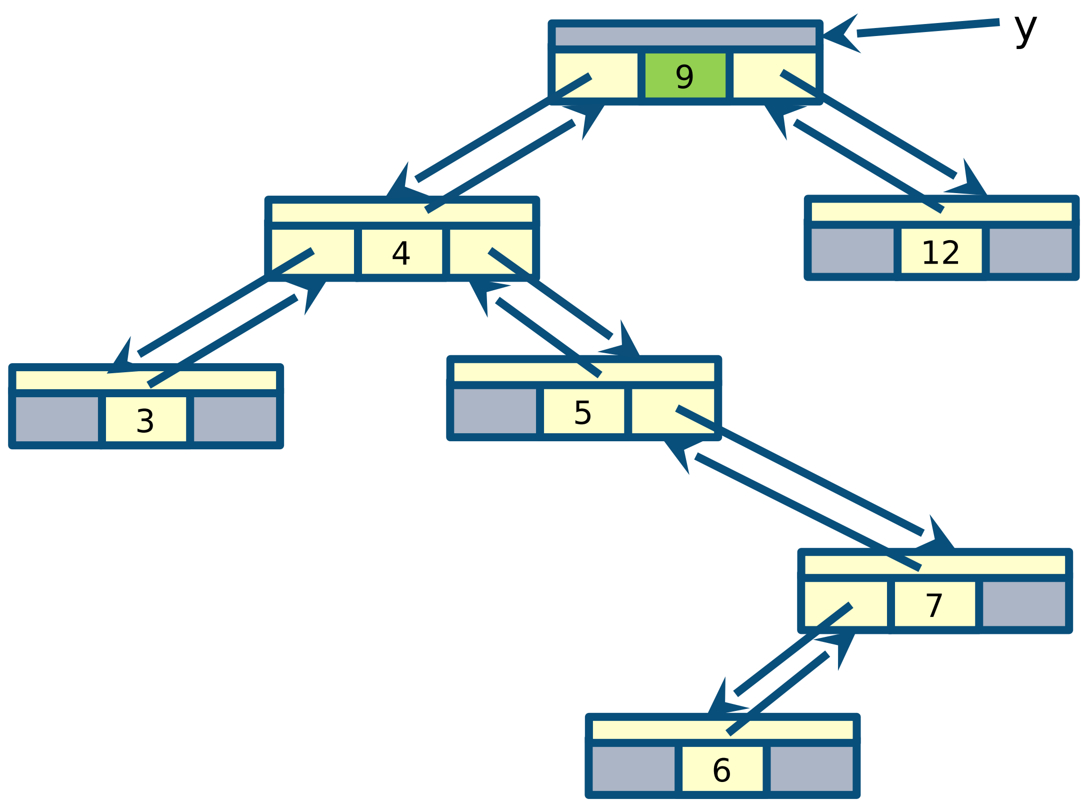
```
DELETE(T,z)
  if z.left = NIL
    TRANSPLANT(T,z,z.right)
  elseif z.right = NIL
    TRANSPLANT(T,z,z.left)
  else y = MINIMUM(z.right)
    if y.p != z
      TRANSPLANT(T,y,y.right)
      y.right := z.right
      y.right.p := y
    TRANSPLANT(T,z,y)
    y.left := z.left
    y.left.p := y
```



– DELETE(T,z) z.key = 8
– Update pointers

Example (case 3)

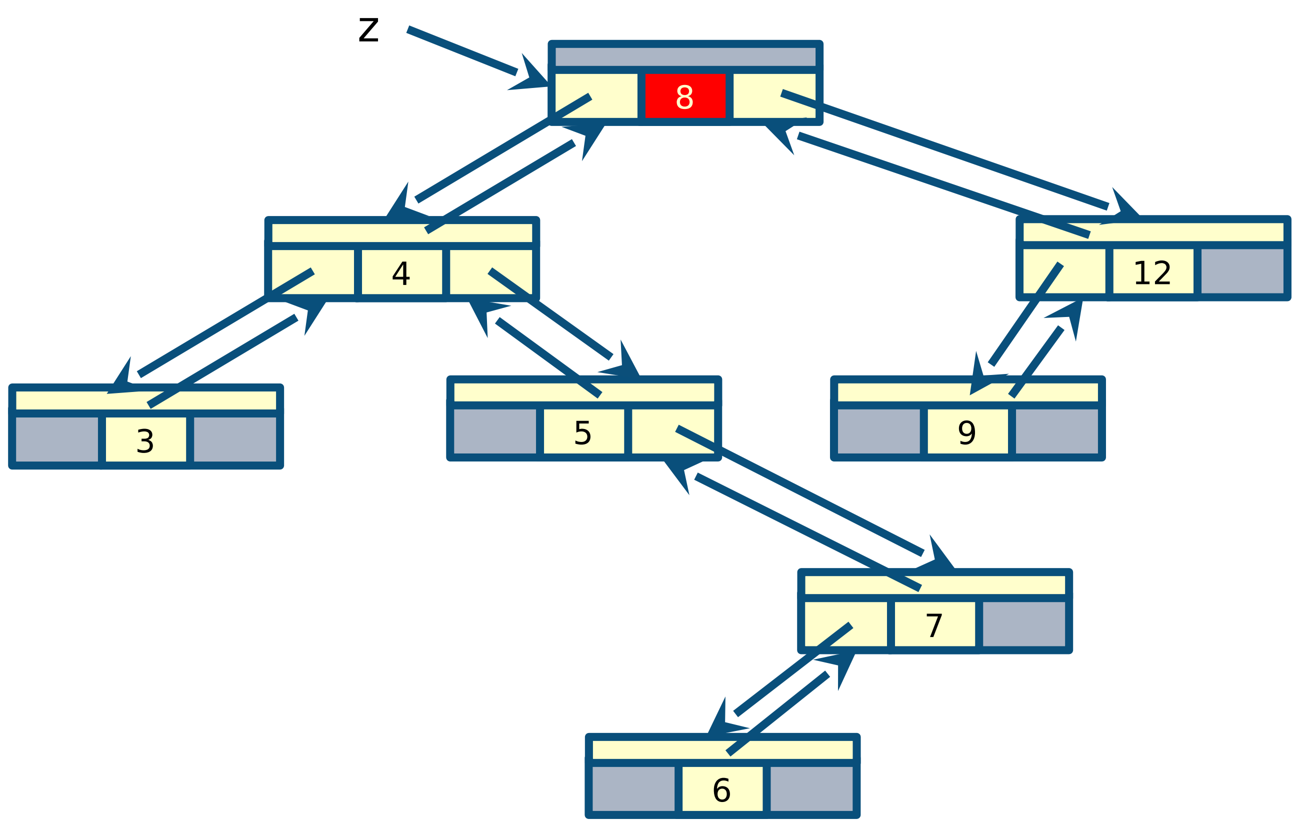
```
DELETE(T,z)
  if z.left = NIL
    TRANSPLANT(T,z,z.right)
  elseif z.right = NIL
    TRANSPLANT(T,z,z.left)
  else y = MINIMUM(z.right)
    if y.p != z
      TRANSPLANT(T,y,y.right)
      y.right := z.right
      y.right.p := y
    TRANSPLANT(T,z,y)
    y.left := z.left
    y.left.p := y
```



- DELETE(T,z) z.key = 8
- Termination

Example (case 4)

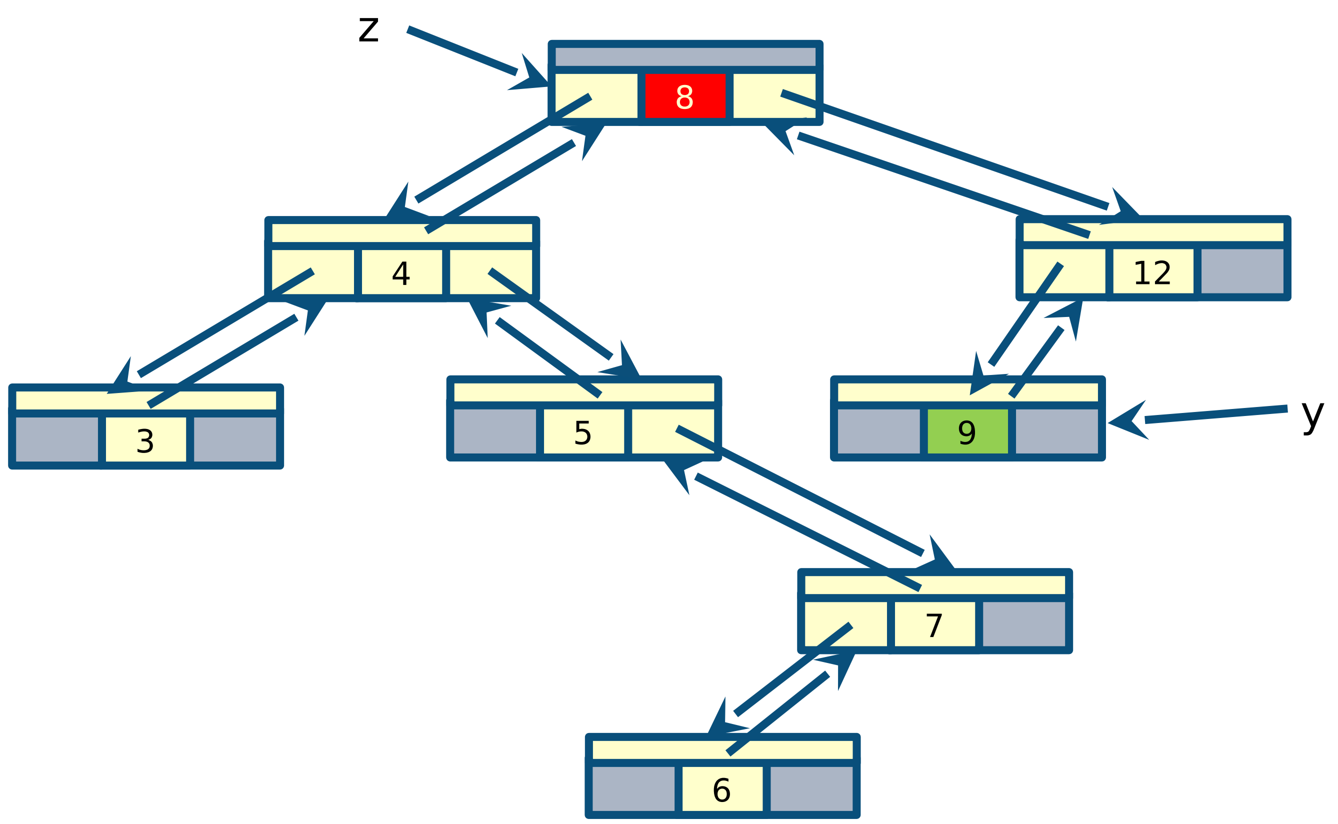
```
DELETE(T,z)
  if z.left = NIL
    TRANSPLANT(T,z,z.right)
  elseif z.right = NIL
    TRANSPLANT(T,z,z.left)
  else y = MINIMUM(z.right)
    if y.p != z
      TRANSPLANT(T,y,y.right)
      y.right := z.right
      y.right.p := y
    TRANSPLANT(T,z,y)
    y.left := z.left
    y.left.p := y
```



– DELETE(T,z) z.key = 8

Example (case 4)

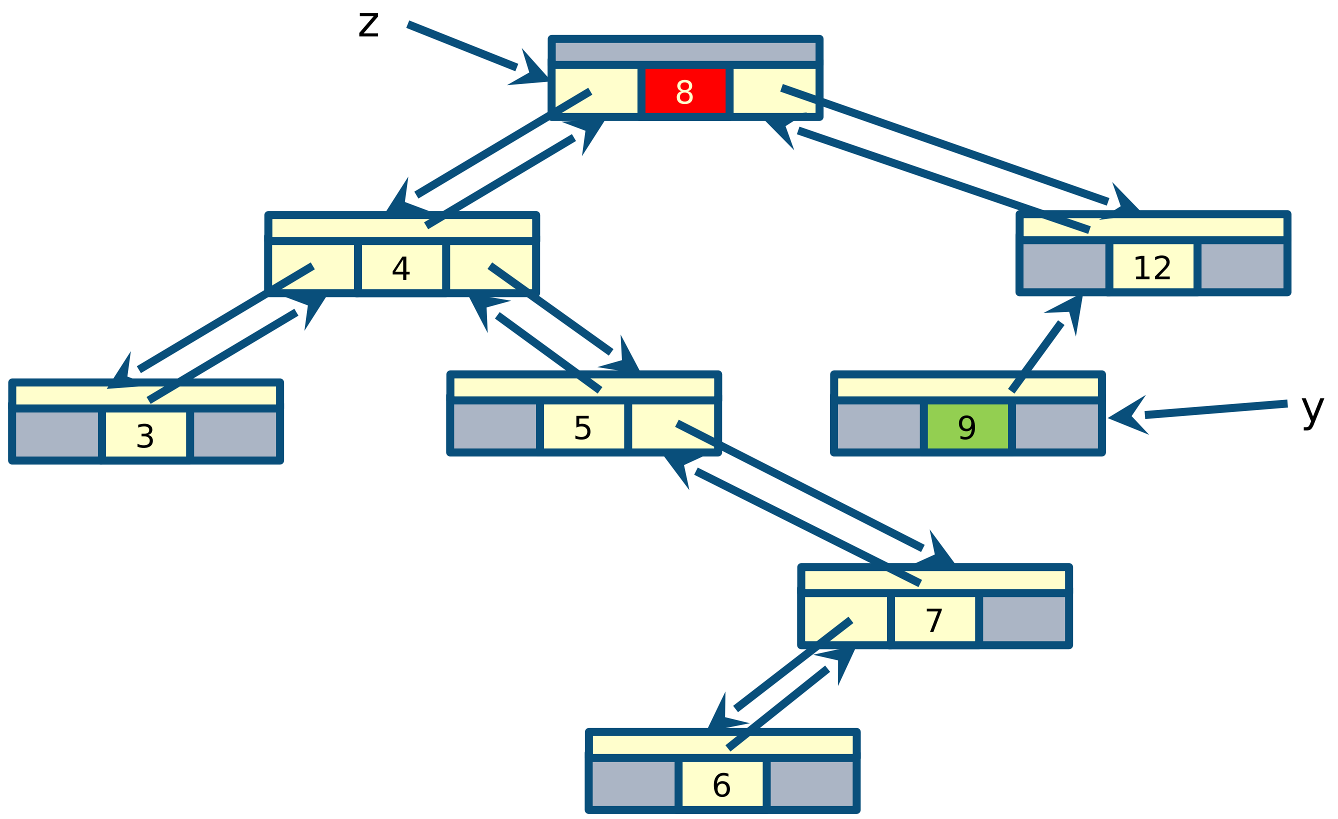
```
DELETE(T,z)
  if z.left = NIL
    TRANSPLANT(T,z,z.right)
  elseif z.right = NIL
    TRANSPLANT(T,z,z.left)
  else y = MINIMUM(z.right)
    if y.p != z
      TRANSPLANT(T,y,y.right)
      y.right := z.right
      y.right.p := y
    TRANSPLANT(T,z,y)
    y.left := z.left
    y.left.p := y
```



- DELETE(T,z) z.key = 8
- Call MINIMUM to find y (successor of z)

Example (case 4)

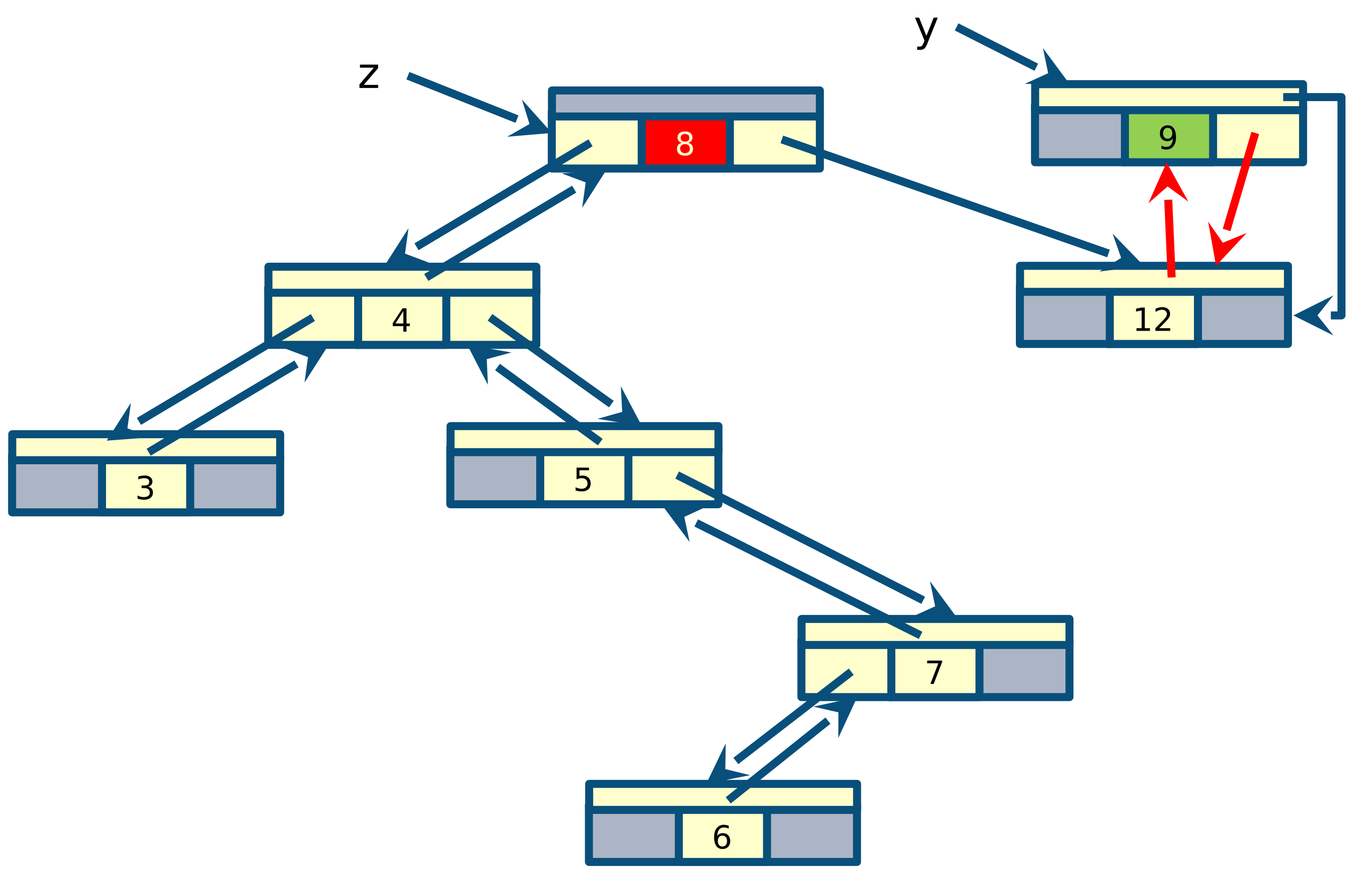
```
DELETE(T,z)
  if z.left = NIL
    TRANSPLANT(T,z,z.right)
  elseif z.right = NIL
    TRANSPLANT(T,z,z.left)
  else y = MINIMUM(z.right)
    if y.p != z
      TRANSPLANT(T,y,y.right)
      y.right := z.right
      y.right.p := y
    TRANSPLANT(T,z,y)
    y.left := z.left
    y.left.p := y
```



- DELETE(T,z) z.key = 8
- z is not the parent of y so replace y with y.right (call TRANSPLANT)

Example (case 4)

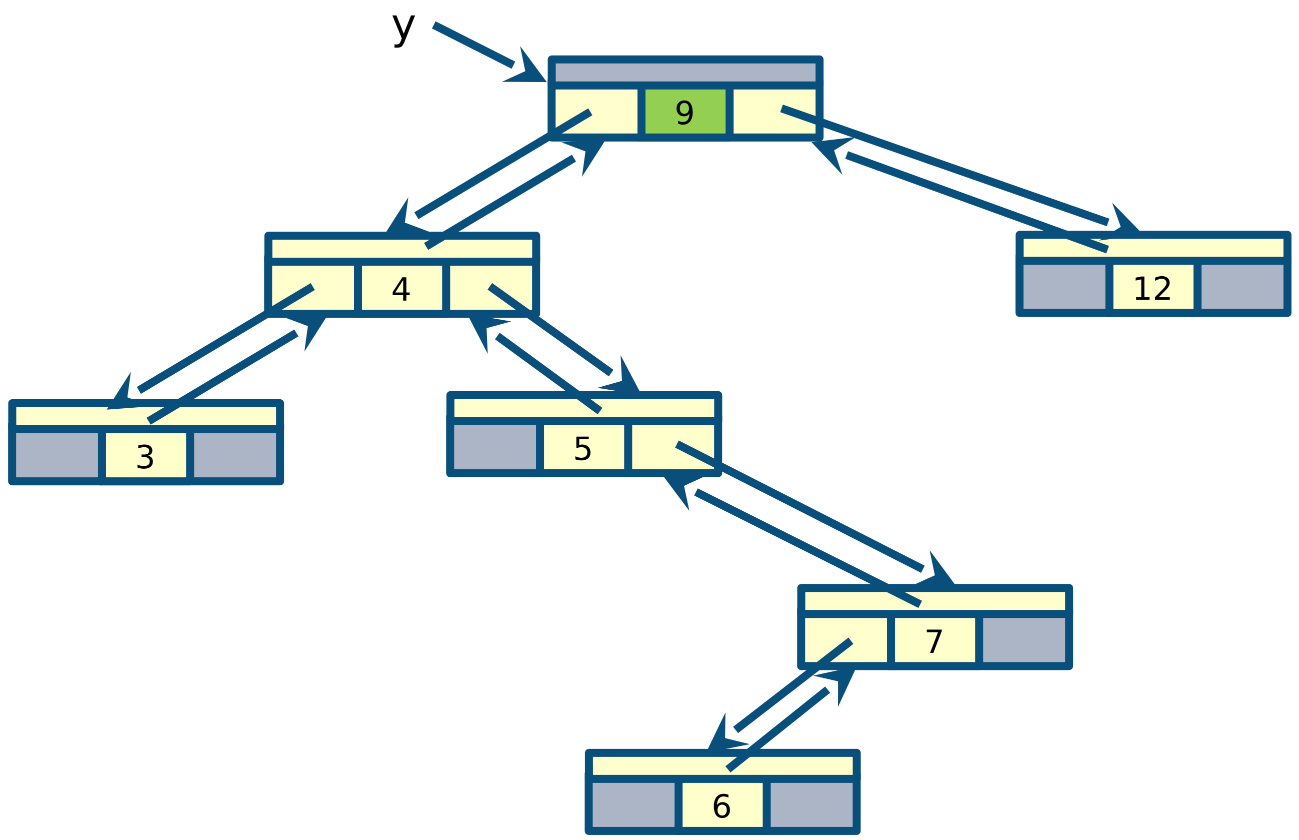
```
DELETE(T,z)
  if z.left = NIL
    TRANSPLANT(T,z,z.right)
  elseif z.right = NIL
    TRANSPLANT(T,z,z.left)
  else y = MINIMUM(z.right)
    if y.p != z
      TRANSPLANT(T,y,y.right)
      y.right := z.right
      y.right.p := y
    TRANSPLANT(T,z,y)
    y.left := z.left
    y.left.p := y
```



- DELETE(T,z) z.key = 8
- Update pointers

Example (case 4)

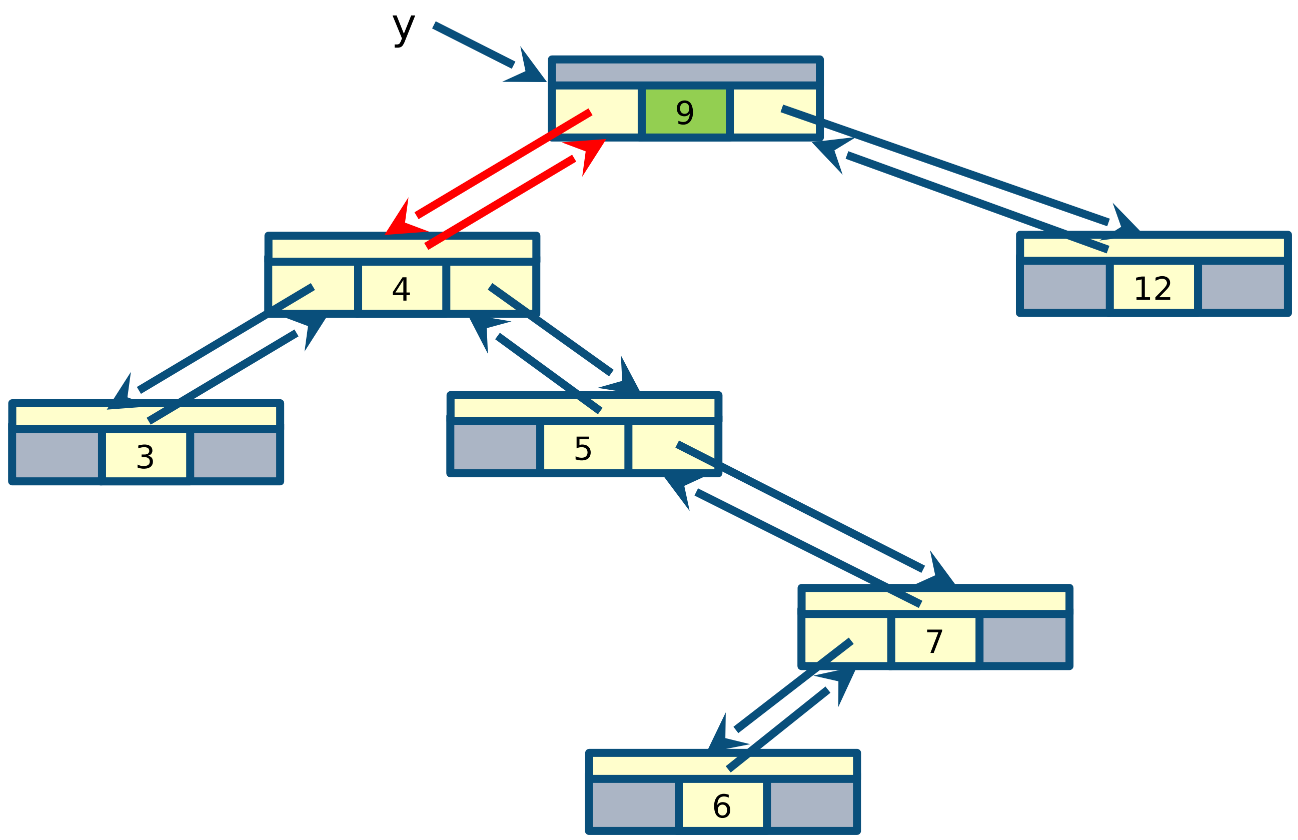
```
DELETE(T,z)
  if z.left = NIL
    TRANSPLANT(T,z,z.right)
  elseif z.right = NIL
    TRANSPLANT(T,z,z.left)
  else y = MINIMUM(z.right)
    if y.p != z
      TRANSPLANT(T,y,y.right)
      y.right := z.right
      y.right.p := y
    TRANSPLANT(T,z,y)
    y.left := z.left
    y.left.p := y
```



- DELETE(T,z) z.key = 8
- Call TRANSPLANT on y

Example (case 4)

```
DELETE(T,z)
  if z.left = NIL
    TRANSPLANT(T,z,z.right)
  elseif z.right = NIL
    TRANSPLANT(T,z,z.left)
  else y = 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
```

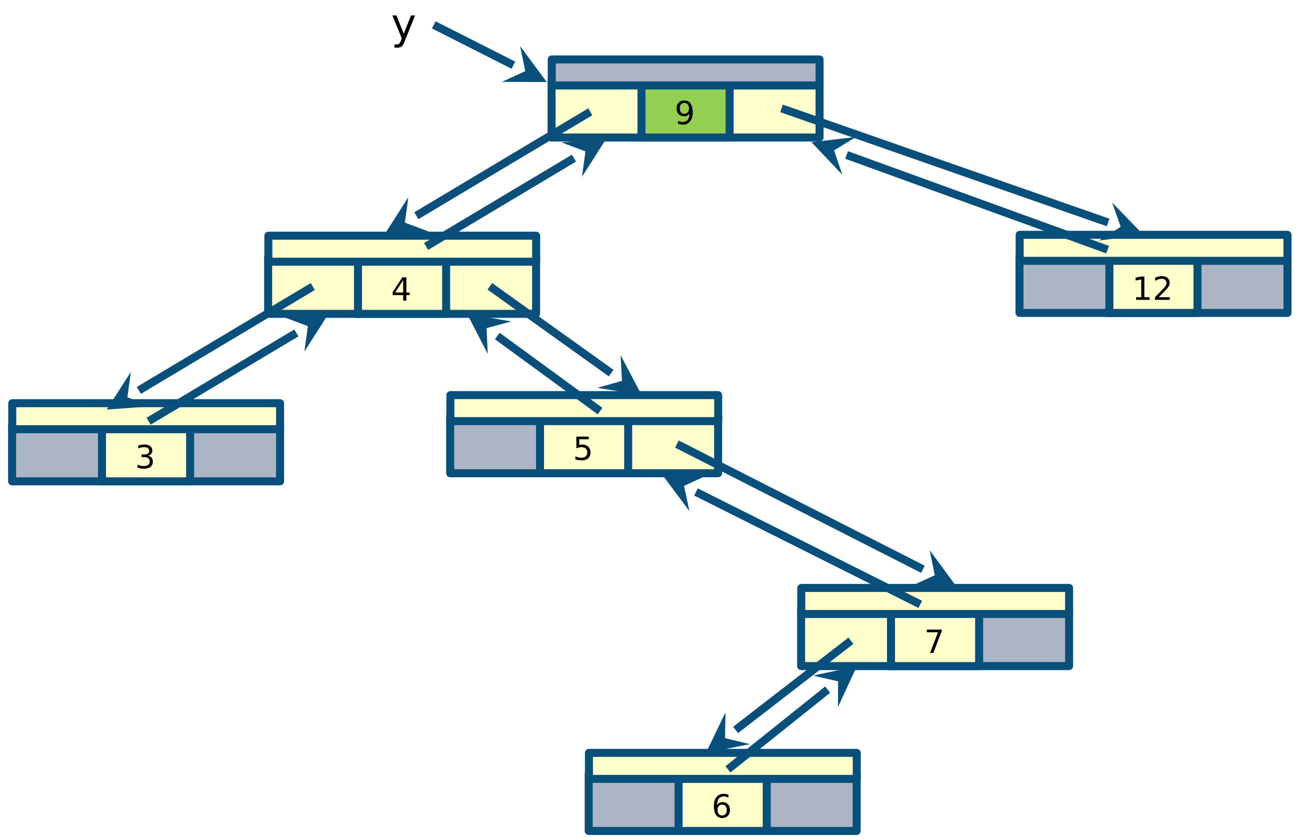


- DELETE(T,z) z.key = 8
- Update pointers

Example (case 4)

```
DELETE(T,z)
  if z.left = NIL
    TRANSPLANT(T,z,z.right)
  elseif z.right = NIL
    TRANSPLANT(T,z,z.left)
  else y = 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
```

- DELETE(T,z) z.key = 8
- Termination



Limitations

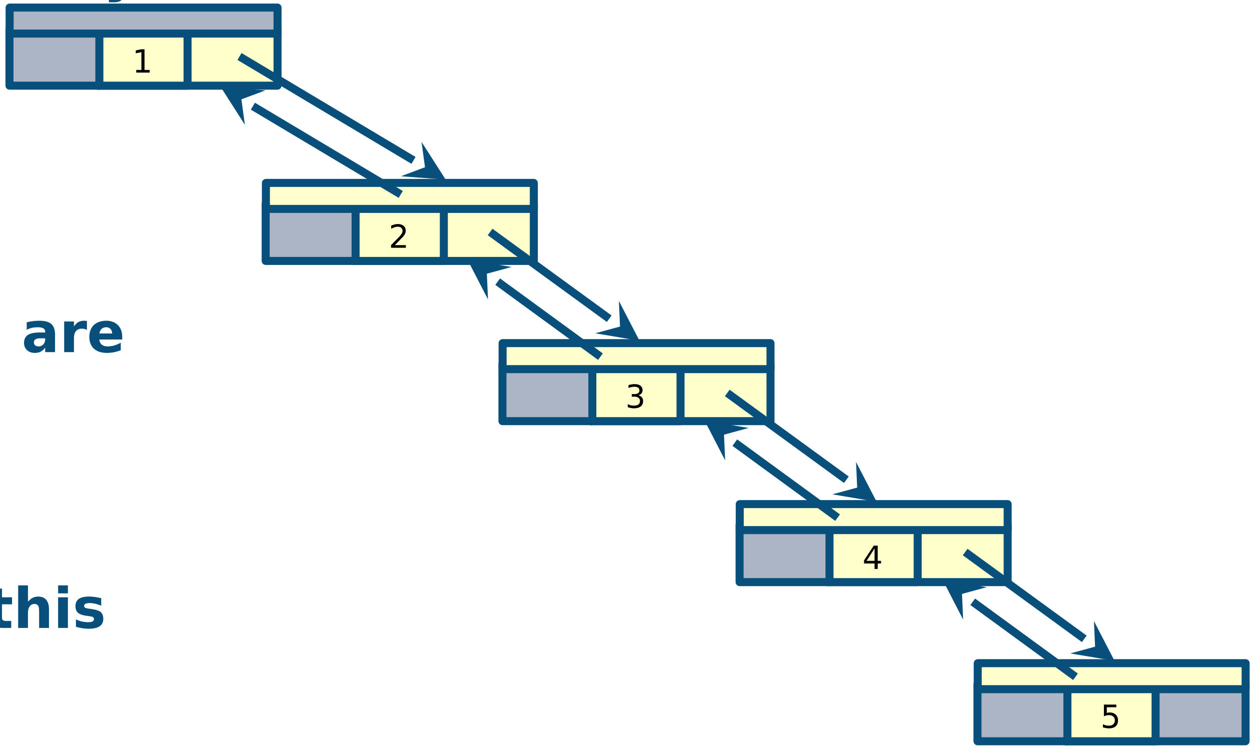
- **Each of the basic operations on a binary search tree runs in O(h) time**
  - h is the height of the tree

- **However, the height varies as items are inserted and deleted**

- **Try to insert elements 1,2,3,4,5 (in this order) into an empty BST**

Limitations

- **Each of the basic operations on a binary search tree runs in O(h) time**
  - h is the height of the tree
- **However, the height varies as items are inserted and deleted**
- **Try to insert elements 1,2,3,4,5 (in this order) into an empty BST**
  - Unbalanced tree with height 4
  - Height is O(n) in unbalanced trees



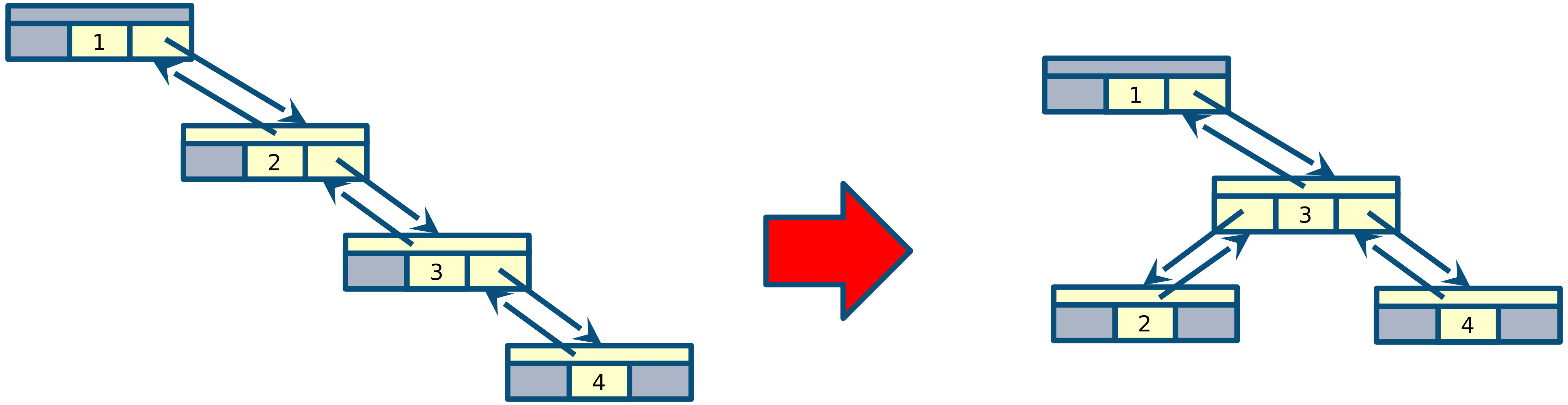
Self-balancing trees

- **Several extensions to the basic BST definition have been introduced to keep the height small as items are dynamically inserted and deleted**
  - Red-Black trees
  - AVL trees
  - B-trees

- **A common method to keep the tree balanced is to perform rotations after each deletion and insertion**
  - Local operation in a search tree that preserves the binary-search-tree property

Example

- Left rotate on 2



Randomly built binary search tree

- **What is the height of a randomly built BST on n distinct keys?**
  - By insertion alone

- **The height on average is O(log n)**
  - The proof is quite involved and is not part of the course
  - See Section 12.4 in the Cormen book

- **Open problem when deletions are also considered**

Duplicate keys

- **Our definition of insertion stores any duplicates in the right subtree**
  - If there are a lot of duplicates this strategy might lead to an unbalanced trees

- **Different approaches have been proposed to alleviate this problem**
  - Keep a list of duplicates linked to each node
  - Add a count attribute to each node
  - Pick randomly either left or right
  - Ignore duplicates

Summary

- **Binary search trees (BSTs)**
- **Querying a tree**
- **Computation of tree parameters**
- **Operations**
  - Insertion
  - Deletion
- **Randomly build BSTs**
- **BSTs with equal keys**