BINARY SEARCH TREE

Data Structures and Algorithms

Waheed Iqbal

Department of Data Science, FCIT
University of the Punjab, Lahore, Pakistan

Introduction

- Tree is an important data structure to maintain and manipulate data specifically for hierarchy relationships

- Important Terminology:
  - Node
  - Parent
  - Child
  - Link

Introduction

- The search tree data structure supports many dynamic-set operations, including:
  - SEARCH
  - MINIMUM
  - MAXIMUM
  - PREDECESSOR
  - SUCCESSOR
  - INSERT
  - DELETE

Binary Search Tree

- Binary search tree is organized as a binary tree
- We represent binary search tree in a linked data structure
- Each node contains key, satellite data, left child, right child, parent references
- The main property of binary search tree is:

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 (Cont.)

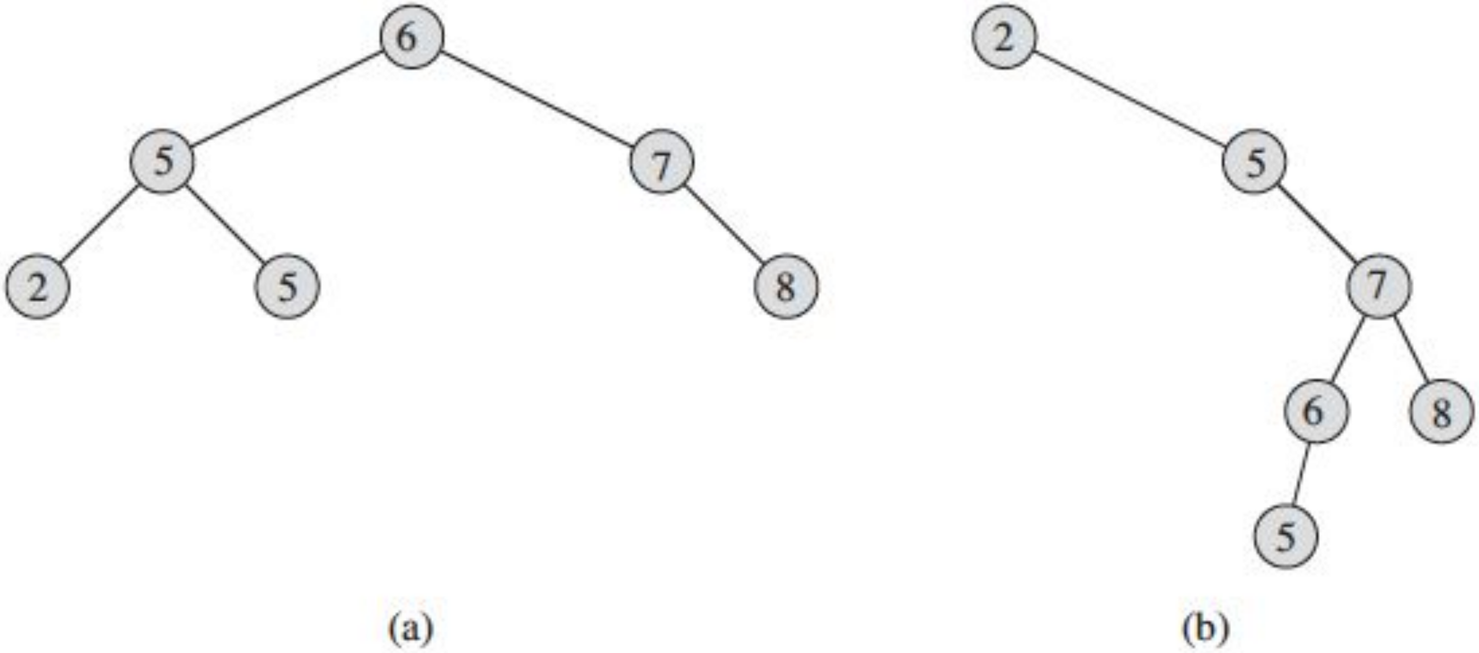


Figure 12.1 Binary search trees. For any node x, the keys in the left subtree of x are at most $x.key$, and the keys in the right subtree of x are at least $x.key$. Different binary search trees can represent the same set of values. The worst-case running time for most search-tree operations is proportional to the height of the tree. **(a)** A binary search tree on 6 nodes with height 2. **(b)** A less efficient binary search tree with height 4 that contains the same keys.

Binary Search Tree Implementation

```
class BinarySearchTree:
    def __init__(self):
        self.root = None

    def insert(self, value):
        if not self.root:
            self.root = TreeNode(value)
        else:
            current = self.root
            while True:
                if value < current.value:
                    if current.left is None:
                        current.left = TreeNode(value)
                        break
                    else:
                        current = current.left
                elif value > current.value:
                    if current.right is None:
                        current.right = TreeNode(value)
                        break
                    else:
                        current = current.right
                else:
                    # Value already exists in the tree
                    break
```

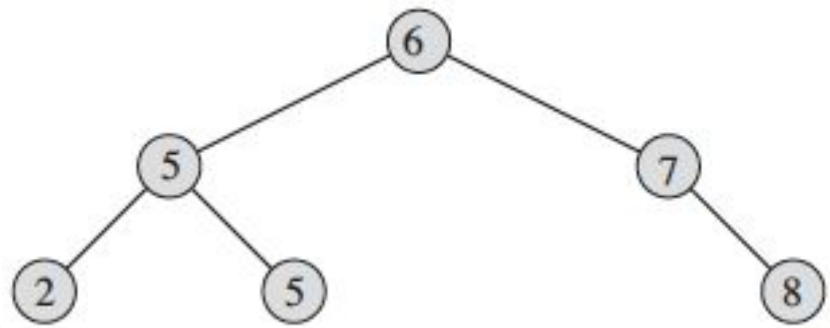
```
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
```

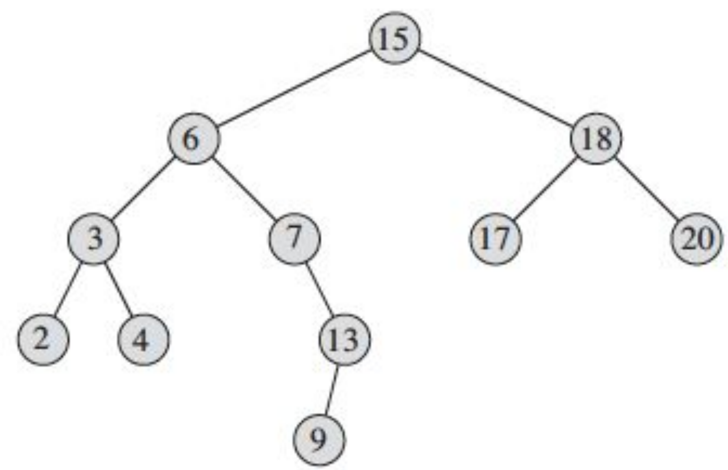
```
def search(self, value):
    current = self.root
    while current:
        if value == current.value:
            return True
        elif value < current.value:
            current = current.left
        else:
            current = current.right
    return False
```

Binary Search Tree (Cont.)

Traversal: Inorder tree traversal use to print sorted nodes

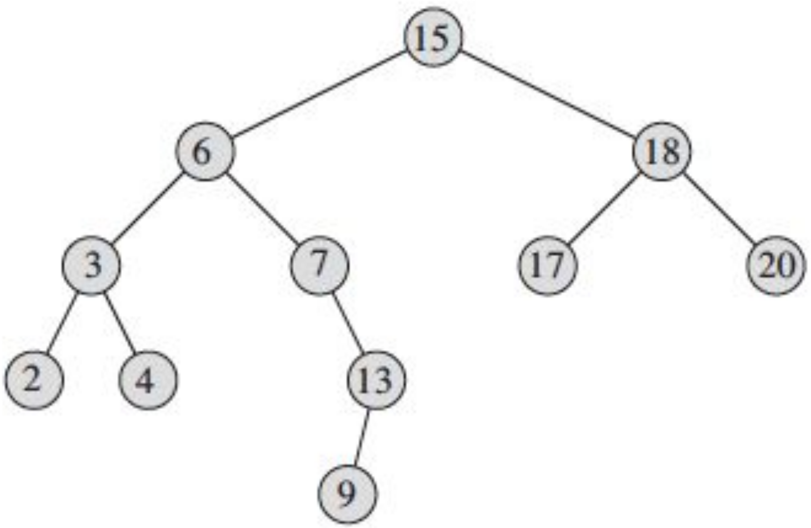
```
INORDER-TREE-WALK(x)
if x ≠ NIL
    INORDER-TREE-WALK(x.left)
    print x.key
    INORDER-TREE-WALK(x.right)
```





Binary Search Tree (Cont.)

Searching:




```
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)
```

```
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
```

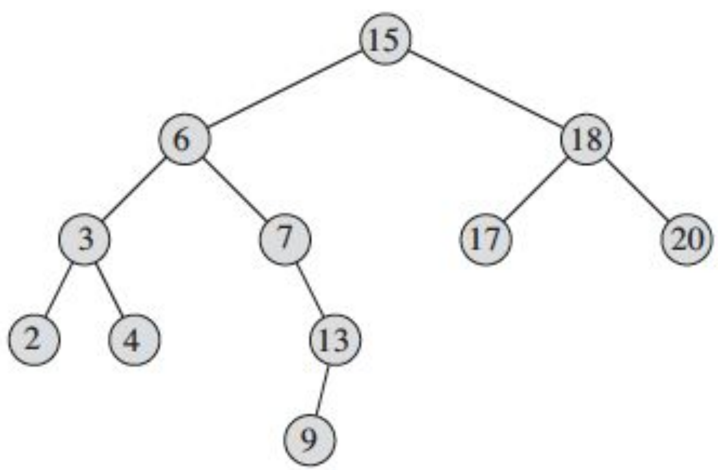
Figure 12.2 Queries on a binary search tree. To search for the key 13 in the tree, we follow the path $15 \rightarrow 6 \rightarrow 7 \rightarrow 13$ from the root. The minimum key in the tree is 2, which is found by following *left* pointers from the root. The maximum key 20 is found by following *right* pointers from the root. The successor of the node with key 15 is the node with key 17, since it is the minimum key in the right subtree of 15. The node with key 13 has no right subtree, and thus its successor is its lowest ancestor whose left child is also an ancestor. In this case, the node with key 15 is its successor.

Binary Search Tree (Cont.)

Minimum and Maximum

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

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

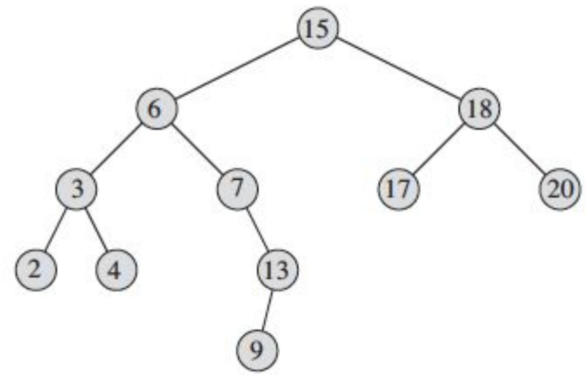


Binary Search Tree (Cont.)

Successor and Predecessor

Successor of a node x is the node with the smallest key greater than x.key

```
TREE-SUCCESSOR(x)
1  if x.right ≠ NIL
2      return TREE-MINIMUM(x.right)
3  y = x.p
4  while y ≠ NIL and x == y.right
5      x = y
6      y = y.p
7  return y
```



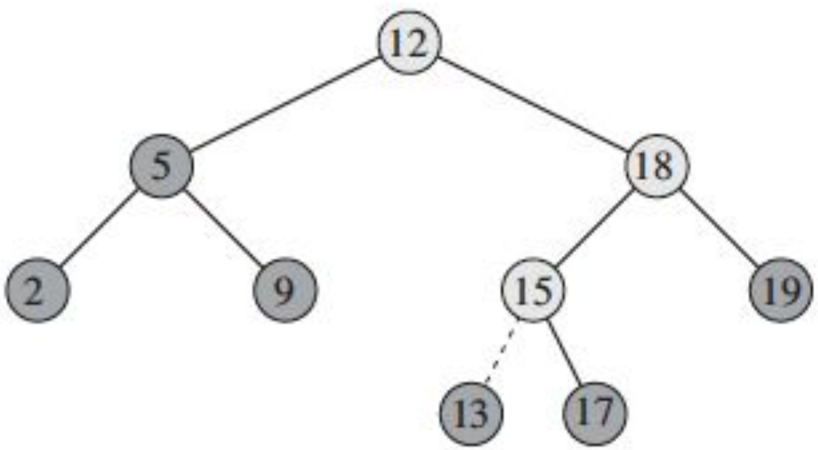
Line 3-7 simply finds a node x which is a left child of its parent y

it returns the successor of a node x in a binary search tree if it exists, and NIL if x has the largest key in the tree.

How would you implement TREE-PREDECESSOR(x)?

Binary Search Tree (Cont.)

Insertion



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
```

Figure 12.3 Inserting an item with key 13 into a binary search tree. Lightly shaded nodes indicate the simple path from the root down to the position where the item is inserted. The dashed line indicates the link in the tree that is added to insert the item.

Binary Search Tree (Cont.)

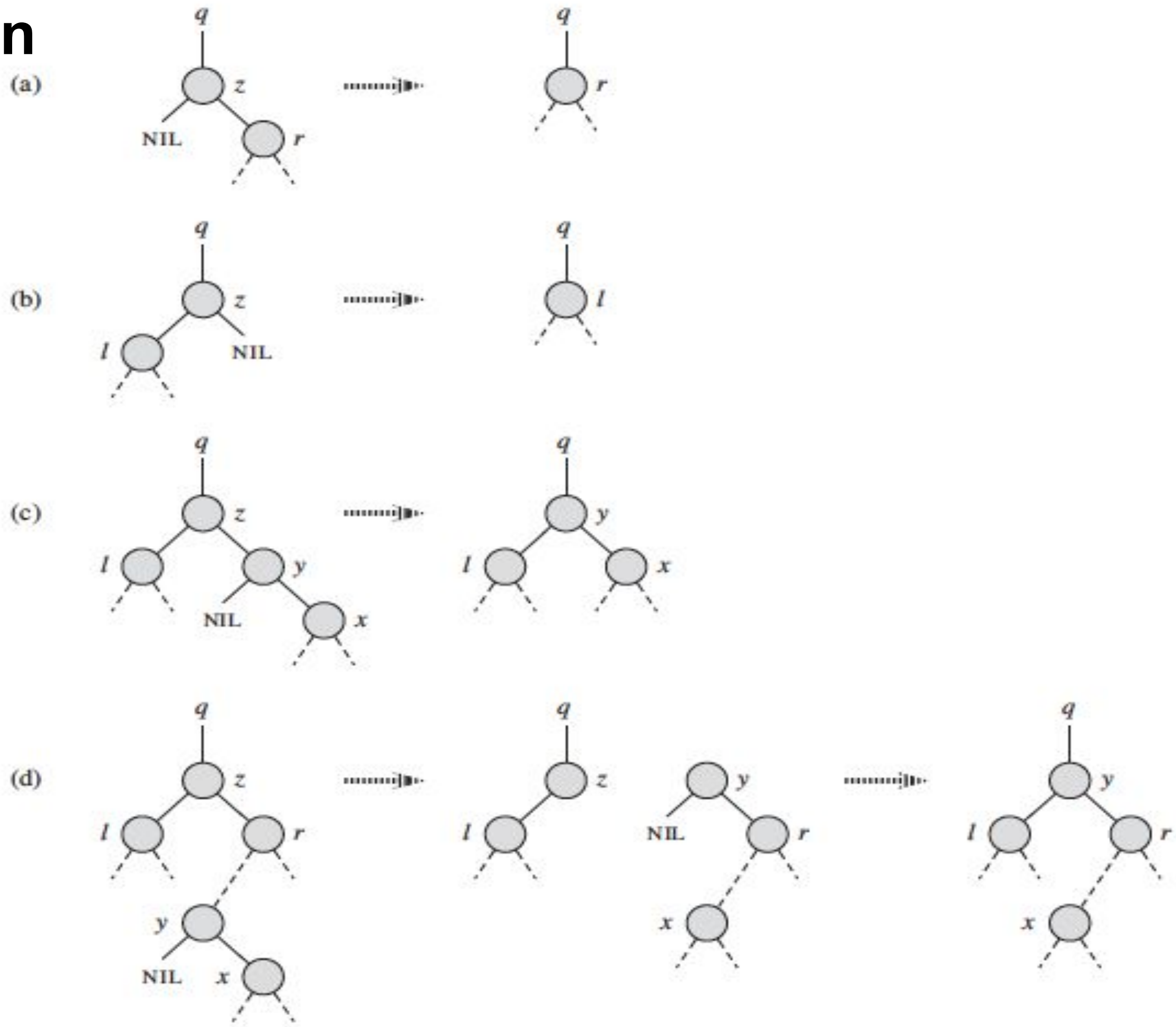
Deletion

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.

Binary Search Tree (Cont.)

Deletion

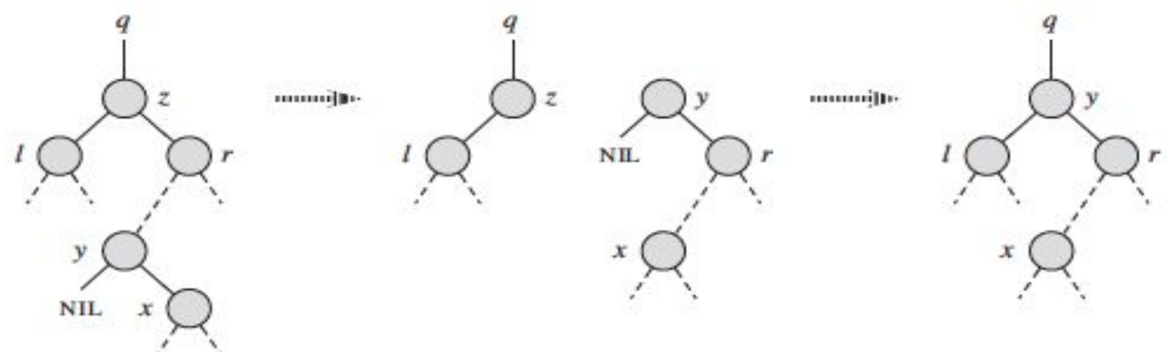


Binary Search Tree (Cont.)

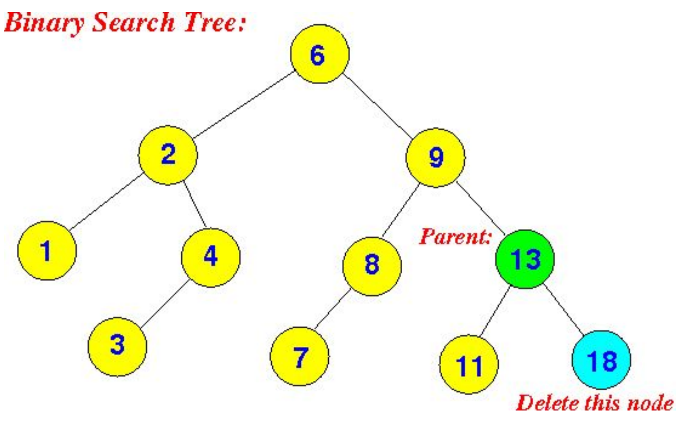
Deletion

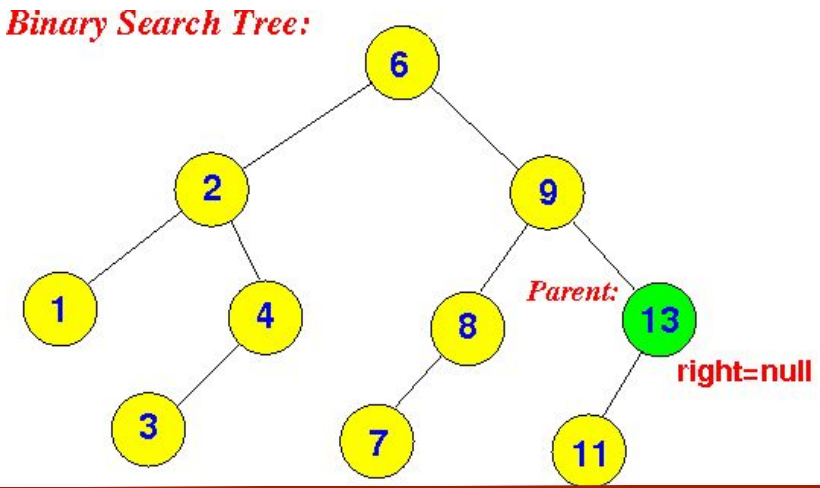
```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSPLANT(T, z, z.right)
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    if y.p ≠ z
        TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
```

```
TRANSPLANT(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
```

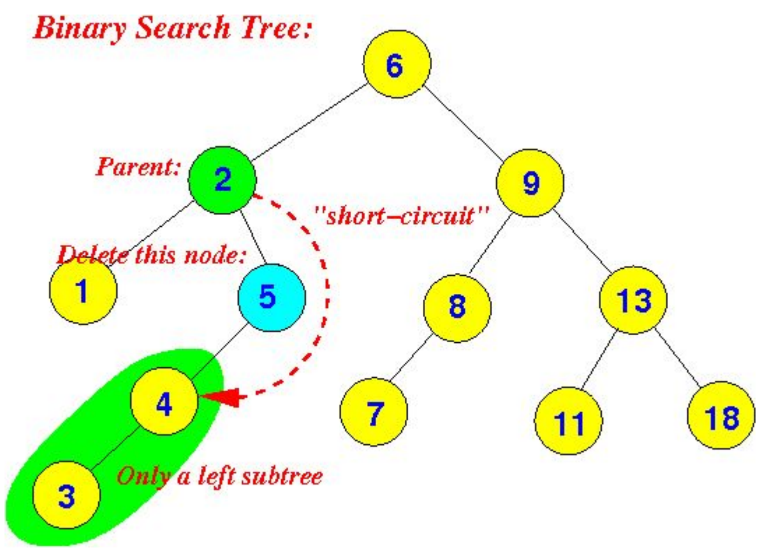


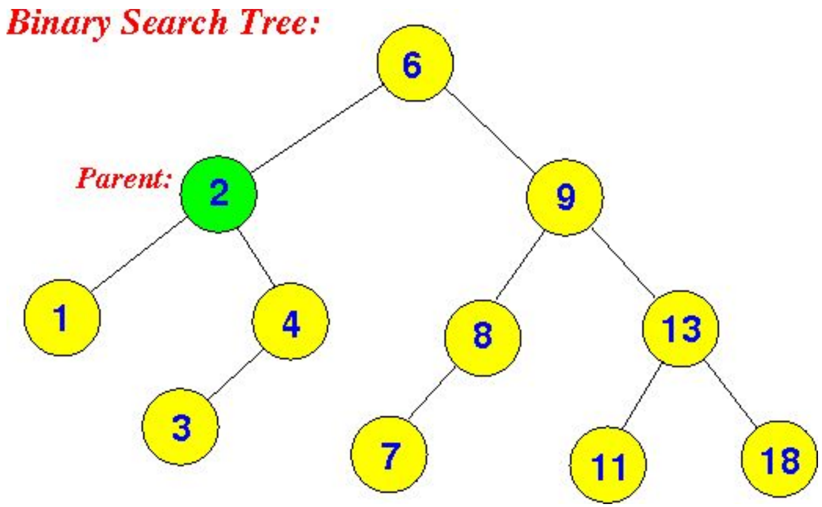
Simple Case 1: Z has no children



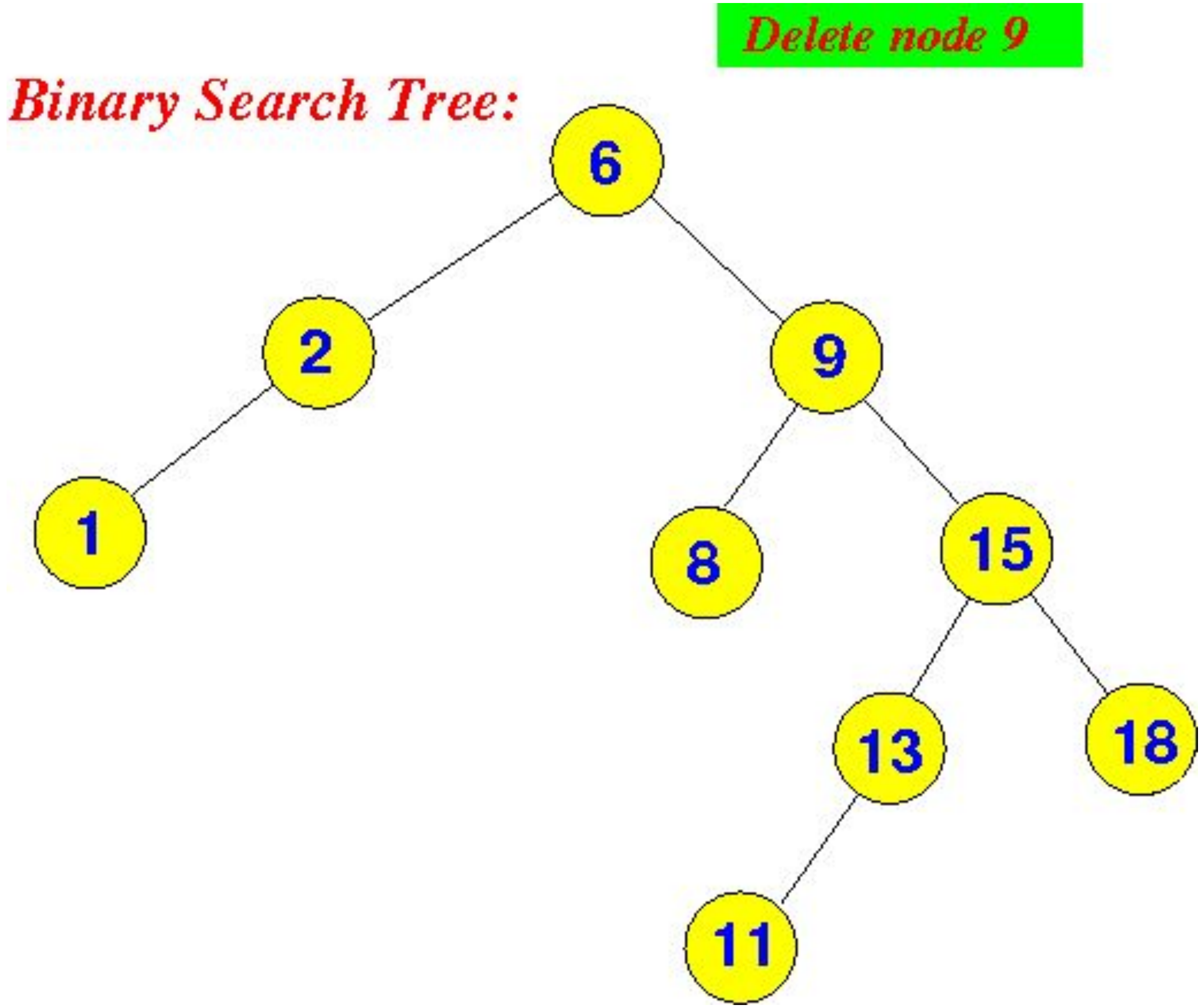


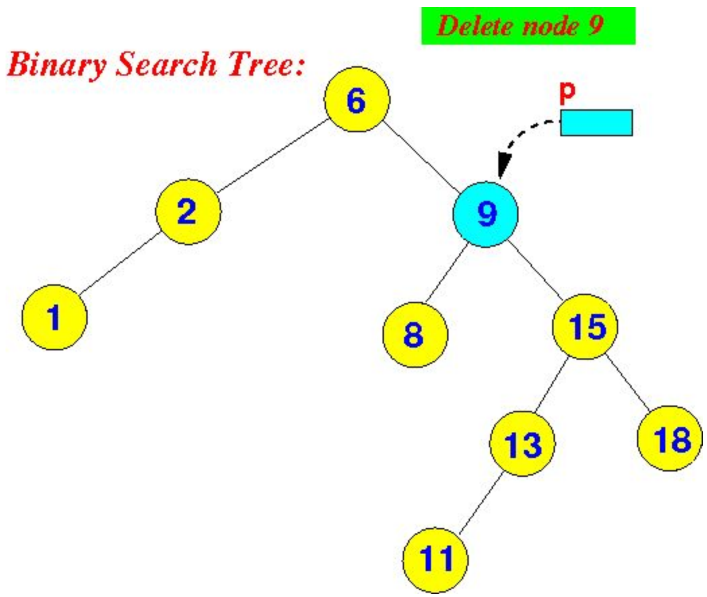
Simple Case 2: Z has only one child



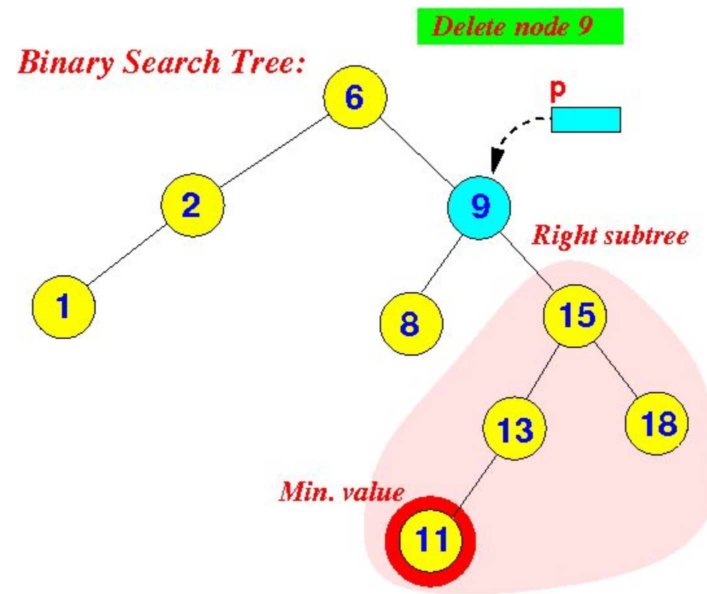


Example #1: BST Delete

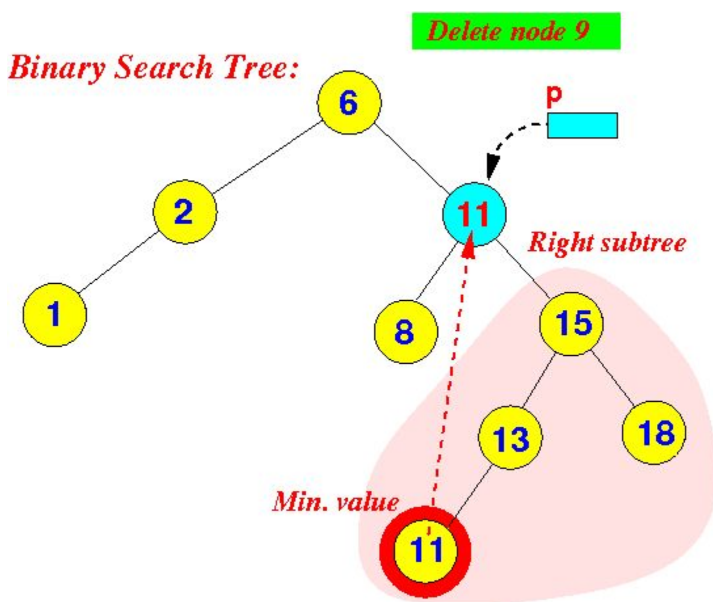



Delete node 9
Binary Search Tree:
p
6
2
9
1
8
15
13
18
11

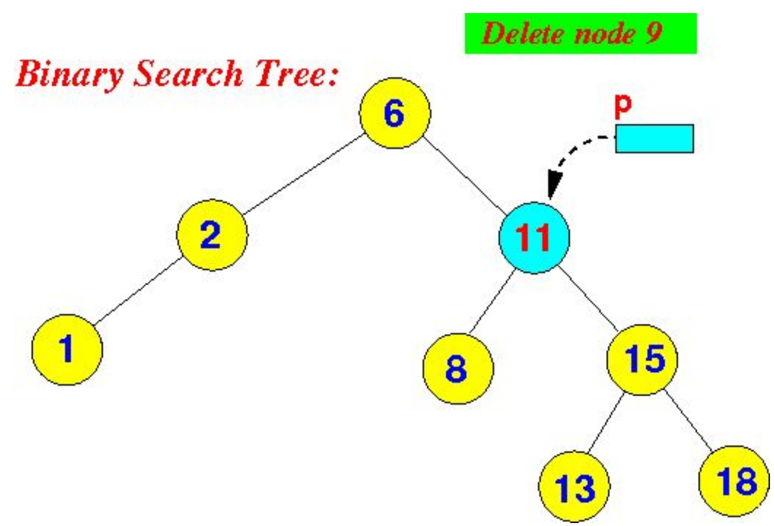
(a)


Delete node 9
Binary Search Tree:
p
6
2
9
Right subtree
1
8
15
13
18
Min. value
11

(b)

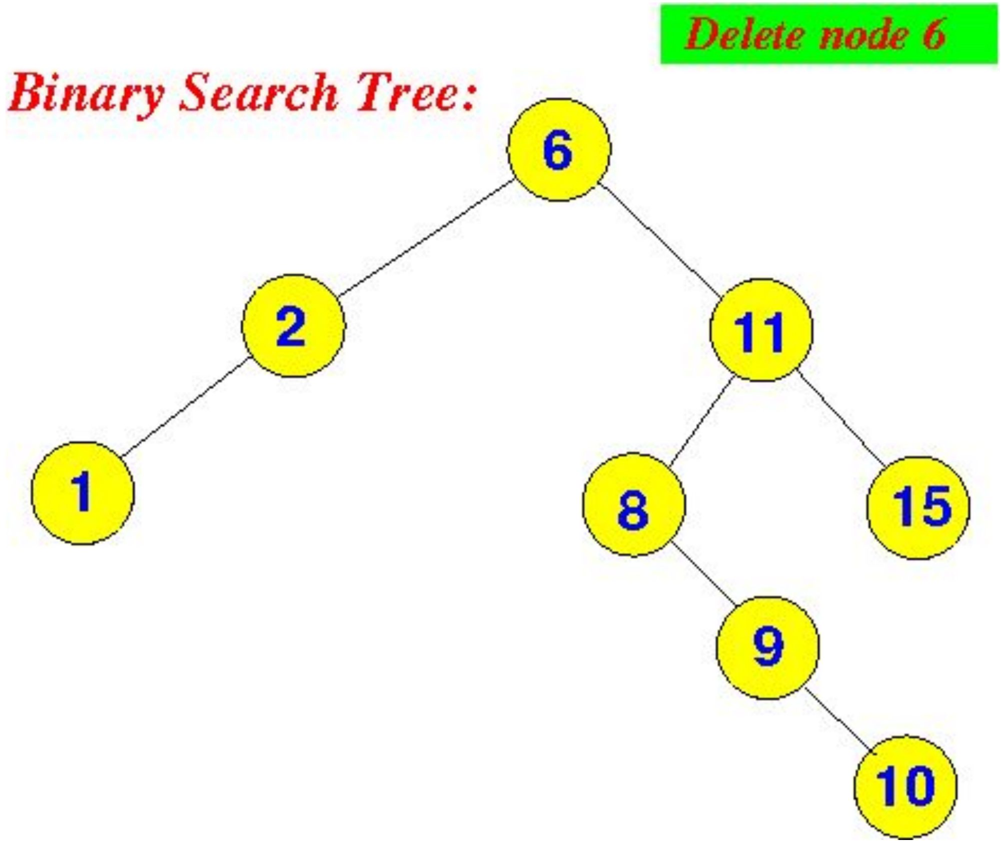

Delete node 9
Binary Search Tree:
p
6
2
11
Right subtree
1
8
15
13
18
Min. value
11

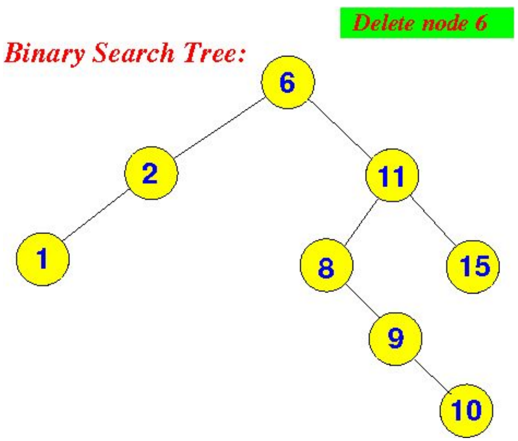
(c)

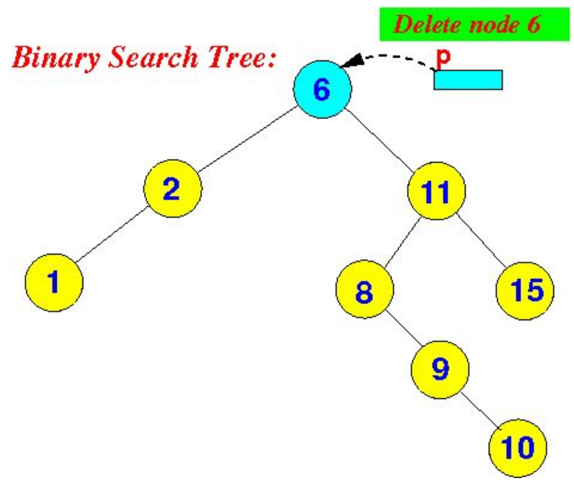

Delete node 9
Binary Search Tree:
p
6
2
11
1
8
15
13
18

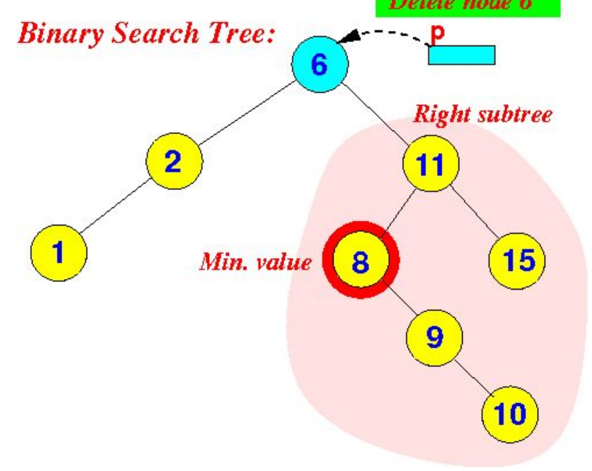
(d)

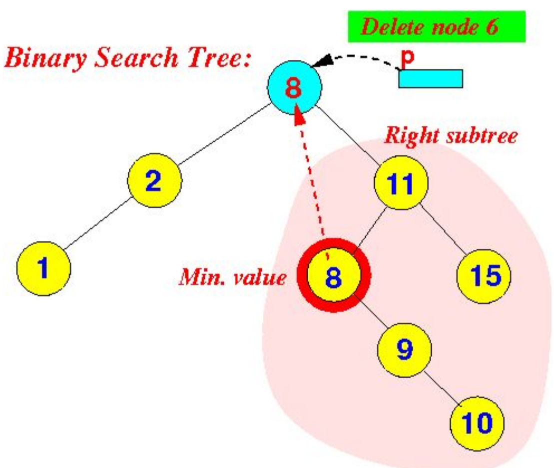
Example #2: BST Delete

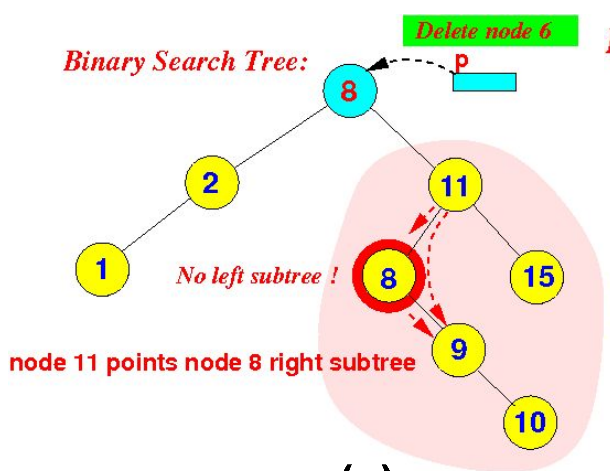


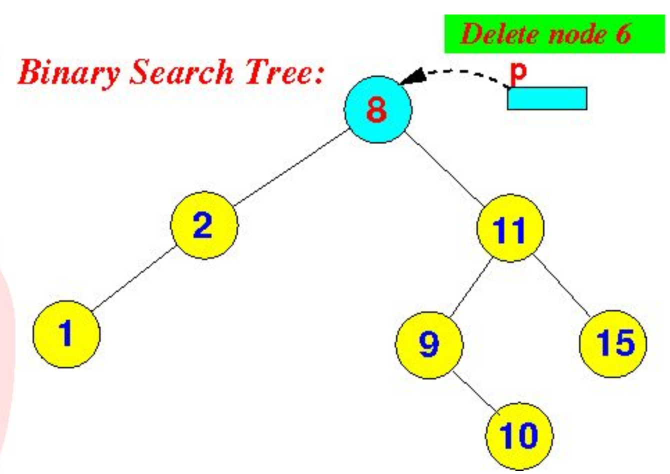

Delete node 6
Binary Search Tree:
6
2
11
1
8
15
9
10
(a)


Delete node 6
Binary Search Tree:
p
6
2
11
1
8
15
9
10
(b)

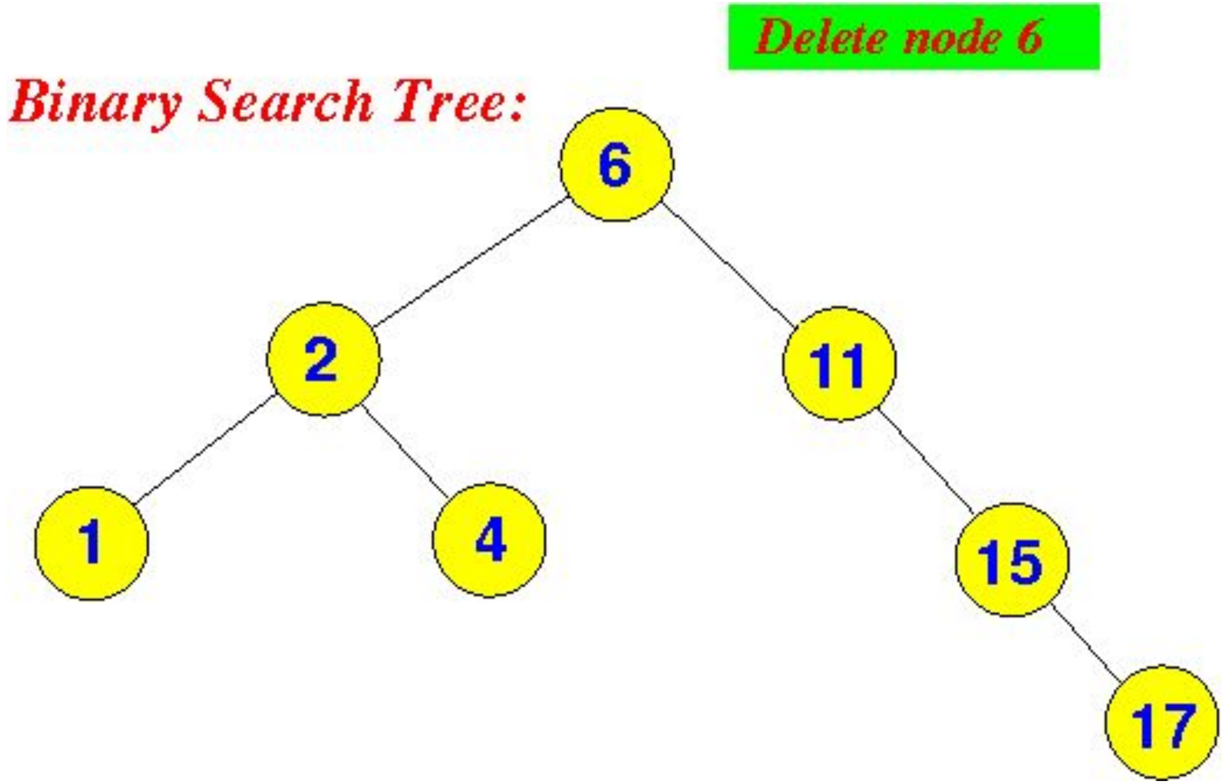

Delete node 6
Binary Search Tree:
p
6
Right subtree
2
11
1
Min. value
8
15
9
10
(c)

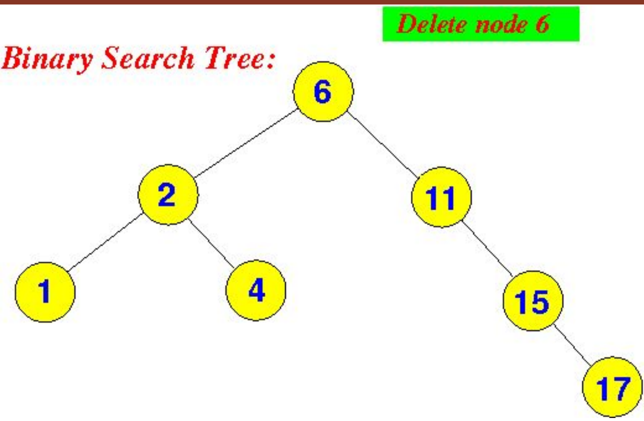

Delete node 6
Binary Search Tree:
p
8
Right subtree
2
11
1
Min. value
8
15
9
10
(d)


Delete node 6
Binary Search Tree:
p
8
2
11
1
No left subtree !
8
15
9
node 11 points node 8 right subtree
10
(e)

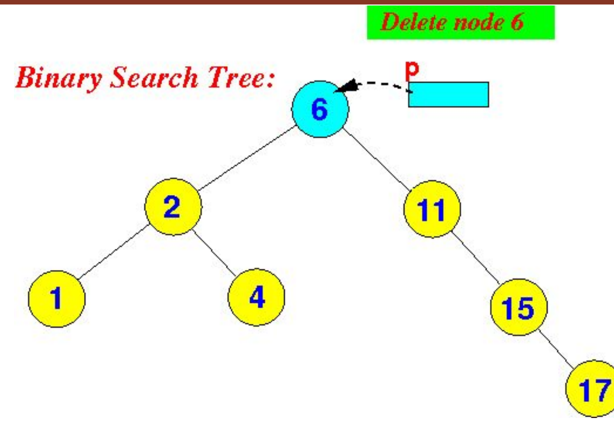

Delete node 6
Binary Search Tree:
p
8
2
11
1
9
15
10
(f)

Example #3: BST Delete

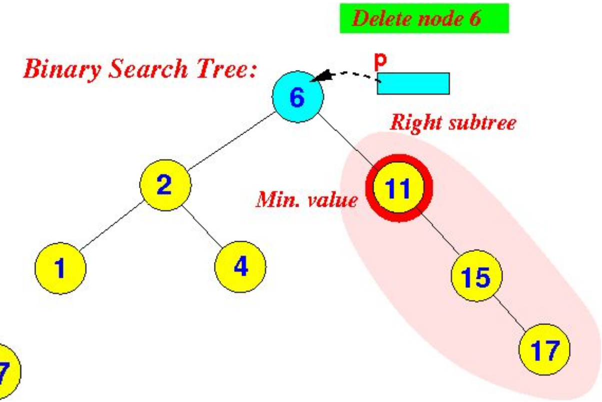



Delete node 6
Binary Search Tree:
6
2
11
1
4
15
17

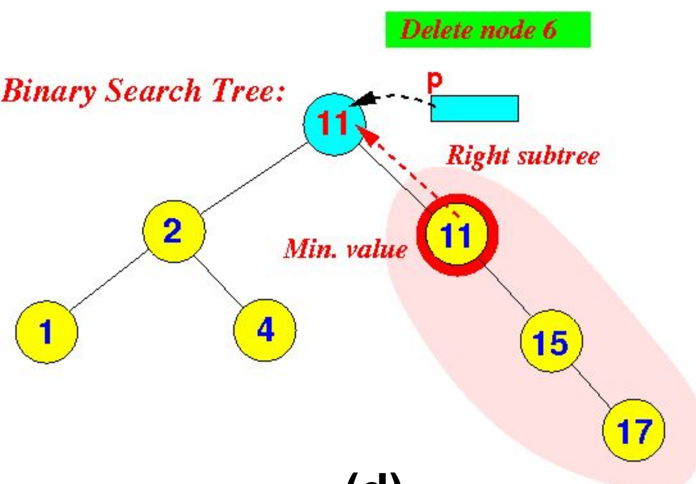
(a)


Delete node 6
Binary Search Tree:
p
6
2
11
1
4
15
17

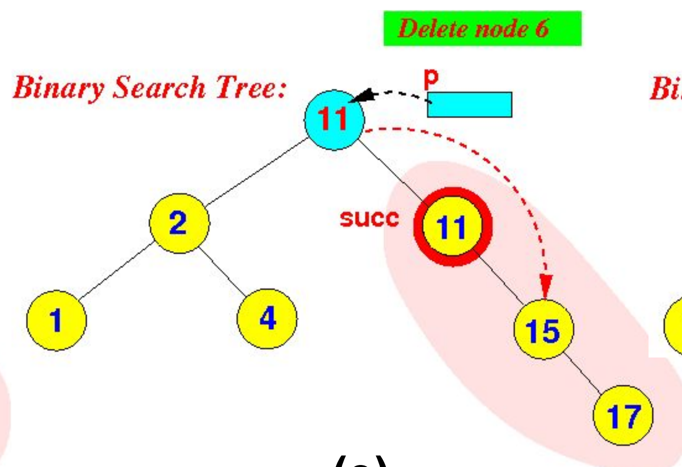
(b)


Delete node 6
Binary Search Tree:
p
6
Right subtree
Min. value
2
11
1
4
15
17

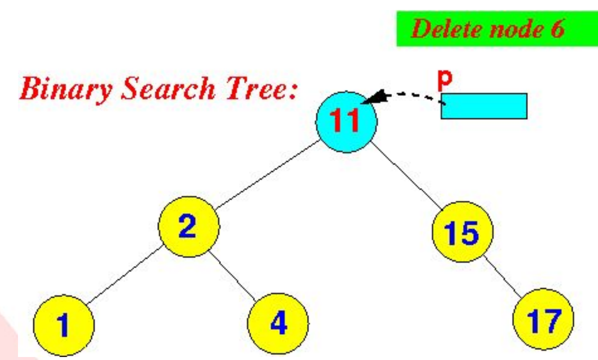
(c)


Delete node 6
Binary Search Tree:
p
11
Right subtree
Min. value
2
11
1
4
15
17

(d)


Delete node 6
Binary Search Tree:
p
11
succ
2
11
1
4
15
17

(e)


Delete node 6
Binary Search Tree:
p
11
2
15
1
4
17

(f)

Credit

- These notes contain material from Chapter 12 of Cormen, Leiserson, Rivest, and Stein (3rd Edition).
- http://www.mathcs.emory.edu/~cheung/Courses/171/Syllabus/9-BinTree/BST-delete2.html