Binary Search Trees

CSE 373

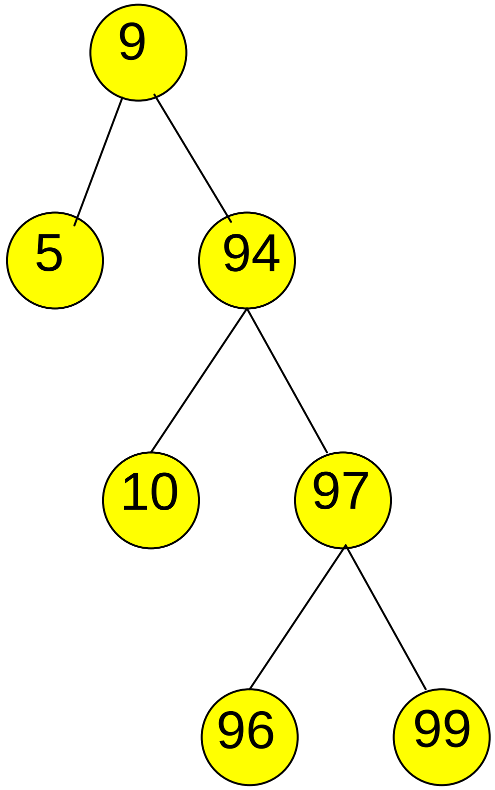
Data Structures

Readings

- Chapter 10 Section 10.1

Binary Search Trees

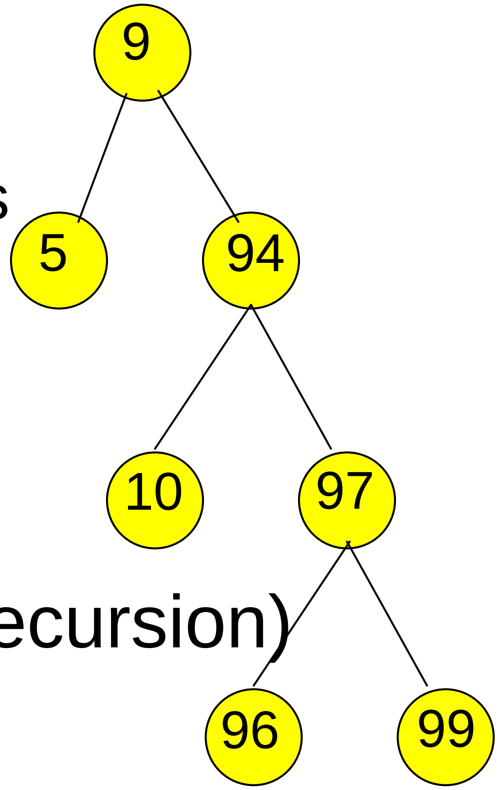
- Binary search trees are binary trees in which
  - › all values in the node's left subtree are less than node value
  - › all values in the node's right subtree are greater than node value
- Operations:
  - › Find, FindMin, FindMax, Insert, Delete



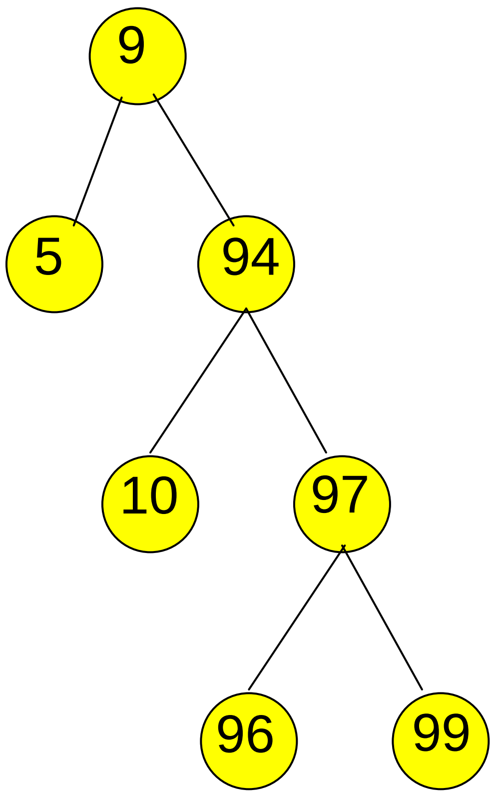
What happens when we traverse the tree in inorder?

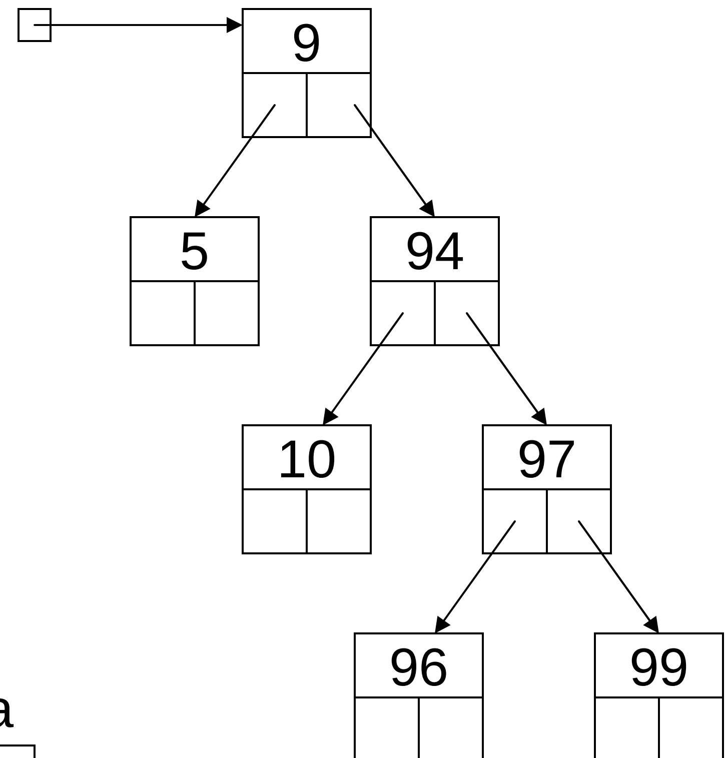
Operations on Binary Search Trees

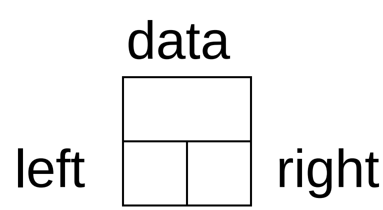
- How would you implement these?
  - › Recursive definition of binary search trees allows recursive routines
- FindMin
- FindMax
- Find
- Insert (but be careful when using recursion)
- Delete (the only tricky one)



Binary SearchTree







Find

```
Find(T : tree pointer, x : element): tree pointer {
case {
  T = null : return null;
  T.data = x : return T;
  T.data > x : return Find(T.left,x);
  T.data < x : return Find(T.right,x)
}
}
```

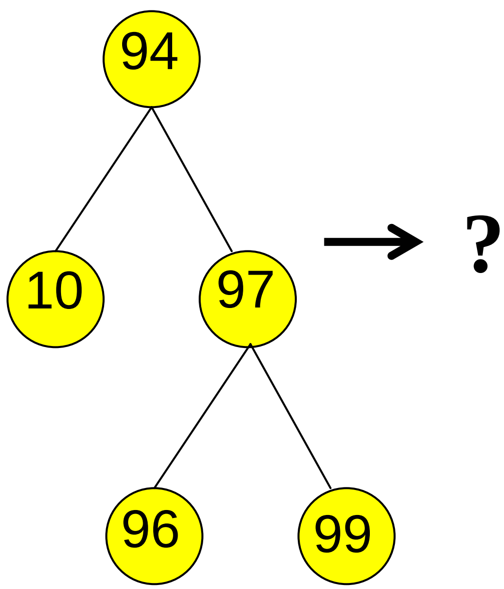
FindMin

- Design recursive FindMin operation that returns the smallest element in a binary search tree.

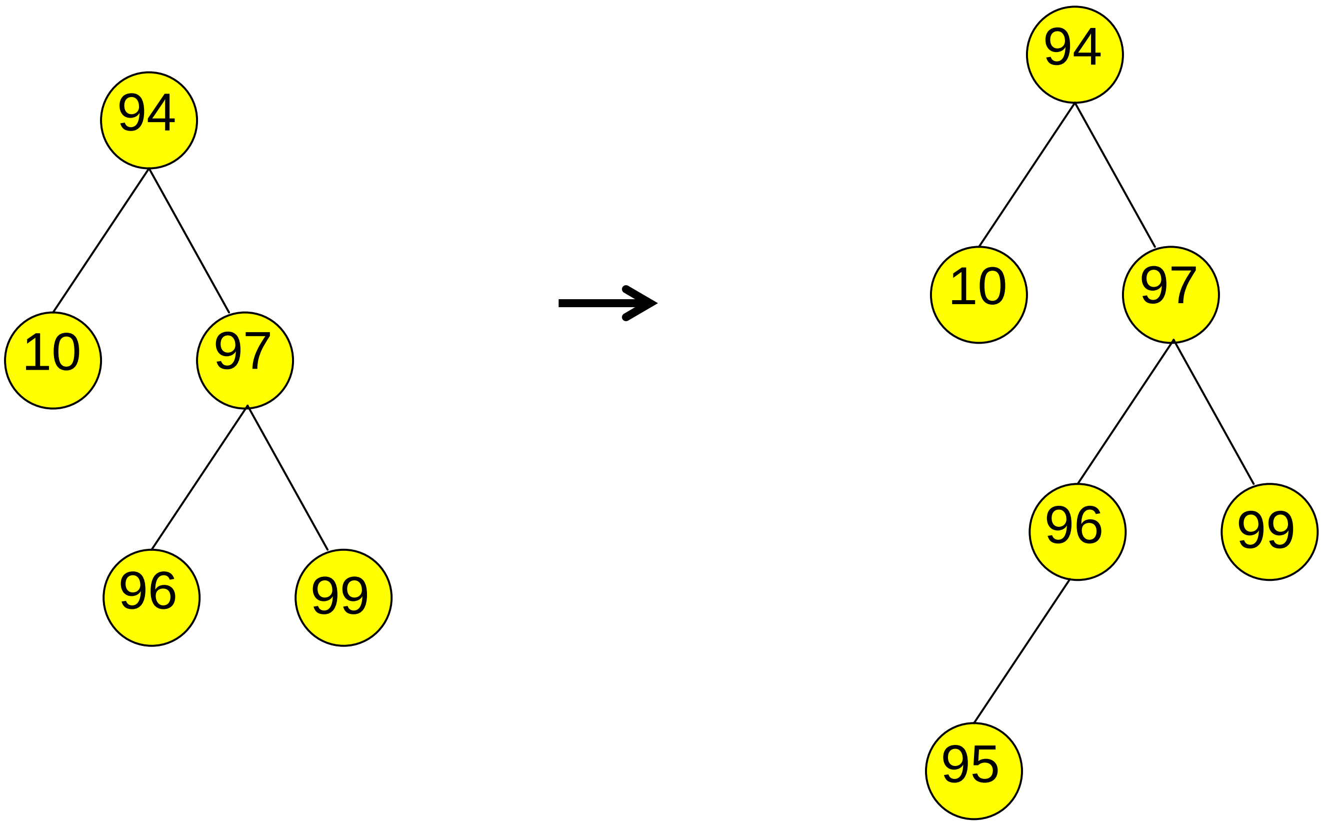
```
FindMin(T : tree pointer) : tree pointer {
// precondition: T is not null //
if T.left = null return T
 else return FindMin(T.left)
 }
```

Insert Operation

- `Insert(T: tree, X: element)`
  - › Do a “Find” operation for X
  - › If X is found → update (no need to insert)
  - › Else, “Find” stops at a NULL pointer
  - › Insert Node with X there
- Example: Insert 95



Insert 95



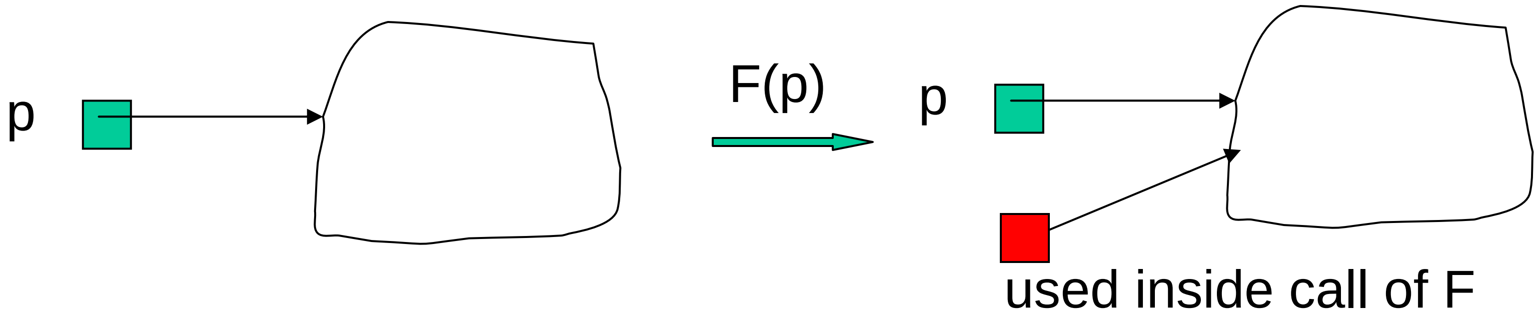
Recursive Insert

```
Insert(T : tree pointer, x : element) : tree pointer {
if T = null then
  T := new tree; T.data := x; return T;//the links to
                                       //children are null
case
  T.data > x : T.left := Insert(T.left, x);
  T.data < x : T.right := Insert(T.right, x);
  T.data = x : break;//Might throw an exception
endcase
}
```

Slight impediment: When a pointer to an object is passed as a parameter a copy of the pointer is made.
This is called “call-by value”

Call by Value vs Call by Reference

- Call by value
  - › Copy of parameter is used



- Call by reference
  - › Actual parameter is used

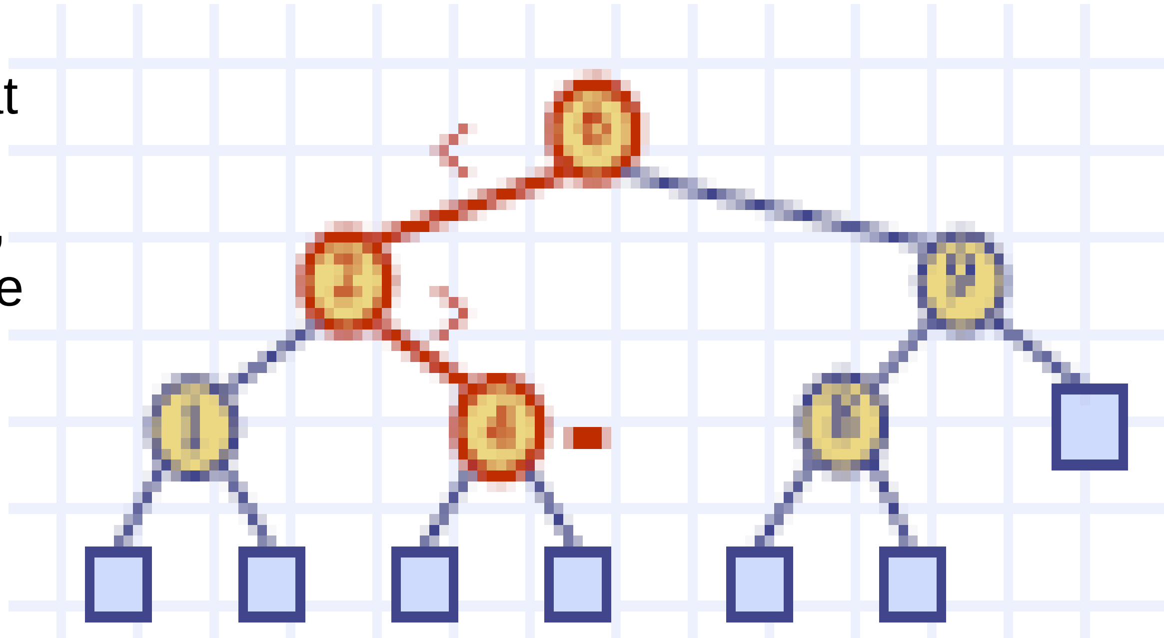
Insert Done with call-by-reference

```
Insert(T : reference tree pointer, x : element) : integer {
if T = null then
  T := new tree; T.data := x; return 1;//the links to
                                       //children are null
case
  T.data = x : return 0;
  T.data > x : return Insert(T.left, x);
  T.data < x : return Insert(T.right, x);
endcase
}
```

Advantage of reference parameter is that the call has the original pointer not a copy. But not available in Java

Binary search tree with external nodes

Each node that carries a key has 2 children, even if they are "null" children



For a tree with N keys, how many external nodes are needed?

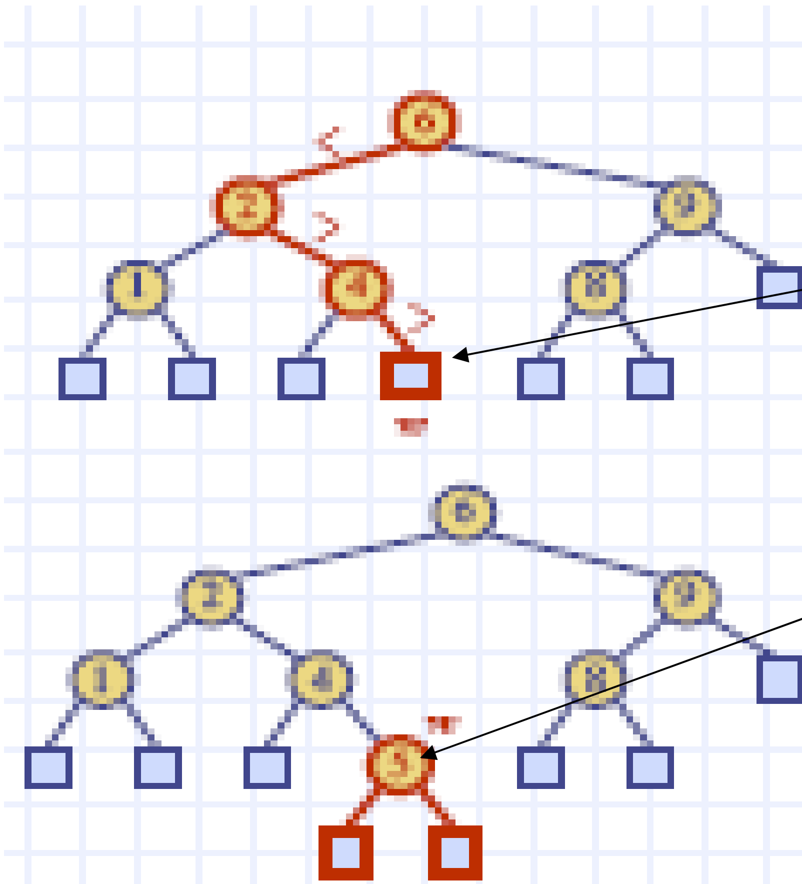
Drawbacks of external nodes

- Extra O(n) space
  - (in fact a little more than double the original!)
- For all practical purposes, have to discard external nodes for traversal, findmin etc…

Advantages of external nodes

- Easier to do insert
- Find the place of insertion
  - › It will be an external node, say v
- Replace the external node with an internal node (and 2 external nodes)

Insert with external nodes



Insert “5”

External node place of insertion

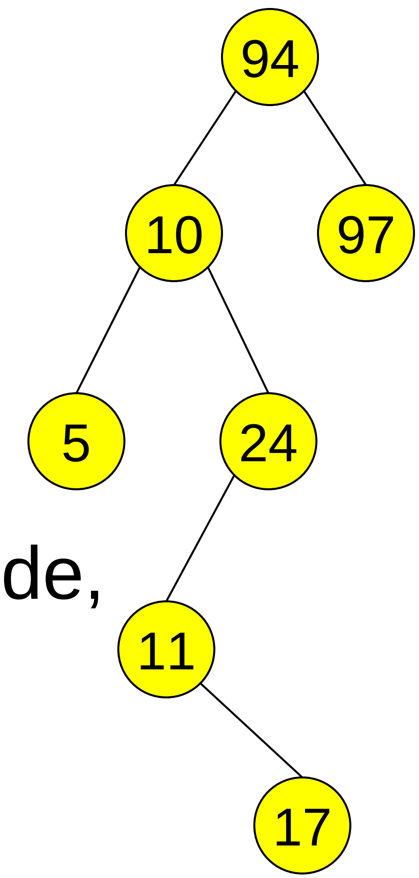
External node replaced by internal node and 2 external node children

Insert (keeping original root)

```
Insert (t : tree pointer, x: element){
//preconditions: tree not empty; element x not in the tree
if ( x < t.key) then {
    if (t.left = null then{  //found place of insertion
         new s; // the two children of s are null
         s.data := x;
         t.left := s;
         return}
     else Insert(t.left,x)}
else     {              // x > t.key
       //do same thing as above replacing left by right
         }
}
```

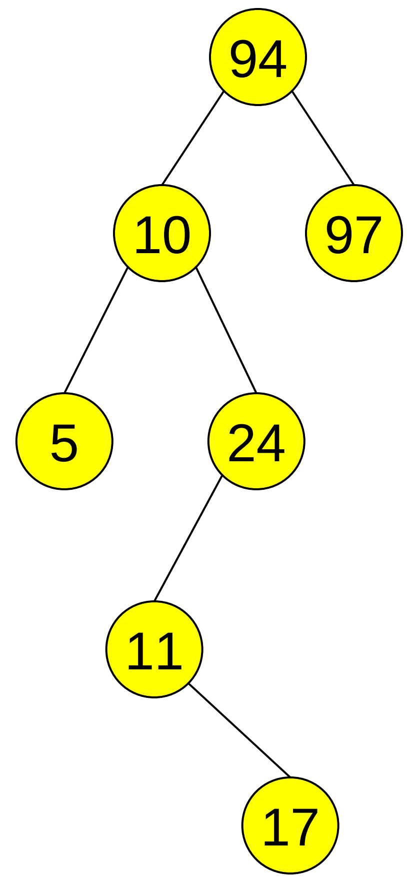
Delete Operation

- Delete is a bit trickier…Why?
- Suppose you want to delete 10
- Strategy:
  - › Find 10
  - › Delete the node containing 10
- Problem: When you delete a node, what do you replace it by?



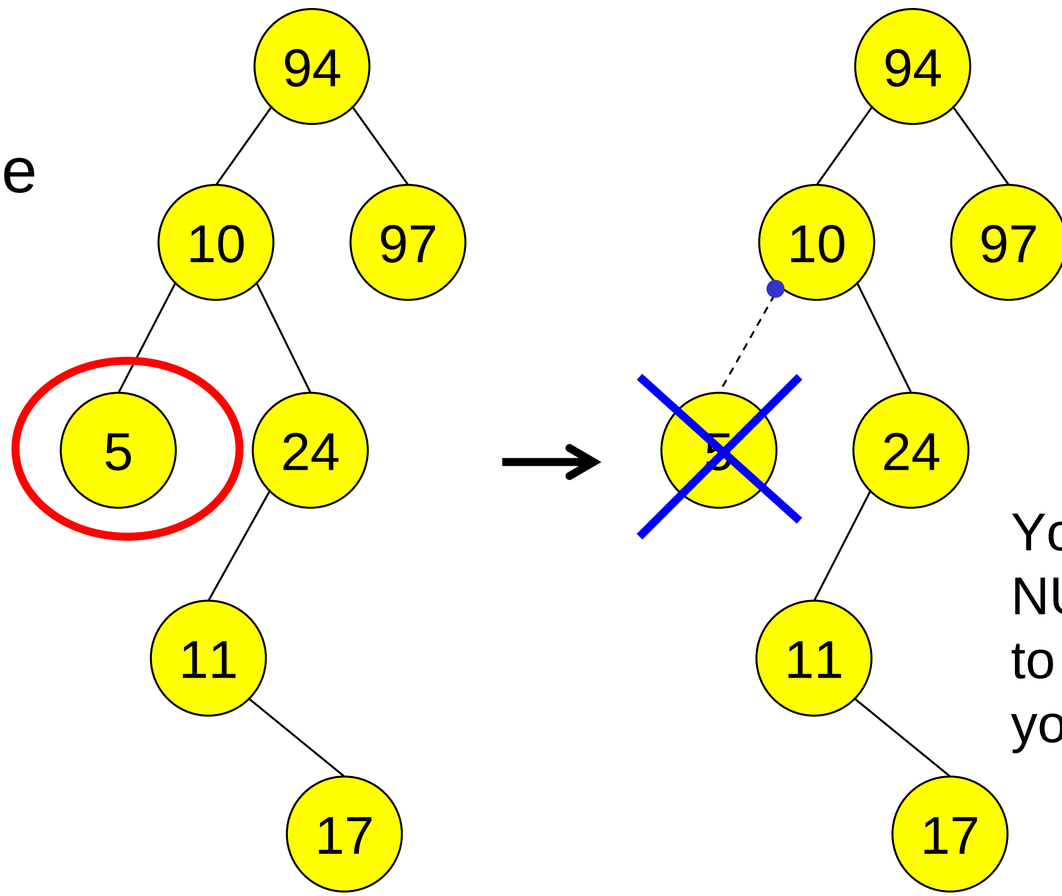
Delete Operation

- Problem: When you delete a node, what do you replace it by?
- Solution:
  - If it has no children, by NULL
  - If it has 1 child, by that child
  - If it has 2 children, by the node with the smallest value in its right subtree (the inorder successor of the node)



Delete “5” - No children

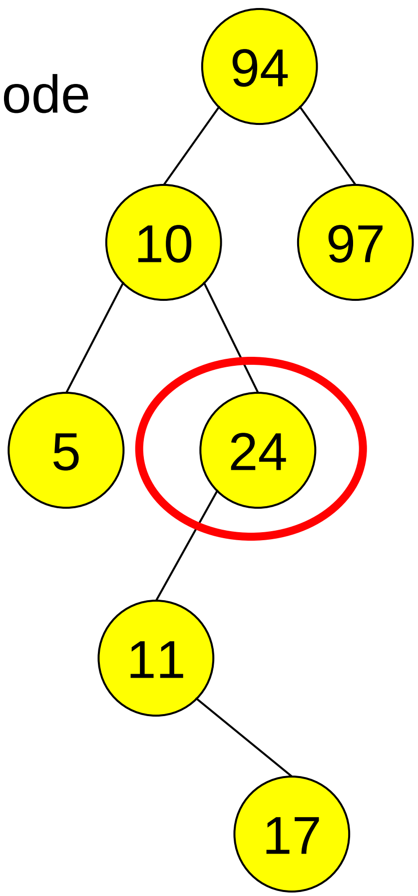
Find 5 node

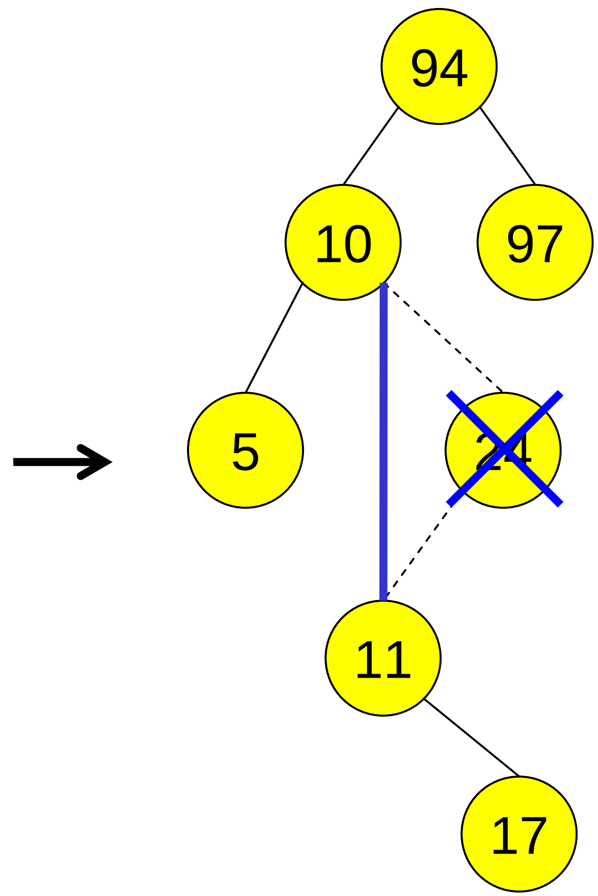


You need to NULL the pointer to the node that you are deleting

Delete “24” - One child

Find 24 node

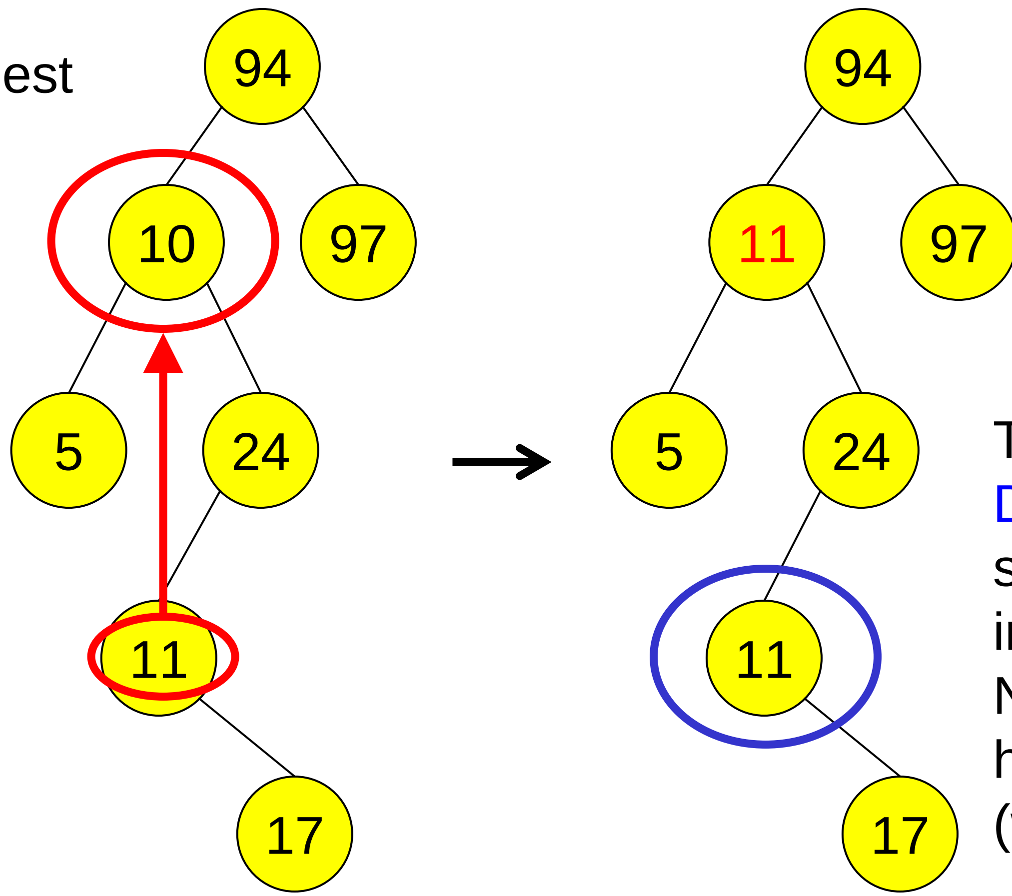




replace the pointer to the Deleted node with a pointer to its child

Delete "10" - two children

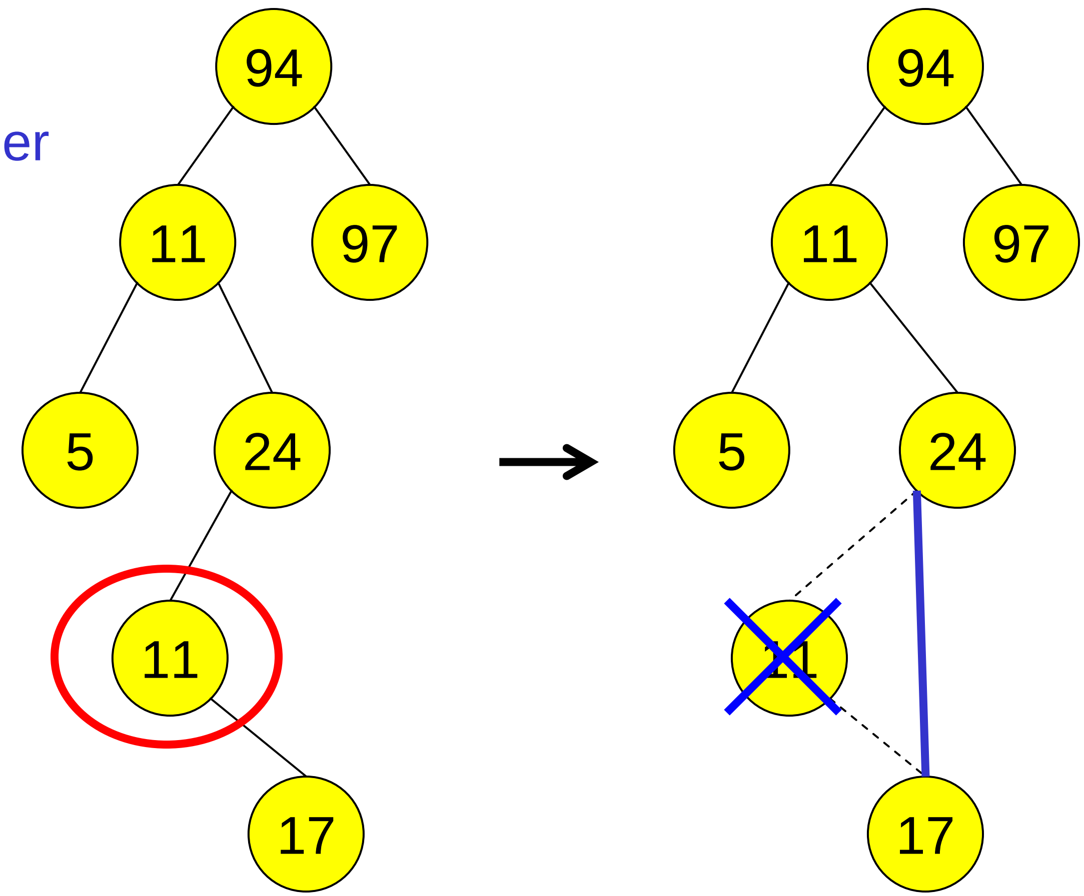
Find 10,
Copy the smallest value in right subtree into the node



Then (recursively) Delete node with smallest value in right subtree
Note: it cannot have two children (why?)

Then Delete “11” - One child

Remember
11 node



Then delete
the 11 node,i.e.,
replace the
pointer to it with
a pointer to its
child