One dimensional searching

Searching in an array

naive search, binary search, interpolation search

Binary search tree (BST)

operations Find, Insert, Delete

Naive search in a sorted array — linear, SLOW.

Array

Sorted array: →

Size = N

363	369	388	603	638	693	803	833	836	839	860	863	938	939	966	968	983	993

Find 993 !

Tests: N

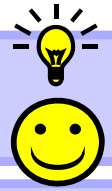
363	369	388	603	638	693	803	833	836	839	860	863	938	939	966	968	983	993

Find 363 !

Tests: 1

363	369	388	603	638	693	803	833	836	839	860	863	938	939	966	968	983	993

Search in a sorted array — binary, FASTER



1 test

Find 863 !

363	369	388	603	638	693	803	833	836	839	860	863	938	939	966	968	983	993
~~363~~	~~369~~	~~388~~	~~603~~	~~638~~	~~693~~	~~803~~	~~833~~	~~836~~	839	860	863	938	939	966	968	983	993

1 test

839	860	863	938	939	966	968	983	993
839	860	863	938	939	~~966~~	~~968~~	~~983~~	~~993~~

1 test

839	860	863	938	939
839	860	863	~~938~~	~~939~~

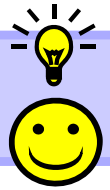
1 test

839	860	863
~~839~~	~~860~~	863

1 test

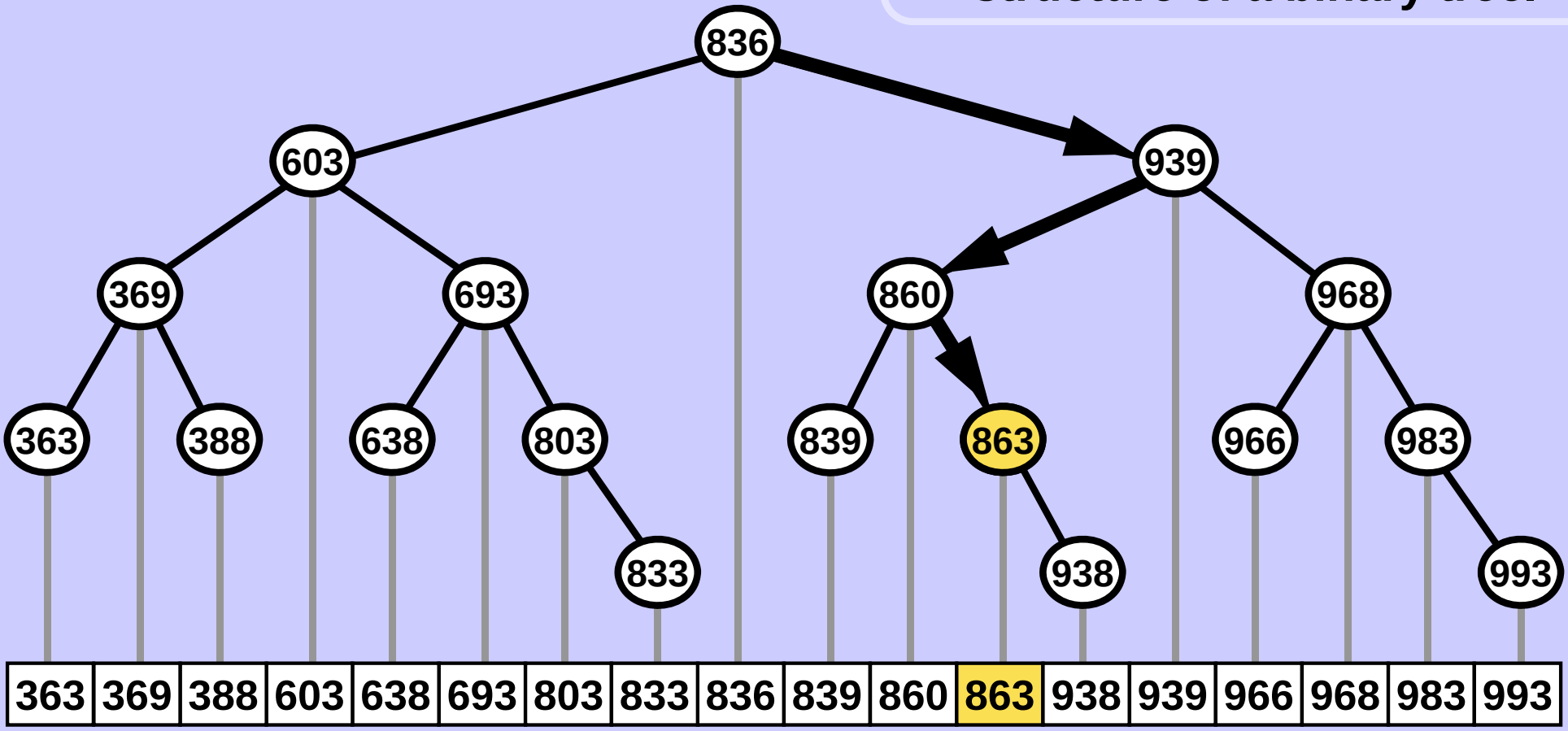
863

Search in a sorted array — binary, FASTER

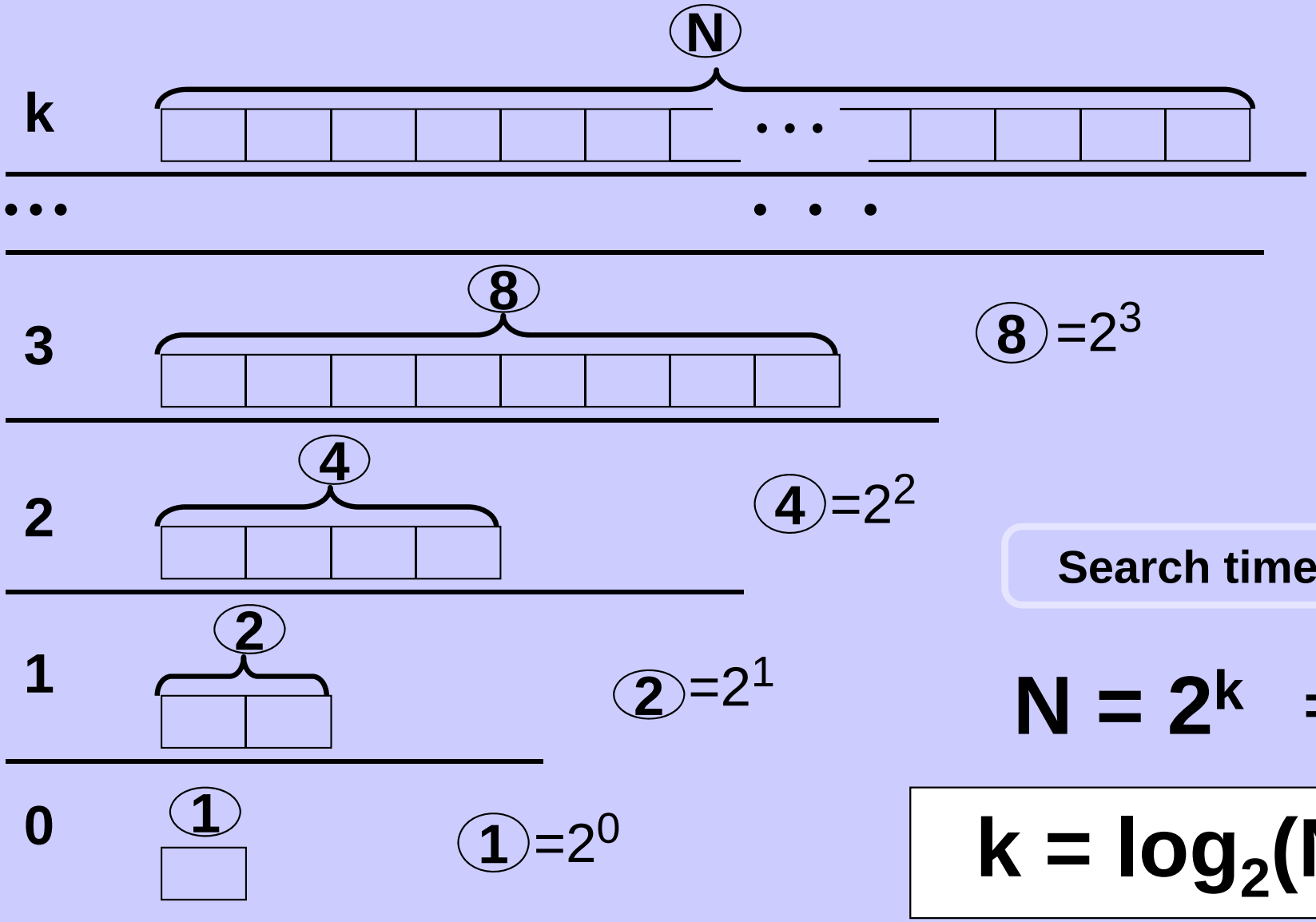


Find 863 !

The search follows the structure of a binary tree.



Search in a sorted array — binary, FASTER



Search time

$N = 2^k \Rightarrow$

$k = \log_2(N)$

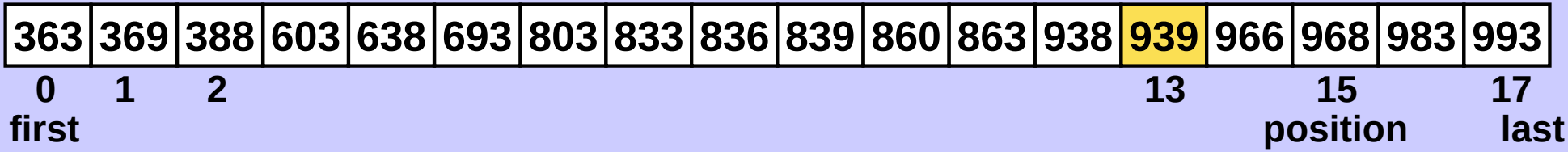
Binary search

```
def binarySearch(arr, value):
    i1 = 0; i3 = len(arr)-1
    while i1 < i3:
        i2 = (i1 + i3) // 2
        if arr[i2] < value: i1 = i2+1
        else:               i3 = i2
    if arr[i1] == value: return i1      # found or
    return -1                           # not found
```

Interpolation search

Array a[ ]

Find q = 939



When the values are expected to be more or less evenly distrubuted in the range the interpolation search might help.
The position of the element should roughly correspond to its value.

$$position = first + \frac{(q - a[first])}{a[last]-a[first]} * (last - first)$$

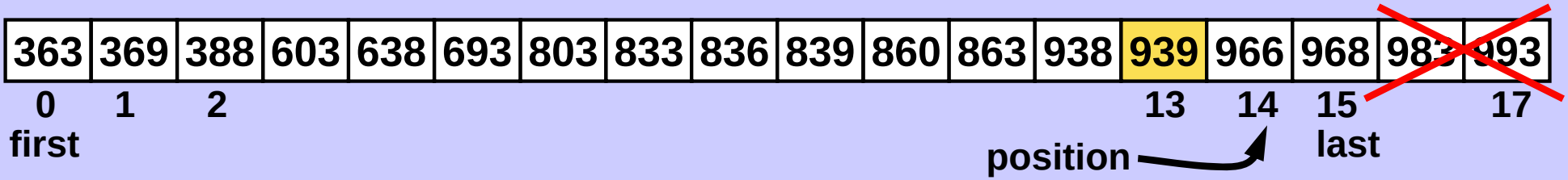
Example

$$position = 0 + \frac{939 - 363}{993 - 363} * (17 - 0) = 15.54$$

Interpolation search

Array a[ ]

Find q = 939



When the element is not found at the first hit then continue the search recursively in the part of the array which was not excluded form the search yet.

$$position = first + \frac{(q - a[first])}{a[last]-a[first]} * (last - first)$$

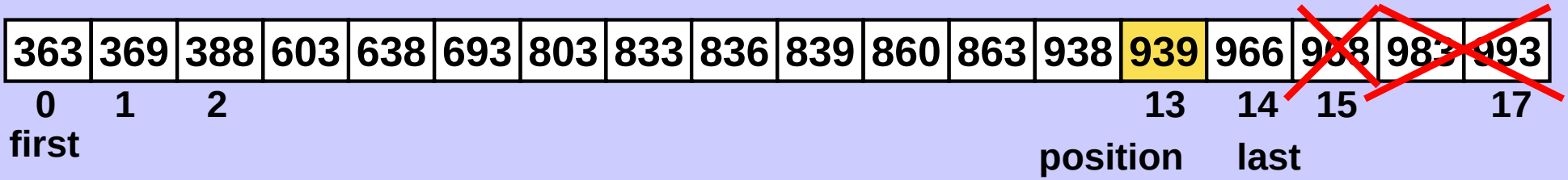
Example

$$position = 0 + \frac{939 - 363}{968 - 363} * (15 - 0) = 14.12$$

Interpolation search

Array a[ ]

Find q = 939



When the element is not found at the first hit then continue the search recursively in the part of the array which was not excluded form the search yet.

$$position = first + \frac{(q - a[first])}{a[last]-a[first]} * (last - first)$$

Example

$$position = 0 + \frac{939 - 363}{966 - 363} * (14 - 0) = 13.37$$

Finished.

Search in a sorted array — speed comparison

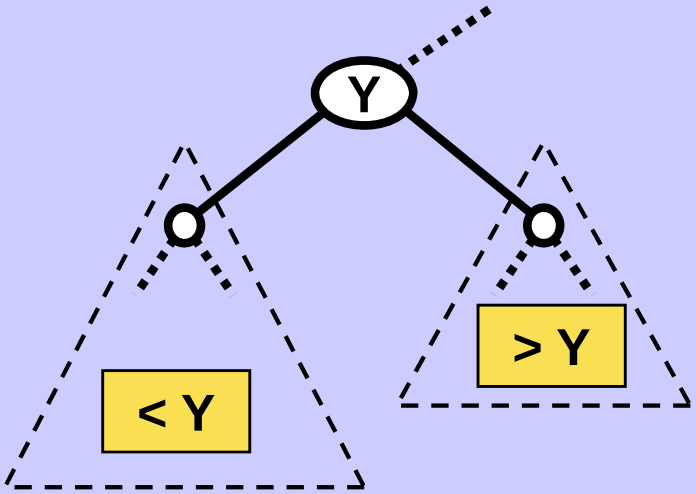
Array size N	Linear search average case	Interpolation search average case	Binary search cca average / worst case	
10	5.5	1.60	4	5
30	15.5	2.12	5	6
100	50.5	2.56	7	8
300	150.5	2.89	9	10
1 000	500.5	3.18	10	11
3 000	1 500.5	3.41	12	13
10 000	5 000.5	3.63	14	15
30 000	15 000.5	3.80	15	16
100 000	50 000.5	3.96	67	18
300 000	150 000.5	4.11	17	19
1 000 000	500 000.5	4.24	20	21
Asymptotic complexity	Obviously $\Theta(n)$	Random uniform distribution $\log_2(\log_2(N)) \in \Theta(\log(\log(N)))$	Due to the binary tree structure $\Theta(\log(n))$	

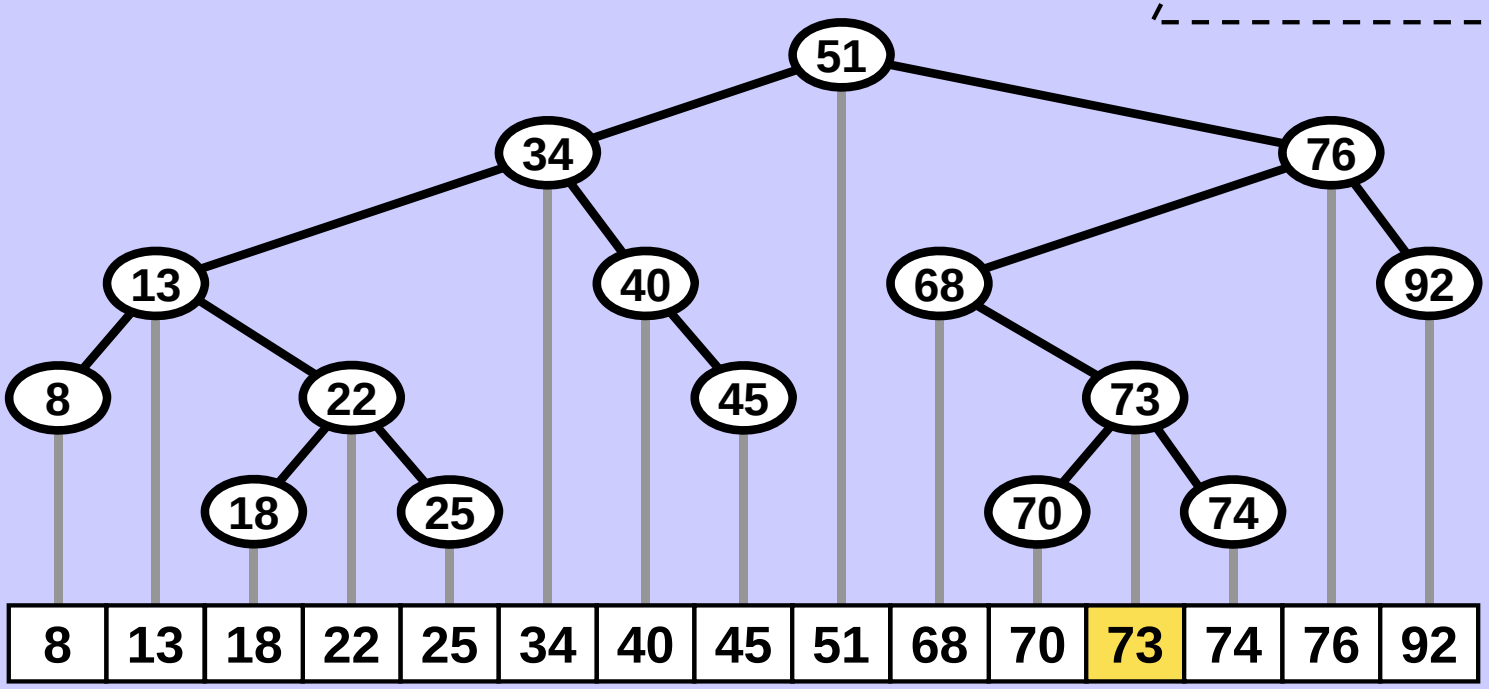
Binary search tree

For each node Y it holds:

Keys in the left subtree of Y are smaller than the key of Y.

Keys in the right subtree of Y are bigger than the key of Y.



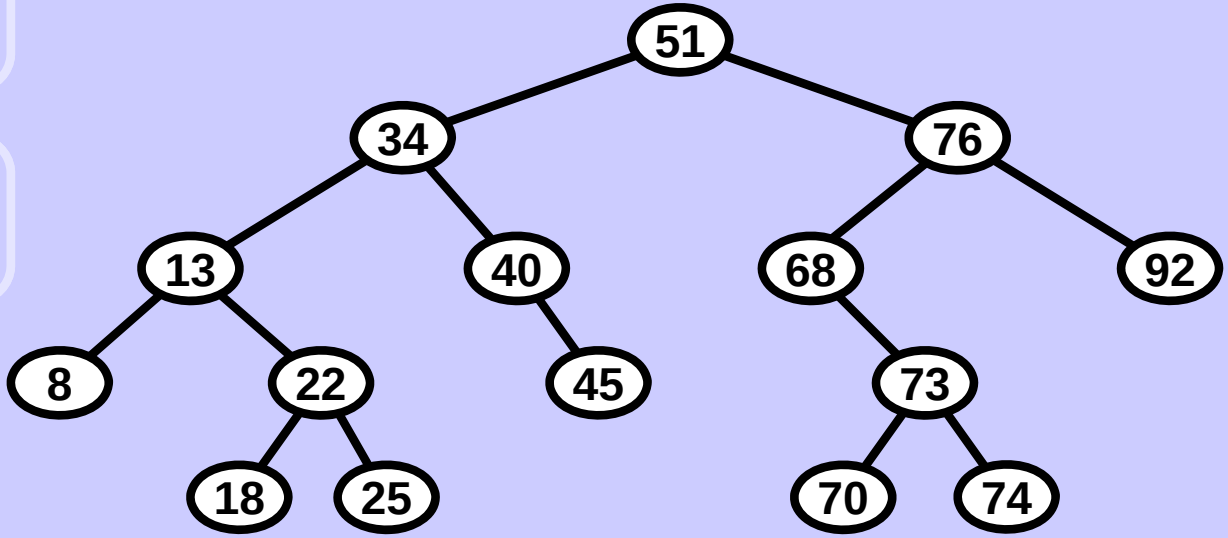


Binary search tree

BST may not be balanced and usually it is not.

BST may not be regular and usually it is not.

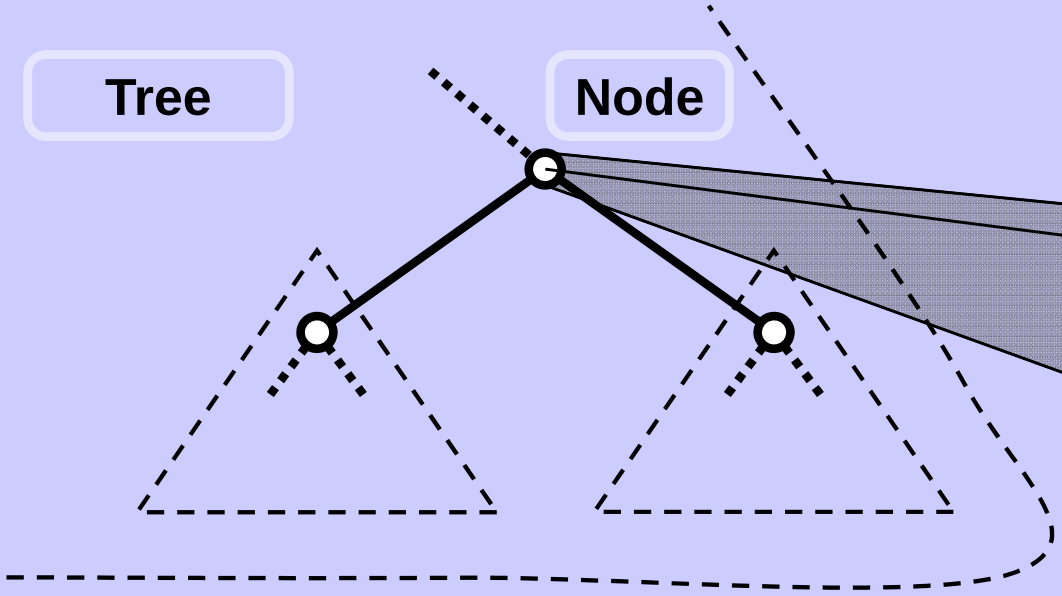
Apply the INORDER traversal to obtain sorted list of the keys of BST.



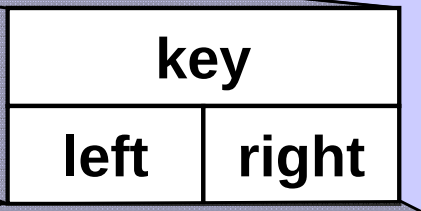
BST is flexible due to operations:

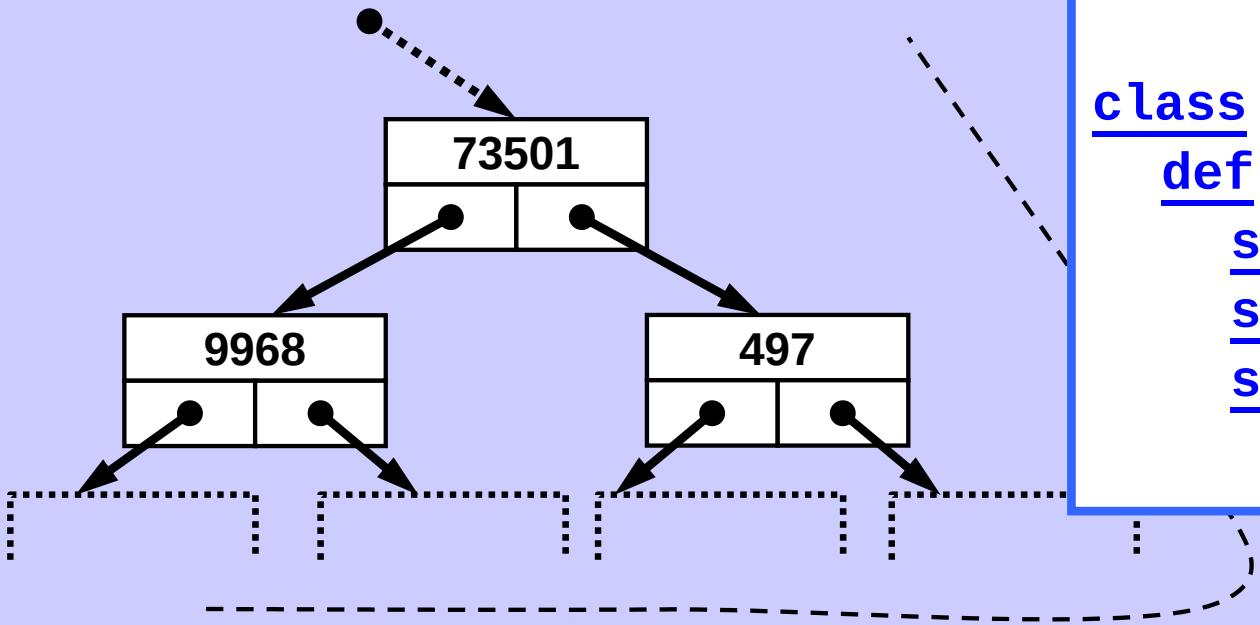
Find – return the pointer to the node with the given key (or null).
Insert – insert a node with the given key.
Delete – (find and) remove the node with the given key.

Binary search tree implementation -- Python



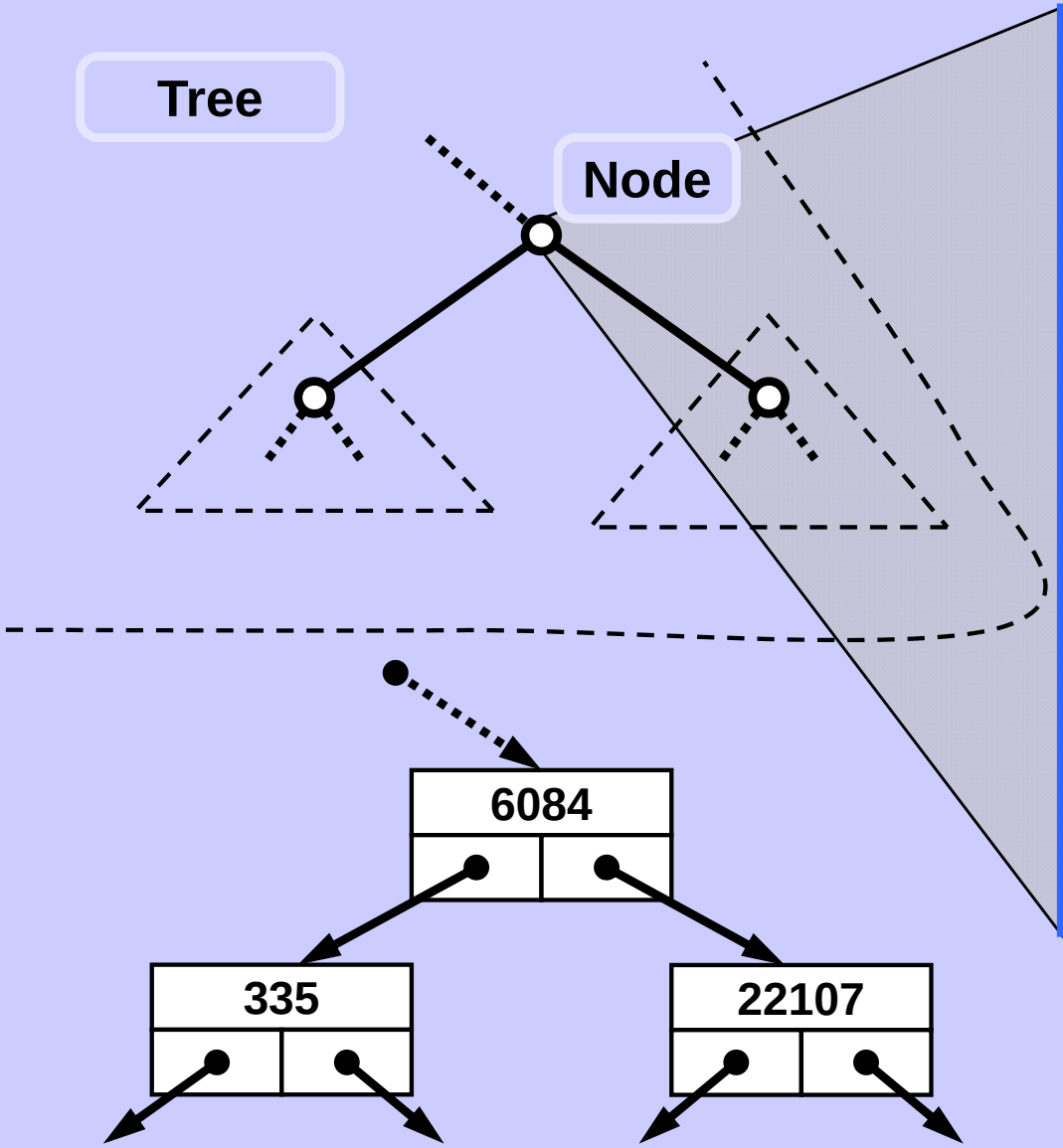
Node representation






```
class Node:
  def __init__(self, key):
    self.left = None
    self.right = None
    self.key = key
```

Binary search tree implementation -- Python

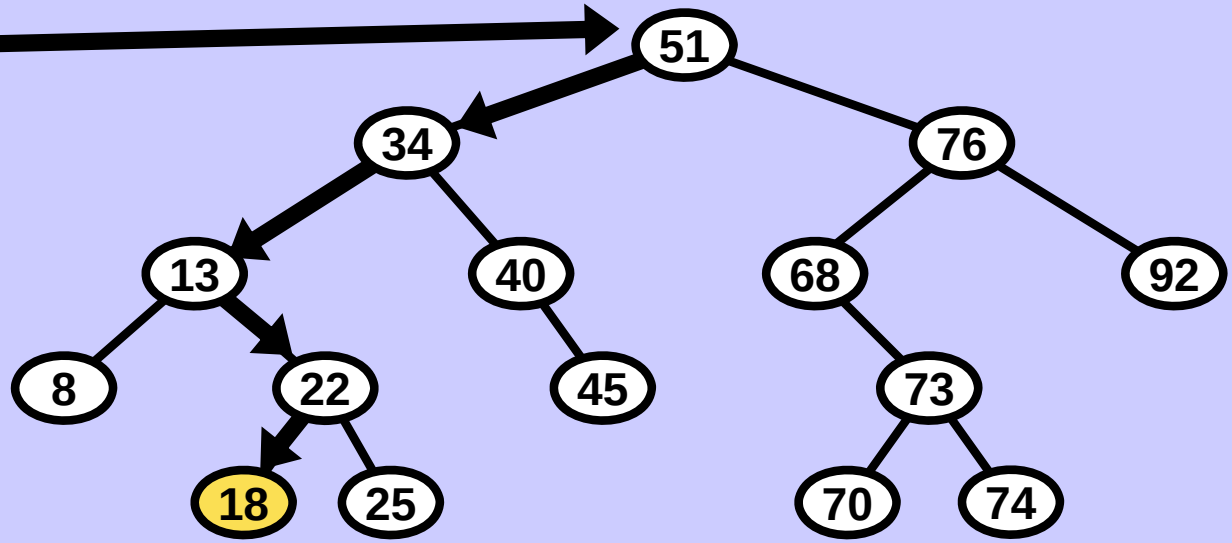



```
class Node:
   def __init__(self, key):
      self.left = None
      self.right = None
      self.key = key


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

Operation Find in BST

Find 18



Each BST operation starts in the root.

Iteratively

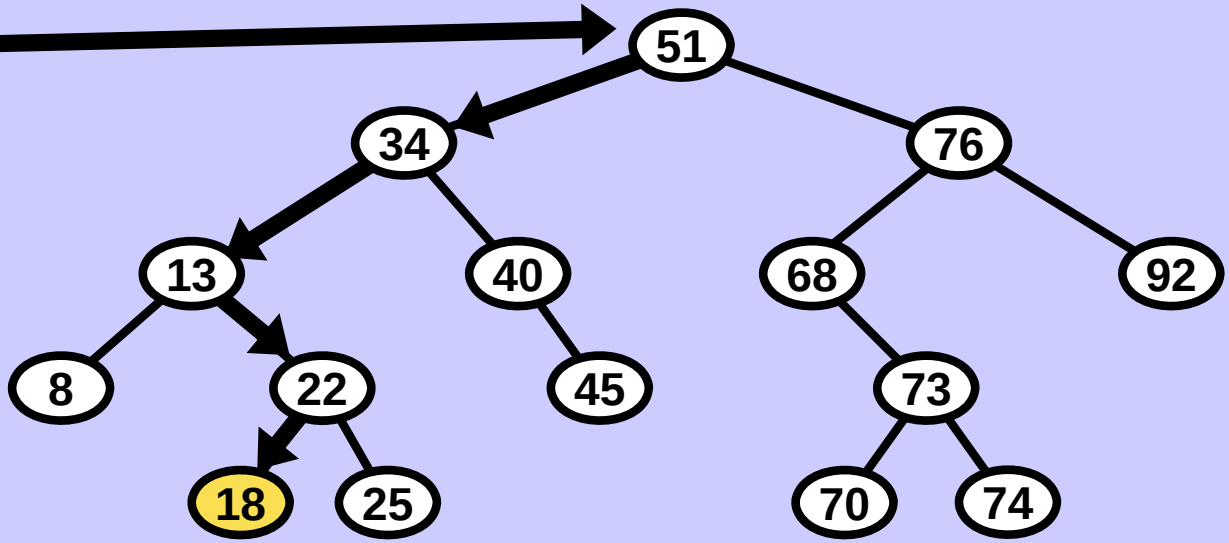
```
def FindIter(self, key, node):
    while(True):
        if node == None     : return None
        if key == node.key  : return node
        if key < node.key   : node = node.left
        else                : node = node.right

FindIter(key, tree.root)  # call
```

Operation Find in BST

Find 18

Each BST operation starts in the root.

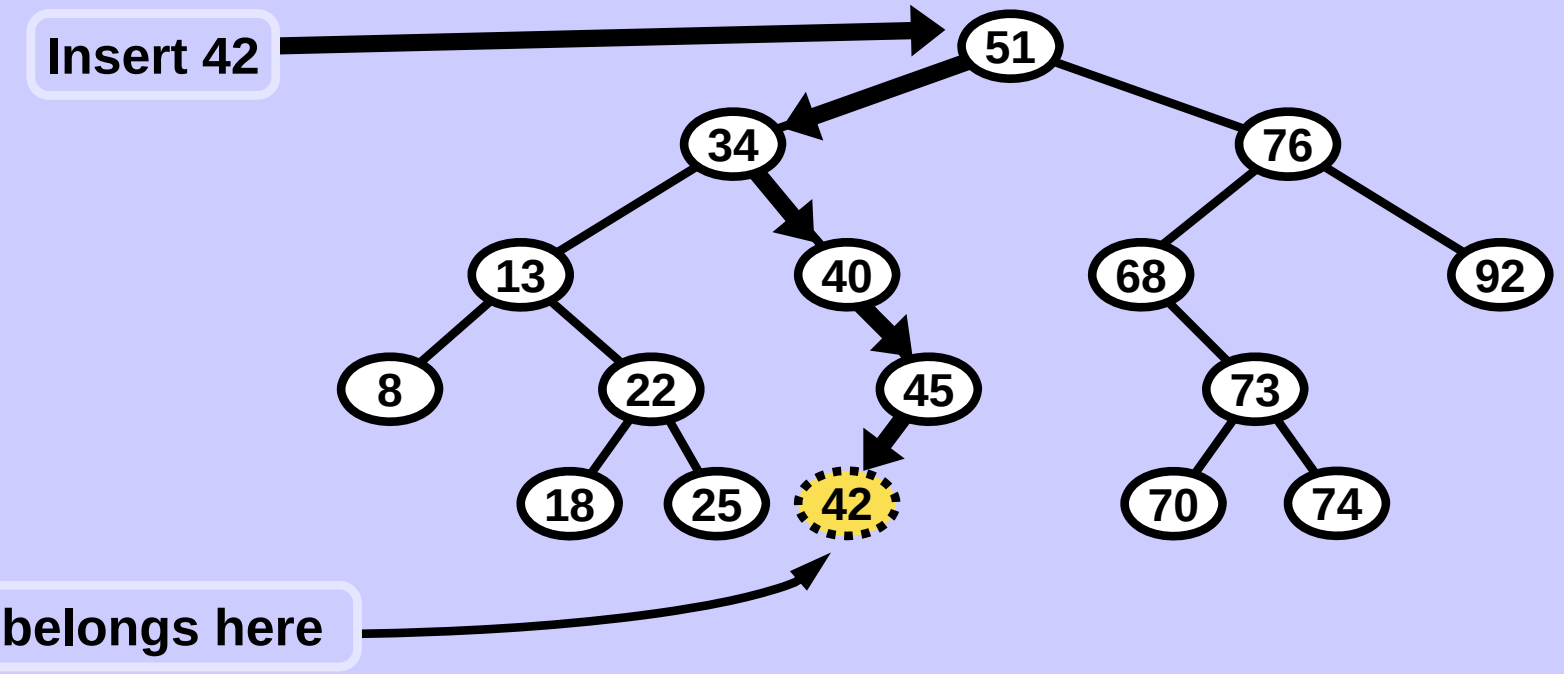


Recursively

```
def Find(self, key, node):
    if node == None     : return None
    if key == node.key  : return node
    if key < node.key   : self.Find(key, node.left)
    else                : self.Find(key, node.right)

Find(key, tree.root)  # call
```

Operation Insert in BST



Insert
1. Find the place (like in Find) for the leaf where the key belongs.
2. Create this leaf and connect it to the tree.

Operation Insert in BST iteratively

```
def InsertIter(self, key):
    if self.root == None:
        self.root = Node(key);
        return self.root

    node = self.root
    while True:
        if key == node.key: return None # no duplicates
        if key < node.key:
            if node.left == None:
                node.left = Node(key); return node.left
            else: node = node.left
        else:
            if node.right == None:
                node.right = Node(key); return node.right
            else: node = node.right
```

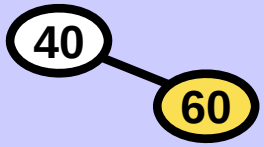
Operation Insert in BST recursively

```
def Insert(self, key, node):
    if key == node.key: return   # no duplicates
    if key < node.key:
        if node.left == None:  node.left = Node(key)
        else:                  self.Insert(key, node.left)
    else:
        if node.right == None: node.right = Node(key)
        else:                  self.Insert(key, node.right)
```

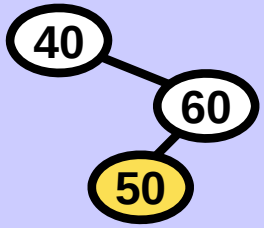
Building BST by repeated Insert

insert 40

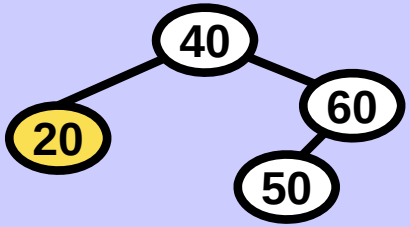
insert 60



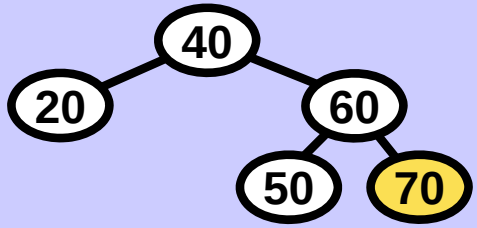
insert 50



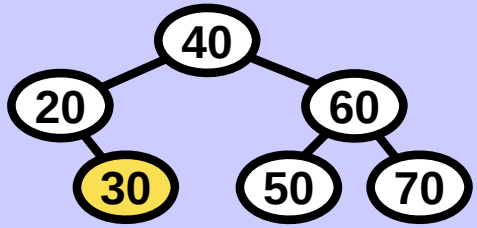
insert 20



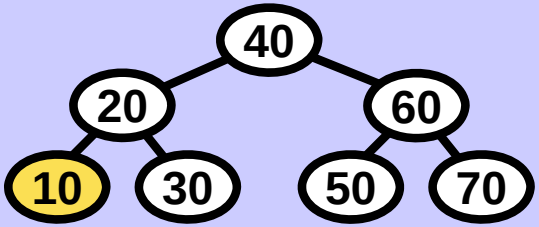
insert 70



insert 30



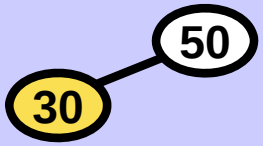
insert 10



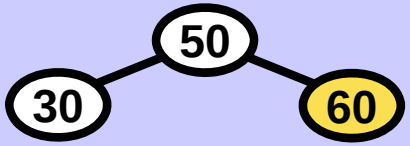
The shape of the BST depends on the order in which data are inserted.

insert 50

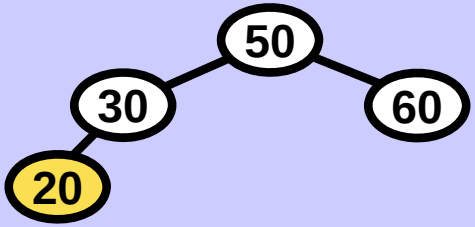
insert 30



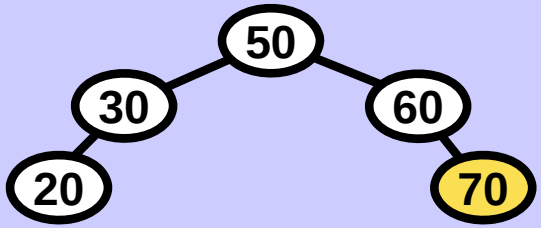
insert 60



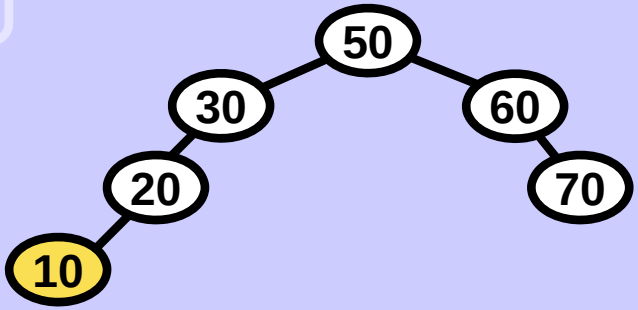
insert 20



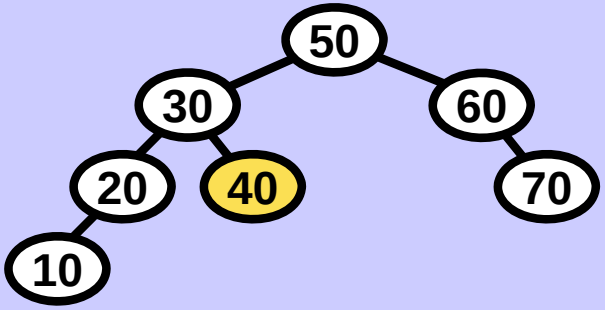
insert 70



insert 10



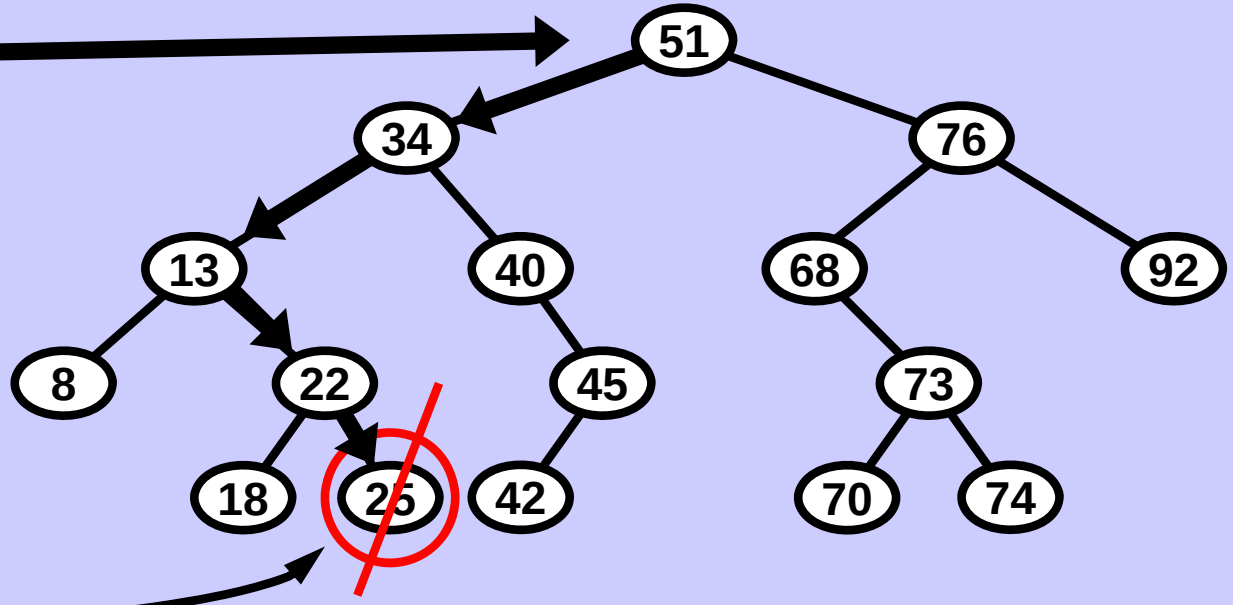
insert 40



Operation Delete in BST (I.)

Delete a node with 0 children (= leaf)

Delete 25

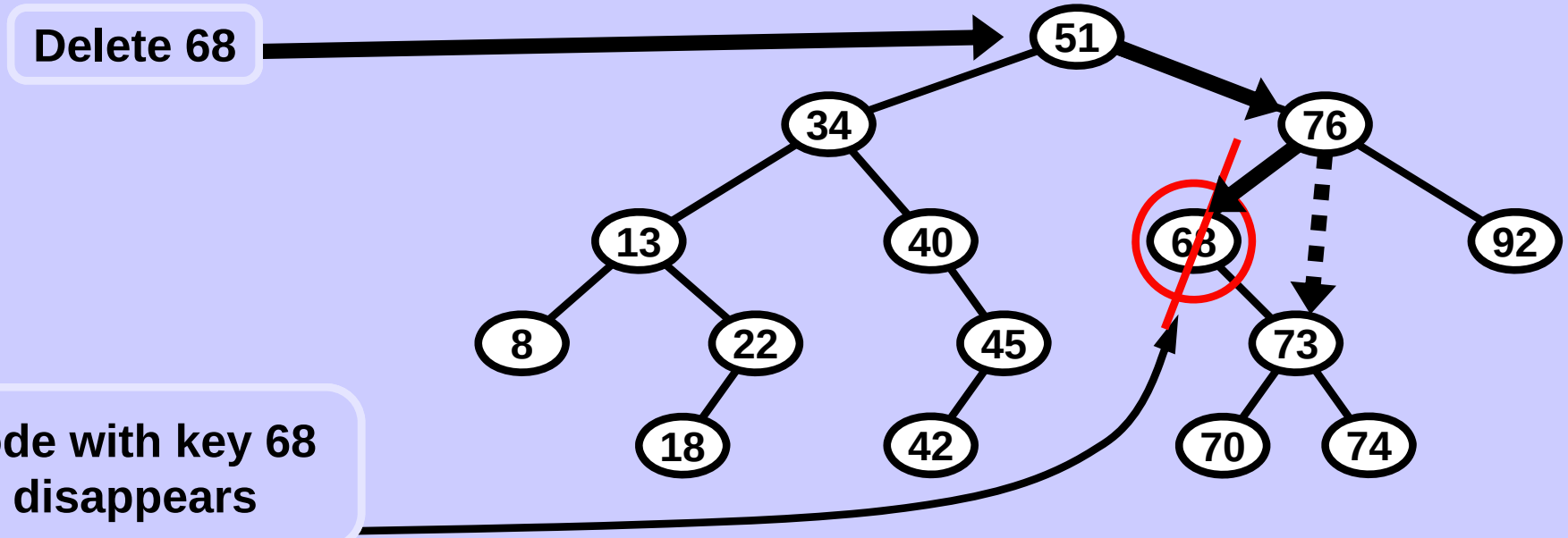


Leaf with key 25 disappears

Delete I. Find the node (like in Find operation) with the given key and set the reference to it from its parent to null.

Operation Delete in BST (II.)

Delete a node with 1 child.



Change the 76 --> 68 reference to 76 --> 73 reference.

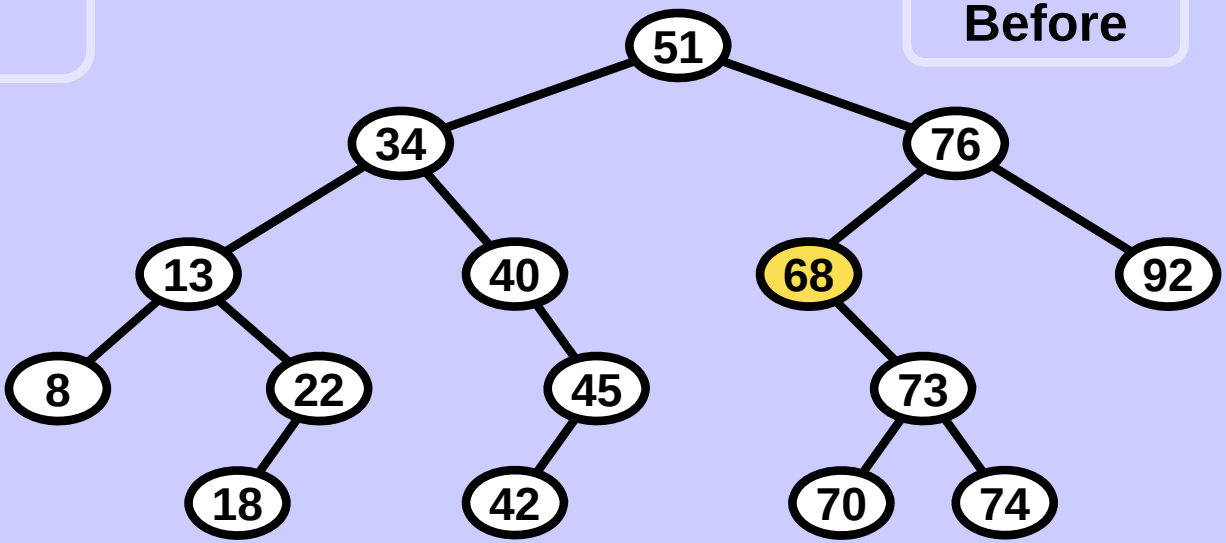
Delete II. Find the node (like in Find operation) with the given key and set the reference to it from its parent to its (single) child.

Operation Delete in BST (II.)

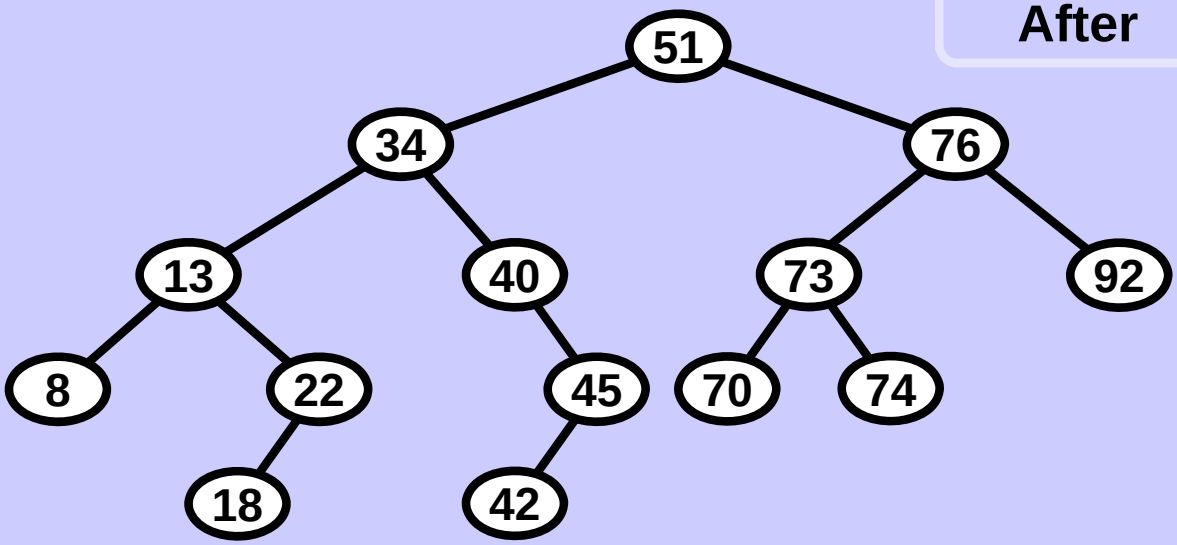
Delete a node with 1 child.

Delete 68

Before

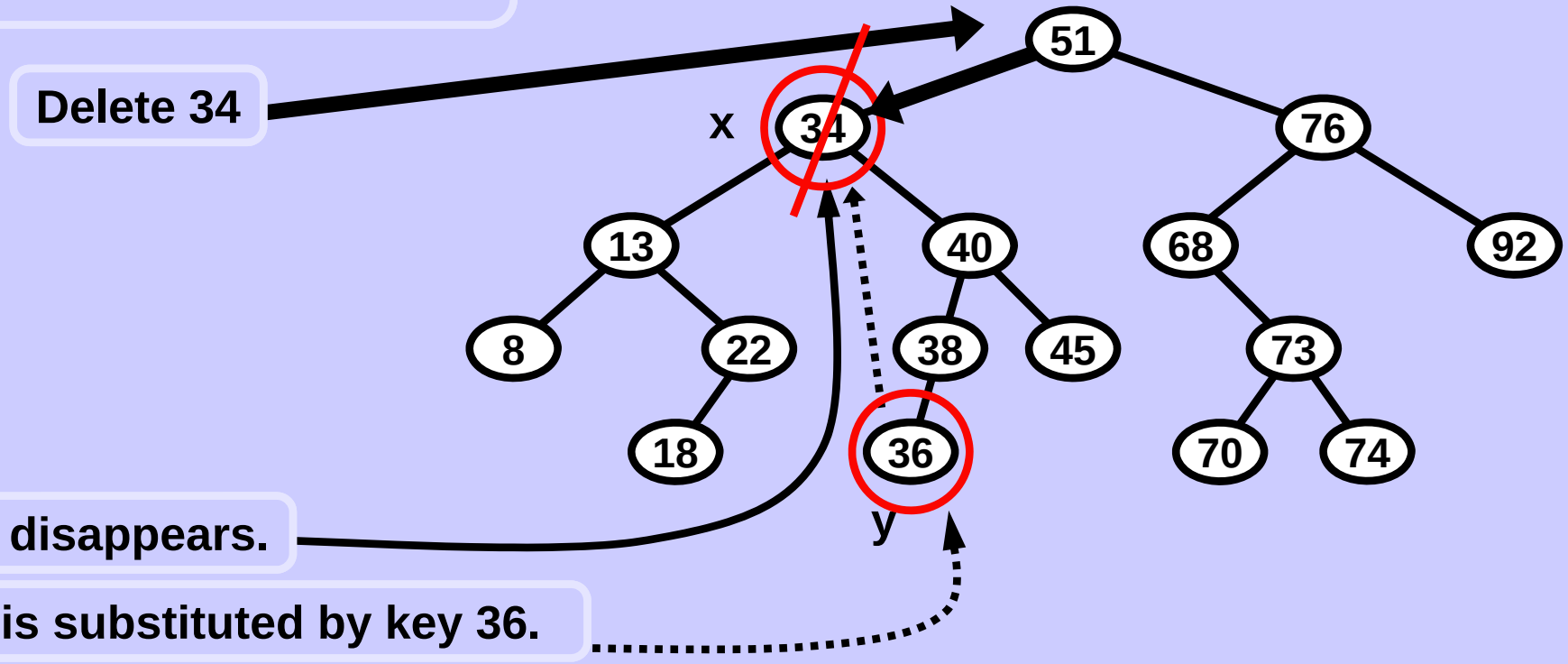


After



Operation Delete in BST (IIIa.)

Delete a node with 2 children.



Delete IIIa.

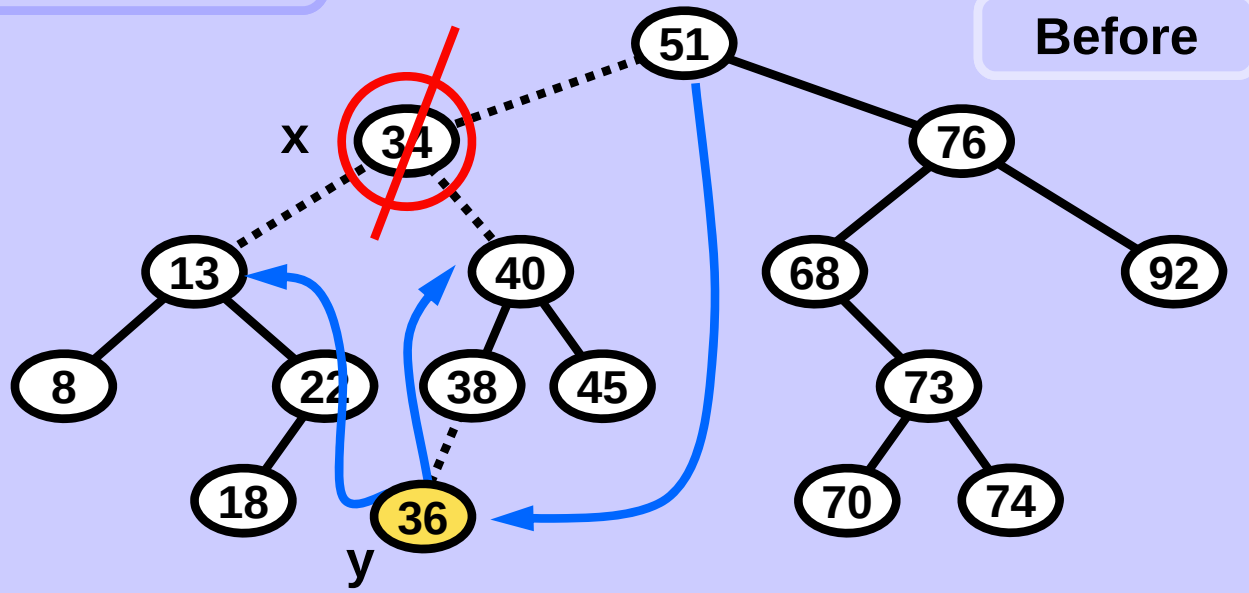
1. Find the node (like in Find operation) with the given key and then find the ***leftmost*** (= smallest key) node y in the ***right*** subtree of x.
2. Point from y to children of x, from parent of y point to the child of y instead of y, from parent of x point to y.

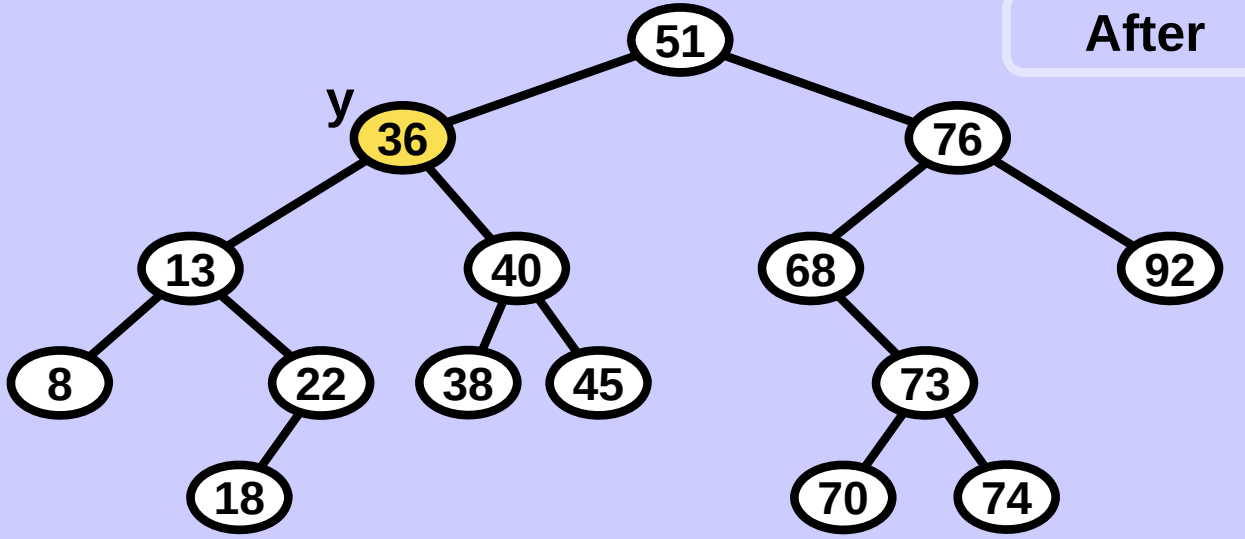
Operation Delete in BST (IIIa.)

Delete 34

old edges/pointers/references

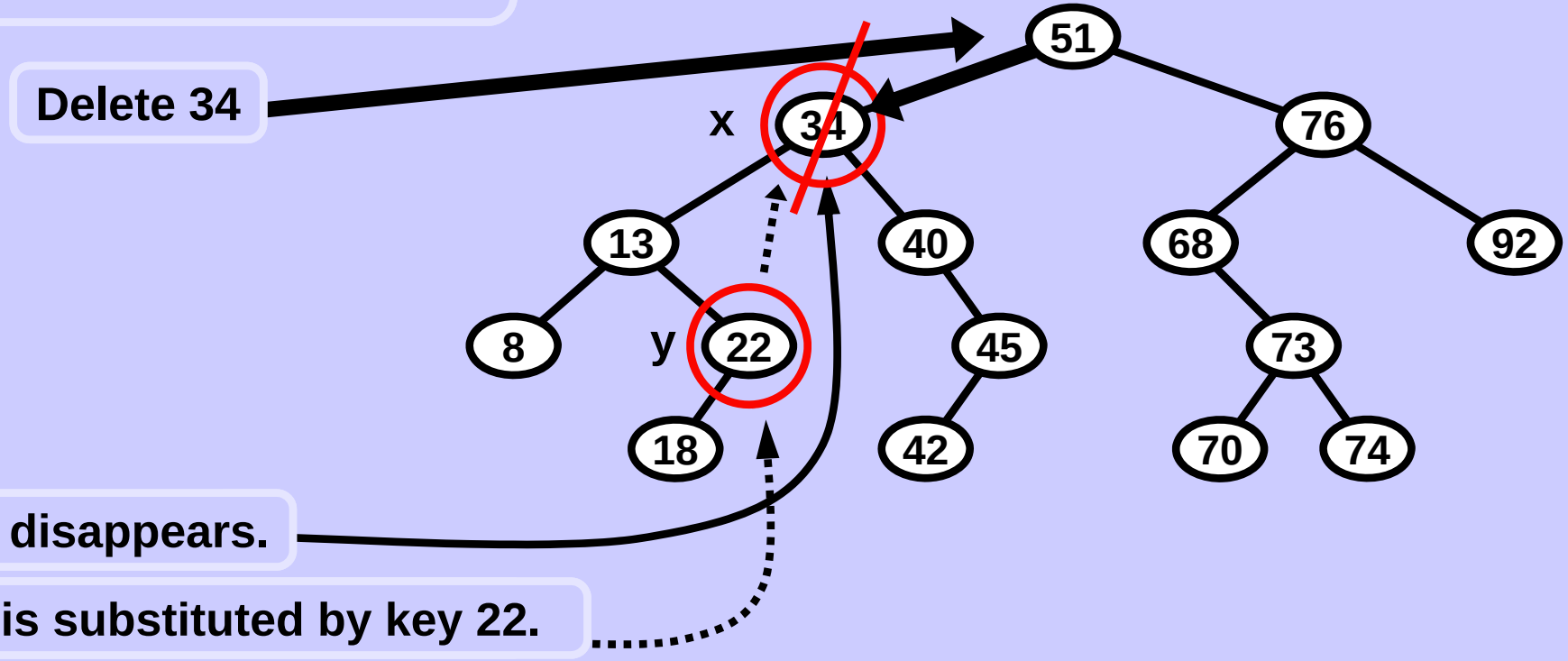
new edges/pointers/references





Operation Delete in BST (IIIb.) is equivalent to Delete IIIa.

Delete a node with 2 children.



Delete IIIb.

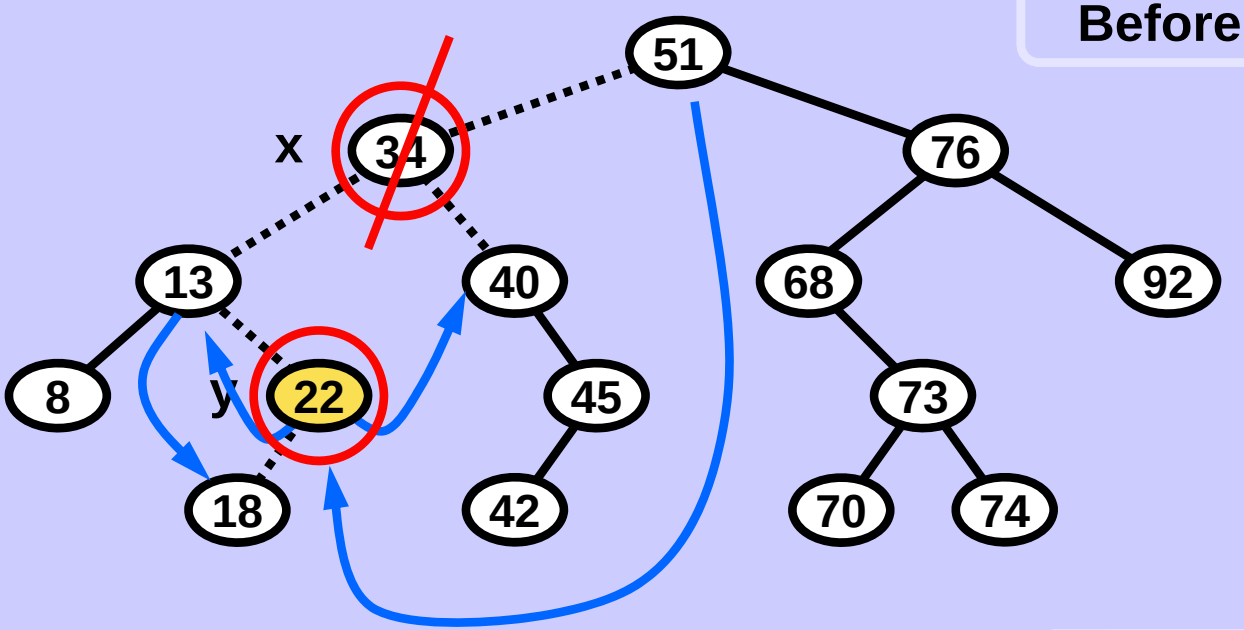
1. Find the node (like in Find operation) with the given key and then find the ***rightmost*** (= smallest key) node y in the ***left*** subtree of x.
2. Point from y to children of x, from parent of y point to the child of y instead of y, from parent of x point to y.

Operation Delete in BST (IIIb.) is equivalent to Delete IIIa.

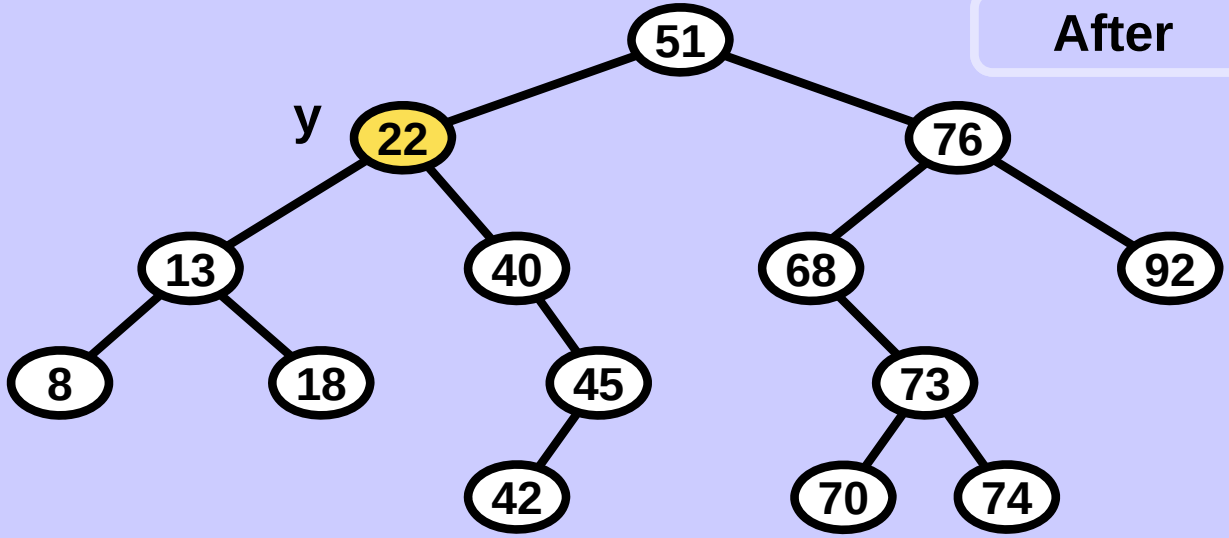
Delete 34

old edges/pointers/references

new edges/pointers/references



The moved node may itself have a child. In such case apply to it the variant Delete II.



Operation Delete in BST

```
def Delete (self, key):
  ...                                  # homework...
```

Asymptotic complexities of operations Find, Insert, Delete in BST

	BST with n nodes	
Operation	**Balanced**	**Maybe not balanced**
Find	$O(\log(n))$	$O(n)$
Insert	$\Theta(\log(n))$	$O(n)$
Delete	$\Theta(\log(n))$	$O(n)$