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

Size = N

Sorted array: →

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

Find 863 !

863 > 836? — 1 test

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

863 > 939? — 1 test

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

863 > 863? — 1 test

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

863 > 860? — 1 test

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

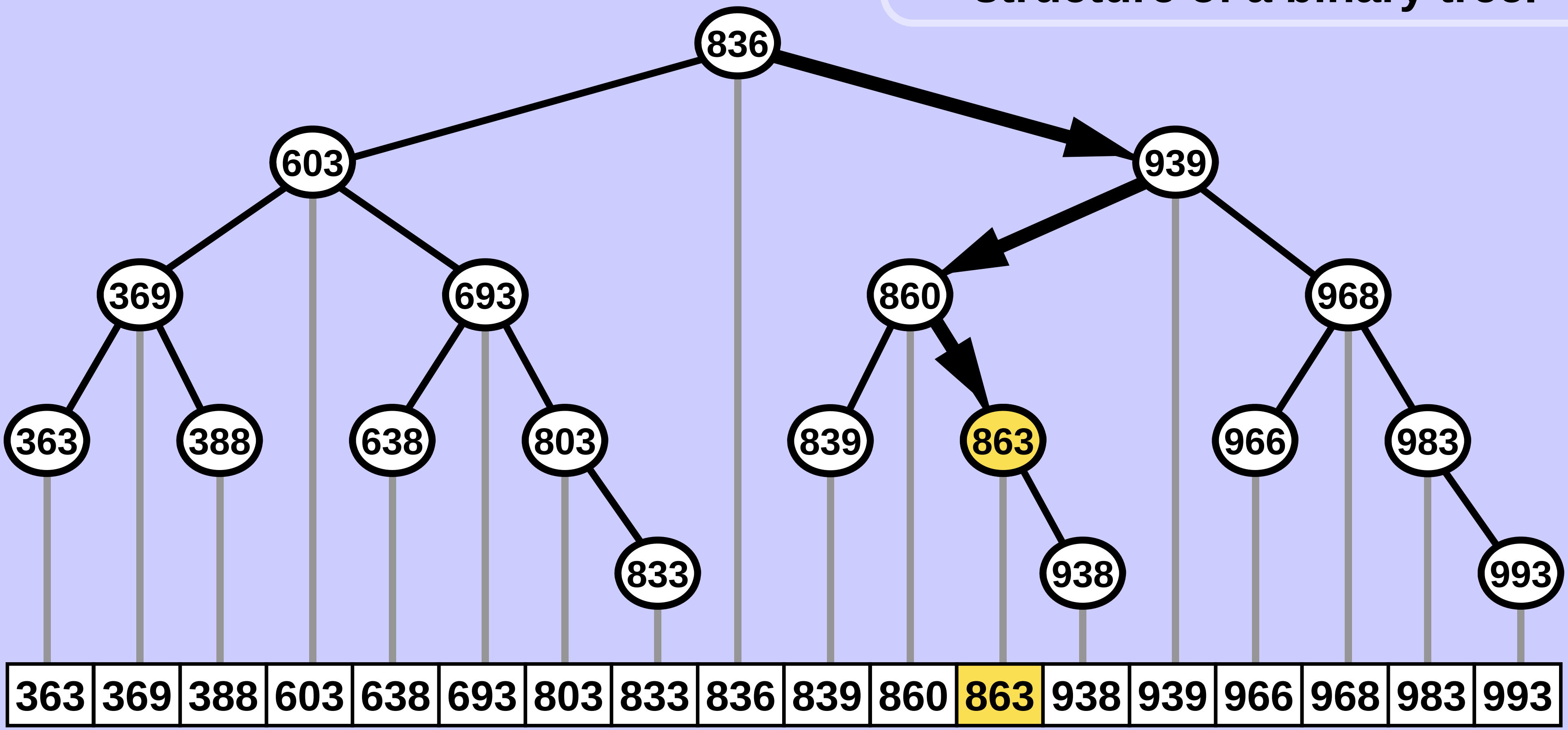
863 == 863 — 1 test

863 !

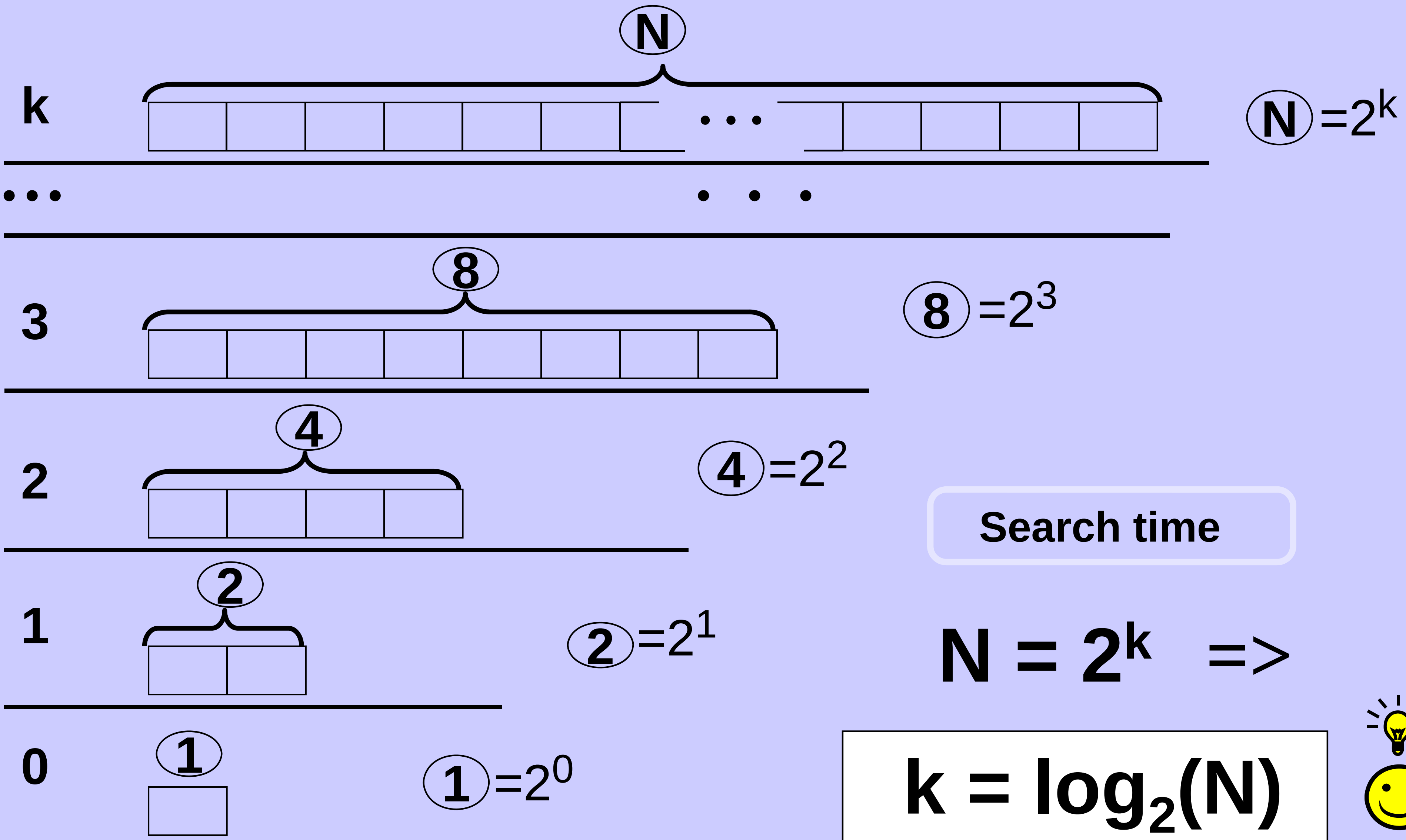
Search in a sorted array — binary, FASTER

Find 863 !

The search follows the structure of a binary tree.



Search in a sorted array — binary, FASTER



Search time

$$N = 2^k \quad =>$$

$$k = \log_2(N)$$

Search in a sorted array — binary, FASTER

Find q = 863 !

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
363	369	388	603	638	693	803	833	836	839	860	863	938	939	966	968	983	993

839	860	863	938	939	966	968	983	993

839	860	863	938	939

839	860	863

863

Typically, on average, the query value is close to a leaf in the search tree.

It takes too much time to test each value in the tree during the search descent to the leaf.

Therefore, the method first finds the exact place where the query value should be located and only then it checks if the value is really there.

5 tests in total

Though the query value 863 was *encountered* already in the 3rd check, its presence in the array was *confirmed* only after the 4th test.

During the search, the current segment is divided to two halves and the unpromissing half is discarded. The final test "Is q in the array?" is performed only once, when the current segment length is 1.

Binary search -- fast variant

```
def binarySearch( arr, value ):
    low = 0; high = len(arr)-1
    while low < high:                # while segment length > 1
        mid = (low + high) // 2      # bug ?
                                 # fix: mid = low + (high-low)/2;
        if arr[mid] < value: low  = mid+1
        else:                high = mid
    if arr[low] == value: return low     # found or
    return -1                            # not found
```

Bug? : When low + high > INT_MAX in some languages overflow appears
https://research.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html

Interpolation search

Array a[ ]

Find q = 939

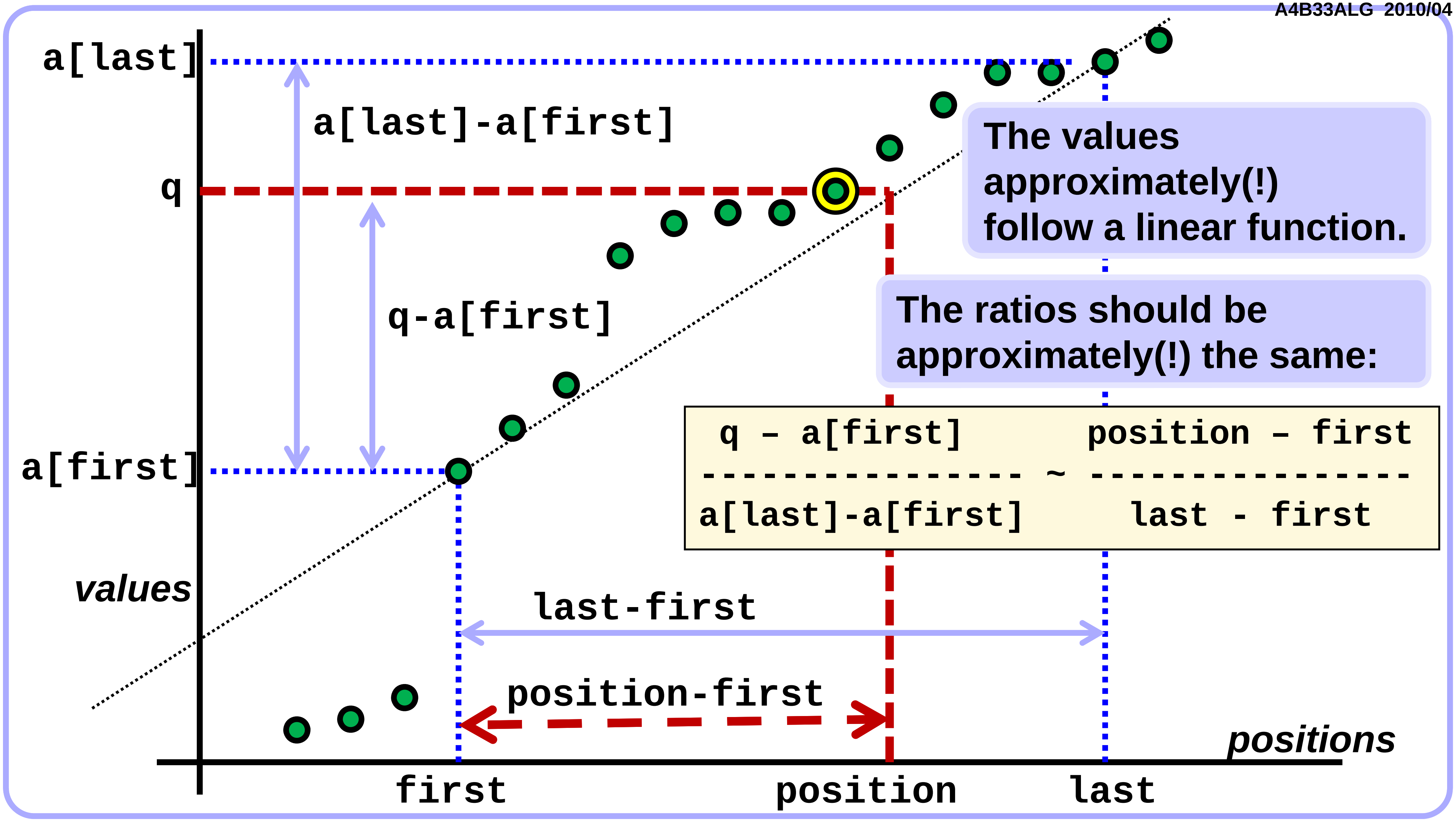
363	369	388	603	638	693	803	833	836	839	860	863	938	**939**	966	968	983	993
0	1	2											13		15		17
first															position		last

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

```
                          (q – a[first])
position = first + ---------------- *(last – first)
                         a[last]-a[first]
```

```
                    939 - 363
position = 0 + ----------- *(17 – 0) = 15.54
                    993 - 363
```

Example



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

Interpolation search

Array a[ ]

Find q = 939

363	369	388	603	638	693	803	833	836	839	860	863	938	939	966	968	~~983~~	~~993~~
0	1	2											13	14	15		17
first														position	last		

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.

```
                        (q – a[first])
position = first + ----------------- *(last – first)
                       a[last]-a[first]
```

Example

```
                 939 - 363
position = 0 + ----------- *(15 – 0) = 14.12
                 968 - 363
```

Interpolation search

Array a[ ]

Find q = 939

363	369	388	603	638	693	803	833	836	839	860	863	938	**939**	966	~~968~~	~~983~~	~~993~~
0	1	2											13	14	15		17
first													position	last			

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.

```
                          (q – a[first])
position = first + ----------------- *(last – first)
                         a[last]-a[first]
```

```
                  939 - 363
position = 0 + ----------- *(14 – 0) = 13.37
                  966 - 363
```

Example

Finished.

Interpolation search

```
def interpolationSearch( arr, q ):  # q is the query
    first = 0;  last = len(arr)-1

    while True:
        # found?
        if first == last :
            if arr[first] == q: return pos
            else:               return -1

        # continue search
        pos = first + round( (q-arr[first])/
                        (arr[last]-arr[first]) * (last-first) )
        if arr[pos] == q: return pos
        if arr[pos] < q:  first = pos+1  # check left side
        else:             last  = pos-1  # check right side
```

Search in a sorted array — speed comparison

	Method		
Array size N	Linear search average case	Interpolation search average case	Binary search all cases
10	5.5	1.60	4
30	15.5	2.12	5
100	50.5	2.56	7
300	150.5	2.89	9
1 000	500.5	3.18	10
3 000	1 500.5	3.41	12
10 000	5 000.5	3.63	14
30 000	15 000.5	3.80	15
100 000	50 000.5	3.96	17
300 000	150 000.5	4.11	19
1 000 000	500 000.5	4.24	20

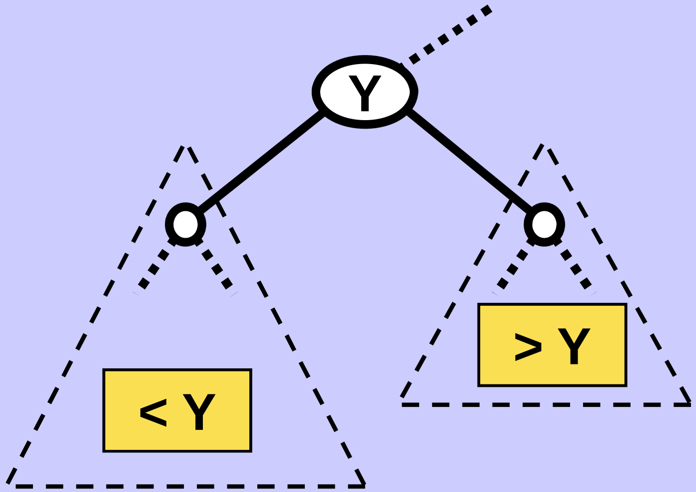
Asymptotic complexity	Obviously $\Theta(n)$	Random uniform distribution $\log_2(\log_2(N)) \in \Theta(\log(\log(N)))$	Due to the binary tree structure $\Theta(\log(n))$

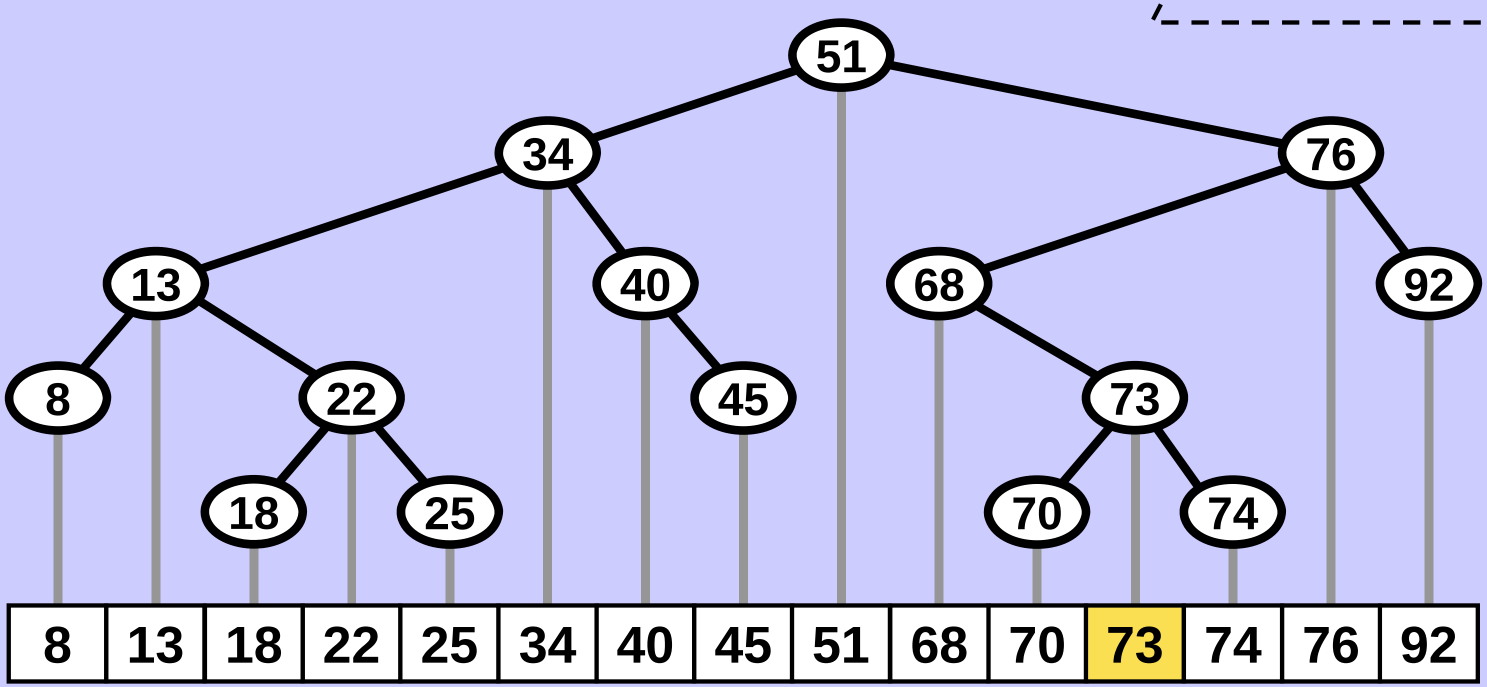
Binary search tree

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.



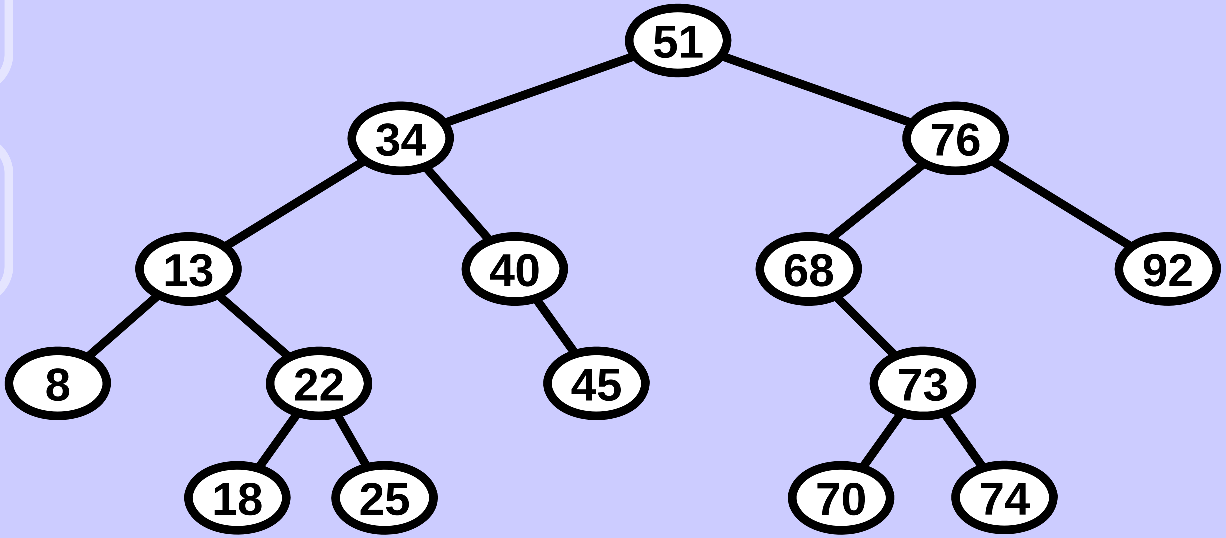


Binary search tree

BST may not be balanced and usually it is not.

BST may not be regular and usually it is not.

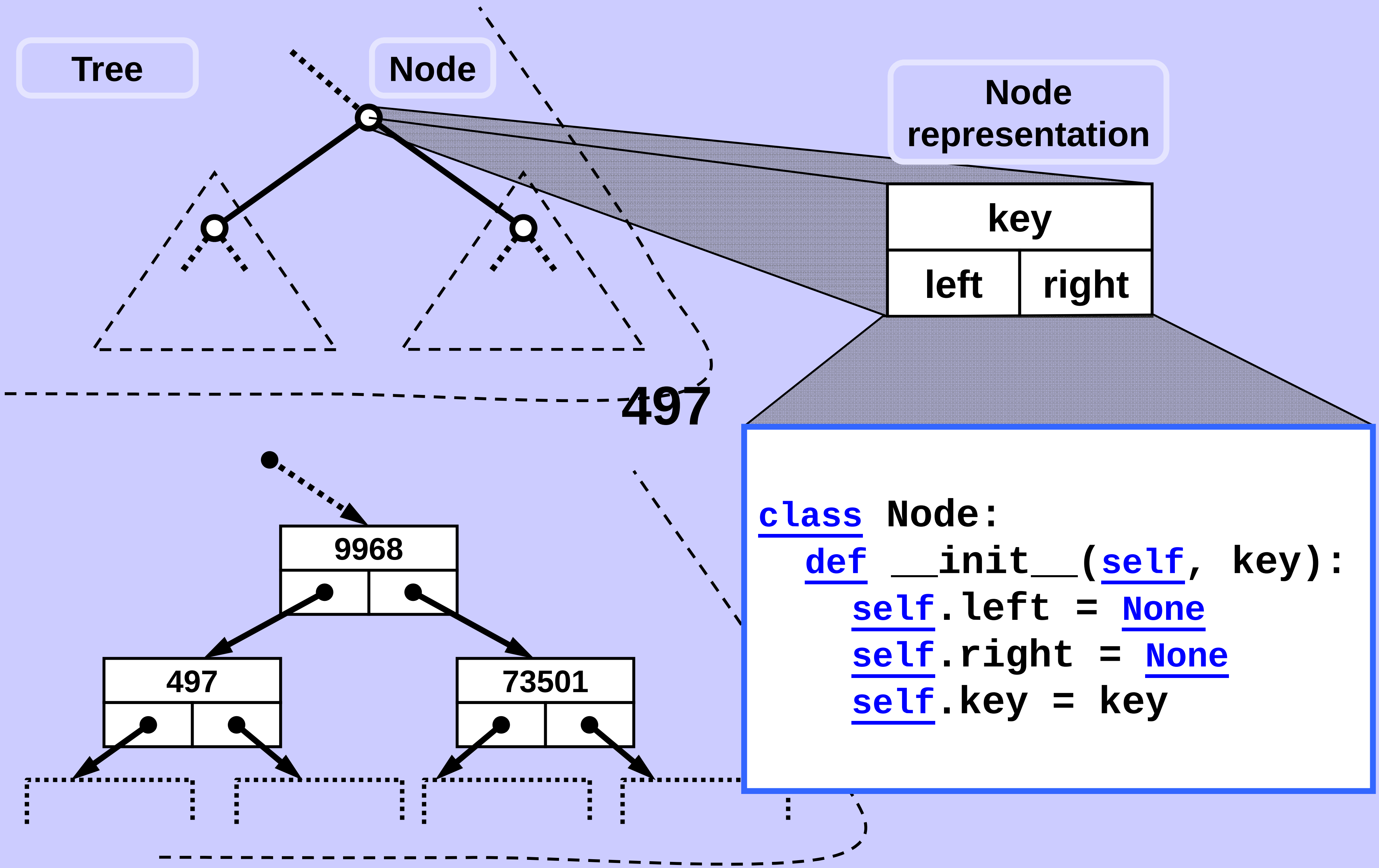
Apply the INORDER traversal to obtain sorted list of the keys of BST.



BST is flexible due to the operations:

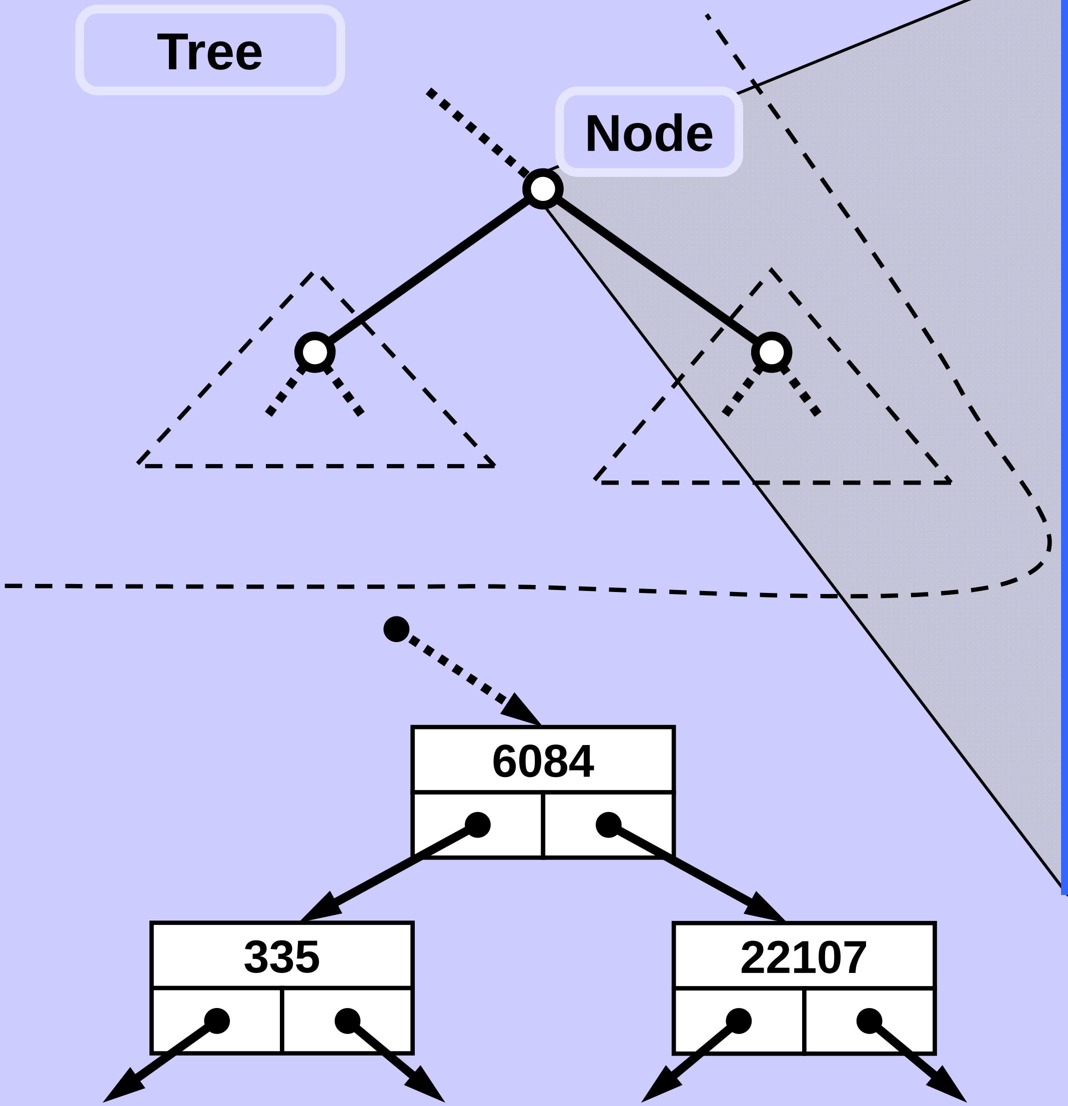
Find – return the pointer to the node with the given key (or null).
Insert – insert a node with the given key.
Delete – (find and) remove the node with the given key.

Binary search tree implementation -- Python




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

Binary search tree implementation -- Python

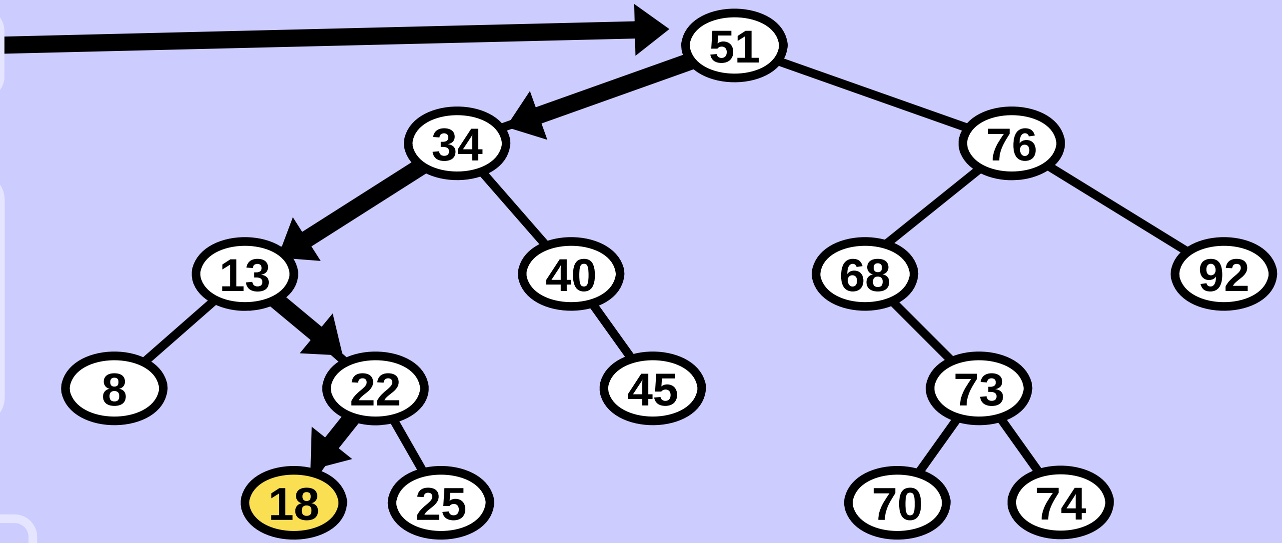



```
class Node:
   def __init__(self, key):
      self.left = None
      self.right = None
      self.key = key


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

Operation Find in BST

Find 18



Each BST operation starts in the root.

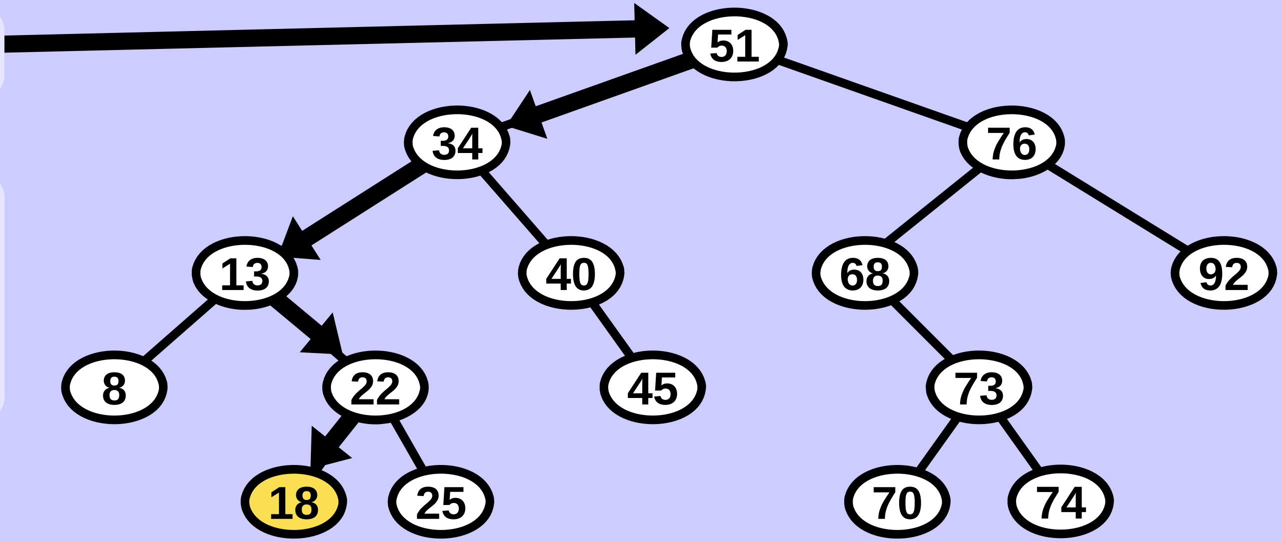
Iteratively

```
def FindIter(se 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



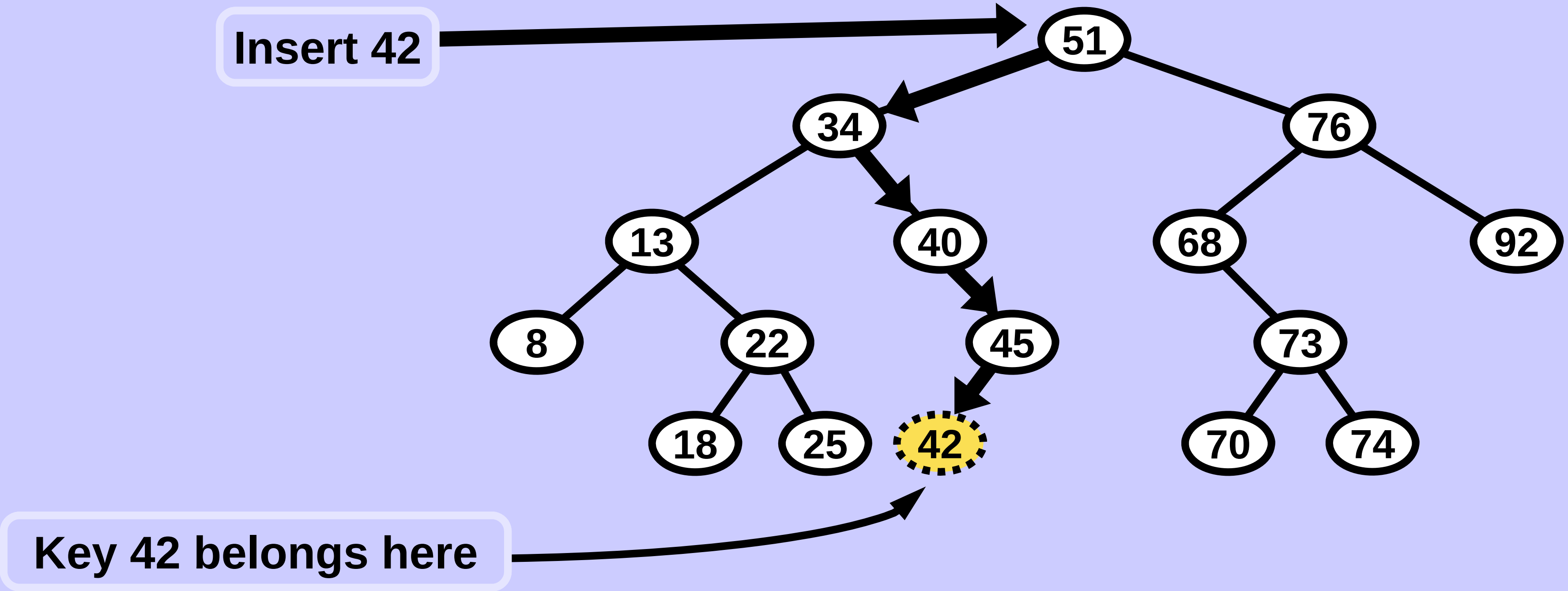
Each BST operation starts in the root.

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:                       # empty tree
        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

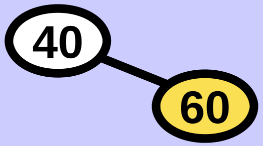
```
def Insert( self, key, node ):
    if key == node.key: return None  # no duplicates
    if key < node.key:
        if node.left == None:  node.left = Node( key )
        else:                  self.Insert( key, node.left )
    else:
        if node.right == None: node.right = Node( key )
       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


# call
if self.root == None:
    self.root = Node( key )
else: Insert( key, self.root )
```

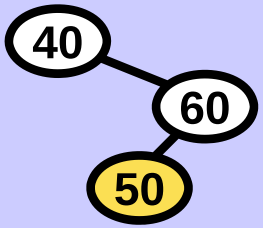
Building BST by repeated Insert

insert 40

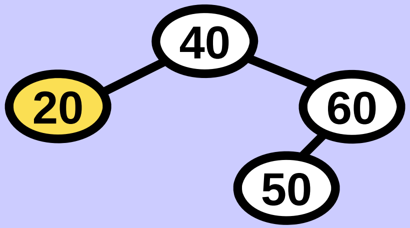
insert 60



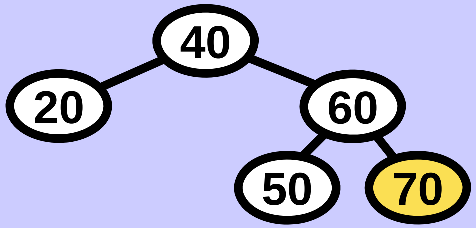
insert 50



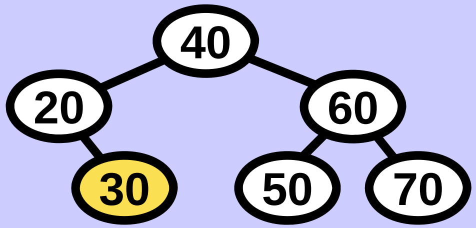
insert 20



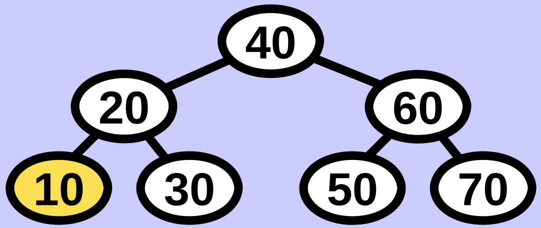
insert 70



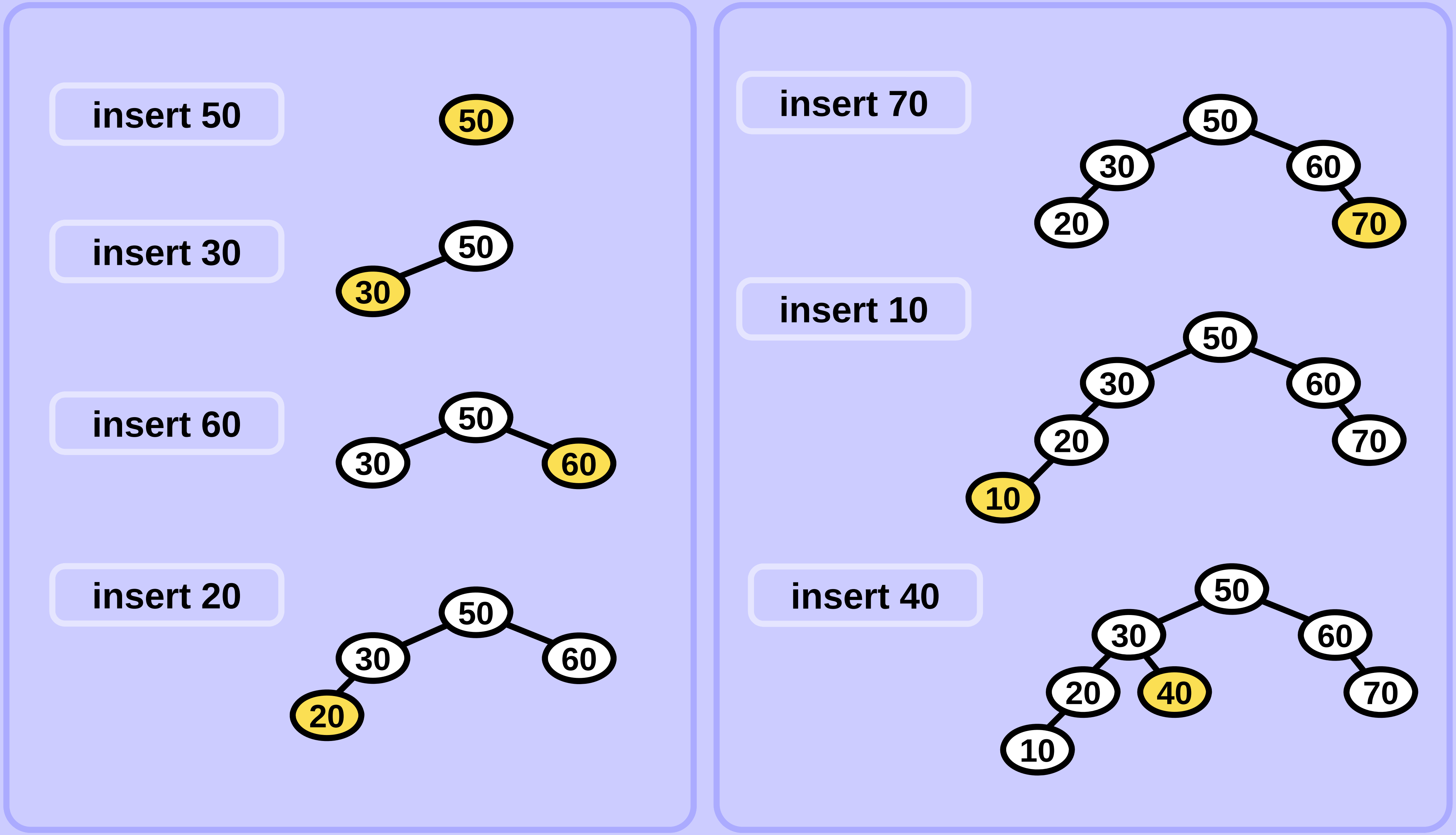
insert 30



insert 10

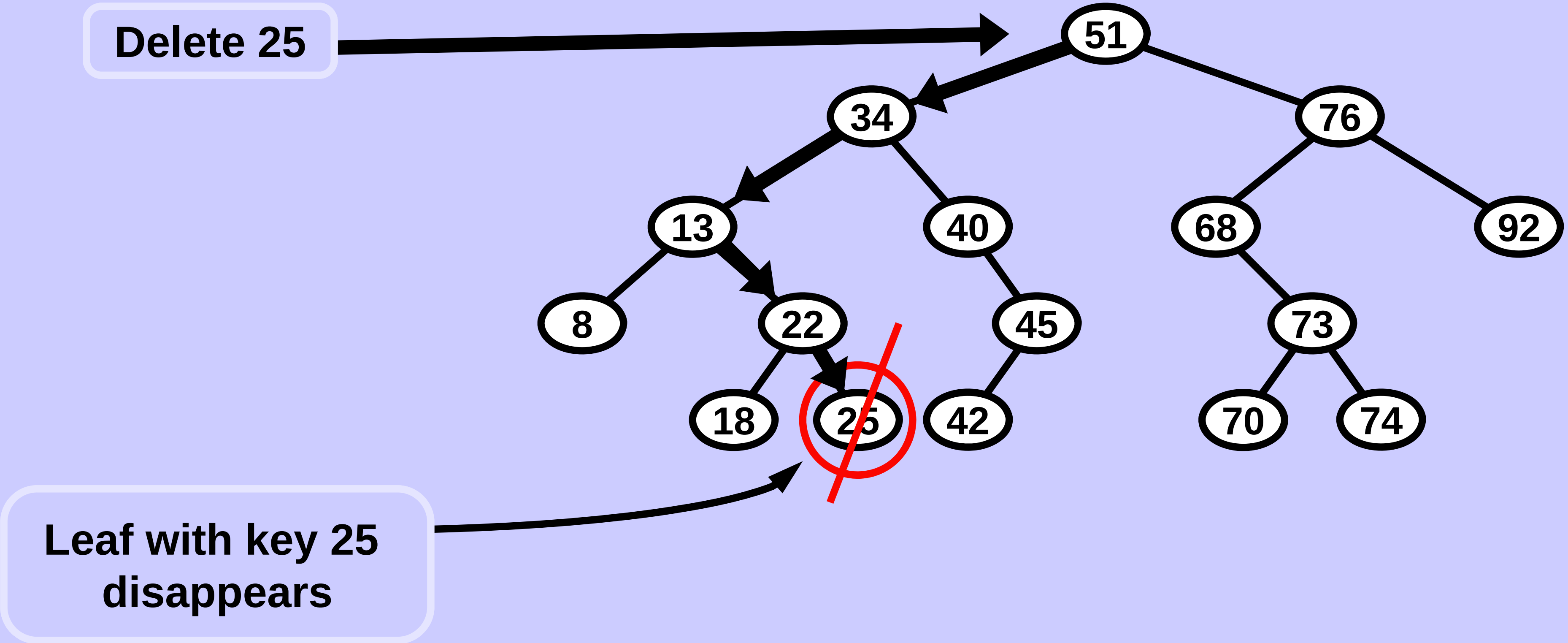


The shape of the BST depends on the order in which data are inserted.



Operation Delete in BST (I.)

Delete a node with 0 children (= leaf)

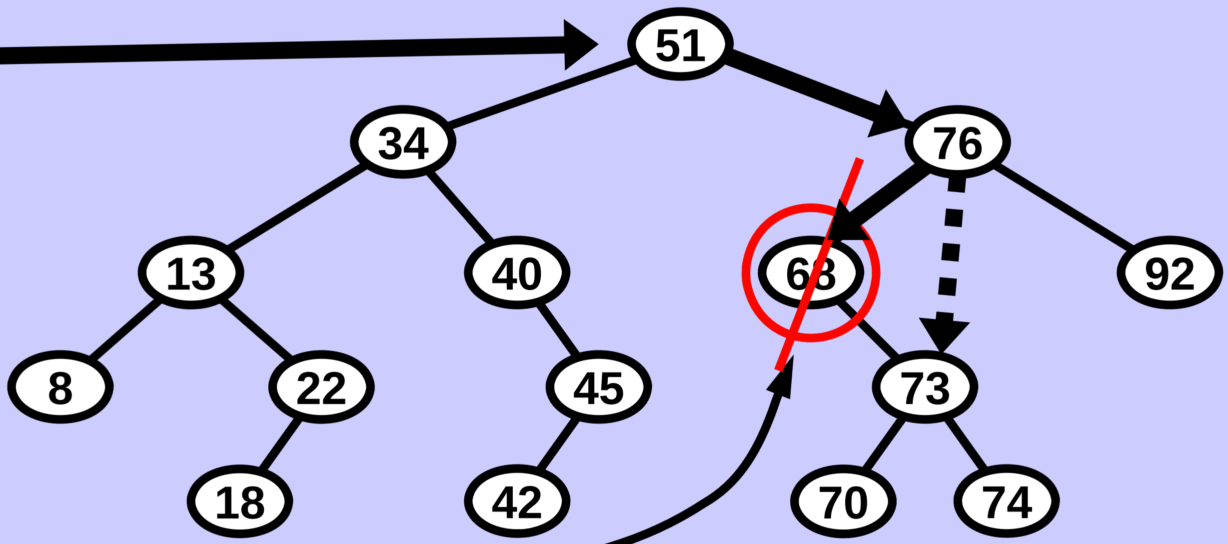


Delete I. Find the node (like in Find operation) with the given key and set the reference to it from its parent to null.

Operation Delete in BST (II.)

Delete a node with 1 child.

Delete 68



Node with key 68 disappears

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

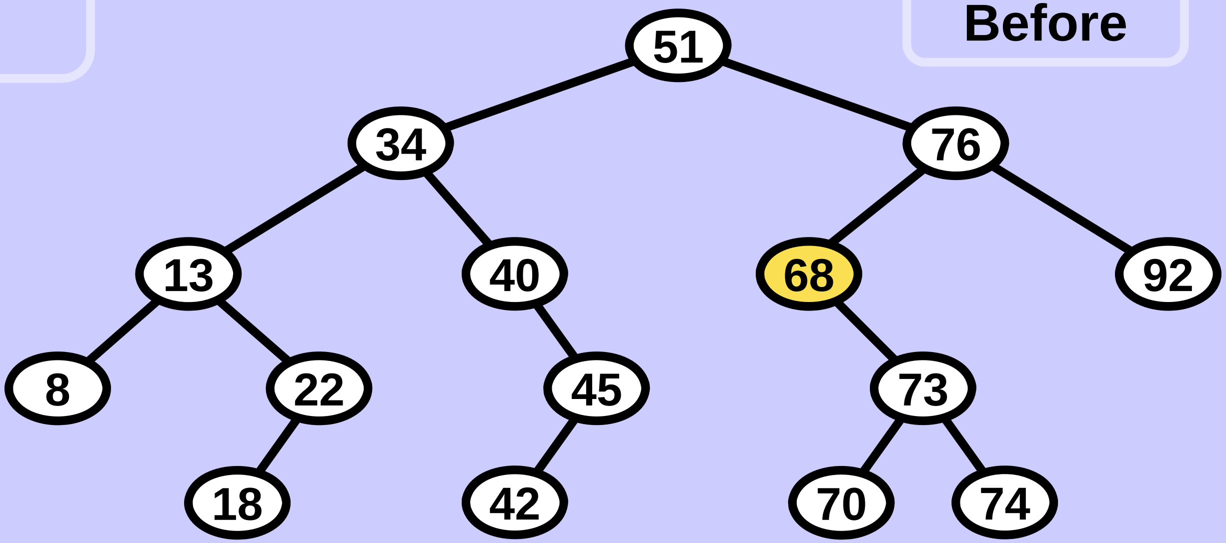
Delete II. Find the node (like in Find operation) with the given key and set the reference to it from its parent to its (single) child.

Operation Delete in BST (II.)

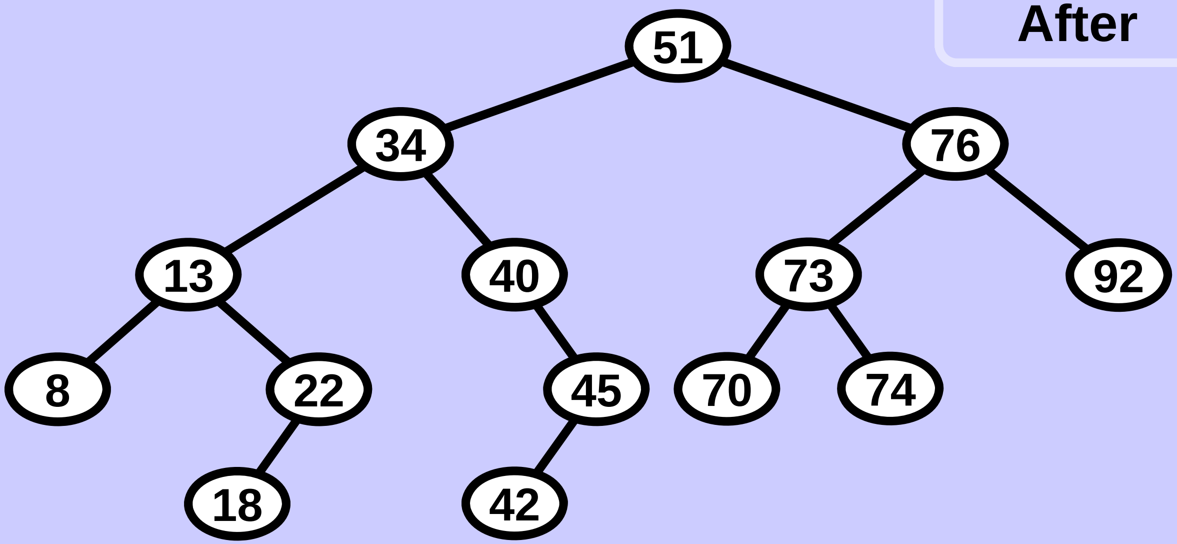
Delete a node with 1 child.

Delete 68

Before

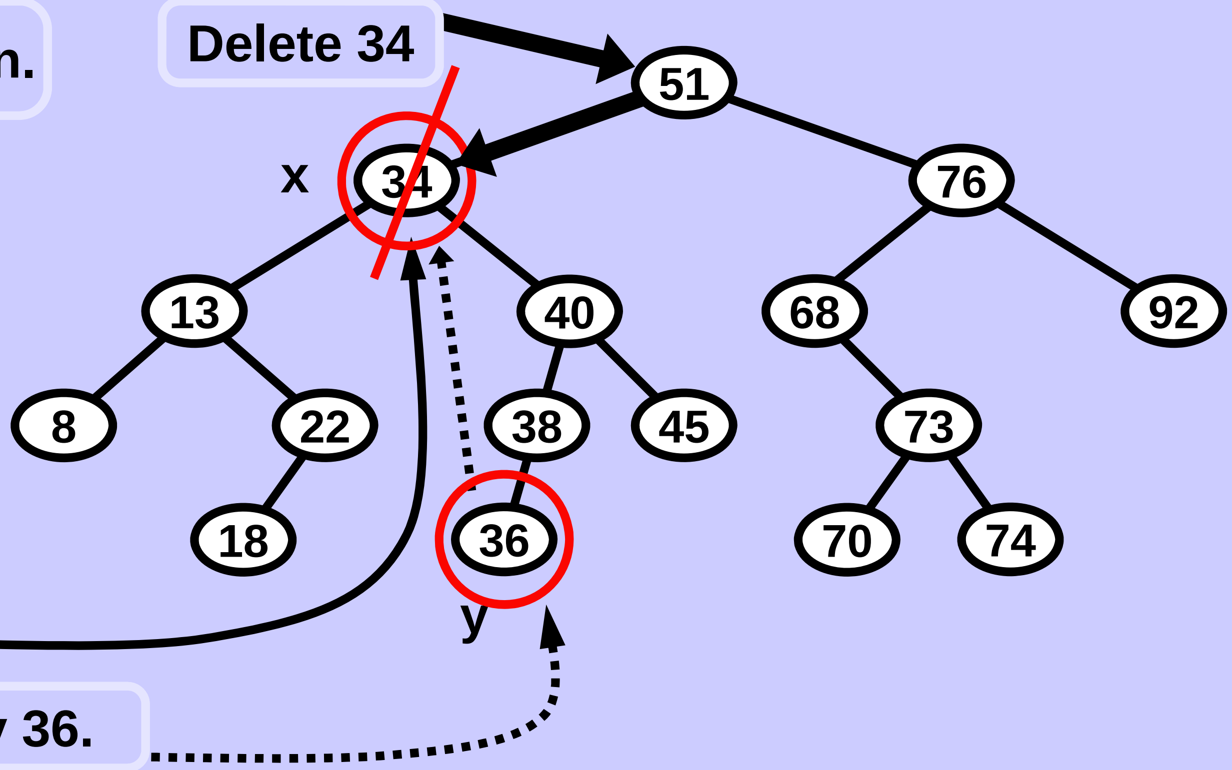


After



Operation Delete in BST (IIIa.)

Delete a node with 2 children.



Concept:

Key 34 disappears.

And it is substituted by key 36.

Implementation:

Delete IIIa.

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

Operation Delete in BST (IIIa.)

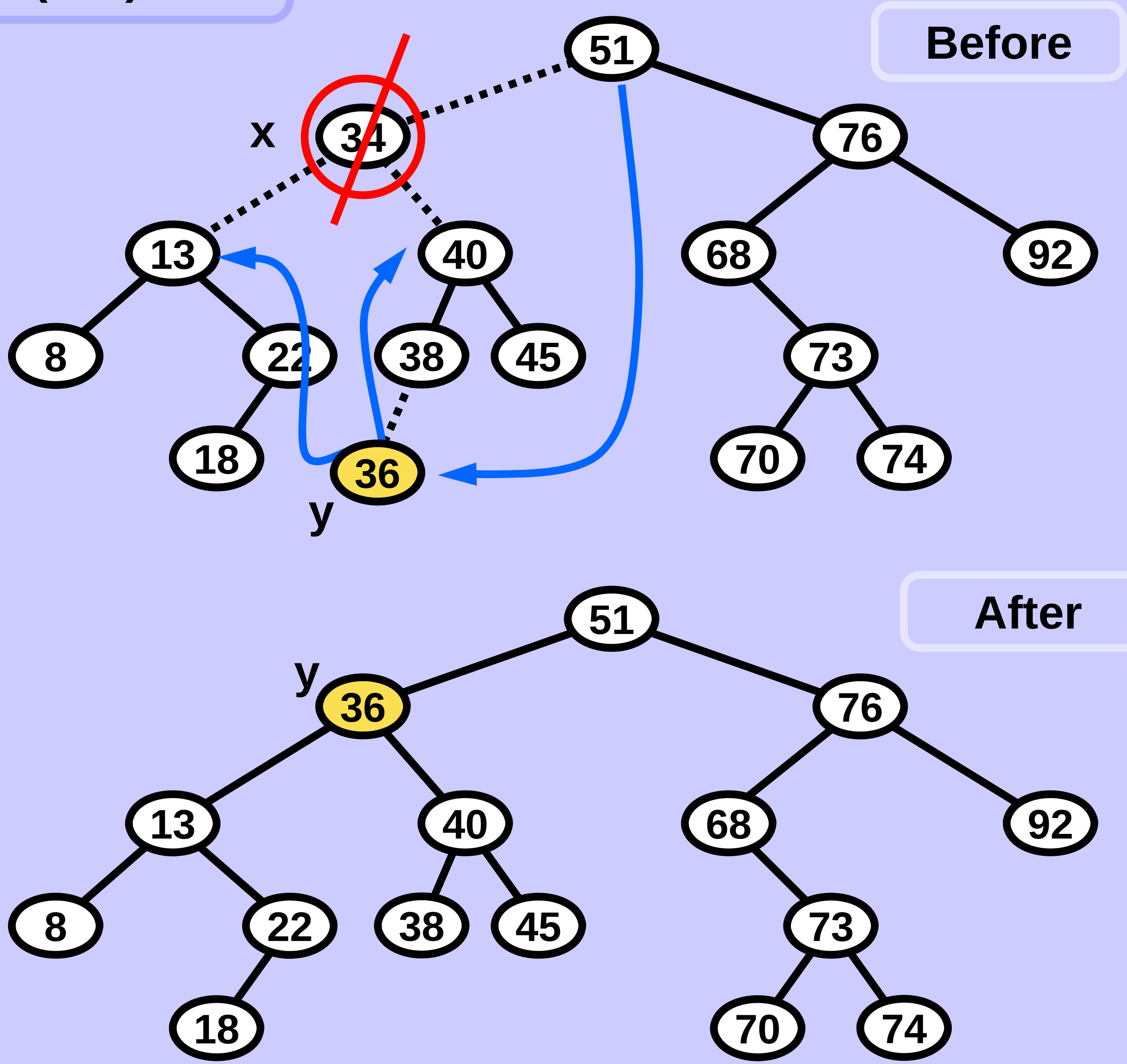
Delete 34

old

edges/pointers/references

new

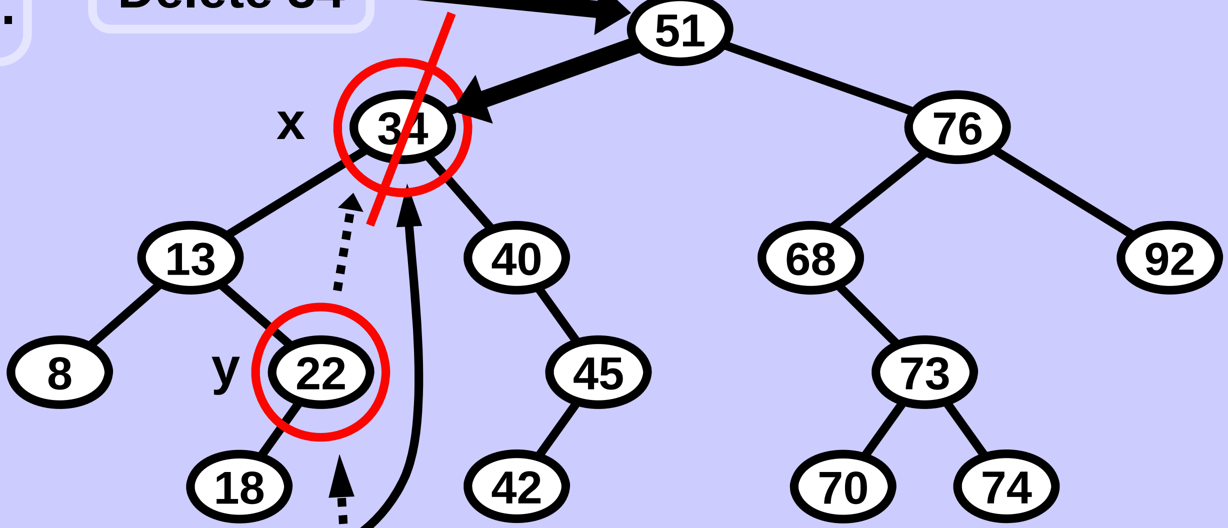
edges/pointers/references



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

Delete a node with 2 children.

Delete 34



Concept:

Key 34 disappears.

And it is substituted by key 22.

Implementation:

Delete IIIb.

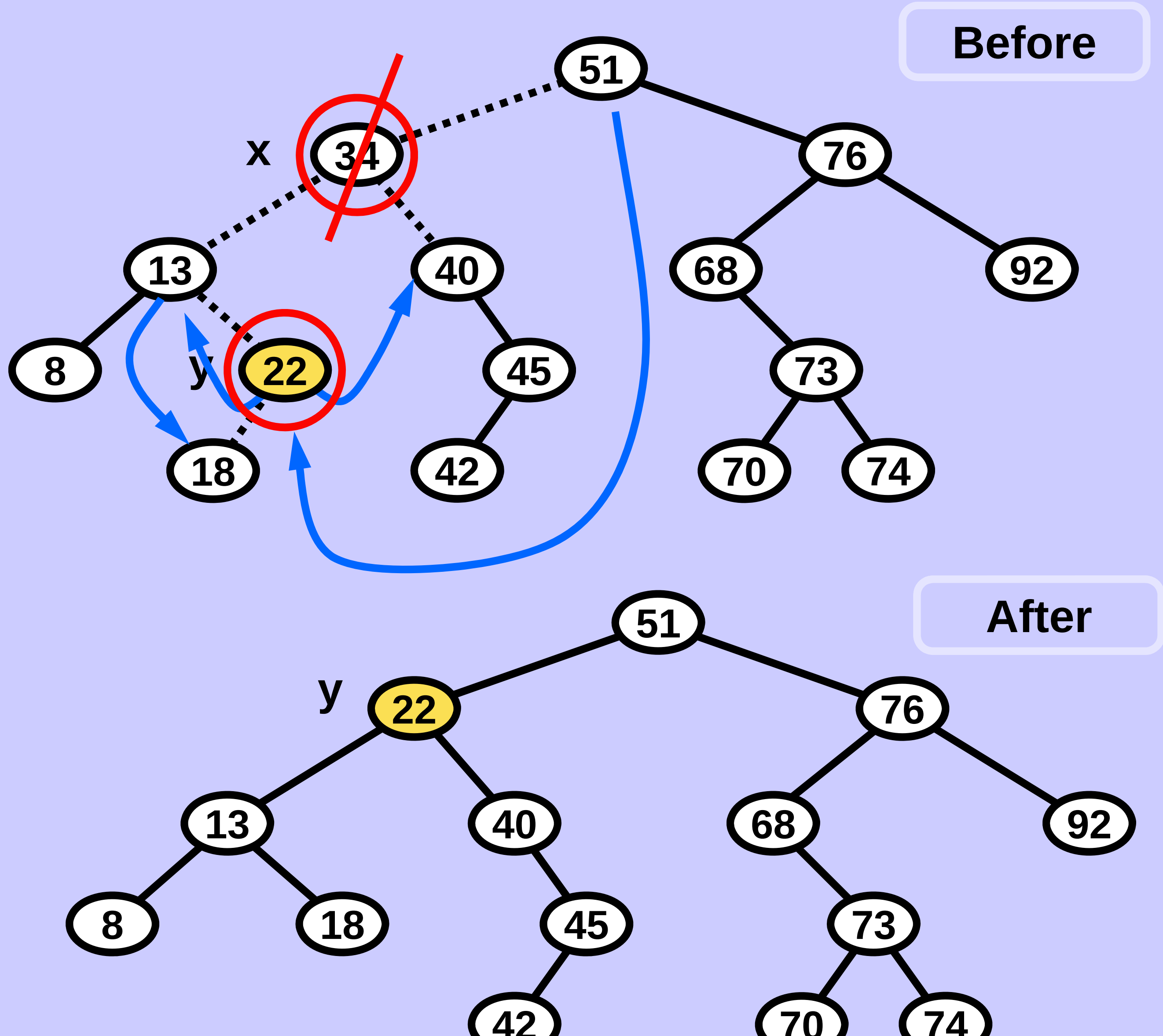
1. **Find the node (like in Find operation) with the given key and then find the *rightmost* (= biggest key) node y in the *left* subtree of x.**
2. **Point from y to children of x, from parent of y point to the child of y instead of y, from parent of x point to y.**

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 it looks as if variant Delete II was applied locally on the moved node.

Operation Delete in BST -- recursively

```
def delete( self, node, parent, key ):

  # not found or search recursively in L or R subtree
  if node == None: return None
  if key < node.key:  self.delete( node.left,  node, key ); return
  if node.key < key:  self.delete( node.right, node, key ); return

  # found in current node, delete the key/node
  if node.left != None and node.right != None:
    # both children
    rightMinNode, rightMinParent = self.findMin( node.right, node )
    node.key = rightMinNode.key
    self.delNodeWithAtMost1Child( rightMinNode, rightMinParent )
  else:
    # single child
    self.delNodeWithAtMost1Child( node, parent )
```

Operation Delete in BST -- support functions

```
def findMin( self, node, parent ):
    while node.left != None:
        parent = node; node = node.left
    return node, parent
```

```
def delNodeWithAtMost1Child( self, node, parent ):
    if node.left is None:
        if node.right == None:
            # leaf, no child
            if parent.left == node: parent.left  = None
            else:                   parent.right = None
        else: # single R child
            if parent.left == node: parent.left  = node.right
            else:                   parent.right = node.right
    else:
        if node.right == None:
            # single L child
            if parent.left == node: parent.left  = node.left
            else:                   parent.right = node.left
```

Operation Delete in BST

Asymptotic complexities of operations Find, Insert, Delete in BST

	BST with n nodes	
Operation	Balanced Not guaranteed !! Must be induced by additional conditions.	Not balanced (expected general case)
Find	O(log(n))	O(n)
Insert	O(log(n))	O(n)
Delete	O(log(n))	O(n)

Additional Fact :
The expected height of a *randomly* built binary search tree on n distinct keys is O (log n). ***source: [CLRS]***

***Randomly*, in this case: Each of the n! permutations of the input keys is equally likely.**

Uniformly random BST Experiment

```
def depth( self ):
    return self._depth( self.root )

def _depth( self, node ):
    if node == None: return -1
    return 1 + max(self._depth(node.left),self._depth(node.right))

def createRandomTree(self, Nkeys ):
    keys = list( range(0,Nkeys) )
    random.shuffle ( keys )
    for key in keys: self.insert( key )
```

```
for i in range( 1, 6 ):
    tree =  BinarySearchTree()
    tree.createRandomTree( 10**i )
    print( tree.N, tree.depth() \
          "%4.1f"%(2*math.log2(tree.N)))
```

Experiment results

Uniformly Random BST
with N nodes

N	depth	2*log2(N)
10	4	6.6
100	11	13.3
1000	19	19.9
10000	30	26.6
100000	37	33.2
1000000	48	39.9