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.

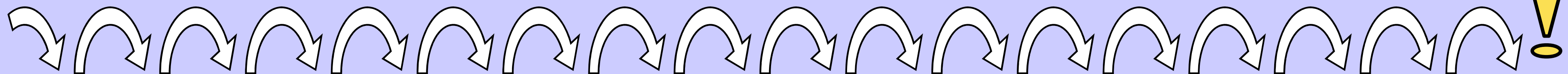
Array

Size = N

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

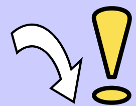
Find 993 !

Tests: N



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

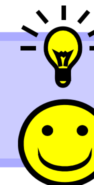
Find 363 !



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 !

2 tests

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

839	860	863	938	939	966	968	983	993

2 tests

2 tests

839	860	863	938	939	966	968	983	993

839	860	863	938

966	968	983	993

2 tests

839	860	863	938

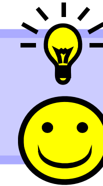
839

863	938

1 test

863	938

Search in a sorted array — binary, FASTER

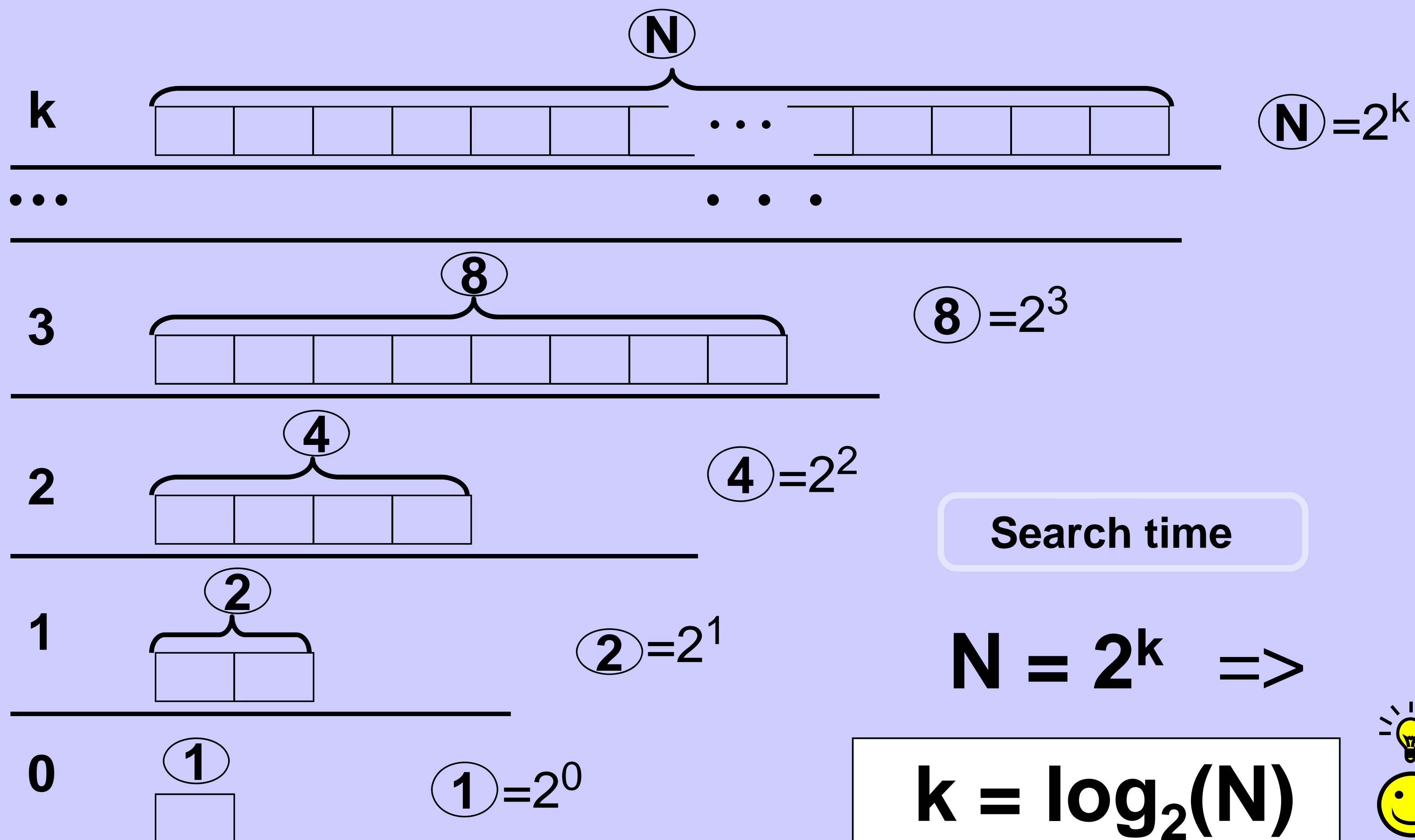


Find 863 !

The search follows the structure of a binary tree.

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



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 first	1	2											13		15 position		17 last

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.

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

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

Example

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 first	1	2											13	14 position	15 last		17

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 first	1	2											13 position	14 last	15		17

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

Finished.

Interpolation search

```
int interpol(int [] arr, int q) {
  int first = 0;
  int last = length(arr)-1;
  do {
     pos = first + round((q-arr[first])/(arr[last]-arr[first])
                             *(last-first) );
     if (arr[pos] < q)      first = pos+1;  // check left side
     else if (arr[pos] > q)  last = pos-1;  //check right side
   } while ((arr[pos] != q) && (first < last));
   return pos;
}
```

Search in a sorted array — speed comparison

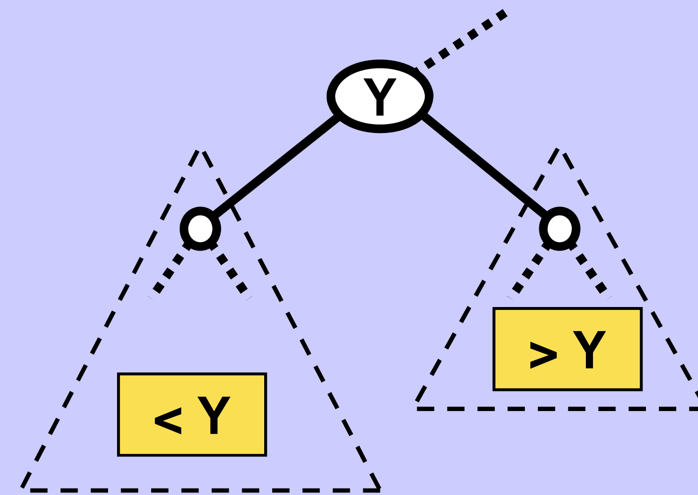
Array size N	Linear search average case	Interpolation search average case	Binary search cca average	/ worst case
10	5.5	1.60	7	9
30	15.5	2.12	9	11
100	50.5	2.56	13	15
300	150.5	2.89	17	19
1 000	500.5	3.18	19	21
3 000	1 500.5	3.41	23	25
10 000	5 000.5	3.63	27	29
30 000	15 000.5	3.80	29	31
100 000	50 000.5	3.96	33	35
300 000	150 000.5	4.11	37	39
1 000 000	500 000.5	4.24	39	41
Asymptotic complexity	Obviously $\Theta(n)$	Random uniform distribution $\log_2(\log_2(N)) \in \Theta(\log(\log(N)))$	Due to the binary tree structure $\Theta(\log(n))$	

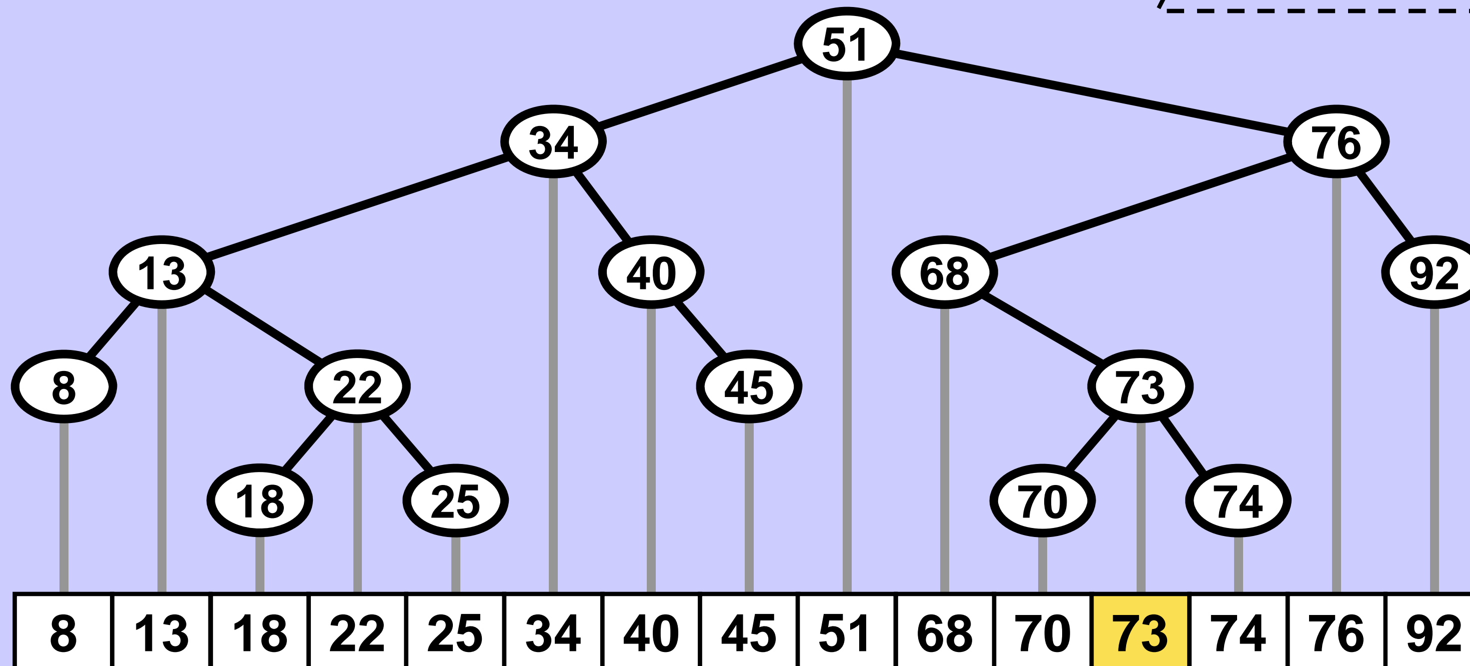
Binary search tree

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.



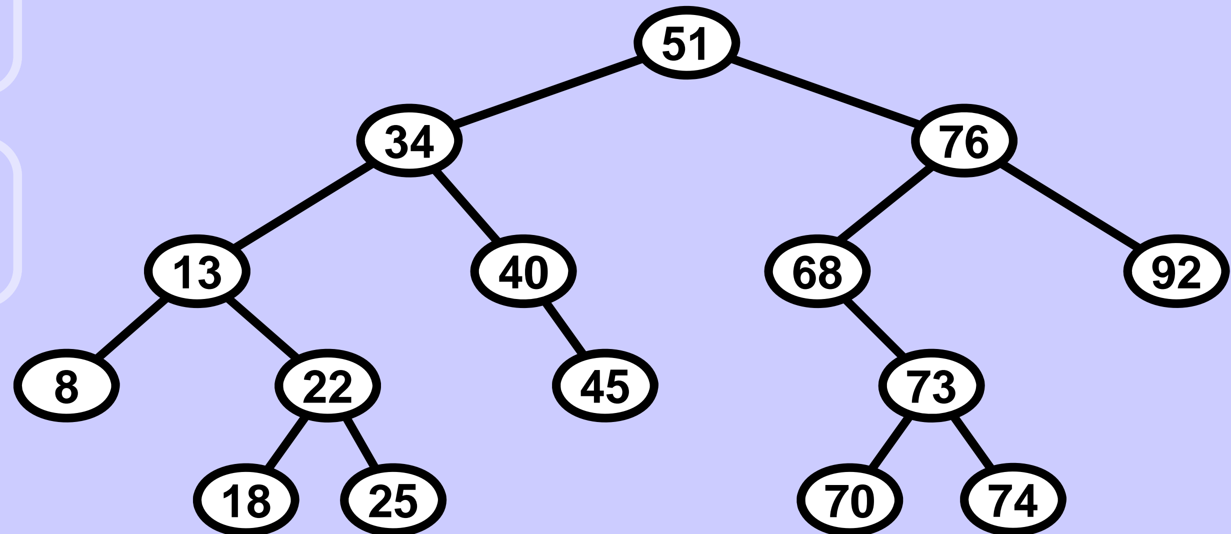


Binary search tree

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 -- C

Tree

Node

Node representation

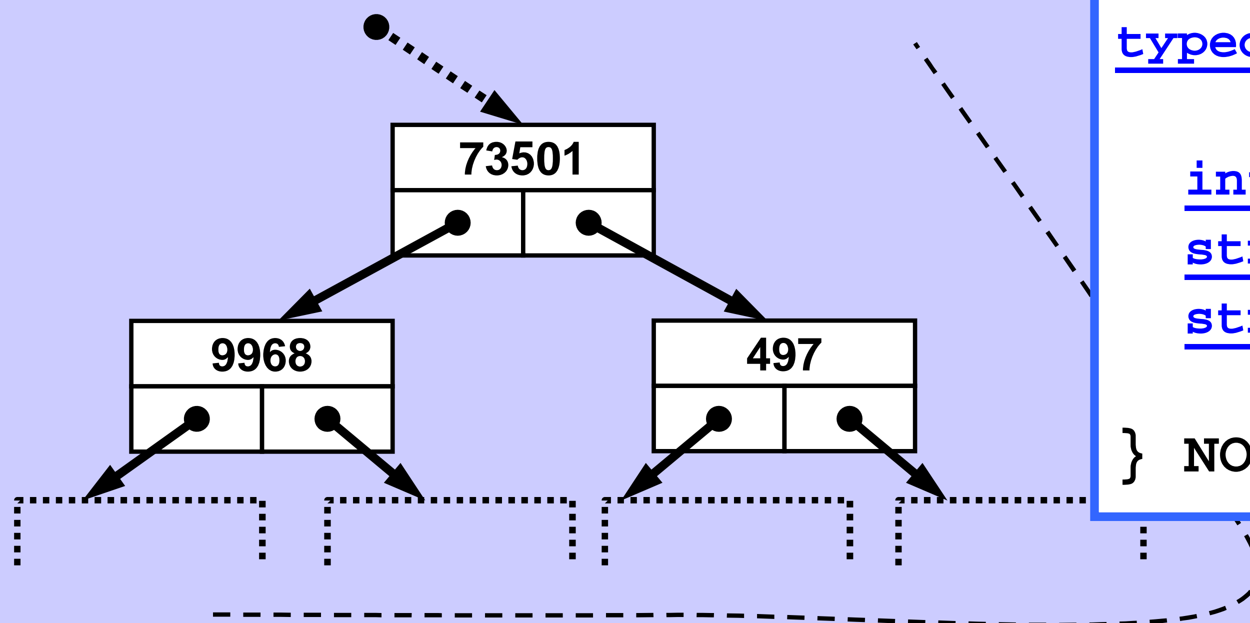
key

left | right

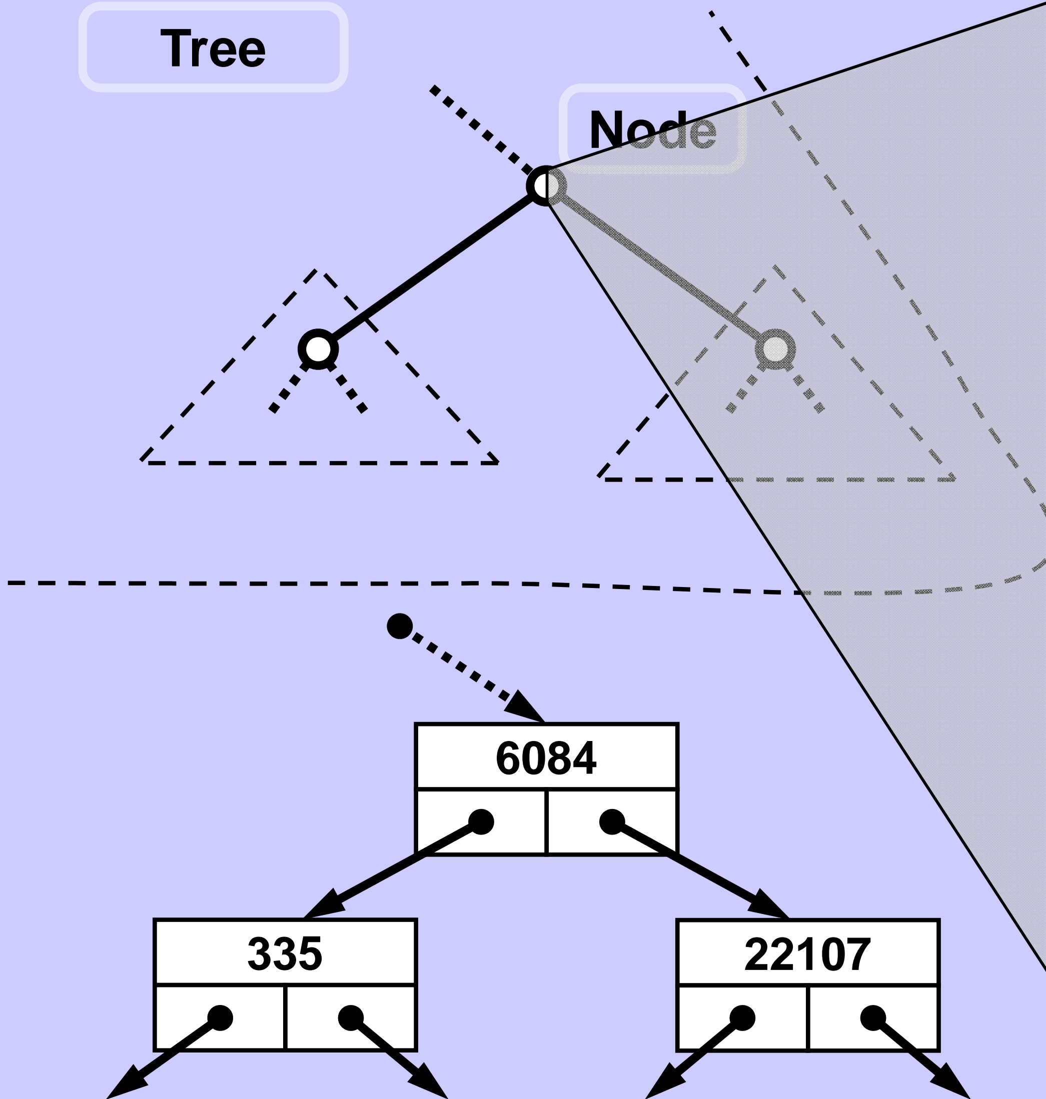
```c
typedef struct Node {

   int key;
   struct Node *left;
   struct Node *right;

} NODE;
```



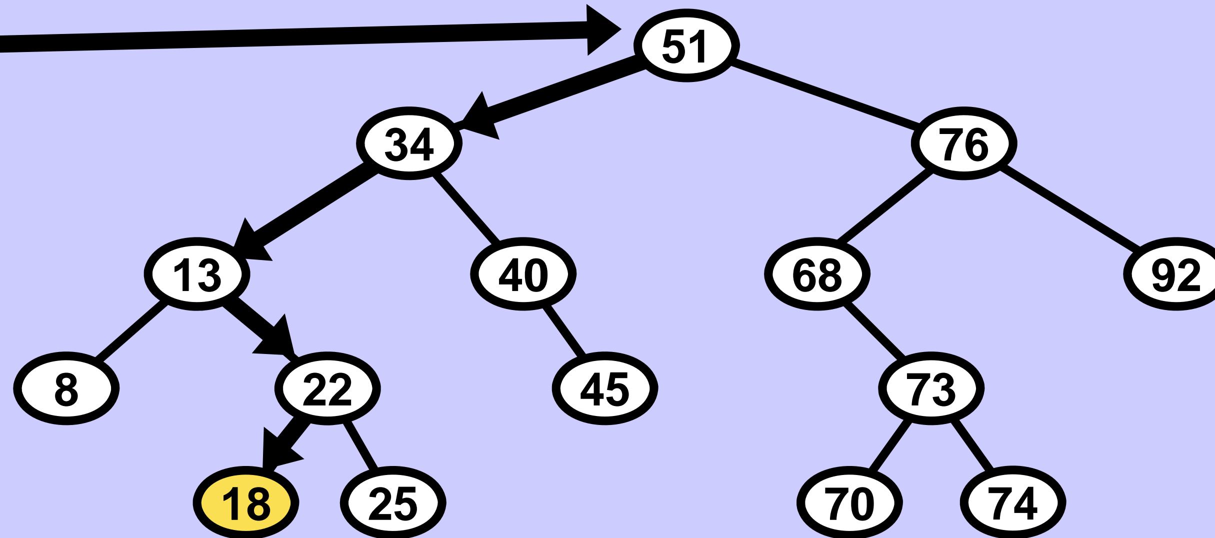
Binary search tree implementation -- java




```java
public class Node {
   public Node left;
   public Node right;
   public int key;
   public Node(int k) {
      key = k;
      left = null;
      right = null;
   }
}

public class Tree {
   public Node root;
   public Tree() {
      root = null;
   }
}
```

Operation Find in BST



Each BST operation starts in the root.

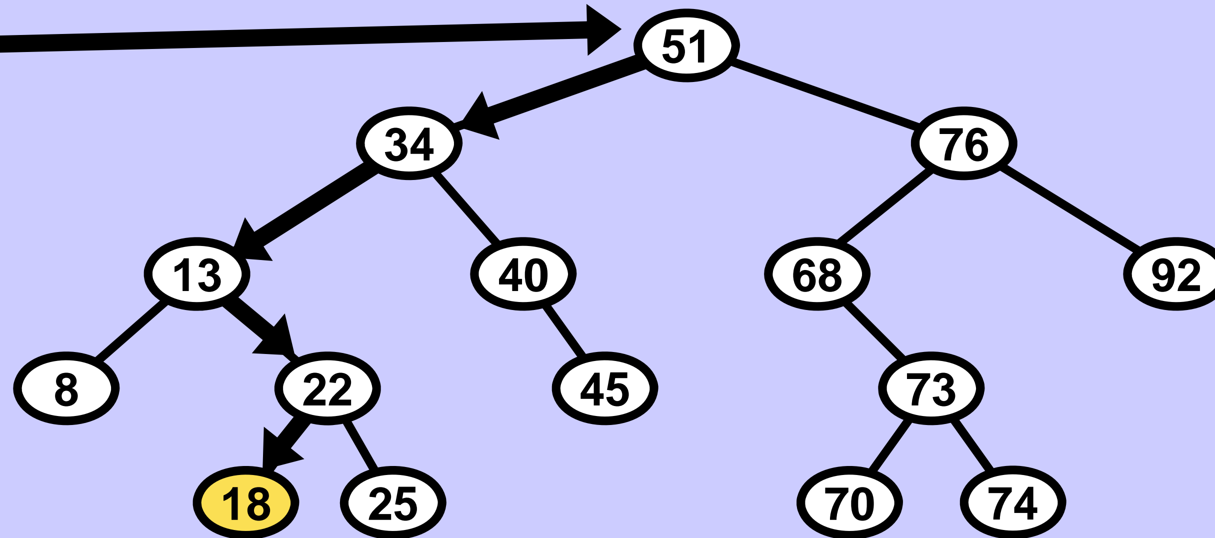
Iteratively

```
Node find(int k, Node node){
   while (true) {
      if (node == null) return null;
      if (node->key == k) return node;
      if (k < node->key) node = node->left;
         else node = node->right;
} }
```

Operation Find in BST

Find 18

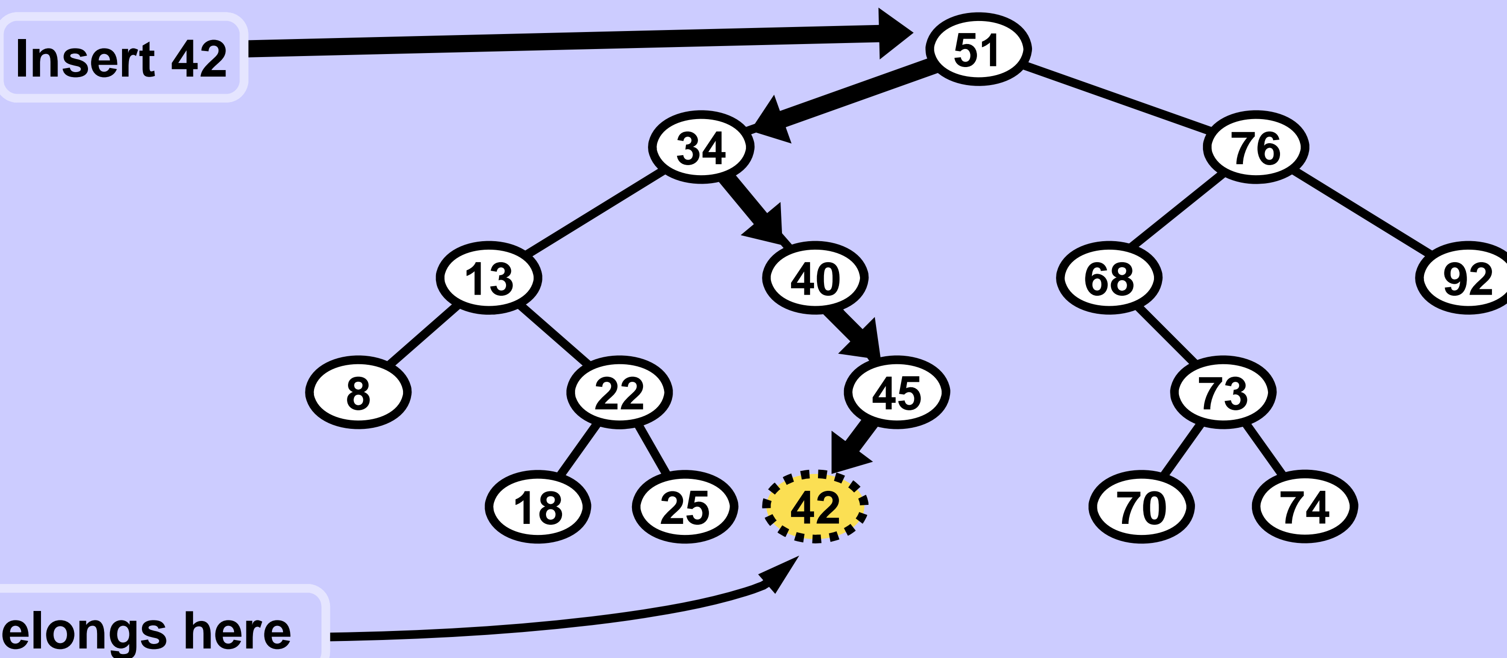
Each BST operation starts in the root.



Recursively

```
Node findRec(int k, Node node){
   if (node == null) return null;
   if (node.key == k) return node;
   if (k < node->key) return findRec(k, node->left);
                else return findRec(k, node->right) ;
} }
```

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

```
Node insert (int k, Node node) {
  if (node == null){                          // empty tree
    Node newNode = ...;                       // create a node with key k
    return newNode;
  }
  while (true) {
    if (node->key == k) return null; //can't insert a duplicate
    if (node->key > k)
       if (node->left == null) {
          Node newNode = ...;                 // create a node with key k
          node->left = newNode;
          return newNode; }
       else node = node->left;
    else                                      // similarly to the right
       if(node->right == null){
          Node newNode = ...;
          node->right = newNode;
          return newNode; }
       else node = node->right;
} }
```

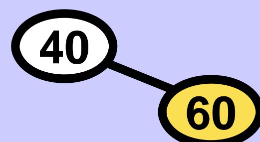
Operation Insert in BST recursively

```
Node insertRec (int k, Node node, Node parentNode) {
  if (node == null){                    // empty tree
    Node newNode = ...;                 // create node with key k
    if(parentNode != null){
       if(parentNode->key > k)
          parentNode->left = newNode;
       else
          parentNode->right = newNode;
    return newNode;
  }
  if (node->key == k) return ;          //can't insert a duplicate
  if (node->key > k)                    // chose direction
     return insertRec(k, node->left, node);
  else
     return insertRec(k, node->right, node);
}
```

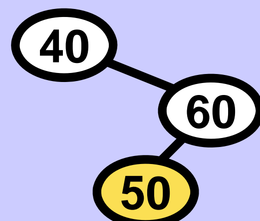
Building BST by repeated Insert

insert 40

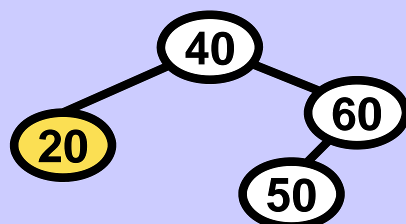
insert 60



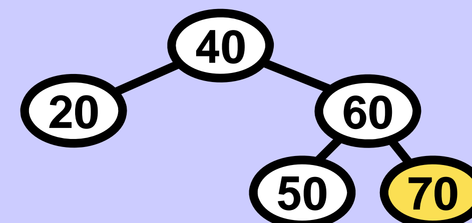
insert 50



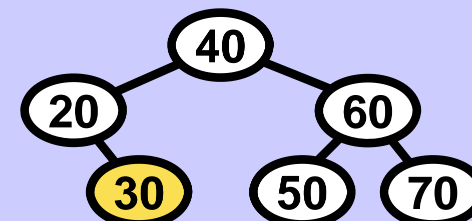
insert 20



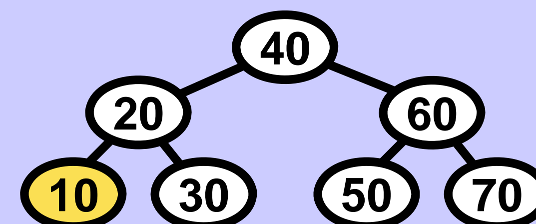
insert 70



insert 30



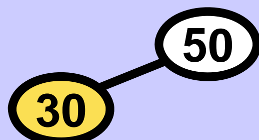
insert 10



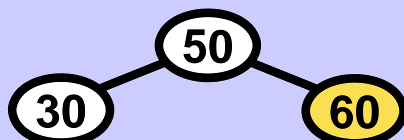
The shape of the BST depends on the order in which data are inserted.

insert 50

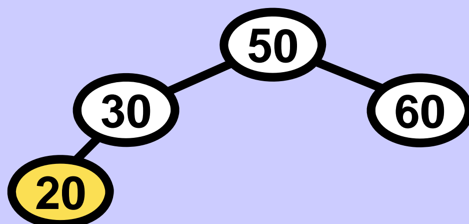
insert 30



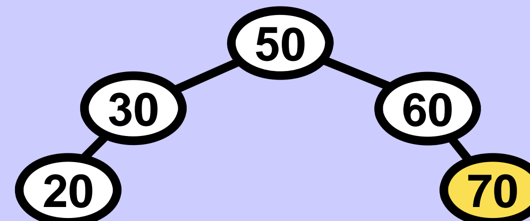
insert 60



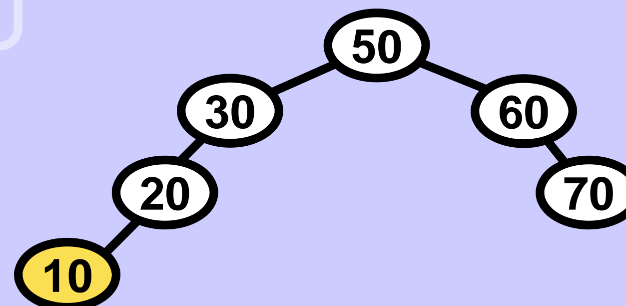
insert 20



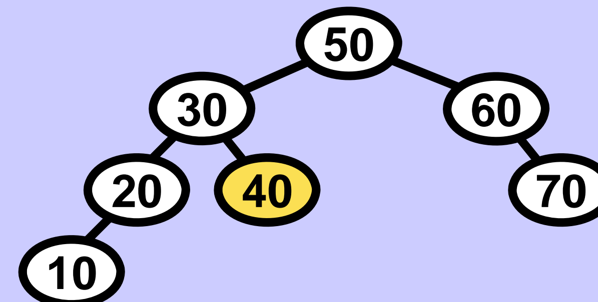
insert 70



insert 10



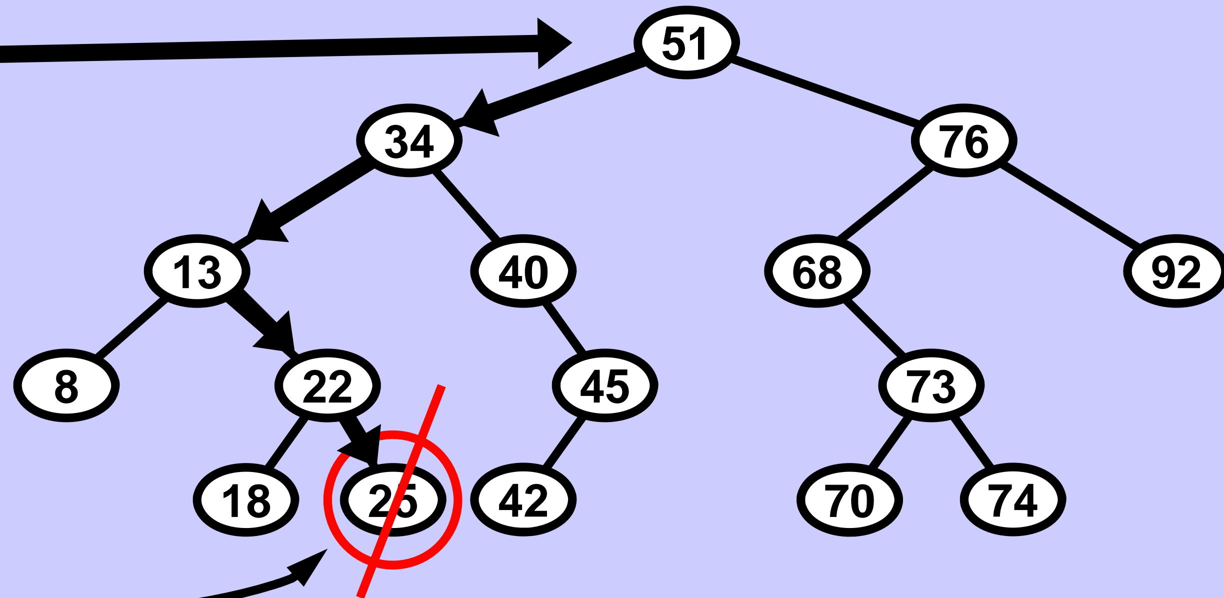
insert 40



Operation Delete in BST (I.)

Delete a node with 0 children (= leaf)

Delete 25



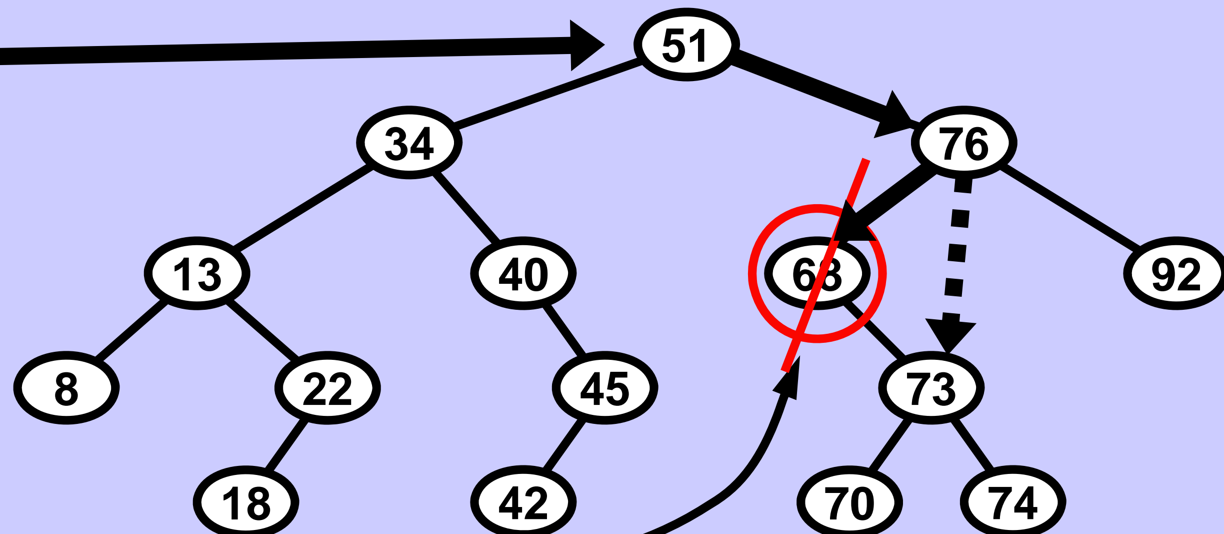
Leaf with key 25 disappears

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.

Delete 68



Node with key 68 disappears

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

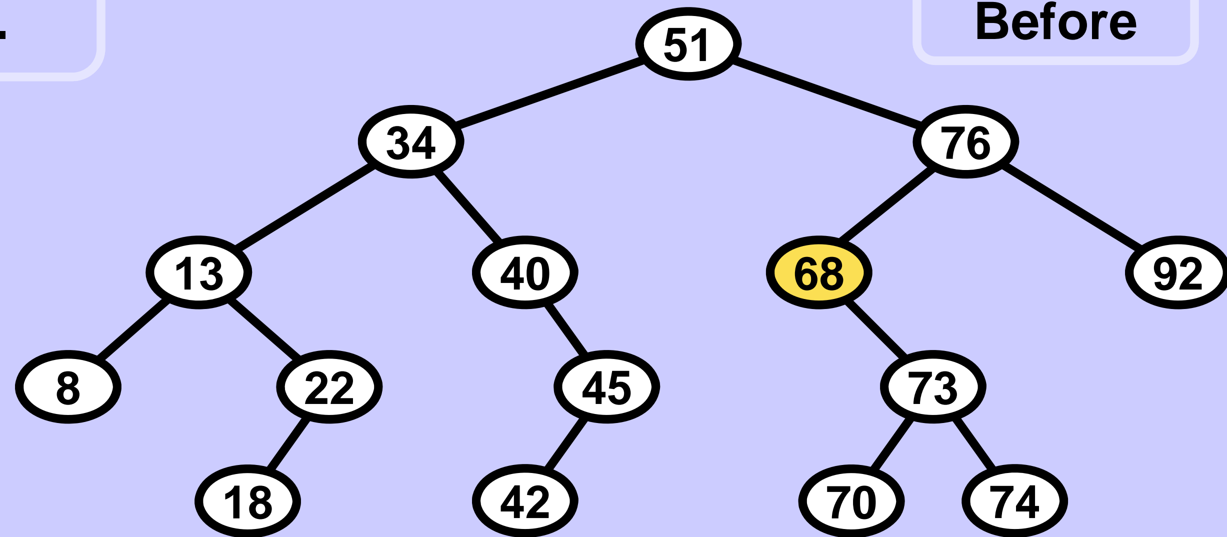
Delete II. Find the node (like in Find operation) with the given key and set the reference to it from its parent to its (single) child.

Operation Delete in BST (II.)

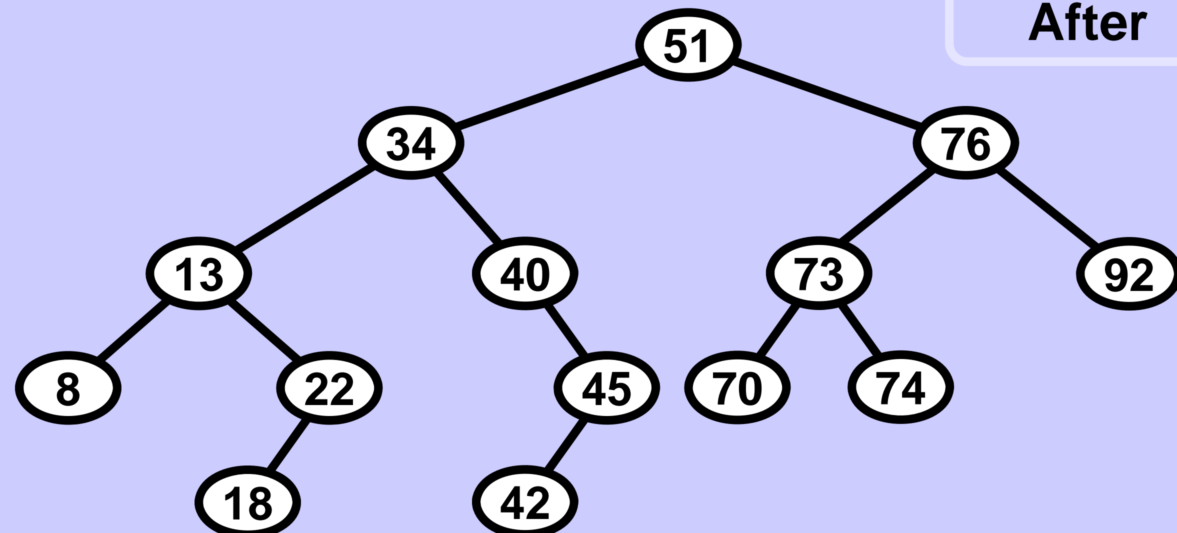
Delete a node with 1 child.

Delete 68

Before



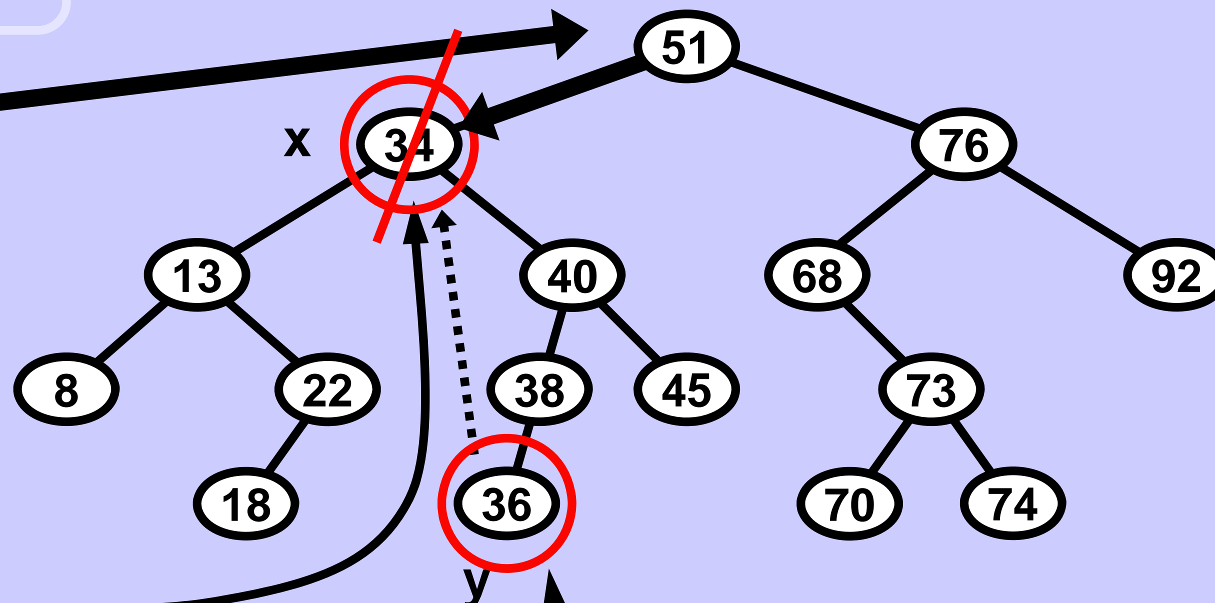
After



Operation Delete in BST (IIIa.)

Delete a node with 2 children.

Delete 34



Key 34 disappears.

And it is substituted by key 36.

Delete IIIa.

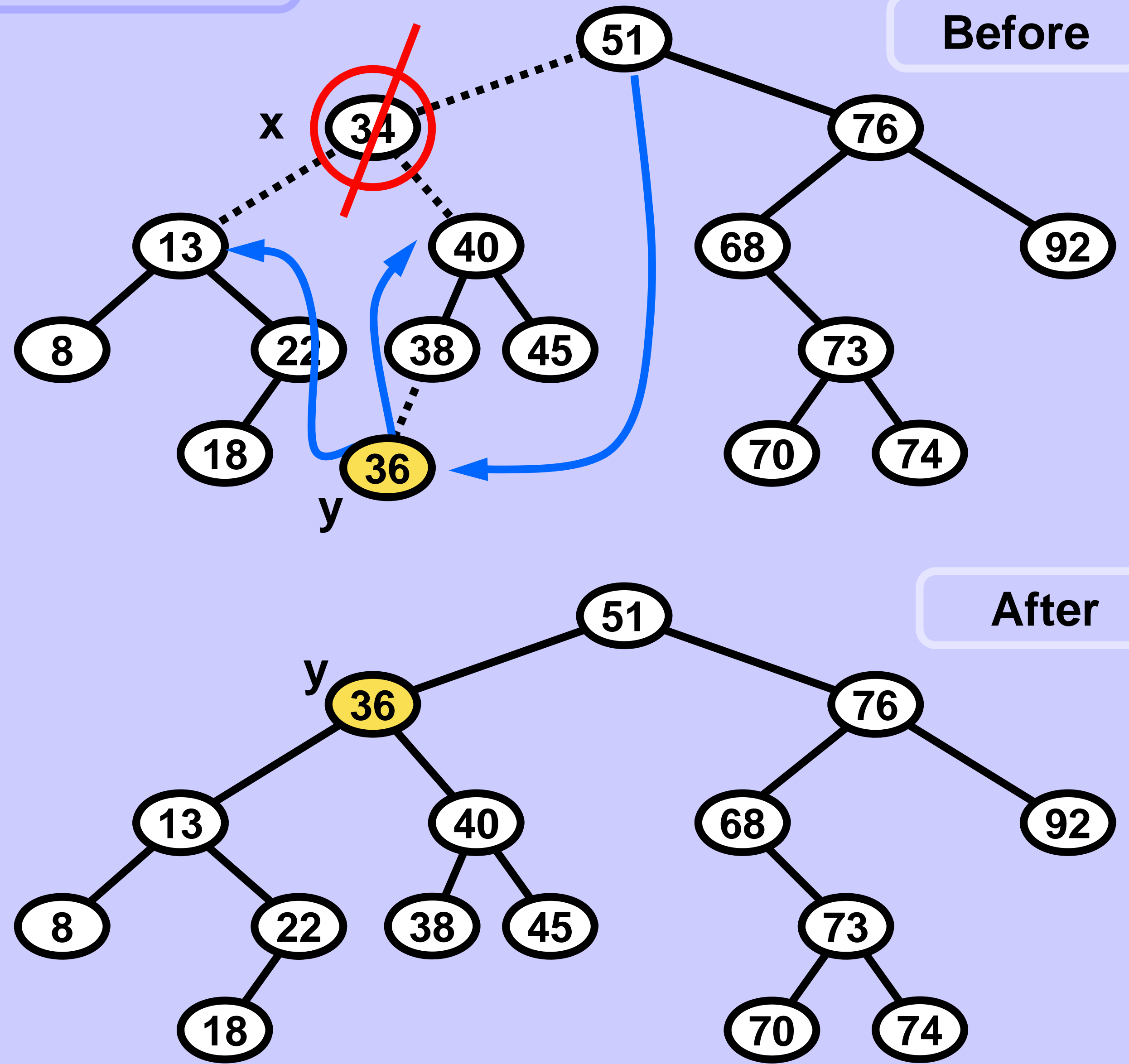
1. Find the node (like in Find operation) with the given key and then find the ***leftmost*** (= smallest key) node y in the ***right*** subtree of x.
2. Point from y to children of x, from parent of y point to the child of y instead of y, from parent of x point to y.

Operation Delete in BST (IIIa.)

Delete 34

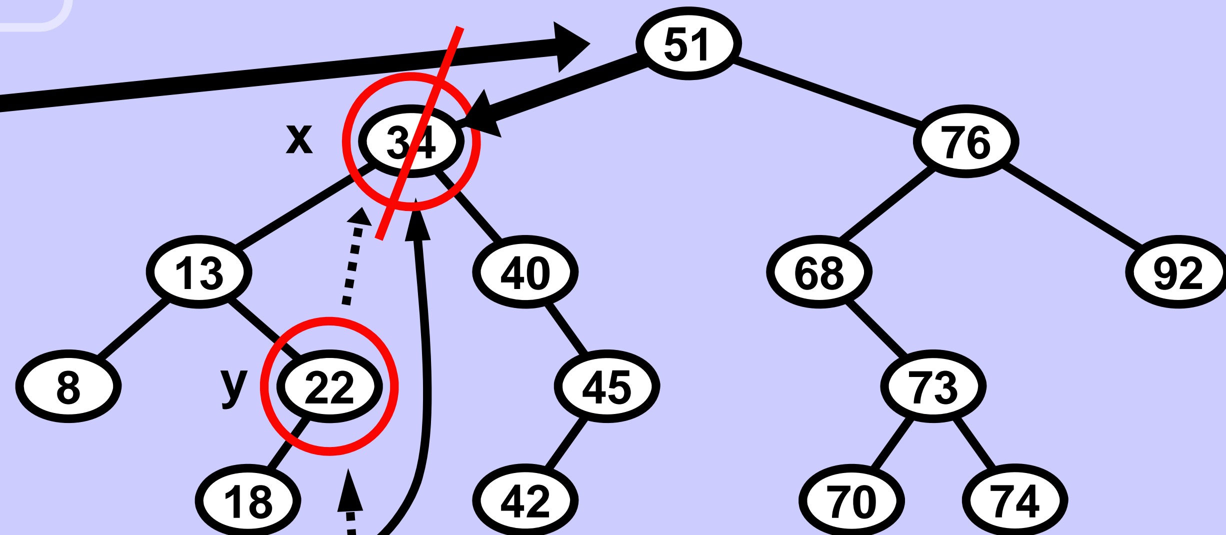
old
edges/pointers/references

new
edges/pointers/references



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

Delete a node with 2 children.



Delete IIIb.

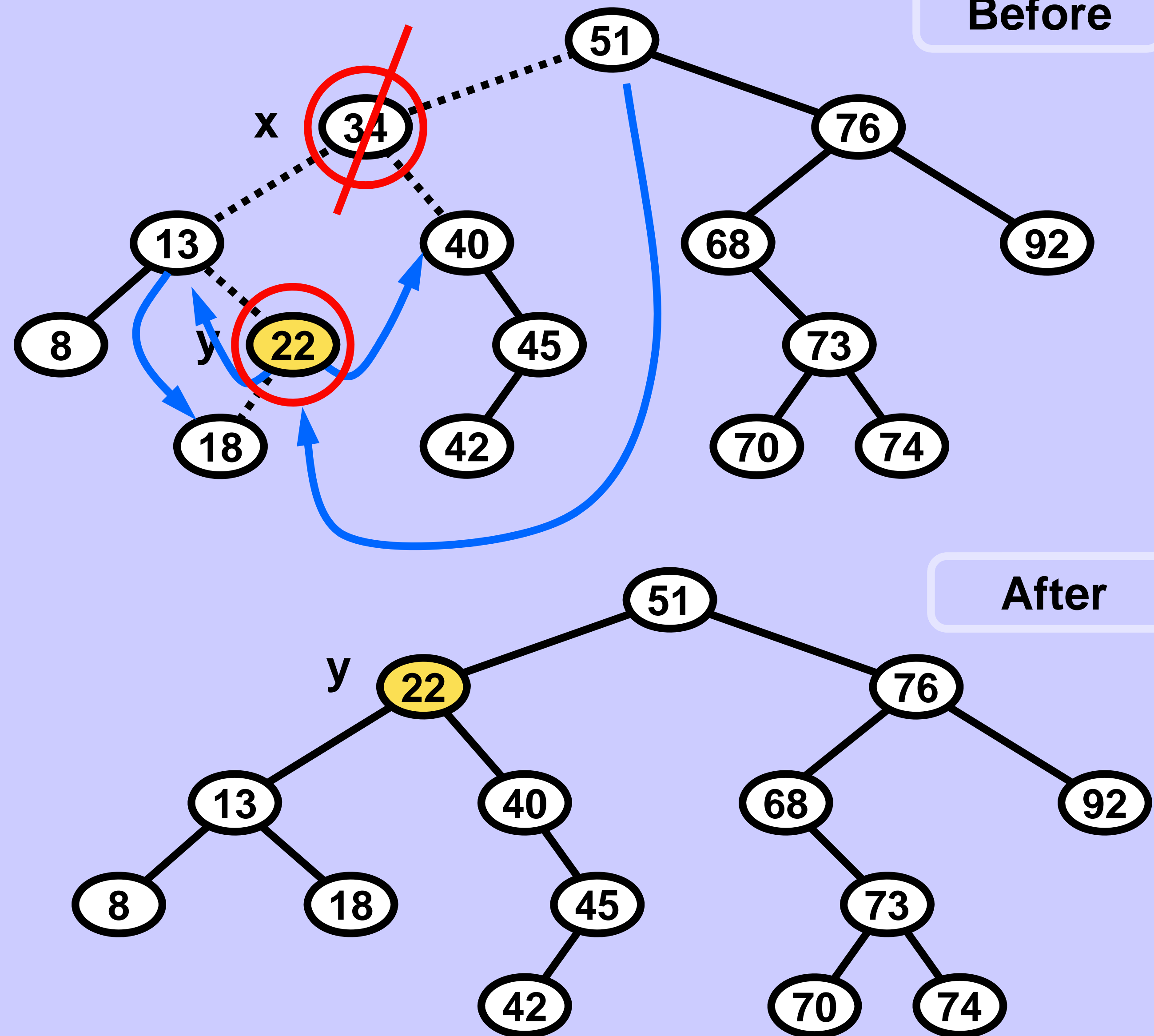
1. Find the node (like in Find operation) with the given key and then find the ***rightmost*** (= smallest key) node y in the ***left*** subtree of x.
2. Point from y to children of x, from parent of y point to the child of y instead of y, from parent of x point to y.

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

```
Node delete (int k, Node node) {
   ...                              // 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	$\Theta(\log(n))$	$O(n)$