Estruturas de Informação

Trees

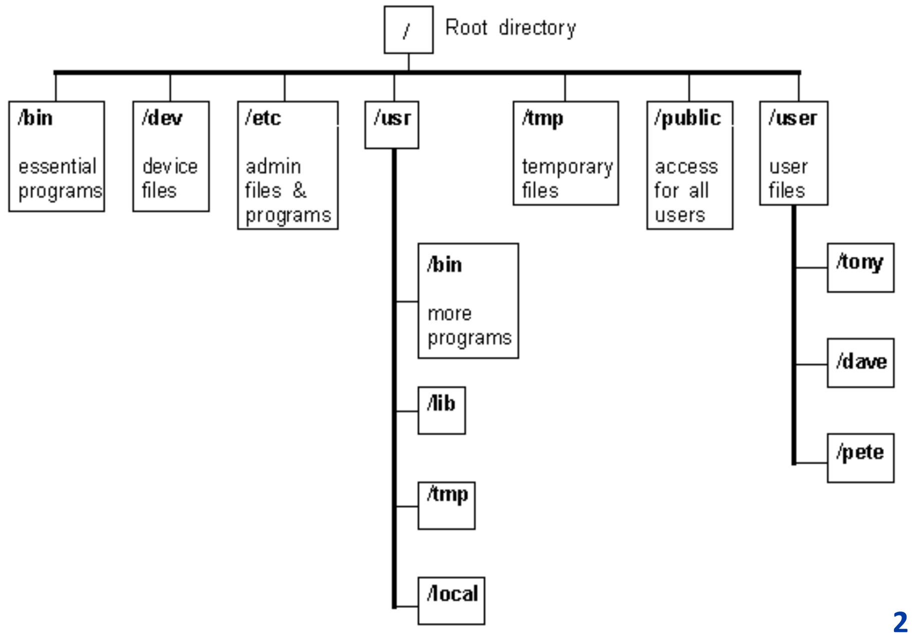
Fátima Rodrigues
mfc@isep.ipp.pt
Departamento de Engenharia Informática (DEI/ISEP)

Trees

- In computer science, a tree is an ADT which stores elements hierarchically
- Tree consists of nodes with a parent-child relation

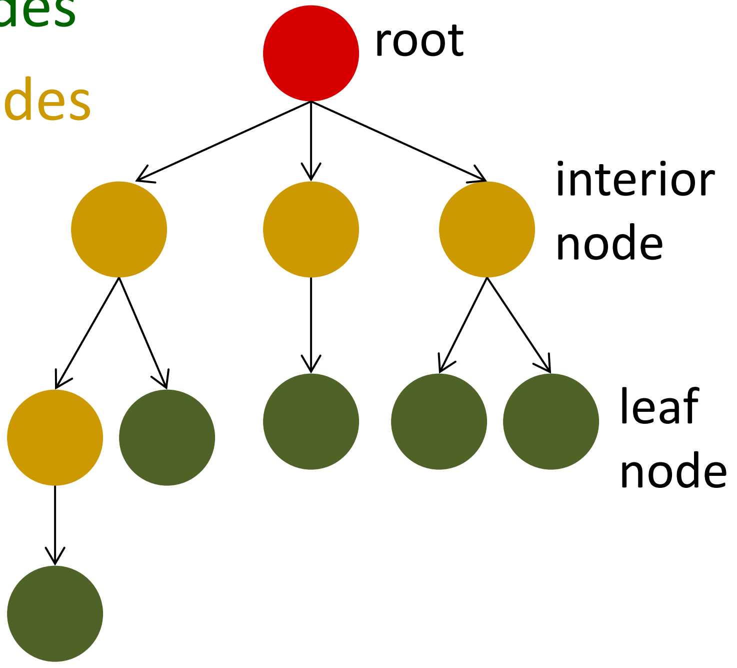
Applications:

- File systems
- Programming environments
- Taxonomies
- Image Representation
- Database Indexes
- ….



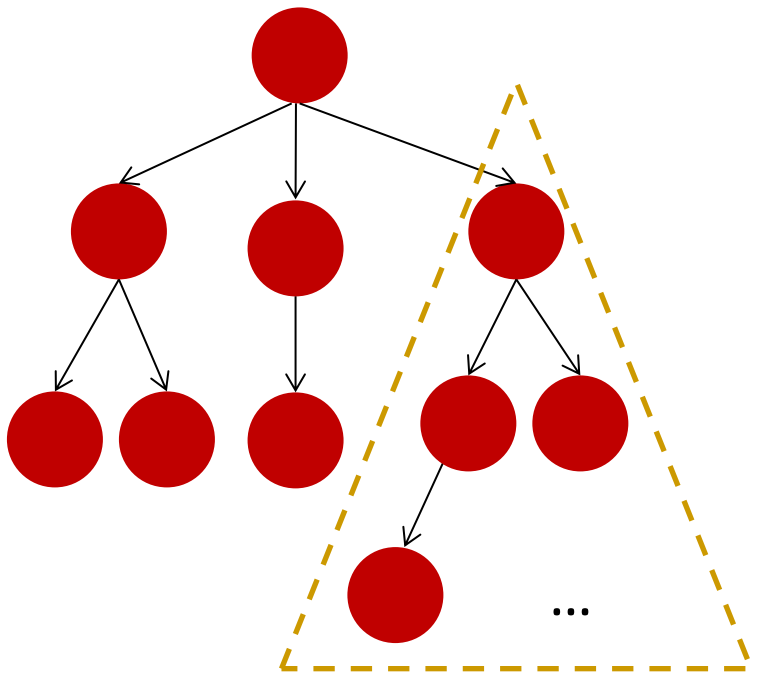
Tree Terminology

- Tree = Set of nodes connected by arcs (or edges)
- Every tree has a single root node – node without parent node
- A parent node points to (one or more) other nodes
- Nodes pointed to are children
- Every node (except the root) has exactly one parent
- Nodes with no children are leaf nodes
- Nodes with children are interior nodes



Tree Terminology

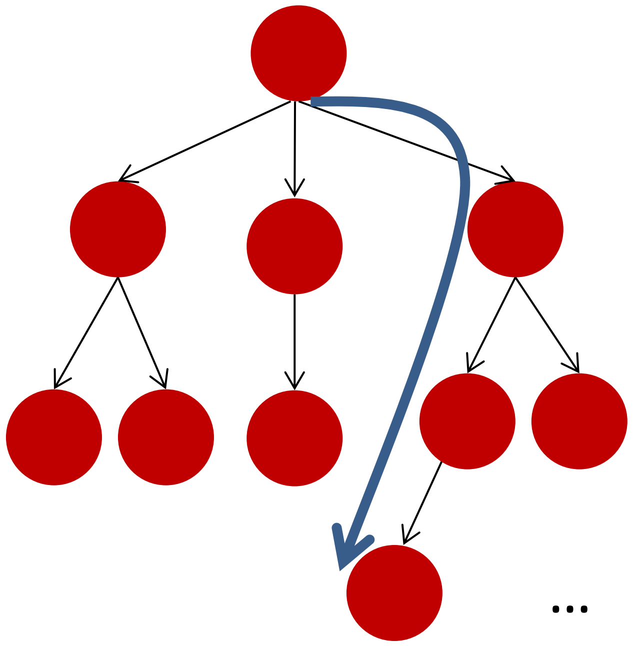
- Any node can be considered the root of a subtree



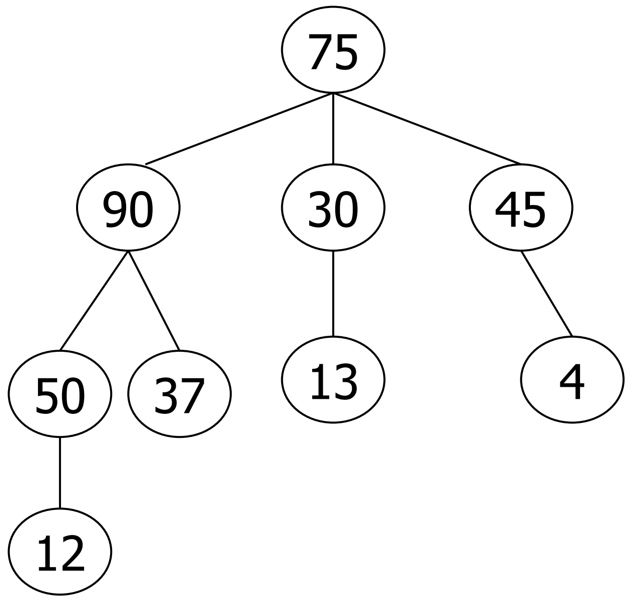
How many subtrees are there?

Tree Terminology

- There is a single, unique path from the root to any node
- A path's length is equal to the number of arcs traversed



Ascendants and descendants of a Node

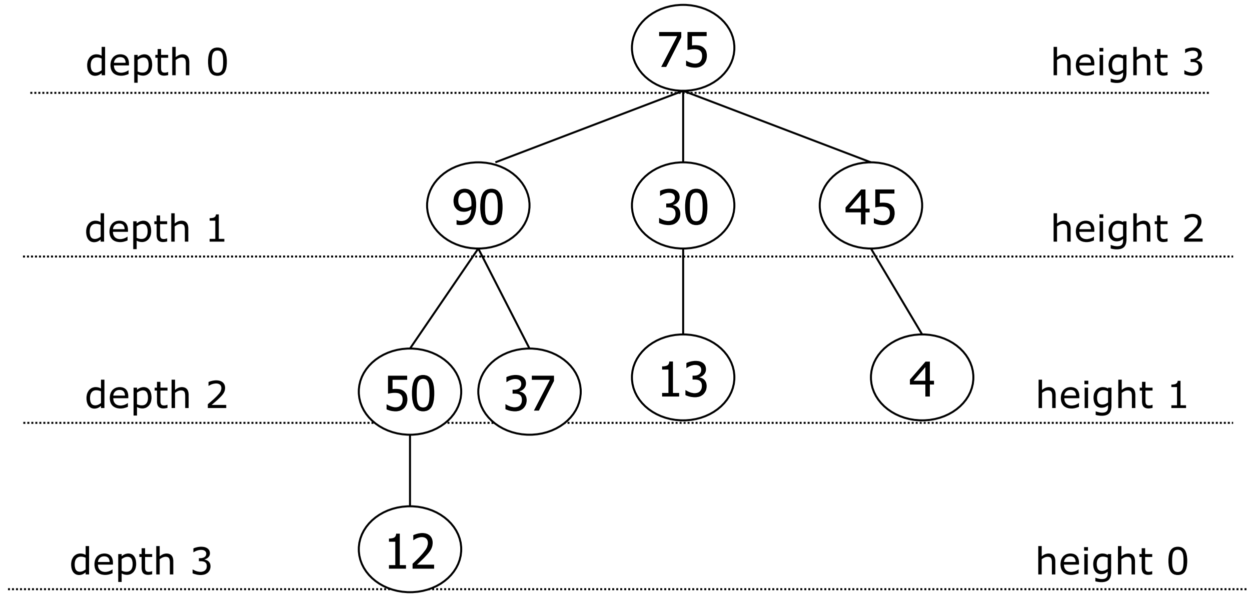


- The ascendants of a node are the nodes that are in the path from the node to the root of the tree. Ascendants of node 12: 50, 90, 75
- The descendants of a node are all the nodes reachable from that node. Descendants of node 90: 50, 37, 12
- All nodes in a tree are descendants of the root

  (except for the root)

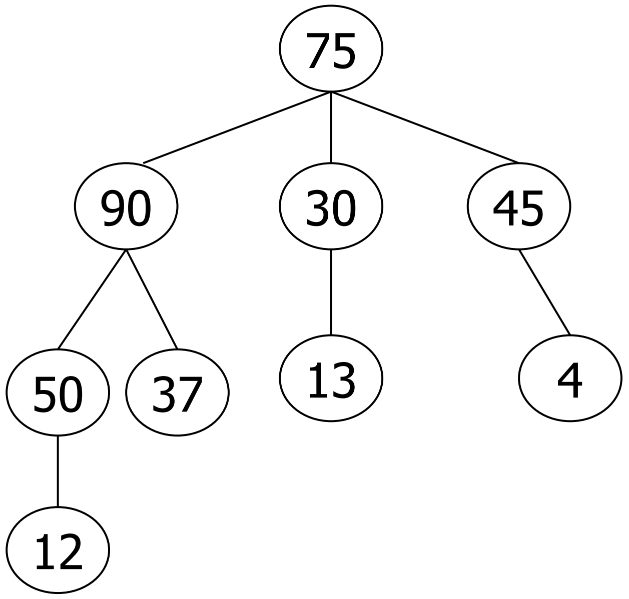
Height and depth of a tree

- Height of a node = max. path length from the node to a leaf
  - Height of a leaf node = 0 (greater depth)
  - Height of the tree = Height of the root
- Depth of a node = path length from the root to that node
  - Depth of the root = 0



Degree of a tree

- Degree of a tree is the maximum degree of its nodes
- Degree of a node is equal the number of its children's



Degree of node 90: 2
Degree of a leaf node: 0
Degree of the tree: 3

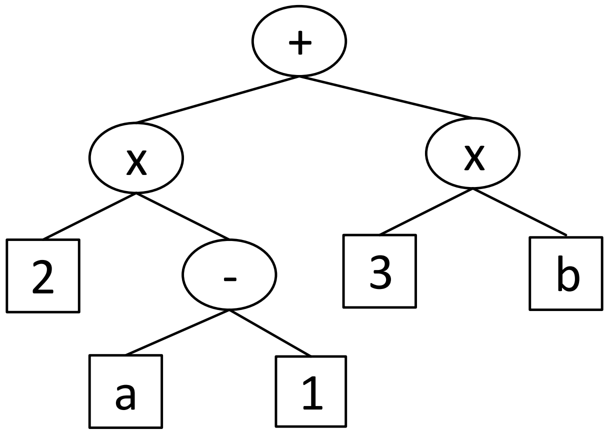
Binary Trees

Binary Tree – Definition

- A binary tree is a special case of a K-ary tree whose nodes have exactly two child references
- A binary tree is a rooted tree in which no node can have more than two children AND the children are distinguished as left and right

Applications:

- Arithmetic expressions
- Decision processes
- Searching
- ....



Arithmetic expression: ((2 x (a - 1)) + (3 x b))

Binary Tree – Properties

In a binary tree

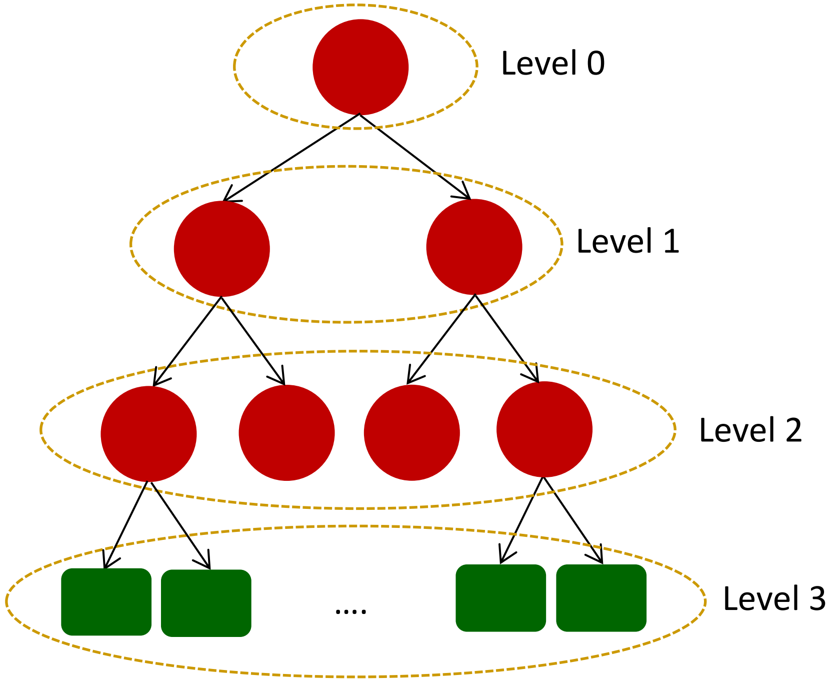
- level 0 has at most 1 = 2^0 node
- level 1 has at most 2 = 2^1 nodes
- level 2 has at most 4 = 2^2 nodes

…

- level d has at most 2^d nodes

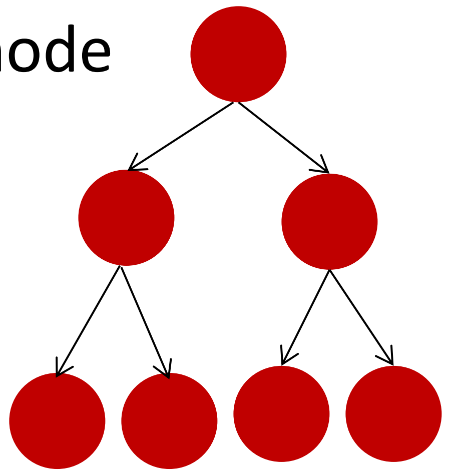
A binary tree of height h has:

- minimum: h + 1 nodes
- maximum: $2^{h+1} - 1$ nodes



Binary Tree – Properties

A full binary tree is a binary tree in which every node is a leaf or has exactly two children



- A full binary tree with n internal nodes has n + 1 leaves

A perfect binary tree is a full binary tree in which all leaves have the same depth

- The number of nodes in a perfect binary tree is $2^{h+1} - 1$ nodes, where h is height

$$n = 2^{h+1} - 1$$

$$2^{h+1} = n + 1$$

$$\log_2 (2^{h+1}) = \log_2 (n + 1)$$

$$h = \log_2 (n + 1) - 1$$

Tree Traversals

Depth-First Traversals

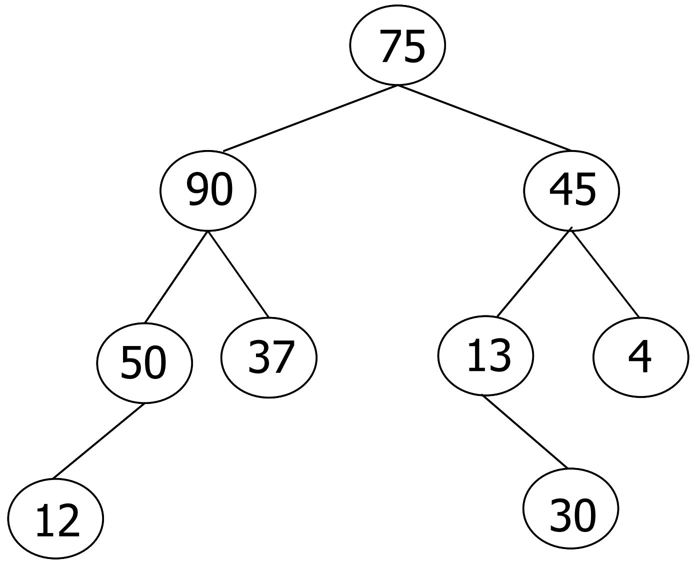
- Pre-order – root, left subtree, right subtree
- In-order – left subtree, root, right subtree
- Pos-order – left subtree, right subtree, root

Breadth-First Traversal

- Level-order – all the positions at depth *d are visited* before the positions at depth *d* + 1

- A breadth-first traversal is a common approach used in software for playing games

Tree Traversals - Exemplification



Depth-First

• Pre-order	75, 90, 50, 12, 37, 45, 13, 30, 4
• In-order	12, 50, 90, 37, 75, 13, 30, 45, 4
• Pos-order	12, 50, 37, 90, 30, 13, 4, 45, 75

Breath-First

• Level-order	75, 90, 45, 50, 37, 13, 4, 12, 30

Pre-Order Traversal

In a preorder traversal the node is visited before both its subtrees, left and right

```
Algorithm void preOrder(Node<E> node){

    if (node == null)
       return;

    visit(node)
    preOrder(node.getLeft())
    preOrder(node.getRight())
}
```

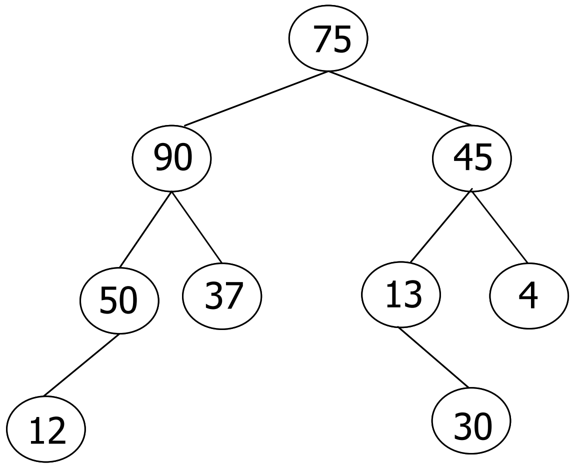
Time Complexity: O(?)

If visit(node) is O(1), then the complexity of preOrder is O(n)

Pre-Order Traversal – iterative algorithm

The iterative algorithm needs an auxiliary stack

```
Algorithm void IterativepreOrder(Node<E> node){

    r = node
    do {
         while (r != null){
            visit(r)
            stk.push(r)
            r=r.getLeft()
         }
         if (!stk.isEmpty()){
            stk.pop()
            r=r.getRight()
         }
    } while (stk.isEmpty() && r != null)
}
```



In-Order Traversal

In an in-order traversal a node is visited after its left subtree and before its right subtree

```
Algorithm void inOrder(Node<E> node){

  if (node == null)
     return;
  inOrder(node.getLeft())
  visit(node)
  inOrder(node.getRight())
}
```

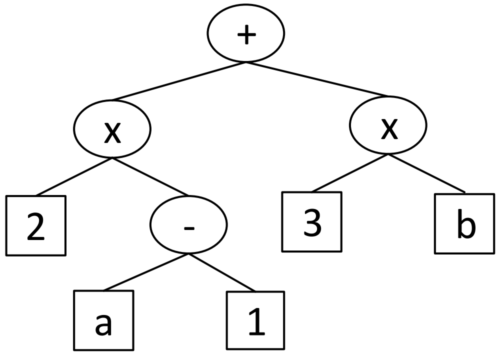
Specialization of In-Order Traversal:

writing of an arithmetic expression

Specialization of In-Order Traversal

Write an Arithmetic Expression

- write operand or operator when visiting node
- write “(“ before traversing left subtree
- write “)“ after traversing right subtree



((2 x (a - 1))+(3 x b))

```
Algorithm void  writeExpression(Node<String> node, String str){

   if (node.getLeft()){
      str += “(“
      writeExpression(node.getLeft(),str)
   }
   str += node.getElement()
   if (node.getRight()){
      writeExpression(node.getRight(),str)
      str += “)“
   }
}
```

Pos-Order Traversal

In a pos-order traversal a node is visited after both its subtrees, left and right

```
Algorithm void posOrder(Node<E> node){

  if (node == null)
     return;

  posOrder(node.getLeft())
  posOrder(node.getRight())
  visit(node)
}
```

Iterative Algorithm: needs two stacks

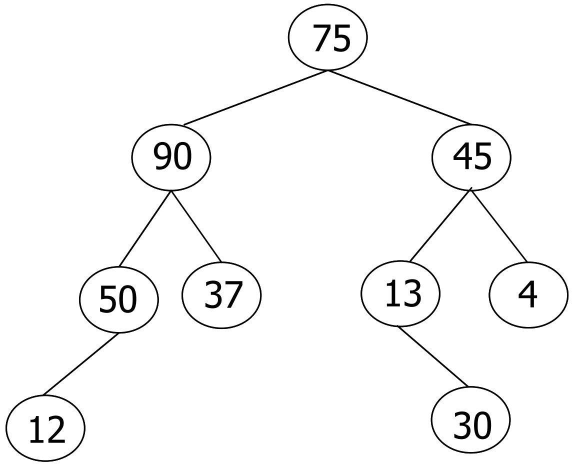
Specialization of a Pos-Order Traversal:

evaluation an arithmetic expression

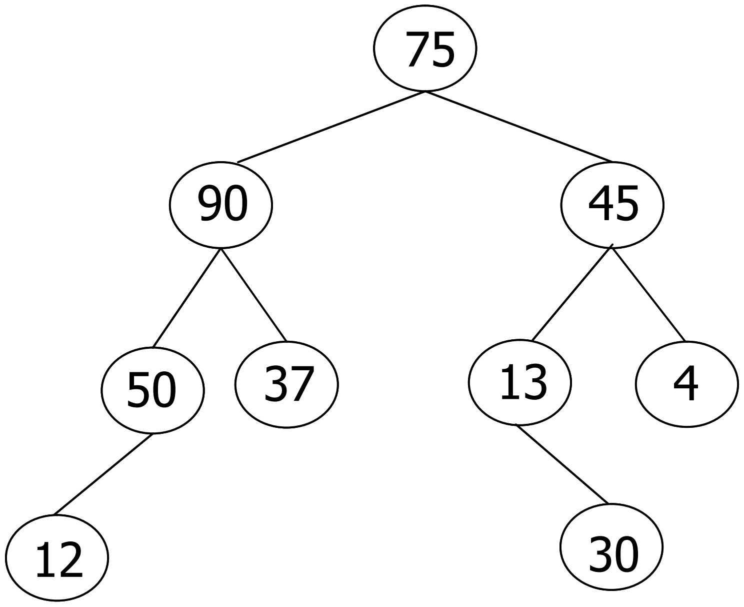
Breadth First Traversal

In a breadth first traversal the nodes of the tree are visited level by level

```
Algorithm void breadthfirst (){
   Initialize queue Q to contain root()
   while (Q not empty) {
      p = Q.dequeue()
      visit(p)
      for (each child c in children(p))
         Q.enqueue(c)
   }
}
```



Search an Element



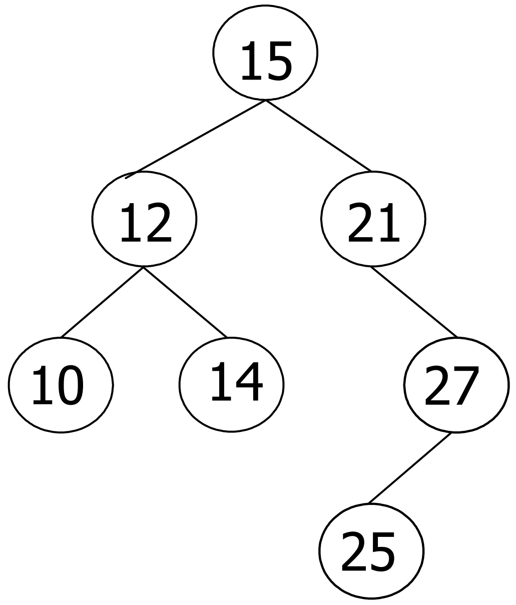
Time Complexity: O(?)

Binary Search Trees

Binary Search Tree (BST)

Is a binary tree where every node value is:

- **Greater than** all its left descendants
- **Less than** to all its right descendants



The elements in the BST must be comparable

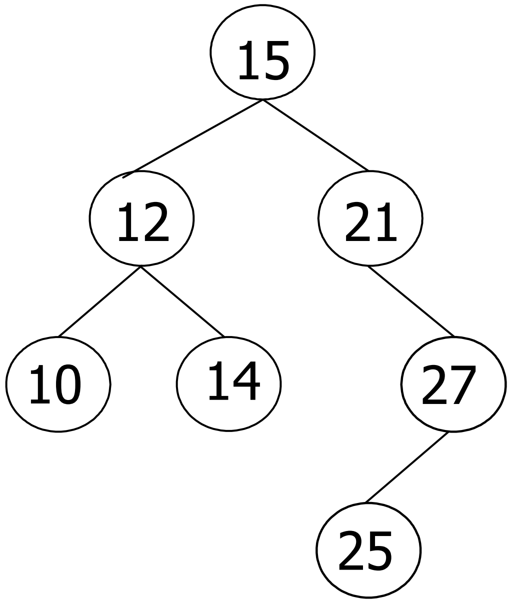
Duplicates are not allowed

Each subtree of a BST is also a BST

Applications:

- Symbol tables in compilers, "assemblers“
- Used in implementing efficient priority-queues (heaps)
- ….

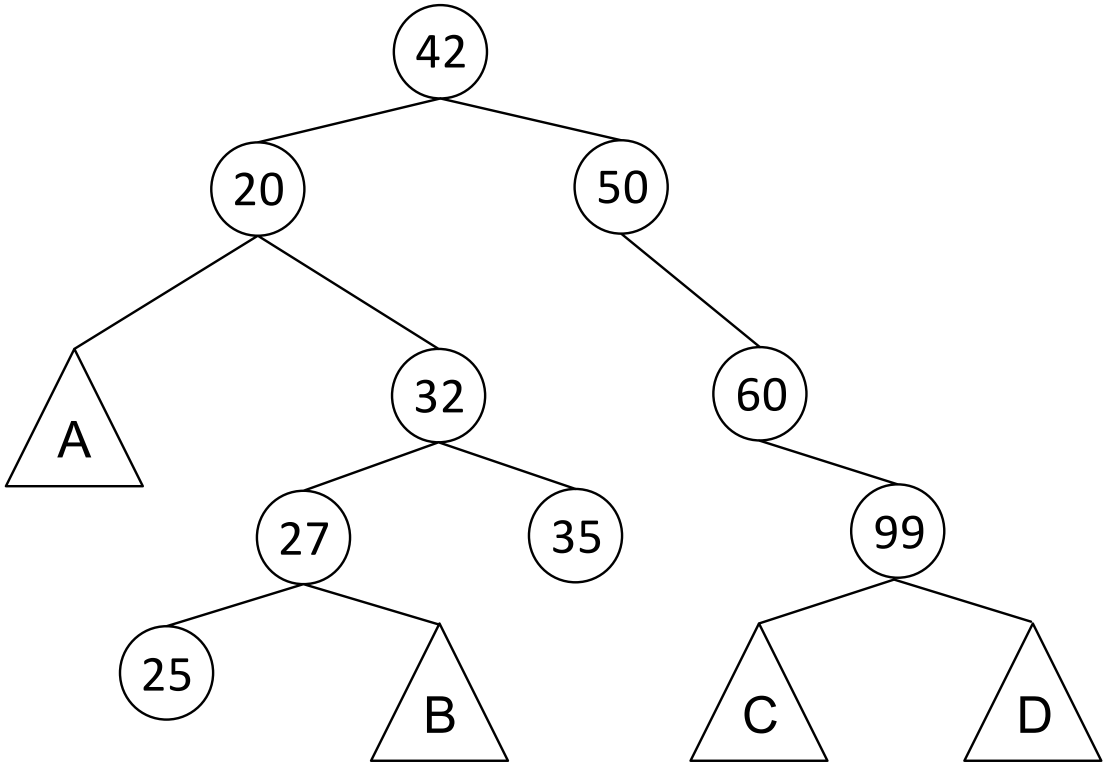
Binary Search Tree



• In-order:	10, 12, 14, 15, 21, 25, 27
• Pre-order:	15, 12, 10, 14, 21, 27, 25
• Pos-order:	10, 14, 12, 25, 27, 21, 15
• Level-order:	15, 12, 21, 10, 14, 27, 25

BST in-order traversal returns elements in sorted order

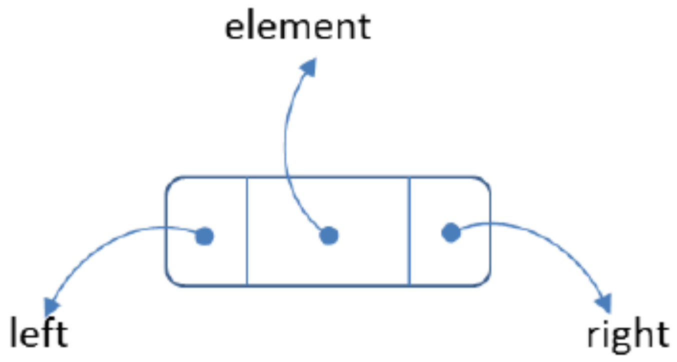
A Binary Search Tree of Integers



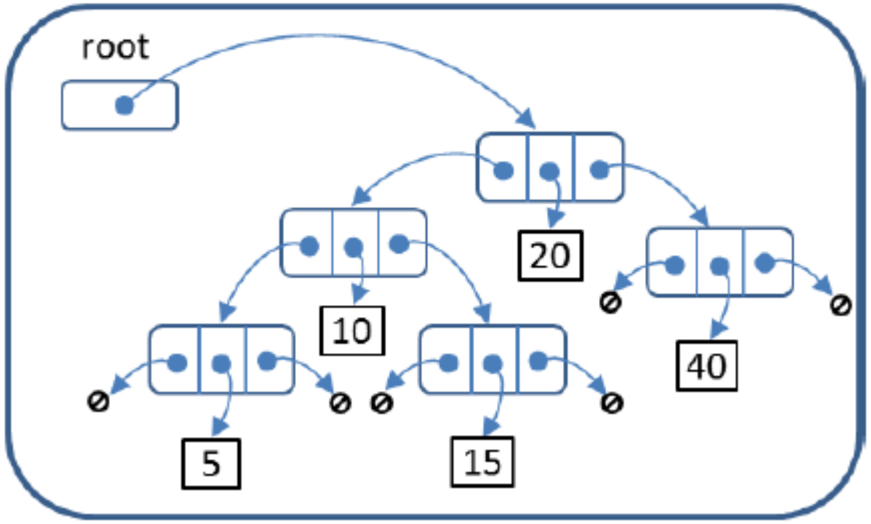
Describe the values which might appear in the subtrees labeled A, B, C, and D

Binary Search Tree ADT

Node

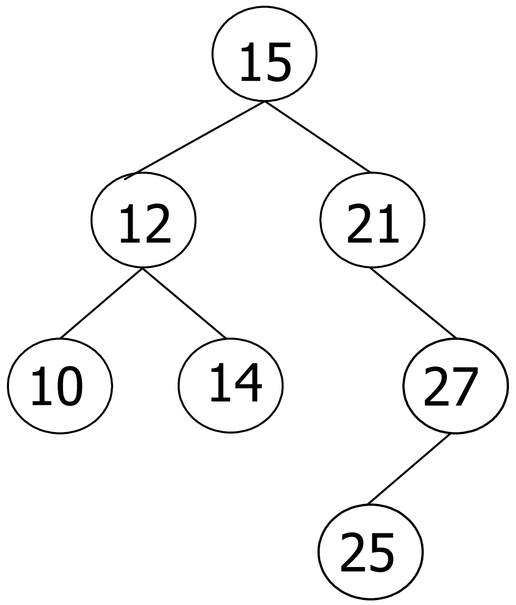


BST



```
public BST()
public boolean isEmpty()
public int size()
public void insert(E element)
public void remove(E element)
```

Search for an Element



- Start at root
- At each node, compare value to node value:
  - Return true if match
  - If value is less than node value, go to left child (and repeat)
  - If value is greater than node value, go to right child (and repeat)
  - If node is null, return false

Time Complexity

The maximum number of comparisons to conclude whether or not the key is in the tree is the maximum height of tree: h

If the tree is (more or less) balanced, all the leaf nodes with the same depth, the height of the tree can be relate with the total number of elements n

$$n = 2^{(h+1)} - 1$$
$$2^{(h+1)} = n + 1$$
$$h+1 = \log_2 (n+1)$$
$$h = \log_2 (n+1) - 1$$

For all values of $n \geq 1$, there is a constant C, such that:

$$\log_2 (n+1) - 1 \leq C \times \log_2 n$$

$T(n) = O(\log n)$

Search for an Element

```
Algorithm Node<E> search(Node<E> node, E elem){
   if (node == null)
      return null
   if (node.getElement() == elem)
      return node
   if (node.getElement() > elem)
      return search(node.getLeft(),elem)
   else
       return search(node.getRight(),elem)
}
```

Time Complexity: O(?)

- **Best case**: the element is at the root
- **Average Case**: the tree is balanced
- **Worst Case**: the element doesn't exist, the tree degenerates in a list

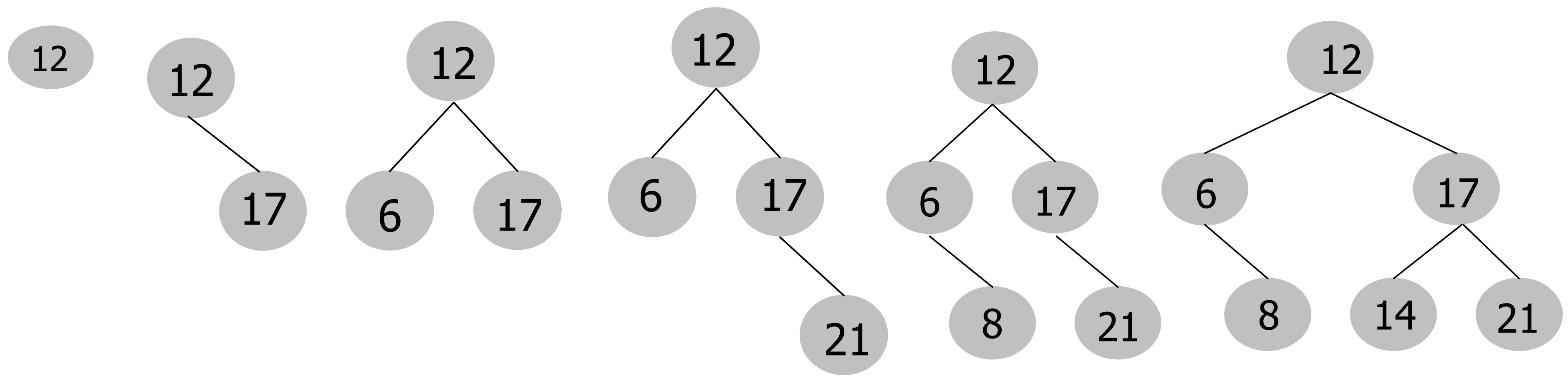
Search for an Element – iterative version

```
Algorithm boolean search(E elem) {
  node = root
  find = false
  while (node != null && !find){
     if (node.getElement() == elem)
        find = true
     if (node.getElement() > elem)
        node = node.getLeft()
     if (node.getElement() < elem)
       node = node.getRight()
  }
  return find
}
```

Insertion

- Start at the root
- successively down the tree from the root choosing the appropriate sub-tree
- Arriving in a leaf, insert in the appropriate side

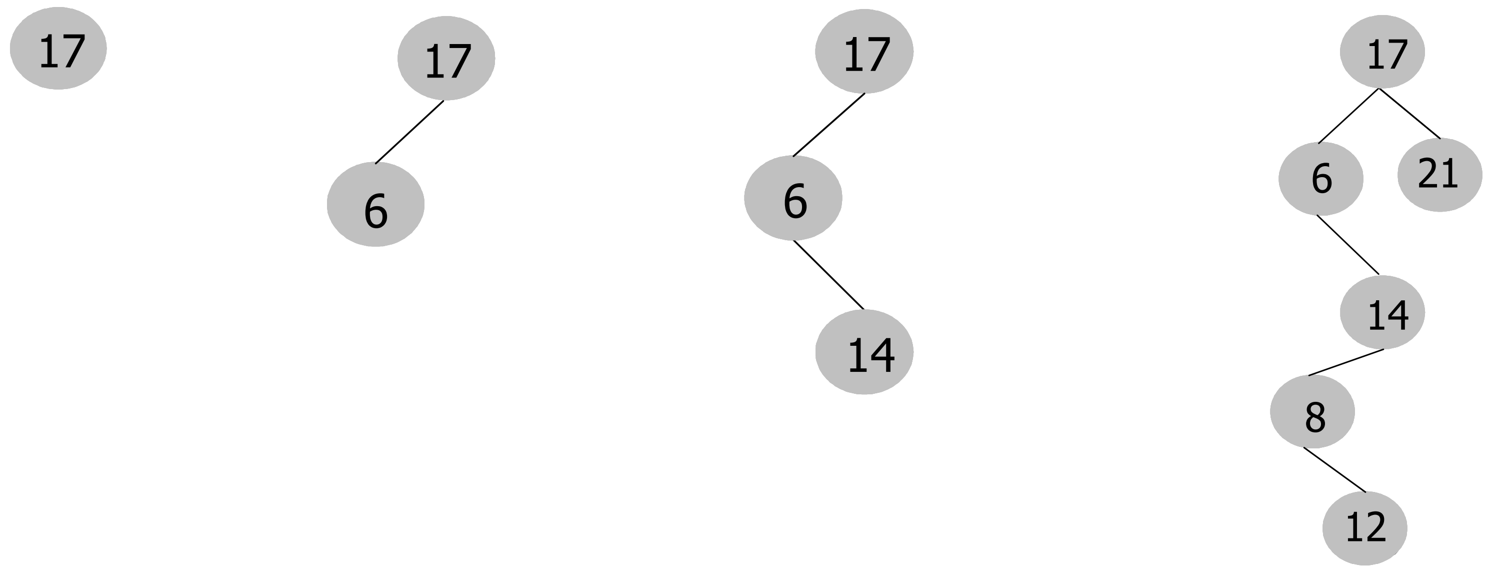
The shape of the tree depends on the order of elements insertion:
12, 17, 6, 21, 8, 14



Insertion

The shape of the tree depends on the order of elements insertion:

17, 6, 14, 21, 8, 12



What happens if the elements are inserted into the tree in ascending or descending order?

Insertion

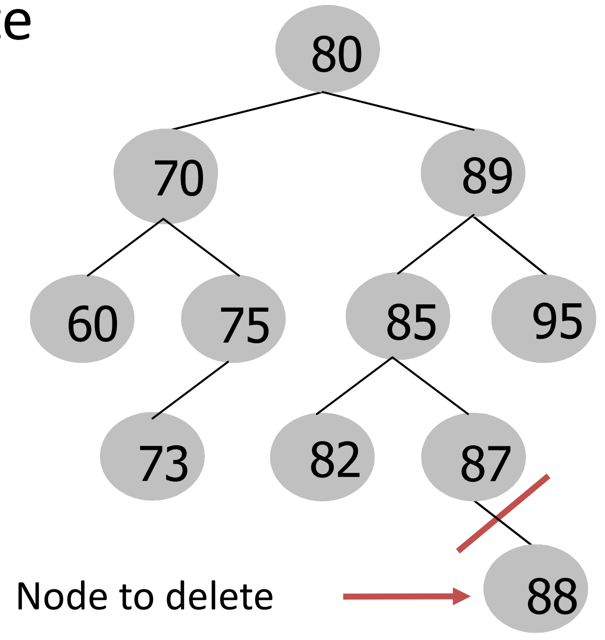
```
Algorithm Node<E> insert(Node<E> node, E elem){

    if (node == null)
       return new Node(elem, null, null)

    if (node.getElement() > elem)
        node.setLeft(insert(node.getLeft(),elem))
    else
        if (node.getElement() < elem)
           node.setRight(insert(node.getRight(),elem))

    return node
}
```

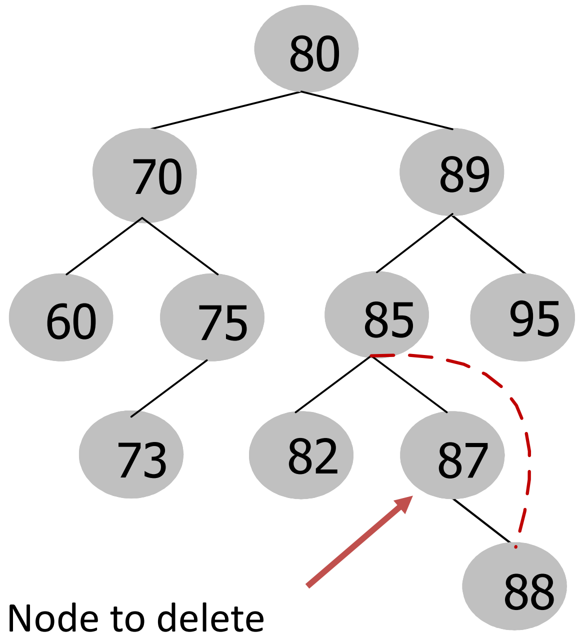
Deletion

When delete a node three cases can happen:

1. the node is a leaf (it hasn't subtrees)
2. the node has only one subtree
3. the node contains two subtrees (left and right)

The first two cases (1 and 2) are solved by adjusting the pointer of the previous node (parent node) that points to the node we want to eliminate





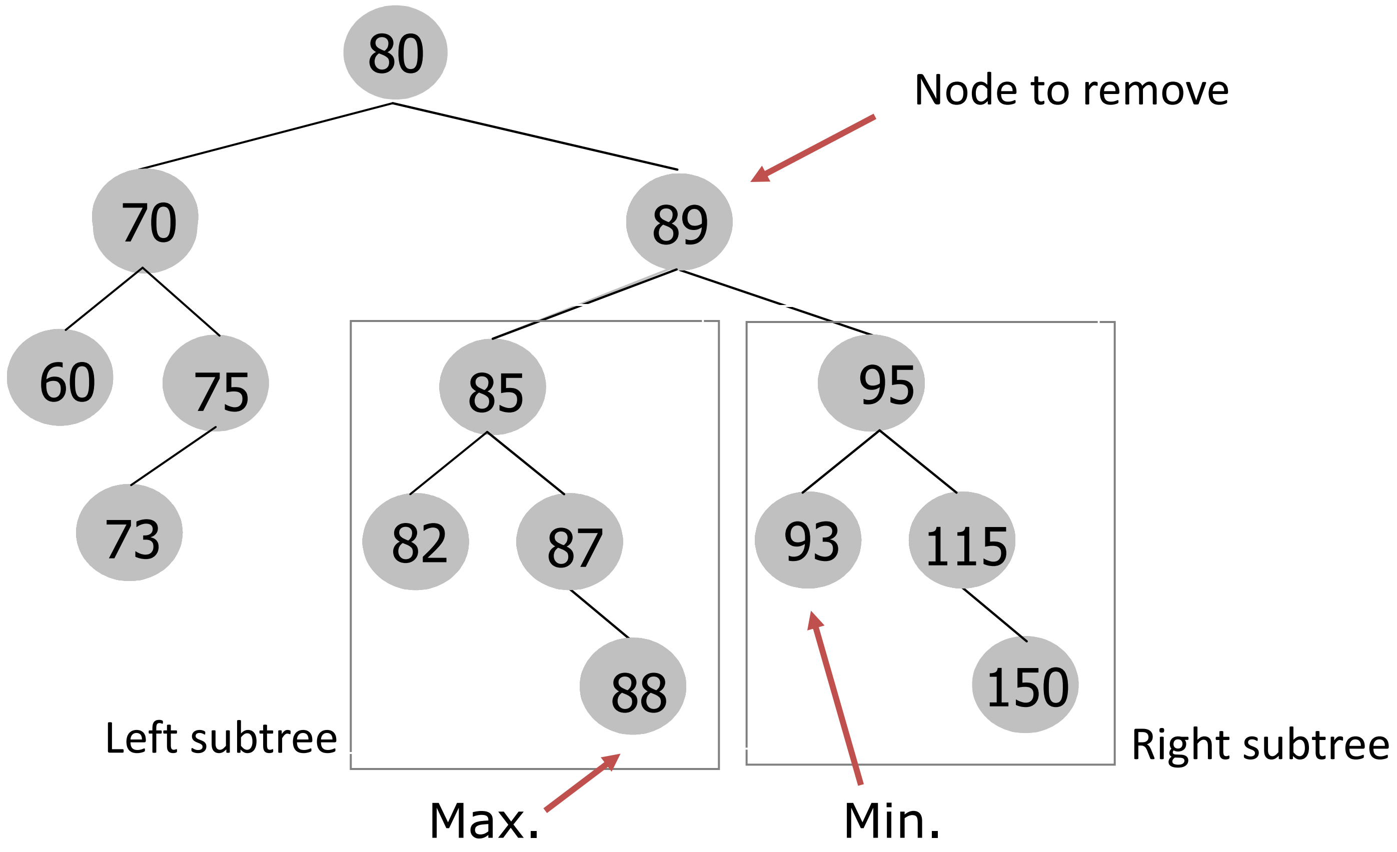
Deletion

Case 3:

- replace the node to eliminate with the greatest node of the left subtree of the node to delete

or

- replace the node to eliminate with the smaller node of the right subtree of the node to eliminate



Deletion

```
Algorithm Node<E> remove(E elem, Node<E> node) {
    if (node == null)
       return null

    if (node.getElement() == elem) {

        if (node.getLeft() == null && node.getRight()== null)
           return null

        if (node.getLeft() == null)
           return node.getRight()

        if (node.getRight() == null)
           return node.getLeft()

        E min = smallestElement(node.getRight())
        node.setElement(min)
        node.setRight(remove(min, node.getRight()))  }
    else if (node.getElement() > elem)
         node.setLeft(remove(elem,node.getLeft()))
    else
         node.setRight(remove(elem,node.getRight()))
    return node }
```

BST tree - Sorting

A BST tree can be used to sort a collection of values:

1. Insert data into the BST tree: O(??)
2. Copy data from BST tree into the collection using the ?? traversal: O(??)

Execution time: O(nlog n)

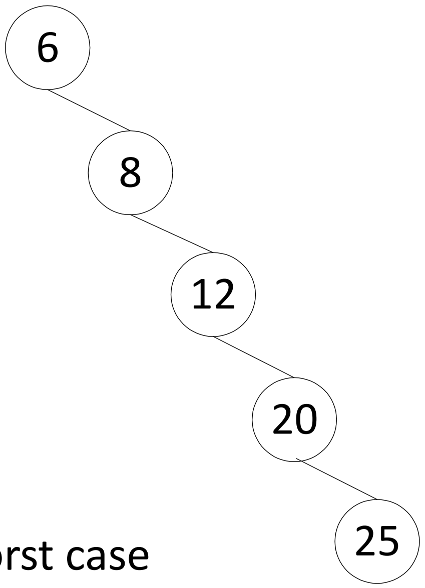
- Matches that of quicksort in benchmarks
- Unlike quicksort, BST trees don’t have problems if data is already sorted or almost sorted (which degrades quicksort to $O(n^2)$)
- However, requires extra storage to maintain both the original data buffer and the tree structure

Performance BST methods

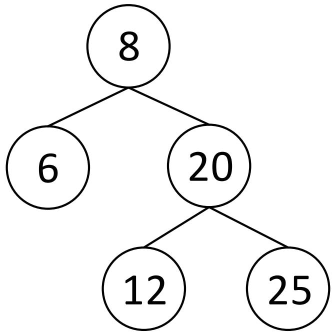
The analysis of search, insert and remove is similar

- In each case, h nodes are visited
- If each node is visited at O(1)
- The methods take O(h) time

The height h is O(n) in the worst case and O(log n) in the best case



worst case



best case

To make sure height h of a tree is always O(log n), the tree must be balanced

Balanced Trees or AVL Trees

Balanced Trees

- In a balanced tree for all of its nodes the height of the left subtree is approximately equal to the height of the right subtree, which guarantees that the height of the tree is always O(logn)

- This is achieved by an extra processing cost on the construction of the tree to maintain it balanced, but this is compensated when the data is often retrieved

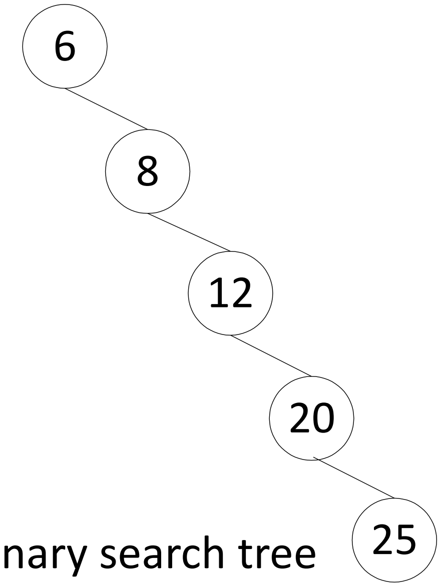
- The idea of maintaining a balanced binary tree dynamically i.e., as nodes are inserted/removed, was proposed in 1962 by two Soviet called Adelson-Velskii and Landis - AVL tree

AVL Tree - Definition

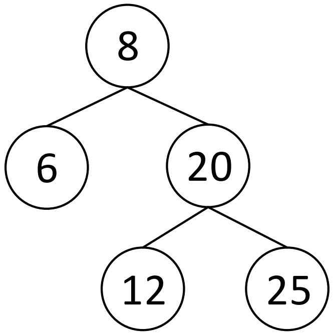
An AVL tree is a binary search tree such that for every internal node the heights of its children trees can differ by at most 1

Thus, each node has a Balance Factor (BF)

BF (node) = height (right subtree) - height (left subtree)



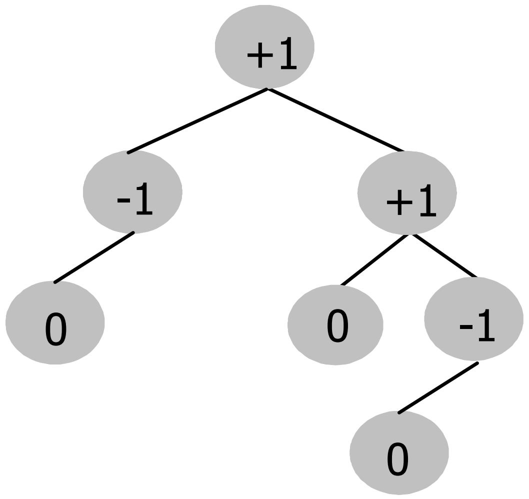
Binary search tree



AVL tree

Balance Factor (BF)

- Negative balance factor of a node means that the height of its left subtree is larger (in at least one node) than the height of its right subtree, left node heavy
- Positive balance factor of a node means that the height of its right subtree is larger (in at least one node) than the height of its left subtree, right node heavy
- Null balance factor of a node means that the height of the left subtree is equal to the height of the right subtree - node balanced

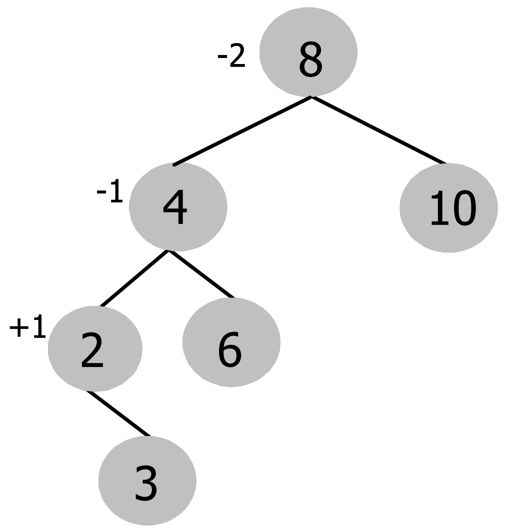


Balancing the Tree

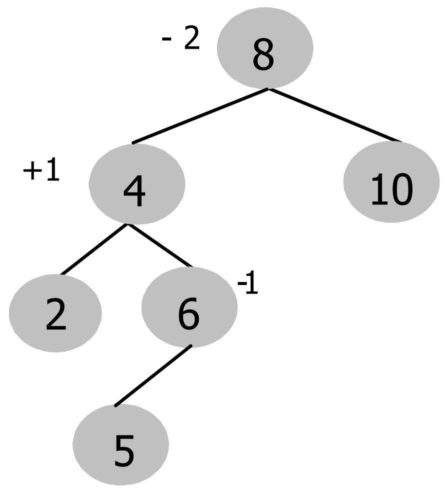
It is necessary whenever the insertion/removal of a node violates the tree balancing property: nodes in the tree with BF $\notin$ [-1,..,1]

The balancing of the tree is achieved with two kind of rotations:

- Simple - when the unbalanced node presents the same BF signal as its child's root node of unbalanced subtree
- Double - when the unbalanced node presents a BF signal contrary to its child's root node unbalanced subtree



Simple rotation

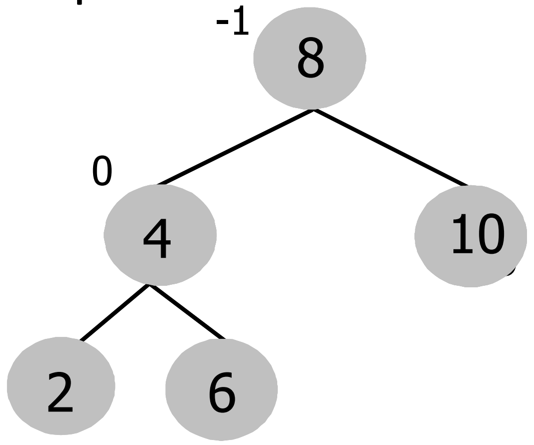


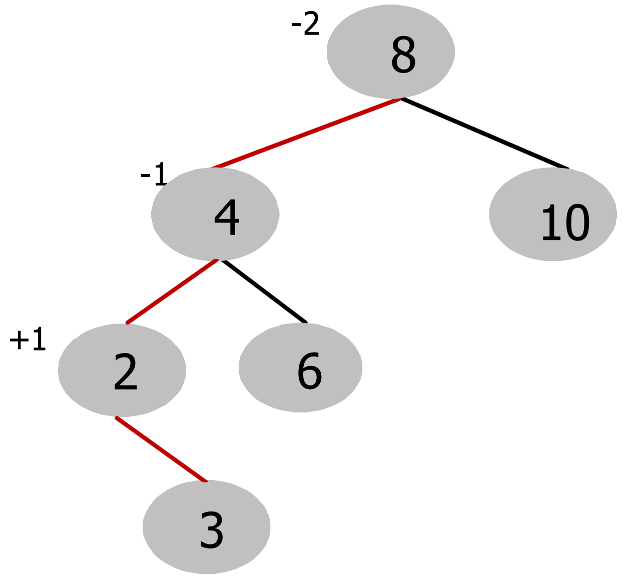
Double rotation

Balancing the Tree

There is a very important property in binary search trees:

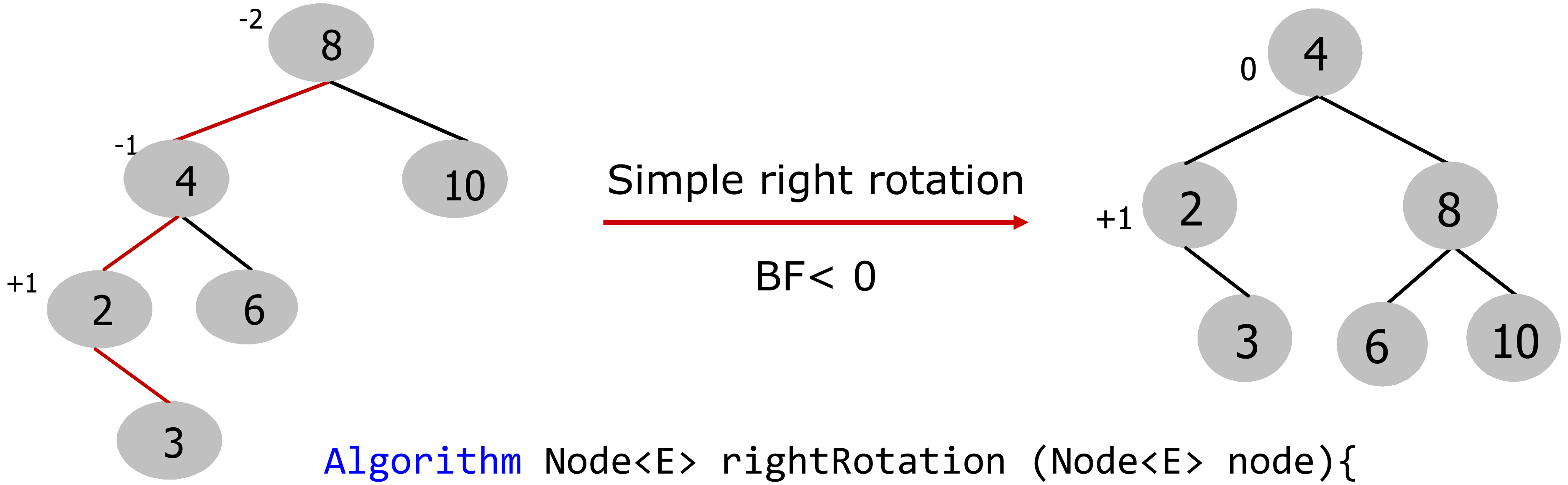
- after insertion/removal of a node it is warranted that only the nodes that are on the path between the root and the element inserted/removed can become imbalanced
- So, the balancing operations are only necessary on the nodes that are on that path





Simple right rotation

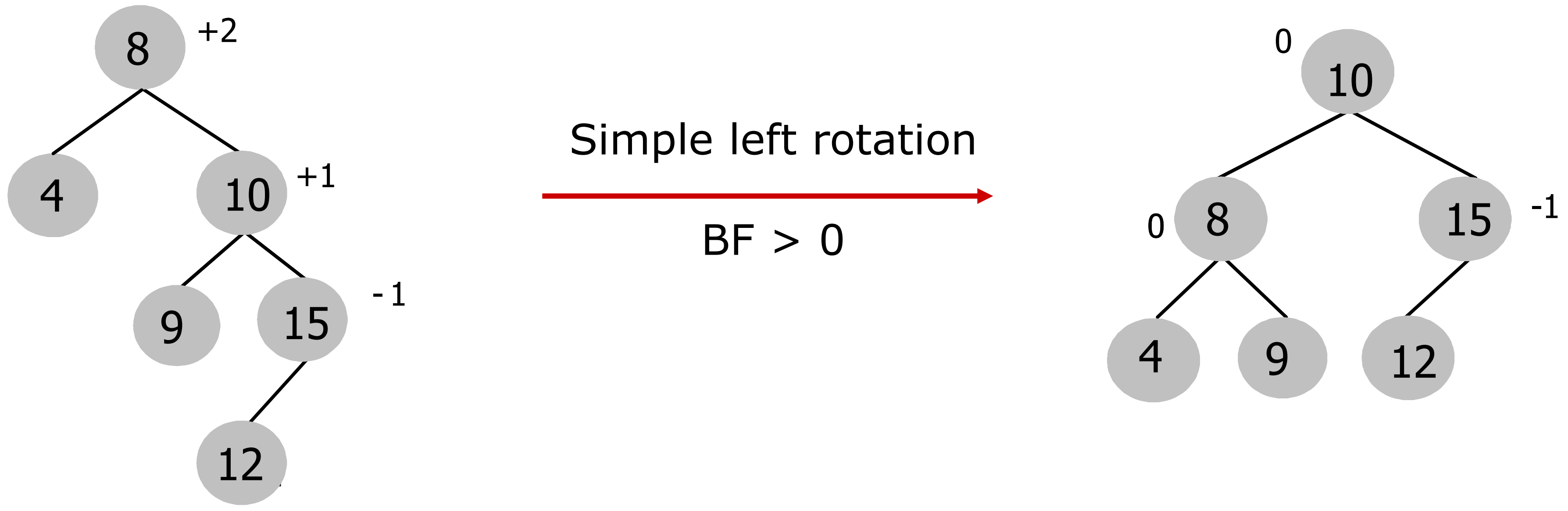
When the rotation is simple, it occurs always in the opposite direction of the tree imbalance



```
Algorithm Node<E> rightRotation (Node<E> node){

        Node<E>  leftson = node.getLeft()

        node.setLeft(leftson.getRight())
        leftson.setRight(node);

        node = leftson

        return node
}
```

Complexity(?)

Simple left rotation

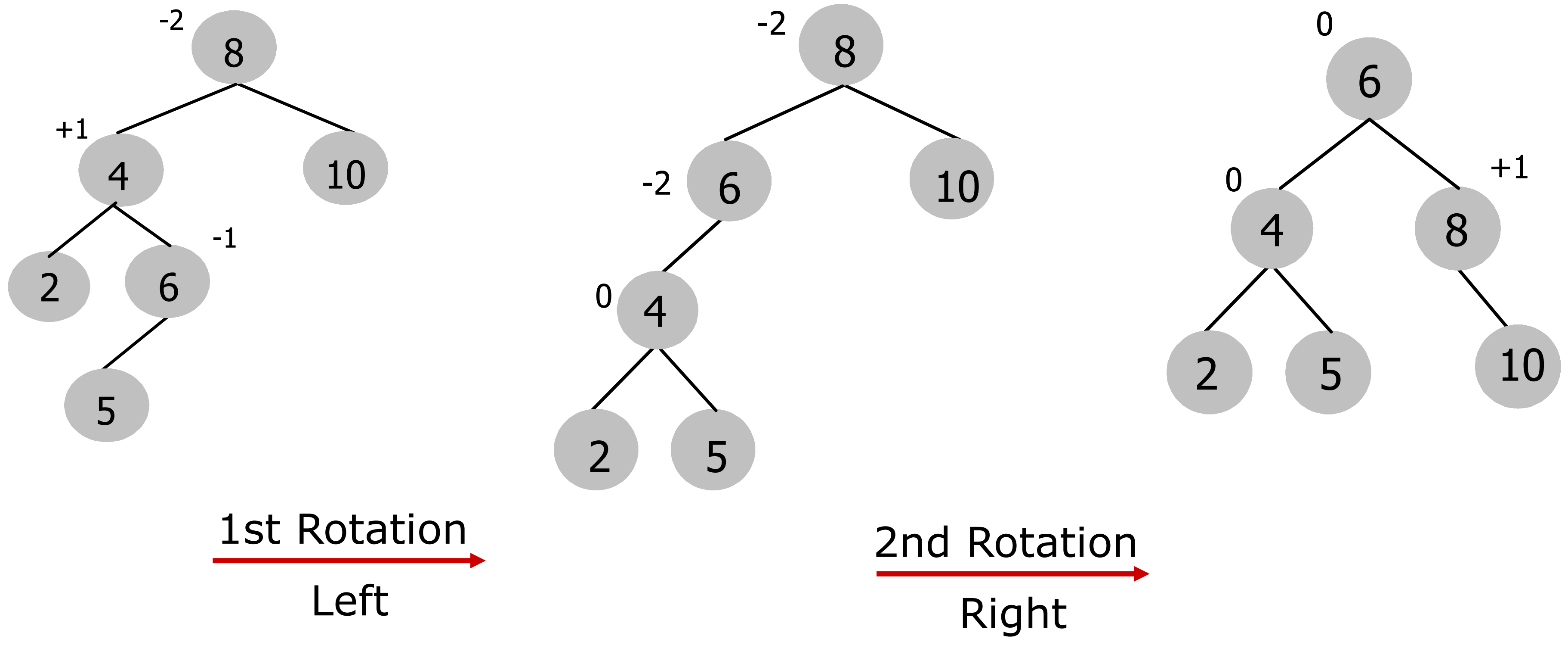


```
Algorithm Node<E> leftRotation (Node<E> node){
        Node<E>  rightson = node.getRight()
        node.setRight(rightson.getLeft())
        rightson.setLeft(node)
        node = rightson
        return node
}
```

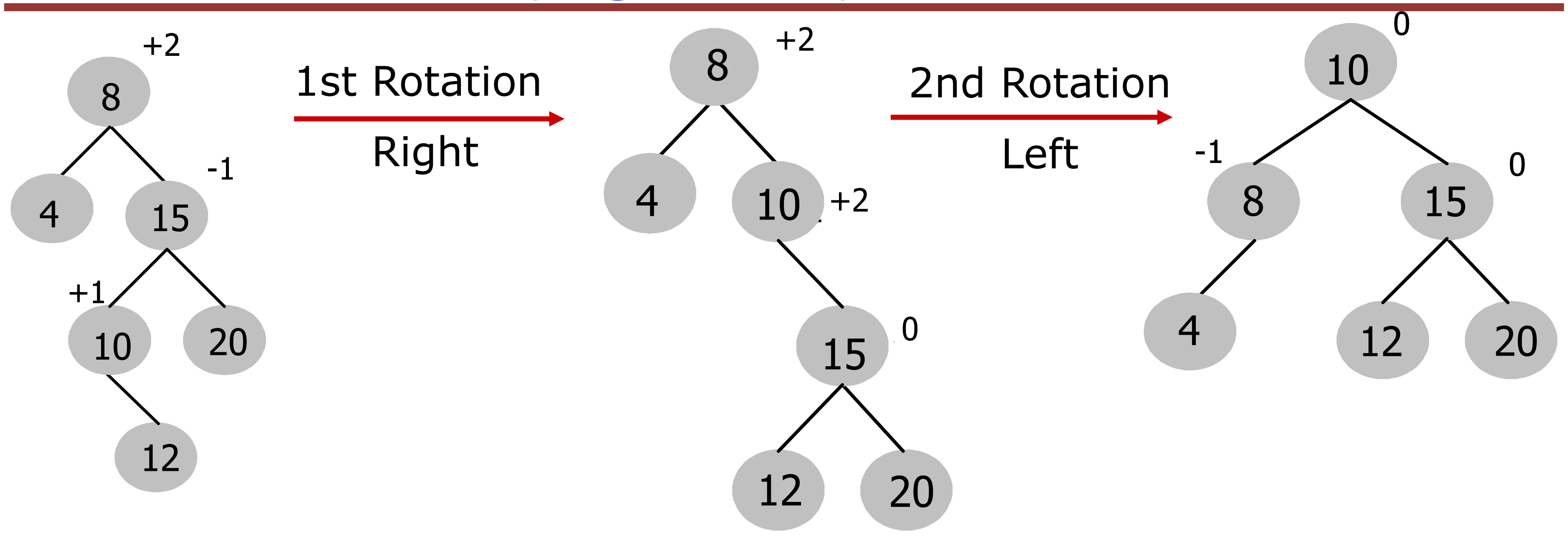
Complexity(?)

Double rotation (Left-Right)

- This case requires rotating the tree child in direction of the imbalance side
- The second rotation always occurs in the opposite direction of the first rotation



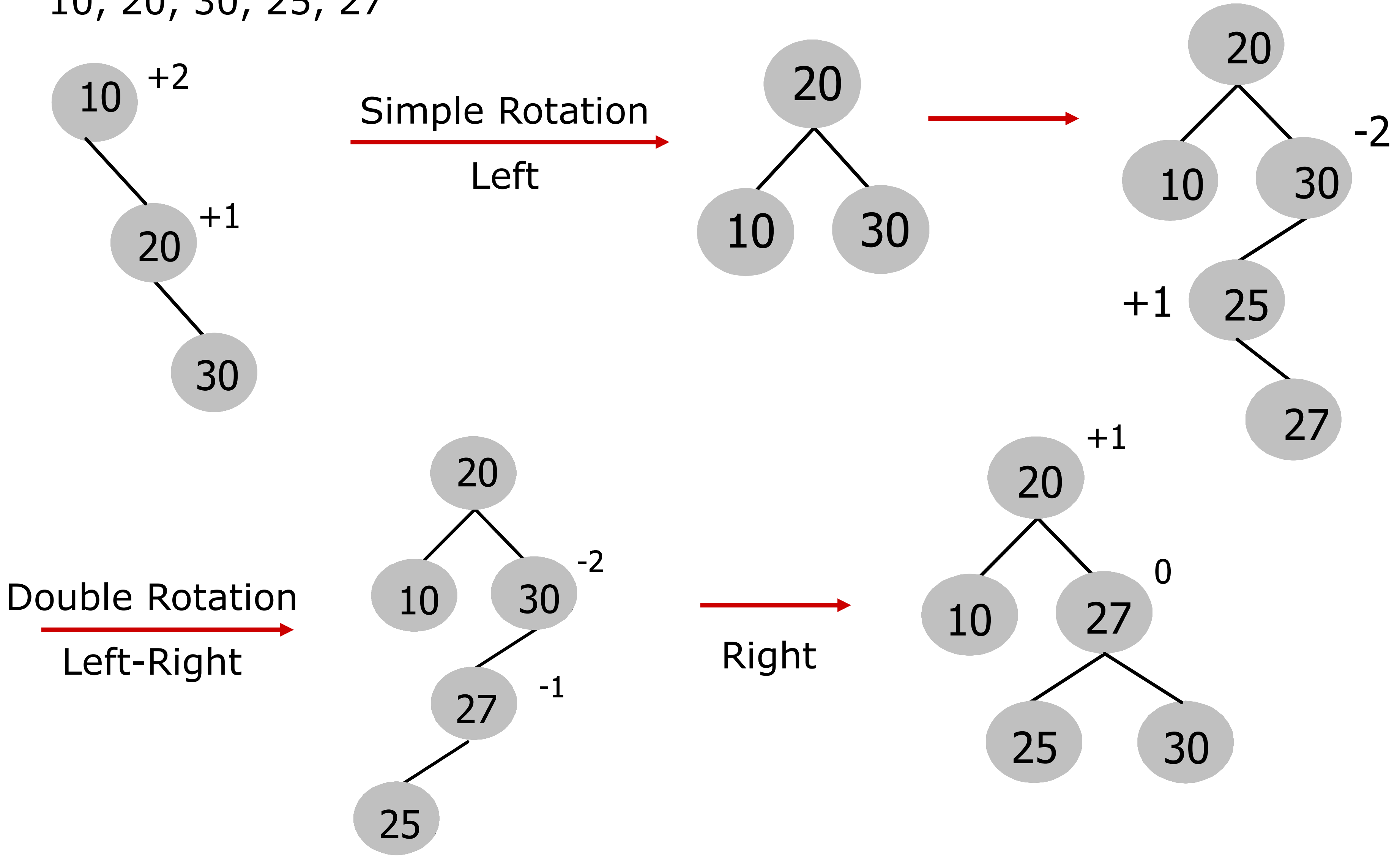
Double rotation (Right-Left)



```
Algorithm Node<E> twoRotations (Node<E> node) {
   if (balanceFactor(node) < 0)  {
      node.setLeft(leftRotation(node.getLeft()));
      node = rightRotation(node) }
   else {
      node.setRight(rightRotation(node.getRight()))
      node = leftRotation(node)  }
   return node
}
```

Insertion in an AVL tree

10, 20, 30, 25, 27

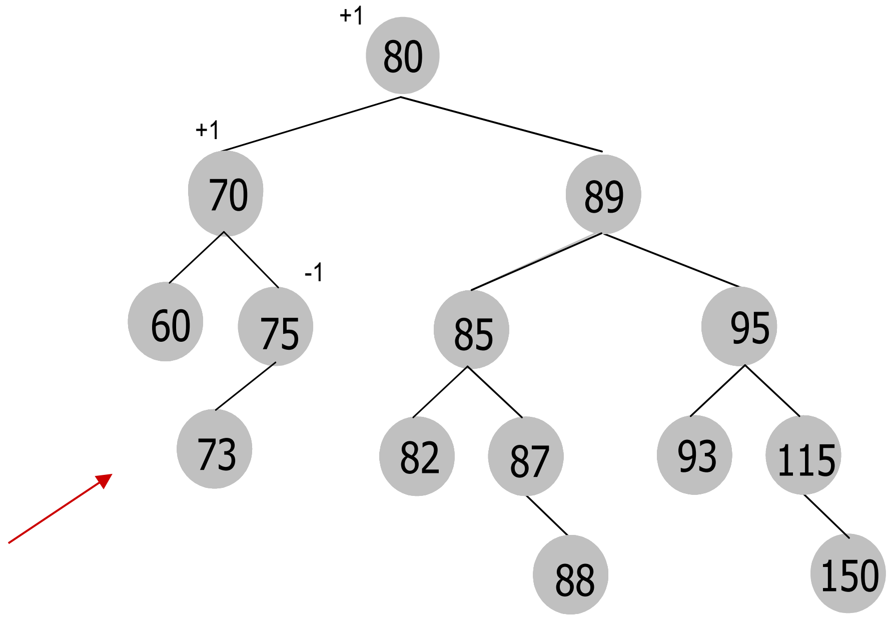


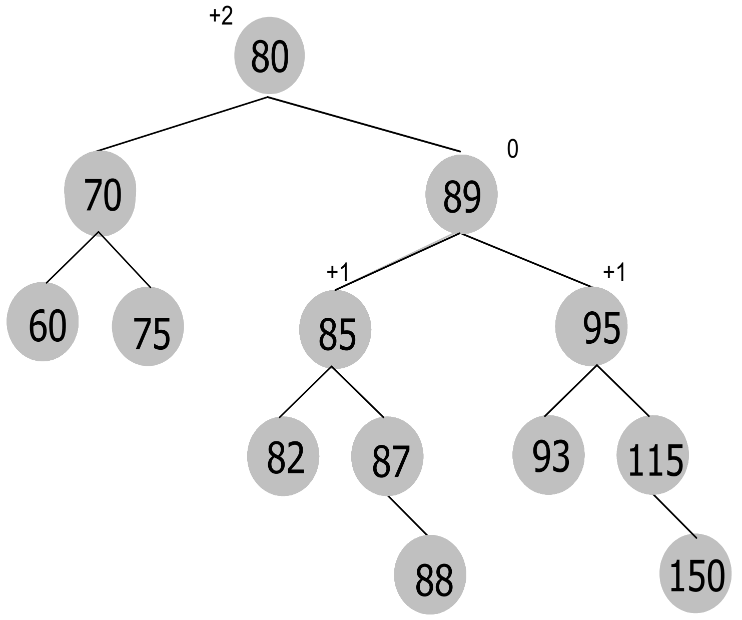
Insertion in an AVL tree

```
Algorithm Node<E> insert(Node<E> node, E elem) {
 if (node == null)
   return new Node(elem, null, null)

 if (node.getElement() == elem)
    node.setElement(elem)
 else
    if (node.getElement() > elem) {
       node.setLeft(insert(node.getLeft(),elem))
       node = balanceNode(node)
    }
    else {
       node.setRight(insert(node.getRight(),elem))
       node = balanceNode(node)
    }
 return node
}
```

Deletion in an AVL tree

After removal of a node, the balance factor of each node on the path between the removed node and the root are recalculated and the necessary rotations are made





Deletion

```
Algorithm Node<E> remove (E elem, Node<E> node) {
    if (node == null)
       return null

    if (node.getElement() == elem) {
       if (node.getLeft() == null && node.getRight()== null)
          return null
       if (node.getLeft() == null)
          return node.getRight()
       if (node.getRight() == null)
          return node.getLeft()
       E smallElem = smallestElement(node.getRight())
       node.setElement(smallElem)
       node.setRight(remove(smallElem, node.getRight()))
       node = balanceNode(node)
    }
    else if (node.getElement() > elem) {
         node.setLeft(remove(elem,node.getLeft()))
         node = balanceNode(node)   }
    else  {
         node.setRight(remove(elem,node.getRight()))
         node = balanceNode(node)   }
    return node
}
```

Binary search tree vs. AVL search tree

- On average 50% of insertions and deletions require rotations
- These lead to a loss of efficiency in the insertion and removal algorithms

Thus, the use of AVL or BST depends on the application:

- applications where the search is the dominant operation should use AVL trees, because they guarantee time complexity O(logn)
- applications where insertions or deletions are the most frequent operations should used binary search tree

Kd-trees

kd-trees

- Invented in 1970s by Jon Bentley
- Kd-trees[1] mean “2d-trees, 3d-trees, 4d-trees, etc”
  – k is the number of dimensions

- A Kd-tree is a binary tree that stores a finite set of points from a k-dimensional space

- Kd-trees are useful for searches involving a multidimensional search key, like nearest neighbour searches or range searches

1 - https://pt.wikipedia.org/wiki/Árvore_k-d

Kd-trees

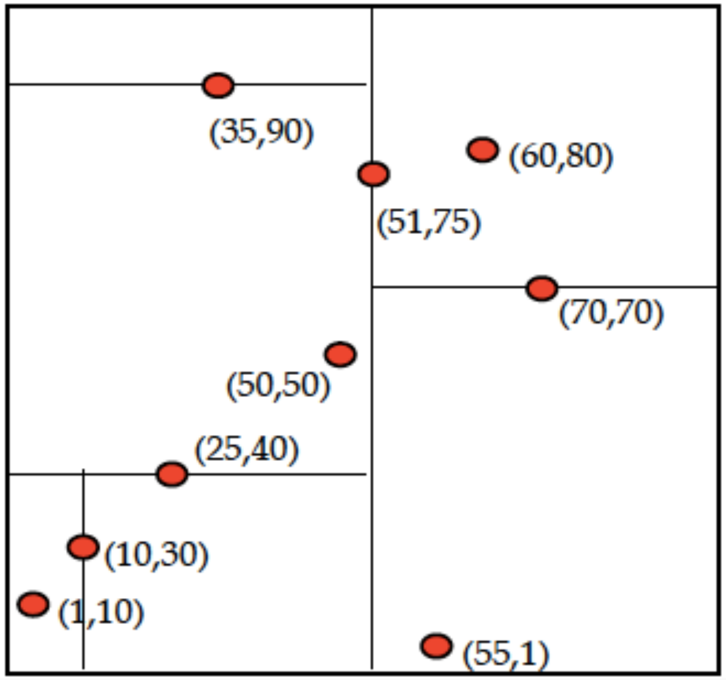
Any internal node in a Kd-tree divides the space into two halves irrespective of the number of dimensions

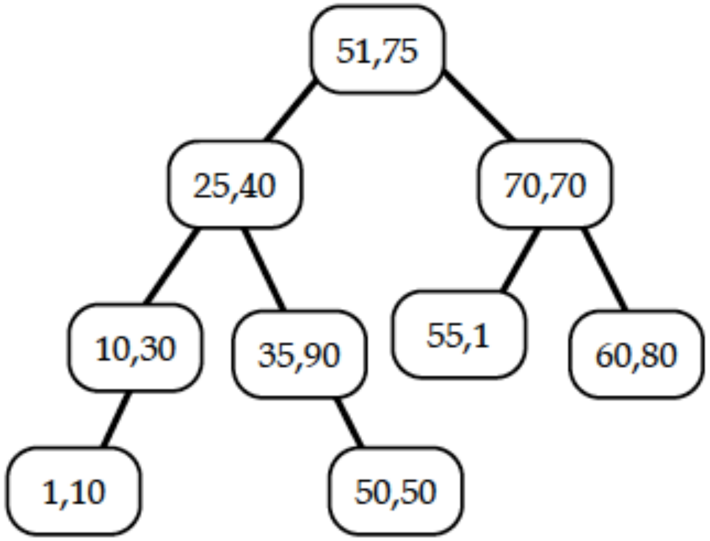
Every internal node represents a hyperplane that cuts the space in two parts:

- for 2-dimensional space, that is a line
- for 3 dimensional space, that is a plane
- ….

Kd-tree Definition

- Kd-tree is a binary search tree representing a rectangular area in D-dimensional space
- The area is divided (and recursively subdivided) into rectangular cells

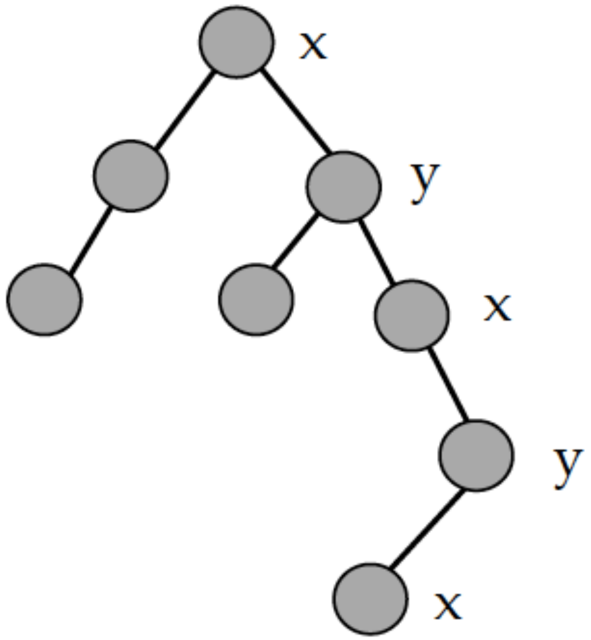




2d-tree example

Idea: Each level of the tree compares against one dimension

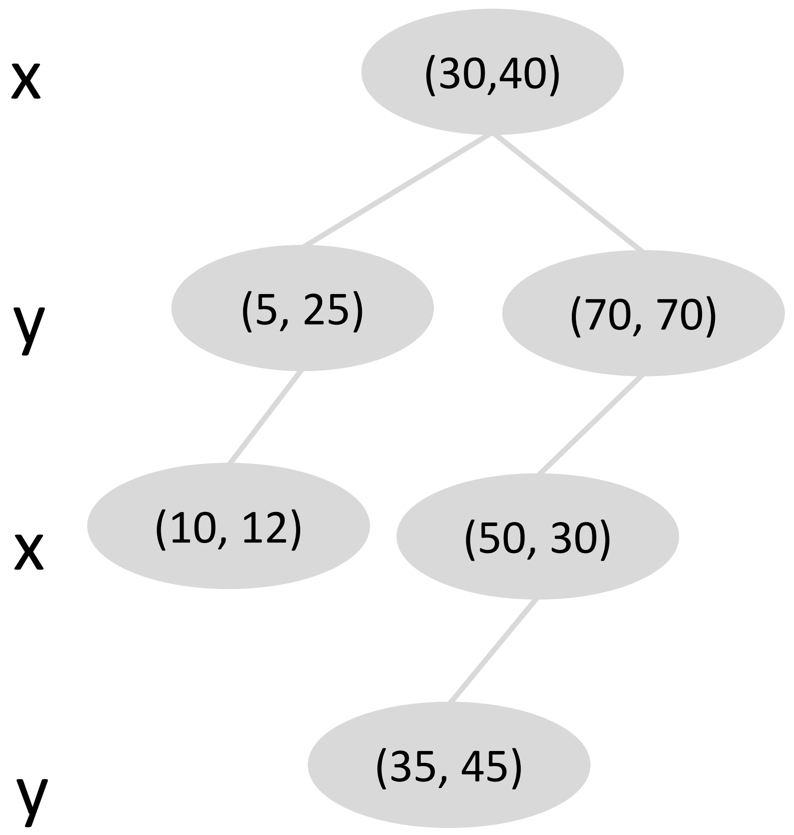
- Each node contains a point P = (x, y)
- Each internal node stores the splitting node along x (or y).
- Partition along x and y axis in an alternating fashion:
  - level 0 is discriminated by the x-coordinate,
  - level 1 by the y-coordinate,
  - level 2 again by the x-coordinate, etc.

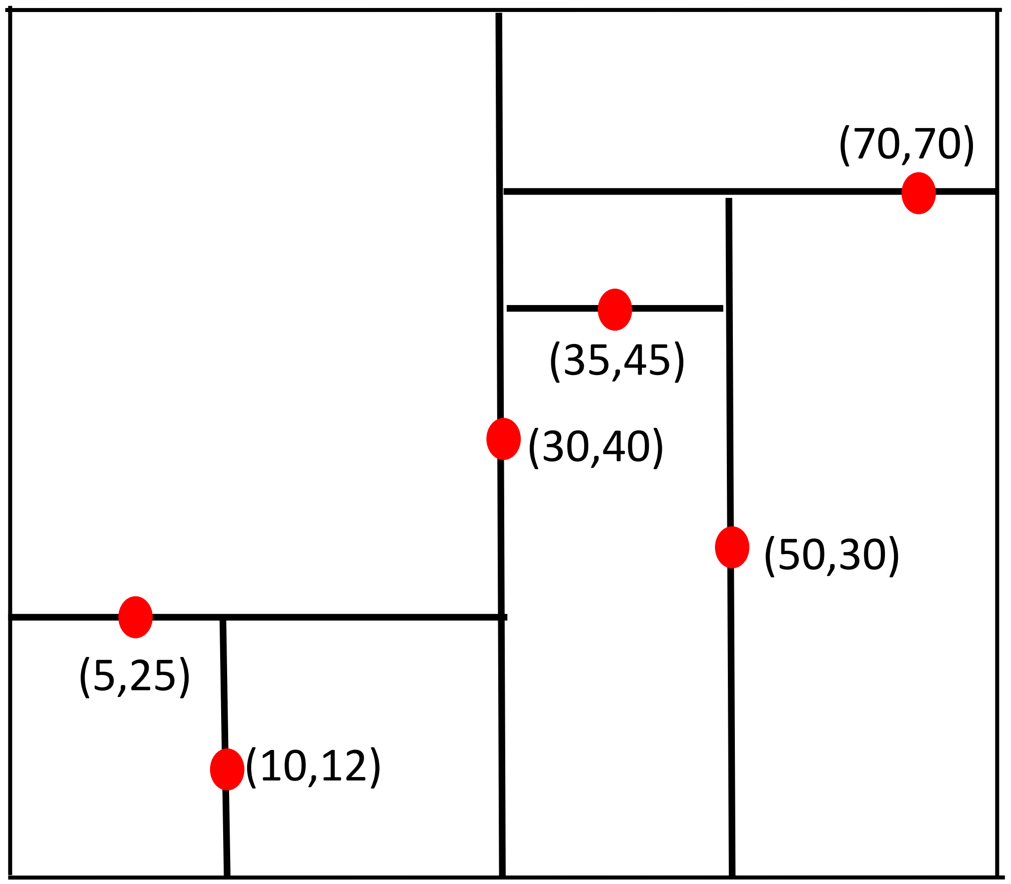


To find (x', y') only compare coordinate from the cutting dimension

kd-tree Insertion

Insert: (30,40), (5,25), (10,12), (70,70), (50,30), (35,45)





Kd-tree ADT

```
public class Node<T> {
    protected Point2D.Double coords;
    protected T info;
    protected Node<T> left;
    protected Node<T> right;
…
}
```

Kd-tree ADT

```
public class KDTree<T> {

private final Comparator<Node<T>> cmpX = new Comparator<Node<T>>(){

        @Override
        public int compare(Node<T> p1, Node<T> p2) {
            return Double.compare(p1.getX(), p2.getX());
        }
    };

private final Comparator<Node<T>> cmpY = new Comparator<Node<T>>(){

        @Override
        public int compare(Node<T> p1, Node<T> p2) {
            return Double.compare(p1.getY(), p2.getY());
        }
    };

private Node<T> root;
```

Kd-tree Insertion

Operation Insertion works analogously as in BST

Find the place for the new node under some of the leaves and insert node there

Do not accept coordinates which are identical to some other already stored in the tree

Kd-tree: Insertion

```
Algorithm Node2D<T> insert(Node2D<T> currentNode, Node2D<T> node,
                                                     boolean divX){
  if (node.coords.equals(currentNode.coords))
     return;
 int cmpResult = (divX ? cmpX : cmpY).compare(node, currentNode);

 if (cmpResult == -1)
    if (currentNode.left == null)
       currentNode.left = node;
    else
       insert(currentNode.left, node, !divX);
 else
    if (currentNode.right == null)
       currentNode.right = node;
    else
       insert(currentNode.right, node, !divX);
}
```

Kd-tree Insertion

Adding points in this manner can cause the tree to become unbalanced, leading to decreased tree performance

Because *k*d-trees are sorted in multiple dimensions, the tree rotation technique cannot be used to balance them — this may break the invariant

If the entire set of points to be inserted are known:

- Points are inserted by selecting the **median** of the points being put into the tree, with respect to their coordinates in the axis being used to create the splitting plane

- This leads to a **balanced kd-tree**, in which each leaf node is about the same distance from the root

Kd-tree Search

Kd-tree supports several search operations:

- *find an element that exists in the tree*

  is analogous to BST

- *nearest-neighbour search*

  returns the nearest-neighbor to a query point

- *K-nearest neighbour search*

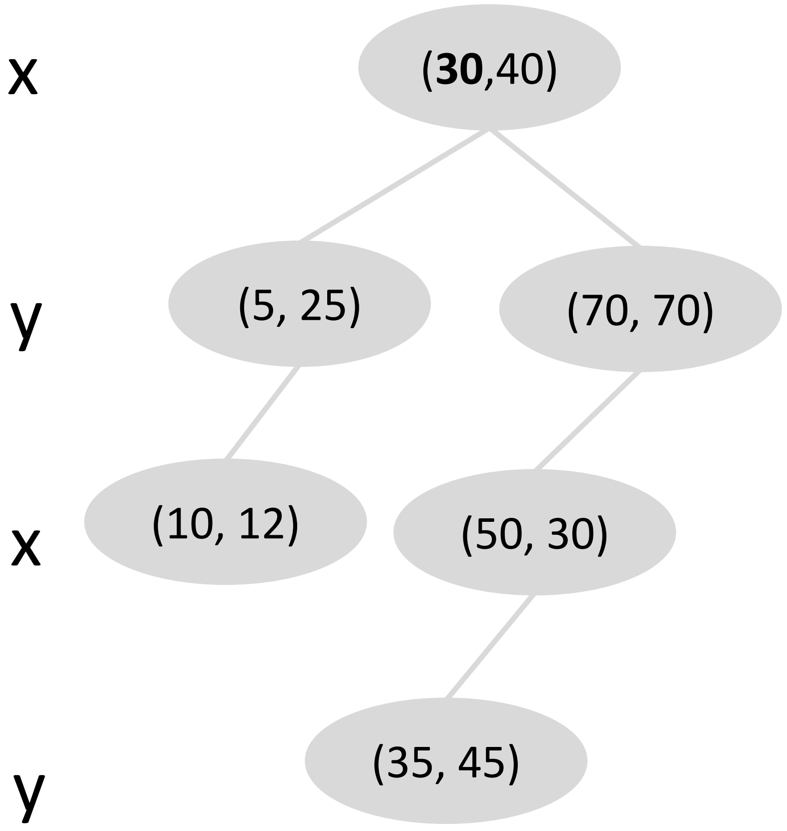
  returns the *k* nearest neighbours to a given query point

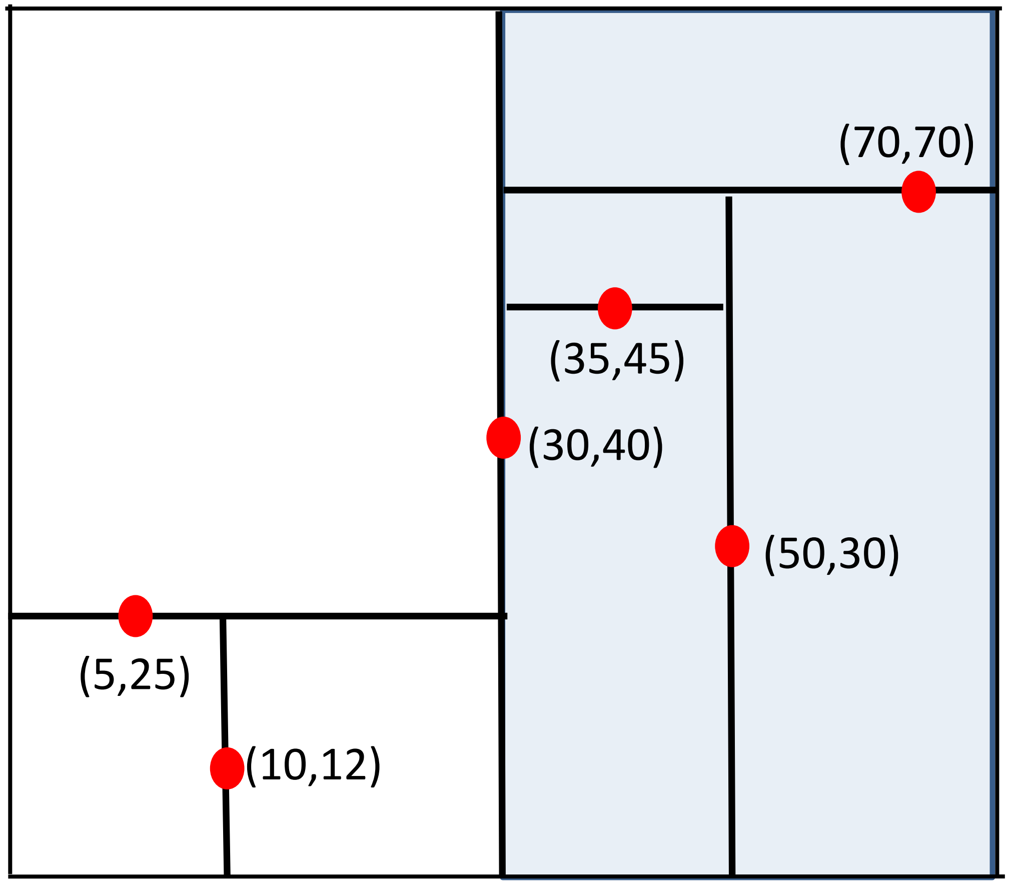
- *Range search*

  returns the neighbours inside a region

kd-tree: Exact Find

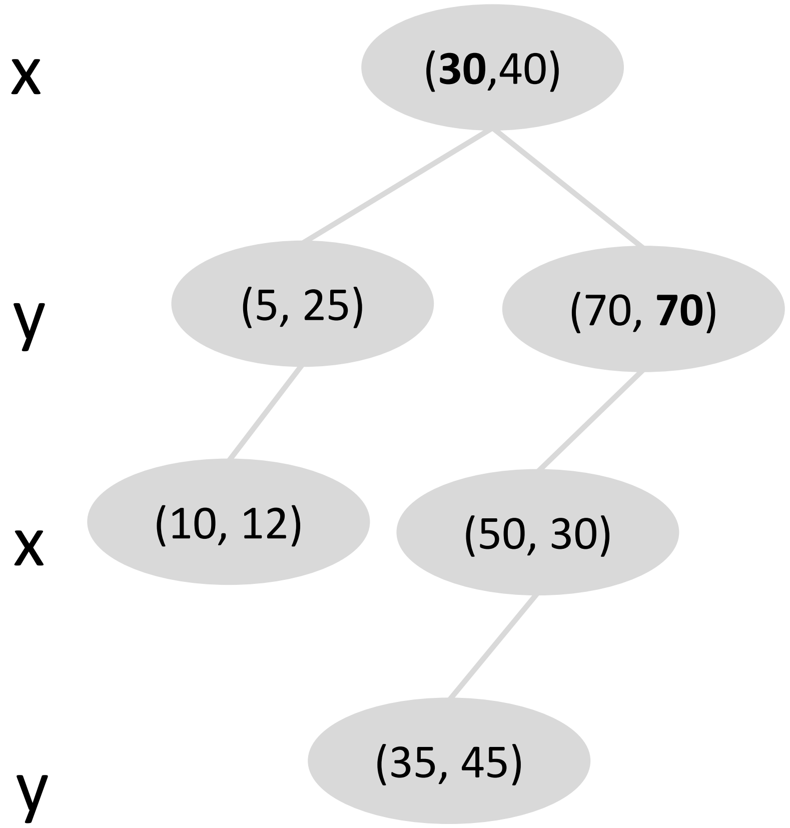
Find: (35,45)

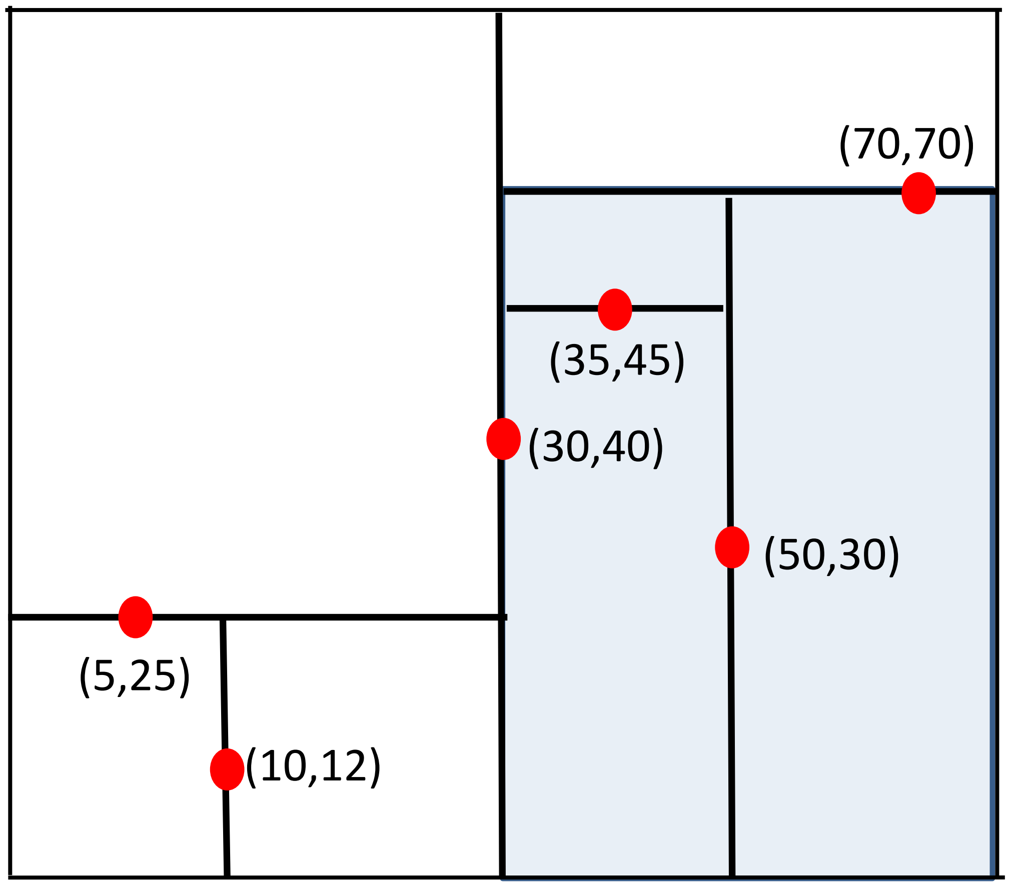




kd-tree: Exact Find

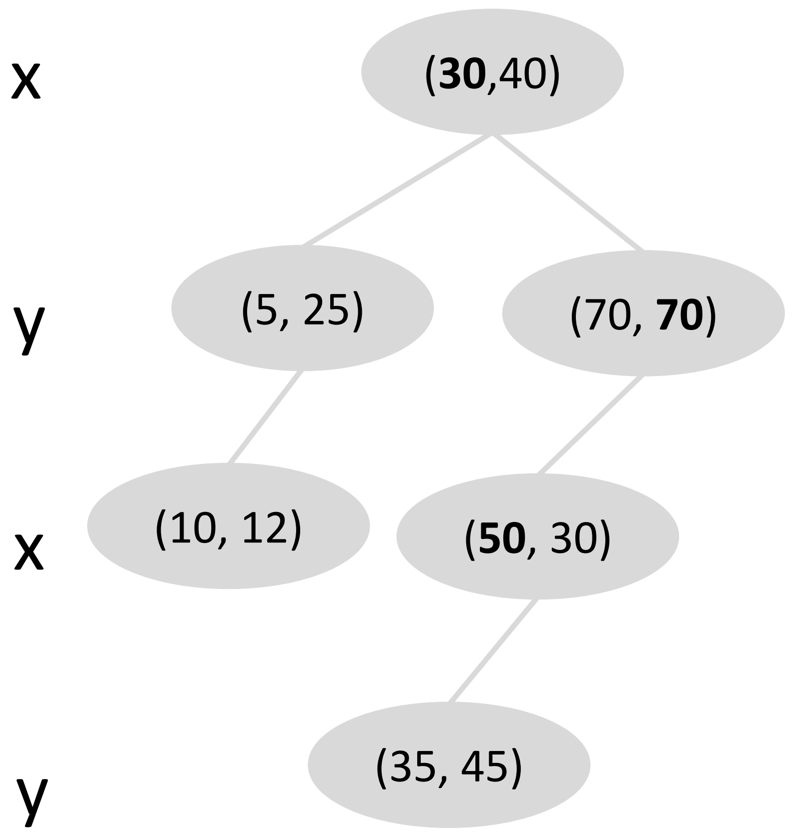
Find: (35,45)

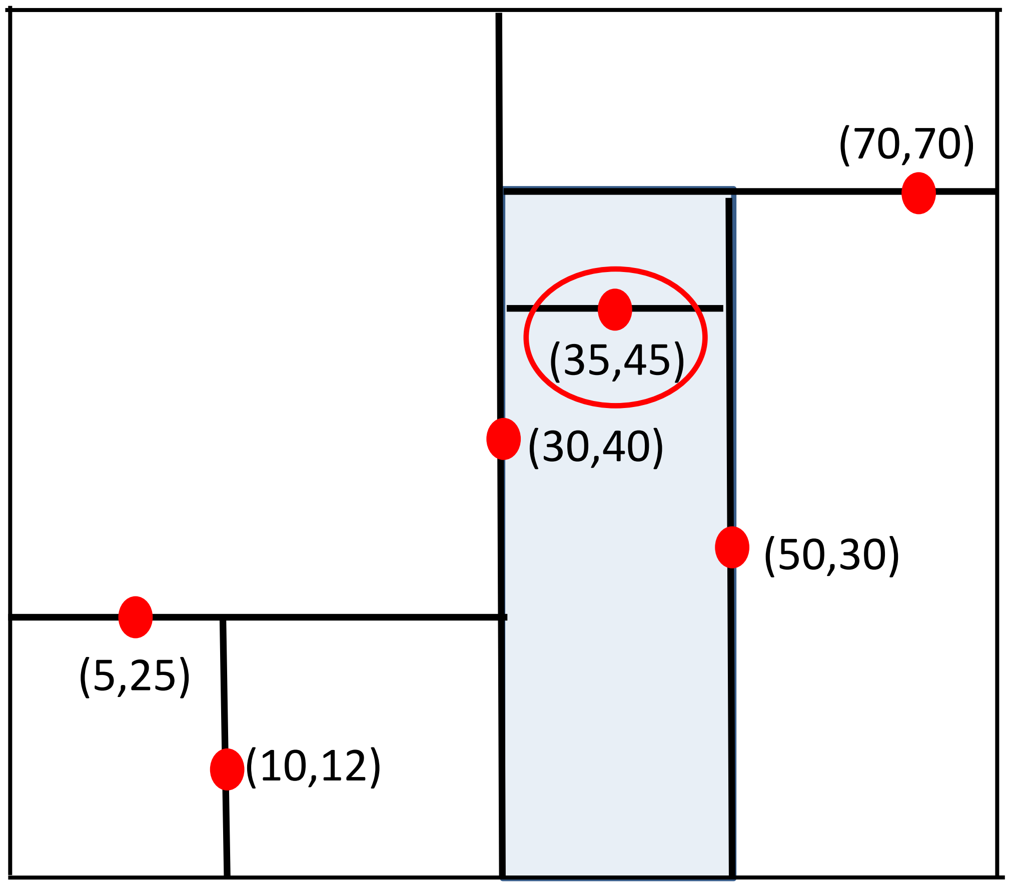




kd-tree Exact Find

Find: (35,45)

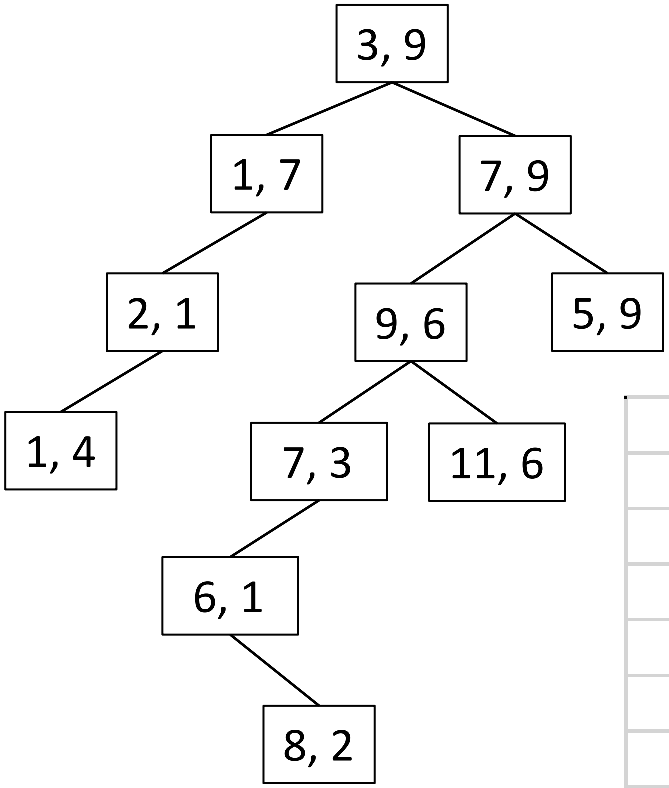


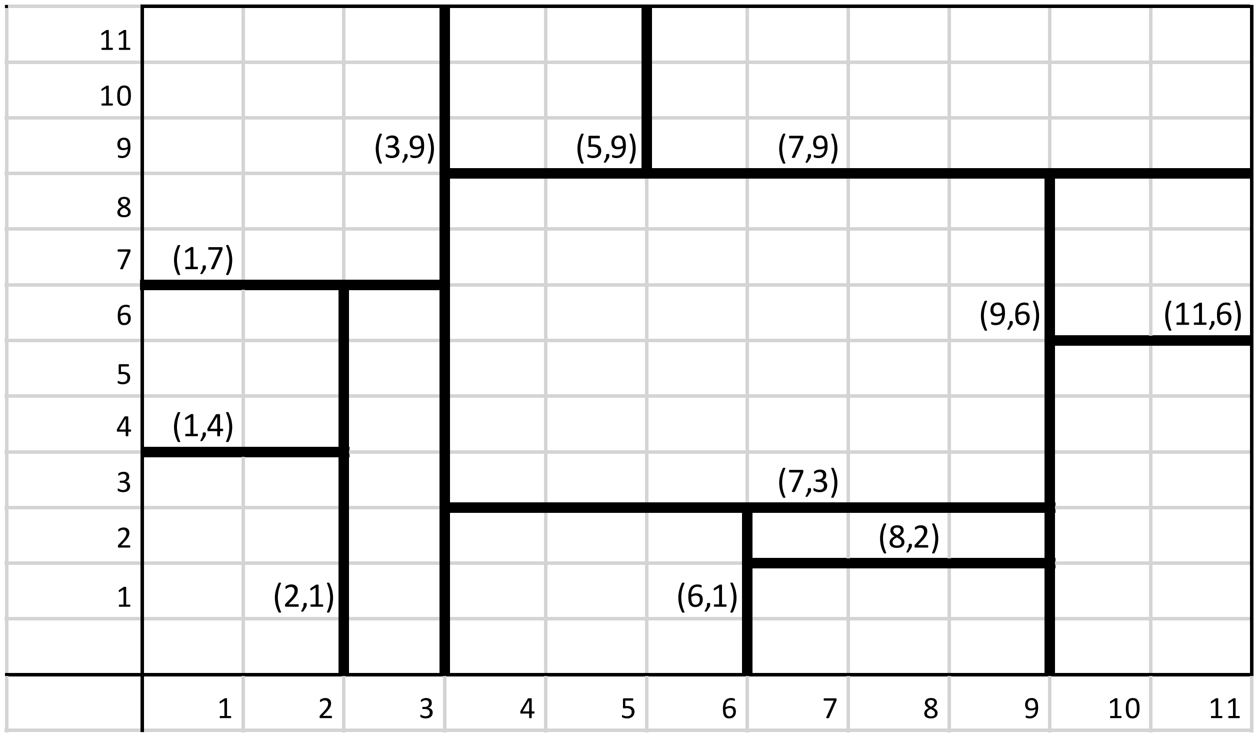


kd-tree: Nearest-neighbour search

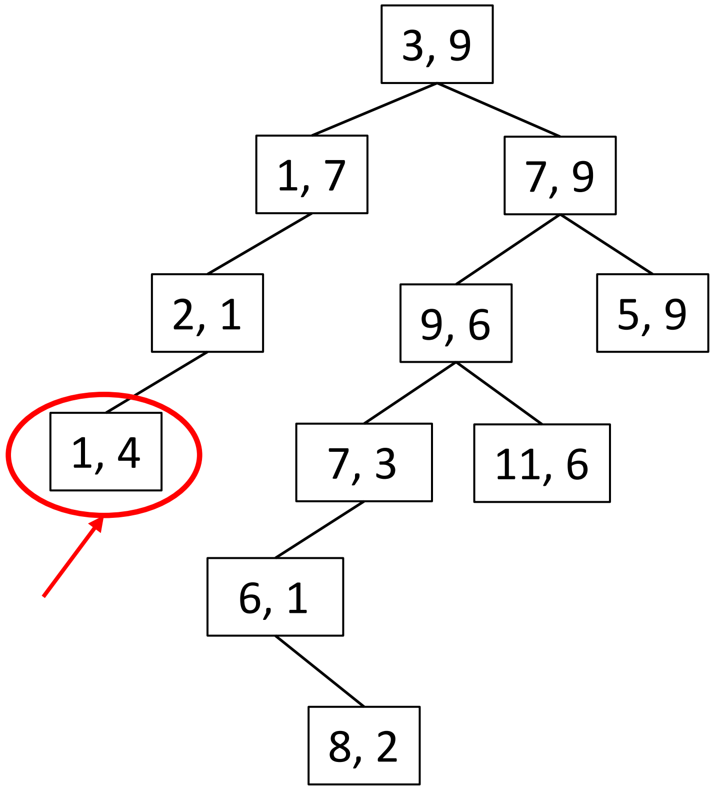
- Search starts in the root and runs recursively in both left and right subtrees of the current node
- During the search needs:
  - to calculate the square of the distance between the current node and the point
  - to calculate the square distance between the point and the plan associated with the subtree
  - to store the closest node

kd-tree: Nearest-neighbour search





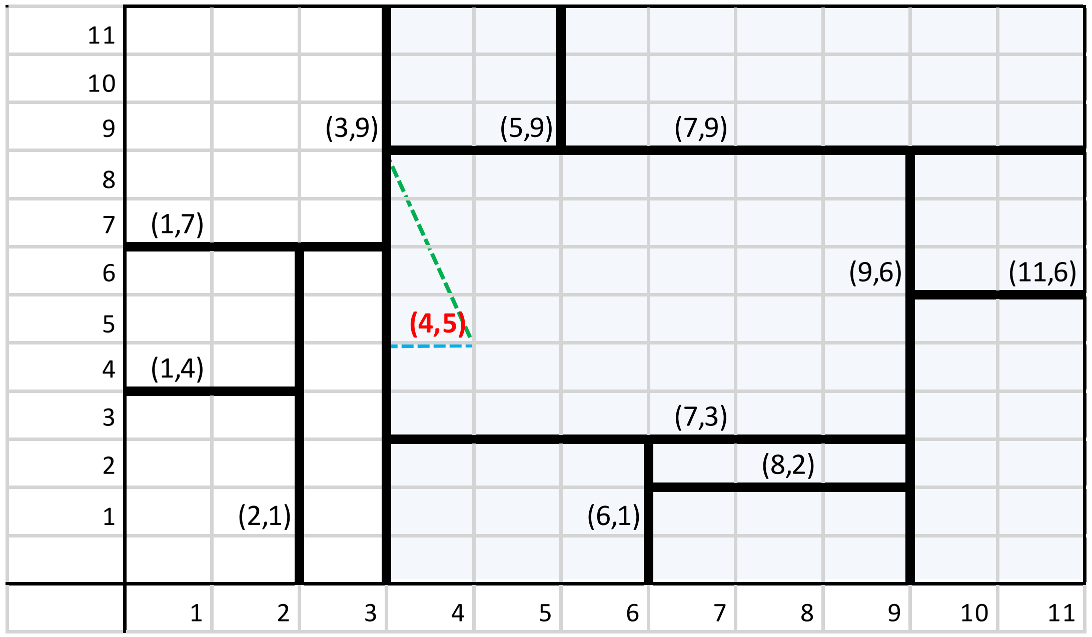
kd-tree: Nearest-neighbour search

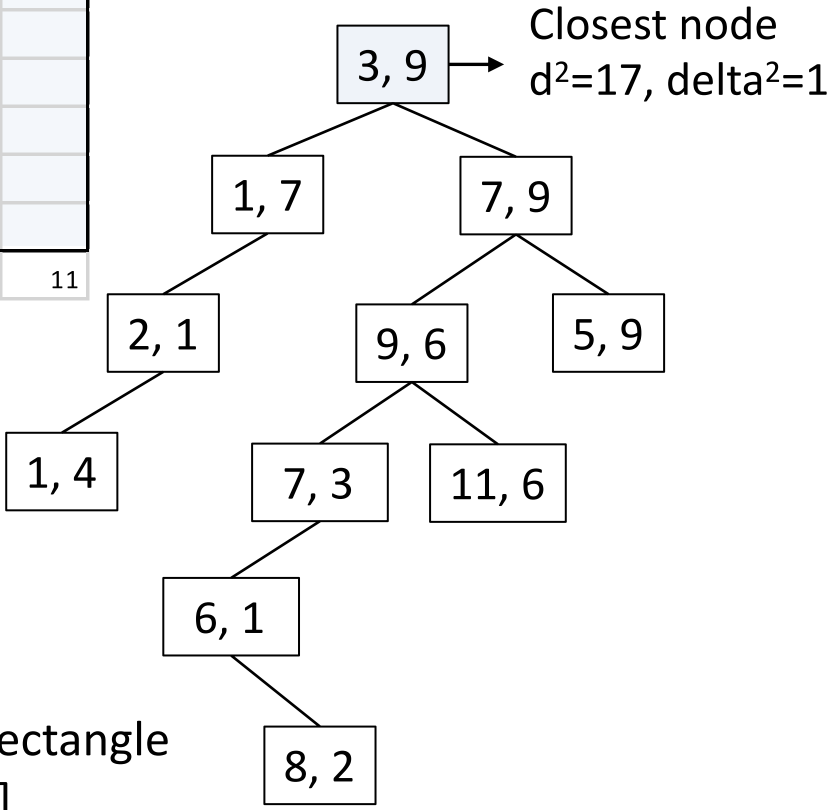


Nearest-neighbour of Point (4,5)

		sq. dist.	delta	sq. delta
3	9	17	1	1
1	7	13	-2	4
7	9	25	-4	16
2	1	20	2	4
9	6	26	-5	25
5	9	17	-1	1
1	4	10	1	1
7	3	13	2	4
11	6	50	-1	1
6	1	20	-2	4
8	2	25	3	9

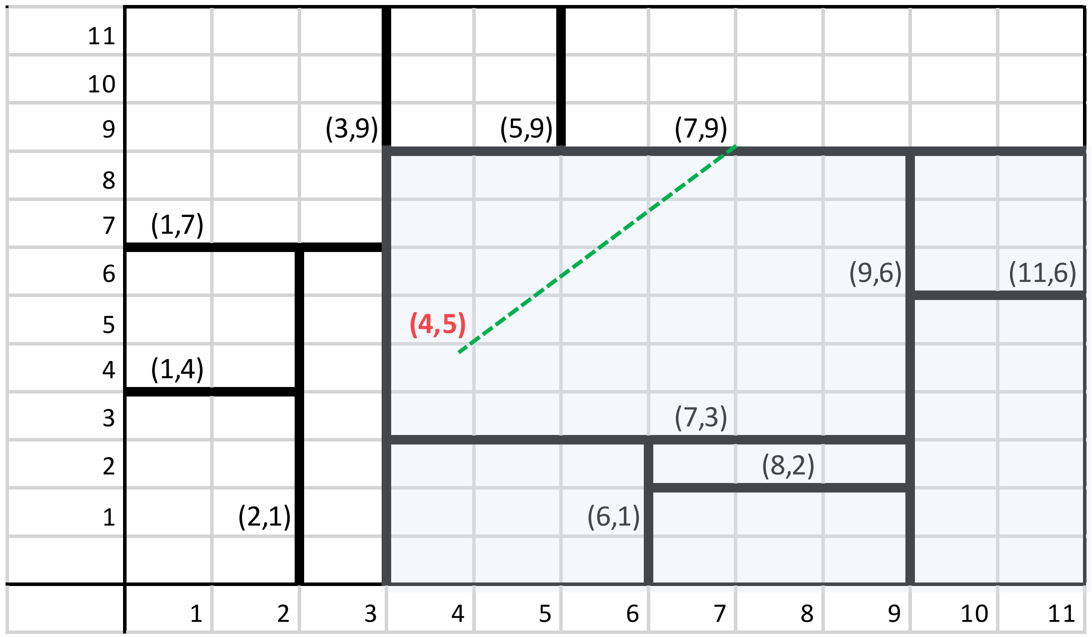
kd-tree: Nearest-neighbour search

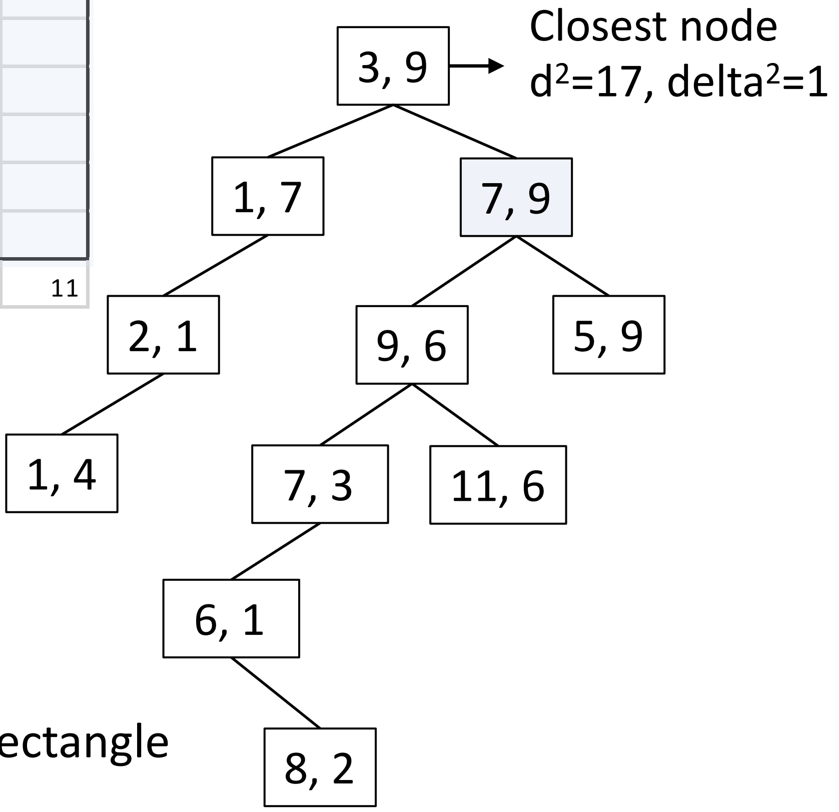




The query point Q = [4, 5] lies inside the (hyper) rectangle associated with the right subtree of the root [3, 9]

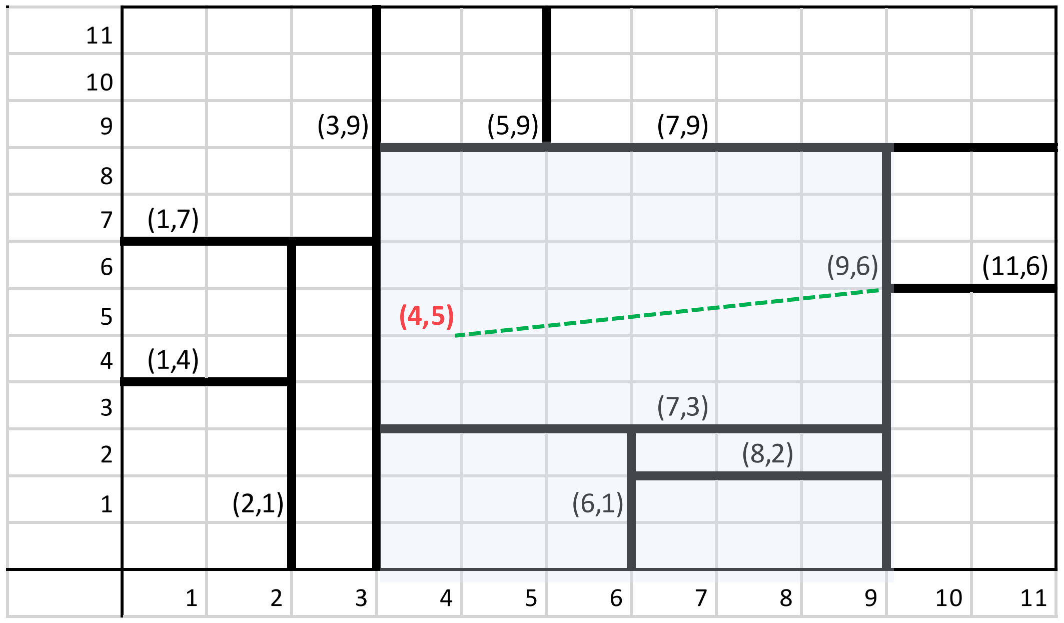
kd-tree: Nearest-neighbour search

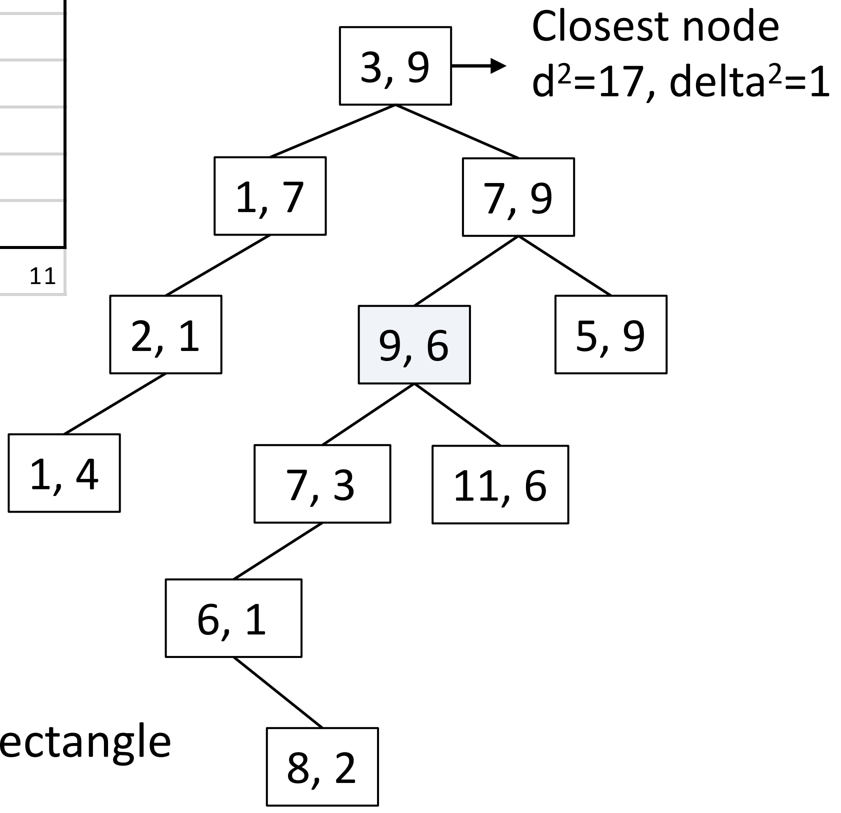




The query point Q = [4, 5] lies inside the (hyper) rectangle associated with the leftt subtree of the root [7, 9]

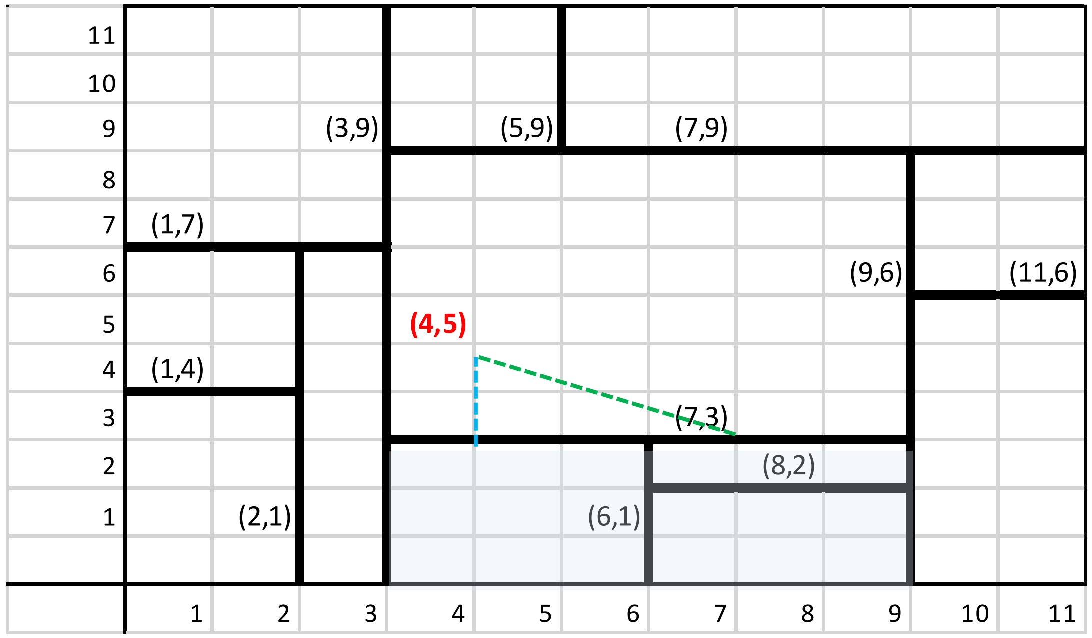
kd-tree: Nearest-neighbour search

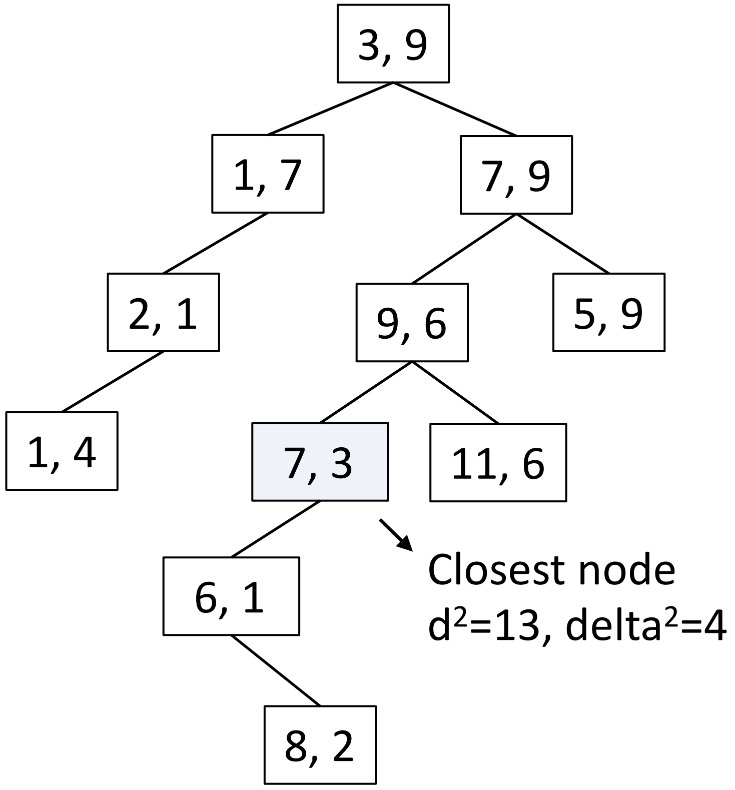




The query point Q = [4, 5] lies inside the (hyper) rectangle associated with the left subtree of the root [9, 6]

kd-tree: Nearest-neighbour search





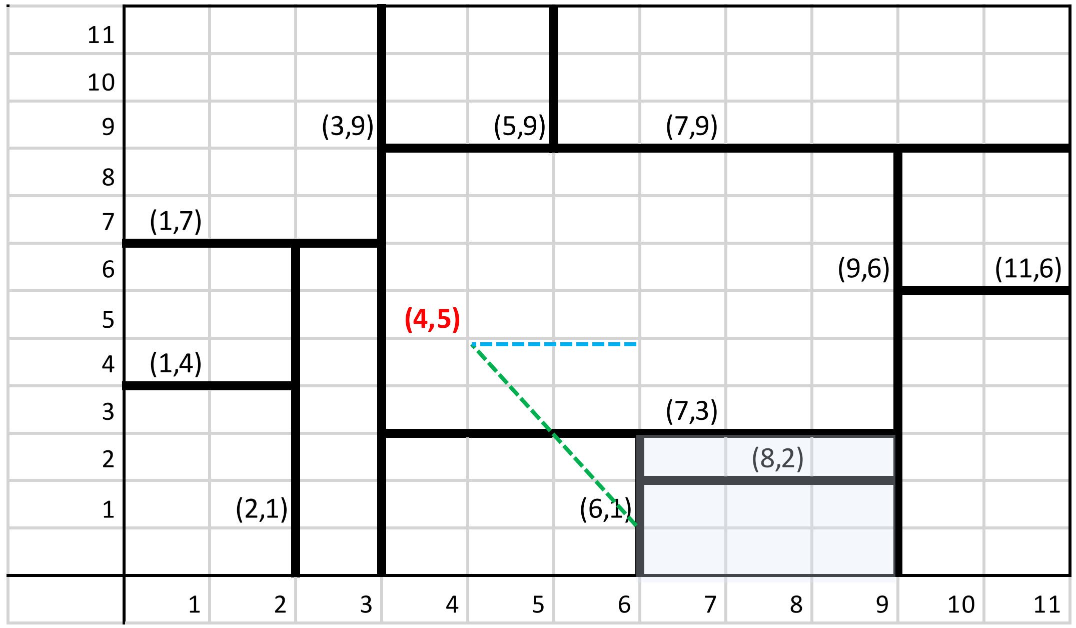
The Point [7, 3] becomes new closest node.

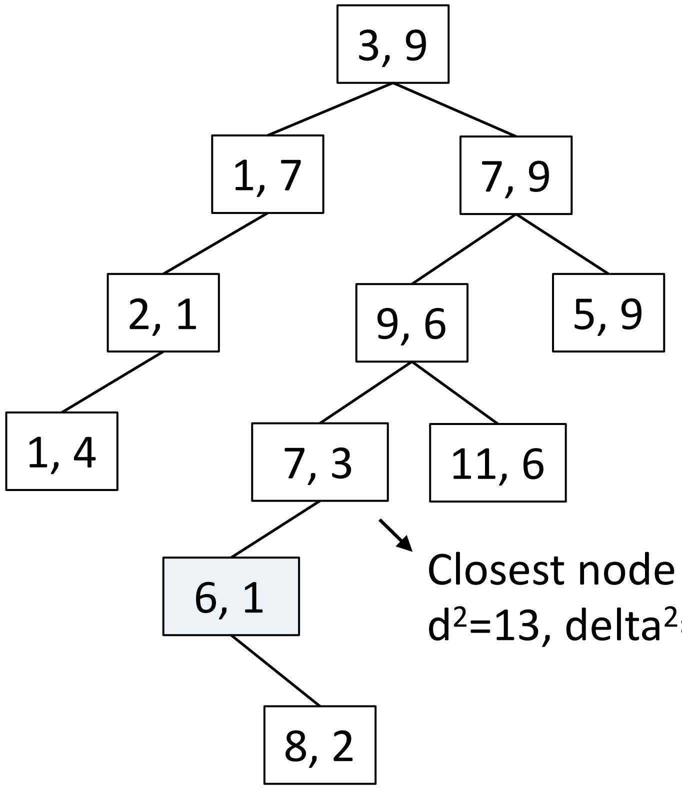
No right subtree - backtraking

Prunning?

delta^2 = 4 < d^2 = 13 - the search continues in the left subtree of [7, 3].

kd-tree: Nearest-neighbour search

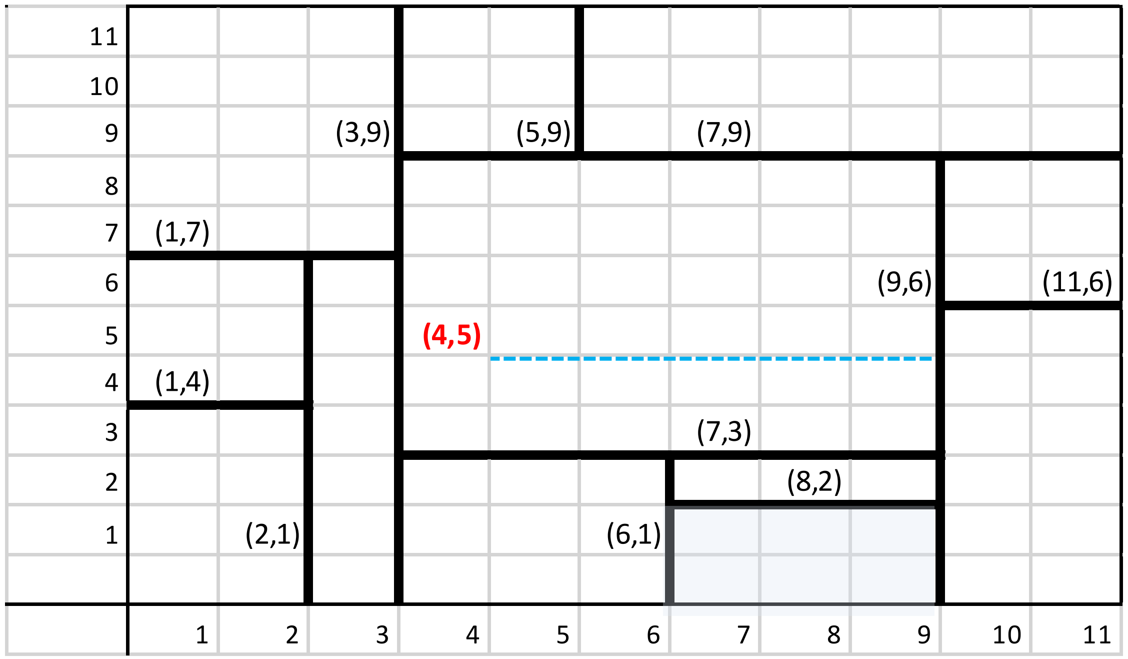


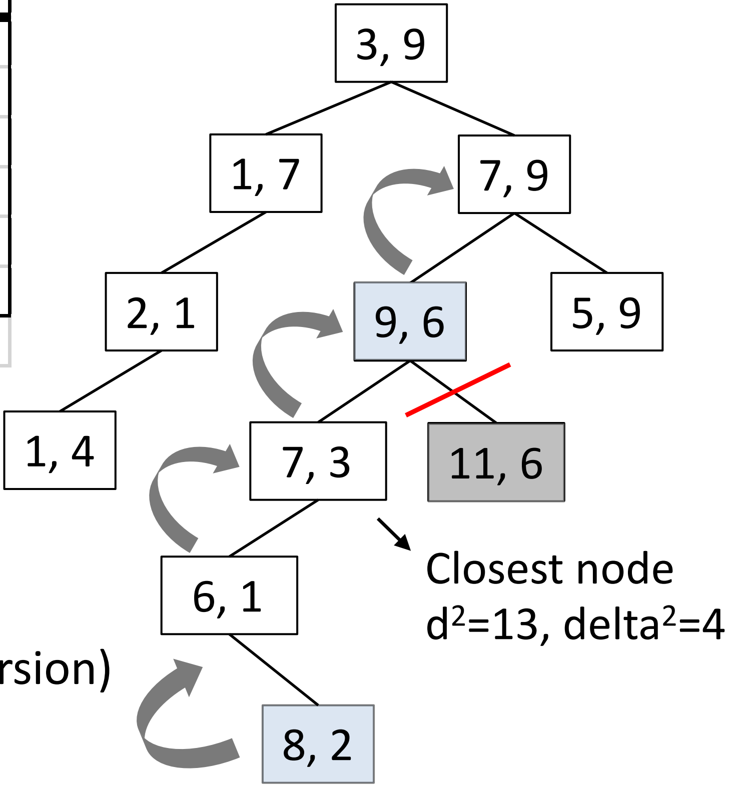


Closest node
$d^2=13$, $delta^2=4$

No left subtree - backtraking
Prunning?
$delta^2 = 4 < d^2 = 13$ - the search continues in the right subtree of [6, 1].

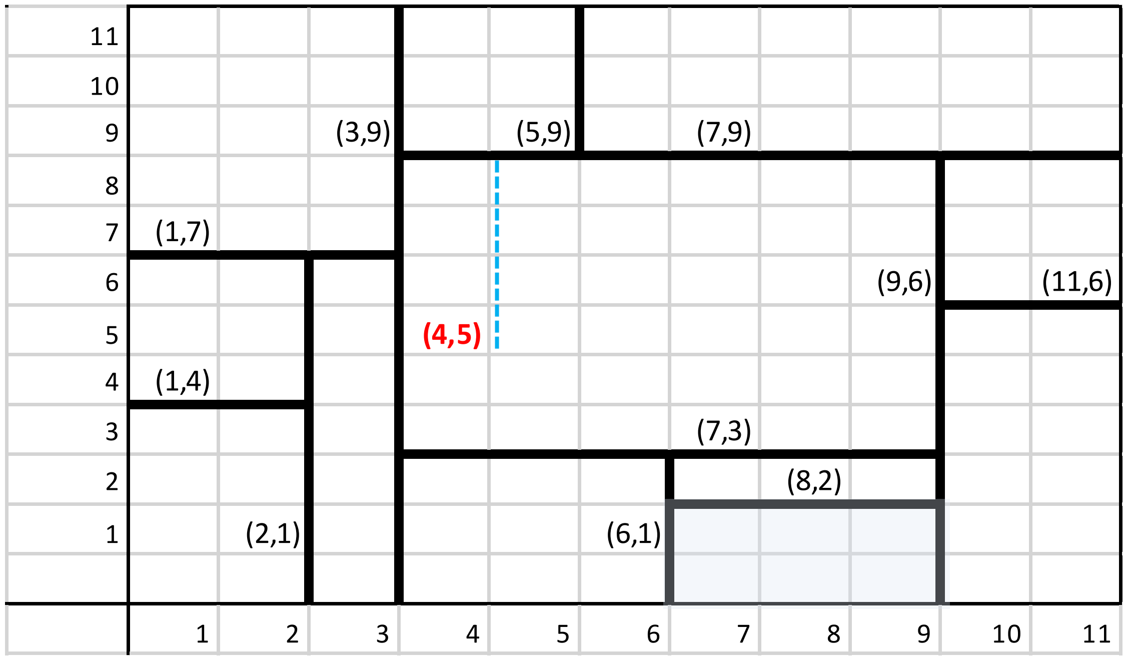
kd-tree: Nearest-neighbour search

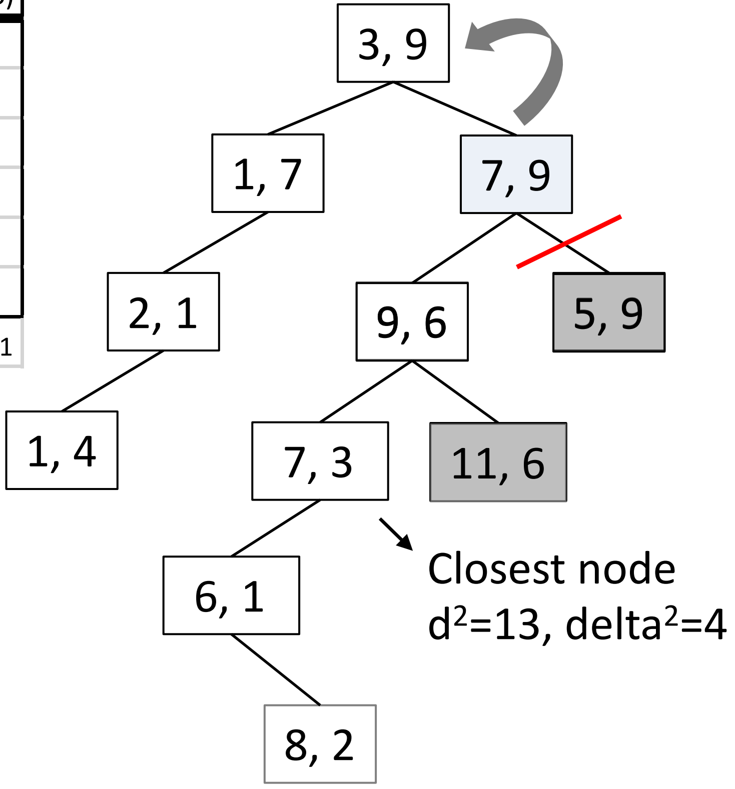




The search has reached a leaf and returns (due to recursion) to the last unexplored branch - right subtree [9,6]
$delta^2 = 25 > d^2=13$. The whole branch is pruned.
The search returns back to the previous unexplored branch.

kd-tree: Nearest-neighbour search



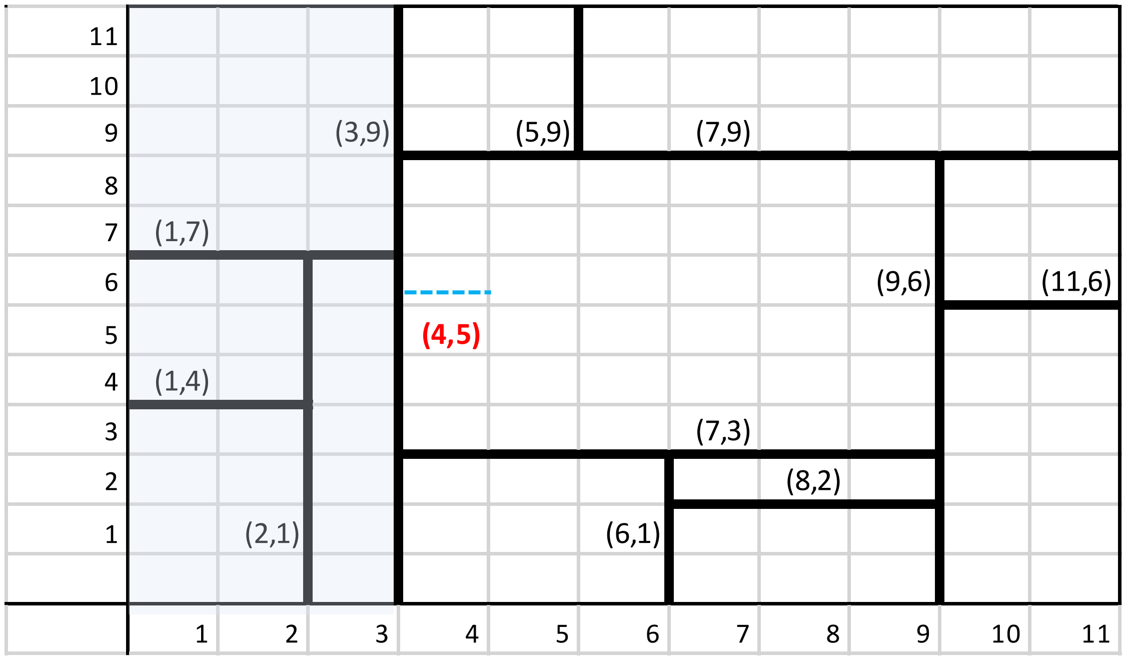


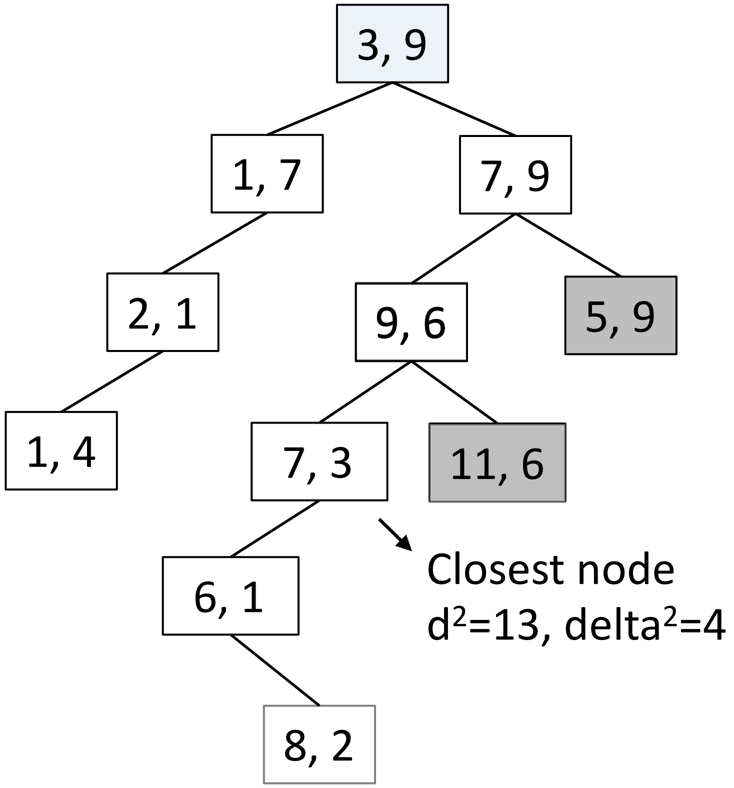
Prunning?

$delta^2 = 16 > d^2 = 13$ - The whole branch is pruned.

The search returns back to the previous unexplored branch.

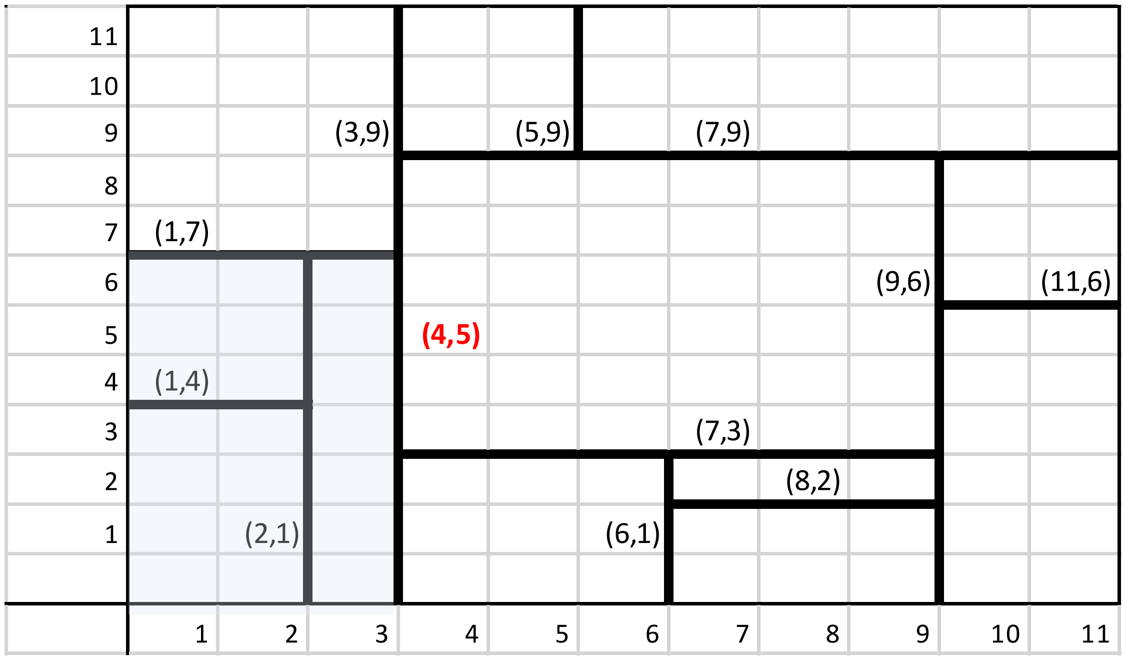
kd-tree: Nearest-neighbour search

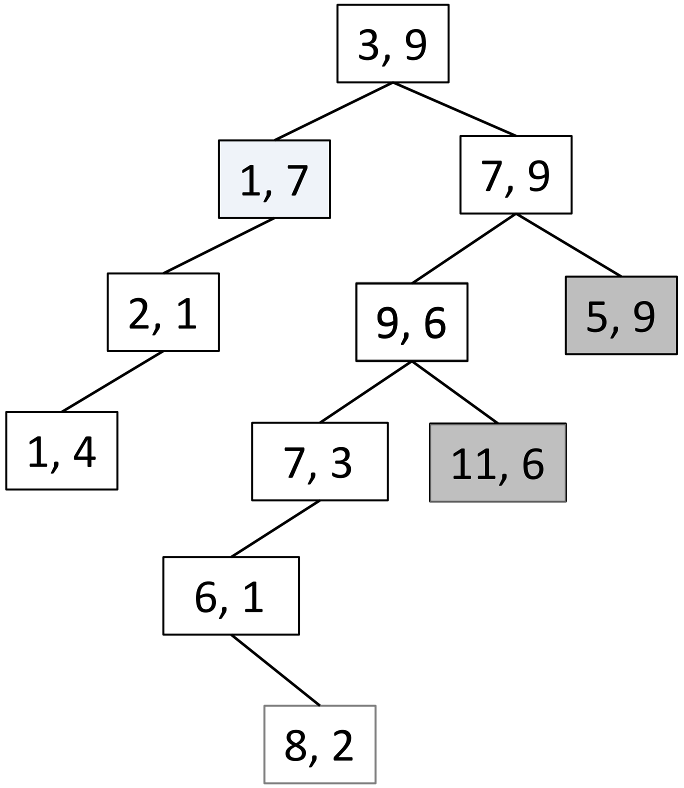




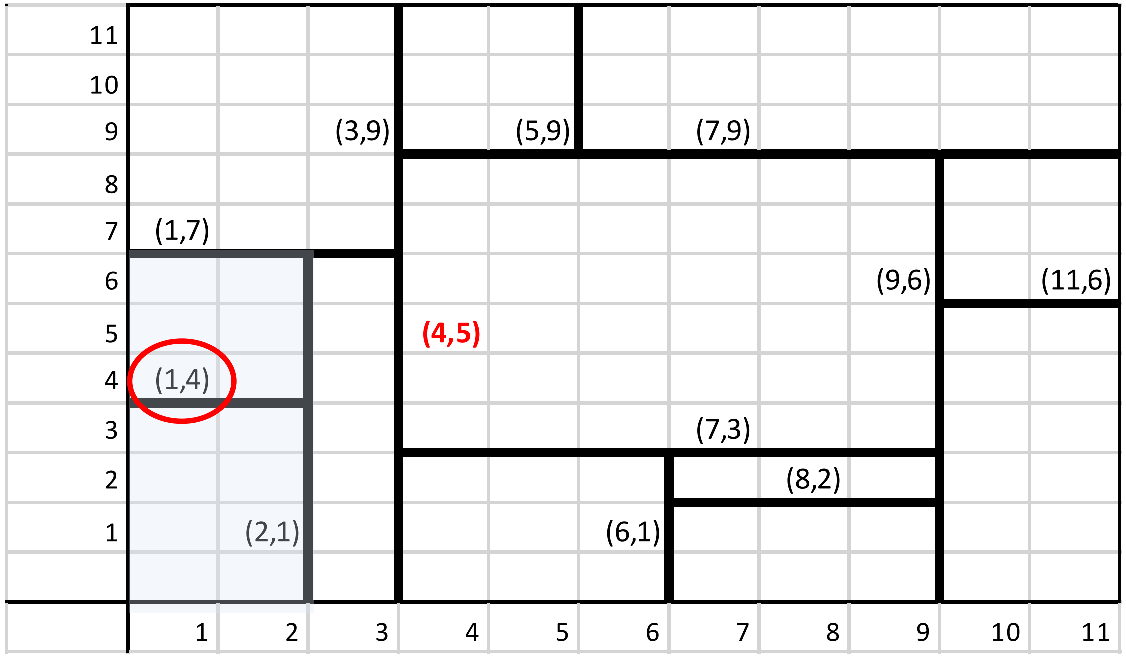
The search returns to node [3,9].
Pruning?
delta^2 = 1 < d^2 = 13 - the search continues in the left subtree of [3, 9].

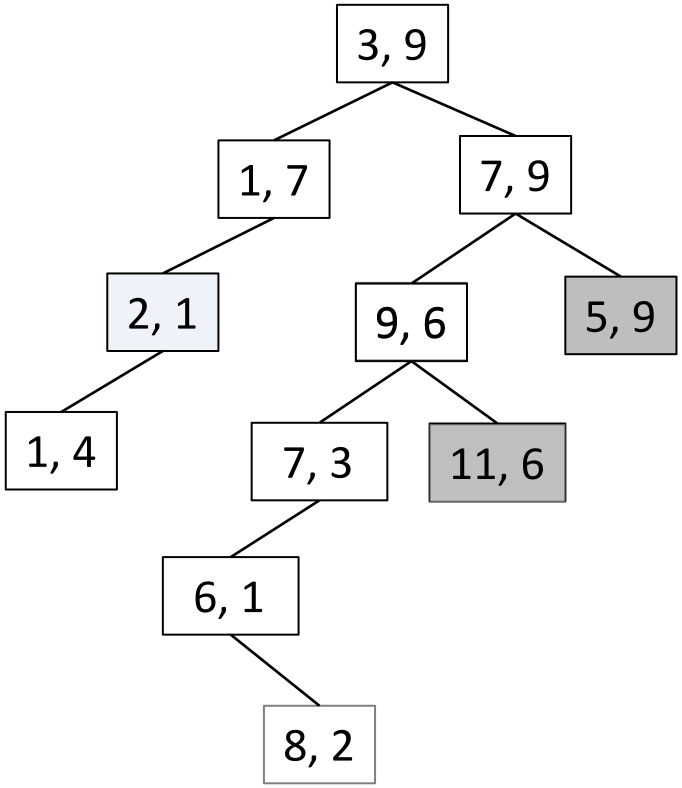
kd-tree: Nearest-neighbour search





kd-tree: Nearest-neighbour search





Closest node
d^2=10, $delta^2$=1

Kd-tree: Nearest-neighbour search

```
Algorithm T findNearestNeighbour(Node<T> node, double x, double y,
                                         Node<T> closestNode, boolean divX){
  if (node == null)
    return null;

  double d = Point2D.distanceSq(node.coords.x, node.coords.y, x, y);
  double closestDist = Point2D.distanceSq(closestNode.coords.x,
                                          closestNode.coords.y, x, y);
  if (closestDist > d){
    closestNode.setObject(node);

  double delta = divX ? x - node.coords.x : y - node.coords.y;
  double delta2 = delta * delta;

  Node<T> node1 = delta < 0 ? node.left : node.right;
  Node<T> node2 = delta < 0 ? node.right : node.left;

  findNearestNeighbour(node1, x, y, closestNode, !divX);

  if (delta2 < closestDist){
        findNearestNeighbour(node2, x, y, closestNode,!divX);

  return closestNode.info;
}
```

Nearest-neighbour search Complexity

Best Case: O(log n)

Worst Case: O(n)

When data points and query point are unfavourably arranged.

This happens only when:

- the dimension D is relatively high, 4,5…, or
- the arrangement of points in low dimension D is very special (artificially constructed)