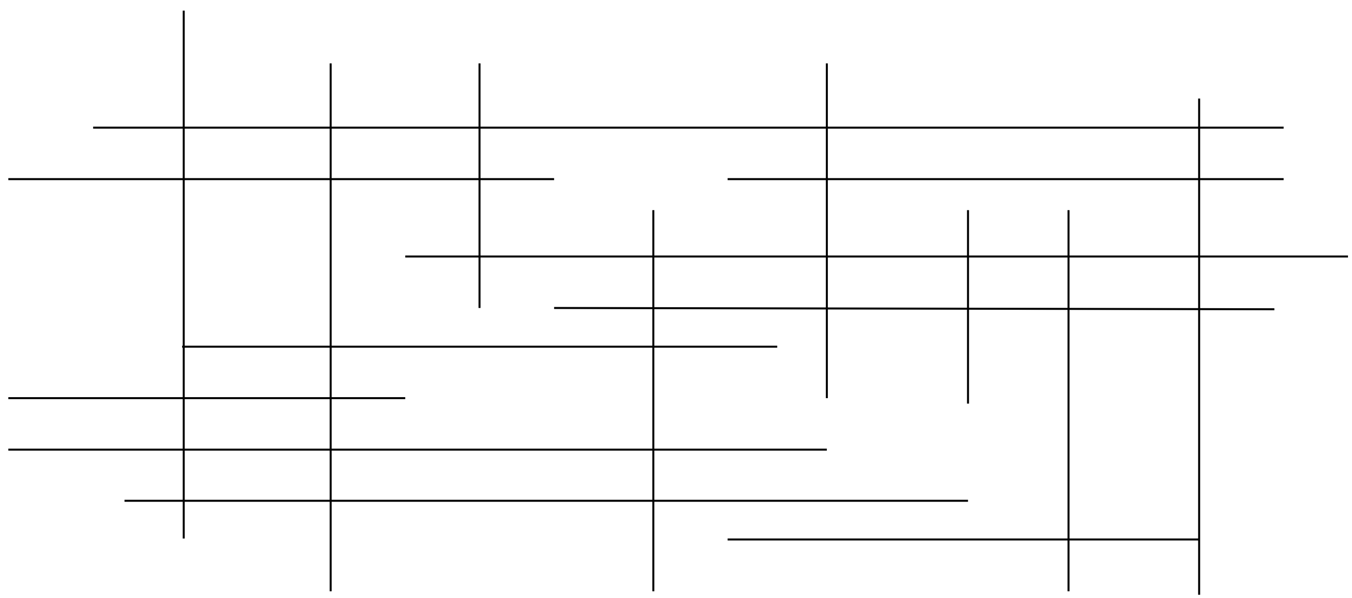
Problem 5: Finding line segment intersections

Problem: given a set of h horizontal and v vertical line segments in the plane, find all intersections.

- **A common type of problem – finding intersections between geometric objects**



h=10, v=8, number of intersections=34

Finding line intersections

- **Brute force algorithm examines all pairs**
  - **each pair can be checked in O(1) time**
  - **but this is never better than O(hv)**
- **No algorithm can be better than O(hv) in the worst case**
  - **there could be hv intersections**
- **Can we get something better when the number of intersections p is small?**
- **Our solution illustrates a powerful general technique: so-called *line-sweep***
- **Algorithm will have O(n log n + p) complexity in the worst case, where n = h + v**

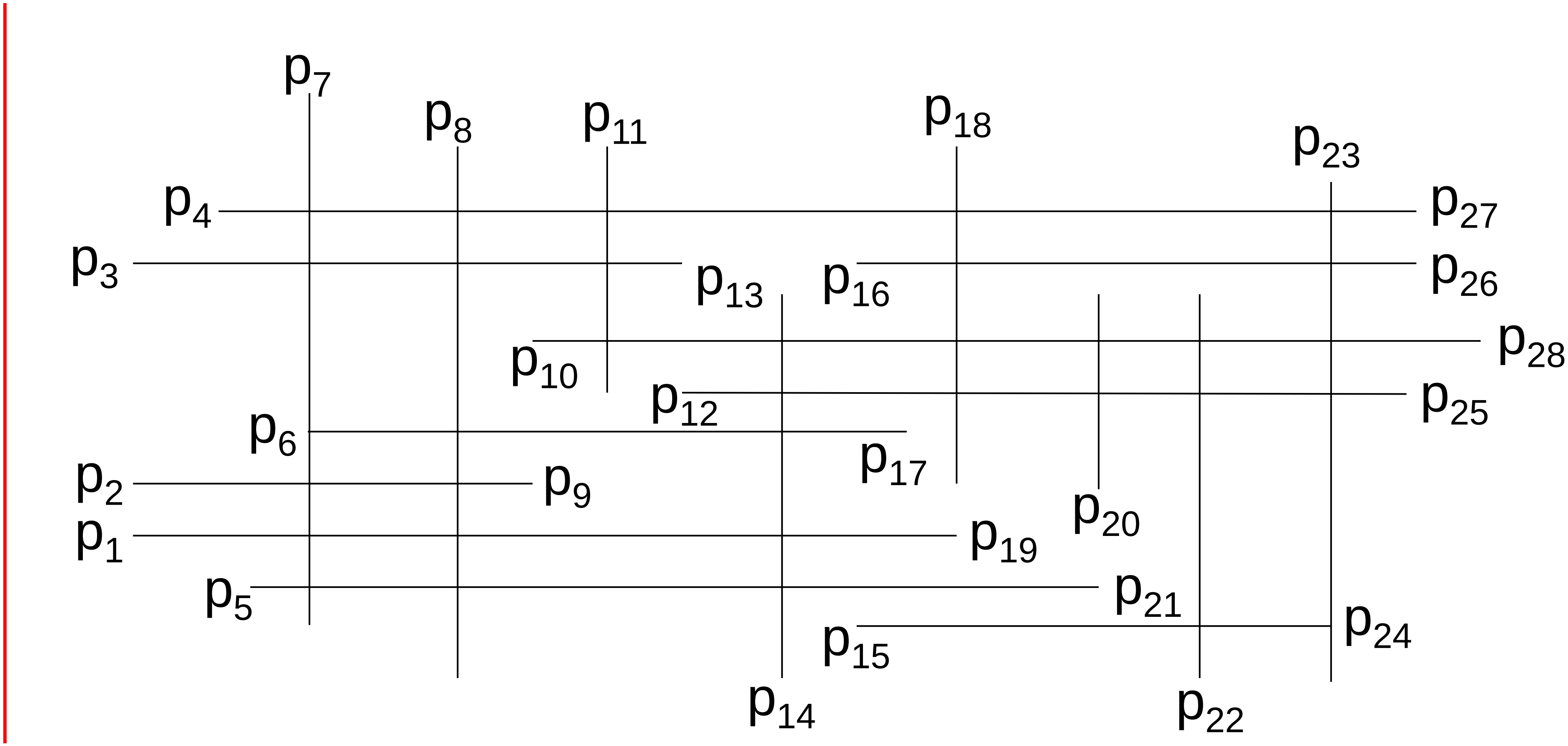
The line-sweep technique

- for brevity, refer to line segments as *lines*
- imagine an infinite vertical line ‘sweeping’ left to right across the plane
- the sweep hits each vertical line once, but hits each horizontal line at its left end point, and leaves it at its right end point
- set up a list of endpoints
  - each vertical line is represented once
  - each horizontal line is represented twice, by its left and right end points
- sort the list by x-coordinate
- assume that no two horizontal lines intersect, and that no two vertical lines intersect

Line-sweep – the basic idea

- **simulate the line-sweep by ‘processing’ the list of endpoints**
- **maintain throughout a set of *candidate* horizontal lines**
  - **those whose left end point has been processed but whose right end point has not**
- **when a vertical line is encountered during the sweep, consider *only the candidate lines* for intersections**
- **will give real speed-up in many cases**
- **but could still have as many as hv comparisons and few (even zero) intersections**
  - **all of the horizontal lines could be candidates throughout the sweep**

The line-sweep algorithm in action

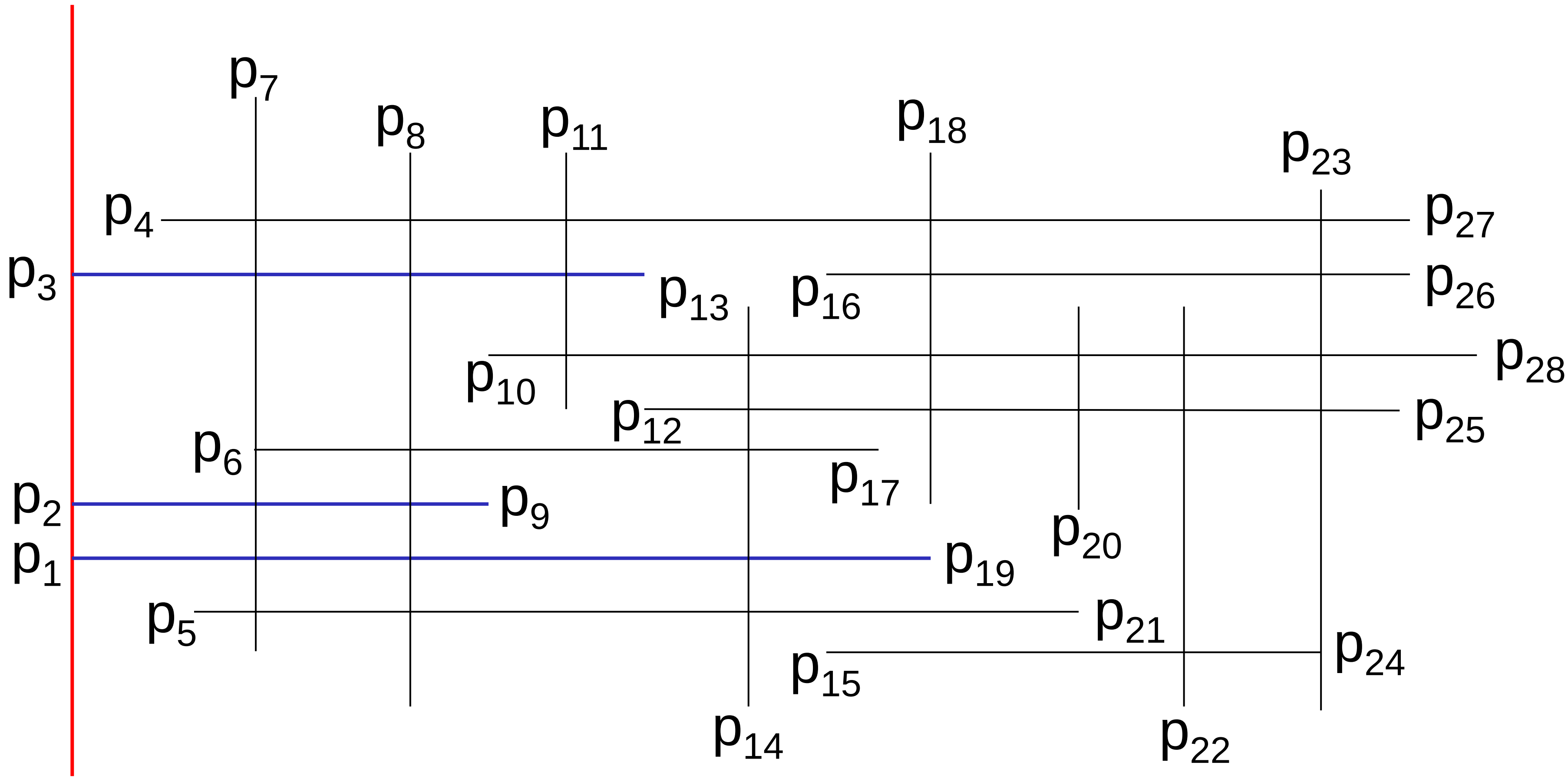


h=10, v=8, number of intersections=34

- 20 horizontal line endpoints
- 8 vertical line endpoints

sorted in increasing order of x-coordinate

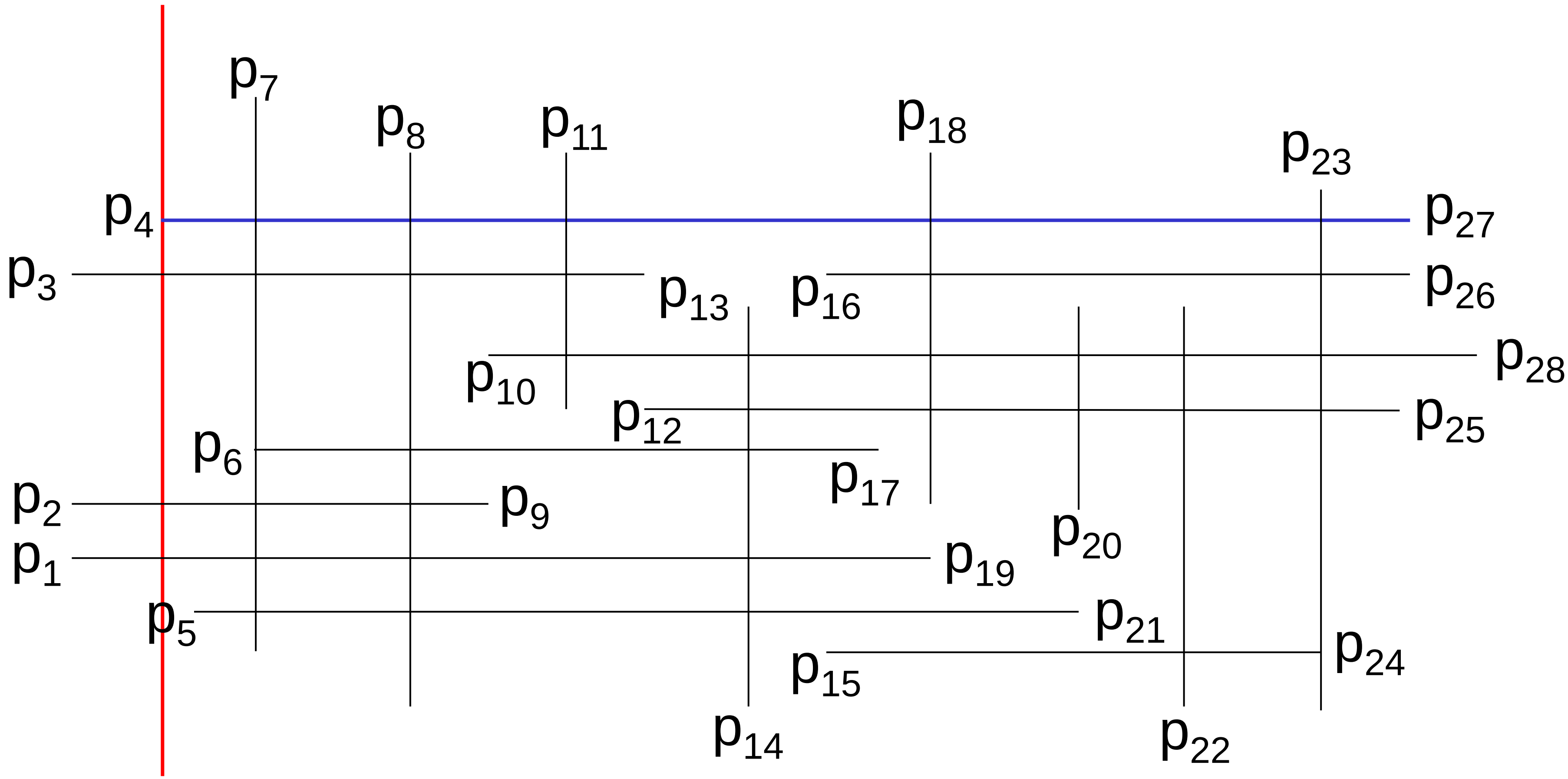
The line-sweep algorithm in action



Candidates: p_1 p_2 p_3

Total number of intersections: 0

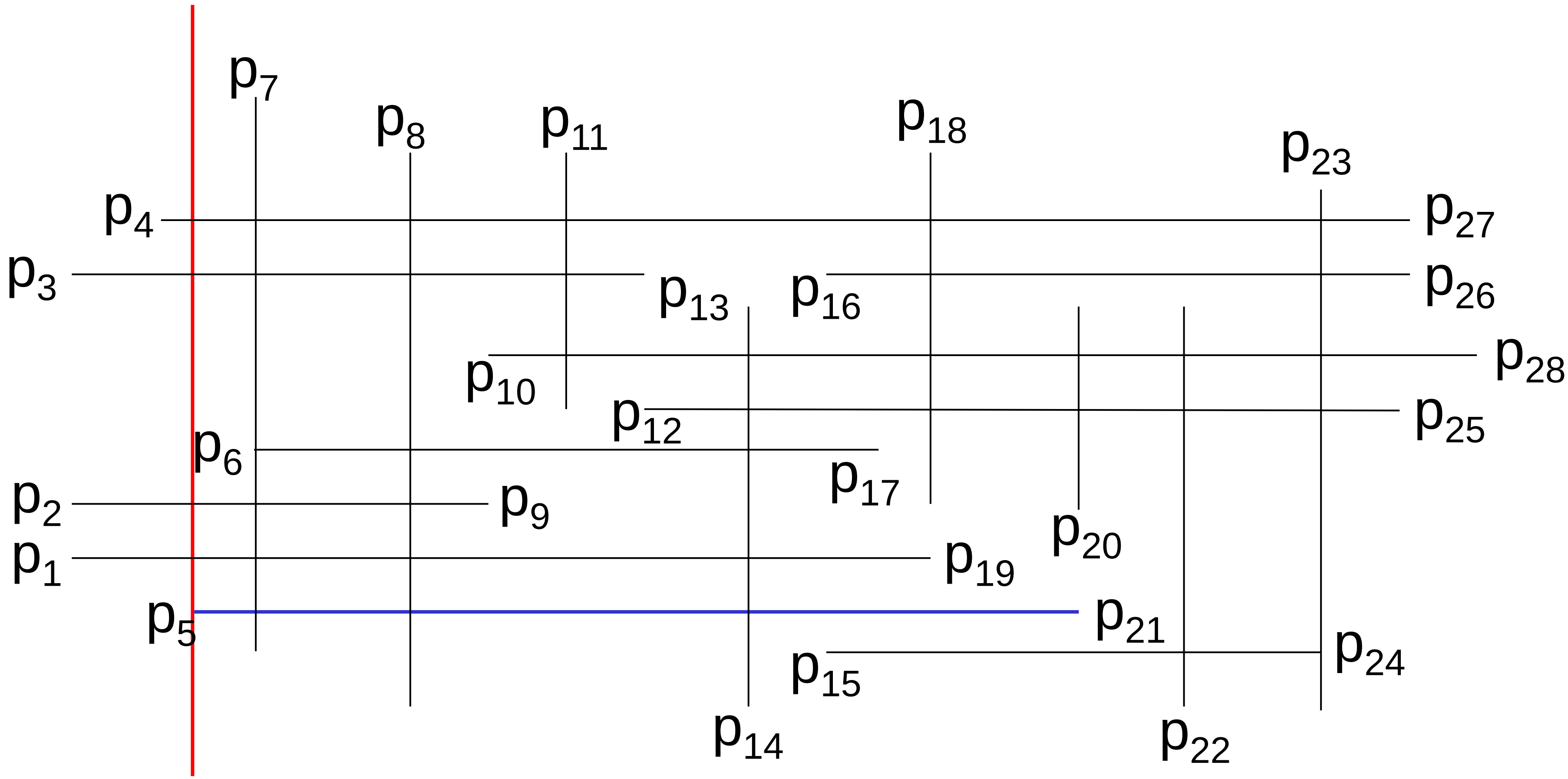
The line-sweep algorithm in action



Candidates: p_1 p_2 p_3 p_4

Total number of intersections: 0

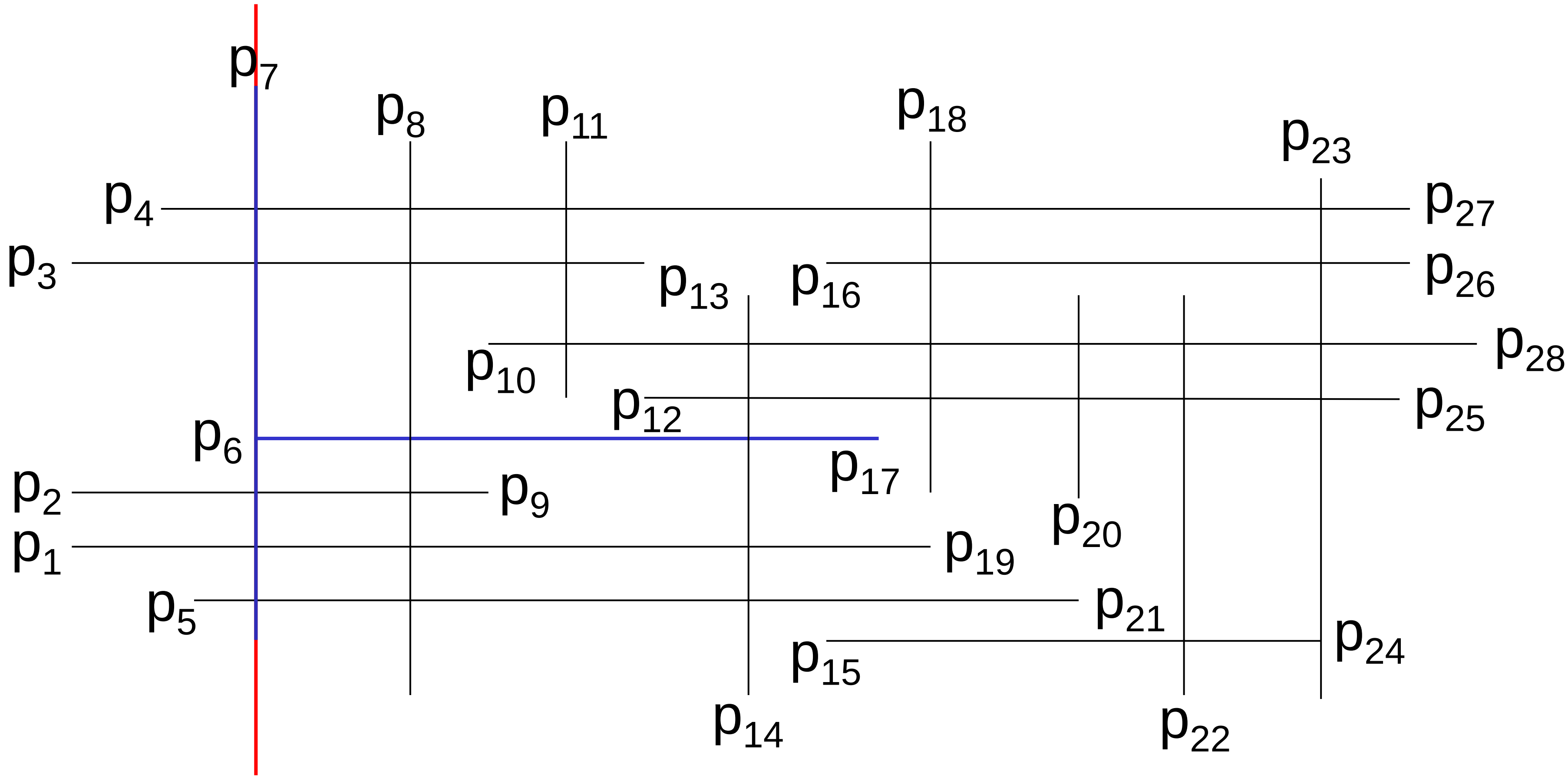
The line-sweep algorithm in action



Candidates: p_1 p_2 p_3 p_4 p_5

Total number of intersections: 0

The line-sweep algorithm in action

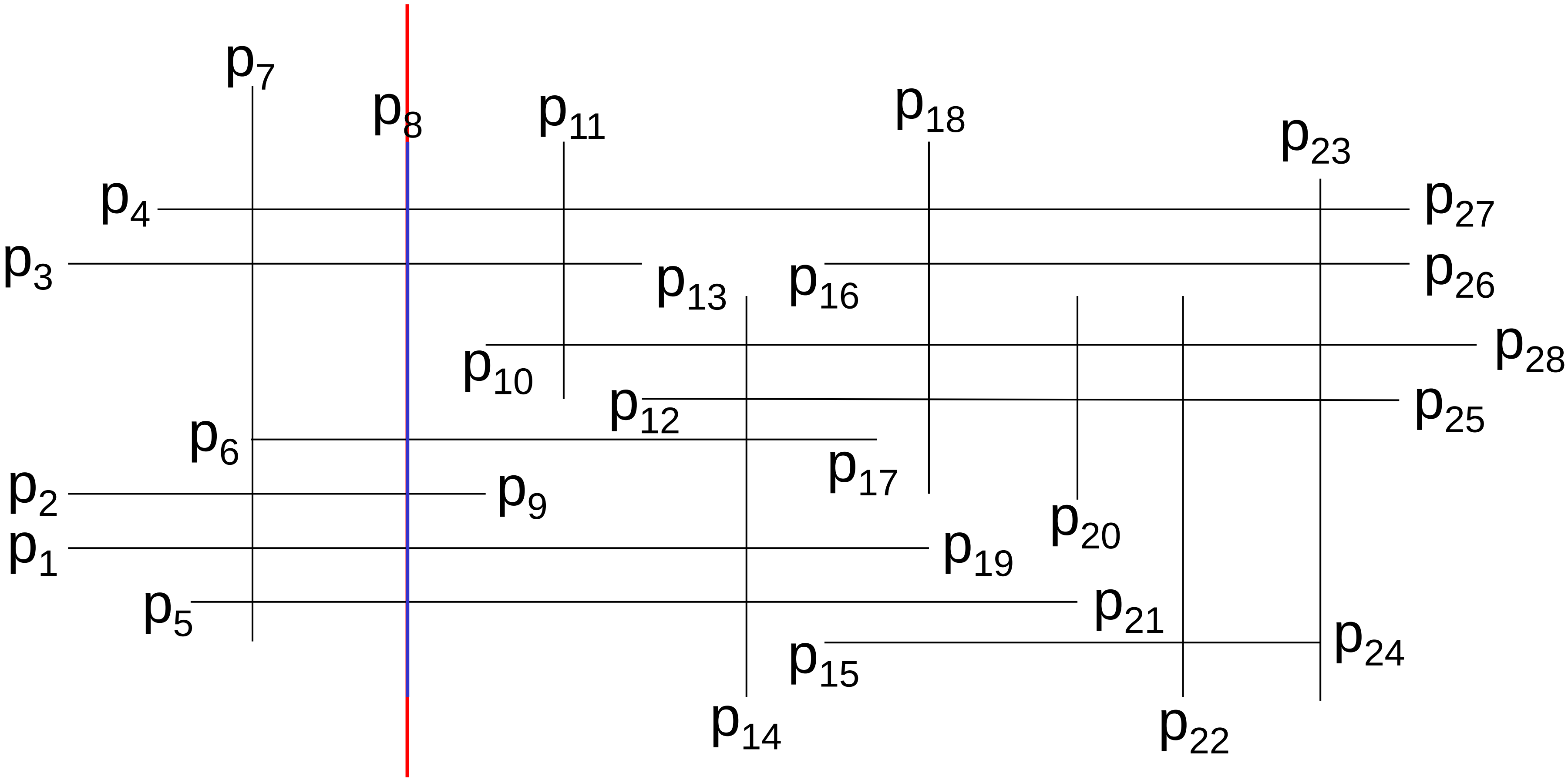


Candidates: p_1 p_2 p_3 p_4 p_5 p_6

Intersections from the vertical line: 6

Total number of intersections: 6

The line-sweep algorithm in action

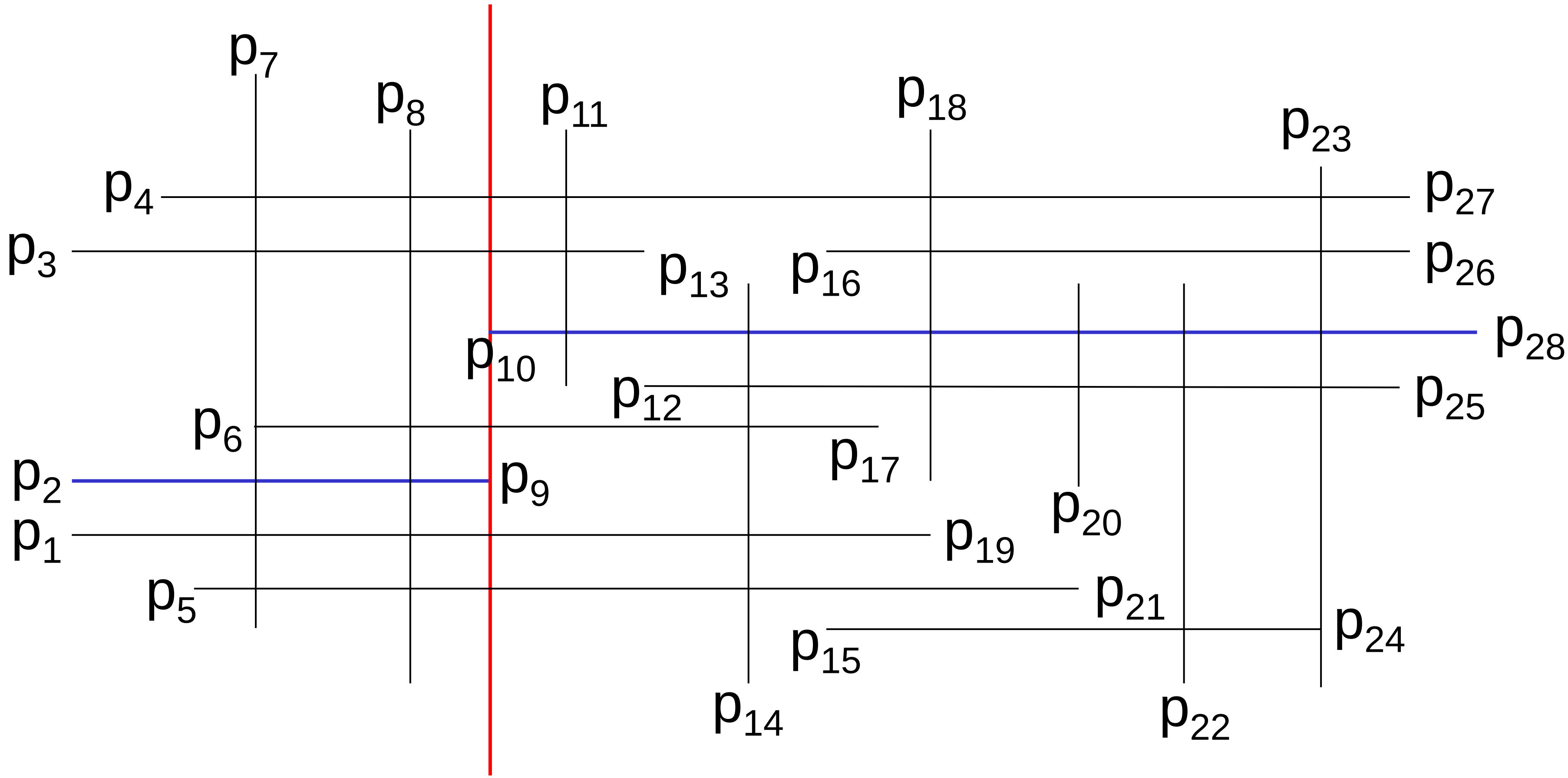


Candidates: p_1 p_2 p_3 p_4 p_5 p_6

Intersections from the vertical line: 6

Total number of intersections:12

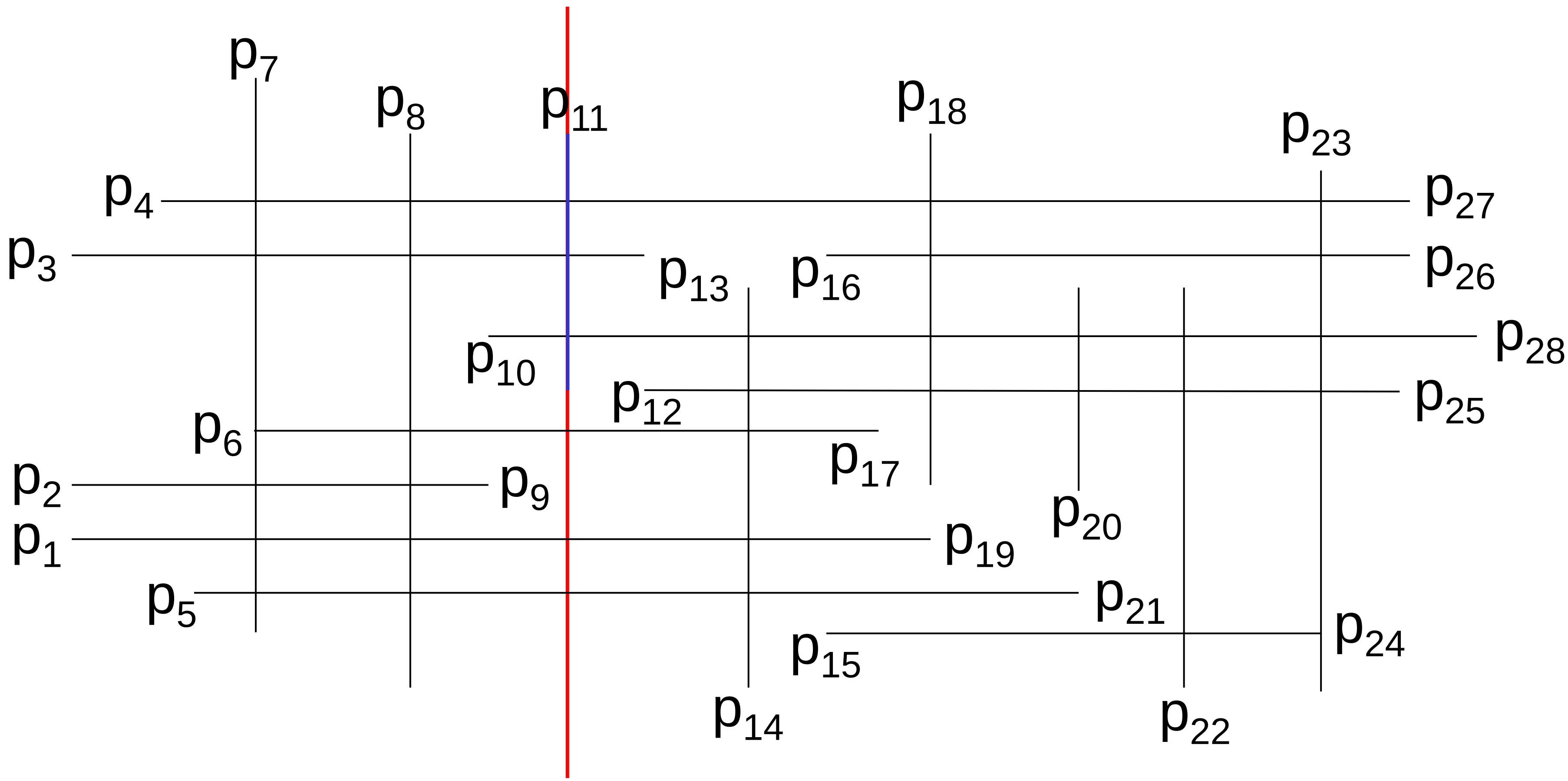
The line-sweep algorithm in action



Candidates: p_1 p_2 p_3 p_4 p_5 p_6 p_{10}

Total number of intersections:12

The line-sweep algorithm in action

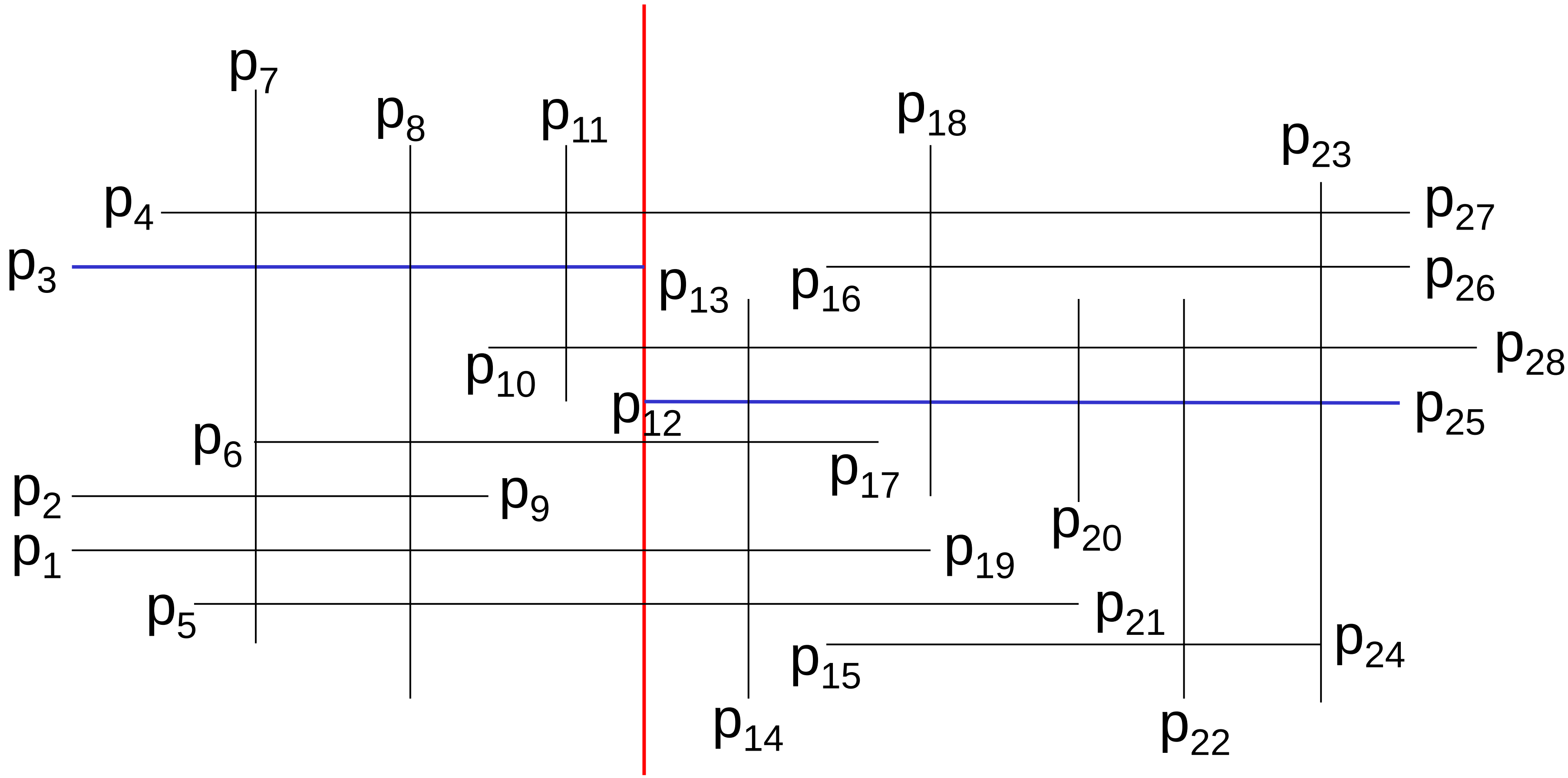


Candidates: p_1 p_3 p_4 p_5 p_6 p_{10}

Intersections from the vertical line: 3

Total number of intersections:15

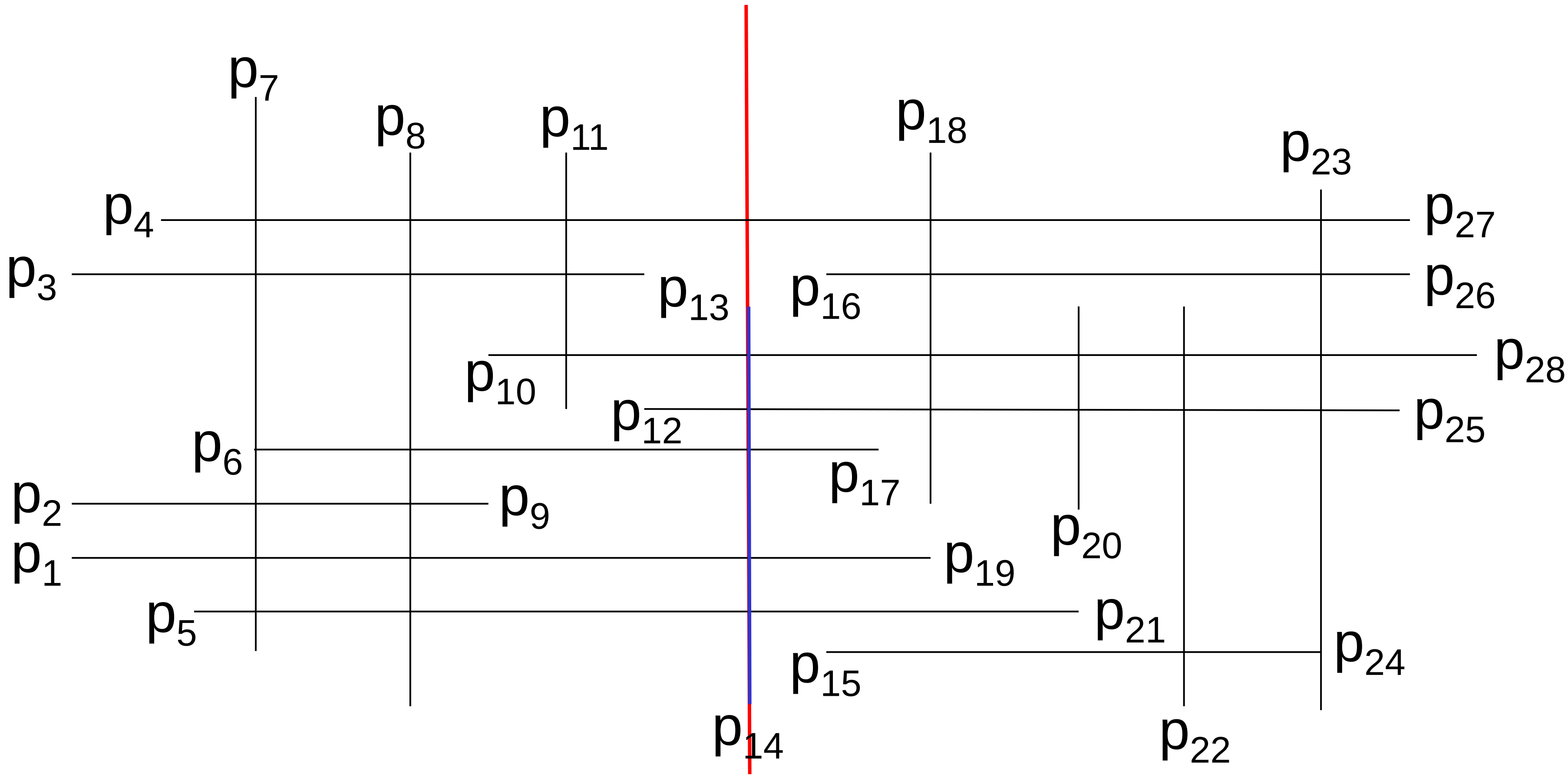
The line-sweep algorithm in action



Candidates: p_1 p_3 p_4 p_5 p_6 p_{10} p_{12}

Total number of intersections:15

The line-sweep algorithm in action

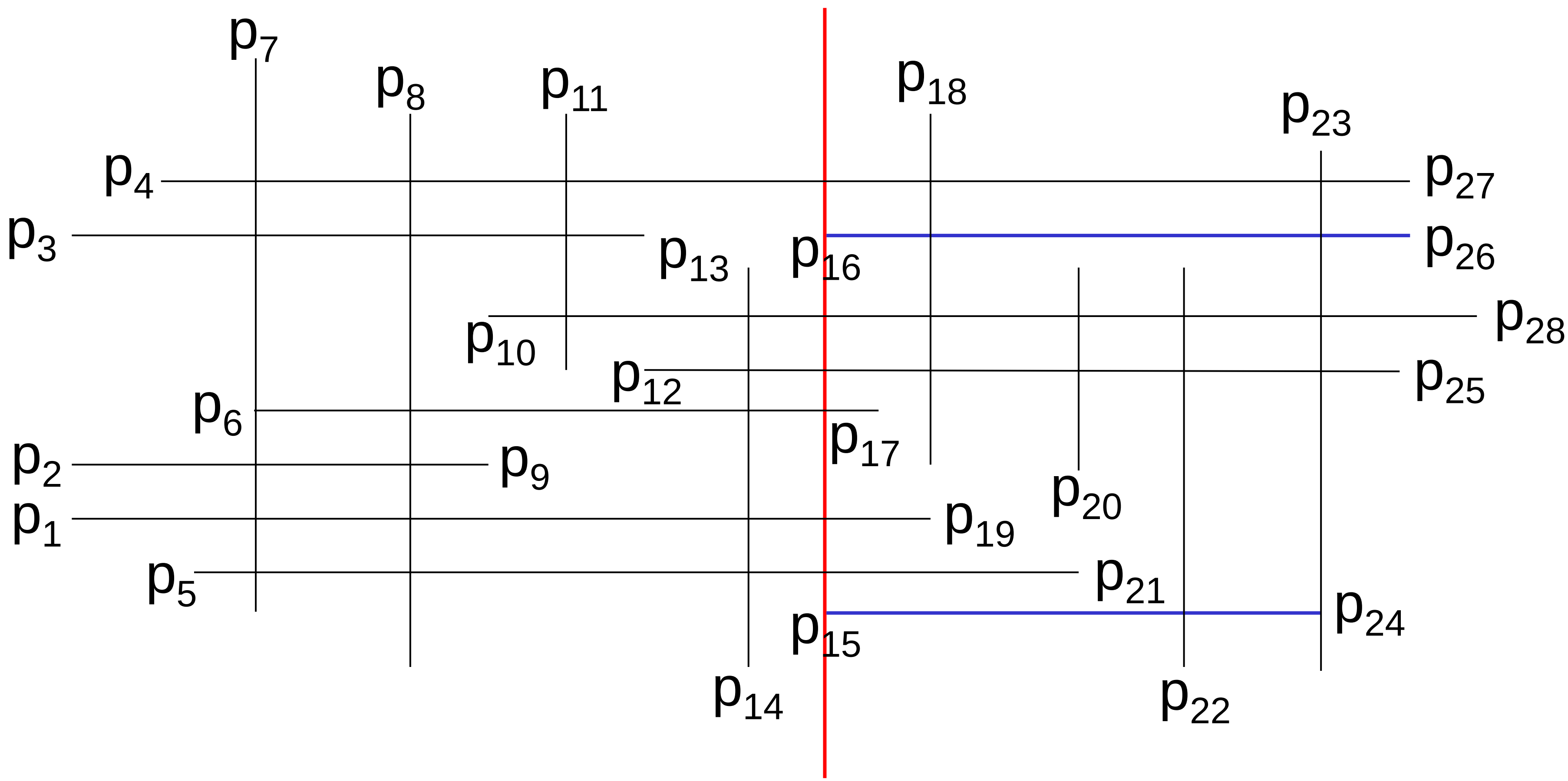


Candidates: p_1 p_4 p_5 p_6 p_{10} p_{12}

Intersections from the vertical line: 5

Total number of intersections: 20

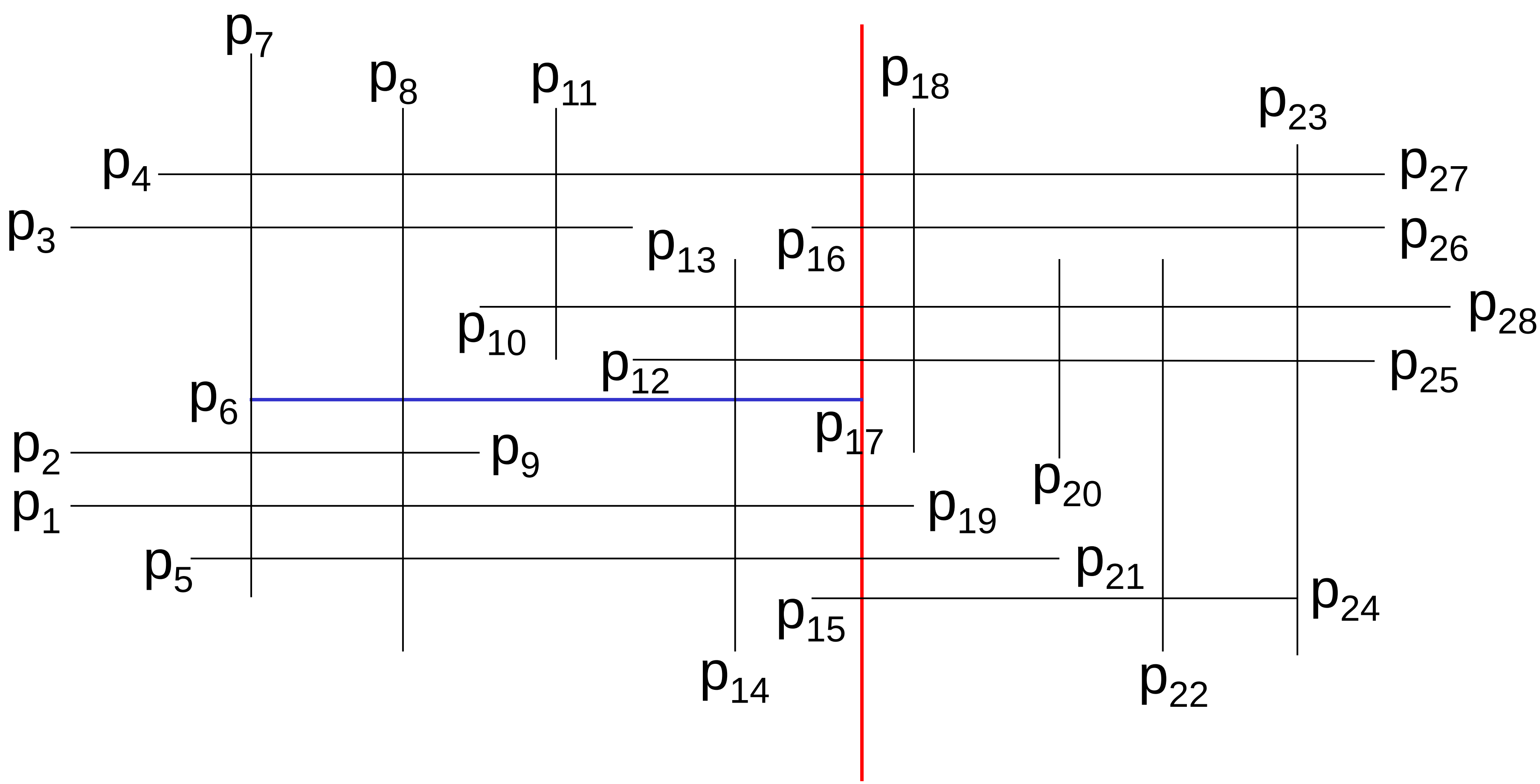
The line-sweep algorithm in action



Candidates: p_1 p_4 p_5 p_6 p_{10} p_{12} p_{15} p_{16}

Total number of intersections: 20

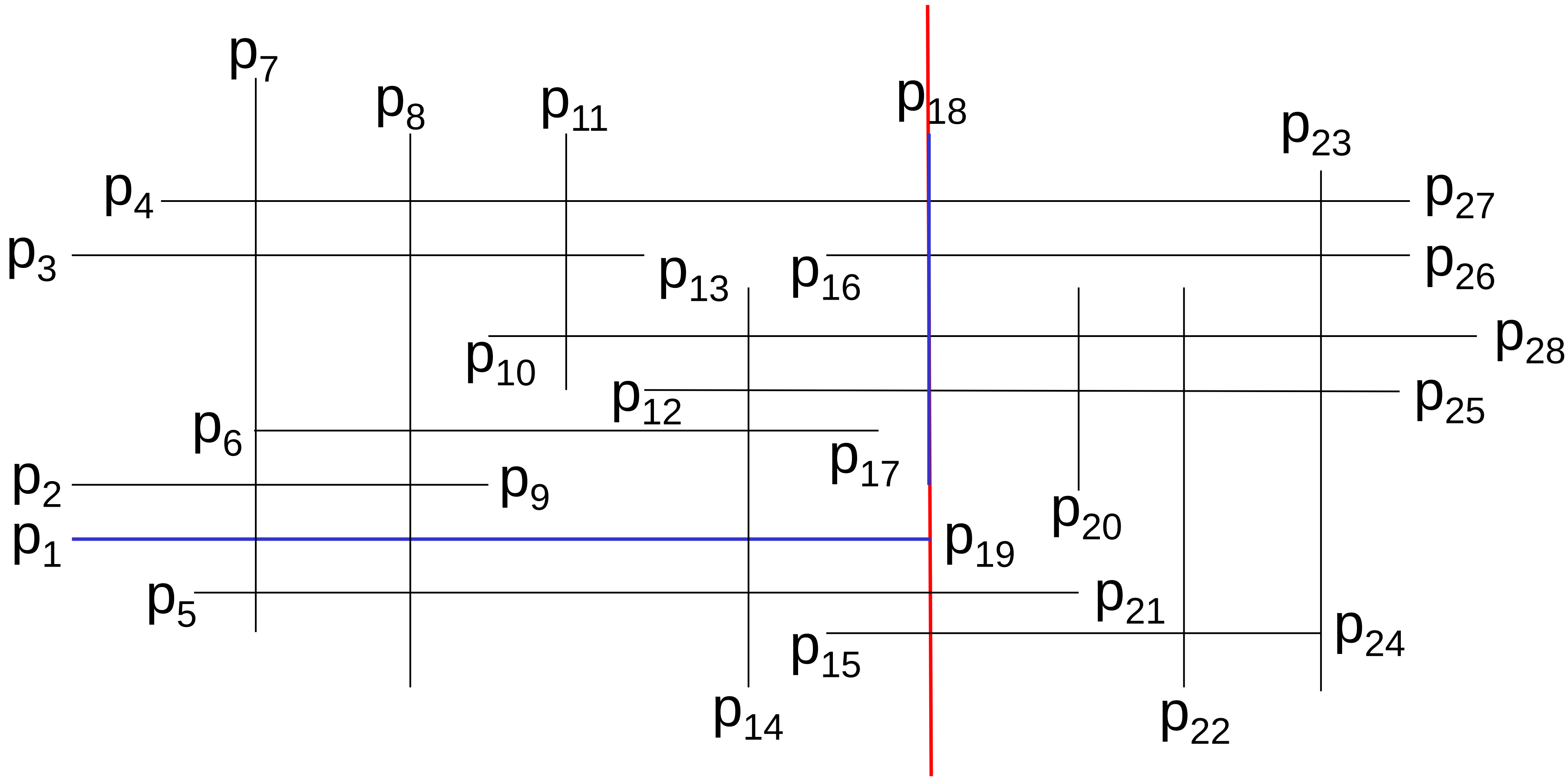
The line-sweep algorithm in action



Candidates: p_1 p_4 p_5 p_6 p_{10} p_{12} p_{15} p_{16}

Total number of intersections: 20

The line-sweep algorithm in action

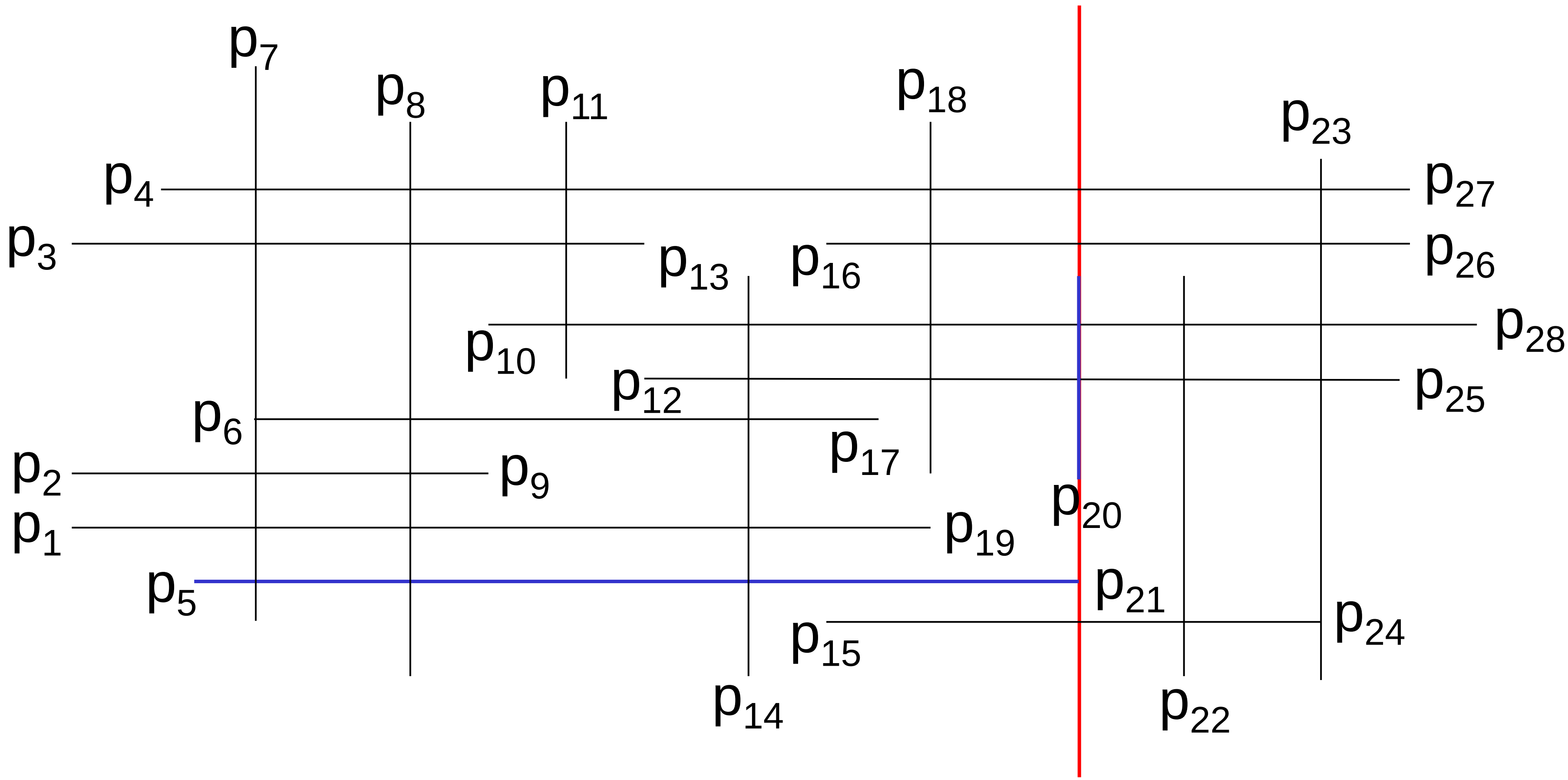


Candidates: p_1 p_4 p_5 p_{10} p_{12} p_{15} p_{16}

Intersections from the vertical line: 4

Total number of intersections: 24

The line-sweep algorithm in action

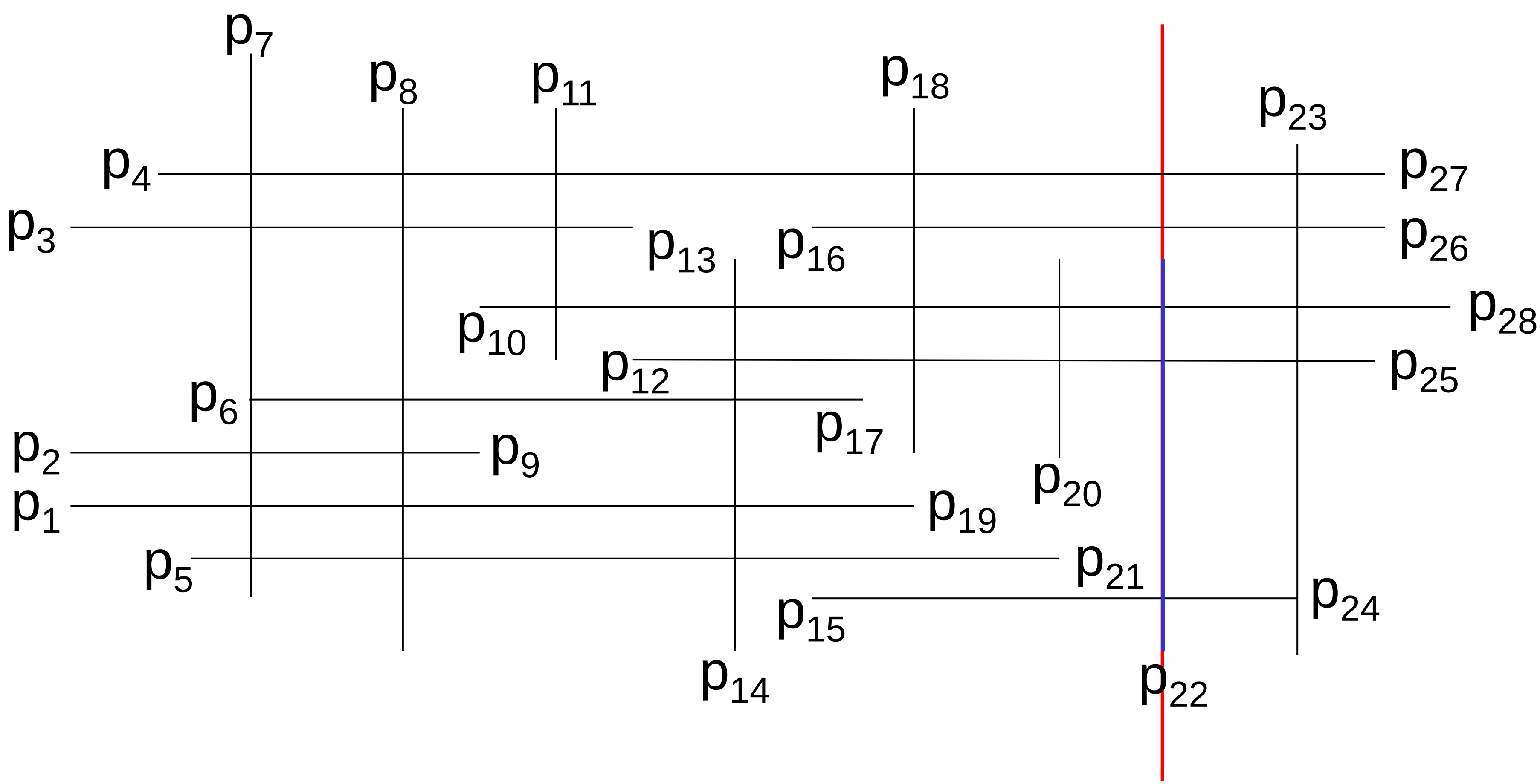


Candidates: p_4 p_5 p_{10} p_{12} p_{15} p_{16}

Intersections from the vertical line: 2

Total number of intersections: 26

The line-sweep algorithm in action

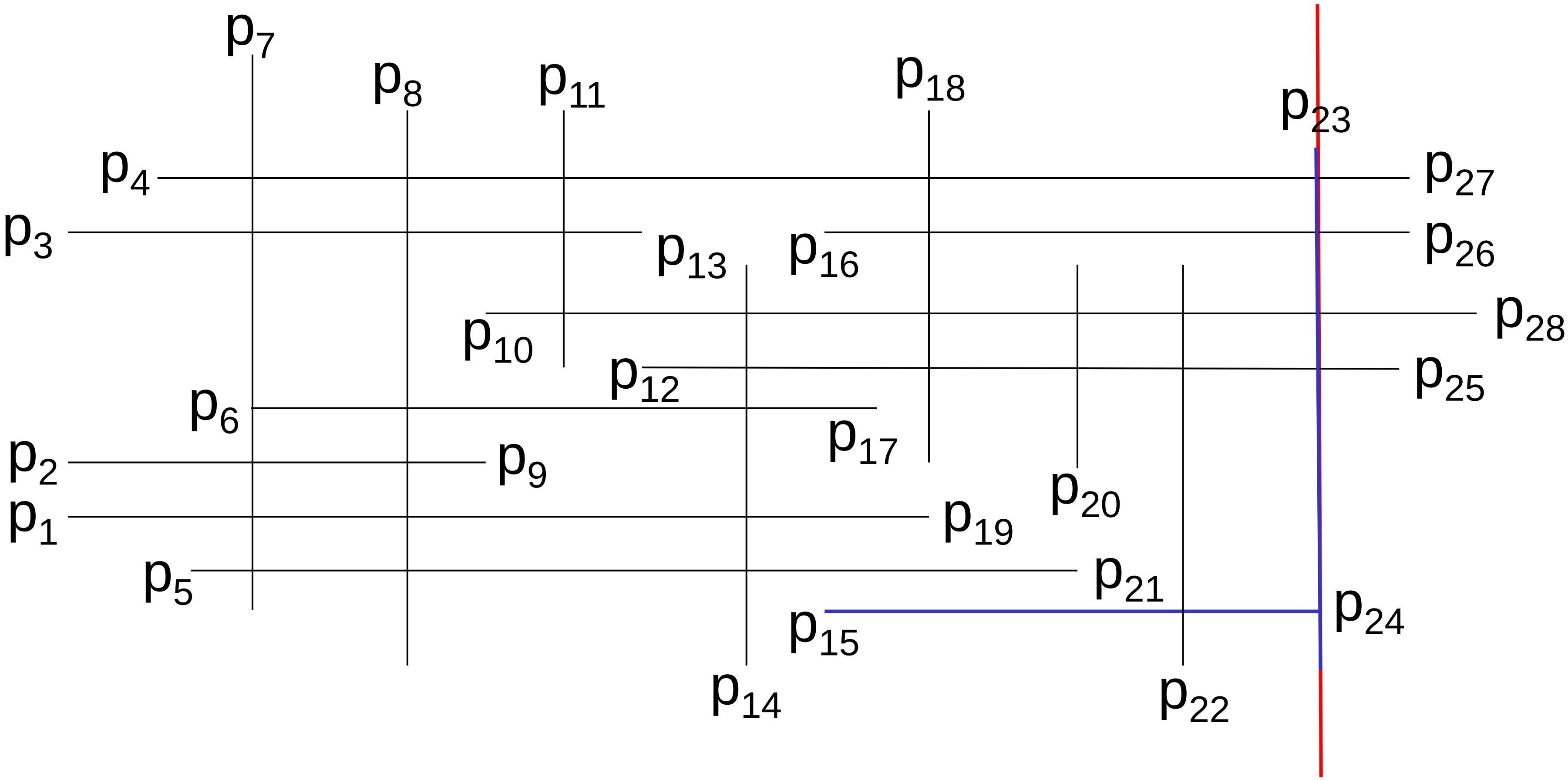


Candidates: p_4 p_{10} p_{12} p_{15} p_{16}

Intersections from the vertical line: 3

Total number of intersections: 29

The line-sweep algorithm in action

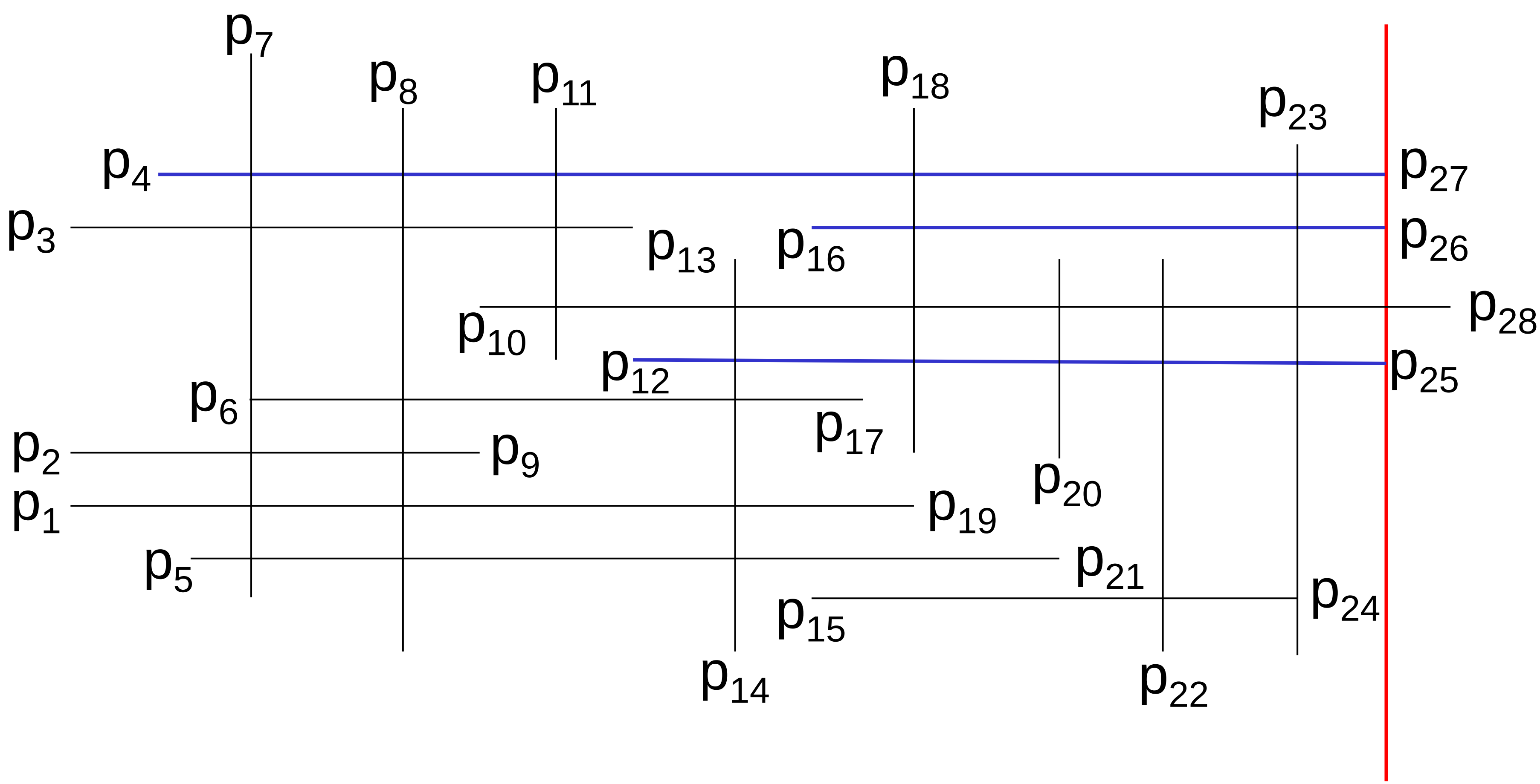


Candidates: p_4 p_{10} p_{12} p_{15} p_{16}

Intersections from the vertical line: 5

Total number of intersections: 34

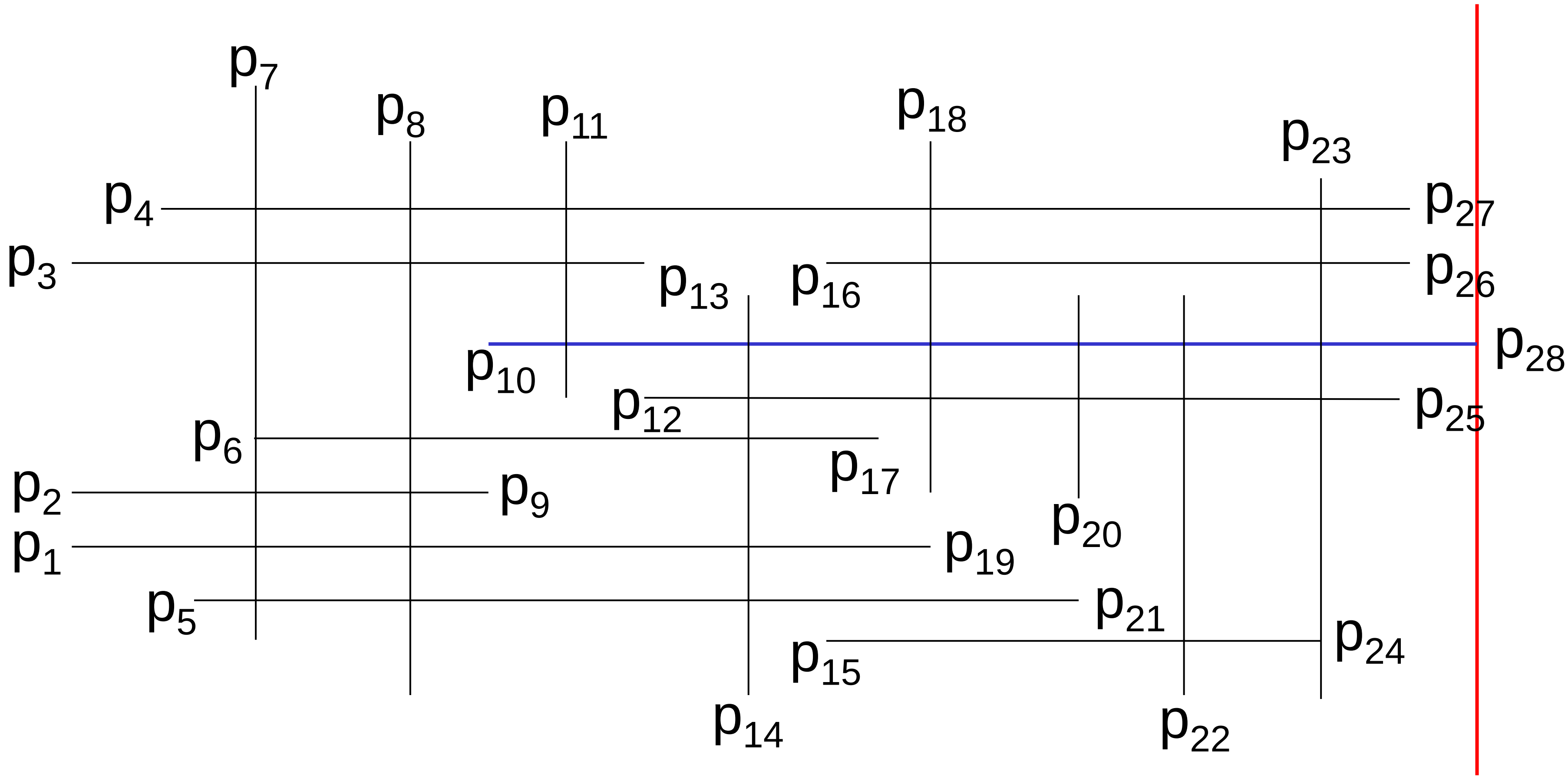
The line-sweep algorithm in action



Candidates: p_4 p_{10} p_{12} p_{16}

Total number of intersections: 34

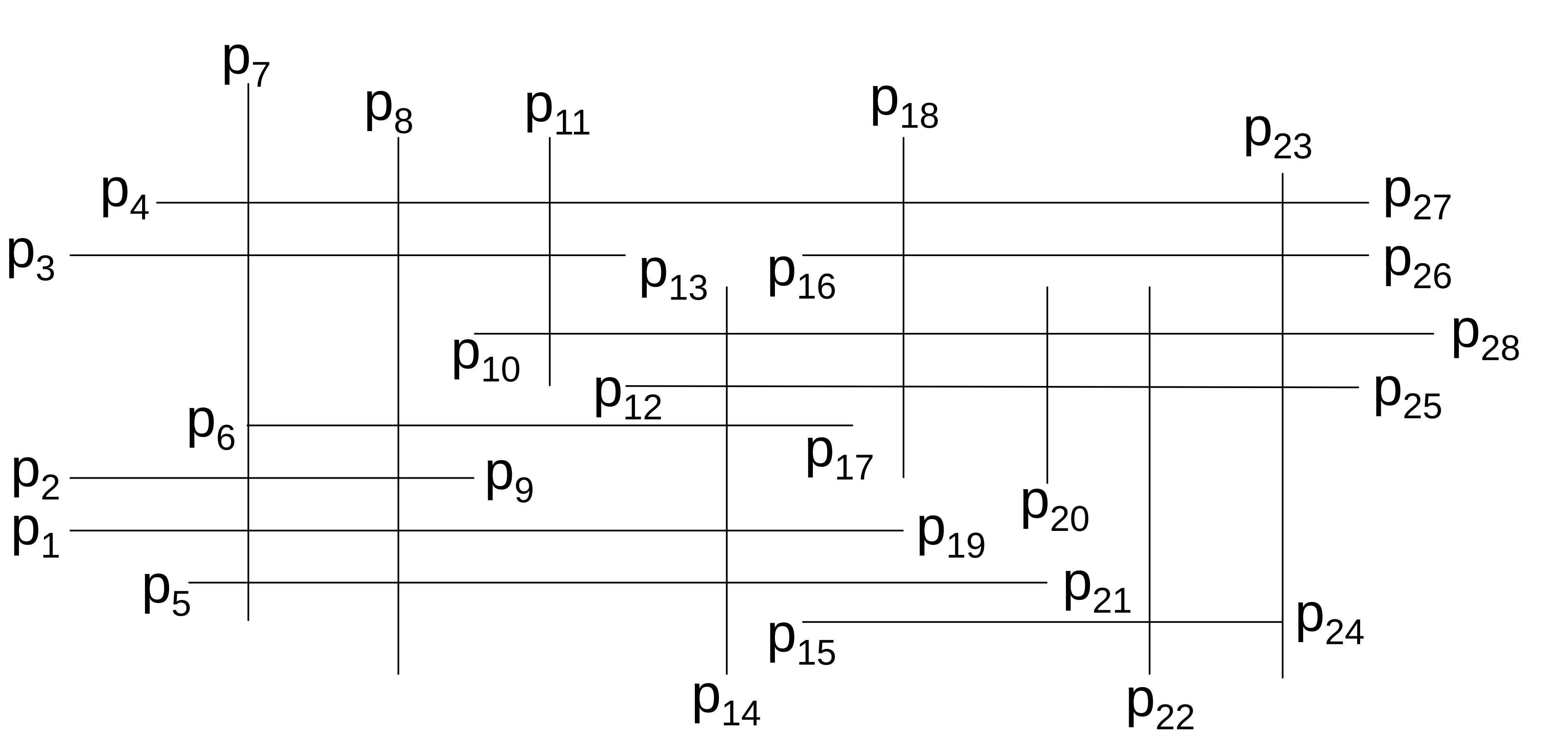
The line-sweep algorithm in action



Candidates: p_{10}

Total number of intersections: 34

The line-sweep algorithm in action



Candidates:

Total number of intersections: 34

The sort on line endpoints

Recall that

- **each vertical line is represented once**
- **each horizontal line is represented twice,**
  - by its left and right end points

Sort the list by x-coordinate, but, if two or more points have the same x-coordinate, break ties so that:

a horizontal line's left endpoint

always comes before

a vertical line endpoint

which in turn always comes before

a horizontal line's right endpoint

The key question

- **How can we represent the candidate set so that, when a vertical line is reached, we don't have to compare it with all of the candidates?**

Developing a solution

- **Refer to the y-coordinate of a candidate as the *value* of the candidate**
- **If current vertical line has endpoints (x,y_1) and (x,y_2), we want to find all candidates with values in the range $y_1 .. y_2$**
- **We should represent the candidate set in a way that facilitates *range searching***
- **But we also need to perform insertions and deletions efficiently**

Efficient range searching

- Consider using an *array* to store the candidates – the goal is to find all entries with value in range $y_1 .. y_2$
- Naïve method is to check all entries – no better than $O(h)$ in the worst case
- Faster method is to sort entries in the array by value
  - then use a binary search for y_1 and then scan along
  - size of array is at most h
  - search is $O(\log h)$ and scan is $O(k)$ if there are k items in range
- But insertion / deletion are no better than $O(h)$ in the worst case
- Is there an alternative data structure?

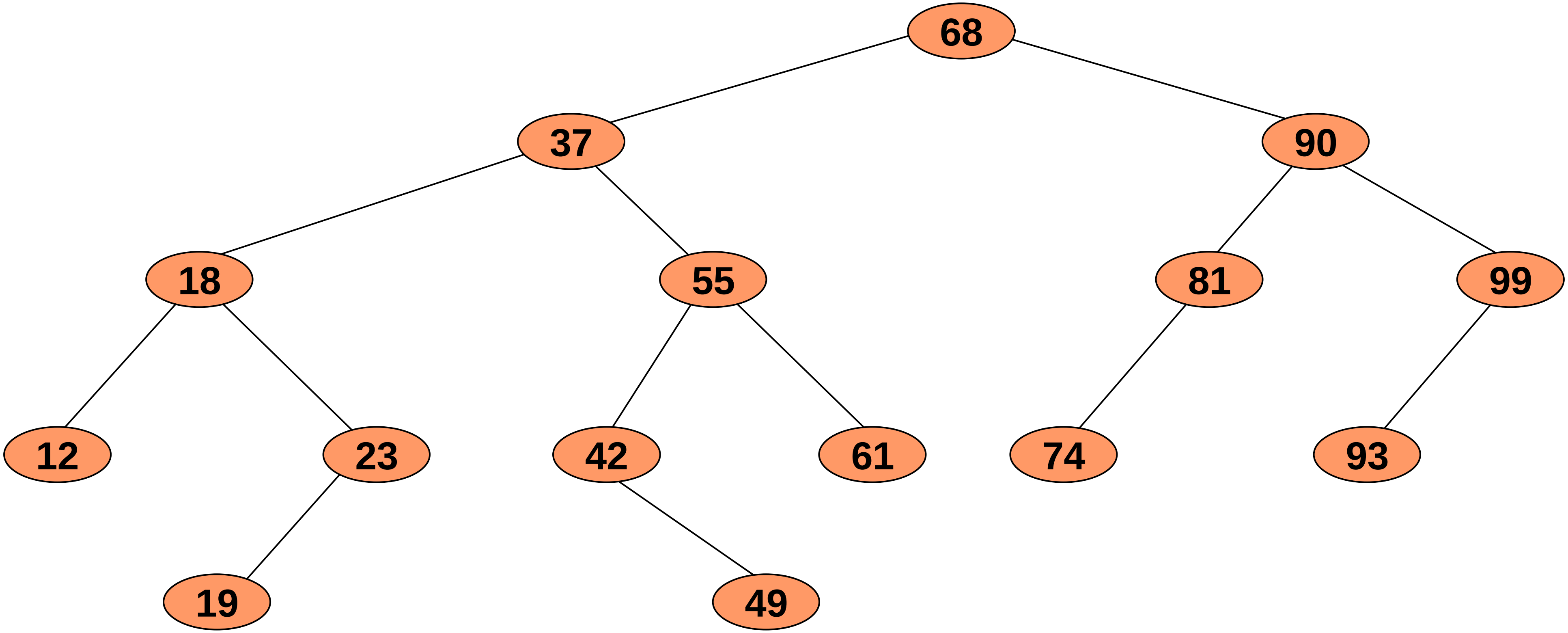
AVL Trees – revision

- **AVL trees are balanced binary search trees**
- **For each node, heights of left subtree and right subtree differ by at most 1**
- **An AVL tree with h nodes has height O(log h)**
- **Search, insertion and deletion are all O(log h)**

Back to range searching problem

- **Consider using an *AVL tree***
  - **search, insert, delete are all O(log h)**
  - **range searching is O(k + log h)**

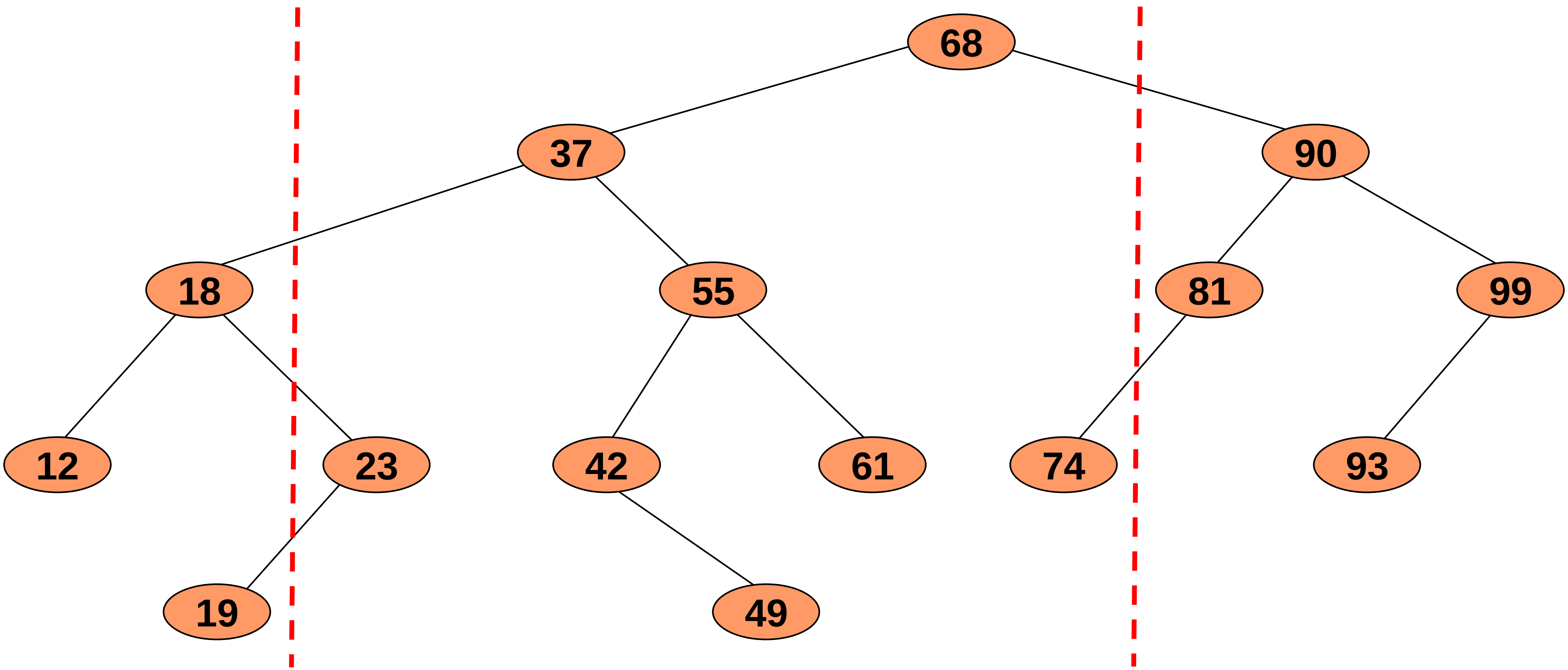
Example – AVL tree representing a set of 15 integers



Suppose we are searching for all items in the range 20..80

An example of a *one-dimensional range search query*

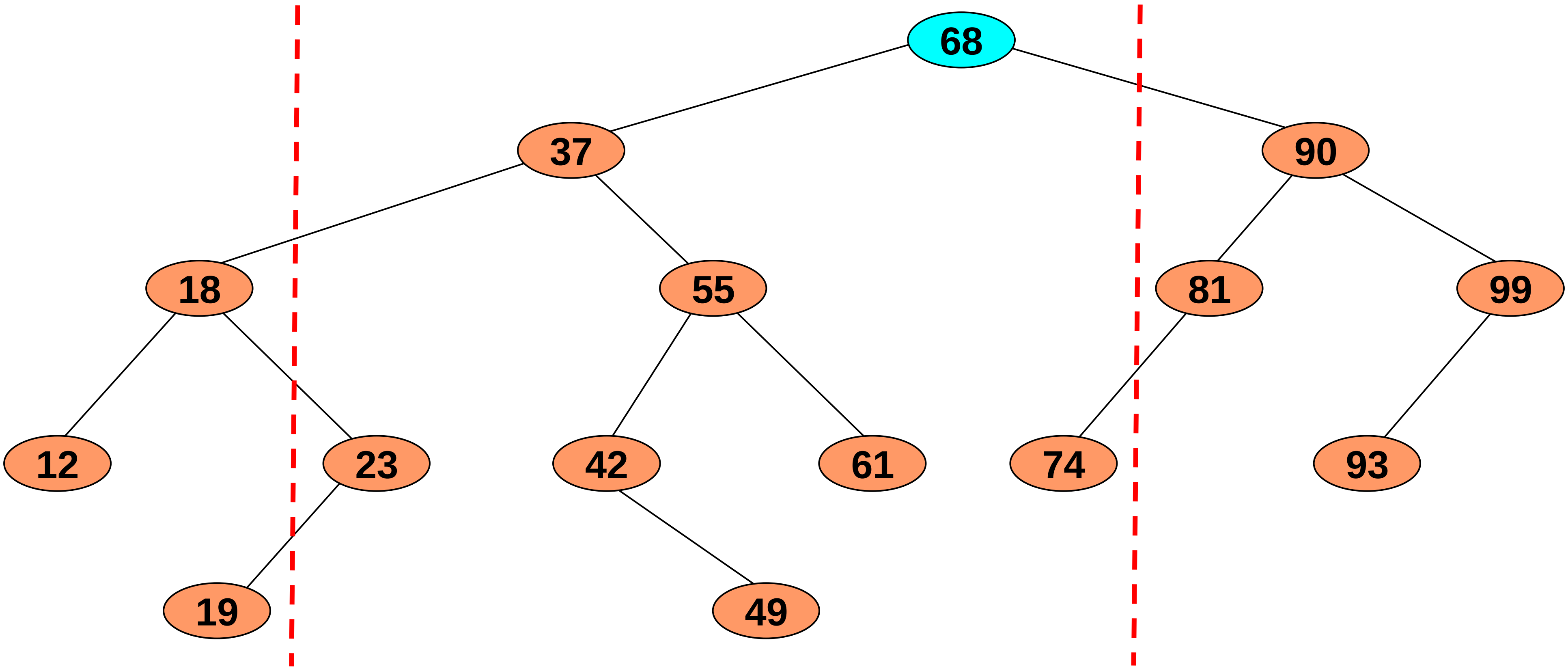
Example – AVL tree representing a set of 15 integers



Suppose we are searching for all items in the range 20..80

An example of a *one-dimensional range search query*

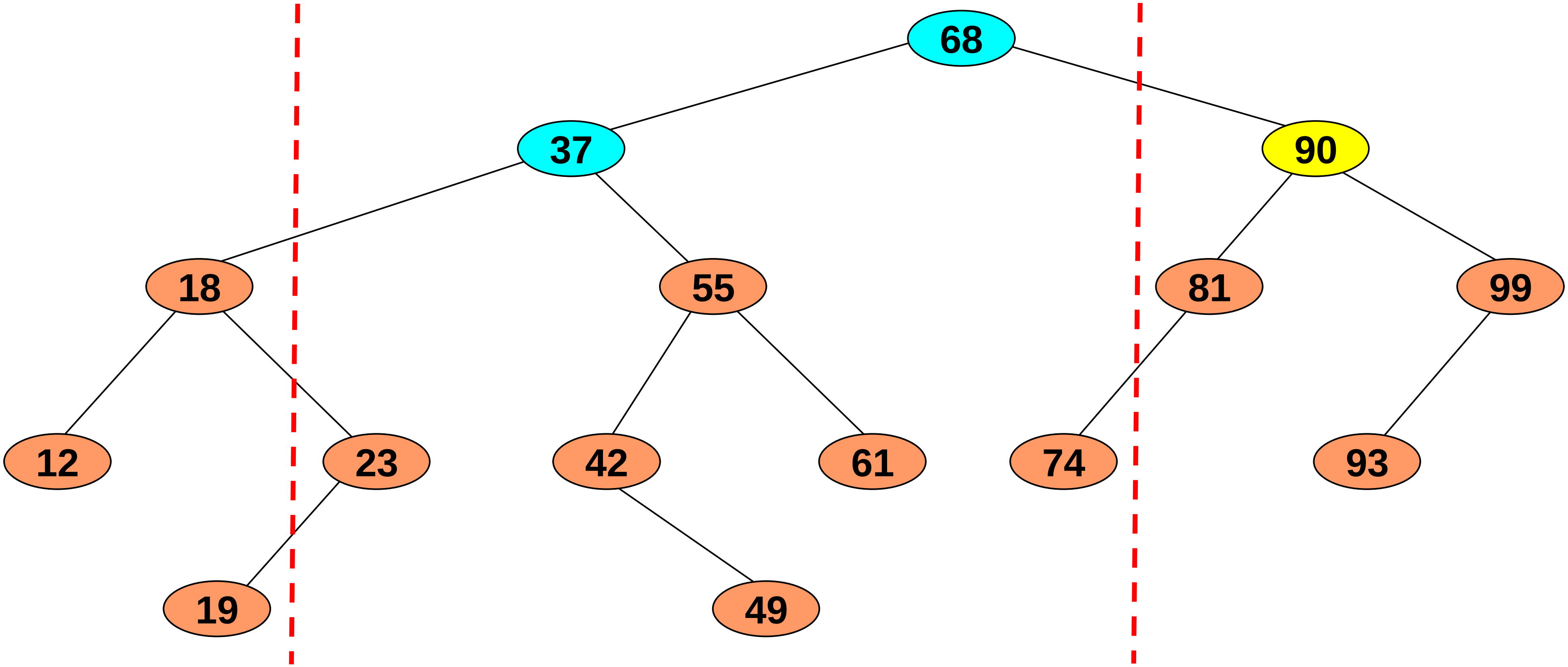
Example – AVL tree representing a set of 15 integers



Suppose we are searching for all items in the range 20..80

An example of a *one-dimensional range search query*

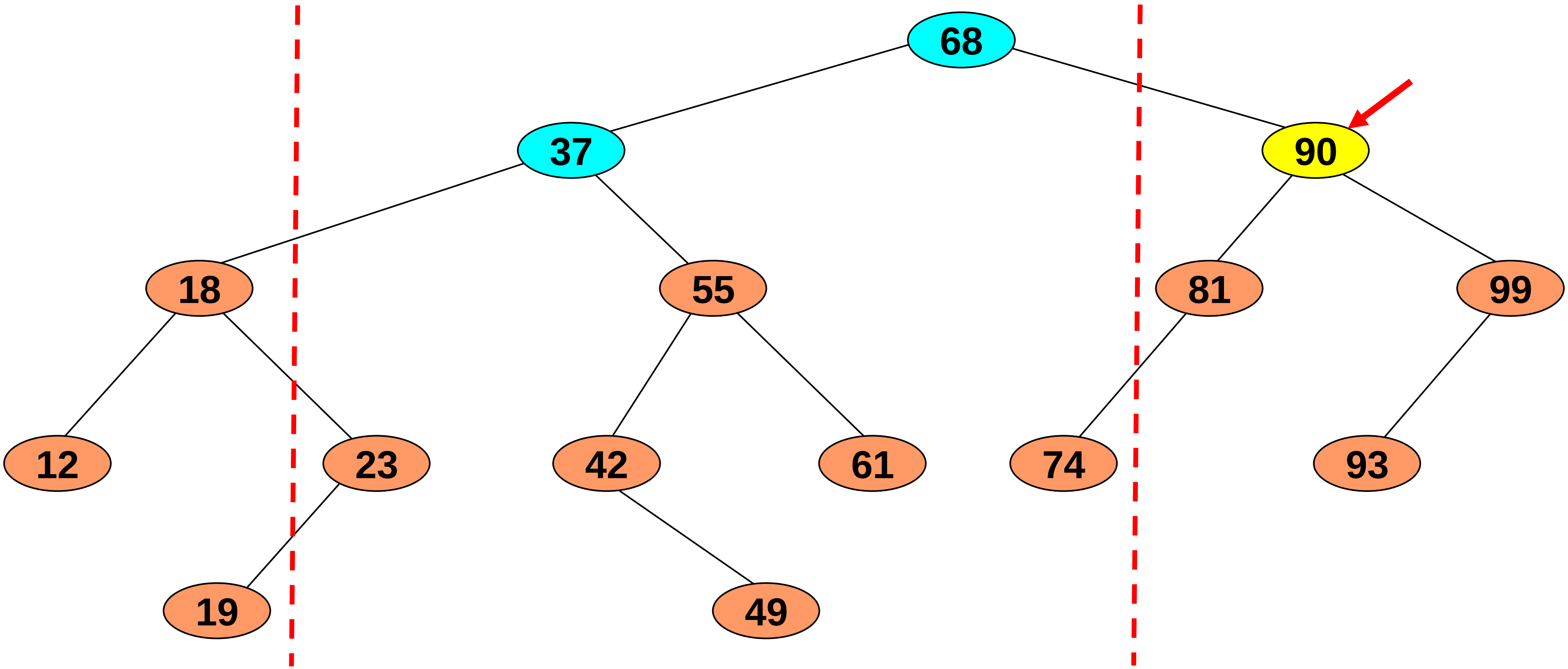
Example – AVL tree representing a set of 15 integers



Suppose we are searching for all items in the range 20..80

An example of a *one-dimensional range search query*

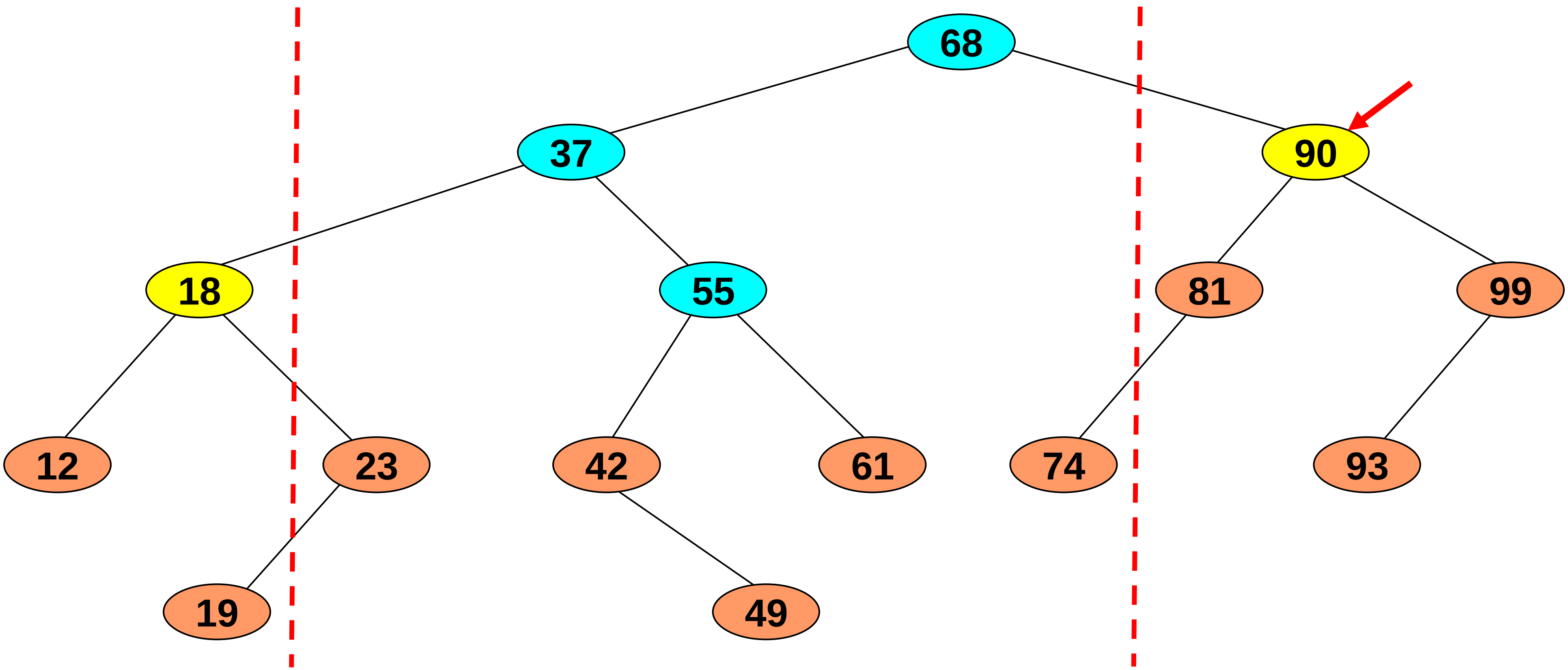
Example – AVL tree representing a set of 15 integers



Suppose we are searching for all items in the range 20..80

An example of a *one-dimensional range search query*

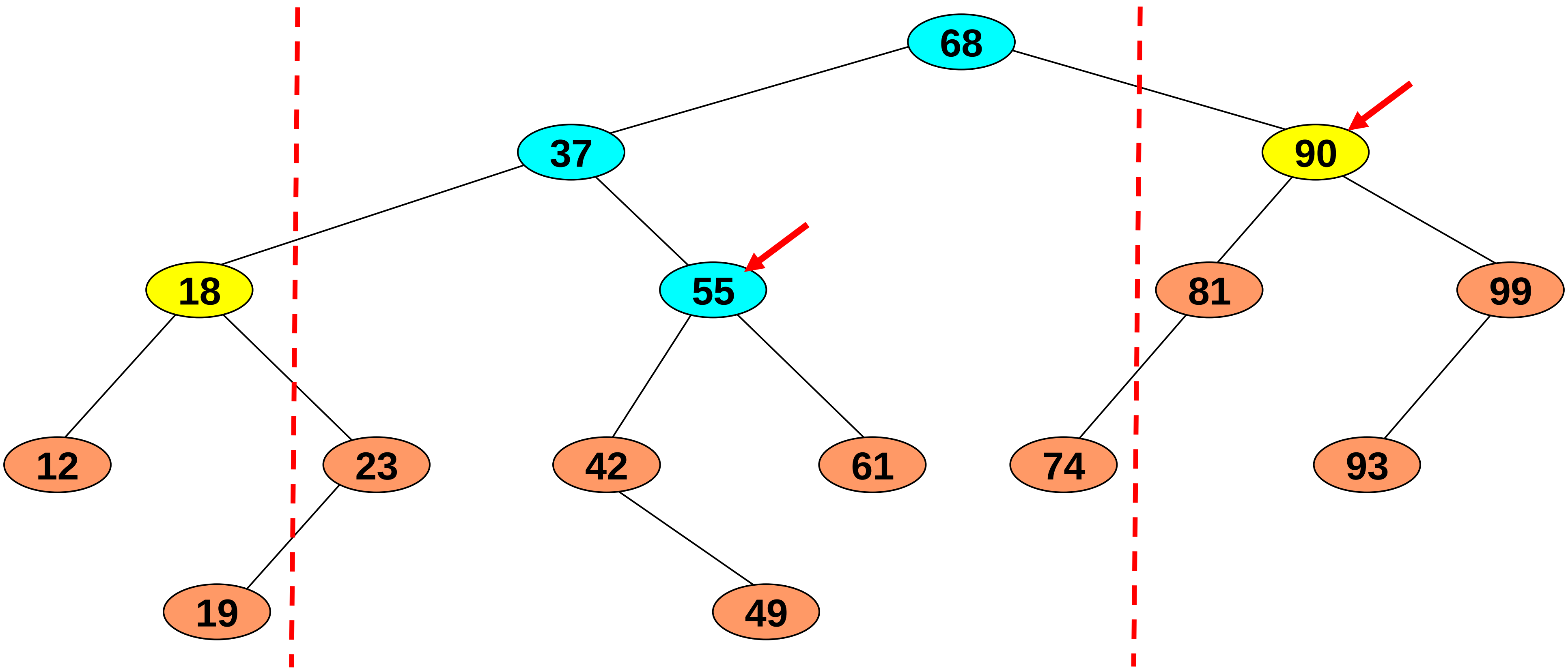
Example – AVL tree representing a set of 15 integers



Suppose we are searching for all items in the range 20..80

An example of a *one-dimensional range search query*

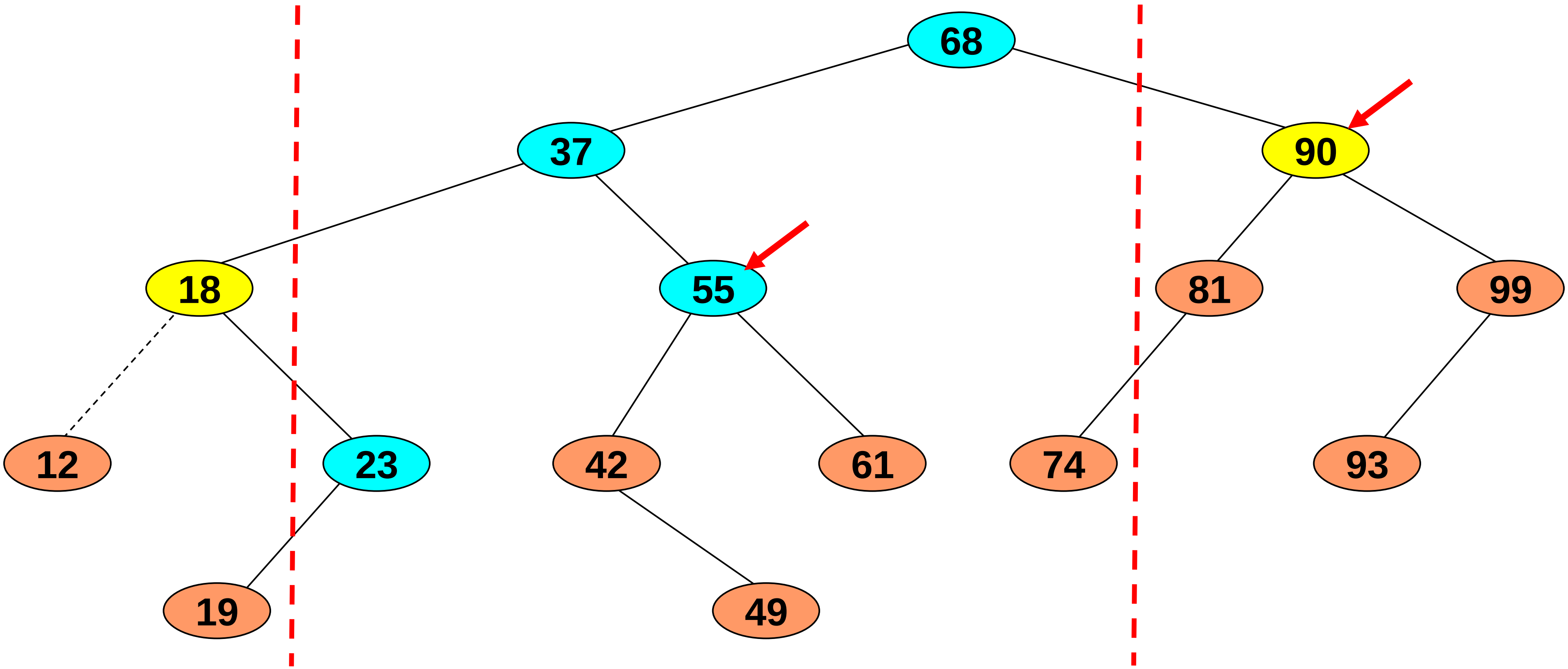
Example – AVL tree representing a set of 15 integers



Suppose we are searching for all items in the range 20..80

An example of a *one-dimensional range search query*

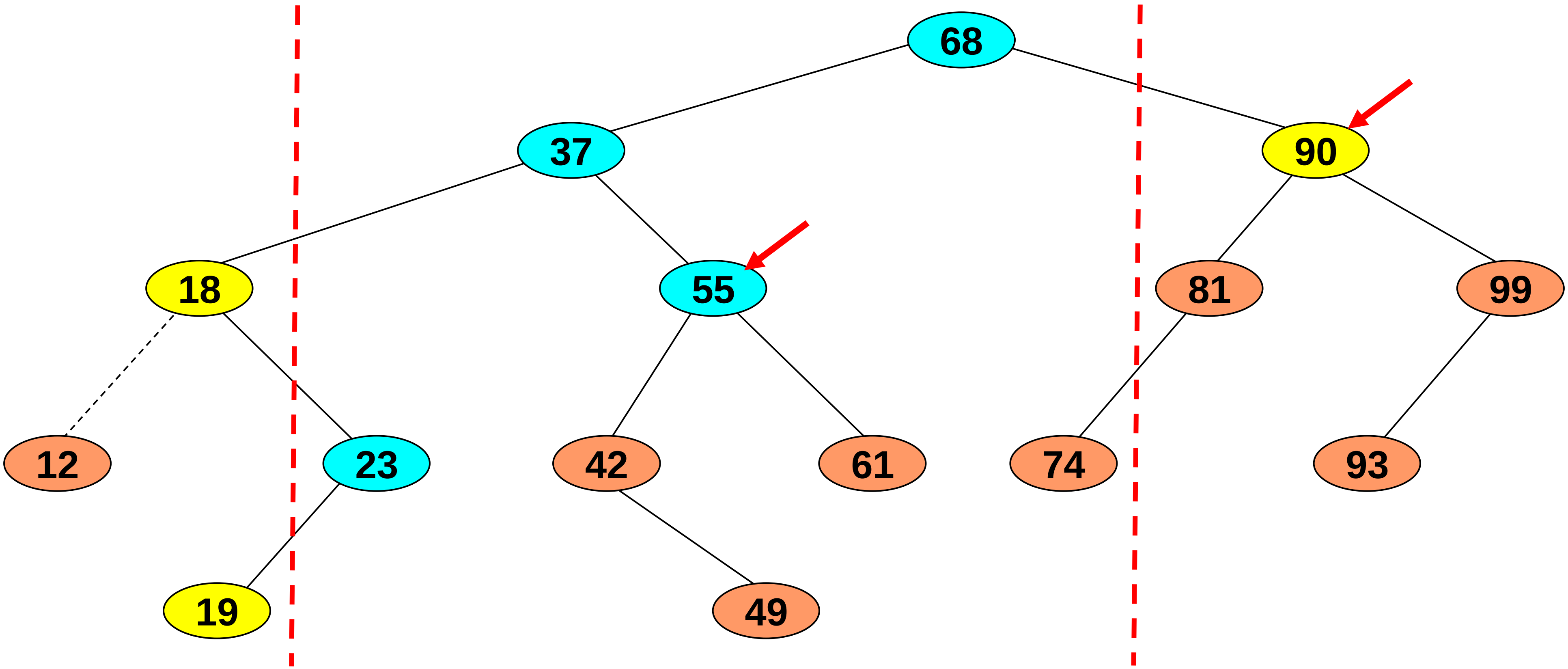
Example – AVL tree representing a set of 15 integers



Suppose we are searching for all items in the range 20..80

An example of a *one-dimensional range search query*

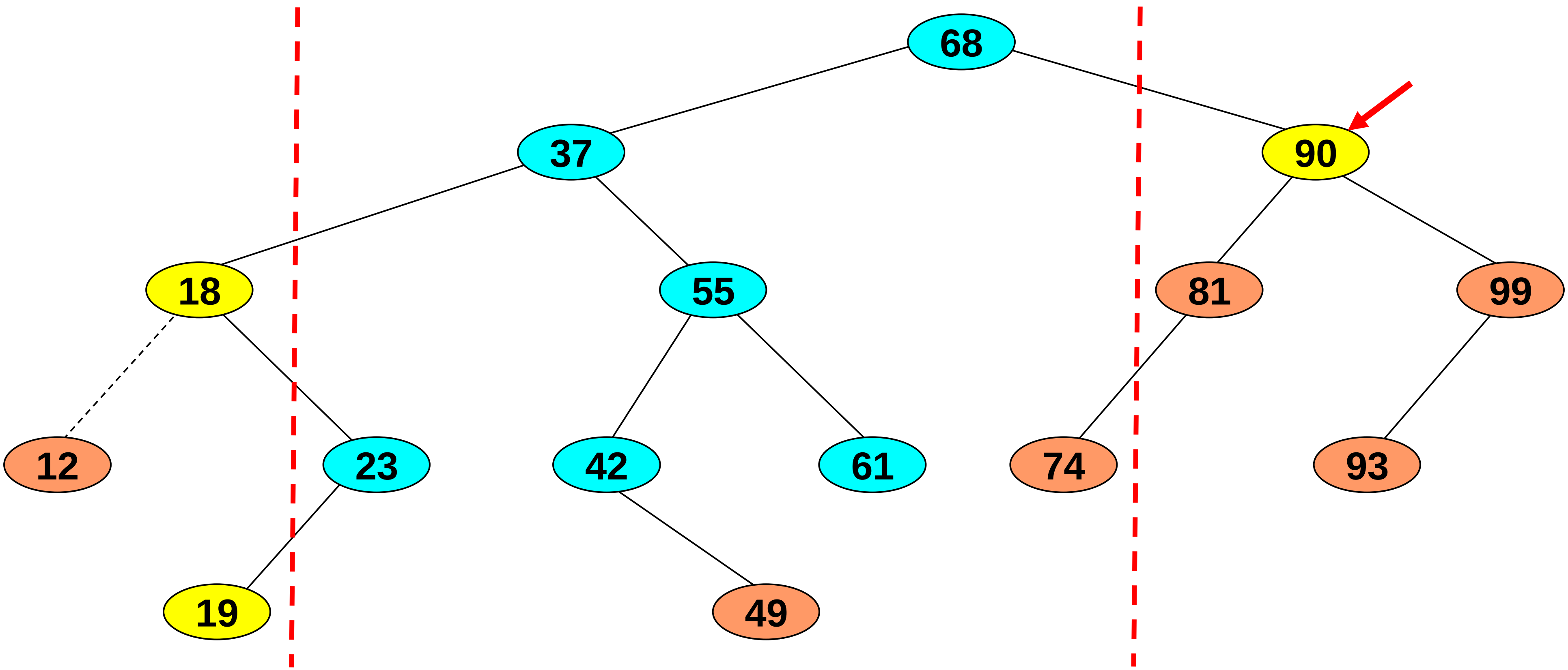
Example – AVL tree representing a set of 15 integers



Suppose we are searching for all items in the range 20..80

An example of a *one-dimensional range search query*

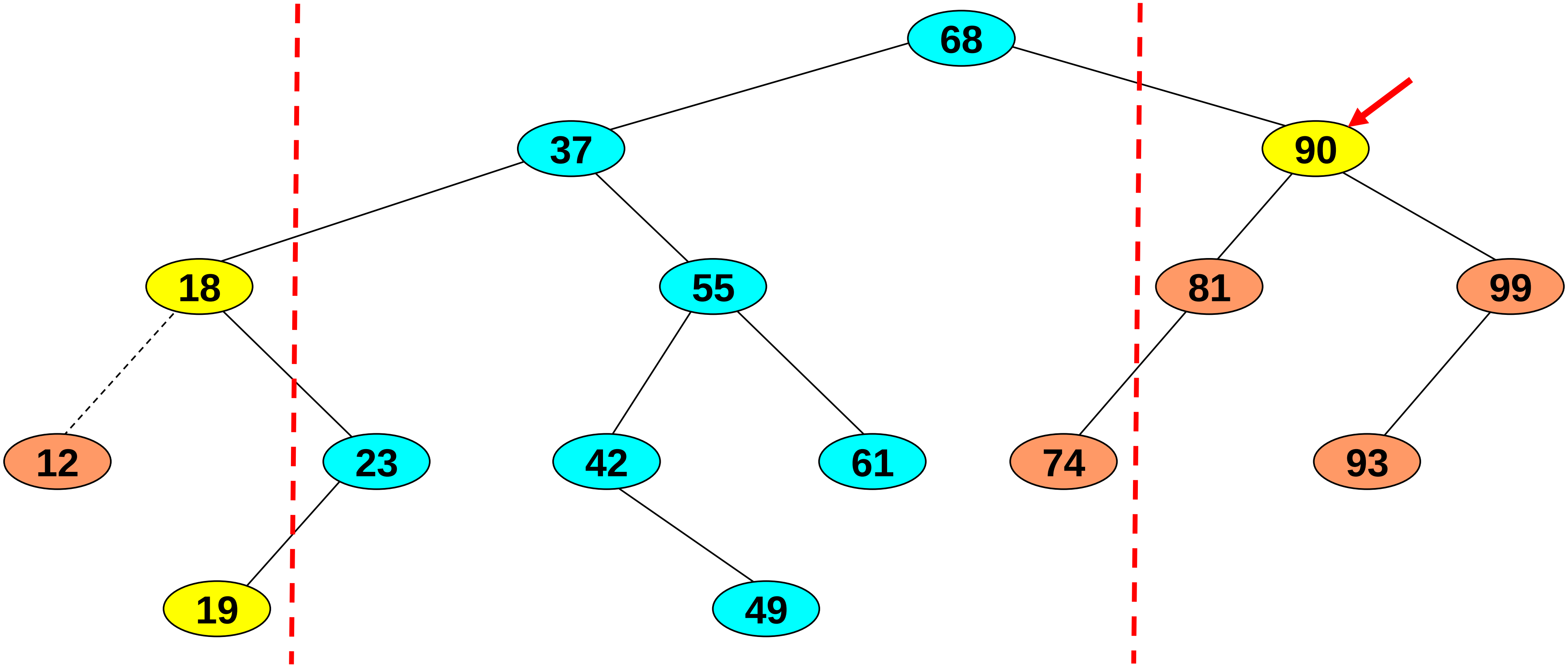
Example – AVL tree representing a set of 15 integers



Suppose we are searching for all items in the range 20..80

An example of a *one-dimensional range search query*

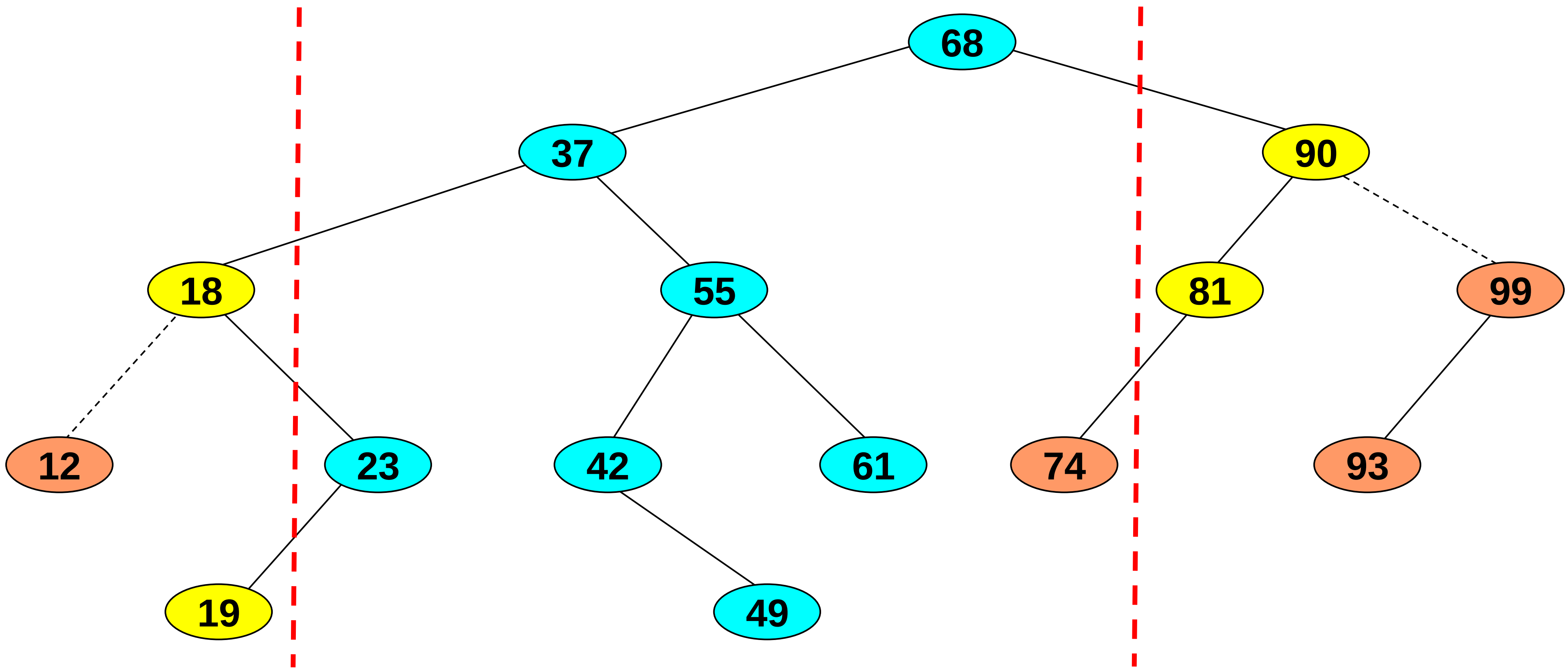
Example – AVL tree representing a set of 15 integers



Suppose we are searching for all items in the range 20..80

An example of a *one-dimensional range search query*

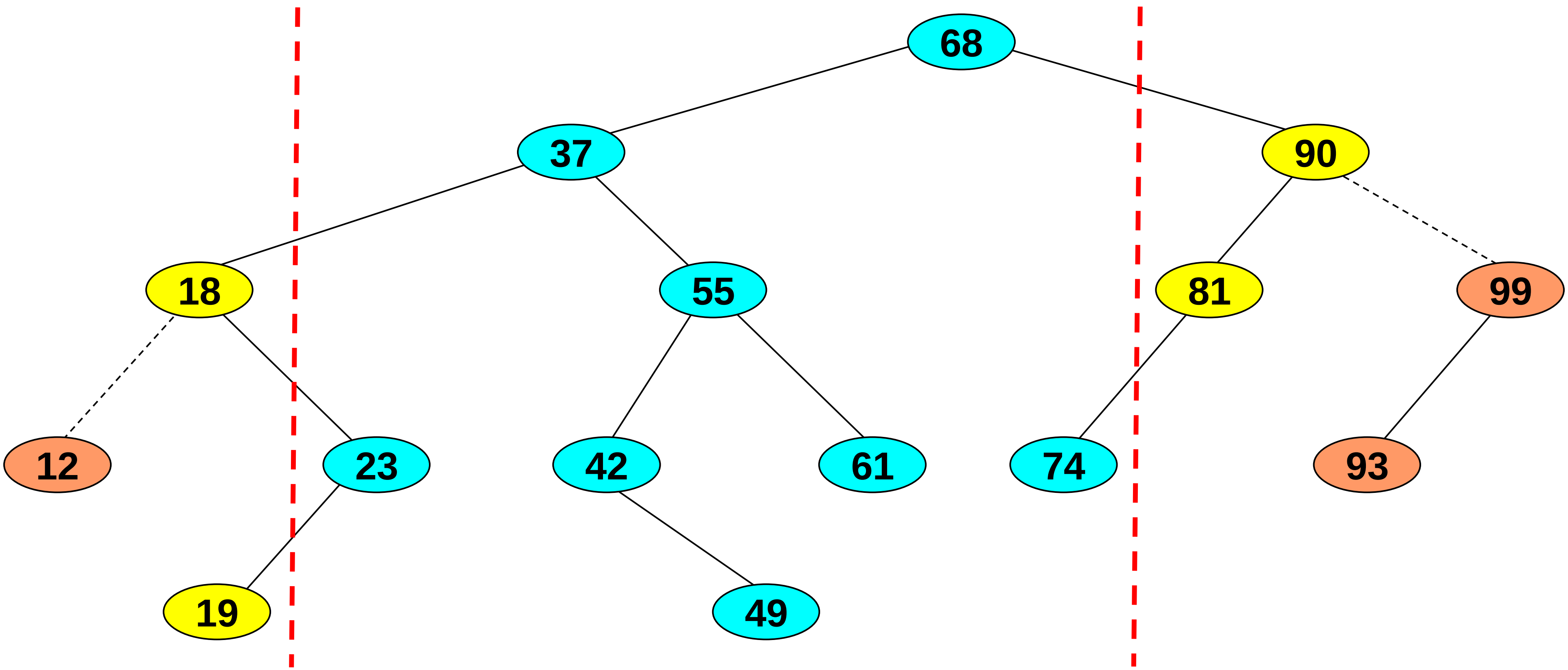
Example – AVL tree representing a set of 15 integers



Suppose we are searching for all items in the range 20..80

An example of a *one-dimensional range search query*

Example – AVL tree representing a set of 15 integers



Suppose we are searching for all items in the range 20..80

An example of a *one-dimensional range search query*

Algorithm for range searching

- Assume that we are searching for all elements in the range **min . . max**
- Partial traverse of tree: at a given node
  - if value < min ignore left branch but explore right branch (if any)
  - if value > max ignore right branch but explore left branch (if any)
  - otherwise include value and explore both branches

Analysis of the algorithm

- Let k be the number of elements of S in the range min . . max and let s = |S|
- Algorithm is O(k + log s)
  - k nodes visited with value in range
  - visit at most 1 node to the left of range, and 1 node to the right of range, at each level, O(log s) levels

Useful data structures

```
/** Class representing a binary tree where each node
  * has a value and a possible left subtree and a
  * possible right subtree
  */
public class Tree {

   public int val;

   public Tree leftSubtree; // null if no left subtree

   public Tree rightSubtree; // null if no right subtree
}

/** Class representing a set of integers */
public class IntSet {

   public ArrayList<int> intSet;
}
```

Range search algorithm for AVL trees

```
/** Input: an AVL tree t containing integer values, and
  * minimum and maximum range values min and max
  * Output: the set of values in t between min and max */
public IntSet rangeSearch (Tree t, int min, int max){
  if (t==null)
    return ∅;
  else
  { int y = t.val;
    Tree lt = t.leftSubtree;
    Tree rt = t.rightSubtree;
    if (y ≥ min && y ≤ max)
      return rangeSearch(lt,min,max) ∪ {y}
                                    ∪ rangeSearch(rt,min,max);
    else if (y < min)
      return rangeSearch(rt,min,max);
    else   // max < y
      return rangeSearch(lt,min,max);
  }
}
```

Summary of the line-sweep algorithm

Assume that h_e (v_e) denotes the horizontal (vertical) line with endpoint e

```
form list E of endpoints of all lines;
sort E on x-coordinate;
create empty AVL-tree C of candidates;
for (Point2D.Double e : E)
  if (e is from a horizontal line)
    if (e is a left endpoint)
      add e to C;
    else  // e is a right endpoint
      remove left endpoint of h_e from C;
  else    // e is from a vertical line
  { let y_1,y_2 be y-coords of v_e endpoints;
    range search C for y_1..y_2;
    output intersection (v_e,h_f) for all
               horizontal lines h_f in the range;
  }
```

Analysis of the line-sweep algorithm

- sorting endpoints is $O(n \log n)$
- the sweep encounters $2h + v$ points
- at each 'horizontal' point, we insert in or delete from the tree, contributing $2h \times O(\log h)$ to the complexity
- at 'vertical' point i, suppose there are n_i intersections with candidates – total time taken to process vertical points is

$$O(n_1 + \log h + n_2 + \log h + \ldots + n_v + \log h) = O(p + v \log h)$$

Overall complexity

$$O(n \log n + h \log h + v \log h + p) = O(n \log n + p)$$

since $h = O(n)$ and $v = O(n)$.