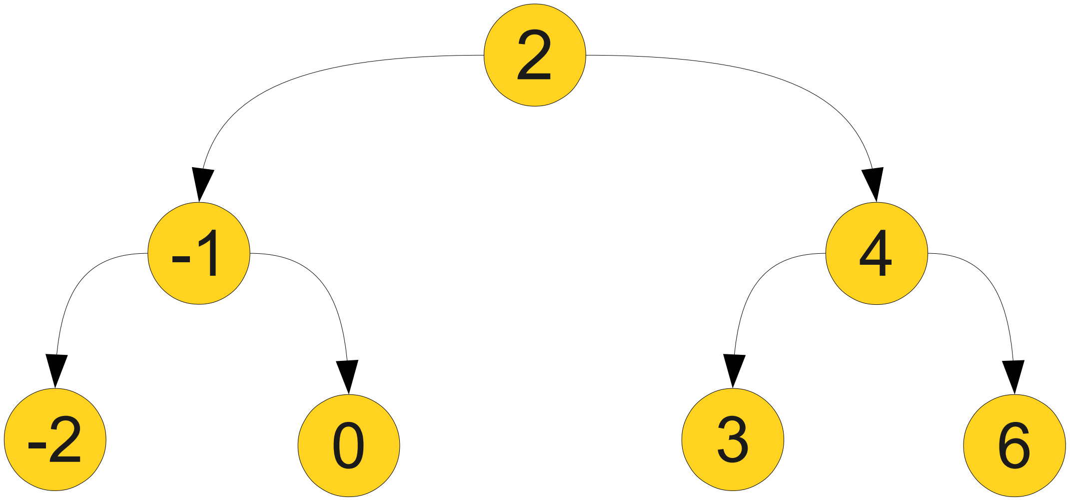
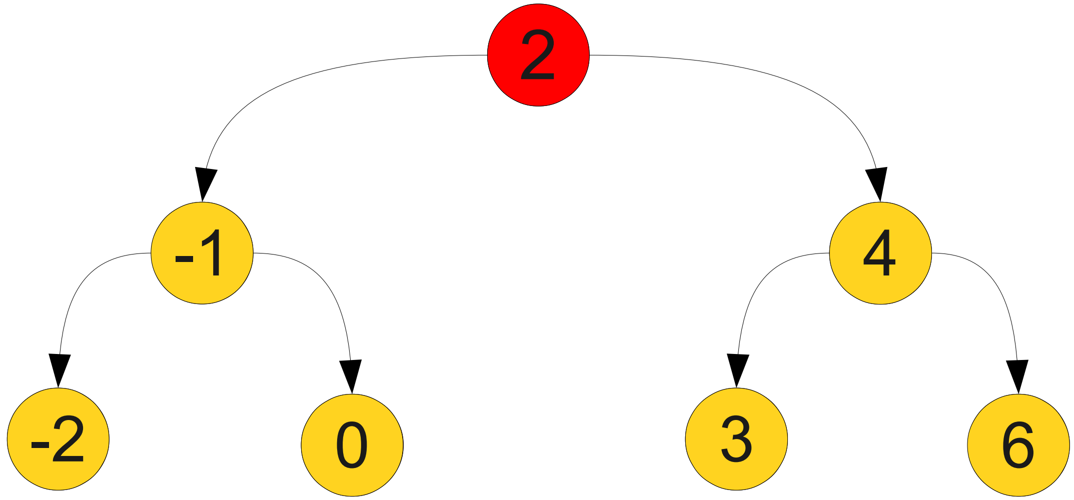
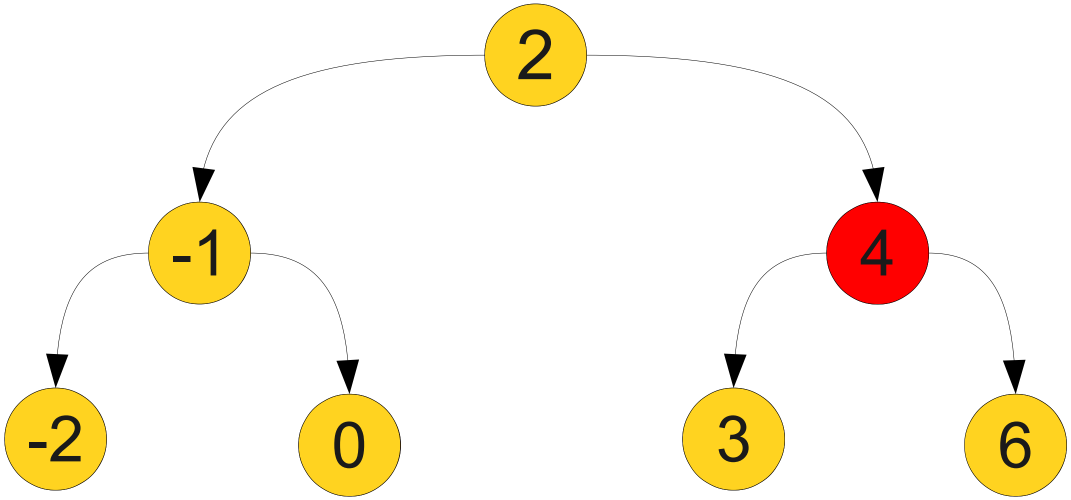
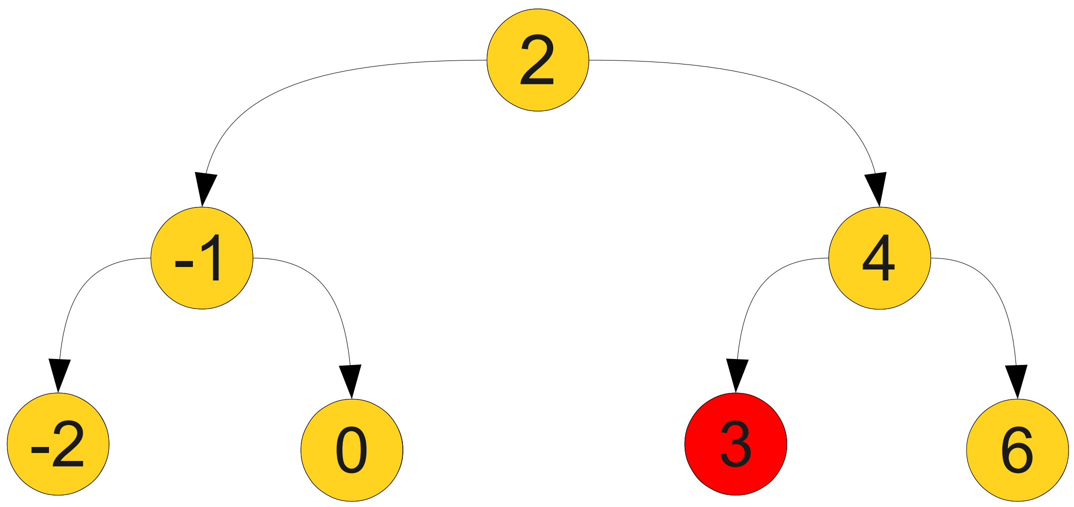
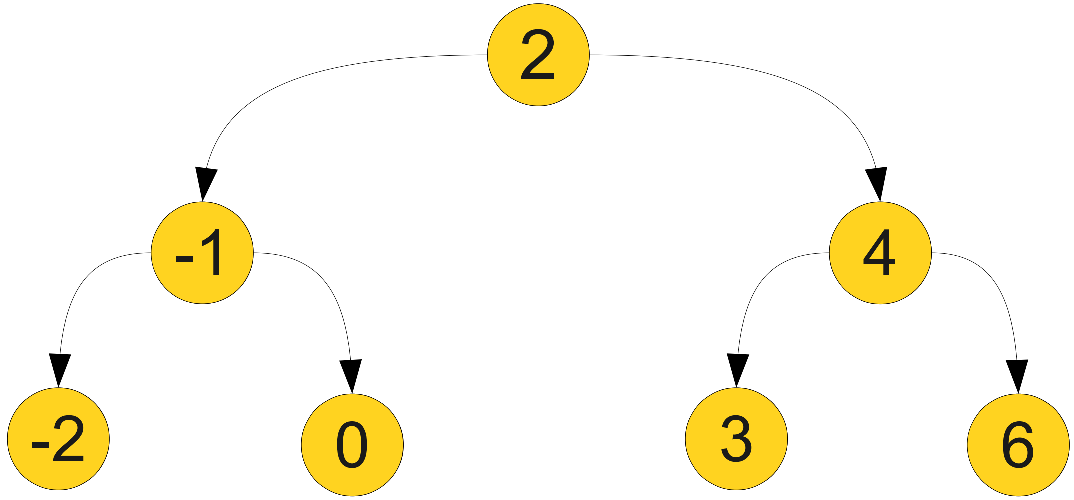
Advanced Data Structures

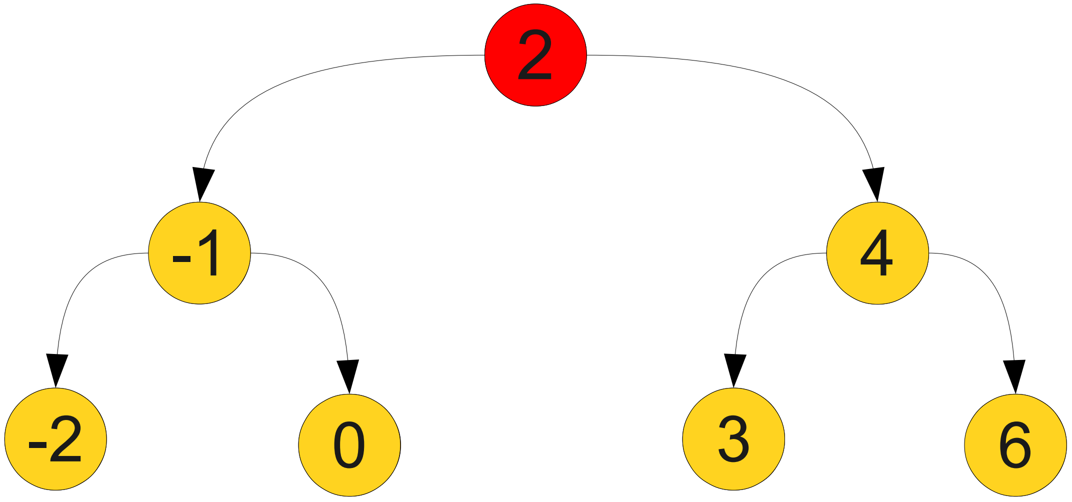

2
-1
4
-2
0
3
6

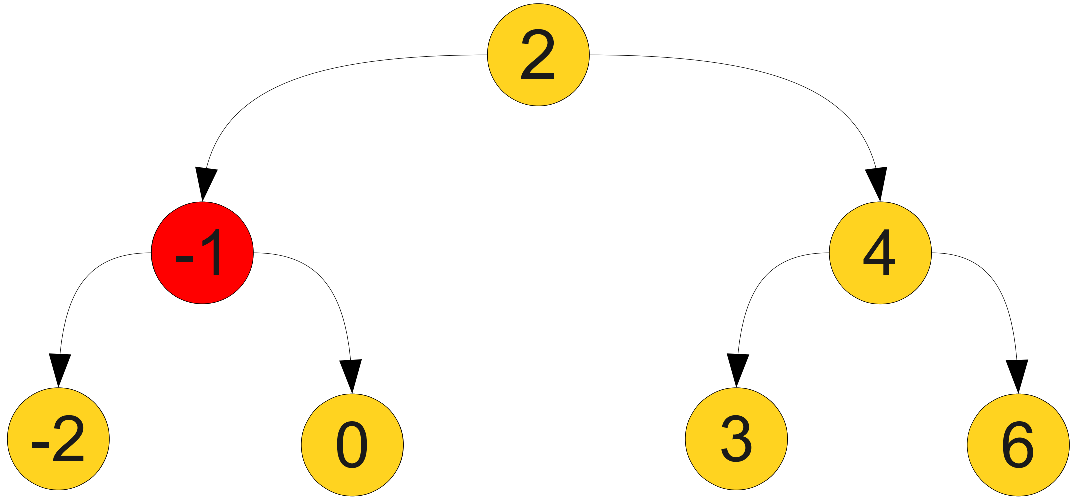

2
-1
4
-2
0
3
6


2
-1
4
-2
0
3
6

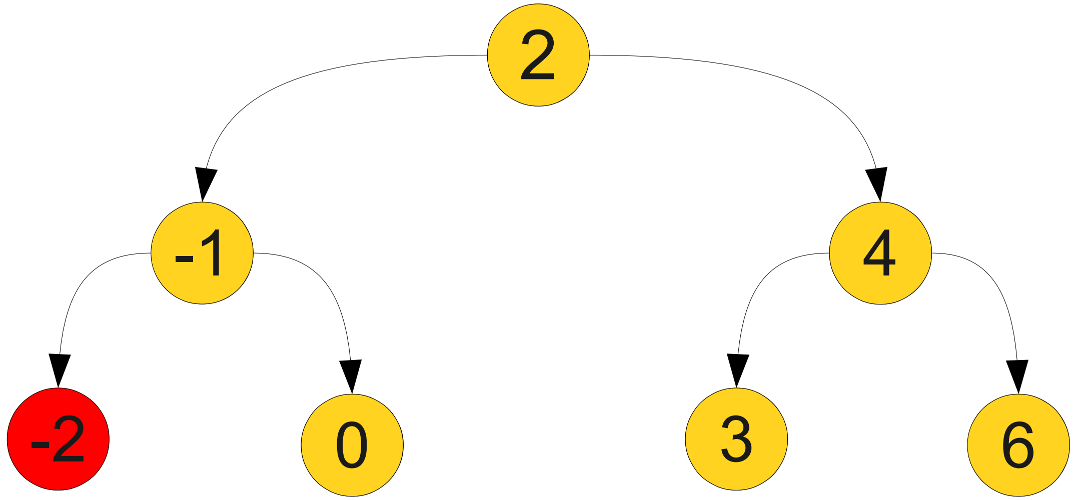

2
-1
4
-2
0
3
6

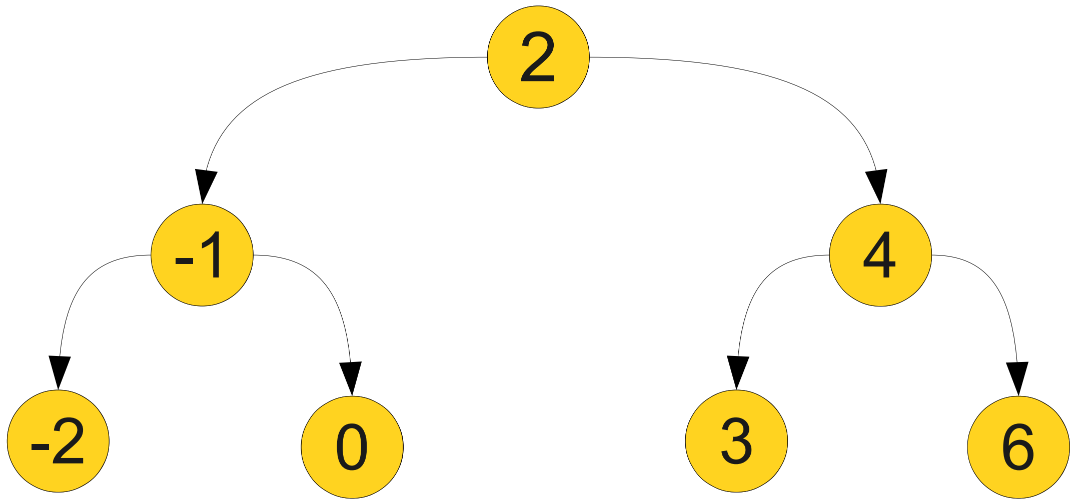

2
-1
4
-2
0
3
6


2
-1
4
-2
0
3
6



2
-1
4
-2
0
3
6

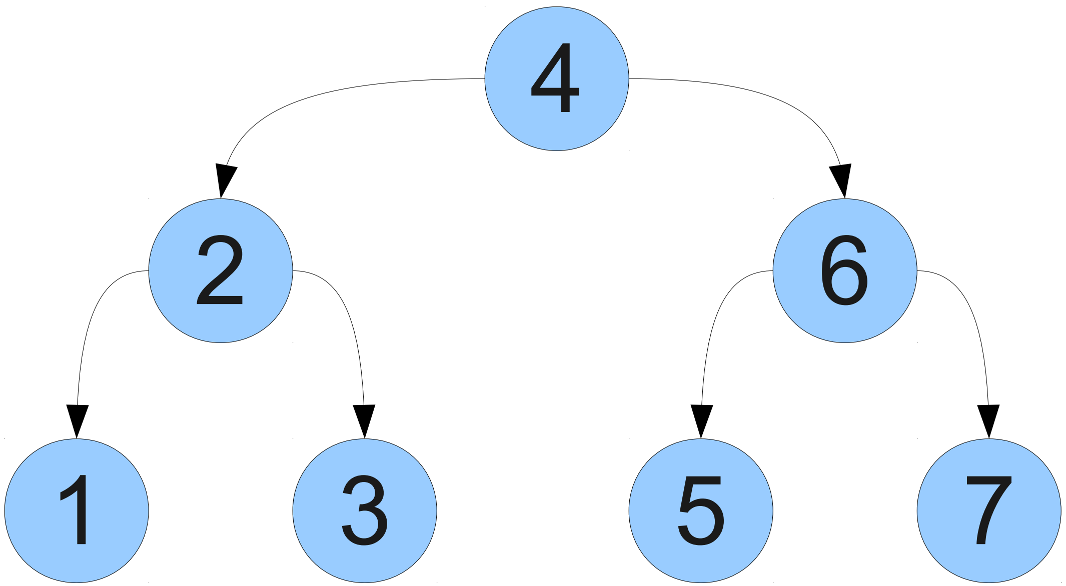

2
-1
4
-2
0
3
6


2
-1
4
-2
0
3
6

Insertion Order Matters

- Suppose we create a BST of numbers in this order:

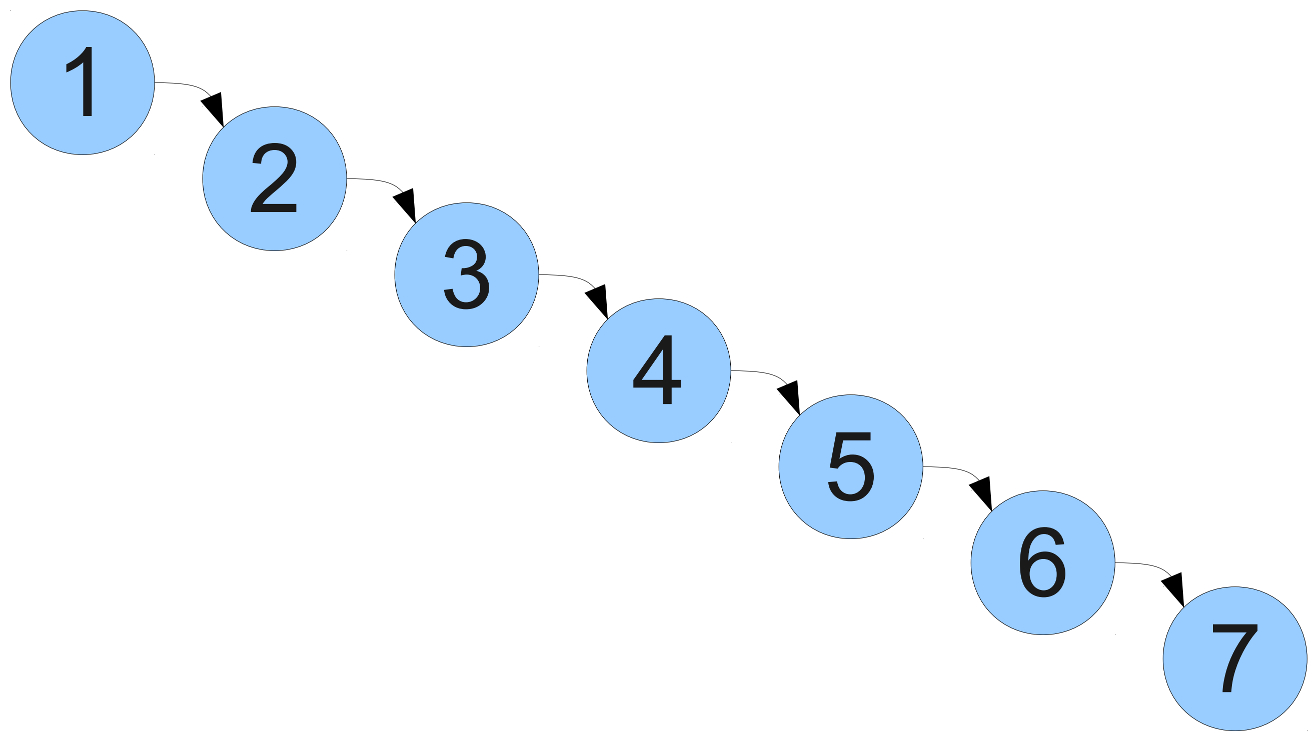
4, 2, 1, 3, 6, 5, 7



Insertion Order Matters

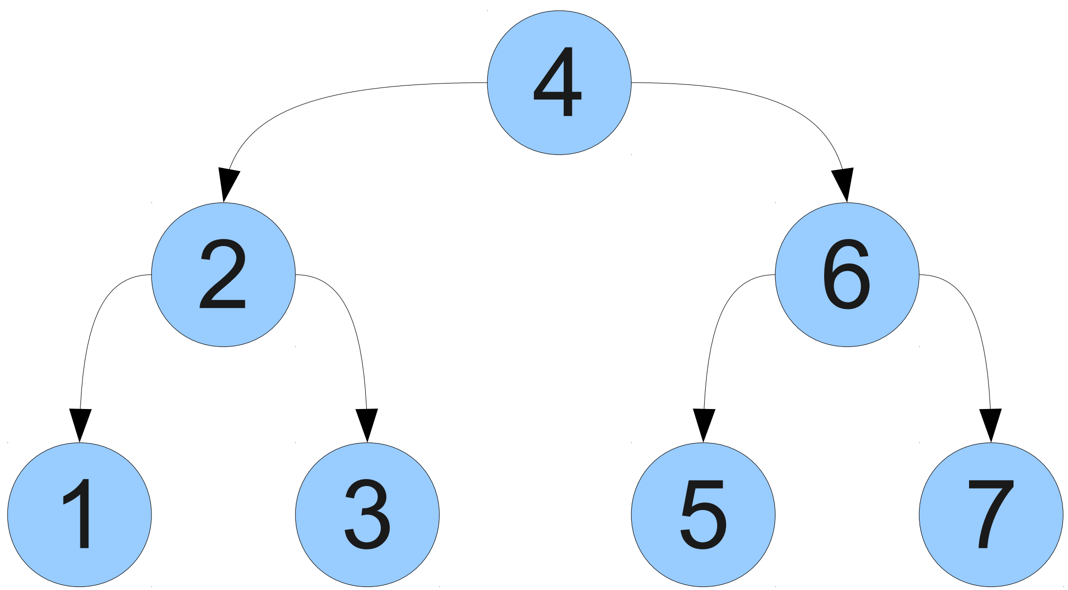
- Suppose we create a BST of numbers in this order:

$$1, 2, 3, 4, 5, 6, 7$$



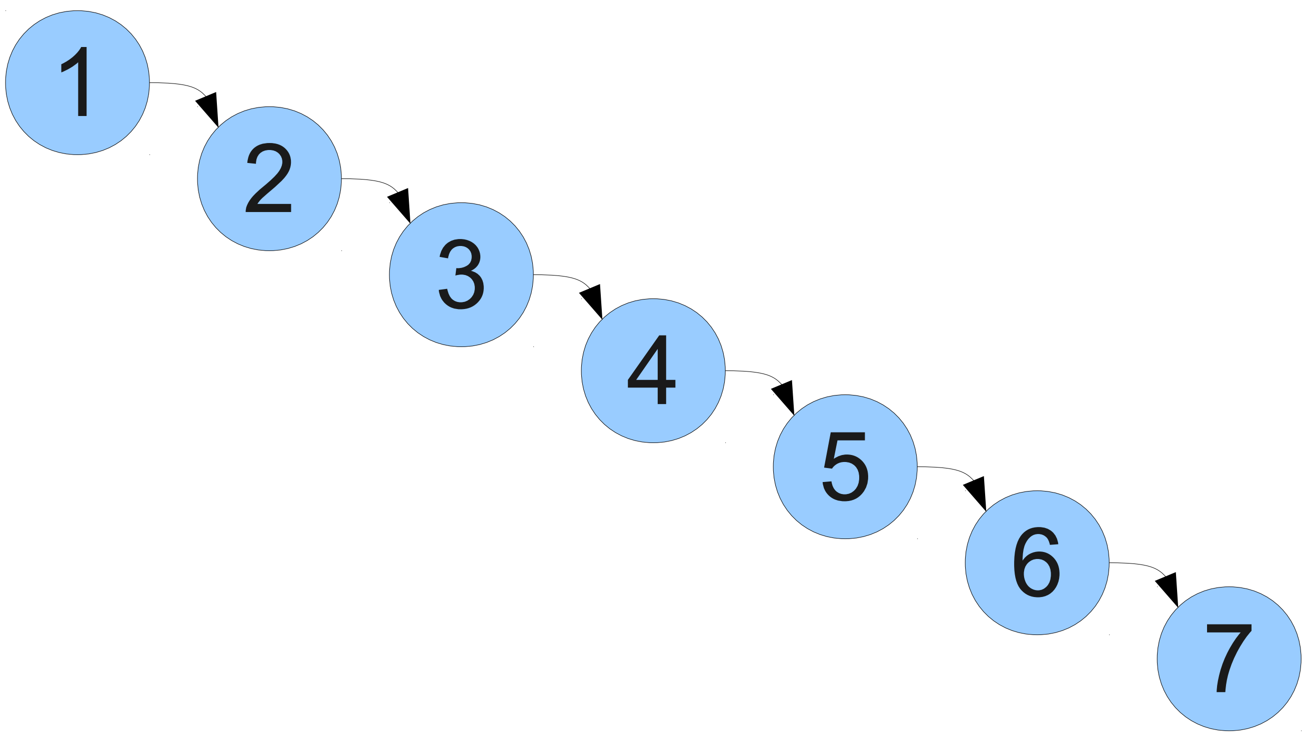
Tree Terminology

- The **height** of a tree is the number of nodes in the longest path from the root to a leaf.



Tree Terminology

- The **height** of a tree is the number of nodes in the longest path from the root to a leaf.



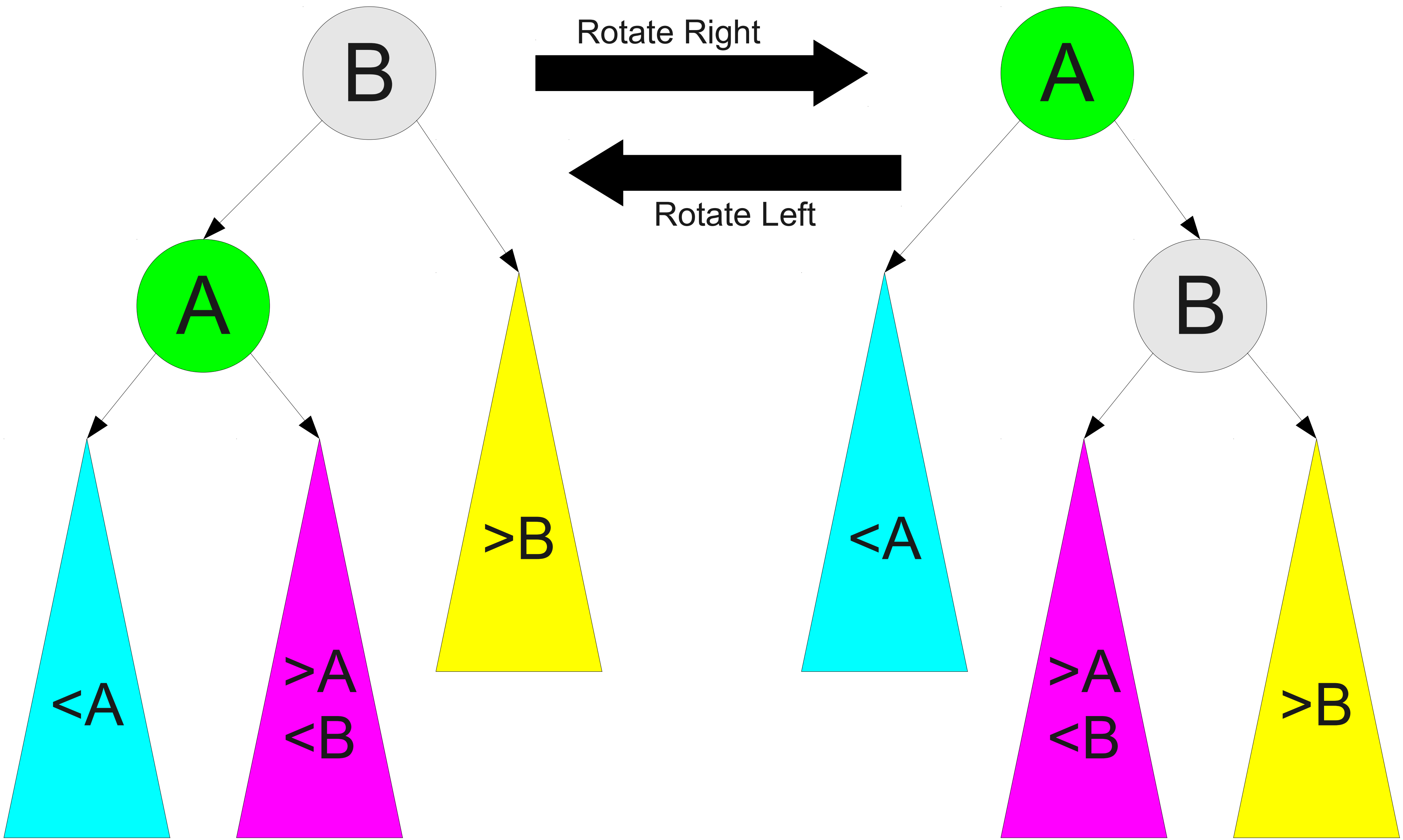
Keeping the Height Low

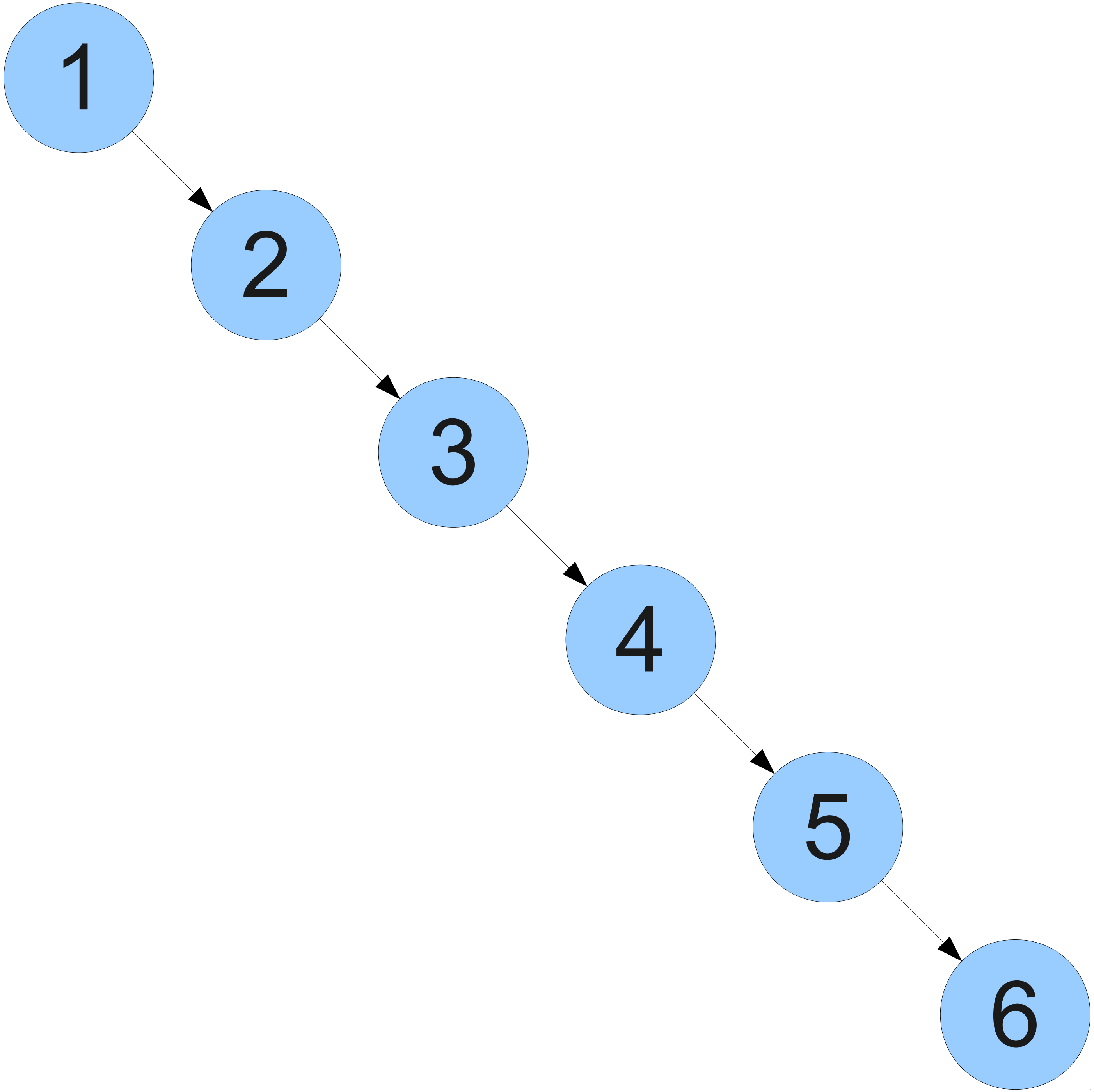
- Almost all BST operations have time complexity based on height:
  - Insertion: $O(h)$
  - Search: $O(h)$
  - Deletion: $O(h)$
- Keeping the height low will make these operations much more efficient.
- How do we do this?

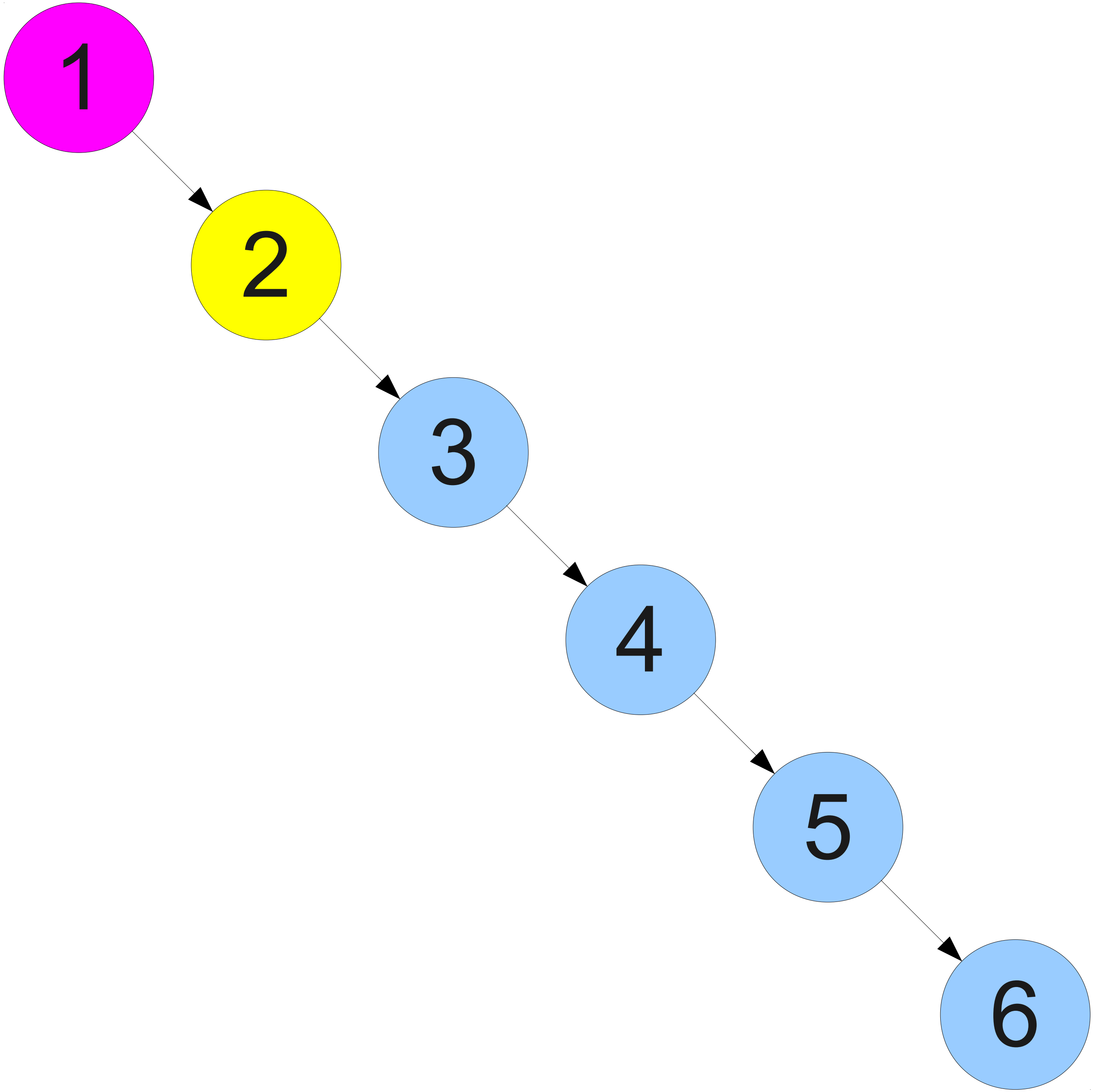
Tree Rotations

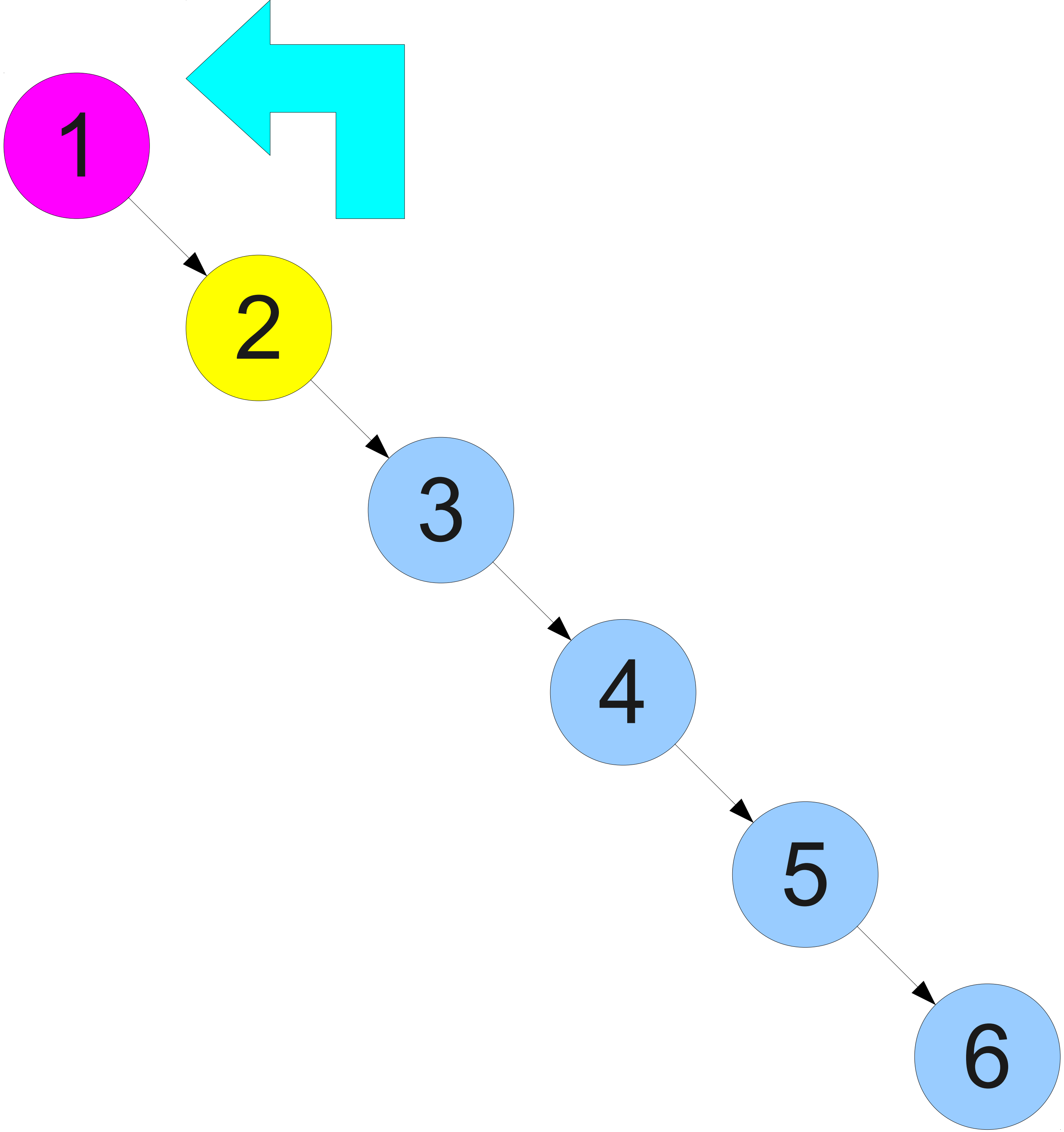
- One common way of keeping tree heights low is to reshape the BST when it gets too high.
- One way to accomplish this is a **tree rotation**, which locally rearranges nodes.

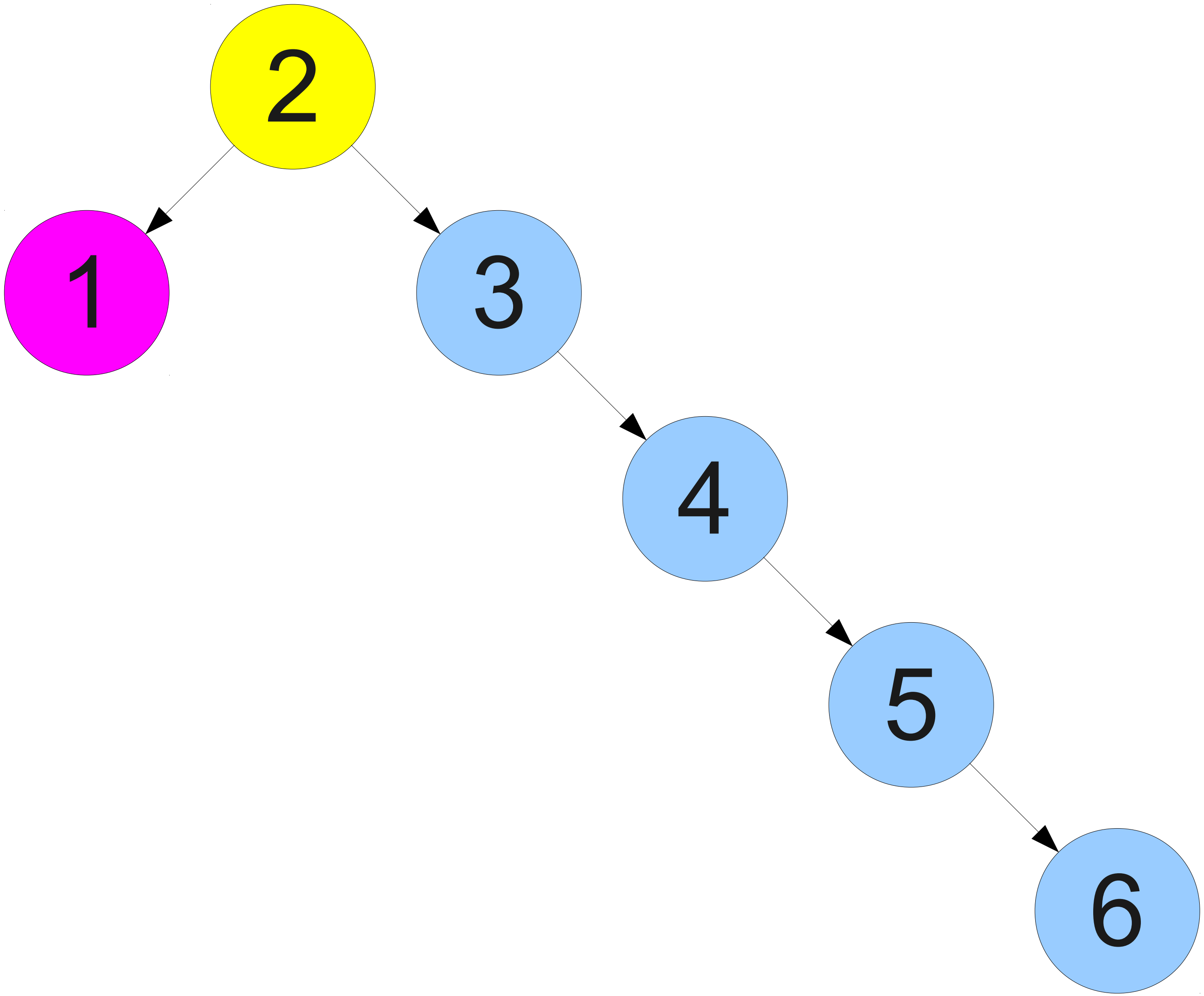
Tree Rotations

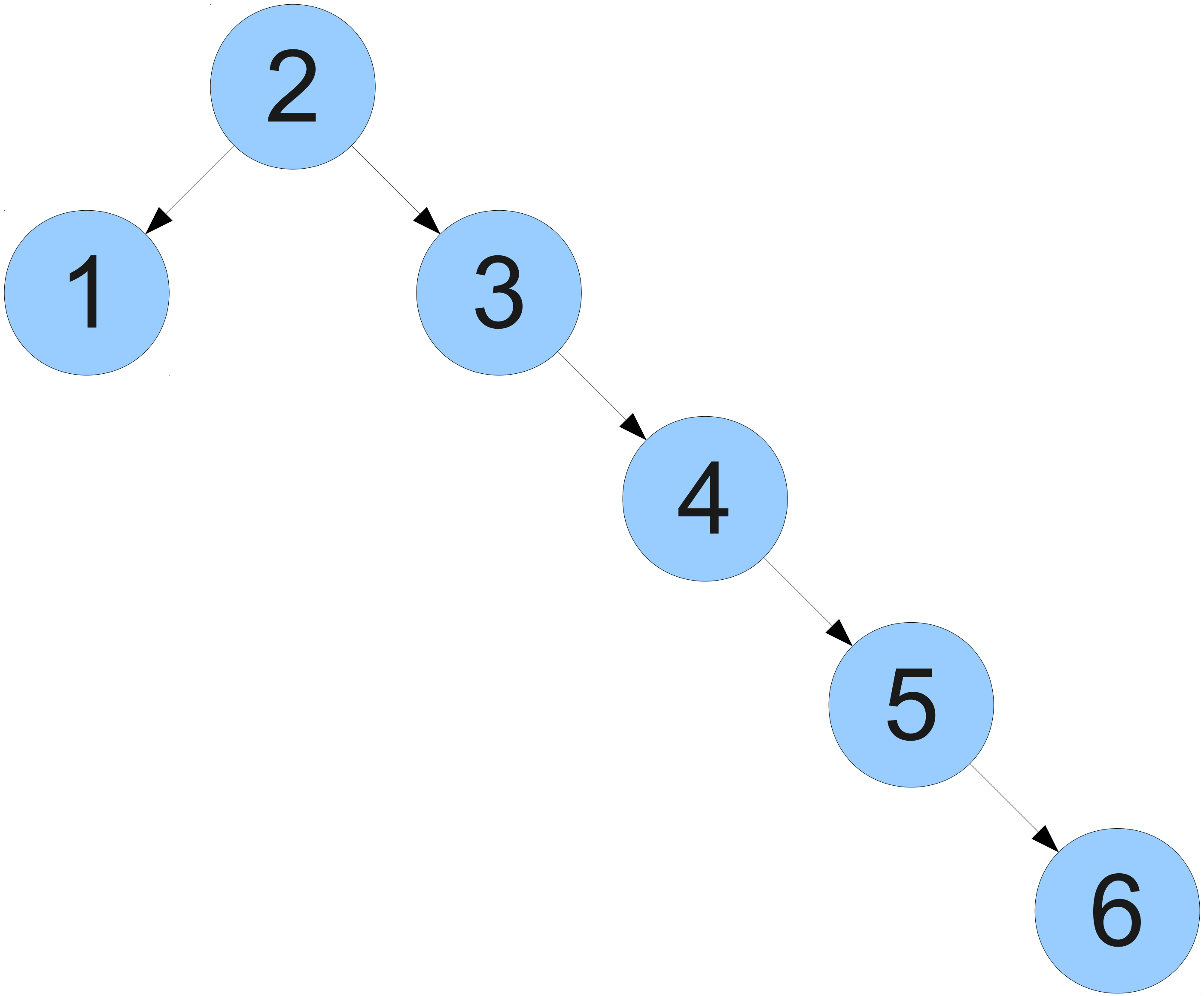


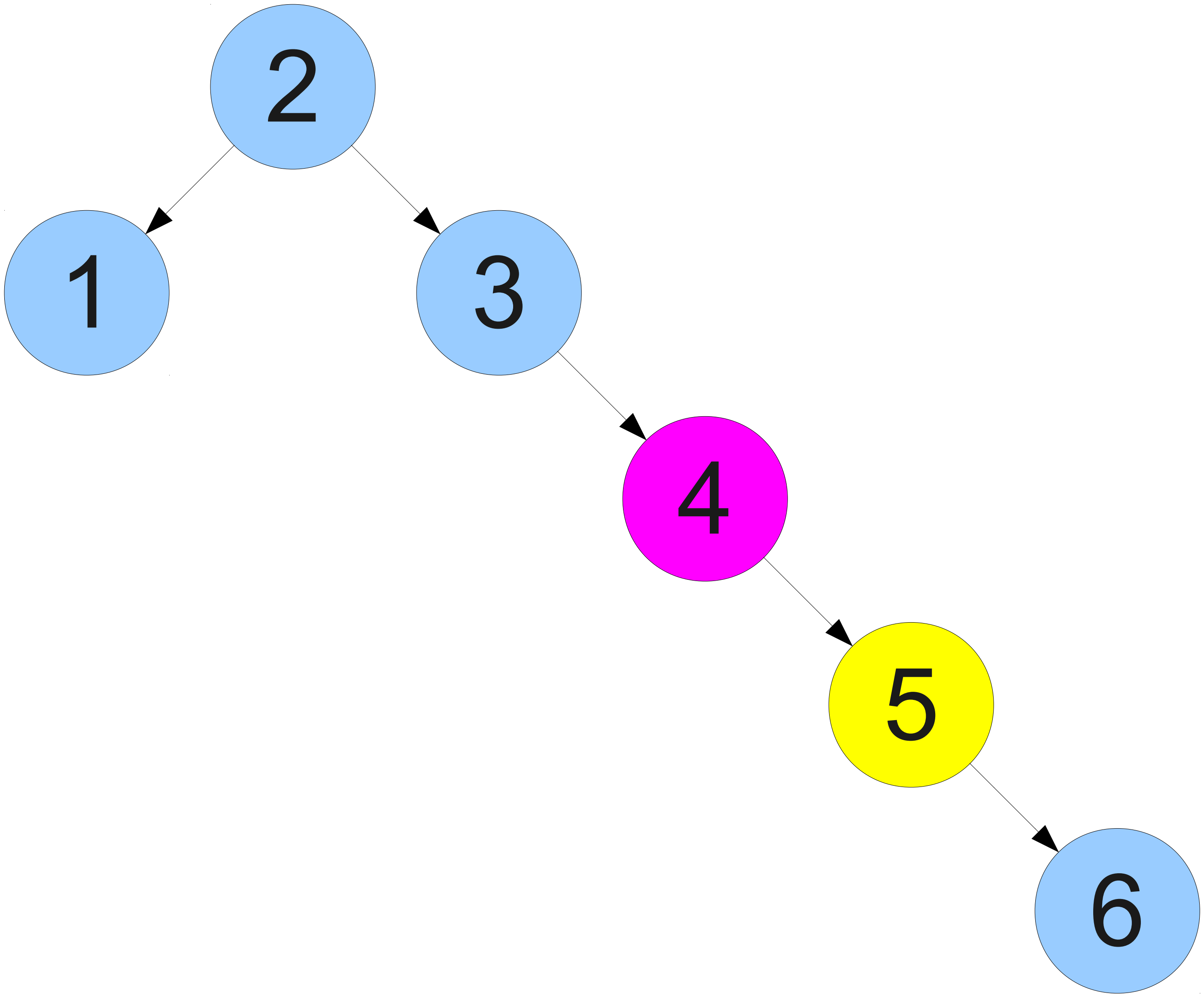

1
2
3
4
5
6


1
2
3
4
5
6

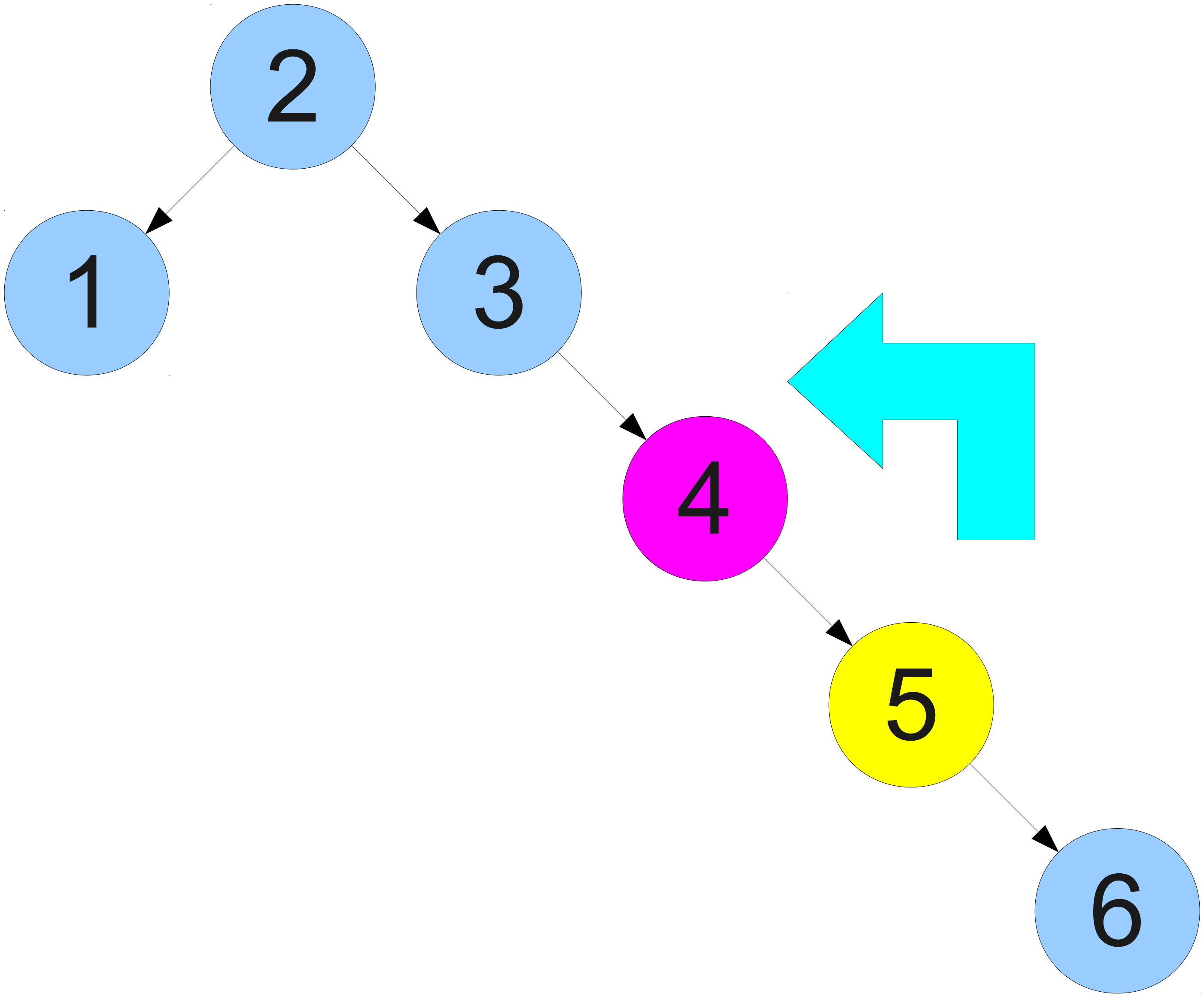

1
2
3
4
5
6


2
1
3
4
5
6

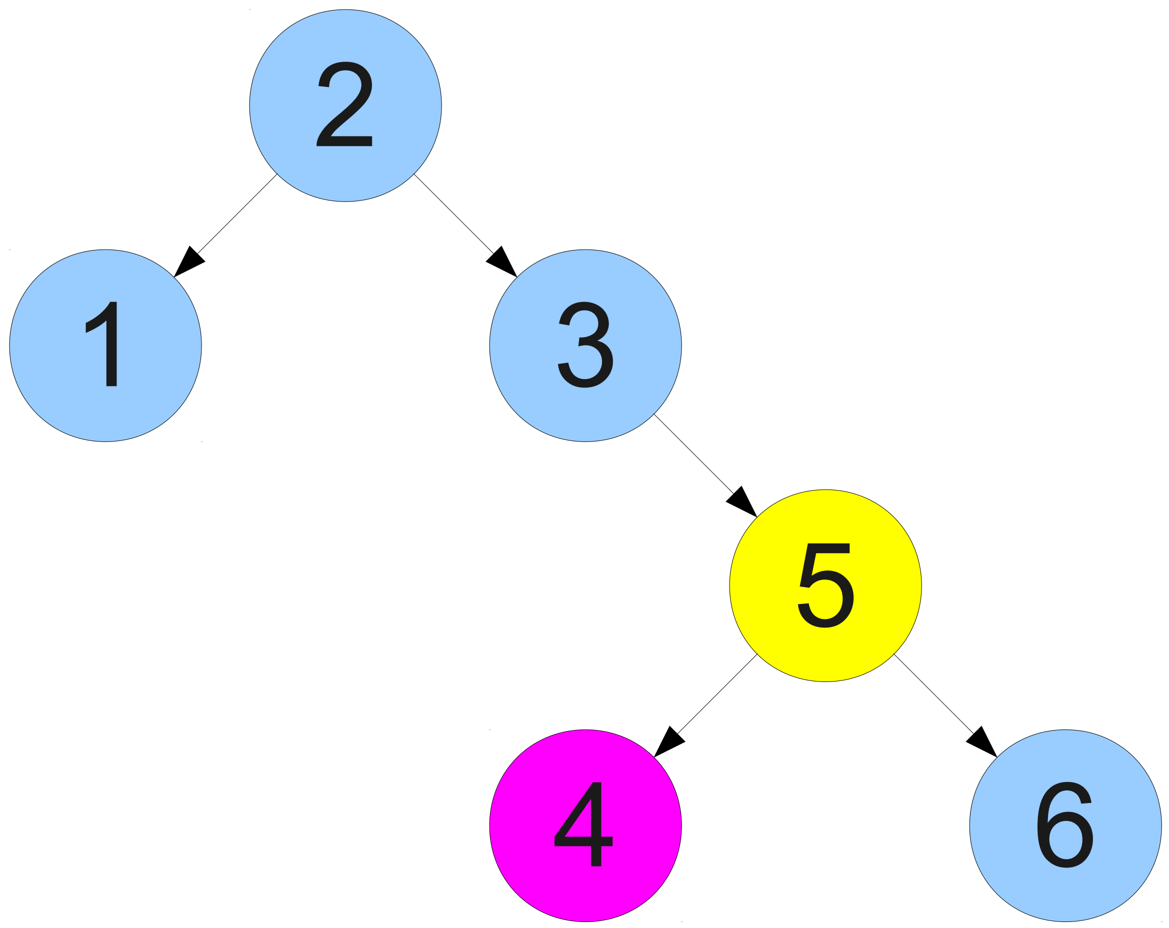

2
1
3
4
5
6

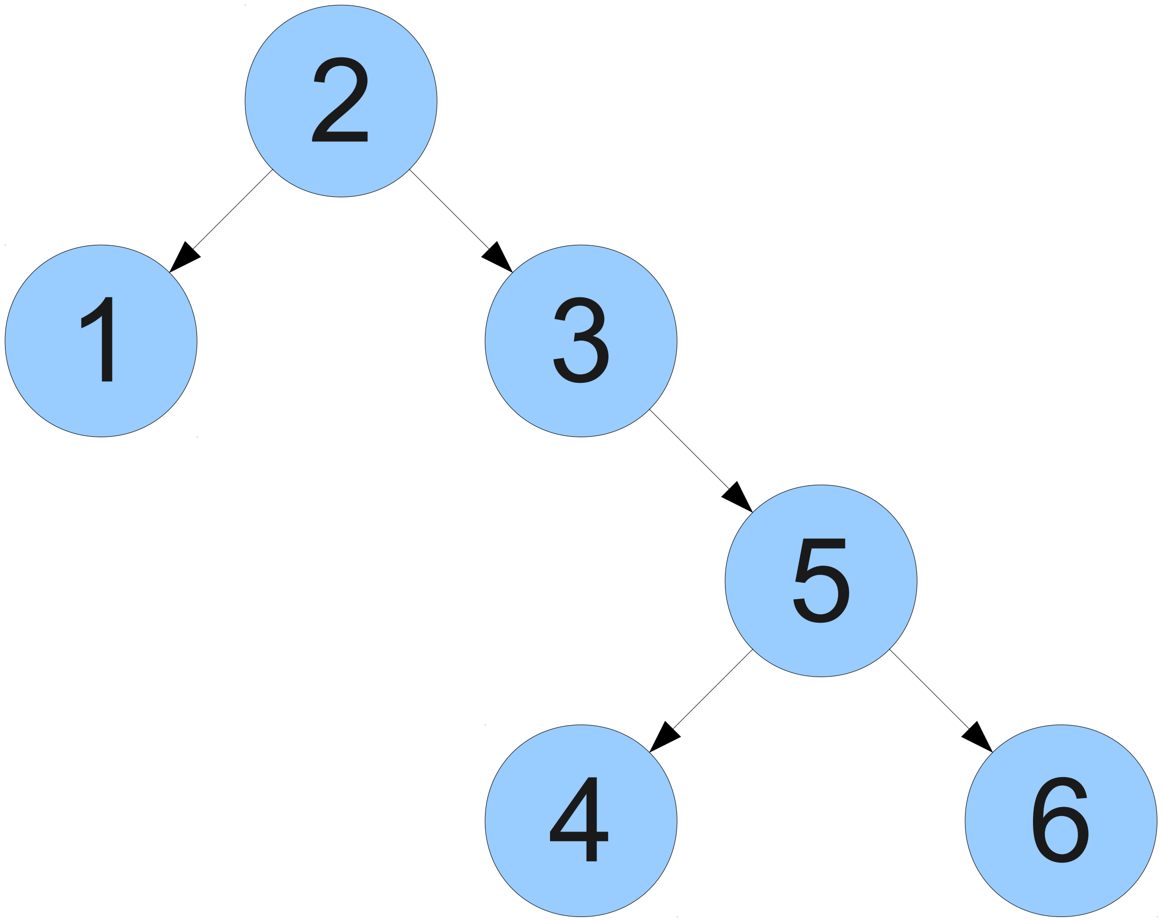


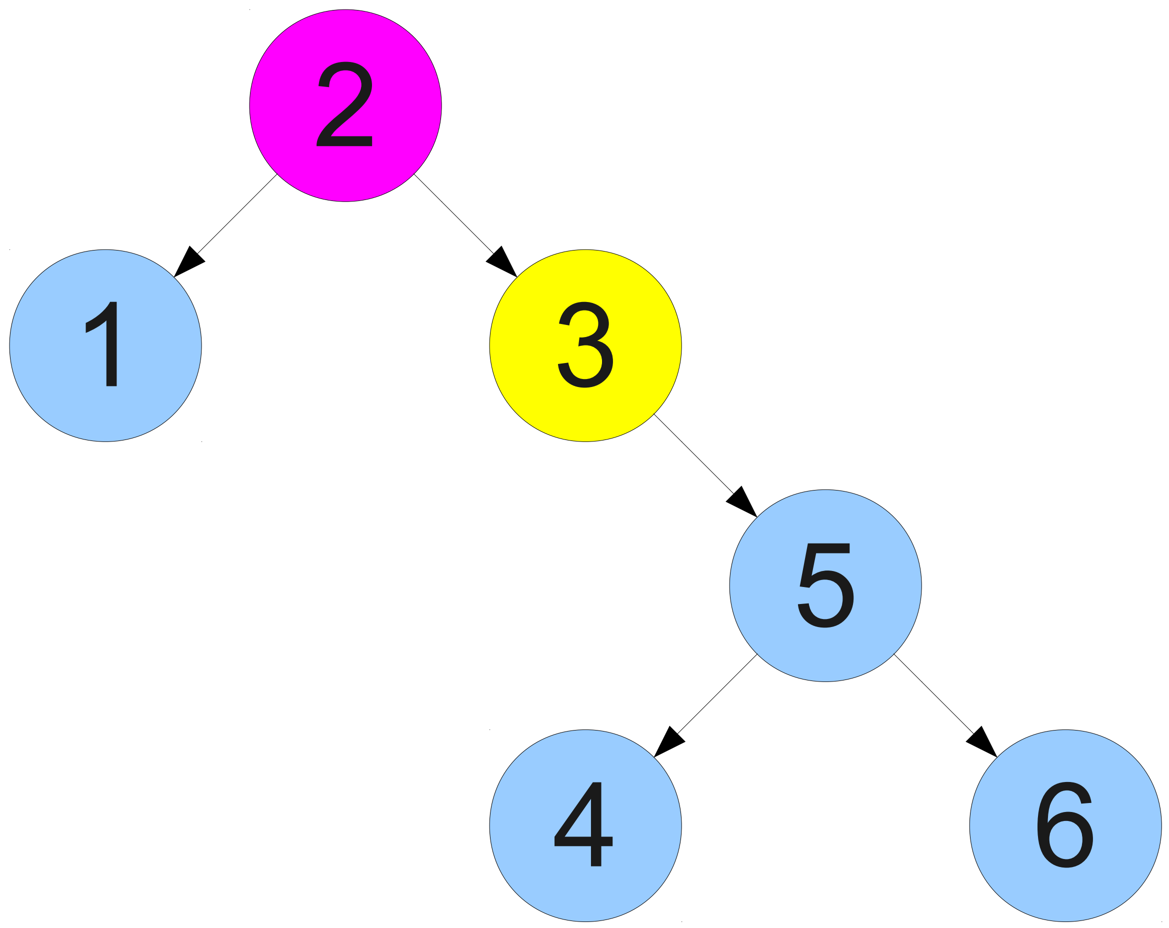
2
1
3
4
5
6

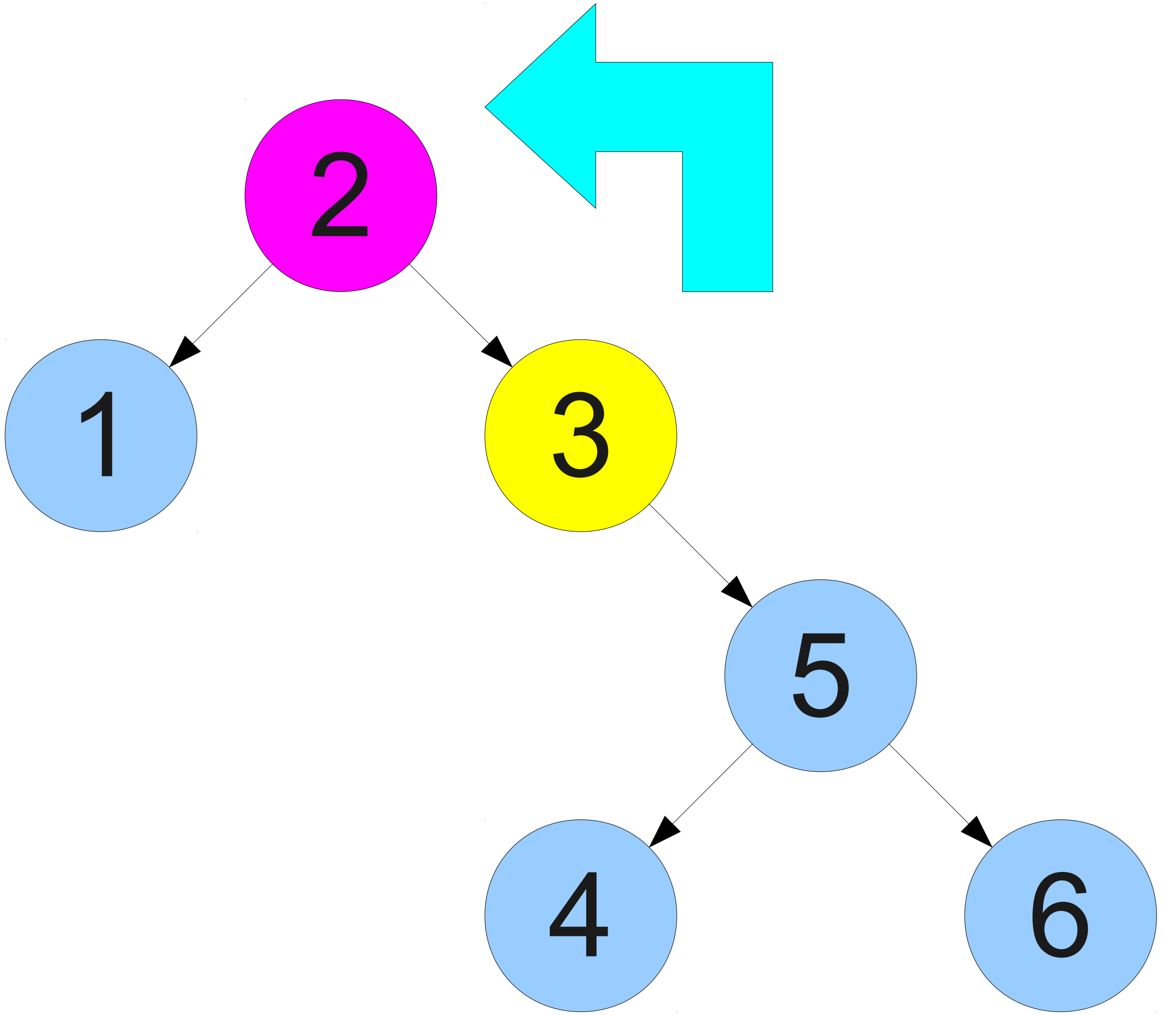


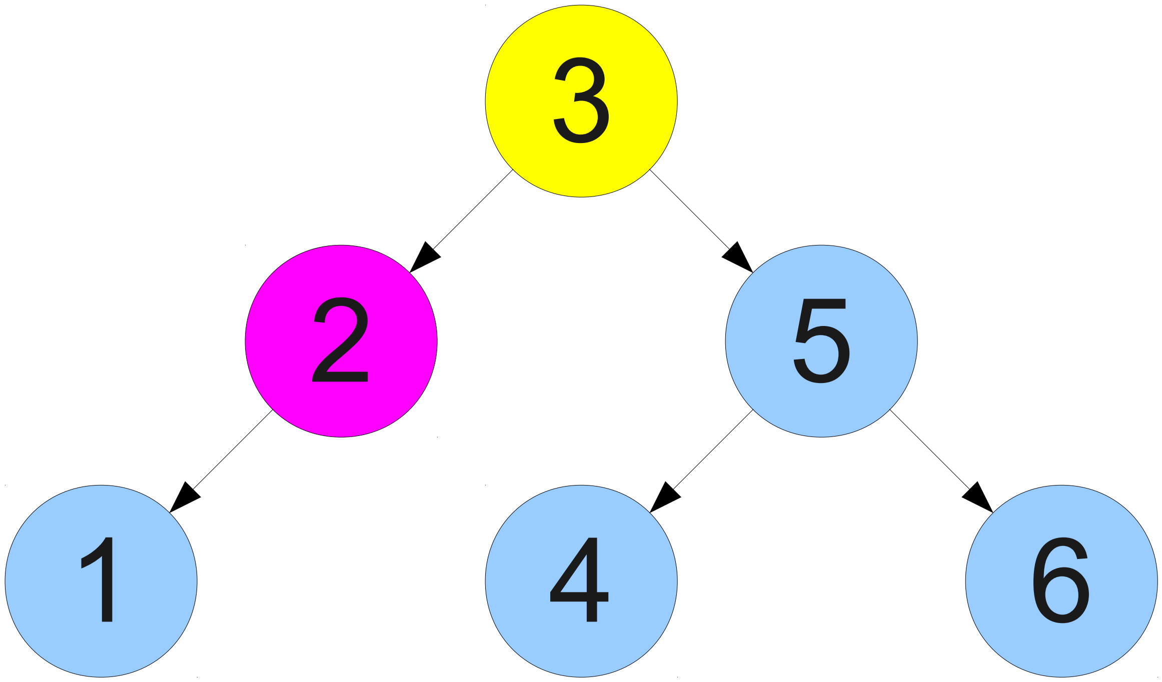
2
1
3
4
5
6


2
1
3
5
4
6


2
1
3
5
4
6


2
1
3
5
4
6


2
1
3
5
4
6


3
2
5
1
4
6

Let's Code it Up!

When to Rotate?

- The actual code for rotations is not too complex.
- Deciding when and where to rotate the tree, on the other hand, is a bit involved.
- There are many schemes we can use to determine this:
  - AVL trees maintain balance information in each node, then rotate when the balance is off.
  - Red/Black trees assign each node a color, then rotate when certain color combinations occur.

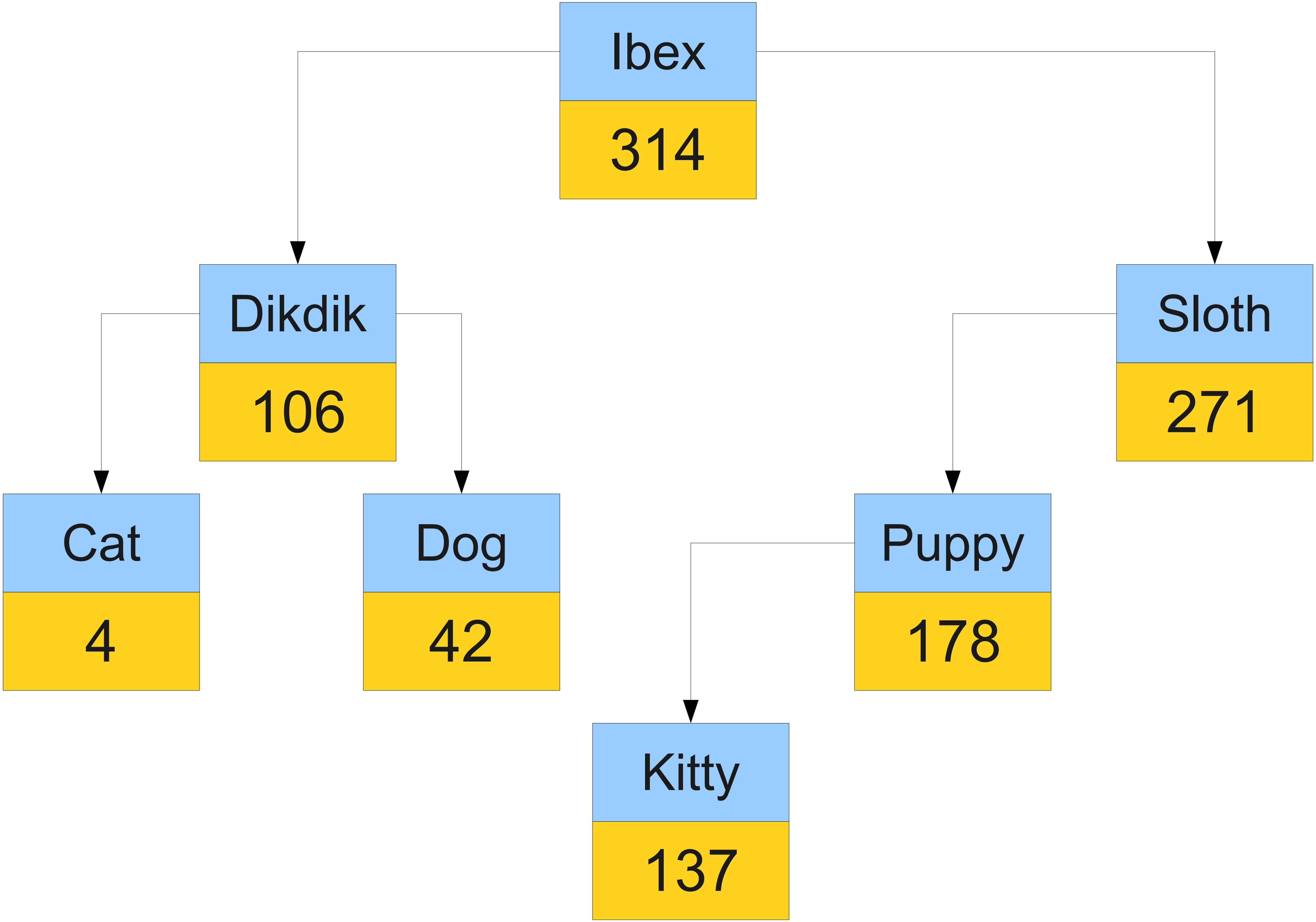
An Interesting Observation

Random Binary Search Trees

- If we build a binary search tree with totally random values, the resulting tree is (with high probability) within a constant factor of balanced.
  - Approximately 4.3 ln n
- Moreover, the *average* depth of a given node is often very low.
  - Approximately 2 ln n.
- If we structure the BST as if it were a random tree, we get (with high probability) a very good data structure!

Treaps

- A **treap** is a data structure that combines a binary search tree and a binary heap.
- Each node stores two pieces of information:
  - The piece of information that we actually want to store, and
  - A random real number.
- The tree is stored such that
  - The nodes are a binary search tree when looking up the information, and
  - The nodes are a binary heap with respect to the random real number.



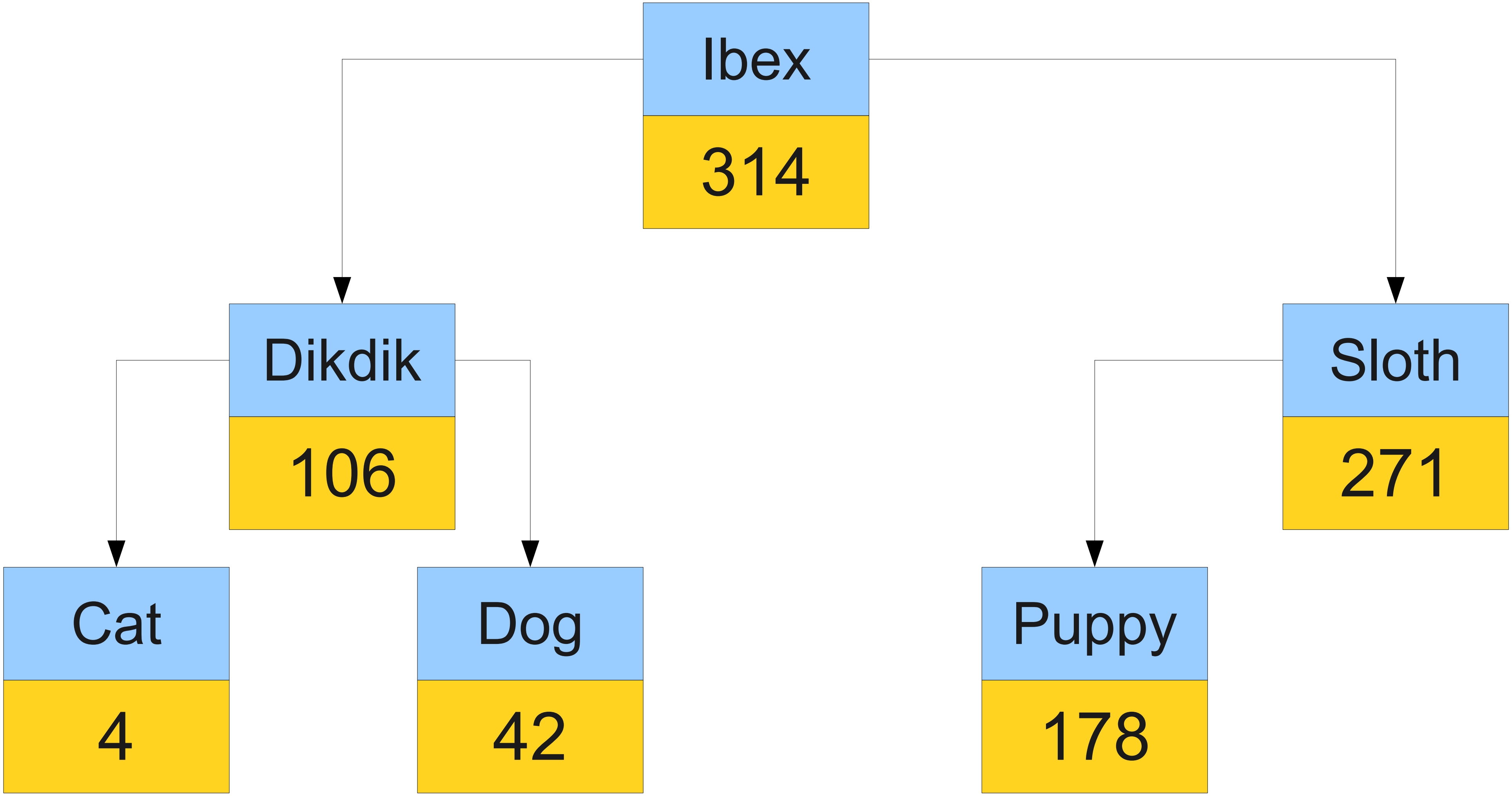
Ibex
314
Dikdik
106
Sloth
271
Cat
4
Dog
42
Puppy
178
Kitty
137

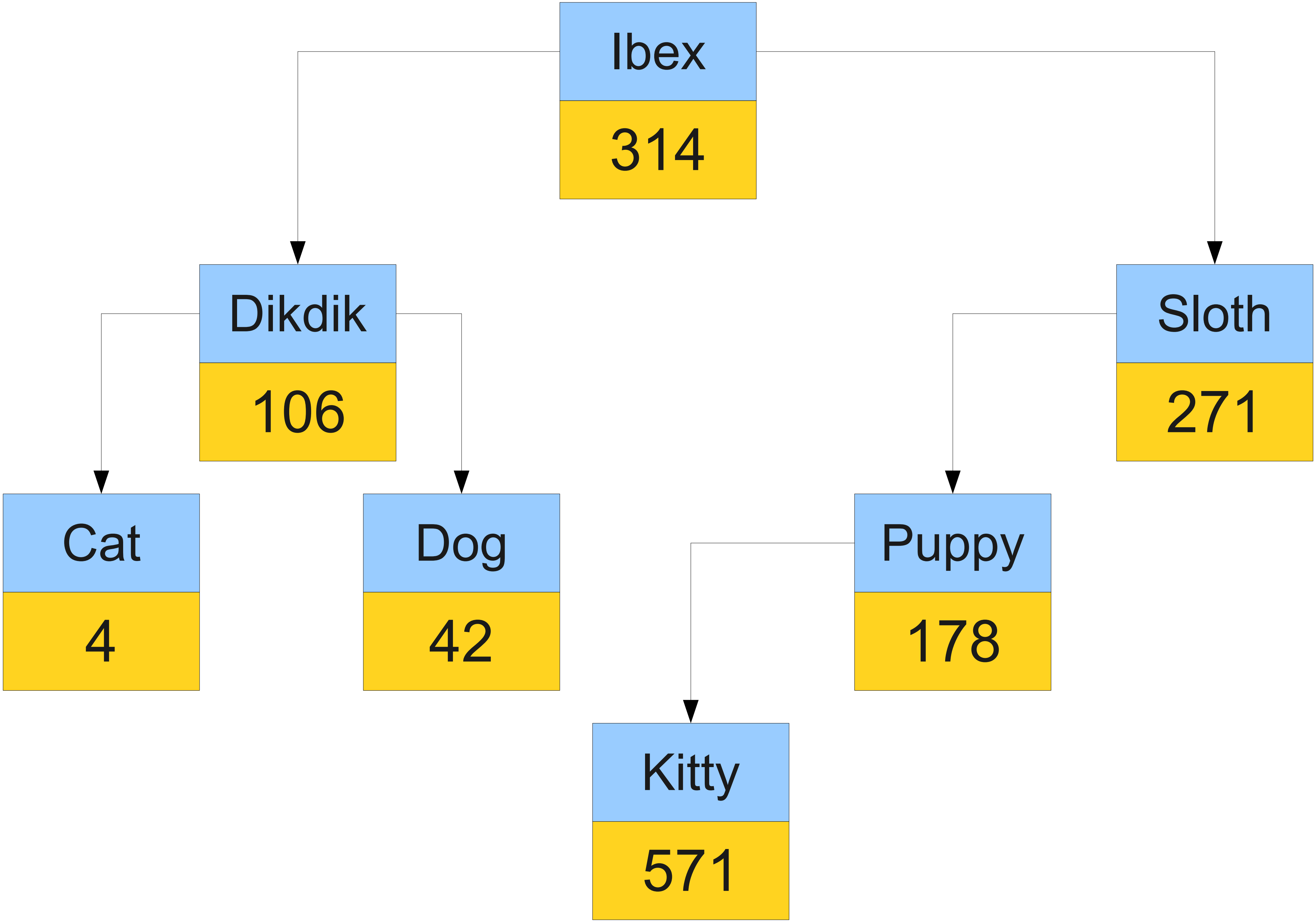
Treaps are Wonderful

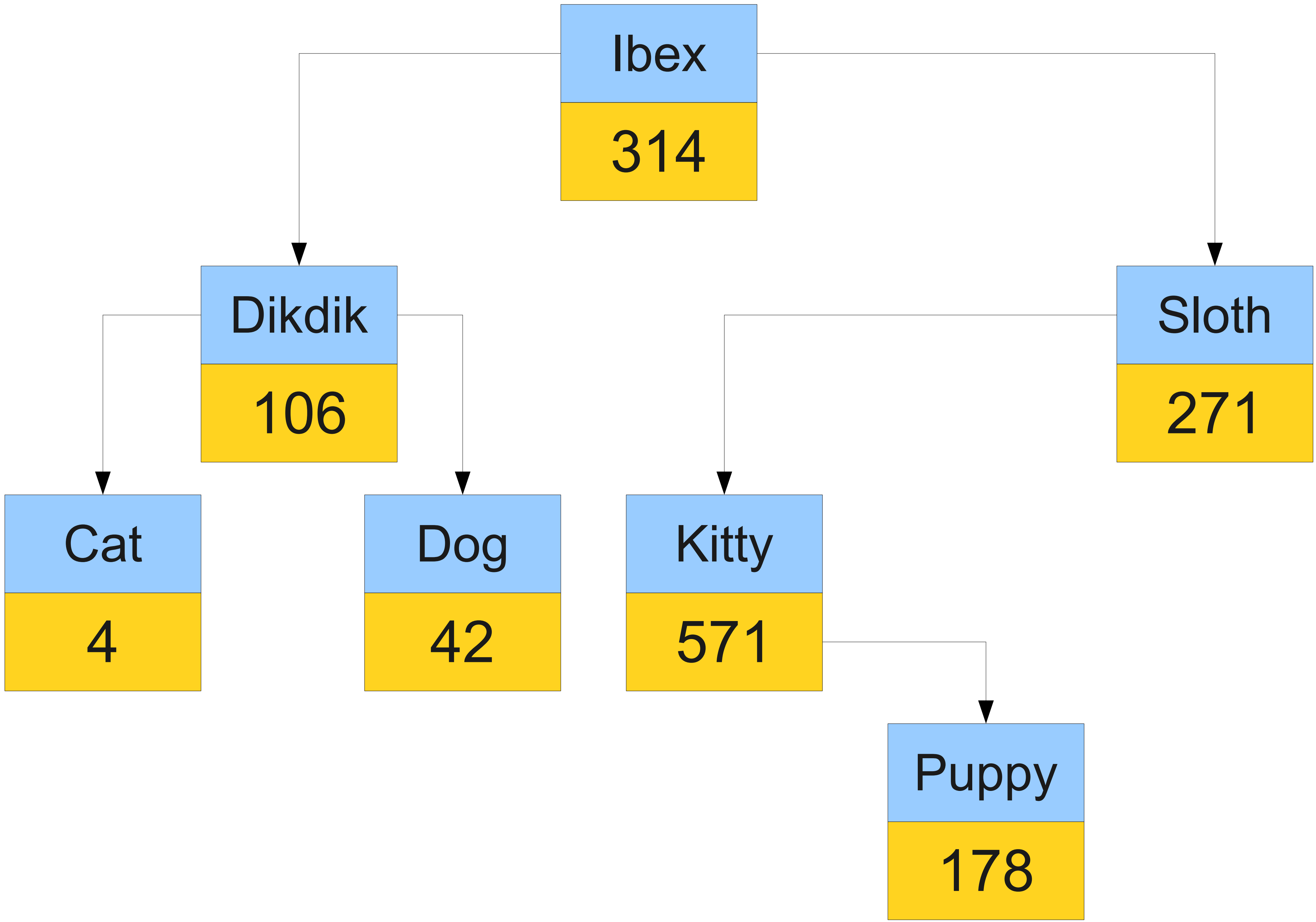
- With very high probability, the height of an n-node treap is O(log n).
- Insertion is surprisingly simple once we have code for tree rotations.
- Deletion is straightforward once we have code for tree rotations.

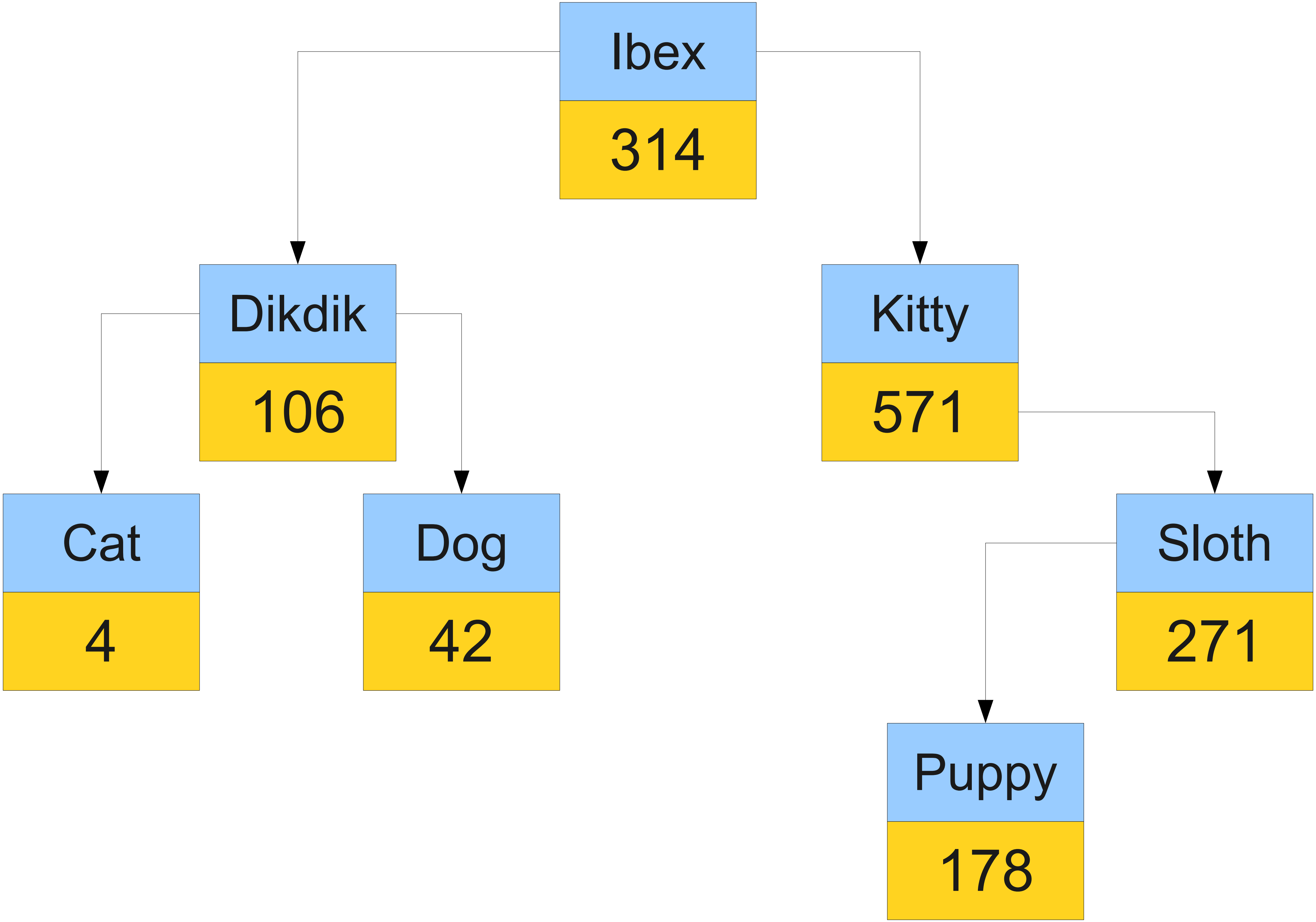
Inserting into a Treap

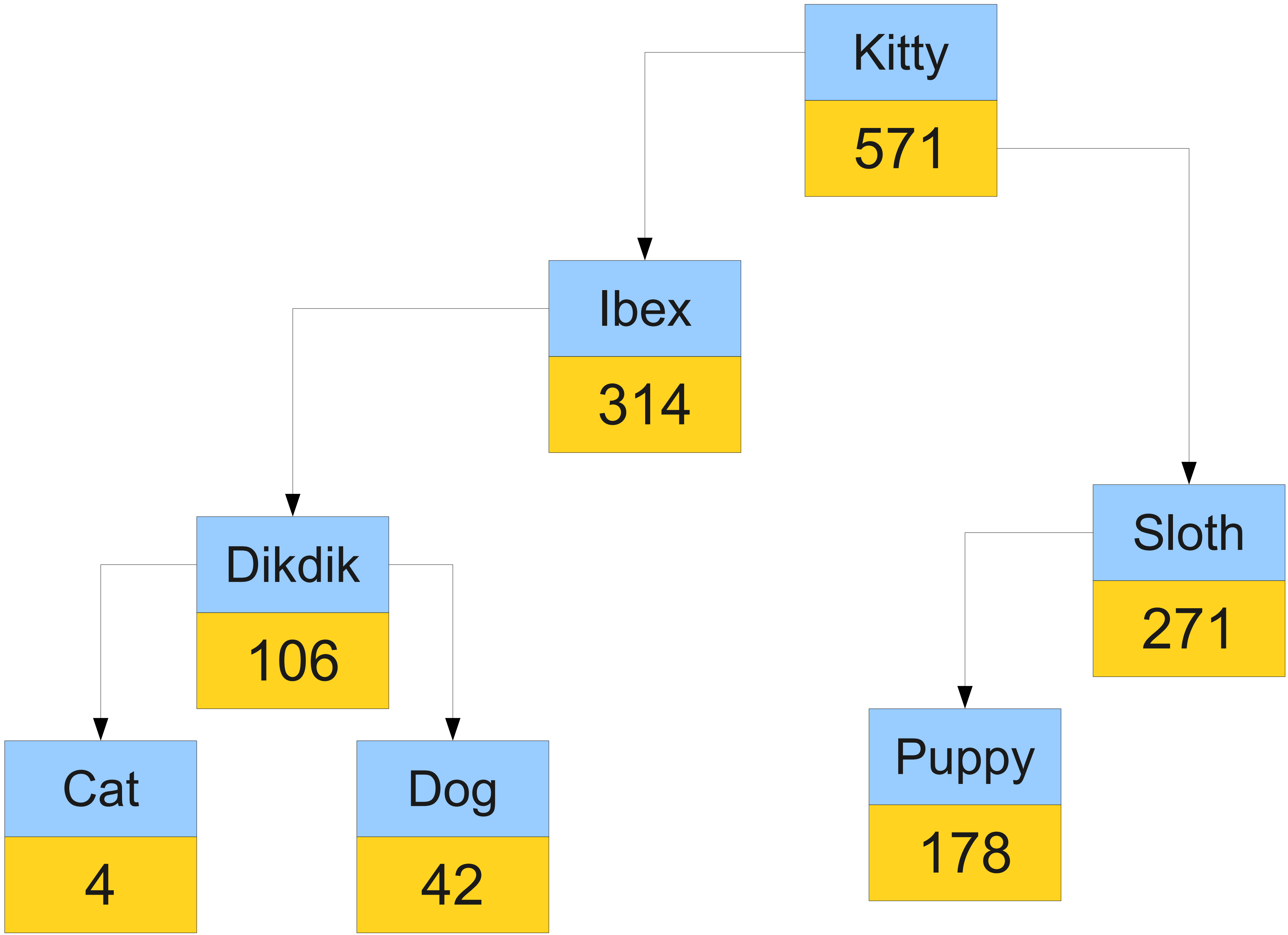
- Insertion into a treap is a combination of normal BST insertion and heap insertion.
- First, insert the node doing a normal BST insertion. This places the value into the right place.
- Next, bubble the node upward in the tree by rotating it with its parent until its value is smaller than its parent.


Ibex
314
Dikdik
106
Sloth
271
Cat
4
Dog
42
Puppy
178


Ibex
314
Dikdik
106
Sloth
271
Cat
4
Dog
42
Puppy
178
Kitty
571


Ibex
314
Dikdik
106
Sloth
271
Cat
4
Dog
42
Kitty
571
Puppy
178

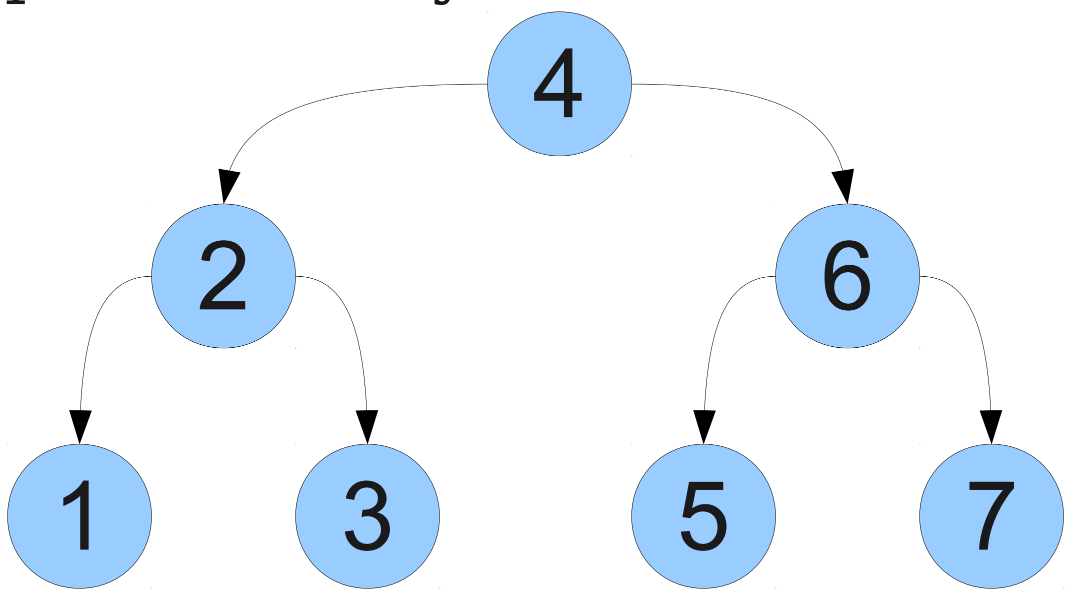

Ibex
314
Dikdik
106
Kitty
571
Cat
4
Dog
42
Sloth
271
Puppy
178


Kitty
571
Ibex
314
Sloth
271
Dikdik
106
Puppy
178
Cat
4
Dog
42

Let's Code it Up!

Removing from a Treap

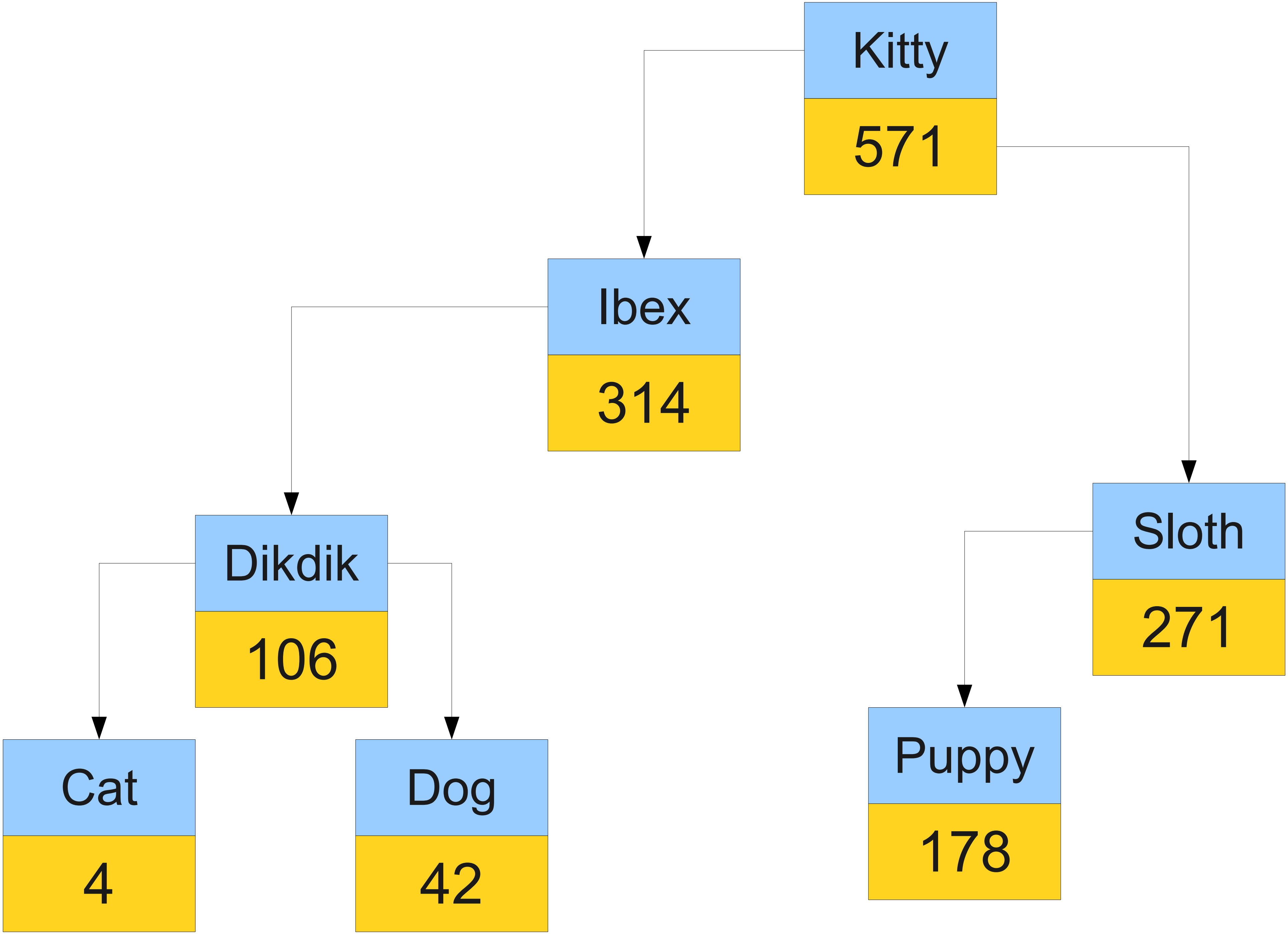
- In general, removing a node from a BST is quite difficult because we have to make sure not to lose any nodes.
- For example, how do you remove the root of this tree?

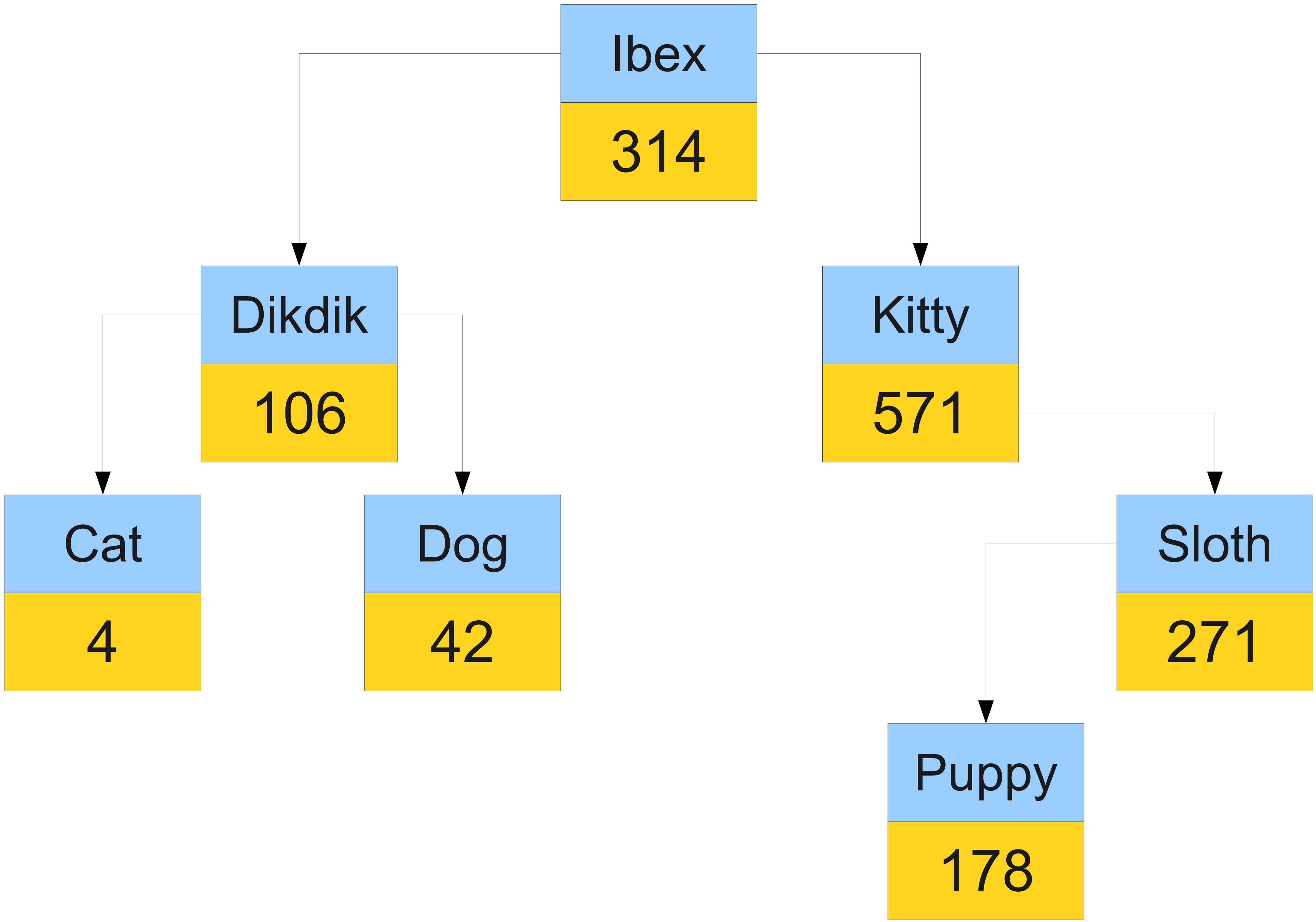


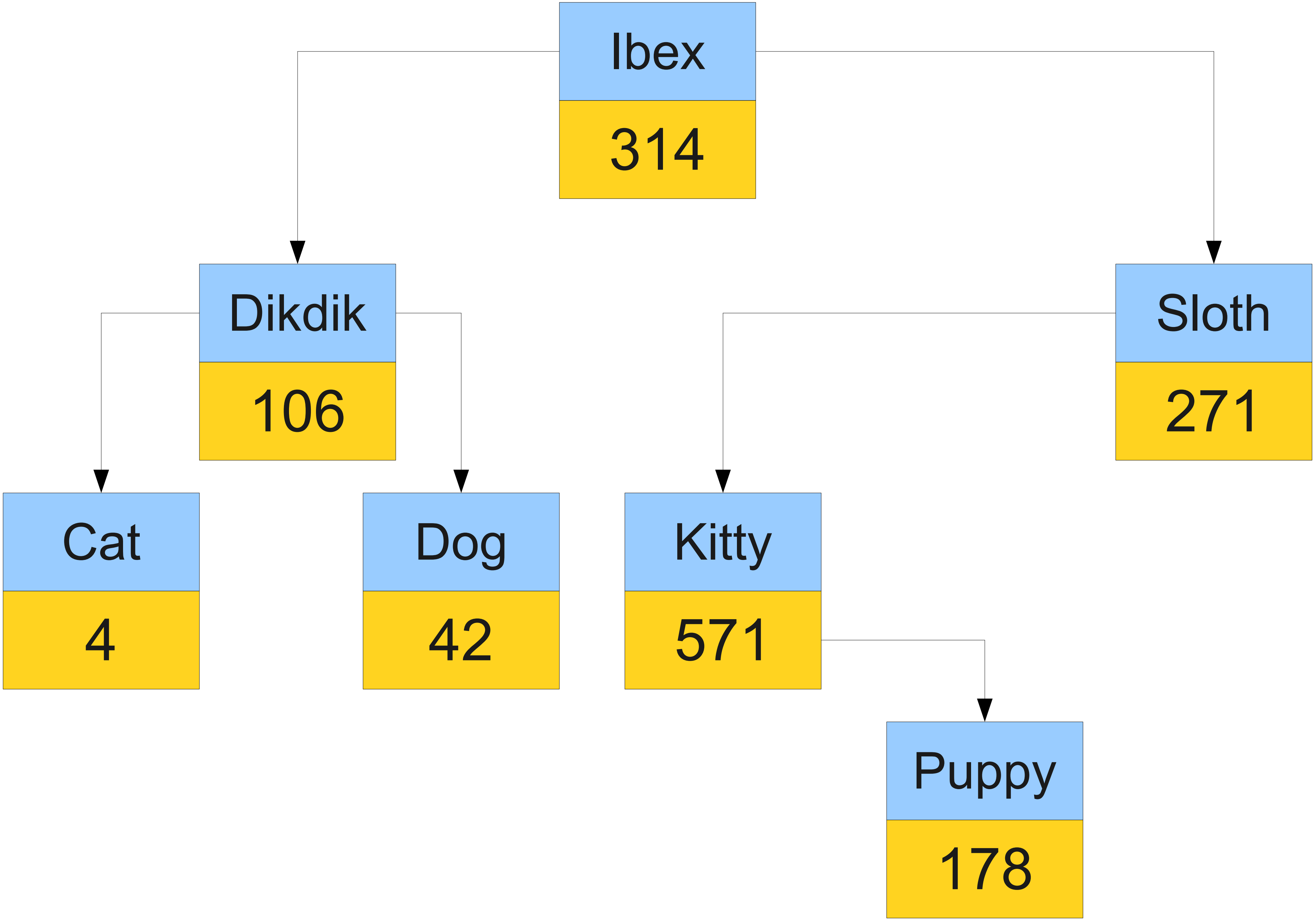
- However, removing leaves is very easy, since they have no children.

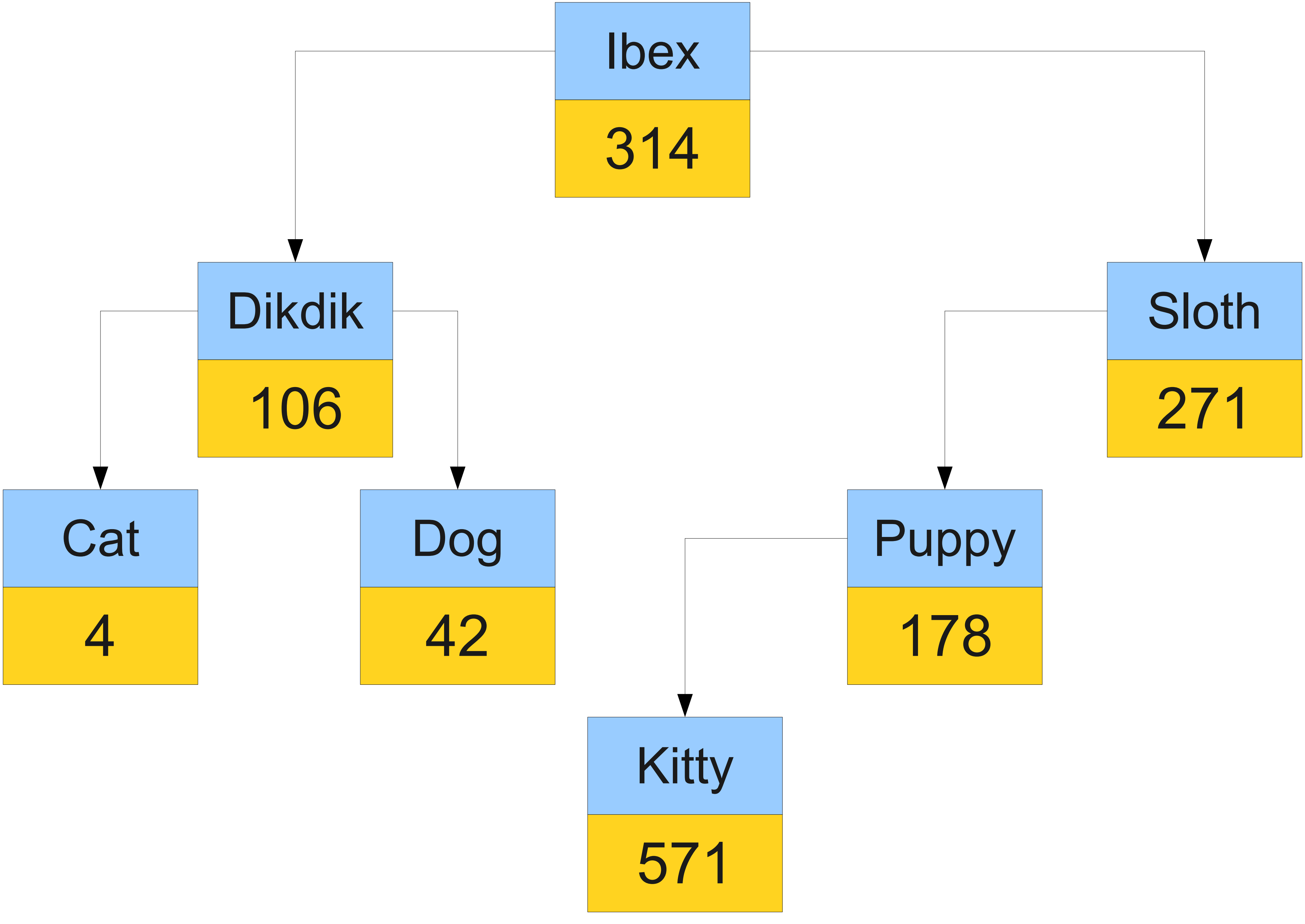
Removing from a Treap

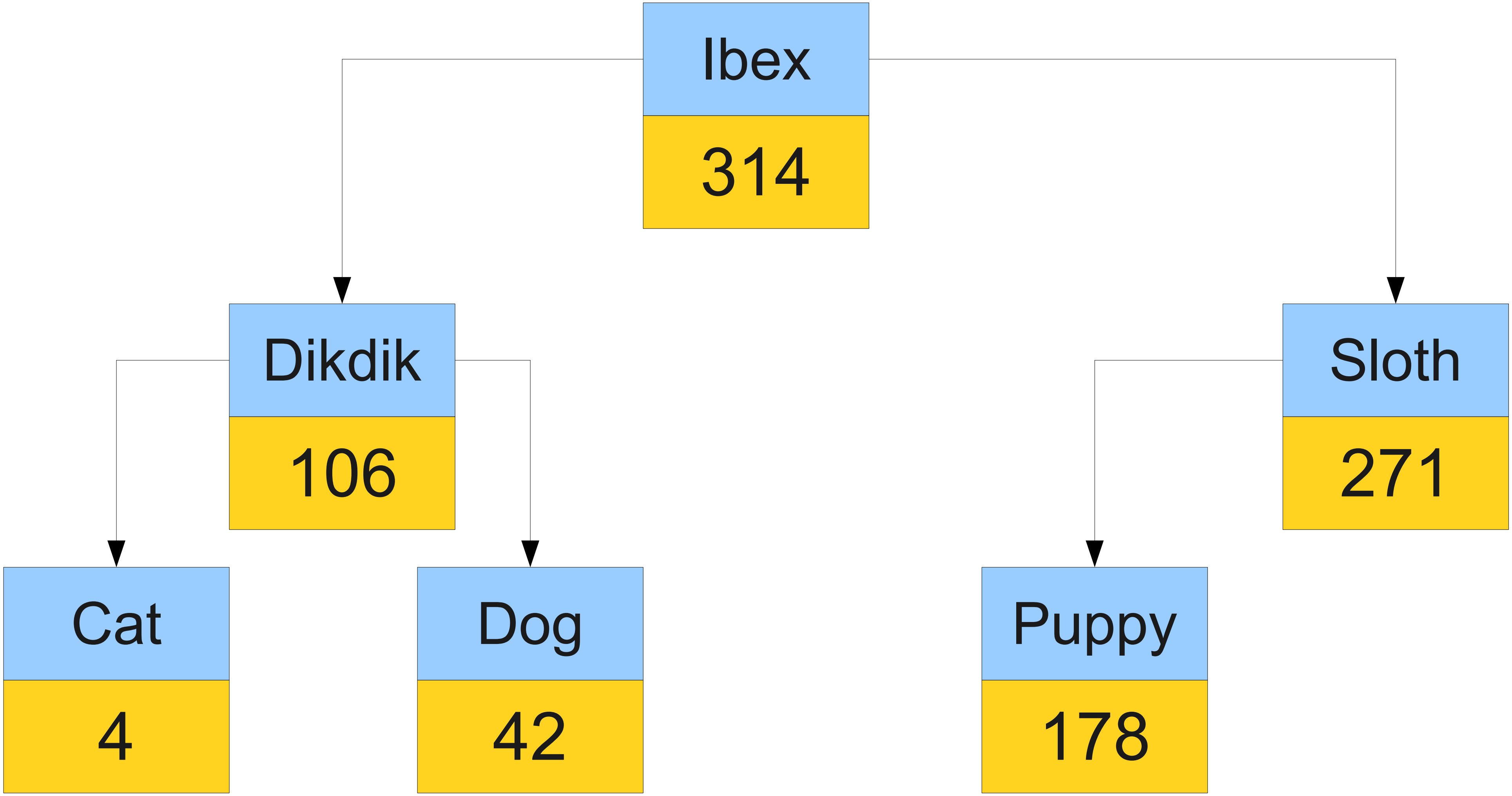
- It would seem that, since a treap has extra structure on top of that of a BST, that removing from a treap would be extremely hard.
- However, it's actually quite simple:
  - Keep rotating the node to delete with its larger child until it becomes a leaf.
  - Once the node is a leaf, delete it.


Kitty
571
Ibex
314
Sloth
271
Dikdik
106
Puppy
178
Cat
4
Dog
42


Ibex
314
Dikdik
106
Kitty
571
Cat
4
Dog
42
Sloth
271
Puppy
178


Ibex
314
Dikdik
106
Sloth
271
Cat
4
Dog
42
Kitty
571
Puppy
178


Ibex
314
Dikdik
106
Sloth
271
Cat
4
Dog
42
Puppy
178
Kitty
571


Ibex
314
Dikdik
106
Sloth
271
Cat
4
Dog
42
Puppy
178

Summary of Treaps

- Treaps give a (reasonably) straightforward way to guarantee that the height of a BST is not too great.
- Insertion into a treap is similar to insertion into a BST followed by insertion into a binary heap.
- Deletion from a treap is similar to the bubble-down step from a heap.
- All operations run in expected O(log n) time.

A Survey of Other Data Structures

Data Structures so Far

- We have seen many data structures over the past few weeks:
  - Dynamic arrays.
  - Linked lists.
  - Hash tables.
  - Tries.
  - Binary search trees (and treaps).
  - Binary heaps.
- These are the most-commonly-used data structures for general data storage.

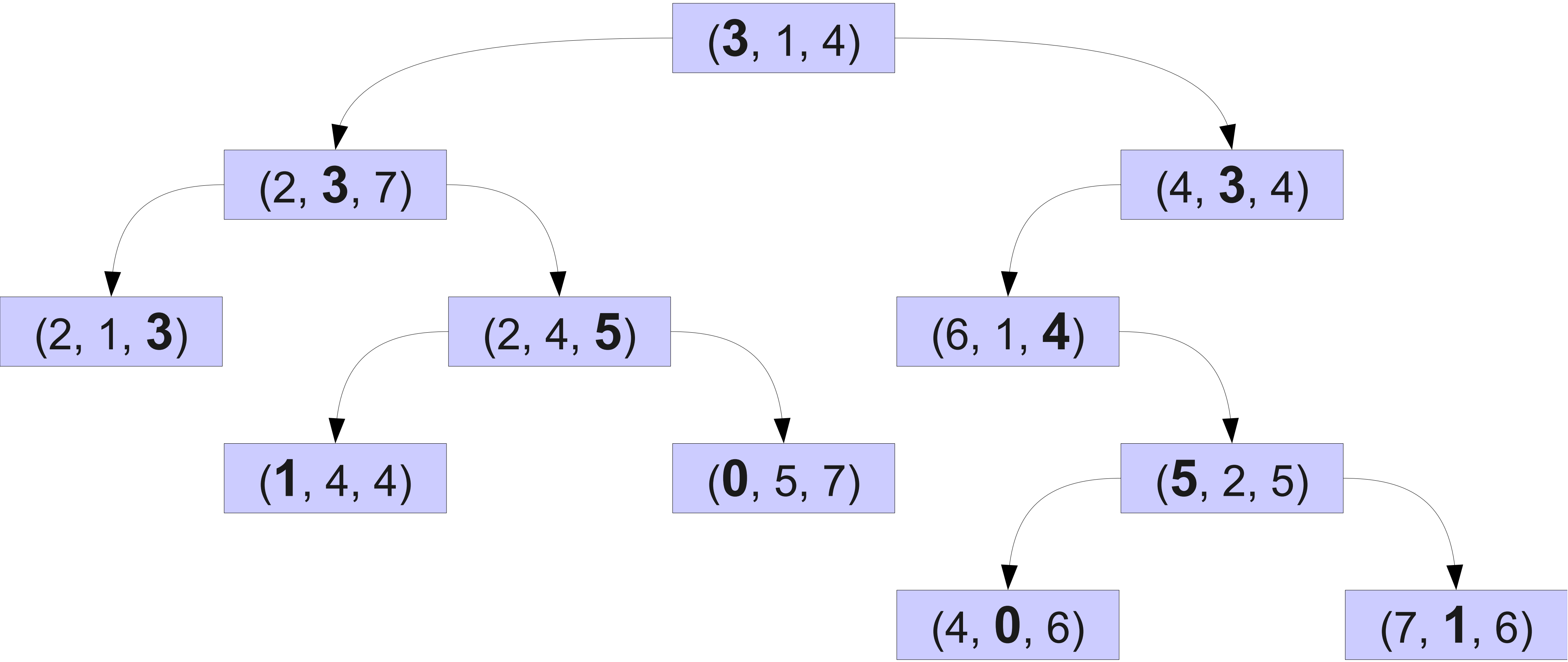
Specialized Data Structures

- For applications that manipulate specific types of data, other data structures exist that make certain operations surprisingly fast and efficient.
- Many critical applications of computers would be impossible without these data structures.

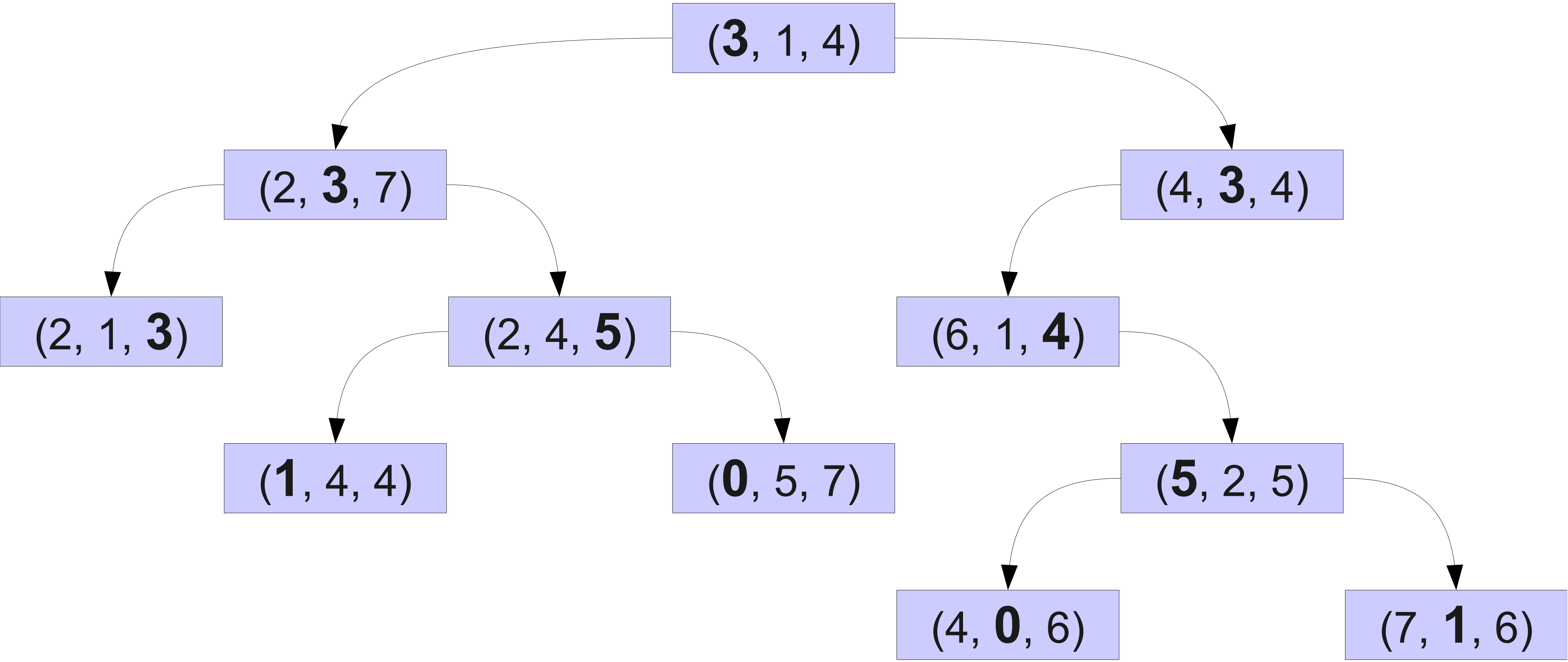
k-d Trees

Suppose that you want to efficiently store points in k-dimensional space.

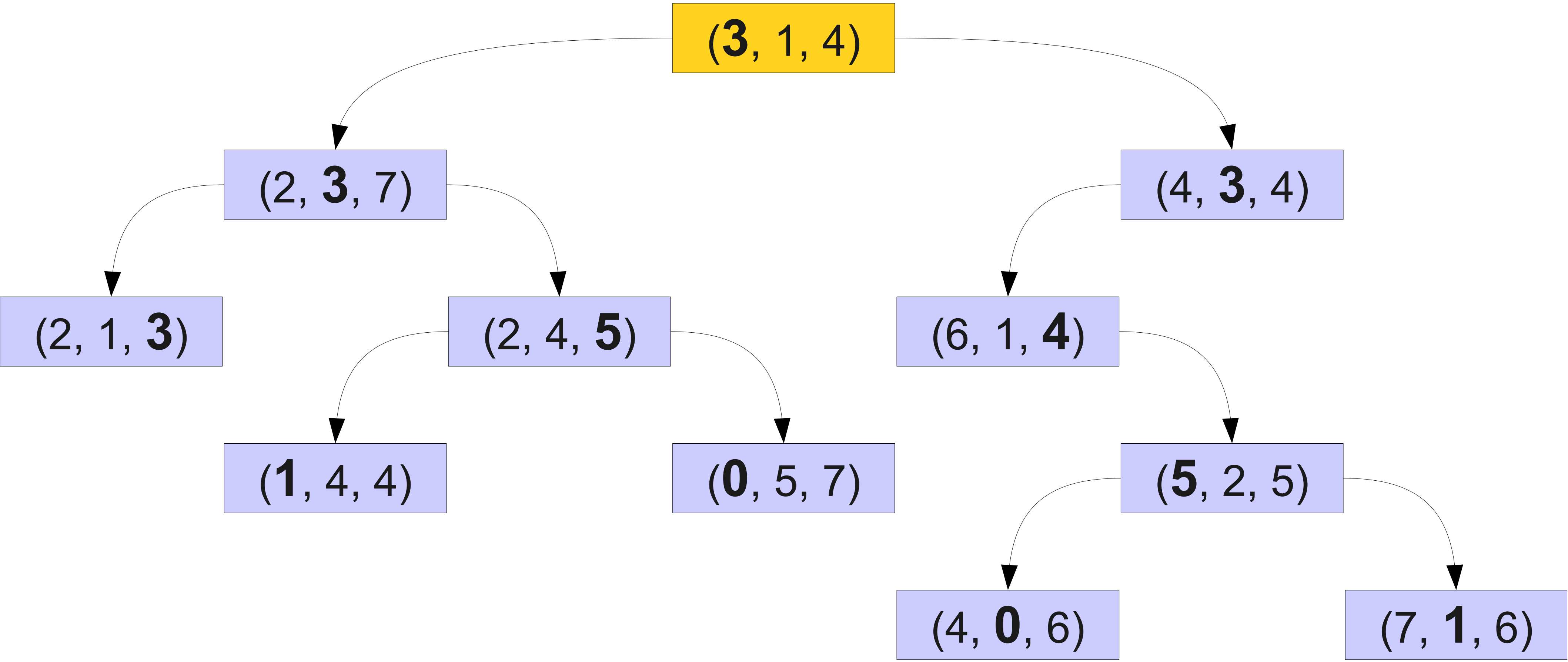
How might you organize the data to efficiently query for points within a region?


(3, 1, 4)
(2, 3, 7)
(4, 3, 4)
(2, 1, 3)
(2, 4, 5)
(6, 1, 4)
(1, 4, 4)
(0, 5, 7)
(5, 2, 5)
(4, 0, 6)
(7, 1, 6)

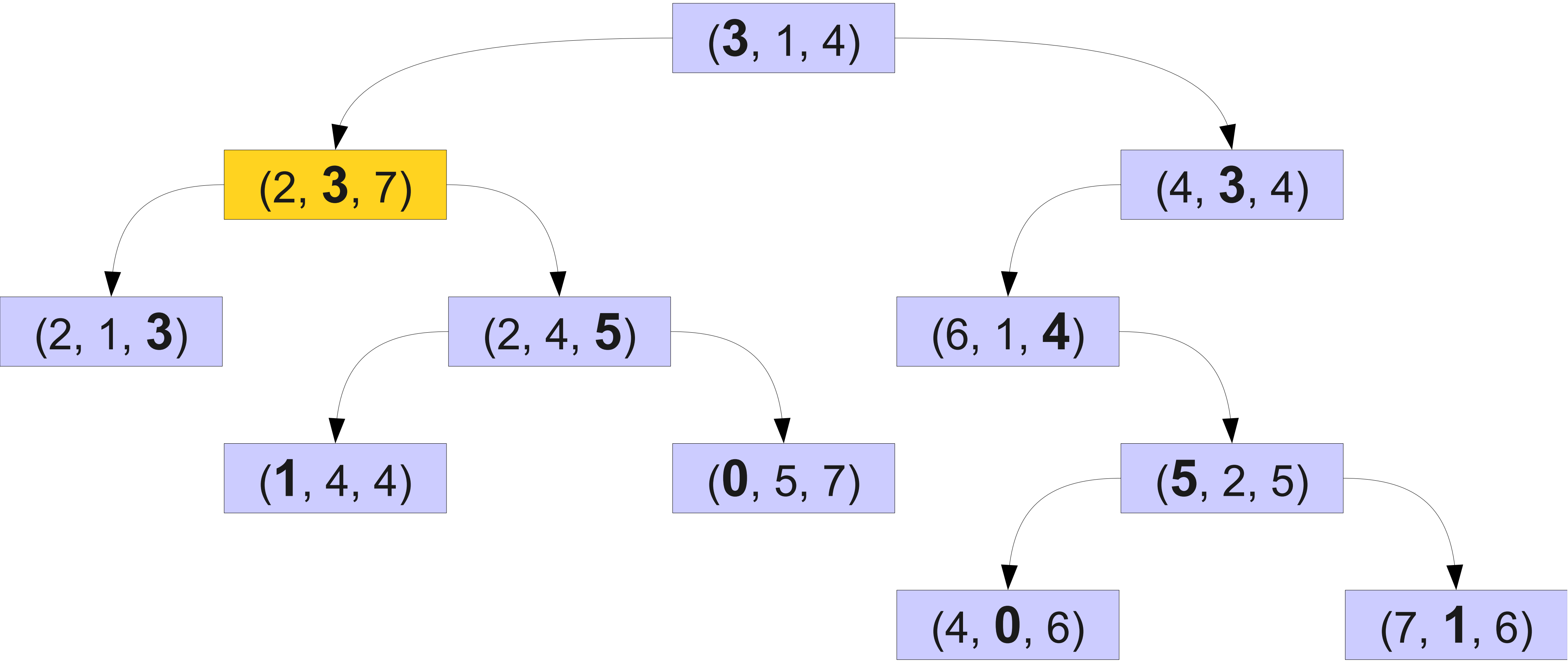
Search for (1, 4, 4)



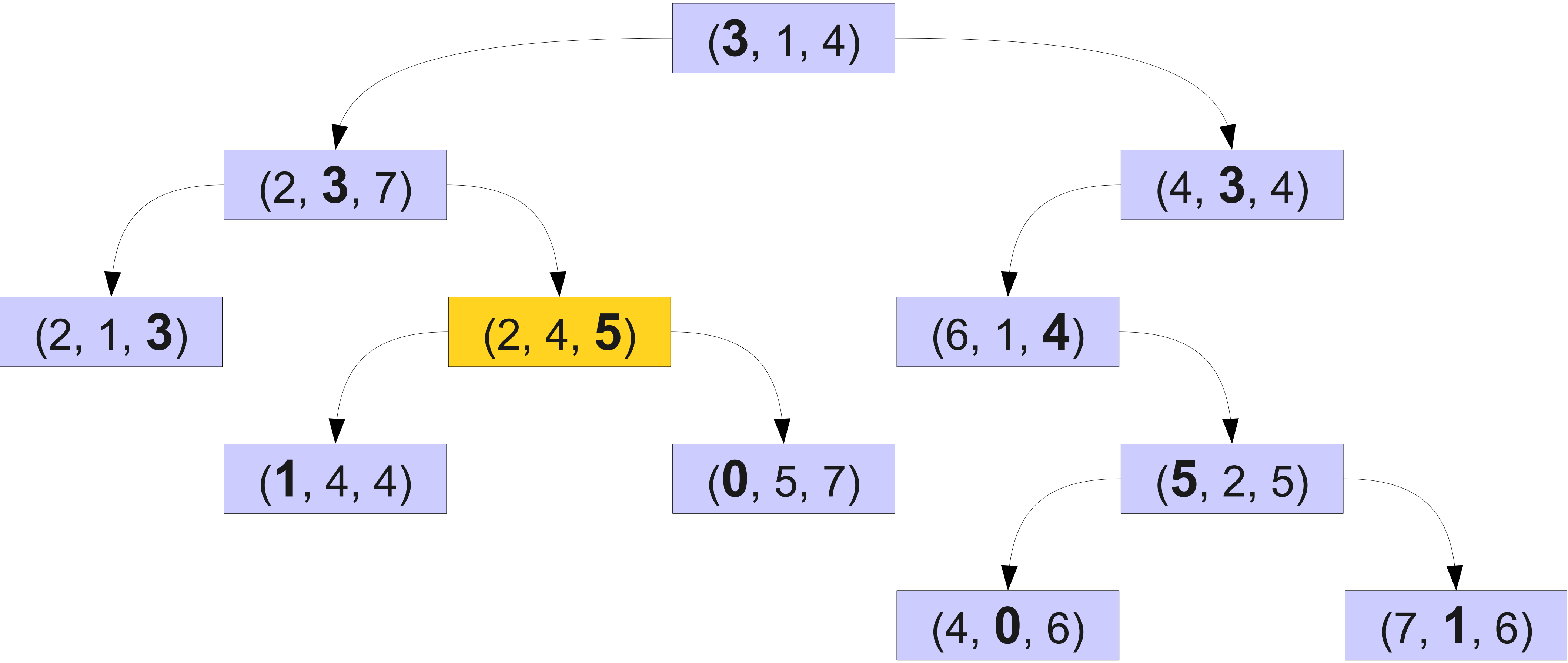
Search for (1, 4, 4)



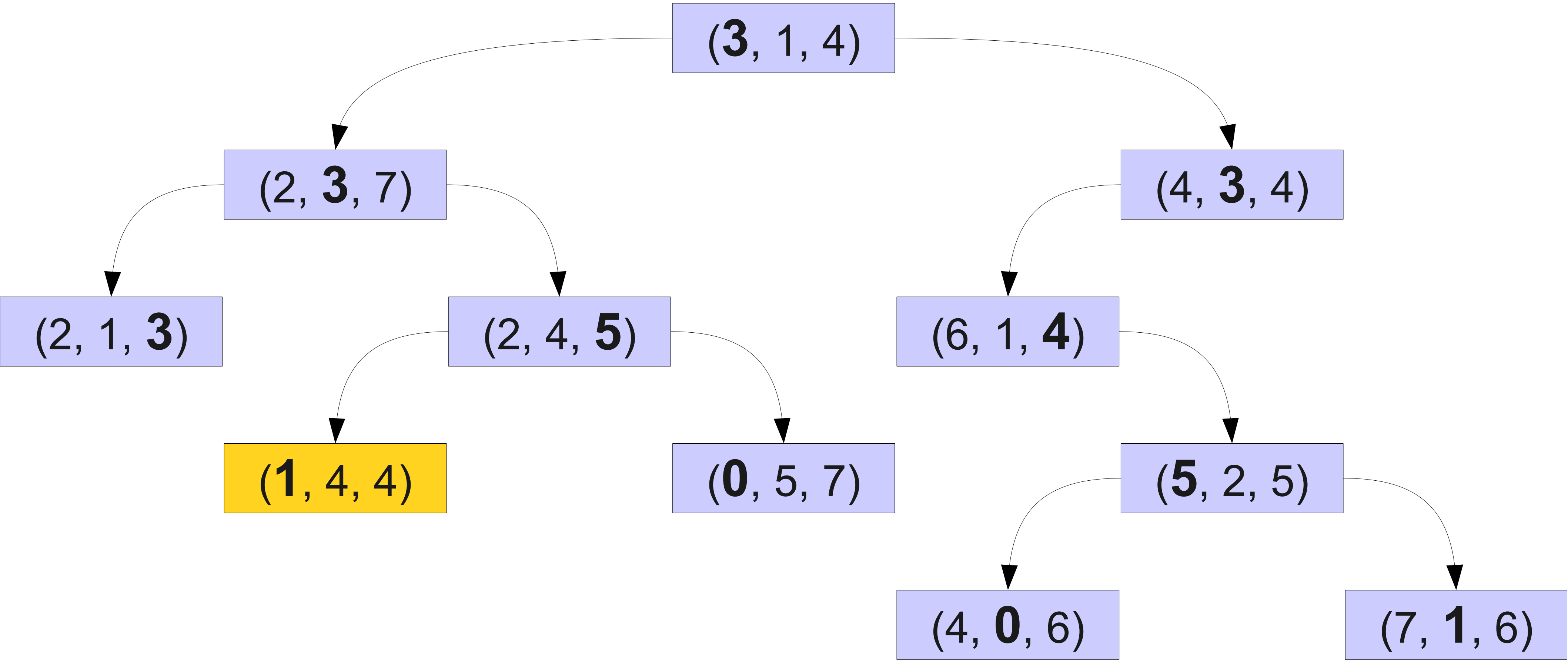
Search for (1, 4, 4)



Search for (1, 4, 4)

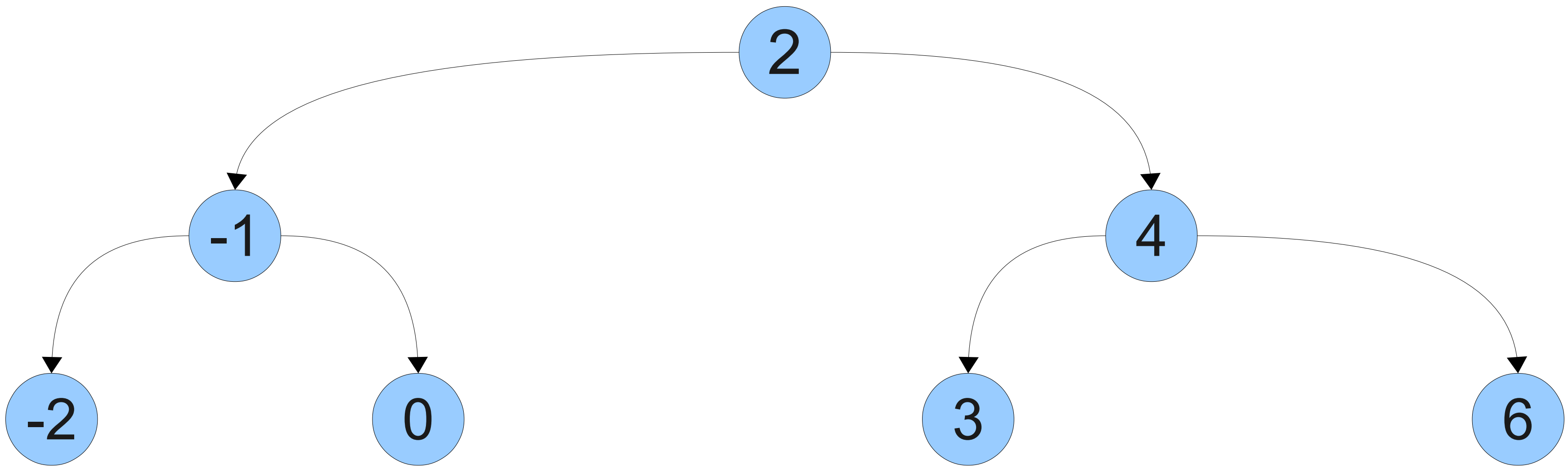


Search for (1, 4, 4)

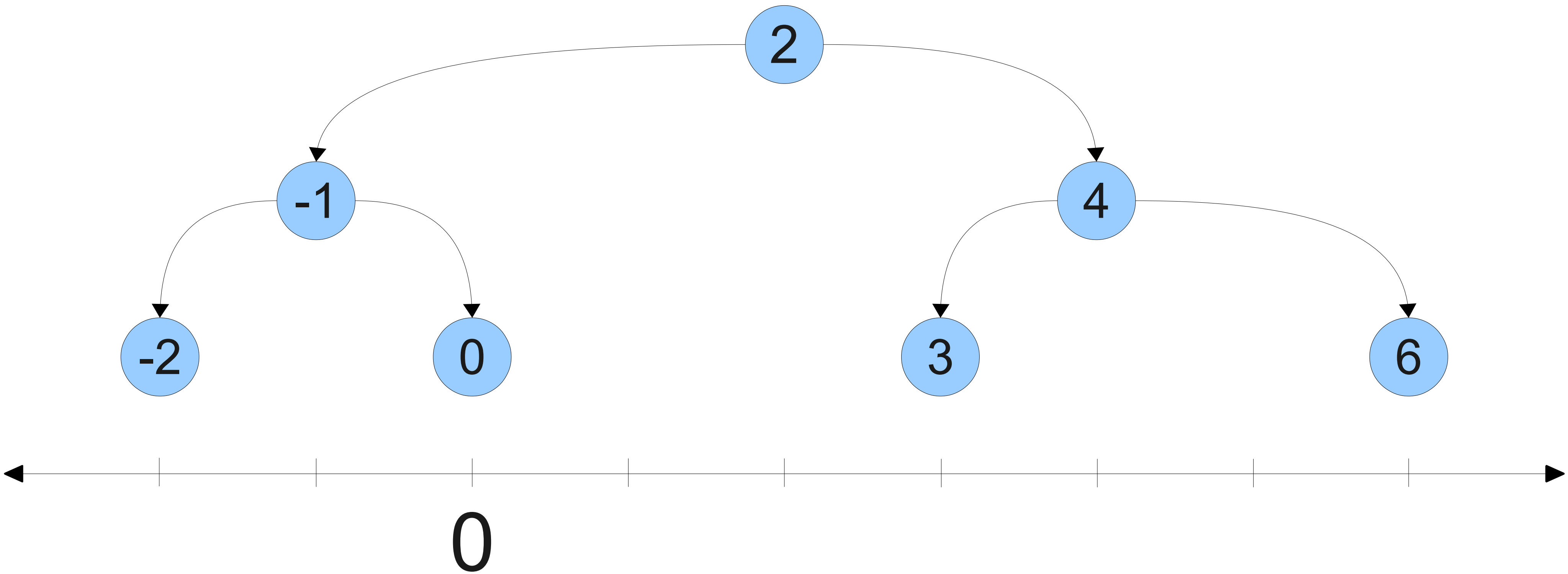


The Intuition

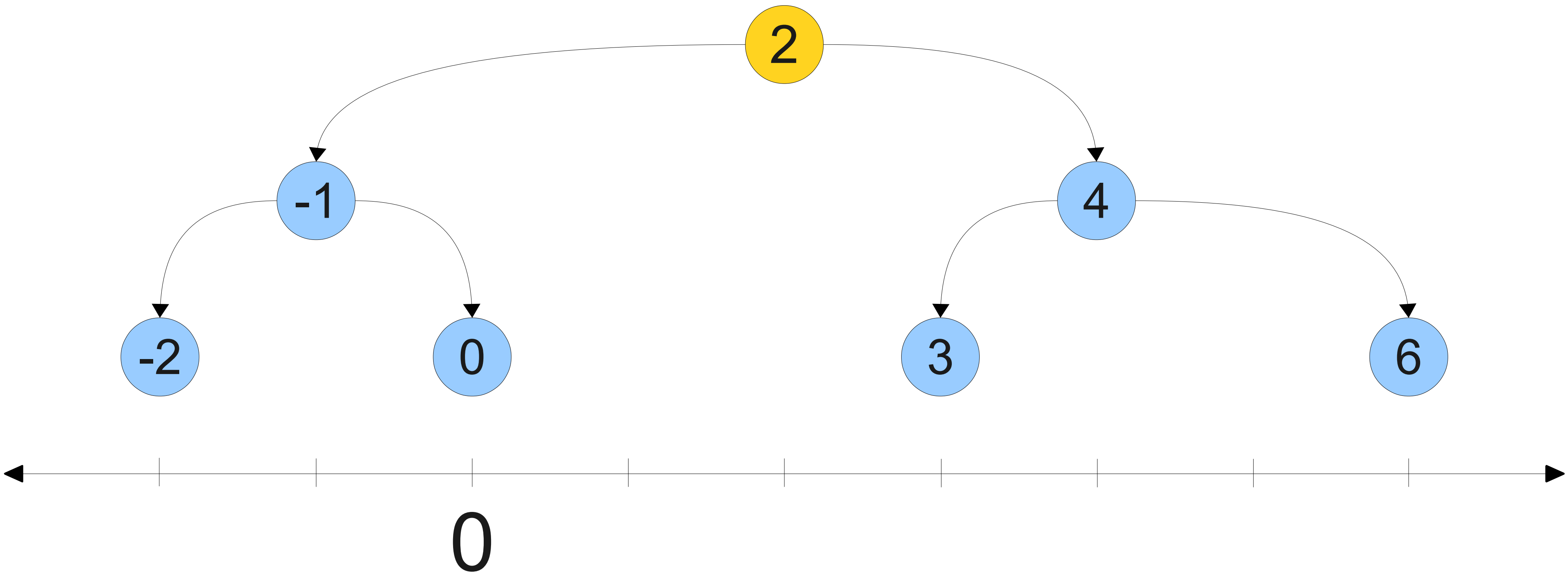
The Intuition



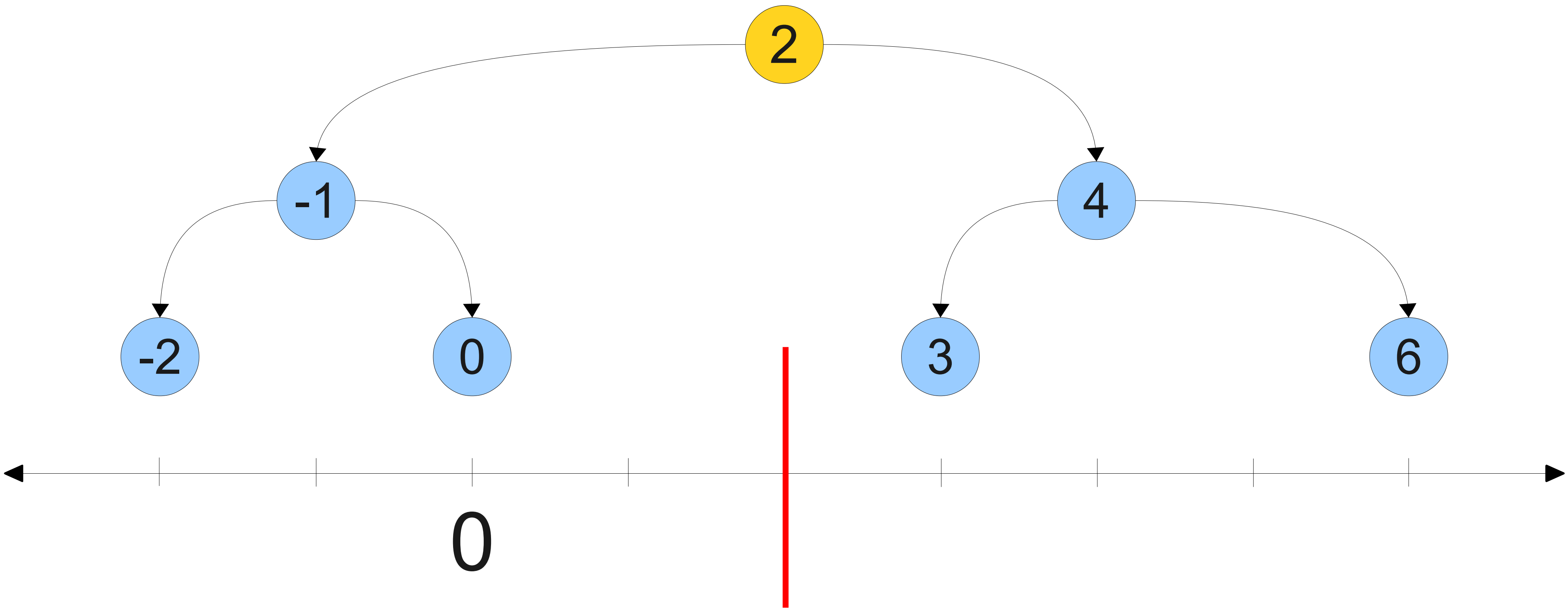
The Intuition



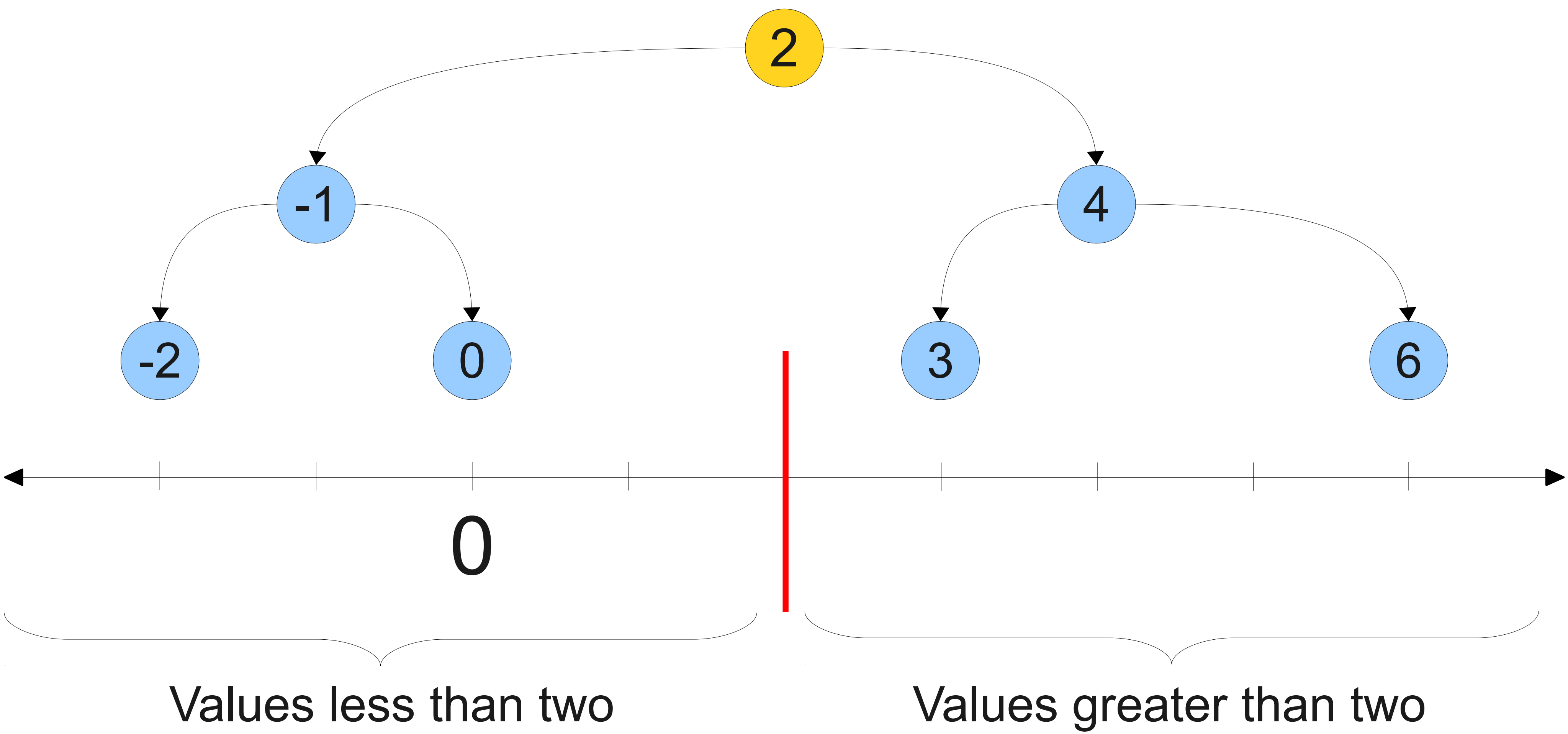
The Intuition



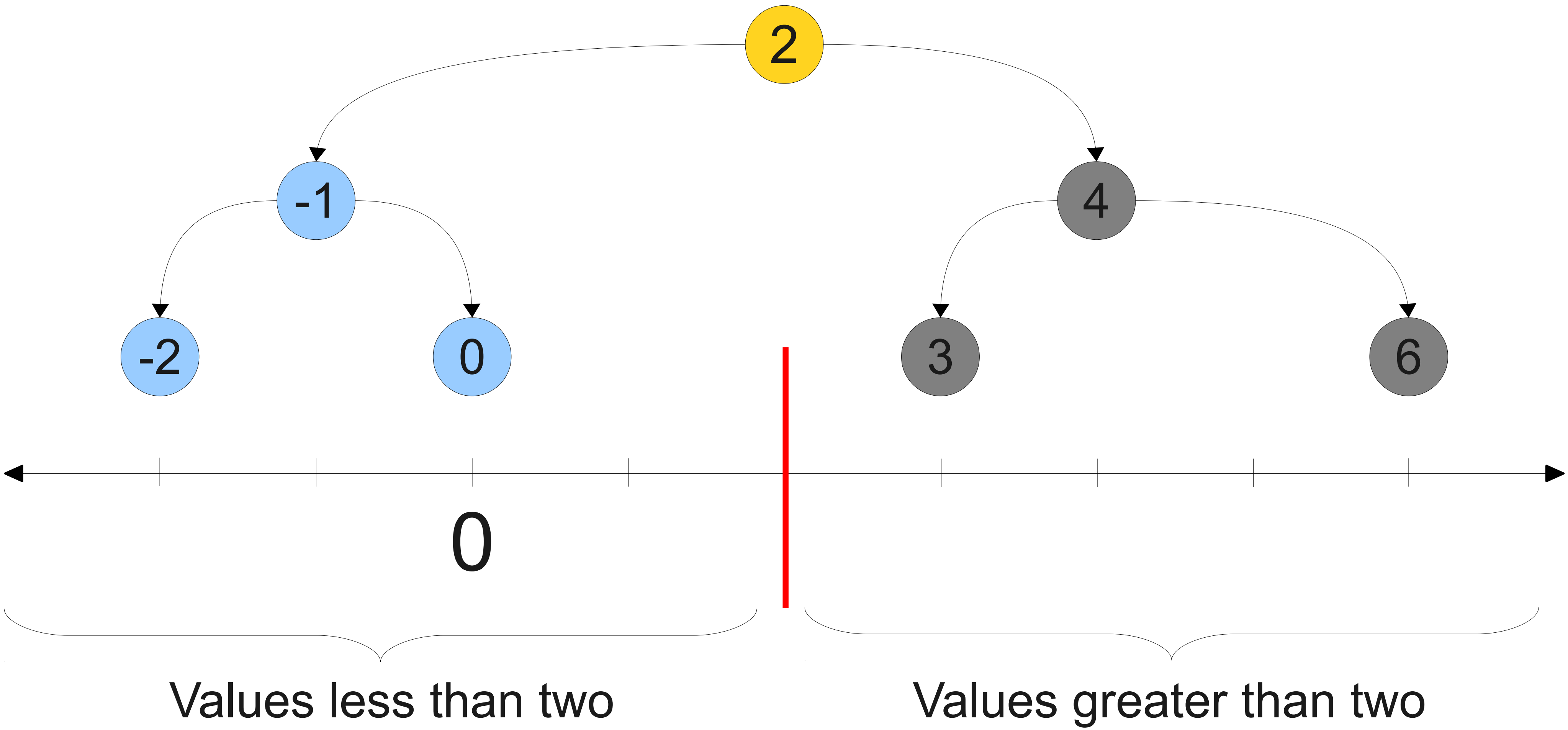
The Intuition



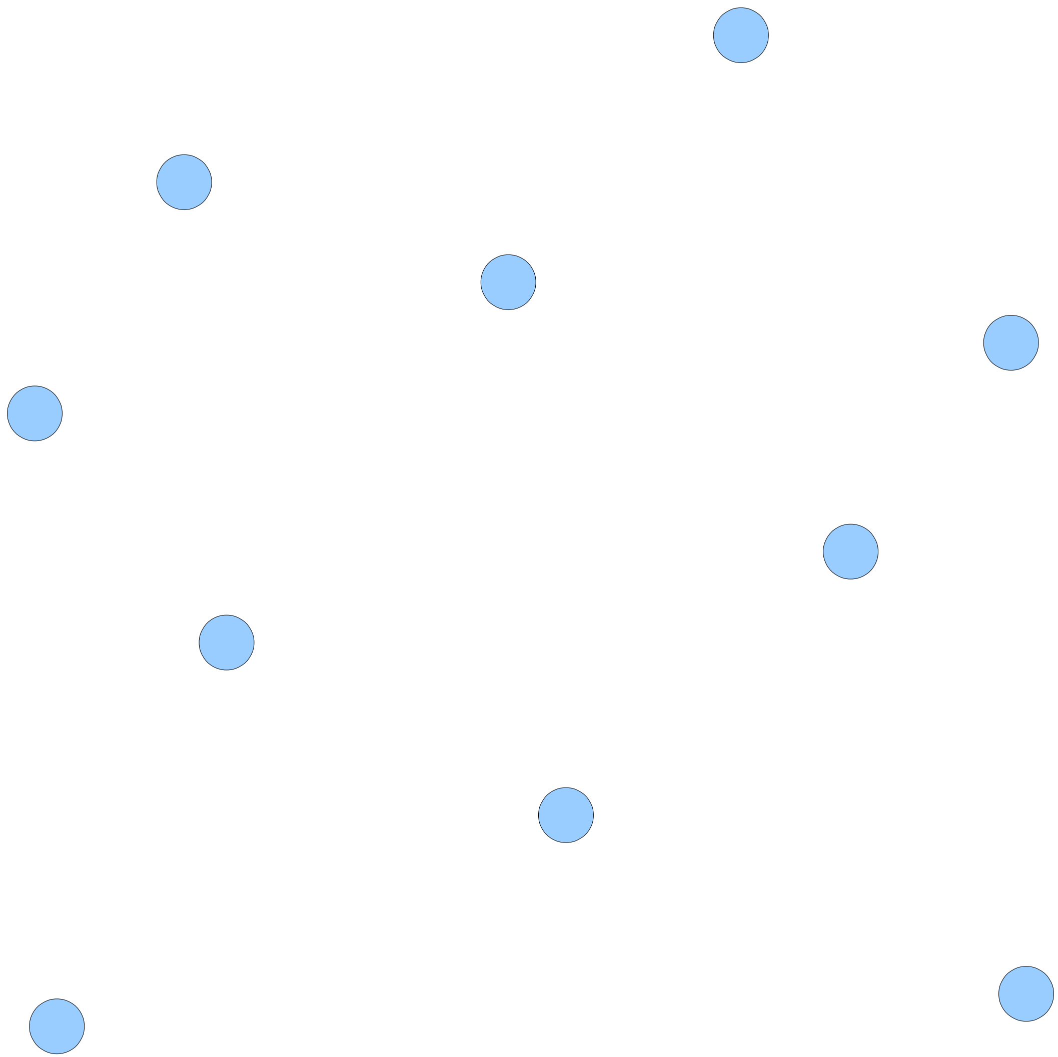
The Intuition

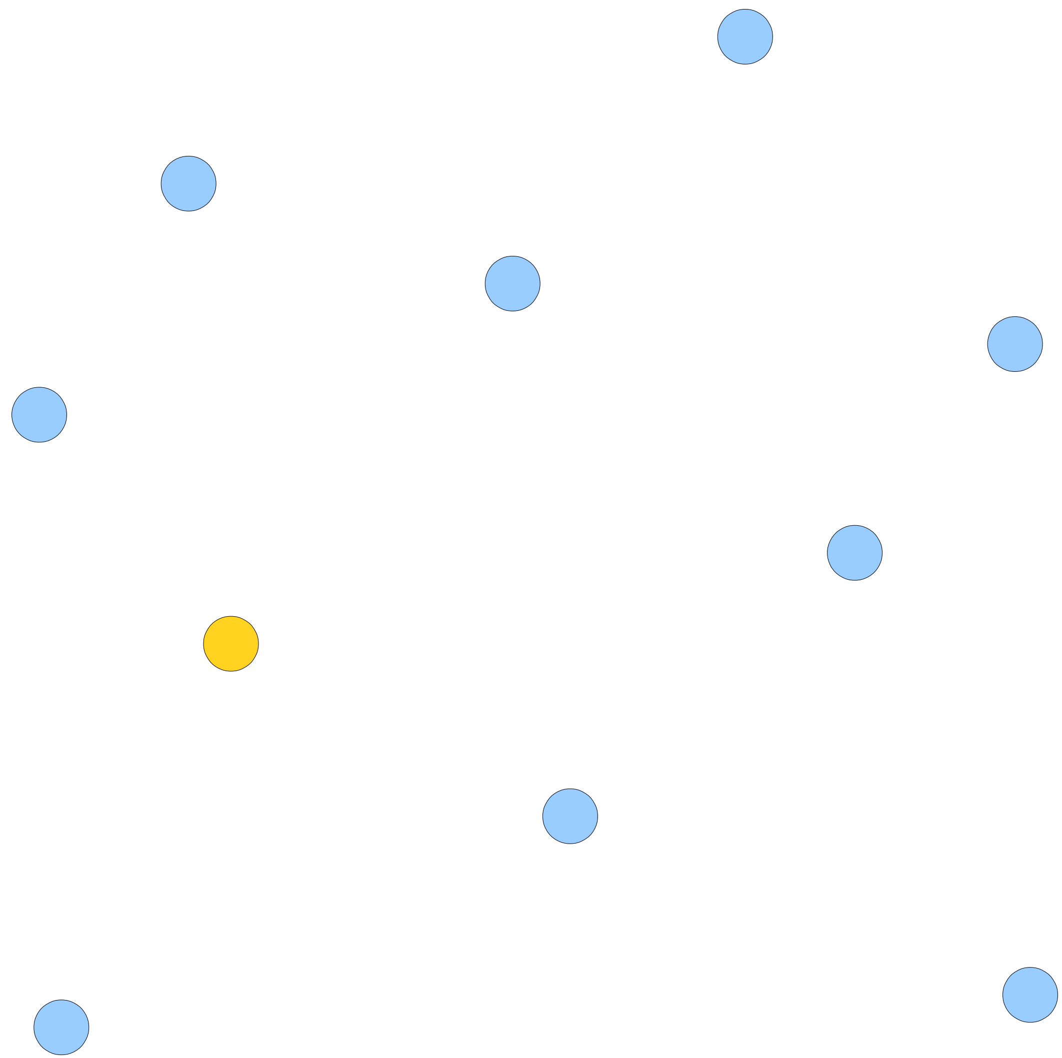


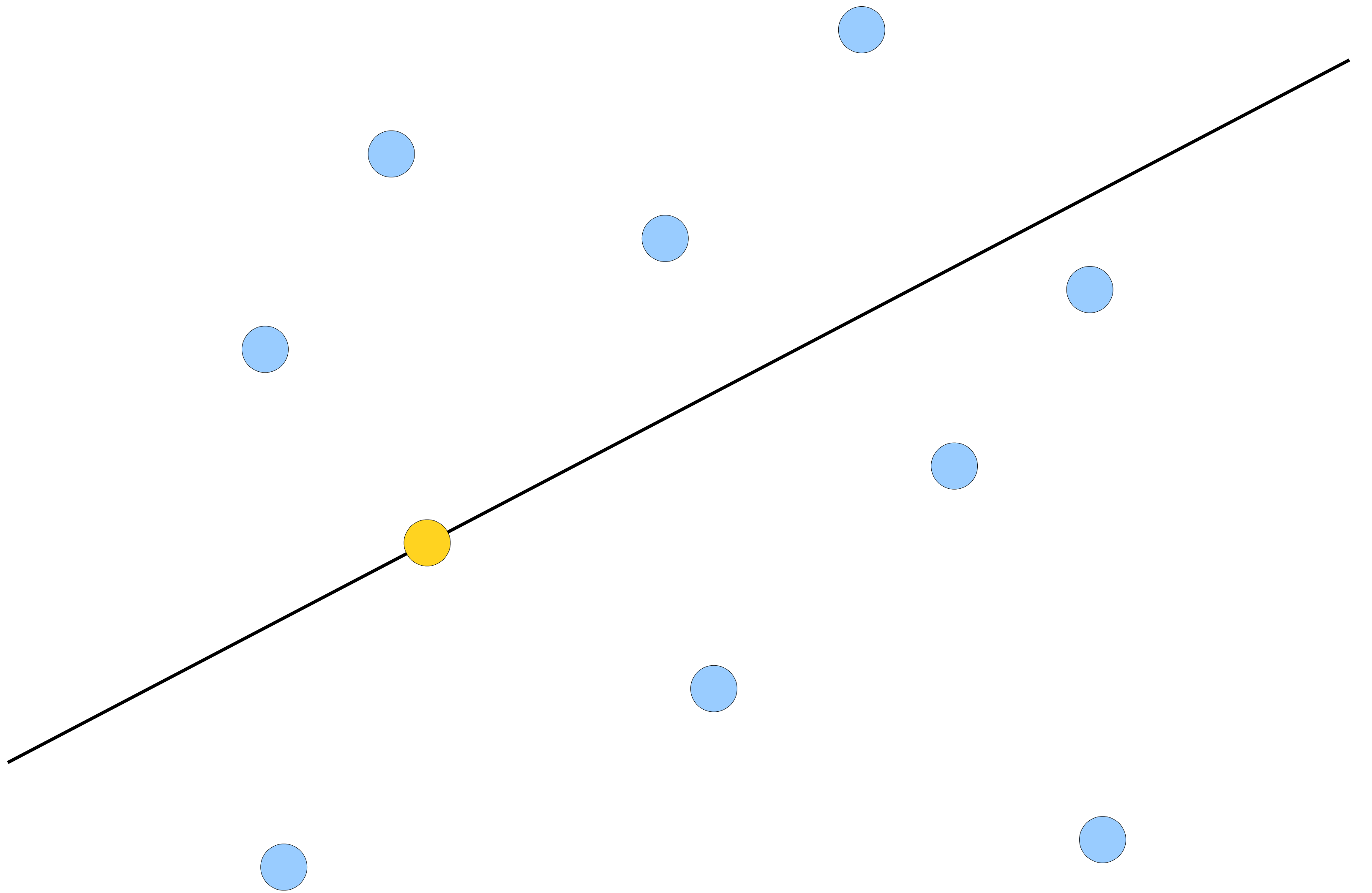
The Intuition

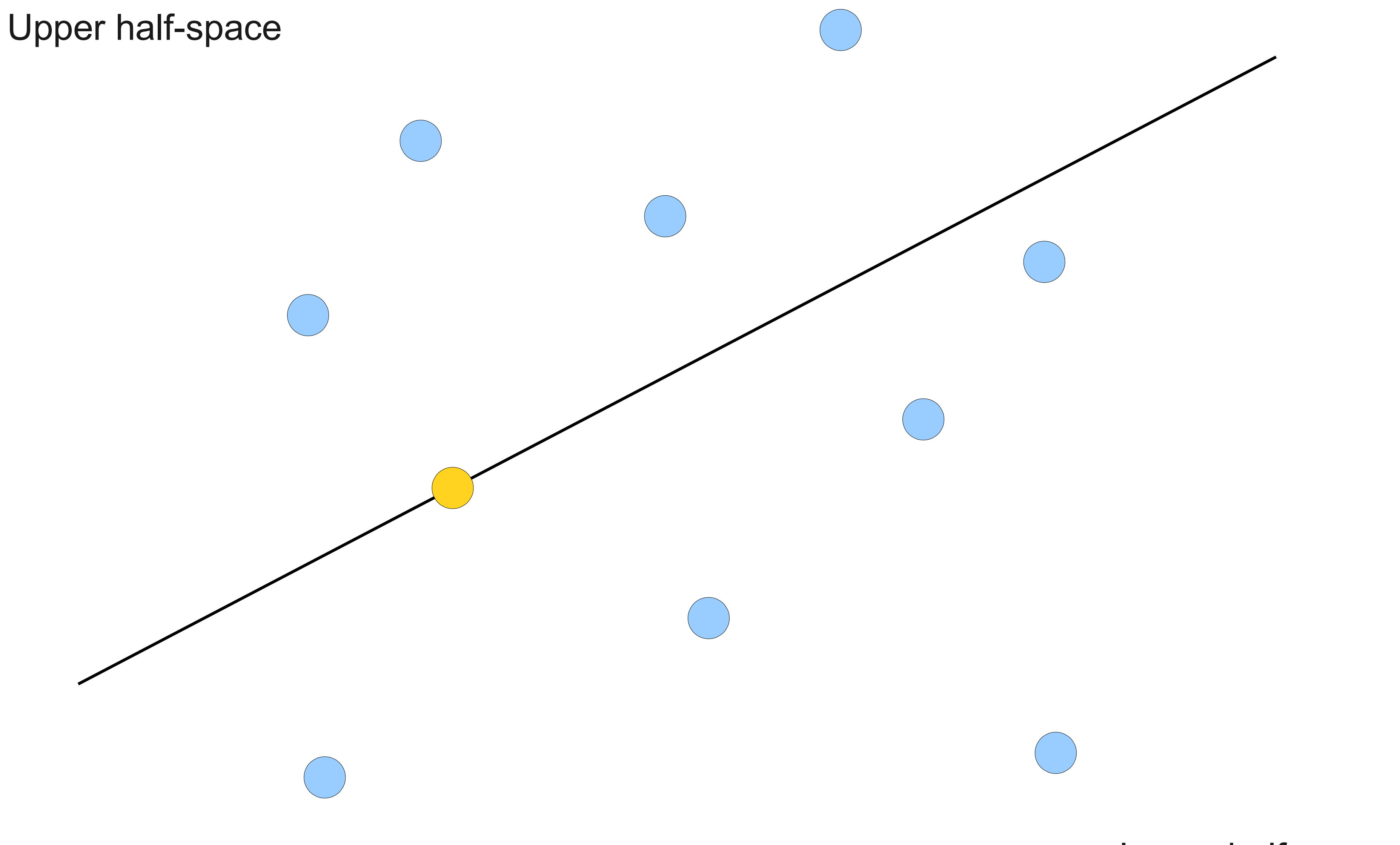


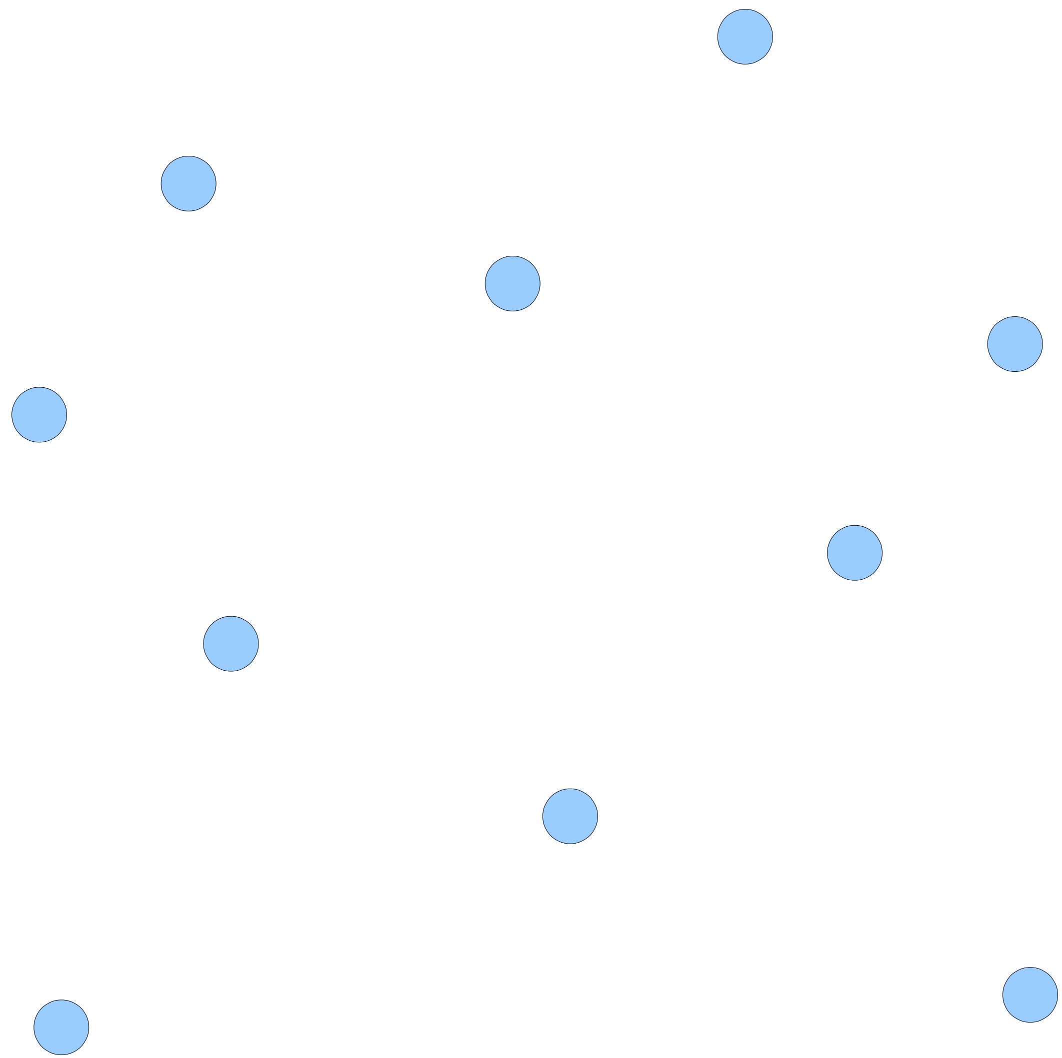
Key Idea: **Split Space in Half**

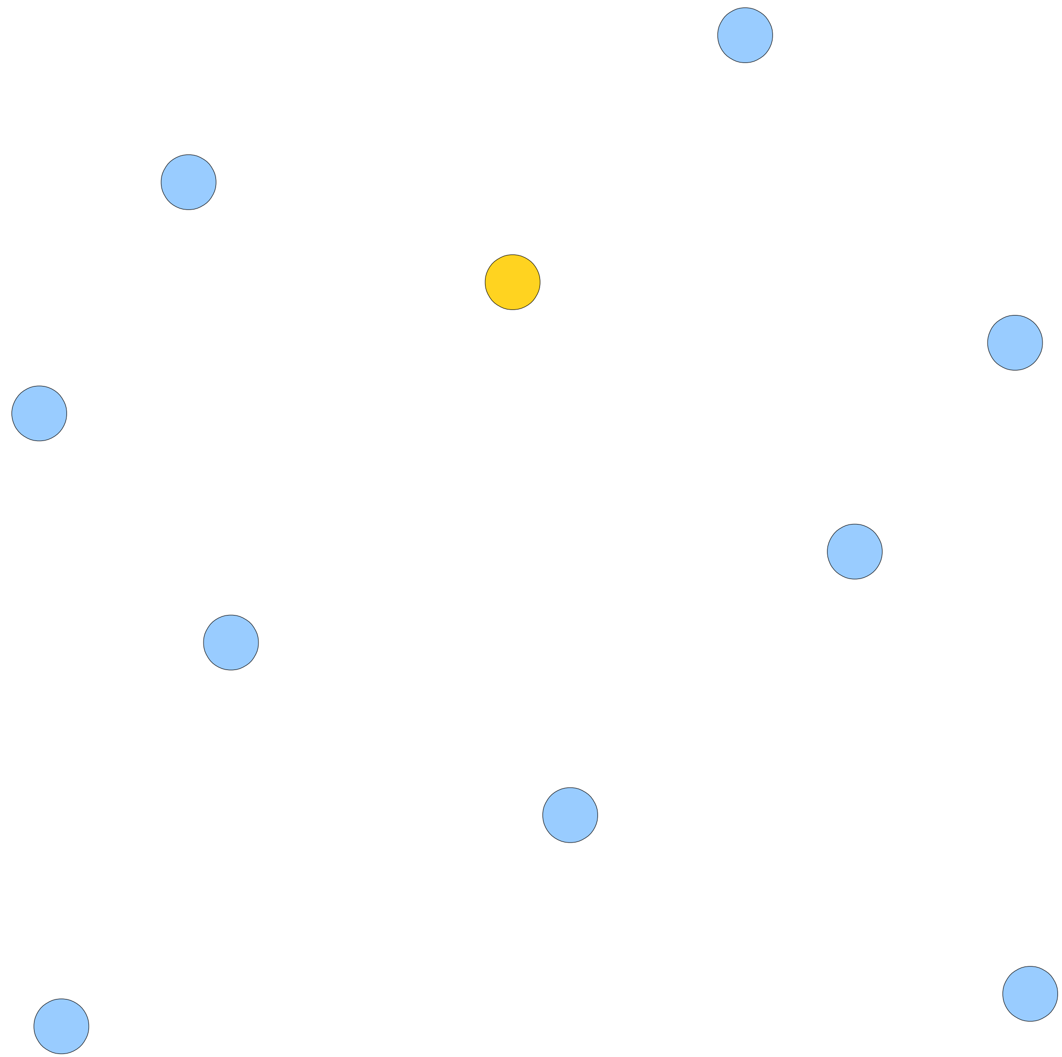


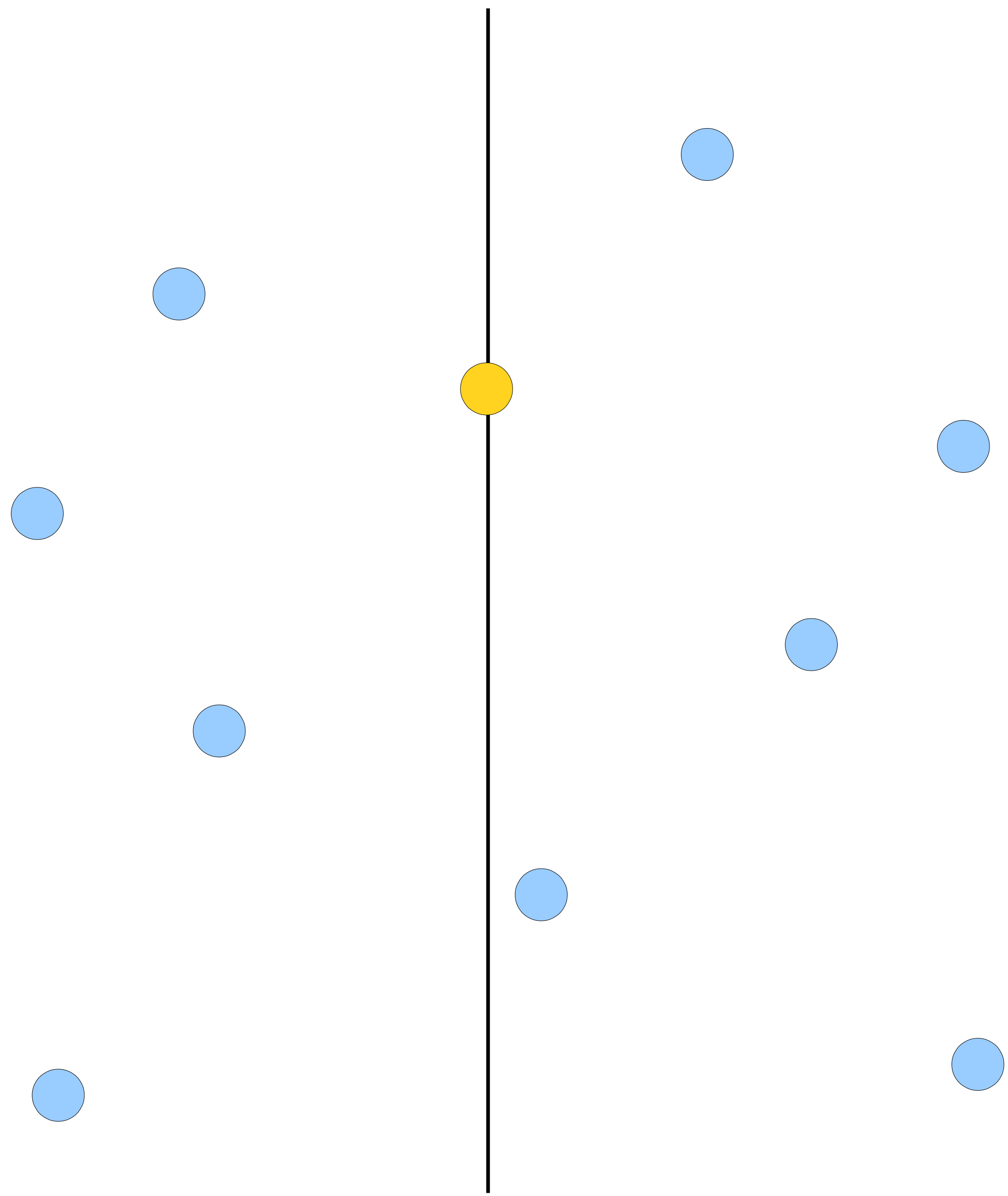


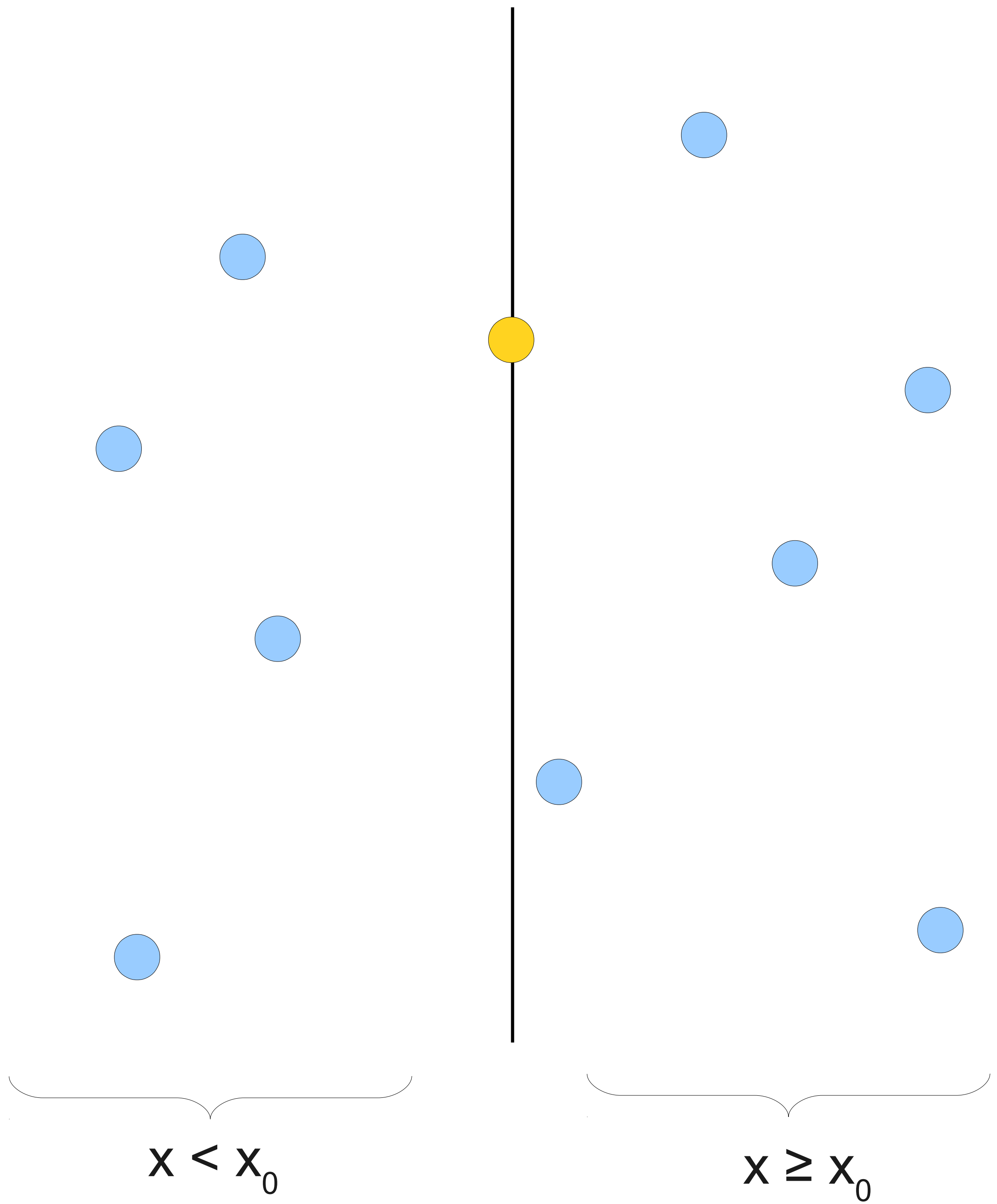


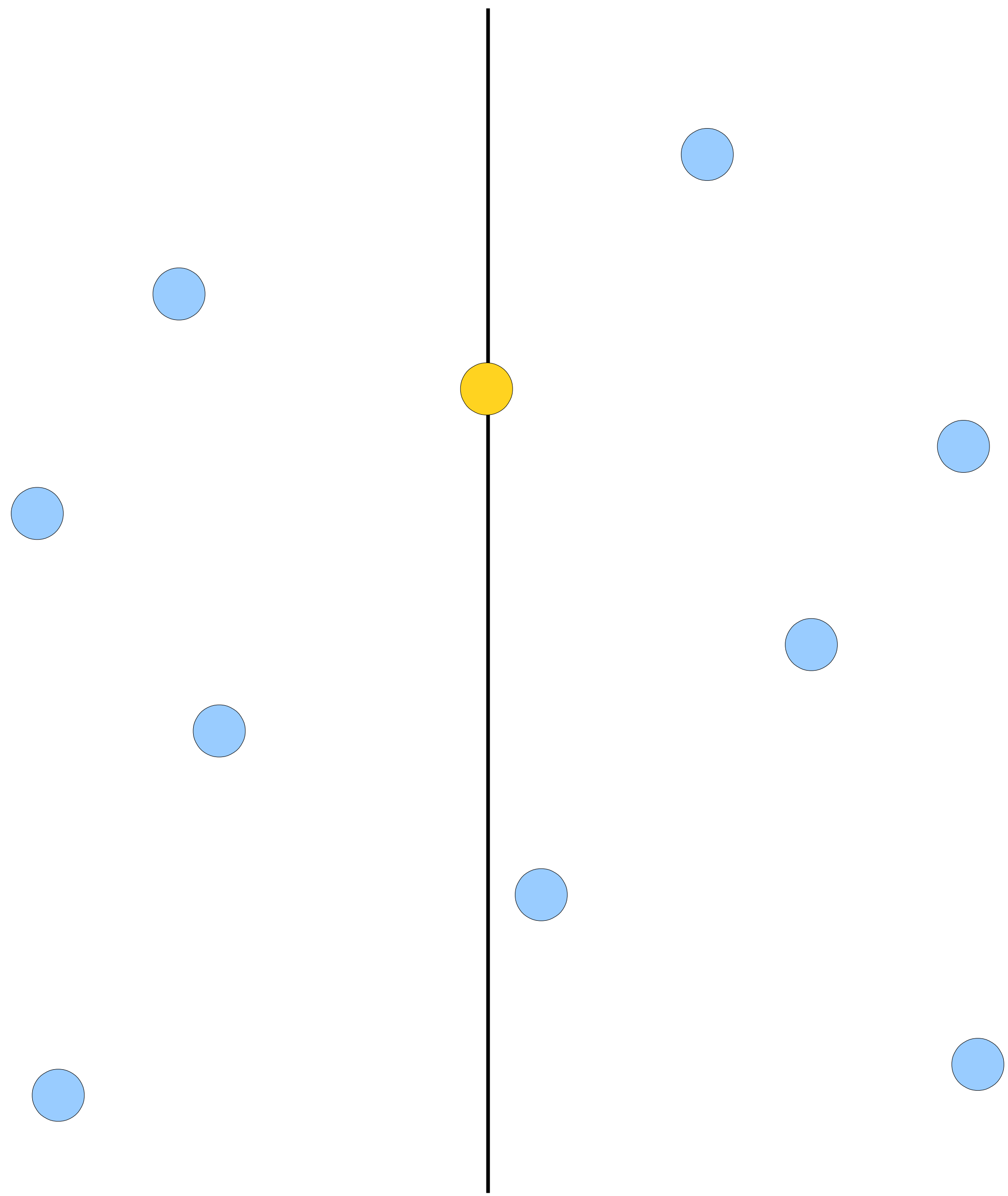

Upper half-space
Lower half-space

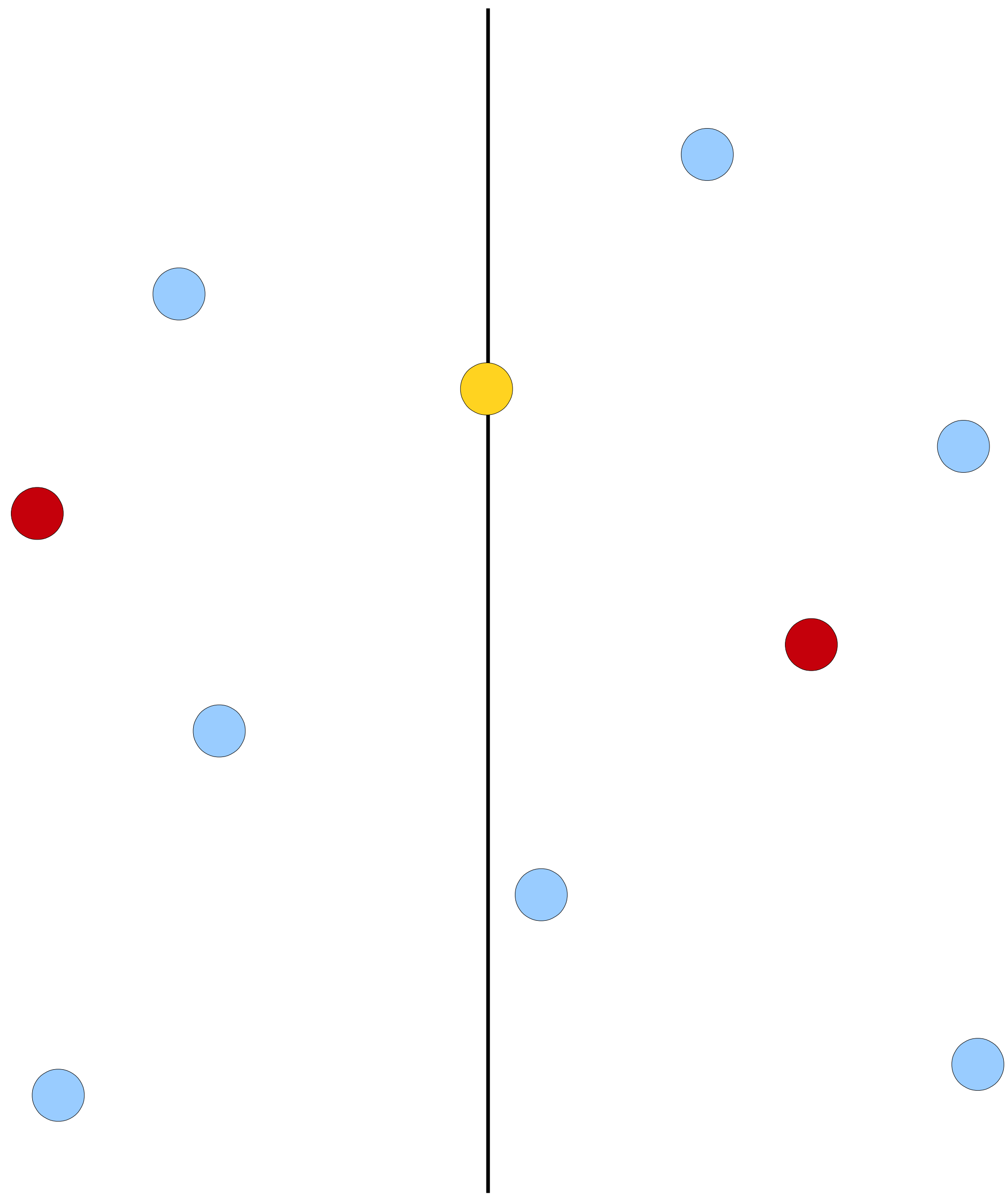


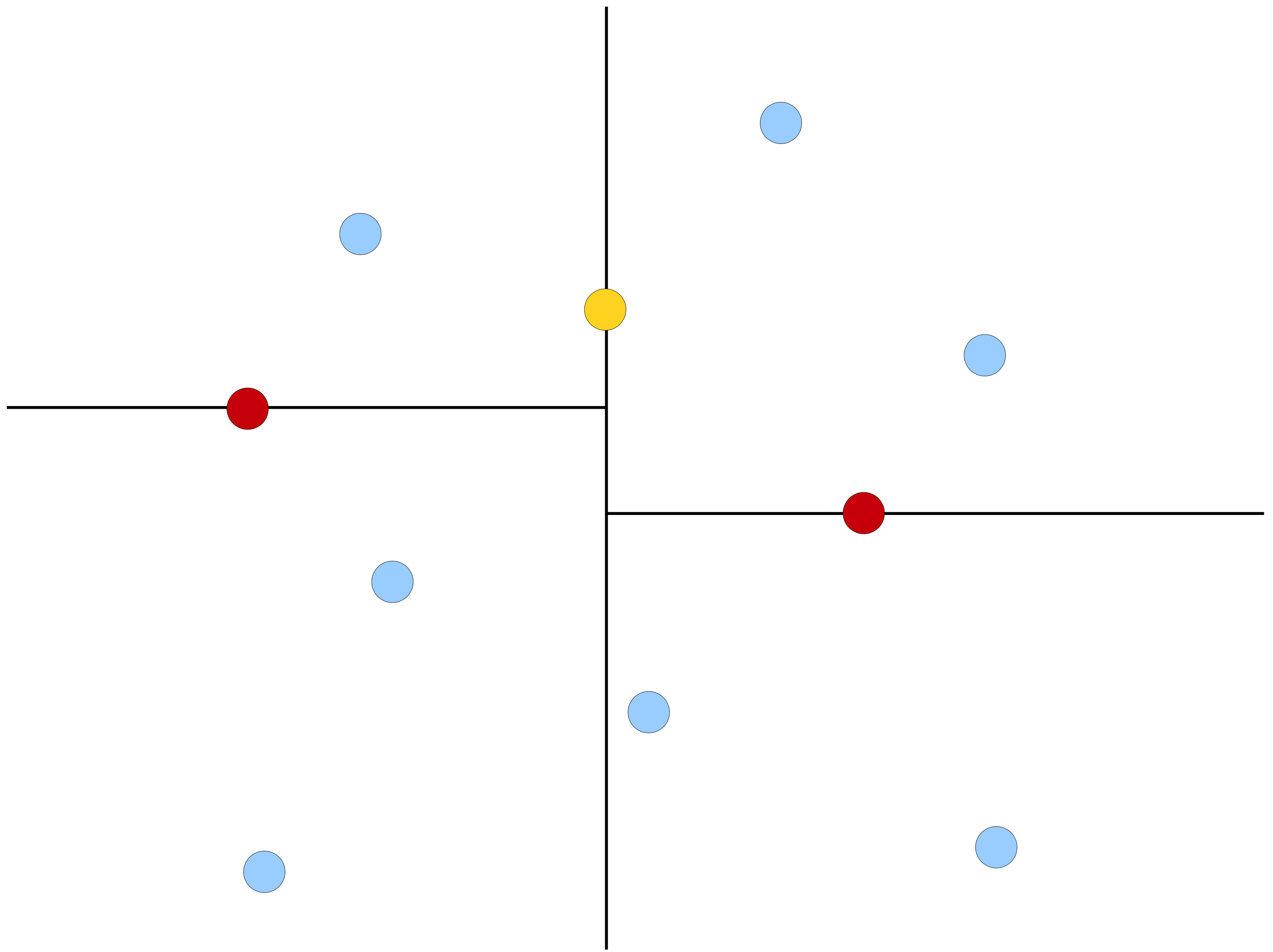


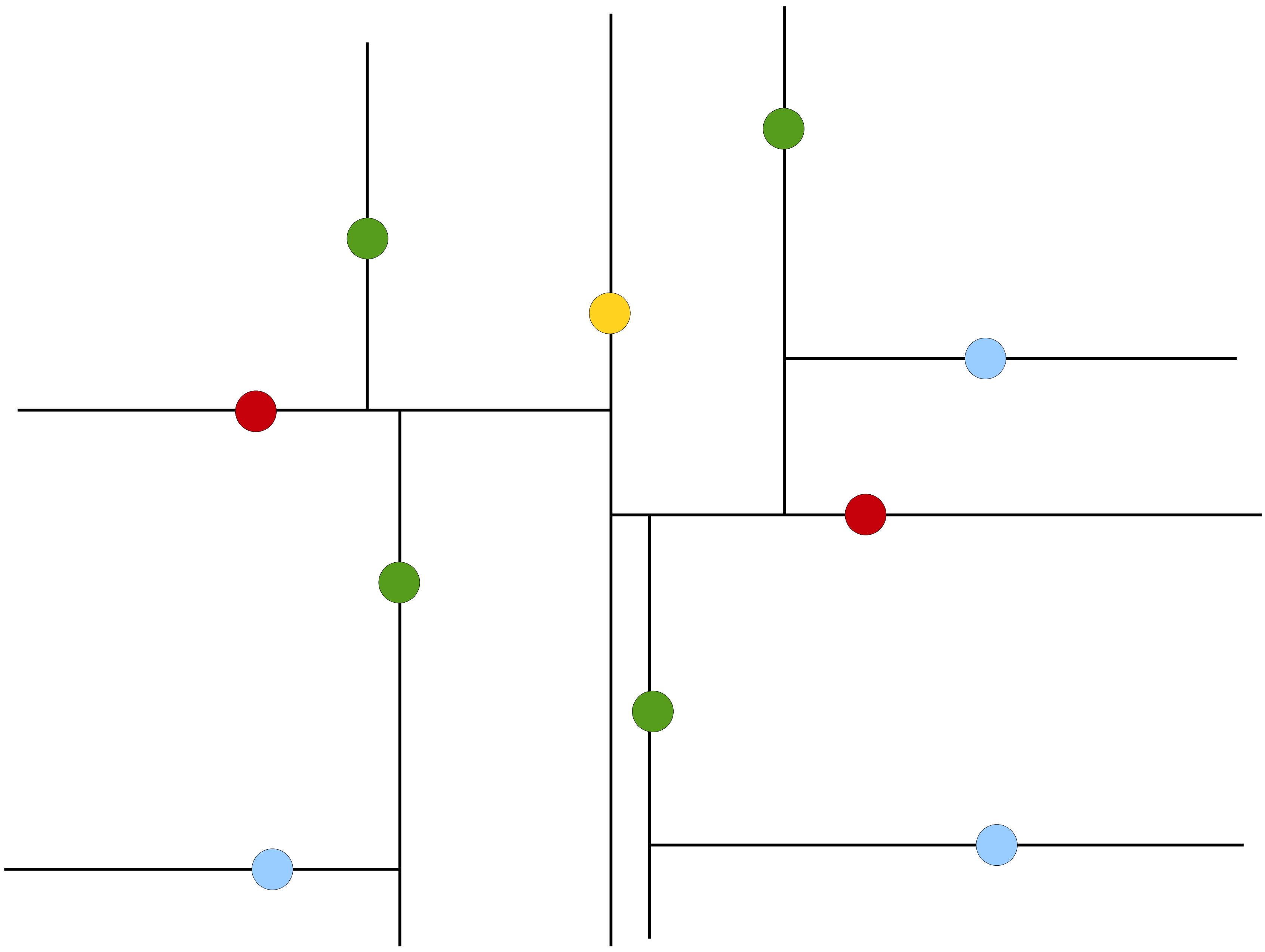


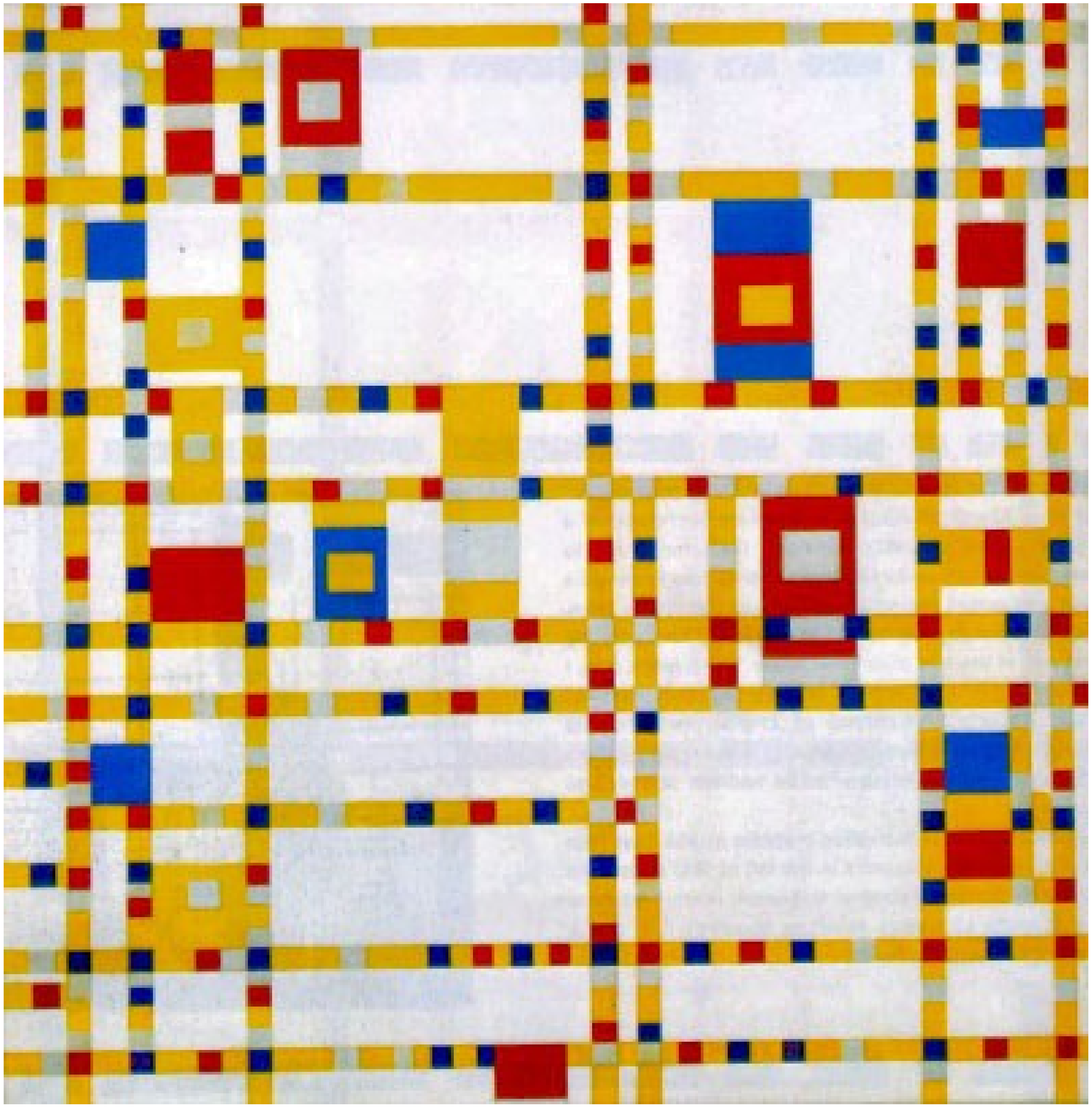

$x < x_0$
$x \geq x_0$



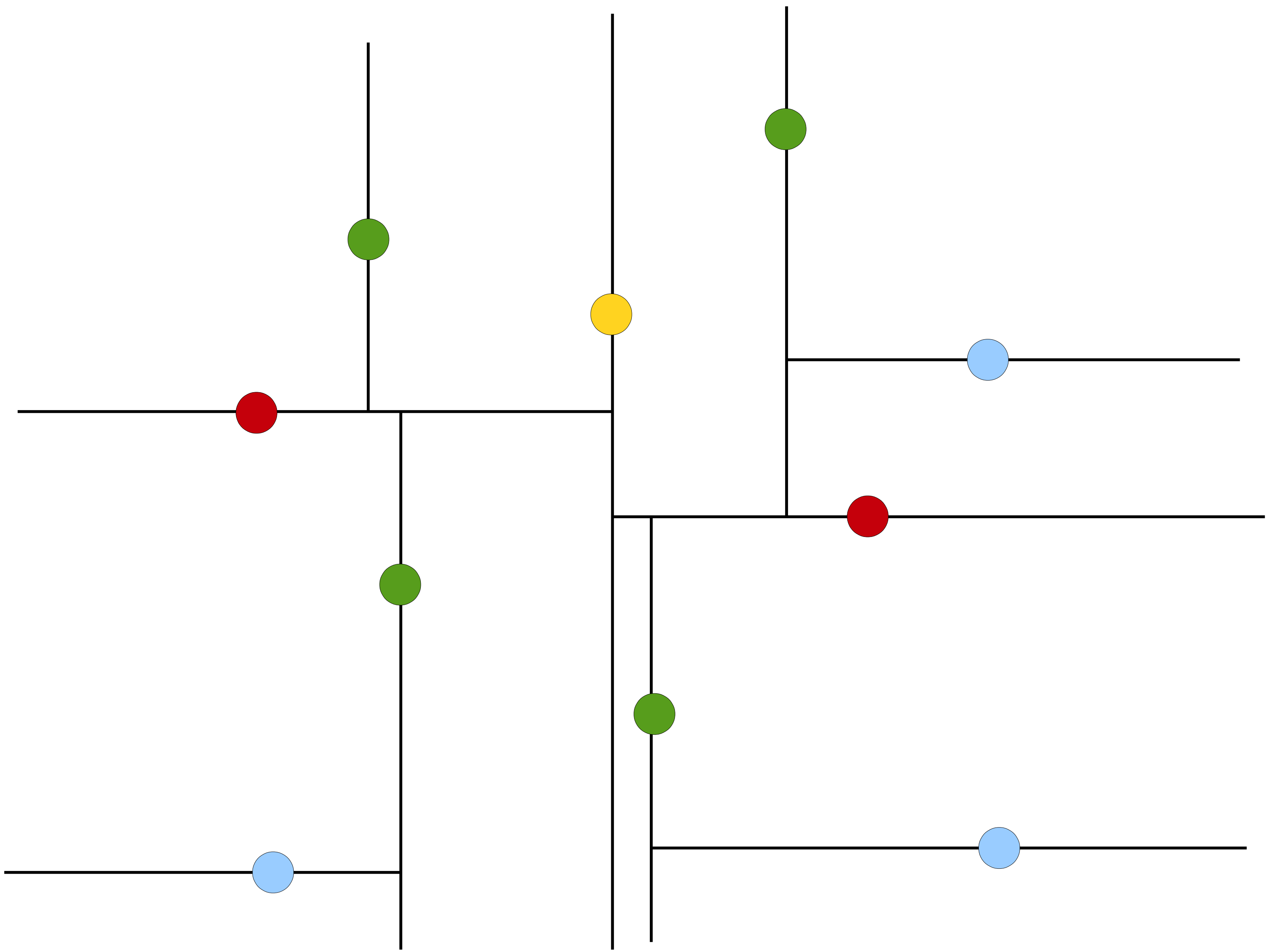


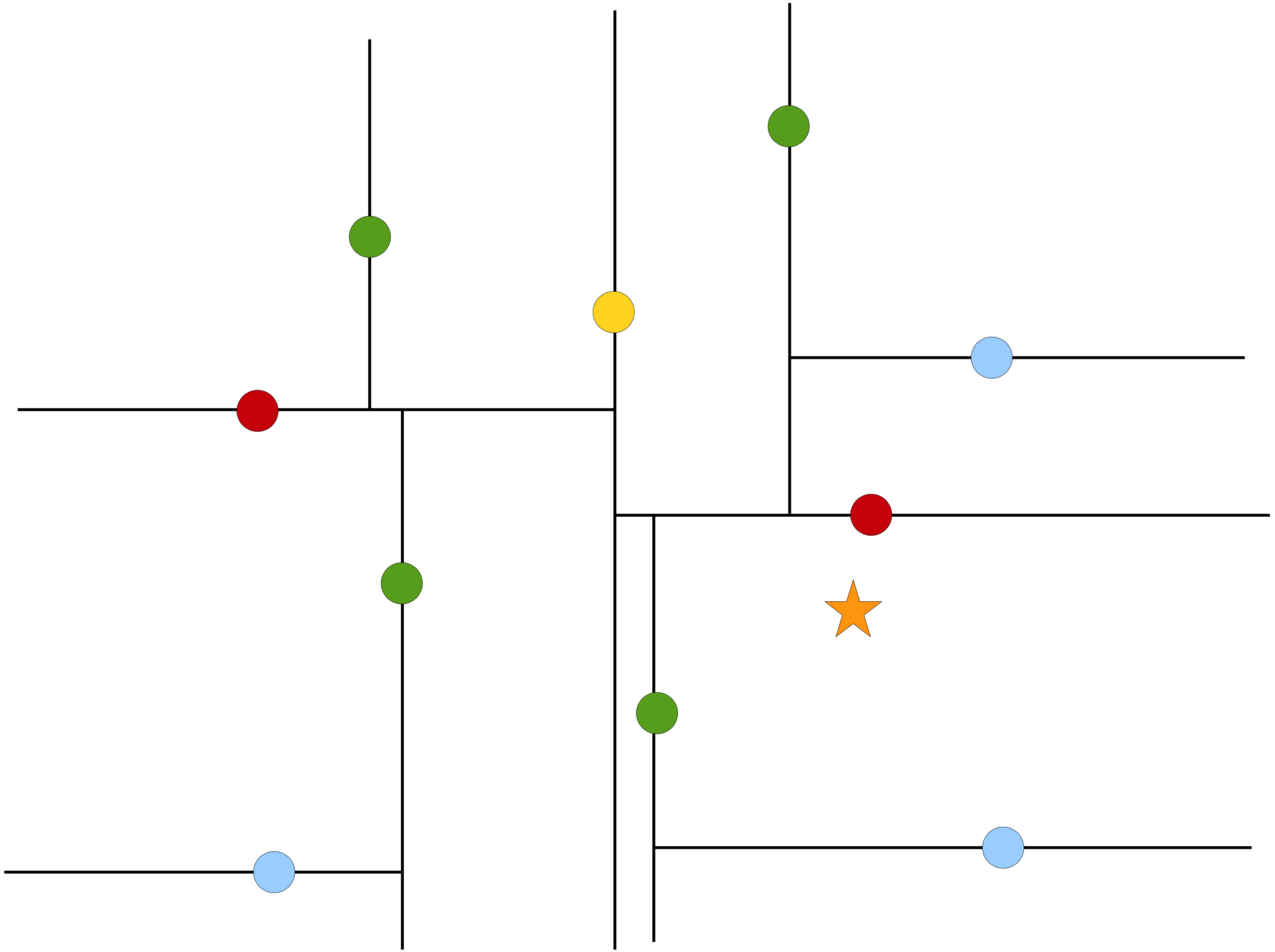


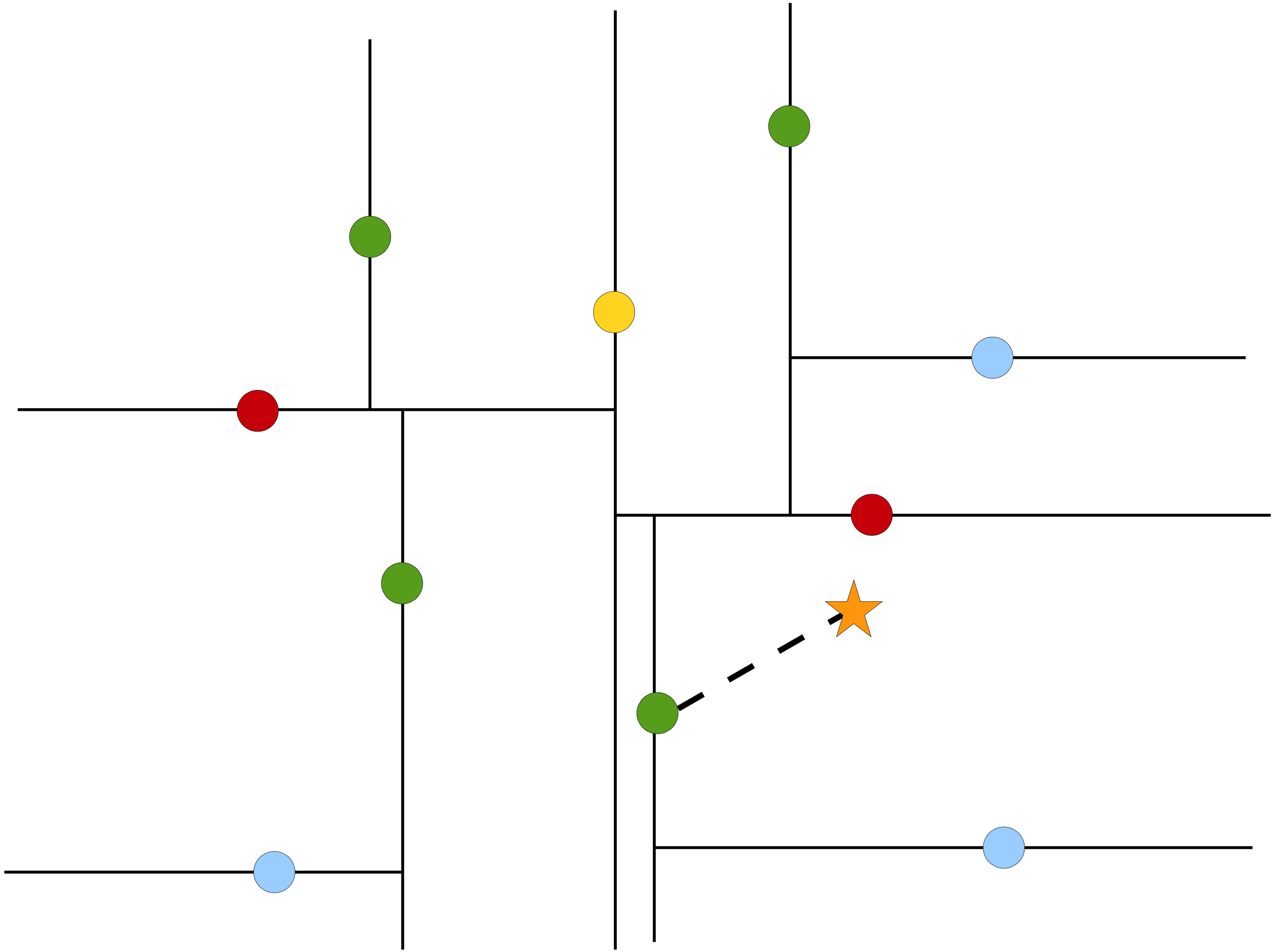


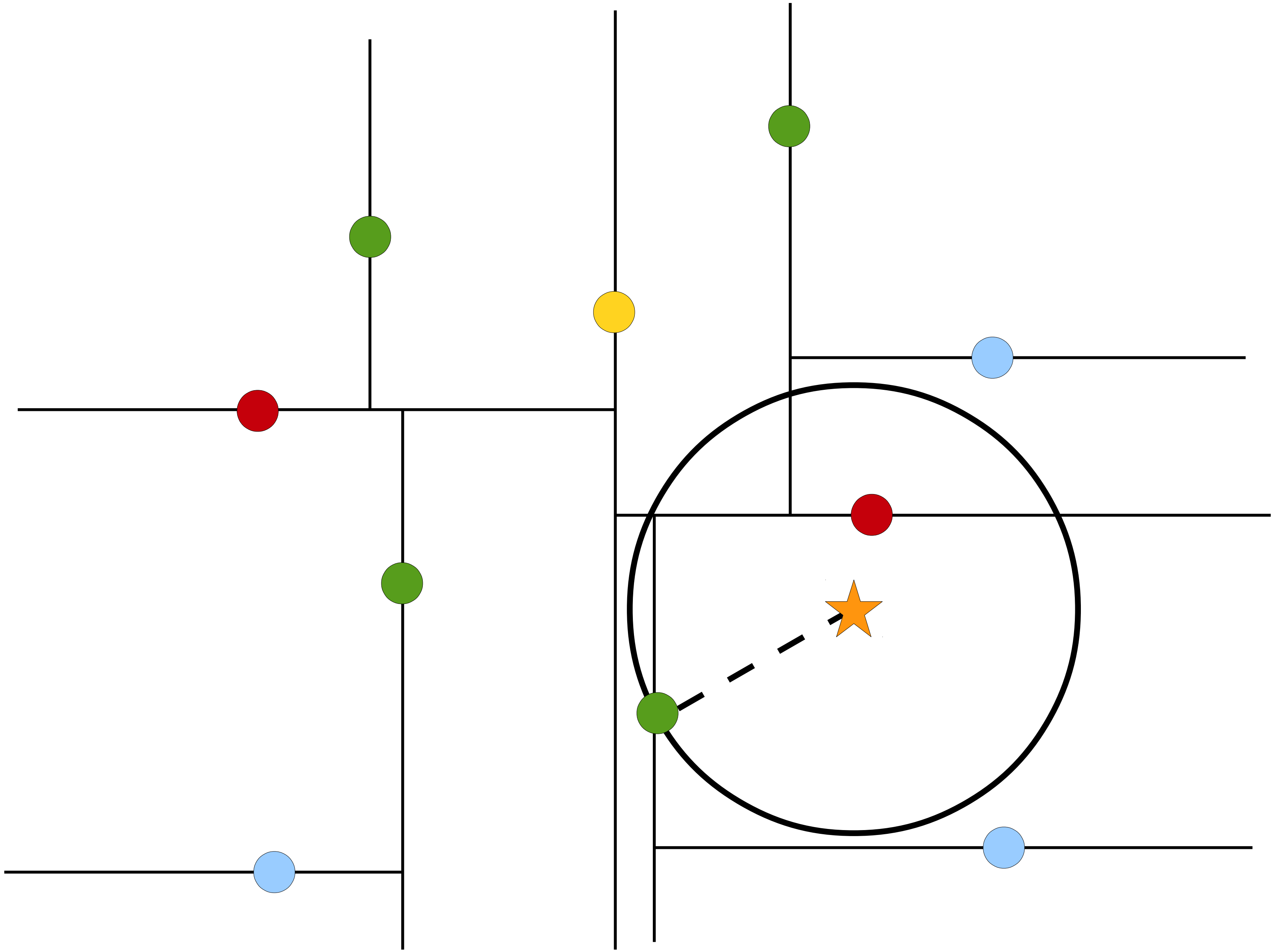


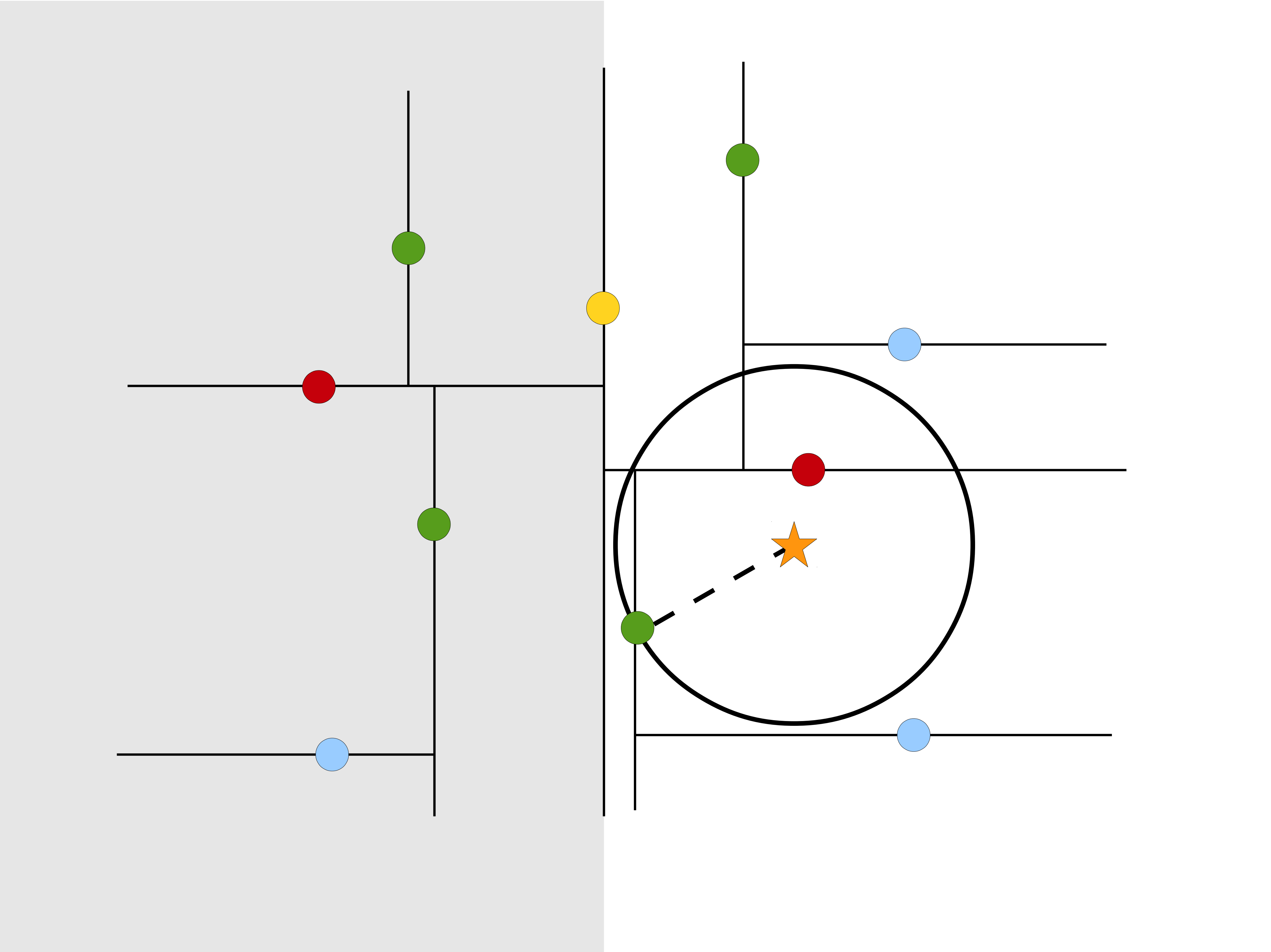
Nearest-Neighbor Lookup

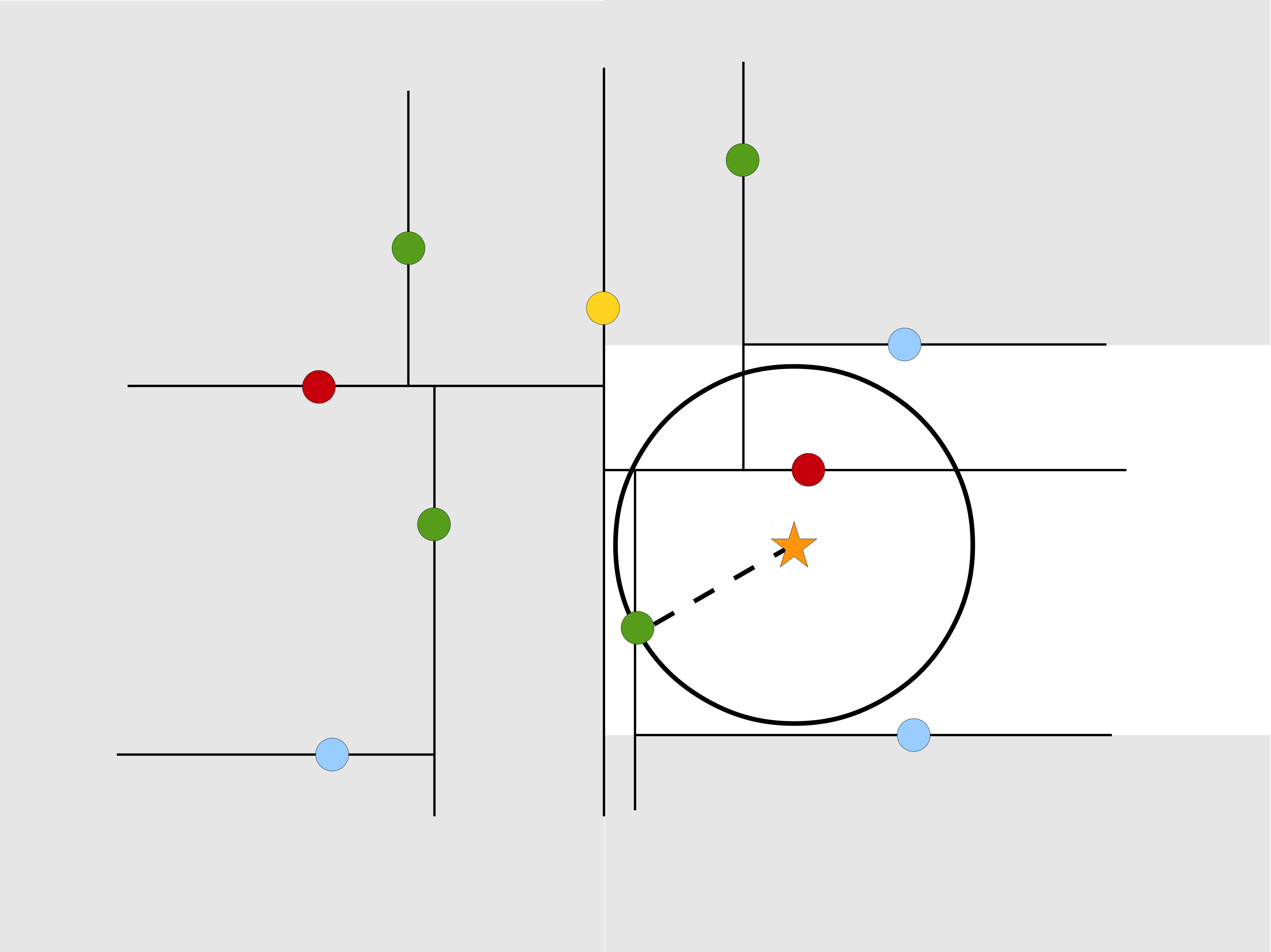


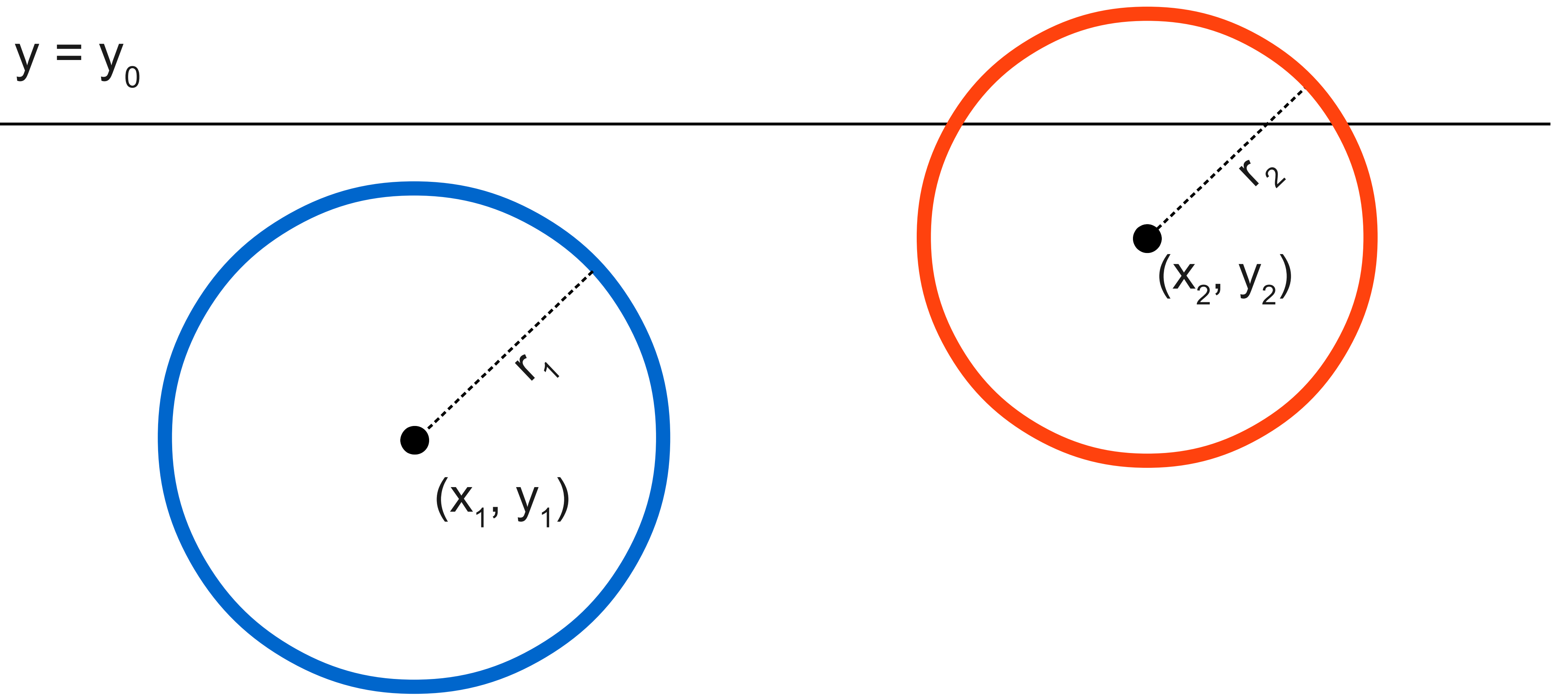




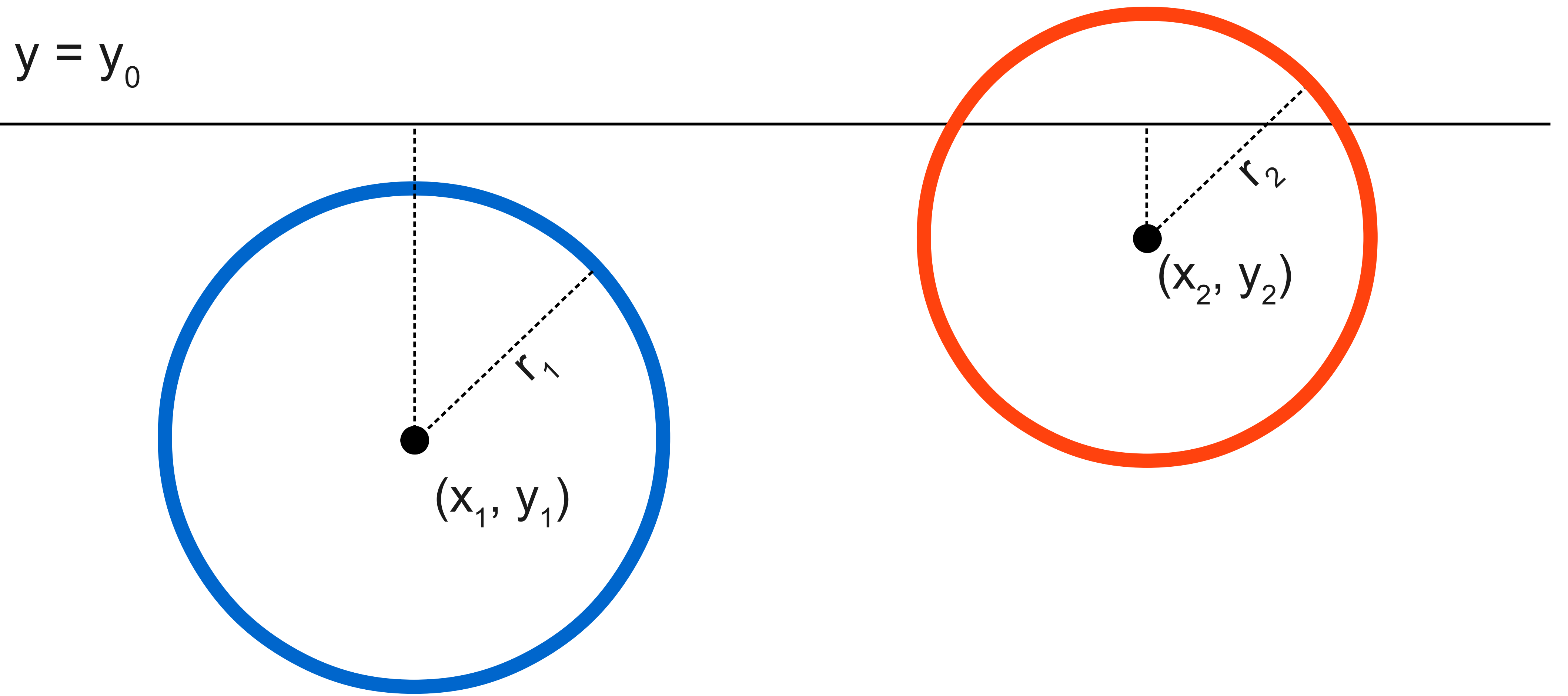




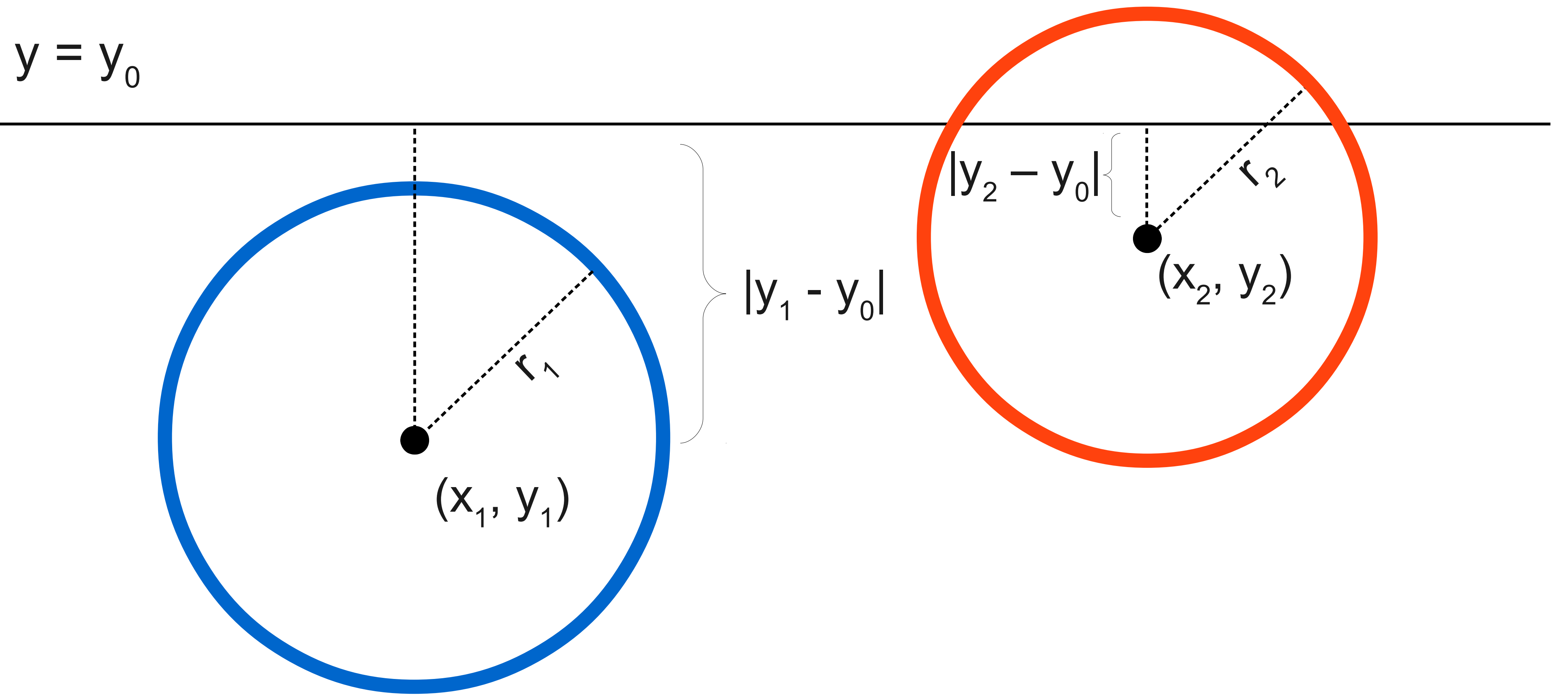




$y = y_0$
r_1
(x_1, y_1)
r_2
(x_2, y_2)



y = y0
r1
(x1, y1)
r2
(x2, y2)


y = y0
|y1 - y0|
r1
(x1, y1)
|y2 – y0|
r2
(x2, y2)

k-d Trees

- Assuming the points are nicely distributed, nearest-neighbor searches in k-d trees can run faster than O(n) time.
- Applications in computational geometry (collision detection), machine learning (nearest-neighbor classification), and many other places.

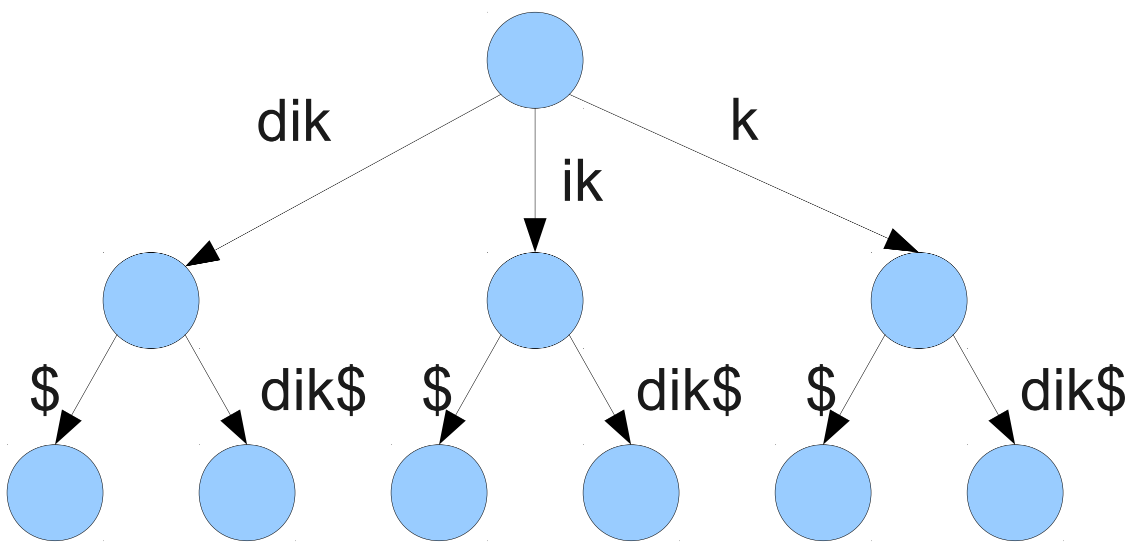
Suffix Trees

String Processing

- In computational biology, strings are enormously useful for storing DNA and RNA.
- Many important questions in biology can be addressed through string processing:
  - What is the most plausible evolutionary history of the following genomes?
  - Are there particular gene sequences that appear with high frequency within a genome?

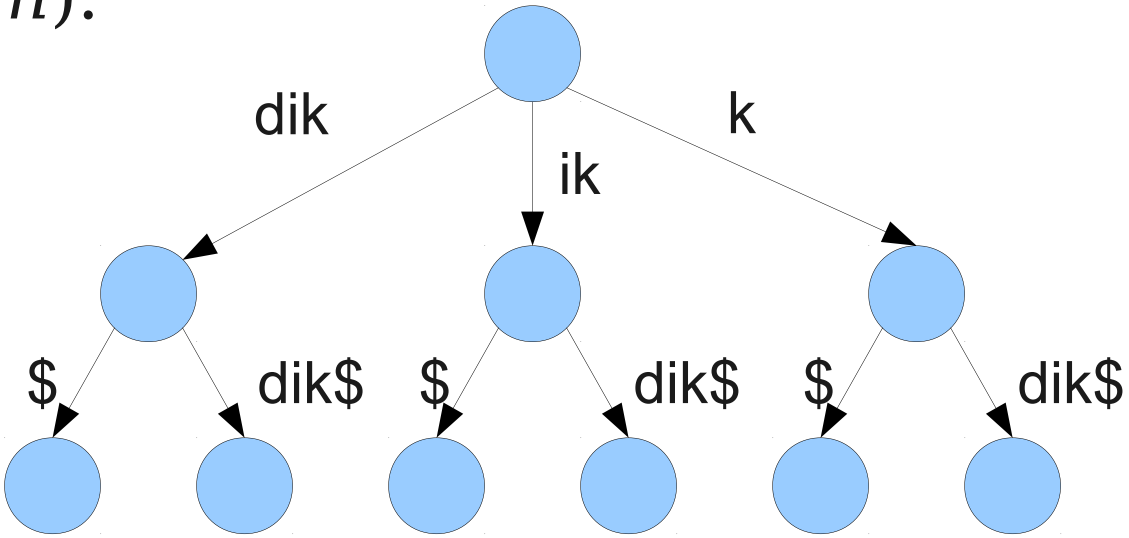
Suffix Trees

- A **suffix tree** is a (slightly modified) trie that stores all suffixes of a string *S*.
- Here is the suffix tree for “dikdik;” the $ is a marker for “end-of-string.”



Suffix Trees

- Important, nontrivial, nonobvious fact: A suffix tree for a string of *n* characters can be built in time O(*n*).
- Given a string of length *m*, we can determine whether it is a substring of the original string in time O(*m*).



Suffix Trees

- Other applications of suffix trees:
  - Searching for one genome within another allowing for errors, insertions, and deletions in time $O(n + m)$.
  - Finding the longest common substring of two sequences in time $O(n + m)$.
  - Improving the performance of data compression routines by finding long repeated strings efficiently.

Bloom Filters

Distributing Data

- Websites like Google and Facebook deal with *enormous* amounts of data.
- Probably measured in hundreds of millions of gigabytes (hundreds of *petabytes*).
- There is absolutely no way to store this on one computer.
- Instead, data must be stored on multiple computers networked together.

Looking up Data

- Suppose you are at Google implementing search.
- When you get a search query, you have to be able to know which computer knows what pages to display for that query.
- Network latency is, say, 2ms between you and each computer.
- If you have one thousand computers to search, you can't just query each one and ask.

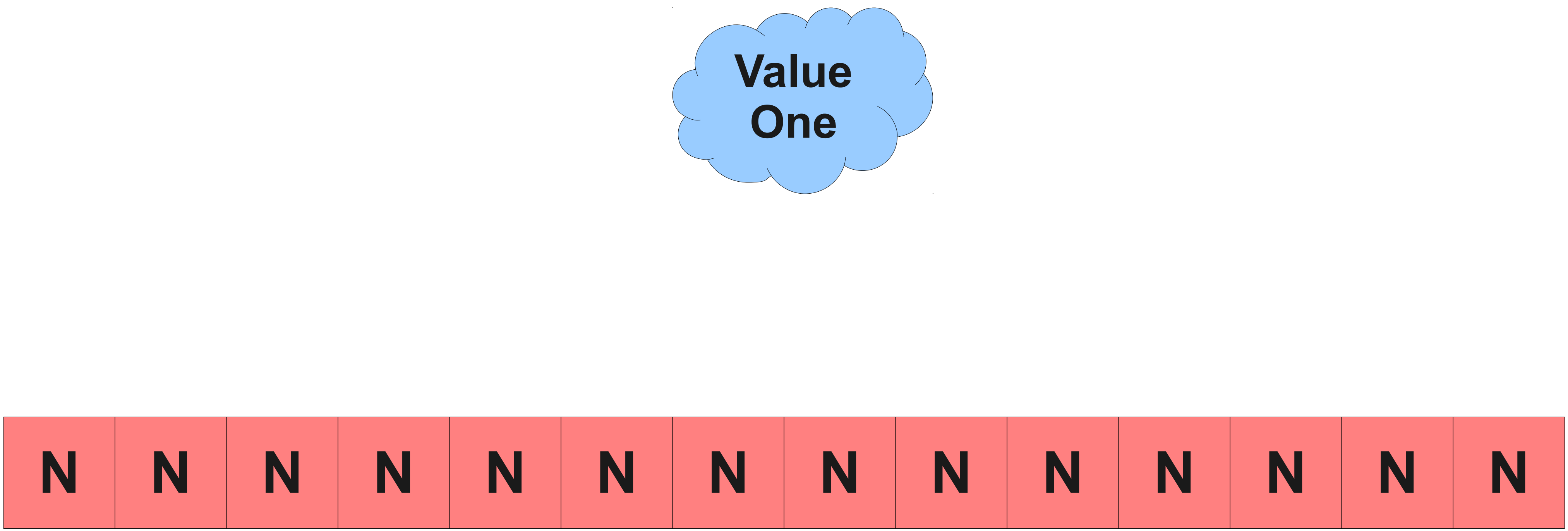
Bloom Filters

- A **Bloom filter** is a data structure similar to a set backed by a hash table.
- Stores a set of values in a way that may lead to false positives:
  - If the Bloom filter says that an object is not present, it is definitely not present.
  - If the Bloom filter says that an object is present, it may actually not be present.

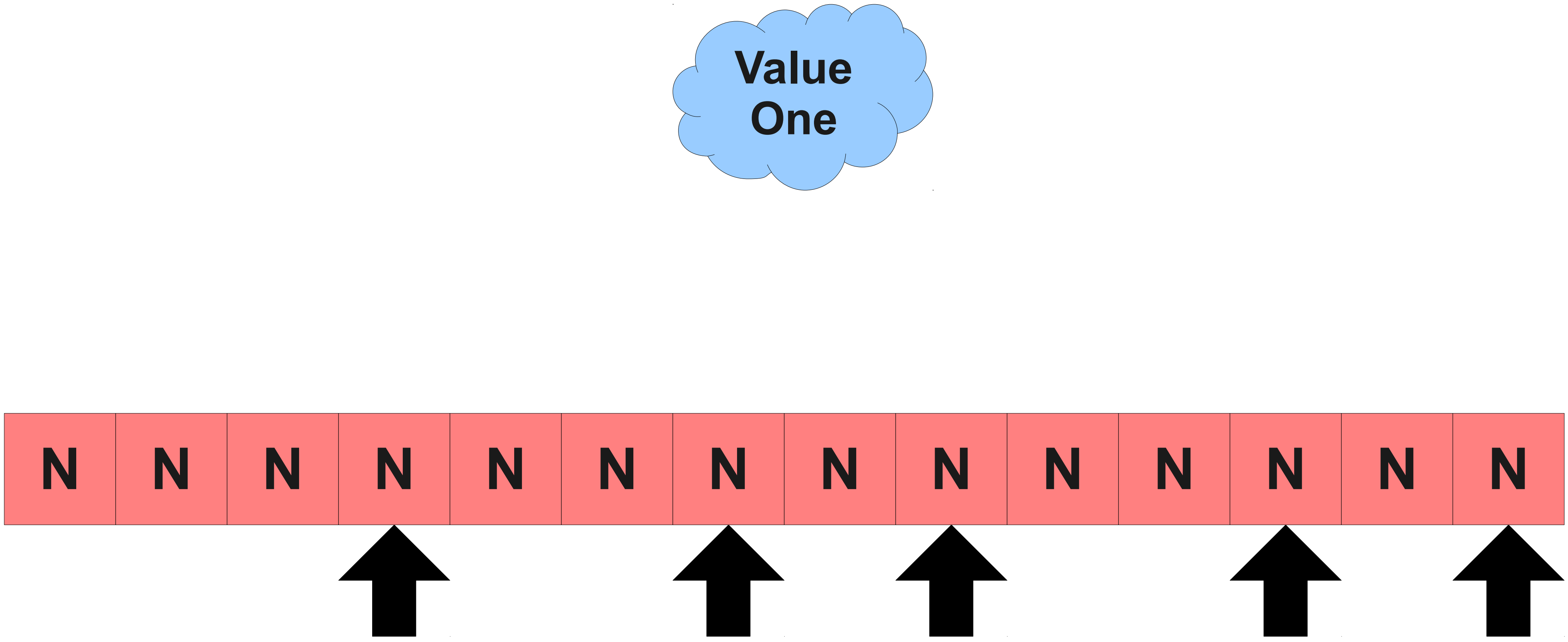
Bloom Filters

N	N	N	N	N	N	N	N	N	N	N	N	N	N

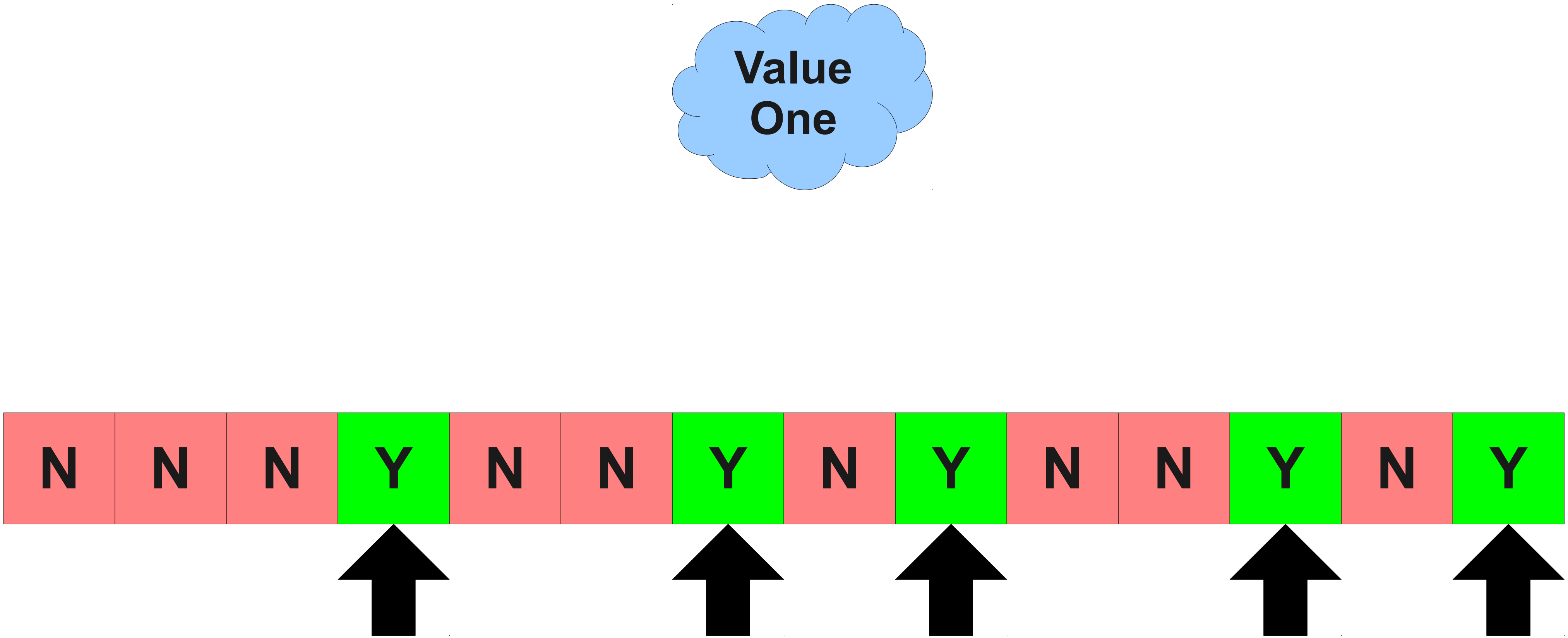
Bloom Filters



Bloom Filters



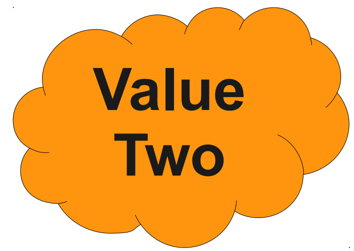
Bloom Filters



Bloom Filters

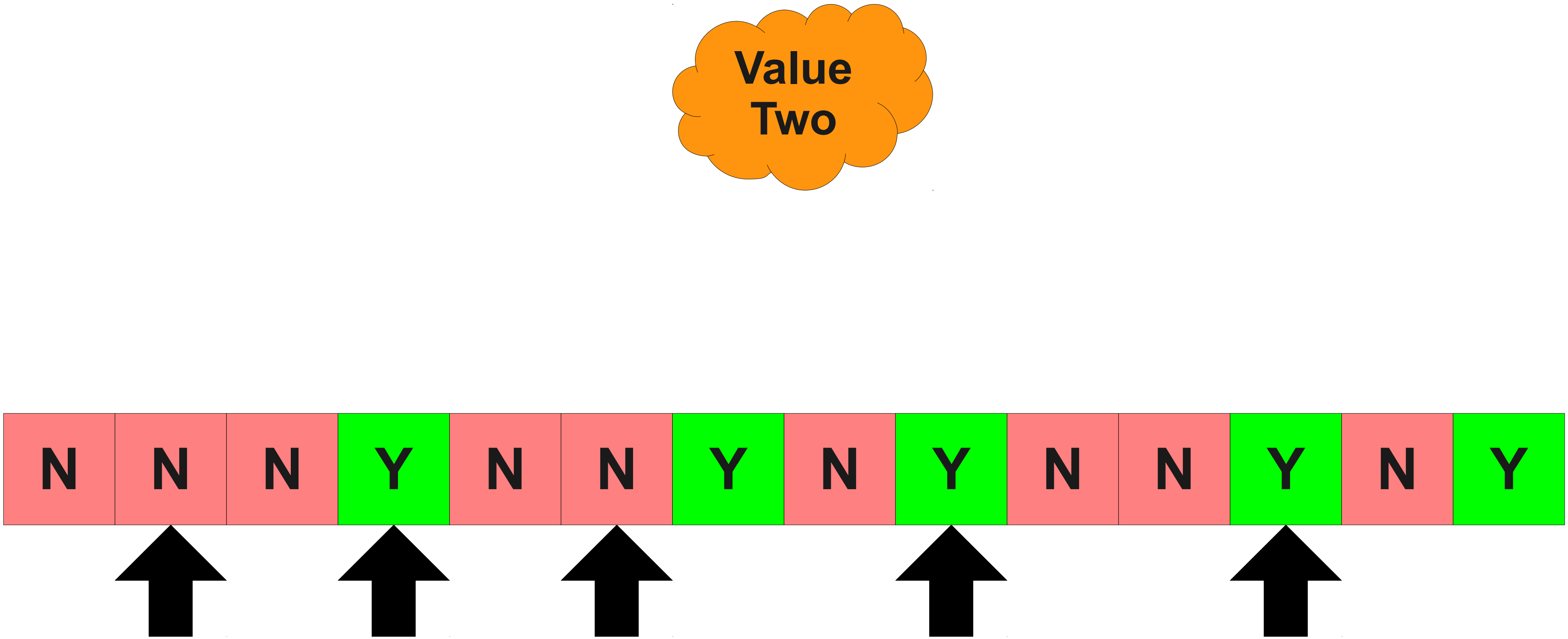
N	N	N	Y	N	N	Y	N	Y	N	N	Y	N	Y

Bloom Filters

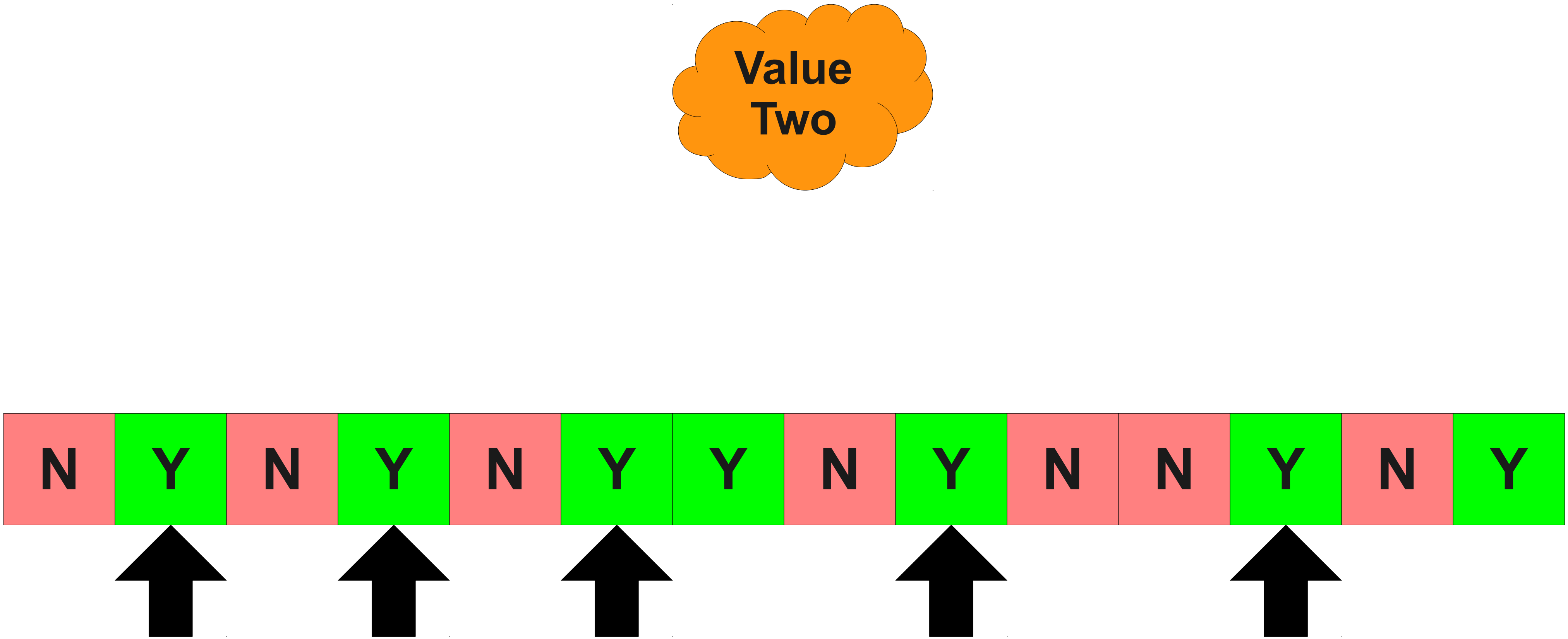


N	N	N	Y	N	N	Y	N	Y	N	N	Y	N	Y

Bloom Filters



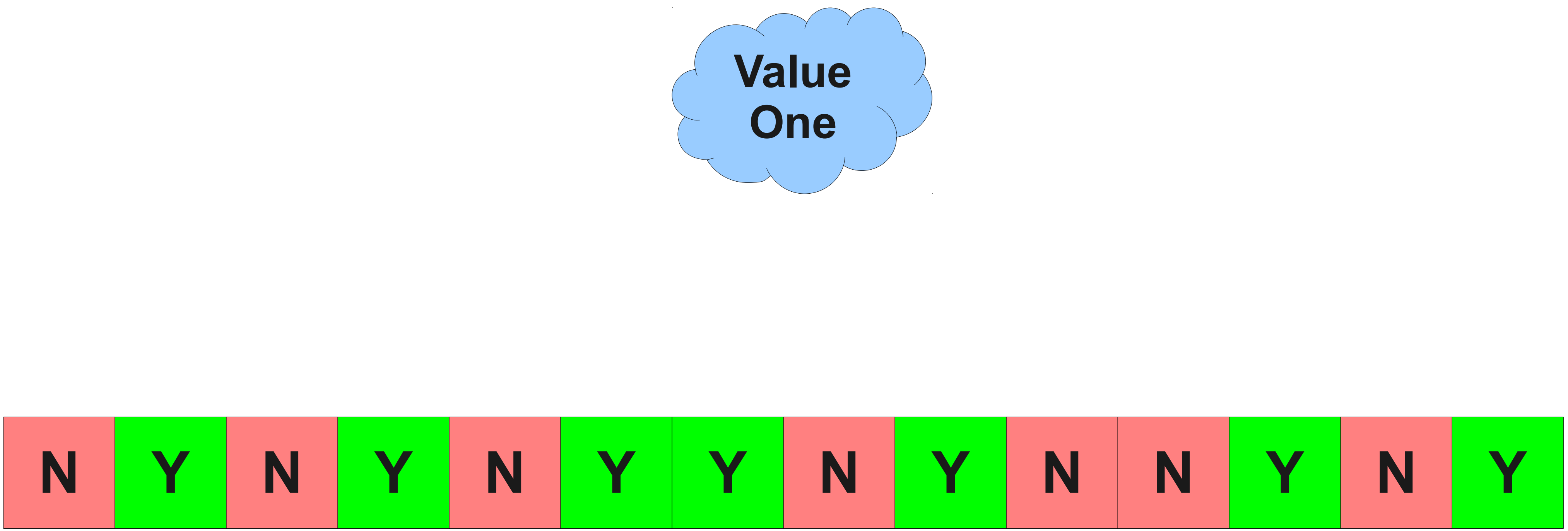
Bloom Filters



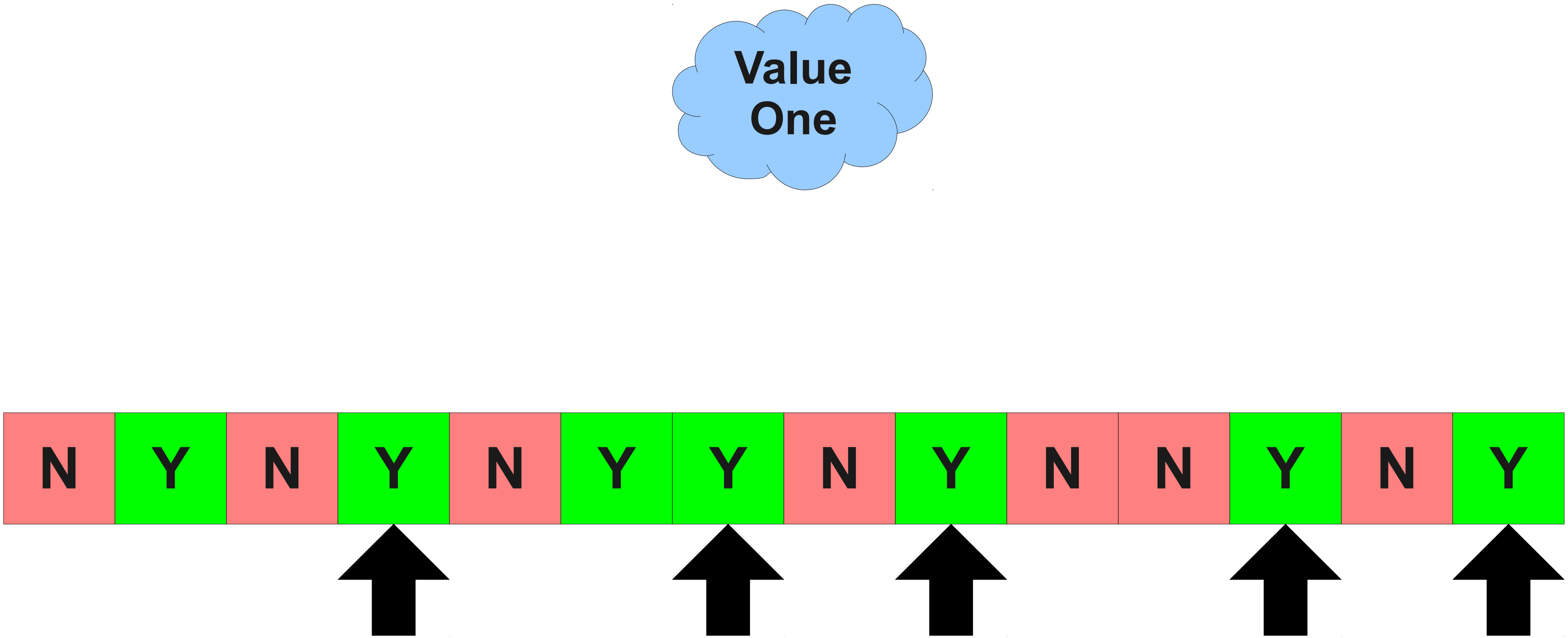
Bloom Filters

N	Y	N	Y	N	Y	Y	N	Y	N	N	Y	N	Y

Bloom Filters



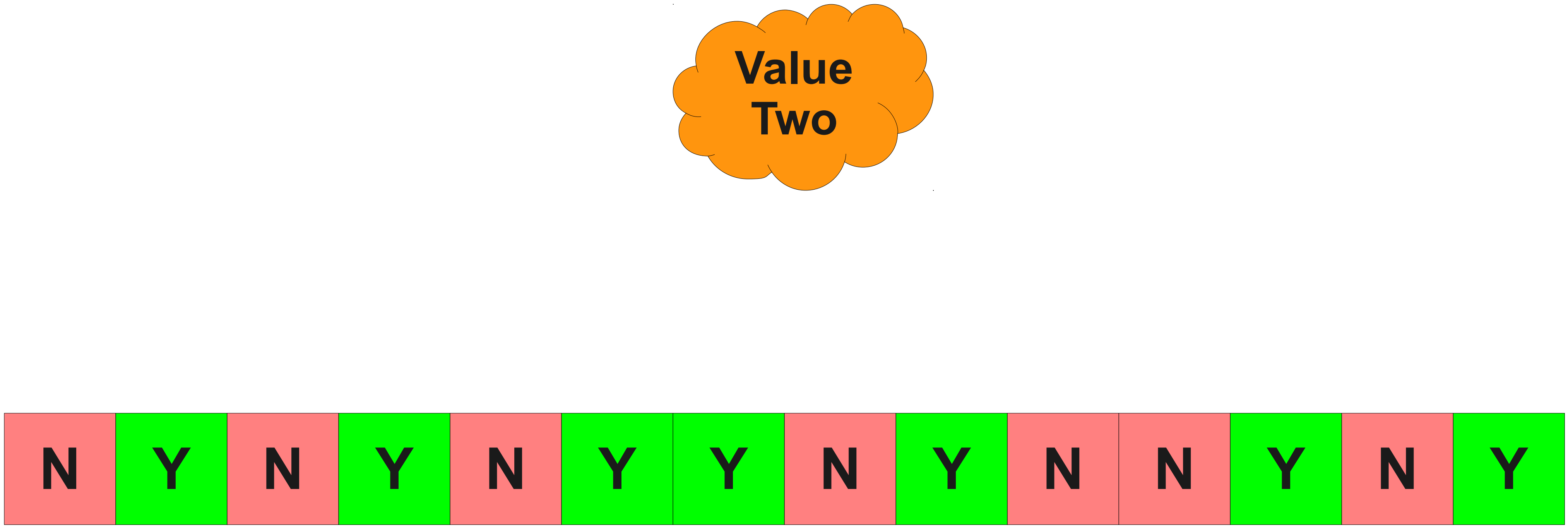
Bloom Filters



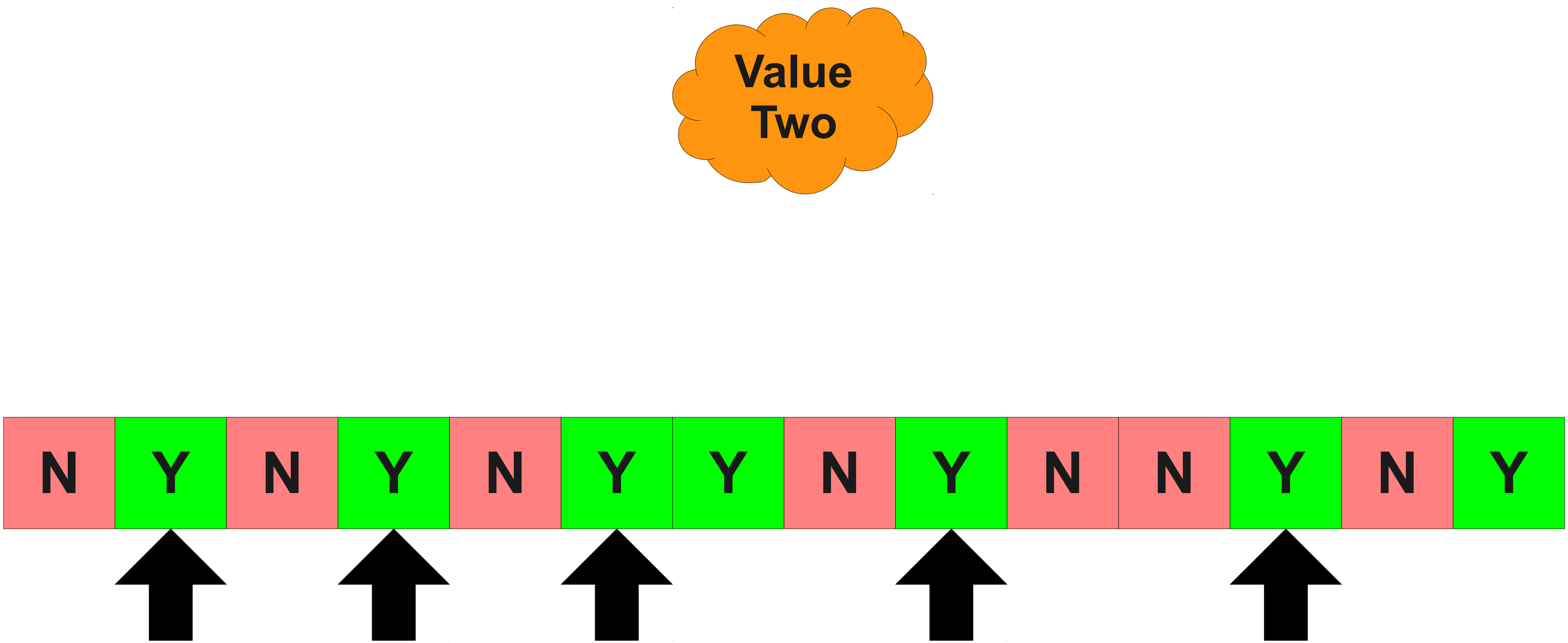
Bloom Filters

N	Y	N	Y	N	Y	Y	N	Y	N	N	Y	N	Y

Bloom Filters



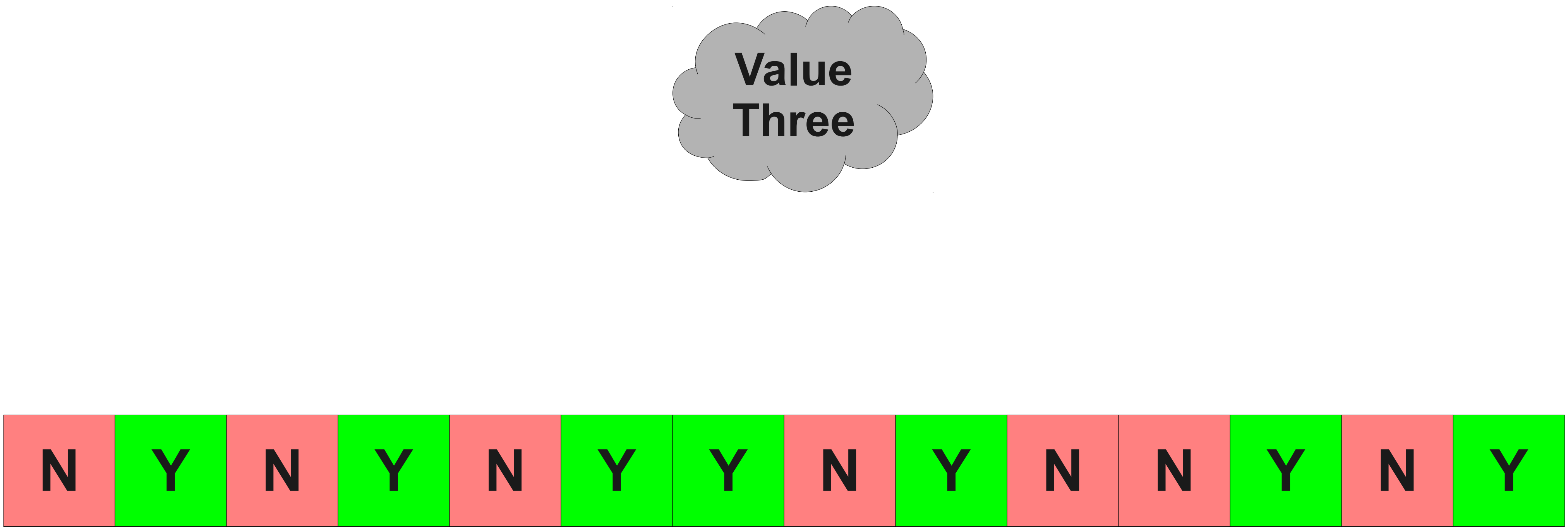
Bloom Filters



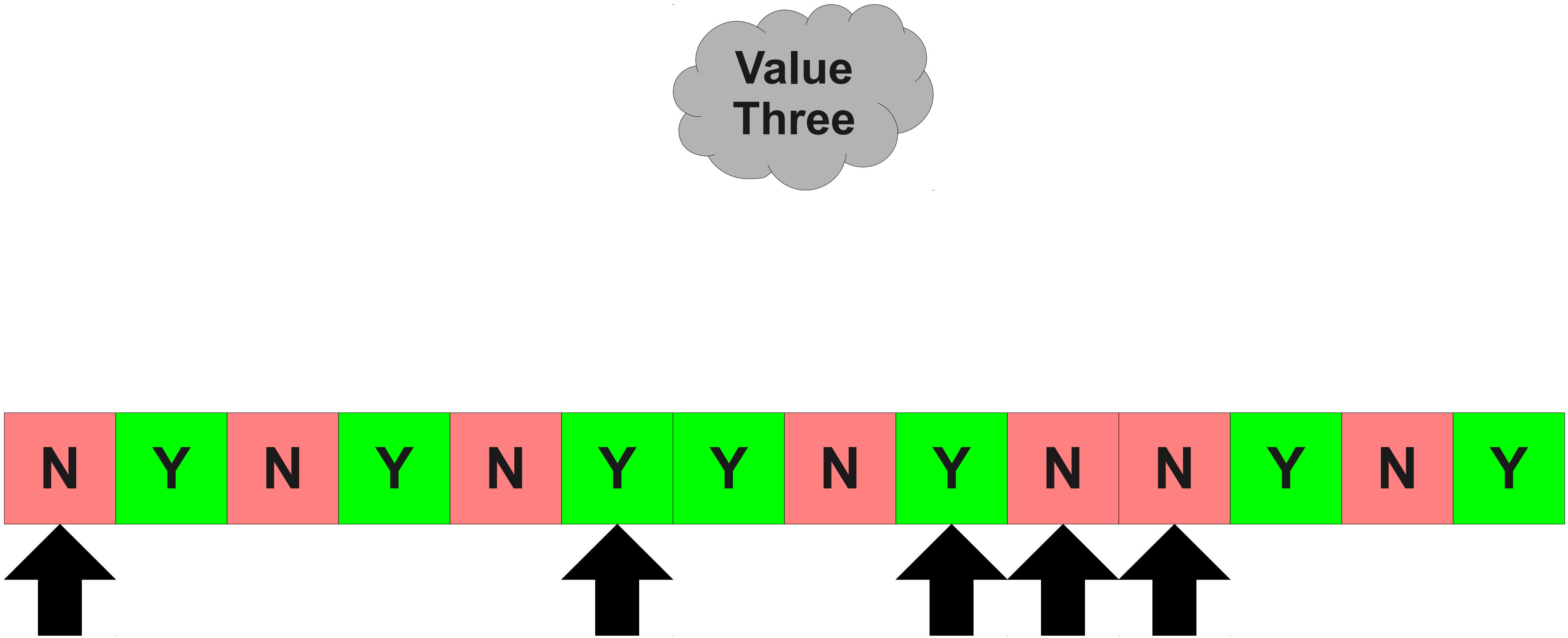
Bloom Filters

N	Y	N	Y	N	Y	Y	N	Y	N	N	Y	N	Y

Bloom Filters



Bloom Filters



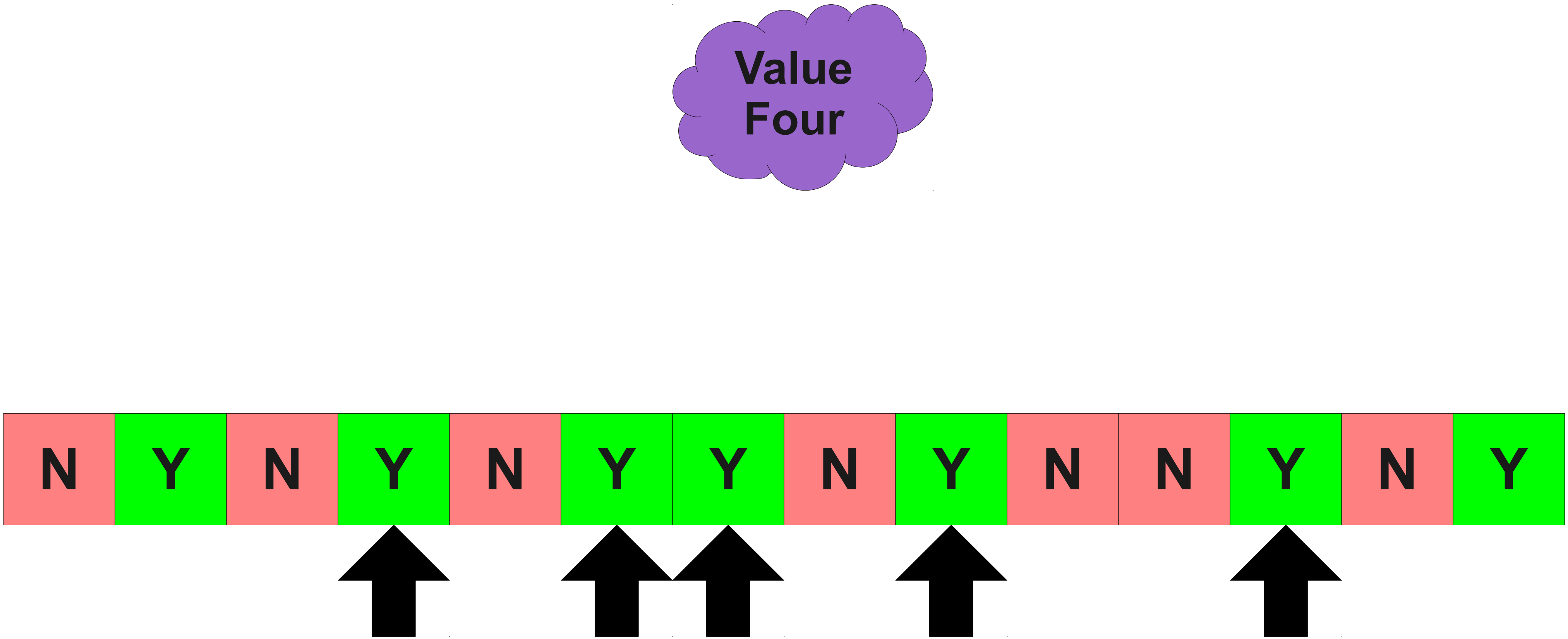
Bloom Filters

N	Y	N	Y	N	Y	Y	N	Y	N	N	Y	N	Y

Bloom Filters

N	Y	N	Y	N	Y	Y	N	Y	N	N	Y	N	Y

Bloom Filters



Bloom Filters and Networks

- Bloom filters can be used to mitigate the networking problem from earlier.
- Have each computer store a Bloom filter of what's stored on each other computer.
- To determine which computer has some data:
  - Look up that value in each Bloom filter.
  - Call up just the computers that might have it.
- Since Bloom filter lookup is substantially faster than a network query (probably 1000-10,000x), this solution is used extensively in practice.

Data structures make it possible to solve important problems at scale.

You get to decide which problems we'll be using them for.