Associative Graph Data Structures AGDS with an Efficient Access via AVB+trees

Adrian Horzyk

horzyk@agh.edu.pl

AGH

AGH University of Science and Technology Krakow, Poland

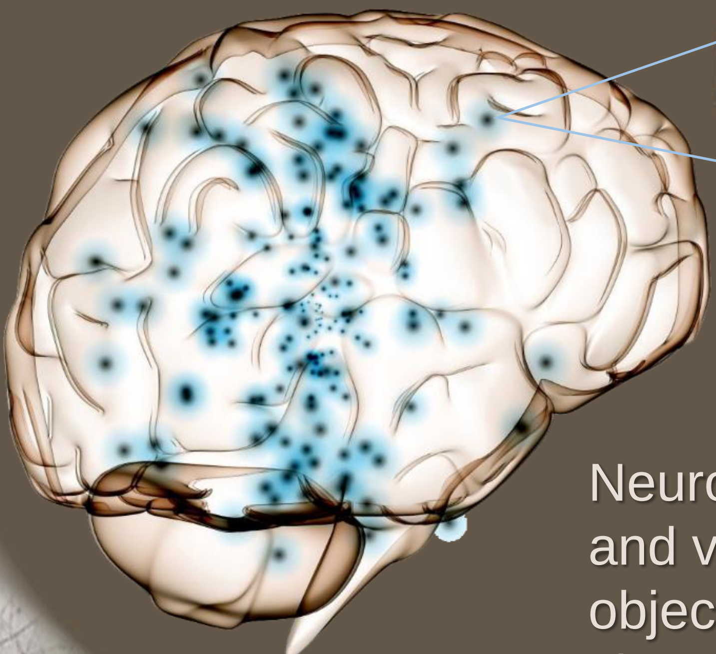
Brains and Neurons

How do they really work?

How we can use brain-like structures
to make computations
more efficient and intelligent?

Brain Structures

Brains consist of complex graphs of connected neurons and other elements.



Neurons and their connections represent input data and various relations between them, defining objects and similarities, proximities, sequence, chronology, context, and establishing causal relationships between them.

Why the brain structures look so complex and irregular?

Data Tables

In computer science, we mostly use tables to store, organize and manage data,

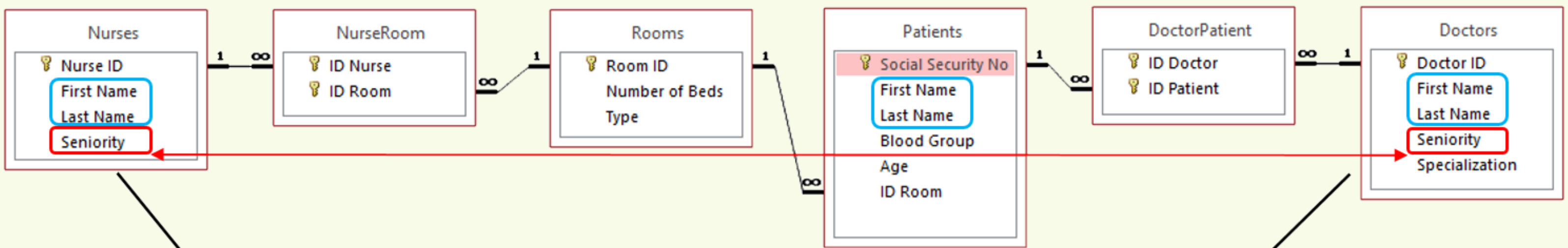
	ATTRIBUTES				
SAMPLE OBJECTS	SEPAL LENGTH	SEPAL WIDTH	PETAL LENGTH	PETAL WIDTH	CLASS LABEL
O1	5.4	3.0	4.5	1.5	Versicolor
O2	6.3	3.3	4.7	1.6	Versicolor
O3	6.0	2.7	5.1	1.6	Versicolor
O4	6.7	3.0	5.0	1.7	Versicolor
O5	6.0	2.2	5.0	1.5	Virginica
O6	5.9	3.2	4.8	1.8	Versicolor
O7	6.0	3.0	4.8	1.8	Virginica
O8	5.7	2.5	5.0	2.0	Virginica
O9	6.5	3.2	5.1	2.0	Virginica

but common relations like identity, similarity, neighborhood, minima, maxima, number of duplicates must be found. The more data we have the bigger time loss we face!

Such relations are not enough!

Relational Databases

Relational databases relate stored data only horizontally, not vertically, so we still have to search for duplicates, neighbor or similar values and objects.



Nurses

Nurse ID	First Name	Last Name	Seniority
N1	Amy	Moon	12
N2	Rose	Jolie	18
N3	Kate	Ford	24
N4	Lisa	Brown	9
N5	Sara	Pitt	4
N6	Kate	Lopez	12

Doctors

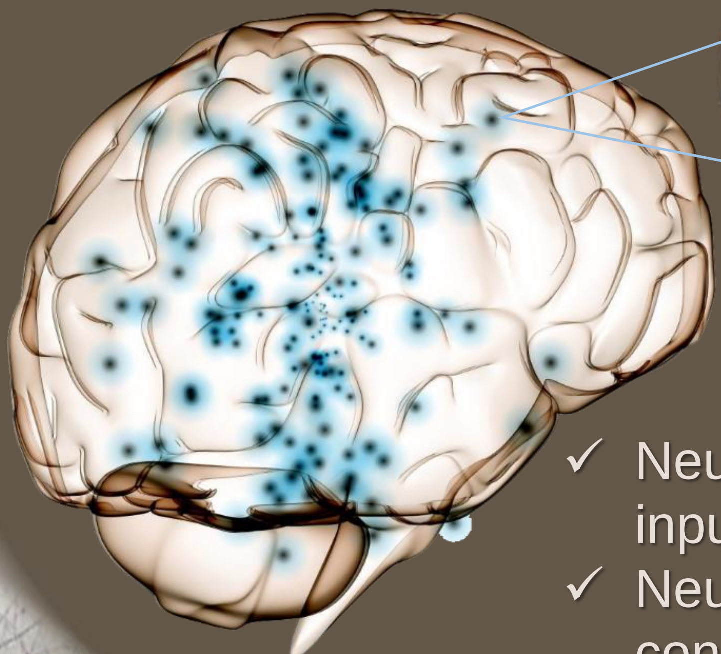
Doctor ID	First Name	Last Name	Seniority	Specialization
D1	Tom	Hanks	18	orthopedics
D2	Jack	Brown	15	surgery
D3	Lisa	Ford	23	pediatrician
D4	Tom	Trump	35	pediatrician
D5	Kate	Smith	7	surgery
D6	Amy	Hanks	12	surgery

Even horizontally, data are not related perfectly and many duplicates of the same categories occur in various tables which are not related anyhow. In result, we need to lose a lot of computational time to search out necessary data relations to compute results or make conclusions.

Is it wise to lose the majority of the computational time for searching for data relations?!

Data Relationships

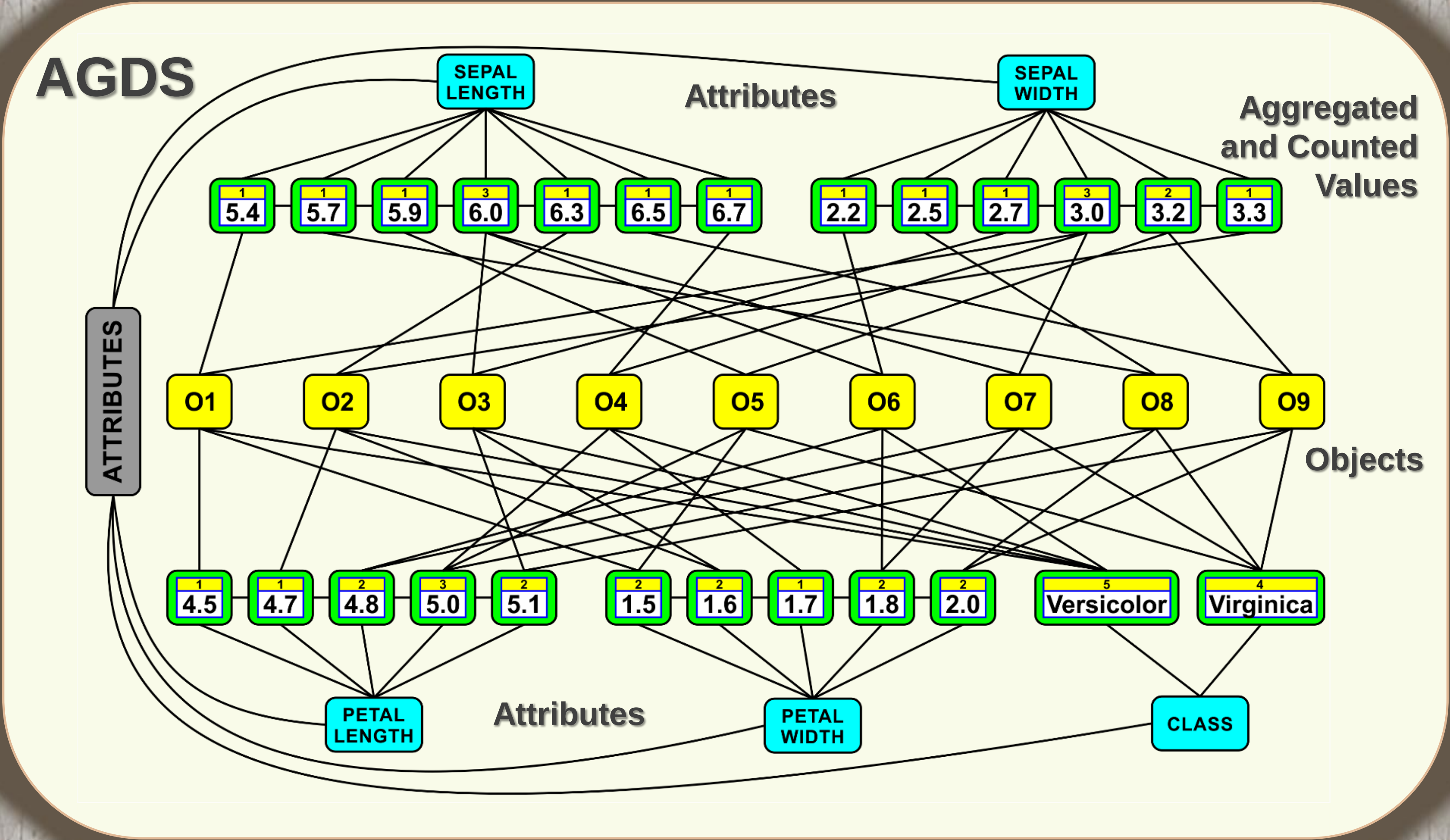
We can find a solution in the brain structures where data are stored together with their relations.



- ✓ Neurons can represent any subset of input data combinations which activate them.
- ✓ Neuronal plasticity processes automatically connect neurons and reinforce connections which represent related data and objects.

Let us use the biologically optimized solution!

AGDS
Associative Graph Data Structure

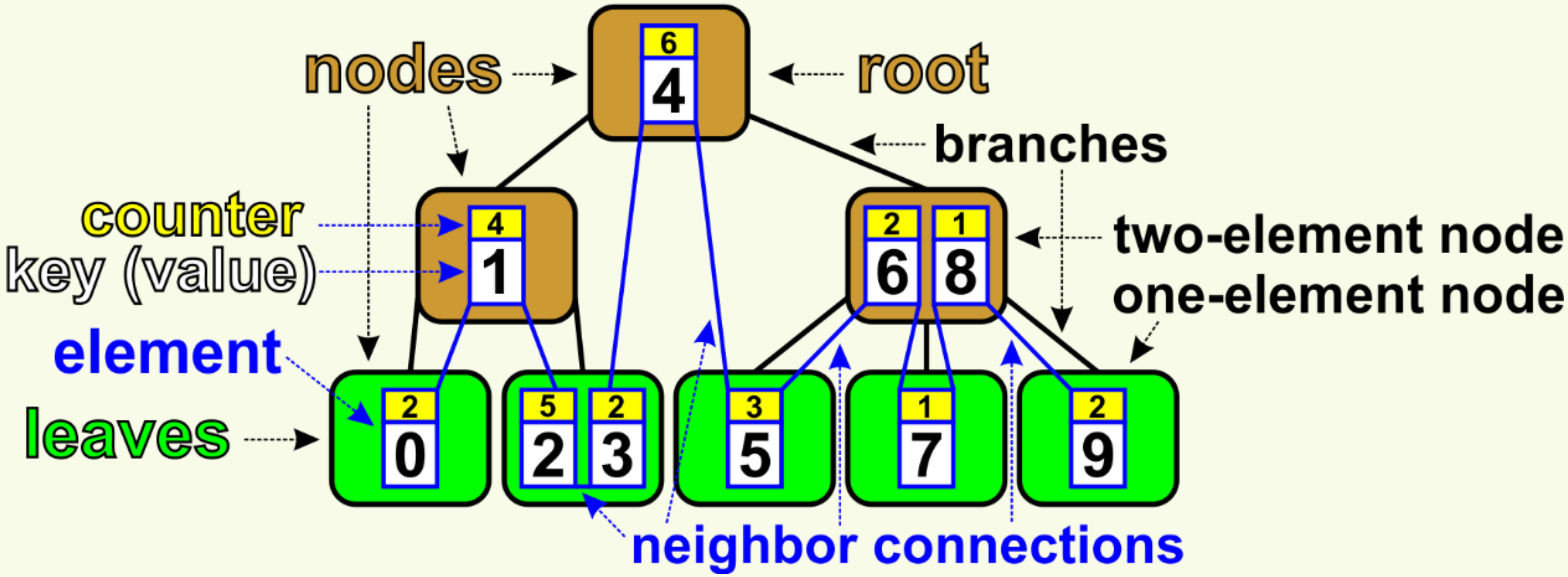


Connections represent various relations between AGDS elements like similarity, proximity, neighborhood, definition etc.

AVB+Trees

Sorting Aggregated-Value B-Trees

An AVB+tree is a hybrid structure that represent sorted list of elements which are quickly accessed via self-balancing B-tree structure. Elements aggregate and count up all duplicates of represented values.



AVB+trees are typically much smaller in size and height than B-trees and B+trees thanks to the aggregations of duplicates and not using any extra internal nodes as signposts as used in B+trees.

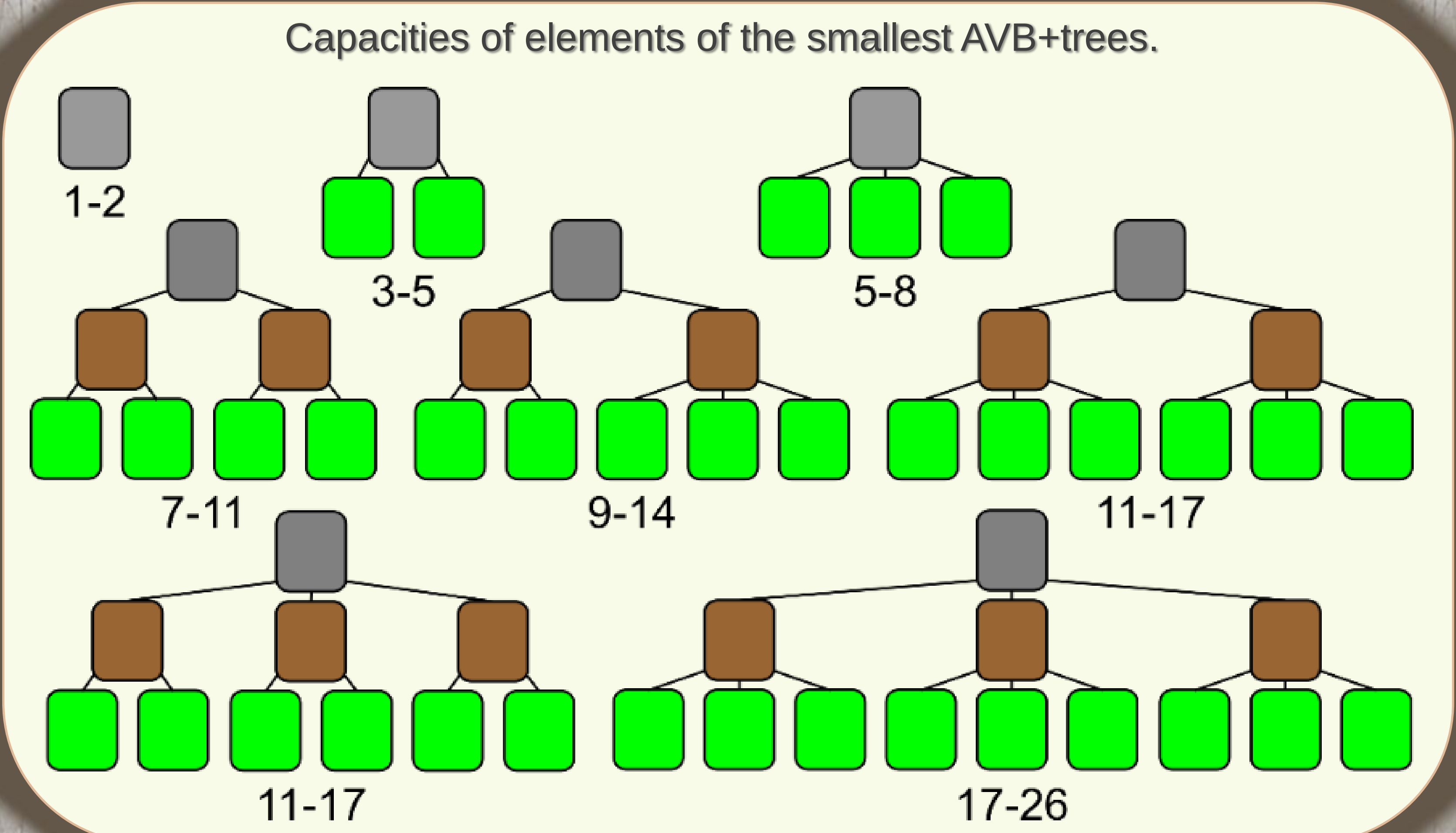
Internal states of APN neurons are updated only at the end of internal processes (IP) that are supervised by the Global Event Queue.

Properties of AVB+trees

- ✓ Each tree node can store one or two elements.
- ✓ Elements aggregate representations of duplicates and store counters of aggregated duplicates of values.
- ✓ Elements are connected in a sorted order, so it is possible to move between neighbor values very quickly.
- ✓ AVB+trees do not use extra nodes to organize access to the elements stored in leaves as B+trees.
- ✓ AVB+trees use all advantages of B-trees, B+trees, and AVB-trees removing their inconvenience.
- ✓ They implement common operations like Insert, Remove, Search, GetMin, GetMax, and can be used to compute Sums, Counts, Averages, Medians etc. quickly.
- ✓ They supply us with sorted lists of elements which are quickly accessible via this tree structure and thanks to the aggregations of duplicates that substantially reduce the number of elements storing values.

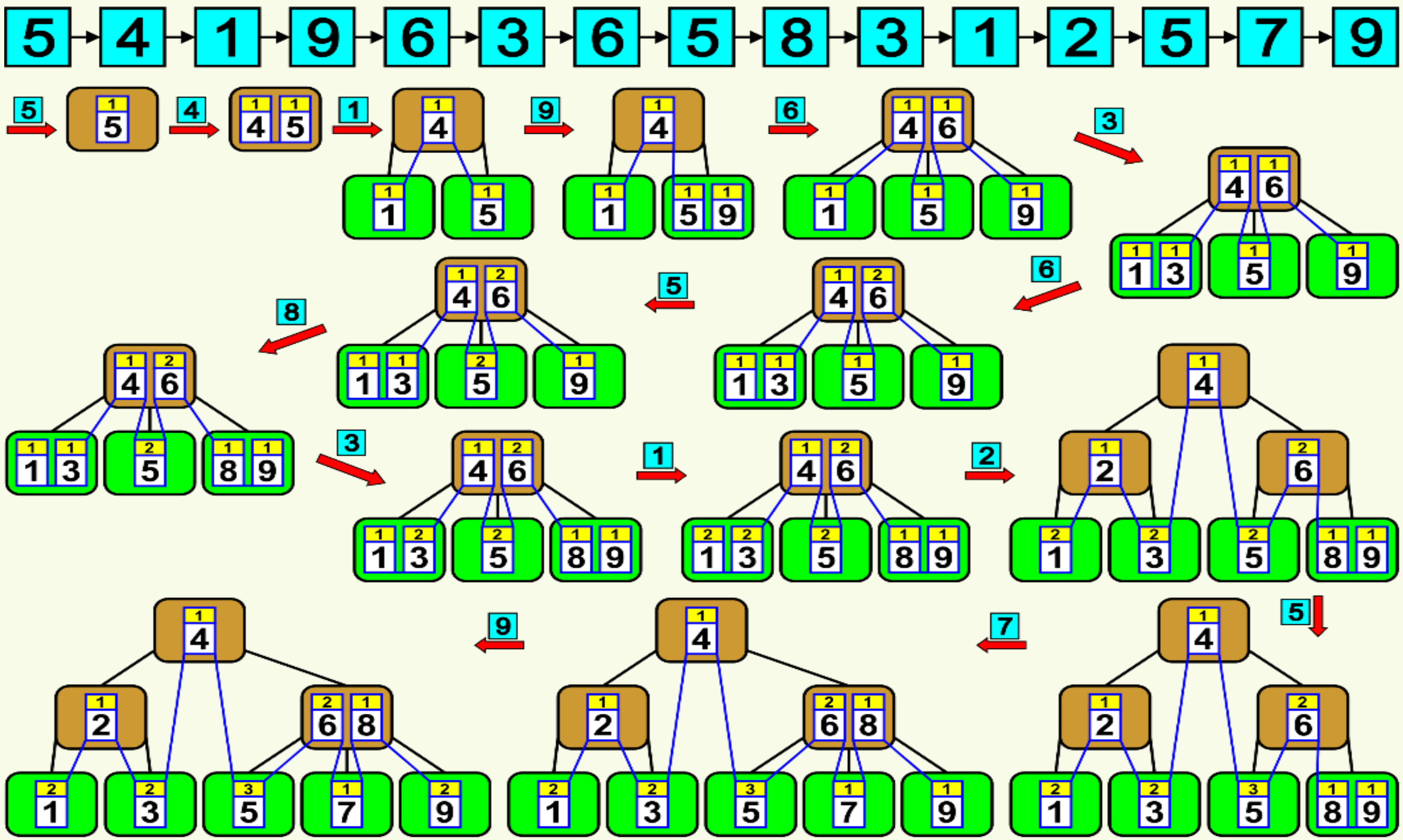
Efficient hybrid structure!

Capacity of AVB+Trees



The same number of elements can be stored by various AVB-tree structures, e.g. 11 or 17 elements!

Insert Operation on AVB+Trees



AVB+trees self-balance, self-sort and self-organize the structure during the insert operation!

Insert Operation

The Insert operation on the AVB+tree is processed as follows:

1. Start from the root and go recursively down along the branches to the descendants until the leaf is not achieved after the following rules:
   - if one of the elements stored in the node already represents the inserted key, increment the counter of this element, and finish this operation;
   - else go to the left child node if the inserted key is less than the key represented by the leftmost element in this node;
   - else go to the right child node if the inserted key is greater than the key represented by the rightmost element in this node;
   - else go to the middle child node.
2. When the leaf is achieved:
   - and if the inserted key is already represented by one of the elements in this leaf, increment the counter of this element, and finish this operation;
   - else create a new element to represent the inserted key and initialize its counter to one, next insert this new element to the other elements stored in this leaf in the increasing order, update the neighbor connections, and go to step 3.

Less than logarithmic expected computational complexity (typically constant) for data containing duplicates!

Insert Operation

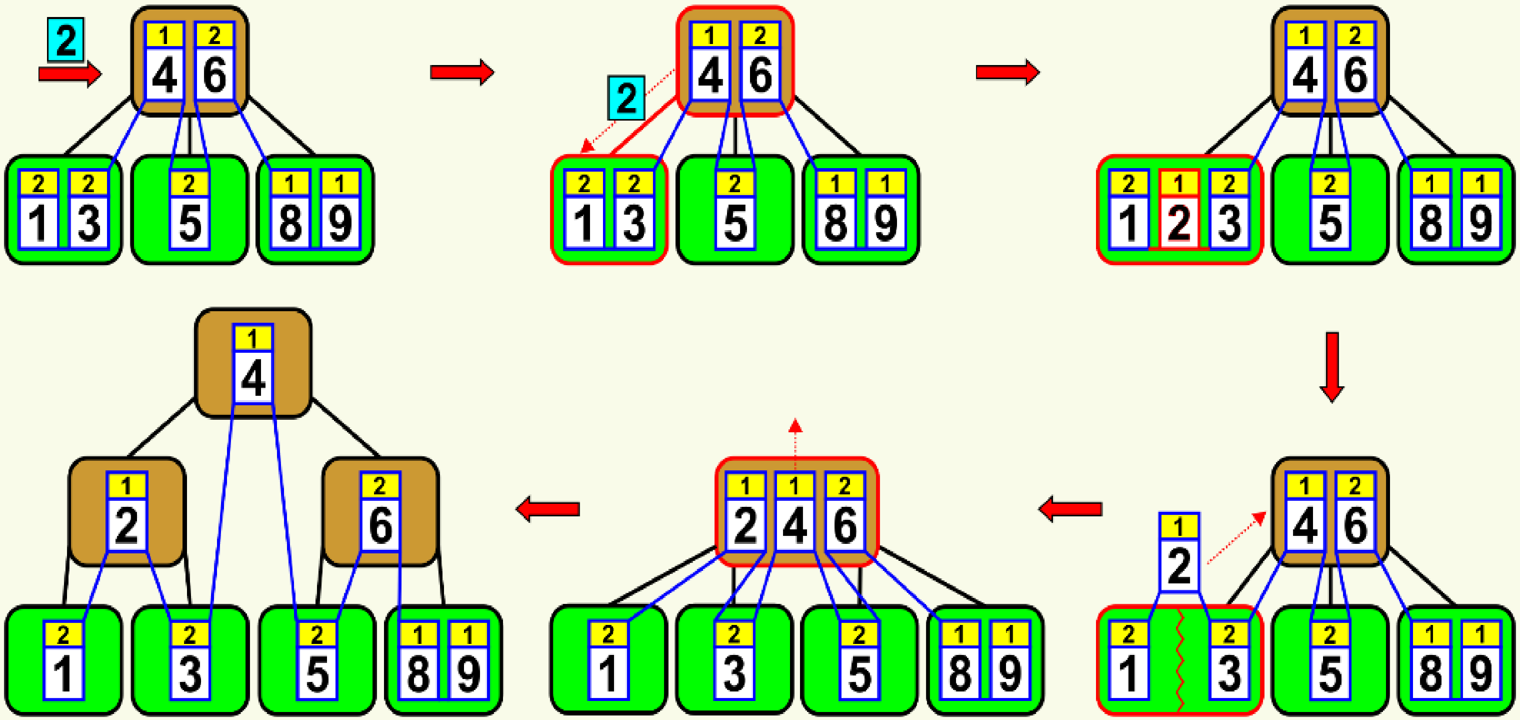
3. If the number of all elements stored in this leaf is greater than two, divide this leaf into two leaves in the following way:

- let the divided leaf represent the leftmost element representing the least key in this node together with its counter;
- create a new leaf and let it represent the rightmost element representing the greatest key in this node together with its counter;
- and the middle element (representing the middle key together with its counter) and the pointer to the new leaf representing the rightmost element pass to the parent node if it exists, and go to step 4;
- if the parent node does not exist, create it (a new root of the AVB+tree) and let it represent this middle element (representing the middle key together with its counter), and create new branches to the divided leaf representing the leftmost element and to the leaf pointed by the passed pointer to the new leaf representing the rightmost element.
  Next, finish this operation.

Less than logarithmic expected computational complexity
(typically constant) for data containing duplicates!

Rebalancing during Insert Operation

A self-balancing mechanism of an AVB+tree during the Insert operation when adding the value (key) „2” to the current structure which must be reconstructed because the node is overfilled and must be divided.



Self-balancing and self-sorting mechanism of the Insert Operation when a node is overfilled and must be divided!

Insert Operation

4. Insert the passed element between the element(s) stored in this node in the key - increasing order after the following rules:

- if the element has come from the left branch, insert it on the left side of the existing element(s) in this node;
- if the element has come from the right branch, insert it on the right side of the existing element(s) in this node;
- if the element has come from the middle branch, insert it between the existing element(s) in this node.

5. Create a new branch to the new node (or leaf) pointed by the passed pointer and insert this pointer to the child list of pointers immediately after the pointer representing the branch to the divided node (or leaf).

Less than logarithmic expected computational complexity (typically constant) for data containing duplicates!

Insert Operation

6. If the number of all elements stored in this node is greater than two, divide this node into two nodes in the following way:

- let the existing node represent the leftmost element representing the least key in this node together with its counter;
- create a new node and let it represent the rightmost element representing the greatest key in this node together with its counter;
- the middle element (representing the middle key together with its counter) and the pointer to the new node representing the rightmost element pass to the parent node if it exists; and go back to step 4;
- if the parent node does not exist, create it (a new root of the AVB+tree) and let it represent this middle element (representing the middle key together with its counter), and create new branches to the divided node representing the leftmost element and to the node pointed by the passed pointer to the new node representing the rightmost element. Next, finish this operation.

Less than logarithmic expected computational complexity (typically constant) for data containing duplicates!

Remove Operation

- ✓ The Remove operation allows to remove a key from the AVB+tree structure and next quickly rebalance and reorganize the structure automatically if necessary.
- ✓ If the removed key is duplicated in the current structure, then only the counter of the element which represents it is decremented.
- ✓ When the removed key is represented by the element which counter is equal one then the element is removed from the node.
- ✓ If this node is a leaf containing only a single element, then the leaf is removed as well, and a rebalancing operation of the AVB+tree is executed.

Less than logarithmic expected computational complexity (typically constant) for data containing duplicates!

Remove Operation

The Remove operation on the AVB+tree is processed as follows:

1. Start from the root and go recursively down along the branches to the descendants until the removed key is found in one of the elements in the nodes after the following rules:

- if one of the elements stored in the node represents the removed key go to step 2;
- else if this node is a leaf, finish this operation without removing the key from the tree because this key was not found;
- else go to the left child node if the removed key is less than the key represented by the leftmost element in this node;
- else go to the right child node if the removed key is greater than the key represented by the rightmost element in this node;
- else go to the middle child node.

Less than logarithmic expected computational complexity
(typically constant) for data containing duplicates!

Remove Operation

2. Decrement the counter of the element representing the removed key and:

- if the decremented counter is greater or equal one finish this operation success-fully;
- else remove the element from the node and go to step 3.

3. If the node where the element was removed is not a leaf go to step 12 else:

- if there is still another element in this leaf after the reduction of the removed element then finish this operation successfully;
- else remove the leaf and go to step 4.

4. If the parent node of the removed node has only a single element, go to step 5 else go to step 6.

Less than logarithmic expected computational complexity (typically constant) for data containing duplicates!

Remove Operation

2. Decrement the counter of the element representing the removed key and:

- if the decremented counter is greater or equal one finish this operation success-fully;
- else remove the element from the node and go to step 3.

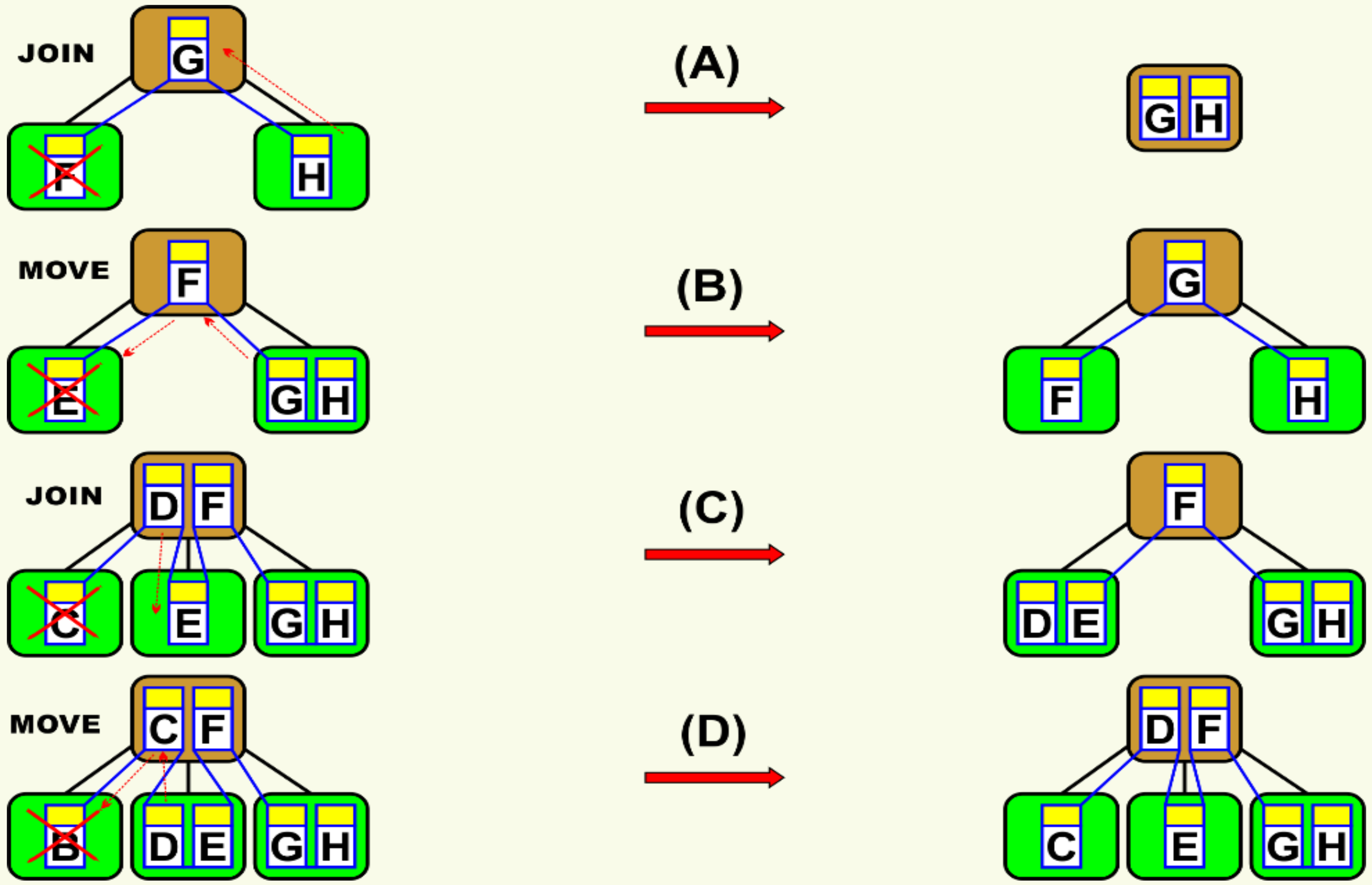
3. If the node where the element was removed is not a leaf go to step 12 else:

- if there is still another element in this leaf after the reduction of the removed element then finish this operation successfully;
- else remove the leaf and go to step 4.

4. If the parent node of the removed node has only a single element, go to step 5 else go to step 6.

Less than logarithmic expected computational complexity (typically constant) for data containing duplicates!

Remove Operation on AVB+Trees



Move and Join Operations on AVB+trees during Remove Operation which reorganize this tree!

Remove Operation

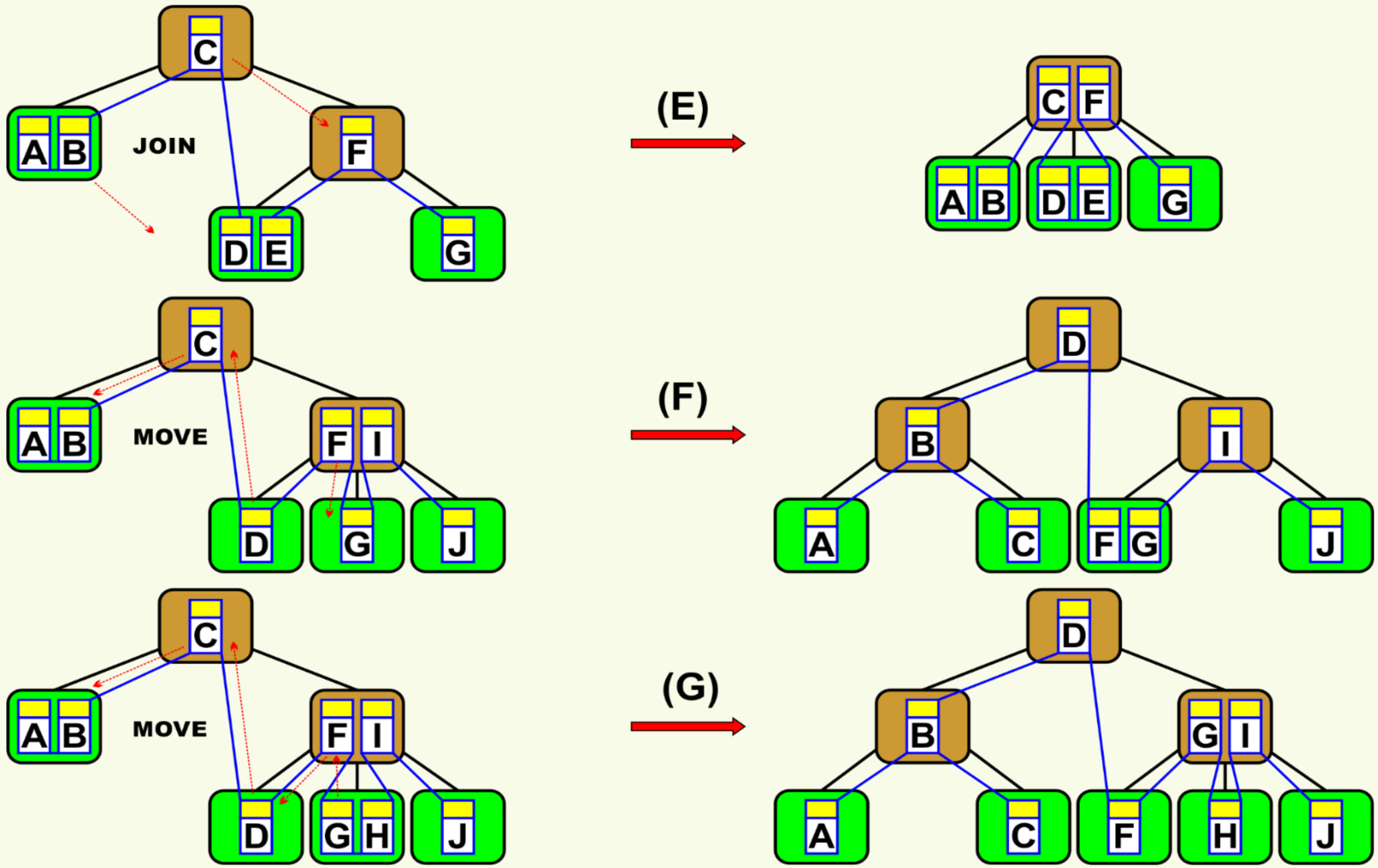
5. If the second child of the parent node of the removed node also only has a single element (Fig. A) than join these two elements together and remove the second child as well, and go to step 7;
else create the removed node again and move the parent element to this node and its neighbor connected element from its second child move to the parent node (Fig. B).

6. For the parent node of the removed element which contains two elements:

- if the second neighbor element connected to the parent element which was connected to the removed element in the removed node is single in its node then move this parent element to this child joining them together in this child node (Fig. C);
- else create the removed node again and move the parent element to this node and its neighbor connected element from its second child move to the parent node (Fig. D).

Less than logarithmic expected computational complexity
(typically constant) for data containing duplicates!

Remove Operation on AVB+Trees



Move and Join Operations on AVB+trees during Remove Operation which reorganize this tree!

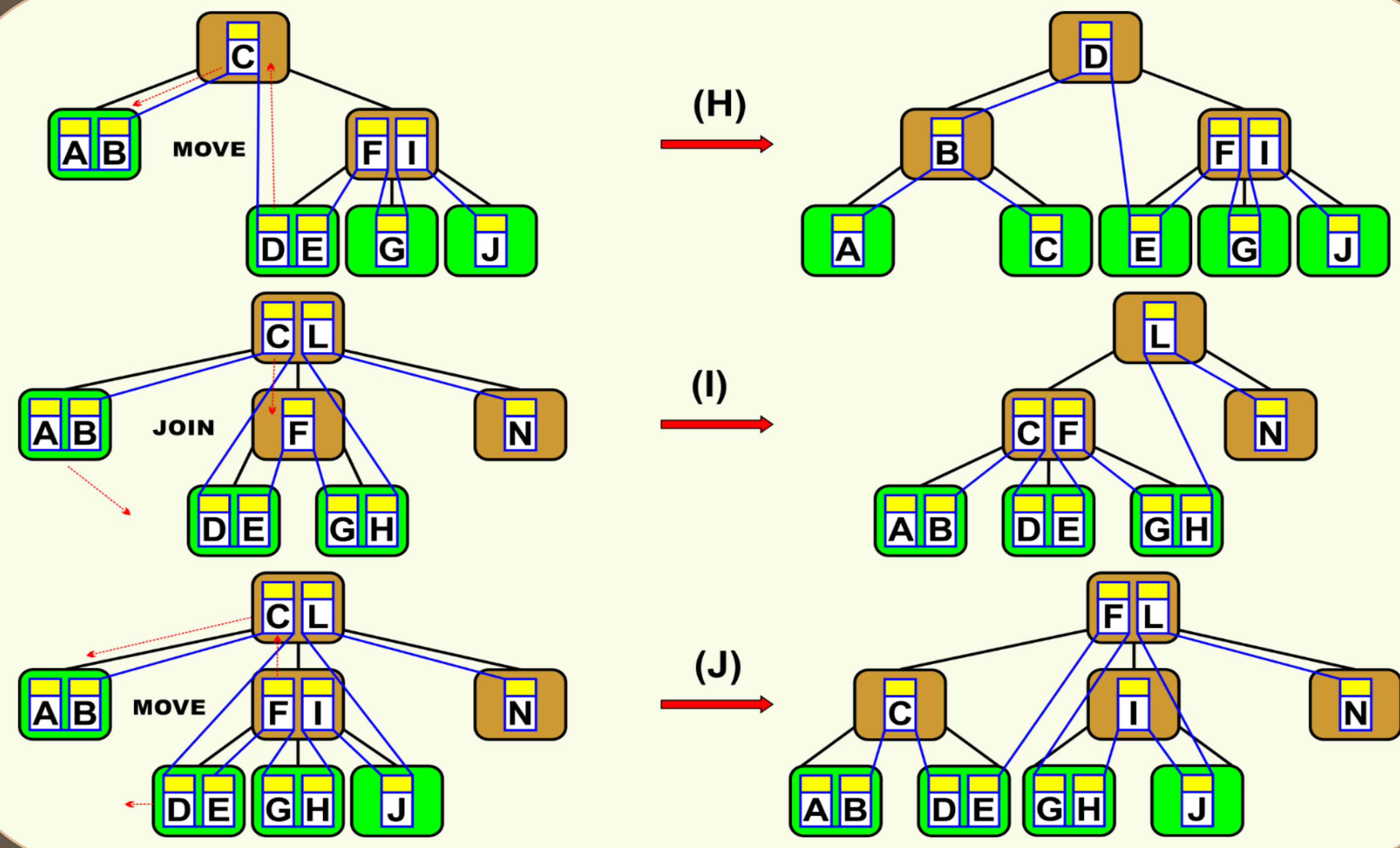
Remove Operation

7. For the joined node, if the parent node has only a single element (Fig. E-H), go to step 8 else go to step 11.
8. If the second child of this parent node has only a single element; go to step 9 else go to step 10.
9. Join parent element with the second child element and move the joined element to the new joined parent (Fig. E) and go to step 7 until this parent is not a root of the tree. When the parent is the root finish this operation successfully.
10. For the second child containing two elements (Figs. F-H):

- if the child of this child connected to the parent element is single in its node (Fig. F-G) move it to the parent and the node from the parent to the reconstructed branch where the nodes have been joined; next, go to step 6 balancing the second child of this parent.
- else for the child of this child connected to the parent element is not alone in its node (Fig. H), move this child to this connected parent node and the parent element to the branch where the nodes have been joined. Next, finish this operation successfully.

Less than logarithmic expected computational complexity (typically constant) for data containing duplicates!

Remove Operation on AVB+Trees



Move and Join Operations on AVB+trees during Remove Operation which reorganize this tree!

Remove Operation

11. For the second child containing two elements (Fig. I-J):

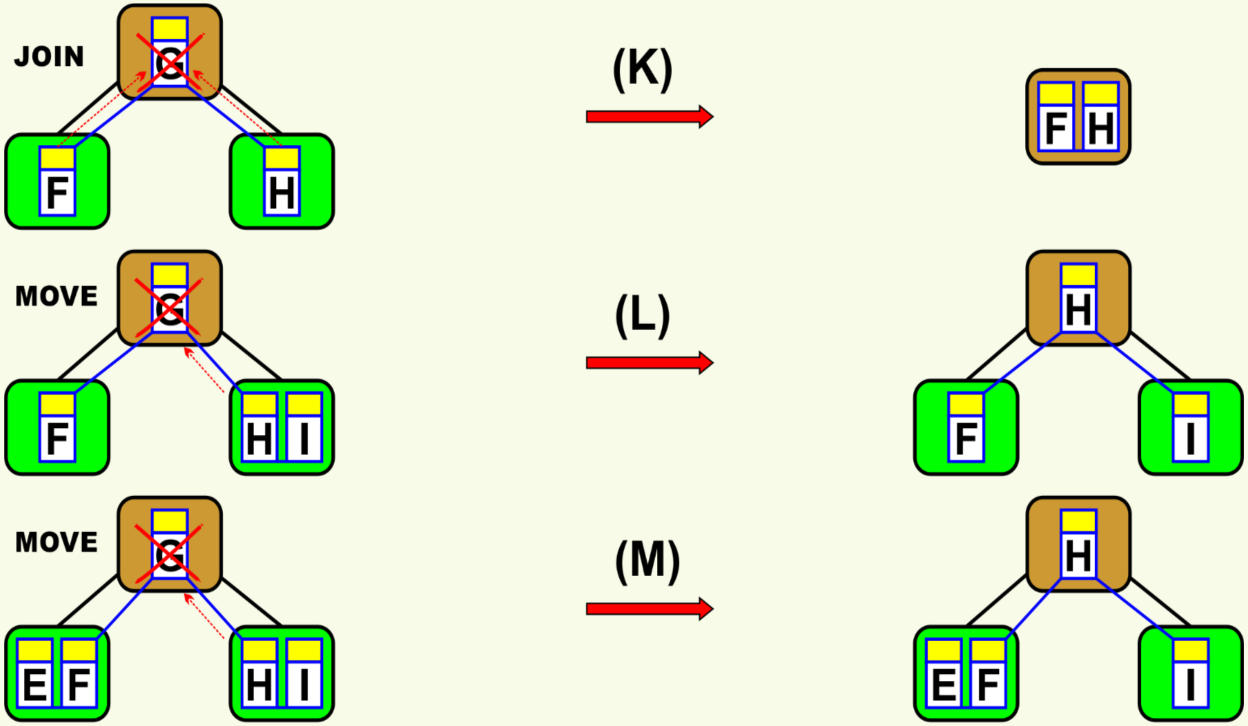
- if one of the neighbor siblings of the joined node has a single element then move the parent element of the joined node to this neighbor sibling and move the joined node to the children of this neighbor siblings (Fig. I).
- else move the connected parent element to the branch where the nodes have been joined, the first closest element from the two-element child to the node and its connected child to the child of the reconstructed branch (Fig. J).
- Next, finish this operation successfully.

12. For elements removed from the non-leaf node (Fig. K-P):

- if this node has only two children go to step 13;
- else go to step 14.

Less than logarithmic expected computational complexity (typically constant) for data containing duplicates!

Remove Operation on AVB+Trees



Move and Join Operations on AVB+trees during Remove Operation which reorganize this tree!

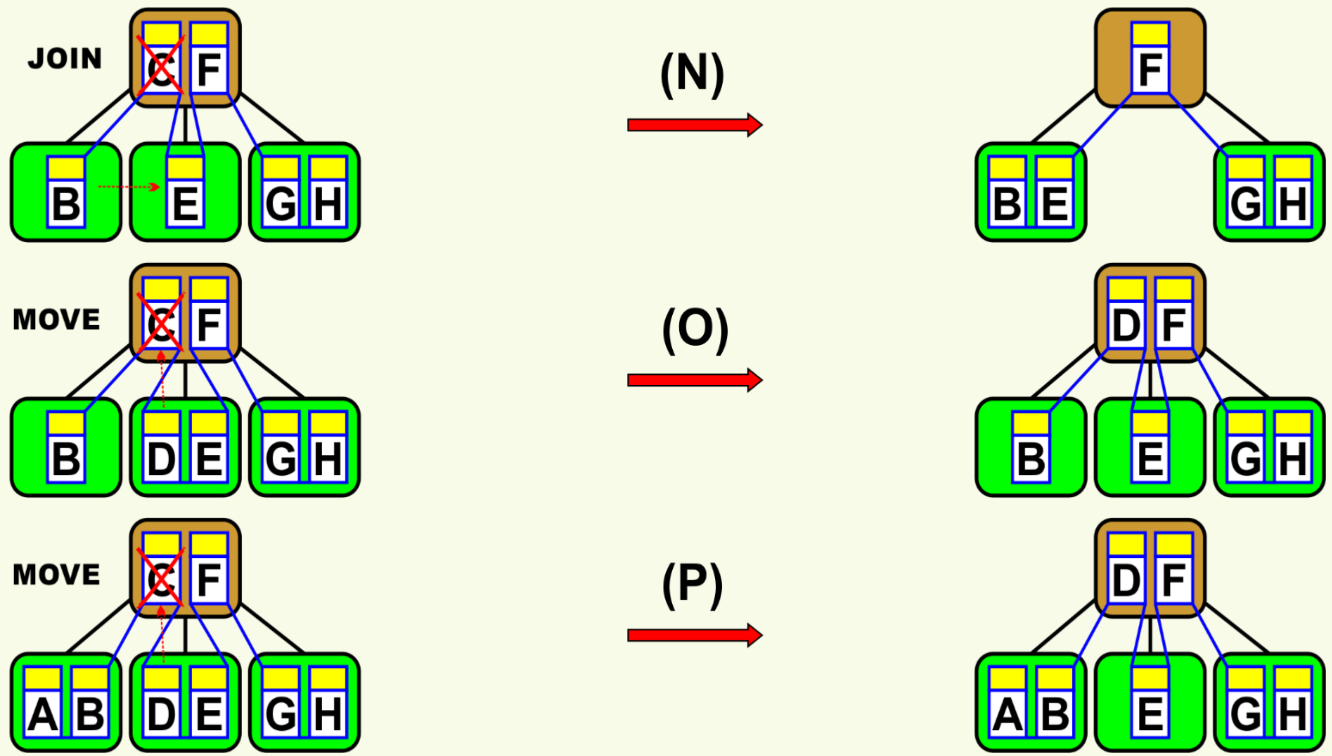
Remove Operation

13. If both two children have only a single element each
then join them together in one node (Fig. K) and go to step 7;
else move one element of the two-element child to the parent to replace
the removed element (Fig. L-M). Next, finish this operation successfully.

14. If both two neighbor children have only a single element each
then join them together in one node (Fig. N);
else move one element of the two-element child to the parent to replace
the removed element (Fig. O-P).
Next, finish this operation successfully.

Less than logarithmic expected computational complexity
(typically constant) for data containing duplicates!

Remove Operation on AVB+Trees



Move and Join Operations on AVB+trees during Remove Operation which reorganize this tree!

Update Operation

- ✓ The Update operation is a simple sequence of Remove and Insert operations because it is not possible to simply update a value in an element because of the structure of AVB+trees which represent various relations.
- ✓ Data can be easily updated (a value can be changed) only in those structures which do not represent relations, e.g. unsorted arrays, lists, or tables.
- ✓ The Update operation on an AVB+tree removes the old key (value) from this structure using the Remove operation and inserts an updated one using the Insert operation.

Less than logarithmic expected computational complexity (typically constant) for data containing duplicates!

Search Operation

The Search operation in the AVB+tree is processed as follows:

1. Start from the root and go recursively down along the branches to the descendants until the searched key or the leaf is not achieved after the following rules:

- If one of the keys stored in the elements of this node equals to the searched key, return the pointer to this element;
- else go to the left child node if the searched key is less than the key represented by the leftmost element in this node;
- else go to the right child node if the searched key is greater than the key represented by the rightmost key in this node;
- else go to the middle child node.

2. If the leaf is achieved and one of the stored elements in this leaf contains the searched key, return the pointer to this element, else return the null pointer.

Less than logarithmic expected computational complexity
(typically constant) for data containing duplicates!

GetMin and GetMax Operations

The GetMin and GetMax operations can be implemented in two different ways dependently on how often extreme elements are used in other computations using an AVB+tree structure:

1. The first way is used when extreme keys are not often used.
   In this case, it is necessary to start from the root node and always go along the left tree branches until the leaf is achieved and in its leftmost element (if there are two) is the minimum key (value) stored in this tree.
   Similarly, we go always along the right branches starting from the root node until the leaf is achieved and in its rightmost element (if there are two) is the maximum key (value) stored in this tree. These operations take log Ň time, where Ň is the number of elements stored in the tree, which is equal the number of unique keys (values) of the data.

Less than logarithmic expected computational complexity (typically constant) for data containing duplicates!

GetMin and GetMax Operations

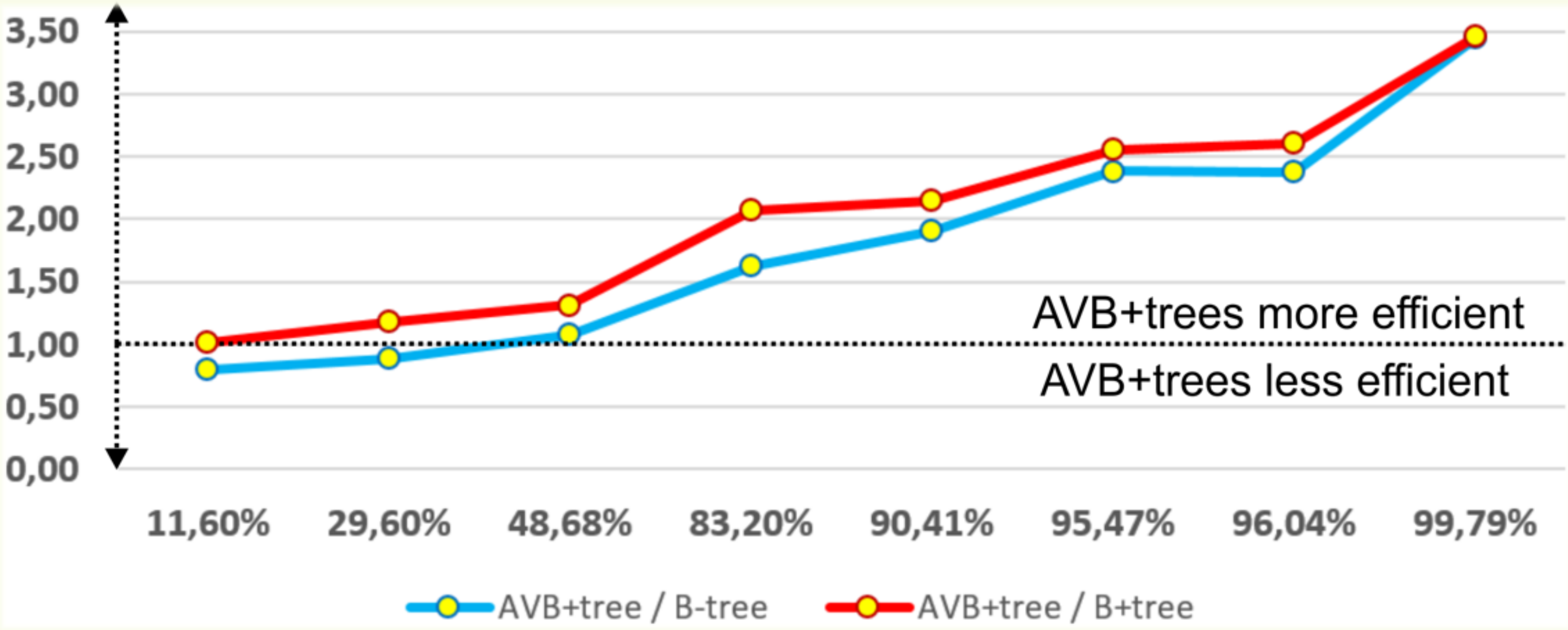
The GetMin and GetMax operations can be implemented in two different ways dependently on how often extreme elements are used in other computations using an AVB+tree structure:

2. The second way is used when extreme keys are often used and should be quickly available (in constant time).
   In this case, the leftmost (minimum) and rightmost (maximum) elements of the leftmost and rightmost leaves appropriately are additionally pointed from the class implementing the AVB+tree.
   If using these extra pointers they are automatically updated when the minimum or maximum element is changed, and
   the minimum and maximum element can be easily recognized because its neighbor connection to the left or right neighbor element is set to null.

Less than logarithmic expected computational complexity
(typically constant) for data containing duplicates!

Comparison of Efficiencies

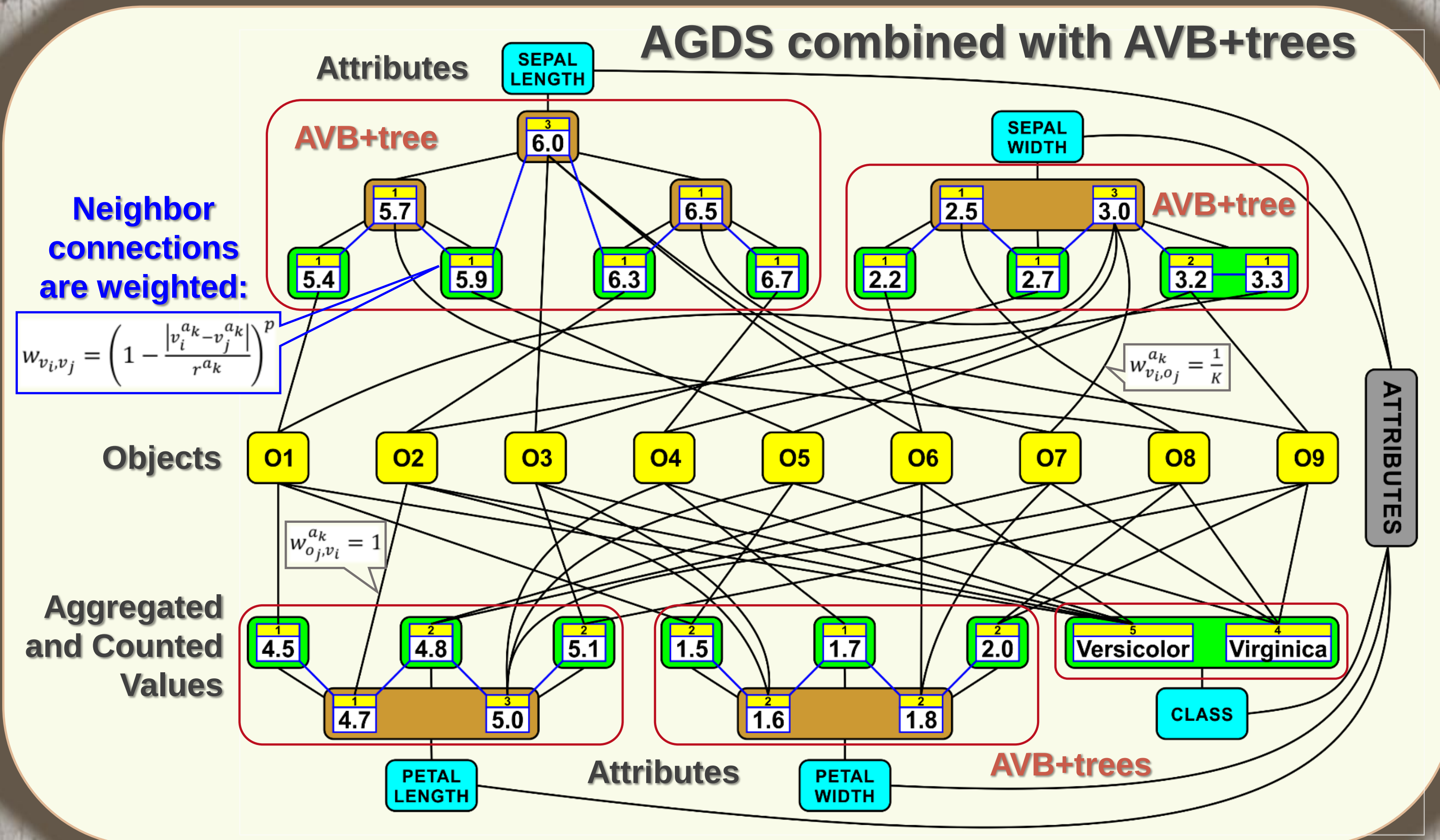
The efficiencies of the same operations on the same datasets from UCI ML Repository were compared on B-trees, B+trees, AVB-trees, and AVB+trees.



The achieved results proved the concept that AVB+trees are always faster than B+trees commonly used in databases, and AVB-trees are usually faster than B-trees when data contain more than 30% of duplicates.

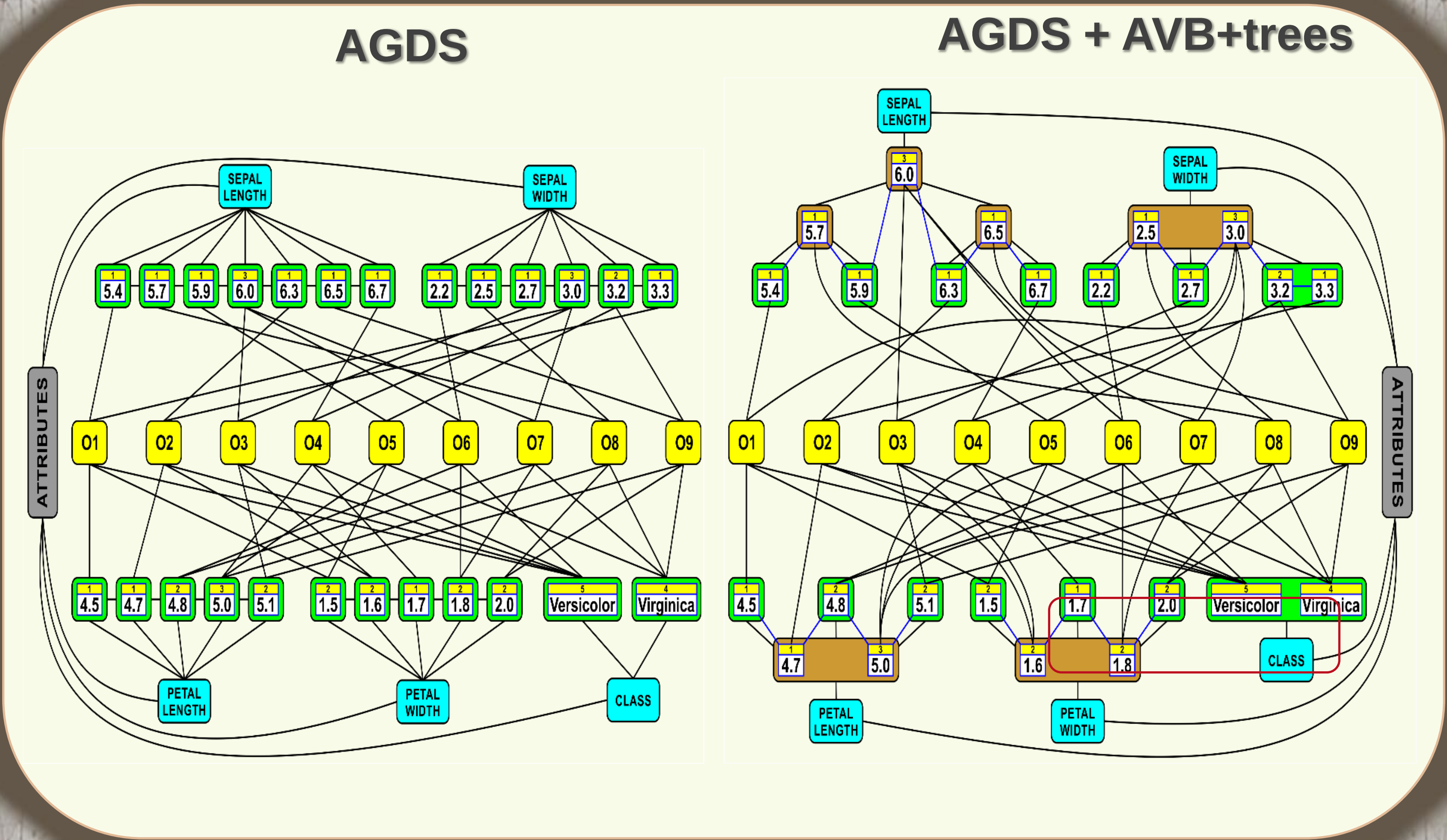
AVB-trees and AVB+trees outperform commonly used B-trees and B+trees in most cases!

AGDS + AVB+trees
as a still more efficient solution



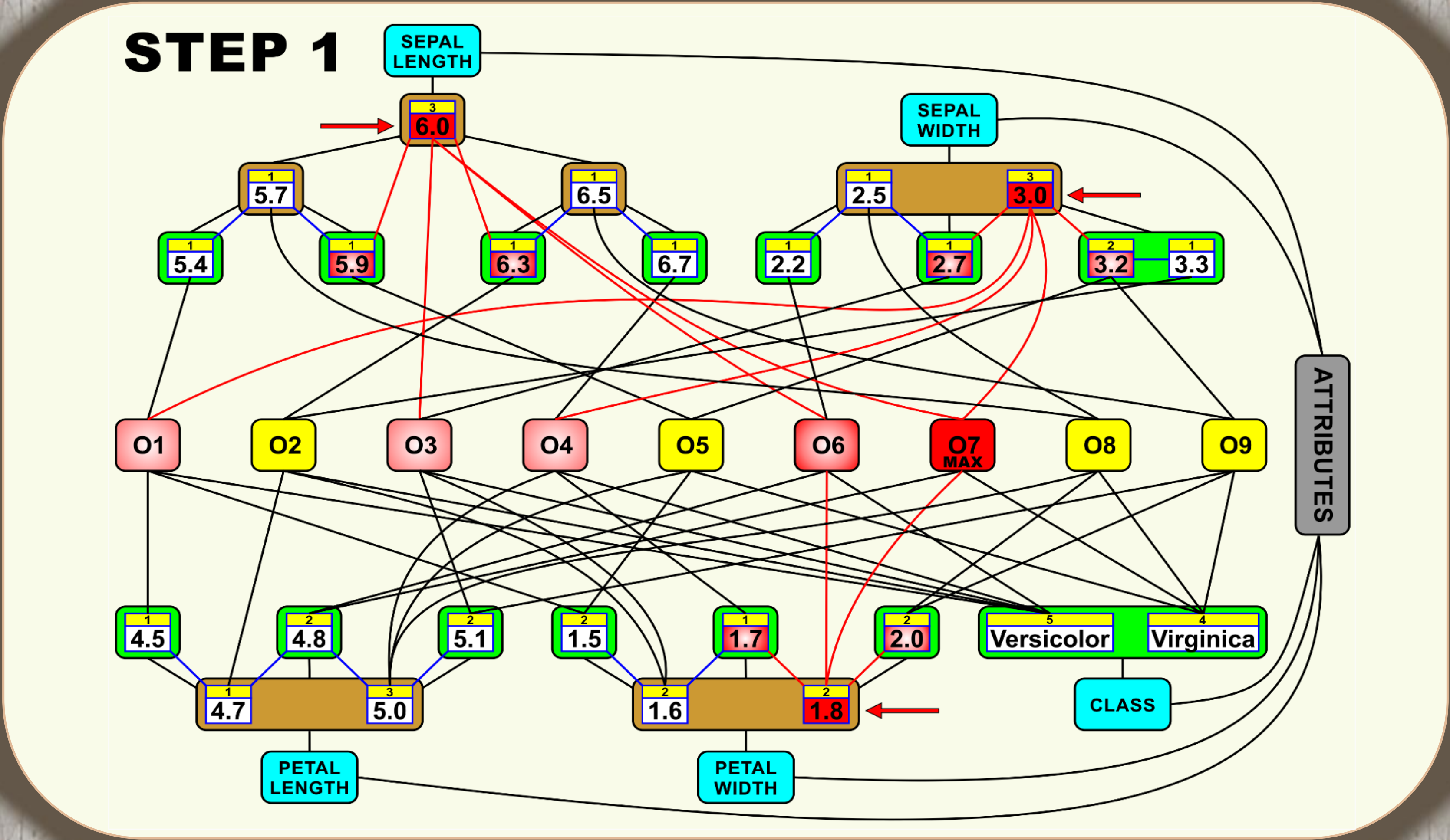
AVB+trees implemented to AGDS structures make the data access faster especially for Big Data datasets and databases.

Comparison of AGDS with AGDS + AVB+trees



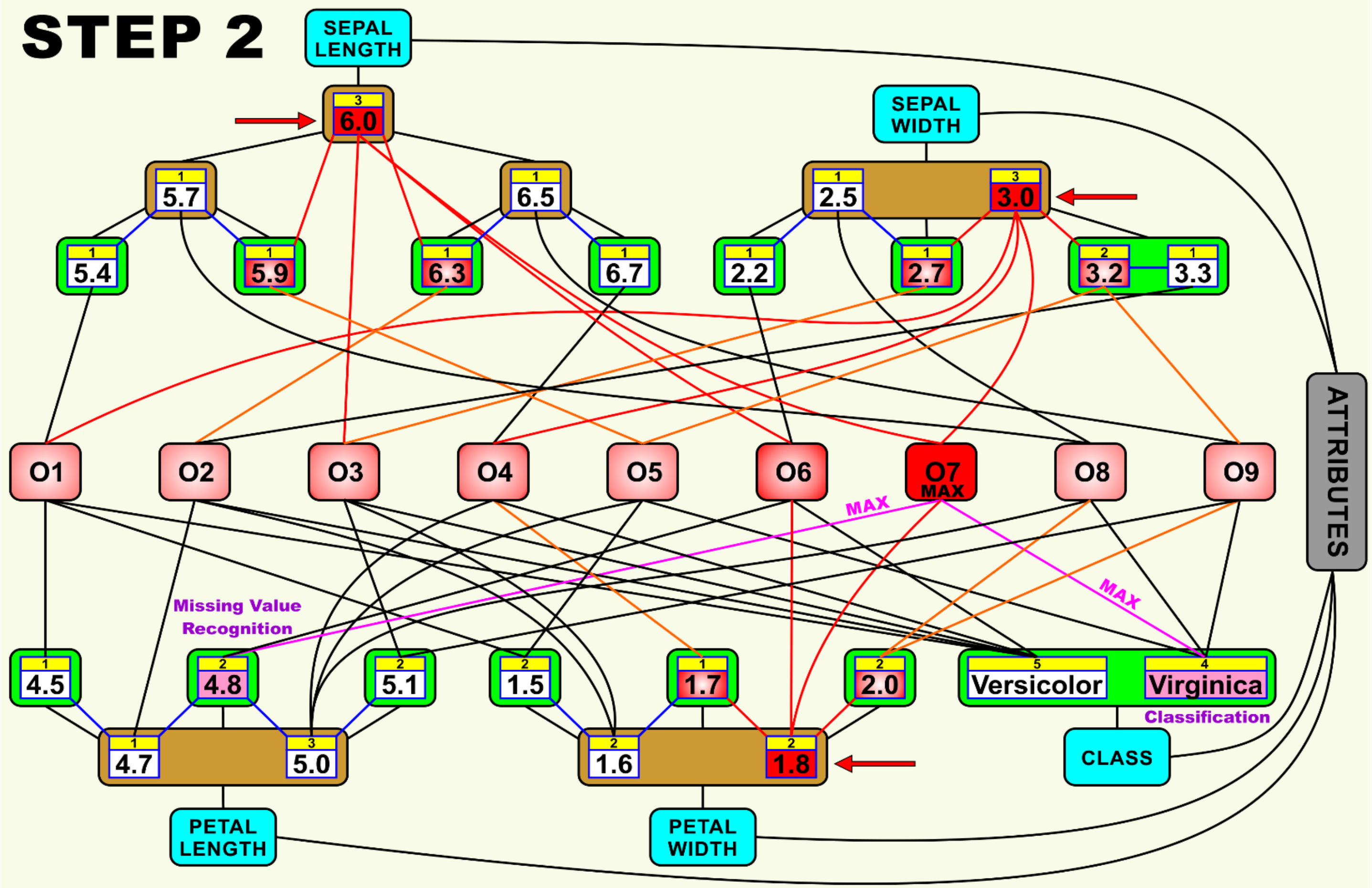
When data contain many duplicates
we practically achieve the constant access
to all data stored in AGDS + AVB+trees.

Inferences on AGDS combined with AVB+trees



We do not need to search for common relations in many (nested) loops but we simply go along the connections and get results.

Inferences on AGDS combined with AVB+trees



Such structures can also be used for very fast recognition, clustering, classification, searching for the most similar objects etc.

Conclusions

- ✓ **AGDS structures combined with AVB+trees provide incredibly fast access to any data stored and sorted for all attributes simultaneously.**
- ✓ **AGDS + AVB+trees stores data together with the most common vertical and horizontal relations, so there is no need to loop and search for these relations.**
- ✓ **Typical operations on AGDS + AVB+trees structures have pessimistically logarithmic time, but the expected complexity on typical real data is constant.**

Questions or Remarks?

1. **A. Horzyk**, J. A. Starzyk, J. Graham, ***Integration of Semantic and Episodic Memories***, IEEE Transactions on Neural Networks and Learning Systems, Vol. 28, Issue 12, Dec. 2017, pp. 3084 - 3095, 2017, DOI: 10.1109/TNNLS.2017.2728203.
2. **A. Horzyk**, J.A. Starzyk, ***Multi-Class and Multi-Label Classification Using Associative Pulsing Neural Networks***, IEEE Xplore, In: 2018 IEEE World Congress on Computational Intelligence (WCCI IJCNN 2018), 2018, (in print).
3. **A. Horzyk**, J.A. Starzyk, ***Fast Neural Network Adaptation with Associative Pulsing Neurons***, IEEE Xplore, In: 2017 IEEE Symposium Series on Computational Intelligence, pp. 339 -346, 2017, DOI: 10.1109/SSCI.2017.8285369.
4. **A. Horzyk**, K. Gołdon, ***Associative Graph Data Structures Used for Acceleration of K Nearest Neighbor Classifiers***, LNCS, In: 27th International Conference on Artificial Neural Networks (ICANN 2018), 2018, (in print).
5. **A. Horzyk**, ***Deep Associative Semantic Neural Graphs for Knowledge Representation and Fast Data Exploration***, Proc. of KEOD 2017, SCITEPRESS Digital Library, pp. 67 - 79, 2017, DOI: 10.13140/RG.2.2.30881.92005.
6. **A. Horzyk**, ***Neurons Can Sort Data Efficiently***, Proc. of ICAISC 2017, Springer-Verlag, LNAI, 2017, pp. 64 - 74, ICAISC BEST PAPER AWARD 2017 sponsored by Springer.
7. **A. Horzyk**, J. A. Starzyk and Basawaraj, ***Emergent creativity in declarative memories***, IEEE Xplore, In: 2016 IEEE Symposium Series on Computational Intelligence, Greece, Athens: Institute of Electrical and Electronics Engineers, Curran Associates, Inc. 57 Morehouse Lane Red Hook, NY 12571 USA, 2016, ISBN 978-1-5090-4239-5, pp. 1 - 8, DOI: 10.1109/SSCI.2016.7850029.
8. **Horzyk, A.**, ***How Does Generalization and Creativity Come into Being in Neural Associative Systems and How Does It Form Human-Like Knowledge?***, Elsevier, Neurocomputing, Vol. 144, 2014, pp. 238 - 257, DOI: 10.1016/j.neucom.2014.04.046.
9. **A. Horzyk**, ***Innovative Types and Abilities of Neural Networks Based on Associative Mechanisms and a New Associative Model of Neurons*** - Invited talk at ICAISC 2015, Springer-Verlag, LNAI 9119, 2015, pp. 26 - 38, DOI 10.1007/978-3-319-19324-3_3.
10. **A. Horzyk**, ***Human-Like Knowledge Engineering, Generalization and Creativity in Artificial Neural Associative Systems***, Springer-Verlag, AISC 11156, ISSN 2194-5357, ISBN 978-3-319-19089-1, ISBN 978-3-319-19090-7 (eBook), Springer, Switzerland, 2016, pp. 39 – 51, DOI 10.1007/978-3-319-19090-7.

Adrian Horzyk

horzyk@agh.edu.pl

Google: Horzyk

AGH

University of Science and Technology in Krakow, Poland