IMPLEMENTING DECISION TREES THROUGH AUGMENTING TREE DATA STRUCTURE FOR CLASSIFICATION PURPOSES

MAJED ABUSAFIYA¹

¹Associate Professor, Al-Ahliyya Amman University, Department of Software Engineering, Jordan
E-mail: majedabusafiya@gmail.com

ABSTRACT

The main goal of this paper is to define a data structure to implement decision trees for classification purposes. A classification problem is defined by a set of items with defined attributes and a set of classifiers, its output is a set of classes of items. The method we followed in this paper is the known problem solving technique called augmenting data structures. The basic tree data structure is chosen. The required information to augment is defined. Also the basic operations to build, update and query this data structure are defined as algorithms. One important issue that was considered in defining this data structure is deal with ill-formed classification. Ill-formed classification may result in having an item ending in zero, one or more final classification classes. One main advantage of this data structure is that it can be used as base for software tools that facilitate the automated and interactive design, update and querying of decision trees for classification purposes.

Keywords: Algorithms, Decision trees; Augmenting data structures; Classification; Ill-formedness

1. INTRODUCTION

A decision tree is a tool that is used to break down a complex decision-making process to a collection of simpler decisions. One main application of the decision tree is classification: given a set of items with attributes and a set of classification rules, these items are grouped into a set of classes. Decision trees have been extensively used in literature. A lot of work was directed towards algorithms to learn the decision tree from input data: IDE3[1], C4.5[3], Sprint[2] and others [4]. A hardware implementation for decision tree was proposed in [4]. In [5] a data mining tool was used to implement decision tree. In [6], a decision tree was constructed by providing a visual interactive interface that helps the user to build the tree. In [7], an implementation of the ID3 decision tree algorithm using Java applets is presented.

To contrast our work from related work, we point to the following: (1) the work in this paper is not related to machine learning nor data mining where the decision tree is automatically generated from a given data set. We assume that the classification rules are given as input. This paper is directed towards the algorithmic construction of the decision tree in the data structure level, (2) our work views a decision tree as a dynamic data structure, so we present algorithms for building and modifying the tree for insertions, deletions and updates, (3) our work can be viewed as a basis to develop software tools to design decision trees for classification. The defined operations eases the adjustment of the tree to correctly define the classification problem, (4) the proposed data structure tolerates ill-formed classification. An ill-formed classification is the classification that is not well-formed. A classification is described to be well-formed if every input item ends in one and only one final class.

A data structure models an abstract object that organizes data into well-defined composition and allows operations for creation, modification, and querying [8]. Many problems can be efficiently solved by defining a suitable data structure (e.g. binary search trees, queues and stacks). In this paper, a similar approach will be taken where we wish to implement decision tree to solve classification problem using a data structure. We will implement the decision tree by using a known
algorithmic problem solving technique that is called augmenting data structures [9]. In this paper, we will augment the basic tree data structure to implement the decision tree for classification purposes. The process of augmenting a data structure to solve a given problem is defined by the following steps: (1) choosing an underlying data structure, (2) determining the additional information to augment, (3) verifying that this additional information is correctly maintained for the basic modifying operations and (3) finally define useful operations for querying data in this data structure. We will follow this process to define the desired decision tree.

This paper is organized as follows: Section 2 presents the information to be augmented into the basic tree data structure to implement decision trees. In Section 3, the operations of the new data structure are presented. In section 4, an example application for the proposed data structure is presented. The paper ends up with a discussion and a set of references.

2. AUGMENTING THE TREE DATA STRUCTURE

To ease the presentation of the proposed data structure, an example will first be given. Let items = \{i_1, i_2, i_3, i_4, i_5, i_6, i_7, i_8\} be the items to classify with the attributes \{f_1, f_2, f_3\}, all with type integer for simplicity. Table 1 shows the values of the attributes for these items.

<table>
<thead>
<tr>
<th>Item</th>
<th>(f_1)</th>
<th>(f_2)</th>
<th>(f_3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i_1)</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>(i_2)</td>
<td>4</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>(i_3)</td>
<td>2</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>(i_4)</td>
<td>7</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>(i_5)</td>
<td>2</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>(i_6)</td>
<td>5</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>(i_7)</td>
<td>6</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>(i_8)</td>
<td>3</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

The conditions upon which the classification is done are defined by classifiers. For our example, the classifiers are shown in Table 2. In this example, we assumed a classifier for each attribute for simplification. However, a classification conditions of a classifier may consider multiple attributes. There is no relation between the number of the classifiers and the number of attributes. For generality, we will assume that a classifier's conditions may overlap. This means that classification is not necessarily disjoint. See for example the second and third classification conditions for classifier2.

<table>
<thead>
<tr>
<th>Classifier</th>
<th>conditions</th>
<th>(f_1)</th>
<th>(f_2)</th>
<th>(f_3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>classifier1</td>
<td>(f_1) is odd &amp; (f_2) is even &amp; (f_3) is prime</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>classifier2</td>
<td>(f_1) is even</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>classifier3</td>
<td>(f_1) is prime</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

The information to be augmented in the tree data structure to implement the desired decision tree is shown in Table 3. The tree in Figure 1 will be used to better illustrate the proposed data structure. A tree \(T\) is composed of number of tree nodes with a \(T.root\) is the root tree node. The classifiers define the classification conditions and will be explained shortly. Items are the set of items to be classified. The final classes of the classification problem will be the items sets within the tree nodes blocks in the last level. The last level tree nodes set are referred to as lastLevelTreeNodes.

A TreeNode is composed of at least one tree node block. A tree node with no tree node blocks is eliminated. Every tree node block has exactly one child tree node (except those that exist in leaf tree nodes). Every tree node has a parent tree node block except the root. Every tree node has a classifier. Tree nodes of the same level share the same classifier. The items of a tree node are the items of its parent tree node block. The items of the \(T.root\) is the input items of the classification problem.

A tree node block has an \(id\) field. This \(id\) field is useful in identifying and locating a tree node block in the tree. An \(id\) is a string of numeric fields that are separated by dots. These numeric fields correspond to defined ranks. For example, in Figure 1, the tree node block whose \(id\) is (2.1.2) exists in the third level and the rank of its classification condition is 2. It is the child of tree node block (2.1). The tree node block (2.1) has the rank of its classification condition to be 1, which – in turn - exists as a child of tree node block (2) in the first level. The tree node block (2) has the rank of its
classification condition to be 2. We will use a slightly different notation to refer to a tree node. For example, we refer to the tree node that contains the tree node block (2.2.1 and 2.2.2) as (2.2.*).

Every tree node block has a set of items. The items of a tree node block are those items - of the containing tree node - that satisfy its classification condition. All tree node blocks must have non-empty items field. As will be shown latter, any tree node block that has an empty items field is eliminated from its tree node. A tree node block knows its containing tree node through treeNode field. A tree node block has a childTreeNode field. Each tree node block will correspond to one of the classes that are defined by the classifier of the level that contains this tree node block. The number of tree node blocks of a tree node equals to the number of the classification conditions of its classifier. However, If none of the items of a tree node satisfy the classification condition of a given tree node block, then this tree node block is dropped from its tree node. So, tree nodes at the same level may have a varying number of tree node blocks. Every item belonging to the items set of a tree node is supposed to join the items set of one of its tree node blocks. However, and for generality and flexibility, an item may join the items of multiple tree node block of the same tree node, if it satisfies the classification conditions of multiple tree node blocks. So an item of a tree node may end up in zero, one or more tree node blocks. If an item - that is a member of a the items of some tree node - does not satisfy any of the classification conditions of the corresponding classifier, then this item will not join any of the tree node blocks of this tree node. We say that this item is blocked at this tree node.

Classifiers is a set of classifiers. The classifiers map items of a tree node to the items of the contained tree node blocks. A classifier is defined as a set of classification conditions (Table 4). A classificationCondition has a rank within its classifier. This rank identifies the rank of a corresponding tree node block in a tree node. It is composed of a condition and a corresponding className (it is a label that identifies a class). As will be shown latter, a classificationCondition is applied on the items of the containing tree node and returns the set of items that satisfy its condition. An item that satisfies a classificationCondition will join the items of the corresponding tree node block. We mean by corresponding tree node block, is the tree node block whose classification condition is classificationCondition. That is, a classification condition is associated with a tree node block in the sense that it identifies its items and both have the same rank. For example, in Figure 1, the classification condition with rank 2 of the second classifier, classified the item $i_4$ into tree node block with rank 2 (with id 1.2) in tree node (1.*). The treeNode field of a classifier is the set of the tree nodes where the classifier is applied. They compose the tree nodes of one level within the tree. The rank field of a classifier specifies the level in the tree where it is applied. No specific ordering for applying the classifiers should be followed. However, we will assume the classifiers in classifiers are ordered and they are ranked in that order. For example, the third level classifier in Figure 1 has the rank value 3, its classification conditions are “$f_3$ is prime” and “$f_3$ is not prime” and its treeNode={1.1.*, 1.2.*, 2.1.*, 2.2.*, 2.3.*).

3. THE AUGMENTED DATA STRUCTURE OPERATIONS

In this section, the algorithms of the basic operations of the augmented data structure are presented. These operations include building, modifying and querying the tree. These operations mainly get or set the information fields of the objects that compose the tree. These algorithms are presented in high-level flowcharts. To save space and to simplify presentation, we will focus on the main steps of the algorithm. Some of the steps will be abstracted and only the most significant fields are updated. Some of the detailed steps and updates on less significant information fields are dropped since they can easily be inferred from the given presented algorithm. For example, a change of a childTreeNode field of a tree node block implies an update on parentTreeNodeBlock of the corresponding tree node.

3.1 BUILD-TREE operation

Building the tree is the main operation of the proposed data structure since the desired classification is a side product of building the tree.
The desired classification is found in the tree node blocks in the last level of the tree. The algorithm BUILD-TREE is shown Figure 2. It has two inputs: items and classifiers.

The tree T is built a level by level: one level for each classifier. First, the T.root tree node is created and root.items is initialized with items. T.classifiers is set to the input classifiers. The currentLevelTreeNodes refers to the set of tree nodes of the current level of the tree. The tree nodes of the next level of the tree are created as children of the tree node blocks of tree nodes in currentLevelTreeNodes. Initially, currentLevelTreeNodes contains the root tree node only. The first classifier from classifiers is chosen to classify the items in root. In a loop, the classification conditions of the first classifier are applied one after another against the the items of the root to find classifiedItems. The classifiedItems is the subset of items that satisfy the current classificationCondition. If classifiedItems is not empty, a newTreeNodeBlock is created, its items field is set to be classifiedItems, its treeNode and classification conditions are set, and then is inserted into the root tree Node. If classifiedItems set is empty, no tree block will be created for this classificationCondition. The construction of the root tree node is complete once all the classification conditions of the first classifier are processed. The next levels of the tree are built by calling BUILD-TREE-LEVEL algorithm once for each classifier. Every call of the BUILD-TREE-LEVEL algorithm will update currentLevelTreeNodes to be the set of the created tree nodes of the new level. The algorithm for BUILD-TREE-LEVEL is shown in Figure 3.

BUILD-TREE-LEVEL algorithm builds a new level in the tree by taking as input a classifier and the currentLevelTreeNodes - which contains the tree nodes that were created in the previous level. There are three nested loops in the algorithm. The first - outer - loop takes the next treeNode from the currentLevelTreeNodes. In the second loop, the tree node blocks of treeNode are taken one by one. A newTreeNode will be created for every treeNodeBlock in treeNode. The newTreeNode will be added to the newLevelTreeNodes. This is needed so that the newly created tree nodes will be available for the next call for the algorithm. In the third inner loop, the classification conditions of classifier are applied - one by one - against the items of the current treeNodeBlock. ClassifiedItems is the set of items of the current treeNodeBlock that satisfy the current classificationCondition. In case the classifiedItems is not empty, a newTreeNodeBlock is created with items field is set to be classifiedItems, its fields are set, and is then added to the newTreeNode. If classifiedItems is empty, no further action is needed and the algorithm proceeds to process the next classification condition. The inner-most loop terminates when all the classification conditions of classifier are applied to the items of the treeNodeBlock being processed. The second loop terminates when all the treeNodeBlocks of treeNode are processed. The outer-most loop terminates when all the tree nodes in currentLevelTreeNodes are processed. Finally, new tree nodes that are empty (i.e. with zero tree node blocks) are eliminated from newLevelTreeNodes and is then assigned to the currentLevelTreeNodes so that it will be available as input for the next call of BUILD-TREE-LEVEL.

To show how BUILD-TREE algorithm works, we will apply it on the classification example given in Figure 1. Initially, the root tree node is created. The items of the root will be the items of the classification problem \( \{i_1, i_2, i_3, i_4, i_5, i_6, i_7, i_8\} \). The first classifier has two classification conditions (Table 2). Applying the first classification condition will result in \( \text{classifiedItems} = \{i_2, i_4, i_6, i_8\} \). A new tree node block will be created, its items field is set to be classifiedItems and is then added to root. The same process is repeated for the next classificationCondition. The algorithm proceeds to process other classifiers by calling BUILD-TREE-LEVEL for the tree nodes in currentLevelTreeNode. In this case it contains only the root tree node. For the second classifier, the tree nodes in currentLevelTreeNodes will be taken one by one. In this case, there is only one tree node which is root. The tree node blocks within root will be taken one by one. A new tree node will be created as a child for every tree node block in root. The classification conditions of the second classifier are applied on the items of the tree node blocks of root. This call of BUILD-TREE-LEVEL ends when the second level of the tree is built and
currentLevelTreeNode is updated to contain the tree nodes of the second level of the tree (i.e. 1.* and 2.*). The third level of the tree is built by a call for BUILD-TREE-LEVEL for the third classifier and the tree nodes of the second level.

3.2 Update operations
3.2.1 Adding new Items
ADD-ITEMS algorithm (Figure 4) adds new items without the need to rebuild the decision tree allowing incremental growth. Its input is the newItems to add. It runs a level by level where in every level of the tree, the newItems will be added into one or more tree node block(s) at one or more tree node(s) in the current level. A new item may be added to multiple tree node blocks in the same tree node and may exist in multiple tree nodes. This may require a creation of a new tree node block if a corresponding tree node block does not exist. Adding new items may require adding new child tree nodes if a tree node block is created. A new item may be blocked at - one or more - tree nodes at some level if it did not satisfy any of the classification conditions of that level. To avoid considering all the tree nodes in the current level, only the tree nodes to which an item in newItems may exist are maintained in the set newItemsTreeNodes. Initially, newItemsTreeNodes set contains the root tree node only. The newItems should be added to the items of the root. In every iteration of the loop-(a) of the algorithm, the newItemsTreeNodes is updated to be the nextLevelTreeNodes. The nextLevelTreeNodes will be calculated to be the tree nodes of the next level that will have the newly added items. The classification conditions of the current level classifier will be applied one after another on the newItems set (loop-(b)). All classification conditions of the classifier need to be checked because an item may exist in multiple tree node blocks. The classifiedItems are the items from newItems that satisfy the current classification condition of the classifier of the current level. If classifiedItems set is empty, nothing is done and the next classification condition is considered. Otherwise, the tree nodes of newItemsTreeNodes are taken one by one (loop (c)). The intersectionItems of classifiedItems and items of the current TreeNode are calculated. If the intersectionItems set is empty, then no new items should be inserted in the current TreeNode. So, the next classification condition should be considered. However, TreeNode is checked if empty (i.e. contains no tree node blocks). If so, this TreeNode need to be eliminated. This is needed for tree nodes that were created as a child tree node (i.e. with no tree node blocks) while processing the previous level and no tree node blocks were added to it. This means that it continues to have no tree node blocks and need to be eliminated. In case intersectionItems set is not empty, then these items need to be added to the corresponding tree node block within current TreeNode. How to add intersectionItems to TreeNode depends on whether if TreeNode contains a TreeNodeBlock for the current classificationCondition. If exists, the intersectionItems should be added to the items of this TreeNodeBlock. The TreeNodeBlock is checked if it has a child. If it has a child tree node, then the TreeNodeBlock.childTreeNode.items is updated by adding intersectionItems. Also TreeNodeBlock.childTreeNode is added to nextLevelTreeNodes. In case, TreeNodeBlock does not have a child tree node, and the current classifier is not the last level classifier, a newTreeNode is created as a child for TreeNodeBlock and its items field is set to be the items of TreeNodeBlock. This newTreeNode is then added to the nextLevelTreeNodes. This in case a TreeNodeBlock for the current classification condition exists in the current TreeNode. However, if such tree node block does not exist, then a new TreeNodeBlock need to be created, setting its items to the intersectionItems and adding it to TreeNode. If the currentLevel is not the last level, a newTreeNode need to be created as a child for this new TreeNodeBlock. The items of this newTreeNode is set to be the items of newly created TreeNodeBlock. This new child tree node should be added to the nextLevelTreeNodes. No need to create such a child tree node if the current level is the last level. Once all tree nodes of newItemsTreeNodes are processed (loop-(c)), then the next classification condition should be processed. Once all the classification conditions of the current level classifier are processed, the algorithm proceeds to the next level classifier. This requires updating the newItemsTreeNodes to be nextLevelTreeNodes. This process needs to be
repeated until either all classifiers are processed or newItemsTreeNodes becomes empty. The newItemsTreeNode may become empty if no tree nodes where added to nextLevelTreeNodes at the end of loop(b). This may happen before processing the last classifier.

Assume that ADD-ITEMS algorithm is called for newItems = \{i_9, i_{10}\} whose attributes are \(f_1=7, f_2=4, f_3=3\) and \(f_4=8, f_5=3, f_6=4\) respectively into the tree in Figure 1. Both items \(i_9\) and \(i_{10}\) will be added to the root.items and the classification conditions of the root’s classifier are applied. The classifiedItems for the first classification condition will be \(i_9\). The root – which is the only tree node in newItemsTreeNodes - will be processed. The intersectionItems of classifiedItems and root.items will be \(\{i_9\}\). IntersectionItems \(\{i_9\}\) will be added to the items of the corresponding tree node block (1). Also the intersectionItems will be added to the items of the child tree node (1.*). The second classification condition is processed next and classifiedItems will be \(i_{10}\). The intersectionItems of classifiedItems and root.items will be \(\{i_{10}\}\). IntersectionItems \(\{i_{10}\}\) will be added to the items of the corresponding tree node block (2). Also the intersectionItems will be added to the items of the child tree node (2.*). The newItemsTreeNodes will be be updated to be tree nodes (1.* and 2.*). In the second iteration of outmost loop, the classification conditions of the second classifiers are applied on the items of the tree nodes of in newItemsTreeNodes. The first classification condition is not satisfied by \(i_9\) nor by \(i_{10}\) (i.e classifiedItems is empty). The second classification condition is satisfied by both \(i_9\) and \(i_{10}\) (i.e classifiedItems is \(\{i_9, i_{10}\}\)). The tree nodes of newItemsTreeNodes (1.* and 2.*) are taken one by one. The intersection between the newItems and items of tree node (1.*) will be \(i_9\). So, \(i_9\) will be added to the items of the corresponding tree node block (1.2). Also the intersection between newItems and items of tree node (2.*) will be \(i_{10}\). So, \(i_{10}\) will be added to the items of the corresponding tree node block (2.2). The child tree nodes are also added to the items of child tree nodes (1.2.*, and 2.2.*). The child tree nodes (1.2.* and 2.2.*) will be added to the nextLevelTreeNodes. The third classification condition will be applied on the newItems. Only tree node \(i_9\) will satisfy this condition (i.e classifiedItems is \(i_9\)). The intersectionItems of classifiedItems and items of newItemsTreeNodes (i.e. 1.* and 2.*) will be calculated. For tree node (1.*), intersectionItems will be \(i_9\). However, there is no corresponding tree node block for this classification condition in tree node (1.*) So tree node block (1.3) will be created and added. The items field of the new tree node block (1.3) is set to be intersectionItems (i.e. This requires the creation of new tree node (1.3.*) as a child of this new tree node block (1.3). Its items is set to be items of the new tree node block (i.e. (1.3.*) This child tree node (1.3.*) will be added to nextLevelTreeNodes. Once all the classification conditions of the second level are processed, newItemsTreeNodes will be updated to be nextLevelTreeNodes. In the third iteration of the outmost loop, newItemsTreeNodes will include only the tree nodes (1.2.*, 1.3.*) and (2.2.*). The first classification condition of the last level will be applied on the newItems. The classifiedItems will be \(i_9\). The tree nodes of newItemsTreeNodes will be taken one by one. For tree node (1.2.*), intersectionItems will be \(i_9\). So, \(i_9\) will be added to the items of tree node block (1.2.1). Next, the tree node (1.3.*) will be considered. IntersectionItems of classifiedItems (i.e. items of tree node (1.3.*) will be \(i_9\). So, a new tree node block (1.3.1) will be created, its items will be intersectionItems (i.e. \(i_9\)). No child tree node need to be created because the current classifier is the last classifier. The last tree node will be considered next (2.2.*). IntersectionItems of classifiedItems (i.e. items of tree node (2.2.*) will be empty. The second classification condition of the last level will be processed next on the tree nodes of newItemsTreeNode. ClassifiedItems will be \(i_{10}\). IntersectionItems will be empty for tree nodes (1.2.*) and (1.3.*). For tree node (2.2.*), intersectionItems will be \(i_{10}\) and will be added to the items of the corresponding tree node block (2.2.2). Since all classifiers are processed, the algorithm terminates. The tree will be as shown in Figure 5.

3.2.2 Deleting Items

DELETE-ITEMS algorithm (Figure 6) deletes items from every containing tree node. An item may exist in many tree nodes in the same level. Tree nodes of the same level that contain itemsToDelete will be referred to as itemToDeleteTreeNode set. For every level, this variable is updated. The algorithm starts with itemsToDeleteTreeNode containing the root. The classification conditions of the current level classifier are applied on itemsToDelete set generating classifiedItems set. This is required to identify the tree nodes blocks where the itemsToDelete exist in the current level. If classifiedItems is empty, we know of that none of
the itemsToDelete exist in any of the tree node blocks of itemToDeleteTreeNodes that correspond to the current classificationCondition. In this case, nothing need to be done more for the current classificationCondition and the next classification condition is checked next. However, if classifiedItems is not empty, two steps need to be done for every tree node in itemToDeleteTreeNodes: (1) every item in itemsToDelete of the corresponding classificationCondition, (2) the child tree node of treeNodeBlock – if any - should be added to the nextLevelTreeNodes. To achieve this, the tree nodes of itemsToDeleteTreeNodes are taken one by one. The intersectionItems of classifiedItems and the items of the current tree node is calculated. If intersectionItems is empty, then the next treeNode of itemsToDeleteTreeNodes should be considered. However, if not empty, intersectionItems should be deleted from the items of the treeNodeBlock – within treeNode - that corresponds to the current classificationCondition. Also, the child tree node of treeNodeBlock – if any - should be added to nextLevelTreeNodes. If due to this deletion, the items of treeNodeBlock becomes empty, then treeNodeBlock should be removed from treeNode. Once all the tree nodes of itemsToDeleteTreeNodes where processed for the current classificationCondition, the algorithm should proceed to process the next classification condition. Once, all the classification conditions of the current classifier are processed, the algorithm proceeds to the next classifier. Before that, the items fields of itemsToDeleteTreeNodes should be updated by deleting itemsToDelete and itemsToDeleteTreeNodes is updated to be nextLevelTreeNodes. If nextLevelTreeNodes was empty, then no more processing is needed and the algorithm should terminate.

To show how DELETE-ITEMS works, it will be called for the items that were added in the previous example \{i_9, i_{10}\} (Figure 5). Initially, itemsToDeleteTreeNodes is set to be the root. The classification conditions of the root's classifier are applied one after another. The first classification condition is satisfied by \(i_9\), so \(i_9\) should be deleted from the items field of tree node block \(J\) and its child tree node (i.e tree node \(1.*\)) is added to nextLevelTreeNodes. The second classification condition is satisfied by \(i_{10}\), so \(i_{10}\) should be deleted from the items of the tree node block \(2\) and its child tree node (i.e tree node \(2.*\)) is added to nextLevelTreeNodes. Both items, \(i_9\) and \(i_{10}\), are deleted from the root.items and itemsToDeleteTreeNodes is updated to be nextLevelTreeNodes. When the second level is processed, itemsToDeleteTreeNodes contains tree nodes \((1.*\) and \(2.*\)) and the classification conditions are applied on itemsToDelete. None of the itemsToDelete satisfy the first condition so we proceed to the next classification condition. The second classification condition is satisfied by the both items in itemsToDelete where classifiedItems will be \{\(i_9, i_{10}\)\}. This means that the tree node blocks of the second classification condition of the second level will be classifiedItems will be \{\(i_9, i_{10}\)\}. This means that the tree node blocks of the second classification condition of the second level may contain one or more of the itemsToDelete. The tree nodes of itemsToDeleteTreeNodes are taken one by one. First, tree node \((1.*\) is taken, the intersectionItems between classifiedItems and the items of tree node \((1.*)\) will be \(i_9\). So, \(i_9\) will be deleted from the corresponding tree node block (i.e \(1.2\)) and its child tree node \((1.2.*)\) will be added to the nextLevelTreeNodes. Second, tree node \((2.*\) is taken next, the intersectionItems between classifiedItems and the items of tree node \((2.*)\) is \(i_{10}\). So, \(i_{10}\) will be deleted from the corresponding tree node block (i.e \(2.2\)) and its child tree node \((2.2.*)\) will be added to the nextLevelTreeNodes. All tree nodes in itemsToDeleteTreeNodes were processed for the second classification condition. So, the third classification condition is taken. ClassifiedItems will be \(i_9\). The tree nodes in itemsToDeleteTreeNode are taken one by one. For the first tree node \((1.*)\), the intersectionItems between classifiedItems and the items of tree node \((1.*)\) will be \(i_9\). So, \(i_9\) will be deleted from the corresponding tree node block (i.e \(1.3\)). The tree node block \((1.3\) becomes empty, so it will be dropped from tree node \((1.*)\). Its child tree node will be lost as well. For the next tree node \((2.*)\), intersectionItems of classifiedItems and the items of tree node \((2.*)\) will be empty, so we proceed to the next classification condition. After processing all classification conditions for the current level, itemsToDelete are deleted from the tree nodes in the itemsToDeleteTreeNodes. The last step to be done for the second level is to update itemsToDeleteTreeNodes to be the nextLevelTreeNodes which includes the tree nodes \((1.2.*)\) and \((2.2.*)\). When processing the third level, the first classification condition will be applied on the itemsToDelete and classifiedItems will include \(i_9\) only. We then take the tree nodes in itemsToDeleteTreeNodes one by one. We take first tree node \((1.2.*)\). The intersectionItems of classifiedItems and the items of tree node \((1.2.*)\) will be \(i_9\). So \(i_9\) is deleted from the corresponding tree node block \((1.2.1\). We then take the next tree node in itemsToDeleteTreeNodes which is \((2.2.*)\).
However, if the current level is not the last level and terminates. No further processing is needed since no distortion occurred on the tree. However, if classifierToDelete level is not the last level of the tree, a fixing of the tree is required starting from the last level up to classifierToDelete's level. The main idea of fixing the tree is merging tree node blocks in currentLevel into new tree node blocks. This merging relies on the id field for the tree node blocks. Before merging, the id fields need to be fixed. This fixing is done by dropping the numeric field of the id that corresponds to the classifierToDelete Level. This dropping results is raising the currentLevel one level above without copying into an upper level. In general, the id field of a tree node block takes the form \(x_1.x_2....X_{n-1}.X_n\) where \(n\) is the rank of the level where the tree node block exists. The id field will be fixed by dropping the numeric field \(x_i\) where \(i\) is the rank of the classifierToDelete. After fixing their ids, these tree nodes need to be merged. Tree node blocks that are mergable are those tree node blocks with identical id numeric fields – after fixing. Merging a set of a mergable tree node blocks will result in a new tree node block whose items field will be the union of the items of the merged tree node blocks. It is possible that a tree node block may not have any other tree node block to merge with. In this case, it is merged by staying as is with no change. After merging tree node blocks of the currentLevel, these merged tree node blocks should be encapsulated into tree nodes. The merged tree node blocks whose id's numeric fields are identical except for the last numeric field should be encapsulated into the same tree node. If a merged tree node block had no matching tree node blocks, it will be encapsulated by itself in a separate tree node. The last step of the loop requires fixing child-parent links between the merged tree node blocks in currentLevel and tree nodes in the next level. The child of a tree node can easily be inferred from its id. In general, the child tree node of a tree node block with id \(x_1.x_2....X_{n-1}.X_n\) will be the tree node \(x_1.x_2....X_{n-1}.X_n.*\) (* is a wild-card meaning any value). To decide whether if a new iteration is needed, the currentLevel is checked if it is in the level that exists just next to classifierToDelete level. If so, no more iterations are needed. In this case, the algorithm replaces the tree nodes of the classifierToDelete level with the tree nodes of currentLevel. Child-parent links between the previous level and currentLevel need to be fixed. If no previous level exists (i.e. classifierToDelete is the root's), this fix is not needed.

To show how the DELETE-CLASSIFER algorithm works, we will apply it on the first level classifier in Figure 1. Initially, currentLevel will be the tree nodes of the last level (i.e. 1.1.*, 1.2.*, 2.1.*, 2.2.* and 2.3.*). The id fields of the tree node

\[ \text{IntersectionItems is empty, so nothing is done. Since all tree nodes of itemsToDeleteTreeNodes are processed, the next classification condition should be considered. The classifiedItems for the second classification condition will be } i_{10}. \text{ The tree nodes of itemsToDeleteTreeNodes are taken one by one. Starting with tree node (1.2.*), the intersectionItems of classifiedItems and items of tree node (1.2.*) will be empty so the next node in itemsToDeleteTreeNodes is processed (i.e. 2.2.*). The intersectionSet will be } i_{10}. \text{ So, } i_{10} \text{ will be deleted from the corresponding tree node block (2.2.2). Since all classification conditions are processed for the current level, the items fields for the itemsTreeNodeBlocks are updated by deleting the itemsToDelete. Since nextTreeNodeBlock is empty, so will be itemsToDeleteTreeNodes and the algorithm terminates. The tree returns back as it was before adding } i_9 \text{ and } i_{10} \text{ (Figure 1).} \]
blocks are fixed by dropping the first numeric field. The first numeric field is dropped because the classifierToDelete level is the first level. After fixing, the id fields for tree node blocks of currentLevel tree nodes will be from left to right – refer to Figure 1- as follows: (1.1), (1.2), (2.1), (1.2), (2.1), (2.2) and (3.1). Next, tree node blocks with identical ids will be merged. Merging will give five tree node blocks. Tree node block (whose id is 1.1.2 before fixing and 1.2 after fixing) and the fourth tree node block (whose id 2.1.2 before fixing and 1.2 after fixing) are merged into tree node block (1.2) in Figure 8. Also, tree node block (whose id is 1.2.1 before fixing and 2.1 after fixing) and the node block (whose id 2.2.1 before fixing and 2.1 after fixing) are merged into tree node block (2.1) in Figure 8. The other tree node blocks are not mergable. Next, merged tree node blocks need to be encapsulated into new tree nodes. The merged tree node blocks with identical new id's numeric fields except for the last field will go to the same tree node. So, the merged tree node blocks (1.1) and (1.2) will end in new tree node (1.*), tree node blocks (2.1) and (2.2) will end in new tree node (2.*), and (3.1) will be in a separate tree node alone. No fixing with lower tree node blocks is needed now because currentLevel is the last level. Since the currentLevel is not the level next to the classifierToDelete level, another iteration is required. The currentLevel become tree nodes of the next upper level (i.e. the second level in Figure 1). It includes the tree nodes (1.*) and (2.*). The id's of tree node blocks are fixed by dropping the first numeric field from the id. The first field is dropped because the classifier to delete is the first level classifier. So, the id's of tree node blocks, of the second level, from left to right will be: (1), (2), (1), (2) and (3). The first and the third tree node blocks will be merged into one tree node block, the second and fourth tree node blocks will be merged into one tree node block and the last tree node block will stay alone without merging (Figure-8). Next, these new tree node blocks are encapsulated into new tree node. This is because their id are composed of one field only. The child-parent links between the currentLevel and the level (that was constructed in the last iteration) are fixed. This fixing is guided by the new ids. No more iterations are needed since the currentLevel is the level next to classifierToDelete level. So, the tree nodes of the classifierToDelete level are replaced with the new tree nodes of currentLevel tree nodes. The final tree looks as shown in Figure 8.

### 3.2.3 Updating classifiers

Updating a classifier means updating its classification conditions. One way to update a classifier is through a couple of operations: deleting the classifier-to-update and then adding the updated classifier. This operation is expensive since it requires appending a new level to the tree. Instead, we will define a number of operations to update a classifier without deleting then adding classifiers. These operations include: adding, deleting and updating a classification condition for a given input classifier.

ADD-CLASSIFICATION-CONDITION operation is shown in Figure 9. The classifier to update is referred to as classifierToUpdate and the classification condition to add as classificationConditionToAdd. The classifierToUpdate and the classificationConditionToAdd are passed as inputs. Adding classificationConditionToAdd to classifierToUpdate may require adding a corresponding tree node block in every tree node in the level of the classifierToUpdate. This also may result in the creation of a new subtrees descending from these new tree node blocks. The tree nodes of the level of the classifierToUpdate are taken one by one. The current tree node is referred to as treeNode. ClassificationConditionsToAdd is applied on the items of the current treeNode to generate classifiedItems. If empty, nothing need to be done and the algorithm proceeds to process the next treeNode. However, if not empty, a new tree node block should be created and added to treeNode. This is done through creating a new tree node block, setting its items field to be the classifiedItems, adding it to the current treeNode, a child tree node is created, its items is set to be the classifierItems, and is added to the nextLevelTreeNodes. Once all the tree nodes in the classifierToUpdate level are processed, the subtrees that descend from these new tree node blocks should be created. This is done through calling BUILD-TREE-LEVEL algorithm. This requires assigning a value to the global variable currentLevelTreeNodes that will be set initially to the nextLevelTreeNodes. The algorithm will be called once for every lower level until the subtrees are completely built.

To show how this algorithm works, a new classification condition “f5 is divisible-by-3” is to be added to the classifier of the second level in Figure 1. Remember that a classification condition may involve any attributes of the items. Initially, currentLevelTreeNodes will be the tree nodes of the
second classifier: (1.*) and (2.*). The algorithm applies the new classification condition on the items of tree node (1.*). Note that the classification condition to add will only be used in the classifierToUpdate level. Only  is satisfies this classification condition, so the new tree node block (1.4) will be created and added to tree node (1.*). Also a new child tree node (1.4.1) will be created and then added to the nextLevelTreeNodes. The next tree node (2.*) is then processed where the classification condition to add will be applied on its items. The items  and  satisfy this condition. So new tree node block (2.4) will be created and added to the tree node (2.*). Also a new tree node (i.e. 2.4.*) will be created as a child of the new tree node block (i.e 2.4) and added to nextLevelTreeNodes. At this point, the tree nodes of the second level are all processed. Now the subtrees that descend from tree node blocks (1.4) and (2.4) need to be built by calling BUILD-TREE-LEVEL algorithm. The final tree is shown as Figure 10.

The DELETE-CLASSIFICATION-CONDITION operation takes as input classifierToUpdate, classificationConditionToDelete (Figure 11). It requires eliminating the corresponding tree node block – if exists - from every tree node in the classificationConditionToDelete level. This will result in the elimination of all tree nodes that descend from the deleted tree node blocks.

To show how DELETE-CLASSIFICATION-CONDITION algorithm works, we will apply it on the tree in Figure 1. Assume we wish to delete the first classification condition of the second classifier. The algorithm starts with the tree node (1.*). It will delete the tree node block (1.1). This result in deleting the descending subtree from tree node block (1.1). The same process is applied on the tree node (2.*). The tree becomes as shown in Figure 12. Note that the tree nodes and tree node blocks that were deleted are kept for illustration but they actually does not exist.

The UPDATE-CLASSIFICATION-CONDITION algorithm is shown in Figure 13. The classification condition to update is viewed as an object whose condition field is changed. We will refer to the updated classification condition – after update - as updatedClassificationCondition. Updating a classification condition means changing the items fields of its tree node blocks. This requires rebuilding the subtrees that descend from these tree node blocks. The tree nodes are updated level by level. Initially, currentLevelTreeNodes is set to be the tree nodes of the level of the classifier to update. Tree nodes in currentLevelTreeNodes are processed one by one. The updatedClassificationCondition is applied on the items of the current treeNode to find classifiedItems. If classifiedItems is not empty, the algorithm checks whether if a corresponding tree node block already exists for this classification condition before update. If it does not exist, a new tree node block needs to be created and added to the current tree node. The fields of the corresponding tree node, whether created or already existed, need to be updated. Its items field is updated to be the classifiedItems set, a new child tree node is created with its items is set to be the classifiedItems and is added to the nextLevelTreeNodes. By this step, the old tree nodes -if any – descending from treeNode are eliminated. If classifiedItems was empty, the corresponding tree node block – if exists - has to be deleted. Once all tree nodes in currentLevelTreeNodes are processed, the subtrees that are rooted at these updated tree node blocks need to be rebuilt. This is done by calling BUILD-TREE-LEVEL operation. This requires setting the currentLevelTreeNodes global variable to be the tree nodes in nextLevelTreeNodes. The BUILD-TREE-LEVEL will be called for lower level classifiers until the subtrees for the tree node blocks of the updated classification condition are rebuilt.

To show how this algorithm works, the first classification condition of the second level classifier (Figure-1) will be updated to be “f2 is less than or equal to 3”. The algorithm will start by updating the tree nodes in the second level. First, the updated classification condition is applied on the items of tree node (1.*), classifiedItems becomes  . So the items of tree node block (1.1) will be and a new empty child tree node (1.1.*) will be added to nextLevelTreeNodes. Next, tree node (2.*) is processed, the updated classification condition is applied on its items and classifiedItems will be  . The items of tree node block (2.1) will be set to be and a new empty child tree node is added to nextLevelTreeNodes. Next, the subtrees rooted at tree node blocks (1.1) and (2.1) are rebuilt by calling BUILD-TREE-LEVEL and the tree will look as shown in Figure 14.

3.3 Complexity Analysis

The factors that determine the space and time complexity of the operations of this data structure are: (a) the number of the classifiers, since
the number of the levels of the tree equals to the number of classifiers, (b) the number of classification conditions per classifier and (c) the number of items to classify. Because of that, and to simplify the analysis, the complexity analysis of the operations will be based on the maximum number of tree node blocks that are processed. For space complexity, upper bounds are based on the number of tree node blocks in the last level for a set of the tree nodes for a subtree rooted at tree node in level v. For example, for Figure 1, \(N_{\text{nodes}}(C_v)\) will be \(1+2+2*3=9\). \(N_{\text{nodes}}(C_v)\) is the maximum number of tree node blocks in the last level for a set C. For example, in Figure 1, \(N_{\text{last level}}(C)\) = \(|C_1|+|C_2|+|C_3|=12\). \(N_{\text{nodes}}(C,v)\) is the maximum number of tree nodes in level v. For \(v=1\), \(N_{\text{nodes}}(C,v)\) equals to 1. In Figure 1, \(N_{\text{nodes}}(C,2)=|C_2|=2\) and \(N_{\text{nodes}}(C,3)=|C_3|=6\).

We need to explore the time that is needed to construct a single tree node block. The steps for building a single tree node block take constant time except for computing the \(\text{items}\) field. Computing the \(\text{items}\) field requires applying the classification condition – of the tree node block - on the \(\text{items}\) of the containing tree node. The time that is needed to compute \(\text{items}\) field is the sum of the times that are needed to apply this classification condition on every single item in the \(\text{items}\) of the containing tree node. The time that is needed to compute the \(\text{items}\) field for a given tree node block b can be formalized as:

\[
\sum_{i=1}^{v-b} t(b,c,\text{item}_i)
\]

Where \(t(b,c,\text{item}_i)\) is the time that is needed to evaluate the classification condition of the tree node block b (i.e. b.c) on item \(\text{item}_i\), where \(\text{items}\) are the items of the tree node that contains b. Note that the attribute values of the item may affect the time needed to evaluate the classification conditions. Consider for example, determining whether an integer attribute is prime. For example, the time that is needed to determine whether an integer attribute value is a prime number depends on the value of this attribute. The other factor that affects this sum, is the size of the \(\text{items}\) set of the containing tree node. To simplify our analysis, we will assume that the time that is needed to apply any classification condition on any item is less than a fixed value \(l_{\text{max}}\). Based on this assumption, the time that is needed to compute the \(\text{items}\) field of any tree node block will depend on the number of the items of the containing tree node. The number
of items of any tree node is bounded by the number of the input items $k$. So, an upper bound on the time that is needed to find the items field of a tree node block will be $k \cdot t_{\text{max}}$ which is $O(k)$. Since other steps in constructing the tree node block take constant time, we can say that the complexity of constructing a tree node block as whole is $O(k)$.

We will study the complexity of the operations of this data structure based on the above definitions and assumptions. Our approach in the analysis can be summarized as follows: (a) it is worst-case analysis, (b) it considers space and run time complexities, (c) space complexity will be measured in terms of the number of tree node blocks to be created, (d) run time complexity will be measured based on the number of tree node blocks that are accessed by the operation and the time needed to construct/change them, (e) it will consider both cases: ill-formed and well-formed classification.

Starting with BUILD-TREE, the space complexity will be $O(N_{\text{Blocks}}(C_i))$ for both ill-formed and well-formed complexities. For time complexity, applying the classification condition is required for every to-be-created tree node block on every item in the containing tree node. We assumed that the time to create a tree node block will be $O(k)$. In well-formed classification an item may exist only in one tree node in any level. So, the number of classification conditions that will be applied on any item will be $N_{\text{Condition}}(C_i)$. The run time to construct a tree for well-formed classification will be bound by $O(N_{\text{Condition}}(C_i).k)$. However, for ill-formed classification, the same item may have the same classification condition applied multiple times because it may exist in multiple tree nodes in the same level, so the run time will be $O(N_{\text{Blocks}}(C_i).k)$.

For ADD-ITEMS, in terms of space complexity, an $N_{\text{Blocks}}(C_i)$ tree node blocks may need to be created. This is because, in extreme worst case, added items to an empty tree may satisfy all the classification conditions. In well-formed classification, in worst case, $|C|$ tree node blocks may need to be created for one item, one tree node block in each level of the tree. For a set of items to add, in worst case $N_{\text{Blocks}}(C_i)$ tree node blocks may need to be created. In best cases, no tree node blocks need to be created where the items to add will join an already existing tree node blocks. For time complexity, in case of ill-formed classification, some of the items to add will be inserted in every tree node block. So, the time is bounded by $N_{\text{Blocks}}(C_i)$. The same applies for the well-formed classification.

For DELETE-ITEMS, in terms of space complexity, no new tree node blocks need to be created. It may even result in deleting existing tree node blocks, along with all its descending tree nodes. This may happen if its items field becomes empty due to items deletion. This applies for ill-formed and well-formed classification. For run time, in case of ill-formed classification, all classification conditions need to be evaluated once to know the rank(s) of the tree node block where the deleted item may exist in the current level. Knowing the rank(s) of the tree node block(s) that contain(s) the items to delete requires evaluating $N_{\text{Condition}}(C_i)$ classification conditions for the items to delete. Once the tree node block rank(s) in the current level where items to delete may exist is found, in worst case, all the tree nodes in the current level need to be updated. This means that we need to update $O(N_{\text{Nodes}}(C_i))$ tree node blocks. So the time will be $O(N_{\text{Condition}}(C_i)+O(N_{\text{Nodes}}(C_i)))$ to evaluate classification conditions and to update tree node blocks. The same analysis applies for well-formed classification.

For ADD-CLASSIFIER operation, in terms of space complexity, it may result in the creation of a number of tree node blocks that is equal to the number of the tree node blocks in the last level, multiplied by the number of the classification conditions of the newly added classifier ($C_{\text{newClassifier}}$). In the worst case, the maximum number of tree node blocks that will be created will be $O(N_{\text{Blocks}}(C_{\text{newClassifier}}))$. This is for both: ill-formed and well-formed classification. For run time, in worst case and for both ill-formed and well-formed classification, every classification condition of the new classifier need to be applied on every item in the tree node blocks of the last level. For ill-formed classification, in extreme worst case, all the items will exist in every tree node block of the last level. The classification condition evaluation will be repeated for all the items for every tree node block. So, the time to
apply the classification conditions will be \( O(N_{\text{lastLevel}}(C) \cdot \left| C_{\text{newClassifier}} \right| k) \). Time is also needed to construct the new tree node blocks. The number of tree node blocks to be created will be \( O(N_{\text{lastLevel}}(C) \cdot \left| C_{\text{newClassifier}} \right| k) \). So the total time will be \( O(N_{\text{lastLevel}}(C) \cdot \left| C_{\text{newClassifier}} \right| k) + O(N_{\text{lastLevel}}(C) \cdot \left| C_{\text{newClassifier}} \right| k) \) which will be \( O(N_{\text{lastLevel}}(C) \cdot \left| C_{\text{newClassifier}} \right| k) \). For well-formed classification, all items should be tested against the new classification conditions. However, an item appears only once in the last level. So applying classification conditions will cost \( O(\left| C_{\text{newClassifier}} \right| k) \). Also, time is needed to build new tree node blocks which will be \( O(N_{\text{lastLevel}}(C) \cdot \left| C_{\text{newClassifier}} \right| k) \). So the total time will be \( O(\left| C_{\text{newClassifier}} \right| k) + O(N_{\text{lastLevel}}(C) \cdot \left| C_{\text{newClassifier}} \right| k) \) where \( i \) is the level of the classifier to update. For run time, the newly added classification condition needs to be evaluated for the items of every tree node in the level of the classifier to update. It also requires applying the classification conditions for the lower level classifiers for the newly created tree nodes due to the new classification condition addition. In the worst case, for ill-formed classification, the new classification condition need to be evaluated for the items of all tree nodes of the level of the classifier to update. This will require \( O(N_{\text{nodes}}(C,i) \cdot k) \) time. It will also require creation of \( N_{\text{nodes}}(C,i) \) tree node blocks in the level \( i \). So the total time to update level \( i \) will cost \( O(N_{\text{nodes}}(C,i) \cdot k) + O(N_{\text{nodes}}(C,i)) = O(N_{\text{nodes}}(C,i) \cdot k) \). Also a whole subtree should be created as a child of each new tree node block that was added in level \( i \). Every subtree will need evaluation of all the classification conditions of lower levels and creation for new tree node blocks. This will cost \( O(N_{\text{blocks}}(C,i+1) \cdot k) \) for one subtree. We have a maximum \( N_{\text{nodes}}(C,i) \) subtrees. So time will be \( O(N_{\text{nodes}}(C,i) \cdot N_{\text{blocks}}(C,i+1) \cdot k) \). The total time to fix the level \( i \) and to build the lower subtrees will be: \( O(N_{\text{nodes}}(C,i) \cdot k) + O(N_{\text{nodes}}(C,i) \cdot N_{\text{blocks}}(C,i+1) \cdot k) = O(N_{\text{nodes}}(C,i) \cdot N_{\text{blocks}}(C,i+1) \cdot k) \). This is for ill-formed. We will not consider well-formed classification here because it does not apply here according to our well-formed classification definition. That is, all the items are already mapped to the existing tree node blocks and no items will map to the new tree node blocks.

**DELETE-CLASSIFIER** requires replacing the tree node blocks in the levels lower than the level of the classifier to delete, with new tree node blocks that are built by merging. So, no significant space overhead is required. For time complexity, the newly created tree nodes do not require re-applying of the classification conditions. However, there is a computational overhead that is required to merge and encapsulate the tree node blocks. A smart implementation will significantly reduce the cost of merging tree nodes to be less than the time needed to evaluate classification conditions, especially for large \( k \). The number of tree node blocks that will be accessed and updated will be bounded by the number of tree node blocks that exist in lower levels of the classifier to delete. So, the time will be bounded by \( O(N_{\text{blocks}}(C,i+1)) \) where \( i \) is the level of the classifier to delete. This applies for ill-formed and well-formed classification.

In worst case, **ADD-CLASSIFICATION-CONDITION** will create a new tree node block in every tree node in the level to update. Each new tree node block may result in the creation of a whole new subtree in the lower levels of the tree. The maximum number of the tree node blocks to be created for every new subtree will be \( O(N_{\text{blocks}}(C,i)) \) where \( i \) is the level of the classifier to update. The number of subtrees that will be added will be equal to the number of tree nodes in the level of the classifier to update \( N_{\text{nodes}}(C,i) \). So the number of tree nodes to be created is bounded by \( O(N_{\text{nodes}}(C,i) \cdot N_{\text{blocks}}(C,i)) \) where \( i \) is the level of the classifier to update. For run time, the newly added classification condition needs to be evaluated for the items of every tree node in the level of the classifier to update. It also requires applying the classification conditions for the lower level classifiers for the newly created tree nodes due to the new classification condition addition. In the worst case, for ill-formed classification, the new classification condition need to be evaluated for the items of all tree nodes of the level of the classifier to update. This will require \( O(N_{\text{nodes}}(C,i) \cdot k) \) time. It will also require creation of \( N_{\text{nodes}}(C,i) \) tree node blocks in the level \( i \). So the total time to update level \( i \) will cost \( O(N_{\text{nodes}}(C,i) \cdot k) + O(N_{\text{nodes}}(C,i)) = O(N_{\text{nodes}}(C,i) \cdot k) \). Also a whole subtree should be created as a child of each new tree node block that was added in level \( i \). Every subtree will need evaluation of all the classification conditions of lower levels and creation for new tree node blocks. This will cost \( O(N_{\text{blocks}}(C,i+1) \cdot k) \) for one subtree. We have a maximum \( N_{\text{nodes}}(C,i) \) subtrees. So time will be \( O(N_{\text{nodes}}(C,i) \cdot N_{\text{blocks}}(C,i+1) \cdot k) \). The total time to fix the level \( i \) and to build the lower subtrees will be: \( O(N_{\text{nodes}}(C,i) \cdot k) + O(N_{\text{nodes}}(C,i) \cdot N_{\text{blocks}}(C,i+1) \cdot k) = O(N_{\text{nodes}}(C,i) \cdot N_{\text{blocks}}(C,i+1) \cdot k) \). This is for ill-formed. We will not consider well-formed classification here because it does not apply here according to our well-formed classification definition. That is, all the items are already mapped to the existing tree node blocks and no items will map to the new tree node blocks.

**DELETE-CLASSIFICATION-CONDITION** will result in deletion of all the tree node blocks of the classification condition to delete. This will also result in the deletion of all tree nodes descend from the deleted tree node blocks. So, there is no space complexity overhead. For run time, no time is needed to apply classification conditions. However, the algorithm needs to locate tree node blocks to be deleted that spread over many tree nodes in the same level. In the worst case, the number of tree nodes to be deleted \( N_{\text{nodes}}(C,i) \) where \( i \) is the level of the classifier to update.

For space complexity, **UPDATE-CLASSIFICATION-CONDITION**, it may create/reconstruct up to \( O(N_{\text{nodes}}(C,i)) \) subtrees where \( i \) is the level of the classifier to update. These subtrees are rooted at the corresponding tree node blocks of the classification condition to update in
the level $i$. Every subtree may contain up to $N_{\text{blocks}}(C_{i+1})$ tree node blocks. So the maximum number of tree node blocks to be created is bounded by $O(N_{\text{Node}}(C,i) \times N_{\text{blocks}}(C_{i+1}))$ tree node blocks. For run time complexity, the updated classification condition should be applied on the items of all the tree nodes of the level of the classifier to update. For ill-formed classification, it costs $O(N_{\text{Node}}(C,i) \times k)$. A reconstruction of all the tree nodes of the level of the classifier to update. For well-formed classification, an item exists in one tree node block in each level. The classification conditions are applied only once on every item. So, the time to evaluate classification conditions will be $O(N_{\text{Conditions}}(C_{i+1}, i)).$ The time needed to build tree nodes is $O(N_{\text{Node}}(C,i) \times N_{\text{blocks}}(C_{i+1}))$. So, the total time will be $O(N_{\text{Conditions}}(C_{i+1}, i)) + O(N_{\text{Node}}(C,i) \times N_{\text{blocks}}(C_{i+1})).$

4. APPLICATION

To implement this data structure, we will give one application where this data structure could be useful. Given a text message $M = \langle w_1, w_2, ..., w_n \rangle$ that is composed of $n$ words. Given a number of narrators $N = \{n_1, n_2, n_3, n_4\}$ that narrated the same message $M$. Some of the words of $M$ are narrated differently by different narrators. For example, the word $w_i$ may have been narrated by $n_1, n_2$ as $w_i^{(1)}$ and by $n_3, n_4$ as $w_i^{(2)}$. The same narrator may have multiple narrations for the same word. The question is: Can we classify the narrators according to the different ways the message $M$ can be read – as a whole - according to their narrations?

In this application, the number of classifiers will be the number of words with multiple narrations. One classification condition of a word classifier is defined for every narration for that word. For example, given that the word $w_i$ was narrated in three different ways $w_i^{(1)}, w_i^{(2)}, w_i^{(3)}$, then this word classifier can be defined by three conditions. One of the three classification conditions of this classifier is: did narrator $x$ narrate $w_i$ as $w_i^{(j)}$?

As an example, let $M = \langle w_1, w_2, w_3, w_4, w_5, w_6, w_7 \rangle$ be a message that is narrated by the narrators $N = \{n_1, n_2, n_3, n_4\}$. Table 5 shows the different narrations for each word of $M$. The first column shows the words of the message. The second column, shows the different narrations for the corresponding word. Some of the words are narrated exactly the same for all narrators ($w_1$ for example). The third column shows the narrators for every narration. The fourth column shows the classification conditions.

This problem can be solved by building the decision tree through calling BUILD-TREE. The set of narrators will be passed as $\text{items}$. The classifiers are composed of a classifier for every word with multiple narrations and a classification condition for every narration for that word. The tree for this example is shown in Figure 16. Note that the tree is composed of three levels because we have three classifiers: for the words $w_3, w_5$ and $w_7$. The narrators are classified into six classes. The classification results are shown in Table 6.

5. DISCUSSION

In this paper, a data structure level implementation of decision tree for classification purposes – is proposed. We used the known problem solving technique that is called data structure augmentation. One contribution of this research is that it presented a new augmented data structure that can be used to do classification. To the best of our knowledge, we could not find a work that explicitly define such data structure. A lot of work exists in literature where a decision tree is used to do classification through machine learning algorithms; where the classes are implicitly learned from the input data. Although our work required the user to explicitly define the classification conditions, it has the advantage of being generic in that it can work on any data set given the items with well-defined attributes and a set of classifiers that are based on these attributes.

Considering ill-formed classification was an essential requirement when designing this data structure. It may seem unrealistic, too-loose, or even useless, to have a classified item ending in zero or in multiple final classes. However, we found that this requirement deserved the effort for
many reasons: (1) it is the more generic case. Allowing ill-formed classification implicitly included well-formed classifications, (2) some classification problems are ill-formed in nature. The application given in this paper is one example, (3) starting with a rough ill-formed classification problem could be iteratively tuned towards a desired well-formed classification using the proposed update operations, (4) choosing to allow ill-formed classification complicated the update operations of the proposed data structure. These operations would have been much simpler and straightforward if a well-formed classification is assumed.

One advantage of our work is that it bridged a large gap towards the implementation of decision tree for classification. The defined augmented data and operations makes translating this data structure towards a running program straightforward for programmers. Moreover, this data structure can be the basis for an interactive GUI software tool that gives the user the flexibility to design the tree for a specific classification problem that is not yet completely defined. It allows interactive incremental definition, tuning and validation of the tree until it is completely built.

The operations that were presented in this paper are the basic operations. However, there are many more useful operations that may be defined for this data structure to allow wider and more precise manipulation of the decision tree. Examples of such update operations include re-ordering of classifiers, re-ordering of classification conditions and updating the attribute values of the items. More querying operations may be defined like generating detailed statistics regarding the blocked items, overlapping classes and about the final classification.

REFERENCES
Figure 1: Decision Tree Example

Table 3: The data structure basic types.

<table>
<thead>
<tr>
<th>Structure</th>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tree</td>
<td>root</td>
<td>TreeNode</td>
<td>The root of the decision tree</td>
</tr>
<tr>
<td></td>
<td>classifiers</td>
<td>list of classifiers</td>
<td>The classification rules</td>
</tr>
<tr>
<td></td>
<td>items</td>
<td>Items</td>
<td>The items to classify</td>
</tr>
<tr>
<td>lastLevelTreeNodes</td>
<td>set of TreeNodes</td>
<td>TreeNodes</td>
<td>Tree nodes of the last level</td>
</tr>
<tr>
<td>TreeNode</td>
<td>parentTreeNodeBlock</td>
<td>TreeNodeBlock</td>
<td>The parent tree node block</td>
</tr>
<tr>
<td></td>
<td>classifier</td>
<td>Classifier</td>
<td>The classifier of treeNode's level</td>
</tr>
<tr>
<td></td>
<td>items</td>
<td>Items</td>
<td>The items of the tree node</td>
</tr>
<tr>
<td></td>
<td>id</td>
<td>String</td>
<td>Identification string</td>
</tr>
<tr>
<td></td>
<td>items</td>
<td>Items</td>
<td>The items - of the containing tree node - that satisfy this tree node block's</td>
</tr>
<tr>
<td></td>
<td>classificationCondition</td>
<td>ClassificationCondition</td>
<td>The classification condition of this tree node block</td>
</tr>
<tr>
<td>TreeNodeBlock</td>
<td>treeNode</td>
<td>TreeNode</td>
<td>The containing tree node</td>
</tr>
<tr>
<td></td>
<td>childTreeNode</td>
<td>TreeNode</td>
<td>The child tree node</td>
</tr>
<tr>
<td></td>
<td>classificationCondition</td>
<td>ClassificationCondition</td>
<td>The classification condition of this tree node block</td>
</tr>
</tbody>
</table>

Table 4: Classifier basic types

<table>
<thead>
<tr>
<th>Structure</th>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>classifier</td>
<td>classificationConditions</td>
<td>list of classification-Conditions</td>
<td>The classification conditions of the classifier</td>
</tr>
<tr>
<td></td>
<td>treeNodes</td>
<td>set of TreeNodes</td>
<td>The tree nodes of the level where the classifier is applied</td>
</tr>
<tr>
<td></td>
<td>rank</td>
<td>integer</td>
<td>The level of the tree where the classifier is applied</td>
</tr>
<tr>
<td></td>
<td>rank</td>
<td>integer</td>
<td>Rank within classifier</td>
</tr>
<tr>
<td>ClassificationCondition</td>
<td>condition</td>
<td>Condition</td>
<td>A boolean condition that is applied on the attributes of the items</td>
</tr>
<tr>
<td></td>
<td>className</td>
<td>String</td>
<td>A label that identifies the class defined by condition</td>
</tr>
</tbody>
</table>
Figure 2: BUILD-TREE Operation
Figure 3: BUILD-TREE-LEVEL operation
Figure 4: ADD-ITEMS operation
Figure 5: Decision tree after calling ADD-ITEM(\(i^9\))
Figure 6: DELETE-ITEMS operation
**Figure 7: DELETE-CLASSIFIER operation**

```
classifierToDelete

Let currentLevel be TlastLevelTreeNodes

classifierToDelete's level is the last level?

fix the ids of the tree node blocks of currentLevel by dropping the i-th field where i is the rank of the classifierToDelete rank

Merge tree node blocks that have an identical id into new tree node blocks

Encapsulate the merged tree node blocks - with identical id numeric fields except the last - into new tree nodes

Fix the child links between the merged tree node blocks and the next level tree nodes guided by the id values

currentLevel is the level next to classifierToDelete Level?

set currentLevel to be the tree nodes of the next upper level

yes

no

delete TlastLevelTreeNodes

replace classifierToDelete level with currentLevel and fix links between the previous level and currentLevel

end
```

**Figure 8: DELETE-CLASSIFIER example**

```
1
 f_3 = 2
 i_1, i_2, i_3

2
 3 \leq f_3 \leq 4
 i_4, i_5, i_6, i_7

3
 f_3 \geq 4
 i_8

1.1
 f_3 is prime
 i_1

1.2
 f_3 is not prime
 i_2, i_3

2.1
 f_3 is prime
 i_4

2.2
 f_3 is not prime
 i_5

3.1
 f_3 is prime
 i_6
```
Figure 9: ADD-CLASSIFICATION-CONDITION example

Figure 10: Figure 1 after adding "f3 is divisible by 3" condition to the second level classifier
Figure 11: DELETE-CLASSIFICATION-CONDITION operation

Figure 12: Figure 1 after deleting the first classification condition of the second classifier.
Figure 13: UPDATE-CLASSIFICATION-CONDITION operation

Figure 14: Figure 1 after updating the first classification condition of the second classifier.
Table 5: The different narrations for the message M.

<table>
<thead>
<tr>
<th>Words</th>
<th>Narrations</th>
<th>Narrators</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>w1</td>
<td>w1</td>
<td>All narrators</td>
<td>-</td>
</tr>
<tr>
<td>w2</td>
<td>w2</td>
<td>All narrators</td>
<td>-</td>
</tr>
<tr>
<td>w3</td>
<td>w3(1)</td>
<td>All narrators</td>
<td>w3=w3(1)</td>
</tr>
<tr>
<td></td>
<td>w3(2)</td>
<td>n1</td>
<td>w3=w3(2)</td>
</tr>
<tr>
<td>w4</td>
<td>w4</td>
<td>All narrators</td>
<td>-</td>
</tr>
<tr>
<td>w5</td>
<td>w5(1)</td>
<td>n2</td>
<td>w5=w5(1)</td>
</tr>
<tr>
<td></td>
<td>w5(2)</td>
<td>n1,n4</td>
<td>w5=w5(2)</td>
</tr>
<tr>
<td></td>
<td>w5(3)</td>
<td>n1,n3,n4,n5</td>
<td>w5=w5(3)</td>
</tr>
<tr>
<td>w6</td>
<td>w6</td>
<td>All narrators</td>
<td>-</td>
</tr>
<tr>
<td>w7</td>
<td>w7(1)</td>
<td>n1,n2,n3</td>
<td>w7=w7(1)</td>
</tr>
<tr>
<td></td>
<td>w7(2)</td>
<td>n4</td>
<td>w7=w7(2)</td>
</tr>
</tbody>
</table>

Figure 15: Decision Tree for narrators and narrations classification

Table 6: Classification results

<table>
<thead>
<tr>
<th>Classes</th>
<th>Narration</th>
</tr>
</thead>
<tbody>
<tr>
<td>{n2}</td>
<td>&lt;w1,w2,w3(1),w4,w5(1),w6,w7(1)&gt;</td>
</tr>
<tr>
<td>{n3}</td>
<td>&lt;w1,w2,w3(1),w4,w5(2),w6,w7(1)&gt;</td>
</tr>
<tr>
<td>{n4}</td>
<td>&lt;w1,w2,w3(2),w4,w5(1),w6,w7(2)&gt;</td>
</tr>
<tr>
<td>{n1,n3,n5}</td>
<td>&lt;w1,w2,w3(1),w4,w5(2),w6,w7(1)&gt;</td>
</tr>
<tr>
<td>{n4}</td>
<td>&lt;w1,w2,w3(1),w4,w5(2),w6,w7(2)&gt;</td>
</tr>
<tr>
<td>{n1}</td>
<td>&lt;w1,w2,w3(1),w4,w5(2),w6,w7(1)&gt;</td>
</tr>
</tbody>
</table>