A B^+-TREE-BASED INDEXING AND STORAGEOFNUMERICAL RECORDS IN SCHOOL DATABASES

.

ABSTRACT

The need for effective indexing and retrieval of data is paramount in any contemporary organization. However, the use of tree data structure had been effective in this regard as evident in literature. This article gives an overview of B^+-tree data structure, its indexing technique and application in indexing and retrieving students' academic records in the school system in order to make such records flexible. The study demonstrates the indexing and arrangement patterns of some numerical data. In essence, it discusses how to adopt the use of B^+-tree data structure to manage some numerical data in order to enhance indexing, retrieval and modifications of such record. It concludes that good record management results in more convenient indexing and retrieval of students' academic records within the school system.

*Keywords: [*Record keeping, File indexing, File processing system, Database representation*]*

1. INTRODUCTION

Before the advent database management system, organizations relied on file processing systems for storage, modification and retrieval of data. As a result of this, end-users became aggravated with file processing because data was stored in many different files, each organized in a different way. Each file was specialized to be used with a specific application. Needless to say, file processing became bulky, costly and non-flexible when it came to supplying needed data accurately and promptly [1].As a result of this, data redundancy was an issue with the file processing system because independent data files produce duplicate data. A similar problem was lack of data integrity and consistency or standardization of data in a file processing system, consequently making maintenance difficult. As a resultfor these, it was obvious that effective functioning of schools'file management system was impossible without a dynamic storage and retrieval file system.

The place of adequate, accessible and modifiable records in a school cannot be over-emphasized. This is because school events unfold on daily basis which are expected to be filed, retrieved and stored as situation demands. Since the continuous nature of students' achievement is imminent, it implies that the tool with which such records are kept need to be flexible, modifiable and makes data easily accessible.

In any automated file management system, the number of disk accesses required to store or retrieval is important. The time required to access and retrieve a word from high-speed memory is in microseconds while the time required to locate a particular record on a disk is measured in milliseconds. The time required for a single disk access is thousands of times greater for external retrieval than for internal retrieval [2]. The goal in external searching is to minimize the number of disk accesses, since each access takes so long compared to internal computation.

The time required to access the data is dependent on the amount of data to be accessed. Since secondary storage devices are block-based, the data block is their information transfer unit instead of bytes or bits. A database management system usually run on an operating system and its disk access mechanism is important. In order to increment disk performance, the operating system usually includes cache techniques to reduce data time access and implements a block access mechanism, the size of which is a multiple of the block size of the disk device [3].

Contemporary schools need to manage more information than ever before. Consequently, without a solid internal infrastructure for teachers, administrators and departments to share data, critical school and students' information can be lost, thereby leading to a host of problems that can effect of a school's image and overall performance [4]. To remain competitive, school needs a simple solution that can run individual function, connect their entire operation, use the web as a key communication tool and simplify day to day operational responsibilities, giving staff more time with students.

Regarding the practice of continuous assessment which is carried out from time to time, the need for an efficient storage and retrieval system cannot be over-emphasized. This is because students are assessed on daily basis on the three assessment areas of school achievement of cognitive, affective and psycho-motor domains. Since the continuous nature of students' achievement is imminent, it implies that the tool with which such records are kept need to be flexible, modifiable and makes data easily accessible.

In contemporary school practice, the operations of generating, storing and maintenance of school records are becoming increasingly complex as a result of the multi-dimensional nature of school activities which had become evidence as a result of the use of modern technologies in educational practice.

Related studies to this work include [5] which highlights the need for specific indexing techniques for fuzzy databases. The study proposes two indexing principles for flexible querying using possibility and necessity measures on fuzzy attributes. In a similar study, [6] as well as [7], proposed an indexing method to improve the performance of flexible query processing on crisp databases which involves creating one index structure for each fuzzy predicate associated to a crisp attribute. Consequently, each possible satisfaction degree of the fuzzy predicate to the rowsatisfy the predicate at this degree. The technique is costly because of the high number of necessary index structures, which raises the storage and maintenance costs. In a similar study, [8], [9] proposed an indexing technique for fuzzy databases based on a crisp multi-dimensional indexing technique. This proposal is applicable on heterogeneous domains and needs only one indexing structure [10]. The experimental results of this study reveal that the proposed indexing technique outperforms the previous one proposed.

The B+-tree is among the most efficient and widely used for indexing numerical values on block-based data stores. This is because they are balanced (i.e. all leaf nodes are at the same depth from the root), their structure matches block-based data, stores data in orderly manage as each node in the tree corresponds to a disk block, and the data are stored in an ordered fashion, making it efficient to perform range queries as well as exact value look-ups.

A B^+-tree is a self-balancing tree data structure that keeps data sorted and allows searches, sequential access, insertion, and deletions in $O(log(n))$. It is optimized for disk-oriented database management system that read / write large blocks of data. Almost every modern database management system that supports order-preserving indexes uses a B^+-tree. The primary difference between the original B-tree and the B^+-tree is that B-trees stores keys and values in all nodes while B+ trees store values only in leaf nodes. Modern B^+-tree implementations combine features from other B-tree variants, such as the sibling pointers used in the blink-tree [11].

A B^+-tree is defined by its order (or branching factor), which indicates the maximal fan- out (amount of children) each internal node can have. [12] explained that the order of a B^+-tree can theoretically be chosen but in practice, it is determined by the size of the disk blocks in such a way that each node of the index corresponds to a disk block. This kind of data structure is a type of balanced tree used to maintain a collection of sorted records by a key value in a way that allows for efficient insertion, retrieval and removal. A B^+-tree stores the indexed keys and a pointer to its associated record in every tree node, so that when the tree is traversed in-order, the records can be retrieved in the order determined by their key

values. To maintain the balance of the tree, each node is allowed to store a minimum number of entries *d*, and a maximum number of entries *2d*. This policy ensures a minimum node usage of 50%. The value *d* is named the order of the tree. Each leaf node is linked to the next leaf node. Since all the entries are stored in the leaf nodes and these nodes are linked, the performance of sequential processing of the indexed data of B^+-tree is higher than of B-trees, especially when the tree is stored in secondary memory [13].

MATERIAL AND METHODS

2.1. Structure of a B^+-tree node

The search algorithm for a record with an associated key, *k* in a B+-tree begins from the root node. In this case, it is necessary to look for the leaf node which should contain *k*. Consequently, in order to determine the path to the leaf node, it is necessary to use the pointer $p_i$between two consecutive entries: k_i−1 and $k_i$fulfilling k_i−1 ≤ $k < k_1$ andp_0 if $k < k_0$ or p_{2d+1} if $k \geq k_{2d}$. If the corresponding leaf node is known, it is necessary to look for an entry for which $k_i = k$. If it exists, the record associated with *k* is pointed by p_i, otherwise there is no record associated to *k*. A B+-tree node structure is depicted in figure 1.

p_o	K_o	p_1	K_1	p_1	K_1	…	p_{2d}	K_{2d}	p_{2d+1}

Figure 1: A B+-tree node structure

When B^+-tree is used in database indexing, this data structure gets a little complicated because it does not just have a key, it also has a value associated with the key. This value is a reference to the actual data record.

2.2. Insertion in a B^+-tree

Inserting an element into a B^+-tree consists of three main events: searching the appropriate leaf**,** inserting the element and balancing/splitting the tree.The following are essential properties of a B^+-tree before an insertion operation be made.

a. The root has a minimum number of children nodes;
b. Each node, except the root can have a maximum of *m* children and at least *m/2* children; and
c. Each node can contain a maximum of *m-1* keys and a minimum of */m/2˩-1* keys.

In order to insert an element into a B^+- tree, the leaf node is considered. If the leaf is not full, the key is inserted into the leaf node in increasing order.

3. RESULTS AND DISCUSSION

3.1. NUMERICAL EXAMPLE 1

Assume the following four-column table needs to be stored on a database:

Name	Gender	Age	Score
John	M	16	14
Simon	M	17	16
Ruth	F	15	19
Bukonla	M	14	21
Sandra	F	15	29
Kunle	M	16	33

Table 1: A four-column table being stored on a database

The system automatically generates index regarding the items in the table. If the scores of the students are used for indexing, figure 2 is a B^+-tree database page representation of the data contained in table 1 above.

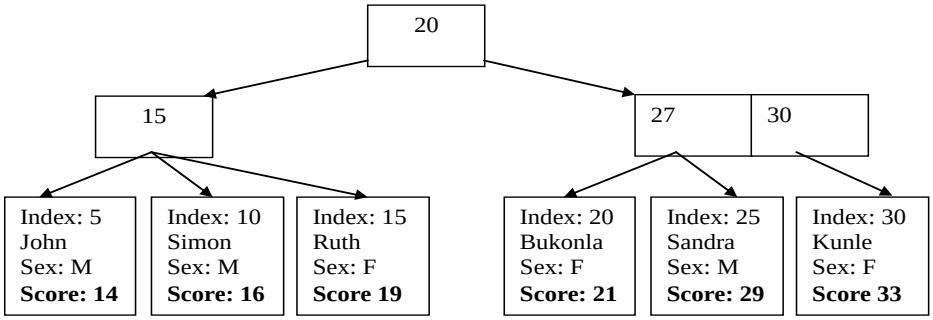


Figure 2: A B^+-tree database page representation of table 1

In figure 2 above, the index values are arranged in binary search tree (BST) order. If for instance, one want to search for index value 21, the value is larger than the parent value of 20, therefore index value 21 is sought to the right of the tree. In essence, there is no need to search for the index value 21 from among index values in the left sub-tree but only at the right sub-tree. This reduces the search time value, consequently reducing search cost and enhancing the feasibility operation on the index value of 21 being sought.

3.2. Numerical Example II

Table 2 contains information about five students in a Mathematics class.

Matriculation Number	Age	Name
23/**0024**	19	Solomon
23/**0035**	20	Mohammed
23/**0042**	18	Kingsley
23/**0048**	18	Noah
23/**0061**	17	Bush

Table 3: Information about five students in a Mathematics class

In table 2, the data being indexed and stored at the tree's leaf nodes is the addresses of records. Figure 3 depicts the abstraction of the index of the values in table 2.

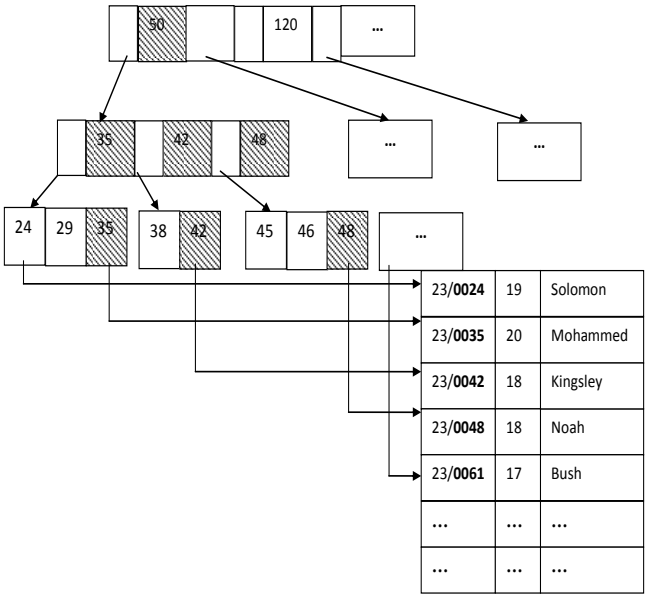


Figure 3: Index representation of table 2

From figure 3, the primary index is built using the matriculation number while the leaf node store the addresses of each record. As a result, the processes involving the location of a record include locating the value of the supplied key, then use the value which is the address of the data to locate the actual record. This type of index is referred to as non-clustered index.

In figure 3, all records are stored in the leaf nodes of the B^+-tree and index and are used as the key to create a B^+-tree while no records are stored on non-leaf nodes. Each of the leaf nodes references the next record in the tree. A database can perform a binary search by using the index or sequential search through every element while traveling through the leaf nodes only.

3.3. Numerical Example III

Consider table 3 below containing the cumulative score (C.S.) of 10 students in Mathematics together with their names and gender. In this case, it is assumed that the minimum score for a pass is 25.

S/N	Name	Gender	Cumulative Score (C.S.)	Remarks (Pass/Fail)
1	Sam	M	38	Pass
2	James	M	27	Pass
3	Murray	F	42	Pass
4	Ahmed	F	18	Fail
5	Lewis	F	59	Pass
6	John	M	53	Pass
7	Mohad	M	25	Pass
8	Isa	F	49	Pass
9	Niyi	F	44	Pass
10	Salako	M	40	Pass

Table 3: Cumulative scores of 10 students with names, gender and performance grade

In order to create a B^+-tree representation of the cumulative scores, the cumulative scores are extracted as follows: **38, 27, 42, 18, 59, 53, 25, 49, 44** and **40**

These values are represented in a B^+-tree of order m=4.
Since order is of m=4, the maximum keys in any node is *m-1* where m=4. Hence the maximum key in any node is 3 while the minimum key in any node is 1. The processes of insertion of each of these elements in the B^+-tree are described below.
The first element is 38 and it is inserted as shown in figure 4 below.

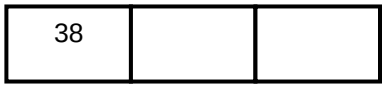


Figure 4: Key value 38 being inserted

The next element is 27. Since 27 is less than 38, the new B^+-tree is as indicated in figure 5.

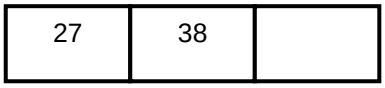


Figure 5: Key value 27 being inserted
The next element is 42. Since 42 is greater than 38, the new B^+-tree is indicated in figure 6.

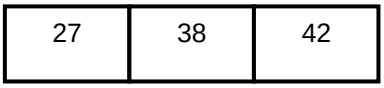
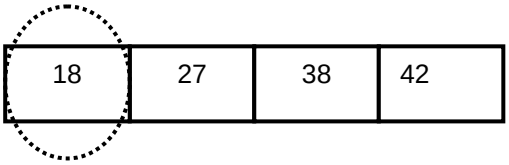


Figure 6: Key value 42 being inserted
The next element is 18. Since 18 < 27, the new B^+-tree is indicated in figure 7.



In figure 5 above, the B^+-tree is of order m=4, then there can be no node with more than three elements. Since figure 5 has 4 keys, we find the middle element(s) which are 27 and 38. In this case, we choose any of these as the middle element, thus making the root node in this case. Let us assume we are left-biased by making the middle element to be 27. Consequently the new B^+-tree is depicted in figure 8 below.

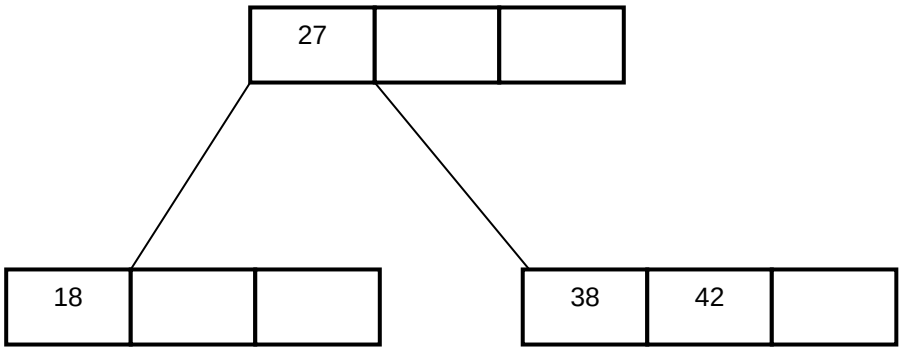


Figure 8: The node being splitted to maintain tree order

In figure 8, it is seen that the binary search tree property of the B-tree is maintained. The root (parent) node has element 27. The element in the left subtree i.e. 18 is lesser than the root element while keys in the right subtree i.e. 38 and 42 are both higher than the root element 27.

The next element to be inserted is 59. Although new elements are not inserted in the root in a B-tree, yet this element is first compared with the root element 27. Since 59 > 27, the new element i.e. 59 is inserted to the right subtree as shown in figure 9 below.

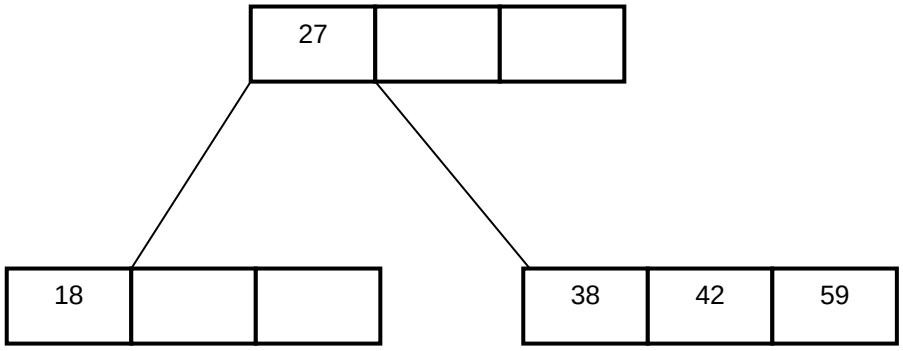


Figure 9: Key value 59 being inserted

The next element to be inserted is 53. This element is compared with the root element 27. Since 53 > 27, the new element i.e. 53 is inserted to the right subtree as shown in figure 10 below.

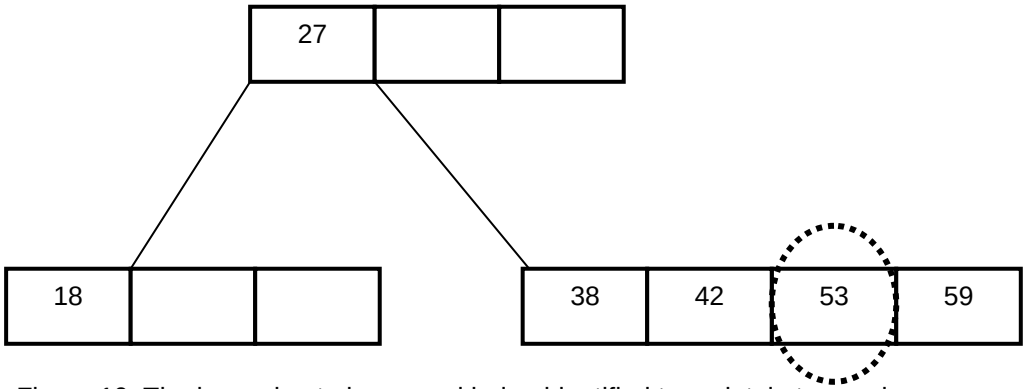


Figure 10: The key value to be moved being identified to maintain tree order

Since the B-tree is of order m=4, there can be no node with more than three keys. Since the right subtree has 4 keys, we find the middle element(s) which are 42 and 53. In this case, we choose any of these two elements as the middle element, thus making another root node in this case. Since we were left-biased in the previous selection, it is better to continue in this way thus making the middle element to be 42. Consequently the new B-tree is depicted in figure 11 below.

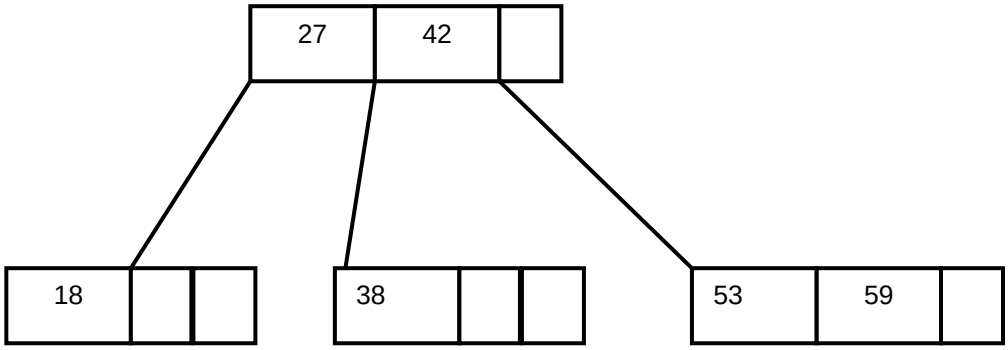


Figure 11: The key values being re-arranged to maintain tree order

The next element to be inserted is 25. Since 25< 27, the new element i.e. 25 is inserted to the right subtree as shown in figure 12 below.

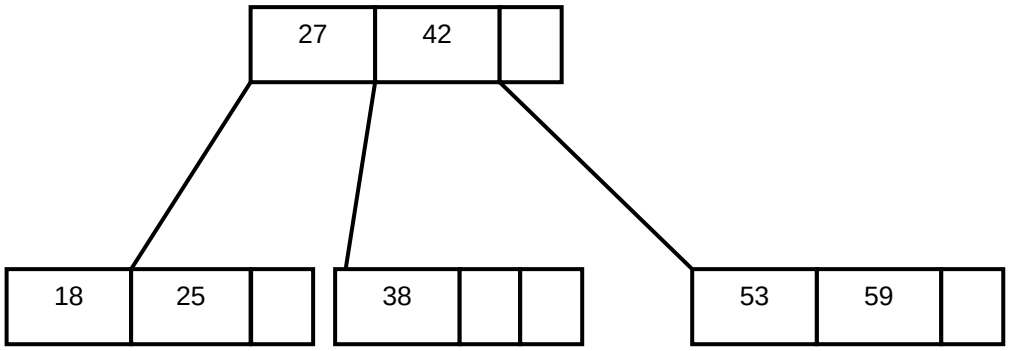


Figure 12: Key value 25 being inserted

The next element to be inserted is 49. Since 49 > 27, the new element i.e. 49 is inserted to the right subtree as shown in figure 13 below.

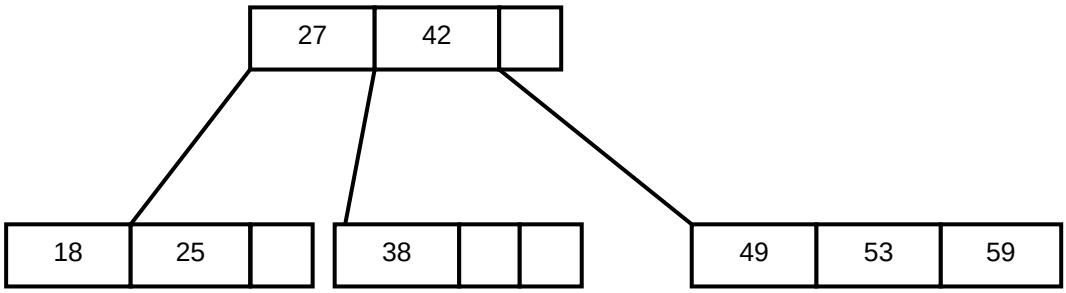
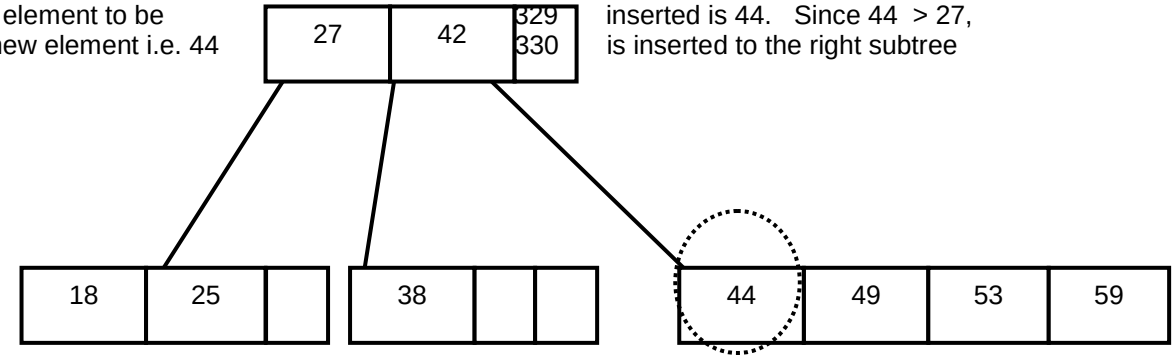


Figure 13: Key value 49 being inserted

The next element to be inserted is 44. Since 44 > 27, 42, the new element i.e. 44 is inserted to the right subtree



as shown in figure 14 below.

Figure 14: Key value 44 being inserted

Being an order m=4 B-tree, there can be no node with more than three elements. Since the right subtree has 4 keys, we find the middle element(s) which are 49 and 53. If element 49 is chosen, the new B-tree is depicted in figure 15 below.

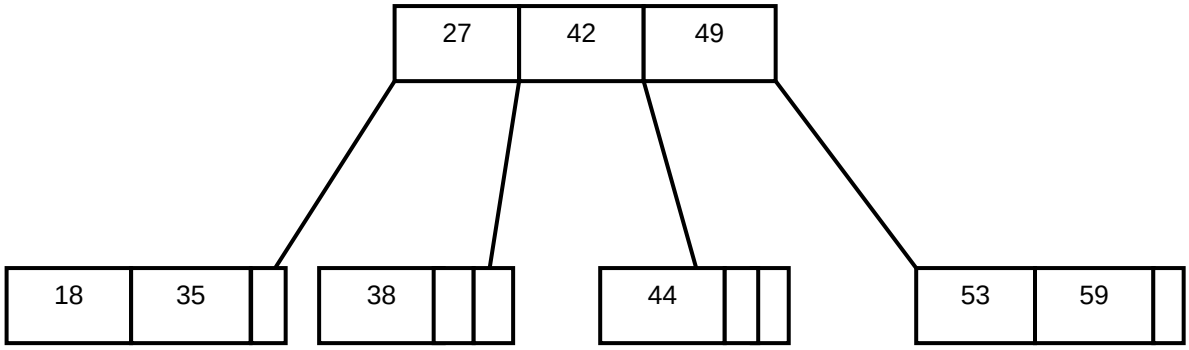


Figure 15: The key values being re-arranged to maintain tree order

The next element to be inserted is 40. Since 40 > 27 but less than 42, the new key i.e. 40 is inserted in-between the subtrees within this range. The new tree is shown in figure 16 below.

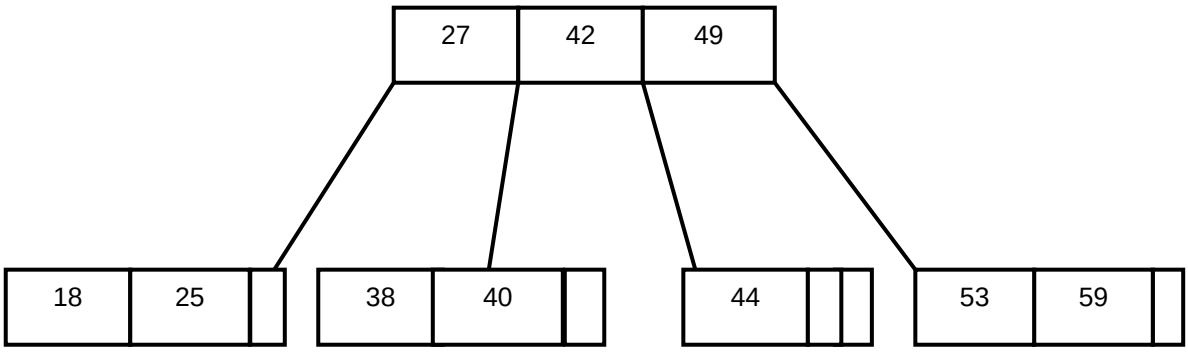


Figure 16: Key value 40 being inserted

The B-tree of order m=4 in figure 14 gives a representation of how the numeric values: **38, 27, 42, 18, 59, 53, 25, 49, 44** and **40** are stored cumulatively as academic performance of the students.The database representation of the B^+-tree in figure 16 is presented in figure 17.

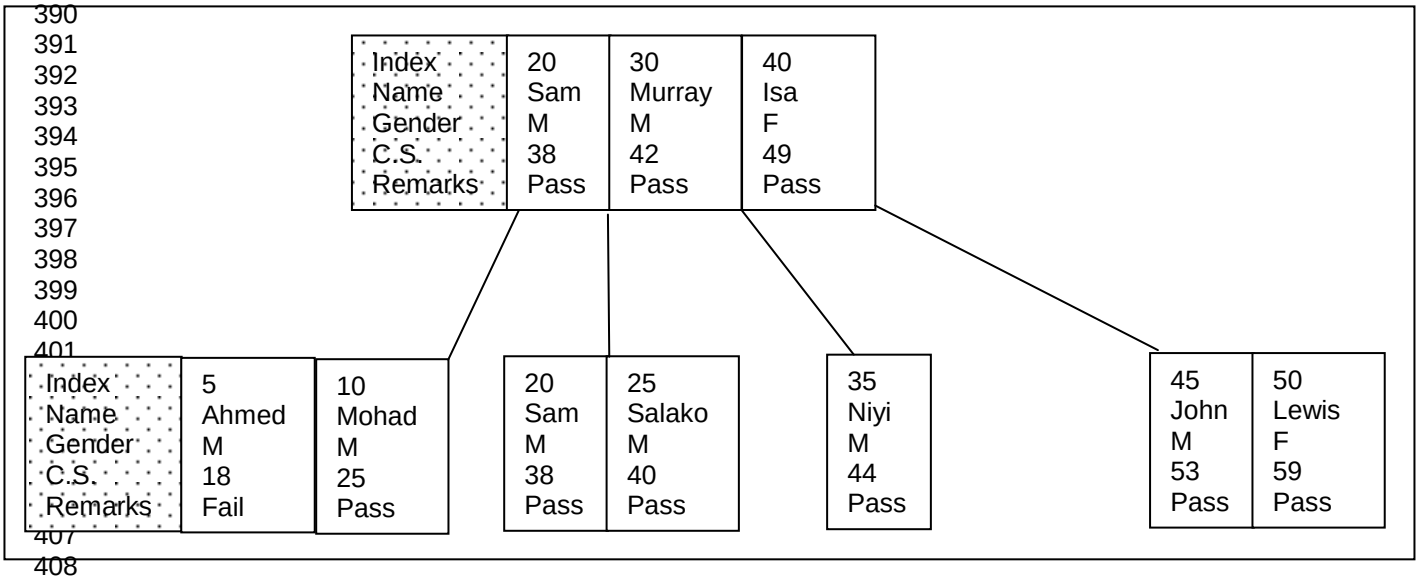


Figure 17: B^+-tree database representation of table 3

In figure 17, the index is a pointer to the database information. This is essential for querying the database and for reference purposes. The processes of insertion and deletion are managed vis-à-vis the index. To be useful and usable, databases must support operations of retrieval, deletion and insertion of data. Since databases are usually large and cannot be maintained entirely in memory, B^+-trees are often used to index the data and to provide fast access. Searching an un-indexed and unsorted database containing *n* key values has a worst case running time of *O(n)*. If the same data is indexed in a B^+-tree, the same search operation will run in *O(log n)*[14]. To perform a search for a single element on a set of five million keys, a linear search will require at most 5,000,000 comparisons. However, if the same data is indexed with a B^+-Tree of minimum degree 10, only 570 comparisons will be required in the worst case.

4. CONCLUSION

A school database keeps a computerized record of students' information which reduces paperwork. Consequently, a school database management system may assist in managing different branches of a school so as to provide a consistent learning environment for students.

Going by the trend in the complexity of data storage and retrieval system in the modern day school system, it is imperative to resort to better means of information storage and retrieval. This is where the concept of B^+-tree in the storage and retrieval of numeric data in the school system becomes imperative. Databases should have an efficient way to store, read, and modify data. B^+-tree provides sequential search capabilities in addition to the binary search, which gives the database more control to search non-index values in a database.

ACKNOWLEDGEMENTS

Nil

COMPETING INTERESTS

Authors have declared that no competing interests exist

AUTHORS' CONTRIBUTIONS

The manuscript was written, read and approved by all the authors for submission. The compilation of references was done by Mr. A. D. Gbadebo.

CONSENT

Nil

ETHICAL APPROVAL

Nil

REFERENCES

[1] Okello, C. O. *Design and Implementation of a Students' Database Management System*: A Case Study of Kampala International University. A Bachelor of Computer Science Project at the Kampala International University. 8-9. 2010

[2] Ahmad, A., Faisal, H. and Jameel, A. M. *E-School –School Management System*. A Bachelor Degree Project in Information Technology. Department of Management Information Systems. University of Palestine. 2016. 5-6.

[3] Kruse R. L., Ryba A. J. Data Structures and Program Design in C++, New Jersey, USA., Prentice-Hall Inc. (2000)

[4] Barranco, C. D., Campaña, J. R. and Medina, J. M. A B+-Tree Based Indexing Technique for Fuzzy Numerical Data. Fuzzy Sets and Systems 159 (2008) 1431 – 1449

[5] De-Mol, Barranco and De-Tré Indexing Possibilistic Numerical Data Using Interval B+-Trees. Fuzzy Sets Systems. (2020) 14(22). 1-17. https://doi.org/10.1016/j.fss.2020.04.011

[6] Bosc, P. and Pivert, O. Fuzzy querying in Conventional Databases. Fuzzy Logic for the Management of Uncertainty, Wiley, New York. , 1992. 645–671.

[7] Petry, F.E. and Bosc, P. Fuzzy Databases: Principles and Applications. International Series in Intelligent Technologies, Kluwer Academic Publishers, Dordrecht, 2016.

[8] Yazici, A. and Cibiceli, D. An Index Structure for Fuzzy Databases.Proceedings Fifth IEEE International Conference on Fuzzy Systems, 2(3) 1996. 1375–1381.

[9] Yazici, A and Cibiceli, D. An Access Structure for Similarity-Based Fuzzy Databases. Information Science 115 (14). 1999. 137-152

[10] Medina, J.M., Barranco, C.D. and Pons, O. Building and Evaluation of Indexes for Possibilistic Queries On A Fuzzy Object-Relational Database Management System, in: 2017 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE), IEEE, 2017. 1–6.

[11] Medina, J.M., Barranco, C.D. and Pons, O. Evaluation of Indexing Strategies for Possibilistic Queries Based on Indexing Techniques Available in Traditional RDBMS. International Journal of Intelligent Systems 31(12) (2016). 1135–1165.

[12] Medina, J.M., Barranco, C.D. and Pons, O. Indexing Techniques to Improve the Performance of Necessity-Based Fuzzy Queries using Classical Indexing of RDBMS, Fuzzy Sets Systems. 351 (2018) 90–107.

[13] Barranco, C. D., Campana, J.R. and Medina, J.M. A Low Implementation Cost Alternative for Indexing Fuzzy Numerical Data. 2007 IEEE International Fuzzy Systems Conference, July 2007.1–6.

[14] Microslav, B. and Petra, G. Analysis of B-Tree Data Structure and its Usage in computer Forensics. Proceedings of the 21st Central European Conference on Information and Intelligent Systems. (2020) 423-429.