Bo-Tree: An Efficient Search Tree

Sumit S Shevtekar[1], Sayeed Khan[2], Sanket Jhavar[3], Harsh Dhawale[4]

[1]Professor, ME Computer Engineering, Department of Computer Engineering, Pune Institute of Computer Technology
ssshevtekar[at]pict.edu

[2]Department of Computer Engineering, Pune Institute of Computer Technology, Pune, India
khansayeed3664100[at]gmail.com

[3]Department of Computer Engineering, Pune Institute of Computer Technology, Pune, India
khansayeed3664100[at]gmail.com

[4]Department of Computer Engineering, Pune Institute of Computer Technology, Pune, India
harshdhawale2404[at]gmail.com

Abstract: ***Large amounts of data are being produced today using a variety of automated data collection tools. This data management and processing requires a time-consuming operation that calls for an effective search algorithm. This study introduces the "BO-Tree, " a productive search tree with a balanced "O" structure. For effective data searching, nodes in the BO-Tree are arranged into a number of levels, with each level being divided into sections. To find data in a BO-Tree, a Reference array is generated and used. Each level and its sections' minimum and maximum values, as well as their corresponding addresses in the BO-Tree, are stored in the reference array. When using a Reference array, BO-Tree is search-efficient in all circumstances of time complexity.***

Keywords: Data collection, data management, data searching, search-efficient, reference array, time complexity

1. Introduction

Developers often struggle with organising vast amounts of data, but thanks to years of research, this difficult chore has become easier. In order to create a data structure for this purpose, numerous techniques are presented. A data structure is a way to store data in a certain format for effective data organisation and data retrieval. Data structures typically fall into one of two categories: Data is stored in linear data structures in a sequential way, with connections between elements. Non-linear data structures, such as Trees and Graphs, store information based on relationships rather than sequentially. Any data structure should perform the responsibilities listed below:

1) Data addition and removal should be simple.
2) It should retrieve the necessary information and declare the search status.

All data structures have a serious problem with algorithm performance, which is measured on the basis of retrieval performance, which should be at a minimum. This study introduces the "BO-Tree, " a revolutionary non-linear data structure that stores vast amounts of data effectively and speeds up searches.

Identify applicable funding agency here. If none, delete this.

2. Related Work

The capability of a data structure to retrieve the stored data and the simplicity of accessing the necessary structure pieces are both considered in its evaluation. In this study, we introduced the "BO-Tree, " a tree that is based on a non-linear data structure. A BO-tree is a group of nodes arranged in the following manner among several levels:

1) Data must first be sorted before being placed into nodes.
2) Sorted data is then entered clockwise into nodes, and data is filled in level order.

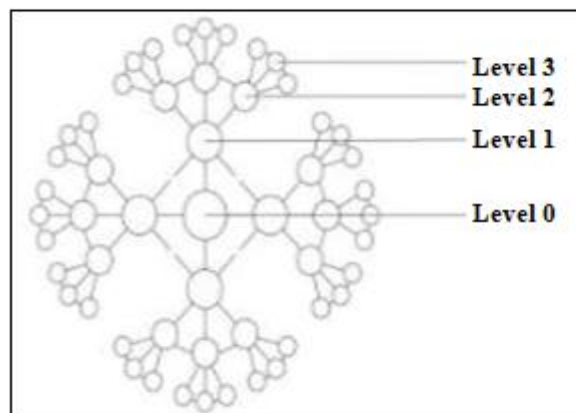


Figure 1: Structure of "BO- tree"

1) Terminology used in this paper

a) Node: Five fields make up the structure: one field stores data, one stores the address of a sibling, and the remaining three fields point to the structure's child nodes. Figure 2 shows that nodes at all levels have the same structure.
b) Level 0: Since the fourth child has no siblings, the Data field is NULL, and the sibling pointer stores the address of the fourth child. When the level rises, it means that the node(s) in the previous level are the current level's parents.

Licensed Under Creative Commons Attribution CC BY

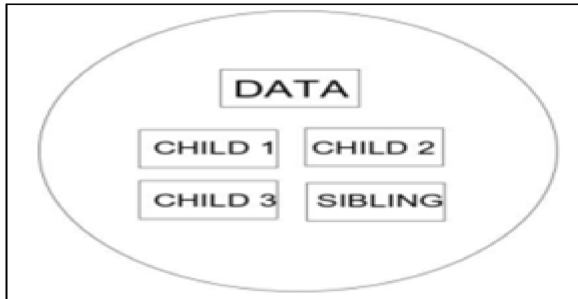


Figure 2: Structure of Node

c) Level 1: The four nodes in Level 1 are each connected to a node in Level 0, and the nodes in Level 1 are linked together. These four Level-1 nodes are children of the Level-0 node and are seen in Figure 3. The data should be placed into level 1 in a clockwise direction, with Node-A holding the data with the lowest value and Node-D holding the highest value. Level-1 contains 4 nodes, so 4*3(level-1) = 4*3(1-1)

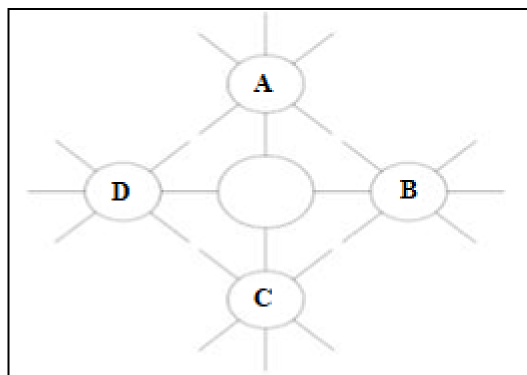


Figure 3: Structure of Level -1 of tree

d) Level- 2: Each Level-1 node has three child nodes, and Level-2 siblings are connected to one another. Level-2 nodes are the offspring of Level-1 nodes. The insertion of data is done clockwise, same like in Level-1. The highest value is kept in the rightmost child of Node-D of Level-1, whereas the least value of Level-2 is kept in the leftmost child of Node-A of Level-1. There are 12 Level-2 nodes, or 4*3(level-1) = 4*3(2-1) = 12.

e) Level- 3: Level-3 nodes are the children of the Level-2 nodes. Each Level-2 node will have 3 child nodes. Node and sibling nodes of this level are interconnected to each other. Here also insertion is in a clockwise manner from left to right. Sample Level-3 tree is shown in fig 4 Number of Level-3 nodes are 36 i.e. 4*3(level-1) = 4*3(3-1)=36

f) Level- K: Each Level-(k-1) node will have 3 child nodes, making Level-k nodes the offspring of Level-(k-1) nodes. From left to right, data is saved and inserted in a clockwise direction. There are 4*3(k-1) nodes in Level-k.

g) Reference Array: A reference array keeps a record of the nodes' addresses, minimum and maximum values in a 2-D array structure. Fig. 5 depicts the array's structure. The number of columns in an array is equal to 2*the number of tree levels. Two columns are used to represent each level: one to record the lowest or maximum value, and the other to store the reference address of the node that has that value. Pair of each two rows in the Reference array, starting at the top, denotes the heritage of Level-1 Nodes.

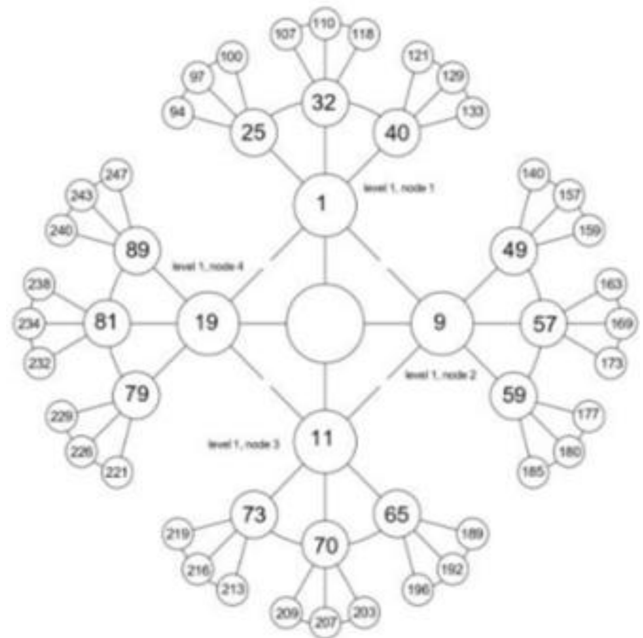


Fig 4. 3 - Level BO-Tree with data inserted

1	ADDR OF 1	25	ADDR OF 25	94	ADDR OF 94
1	ADDR OF 1	40	ADDR OF 40	133	ADDR OF 133
9	ADDR OF 9	49	ADDR OF 49	140	ADDR OF 140
9	ADDR OF 9	59	ADDR OF 59	185	ADDR OF 185
11	ADDR OF 11	65	ADDR OF 65	189	ADDR OF 189
11	ADDR OF 11	73	ADDR OF 73	219	ADDR OF 219
19	ADDR OF 19	79	ADDR OF 79	221	ADDR OF 221
19	ADDR OF 19	89	ADDR OF 89	247	ADDR OF 247

Fig 5. Structure of Reference Array for the 3-level BO-Tree

INode A values, as well as the lowest and maximum values of its children, are stored in row 0 of the above array. The Node-B in Level-1 and its children's minimum and maximum values are stored in Rows 2 and 3. The Node-C in Level-1 and its children are stored in Rows 4, 5 and 6, respectively, while the Node-D in Level-1 and its children are stored in

Rows 6, 7. The address values of the nodes corresponding to columns 0, 2, 4 are stored in columns 1, 3, 5.

2) Operations in "BO- Tree" 1. kth–Level tree traversal:

A way of sequentially accessing the data in the tree from Level-0 to the last node is known as traversing the tree. Take into account a Kth-Level tree where L is the level incremental variable and the method loops from L to K. A node pointer is used to navigate a tree.

Algorithm
Node *tmp;

while (L¡=k)

Licensed Under Creative Commons Attribution CC BY

while(i¡8)

tmp=reference_array[i][j];

while(tmp != NULL)

print(tmp→data);

tmp = tmp→sibiling;

end while

i=i+2;

end while

j=j+2;

i=0;

end while

2. Search operation in “BO-Tree”.

The search function is the “BO- Tree’s” key benefit. The steps listed below can be used to retrieve the search element. 1. Determining the search element’s level.

3) **Determining the level and section of the search element.**

4) **Determining the parent node of the preceding level’s search element.**

5) **Finding the search element among the children of the previously found parent node.**

The rightmost children of Level-1’s Node-D store the maximum value elements of a level, and the same values are also stored in the Reference array along with their addresses. The level of the search element can be determined by comparing the maximum values of each level from the reference array with the search element. Once the level of the search element has been determined, we can use that level’s maximum values for the sections of the tree to determine which section the search element may be present in.

Once the section has been located, its parent level is recognised and its data in the child-3 of each node of its parent level is compared with the lookup element. The child node whose data exceeds the search element is chosen. The child of the chosen node is now compared to the search element. Display ”Element is found” in the tree if the search element is discovered; else, display ”Element is not found.” The outcomes section of this procedure is taught with a condensed example. In Fig. 6, the search process in BO-Tree is shown.

Algorithm

Let ‘s’ be the search element in a K-level tree. while

(s>reference _ array[7, i] i<2k)

i=i+2;

end while

while (reference_array[j, i] j<8)
j=j+2;

Licensed Under Creative Commons Attribution CC BY

end while

if (i>2k j>8)

print ("not found");

end if

Node *ptr = reference array[j, i+1]; Node *ptr1= reference array[j-1, i+1]; if(ptr→data == s || ptr1→data ==s) print("found element"); else

ptr= reference array[j-1, i-1];

while (ptr != NULL)

ptr1= ptrchild3;

if (ptr1data <=s)

search 3childs of ptr for search element if (found)

display ("found element"),

else

display ("not found");

break;

end if

ptr=ptrsibiling;

end if

end while

end if

The maximum number of comparisons needed to locate a search element in a K-Level tree is K+4*3 (k-1) +2, where K is the level of the element, 4*3 (k-1) is to locate the parent node of the search element, and two comparisons in the children of the parent node are required because one child has already been compared.

In level 2 Section C, compare 207 to the rightmost child of each node.

4. Since the rightmost child of 207 is 209, which is higher than 207, we can determine that parent of 207 is 70.

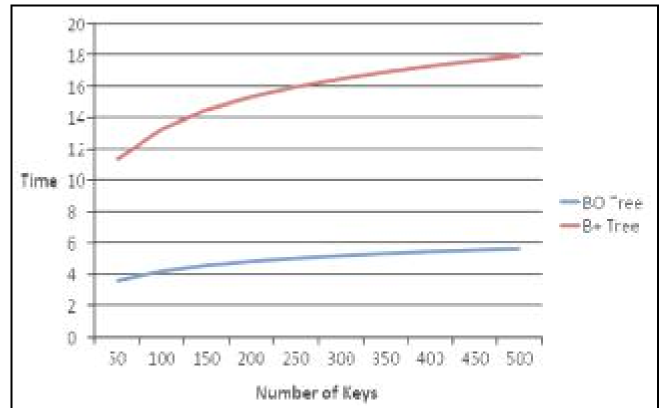
5. Now contrast 70's other two children with 207. The display element is finally located. Figure 6 illustrates the comparison of the BO-tree and B+-tree, and it is very evident that our data structure works better. all cases of time complexity.

A. Best Case Analysis

The ideal situation is when the time complexity is O and the needed data item can be found at either the minimum or maximum entry of the kth level (1). For instance, in Fig. 3, searching for every element in the Reference array has an O (1) +level number-1 time complexity.

B. Average Case Analysis

The typical situation is when the necessary data item is located at the level's lowest node or maximum node, respectively, and the temporal complexity is O(log3(n/4). For instance, searching for items in fig. 3's level 3 external nodes of sections results in time complexity of $O(\log_3(n/4))$.



Figure

3. Results

Java experimentation was done to gauge the suggested data structure's computational complexity. Let's look at an illustration of the BO-Tree search operation; Section 3.2 describes how to search an element in the BO-Tree. Let 207 be the search element in the Fig. 4's 3-level BO- Tree. The steps of this technique are illustrated as follows:

1) To begin with, compare level 207 to each member in the last row of the reference array in figure 5.
2) Since 207 is less than 247, we may determine that 207 is at level 3. Now to locate the level 3 section of the 207. In column 5, compare 207 to the odd number of rows.
3) Because 207 is less than 219, we may determine that Section-C contains it. Now to locate the 207's parent node.

Worst Case Analysis

The worst case scenario is when the necessary data item is located between the maximum and minimum nodes of the appropriate level and time complexity O(log3(n/4))

For instance, searching for items in fig. 3's internal node of sections at level yields an O(log3(n/4))-time complexity. The data structure discussed above has many uses since it can quickly search through a huge amount of data to find the specific data item that is needed. Where there is a need for both massive data storage and an effective search method, such as in the share market database, BO-tree can be employed.

The data structure introduced above has several applications, as it has efficient time complexity for searching the required

Licensed Under Creative Commons Attribution CC BY

data item from a large amount of data. BO–tree can be used, where there is the necessity to store a huge amount of data and for an efficient search technique, such as in the share market database

4. Conclusion

This work introduced a novel, effective search tree called ”BO- Tree, ” which is level-structured with sections at each level. To hold the maximum and minimum values of each level and its sections, BO-Tree uses a Reference array. The search operation uses a reference array, which reduces the worst-case and best-case time complexity to O. (1). When compared to B+-tree, BO-tree has a more effective search operation algorithm. Research fields with large datasets, such as Big Data analytics and Data Science, can use BO-tree.

References

[1] Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., Stein Clif-ford (2009). Introduction to Algorithms, Third Edition (3rd ed.). The MIT Press. ISBN 978-0262033848.
[2] Black, Paul E. ”data structure”. Dictionary of Algorithms and Data Structures. National Institute of Standards and Technology, 2004.
[3] J. Harris and A. Greca, ”The evolution of data structures, ” 34th Annual Frontiers in Education Savannah, GA, 2004, pp. S3H/9-S3H1Vol.3.doi: 10.1109/FIE.2004.140879050.
[4] K. Kumar, M “Y-Trees: An extending non-linear data structure for better organization of large-sized data”, 2017 Third International Conference on Research in Computational Intelligence and Communication Networks (ICRCICN), Kolkata, 2017
[5] R. Bhavani Yerram, Jaya Krishna Bhonagiri, ” An Efficient Sorting Algorithm for binary data”, 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT), IIT Kharagpur, 2020.
[6] Y. Chandramouli, B., Prasaad, G., Kossmann, D., Levandoski, J., Hunter, J., and Barnett, M., “Faster: an embedded concurrent key-value store for state management, ” Proceedings of the VLDB Endowment, vol. 11, no. 12, pp. 1930–1933, 2018.
[7] Atiyah, A., Jusoh, S., and Almajali, S., “An efficient search for context-based chatbots, ” in Proceedings of the 8th International Conference on Computer Science and Information Technology. IEEE, 2018.

Licensed Under Creative Commons Attribution CC BY