Global Journal of Computer Science and Technology: C
Software & Data Engineering
Volume 16 Issue 3 Version 1.0 Year 2016
Type: Double Blind Peer Reviewed International Research Journal
Publisher: Global Journals Inc. (USA)
Online ISSN: 0975-4172 & Print ISSN: 0975-4350

Bottom-Up Update Mechanism for Re-Structured Complete Binary Trees

By Mevlut Bulut

University of Alabama, United States

Abstract- This paper introduces a bottom-up update mechanism together with a non-recursive initial update procedure that reduces the required extra memory space and computational overhead. A new type of tree is defined based on a different geometrical interpretation of Complete Binary Trees. The new approach paves the way for a special and practical initialization of the tree, which is a prerequisite for an implementation of unilateral update operation. The details of this special initialization and the full update procedures are given for Complete Binary Trees. In addition, a comparison is on is made between the introduced update method and the bilateral update methods in terms of different performance related metrics.

Keywords: *data structure, complete binary tree, CBT, sCBT, unilateral update, bottom-up update, replacement selection.*

GJCST-C Classification : *I.1.2, I.2.2*

Strictly as per the compliance and regulations of:

Bottom-Up Update Mechanism for Re-Structured Complete Binary Trees

Mevlut Bulut

Abstract- This paper introduces a bottom-up update mechanism together with a non-recursive initial update procedure that reduces the required extra memory space and computational overhead. A new type of tree is defined based on a different geometrical interpretation of Complete Binary Trees. The new approach paves the way for a special and practical initialization of the tree, which is a prerequisite for an implementation of unilateral update operation. The details of this special initialization and the full update procedures are given for Complete Binary Trees. In addition, a comparison is made between the introduced update method and the bilateral update methods in terms of different performance related metrics.

Keywords: data structure, complete binary tree, CBT, sCBT, unilateral update, bottom-up update, replacement selection.

I. Introduction

At the center of the modern programming paradigm rises the art of obtaining the maximum performance out of a given computer system with limited resources, e.g. computational power, memory or I/O operation capabilities. In designing comparison based algorithms such as searching and sorting, in order to circumvent these limitations, tree formation was suggested a long time ago[1] and it has been widely used since then. The main idea of forming a tree or treating a given array as a tree is to minimize the number of comparisons as close to the theoretical minimum as possible. Although there are many different techniques for the formation (or branching), setup (usage of nodes and node hierarchy), traversing (top-down, bottom-up; preorder, in order, etc.), and initialization of trees (recursive and iterative) new attempts are still being made to improve the efficiencies of these algorithms by optimizing the usage of the limited resources.

As explained in the next section, a new definition for the root node together with a new geometric interpretation of tree formation is proposed. Although the introduced novelties do not change the number of comparisons for the basic tree operations, it brings considerable reduction in required memory space, computational overhead, and number of accessed memory locations.For all the graphical descriptions, only Complete Binary Tree (CBT) structures will be used throughout the article, however the introduced concepts can be applied to other types of trees as well.

The bottom-up update mechanism can simply be described as a unilateral traversing of the nodes from a leaf host to the root. Unlike the bilateral update mechanism, which is based upon comparing two sister node contents followed by the registration of the winner in the parent node, the unilateral update mechanism requires that the overall winner of the previously done consecutive comparisons should be compared to the content of the parent node. If the parent node content is not the winner of this comparison, then the consecutive parent nodes are checked until a parent node content wins, at which point the winner item and the parent node content are swapped. The iteration of this procedure goes on until the root node is reached, where the global winner is registered.

This article introduces a modified bottom-up update mechanism which differs from the previously suggested unilateral implementations[2] in terms of the required auxiliary memory space, the initial update technique, and the overhead reduction during the update operations thanks to the elimination of the redundant nodes from CBTs. As a result, the overall implementation of a bottom-up update operation gets simpler, lighter, and faster.

II. Geometric Definition

Analogous to real trees, the definition of an abstract tree with a stem is suggested (Figure-1).The zeroth node is placed at the end of the stem and utilized as the root of the tree. A CBT with such a structure can be called a stemmed CBT (like most of the trees in the real world). Any Stemmed CBT (sCBT) can be decomposed into smaller sCBTs. In this regard, the smallest sCBT shell encompass two nodes, one of which characterizes the body of the tree and the other one is the root. This definition leads to a new way to compose and decompose a given tree. Figure-2depicts how two minimal sCBTs are combined together. One can decompose a given sCBT along a path from a leaf node to the root. In cases, the sCBT is utilized for replacement selection[3] or priority queue applications [4] then the logic dictates the path of the overall winner to be chosen as the decomposition path. The decomposition will be outlined in the 'initial update' section.

Author: 615 Idlewild Cir Apt 25 Birmingham, Alabama, USA.
e-mail: mevlutbulut@yahoo.com

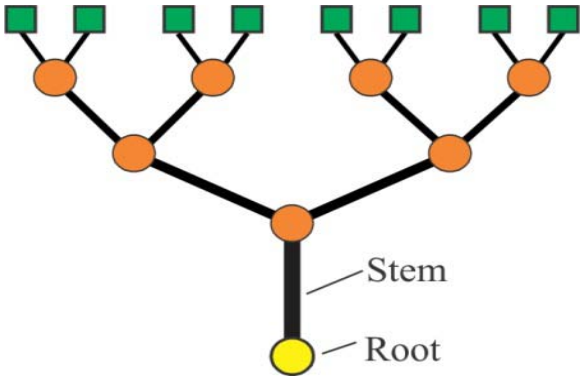

Figure 1: Proposed abstract tree

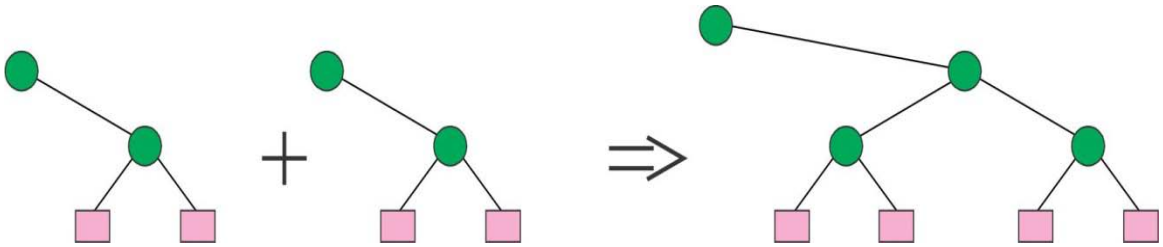

Figure 2 : Two minimal sCBTs are combined through modifying one bond and reforming another, without introducing any new node. Here, note that two regular CBTs cannot be combined without adding a new node.

III. Unilateral Update Versus Bilateral Update

A CBT setup with loser elements rather than winner elements was first suggested by[5] with a coined name 'loser tree', as opposed to 'winner tree', based upon the fact that each and every key appearing in an internal node is a loser exactly once, champion being the only exception. Although they are all losers exactly once, they are the winners of all comparisons up to their current levels. This property is not so different from the case of so called 'winner tree' setup. The logic is the same: both of them promote the winner towards the root. Therefore, there is no point for calling one of them a 'loser tree' and the other one a 'winner tree'. The difference between these two tree setups is that their geometries are different. The difference is dictated by the geometry not by the selection procedure. Therefore, 'winner tree' and 'loser tree' naming convention is abandoned here, instead CBT and sCBT are used to imply the two different geometries and the corresponding bilateral and unilateral update mechanisms respectively.

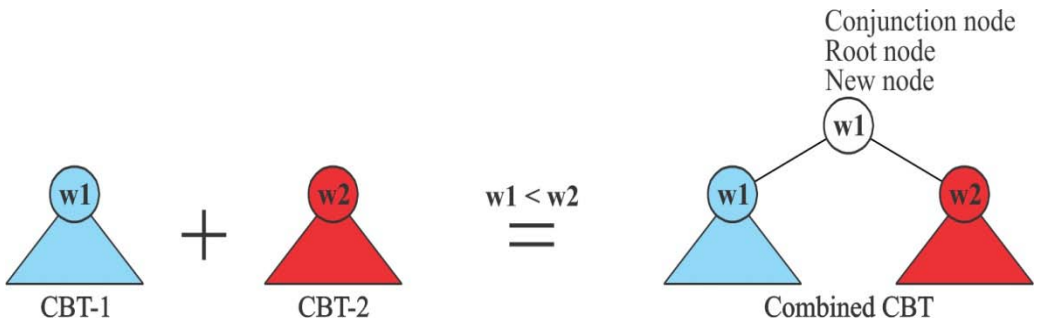

Figure 3 : The winners of two CBTs are compared and the winner (in this case the smaller) is written into the conjunction node serving as the root of the combined CBT. During this operation, three nodes are accessed and the root node should be introduced as a new node.

The comparison operation can be regarded as a procedure to compose two sub-trees. Figure-3 and Figure-4 show how a comparison between the winners of two sub-trees is implemented and how the winner is promoted in CBT and sCBT cases respectively. Note that the procedure of combining two CBTs is not possible without adding a new node, whereas in the sCBT case, there is no need for a new node.

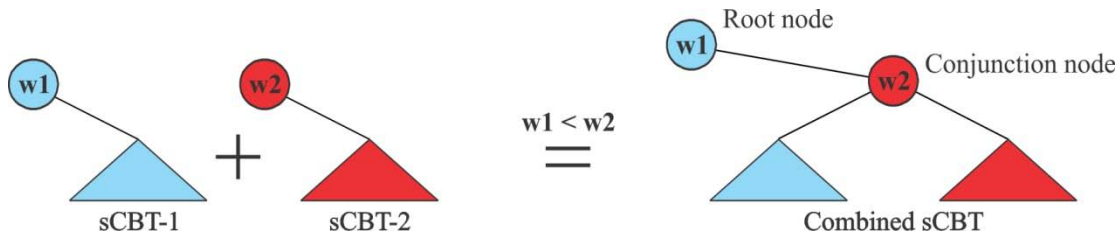

Figure 4 : When combining two sCBT swith the 'smaller wins' rule, we find the winner of the two keys hosted by the two roots then register the winner at the root of the combined sCBT, leaving the loser one in the conjunction node. During this operation, only two nodes are accessed. No extra node is required.

IV. Initial Update

An sCBT is said to be properly initialized only if every node along the winner path hosts the winner of the corresponding sub-sCBT (a node can be the root of either the left or the right block; whichever side hosts the content of the root constitutes the body of the sub-sCBT) and every sub-sCBT also exhibits this same property. Figure-5 depicts the way we can see a properly initialized sCBT. We regard the initialized sCBT as consisted of smaller sCBTs along the path of the winner key, from the winner leaf to the root. All the node contents that lose against the winner are the winners of their own sub- sCBTs.

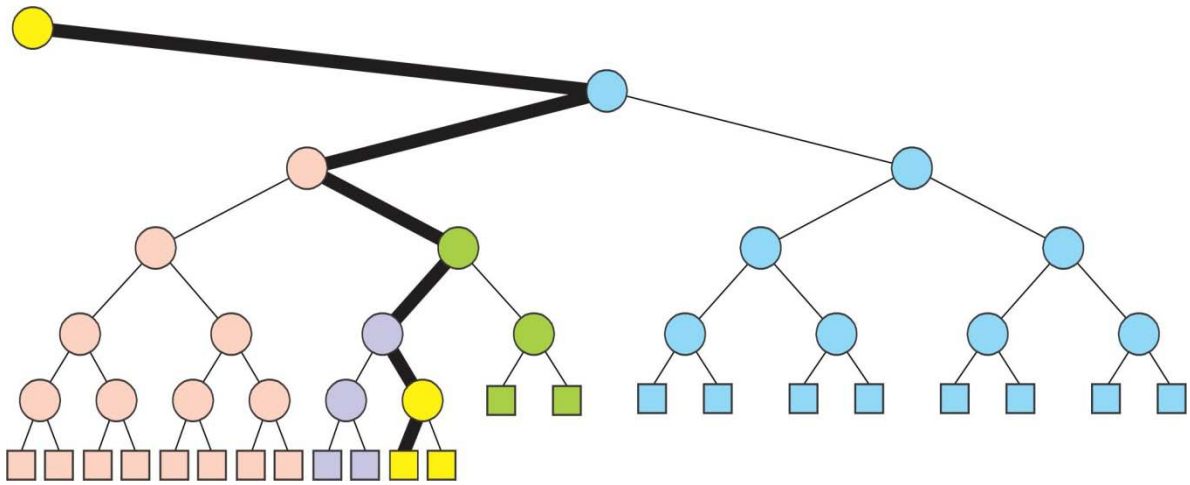

Figure 5 : An sCBT as comprised of smaller sCBTs along the winner path from the leaf node to the root. All the nodes along this path should host the winners of their own sub-sCBTs.

The new idea about initializing an sCBT is that the global tree can be thought of as a composition of already initialized smaller sCBTs. There are two different ways an initialized sCBT can be achieved:

1. Start with the maximum number (N/4) of minimal sCBTs at the lowest level of the tree; grow them independently while merging them as necessary.
2. Start with a minimal sCBT, enlarge it by adding two new leaves and update the obtained sCBT, and repeat this operation until the targeted sCBT size is reached.

In the first way, initialization starts with the non-interfering minimal sCBTs at the bottom of the sCBT and proceeds upward by growing and/or combining them until the whole tree size is reached. Following the initial update, the root (the zeroth node) contains the index of the winner element of the given key array and all the other nodes contain the indexes of the winner elements of their own sub-sCBTs. Figure-6 visualizes this method by the color coded update paths. Initializing an sCBT consisting of just two nodes requires only one comparison between the two leaves hanging from the only body node of this sCBT. After the comparison, the loser is stored in the lower node, while the winner is stored in the upper node. When all depth-1(below the root node, there is only one node) sCBTs are initialized, then the initialization of depth-2 sCBTs starts. To initialize a depth-2 sCBT, we start comparing the two new leaves that come into the picture when we grow the previously initialized depth-1 sCBT into a depth-2 sCBT. The loser of this comparison is stored into the first parent of these leaves and the winner is kept at hand to be compared to the content of the next parent node (which was the winner of the depth-1 sCBT). If it loses the comparison against the content of the next parent

node, they are swapped and the one next parent node will host the winner leaf index of the whole depth-2 sCBT (green update paths inFigure-6). Then the procedure goes on to depth-3, depth-4, and so on until the whole tree is initialized.

In this way, all sub-sCBTs with the same depth can be handled in a sub- loop, allowing any depth specific variable to be calculated faster. One such variable is the index of the root node of a givensub-sCBT, which can be found by right shifting the index of the leftmost bottom node of that sub-sCBT until the least significant bit disappears. Here is a suggested C++ code to find the root index for a given leftmost node:

```
unsigned long level;
_Bit Scan Forward(&level, leftmost Node);
root= leftmostNode>> (level+1);
```

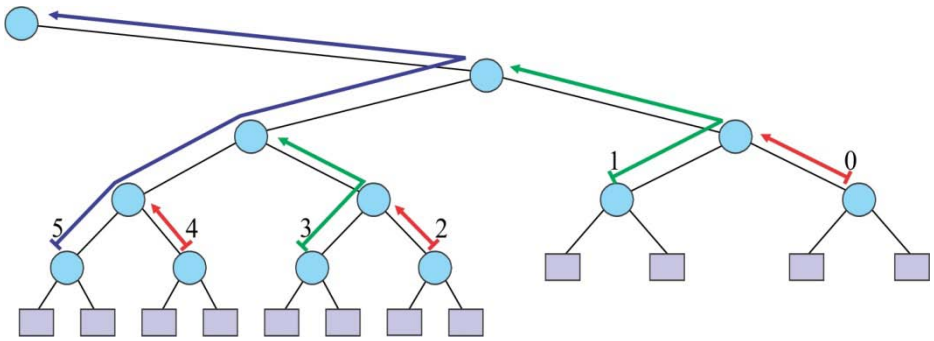

Figure 6 : A graphical depiction of two different ways to implement the initial update operation for a given sCBT. The first way is to initialize the constituent sCBTs from the smallest to the largest as indicated by the color coding in the figure, in the order of red, green, and blue. The second way is to start updating them from right to left as identified by the ascribed counting numbers from zero to five in the figure.

Figure-7 shows that the indexes of the root nodes of the same depth sCBTs form a sequential array when they are traversed from the end of the tree array towards its head (in this example the sequential array is 5; 4; 3). This gives an easy way of finding the root indexes during the initial update. The provided C++ code following the 'Redundant Tree Nodes' section uses the advantage of this first technique. As an example, Figure-7 depicts an sCBT with 12 lexical leaves. By following the sub-figures from a) to d), the initialization of this sCBT can be followed step by-step.

The second way for initial update requires the initialization of sub-sCBTs starting from rightmost depth -1 sCBT and growing/going to the left while initializing the next available size/initializable sCBT on the way. Figure-6 shows the sequence of these consecutive update paths by ordinal numbers from zero to five for the initialization of the depicted sCBT.

The advantage of the second technique is that all sub sCBTs can be processed in a single loop. Depending on the node hierarchy being used, there are some cases where this second technique becomes faster and easier to implement. However, for the simple node hierarchy used throughout this article, the implementation of the first technique proves to be more efficient.

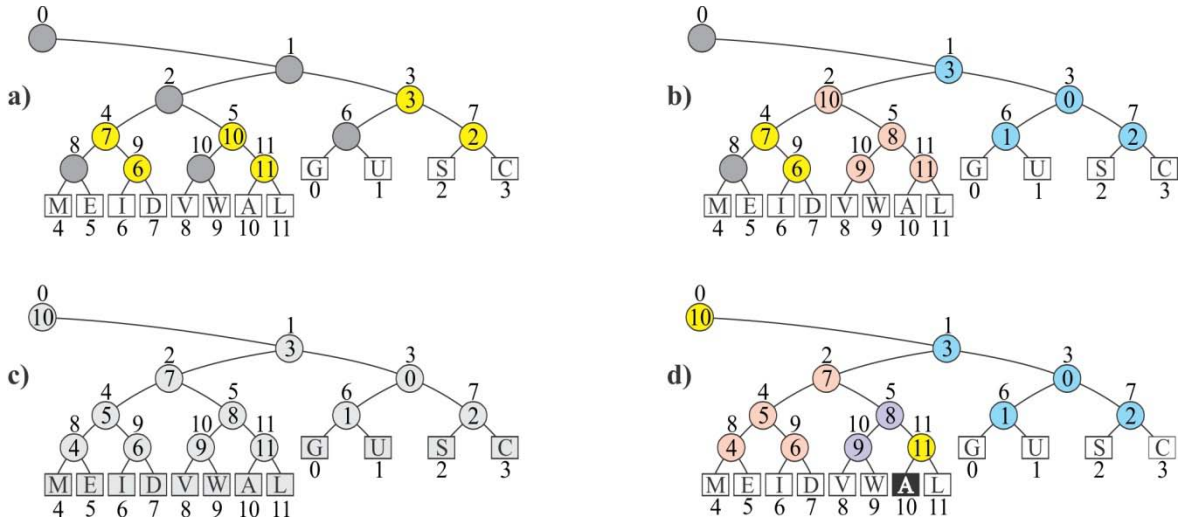

Figure 7: A lexical array of size 12 is used as the leaves of the sCBT in order to demonstrate the introduced initial update procedure using the first of the two suggested methods. a) Only the sub-trees with a depth of one are initialized, in b) the ones with a depth of two and in c) with a depth of four (which is the whole sCBT) are initialized. Here there is no sub-sCBT with a depth of three. In d) the decomposition of the sCBT along the winner path is visualized by using different colors for each sub-sCBT.

V. Redundant Tree Nodes

If a tree node is written but never read, then writing that node is considered redundant. In the case of sCBT and the proposed unilateral update mechanism combination, the bottom nodes, or in other words, the immediate parent nodes of the leaves are all redundant. This is because they host the loser keys not the winner ones. Thus, we can implement the sCBT and the proposed update mechanism by using only N/2 tree nodes. After comparing the sister leaves, we register only the winner to the grandparent node (we can think of the immediate parent node as a ghost node). Figure-8 displays a worked out example of such an sCBT using a lexical key array.

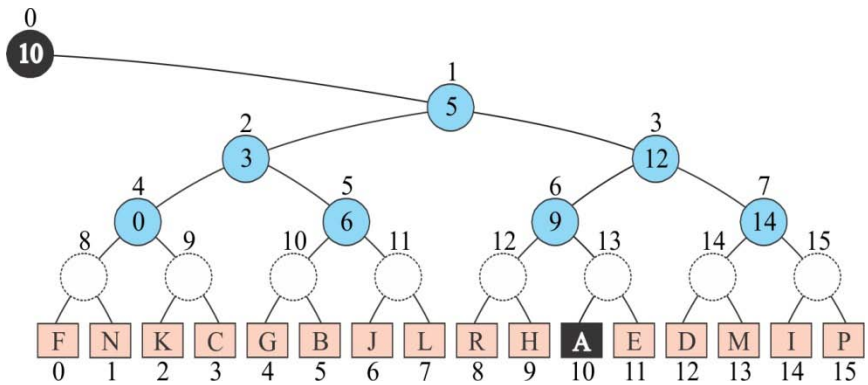

Figure 8 : Leaving out the redundant tree nodes. During the proposed unilateral update procedure, the lowest level tree nodes are not read at all, therefore there is no point of using them to write the indexes of the looser leaves. This reduces the number of required nodes to implement an sCBT to N/2.

VI. Update Mechanism

When an update is required after a new key is assigned to the winner leaf, a unilateral update procedure is implemented: First, the new key is compared to its sister, and then the winner of this comparison is kept at hand as the new winner candidate. Then this new candidate is compared to the hosted keys along the winner path. Wherever the key at hand loses the comparison, it is registered there and the previously registered key in that node is taken as the new winner candidate.This procedure goes on until the root node is reached, where the final winner is registered.

The following is a working C++ code for the proposed initial update and the proposed unilateral update methods. Initial update method follows the first technique explained in'Initial Update'section. Although

the graphical examples up to this point all use 'even number of leaf nodes', the provided code takes care of odd cases by the additional lines marked with (**). If N is guaranteed to be even, then these lines can be safely removed from the code.

```
// int N; //the size of the keys array.
//float*Keys; // the given array containing the keys.
//int offset=N,*sCBT=new int[(N+1)>>1];//"+1" is necessary for odd N cases.
//sCBT:auxiliary integer array used for the formation of stemmed complete binary tree.
// int max ID= N-1;
Void Initial Update ()
{
Int h = N-1;   //h: host, immediate parent node for a pair of leaves.
If (N&1) {sCBT[h>>1]= h; h--; offset++;} // (**)

For (int jump = 2, UpNode = h>>1, Tail= maxID>>1; ;UpNode --)
   {
        Int w= 2*h - offset; if(Keys[w^1] < Keys[w]) w ^= 1;
        For (int n= h>>1; n >UpNode; n>>= 1)
if (Keys[sCBT [n]] < Keys[w]) swap(sCBT[n],w);
        sCBT [UpNode]= w;

        h-= jump;  if(h > Tail) continue;
        h<<= 1;
        if(UpNode> 1) jump<<=1; else{ if(UpNode ==0) break; if(h < Tail) h <<= 1;}
   }
}
//w: winner, it was the index of the previous winner key, when a new value is assigned//to the winner key, the sCBT
// should be updated accordingly. This update procedure  will provide the index of the new winner key.
voidUpdate_sCBT()
{
int w= *sCBT;
if((w^1)!=N) // (w^1)==N can happen only ifN is odd. (**)
        if(Keys[w^1] <Keys[w]) w ^= 1;//loser doesn't need to be registered anywhere.

for(int node= (w + offset)>>2; node> 0 ;node >>= 1)
{
        Int const guest= sCBT [node];//guest: index of the registered key in the node.
        if (Keys[guest] <Keys[w]){sCBT [node]= w; w= guest;}
}
*sCBT= w;

}
```

VII. Results and Discussion

The benefits of the introduced unilateral update mechanism compared to the bilateral update mechanism can be itemized as follows:

1. Every key index appears in the tree at most once. More precisely, half of the key indexes will appear in the tree only once, while the other half will not have any appearance in the reduced sCBT approach. If there is a necessity for a specific application, sCBT can also be formed using N nodes, in which case, the entire key indexes will appear in the tree once and only once. In the bilateral update, some leaves are registered as many as log N times while some others are not registered at all.
2. Except for the computation (or identification) of a leaf level sister, neither is there a need for any sister node computation nor a need for accessing its content.
3. Reduction in the number of read nodes by 50%.
4. During a unilateral update, the number of writes can be between 1 and log N depending on the results of the comparisons, whereas during a bilateral update, log N writes are necessary for every update operation. The number of writes in bilateral updates can be reduced by checking the previous guest

index of a node in order to avoid re-storing the index which is already there. But this will bring extra overhead of log N integer index comparisons.

5. The required number of tree nodes is reduced by 50% in comparison to the required number of nodes for the bilateral update mechanism implemented on a CBT.

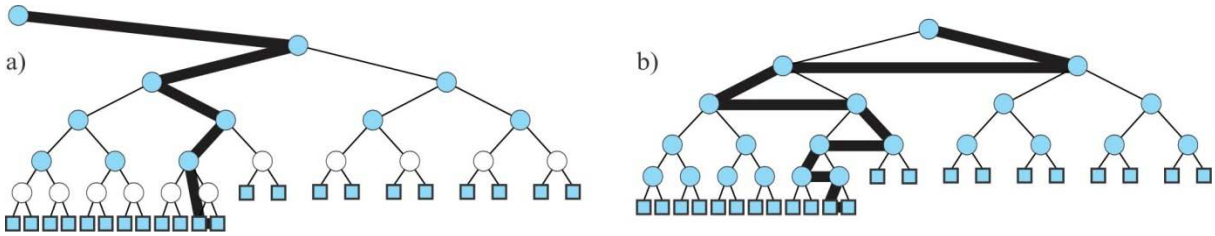

Figure 9 : Nodes visited during a) unilateral update on an sCBT, b) bilateral update on a CBT.

Table-1 shows the algebraic quantities for the two different update mechanisms in five different metrics, whileFigure-9 depicts the visited nodes and the update paths side by side for these two update mechanisms, in order to help visualize the differences.

Table 1 : Comparison between unilateral update and bilateral update for a full update operation on a complete binary tree comprising of N leaves.

Type of Update / #of	Required Tree Nodes	Comparisons	Accessed Nodes	Sister Node Computations	writes	reads
Unilateral Update	N/2	Log N	Log N	0	1≤ ≥Log N	Log N
Bilateral Update	2N	Log N	2Log N	Log N	Log N	2Log N

In terms of initial update cost, there is not much difference between the unilateral and the bilateral update methods. Both of them require exactly N comparisons. However, the number of accessed nodes, writes, and reads are different. In the case of a bilateral update on a CBT, N nodes are accessed, N reads and N writes are implemented. On the other hand, the initial update of an sCBT accesses N/2 nodes, and implements N/2 reads and a minimum of N/2 writes (in the worst case scenario, number of writes can be equal to N if all the comparisons require the swapping of node content and the winner candidate at hand). Table-2 summarizes these quantities.

Table 2 : Comparison between unilateral and bilateral initial update operations on a complete binary tree comprising of N leaves.

Type OfInitial Update / #of	Comparisons	Accessed Nodes	writes	reads
Unilateral Initial Update	N	N/2	N/2≤ ≥ N	N/2
Bilateral Initial Update	N	N	N	N

VIII. Numerical Comparisons

A test run for a given number of keys was repeated 10 times but only the averages were used for graphing. For the obtained numerical results, the maximum encountered error (standard deviation divided by average) was less than 3%. The computer used for the presented results was a Dell OptiPlex 790 with an Intel Core i5-2400 CPU @3.10 GHz and 8GB RAM. The operating system of the test computer was Windows 7 enterprise 64-bit edition. For coding, Visual C++ 2010 programming environment was used. The compilations were done with SSE2 and maximize-speed options enabled.

A uniform distribution (0.0<x<1.0) was used to generate random key values for the hold model[6]. CBTs were constructed using the given number of initial keys. Then a loop of N hold operations was performed for timing. Timing was achieved by counting the total number of CPU cycles between the beginning and the end of the computational block by using the CPU clock register. The accumulated number of CPU cycles was divided by number of given keys to get an average cost

for one hold operation. The presented empirical results have been scaled to the scores of the implementations running on the same test system based on reference CBT that Marin used [6].

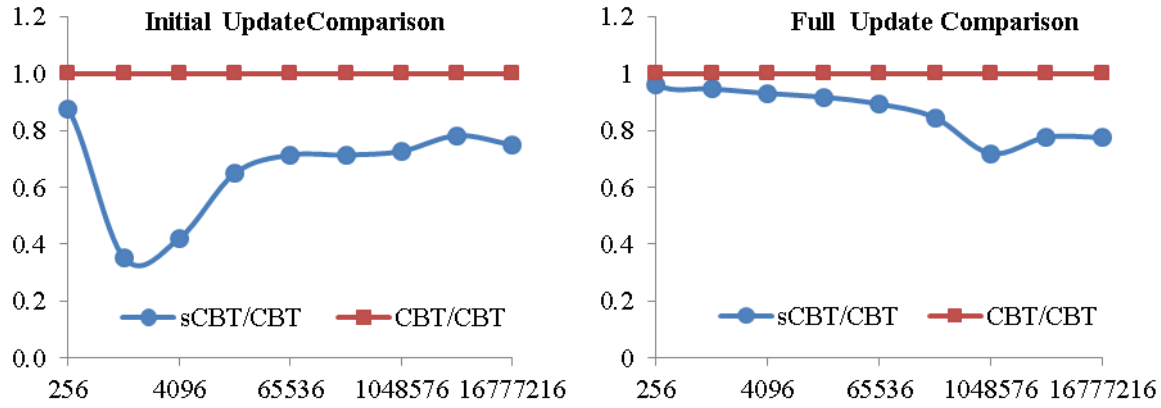

Figure 10 : Comparison of numerical performance results for the introduced unilateral update method and the reference bilateral update method. The left graph shows comparison results of unilateral and bilateral initial update methods while the right one shows the results of full update operations for the update mechanisms. The horizontal axis shows the number of keys, while the vertical axis shows the test scores scaled to the score of the reference structure (CBT) for the same test. The maximum number of keys used for the tests is 2^{24}.

[Fig. 10] presents the obtained results for the test system in two categories: Initial update comparisons and full update comparisons. In the case of initial update comparisons, introduced unilateral initial update performs at least 20% better than bilateral initial update except when the number of keys is very small. This should be because of the smaller footprint of the bilateral initial update code as can be seen in the fallowing lines compared to the code for unilateral initial update given earlier.

```
//intN; //the size of the keys array.
//float *Keys; // the given array containing the keys.

//int*CBT=new unsigned [2*N];
//CBT:auxiliary integer array used for the formation of complete binary tree.
voidInitialUpdate() //Initial Update CBT
{
for(int n=0; n < N; n++) {CBT[N+n]= n;}
for(intn=2*N-1; n > 1; n -= 2)
{if(Keys[CBT[n]] < Keys[CBT[n-1]]) CBT[n/2]= CBT[n];
else CBT[n/2]= CBT[n-1];}
}
```

Full update comparisons show that the superiority of unilateral update gets better as the number of keys increases and it stabilizes around 20%for cases the bulk of the data remains outside the cache memory.

IX. Conclusion

A new graphical formation of binary trees is introduced. As a result of this formation, binary trees can be decomposed or composed without adding or deleting any nodes regardless of their leaf and node hierarchies. This new formation leads to a unilateral bottom-up update mechanism that promises acceleration by reducing computational overhead, auxiliary memory field, and memory operations. When the suggested sCBT structure is used to produce the initial runs for external sorting [7], it will increase the average length of the runs, since larger size trees can be established in a given amount of cache memory thanks to the elimination of redundant tree nodes. The suggested unilateral update mechanism can be coupled with different leaf hierarchies such as Super CBT [8] and/ or with different node hierarchies such as hardware conscious trees[9].

References Références Referencias

1. Kirchhoff G. Ueber die auflösung der gleichungen auf welche man bei der untersuchung der linearen vertheilung galvanischer ströme geführt wird. Ann Phys 1847; 148: 497-508.
2. Sahni S. Structures, algorithms, and applications in C++. 2nd ed. Summit, NJ, USA: Silicon Press, 2005.
3. Friend EH. Sorting on electronic computer systems. J ACM 1956; 3: 134-168.
4. Marín M, Cordero P. An empirical assessment of priority queues in event-driven molecular dynamics simulation. Comput Phys Commun 1995; 92: 214-224.
5. Knuth DE. The art of computer programming, 2nd ed. San Francisco, CA, USA: Addison-Wesley, 1998.

6. Marín M. An empirical comparison of priority queue algorithms. Technical Report. Oxford University, 1997.
7. Martinez Palau X, DominguezSal D, LarribaPey JL. Twoway replacement selection. Proceedings of the VLDB Endowment; September 2010; 3: pp. 871-881.
8. Bulut M. ReducedCBT and SuperCBT, two new and improved complete binary tree structures. Turk J Elec Eng&Comp Sci 2016; 24: 2150-2162
9. Kim JC, Chhugani NS, Sedlar E, Nguyen AD, Kaldewey T, Lee VW, Brandt SA, Dubey P. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In: 2010ACM SIGMOD/PODS Conference; 2010; Indianapolis, Indiana, USA.

This page is intentionally left blank