ISSN:
Print - 2277 - 0593
Online - 2315 - 7461

Journal of Natural Science, Engineering and Technology

PERFORMANCE APPRAISAL OF TREAP AND HEAP SORT ALGORITHMS

***[1]A. D. GBADEBO, [2]A. T. AKINWALE AND [3]S. AKINLEYE**

*[1]Department of Computer Science, Michael Otedola College of Primary Education, Noforija–Epe, Lagos State, Nigeria
[2]Department of Computer Science, Federal University of Agriculture, Abeokuta, Nigeria
[3]Department of Mathematics, Federal University of Agriculture, Abeokuta, Nigeria
***Corresponding Author**:gbuyi2010@yahoo.com **Tel**.

ABSTRACT

The task of storing items to allow for fast access to an item given its key is an ubiquitous problem in many organizations. Treap as a method uses key and priority for searching in databases. When the keys are drawn from a large totally ordered set, the choice of storing the items is usually some sort of search tree. The simplest form of such tree is a binary search tree. In this tree, a set X of n items is stored at the nodes of a rooted binary tree in which some item $y \in X$ is chosen to be stored at the root of the tree. Heap as data structure is an array object that can be viewed as a nearly complete binary tree in which each node of the tree corresponds to an element of the array that stores the value in the node. Both algorithms were subjected to sorting under the same experimental environment and conditions. This was implemented by means of threads which call each of the two methods simultaneously. The server keeps records of individual search time which was the basis of the comparison. It was discovered that treap was faster than heap sort in sorting and searching for elements using systems with homogenous properties.

Keywords: Algorithm, heap, node, treap, tree

INTRODUCTION

A sorting algorithm is an algorithm that puts elements of a list in a certain order. The most – used arrangements are numerical and lexicographical orders. Most sorting algorithms work by comparing the data being sorted. In some cases, it may be desirable to sort a large chunk of data based on only a portion of that data. The piece of data actually used to determine the sorted order is called the key. Although there are many balanced trees which provide a wide variety of low-cost operations on ordered sets, only treap have the advantage of both being simple and general – in addition to insertion and deletion, they easily support efficient searching, joining and splitting.

The application of a purely binary tree is a data structure called a heap. Heaps are unusual in the menagerie of tree structures in that they represent trees as arrays rather than linked structures. The value stored at any node is at least as large as the values in its two children. The most useful property of a heap is that the largest node in the tree is always at the root. This paper overviews the basic functionalities of both treap and heap sort. Consequently, we made an overview of both algorithms and also compare time per-

formance of both sorting methods with regards to search time for items in similar databases under the same conditions.

LITERATURE REVIEW

Treaps

Treap was first described by *Aragon and Seidel* (1989) as aportmanteau of tree and heap. It is a Cartesian tree in which each key is given a (randomly chosen) numeric priority.

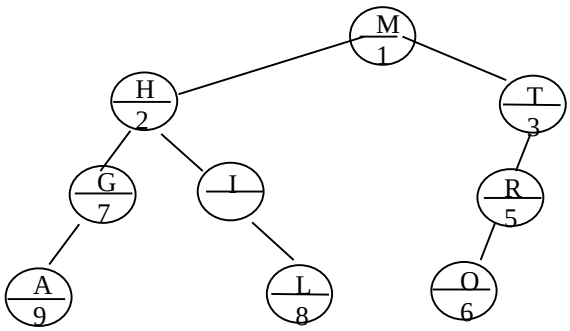

Figure 1: A sample treap showing search keys and priorities

Treap uses randomization to maintain balance in dynamically changing search trees. Let X be a set of n items, each of which has associated with it a *key* and a *priority*. The keys are drawn from some totally ordered universe and so are the priorities. The two ordered universes need not be the same. A treap for X is a rooted binary tree with node set X that is arranged in in-order with respect to the keys and in heap-order with respect to the priorities (Aragon and Seidel 1989). "In-order" means that for any node x in the tree $y.key \leq x.key$ for all y in the left subtree of x and $x.key \leq y.key$ for y in the right subtree of x. "Heap-order" means that for any node x with parent z, the relation $x.priority \leq z.priority$ holds. It is easy to see that for any set X, such a treap exist with the assumption that all the priorities and all the keys of the items in X are distinct. The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point. The left and right children of the root are binary search trees for the sets: $X_< = \{ x \in X \mid x.key < y.key \}$ and $X_> = \{ x \in X \mid y.key > x.key \}$ respectively.

Sequential algorithms

Seidel and Aragon (1996) cited in Blelloch and Reid-Miller (2011)described how to perform many operations on treap. These include the following:

Split: To split a tree rooted in r by key value a, *split* follows the in-order access path with respect to the key value a until either it reaches a node with key value a or a leaf node. When the root key is less than a, the root becomes the root of the "less than" tree. Recursively, *split* splits the right of the

root by *a* and then makes the resulting tree with keys less than *a* the new right child of the root. It also makes the resulting tree with keys greater than *a* the "greater than" tree. Similarly, if the root key is greater than *a*, *split* recursively splits the left child of the root. If the root key is equal to *a, split* returns the root; the left and right children as the "less than" and "greater than" trees respectively. The expected time to split a treap into treaps of size *n* and *m* is *O(lgn + lgm)*. Fig. 2a shows a treap while fig. 2b shows the result of a *split* on the treap shown in fig. 2a.

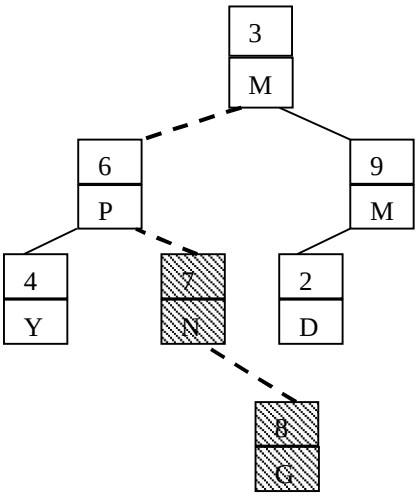

Figure 2a

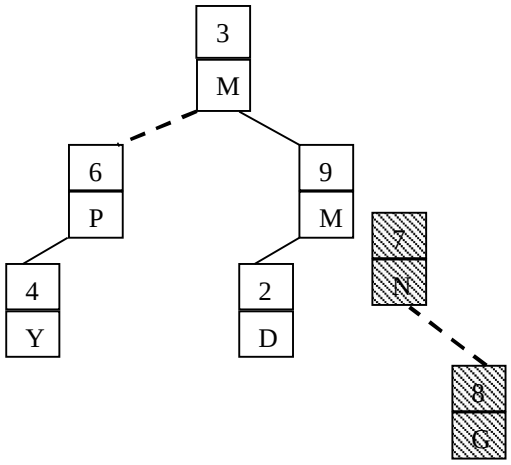

Figure 2b

(*L*, *x*, *G*) = *split* (*T*, key). Split *T* into two trees, *L* with key values less than key and *G* with key values greater than key. If *T* has a node *x* with key value equal to key, then *x* is also returned. The recursive pseudocode implementation of *split* is given below.

procedure TREAP-SPLIT (T:treap, k:key, T1, T2: treap)
TREAP-INSERT ((k, ∞), T)
[T1, T2] ← T → *l*child, *r*child]
Recursive pseudocode implementations of *split*

Join:

To join two treaps T_1 with keys less than *a* and T_2 with keys greater than *a*, *join* traverses the right spine of T_1 and the left spine of T_2. A left (right) spine is defined recursively as the root plus the left (right) spine of the left (right) subtree. To maintain the heap-order, *join* interleaves pieces of the spine so that the priorities descend all the way to a leaf. The expected time to join two treaps of size *n* and *m* is *O(lgn + lgm)*.

T = join (T_1, T_2). Join T_1 and T_2into a single tree *T*, where the largest key value in T_1 is less than the smallest key value in T_2. The recursive pseudocode implementation of *join* is given below.

procedure TREAP-JOIN (T1, T2, T :treap)
T ← NEWNODE ( )

T→ [*l*child, *r*child] ← [T1, T2]
ROOT - DELETE (T)
Recursive pseudocode implementation of *join*.

Parallel algorithms

In the parallel setting, Blelloch and Reid-Miller (2011) view each treap as an ordered set of its keys and considered the following operations.

Union: To maintain the heap-order, *union* makes *r* the root with the largest priority, the root of the result treap. If the key of *r* is *k*, then to maintain the key in-order, *union* splits the other treap by *k* into a "less than" tree with key values less than *k* and "greater than" tree with key values greater than *k* and possibly a duplicate node with a key equal to *k*. This recursively (and in parallel), it finds the union of the left child of *r* and the less-than tree and the union of the right child of *r* and the greater-than tree. The result of the two union operations becomes the left and right subtrees of *r*.

T = union (T_1, T_2). Derives the union of treaps T_1 and T_2 to form a new treap T.

Intersection: Galperin and Rivest (2003) maintained that as with *union*, *intersection* starts by splitting the treap with the smaller priority root by *k*, the key of the root with the greater priority. It then finds the intersection of the two left subtrees, which have keys less than *k* and the intersection of the two right subtrees which have keys greater than *k*. If *k* appeared in both trees, then these results become the left and right children of root used to split. Otherwise, it returns the *join* of the two recursive call results.

T = intersect (T_1, T_2). Derives the intersection of treaps T_1 and T_2 to form a new treap T.

Difference: To find the difference of two treapsT_1, T_2, Seidel and Aragon (1996) explain that *diff* splits the treap with the smaller priority root by *k*, the key of the root of the other treap. Then, it finds the difference of the two left subtrees, which have keys less than *k*, and the difference of the two right subtrees which have keys greater than *k*. Since difference is not symmetric, *diff* considers two instances viz: when T_2 is the subtrahend (the set specifying what should be removed) and when T_2 is not, as specified by the Boolean *x2.is.subtr* (Blelloch and Reid-Miller, 2011). If T_2 is the subtrahend and did not contain *k*, then it sets the left and right children of the root of T_1, T_1 to the results of the recursive calls and returns this root. Otherwise, it returns the *join* of the results of the recursive calls.

T = diff (T_1, T_2). Remove from T_1 nodes that have the same key values as node in T_2, returning its new root T.

Pseudocode of some other treap operations: creation, insertion, deletion and rotation

function EMPTY-TREAP ( ):treap
tnull→[priority, lchild, rchild] ← [- ∞ tnull, tnull]
return*(tnull)*

```
procedure TREAP-INSERT( k, p ) : item,  T :treap )
ifT  =tnull then T  ← NEWNODE ( )
T→  [key, priority, lchild, rchild ] ←  [k, p, tnull, tnull]
else if k <T→ key then TREAP-INSERT ( ( k, p ) , T→ lchild
if T→ lchild→priority> T→ prioritythen ROTATE-RIGHT (  T )
else if k >T→ key then TREAP-INSERT ( ( k, p ) , T→ rchild
if T→ rchild→priority> T→ prioritythen ROTATE-LEFT (  T )
else( * key k already in treapT *)
procedure TREAP-DELETE ( k: key, T :treap )
tnull→key ← k
REC-TREAP-DELETE ( k, T )
procedure REC-TREAP-DELETE ( k:  key,  T:treap )
if k < T→ keythen REC-TREAP-DELETE (k,T→ lchild)
else ifk > T→ keythen REC-TREAP-DELETE
( k, T→ rchild )
else ROOT - DELETE ( T )
procedure ROOT-DELETE ( T: treap )
if IS-LEAF-OR-NULL ( T ) then T  ←  tnull
else if T → lchild→ priority> T→ rchild→ prioritythen ROTATE-RIGHT ( T )
ROTATE-DELETE ( T→ rchild )
elseROTATE-LEFT ( T )
ROOT-DELETE ( T→ lchild )
procedureROTATE-LEFT ( T:treap )
[ T, T→ rchild, T →rchild→ lchild] ← [ T→ rchild, T→ rchild→lchild,  T ]
procedureROTATE-RIGHT ( T:treap )
[ T, T→ lchild, T →lchild→ rchild] ← [ T→ lchild, T→ lchild→rchild,  T ]
function IS-LEAF-OR-NULL ( T:treap ) : Boolean
return( T→ lchild= T→ rchild )
```

Source: Seidel and Aragon (2001)

Treap Operations of creation, insertion, deletion and rotation. Call-by-reference semantics was used. The global variable *tnull* points to a sentinel node whose existence is assumed.

[…]← […] denotes parallel assignment.

Heap sort

The (binary) heap data structure is an array object that can be viewed as a nearly complete binary tree as shown in Fig. 3. Each node of the tree corresponds to an element of the array that stores the value in the node. The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point. *Melhornand Naher* (2002) explains that an array A that represents a heap is an object with two attributes: *length* [A], which is the number of elements in the array, and *heap-size*[A], the number of elements in the heap stored within array A. That is, although A[1 . . *length*[A]] may contain valid numbers, no element past A[*heap-size*[A]], where *heap-size*[A] $\leq$ *length*[A], is an element of the heap.

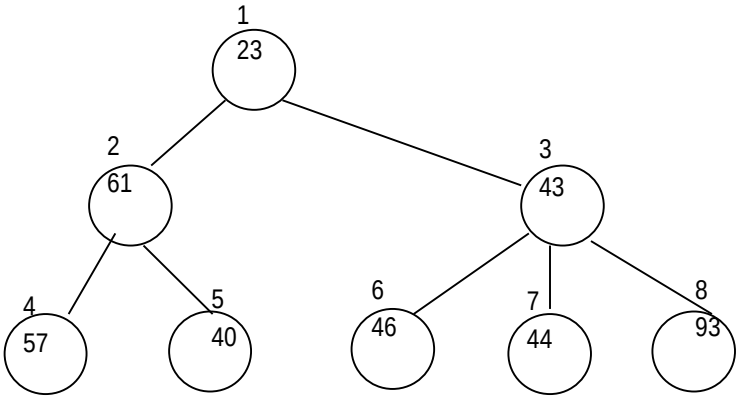

Figure 3a:A max-heap viewed as a binary tree

In fig. 3a, the number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array.

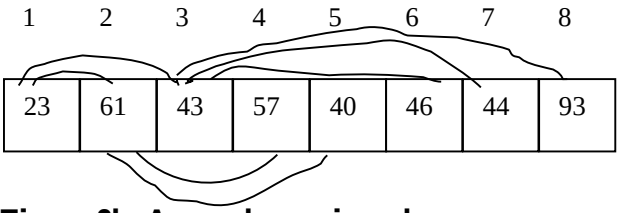

Figure 3b: A max-heap viewed as an array

MAX-HEAPIFY*(A, i)*
1 *l* ← LEFT*(i)*
2 *r* ← RIGHT*(i)*
3 **if** *l* ≤ *heap-size*[*A*] and *A*[*l*] >*A*[*i*]
4 **then** *largest* ←*l*
5 **else** *largest* ←*i*
6 **if** *r* ≤ *heap-size*[*A*] and *A*[*r*] > *A* [*largest*]
7 **then** *largest* ←*r*
8 **if** *largest* _= *i*
9 **then** exchange *A*[*i*] ↔ *A*[*largest*]
10 MAX-HEAPIFY*(A, largest)*

Action of MAX-HEAPIFY

At each step, the largest of the elements *A*[*i*], *A*[LEFT*(i)*], and *A*[RIGHT*(i)*] is determined, and its index is stored in *largest*. If *A* [*i*] is largest, then the subtree rooted at node *i*is a max-heap and the procedure terminates. Otherwise, one of the two children has the largest element, and *A*[*i*] is swapped with *A* [*largest*], which causes node *i* and its children to satisfy the max-heap property. The node indexed by *largest*, however, now has the original value *A*[*i*], and thus the subtree rooted at *largest* may violate the *maxheap* property. Consequently, MAX-HEAPIFY must be called recursively on that subtree. The running time of MAX-HEAPIFY on a subtree

of size n rooted at given node i is the $\Theta(1)$ time to fix up the relationships among the elements $A[i]$, A[LEFT(i)], and A[RIGHT (i)], plus the time to run MAX-HEAPIFY on a subtree rooted at one of the children of node i. The children's subtrees each have size at most $2n/3$. The worst case occurs when the last row of the tree is exactly half full and the running time of MAX-HEAPIFY can therefore be described by the recurrence.

Building a heap

The procedure MAX-HEAPIFY could be used in a bottom-up manner to convert an array $A[1 . . n]$, where $n = length[A]$, into a max-heap. The procedure BUILD-MAX-HEAP goes through the remaining nodes of the tree and runs MAX-HEAPIFY on each one.

BUILD-MAX-HEAP(A)
1 $heap\text{-}size[A] \leftarrow length[A]$
2 **for** $i \leftarrow \lfloor length[A]/2 \rfloor$ **downto** 1
3 **do** MAX-HEAPIFY(A, i)

Procedure in building a heap

The time required by MAX-HEAPIFY when called on a node of height h is $O(h)$.

The heapsort algorithm

The heapsort algorithm starts by using BUILD-MAX-HEAP to build a max-heap on the input array $A[1 . . n]$, where $n = length[A]$. Since the maximum element of the array is stored at the root $A[1]$, it can be put into its correct final position by exchanging it with $A[n]$. If node n is discarded from the heap (by decrementing $heap\text{-}size[A]$), $A[1 . . (n - 1)]$ can easily be made into a max-heap *(Cormen, et al 2002)*. The children of the root remain max-heaps, but the new root element may violate the max-heap property. All that is needed to restore the max-heap property, however, is one call to MAX-HEAPIFY$(A, 1)$, which leaves a max-heap in $A[1 . . (n - 1)]$. The heap sort algorithm then repeats this process for the *maxheap* of size $n - 1$ down to a heap of size 2.

HEAPSORT(A)
1 BUILD-MAX-HEAP(A)
2 **for** $i \leftarrow length[A]$ **downto** 2
3 **do** exchange $A[1] \leftrightarrow A[i]$
4 $heap\text{-}size[A] \leftarrow heap\text{-}size[A] - 1$
5 MAX-HEAPIFY$(A, 1)$

The heap sort algorithm

Source: Mikkel(2008)

The HEAPSORT procedure takes time $O(n \lg n)$, since the call to BUILD-MAXHEAP takes time $O(n)$ and each of the $n - 1$ calls to MAX-HEAPIFY takes time $O(\lg n)$.

MATERIALS AND METHODS

Experimentation environment

The researchers created a database containing 500 files. Each of these files has arbitrary numerical values. This was stored on a desktop computer with the properties indicated in table 1. It was where data values are being searched for using both treap and heap sort simultaneously. The performance of both methods gave rise to their comparison.

Table 1: Structure of the experimentation environment

Attributes	Design
Processor	Dual core 64 bits processor, 2.40GHz
Memory	4GB
Windows	8.1 Pro with Media Centre

Implementation

Both algorithms were coded in java in order to be used for searching and sorting data stored in a given database which was used for both methods. The data set used in the experiment was randomly generated. Each file was randomly loaded with different sizes of bytes of data. The implementation was client-server based. The clients start the program which generates two threads (i.e. two processes). Each thread calls each method: treap and heap sort. Arbitrary values were searched for using both methods simultaneously under the same experimental conditions. Time taken to search for and discover or otherwise the given values were monitored and stored by the server. The experiments were repeated ten times using both methods simultaneously and an average of sort times was taken.

RESULTS

The graphical representation of the result of implementation of treap and heap sort algorithms for searching for existing elements in the database is as shown in fig. 4.

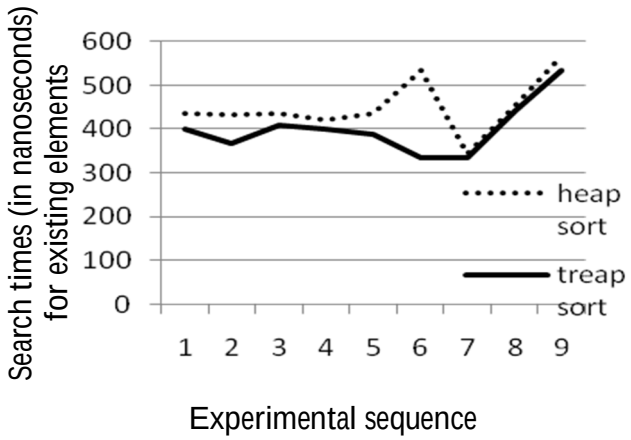

Figure 4: Search times for existing elements in the database using treap and heap sort.

The graphical representation of the result of implementation of treap and heap sort algorithms for searching for non - existing elements in the database is as shown in fig. 5.

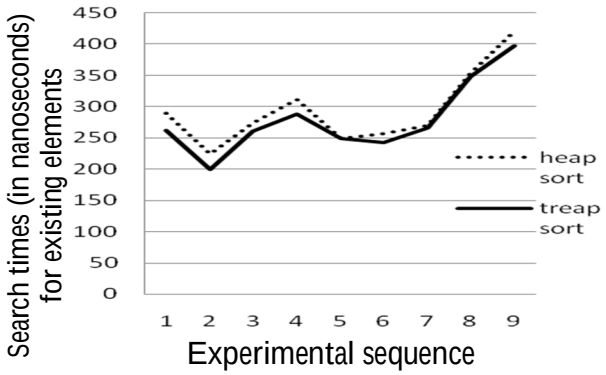

Figure 5: Search times for non-existing elements in the database using treap and heap sort.

DISCUSSION

The comparison of search times for existing elements using treap and heap sort is presented in fig. 4. Since time is an important factor in many contemporary experiments, treap took shorter time to search through the database, discover and report the existence of an element in the database. It is obvious from this figure that treap utilized lesser time to search for items in the database compared to heap sort. For instance, in fig 4, while it took treap about 400 and 520 nanoseconds to search for the first and tenth items, the same task took heap sort about 435 and 560 nanoseconds respectively. The expected height of a treap is O(lgn). Consequently the expected time to search for a value in the treap is O(lgn). The means that a treap on *n* nodes is equivalent to a randomly built binary search tree on *n* nodes since assigning priorities to nodes as they are inserted into the treap is the same as inserting the *n* nodes in the increasing order defined by their priorities. Assigning the priorities randomly results in getting a random order of *n* priorities, which is the same as a random permutation of the *n* inputs. This can be viewed as inserting the *n* items in random order. The time to search for an item is O(h) where *h* is the height of the tree. The comparison of search times for non-existing elements using heap sort and treap is presented in fig. 5. It is obvious from this figure that treap utilized lesser time to search through the database for an element. In essence, treap took shorter time in searching and reporting the existence or otherwise of searched elements than heap sort. The relative shorter time taken by treap in searching and reporting the existence or otherwise of searched elements than heap sort could be connected with the use of priority when searching through the database. This, no doubt, improved the performance of treap in many applications relating to searching through databases. The high speed involved in searching and sorting in database may also be attributable to the fact that treaps are self-balancing search trees with randomized data structures.

CONCLUSION

The comparison of both methods indicates that treap took shorter time for sorting compared to heap sort. However, while heap sort only uses a key to search and consequently sort given data, treap uses a key and a priority to sort and search for given data. This enhances the efficiency of treap and thus gave it a comparative advantage over heap sort. Treap has the advantage of being faster in terms of searching for given elements from among elements in an array compared to heap sort. In essence therefore, the use of treap can be very instrumental considering the vast expansion in technological requirements of today's business world.

REFERENCES

Aragon, C.R. and Seidel, R. 1989. Randomized Search Trees. *Proceedings of the 30th Symposium on Foundations of Computer Science (FOCS) Washington, D.C.*: IEEE Computer Society Press, pp. 540–545.

Blelloch, G.E. and Reid-Miller, M. 2011. Fast Set Operations Using Treaps. 10th Annual ACM Symposium: *Proceedings of the 10th ACM Symposium on Parallel Algorithms and Architectures*. New York, USA: ACM, pp. 16–26.

Cormen, H.T., Leiserson, C.E., Rivest, R.L. and Stein, C. 2002. *Introduction to Algorithms 2nd Edition.* The Massachusetts Institute of Technology. pp. 150-157.

Galperin, I. and Rivest, R.L. 2003. Scapegoat Trees. *Proceedings of the 4th ACM-SIAM*

Symposium on Discrete Algorithms, 165-174.

Melhorn, K. and Naher, S. 2002. Algorithm Design and Software Libraries: Recent Development in the LEDA Project. *Algorithms, software, Architectures, Information Processing.* Vol. 9.Elsevier Science Publishers.

Mikkel, T. 2008. Faster deterministic sorting and priority queues in linear space. *Proceedings of the 9th ACM-SIAM Symposium on Discrete Algorithms.* pp. 550–555.

Seidel, R. and Aragon, C.R. 2001. Randomized Strategies for Maintaining Balance in Binary Search Trees. *Algorithmica,* 23: 218-223.

Seidel, R. and Aragon, C.R. 1996. Randomized Search Trees. *Algorithmica,* 16(4 &5): 464–497.

(*Manuscript received: 24th June, 2018; accepted: 24th September, 2019).*