UPGRADE
The European Journal for the Informatics Professional
http://www.upgrade-cepis.org
Vol. V, No. 5, October 2004

UPGRADE is the European Journal for the Informatics Professional, published bimonthly at <http://www.upgrade-cepis.org/>

UPGRADE is the anchor point for **UP**ENET (**UP**GRADE European NETwork), the network of CEPIS member societies' publications, that currently includes the following ones:
- Mondo Digitale, digital journal from the Italian CEPIS society AICA
- Novática, journal from the Spanish CEPIS society ATI
- Pliroforiki, journal from the Cyprus CEPIS society CCS
- Pro Dialog, journal from the Polish CEPIS society PTI-PIPS

Publisher
UPGRADE is published on behalf of CEPIS (Council of European Professional Informatics Societies, <http://www.cepis.org/>) by Novática <http://www.ati.es/novatica/>, journal of the Spanish CEPIS society ATI (Asociación de Técnicos de Informática <http://www.ati.es/>).
UPGRADE is also published in Spanish (full issue printed, some articles online) by Novática, and in Italian (abstracts and some articles online) by the Italian CEPIS society ALSI <http://www.alsi.it> and the Italian IT portal Tecnoteca <http://www.tecnoteca.it/>.
UPGRADE was created in October 2000 by CEPIS and was first published by Novática and INFORMATIK/INFORMATIQUE, bimonthly journal of SVI/FSI (Swiss Federation of Professional Informatics Societies, <http://www.svifsi.ch/>).

ISSN 1684-5285

Next issue (December 2004):
"Cryptography"
(The full schedule of **UP**GRADE is available at our website)

* This monograph will be also published in Spanish (full issue printed; summary, abstracts and some articles online) by **Novática**, journal of the Spanish CEPIS society ATI (*Asociación de Técnicos de Informática*) at <http://www.ati.es/novatica/>, and in Italian (online edition only, containing summary abstracts and some articles) by the Italian CEPIS society ALSI (*Associazione nazionale Laureati in Scienze dell'informazione e Informatica*) and the Italian IT portal Tecnoteca at <http://www.tecnoteca.it>.

Data Architecture

A Disquisition on The Performance Behaviour of Binary Search Tree Data Structures

Dominique A. Heger

From a performance perspective, applications and operating systems are faced with the challenge to store data items in structures that allow processing fundamental operations such as insert, search, or remove constructs as efficiently as possible. Over the years, a variety of structures have been proposed, focusing on the efficient representation of data items. Some of the structures include direct addressing schemes such as hash tables, while others incorporate comparison schemes such as binary search trees. This study briefly elaborates on the internal characteristics of 5 tree-based data structures and focuses on their performance behaviour under various workload conditions. The conducted empirical studies revolve around expected run-time performance, as well as key-comparison and rotational behaviour. The goal was to identify the most efficient data structure under different workload scenarios. The 5 data structures chosen for this study represent 2 balanced (AA and red-black) and 3 unbalanced (treap, skip list, and radix) binary search tree implementations, respectively.

Keywords: Data Structures, Performance, Binary Tree

1 Introduction

Binary search trees are the most basic (non-linear) data structures utilized in the realm of application and operating system development. Their wide range of applicability can be explained by their fundamentally hierarchical nature, a property induced by their recursive definition. A binary tree structure can be defined as a finite set of nodes that are either empty or consist of a root and the elements of two disjoint binary trees, referred to as the left and right subtrees of the root. Binary tree structures support 2 primary application categories. First, they may represent hierarchical structures and second, they may be utilized to implement efficient data storage and retrieval mechanisms. In a generic setup, the individual tree components consist of 3 fields. First, a data field that holds the key. Second, a pointer to the root-node of the left subtree and third, a pointer to the root-node of the right subtree. In such a tree representation, NULL-pointers indicate empty subtrees, and the argument can be made that this representation is not space-efficient, as most of the pointers are referencing NULL. An alternative to such a design is known as a threaded binary tree structure, a tree construct that utilizes the space more effectively [12]. Instead of pointing to NULL, the leaf node pointers are linked to expedite lookup and tree traversal operations. In general, the challenge faced is to differentiate among the high-level tree features and operations, as well as the representation model, in an effective way that does not break the algorithms. Another venerable issue is that the tree balancing mechanisms (the maintenance operations per se) are from a performance perspective rather expensive, as well as complex to implement. The goal of this study was threefold. First, to quantify the performance behaviour of the red-black, the AA, the treap, the skip-list, and the radix tree data structures under varying workload conditions [5][17][20][21]. The focus was on implementation complexity, expected time complexity, key comparison, as well as on the restructuring operations (in the case of the 2 balanced binary search tree implementations). Second, to analyse the impact that some rather simple code-changes in the treap implementation have on the key comparison behaviour. Third, to quantify the performance delta of the tree traversal operation in a threaded and a non-threaded red-black tree environment. Of the discussed data structures, the AA and the red-black tree represent balanced structures, where all the individual operations (insert, remove, search) are bounded by an asymptotic upper bound of $O(\log n)$. In the case of the treap, the radix, and the skip-list implementation, the underlying unbalanced binary search tree structures result in performing the individual operations in an expected time complexity of $O(\log n)$ as well. With theses 3 data structures though, an ergodicity of $O(n)$ exists. Some other tree constructs such as AVL trees [5] or hash-based solutions were not incorporated into this study. The reader is referred to [27] for a comprehensive discussion on dynamic hashing.

2 Red-Black Trees

Binary search trees perform best when they are either balanced, or the path length from the root to any leaf node is within some bounds. The red-black tree algorithm represents a method for *balancing* trees [5]. Red-black trees are a variation of the classic binary

Dominique Heger has been with IBM for over 9 years. Prior to his work at IBM, he spent 5 years with Hewlett-Packard. His focus is on operating systems performance, performance modelling, algorithms and data structures, and I/O scalability. He has been part of several research projects that focused on UNIX scalability, and has published many systems performance and modelling related papers. He holds a PhD from NSU (Nova Southeastern University), Florida, USA, in Information Systems.
<dheger@us.ibm.com>

search trees (BST) that utilize a rather efficient mechanism for keeping the tree in balance. The name derives from the fact that each node is coloured *red* or *black*, and that the colour of the node is instrumental in determining the balance of the tree. During insert and delete operations, nodes may be rotated to maintain the tree balance. In general, both average and worst-case search time complexity equals to *O(log n)*. More specifically, the red-black tree design incorporates the following properties:

1 Every node is coloured red or black
2. The root node has to be black
3. Every leaf is a NIL node, and is coloured black
4. If a node is red, then both its children are black
5. Every simple path from a node to a descendant leaf contains the same number of black nodes

The number of black nodes on a path from the root to a leaf is known as the *black-height* of a tree. The properties mentioned above guarantee that any path from the root to a leaf is no more than twice as long as any other. All operations on the tree must maintain the properties listed above. In particular, operations that insert or delete items must abide within these very specific rules [5]. The amount of memory required to store a red-black node should be kept to a minimum. This is especially true if many small nodes are being allocated. In most cases, each red-black tree node has a left, a right, and parent pointer. In addition, the colour of each node has to be recorded. Although this requires only one bit, more space may be allocated to ensure that the size of the structure is properly aligned. To reiterate, on a static red-black tree implementation, the operations *minimum*, *maximum*, *search*, *successor*, *predecessor* can be executed in *O(log n)* time. Tree maintenance operations such as insert or delete require dynamic changes to the tree structure, and therefore require rather sophisticated implementations to meet the *O(log n)* time criteria. It has to be pointed out that a simple rotation is being executed in *O(1)* time. A threaded (red-black) search tree represents a data structure where the un-utilized child pointers are used to point to either the successor (right child pointer) or the predecessor (left child pointer) nodes, respectively. From an implementation perspective, the pointers have to be *flagged* to disclose if they represent a normal or a threading scenario [5]. One of the benefits of threading a tree structure is that it is feasible to process an in-order traversal in constant space, as it is not necessary to remember the entire path from the root to the current position. Therefore, a threaded tree structure represents a stack free solution that is beneficial if lookup (find) and tree traversal operations dominate the workload.

3 AA-Tree Data Structure

Andersson [1] introduced the AA-Tree design in 1993, as basically a quest to present new maintenance algorithms for balanced tree structures. Additional work by Weiss (1996) resulted into a much broader dissemination of the AA-Tree design [21]. The AA-Tree is considered as a simpler to code variant of the red-black tree and satisfies the following properties:

1. Every node is coloured red or black
2. The root node has to be black
3. Every leaf is a NIL node, and is coloured black
4. If a node is red, then both its children are black
5. Every simple path from a node to a descendant leaf contains the same number of black nodes
6. Left children may not be red.

The advantages of an AA-Tree design (compared to red-black trees) are that half the restructuring cases are eliminated, and that the delete operation is substantially simplified. In other words, if an internal node has only one child, that child has to be a red right child. Further, it is always possible to replace a node with the smallest child in the right subtree, as it either will represent a leaf node or it will have a red child. In the AA design, the balancing information is stored in each node as *the level*. The actual level is defined by the rules that (1) if a node is a leaf, its level is set to 1. (2) If a node is red, its level equals to the level of its parent. (3) If a node is black, its level equals to 1 less than the level of its parents. The level represents the number of left links to a NULL (or NIL) reference. The AA design further introduces the term horizontal link, in the sense that a *horizontal link* represents a connection between a node and a child with equal levels. In other words, horizontal links can be referred to as right references.

Based on the AA design, (1) it is not possible to have 2 consecutive horizontal links in the tree. (2) Nodes at level 2 or higher have to have 2 children, and (3) that if a node has 0 right horizontal links, its 2 children have to be at the same level. Compared to the red-black tree implementation, the vast number of rebalancing cases is simplified in the AA design by utilizing two rather simple maintenance operations labelled as *skew* and *split*. The skew operation removes left horizontal links, whereas the split operation addresses the issue of removing consecutive horizontal links (that are by design not allowed). Both operations are part of the AA insert and delete maintenance set. All the unbalanced situations that are imaginable in an AA-Tree based scenario can be eliminated by a sequence of at most 3 skew and 2 split operations, respectively. This statement holds true based on the fact that the maintenance work may affect a higher level, and therefore has to be propagated upward in a recursive manner. The fact that the left children may not be red greatly simplifies the delete operation (compared to the red-black paradigm), and therefore an AA-Tree solution should be considered if delete operations represent a significant portion of the actual workload profile.

4 Treap Data Structure

A *treap* is the basic data structure underlying randomized search trees. The name itself refers to synthesizing a tree and a heap structure [5], [19]. More specifically, assuming that *x* represents a set of items where each item is associated with a *key* and a *priority*. A *treap* for a set *x* represents a special case of a binary search tree, in which the node set is

Operation	Time Complexity
Find/Search/Access	O(log n)
Insert	O(log n)
Delete	O(log n)
Rotations per update	2
Update – rot. subtree s = O(s)	O(log n)
Update – rot. subtree s = O(s log^k s)	O(log^k+1*n)
Joining 2 trees (sizes m & n)	O(log max{*m,n*})
Splitting a tree (into size m & n)	O(log max{*m,n*})

Table 1: Treap Performance Characteristics – Generic Operations.

Operation	Time Complexity
Insert with handle	O(l)
Delete with handle	O(1)
Finger search over distance d	O(log d)

Note: handle, finger, split, and join operations require additional pointers.

Table 2: Treap Performance – Advanced Operations.

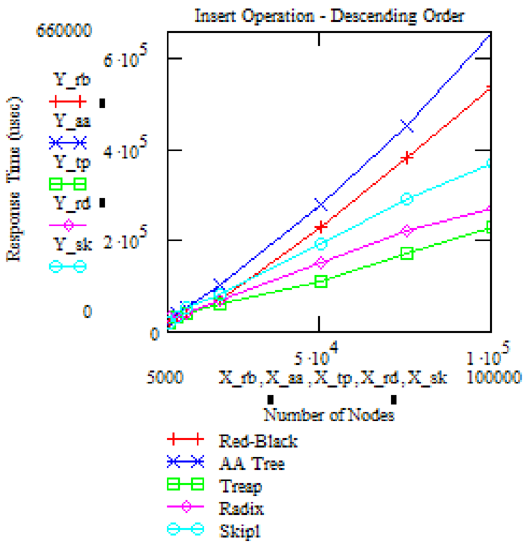


Figure 1: Insert Operations.

arranged *in order* (in respect to the keys) as well as in *heap fashion* (in regards to the priority). Further, assuming that *t* represents the treap structure storing a set of items *x*. Given the scenario where the key of an item is known, the location in *t* can easily be determined via a simple search tree algorithm. In a treap, the access or search time is proportional to the depth of an element in the tree. An insert of a new item *z* into *t* basically consists of a 2-step process. The first step consists of utilizing the item's key to attach to *t* at the appropriate leaf position and second, to use the priority of *z* to rotate the new entry up in the structure until the item locates the parent node that has a larger priority. The process of deleting an item *z* from a treap structure *t* represents the reversed scenario. The first step consists of locating the item, and second to rotate the item down in the tree structure until it becomes a leaf, where the item can be removed.

In some implementations, treap *split* and *join* operations may be necessary. A split operation is used to separate a set of items *x* into 2 sets (*x1* and *x2*). The separation utilizes a heuristic where items are being placed in the 2 sets based on the item's key values in comparison to the key of a reference element *a*. To accomplish the split, the operation inserts an item with key *a* that is affiliated with an infinite priority. Based on the heap-order property, the new item has to represent the root of the heap. Based on the *in-order* property, the left subtree represents the treap *x1*, whereas the right subtree represents the treap *x2*. In a similar fashion, the join operation is utilized to combine the two sets *x1* and *x2* into a single construct. The assumption made is that the keys in *x1* are smaller than the keys in *x2*. The implementation of the join operation creates a dummy root item, where the left subtree consists of *x1* and the right subtree represents *x2*. In a second step, the join operation performs a delete on the dummy root item, finalizing the combined treap structure. In some circumstances, *handles* or *fingers* are being used to expedite some of the maintenance operations. To illustrate, in the case that a handle is referring to a specific node *x*, deleting the node *x* can be accomplished by only rotating it down into a leaf position and freeing the item, circumventing the otherwise necessary search operation. In a similar fashion, to insert a new item *x* where a handle to either the successor or the predecessor *y* of node *x* is available, the search for the location for *x* can start at the reference point *y* (instead of at the root item). The term *finger search* for a node y in a treap refers to following the unique path between *x* and *y*, where node *x* incorporates a handle that points to it. Another aspect of treap implementations is that split and joint operations can be processed more efficiently if handles are available to the min and max key items, respectively. A randomized search tree that stores *n* items reveals the expected asymptotic upper bound time complexity (see Table 1 and Table 2).

The time complexity for a successful *search* operation in a treap environment is proportional to the number of ancestors of *x*, and can be expressed as *O(log n)*. An unsuccessful search for a key that falls between the keys of successive items (x^{-} & x^{+}) takes on an expected time complexity of *O(log n)* as well [14]. In order to *insert* an item into a treap, the first step is to locate its leaf position (based on its key value), and in second step to rotate the item up in the tree structure based on its priority. The number of rotations can at most be equal to the search path, hence the time to insert an item is proportional to the time required to complete an unsuccessful search, which as already discussed (in expectation) equals to *O(log n)*. In the case of a *delete* operation, the insert operation is being inverted, therefore the conclusion that the time complexity equals to *O(log n)*. The number of downward rotations during a delete operation equals to the sum of the length of the right spine of the left subtree of *x*, and the left spine of the right subtree of *x*, respectively. A scenario that (in expectation) is < 2 for a randomized binary search tree.

5 Skip List Data Structure

A skip list represents an ordered linked list, in which every node contains a variable number of links to other nodes in the structure [13][16]. To illustrate, the n^{th} link of a given node points to subsequent nodes in the list, and by design, skips over some number of intermediary nodes. Therefore, these skipped nodes have fewer than n links associated. As most nodes have a variable number of links, a skip list can be referred to as a collection of linked lists of different levels. In order to quickly traverse the structure, seeking for some target key, the search operation seeks on the upper level list until either the target data is encountered, or the operation locates a node with a key that is smaller than the target. At this point, that particular node links to a subsequent node. In this case, the search continues by repeating the same procedure (now starting at the node that incorporates the smaller value than the target) and by continuing on the skip list. Skip lists can be considered as a probabilistic alternative to balanced trees.

Skip lists have balance properties that are similar to the search trees that are built via random insertions. Balancing a data structure probabilistically is easier than explicitly

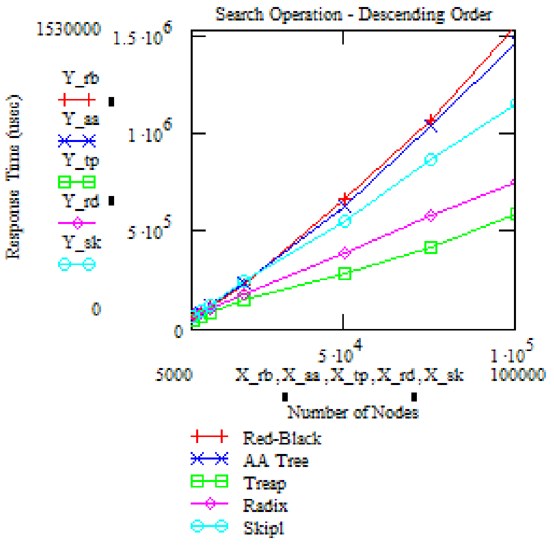


Figure 2: Search Operations.

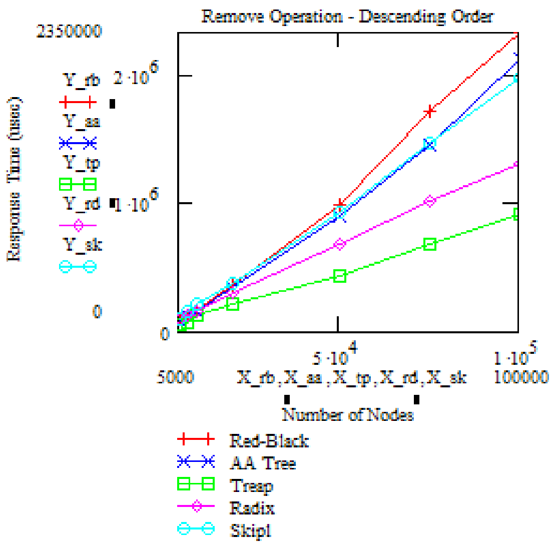


Figure 3: Remove Operations.

maintaining the balance. For many applications, skip lists represent a more natural layout than tree structures, and therefore are generally leading towards simpler algorithms. The ramification is that the simplicity of skip list algorithms allows easier implementations, and provides in some cases a (constant factor) speed improvement compared to the balanced and self-adjusting tree algorithms. Skip lists are rather space efficient. They can easily be configured to hold (on average) 1 1/3 pointers per element, and do not require balance or priority information to be stored within each node. The varying size of the nodes may be regarded as a disadvantage of skip lists. As a s skip list is balanced in a probabilistic fashion (by using a random number generator), the average search, insert, and delete operations are processed in an expected time complexity of *O(log n)*. The probability of encountering significantly worse performance is rather slim, but nevertheless exists. In other words, as the balance criteria is chosen randomly, the chance of encountering the *O(n)* worst case scenario is very small, as any input sequence into a skip list will not consistently produce the worst case performance scenario.

6 Radix Tree Data Structure

A standard radix search tree design is similar to a digital search tree [2][5][21]. However, in a radix search tree, all data items are stored as leave objects, and therefore the internal nodes of the radix tree do not have any key values associated with them. An internal node's child represents either another internal node or an actual data item. During a search operation, the *individual bits in the search key are examined*, and either the left or the right pointer to a child node is being activated according to the specific bit value. Therefore, unlike the digital search tree, the radix search tree does not have to encounter any key comparison overhead per se at each node that is being traversed. Instead, the traverse operation continues until the corresponding bit in a child node's *link filed* is zero. The child link entity refers to a two-bit field entry, where bits 0 and 1 specify the child pointers. In either case, if the bit value equals to zero, the pointer references either NULL or points to a data item. Otherwise, if the bit is 1, the pointer references another node in the radix tree. In other words, if the child link field equals to 0, either a NULL pointer or a data item has been located. Further, the root node always remains in the radix search tree, even in the case when there are no items in the tree. From a performance perspective, the number of nodes, as well as the length of the key value govern the efficiency of a radix tree. In general, large key values have a rather detrimental impact on performance.

Along these lines, the radix sort is a rather good illustration of how lists and deques can be combined with other container's [5]. In the case of a radix sort, a vector of deques is manipulated, similar to a hash table. In a radix sort, the values are successively ordered on a per *digit position* basis, normally from right to left (straight radix sort). This is accomplished by copying the values into buckets, where the index for the bucket is determined based on the position of the digit being sorted. The straight radix sort algorithm operates in *O(nk)* time, where *n* represents the number of items, and *k* refers to the average key length. The greatest disadvantage of a radix sort algorithm is that the implementation can not be constructed to execute *in place*. Therefore, *O(n)* additional memory space is required. Furthermore, a radix sort implementation requires 1 pass for each symbol of the key, and therefore is rather inefficient if long key values are processed. The reader is encouraged to consult [26] for a comprehensive discussion on radix trees, extendible hashing, B-Trees, and performance.

7 Benchmark Results

To conduct the performance comparison, all the data structures were implemented in *ANSI C*. The implementation of the data structures were based on work conducted in [5][12][16][17][21]. Where applicable, the same random number generator and the same seed were used throughout the study. All the data structures were exposed to the same workload scenarios. The analysis was decomposed in 3 sections. Section 1 focused on the individual insert, search, and remove performance. The operations were benchmarked either in an ascending, descending, or random order while scaling the number of nodes from 5,000 to 100,000. Next to the response time comparison, the study introduced the term *aggregate structure performance factor*, describing the mean performance of a data structure as quantified over the set of invocation scenarios used in this study. To illustrate, the insert performance was quantified based on ascending, descending, and random data distributions. Therefore, the overall consistency factor for the insert operation incorporates the 3 invocation scenarios. Section 1 further discusses the performance behaviour based on a mixed workload profile, consisting of a chain of insert, search, and remove operations, respectively. Section 2 quantified the data structure performance focusing on the number of key comparisons and (where applicable) the number of rotate operations. For the treap data structure, code changes surround-

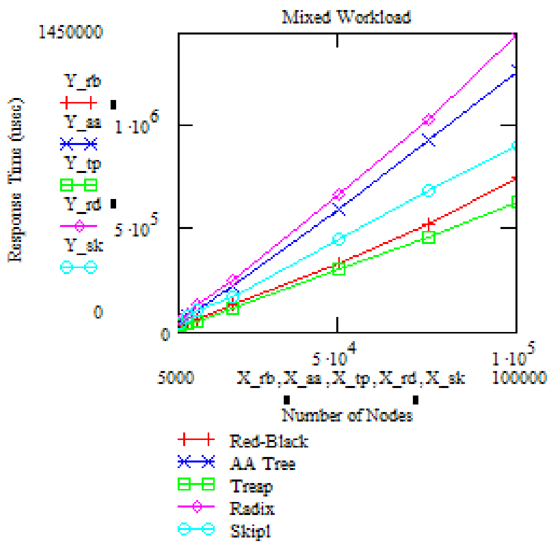


Figure 4: Mixed Workload.

ing the placement of the *equality*, the *less than*, and the *greater than* operations are proposed and analysed. Section 3 discusses the performance of the tree traversal operation, comparing a standard red-black tree and a threaded red-black tree implementation.

The test environments for the benchmarks in Section 1 and 2 consisted of a single CPU, 256MB memory, Linux 2.6 system that was equipped with a single disk configured with the XFS file system. For the benchmarks described in Section 3, a 4-way, 1GB memory system, configured with a 5-disk RAID-5 I/O subsystem that utilized the Linux 2.6 and the JFS file system was used. All the benchmarks were executed 100 times. The performance numbers reported in this study reflect the mean over all the test runs.

Nodes	Operation	Treap	Radix	Skip
10,000	Insert	98,771	299,953	247,439
	Search	169,977	320,032	251,764
	Remove	108,758	309,974	184,254
50,000	Insert	555,109	1,499,955	1,420,190
	Search	1,007,434	1,600,032	1,489,230
	Remove	605,099	1,549,975	1,512,343
100,000	Insert	1,219,770	2,999,956	2,932,223
	Search	2,070,369	3,200,032	2,969,512
	Remove	1,319,758	3,099,975	2,865,642

Table 3: Key Comparisons – Unbalanced Data Structures.

7.1 Insert, Search, and Remove Operations

The basic data structure benchmarks were conducted on the single CPU system. In the case where the nodes were inserted in descending order, the treap outperformed the other data structures by a rather significant margin (Figure 1). As discussed, the treap reflects a light-way data structure compared to either a red-black or the AA implementation, respectively. Therefor, the insert operations are completed more efficiently, as the expensive maintenance functions embedded in the balanced data structures are much more relaxed in a treap implementation. The delta between the fastest (the treap) and the slowest (the AA tree) structure equalled to 430 milliseconds (at the 100,000-node level). At the 10,000-node level, all 5 data structures reported mean response time values within 10 milliseconds. In the case the nodes were either inserted in an ascending or random order, the radix tree proved to be the most efficient solution (Appendix A).

From a *structure performance factor* perspective, at the 100,000-node level, the radix tree's insert operations outperformed the other data structures. Further analysing the fluctuation among the different insert scenarios (ascending, descending, and random) revealed that the red-black tree performed most consistently. At the 100,000-node level, the fluctuation among the ascending, the descending, and the random insert operations was approximately 40 milliseconds. This can be compared to a delta of 440 and 130 milliseconds for the treap and the radix tree, respectively. The insert benchmarks disclosed that the skip list and the AA tree experienced scalability issues, especially in the random insert scenario.

The benchmarks conduced revolving the search operations in a descending order revealed a similar picture (Figure 2). From a mean response time perspective, the treap data structure outperformed the skip list, as well as the radix tree, whereas the latter two data structures were able to outperform the more complex red-black and AA tree structures. At the 100,000-node level, the delta between the fastest (the treap) and the slowest (the red-black tree) data structure was 960 milliseconds. At the 10,000-node level, the difference between the most (the treap) and the least (AA and skip list) efficient implementations equalled to 40 milliseconds. The search benchmarks conducted in ascending order disclosed a similar behaviour as experienced for the insert operations (see Appendix A). The radix tree outperformed the treap, which outperformed the other 3 implementations. At the 100,000-node level, quantifying the aggregate *structure performance factor* showed the radix tree and the treap in a dead heat. At the same time, the other 3 data structures trailed by 530 milliseconds, 725 milliseconds, and 790 milliseconds for the skip list, the AA, and the red-black tree structures, respectively.

From a consistency perspective (smallest delta between the search scenarios), the radix tree outperformed the red-black and the AA tree, which outperformed the treap and the skip list. At the 100,000-node level, the delta between the ascending and the descending search operations was 10 milliseconds for the radix tree, 150 milliseconds for the red-black tree, and 380 milliseconds for the treap, respectively. Benchmarking the remove operations in descending order revealed that the treap was once again capable of outperforming the other 4 data structures (Figure 3). The mean delta between the treap and the slowest structure (red-black tree) at the 100,000-node level equalled to 1,430 milliseconds. At the 10,000-node level, the difference between the treap and the least efficient implementation (skip list) equalled to 90 milliseconds. Analysing the remove performance in ascending

Nodes	Operation	AA	Red-Black
10,000	Insert	168,244	211,383
	Search	114,707	128,853
	Remove	114,037	103,671
50,000	Insert	1,016,999	1,286,225
	Search	685,766	754,794
	Remove	697,229	639,197
100,000	Insert	2,183,976	2,772,389
	Search	1,471,511	1,609,564
	Remove	1,494,441	1,378,359

Table 4: Key Comparisons – Balanced Structures.

Nodes	Operation	AA	Red-Black
10,000	Insert	9,982	9,976
	Height	18	24
	Remove	3,340	2,489
50,000	Insert	49,976	49,971
	Height	21	29
	Remove	16,676	12,468
100,000	Insert	99,987	99,969
	Height	22	31
	Remove	33,339	24,985

Table 5: Height and Rotations – Balanced Structures.

order disclosed the treap as the most efficient implementation (Appendix A). As the mathematical and structural analysis of the red-black and the AA tree design suggested, the AA tree outperformed the red-black implementation in every remove scenario that was benchmarked in this study. Analysing the aggregate *structure performance factor* at the 100,000-node level showed the treap with the lowest mean response time, followed by the radix, the skip list, the AA, and the red-black tree.

From a consistency perspective (smallest delta between the remove scenarios), the radix and the red-black tree outperformed the other 3 implementations, underlying the robustness of these data structures under different operation patterns. To further quantify the performance behaviour of these data structures, the study utilized a mixed workload profile. The profile triggered a chain of insert (100% of the nodes in ascending order), search (randomly for 10% of the nodes), remove (randomly for 50% of the nodes), and search (randomly for 10% of the nodes) operations. The conduced benchmark runs showed that at every node level, the treap outperformed the other 4 data structures (Figure 4). At the 100,000-node mark, the treap outperformed the slowest data structure (radix tree) by 820 milliseconds. Decomposing the conducted test runs into a small (5,000 to 50,000 nodes) and a large (greater than 50,000 up to 100,000 nodes) category, and conducting the analysis accordingly did not change the performance picture in any significant way. In the small mixed workload category, the treap represents the most efficient implementation, whereas the radix tree encounters a rather steep increase in response time at the 20,000 and the 50,000-node levels, respectively. The same behaviour is reflected in the large mixed workload category.

Overall, the red-black tree performed well at every node level, as the data structure was capable of outperforming the more light-way implementations of the radix tree and the skip list at every measured data point.

7.2 Key Comparisons and Rotations

The next few experiments in this study focused on the number of key comparisons performed by the data structures while processing a certain workload. The results in Table 3 and 4 outline key comparisons for a descending permutation, attempting to model a realistic situation where the inserted elements are in a nearly sorted order. Evaluating the mean number of key comparisons (across the 3 operations) showed the treap as the most efficient implementation at all the benchmarked node levels. The radix tree represents the structure that processes the most key (actual bit) comparisons. Despite processing more key comparisons, the simplified AA remove function outperforms the red-black tree implementation from a response time perspective. As the design suggests, the 2 balanced tree structures disclose the lowest number of key comparisons for the search operation. The number of key comparisons processed by the 2 balanced structures (while operating on remove scenarios) are in line with the most efficient (treap) unbalanced data structure, and clearly outperform the other 2 (radix and skip list) solutions. The rebalancing operations necessary in these 2 data structures though squander that advantage, which is reflected in the response time behaviour (Figure 3). Evaluating the key comparison behaviour on random data sets revealed that the 2 balanced data structures outperformed the 3 unbalanced solutions. The analysis showed that the red-black tree slightly edged the AA implementation at all the benchmarked node levels (see Table 3 and Table 4).

In order to further investigate the key comparison behaviour, and the impact on response time, the study varied (in the treap solution) the order in which the *equality*, the *less than*, and the *greater than* operations were processed. The following pseudo code documents the 2 experiments conducted for the treap search operation.

```
Option 1:
1 f key_searched = key_current then found
2 else if key_searched < key_current go to left child
3 else go right

Option 2:
1 f key_searched = key_current then found
2 else if key_searched > key_current go to right
child
3 else go left
```

The benchmarks conduced for the treap search operation (at various node levels and random input sets) revealed that option 2 outperformed option 1 (response time wise) by approximately 4%. The study further showed that moving the comparison in line 1 further down and executing the greater than operation first, results in fewer key comparisons but a higher overall response time.

For the red-black and the AA structures, Table 5 discusses the height and the number of rotations executed at different node levels with a descending input set. Both structures revealed almost identical numbers of rotations while inserting the data items. While processing remove operations, the red-black tree executed approximately 25% less rotations than the AA implementation. In all the benchmarks utilizing ordered data sets, the AA tree presented a significantly flatter tree hierarchy than the red-black implementation. As outlined in Table 5, a height delta of 6, 8, and 9 at the 10,000, the 50,000, and the 100,000 node-levels was reported. Studies conducted on a random data set revealed that the red-black tree executed on the insert as well as the remove operations fewer rotations than the AA tree. Further, with a random sample set, the height of the tree structures only varied by 2, 3, and 3 at the 10,000, the 50,000, and the 100,000-node levels, respectively (see Table 5).

The benchmarks revealed that the compiler, the systems architecture, and the time complexity of the key comparisons significantly impacts the response time behaviour. The search operations for all the tree-based structures were essentially identical. Despite the similar search solutions, methods that executed fewer key comparison operations not always revealed the most efficient response time behaviour. Processing 50,000 search operations in ascending order resulted in 1,600,032 and 1,067,079 key comparisons for

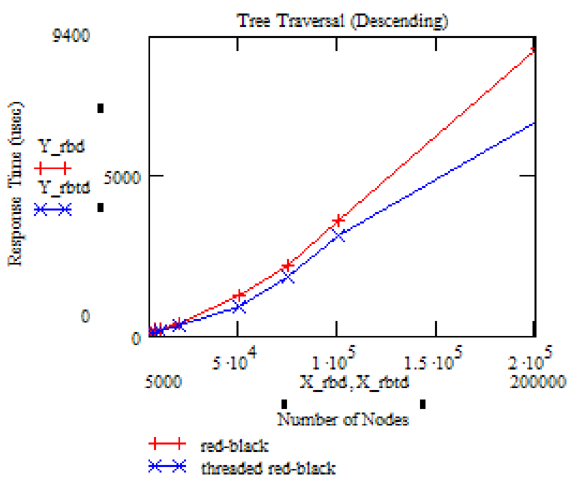


Figure 5: Tree Traversal.

the radix tree and the treap data structures, respectively. From a response time perspective, the radix tree outperformed the treap for this data point by 110 milliseconds. Similar tests conduced on an larger SMP system (running a commercial UNIX flavour) revealed that based on the different compiler architecture, instruction pipelining features, and cache replacement polices, a slightly different execution behaviour of some of the data structure operations. The results presented in this Section for the treap, the skip list, and the red-black tree data structures are comparable to the performance data reported in studies conducted by Sahni [24] and Papadakis [25].

7.3 Threaded Red-Black Tree Performance

The final experiment conducted in this study focused on quantifying the performance behaviour of the tree traversal operation utilizing a regular and a threaded red-black tree solution. The benchmark was conducted on the SMP system discussed in Section 6.0. The benchmark results depicted in Figure 5 reveal the improved traversal behaviour of the threaded red-black solution. Additional tests conducted on random and ascending data sets disclosed the same pattern, as in all the experiments, the threaded implementation outperformed the generic red-black data structure by an average of 12% (Appendix A). Analysing the insert performance of the 2 data structures showed the complexity increase of maintaining the additional pointers in the threaded solution though, as the regular red-black tree implementation outperformed the threaded data structure by an average of 5%.

8 Summary and Conclusion

The empirical analysis conducted for this study supports the mathematical abstractions for the tree data structures. In other words, the theoretical study of the tree structures and the resulting performance claims were highlighted through the conducted benchmarks. To illustrate, the AA implementation was capable of outperforming the red-black tree structure in all the remove cases. To summarize, in the mixed workload environment, the treap data structure outperformed the other 4 implementations by a rather significant margin. In the insert scenarios, the radix and the treap structure outperformed the more complex AA and red-black data structures. A similar picture was drawn by the search operation based benchmarks. It is interesting to point out that in the case of random insert operations, the red-black tree outperformed the treap in the case that 20,000+ nodes were populated into the data structure. The study revealed that based on the mixed workload profile, the treap represents the most efficient implementation whereas overall, the red-black data structure excelled from a consistency perspective. In other words, the red-black tree provided a rather consistent response time behaviour under varying workload patterns. The more light-way data structures (such as the treap or the skip list) on the other hand were rather fluctuation prone (a statement made based on the ascending, descending, and random operations executed at the same node level). Based on the mathematical and empirical study, the ramification is that the implementation of a red-black tree data structure in operating systems is considered as an effective and (to a lesser extent) efficient solution. The consistency factor reported in this study more than justifies the usage of a red-black tree implementation. To address the efficiency issue, one approach may be to explore the possibility of converting a standard red-black tree data structure into a threaded implementation, a design change that would allow expediting the search and traversal operations. As a search operation is part of any remove scenario (the node has to be located first), the remove operation benefits from the enhancement as well. The actual tradeoff revolves around faster lookup operations and increased pointer maintenance. Further, the treap has to be considered as a valuable alternative to any data structure. To illustrate, despite all 3 unbalanced solutions representing rather simple data structures, the treap outperformed the radix tree and the skip list in the mixed workload scenarios, whereas the radix tree represented the least efficient solution of all the benchmarked data structures. This study further discussed the performance gain that is possible by re-ordering the logical operations used in the data structures, and addressed the impact of compiler, systems architecture, and time complexity on the response time behaviour.

References

[1]
A. Andersson. “Balanced Search Trees Made Simple”, WADS, 1993.

[2]
A. Andersson, S. Nielsson. “A New Efficient Radix Sort”, FOCS, 1994.

[3]
S. Baase. “Computer Algorithms, Introduction to Design and Analysis”, 3rd ed., Addison-Wesley, 2000.

[4] M. Black. "Skip Lists vs. B-Trees”, CSI Essex, 2001.

[5]
T. Cormen. “Algorithms”, Second Edition, MIT Press, 2001.

[6]
M. Garey, D. Johnson. “Computers and Intractability: A Guide to the Theory of NP-Completeness”, Freeman, 1979.

[7]
G. Gonnet, R. Baeza-Yates. “Handbook of Algorithms and Data Structures”, 2nd. ed., Addison-Wesley, 1991.

[8]
R. Graham, D. Knuth, O. Patashnik. “Concrete Mathematics”, Addison-Wesley, 1989.

[9]
T. Hagerup, C. Rueb. “A Guided Tour of Chernoff Bounds”, 1990.

[10]
D. Hochbaum. “Approximation Algorithms for NP-Complete Problems”, PWS, 1997.

[11]
E. Horowitz and S. Sahni, Fundamentals of Computer Algorithms, Computer Science Press, 1978.

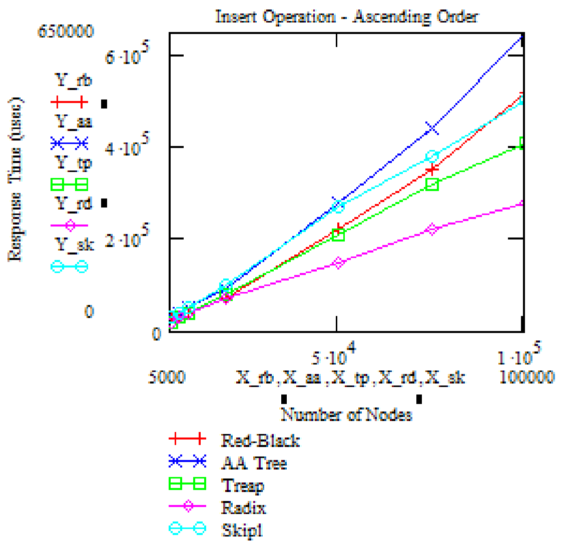
[12]
D. Knuth. “The Art of Computer Programming”, Volumes 1 and 3, Addison-Wesley, 1997 and 1998.

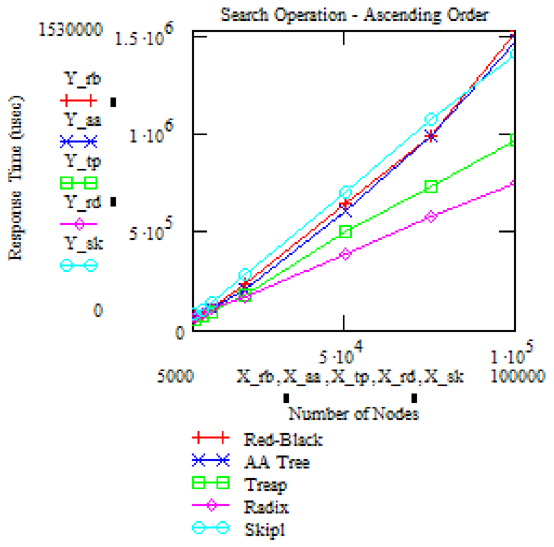
[13]
P. Messeguer. "Skip Trees, an Alternative Data Structure to Skip Lists in a Concurrent Approach", 1997.

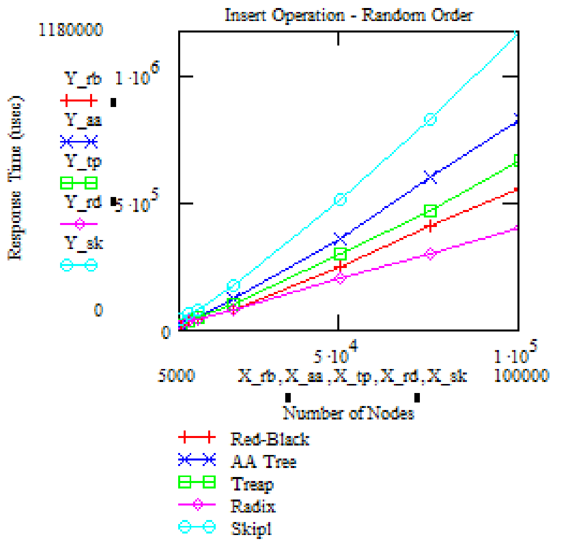
[14]
R. Motwani, P. Raghavan. “Randomized Algorithms”, Cambridge Univ. Press, 1995.

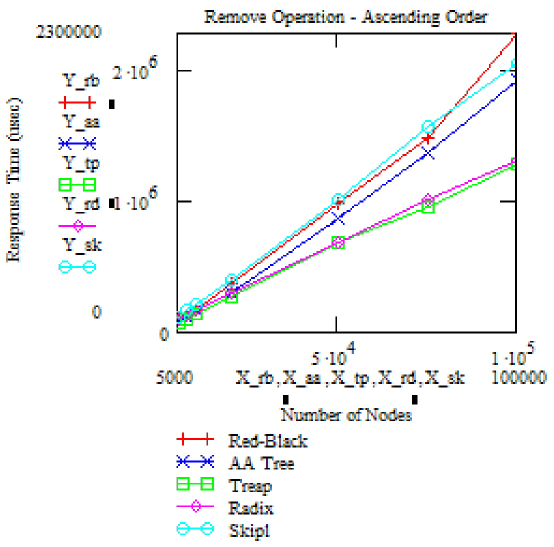
[15]
C. Papadimitriou. “Computational Complexity”, Addison-Wesley, 1994.

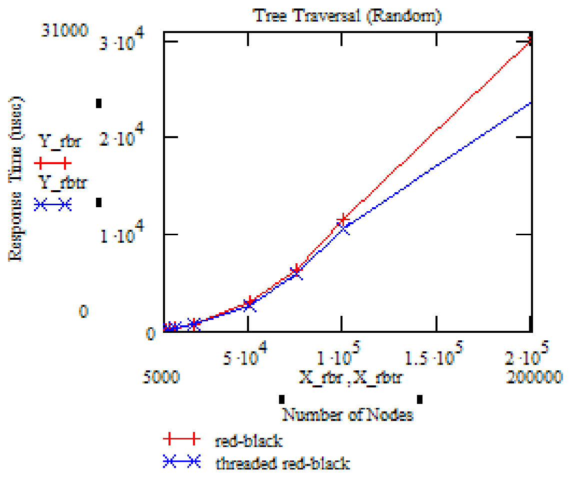
Appendix A: Additional Benchmark Charts

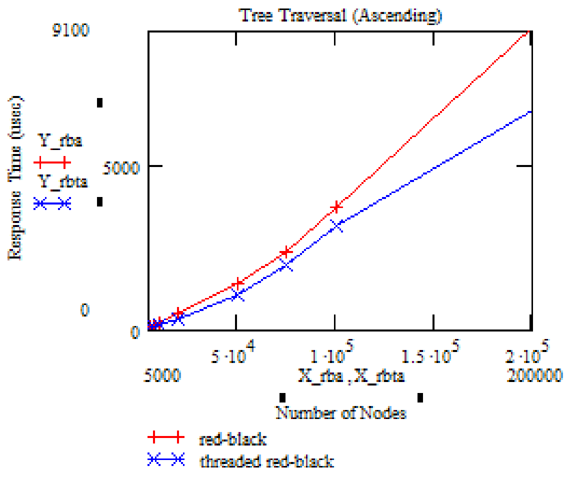












[16]
W. Pugh. "Skip Lists – A Probabilistic Alternative to Balanced Trees", ACM, 1990.

[17]
R. Sedgewick. "Algorithms" 2nd ed., Addison-Wesley, 1988.

[18]
R. Sedgewick, F. Lajolet. "An Introduction to the Analysis of Algorithms", Addison Wesley, 1996.

[19]
R. Seidel, C. Aragon. "Randomized Search Trees", Algorithmica 16, 1996.

[20]
C. Van Wyk. "Data Structures and C Programs" Addison-Wesley, 1988.

[21]
M. Weiss. "Data Structures and C Programs" Addison-Wesley, 1997.

[22]
N. Wirth. "Algorithms + Data Structures = Programs", Prentice Hall, 1978.

[23]
A. Harrison. "VLSI Layout Compaction using Radix Priority Search Trees", 1991.

[24]
S. Sahni, S. Cho. "A New Weight Balanced Binary Search Tree", University of Florida, TR 96-001, 1996.

[25]
T. Papadakis. "Skip Lists and Probabilistic Analysis of Algorithms", Ph.D. Dissertation, U. of Waterloo, 1993.

[26]
R. Fagin, J. Nievergelt, N. Pippenger, H. Strong. "Extendible Hashing – A Fast Access Method for Dynamic Files", ACM Transactions on Database Systems, 1979.

[27]
R. Enbody, H. Du. "Dynamic Hashing Schemes", ACM Computing, 1998.

News & Events

CEPIS Present in the European e-Skills 2004 Conference Long Term Strategies for E-Skills Development in Europe (Press Release)

The European Union should adopt a comprehensive strategy for improving ICT skills and training across all sectors, at all levels and for all citizens. This was one of the main messages of the European e-Skills 2004 Conference which ended Tuesday, 21 September at Cedefop in Thessaloniki, Greece. More than 150 experts took part in this major event, two years after the European e-Skills Summit organised by the Commission and the Danish Presidency in Copenhagen in 2002.

Among the participants, there were several representatives of EU Member States and acceding countries, of five Directorates General of the European Commission (Enterprise and Industry, Education and Culture, Employment and Social Affairs, Information Society and Eurostat) as well as the European Investment Bank, senior executives from leading ICT companies such as Microsoft, Nokia, Cisco Systems, IBM, Certiport, CompTIA etc. and researchers, academic and training world, representatives of European and international professional ICT associations (Council of European of Professional Informatics Professionals), consortia (Career Space, e-Skills Certification Consortium, e-Learning Industry Group, Project Management Institute) and delegations of the social partners (EICTA, Uni Europa and European Metal Workers Federation).

The Synthesis Report of the European e-Skills Forum "E-Skills in Europe: Towards 2010 and beyond" which constituted the basis for the discussions during this event pinpointed the threats of moving European ICT jobs to low-cost countries such as India and China (offshore or international outsourcing). For example, it is expected that by 2010 about 272.000 jobs will be lost in the UK alone due to international outsourcing. There is a tendency for companies to outsource services such as call centres, commercial handling and accounting, to countries with low labour costs. Central and Eastern European Countries and the new Member states, notably the Czech Republic, are increasingly attracting foreign direct investments from ICT companies because of their comparatively lower level of salaries and relative high skill level of their labour force. Highly skilled people are also being recruited in Europe, however, at lower speed.

This creates a serious dilemma for the EU Member States. On the one hand, their firms can lower labour costs by moving (in part or entirely) to low-cost countries, and thus improve competitiveness internationally. But at the same time, losing jobs in the ICT sector threatens social cohesion: ICT has been the main source of new employment in a time when more traditional sectors have been shedding employment opportunities. Mismatches and skill gaps persist however as many ICT jobs remain vacant due to the lack of qualified personnel. The number of current ICT specialists in Europe is 3.7 million and is estimated to reach 5.1 million by 2010.

Among priority actions discussed for 2005, the European e-Skills 2004 Conference also concluded that the European Commission should support alongside Cedefop and industry partners a "European level ICT skills meta- or reference framework" for better planning of investments in training and skills and must also further develop common principles for quality standards and for certification, whether public or private, profit or non-profit oriented. For these purposes it was proposed to create a European network of e-skills experts and a policy advisory group to develop foresight scenarios and further promote e-skills policies at the European level. The conference also proposed the creation of a European ICT career portal and of a central link between all educational institutions working in ICT, whether public or private.

More information at <http://www.eskills2004.org/>.

September 24, 2004

EUCIP News

Norway: EUCIP[1] and Abelia – Measuring Life Long Learning

Abelia (the Association of Norwegian ICT and Knowledge-based enterprises) was host to the conference "Measuring Life Long Learning" held on 14 September in Oslo in conjunction with partners Mintra AS, Norsk Test, EUCIP Norway, NITH and Energibedriftenes Landsforening (EBL).

The schedule contained sessions on net-based and interactive testing and a presentation about EUCIP by Renny Bakke Amundsen, in conjunction with one of Norway's largest learning providers NITH who are accredited to run the EUCIP programme in Norway.

13th September 2004

The 9th World Multi-Conference on Systemics, Cybernetics and Informatics: Call for Papers

This conference will take place in Orlando, Florida, USA, from July 10–13, 2005.

SCI 2005 is an international forum for scientists and engineers, researchers and, consultants, theoreticians and practitioners in the fields of Systemics, Cybernetics and Informatics. It is a forum for focusing into specific disciplinary research, as well as for multi, inter and trans-disciplinary studies and projects. One of its aims is to relate disciplines fostering analogical thinking and, hence, producing input to the logical thinking.

The conference´s Call for papers can be found at <http://www.iiisci.org/sci2005/website/callforpapers.asp>.

The best 10% of the papers will be published in Volume 3 of SCI Journal, <http://www.iiisci.org/Journal/SCI/Home.asp>. 12 issues of the volumes 1 and 2 of the Journal have been sent to about 200 university and research libraries. Free subscriptions, for 2 years, are being considered for the organizations of the Journals' authors.

We are emphasizing the area of Wireless/Mobile computing.

You can find information about the suggested steps to organize an invited session in the Call for Papers and in the conference web page: <http://www.iiisci.org/sci2005>.

If by any reasons you are not able to access the page mentioned above, please, try the following pages:
<http://www.iiis.org/sci2005>.

More information at
<http://www.iiisci.org/sci2005>.

1. EUCIP (European Certification of Information professionals, <http://www.eucip.com>) is a new pan-European qualification scheme, promoted by CEPIS, for people entering the IT profession and for IT professionals wishing to continue their professional development. EUCIP has been developed as an independent, globally recognised scheme for IT professionals in a similar fashion to the ECDL (European Computer Driving Licence) which is aimed at the IT User. The qualification will enable existing IT professionals to document their competencies and skill sets for employers or prospective employers and in addition, increase their market value.