Self-Adjusting of Ternary Search Tries Using Conditional Rotations and Randomized Heuristics

Ghada Hany Badr* and B. John Oommen †

Abstract

A *Ternary Search Trie* (TST) is a highly efficient dynamic dictionary structure applicable for strings and textual data. The strings are accessed based on a set of access probabilities and are to be arranged using a TST. We consider the scenario where the probabilities are not known *a priori*, and is time-invariant. Our aim is to adaptively restructure the TST so as to yield the best access or retrieval time. Unlike the case of lists and binary search trees, where numerous methods have been proposed, in the case of the TST, currently, the number of reported adaptive schemes are few. In this paper, we consider various self-organizing schemes that were applied to Binary Search Trees, and apply them to TSTs . Three new schemes, which are the splaying, the conditional rotation and the randomization heuristics, have been proposed, tested and comparatively presented. The results demonstrate that the conditional rotation heuristic is the best when compared to other heuristics that are considered in the paper.

Keywords : *Ternary Search Tries, Self Adjusting Data Structures, Adaptive Data Structures.*

1 Introduction

The concept of tries [7, 10, 22] was originally conceived by Brandais [15] and later given *that* name by Fredkin [18], which he derived from the word *re***trie***val* in *information retrieval systems*. Tries are one of the most general-purpose data structures used in computer science today [10], because their wide application improve most performance criteria.

Tries are widely used for the efficient storage, matching and retrieval of strings over a given alphabet. The primary challenge in implementing tries is to avoid using excessive memory for the trie nodes that are nearly empty [13]. When it comes to implementation [24], several options are possible depending on the decision structure chosen to guide the descent in the subtrees. One way to implement tries efficiently is to represent each node in the trie as a Binary Search Tree (BST). When the nodes are implemented as BSTs, the trie is given the name Ternary Search Tree (TST).

**Ph.D student*. This author can be contacted at: School of Computer Science, Carleton University, Ottawa, Canada : K1S 5B6. e-mail address: badrghada@hotmail.com.

†*Fellow of the IEEE*. This author can be contacted at: School of Computer Science, Carleton University, Ottawa, Canada : K1S 5B6. e-mail address: oommen@scs.carleton.ca.

More specifically, we consider the problem of performing a sequence of access operations on a set of strings $S = \{s_1, s_2, ..., s_N\}$. We attempt to solve this problem by representing the set of strings by using the above mentioned TST [24]. We assume that the strings are accessed based on a set of access probabilities $P = \{p_1, p_2, ..., p_N\}$, and are to be arranged using the TST. We also assume that P is not known *a priori*, and that it is time-invariant[1]. In essence, we attempt to achieve efficient access and retrieval characteristics by not performing any specific estimation process.

This paper demonstrates how we can apply two self-adjusting heuristics, that have been previously used for BSTs, to TSTs. These heuristics are, namely, the **splaying** [3, 34] and the **Conditional Rotation** [3, 32] heuristics. We have also applied another balancing strategy used for BSTs, to TSTs, which is the basis for the **Randomized** search trees, or Treaps [8, 17, 36]. In an earlier paper [34], Oommen and his co-authors showed that the conditional rotation heuristic is the best for the BST when the access distribution is skewed. In this paper, we see from experimental results that this heuristic, when modified and applied to TSTs, is also the best for the TST when the access distribution is skewed. The conditional rotation heuristic overcomes the disadvantage of expensive local adjustments during the accesses by doing rotations only if the adjustments anticipate a lower overall expected cost. We also showed that the conditional rotation heuristic is also good when the access distribution is almost uniform. In [22], Heinz *et al.* have shown through experimentation that TSTs are slower and less compact than *hash* tables and *Burst* tries, but our intention is to examine different balancing and self-adjusting heuristics that were previously used for the BSTs, and to apply *them* to TSTs. The advantage of this is to allow the TST to adapt itself according to the data without any parameter adjustments. The principal objective of our study is to demonstrate the improvements gained by different heuristics when applying them to TSTs, and to compare this with the original TSTs.

The growth in cache size and CPU speed has led to the development of cache-aware algorithms, of which a crucial feature is that the number of random memory accesses must be kept small. All the papers we consider here possess the same framework, namely that they don't consider caching. The main aim of our heuristic is to try to have the most likely items in the "top" nodes of the TST, which could then be kept in the cache so that the number of random memory accesses will be small. From that perspective, the data used in our experiments should really to considered to be of a "moderate" size, namely, that of a size in which the data can be entirely cache resident.

The paper organization is as follows: Section 2 describes tries and the possible applications in which they can be used. Section 3 presents the TST, the main structure that we will use, and describes how the rotation operation is achieved in the TST. Section 4 gives a brief literature survey about the self-adjusting heuristics used for BSTs that we will apply to the TST. Section 5 describes, in detail, how each of these methods can be applied to the TST . Section 6 gives the experimental setup and the different data sets used in the experiments, describes the experiments done, and presents a comparative survey of results. Section 7 briefly explains other heuristics that have been applied to the BST and can be applied to the TST. Section 8 concludes the paper.

[1]The body of literature available for non-stationary distributions is scanty. Little work has been done for lists where the distribution changes in a Markovian manner. Otherwise, almost all the work for non-stationary environments involves the non-expected case analysis, namely the amortized model of reckoning. Since this is not the primary focus of this paper, we believe that, in the interest of brevity, and of being concise, it is better to not visit these aspects here.

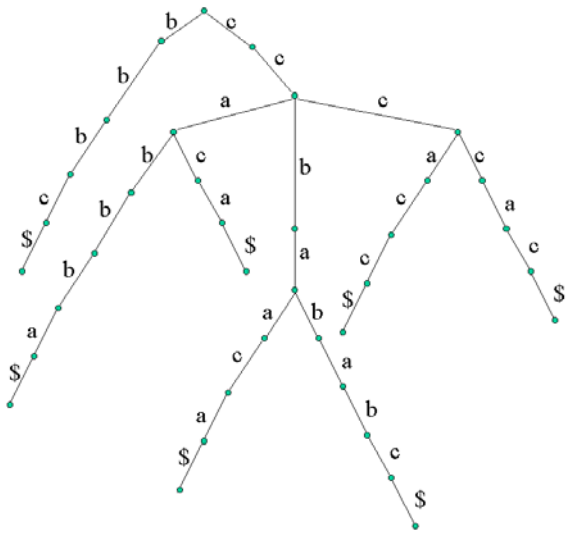

Figure 1: An example for the standard trie for the set {bbbc, ccabbba, ccaca, ccbaaca, ccbababc, cccaca, ccccac}.

2 Tries and Their Applications

A *trie* is an alternative to a BST for storing strings in a sorted order [22]. Tries are both an abstract structure and a data structure that can be superimposed on a set of strings over some fixed alphabet [24]. As an abstract structure, they are based on a splitting scheme, which, in turn, is founded according to the letters encountered in the strings. This rule can be described as follows:

If S is a set of strings, i.e., the input is a set of n strings called $S_1, S_2, \ldots S_n$, where S_i consists of symbols from a finite alphabet, $A = \{a_j\}_{j=1}^{r}$, and has a unique terminal symbol which we call \$, then the trie associated with S is defined recursively by the rule [24]:

$$trie(S) = (trie(S\backslash a_1), ..., trie(S\backslash a_r)).$$

where $S\backslash a_i$ means the subset of S consisting of strings that start with a_i, stripped of their initial letter a_i. The above recursion is halted as soon as S contains less than two elements. The advantage of the trie is that it only maintains the minimal prefix set of characters that is necessary to distinguish all the elements of S. Some examples of alphabets are: $\{0, 1\}$ for binary files, the 256-set ASCII set and the 26 / 52-Symbol English alphabet. Tries are not only used to search for strings in normal text, but can also be used to search for a pattern in a picture.

Clearly the tree $trie(S)$ supports the search for any string S_i in the set S by following an access path dictated by the successive letters of S_i [24]. In a similar manner, the trie can be implemented to deal with insertions and deletions, so that it is a fully dynamic dictionary data type.

Figure 1 shows an example of a trie constructed from the following 7 strings for the alphabet {a, b, c}: {bbbc, ccabbba, ccaca, ccbaaca, ccbababc, cccaca, ccccac}. The labels on the branches shows the character that leads from the previous node to the next node, independent of how the node is implemented.

When it comes to implementation [24], several options are possible depending on the decision structure chosen

to guide the descent in the subtrees. Three major choices present themselves:

- The "array-trie" uses an array of pointers to access subtrees directly; this solution is adequate only when the cardinality of the alphabet is small (typically for binary strings), for otherwise, it creates a large number of null pointers. But search in an array-trie is fast, requiring only a single pointer traversal for each letter in the query string. In other words, the search cost is bounded by the length of the query string.
- The "list-trie" structure remedies the high storage cost of array-tries by linking sister subtrees at the expense of replacing direct array access by a linked list traversal.
- The "bst-trie" uses binary search trees (BST) as the subtree access method, with the goal of combining the advantages of array-tries in terms of the time cost, and list-tries in terms of storage cost.

Tries are widely used for the efficient storage, matching and retrieval of strings over a given alphabet. Tries are also extensively used to represent a *set* of strings [22], that is, for dictionary management. The range of applications [22, 35] encompasses natural language processing, pattern matching, approximate pattern matching, information theory, telecommunications, wireless communications, molecular biology, game theory, coding theory, source coding, stock market analysis, searching for reserved words for a compiler, for IP routing tables, text compression, dynamic hashing, partial match retrieval of multidimensional data, searching and sorting, conflict resolution algorithms for broadcast communications, data compression, coding, security, genes searching, DNA sequencing, genone maps, the double digest problem, and so forth. Balanced tries (represented in a single array) are used as a data structure for dynamic hashing. This broad range of applications justifies the perception of tries as a general-purpose data structure, whose properties are well-understood.

3 Ternary Search Tries (TSTs)

Bentley and Sedgewick [12, 13] proposed the Ternary Search Trie (TST) as an implementation for the BST-trie, where the trie nodes are binary search trees. Consequently, in this case, the the character s_i will be used to search the BST representation of the current node to access the next node. Figure 2 shows the corresponding TST for Figure 1. If we search for the string "ccaca$", we will use the characters to search each BST that represents the nodes as we go along the path till we finish the string including the $, which means that the string exists. The figure shows the implementation of the "BST-trie" as a TST. In the TST, whenever we find the current character in the string equal to the character stored in the node, we follow the middle pointer to the next BST searching for the next character in the string. If we search for "ccb$", we will reach null pointer before we finish the string, which means that the string is not contained in the TST.

TSTs are efficient and easy to implement. They offer substantial advantages over both BST and digital search tries. They are also superior to hashing in many applications for the following reasons:

- A TST [12, 24] may be viewed as an efficient implementation of a trie that gracefully adapts to handle the problem of excessive memory for tries at the cost of slightly more work for the "full" nodes.

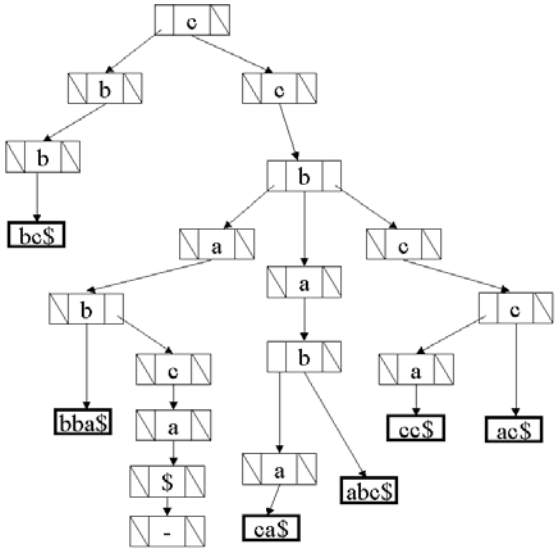

Figure 2: The corresponding TST for the set {bbbc, ccabbba, ccaca, ccbaaca, ccbababc, cccaca, ccccac}.

- TSTs combine the best of two worlds: The low space overhead of BSTs and the character-based time efficiency of tries.
- TSTs are usually substantially faster than hashing for unsuccessful searches[2].
- TSTs grow and shrink gracefully; hash tables need to be rebuilt after large size changes.
- TSTs support advanced searches, such as partial-matches and near-neighbor searches.
- TSTs support many other operations, such as traversal, so as to report the items in a sorted order.

The number of nodes in a TST is constant for a given input set, independent of the order in which the nodes are inserted [12]. The time for an access [34] in the TST, is bounded by the length of the string we search for, plus the number of left and right edges traversed. Thus, to minimize the access time we want to keep the depths of the solid subtrees small. Using this, we can extend the standard BST techniques to the TST, as we will see in the next sections.

3.1 Rotation in TST

TST's are amenable [34] to the same restructuring primitive as BSTs, namely the *rotation* operation, which takes $O(1)$ time, and which rearranges left and right children but not their middle children. This operation has the effect of raising (or promoting) a specified node in the tree structure while preserving the lexicographic order of the elements There are two types of rotations: *right* rotations and *left* rotations depending on whether the node to be raised is a left or right child of its parent respectively. Figure 3 shows the rotation operations as applicable to the TST.

The properties of rotations performed at node i are:

[2]This claim depends on properties such as the load factor of the hash table. We are grateful to the anonymous referee who pointed this out to us.

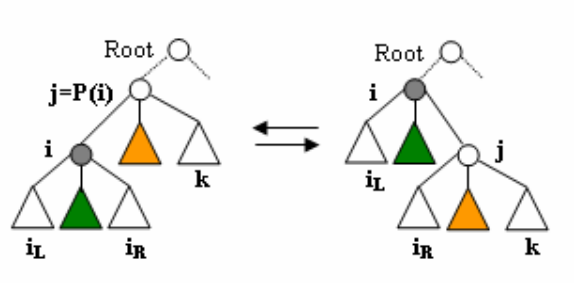

Figure 3: Rotation operations as applicable to the TST.

1. The subtrees rooted at i_L and i_R remain unchanged.
2. After a rotation is performed, i and parent of i, $P(i)$, interchange roles i.e., i becomes the parent of $P(i)$.
3. Except for $P(i)$, nodes that were ancestors of i before a rotation remain as ancestors of i after it.
4. Nodes that were not ancestors of i before a rotation do not become ancestors of i after it.
5. The middle nodes are not affected.

4 Self-Adjusting Heuristics

In the literature a lot of attention has been given to trees with bounded height, or balanced trees. The reason for this is obvious, since the worst-case access time is proportional to the height of the tree. However, balanced trees [5, 19] are not as efficient if the access pattern is nonuniform; furthermore, they also need extra space for storage of the ”balance” information [34]. These trees are all designed to reduce the worst-case time per operation. However, in some typical applications of search trees, the system encounters a sequence of access operations, and what matters is the total time required for the sequence and not the individual times required for each access operation. In such applications, a better goal is to reduce the ”amortized times” of the operations, which is the average time of an operation in a worst-case sequence of operations. In other applications, one seeks to minimize the average asymptotic cost of the data structure.

One way to obtain amortized average case efficiency is to use a ”self adjusting” data structure [3]. The structure can be in any arbitrary state, but after each access operation (or a set of access operations) a restructuring rule is applied, which, in turn, is intended to improve the efficiency of future operations.

Self adjusting data structures have several possible advantages [3] over balanced, or otherwise explicitly, constrained structures. These advantages, cited from [3], are:

1. Ignoring constant factors, self adjusting structures are never much worse than constrained structures, and since they adjust according to usage, they can be much more efficient if the access distribution is skewed.
2. They often need less space, since no balance or other constraint information is stored.
3. Their access and update algorithms are conceptually simple, and very easy to implement.

Although these advantages have been cited in [3], they have to be qualified[3]. With regard to the first advantage, it is true that, in general, adaptive structures possess these advantages, but for some structures, for example, for splay trees, there is a reduction in worst-case complexity, from $O(n^2)$ to $O(n\ log\ n)$ total cost. But in this case the possible savings are by a constant factor, bounded above by the alphabet size. With regard to the second advantage, adaptive structures only require less space than cost-balanced structures, but not less than other structures, in general. Indeed, because these structure require parent pointers, they generally require more space than unbalanced structures. Finally, the third advantage cited in [3], is arguable, as anyone who has attempted to teach adaptive structures to undergraduate students can attest.

Self adjusting structures have two possible disadvantages:

1. They require more local adjustments, especially during accesses, but explicitly constrained structures need adjusting only during updates, and not during accesses.

2. Individual operations within a sequence can be expensive, which may be a drawback in a real-time application.

As we shall see presently, numerous methods have been proposed for the self-adjusting of lists and binary search trees. For tries, most of the work that has been done, has attempted to change the structure of the nodes of the tries, or even change the structure of the trie itself, in order to save more storage (compression) and/or to render the access time more efficient [1, 6, 10, 11, 28]. A little work has been done in the self-adjusting of tries based on the underlying access distribution. To the best of our knowledge, the only work directly pertaining to this is the Burst trie [21, 22]. The Burst trie starts with a single container, implemented in [21, 22] as a BST with a move-to-front heuristic. When the number of nodes in the container starts to be large, (as per a pre-defined criterion), it "bursts" to form a node of the trie that points to smaller containers, and so on. Although it is an elegant data structure which is very efficient for strings with skewed distribution, the disadvantage of the Burst trie, as we see, is that it needs a number of parameters which are to be tuned or adjusted. These may have to be varied according to the data for which it is used. The worst case for the Burst trie is where the strings are of equal lengths and have a relatively flat probability distribution. For text data, its performance depends on the fact that words most commonly used are short, and are stored wholly within the access trie. Rare strings are held in containers, but because they are typically long, there is no loss of efficiency [22]. The cost of searching the container is offset by the saving of not having to traverse a large number of trie nodes.

Indeed, if we work with the assumption that the structure of the node is not permitted to be changed in any restructuring operation, to the best of our knowledge, there is no work reported for creating and manipulating self-adjusting TSTs. The only work reported, is for extending the splaying operation introduced by Sleator and Tarjan in [34] for BSTs, which has since been applied for muti-way trees [33, 38].

In [22], Heinz *et al.* have shown, through experimentation that TSTs are slower and less compact than *hash* tables and *Burst* tries. We attempt to examine different balancing and self-adjusting heuristics that were previously used for BSTs, and to apply them to TSTs. The advantage of this is to allow the TST to adapt itself according to the data

[3]We are grateful to anonymous Referee for pointing out these qualifications.

without any parameter adjustments.

The problem of constructing efficient BSTs has been extensively studied. The optimal algorithm due to Knuth [25], uses dynamic programming and produces the optimal BST using $O(n^2)$ time and space. Alternatively, Walker and Gotlieb [37] have used dynamic programming and divide-and-conquer techniques to yield a nearly-optimal BST using $O(n)$ space and $O(nlogn)$ time.

Self-adjusting BSTs are closely related to the subject of self-organizing lists. A self-organizing list is a linear list that rearranges itself such that after a long enough period of time, it tends towards the optimal arrangement with the most probable element at the head of the list, and the rest of the list being recursively ordered in the same manner. Many memory-less schemes have been developed to reorganize a linear list dynamically [9, 14, 20, 23, 25, 26]. Among these are the move-to-front rule [25, 26] and the transposition rule [31]. These rules, their extensions, and their analytic properties are discussed extensively in the literature [9, 14, 20, 23, 25, 26]. Schemes involving the use of extra memory have also been developed; a review of these is found in [23]. The first of these uses counters to achieve estimation. Another is a stochastic move-to-rear rule [29] due to Oommen and Hansen, which moves the accessed element to the rear with a probability which decreases each time the element is accessed. A stochastic move-to-front [29] and a deterministic move-to-rear scheme [30] due to Oommen *et. al.* have also been reported.

The primitive tree restructuring operation used in most BST schemes is the well known **Rotation** [2]. A few memory-less tree reorganizing schemes which use this operation have been presented in the literature among which are the **Move-to-Root** and **Simple Exchange** rules. These rules are analogous in spirit to the move-to-front and transposition rules respectively for linear lists. The Move-to-Root Heuristic was historically the first self-organizing BST scheme in the literature [4] and is due to Allen and Munro. It is both conceptually simple and elegant. Each time a record is accessed, rotations are performed on it in an upwards direction until it becomes the root of the tree. The idea is that a frequently accessed record will be close to the root of the tree as a result of it being frequently moved to the root, and this will minimize the cost of the search and the retrieval operations. Allen and Munro [4] also developed the Simple Exchange rule, which rotates the accessed element one level towards the root, similar to the transposition rule for lists. Contrary to the case of lists, where the transposition rule is better than the move-to-front rule [31], they show that whereas the Move-to-Root scheme has an expected cost that is within a constant factor of the cost of a static optimum BST, the simple exchange heuristic does not have this property. Indeed, it is provably bad.

Sleator and Tarjan [34] introduced a technique, which also moves the accessed record up to the root of the tree using a restructuring operation called **splaying** which is a multi-level generalization of rotation. Their structure, called the splay tree, was shown to have an amortized time complexity of $O(logN)$ for a complete set of tree operations which included insertion, deletion, access, split, and join. This heuristic is rather ingenious. It is a restructuring move that brings the accessed node up to the root of the tree, and also keeps the tree in a symmetric order, and thus an in-order traversal would access each item in order from the smallest to the largest. Additionally it has the interesting side effect of tending to keep the tree in a form that is nearly height-balanced apart from also capturing the desired effect of keeping the most frequently accessed elements near the root of the tree. The heuristic is somewhat similar to the Move-to-Root scheme, but whereas the latter has an asymptotic average access

time within a constant factor of the optimum when the access probabilities are independent and time invariant, the splaying operation yields identical results even when these assumptions are relaxed. More explicit details and the analytic properties of the splay tree with its unique "two-level rotations" can be found in [32, 34]. Splaying at a node x of depth d takes $\Theta(d)$ time. That is, the time required is proportional to the time needed to access the item in x. Splaying not only moves x to the root, but roughly halves the depth of every node along the access path. On an n-node splay tree all the standard search tree operations have an amortized time bound of $O(logn)$ per operation. Thus the splay trees are as efficient as balanced trees when the total running time is the measure of interest. In a splay tree, whenever a node is accessed via standard operations, it is splayed, thus making it the root. More details about splaying will be shown when it is applied to the TST in Section 5.1.

The disadvantage of the splay tree is that the cost of the access operation is high due to the large amount of restructuring operations done. Also, the splaying rule always moves the record accessed up to the root of the tree. This means that if a nearly optimal arrangement is reached, a single access of a seldomly-used record will disarrange the tree along the entire access path [8, 32] as the element is moved upwards to the root. Thus the number of operations done on every access is exactly equal to the depth of the node in the tree, and these operations are not merely numeric computations (such as those involved in maintaining counters and "balancing functions") but rotations. Thus the splaying rule can be very expensive. Moreover [8], this is undesirable in a caching or paging environment where the writes involved in the restructuring will dirty memory locations or pages that might be otherwise stay clean.

To overcome the disadvantages of splaying, Oommen *et. al.* [32] proposed a heuristic that concentrates on these problems. They introduced a new heuristic to reorganize a BST so as to asymptotically arrive at an optimal form. It requires three extra memory locations per record. Whereas the first counts the number of accesses to that record, the second counts the number of accesses to the subtree rooted at that record, and the third evaluates the **Weighted Path Length** (WPL) of the subtree rooted at that record. The paper [32] specifies an optimal updating strategy for these memory locations. More importantly, however, it also specifies how an accessed element can be rotated towards the root of the tree so as to minimize the overall cost of the tree. Finally, unlike most of the algorithms that are currently in the literature, this move is not done on every data access operation. It is performed if and only if the overall WPL of the resulting BST decreases. Oommen *et. al.* [32] also present a space-optimized version which only requires a **single** additional counter per node. Using this memory location they designed a scheme identical to the one described above. The details of applying this method is shown in [32]. We will discuss it in more detail when we apply it to the TST.

One Balancing strategy for BSTs, that we will apply to the TST, is Randomized Search trees, or Treaps [8, 17, 36]. 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 [8]. 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 *inorder* with respect to the keys and in *heaporder* with respect to the priorities, where the latter items are to be understood as follows:

- ***Inorder*** 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.

- ***Heaporder*** 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 exists. The item with largest priority becomes the root, and the allotment of the remaining items to the left and right subtree is then determined by their keys. Put differently, the treap for an item set X is exactly the binary search tree that results from successively inserting the items of X in the order of their decreasing priorities into an initially empty tree using the usual leaf insertion algorithm for binary search trees. Given the key of some item $x \in X$, the item can easily be located in the treap using the usual search tree algorithm.

We now address the question of updates: The *insertion* of a new item z into T can be achieved as follows: At first, using the key of z, attach z to T in the appropriate leaf position. At this point the keys of all the nodes in the modified tree are in inorder. However, the heaporder condition might not be satisfied, i.e. z's parent might have a smaller priority than z. To reestablish heaporder, we simply rotate z up as long as it has a parent with smaller priority (or until it becomes the root). *Deletion* of an item x from T can be achieved by "inverting" the insertion operation: First locate x, then rotate it down until it becomes a leaf, where the decision to rotate left or right is dictated by the relative order of the priorities of the children of x, and finally the leaf is "clipped away" or deleted.

The random search tree for X is defined [8] to be a treap for X where the priorities of the items are independent, identically distributed continuous random variables.

5 Proposed Restructuring Methods for Ternary Search Tries (TST)

n this section we demonstrates how we can apply two self adjusting heuristics, that have been previously used for BSTs, to TSTs. These heuristics are the **splaying** [3, 34] and the **Conditional Rotation** [3, 32] heuristics. We have also applied another balancing strategy used for BSTs, to TSTs, which is the basis for the **Randomized** search trees, or Treaps [8, 17, 36]. The splaying method has already been proposed (albeit, less formally) applied to the TST [34], but the other two schemes which we suggest are newly proposed to be applied to the TST.

5.1 Splaying for TST

Sleator and Tarjan [34] introduced the idea for extending the splaying as a restructuring heuristic after each access for the TST. We will call the resulting data structure the *Splay-TST*.

To splay [34] at a node x, we proceed up the access path from x, one or two nodes at a time, carrying out the same splaying steps as in the BST, with the additional condition that whenever x becomes a middle child, we continue from $p(x)$ instead from x. Formally, the following splaying step is repeated until x is the root of the current BST (Figure 4):

- ***Case 1 (zig)***: If $p(x)$, the parent of x, is the root of the tree or the middle node of its parent, and x is not a middle child of $p(x)$, rotate the edge joining x with $p(x)$. If x is a middle child of $p(x)$, then splay $p(x)$.
- ***Case 2 (zig-zig)***: If $p(x)$ is not the root, and x and $p(x)$ are both left or right children, rotate the edge joining

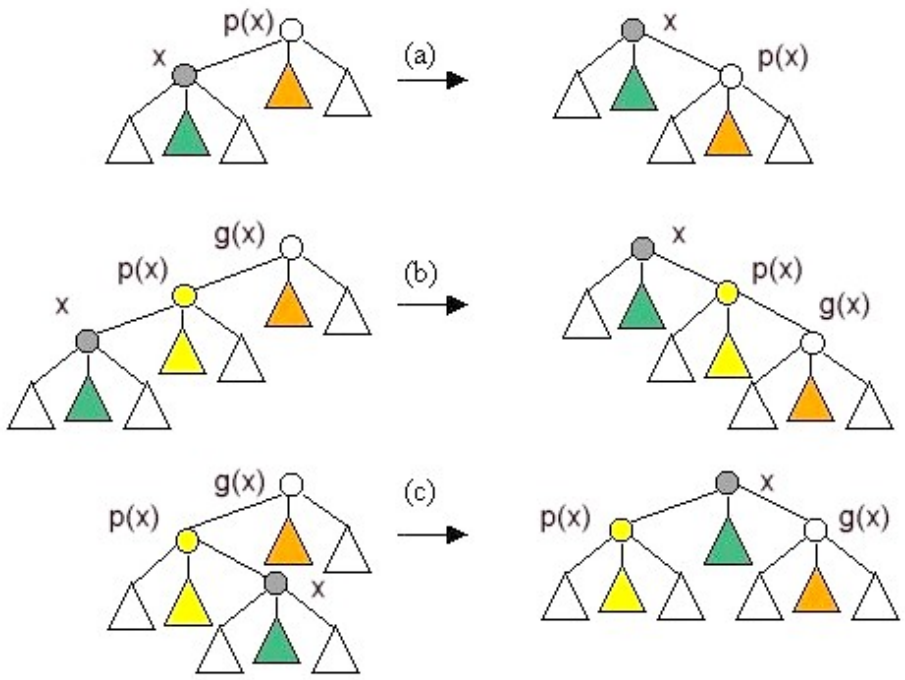

Figure 4: Different cases for the splaying operation for the TST. Node x is the current node involved in the splaying. The cases are: (a) Zig. (b) Zig-Zig. (c) Zig-Zag.

$p(x)$ with its grandparent, $g(x)$, and then rotate the edge joining x with $p(x)$. If x is a middle child of $p(x)$, then splay $p(x)$.

- ***Case 3 (zig-zag)***: If $p(x)$ is not the root, and x is a left child and $p(x)$ a right child, or vice-versa, rotate the edge joining x with $p(x)$, and then the edge joining x with the new $p(x)$. If x is a middle child of $p(x)$, then splay $p(x)$.

These steps are performed from the bottom of the TST towards the root until the recursion terminates with Case (1). Whenever we reach the root of a current BST during the splay process, we continue splaying for the parent node unless the root of the current BST is the root of the whole TST.

To state Sleator and Tarjan's results for the BST, we introduce the following notation. Let $w(i)$ be the weight of a node x_i, which is an arbitrary positive value, subjectively assigned to the nodes. Also let $s(i)$ be the size of a node x_i, which is the sum of the individual weights of all the records in the subtree rooted at x_i. Similarly, let $r(i)$ be the rank of a node i, defined as $r(i) = log_2(s(i))$. Then, if p_i is the access probability of x_i, the following are true for a sequence of m accesses on an N-node splay tree:

Theorem 1. *(Balance theorem) The total access time is* $O((m+n) \cdot logn + m)$.

Theorem 2. *(Static optimality theorem) If every record is accessed at least once, the total access time is*

$$O(m + \sum_{i=1}^{n} p_i \cdot log(\tfrac{m}{p_i})).$$

Theorem 3. *(Static finger theorem) If f is any fixed item, the total access time is*

$$O(n \cdot logn + m + \sum_{j=1}^{m} log(|i_j - f| + 1)),$$

where the items are assumed to be numbered 1 through n in a symmetric order, and the sequence of accessed items is $i_1, i_2, \ldots, i_m$.

Theorem 4. *(Working set theorem) Suppose the accesses are numbered from 1 to m in the order in which they occur. For any time instant j at which the record accessed is x_j, let $t(j)$ be the number of distinct items accessed before j, subsequent to the last access of item x_j, or subsequent to the beginning of the sequence if j is the first access to x_j. Then the total access time is:*

$$O(n \cdot logn + m + \sum_{j=1}^{m} log(t(j) + 1)).$$

Theorem 5. *(Unified theorem) The total time of a sequence of m accesses on an n-node splay tree is*

$$O(n \cdot logn + m + \sum_{j=1}^{m} \textit{ log min } \{\frac{m}{s_{i_j}}, |i_j - f| + 1, t(j) + 1\}),$$

where f is any fixed item, and $t(j)$ is defined as in Theorem 4 above.

Sleator and Tarjan also derived upper bounds for each of the operations supported by the splay tree. In the interest of brevity, we shall only present the upper bounds for the access operation. The details of the other operations can be found in [34].

Let W be the total weight of the tree included in the access operation. For any item i in a tree T let $i-$ and $i+$ denote the item preceding and the item following i, respectively, in the symmetric order, where $w(i-) = \infty$, if $i-$ is undefined, and $w(i+) = \infty$ if $i+$ is undefined. Then, the upper bound on the amortized time complexities of the access operation is:

$$3log(\frac{W}{w(i)}) + 1 \text{ if } i \in t;$$

$$3log(\frac{W}{min\{w(i-), w(i+)\}}) + 1 \text{ if } i \notin t;$$

Theorems 1 through 4 are particularly important, because they imply that:

1. Over a long enough sequence of accesses, a splay tree is as efficient as any type of uniformly balanced tree.
2. A splay tree is as efficient as any fixed search tree, including the optimum tree for the given access sequence.
3. Splay trees support accesses in the vicinity of a fixed finger with the same efficiency as finger search trees.
4. The most recently accessed items, which can be conceptually imagined to form a "working set", are the easiest to access.

Theorem 5 unifies all of the above into a single result.

The amortized analysis of the Splay-TST is given in [34], and it shows that the amortized time to access a string s in a Splay-TST is $O(|s| + logM)$, where M is the number of strings stored in the tree.

5.2 Randomized TST

As we saw in Section 4, random BST adjustments are done during insertions, and no adjustments are done during searching. The price that we pay, of course, is the extra memory requirement for the priority of each node. In [8], the authors showed that it is possible to implement randomized search trees in such a manner that no priorities are stored explicitly. They offered three different methods. The first uses hash functions to generate priorities on demand. The second method stores the nodes of the tree in a random permutation and uses the node addresses themselves as priorities. The last method recomputes priorities from subtree sizes. In our implementation we explicitly utilized stored priorities, because the access time was our primary consideration.

The randomization is usually used when the distribution of the data accesses is uniform, and the items are all equally likely to be accessed. This heuristic is included to study the effect of randomization on TSTs, and to see how its performance is compared with the performance of other heuristics for uniformly distributed data sets. The effect of randomization for non-uniform distributions is typically studied using a few traditional distributions such as the Zipf's, exponential etc. We shall discuss these aspects in a subsequent section when the properties of TSTs are experimentally verified.

Aragon and Seidel [8] analyzed a number of interesting quantities related to Randomized Binary Search Trees (RBSTs) and derive their expected values. Some of the quantities are:

- $D(x)$, the depth of node x in a RBST, which is the number of nodes in the path from x to the root.
- $S(x)$, the size of the subtree rooted at node x, which is the number of nodes contained in that subtree.

Let n be the number of nodes in an RBST. Then the following lemmas and theorem (whose proofs are omitted) are stated in [8].

Lemma 1. *Let T be the treap of n items, and let $1 \leq i, j \leq n$. Then, assuming that all the priorities are distinct, x_i is an ancestor of x_j in T iff among all x_h, with h between i and j, the item x_i has the largest priority.*

Theorem 6. *Let $1 \leq l < n$. In a randomized search tree of n nodes the following expectations hold:*

- $E_x[D(x_l)] < 1 + 2 \cdot ln\ n$.
- $E_x[D(x_l)] < 1 + 2 \cdot ln\ n$.

Lemma 2. *In a randomized search tree with $n > 1$ nodes, we have for index $1 \leq l \leq n$ and any $c > 1$*

$$P_r[D(x_l) \geq 1 + 2cln\ n] < 2(n/e)^{-cln(c/e)}.$$

This balancing method can also be applied to the TST, if we take into consideration that searching for a string s in a TST is as if we search for each character of the string in a *separate* BST. So, each of these BSTs could be balanced using the "Random Tree" property.

In other words, during insertion of s in a TST each character c in s is assigned a random priority, and then inserted in its corresponding BST, T. At this time, if the character is found in T, we skip to the next character, and

no adjustments are done in T. If the character is not found, an insertion is done at the proper leaf position, so as to maintain the inorder property. In order to keep the heaporder property, we simply rotate c up as long as it has a parent with smaller priority, or until it becomes the root of *its* BST, T. This is done for every character of the string during the insertion process.

The search is done using the usual search algorithm of the TST. We will call the resulting data structure the *Rand-TST*.

As shown in [8] and the theorems presented above, the expected cost of searching a RBST is $O(log(n))$. As a simple extension, the expected cost of searching the Rand-TST is $O(|S| + log(M))$, where $|S|$ is the size of the string searched for, and M is the total number of strings stored in the tree.

5.3 Conditional Rotations for TST

As mentioned, the basic philosophy we have adopted to achieve adaptive TST restructuring is to adopt the same adaptive restructuring scheme for the tree associated with *every* BST along the search path of the string. We shall now see how we can adapt the conditional rotation heuristic for this purpose. In this heuristic, the rotation is not done on every data access operation. It is performed if and only if the overall WPL of the entire BST rooted at the *current middle* node decreases. So the net effect is as if we are invoking the same scheme for every BST that represents a character in the string. Clearly the maximum number of rotations performed is $\mid s \mid$. We will call the resulting data structure the *Cond-TST*.

The reader must observe the difference between the current scheme and the original conditional rotation scheme introduced in [32]. In the latter there was only one BST which was adaptively restructured. In this case, there is a BST at *every* node, and this restructuring is done only if the "estimated" cost of the *middle* node for *that* BST decreases. Thus the convergence results are true for the TST as a consequence of the result being true for every single BST of every node of the TST. Observe too that in this way, we are trying to diminish the problem associated with splaying, which does too many rotations during the access operation. We also try to gain the advantages of self adjusting strategies and the advantages of balancing strategies with the additional expense of only a single extra memory location per node.

In [32] Cheetham *et. al.* developed two algorithms for this heuristic when applied to the BST. The first requires three extra integer memory locations for each record in the BST, and the second, the space-optimized version, requires only a single extra memory location. We will first discuss the properties of the first one, and later discus the optimized version, but with respect to the TST. We will use the optimized algorithm to test the heuristic for the TST.

For each node i of every BST t_c in the TST, T, we include three extra memory locations. The current BST t_c is the BST rooted at the c^{th} middle node traversed during the access. The three memory locations are:

- ***First counter*** $\alpha_i(n)$: total number of accesses of a node i in t_c up to and including time n.

- ***Second counter*** $\tau_i(n)$: the total number of accesses to the subtree, of t_c, rooted at node i. Clearly $\tau_i(n)$ satisfies:

$$\tau_i = \sum_{j \in T_i} \alpha_j \quad \forall j \in t_c \; of \; T. \tag{1}$$

- ***Third counter*** $\kappa_i(n)$: the WPL of the subtree, of t_c, rooted at node i at time instant n. Let $\lambda_i(n)$ be the path length of i from the root of t_c. Then:

$$\kappa_i = \sum_{j \in T_i} \alpha_j \cdot \lambda_j \quad \forall j \in t_c \; of \; T. \tag{2}$$

  Equivalently,

$$\kappa_i = \sum_{j \in T_i} \tau_j \quad \forall j \in the \; BST \; t_c \; of \; T. \tag{3}$$

These quantities (α, τ and κ) are maintained for every node in t_c, and are updated after every access operation after we find the character sought for in t_c. In order to avoid a traversal of the entire BST, t_c, for updating τ and κ, Cheetham *et. al.* noted the following recursively computable properties for the quantities τ and κ applied to t_c:

$$\tau_i = \alpha_i + \tau_{iL} + \tau_{iR} \tag{4}$$

$$\kappa_i = \alpha_i + \tau_{iL} + \tau_{iR} + \kappa_{iL} + \kappa_{iR} \tag{5}$$

where iR and iL denote the right and left children of the node i.

These properties imply that to calculate τ and κ for any node, it is necessary only to look at the values stored at the node and at the left and right children of the node.

Observe that in order to update α, only a single top-down pass on the TST is required. This can be done by incrementing the α values as we traverse down the access path. Using Equations (4) and (5), we present a simple scheme to calculate the α, τ, and κ fields of the nodes of t_c after each access. The recursions motivating the scheme are highlighted in the following theorem.

Theorem 7. *Let i be any arbitrary node in the TST, T. Node i stores the current character of the string we search for. At this time, we have finished searching the current BST t_c, and then we move to the next BST rooted at the middle child of i. On accessing $i \in T$, the following updating schemes for α, τ, and κ are valid whether or not a rotation is performed at i.*

- ***Updating*** α

  We need to update α only at node i as per:

$$\alpha_i = \alpha_i + 1$$

- ***Updating*** τ

  We need to update the nodes in the access path from the root of the current BST to the node before the parent $P(i)$

according to the rule:

$$\tau_j = \tau_j + 1$$

If a rotation is performed, i and $P(i)$ are updated by applying:

$$\tau_{P(i)} = \alpha_{P(i)} + \tau_{P(i)L} + \tau_{P(i)R}$$

followed by

$$\tau_i = \alpha_i + \tau_{iL} + \tau_{iR}$$

If no rotation is performed, they are updated as other nodes in the path.

- ***Updating*** κ

  We need to update the nodes in the access path from the root of the current BST to node i by applying:

$$\kappa_j = \alpha_j + \tau_{jL} + \tau_{jR} + \kappa_{jL} + \kappa_{jR}$$

  from node i upwards to the root of the current BST.

Proof: The proof of these updating rules follows from the corresponding proofs of the respective updating rules for the case of the BST in [32]. Since the updating is done for independent BST's in the TST, T, they are true for *every* BST, and so they are true for the entire TST. The arguments are not included here to avoid repetition. □

The main contribution of Cheetham *et. al.* [32] is their proposal of a criterion function θ_i which can be used to determine whether a rotation should be applied at node i or not. Let θ_i be $\kappa_{P(i)} - \kappa'_i$, where the primed quantity is a post-rotational quantity (i.e., it is the value of the specified quantity after the rotation has been performed). The criterion function, θ_i reports whether performing a rotation at node i will reduce the κ-value at $P(i)$ or not. This is of interest because it can be shown that the κ-value of the *entire* current BST is reduced by a rotation if and only if θ_i is reduced by the *local* rotation at i (referred to as a κ-lowering rotation). The result is also true for the TST and is formally given below.

Theorem 8. *Any κ-lowering rotation performed on any node in the current BST, t_c, will cause the weighted path length of t_c to decrease, and so this will decrease the accumulated WPL of the middle nodes along the search path of the entire TST, T.*

Proof: This theorem is true for every BST of T as shown in [32], and so it is true, in particular, for t_c. Since we decrease the WPL of every BST whose root is a middle node in the search path, the accumulated WPL of the middle nodes will decrease, and so the result will also be true for the entire TST, T. □

Assuming that the average path length of the current BST with n nodes is $k_1 logn$, the original algorithm requires $2k_1 logn$ time, and $3k_1 n$ space over and beyond what is required for the tree itself.

Now that we are familiar with the original algorithm, we shall examine a space-optimized version that requires only n extra memory locations and $logn$ time for every BST. This version is based on the observation that the values

of α_i and κ_i are superfluous because the information stored in the α_i values is also stored in the τ_i values and the values of κ_i can be expressed in terms of τ_i and α_i, as in Equation (5).

To see the effect of any tree restructuring operation, the correct method would be to perform the operation and to compare the cost before and after it is done. The reason for this is that the operation can be done anywhere in the tree, but the effect of the operation must be observed at the root of the tree. Cheethman *et. al.* [32] were able to show the existence of a function ψ_i, which can be locally computed and simultaneously "forecast" the effect at the global level. As in [32], we state a new criterion function, ψ_i, defined only in terms of τ_i and α_i and show that a local evaluation of ψ_i is sufficient to anticipate whether a rotation will *globally* minimize the cost of the overall TST or not. If $\psi_i > 0$, then the rotation should be done as it will globally minimize the cost of the whole TST. This result is formally stated as follows.

Theorem 9. *Let i be the accessed node of the current BST t_c, in the TST, T, and let $\kappa_{P(i)}$ be the weighted path length of the BST rooted at the parent $P(i)$ if no rotation is performed on node i, and i is not the middle node of $P(i)$. Let i_L and i_R be the left and right children of $P(i)$ respectively. $P(i)$ will be assumed null if i is the middle child or the root of T. Let κ_i' be the weighted path length of the BST rooted at node i if the rotation is performed. Furthermore, let ψ_i be defined as follows:*

$$\psi_i = \alpha_i + \tau_{iL} - \alpha_{P(i)} - \tau_{B(i)} \text{ if i is a left child};$$

$$\psi_i = \alpha_i + \tau_{iR} - \alpha_{P(i)} - \tau_{B(i)} \text{ if i is a right child}.$$

Then, if $\theta_i = \kappa_{P(i)} - \kappa_i'$,

$$\psi_i \geq 0 \text{ if and only if } \theta_i \geq 0.$$

Proof: This is one of the fundamental results in [32]. Typically, when a restructuring is to be done, the operation will cause the cost of the entire data structure to change. However, Cheetham *et al* showed that in the case of the conditional rotation, it suffices to do a local computation so as to determine whether the global, overall cost of the tree will decrease. This computation involves the quantity ψ_i for the BST. If ψ_i is positive, it can be shown that the cost of the overall BST decreases.

The question of extending this concept to the TST is actually quite straightforward. Since the actual contents of the nodes are unchanged, and since the TST comprises of individual BSTs which are "disjoint", we can determine whether we will achieve a global optimization by merely computing the value of ψ_i for each separate BST. Since the cost of every BST can be guaranteed to be decremented whenever the corresponding ψ_i is positive, and since the decrease of the cost of any one BST does not effect the increase / decrease of the cost *of any other* BST, it is straightforward to see that the cost of the overall TST is also decreased whenever all the ψ_i's are positive. But since the other ψ_j's, (other than the one that is local to the current BST's) are unchanged, the fact that the current ψ_i is positive suffices to guarantee that the global cost of the overall TST also decreases. Hence the theorem ! □

Observe that we need not maintain the κ fields because ψ_i only requires the information stored in the α and

τ fields. This implies that after the search for the desired record (requiring an average of $O(logn)$ time) and after performing any reorganization (which takes constant time), the algorithm does not need to update the κ values of the ancestors of i. Also note that at node i, the α values may be expressed in terms of the τ values, i.e. $\alpha_i = \tau_i - \tau_{iL} - \tau_{iR}$, and need not be explicitly stored. Therefore, for every node of every BST in the TST, we need only one extra memory location, and a second pass of the current BST is not required. Consequently, the modified algorithm, requires only n extra memory locations and $O(|S| + log(M))$ time, where $|S|$ is the size of the string searched for, and M is the total number of strings stored in the tree.

6 Experimental Results

The adaptive restructuring mechanisms introduced in this paper, which are applicable for the TST, have been rigorously tested so as to evaluate their relative performance. In all brevity, we mention that the performance of the new strategy for the cond-TST introduced here are, in our opinion, quite remarkable.

The principal objective of our study was to demonstrate the improvements gained by different heuristics when they were applied to TSTs, and to compare them with the original TST. Additionally, we intended to investigate the performance of the different adjusting heuristics when the access distributions varied. The experiments compare the performance of the four data structures presented in the previous sections, namely the TST, the Splay-TST, the Rand-TST, and the Cond-TST. The comparison is made in terms of the running time measured in seconds, and the space requirements in MBs. For our comparisons we conducted four[4] sets of experiment on four benchmark data sets as explained below.

6.1 Data Sets

Four benchmark data sets were used in our experiments. Each data set was divided into two parts: a *dictionary* and *test documents*. The dictionary was the words or sequences that had to be stored in the TST. The test documents were the "files", for which the words were searched for in the corresponding dictionary.

The four dictionaries we used[5] were as follows:

- Dict[6]: This is a dictionary file used in the experiments done by Bentley and Sedgewick in [12].
- The *Webster's Unabridged* Dictionary: This dictionary was used by Clement *et. al.* [1, 24] to study the performance of different trie implementations including the TST. The alphabet size is 54 characters.
- 9-grams dictionary: This dictionary consisted of the unique words found in the Genomic data used by Heinz *et. al.* [22] for testing the Burst trie. The genomic data is a collection of nucleotide strings, each being typically, thousands of nucleotides in length. It is parsed into shorter strings by extracting n-grams of length

[4]The paper was initially submitted with tests from only two data sets. We are grateful to the Referees who recommended the additional tests. It certainly improved the quality.

[5]We requested the authors of the Burst trie paper [22] for the data that they used. Unfortunately, the data was copyright protected and we did not have the financial resources to purchase its license.

[6]The actual dictionary can be downloaded from http://www.cs.princeton.edu/rs/strings/dictwords.

9, which are utilized to locate regions where a longer inexact matched may be found. The alphabet size is four characters.

- NASA dictionary: The dictionary consisted of unique words extracted from the *NASA* file. This is a large data set that is freely available, and consists of a collection of file names accessed on some web servers[7].

The corresponding four test documents were:

- Dict documents: These were five artificially created documents from the dictionary described above. Each document was created with a large number of strings that followed a certain distribution. These documents were used to simulate five types of access distributions, namely the Zipf's, the exponential, the two families of the wedge distribution, and the uniform distribution.

- Herman Melville's novel *Moby Dick*: This complete text [1, 24] is available on the WWW. The average length of the words in *Moby Dick* was 5.4 characters.

- Genome[8]: The Genome document did not have the same skewed distribution that is typical of text, as in the case of the first two sets. It was fairly uniform, with even the rarest n-grams occurring hundreds of times, and did not even show much locality. This data earlier demonstrated a poor performance in the Burst trie structure [22]. The document consisted of the parsed 9-grams obtained from the genome data, and was used to test the case where the strings searched for were of equal lengths.

- NASA document: This document showed characteristics that were different from the English text and had strings of relatively larger lengths than the other data sets. As we shall see, the strings had the property that they shared common prefixes, which, in turn, gave a perfect TST when it was used to store strings of this type.

The statistics of these data sets are shown in Table 6.1, and snapshots of the different dictionaries used are given in Table 6.1. The first two data sets have very similar dictionaries, which are actually English words, and so we show only a snapshot for the dictionary of Dict data set. As mentioned earlier, since the data used in our experiments is entirely cache resident, all these sets, really, should considered to be of "moderate" size.

The data used in our experiments are considered moderate size, which are .

6.2 Methodology

We considered four sets of experiments. Each set corresponded to each of the data sets. In each experiment, we stored the dictionary in the TST, and the strings in the corresponding document were used in the search process. The words of the dictionary were stored in the order that they appeared in the dictionary. For each data set, 100 parallel experiments were done for a large number of accesses so that a statistically dependable ensemble average

[7]This dictionary can be obtained from: http://cg.scs.carleton.ca/ morin/teaching/tds/src/nasa.txt.gz. We are grateful to Prof. Pat Morin, from Carleton University, for providing us this source.

[8]This file could be obtained from: http://goanna.cs.rmit.edu.au/řsinha/resources/data/set6_genome.zip

	Dict	Moby	Genome	NASA
TST & Size of dictionary	232KB	944KB	2.75MB	548KB
Size of document	1.3GB	1.02GB	301MB	2.34GB
number of words in dictionary	25481	91479	262024	15700
number of words in document	150000000	185408000	31623000	78494851
min word length	1	1	9	1
max word length	22	21	9	99

Table 1: Statistics of the data set used in the experiments.

Dict	Genome	NASA
spite	ccacggacc	$/cgi-bin/imagemap/countdown70?181,275$
et	gactggcca	$/shuttle/missions/sts-70/sts-70-patch-small.gif$
I'll	ggattgaaa	$/shuttle/missions/sts-68/news/sts-68-mcc-11.txt$
disulfide	tgtttattg	$/shuttle/missions/sts-70/o-ring-problem.gif$
windbag	ttggtgaag	$/shuttle/missions/sts-70/movies/woodpecker.mpg$
Eleazar	cagcatgct	$/shuttle/resources/orbiters/endeavour.gif$
radiography	ttcaatcat	$/shuttle/missions/sts-68/news/sts-68-mcc-12.txt$
marmalade	acaacgaaa	$/shuttle/missions/sts-68/sts-68-patch-small.gif$
concatenate	acggtgcca	$/images/op-logo-small.gif$
rib	gtctccgtc	$/shuttle/missions/sts-73/mission-sts-73.html$
sore	taataagtc	$/shuttle/missions/sts-73/sts-73-patch-small.gif$
grantee	actgcgaaa	$/shuttle/missions/sts-71/images/KSC-95EC-0911.gif$
Jolla	atcctggtt	$/shuttle/technology/sts-newsref/sts_asm.html$
taper	atcgcctct	$/shuttle/technology/images/srb_mod_compare_6-small.gif$
Almaden	gtgaagtcg	$/shuttle/technology/images/srb_mod_compare_1-small.gif$
yearn	gtaatggtt	$/images/shuttle-patch-logo.gif$

Table 2: Snapshots of the strings in Dict, Genome and NASA data sets.

could be obtained. In this paper, we have only tested the schemes for cases of successful access operations, with the primary aim being that of demonstrating the improvements gained by different heuristics when applying them to TSTs themselves.

To obtain the simulated five documents for the first data set, the words of the dictionary were first randomly arranged in a second file, and then the resultant random file was used to generate a large search file for each distribution, and finally, the resultant search files were utilized to search each data structure in such a way that the same identical search was made for each of them. Subsequently, 150,000,000 accesses were performed on each tree, each accessed item being randomly chosen. The "randomness" of the accesses were specified by the following four types of probability distributions:

1. Zipf's distribution. In this case, the individual probabilities obey the following:

$$p_i = \frac{k_1}{i}, \text{ where } k_1 = \frac{1}{(\sum_{i=1}^{n} \frac{1}{i})}.$$

2. Exponential distribution. Here the individual probabilities obey:

$$p_i = \frac{k_2}{2^i}, \text{ where } k_2 = \frac{1}{(\sum_{i=1}^{n} \frac{1}{2^i})}.$$

3. Two types of wedge distributions. In both these cases $\{p_i\}$ satisfied :

$$p_i = k_3 \cdot (n - i + 1) \text{ where } k_3 = \frac{1}{(\sum_{i=1}^{n} i)}.$$

4. Uniform distribution. Here the individual probabilities obey:

$$p_i = \frac{1}{n}.$$

The difference between the two wedge distributions is that the first had the probabilities chosen in such a way that the items, when listed in order from the highest access probability to the lowest, were also in a lexicographic order from the smallest item to the largest. The second distribution was chosen such that the items had an inverse relationship between their probability mass orderings and their lexicographic orderings.

In all these cases, to make the documents realistic, the actual records are queried by the *exact* Zipf's, Exponential and Wedge distributions respectively, as defined by the above equations, whenever the precision of the machine architecture permitted it. However, as the index of the records increases (decreases in the case of the second type of wedge distribution), it is clear that the probability mass will keep decreasing. Clearly, beyond a certain limit, the probability of accessing a record will fall below the underflow limitation of the machine. Subsequent to this cut-off point, the distribution was approximated by setting all the remaining probabilities to have the smallest non-underflow value permitted in the exact distribution. We have thereby effectively rendered the distribution exact whenever possible, and forced it to be uniform thereafter as the index of the record increases (decreases in the case of the second type of wedge distribution).

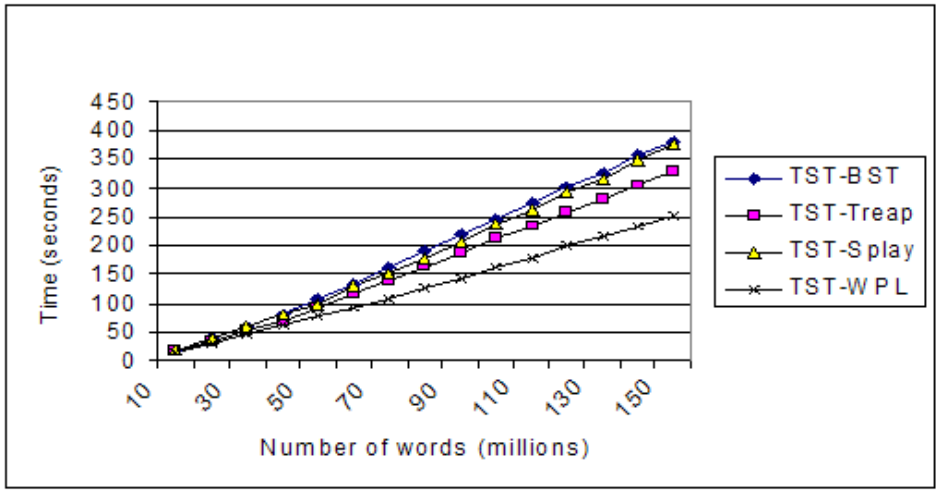

Figure 5: The accumulated time in seconds needed for all the data structures studied for various numbers of words, when the *Dict* dictionary was used, and the document was skewed by the Zipf's distribution.

The other three sets of experiments compared the performance of the data structures on the other three real documents. First, the words in each dictionary were inserted (as above) as they appeared in the dictionary on each data structure. Then, against this dictionary, the words in the corresponding document were searched for in their order of occurrence in the document. Again, 100 parallel experiments are done for a large number of accesses, so that a statistically dependable ensemble average could be obtained.

6.3 Analysis of Results

We now present the results obtained for each data set. The results can be summarized according to each data set as follows:

- *Dict data set*: Figures 5 through 8 shows the results for the different simulated documents. Each figure shows the accumulated time needed for successful search operations for different number of words, and for different probability distributions. From the results we see that the Cond-TST is the best. For example, for 140 million search operations with Zipf's distribution, the Cond-TST took 235 seconds, the Splay-TST took 347 seconds, and the Rand-TST required 306 seconds. The original TST (without any adjusting) took 356 seconds[9]. Observe Figure 9 which shows the time needed for the last 50 million successful searches.

- *Moby data set*: Figure 10 shows the results for the second set of experiments which was done on real-life data set. The results confirm that the Cond-TST is the best with respect to the data structures used in the comparison. For example, for 140 million search operations, the Cond-TST took 159 seconds, the Splay-TST took 261 seconds, and the Rand-TST required 225 seconds. The original TST (without any adjusting) took 173 seconds.

[9]The time shown here may be seen to be larger than the time give by the Burst trie. The time depends on the data and the implementation. Our main goal in the paper was to see the effect of adjusting heuristics on the TSTs, and all the heuristics were built using the same implementation for the basic TST. Also, the time for parsing and printing the results are included with the time presented in the results.

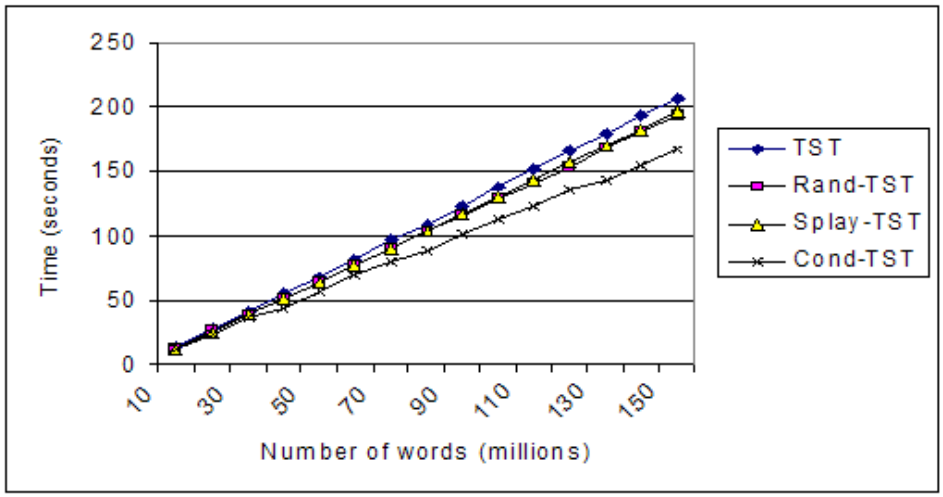

Figure 6: The accumulated time in seconds needed for all the data structures studied for various numbers of words, when the *Dict* dictionary was used, and the document was skewed by the Exponential distribution.

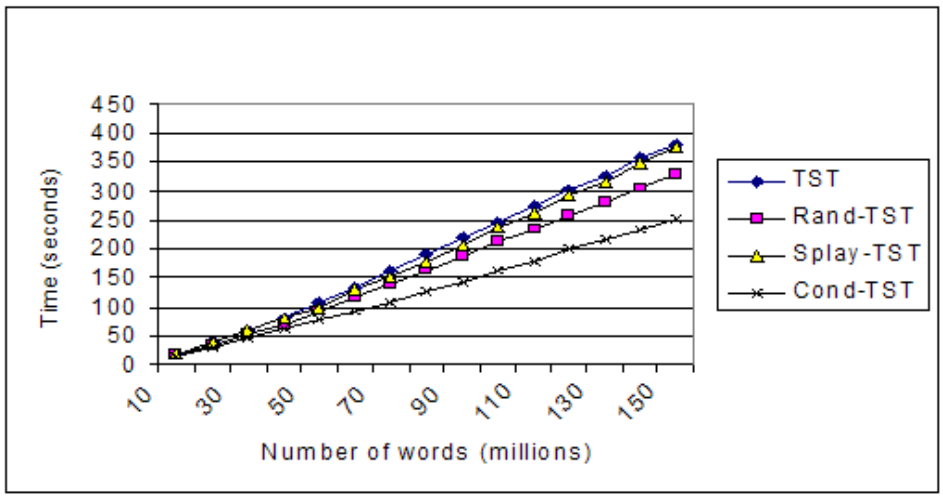

Figure 7: The accumulated time in seconds needed for all the data structures studied for various numbers of words, when the *Dict* dictionary was used, and the document was skewed by the wedge distribution.

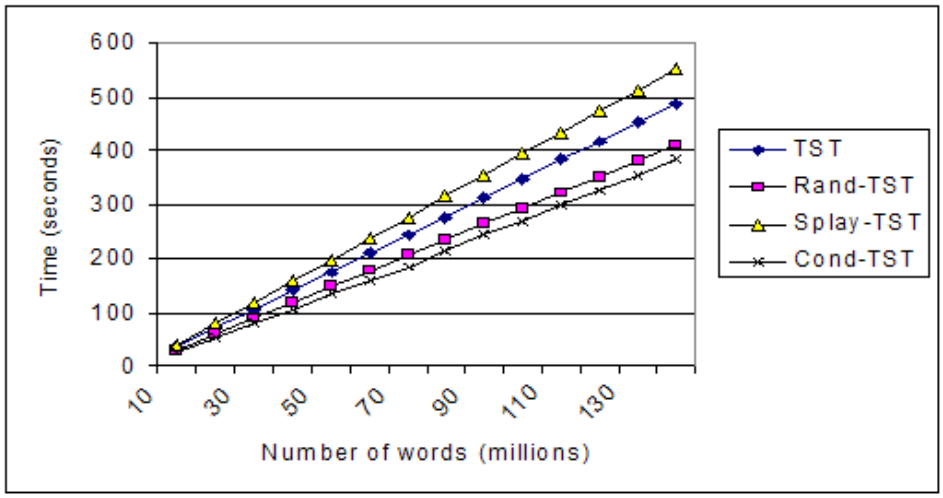

Figure 8: The accumulated time in seconds needed for all the data structures studied for various numbers of words, when the *Dict* dictionary was used, and the document was skewed by the Uniform distribution.

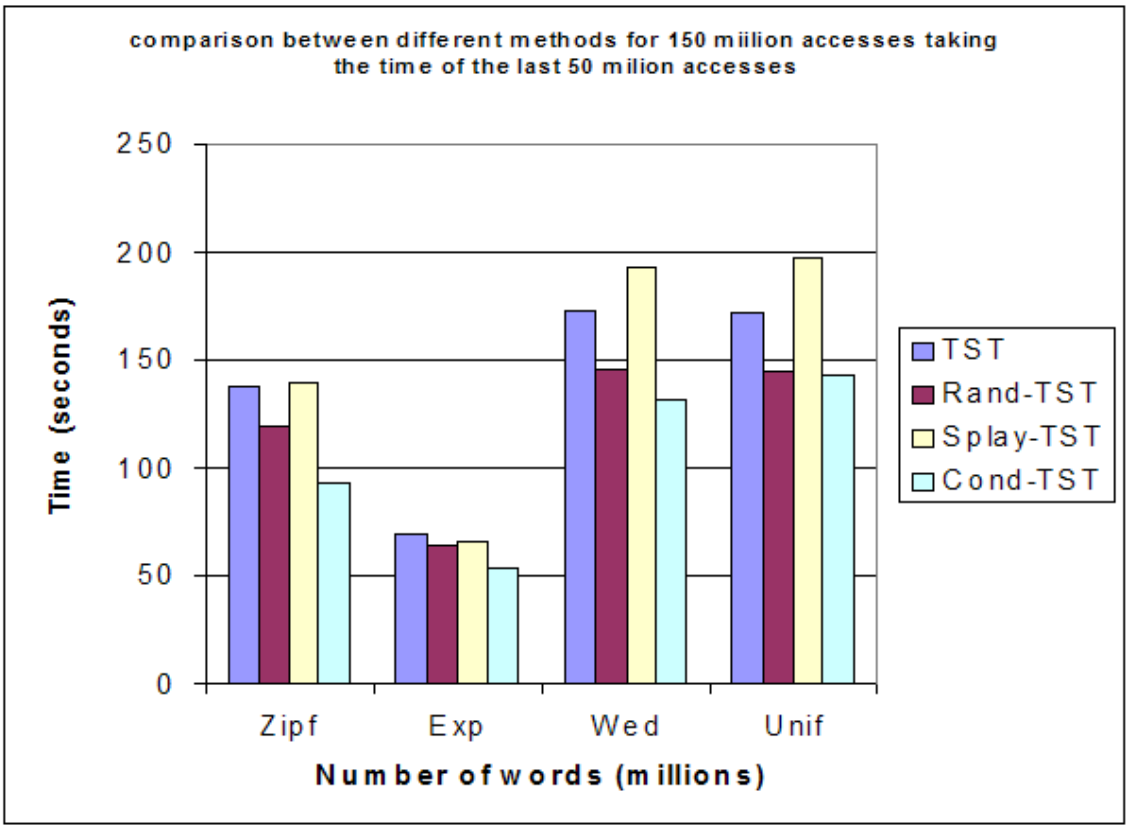

Figure 9: A comparison between the different data structures when various *Dict* documents were accessed against the *Dict* dictionary for 150 million accesses, when the ensemble average of the time required for the last 50 million accesses was measured.

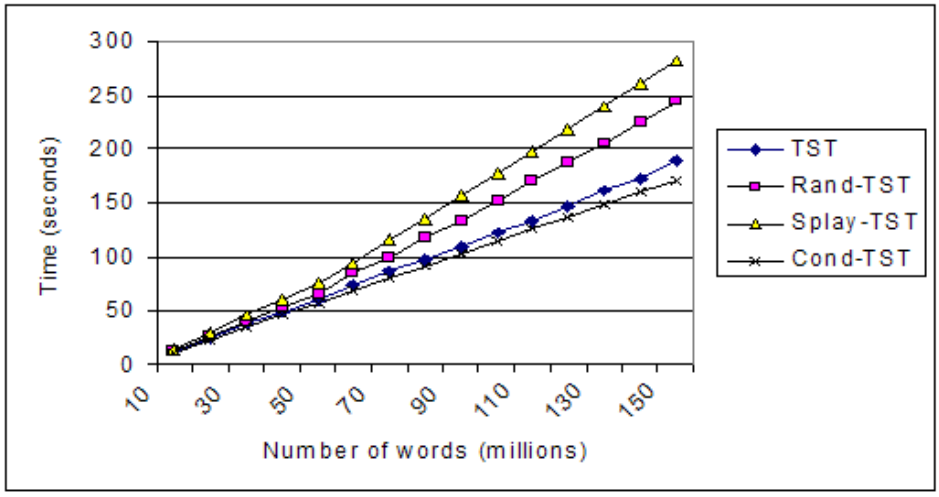

Figure 10: The accumulated time in seconds needed for all the data structures studied for various numbers of words in *Moby Dick* was searched against the *Webster's Unabridged Dictionary* in the order of occurrence in the novel.

- *Genome data set*: In this case, the cond-TST gave a time comparable with respect to the TST. This data set gave a bad performance for the Burst trie with respect to the TST, and that was why we used this data set to see show how our heuristic compared to the TST. Figure 11 presents the results, from which we can see that, for 30 million search operations, the Cond-TST took 88 seconds, the Splay-TST took 117 seconds, and the Rand-TST required 88 seconds. The original TST (without any adjusting) took 85 seconds.

- *NASA data set*: This data set was used to test the case when the data itself was stored in a manner favorable to the TST, and for which the data itself led to the best possible TSTs. Since the strings had so many common prefixes, the dictionary led to a nearly optimal TST. Figure 12 shows that result of comparing the new heuristics with the basic TST. The results show that the Cond-TST has the ability to learn when to stop "self-adjusting", and to learn that for some data that tree will not need adjusting at all!. In this case, the negative side effect for the splaying and randomizing the TST become apparent, because they disturb the original data so as to lead to a TST that is inferior to the one originally stored. For example, for 70 million search operations, the Cond-TST took 190 seconds, the Splay-TST took 275 seconds, and the Rand-TST required 270 seconds. The original TST (without any adjusting) took 206 seconds.

From the results presented above, we see that the Cond-TST has the ability to learn the characteristics of the data set and to adjust the TST according to these characteristics. It overcomes the side effect of the Burst tries which needs parameter adjustments, and the disadvantage of the splaying heuristic which takes a very large time for adjusting the structure even when it is not needed. Also, the Cond-TST is much better than the randomized heuristics which are only suitable for uniformly distributed data sets. The *minor* disadvantage of the Cond-TST is the marginal extra space requirement needed for storing the access information. This disadvantage is minimal (i.e., it requires only linear extra space) considering the availability of inexpensive memory in present-day systems. The space requirements are summarized in Table 6.3.

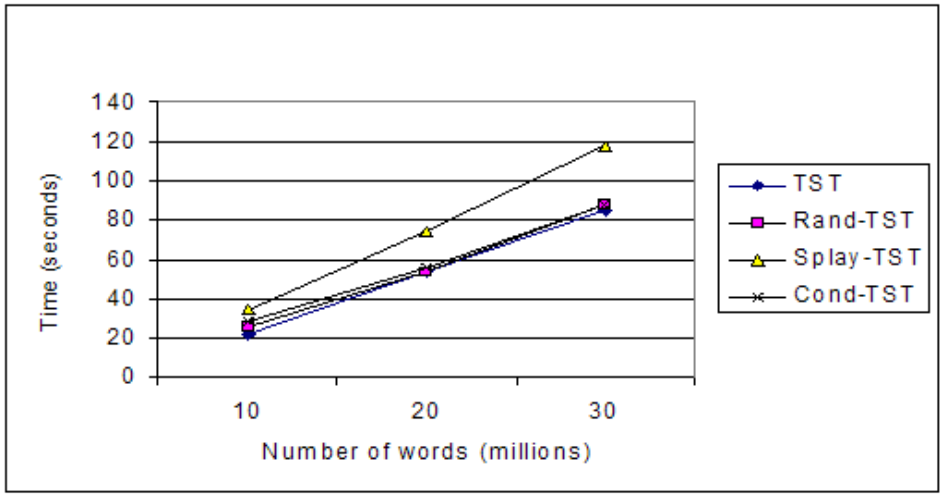

Figure 11: The accumulated time in seconds needed for all the data structures studied for various numbers of words in the *Genome* document when searched against the *Genome* dictionary in the order of occurrence in the document.

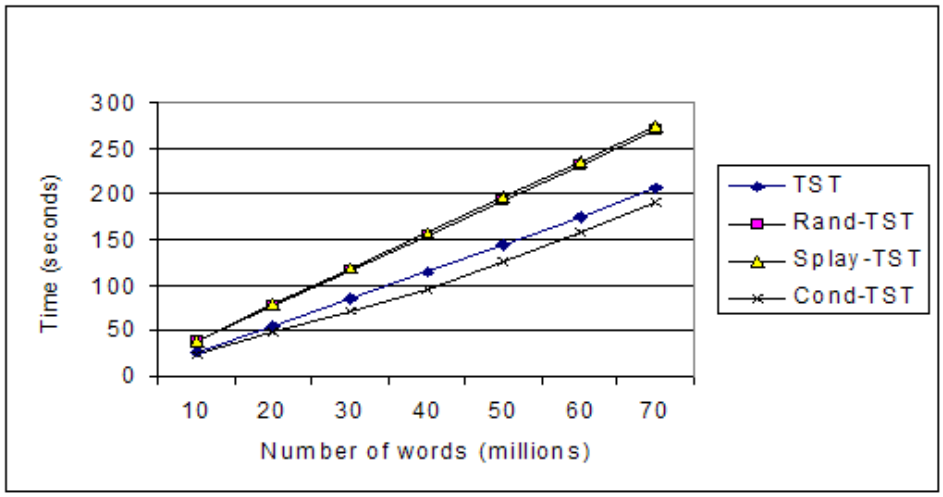

Figure 12: The accumulated time in seconds needed for all the data structures studied for various numbers of words in the *NASA* document when searched against the *NASA* dictionary in the order of occurrence in the document.

Heuristics	Dict	Moby	Genome	NASA
TST & Splay-TST	3183960	11053536	20967168	2724432
Rand-TST & Cond-TST	3714620	12895792	24461696	3178504

Table 3: Space required, in KB, by the different data structures for the different dictionaries used.

7 More Related Work

The literature also records various schemes which adaptively restructure the BST with the aid of additional memory locations. The two outstanding schemes in this connection are the **monotonic tree** and Mehlhorn's D-Tree [14, 25]. The monotonic tree is a dynamic version of a tree structuring method suggested by Knuth [25] as a means to structure a nearly-optimal static tree. The static monotonic tree is arranged such that the most probable key is the root of the tree, and the subtrees are recursively ordered in the same manner. The static version of this scheme behaves quite poorly [27]. Walker and Gotlieb [37] have presented simulation results for static monotonic trees, and these results also indicate that this strategy behaves quite poorly when compared to the other static trees known in the literature.

Bitner suggested a dynamic version of this scheme [14], which could be used in the scenario when the access probabilities are not known *a priori*. Each record has one extra memory location, which "counts" the number of accesses made to it. The reorganization of the tree after an access is then very straightforward. When a record is accessed, its counter is incremented, and then the record is rotated upwards in the tree until it becomes the root of the tree, or it has a parent with a higher frequency count than itself. Over a long enough sequence of accesses this will, by the law of large numbers, converge to the arrangement described by the static monotonic tree. Although this scheme is intuitively appealing, Bitner determined bounds for the cost of a monotonic tree, and showed that it is largely dependent on the entropy, $H(S)$ of the probability distribution of the keys. If $H(S)$ is small, then the monotonic tree is nearly optimal; but if $H(S)$ is large, it will behave quite poorly. Bitner also stated a result, that proved that the expected entropy of a randomly chosen probability distribution is $log(N) - ln2$, which is nearly the maximum entropy attainable. He concluded from this that, on the average, the monotonic tree scheme will behave poorly . The experimental results [32] support this viewpoint.

As opposed to the above, Mehlhorn's D-tree is a BST scheme significantly different from the Monotonic Tree. At every node the D-tree maintains counters which record the weights of the two subtrees at the node, where the weight of a subtree is defined as the number of leaves in that subtree. D-trees permit multiple leaves for the same object, for indeed, each time an object is searched, the number of leaves referring to that object is increased by unity. The searching technique in the scheme ensures that all searches will be properly directed to corresponding objects at the leaf level. In any actual implementation of the D-tree, both search-time and space can be saved by coalescing a significant number of leaves into a single "super-leaf". The D-tree uses the weights of the two children subtrees at each node as the input parameters to a balancing function which provides a numeric measure for how "balanced" the subtree itself is. On executing a search for a record, if the balancing function of any node in the tree exceeds this threshold, single and/or double rotations are executed at strategic nodes along the search path which ensure that the D-tree remains balanced after the rotations are executed. This scheme is closely related to the $BB[\alpha]$-trees also described by Mehlhorn. The latter uses the weight (defined as in the D-tree) of the subtrees of a tree as a method of quantifying how balanced the tree is. A node in the $BB[\alpha]$-tree scheme is considered to be balanced if the balancing function, which takes the weight of the left and right subtrees as parameters, returns a value which is bounded by the variable α. If the tree is reckoned to be "unbalanced", just as in the case of the

AVL-tree [25], the $BB[\alpha]$-tree reorganizes the nodes using single or double rotations.

All these techniques that have been used previously for BSTs can now be applied to TSTs. Such a study is open. In this study, we have chosen three techniques that were previously demonstrated to be among the most superior ones when applied to BSTs. This has motivated us to use them for TSTs. We have also shown that their performance is superior (with respect to time) when compared to TSTs without any balancing heuristics.

8 Conclusion

In this paper we have applied two self-adjusting heuristics, that have been previously used for BSTs, to TSTs. These heuristics are, namely, the **splaying** (Splay-TST) and the **Conditional Rotation** (Cond-TST) heuristics. We have also applied another balancing strategy used for BSTs, to TSTs, which is the basis for the **Randomized** search trees, or Treaps, (Rand-TST). The formal properties of the data structure have been stated. Numerous experiments were done to investigate the performance of the different adjusting heuristics when applied to TSTs so as to see how they perform in the context of different probability distributions characterizing various benchmark data sets. The comparison (made on "moderate-size" data sets) was measured in terms of the running time, quantified in seconds. From the results we unequivocally conclude that the Cond-TST is the best scheme that can be used to improve the performance of the original TST and it has the ability to learn when to stop "self-adjusting" whenever the tree will not need further adjustments. The improvement shown in the paper is with respect to the original form of the TST.

References

[1] A. Acharya, H. Zhu, and K. Shen. (1999) Adaptive algorithms for cache-efficient trie search. *ACM and SIAM Workshop on Algorithm Engineering and Experimentation ALENEX'99*, Baltimore, Maryland, January 15-16.

[2] G. M. Adel'son-Velski'i and E.M. Landis. (1962) An algorithm for the organization of information. *Sov. Math. Dokl.*, vol. 3, pp. 1259–1262.

[3] S. Albers and J. Westbrook. (1998) Self-organizing data structures. *Amos Fiat and Gerhard Woeginger*, pp. 31–51, Online Algorithms: The State of the Art, Springer-Verlag.

[4] B. Allen and I. Munro. (1978) Self-organizing binary search trees. *Journal of ACM*, vol. 25, pp. 526–535.

[5] A. Andersson. (1999) General balanced trees. *Journal of Algorithms*, vol. 30, pp. 1–28.

[6] A. Andersson and S. Nilsson. (1993) Improved behaviour of tries by adaptive branching. *Information Processing Letters*, vol. 46. no. 6, pp. 296–300.

[7] J. Aoe, K. Morimoto, and T. Sato. (1992) An efficient implementation of trie structures. *Software-Practice and Experience*, vol. 22, no. 9, pp. 695–721.

[8] C. Aragon and R. Seidel. (1989) Randomized search trees. *Proc. 30th Symp. on Foundations of Computer Science*, pp. 540–545.

[9] D.M. Arnow and A.M. Tenenbaum. (1982) An investigation of the move-ahead-k rules. In *Congressus Numerantium, Proceedings of the Thirteenth Southeastern Conference on Combinatorics, Graph Theory and Computing*, Florida, February, pp. 47–65.

[10] 2000 / 334 (2000) Fast and space efficient trie searches. Ecole Polytechnique Fdrale de Lausanne, EPFL Swtzerland.

[11] (2001) P. Bagwell. Ideal hash trees. Es Grands Champs, 1195-Dully, Switzerland.

[12] J. Bentley and R. Sedgewick. (1997) Fast algorithms for sorting and searching strings. *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, New Orleans, pp. 360-369.

[13] J. Bentley and R. Sedgewick. (1998) Ternary search trees. *Dr. Dobb's Journal*.

[14] J.R. Bitner. (1979) Heuristics that dynamically organize data structures. *SIAM J.Comput.*, vol. 8, pp. 82–110.

[15] R. Brandais. (1959) File searching using variable length keys. *In Proceedings of Western Joint Computer Conference*, vo. 15, pp. 295–298.

[16] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. (1990) *Introduction to Algorithms*. The MIT Press, Cambridge, MA.

[17] L. Devroye. (1988) Applications of the theory of records in the study of random trees. *Acta Informatica*, vol. 26, pp. 123–130.

[18] E. Fredkin. Trie memory. (1960) *Communication of the ACM*, vol. 3, pp. 490–499.

[19] I. Galperin and R.L. Rivest. (1993) Scapegoat trees. *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, Austin, Texas, United States, pp. 165–174.

[20] G.H. Gonnet, J.I. Munro, and H. Suwanda. (1981) Exegesis of self-organizing linear search. *SIAM J.Comput.*, vol. 10, pp. 613–637.

[21] J. Zobel H.E. Williams and S. Heinz. (2001) Self-adjusting trees in practice for large text collections. *Software Practice and Experience*, vol. 31, no. 10, pp. 925–939.

[22] S. Heinz, J. Zobel, and H. Williams. (2002) Burst tries: A fast, efficient data structure for string keys. *ACM Transactions on Information Systems*, vol. 20, no. 2, pp. 192–223.

[23] H.J. Hester and D.S. Herberger. (1976) Self-organizing linear learch. *ACM Computing Surveys*, pp. 295–311.

[24] P. Flajolet J. Clement and B. Vallee. (1998) The analysis of hybrid trie structures. *Proc. Annual ACM-SIAM Symp. on Discrete Algorithms*, San Francisco, California, pp. 531–539.

[25] D.E. Knuth. (1973) *The Art of Computer Programming*, volume 3. Addison-Wesley, Reading, Ma.

[26] J. McCabe. (1965) On serial files with relocatable records. *Operations Research*, vol. 12, pp. 609–618.

[27] K. Mehlhorn. (1975) Nearly optimal binary search trees. *Acta Informatica*, vol. 5, pp. 287–295.

[28] D. R. Morrison. (1968) Patricia: a practical algorithm to retrieve information coded in alphanumeric. *Journal of ACM*, vol. 15, no. 4, pp. 514–271.

[29] B.J. Oommen and E.R. Hansen. (1987) List organizing strategies using stochastic move-to-front and stochastic move-to-rear operations. *SIAM Journal of Computing*, vol. 16, pp. 705–716.

[30] B.J. Oommen, E.R. Hansen, and J.I. Munro. (1990) Deterministic optimal and expedient move-to-rear list organizing strategies. *Theoretical Computer Science*, vol. 74, pp. 183–197.

[31] R.L. Rivest. (1976) On self-organizing sequential search heuristics. *Comm. ACM*, vol. 19, pp, 63–67.

[32] B.J. Oommen R.P. Cheethman and D.T.H. Ng. (1993) Adaptive structuring of binary search trees using conditional rotations. *IEEE Transactions on knowledge and data engineering*, vo. 5, no. 4, pp. 695–704.

[33] M. Sherk. (1995) Self adjusting k-ary search trees. *Journal of Algorithms*, vol. 19, no. 1, pp. 25-44.

[34] D.D. Sleator and R.E. Tarjan. (1985) Self-adjusting binary search trees. *Journal of the ACM*, vol. 32, no. 3, pp. 652–686.

[35] W. Szpankowski. (2001) *Average Case Analysis of Algorithms on Sequences*. John Wiley and Sons, New York.

[36] J. Vuillemin. (1980) A unifying look at data structures. *Journal of the ACM*, vol. 23, no. 4, pp. 229–239.

[37] W.A. Walker and C.C. Gotlieb. (1972) A top-down algorithm for constructing nearly optimal lexicographical trees. *Graph Theory and Computing*, Academic Press, New York.

[38] I. H. Witten and T. C. Bell. (1990) Source models for natural language text. *International Journal on Man Machine Studies*, vol. 32, pp. 545–579.

Appendix

In this Appendix we present the different algorithms used for the access operation of the TST when the corresponding self adjusting and balancing heuristics discussed in the body of the paper are invoked. The presentation of the algorithms follows the style used in the standard MIT-Press text book [16].

The pseudo-code for the access operation supported by the Splay-TST is given in Algorithms 1 and 2. The pseudo-code for the insertion operation supported by the Rand-TST is given in Algorithms 3 and 4. The pseudo-code for the space-optimized version of the conditional rotation heuristic when applied to the TST, T, is given in Algorithm 5.

Algorithm 1 Algorithm Splay-TST-Access

Input: A Ternary search trie T and a string s to be searched for whose characters are s_i. The end of string s is marked with \$.
Output: The restructured tree T', and the record A_i.
Method:

```
1:  if T is empty then
2:     Return NULL
3:  end if
4:  if T ↑.key = s_i = $ then
5:     perform Splay on node i of T
6:     Return record A_i
7:  else
8:     if T ↑.key = s_i then
9:        perform Splay-TST-Access on T ↑.MiddleChild and with character s_{i+1}
10:    else
11:       if T ↑.key < s_i then
12:          perform Splay-TST-Access on T ↑.RightChild
13:       else
14:          perform Splay-TST-Access on T ↑.LeftChild
15:       end if
16:    end if
17: end if
18: End Algorithm Splay_Tree_Access
```

Algorithm 2 Splay

Input: A Ternary Search Tree T and a node i in T.
Output: A restructured tree T' with i as root.
Method:

```
1: if i is the root of T then
2:    Return T
3: else
4:    if i the root of the current BST then
5:       Splay P(i)
6:    else
7:       if i is a left child then
8:          if P(i) is the root of T then {Case 1}
9:             Right_Rotate P(i)
10:            Return T
11:         else
12:            if P(i) is the root of current BST then {Case 1}
13:               Right_Rotate P(i)
14:               Splay P(P(i))
15:            else
16:               if P(i) is a left child then {Case 2}
17:                  Right_Rotate the parent of P(i)
18:                  Right_Rotate P(i)
19:               else
20:                  Right_Rotate P(i) {Case 3}
21:                  Left_Rotate P(i)
22:               end if
23:               Splay i' which is the post-rotational node i
24:            end if
25:         end if
26:      else {node i is a right child}
27:         if P(i) is the root of T then {CASE 1}
28:            Left_Rotate P(i)
29:            Return T
30:         else
31:            if P(i) the root of the current BST then {Case 1}
32:               Left_Rotate P(i)
33:               Splay the parent of P(i)
34:            else
35:               if P(i) is a Right child then {Case 2}
36:                  Left_Rotate the parent of P(i)
37:                  Left_Rotate P(i)
38:               else
39:                  Left_Rotate P(i) {Case 3}
40:                  Right_Rotate P(i)
41:               end if
42:               Splay i' which is the post-rotational node i
43:            end if
44:         end if
45:      end if
46:   end if
47: end if
48: End Algorithm Splay
```

Algorithm 3 Rand-TST-Insert

Input: A Ternary search trie T and a string s to be inserted for whose characters are s_i. The end of string s is marked with \$.
Output: The restructured tree T'.
Method:

1: **if** T is empty **then**
2: add new node T with key $= s_i$ with new random priority value
3: **end if**
4: **if** $T\uparrow$.key $= s_i =$ \$ **then**
5: we have finished
6: **else**
7: **if** $T\uparrow$.key $= s_i$ **then**
8: {we don't need Rebalance here}
9: **if** $T\uparrow$.MiddleChild = NULL **then**
10: $T\uparrow$.MiddleChild $\leftarrow$ add new node T with key $= s_i$ with new random priority value
11: **end if**
12: $T\uparrow$.MiddleChild $\leftarrow$ perform Rand-TST-Insert on $T\uparrow$.MiddleChild but with s_{i+1}
13: **else**
14: **if** $T\uparrow$.key $< s_i$ **then**
15: n$\leftarrow T\uparrow$.RightChild
16: **if** $T\uparrow$.RightChild = NULL **then**
17: $T\uparrow$.RightChild $\leftarrow$ add new node T with key $= s_i$ with new random priority value
18: n $\leftarrow T\uparrow$.RightChild
19: Perform Rebalance on $T\uparrow$.RightChild
20: **end if**
21: perform Rand-TST-Insert on n but with s_{i+1}
22: **else**
23: n$\leftarrow T\uparrow$.LeftChild
24: **if** $T\uparrow$.LeftChild = NULL **then**
25: $T\uparrow$.LeftChild $\leftarrow$ add new node T with key $= s_i$ with new random priority value
26: n $\leftarrow T\uparrow$.LeftChild
27: Perform Rebalance on $T\uparrow$.LeftChild
28: **end if**
29: perform Rand-TST-Insert on n but with s_{i+1}
30: **end if**
31: **end if**
32: **end if**
33: **Return** T
34: **End Algorithm Rand-TST-Insert**

Algorithm 4 Rebalance

Input: A node n in a Ternary search trie T to be rebalanced
Output: The restructured tree T'.
Method:

1: **if** $P(n) \neq NULL$ and $P(n)\uparrow .periority > n\uparrow .periority$ and n is not a middle child **then**
2: **if** n is a Right child **then**
3: Left_Rotate n
4: **else**
5: Right_Rotate n
6: **end if**
7: **end if**
8: **End Algorithm Rebalance**

Algorithm 5 Optimized-Cond-TST

Input: A Ternary Search Trie T and a search string S assumed to be in T with characters s_i.
Output: The restructured tree T', and a pointer to record A stored at the end of the string S.
Method:

1: **if** T is empty **then**
2: **Return** NULL
3: **end if**
4: $\text{J} \leftarrow T$
5: $\tau_j \leftarrow \tau_j + 1$ {update τ for the present node}
6: **if** $s_j = s_i$ **then**
7: { We reached the end of the current BST}
8: **if** node j is a left child in the current BST **then**
9: $\psi_j \leftarrow 2\tau_j - \tau_{jR} - \tau_{P(j)}$
10: **else**
11: **if** node j is a right child in the current BST **then**
12: $\psi_j \leftarrow 2\tau_j - \tau_{jL} - \tau_{P(j)}$
13: **else**
14: {j is also the root of current BST}
15: $\psi_j \leftarrow 0$
16: **if** $s_{i+1} \neq$ NULL **then**
17: perform Optimized-Cond-TST on $j\uparrow$.MiddleChild but with s_{i+1}
18: **end if**
19: **end if**
20: **end if**
21: **if** $\psi_j > 0$ **then**
22: rotate node j upwards
23: recalculate $\tau_j, \tau_{P(j)}$
24: **end if**
25: **if** $s_j = s_i = \$$ **then**
26: {We reached the end of the string}
27: **Return** record A_i
28: **end if**
29: **else**
30: **if** $s_j < s_i$ **then**
31: {search the subtrees}
32: perform Optimized-Cond-TST on $j\uparrow$.RightChild
33: **else**
34: perform Optimized-Cond-TST on $j\uparrow$.LeftChild
35: **end if**
36: **end if**
37: **END Algorithm Optimized-Cond-TST**