AN EXPERIMENTAL STUDY OF DYNAMIC BIASED SKIP LISTS

by

Yiqun Zhao

Submitted in partial fulfillment of the requirements
for the degree of Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
August 2017

Table of Contents

List of Tables

List of Figures

Abstract

Skip lists allow search, insert, and delete operations in $O\big(\log n\big)$ average time. However, they show a lack of efficiency when dealing with data exhibiting locality of reference, meaning the keys are searched or updated non-uniformly. The dynamic biased skip list $\big(\text{DBSL}\big)$ [28, 29] allows $O(\log r(k))$ amortized time per operation, where $r(k)$ is the number of distinct keys accessed after k was last accessed.

The performance of DBSL has not been fully evaluated. We implemented DBSL and compared it with move-to-front lists, red-black trees and splay trees, using real-world and synthetic data. Our results show that when a highly biased data sequence contains delete operations, DBSL is more efficient compared to the others. When the degree of bias is moderate, DBSL is not competitive against splay trees, but is more efficient than red-black trees and skip lists. When the data sequence is less biased, DBSL is the least efficient.

List of Abbreviations and Symbols Used

$r(k)$	Number of distinct keys accessed after k was last accessed
$r_{max}(k)$	Maximum rank which k achieves in its lifespan
m	Length of a sequence or number of accesses of a dataset
n	Number of distinct keys in a sequence or a dataset
MFL	Move-to-front list
ST	Splay tree
R-ST	Randomized splay tree
W-ST	W-splay tree
RBT	Red-black tree
SL	Skip list
DBSL	Dynamic biased skip list
H-DBSL	Dynamic biased skip list with the hybrid search algorithm described previously

Acknowledgements

I would like to express my gratitude to my supervisor, Dr. Meng He, for his generous guidance, encouragement and patience throughout my program. This work could not be done without his valuable expertise and comprehensive advices.

I am sincerely grateful to my thesis committee, Dr. Norbert Zeh and Dr.Nauzer Kalyaniwalla, for reading my thesis and giving me insightful comments.

I would like to thank my father for his unceasing encouragement, and thank my mother for her always being supportive while alive.

Chapter 1

Introduction

A dynamic table data structure maintains a set of keys from a totally ordered universe to support the following operations:

- **Search(k)**: Look for a certain key k in the data structure. If k exists, return a pointer to its location. Otherwise, return the position in which k should be inserted.
- **Insert(k)**: Add key k into the data structure. This operation is completed only if k does not exist in the current data structure.
- **Delete(k)**: Remove key k from the current data structure. If k is not maintained by the container, do nothing.

The problem of designing an efficient dynamic table is fundamental in computer science; therefore, many data structures have been designed. They include classic data structures such as AVL trees [3], skip lists [71], BB[x] [60], (a,b)-trees [16], red-black trees [14] and splay trees [76]. These data structures support search, insert and delete functions in logarithmic time. However, many data structures do not consider that, in many applications, requests often show locality of reference. This means that within any time interval, only a small portion of data is accessed, some keys are accessed more frequently while other keys are accessed very rarely. Locality of reference exists in many applications such as those for memory, cache and buffer management [23, 74, 75, 37, 47], and also in those for requests between two web servers [9, 6, 18]. Locality of reference can be used to predict future behaviors [44, 13, 32]. Let us consider the example of handling requests between two web servers. Since the reference streams exhibit locality, caching techniques can be used to find objects that are more likely to be accessed again in a near future, thus the communication time between the two servers can be reduced. Therefore, exploiting locality of reference has been widely applied as a programming optimization technique. Extensive theories

have been researched to take query patterns into account to create data structures that are more time efficient, in either the amortized or the worst-case sense, than those only providing $O(\log n)$[1] time per search operation, where n is the total number of distinct keys in the sequence.

In practice, a sequence of operations on the data often shows locality of reference. Special biased data structures are designed to deal with such a biased request sequence. The most frequently needed keys can be accessed faster than those that are rarely needed. Given a sequence of operation on a data structure, the *working set* [76] of a key k is defined as a set of distinct keys in the data structure that have been accessed since last time k was accessed. Let the *rank* of k represent the size of k's working set, denoted as $r(k)$. A data structure is recognized as a data structure with the *working set property* if it can access the key k in $O(\log r(k))$ time. One of the most well-known data structure that has the working set property is the splay tree [76]. Some modified splay trees also satisfy the working set property, such as the R-splay tree [5], and the W-splay tree [4].

In this paper, we study an extension of the skip list which is the dynamic biased skip list [28] and conduct experiments to evaluate its performance. The dynamic biased skip list can be used to exploit bias and access an existing key k in $O\big(\log r\big(k\big)\big)$ time, which satisfies the working set property. Even though a splay tree supports an amortized $O\big(\log r\big(k\big)\big)$ time for accessing an existing key k, an insertion or a deletion in a splay tree takes $O\big(\log n\big)$ time per operation. In the dynamic biased skip list, an updating operation can be done in $O\big(\log r_{\max}\big(k\big)\big)$ amortized time, where $r_{\max}\big(k\big)$ is the maximum rank that the key k achieves during its lifetime.

1.1 Dynamic Data Structures

The study of a dynamic dictionary is a fundamental topic in computer science. There are a lot of data structures that support dynamic operations in logarithmic time. In this section, we will briefly discuss related dynamic data structures.

[1] Throughout this thesis, $\log n$ represents $\log_2 n$.

1.1.1 Move-to-front Lists

A move-to-front list is a dynamic linked list. When a key is found, it will be reallocated as the new head of the list, so it can be found quickly in a short while. Searching for an existing key k takes $O\big(r\big(k\big)\big)$ time. If the access sequence shows considerable bias, the searching performance can be improved to $O\big(1\big)$. Although a move-to-front list does not have the working set property and suffers an $O(n)$ worst-case operation time, it is still a favourable choice for biased data patterns because the most popular keys stay near the head and are quickly accessible.

1.1.2 Red-black Trees

A red-black tree is a self-balancing binary tree in which each node is holding additional color information and maintains some invariants on colors to keep balanced. After an insertion or a deletion, the tree will adjust the colors of related nodes to maintain the color requirements.

A red-black tree supports each search, insert and delete operation in $O\big(\log n\big)$ worst-case time, which is better than most binary trees that suffer an $O(n)$ worst-case time per operation. Since red-black trees work particularly well in handling frequent updates, they are widely applied in Linux kernels or as a mapping storage [57, 19, 55].

1.1.3 Splay Trees

A splay tree, invented by Sleator and Tarjan [76], is a self-adjusting binary tree in which each node does not keeps extra information for balancing. A splay tree uses a special scheme, called splaying, to restructure itself. It moves the most recently accessed node to the root of the tree with a series of specific rotations. Such reconstruction applies in each operation. Consequently, the most popular keys are more likely to stay near the root and can be accessed more quickly in the future. Splay trees support search, insert and delete operations in amortized $O\big(\log n\big)$ time. Moreover, the working set theorem for splay trees states that the cost of accessing an existing key in a splay tree is $O(\log r(k))$ amortized.

The splay tree's working set property inspired many researchers to extend the

splay tree model. An randomized splay tree [5] uses a random parameter, p, called splay probability, to decide whether to splay or not. It supports an $O\big(\log r\big(k\big)\big)$ amortized expected access time. In practice, it can achieve an improvement of up to 25% over the original deterministic splay tree when the access sequence is generated by *uniform distribution* and *Zipf's distribution* [70]. However, such an improvement becomes insignificant when the random access sequence has a high level of locality of reference. Another extension of the splay tree is called W-splay tree [4]. A W-splay algorithm uses a special method to calculate the cost of splaying, which takes the current working set into consideration and splays only when necessary. The W-splay tree maintains $O\big(\log r\big(k\big)\big)$ amortized time per search operation.

1.1.4 Skip Lists

Skip lists, invented by Pugh [71], are fascinating. It is built in layers and each layer is an ordered linked list. The lowest level contains all the keys. The next higher layer contains keys that are copied up from a lower level with a certain preset probability p, where $0 < p < 1$. This procedure is repeated until a level with no keys copied from a lower layer is encountered. A skip list supports an expected logarithmic time on accessing and updating, and is easy to be implemented. Such characteristics make skip lists widely studied. Papadakis et al.[64] theoretically proved the upper bound of each search, insertion and deletion operation is expected in $O\big(\log n + 1\big)$. Etim [31] executed experiments between skip list, splay trees, non-recursive AVL trees and recursive 2-3 trees, and found that skip lists are as fast as optimized balanced trees and faster than randomized trees. In 1990, Pugh [72] proposed an optimization to reduce the number of key comparisons in a sequential search algorithm. In the optimized search algorithm, the target key is compared against any existing key no more than once so that redundant comparisons are avoided. Pugh showed the upper bound of the expected successful search cost is $O(\log_{\frac{1}{p}} n)$, using this optimized search algorithm. Kirschenhofer et al. [48] further analyzed the search cost for both successful and unsuccessful searches. It is also pointed out in [72] that skip lists support not only search, insert, delete operations, but also the following three operations:

- JOIN$\big(S_1$,$S_2\big)$. Given two ordered sets S_1 and S_2, where all the keys in S_1 are smaller than those in S_2. Merge to get a new set S so that $S = S_1 \cup S_2$.

Merging two skip lists of sizes n and m respectively takes $O(\log \max(n, m))$ expected time.

- SPLIT$\big(S,\, k\big)$. Given an ordered set S, split S into two subsets S_1 and S_2, where all keys in S_1 are smaller than or equal to key k and those in S_2 are larger than key k. Splitting one skip lists of size n takes $O(\log n)$ expected time.

- FINGERSEARCH$\big(k_1,\, k_2\big)$. Given that key k_1 already in the data structure, search for k_2 in the data structure, starting from the position of k_1. The skip list supports a finger-search operation in $O(d)$ expected time, where d is the search path distance between k_1 and k_2.

Munro, Papadakis and Sedgewick [59] applied deterministic algorithms on skip lists and proved that the deterministic skip lists are competitive with balanced trees in terms of both time and space. The deterministic skip list guarantees at maximum $2n$ pointers and a $\Theta\big(\log n\big)$ worst-case time per update operation, where n is the number of distinct keys in the data structure. Bose et.al [17] developed the skip lift, which improves the performance of a regular skip list. For each update operation, it needs only $O\big(1\big)$ worst-case time for structural changes. Golovin proposed a B-skip-list [35] inspired by B-trees, which can be used in file systems and databases. The B-skip-list supports search, insert or delete operations in $O\big(\log_B n\big)$ expected I/O transfers, where B is usually chosen to match the size of a cache block.

Skip lists can be applied to distributed systems. In 2003, Aspnes and Shah [10] extended skip lists using a graph algorithm and named this extension skip graph. In a skip graph, each level may contain several doubly linked lists and each node on that level belongs to one of these lists. The lowest level L_0 in a skip graph contains all the keys in a doubly linked list. level L_1, which is immediately above L_0, is divided into 2 lists, and level L_i is divided into 2^i lists when $i \leq \lceil \log n \rceil$. Skip graphs are used in distributed systems where resources are stored in separate nodes. Each node x in the skip graph has two fields: a key and a membership vector $m(x)$ of an infinite word over a fixed alphabet that x is in. In which sub-list each node belongs to is decided by the node's membership vector. When a sub-list can be identified by a finite word w and w is a prefix of $m(x)$, x is contained by this sub-list. In practice, each node can be hosted by different machines. Experimental results indicate the skip graph

has a better fault tolerance performance over other distributed data structures, since repairing errors in skip graph is simpler and more straightforward. Erway et al. [30] developed a dynamic provable data possession $\big(\text{DPDP}\big)$ scheme based on skip list models. Provable data possession $\big(\text{PDP}\big)$ technique is used when a client needs to know if a server stores the data which is previously stored by the client and if the data is not tampered. Classic PDP algorithms [11] can only be applied on static files, because the clients are not allowed to update the sorted files. The DPDP scheme, on the other hand, can still guarantee data possession with updates.

Skip lists can be also extended as networking techniques. An overlay network can be seen as a structure built on top of another network and its nodes are connected using virtual links but not real links. Those virtual links are representing paths in the underlying network. Using the skip graph's basic structure, Harvey et al. [38] developed a skipnet as an overlay network. A skipnet provides a fine data placement scheme and guarantees the locality of routing, that classical overlay networks can not support. Clouser, Nesterenko, and Scheideler [21] introduced a distributed skip list called Tiara. By using a self-stabilizing algorithm, Tiara can better tolerate faults or inconsistencies that are common in peer-to-peer systems, and can reach an eventual self-stabilization without previous knowledge of the current network situation. Nor, Nesterenko and Scheideler [61] developed a deterministic self-stabilizing skip list used for asynchronous message-passing systems. Wang and Liu [82] developed an overlay network for real-time media distribution, based on skip list algorithms. The authors also executed experiments to evaluate the performance of this overlay network, and claimed it is capable with frequent videocassette recorder $\big(\text{VCR}\big)$ operations, such as pause, resume, fast-forward, rewind and stop, at a low cost and still maintains a satisfying playback quality, which other existing media networks can rarely achieve.

Ergun et al.[29] proposed the dynamic biased skip list, which takes the access pattern into consideration and supports a $O\big(\log r(k)\big)$ amortized time for successful search, insert and delete operations. Bagchi and Buchsbaum [12] developed a variant of skip list for generally biased access sequences using a weight function. Given n elements, let w_i be the number of times that the item i is accessed, where $1 \leq i \leq n$, and $W_S = \sum_{1 \leq i \leq n}^{i} w_i$. This biased skip list accesses an item i in $O\big(\log \frac{W_S}{w_i}\big)$ time, which is asymptotically optimal. Bagchi and Buchsbaum showed the biased skip lists

also support these three operations: joining two biased skip lists S_1, S_2 is bounded by $O(\log(\frac{W_1}{w_{1_{\max}}}) + \log(\frac{W_2}{w_{2_{\min}}}))$, where W_1 and W_2 are the total weights of S_1 and S_2 respectively, $w_{1_{\max}}$ is the weight of the largest key in S_1, and $w_{2_{\min}}$ is the smallest key's weight in S_2. Splitting one biased skip list S into two takes $O(\log(\frac{W}{\min(k^-, k^+)}))$, where W is the total weight of all keys in S, k^- and k^+ are k's largest in-order predecessor and smallest in-order successor. A finger-search operation in such a biased skip list uses $O(\log \frac{W(k_1, k_2)}{\min(w_{k_1}, w_{k_2})})$ expected time, where $W(k_1, k_2) = \sum_{k_1 \le u \le k_2} w_u$. The biased skip list maintains a biased dictionary and is theoretically comparable with some other weighted data structures in worst-case scenarios, such as biased search trees [15] and weighted randomized search trees [8], which will be described in Section 1.1.5. This biased skip list, however, may lack efficiency in practice since each key's weight must be determined in advance.

1.1.5 Other Biased Data Structures

Additional novel data structures have been designed for biased data patterns. For example, a biased search tree, developed by Bent, Sleator and Tarjan [15], holds an worst-case $O\big(\log \frac{W_s}{w_k}\big)$ bound for each search, insertion and deletion operation, where W_s and w_k hold the same definitions as the previously defined variables. Seidel and Aragon [8] developed a weighted randomized search trees. Such a randomized search tree can be used to execute search, insert, delete and finger-search operations on key k taking $O\big(\log \frac{W_s}{w_k}\big)$ time in expected cases, and execute join or split operations in $O(1 + \log \frac{W_1}{w_{T_1 max}} + \log \frac{W_2}{w_{T_2 min}})$ expected time, where W_1, W_2 are the total weights of the two trees, $w_{T_1 max}$, $w_{T_2 min}$ are the maximum weight of a key in the first tree T_1 and the minimum weight of a key in the second tree T_2, respectively. Using an underlying splay tree scheme, Iacono [41] developed a set of splay trees called the splay tree forest. A splay tree forest can access an existing item in $O\big(\log r\big(k\big)\big)$ amortized time and report a non-existing item in $O\big(\log n\big)$ time under the worst-case scenario.

1.2 Our Work

Our work is motivated by the work of Ergun et al. [28] on a dynamic extension of the skip list which takes advantage of the current working set property. The dynamic biased skip list theoretically satisfies an $O\big(\log r\big(k\big)\big)$ expected time for search, insertion and deletion operations.

Ergun et al. [29] implemented a biased skip list and executed experiments with move-to-front lists and original skip lists. However, the dynamic biased skip list is not fully implemented and tested, and there remains a large amount of work to be done. For example, they only tested search and insert operations, and the delete functions were not coded. More real-world data sets should be used to evaluate the performance, and more self-adjusting data structures should be tested against the biased skip list, especially splay tree models, which have the working set property.

In our experimental studies, we first implemented the dynamic biased skip list with search, insert, delete operations. Then, we applied a hybrid search algorithm on the dynamic biased skip list using real-world data sets; this search algorithm examines a small set of recently accessed keys sorted by rank before searching from the head position of a biased skip list. After changing parameter values in the hybrid data structure, we found the optimized configured dynamic biased skip list for each data set. Finally, dynamic biased skip lists were compared with move-to-front lists, original skip lists, red-black trees, splay trees, R-splay trees and W-splay trees using both real-world and synthetic data. Our experimental results suggested that dynamic biased skip lists have the potential to be a better choice for highly biased data patterns with search, insertion and deletion operations. However, the move-to-front list was the fastest data structure when the sequence contains only search requests. Splay trees and splay tree extensions are also able to take biased access patterns into consideration and result in better performances than ordinary skip lists and red-black trees.

1.3 Organization of The Thesis

The remainder of this thesis is organized as follows. In Chapter 2, we introduce the move-to-front list, the original skip list, splay tree, two splay tree extensions and the red-black tree. The move-to-front list is discussed since the dynamic biased skip list

applies the move-to-front algorithm as an underlying scheme. We thoroughly discuss the self-adjusting data structures mentioned above so that readers can gain a deep understanding about the code in our paper.

Chapter 3 demonstrates the dynamic biased skip list. We start the chapter by discussing a biased skip list, which provides fast lookup responses but lacks efficiency in construction and insert/delete operations. Following that, we explain a simpler construction method that avoids unnecessary copies and apply a lazy updating scheme to prevent frequent structure adjustments.

Our experimental results are given in Chapter 4. First, we determine the optimal dynamic biased skip list parameters for each data set by experimenting with different parameter values. After that, we compare move-to-front lists, skip lists, red-black trees, splay trees and their variants with dynamic biased skip lists using real-world data. Experimental studies with synthetic data are conducted in the last section.

Chapter 5 presents our conclusions, not only for the dynamic biased skip list, but also for the other data structures we implemented.

Chapter 2

A Review of Different Search Structures

This chapter introduces move-to-front lists, original skip lists, red-black trees and splay tree extensions that will be compared with the dynamic biased skip list in our experimental studies.

2.1 Move-to-front Lists

The move-to-front list [74] is a simple structure which optimizes a linked list by putting the last accessed key at the head of the list. This heuristic adapts quickly to bias changes in data queries. The move-to-front scheme is used in the dynamic biased skip list as an underlying filter for highly skewed data patterns. Also, the move-to-front list is tested as a benchmark against other data structures.

In a move-to-front list, the search algorithm starts from the head node and follow the links to find the target value. If the target key k is found, the nodes preceding and following k are linked and k is moved to the head of the list. If k is not present in the current list, the search procedure will terminate until the end of the list is reached. Search, insert and delete operations in a move-to-front list are completed in $O(n)$ worst-case time. Figure 2.1 illustrates the search procedure in a move-to-front list.

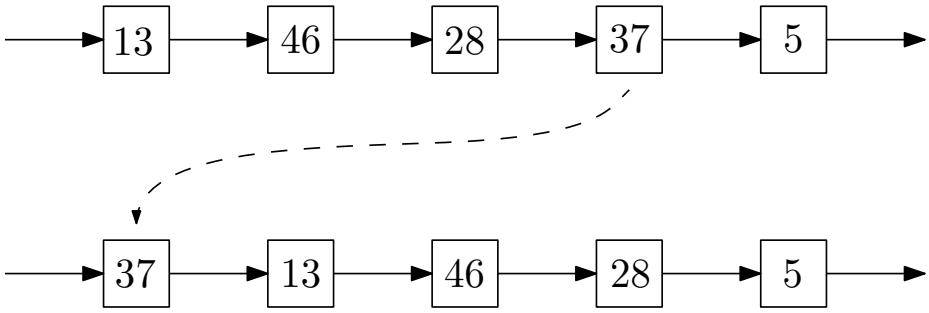


Figure 2.1: Search for 37 in a move-to-front list

Move-to-front lists can be used in caches or files systems [43, 42], or in a data

compression algorithm [20, 80]. If the access sequence shows locality of reference, the frequently accessed are more likely to stay near the head position and thus can be found more quickly. A move-to-front list thus supports efficient operations for highly biased data sets. However, a move-to-front list lacks efficiency with uniformly distributed queries or queries that include insertions and deletions.

2.2 Red-black Trees

Red-black trees are binary trees in which each node holds additional information to keep the tree balanced. This extra information is a colour, which is either red or black. A red-black tree satisfies the following properties:

1. Each node is either red or black.
2. The root node is black. All leaf nodes (NIL nodes) are black.
3. A red node's parent must be black.
4. All paths from the root to a leaf node contain the same number of black nodes. The number of black nodes along a path from the root node to any leaf node is called the *black height* of the tree.

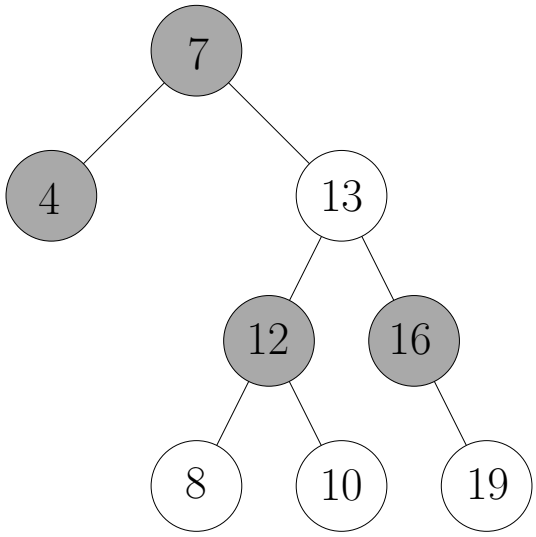


Figure 2.2: A red-black tree

Figure 2.2 is an example of a red-black tree, in which the dark colored nodes represent the black nodes and the light coloured nodes represent the red nodes. The NIL nodes are omitted in this figure. Red-black trees have been widely studied, and have the ability to deal with frequent insertions and deletions [62, 73, 78, 40]. Therefore, red-black trees can be used as a good benchmark for the dynamic biased skip list in regards to experiments with update operations.

By keeping these colouring requirements, a red-black tree can always maintain a logarithmic height. Let us consider a red-black tree of n nodes, and remove all the red nodes without changing the black height of tree. Then reconnect each black node to its closest ancestor. The 'trimmed' tree has leaves at the same depth, which is the black height $\big($denoted as $bh\big)$. Each internal node in this tree has 2 to 4 children. Such a tree contains at least $\Omega(2^{bh})$ nodes. Adding back the red nodes, if n is $\Omega(2^{bh})$, bh is $O(\log n)$. Since there are not two adjacent red nodes along any path, the longest path is at most $2bh$. Therefore, the height of a red-black tree is always bounded by $O(\log n)$.

Searching for key k in a red-black tree is the same as searching in an ordinary binary tree. The search procedure starts from the tree's root node. Since the red-black tree is built in sorted order, k and the key of the current node can be compared. If k is equal to the current node, the target is found and the search operation terminates. If k is larger than the current node, the search operation proceeds to the current node's right child. Otherwise, it follows the left link to reach the left subtree. This process is iterated until a node that has a key equal to k is reached, or the remaining subtree is null.

An insertion operation starts with an unsuccessful search that returns the position in which to insert the element. After adding the new element into the tree as done in a plain binary search tree, two operations are performed to resolve the colour properties. Since the colour properties may be violated after an insertion, a *recolour* operation is need to be done to solve that. If a single recolour operation does not work, more nodes will be recoloured, and the tree structure will be adjusted using *rotations*. Splay trees, AVL trees and some other tree structures also use rotations for reconstruction.

A rotation moves a child node up and its parent node down, thus changing the

structure of the tree without changing the in-order traversal key ordering. There are two kinds of rotations: left-rotation and right-rotation. Let x denote the current node, and p denote the parent node of x. A left-rotation is needed when x is the right child of p. x will be moved up as a new parent, thus making p the left child of x. Figure 2.3 shows an example of the left-rotation. c_l and c_r represent the left and right child of x, respectively. s is an additional child of parent p. A detailed description of this process is provided in Algorithm 1. In the pseudocode, *left* represents the left child of the current node and *right* represents the right child of the current node. The parent node of current node is denoted as *parent*. *Root* is the root node of the tree and x is the right child of p, which follows the same pattern as Figure 2.3. Similarly, a right-rotation moves x, which is now the left child of p, up as a new parent of p and makes p the right child of x. How a red-black tree rebalances itself using recoloring and rotations can be found in [22].

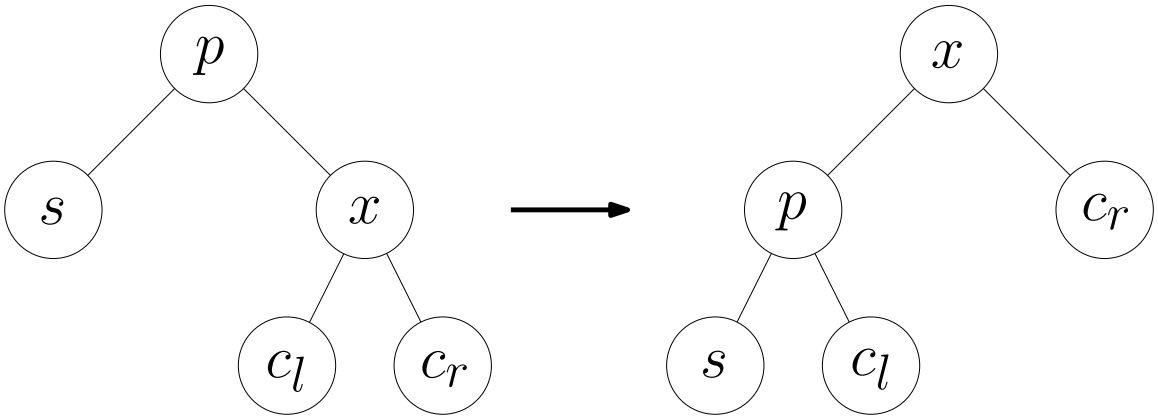


Figure 2.3: A left rotation

A red-black tree guarantees $O\big(\log n\big)$ worst-case time per operation. Some researchers [62, 73, 40] have pointed out that red-black trees may outperform other binary trees when handling frequent insert/delete operations, but may not be as efficient as other self-adjusting trees that do searches solely [78, 25]. Red-black trees are widely used in many applications, such as:

1. HashMap in Java 8 [55]. A HashMap has a number of buckets, and entries are stored in them. The implementation of HashMap uses red-black trees to represent the entries in a bucket once the bucket size exceeds a certain constant.

Algorithm 1 Left-rotation(p)

```
p.right ← x.left
x.left.parent ← p
x.parent ← p.parent
if p.parent=NULL then
    Root ← x
else if p = p.parent.right then
    p.parent.right ← x
else
    p.parent.left← x
x.left ← p
p.parent ← x
```

2. C++ STL, such as map, multimap, multiset [57, 19].

3. Completely Fair Scheduler [63] in Linux kernels. CFS is implemented as a red-black tree to handle CPU resources for executing processes and optimize CPU utilizations. It uses a red-black tree to keep track of future tasks.

2.3 Splay Trees

We briefly review splay trees and two splay tree extensions: randomized splay trees and W-splay trees. The randomized splay tree was introduced by Albers and Karpinski [5] and the W-splay tree was introduced by Aho, Elomaa and Kujula [4].

2.3.1 Original Splay Tree

The splay tree is a self-adjusting binary search tree devised by Sleator and Tarjan [76]. It is theoretically proved to satisfy the Static Optimality Theorem, which means that a splay tree performs as well as an optimum static binary search tree. A splay tree is easy to implement since each node of the tree holds no additional information for maintaining the tree's balance.

A splay tree supports SEARCH, INSERT, DELETE, JOIN and SPLIT operations. The fundamental idea behind a splay tree, is called 'SPLAY'. This technique

is automatically performed with every single operation.

The essential idea of SPLAY(x) is to move the current node x up to the root position using a series of rotations. This procedure enables a key to be accessed quickly in the near future. Different from the single rotation operation in a binary search tree, the splaying operation is usually done by combining two rotations in a pair. If the node x is rotated to the root position using only single rotations, the running time can be $\Theta(n)$ per access [76]. This indicates, using only single rotations to adjust the structure shows a lack of amortized efficiency. Splaying performs rotations in pairs. Let p denote the parent of x, and g denote the grandparent of x $\big($which is the parent of $p\big)$. If the path from p to x goes through p's left link, it is called a *zig* case. Otherwise, it is a *zag* case. Splaying is performed with two rotations, except when p is the root node of the tree, in which case a simple left rotation or a right rotation is performed.

Depending on the positions of x, p and g, there are four cases of double rotations: zig-zig, zag-zag, zig-zag, and zag-zig. These four cases are the only possibilities in making two moves along a path. SPLAY(x) repeats the following steps until x becomes the root of the tree.

- Zig-zig case/ Zag-zag case

  In a zig-zig or a zag-zag case, x and p appear on the same side of their parent, both left or both right. Figure 2.4 shows the example of zig-zig case. A rotation is needed at g, and p and x will be moved up one level. Following that, a rotation at p is performed, which takes x up one more level. A zag-zag case can be solved symmetrically.

- Zig-zag case/ Zag-zig case

  These two cases cover the remaining conditions. In this case, p and x are different types of children. Figure 2.5 presents the procedure for dealing with a zig-zag case, where x is the right child of p and p is the left child of g. Two rotations will be performed: a rotation at p, taking x one level up to become the parent of p and a rotation at g, making x the parent for both p and g.

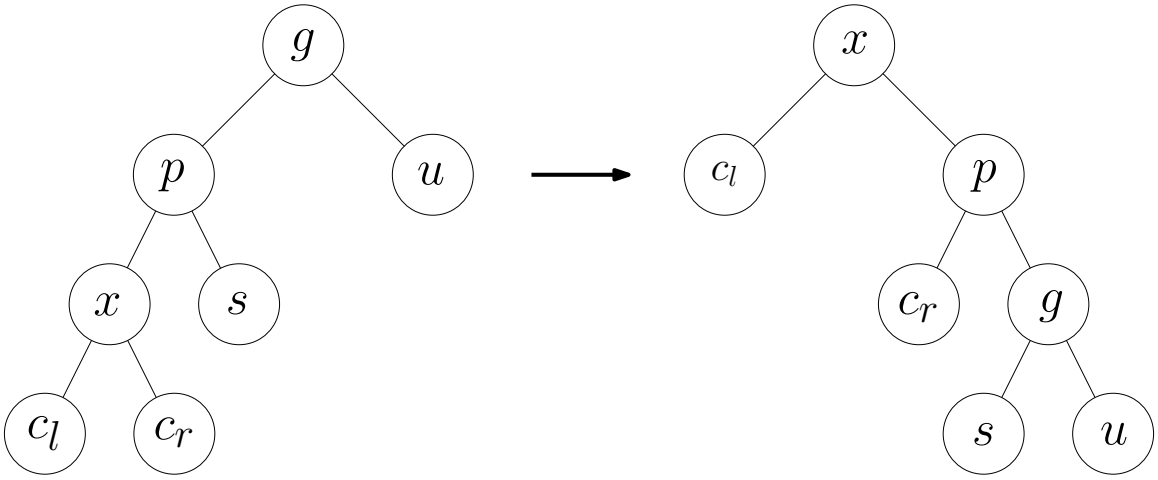


Figure 2.4: Zig-zig case

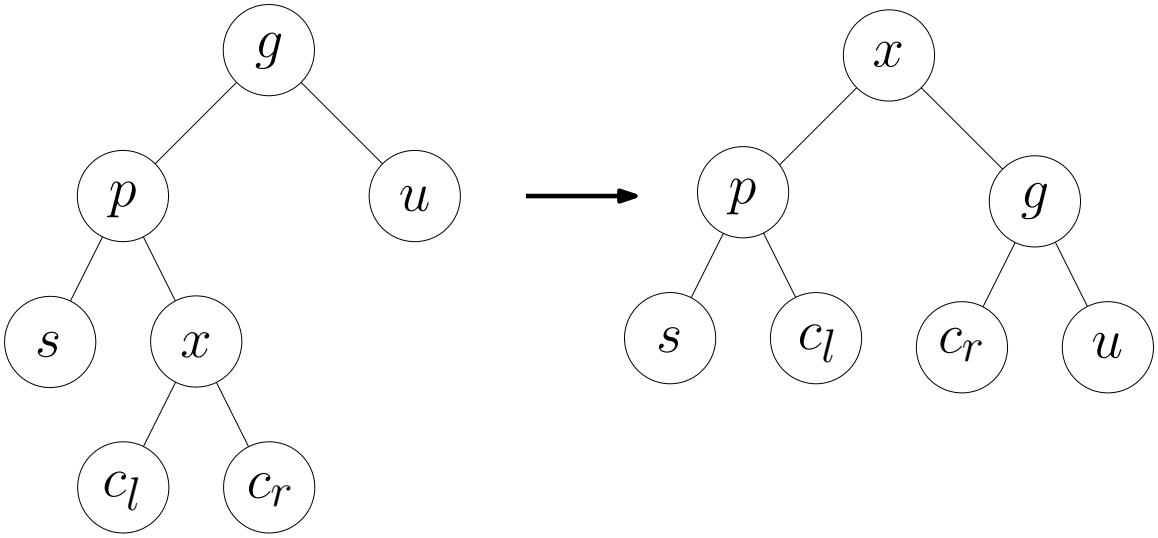


Figure 2.5: Zig-zag case

A splay tree favours the following operations in logarithmic time:

1. SEARCH(k)

   To search for a certain key k in a splay tree, the first step is same as that in a binary tree. If the target key is found, it is splayed to the root position. If the key does not exist, the last non-NULL node on the search path will be lifted up to the root position by splaying.

2. INSERT(k)

To insert a key k into the tree, a binary search will be triggered in order to provide a proper position for this insertion. After the key is inserted, a splaying operation on the newly inserted node is performed, so this node becomes the new root of the tree.

3. DELETE(k)

   To delete a node x of key k, x will be found by searching the entire tree. If x has no children, it can be removed directly. Otherwise, either x's in-order predecessor or successor needs to be found. The node copies its key to x, and then is deleted. Finally, splaying is performed on the parent of the removed node, making the parent node the new root of the tree.

4. JOIN(T_1,T_2)

   Assuming all the keys in tree T_1 are smaller than those in tree T_2. To merge the two splay trees, the first step is to search for the largest key k_{max} in T_1 and rotate it to the root. Then, in the resulting tree, k_{max} will have no right children. Following that, T_2 is attached as a right subtree of k_{max}, so that the two trees are jointed.

5. SPLIT(k,T)

   The split operation returns two subtrees of tree T, where all keys in one subtree are smaller than or equal to key k and all keys in the other subtree are larger than k. Assume that k already exists in the tree. A splay operation is executed on k so that k becomes the root of the tree. Then the right subtree of k will be spliced out, so that the two required trees are obtained. If k is not maintained by the tree, the first step is to find a key k' which is the largest element smaller than k in the splay tree, and the following steps are the same as previous but performed on k'.

Sleator and Tarjan [76] showed some theorems that splay trees satisfy, such as the Balance Theorem and the Working set Theorem.

Theorem 1 (Balance Theorem) *The total access time of a sequence with length m and n distinct keys is at most $(1+d)(3m\log n+m)+(1+d)n\log n$.*

Theorem 2 (Working Set Theorem) *The total access time of a sequence with length m and n distinct keys is at most* $(1+d)(6\sum_{j=1}^{m}\log(r(j)+1)+4m)+(1+d)(2n\log n+n)$.

Splay trees can be applied for data compression and encryption [45, 36, 58]. In data compression algorithms, the input text can be compressed by using fewer bits to encode more frequently occurring characters. Since the occurring characters exhibit locality of reference, splay trees can be used in those applications for compressing the input text. Besides, a node in a splay tree only requires two pointers with no additional information. This implies a splay tree has a better cache performance [85, 33, 81] when memory is limited. Moreover, with the working set property, splay trees are particularly useful in garbage collection algorithms [46].The limitation of splay tree algorithm is the height of a splay tree can be linear [81, 79] in worst-cases.

2.3.2 Randomized Splay Tree

As discussed in the last section, splay trees support amortized logarithmic running time per operation and have the working set property. By adapting itself with each search or update operation, a splay tree makes the most frequently accessed elements stay near the root of the tree. However, splay trees have the disadvantage that they rotate with every access, which may lead to a high overhead. Based on this observation, Albers and Karpinski [5] developed a randomized splay tree. In a randomized splay tree, the splaying is executed with a certain probability $p \in \big[0,1\big]$. Operations within a randomized splay tree are only slightly different from that of one regular splay tree. When searching returns a node, a random float number in the range of $\big[0,1\big]$ is generated. If the number is smaller than p, the current node is splayed up to the root position. Otherwise, the tree remains the same. From this randomized splaying scheme, one can observe that if p is set close to 1 or the input pattern is highly biased, the most 'popular' elements will still have a high possibility of staying near the root position. Albers and Karpinski [5] also theoretically approved that the randomized splay tree satisfies the Balance Theorem and the Working Set Theorem.

Theorem 3 (Balance Theorem) *The expected total access time of a sequence with length m and n distinct keys is at most* $(1+pd)(3m\log n+m)+(\frac{1}{p}+d)n\log n$.

Theorem 4 (Working Set Theorem) *The expected total access time of a sequence with length m and n distinct keys is at most* $(1+pd)(6\sum_{j=1}^{m}\log(r(j)+1)+4m)+(\frac{1}{p}+d)(2n\log n+n)$.

2.3.3 W-splay Tree

In experimental studies, randomized splay trees gain some improvements upon original splay trees by reducing the number of splayings using a random parameter. However, whether the amount of splay operations could be reduced more efficiently is left an open problem. Aho, Elomaa and Kujula [4] developed another splay tree extension call 'W-splay tree', which takes the current working set into consideration and reduces the number of splaying steps. The *working set* [26] in a W-splay tree is slightly different from what we previously defined. It is defined as follows: access requests to keys often exhibit locality of reference. At any time interval, only a small portion of data may be accessed frequently while the other keys stay 'unpopular'. Such a small portion of keys are defined as the current working set. The main idea of W-splay algorithm is to monitor the need for splaying by maintaining a *counter*, whose initial value is 0. Each access contributes to a change of the counter in the following way:

$$counter = counter \cdot d + depth - limit_w. \tag{2.1}$$

In this equation, *depth* is the number of edges that are traversed to find the target key. d and $limit_w$ are two variables that affect the value of *counter*, which will be described later. When the value of the counter is non-negative, the tree splays; otherwise, it stays still. d is a discount factor in a range of $\big[0,1\big)$, which regulates the effect that the previous value of *counter* has on its current value. Taking discounted history into account is necessary, since it ensures that accessing a non-popular key does not restructure the tree. In practice, the authors found out the value of d does not overly affect the performance of the W-splay tree. Therefore, they suggest $d = 0.9$ as an acceptable value of d. $limit_w$ is the maximum value for acceptable depths for a working set w, it is initialled as follows,

$$limit_w = c\log(s_w + 1), \tag{2.2}$$

where c is a constant, and s_w is the size of w, which is approximated using the method shown in [26]. Changes to the value of *limit* are allowed with every access in the W-splay scheme:

$$limitChange = p\big(\frac{a}{a+b} - \frac{1}{d}\big). \tag{2.3}$$

Assume the sequence s_1,s_2,...s_{i-1} is accessed and the access s_i is being operated. a is the number of splayed accesses in s_1, s_2,...,s_{i-1}, b is the number of non-splayed accesses before executing s_i. Notice that $a+b=i$. p is a factor defining the adaptive depth. If the value of *limitChange* is positive, it will be added to $limit_w$. Thus the variable $limit_w$ can be adjusted with every splaying operation. If the value is negative, then the value of $limit_w$ does not change.

The W-splay algorithm makes it possible to take advantage of the working set and reduces the number of costly splaying operations by monitoring the current working set. In addition, Aho, Elomaa and Kujula [4] proved that the W-splay tree conforms to the balance theorem of a splay tree:

Theorem 5 (Balance Theorem) *The total access time of a sequence with length m and n distinct keys is at most $2m_n limit_w + m_s(1+d)(3\log n + 1) + (1+d)n\log n$.*

Based on the experimental results in [4], the access time was shorter in W-splay trees than in ordinary splay trees. We will execute experiments using real-world data sets to further test the performance of W-splay tree.

2.4 Skip Lists

Skip lists [71] can be seen as a simulation of a binary tree built via a hierarchy of linked lists. A skip list allows for lookups in $O\big(\log n\big)$ expected time, where n is the number of distinct keys held by the data structure. A skip list contains several levels and is built in a bottom-up fashion. Each level is an ordered linked list. The lowest level L_0 contains all keys in sorted order. To construct level L_1, which is one level higher than L_0, each node in level 0 is copied up to next level with a certain probability. In our study, the copying-up probability is set to $\frac{1}{2}$, which is the most commonly used value. This is repeatedly performed until a level with no keys copied up is encountered. On average, $\frac{n}{2}$ keys are in level L_1, $\frac{n}{4}$ keys are in level L_2, $\frac{n}{8}$

keys are in level L_3, and in general, $\frac{n}{2^i}$ keys are in level L_i. The expected height of a skip list is thus bounded by $\log n$. The structure of a skip list can be viewed as a two-dimensional collection of nodes arranged horizontally into levels and vertically into columns.

A search operation in a skip list starts from the leftmost key on the highest level. Then the search traverses the data structure using horizontal pointers until the current element is greater than or equal to the target value. If the current element is equal to the target, then the target is found. If current element is greater than the target, or the end of current level is reached, the procedure returns to the previous node and drop down one level. By repeating this procedure, either the target is found, or the bottommost level is reached, where the target key is in the range of $\big(k_1, k_2\big)$ and k_1 and k_2 are two adjacent nodes on the lowest level.

We now analyze he expected running time for each access. Suppose k is found at one level, then we analyze the search path reversely. The level at which k is found is defined as level l_1. The next higher level is l_2, and so on. The analysis counts the expected number of links from the current location of k to the leftmost and topmost key in the data structure. Let S represent the search path on a key k, $c(i)$ represent the number of links traversed on level l_i, h be the expected height of the skip list. The expected length of a search procedure is

$$E[S] = E[h + \sum_{i=0}^{\infty} c(i)] = E[h] + E[\sum_{i=0}^{\log n} c(i)] + E[\sum_{i=\log n+1}^{\infty} c(i)] \tag{2.4}$$

The expected height $E(h)$ is $O(\log n)$. Since the copy-up probability is $\frac{1}{2}$, the expected number of keys traversed on one level before finding a link to go to a higher level is thus 1. The probability of a node that has more than r levels is $\frac{1}{2^r}$, so the

expected number of nodes in level r is $\frac{n}{2^r}$. Thus,

$$\begin{aligned} E[S] &= E[h] + E[\sum_{i=0}^{\log n} c(i)] + E[\sum_{i=\log n+1}^{\infty} c(i)] \\ &= E[h] + E[\sum_{i=0}^{\log n} 1] + E[\sum_{i=\log n+1}^{\infty} \frac{n}{2^i}] \\ &\leq E[h] + E[\sum_{i=0}^{\log n} 1] + E[\sum_{i=0}^{\infty} \frac{1}{2^i}] \\ &\leq O(\log n) \end{aligned}$$

The expected search cost in a skip list is bounded in $O(\log n)$. An example of the search operation in a skip list is shown in Figure 2.6. The bold line shows the search path for key 42.

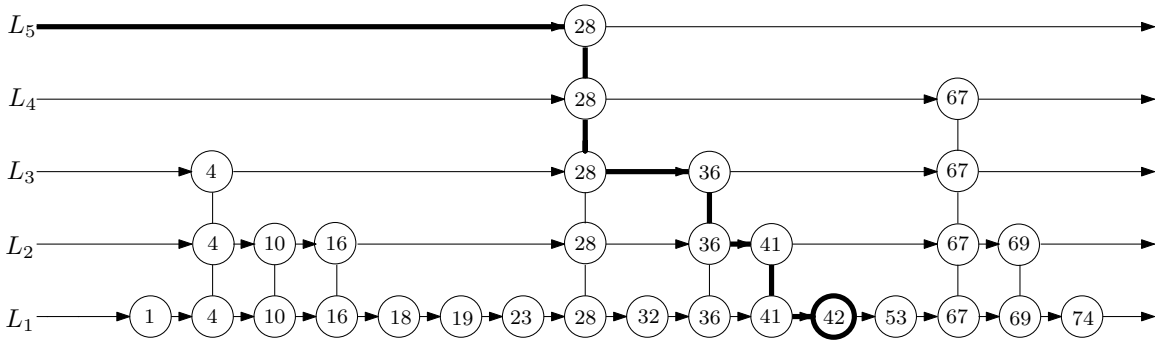


Figure 2.6: Search for key 42 in a skip list

The insertion of key k into a skip list starts with a search operation, which returns the correct position on the bottommost level in which to insert k. After determining the correct position, the node of key k is immediately inserted and copied up to a higher level based on the copying-up probability. The key will be copied to level L_2 with a probability of 1/2 and to L_3 with a probability of 1/4. This is continuously performed until k no longer needs to be copied up. The skip list's deletion operation also starts with executing a search operation. If target key k is found, all copies of k are removed, which can be easily accessed by climbing down the column of k. The number of copies of k is $O\big(\log n\big)$ as expected, and the number of pointers that must be updated is no more than twice the number of levels in the list. The assignments to

new nodes or deletions of an existing node can be done in constant time. Therefore, it is safe to predict that the skip list's average running time for completing an insertion or a deletion is $O\big(\log n\big)$.

A skip list has been used for:

1. The implementation of the QMap class in Qt application, which is a cross-platform application development framework for desktop, embedded and mobile. The framework of Qt is written in C++ and QMap class $\big($up to Qt4$\big)$ is a template class that uses a skip-list as a key/value dictionary [77].

2. ConcurrentSkipListSet and ConcurrentSkipListMap in the Java 1.6 API [39, 27].

3. Databases as a index of key/value pairs, like MemSQL [24, 52], skipDB [34].

4. Distributed applications [30], where the nodes of the skip list represents the computer systems and pointers represent a network connection.

Skip lists have drawbacks as well. First, there is a very small possibility that a skip list will become a linked list, where all keys are stored in the same level. Moreover, due to the coin-flip scheme and linked hierarchy, a skip list may not be perfectly balanced thus each search, insert and delete operation may have an $O(n)$ worst-case running time [71]. Plenty of research on skip lists [65, 49, 64, 48] claims that a skip list guarantees $O\big(\log n\big)$ expected running time for each operation. We will execute experiments using both real-world and synthetic data sets with skip lists.

Chapter 3

Dynamic Biased Skip Lists

In this chapter, we introduce the dynamic biased skip list, developed by Ergun, Sahinalp, Sharp and Sinka [28, 29]. First we demonstrate a biased version of the skip list, which takes advantage of query patterns using a rank assignment scheme. This biased skip list supports a successful search in $O\big(\log r\big(k\big)\big)$ time and an unsuccessful one bounded by $O\big(\log n\big)$. Also it favors an insert/delete operation in $O\big(\log n\big)$ expected time. Then we introduce a modification of the biased skip list which supports updates in $O\big(\log r\big(k\big)\big)$ time.

3.1 Biased Skip List

The biased skip list is an extension of skip lists. Each existing key in the biased skip list is given a unique rank in the range [1,2,...,n], where n is the number of keys maintained by the data structure. When a key k with rank i is accessed, it is given a new rank of 1. All the keys with a rank $i^{'} < i$ will have to update their ranks to $i^{'} + 1$. If a key is not previously maintained by the data structure, it will be given rank 1 , and all the existing keys increase their ranks by 1. Thus, by maintaining the ascending rank order, the least recent access order of keys can be recorded.

Let *rank* of key k, denoted as $r\big(k\big)$, represent the number of unique keys accessed since last time k was accessed. This is k's current rank. Let $r_{max}\big(k\big)$ represent the maximum rank in k's lifespan. A conceptual move-to-front list is applied to keep the keys in ascending rank order. The list is divided into *classes* conceptually: class C_1 contains the first key with the smallest rank, class C_2 contains the next 2 keys in rank order, and following this pattern, in general, class C_i contains 2^{i-1} keys. Consider a biased skip list of n distinct keys. Based on the rank assignment scheme and class division scheme, the number of classes is $\lceil \log n \rceil$. A biased skip list is constructed in a bottom-up fashion like a regular skip list, in which each level maintains keys in sorted order. Assume the distinct keys are partitioned into c classes. The levels in a

biased skip list, from the bottommost level to the topmost one, are defined as level L_{2c-1}, $L_{2(c-1)}$, $L_{2(c-1)-1}$,..., L_2, L_1. They are constructed as follows :

- Level L_{2c-1}
  - The lowest level.
  - Maintains all keys in sorted order.
- Level L_{2d}, for $d = 1, 2, ..., c-1$
  - Contains all keys from classes C_1, C_2..., C_d.
  - Picks up some keys from Level L_{2d+1} with a probability of $\frac{1}{2}$.
- Level L_{2d-1}, for $d = 1, 2, ..., c-1$
  - Contains all keys from classes C_1, C_2..., C_{d-1}.
  - Picks up some keys from Level L_{2d} with a probability of $\frac{1}{2}$.

The conceptual move-to-front list can be implemented using extra links. In our design, we call it a *rank keeper*. For each key, we pick one of its copies that appears higher than any other copies as an *entry* , and use pointers to connect all the entries by rank ascending order. Based on the construction algorithm, all keys from class C_d, $1 \leq d < c$ will be automatically copied to Level L_{2d}. This level is called the *default level* of the key. Figure 3.1 gives an example of the construction procedure for a biased skip list.

Construction of a biased skip list of n keys takes $O(n)$ expected time, assuming the keys are in sorted order. The problem can be seen as counting the number of copies of all the keys. The copies of all keys can be divided into two parts: the automatic copies and the random copies. The random copies are created with a probability of $\frac{1}{2}$. Therefore, the expected number of random copies of a key is

$$\sum_{i=1}^{2c} \frac{1}{2^i} \leq \sum_{i=1}^{\infty} \frac{1}{2^i} \leq 1. \tag{3.1}$$

Now let us count the number of automatic copies. In class C_c, the number of keys is at most $\lfloor \frac{n}{2} \rfloor$. Those keys are automatically inserted into Level L_{2c+1}, thus the

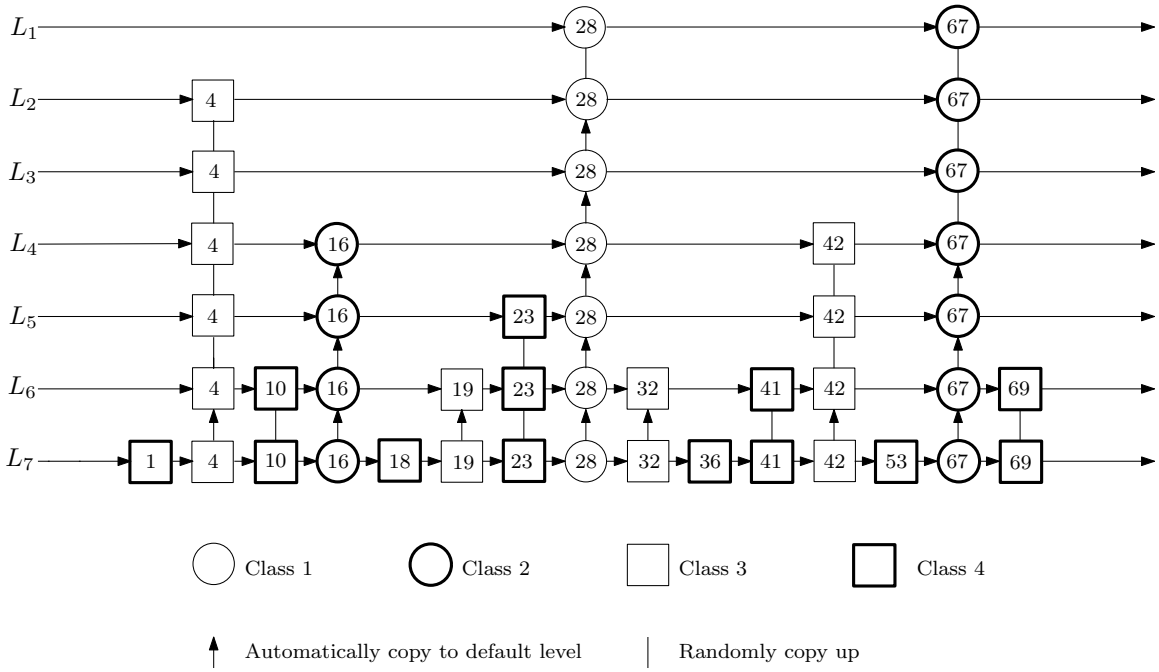


Figure 3.1: An example of biased skip list

copies from Class C_c are bounded in $O(n)$. In class C_d, there are at most $\lfloor \frac{n}{2^{c-i+1}} \rfloor$ keys and these keys will have $2(c-i)$ copies. Since the number of classes is bounded by $O(\log n)$, the total number of automatic copies of n keys is

$$\sum_{i=1}^{\log n} \frac{n \cdot 2(c-i)}{2^{c-i+1}} \leq O(n) \tag{3.2}$$

The first step in searching for a certain key k in the biased skip lists is exactly the same as searching in a regular skip list. After the key is successfully found, some following operations need to be performed. First, k will be reallocated as the new head of the rank keeper by adjusting related pointers. Also k will be given rank 1 and be lifted to Level L_2 in which all class C_1 elements should stay. Second, due to the class assignment rules, all classes are of fixed sizes. This means that the ranks of some keys are needed to be re-assigned due to the change of k's rank. Assume that k had a rank of i and belonged to class C_d. All keys that have ranks smaller than i should increase their ranks by 1. If k is the last element in class C_d, then all elements that are the last elements in class C_1 to class C_d should lower their default levels by 2. Otherwise, the last elements from class C_1 to class C_{d-1} are adjusted with the same operations mentioned above. Notice that it is very costly to increment ranks

of each affected key. To reduce the expense, we use an extra array to store all nodes located at the tail positions in each class, since only *class tail nodes* will be moved to the next class and be reallocated down two default levels in the data structure.

A successful search algorithm can be found in Algorithm 2. The *skipListSearch*(k) returns the position of k, following the search algorithm of a regular skip list. The *classInfo* represents the class number of the key. The array *Tail* stores the last elements from all classes, where *Tail[i]* returns the tail node of Class C_i. The *degrade* operation lowers the class tail node's default levels by 2 since they have a larger class number. In practice, this can be done by removing the highest two copies of a key, or if the key has no more than two copies, its copies are removed and the key is immediately inserted into a specific level. After that, the entry of each key is inserted to its previous position in the rank keeper. Similarly, in the *upgrade* operation on the newly-found node x, we simply copy the node up to its default level L_2, randomly copy it up to a higher level L_1, and move it as the new head of the rank keeper. Its previous successor *next* and predecessor *prev* in the rank keeper are linked after that.

Algorithm 2 Search algorithm in biased skip list

```
x ← skipListSearch(k)
num ← x.classInfo
x.classInfo ← 1
if x =Tail[num] then
    Tail[num] ← x.prev
for (i = 0; i < num; i++) do
    y ←Tail[i]
    Tail[i]← y.prev
    y.classInfo ← y.classInfo+1
    degrade(y)
upgrade(x)
```

A searching operation in a biased skip list takes $O\big(\log r\big(k\big)\big)$ time for a successful search, and $O\big(\log n\big)$ time for an unsuccessful search, where n is the number of distinct keys in the data structure. Let us first analyze a successful search. Assume that the target k is in Class C_d. According to the class assignment scheme, it is clear that

$d \leq \log r(k) + 1$. Searching for the target k involves two parts: traversing horizontal and vertical pointers. Since k is guaranteed to be found at Level L_{2d}, the vertical distance to travel is at most $2d$.

Now let us count the expected number of horizontal links traversed on each level. Class C_1 contains only 1 key, which is inserted to L_2 and will be randomly copied to L_1. Thus the expected number of node on the topmost level is at most 1. The expected number of links is 2 at maximum. Suppose that level L_x is reached by dropping from the node a_{x-1} on Level L_{x-1}. Assume that the node right next to a_{x-1} on L_{x-1} is a'_{x-1}. According to the construction scheme, it is clear that $a_{x-1} < k < a'_{x-1}$. Let a_x and a'_x represent the copies of a_{x-1} and a'_{x-1} on L_x, respectively. The number of links we travelled on L_x, in the worst case, is the number of links between a_x and a'_x. According to the randomized copy-up scheme with the copy-up probability of $\frac{1}{2}$ that all keys are subject to, the expected number of keys that are not copied to a higher level between two keys that have been copied to a level above is 1. Also, notice that some of the keys between a_x and a'_x are copied up in a mandatory manner if the key's default level is higher than L_x, so the expected number of keys that are not copied up between a_x and a'_x is at most 1. Therefore, the number of links traversed on Level L_x is at most 2. In general, a search involves traversing $2d$ levels with at most 2 links on each level in expectation, so the expected search time for an existing key k is bounded by $O(d)$.

After locating k, the class tail nodes that have a class number smaller than d need to be updated. This is done in the for loop in Algorithm 2. We previously presented the algorithm for degrading a tail node: delete the two copies appearing on the two highest levels, and link it again into rank keeper. Inserting or deleting a node, assigning new class numbers and adjusting pointers in rank keeper all take constant time. Thus the cost over the for loop is bounded by the number of affected class tail nodes, which is $O(d)$. In the following *upgrade* operation, the newly found node x needs to be copied up to at most Level L_1. Since k is guaranteed to be found at least on Level L_{2d}, the total number of new copies of k is no more than $2d-1$. The time for joining x's predecessor and successor by moving x as the new head in rank keeper takes constant time. Thus, the expected time on operations after finding k is bounded by $O(d)$ as well. Summing up the two results shows that the entire search

procedure takes $O\big(d\big)$ expected time, which is $O\big(\log r\big(k\big)\big)$.

For an unsuccessful search, the number of links traversed on each level is no more than 2, as previously analyzed. The search procedure will end when the bottommost level is reached. Since the total number of levels in a biased skip list is $2c + 1$, the expected time for an unsuccessful search is $O\big(\log n\big)$, where c is the number of classes in the data structure, and $c = \lceil \log n \rceil$.

A biased skip list accesses an existing key in $O\big(\log r\big(k\big)\big)$ time, which satisfies the working set property. However, an insertion or deletion operation lacks efficiency due to the mandatory copying-ups and moving-downs of affected keys. For example, to insert a key k into a biased skip list, knowing that k will not be found, it will be inserted at the bottommost level and copied up to Level L_2 or Level L_1. Each class tail node must be moved two levels down. The whole insertion operation therefore takes $O\big(\log n\big)$ expected time. Deleting a certain key k from class C_d starts with locating k in the data structure. Then all the copies of k can be removed by traversing down. After that, each key that is the first element in class $C_{d'}$, for all $d < d^{'} \leq c$, will be moved up two default levels. The entire deletion operation is completed in $O\big(\log n\big)$ expected time.

Based on this observation, two modifications are developed to improve the performance of insert and delete operations in a biased skip list: a simpler construction scheme and a lazy-updating algorithm.

3.2 Modifications

3.2.1 Improved Construction Scheme

As described in the previous section, a biased skip list supports efficient searching operations. However, large numbers of copies of a key, especially of a frequently-accessed key, will slow down the progress of an insertion or a deletion. Besides, searching for a 'unpopular' key is costly since lower levels contain more keys, including ones with small ranks. To improve these, a biased skip list can be modified with a simpler construction. Assume that c is the total number of classes in a biased skip list. $c = \log r\big(n\big)$, where n is the number of distinct keys.

- Level L_{2c-1}

  - All keys in Class C_c.

- Level L_{2d}, for $d = 1, 2, ..., c-1$
  - All keys in Class C_d.
  - Picks up some keys in Level L_{2d+1} with a probability of $\frac{1}{2}$.
  - Copies down some keys in Level L_{2d-1} after L_{2d-1} is built up, with a probability of $\frac{1}{2}$. Those keys can be further copied down to lower levels with a a probability of $\frac{1}{2}$.

- Level L_{2d-1}, for $d = 1, 2, ..., c-1$
  - Picks up some keys in Level L_{2d} with a probability of $\frac{1}{2}$.
  - Copies down some keys in Level L_{2d-2} after Level L_{2d-2} is built up, with a probability of $\frac{1}{2}$. Those keys can be further copied down to lower levels with a a probability of $\frac{1}{2}$.

Figure 3.2 illustrates the improved construction progress of a biased skip list.

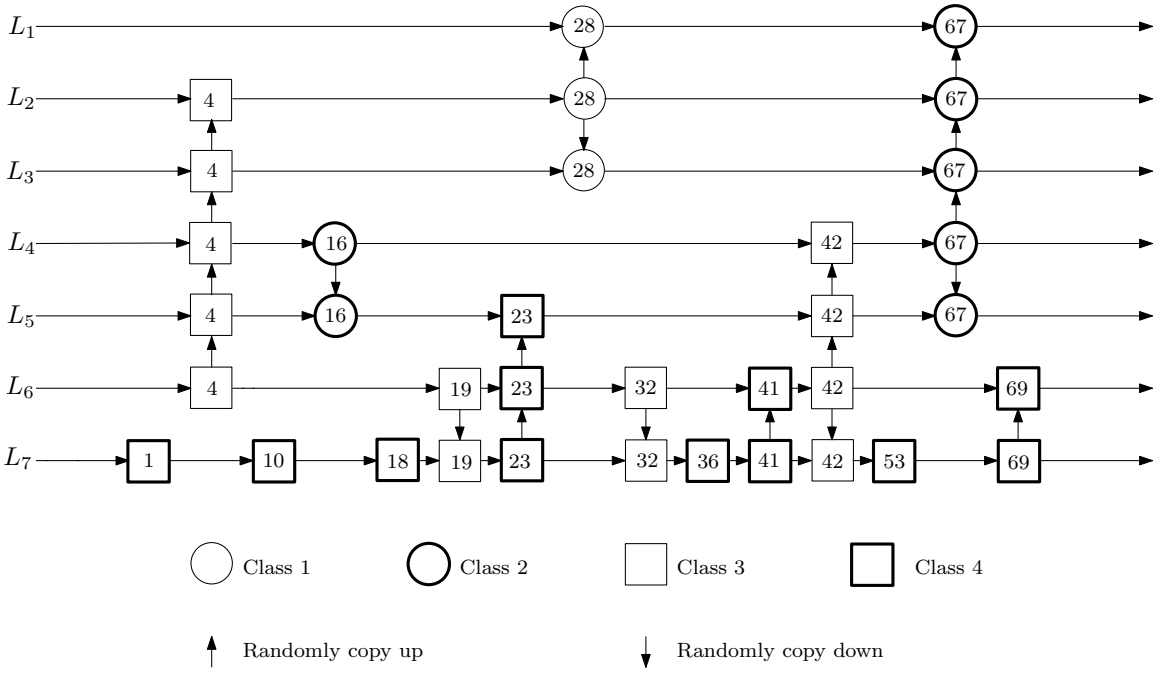


Figure 3.2: Improved BSL

In this improved construction procedure, the newly inserted element does not need to be inserted at the bottommost level and copied up to Level L_2. Instead, each key stays on its default level and is copied up or down randomly. It is emphasized that the importance of the copying-down step is different from that in an original biased skip list. Suppose that each level contains only default keys and random keys are picked up from a lower level. An extreme situation may occur in which one class contains keys in a small range of value, but its adjacent two classes contain keys of different ranges. See Figure 3.3 for an example. The bold line represents the search path for key 69. In such cases, a search procedure may have to traverse back and forth in the entire biased skip list until it finds the target key, which will bring negative effects on the performance of the data structure. Therefore in the new design of the biased skip list, each level can have keys not only from the lower levels but also from higher levels. In this way some improvements in efficiency can be achieved.

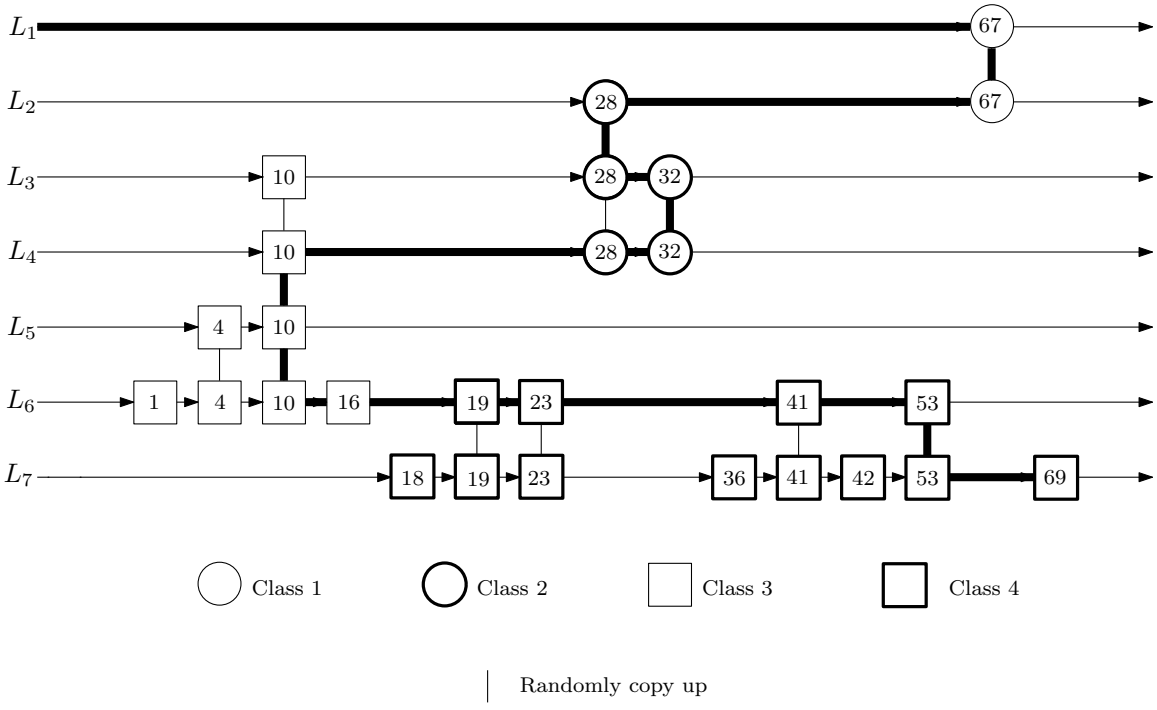


Figure 3.3: An extreme situation without randomly copying keys down

The expected construction time of the improved skip list is $O\big(n\big)$, if the keys are given in sorted order. Inserting all keys from class C_c into level L_c takes $O(|C_c|)$ time. For any $d < c$, level L_{2d} contains keys from class C_d and keys randomly picked up from a lower level, its size is in $O(|C_d|)$, level L_{2d-1} is built from a previous level L_{2d}

so that its size is also $O(|C_d|)$. Since the number of keys which are copied down from a higher level decreases faster than the expected number of keys on the level, it is reasonable to argue the size of each level is $O(|C_i|)$. The number of times that a key being copied up is

$$\sum_{i=1}^{2d} \frac{1}{2^i} \leq \sum_{i=1}^{\infty} \frac{1}{2^i} \leq 1. \tag{3.3}$$

Similarly, the number of times that this key being copied down is also less than 1. The cost of constructing class C_d is thus $O(|C_i|)$. Notice that to copy a key down or to copy it up, we need to go to its left neighbour on the same level, in the aim of finding a link to reach a lower level. And after reaching the lower level, we may need to go to a right neighbour to find the proper position to insert the key. Ergun et al. [28] showed that such traversing time on each level is bounded by $O(1)$ in expectation. Summing up the results mentioned above, the entire construction procedure is done in $O(n)$ expected time.

3.2.2 Lazy Update Scheme

The class assignment scheme fixes the class sizes in a strict manner. Therefore, an insertion will be costly, affecting all the class node tails and causing $O\big(\log n\big)$ adjustments. In this case, the *lazy update scheme* should be applied. To apply the scheme, flexible class sizes are allowed. For any class C_i, its size ranges from 2^{i-1} to 2^{i+1}, with its default size 2^i. Such a lazy update rule makes most insertions or deletions more 'localized' in the data structure. This means, only when the number of keys in a class reaches the upper bound of class size, keys need to be moved from one class to another. Otherwise, the key will be inserted with few readjustments on the structure. A biased skip list with both the lazy update scheme and improved construction is called a *dynamic biased skip list*.

A successful search in a dynamic biased skip list takes $O\big(\log r\big(k\big)\big)$ expected time, and an unsuccessful search takes $O\big(\log n\big)$ time. Although the size of classes are flexible, an existing key k that belongs to class C_d is still guaranteed to be found at level L_{2d}. The number of vertical steps to find k is at most $O(\log r(k))$. Let us bound the number of horizontal links on each level while we go down. With a similar argument to that shown in the analysis for a biased skip list, the expected number

of links that need to be visited until k or a key greater than k is found is at most 2. If k is not found on one level, later actions may be traversing to the left side to find a way down to a lower level, and after the lower level is reached, traversing to the right. Following the same logic in the analysis of the improved construction time, the expected number of such links is constant. After k is found, all the class tail nodes that has a class number smaller than d needs to be updated. The update operations include rewriting their ranks, increasing their class number by 1 and moving their default levels down by 2, and take no more than $O(\log r(k))$ time in expectation, as we analyzed previously. Therefore, the expected time of the entire search operation is $O(d)$, which is $O\big(\log r\big(k\big)\big)$. If the key k is not maintained by the data structure, the expected operation time is $O(\log n)$ since all the $O(\log n)$ levels need to be visited and $O(\log n)$ class tail nodes need to be updated.

As analyzed in the last paragraph, a search operation on key k involves updating operations on all the class tail nodes that has a class number smaller than k's class number before k is accessed. The updating operations include moving the class tail nodes to next class and moving them down by 2 levels. Therefore, the size of each class stays the same after a search operation and will be changed only with insertions or deletions. When a key k is inserted into the dynamic biased skip list, it is given rank 1 and set into Class C_1. Random copies of k are added into Level L_1 and into lower levels determined by a coin-flip result. If the number of nodes in Class C_1 reaches its upper bound, then half of the keys that have larger ranks will be moved to Class C_2 and degraded with default levels moved down in the data structure. Let us carry out the same operations on Class C_2. If a Class C_i reaches its upper bound 2^i, then 2^{i-1} higher ranked keys in Class C_i are moved to class C_{i+1}, with default levels of those keys moved down by 2. This enables Class C_i to return to its default size. This procedure is executed recursively until the size of the current class is under its upper limit.

A costly case may occur when all classes are full, an insertion will bring movements to all classes and the whole data structure is reconstructed. However, it can still be proved [29] that an insertion or a deletion in a dynamic biased skip list costs $O\big(\log r_{\max}\big(k\big)\big)$ amortized time.

3.3 Hybrid Search Algorithm

As discussed in the last subsection, the implementation of a dynamic biased skip list contains the skip list structure and a move-to-front list component, the rank keeper. Inspired by this design, Ergun et al. [29] proposed a hybrid search algorithm but did not explicitly describe how it works. We further implemented the modified algorithm based on the following observation: a regular search procedure starts from the topmost and leftmost key in the skip list, and traverses to the right or down to locate the target value. However, each layer in the dynamic biased skip list is an ordered linked list. If the target is frequently accessed with a large value, a search has to travel for a long distance to reach the target key. To avoid such cases, the search can start from the rank keeper, where keys are linked by rank ascending order. A search operation in this hybrid data structure has two steps:

1. Set a maximum number of links for traversing in rank keeper. The ratio between this number and the default size of Class C_1 is denoted as t.

   The search operation will start looking for the target key in the rank keeper. If the target is found within the maximum number of links, make it the new head in the rank keeper without adjusting the structure of the dynamic biased skip list. Then the procedure ends with a successful search.

2. If the target cannot be found from the previous step, the searching starts from the topmost and leftmost key in the dynamic biased skip list. This is the same as searching in an original skip list.

Notice that it is reasonable to set t under 1. Otherwise, an adverse situation may occur if the target key is not maintained by the first class. For example, if the specific key can not be found in the first step of the hybrid algorithm, it can not be found on the first two levels either(with a high possibility). In this case, we have to start looking for it from the head of dynamic biased skip list and traverse the first two levels again. This is very costly since searching in the rank keeper is a linear search and therefore, it requires more time and leads to less efficient performances. Moreover, it should be avoided to turn the biased skip list search procedure into a searching operation in a move-to-front list. Therefore, the maximum value of t is 1 in our hybrid algorithm.

In addition, Ergun et al. [29] performed 2 sets of experiments with this hybrid search algorithm on a biased skip list, but they did not examine the optimal values of t for different biased skip list configurations. In our study, we further evaluate the performances of this hybrid data structure.

Chapter 4

Experiments

In Ergun et al.'s implementation studies [29], the authors completed the following experiments on biased skip lists:

- Analysis of search times with different Class C_1 default size using synthetic data sets.
- Comparison of the search times between biased skip lists, move-to-front lists, skip lists and binary tries, using synthetic data sets.
- Comparisons of the search times between biased skip lists and move-to-front lists with disabled caches, using synthetic data sets.
- Comparisons of biased skip lists with move-to-front lists and skip lists, using a real-world data set. In this experiment, the operation sequence includes a small portion of insertion operations and while the remaining are search operations.

The dynamic biased skip list was not implemented in this paper. Also the performances of biased skip lists/dynamic biased skip lists are not fully evaluated for the following reasons:

- Insert/delete operations are not completely included.

  The authors in [29] executed one experiment with insertion operations on biased skip list. In this experiment, the $1.5 * 10^4$ insertions were a tiny portion compared with the operation sequence length which is $1.8 * 10^6$. The remaining experiments were done with all keys inserted at the beginning. In addition, the delete operation was not implemented in any experiments.

- The biased skip list is tested with only one real-world data set.

  The biased skip lists experiments were conducted with one real-world data set in [29]. This database is highly biased, with the average rank of all keys being only 25.

- Comparisons between biased skip lists and other data structures are not enough.

  In [29], the biased skip list was compared against the move-to-front lists, original skip lists and binary tries. As mentioned previously, data containers that exhibits the working set property work well for biased data patterns, of which the biased skip list is one. However, move-to-front list suffers an $O(n)$ average running time, skip lists and binary tries do not exhibit the working set property. This means that a comparison between these data structures will not provide a convincing conclusion.

We fully implemented the dynamic biased skip list and tested it using both real-world data sets and synthetic data with different degrees of bias. Its performance was then compared to that of move-to-front lists, skip lists, splay trees, randomized splay trees, W-splay trees and red-black trees. Some data structures were already implemented and their code were available in the public domain. The skip list implementation can be found at [71]. The code for red-black trees is provided by Martinian [2]; and splay tree implementation is provided by Buricea [1]. We implemented the move-to-front lists and dynamic biased skip list ourselves. In addition, based on the implementation of [1], we further implemented the randomized splay trees and W-splay trees.

A list of abbreviations used in our experiment is given below:

- MFL: move-to-front list
- ST: splay tree
- R-ST: randomized splay tree
- W-ST: W-splay tree
- RBT: red-black tree
- SL: skip list
- DBSL: dynamic biased skip list
- H-DBSL: dynamic biased skip list with the hybrid search algorithm described previously

This chapter is organized as follows: in Section 4.1, we introduce some abbreviations used in our discussions, the real-world data sets that are used for testing and other information regarding the experimental setup. In Section 4.2, we describe the experiments for changing the sizes of Class C_1, which follows the experimental procedure outlined in [29]. In Section 4.3, we test different values of t to find the optimal parameters for our data structure. In Section 4.4, we compare move-to-front lists, skip lists, red-black trees, splay trees, randomized splay trees and W-splay trees with dynamic biased skip lists using real world data. Section 4.5 then repeats the comparison experiments using synthetic data of varying degrees of bias measured by the average rank of keys in the sequence.

4.1 Experimental Setup

All the experiments were performed on a machine with an Intel Core i5-3550 processor at 3.30 GHz, with 64KB of L1 Cache, 256KB L2 Cache, 6144KB L3 Cache and 16GB RAM with Physical Address Extension installed on a 32-bit Ubuntu Linux system. The PC runs on Ubuntu 12.10 with a 3.5.0-51-generic kernel. The data structures tested in our experiments are written in C++ and compiled using gcc-4.7.2 with optimization level O3.

We find 8 realistic data sets with different degrees of bias. To determine the bias degree of a data set, we define a data set's rank be calculated as the average rank of each access. The rank of each access is calculated as follows: the first access will be given rank as 1 and inserted into an empty move-to-front list. Following this pattern, if an access is performed on an existing key k, we start from the head of the move-to-front list, and count the number of distinct keys before k is found as the access's rank; otherwise it will be given a rank of 1. Thus we can get the rank of the whole sequence and calculate the data set's rank. Table 4.1 gives a description of these data sets.

The LBL trace data (short) and the Wiki adminship election data show a high degree of bias, with intensive access for a small amount of keys. On the other hand, the Amazon product reviews, Query log data and Pizza requests data sets contain keys with even access possibilities. The remaining data sets, the Accesses to Amazon website, LBL trace data (long) and the Twitter updates, have bias that lie in between

Dataset	Accesses	Distinct keys	Rank	Description
LBL trace data (short) [67]	178,995	1,622	19	TCP packages between the Lawrence Berkeley Laboratory and other websites in 2 hours. Keys are local PC addresses.
Wiki adminship election [51]	994,452	2,931	33	Voting records of candidates for administrator positions. Keys are election IDs.
Accesses to Amazon website [53]	716,064	6,452	283	History of users accessing targeted branch websites of Amazon.com. Keys are user IP addresses.
LBL trace data (long) [68]	7,822,816	13,783	442	TCP packages between the Lawrence Berkeley Laboratory and other websites in 30 days.
Twitter updates [84]	5,108,859	5,179	515	Collection of a portion of twitter users and their updates. Keys are twitter user IDs.
Amazon product reviews [56]	194,439	10,429	1,904	Product reviews on cell phones and accessories. Keys are product IDs.
Query log data [66]	3,558,411	65,516	3,926	A collection consists of web queries collected over three months. Keys are different IP addresses.
Pizza requests [52]	1,323,096	16,736	4,165	A collection of textual requests for a free pizza from the Reddit community. Keys are user IDs.

Table 4.1
Real-world data sets used in experiments

the two extremes. We use these three groups of data to make our experiments credible.

We also conduct experiments with synthetic data. Our synthetic data is generated in the same way as [29], where our operation sequence is of length 10^8 and contains 10^6 unique keys:

1. Randomly generate a sequence S^{data} of 10^6 distinct keys.

2. Generate a sequence S^{rank} representing the rank of each access sequence. Bias

in S^{rank} is simulated with geometric distribution. The geometric distribution is a discrete random sampling distribution which represents the number of failed Bernoulli trials before a success is got. If the probability of success in such a trial is p, then the probability of the x-th trial is the first success is

$$P(k) = p(1-p)^{x-1}, \tag{4.1}$$

for x = 1,2,3...

3. Set the (S_1^{rank})-th key in S^{data} as the first element, and move the key as the first key in S^{data}. The (S_2^{rank})-th key is then set as the second element in the desired sequence, and is moved to the head in S^{data}. This pattern is repeatedly done until we get a sequence of 10^8 operations.

4.2 Analysis of Different C_1 Sizes

As we discussed in the last chapter, the expected search time for an existing key k is bounded by $O\big(d\big)$, where d is the number of the class that k belongs to. Therefore, the default size of Class C_1 may affect the performance of the DBSL since it determines in which class k stays. In this section, we change the default sizes of Class C_1 with different operation sequences.

We first show how the running time changes as the size of Class C_1 increases. We change Class C_1 default size from 2^2 to 2^{11}. Also, to test the performance of update operations thoroughly, we start from an empty DBSL and vary the ratios between insert and delete operations for the operation sequence in our experimental studies. The ratio of deletion operations are set from 0 to 25%. Each access sequence is either an insertion or a deletion, in this way the data structure can be built up with successful insertions. If the insertion is on a key that already exists, only a search operation will be performed.

We select the Wiki adminship election data set as the highly biased data pattern for testing the performance of DBSL. This data set has 2931 distinct keys, $994,452$ operations, and its average rank is 33. The LBL trace data $\big($short$\big)$ that has the same degree of bias, returned an optimal C_1 size as 2^4, which is the same as the result from using the Wiki adminship election data. Therefore, we just put one experimental

result to represent experiments on highly biased data sets. For a less biased database, we pick two data sets: a) the accesses to Amazon website data set, which has 6452 keys, 7,822,816 access requests, rank of 283, and has an optimal class C_1 size of 2^7 and b) the Twitter updates data set, which is ranked 515 with $5,108,859$ operations and 5179 unique keys. The optimal class C_1 size using Twitter updates data sets is 2^7 when the access sequence contains 25% deletions and 2^8 under other circumstances. The result using the long LBL trace data, which is of roughly the same degree of bias, returned 2^7 as an optimal C_1 size, which is the same result as using the Amazon website data set. For data sequences with a low degree of bias, we pick the Amazon product review data set as a representative, which has $194,439$ operations on $10,429$ distinct keys. Our results indicate an optimal C_1 size can optimize the performance of DBSL, and that as the degree of bias increases, the optimal size of C_1 increases as well. All the experiments are executed for 5 times and the final result is calculated as the average value of the 5 results, so the deviation is minimized.

Our results are seen in Figures 4.1, 4.2, 4.3 and 4.4. The vertical axis represents the total run time for each data set, measured in seconds (s), and each line in the chart represents different ratios between insert and delete operations. Results from the rest of data sets can be found in Appendix A.1.

We observe from all the experimental results that, as we increase the size of Class C_1, the total running time will first decrease and then increase. As we discussed in the last section, the operation time is affected from three sub-procedures. We take a search operation for example, assume the target key is k in Class C_i:

1. Locating k in the dynamic biased skip list, including traversing horizontally and vertically.

2. Lifting k to higher levels.

3. Moving down related class tail nodes that has a smaller class number than i.

When the size of C_1 doubles, there will be a fewer number of classes in the whole data structure, thus the total height is reduced by 2. The sub-procedures 2 and 3 will be shortened. However the number of keys on each level also doubles in expectation. Therefore, the performance of the sub-procedure 1 should be analyzed with different C_1 sizes. When the C_1 size increases from a small value, the enlarged C_1 can still

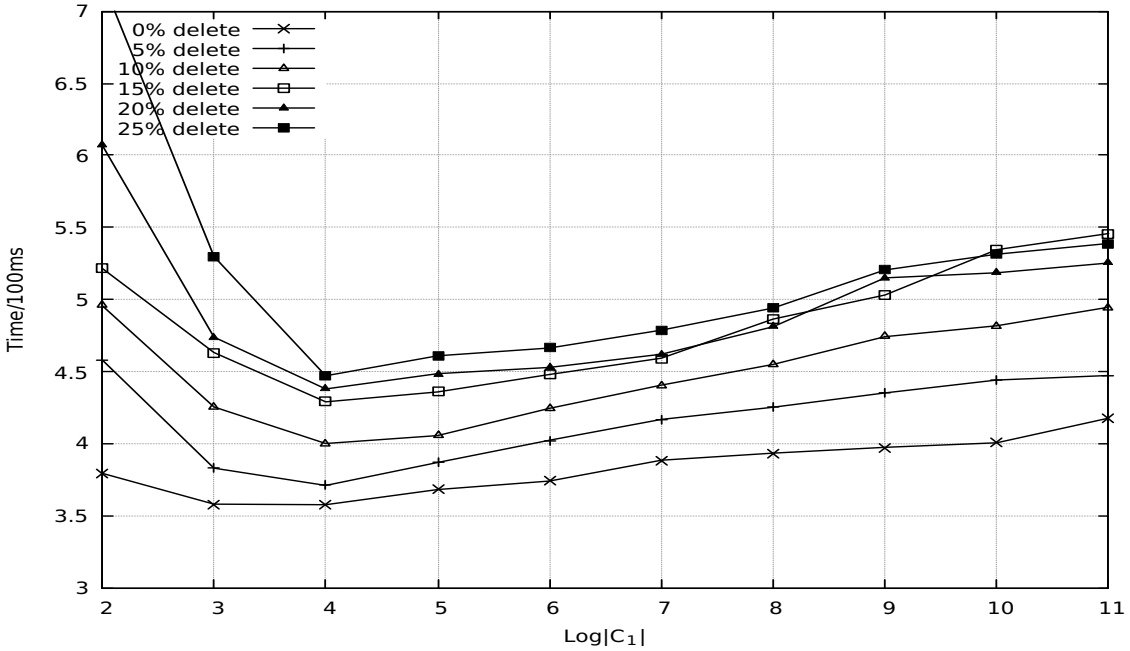


Figure 4.1: Varying Class C_1 sizes on Wiki adminship election data (average rank 33) with varying deletion ratios

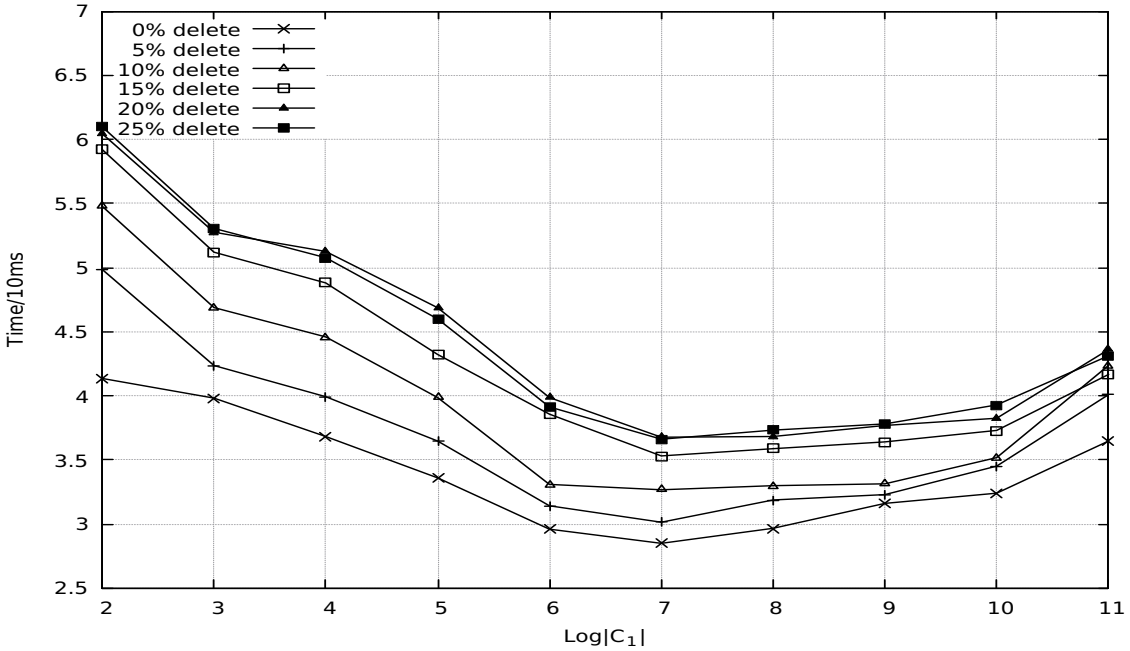


Figure 4.2: Varying Class C_1 sizes on accesses to Amazon website data (average rank 283) with varying deletion ratios

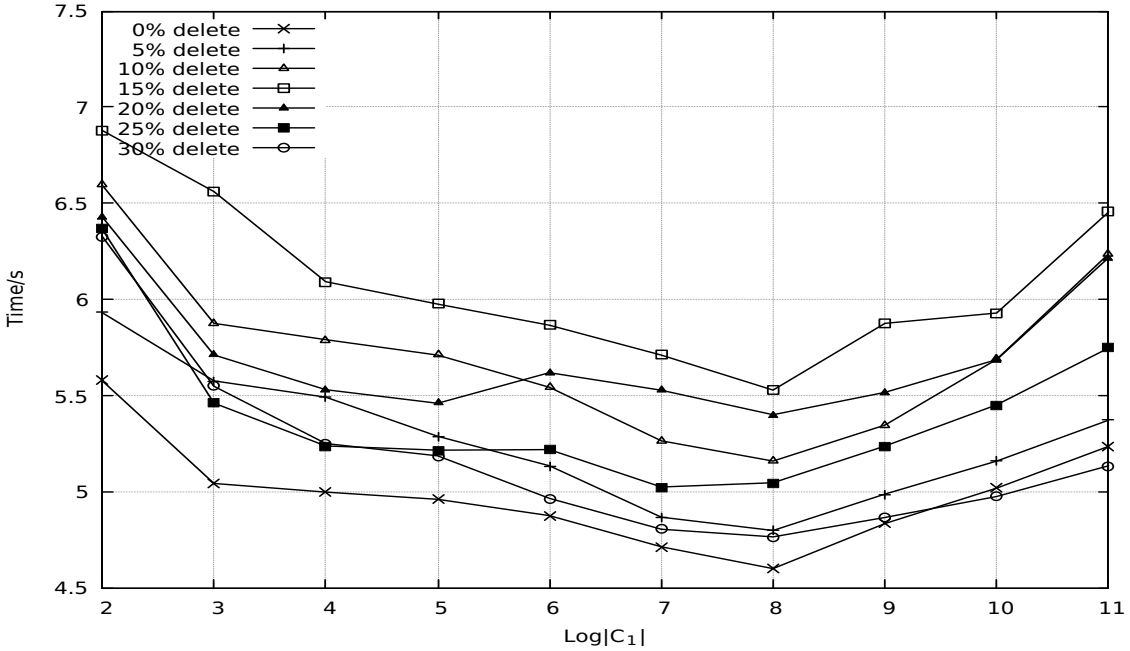


Figure 4.3: Varying Class C_1 sizes on Twitter updates data (average rank 515) with varying deletion ratios

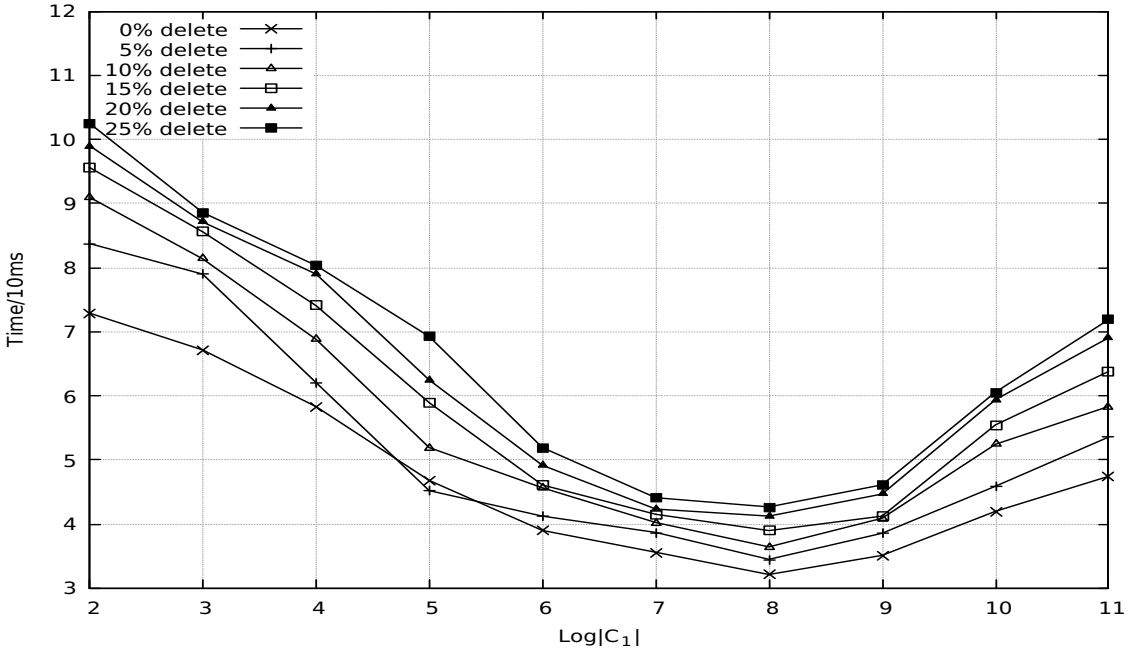


Figure 4.4: Varying Class C_1 sizes on Amazon product reviews (average rank 1,904) with varying deletion ratios

hold a portion of frequently accessed keys. The time of sub-procedure 1 will not be affected too much due to the small number of keys from each class. That explains why the time drops at the beginning, when the size of C_1 increases from small.

We also noticed, running time starts to increase when the default size of Class C_1 passes a 'limit value'. This is because the time spent on the first sub-procedure becomes the most expensive part. For example, if we enlarge the default size of C_1 from 512 to 1024, although it will cause the number of classes reduced by 1, copies of the target key reduced by 2 and number of class tail node to be updated reduced by 1. However, there will be more than 512 keys on the first two levels, more than 1024 keys on the next two levels, in which some 'unpopular' keys are included. This brings unnecessary comparisons in the search procedure. Also notice that the searching on each level is a linear search which suffers a $O(n)$ average running time. Based on this observation, we can conclude, the choice of the C_1 default size can optimize the performance of DBSL.

Figure 4.3 represents an interesting phenomenon: when there are 25% deletion operations, a smaller default size of Class C_1 yields a slightly better result. This is the only data set which has a changing optimal class size when changing the percentage of deletion operations. However, if we continue to increase the deletion operation ratio to 30%, the optimal C_1 size is back to 2^8. The experimental results suggest that, even if the operation sequences can affect the optimal C_1 size, it is a smaller variation compared with the degree of data bias.

The optimal Class C_1 sizes for each data set can be seen in Figure 4.5. Comparing the results with data sets of different degrees of bias, we observe that when the rank of the data set increases, the optimal Class C_1 size increases. Following the same logic described in last paragraph, a small Class C_1 size, which is the optimal choice for a highly biased data pattern, can be extremely expensive with a unbiased data pattern. Since the capacity of each level is low, the overhead for maintaining smaller classes is the most costly part in a search procedure. A larger C_1 size will reduce this kind of overhead for a less biased data set. Therefore, the size of the topmost class can be applied to adjust the configuration of DBSL to best suit the input bias.

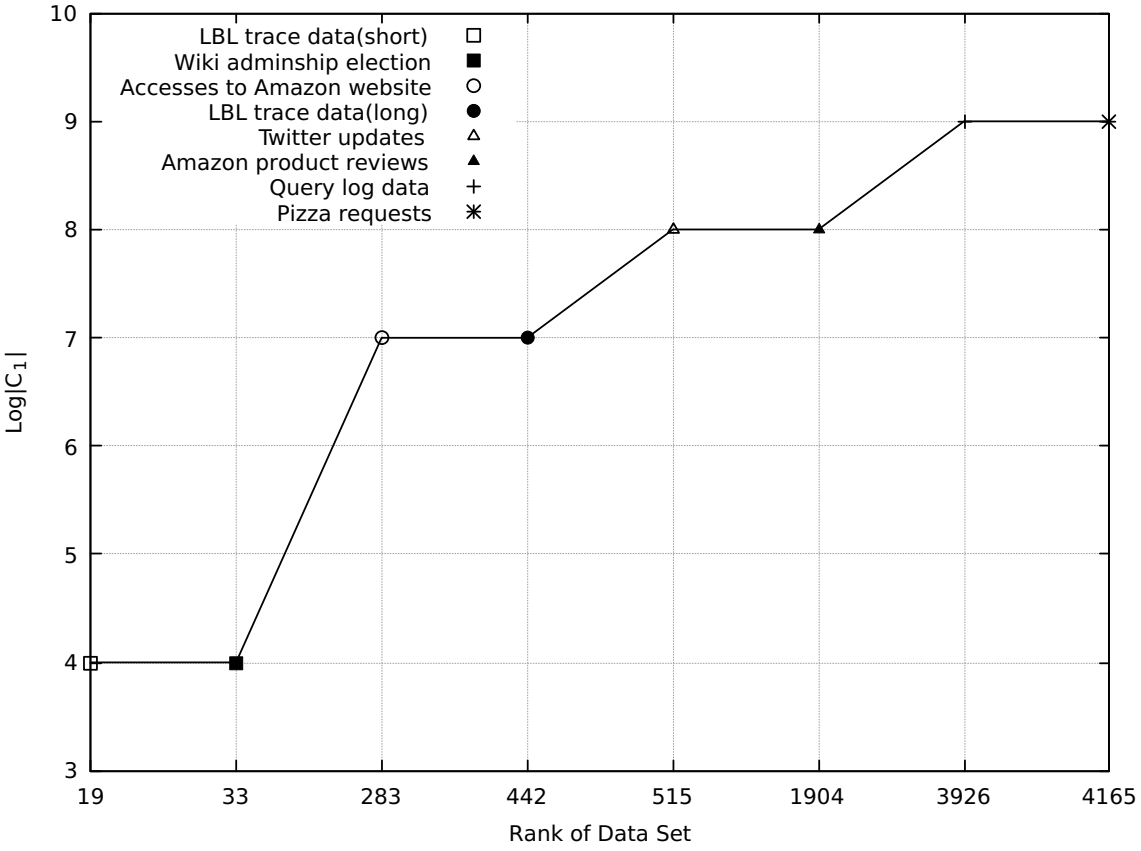
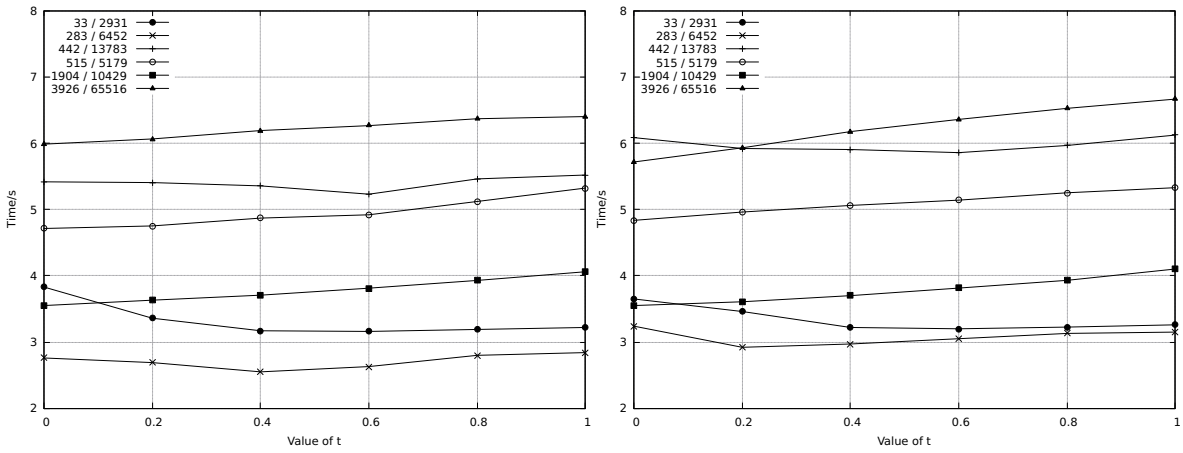


Figure 4.5: Optimal Class C_1 sizes with different data sets

4.3 Analysis of Varying Value of t in H-DBSL

The design of the hybrid search algorithm aims to take advantage of the rank keeper thus speeding up the search time. Next, we evaluate the performance of H-DBSL by changing t. Recall that the largest possible value of t is 1, which means that the maximum length we search in the rank keeper is the value of current Class C_1 default size. In our experimental evaluations, the value of t starts from zero and is incremented by 20% until it reaches the default size of Class C_1. We execute experiments with the size of Class C_1 fixed while changing t. Figure 4.6 displays the results using data sets with all degrees of bias, where '33/2931' means the rank of data is 33 and the number of distinct keys is 2931. In Figure 4.6a, the fixed C_1 size is 128, Figure 4.6b is the result with C_1 size of 512. The experiments using Wiki adminship data (rank 33), Accesses to Amazon data (rank 283) and Amazon product review data (rank 1904) are executed for 100 times, LBL trace data (rank 442), Twitter updates data (rank 515) and Query log data (rank 3926) are executed for 10 times so that all the running times are measured with seconds and fit into one coordinate.

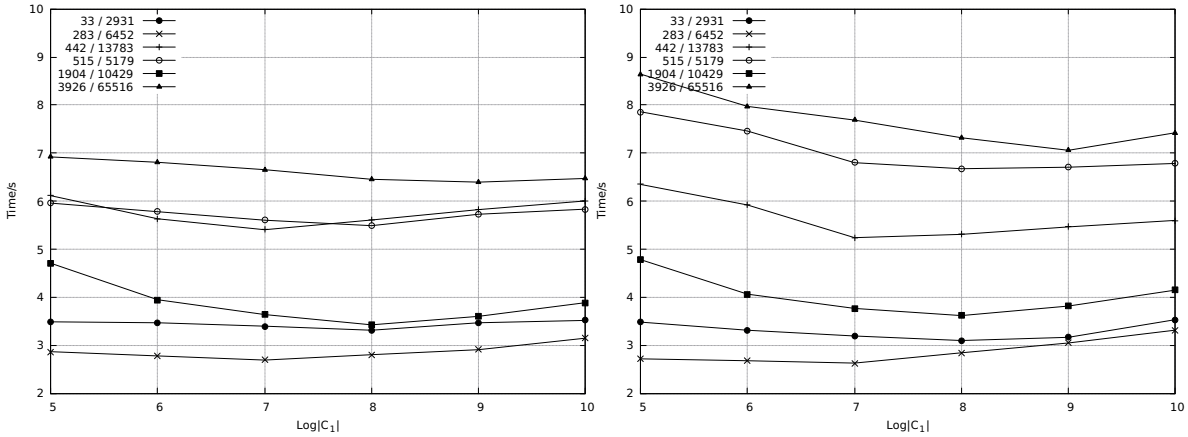


(a) Fixed Class C_1 size is 128 (b) Fixed Class C_1 size is 512

Figure 4.6: Change t on fixed C_1 sizes using different data sets

We observe, for those non-uniformly distributed data patterns, changing t from 0 to 1 will first bring an improvement in performance, and then the running time increases. This indicates that starting the search in rank keeper is beneficial for capturing bias with a biased data set. However, the hybrid algorithm is less efficient when compared to the original search algorithm for data sets with a rank no smaller

than 515. This is easy to explain, since the data is less biased, searching for the first $t \cdot |C_1|$ elements in the rank keeper will more likely result in failure, making the total running time larger.



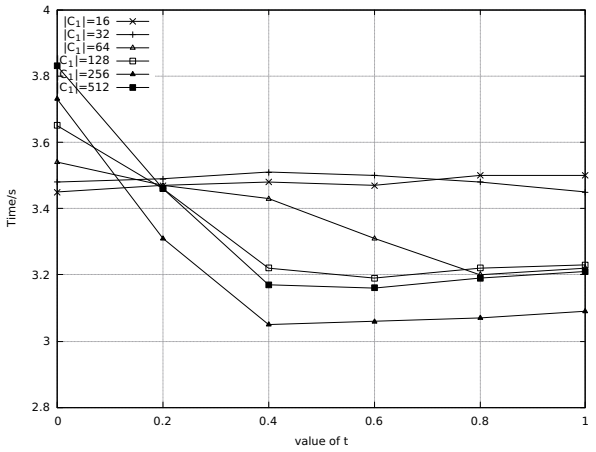
(a) t is 0.2 (b) t is 0.6

Figure 4.7: Change C_1 sizes on fixed t values using different data sets

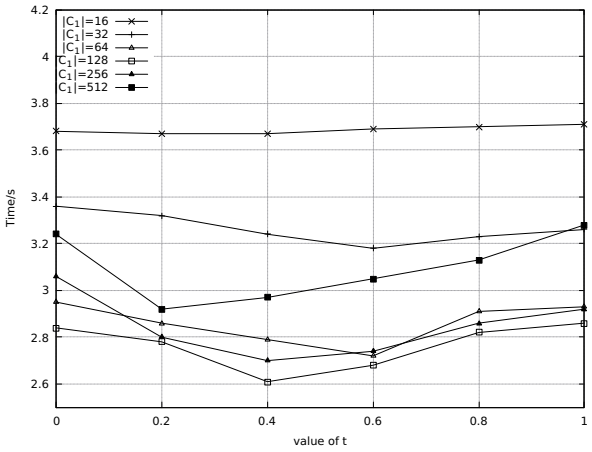
Now we set the ratio t fixed, and change the sizes of C_1, the results can be seen in Figure 4.7. For each data set, no matter how t changes, the optimal size of C_1 does not change. This indicates, for a mostly unbiased data set, start to search in the rank keeper first will bring a small improvement in practice, C_1 is the major factor to affect the performance.

We finally give the results of varying C_1 and varying t, using a data set of different degree of bias. Figure 4.8 represents the result of changing the ratio of t with different class C_1 sizes, using three different data sets. The access sequences contain insert and search operations only. For a highly biased data sequence, if the size of C_1 is small, changing t does not affect the performance of H-DBSL much. However, if the size of C_1 increases, increasing the value of t brings improvement of the performance. Notice that, when $|C_1| = 64$, the optimal value of t is 0.8, and when $|C_1|$ keeps increasing, the optimal value of t drops to 0.4. This is because, when C_1 is smaller, the frequently accessed keys can be found in the rank keeper with a large value of t, since the degree of bias is high and the number of frequently accessed keys is small. In contrast, when the size of C_1 increases, a smaller t ensures that the most popular keys can still be found in the rank keeper in such a hybrid search algorithm. When

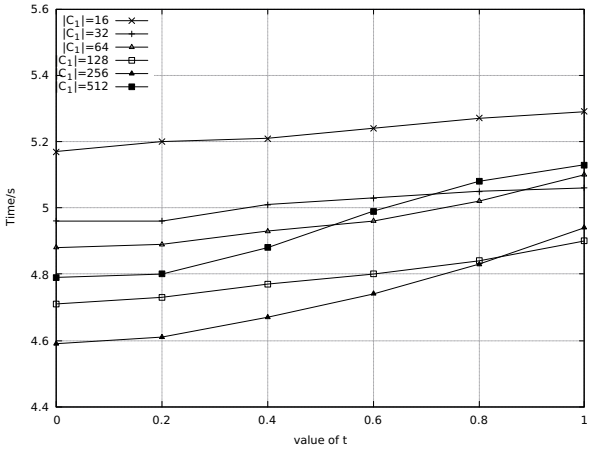
the sequence's degree of bias drops, see Figure 4.8b, this pattern still exists: if $|C_1|$ is small, the hybrid search algorithm performs slightly better than the regular search algorithm. If $|C_1|$ increases from 32 to 512, the optimal t drops from 0.6 to 0.2. But when the rank of data reaches 515, see Figure 4.8c, increasing the value of t decreases efficiency. This is because, since the degree of bias of data decreases, some keys can not be found in the rank keeper. The search algorithm has to start from the head position of the dynamic biased skip list. Therefore the searching in the rank keeper is highly unnecessary and decreases the efficiency of the entire search operation.



(a) Wiki adminship data (average rank 33)



(b) Accesses to Amazon website data (average rank 283)



(c) Twitter updates data (average rank 515)

Figure 4.8: Different t with different C_1 using data sets of different degrees of bias without delete operations

The optimal pairs of C_1 and t for each data set under different deletion ratios

can be seen in Table 4.2. In this Table, '128/0.4' means that C_1 size is 128 and t is 0.4. The best configurations for the highly biased data sets (the first two rows in the Table) are '256/0.4'. When the bias of data is extremely high, this hybrid search algorithm excels since most frequently accessed keys can easily be found with a few links from the head of the rank keeper. Notice that, '256/0.4' is a general good combination for most data sequences. For each data sequence, we also show the different between the running time under the optimal pair and the running time under '256/0.4'. '9.0%' means the operation time using this the current pair is 9.0% faster than the running time using the pair '256/0.4'.

We conduct experiments on the optimal C_1 sizes with varying t and varying deletion ratios. Those experimental results can be seen in Appendix A.2.

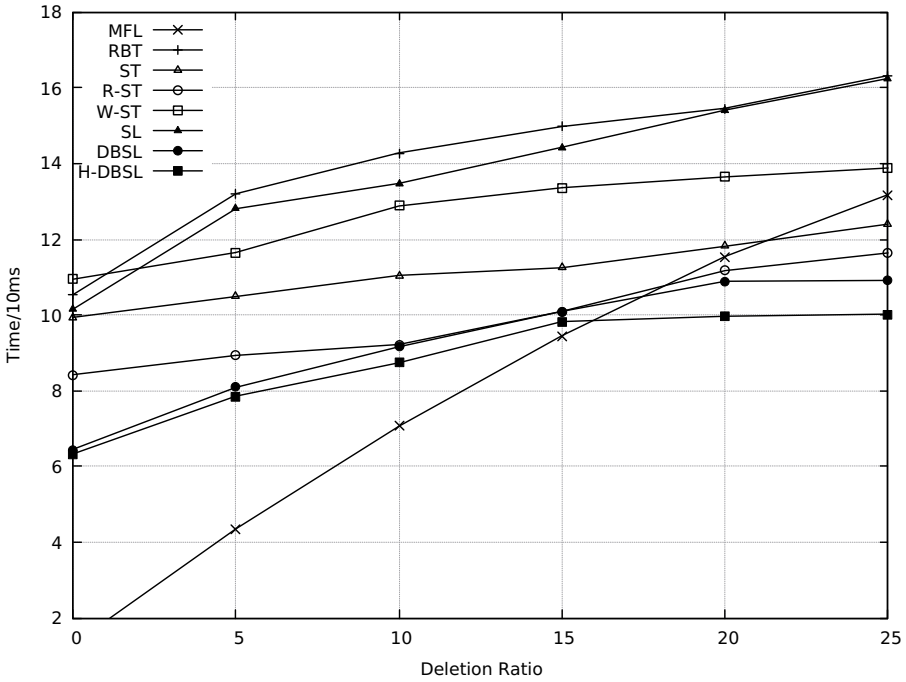
4.4 Comparison of Data Structures using Real-World Data

In this section, we present the experimental results for the comparisons between the MFL, SL, RBT, ST,R-ST,W-ST, DBSL and H-DBSL, using all 8 real-world data sets. Based on the results in Section 4.2 and Section 4.3, we use the best values of Class C_1 and t. The R-ST and W-ST models are set with the optimum splaying parameters determined in [5] and [4]. We execute experiments using highly biased data sets, see Figure 4.9, using data sets with moderate degree of bias, see Figure 4.10, and using data exhibiting low bias, see Figure 4.11.

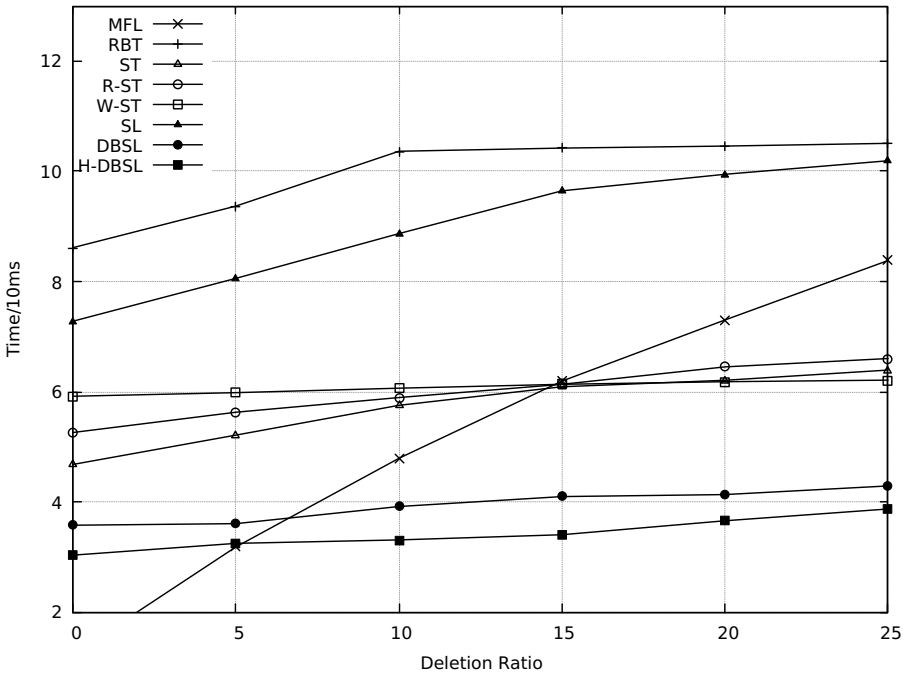
When the data set is highly biased and only search operations are performed, MFL's performance exceeds all the other data structures due to its simple structure. However, when there are more insertions/deletions, the operation time of MFL increases quickly. Except MFL, H-DBSL shows the best performance among all the other data containers with highly biased data. We can observe that the H-DBSL consistently outperforms DBSL at any ratios of deletion operation. The hybrid search algorithm, which takes its advantage from the underlying move-to-front schemed rank keeper, makes search operations more efficient than that in DBSL. Also notice that, the default size of C_1 in H-DBSL is larger than that of DBSL, so that fewer structural changes will take place with more insertions/deletions. DBSL is slightly slower than the H-DBSL but still faster than other binary trees and the original skip list. However, if we continue decrease the degree of bias of data, the two DBSL models become

	Deletion Ratios					
Data sets	0	5%	10%	15%	20%	25%
LBL trace data (short)	256/0.4 (0)	256/0.4 (0)	256/0.4 (0)	256/0.4 (0)	256/0.4 (0)	256/0.4 (0)
Wiki admin-ship election	256/0.4 (0)	256/0.4 (0)	256/0.4 (0)	256/0.4 (0)	256/0.4 (0)	256/0.4 (0)
Accesses to Amazon website	128/0.4 (9.0%)	128/0.4 (8.6%)	128/0.4 (9.1%)	128/0.4 (8.0%)	128/0.2 (7.5%)	128/0.2 (7.7%)
LBL trace data (long)	128/0.6 (3.7%)	128/0.2 (5.7%)	128/0.2 (5.3%)	128/0.2 (7.8%)	128/0.2 (7.5%)	128/0.2 (9.8%)
Twitter up-dates	256/0 (5.0%)	256/0 (7.9%)	256/0 (10.4%)	256/0 (8.1%)	256/0 (11.2%)	128/0 (13.7%)
Amazon product reviews	256/0 (10.8%)	256/0 (12.3%)	256/0 (14.6%)	256/0 (16.5%)	256/0 (15.3%)	256/0 (15.2%)
Query log data	512/0 (12.8%)	512/0 (11.9%)	512/0 (10.1%)	512/0 (11.9%)	512/0 (8.6%)	512/0 (7.9%)
Pizza re-quests	512/0 (12.2%)	512/0 (12.3%)	512/0 (14.2%)	512/0 (16.2%)	512/0 (17.8%)	512/0 (17.9%)

Table 4.2
Optimized combination of C_1 and t for different data sets

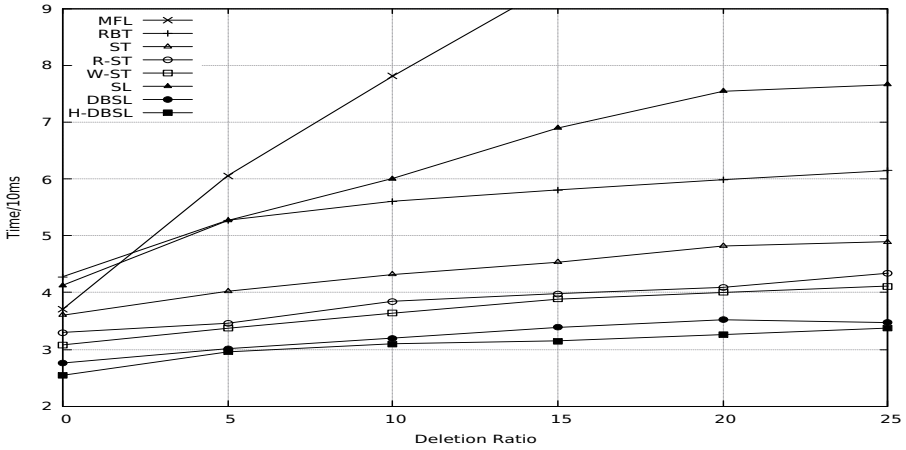


(a) short LBL trace data(rank 19)

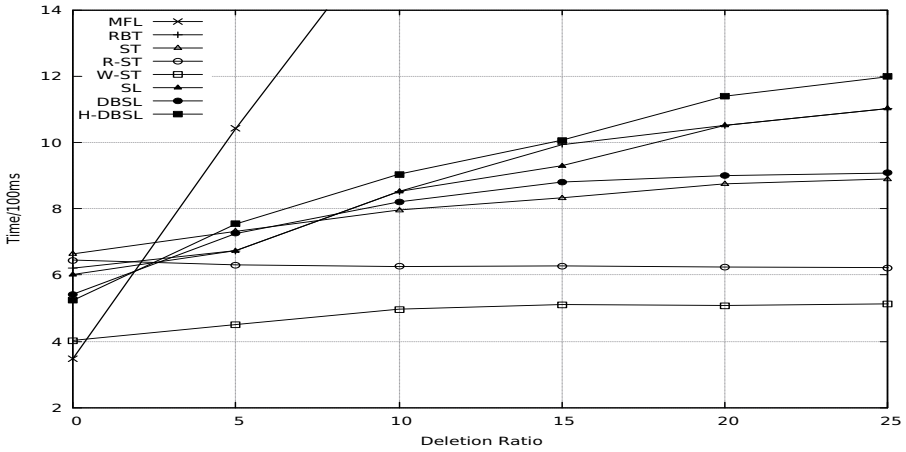


(b) Wiki adminship data (rank 33)

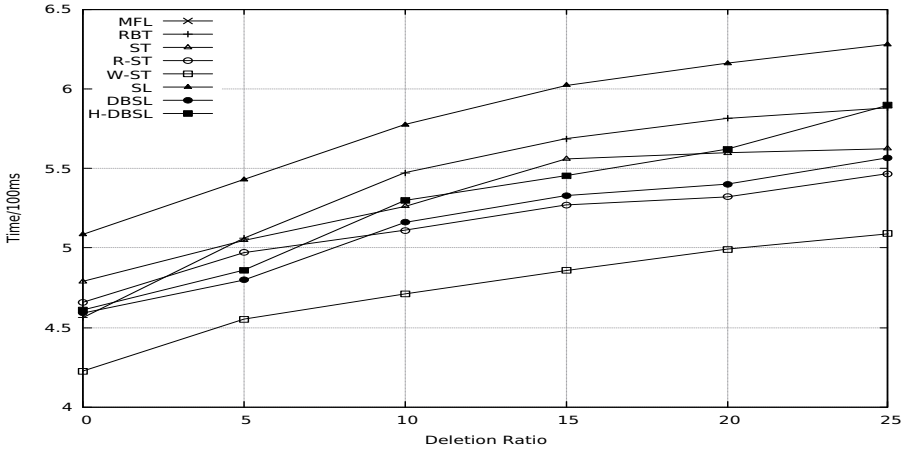
Figure 4.9: Operating times on different data structures with different deletion ratios; data sets are of high degree of bias



(a) Accesses to Amazon website data (rank 283)

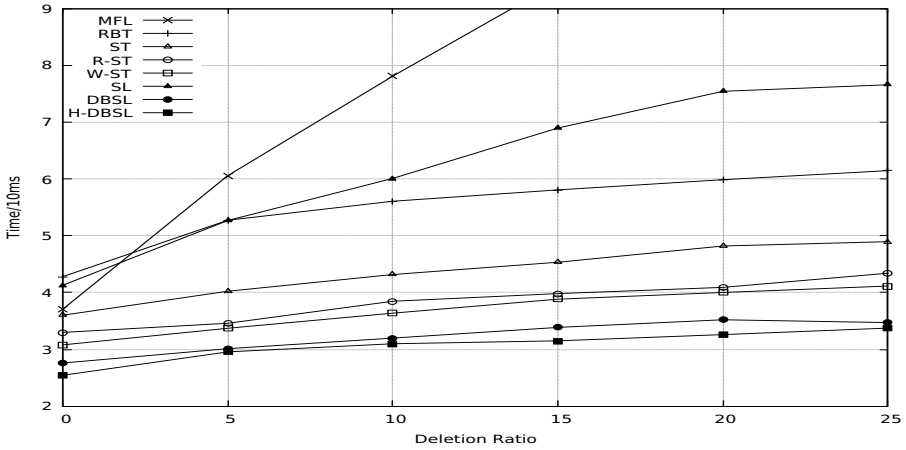


(b) long LBL trace data (rank 442)

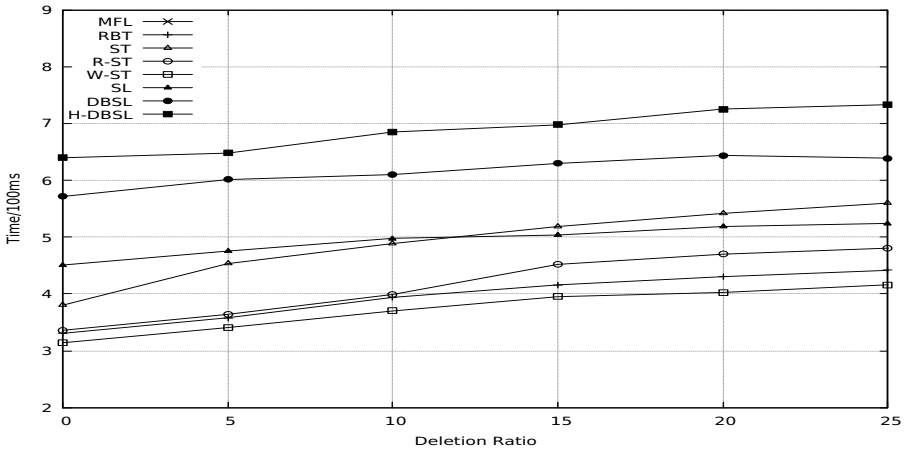


(c) Twitter updates data (rank 515)

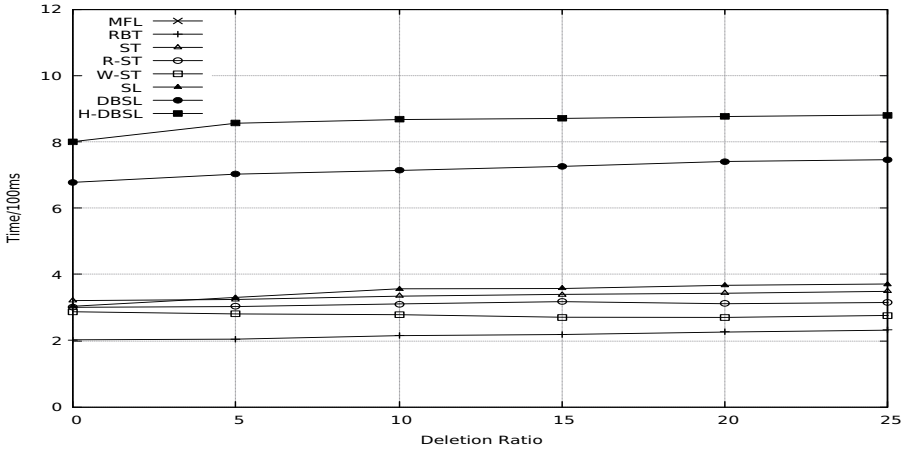
Figure 4.10: Operating times on different data structures with different deletion ratios; data sets are of moderate degree of bias



(a) Amazon product reviews data (rank 1904)



(b) Query log data (rank 3926)



(c) Pizza requests data (rank 4165)

Figure 4.11: Operating times on different data structures with different deletion ratios; data sets are of low degree of bias.

less efficient. When the rank of data reaches 1904, both H-DBSL and DBSL are not competitive with other binary trees or the skip list. As the data is showing less locality of reference, keys are staying evenly on each level, and costly re-adjustments have to be done with every search or unsuccessful insertion. Therefore, maintaining the large number of pointers becomes a big overhead on performance. Also notice that the performance of H-DBSL is even worse than that of DBSL. With non biased access distributions, keys are accessed with uniform possibilities. Therefore, it is highly possible that starting the search in the rank keeper will result in wasted time.

The three splay trees outperform both the skip list and the red-black tree by a wide margin with data of small ranks. We have discussed the working set properties of splay trees in Chapter 3. The splaying operation moves the most frequently accessed keys near the root, from which the splay trees benefit when dealing with highly biased data. Randomized splay trees are better than W-ST and ST, and ST is the slowest among the three. This result indicates the calculation in W-splay algorithm is less efficient than just reducing splaying at some possibility, but is still better than deterministic splaying. As the degree of bias drops, W-ST is superior to R-ST and ST at any degree of bias and at any ratios between insertions and deletions. Both splay tree extensions are more efficient than the original splay tree. This indicates the two methods for reducing the chance of splaying bring efficiency improvements for the deterministic splaying operations in ST. Since the W-ST outperforms R-ST and ST under most circumstances, and is only slightly slower than R-ST when the bias degree of data is extremely high, it is safe to conclude that W-ST is the more favourable choice compared to the other two splay tree models.

SL and RBT are slower than splay trees and DBSL models when data exhibits locality of reference. SL performs better than RBT. However, RBT plateaus earlier than SL. When the degree of bias decreases, RBT exhibits a faster processing time than SL. SL turns out to be the least efficient with sequences that have low degrees of bias among the binary tree structures. This can be explained from the cache behaviors of SL. SL is not cache-friendly [71] since it does not adjust its own structure to changes in locality of reference. To be more specific, even if two related keys are both frequently accessed, they may still stay far away from each other or even

on different pages. This may lead to cache misses due to the jumps between non-contiguous parts of allocated memory. We can also explain the phenomenon by discussing the way in which these data structures are maintained. To maintain a SL, newly inserted keys only have a 50% probability of having just one node in the list structure, and a 50% probability of being continuously copied-up. Additionally, each node has approximately four pointers on average. This means SL has a big overhead for keeping itself 'seemingly adaptive' with query distributions. Pugh [71] theoretically proved that the average number of comparisons performed in a search operation is $\frac{3}{2}\log n + \frac{7}{2}$ in a skip list, while in a binary tree like RBT, the number is just $\log n$ [50]. SL cannot actually guarantee being balanced nor be cache-friendly with a large number of distinct keys, so it has the largest overhead while running. Notice that, when data pattern becomes almost non-biased, RBT turns out to be as competitive as W-ST. Results from [69, 83, 4, 62, 73, 78] also indicate that the RBT is superior for sequences with a low degree of bias: with the balancing scheme, the total height of the tree is maintained in $O\big(\log n\big)$.

We also provide the performances of each data structure under different deletion operation ratios in Appendix A.3.

4.5 Comparison with Data Structures using Generated Data

We follow the same pattern of experiments from last section, except now we use the synthetic data sets. The average rank of the data sets changes from 5 to 500. Based on previous experimental results, although the optimal default size of Class C_1 in DBSL changes with the degree of bias, the size of 128 works well overall for data sets with average rank lower than 500. Similarly, in the H-DBSL, the value of t is 0.4 and default size of C_1 is 256 for generic good performances. Thus, in the following experiments, we always apply $|C_1| = 128$ with DBSL and $|C_1| = 256, t = 0.4$ with H-DBSL.

We first test the search operations only. Running times of different data structures are shown in Figure 4.12. Most self-adjusting data structures perform well when the rank of data set is low. We present a partly enlarged figure of comparisons for the case where the average rank of the data is lower than 200, in Figure 4.13.

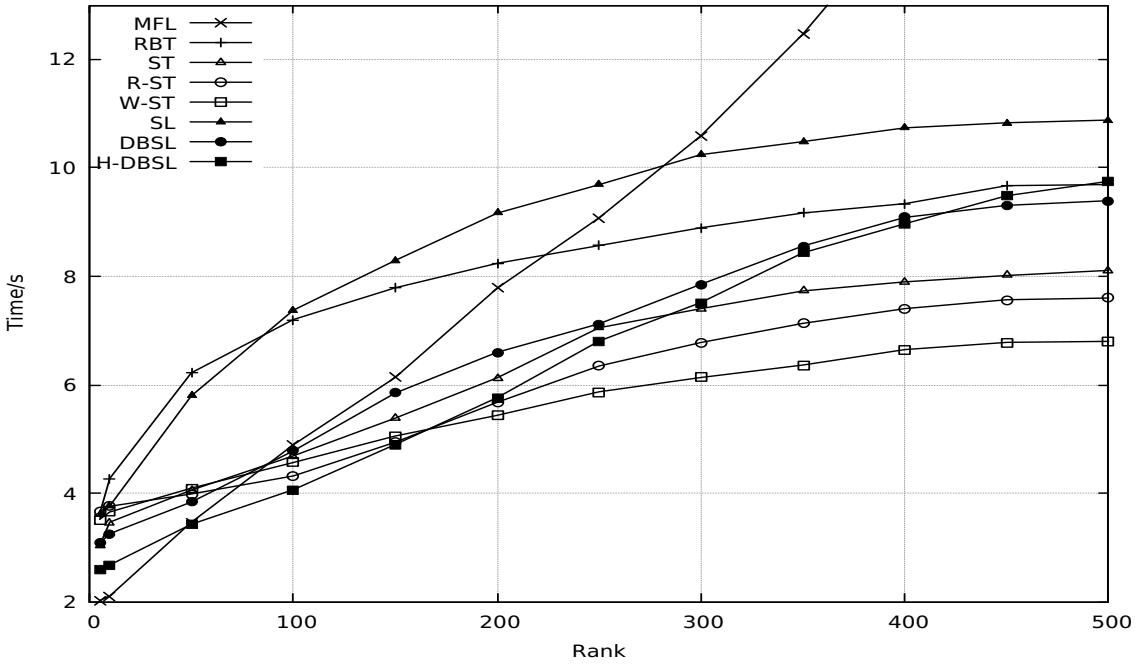


Figure 4.12: 10^8 searches on different data structures

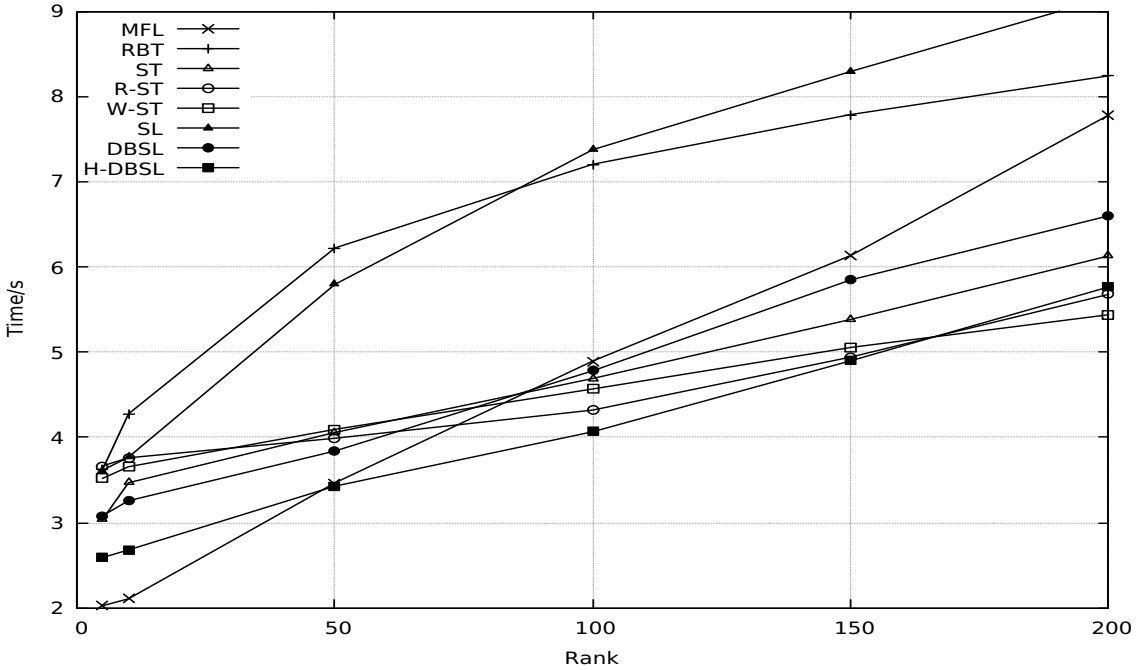


Figure 4.13: 10^8 searches on different data structures; rank of data ranges from 5 to 200

When the rank of data is under 100, MFL returns the best performance as compared to any other data containers. The running time of MFL increases dramatically when the degree of bias of the access sequence drops. When the rank of data is larger than 300, MFL is not competitive at all. The relationship between the rank of data and MFL's running time can be recognized as linear relationship. An access sequence in a MFL takes $O(r(k))$ time for searching for an existing key k, and $O(n)$ time for a non-existing key, where n is the total number of stored distinct keys. We believe that if we introduce more insertion/deletion operations into the query data, the performance of MFL will be even worse.

The DBSL and H-DBSL runs second to MFL with highly biased data, but are exceeded marginally by W-ST when data rank increases to 50. H-DBSL loses its superiority when data rank is larger than 100. Both DBSL and H-DBSL outperform the initial skip list at all rank degrees, which indicates that the maintaining of extra pointers in DBSL models can bring efficiency improvement in practice. R-ST is slightly faster than ST when data rank is small. When the rank of data is under 50, W-ST is less competitive as St or R-ST. However, W-ST has a better performance than both of the other two splay trees when the rank of data is higher. This indicates, the time spent on evaluating the current working set in W-splay algorithm is less than that spent on random splayings or deterministic splayings. RBT and SL are not as efficient as other data structures. RBT processes data at least one order of magnitude slower than SL. But we still believe RBT can provide a better result than SL since the running time of RBT plateaus when rank is 400 while the running time of SL keeps increasing. The figure also indicates that, even though each of the data structures $\big($except MFL$\big)$ has a logarithmic pattern between the running time and the rank of data, performances of STs and DBSLs are slightly better than those of SL and RBT. This is because, accessing an existing key k in STs and DBSLs takes $O(\log r(k))$ time, while in SL and RBT it takes $O(\log n)$ time. In highly biased data sequences, $r(k)$ is often smaller than n, and therefore STs and DBSLs are preferred under such circumstances.

In the following experiments, we set 10% of the query operations as deletions. The results are shown in Figure 4.14. Figure 4.15 is a partly enlarged figure of 4.14, where the rank of data is between 5 to 200.

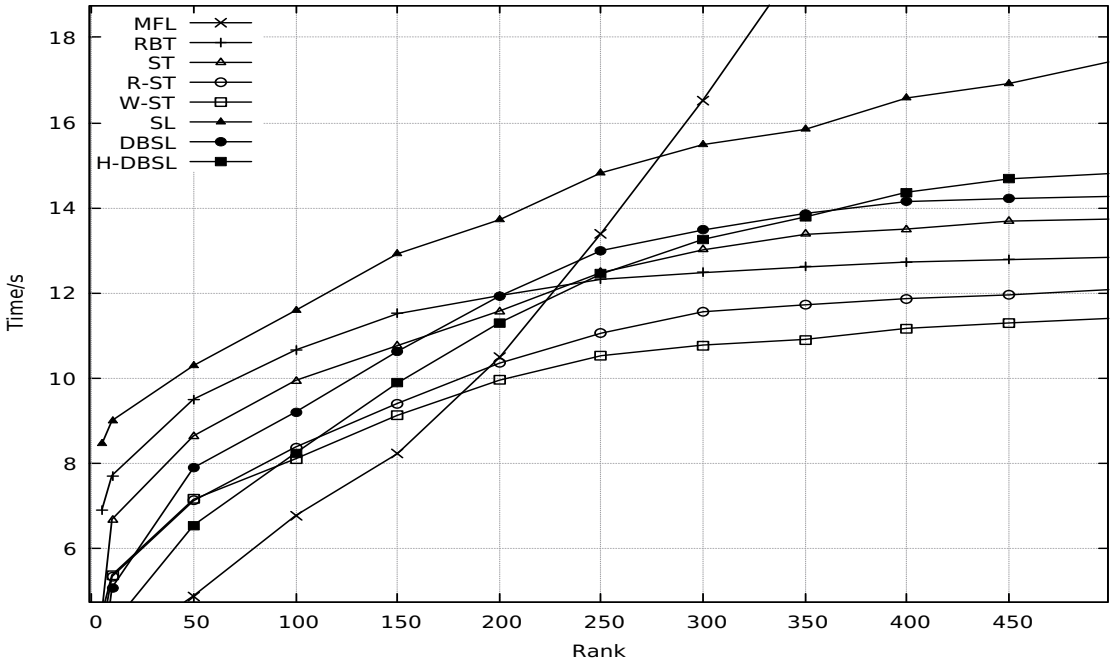


Figure 4.14: 10^8 requests including 10% deletions on different data structures;

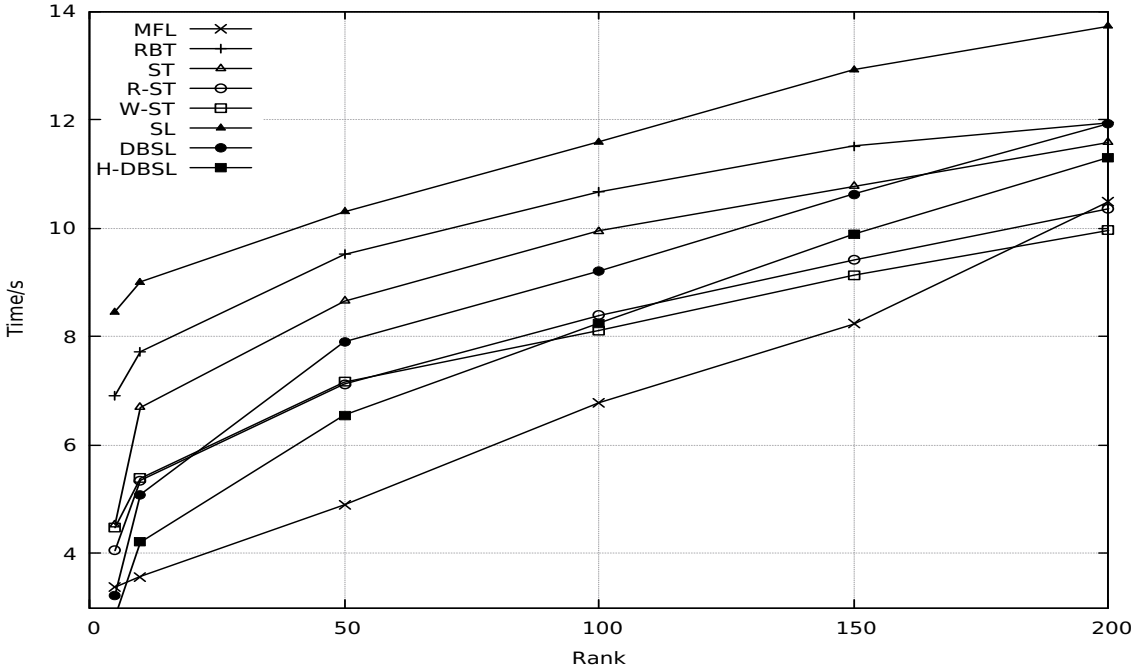


Figure 4.15: 10^8 requests including 10% deletions on different data structures; rank of data ranges from 5 to 200

As expected, if we decrease the degree of bias in a data set and introduce more insertion/deletion requests, MFL's performance is not competitive with any other data structures. The performances of H-DBSL and DBSL are only slightly better than splay trees when data rank is small. H-DBSL catches up with original DBSL before data rank reaches 400, which is earlier than that shown in Figure 4.12.

Another observation is that RBT presents as a favourable choice over ST and SL. This suggests RBT may be an ideal tool for dealing with continual update operations. We carry out experiments on data set that includes 20% deletion operations. The results can be seen in Figure 4.16 and Figure 4.17.

The performance of MFL is rather bad, as we expected. The experimental result depicts that RBT outperforms the other data structures except W-ST. Although the implementation of RBT is complicated due to the large amount of edge cases, RBT is extremely efficient when insert/delete operations are relatively frequent. In a RBT, the shape of the tree is constrained at all times, consequently the height of the tree is bounded under $O\big(\log n\big)$ no matter what kind of operation is applied. In this way RBT make less structural changes to balance itself, which makes it faster to respond to insertions or deletions. This implies that RBT is the best choice for dealing with a number of insert/delete operations if the bias degree of data is unknown. SL is not comparable with RBT or ST at any deletion ratios. In such a random skip list, an insertion has to spend time on generating random bits for making copy-up decisions. This prevents SL from being a comparable alternative to other binary search structures.

W-ST and R-ST both inherit the working set property from ST. ST must rotate with every single operation due to the mandatory splaying scheme. In contrast, R-ST reduces the times of splaying by a random factor and W-ST reduces splaying by observing current working set. These two methods both improve the performance of regular splay trees.

We observe that DBSL favours an overall competitive ability for dealing with biased data patterns, especially with frequent updating operations. H-DBSL catches up with DBSL's performance when rank of data reaches 300. Summing up the comparisons of H-DBSL and DBSL from Figure 4.12, 4.14, and 4.16, we suggest using DBSL when the update operations are frequent. Also, we consider the running times

of DBSL and W-ST when rank of data is 500. Although DBSL is slower under any deletion ratios, the difference from Figure 4.16 is noticeable smaller than that in Figure 4.12. This indicates the lazy-updating scheme can bring improvement in practice, especially with frequent updating operations.

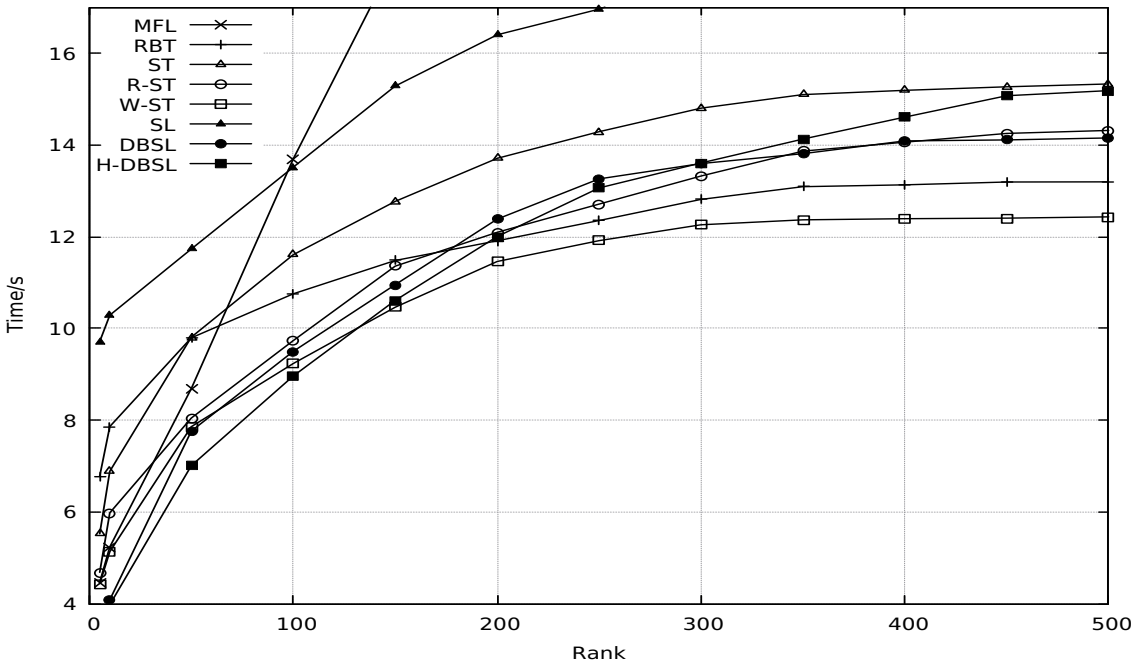


Figure 4.16: 10^8 requests including 20% deletions on different data structures;

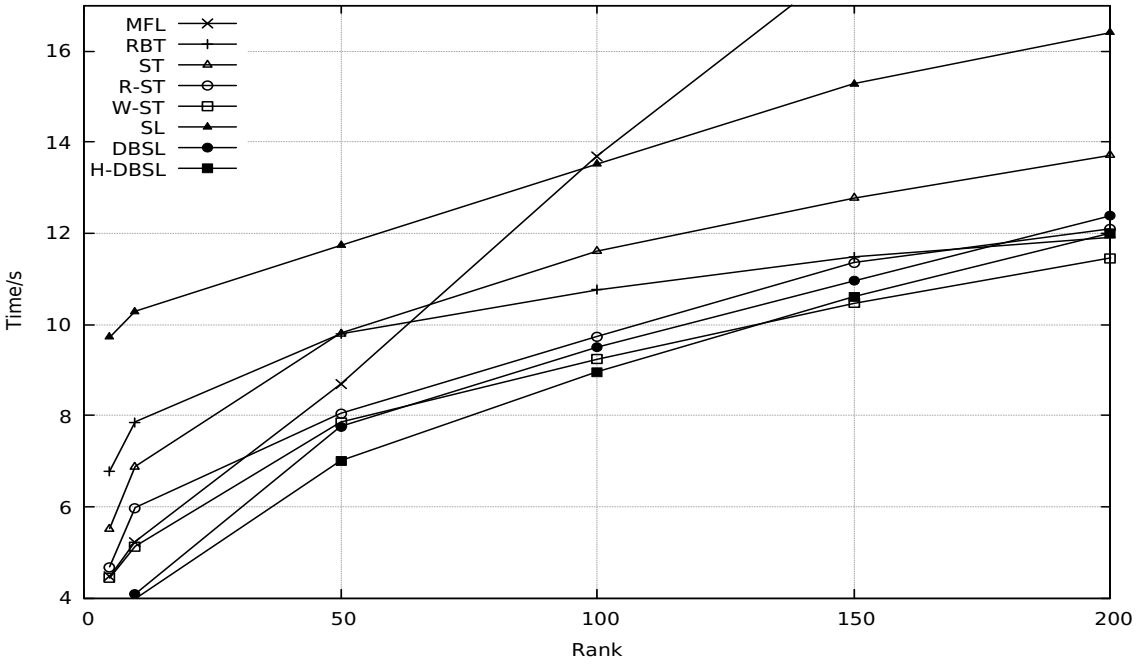


Figure 4.17: 10^8 requests including 20% deletions on different data structures; rank of data ranges from 5 to 200

Chapter 5

Conclusions

In our work, we mainly studied the dynamic biased skip list and executed experiments to evaluate its performance. The dynamic data structure maintains a move-to-front list component to store the most recently accessed keys, and moves the frequently needed keys to higher levels. Thus, those keys can be accessed faster in the near future. The dynamic biased skip list supports $O\big(\log r\big(k\big)\big)$ amortized time with each successful search, insertion and deletion, and $O\big(\log n\big)$ amortized time for a failed search, where $r\big(k\big)$ is the number of distinct keys accessed after the last time k was accessed, and n is the total number of keys in the structure.

We implemented the dynamic biased skip list first. Then we fully evaluated its performances using both synthetic and real-world data sets. We also compared DBSL with additional self-adjusting data structures, such as move-to-front lists, skip lists and red-black trees. In addition, we noticed that splay trees have working set properties to quickly access biased data patterns, so we included comparison with splay trees, randomized splay trees, and W-splay trees as well.

Our research and experimental results suggest that when dealing with highly biased data sets with only search operations, DBSL and H-DBSL work better than the tree structures and are only slightly slower than MFL. If we introduce some insert and delete operations into the data set, DBSL models excel. Based on the results, we can safely conclude, the H-DBSL can take great advantage from the hybrid algorithm with data exhibits high bias. If the degree of bias drops, although DBSL is not competitive against splay trees, it still supports a faster operation time than RBT and SL. However, if the data pattern exhibits no bias, the maintenance of DBSL or H-DBSL will bring the largest overhead in performance and the two models are no longer comparable with the other data structures.

The hybrid search algorithm outperforms the regular search algorithm with data showing considerable bias. On the contrary, the hybrid data structure suffers the

worst running time with uniformly distributed patterns, since searching in the move-to-front list will likely end in a failure.

W-ST is overall better than the other two splay tree models. Only when bias is extremely high will the performance of W-ST be worse than that of R-ST and ST. RBT is capable of doing frequent insert/delete operations, so we suggest using it to deal with data sets that need to be intensively updated. SL is not competitive with ST in all of our experiments, and is only slightly better than RBT with highly biased sequences containing solely search operations, although it supports search, insert, delete operations in $O\big(\log n\big)$ expected time.

Further work can be done based on this thesis. For example, we determined the optimum Class C_1 sizes of DBSL dealing with different data sets. However, without the knowledge of the data's degree of bias, DBSL cannot work under the best configuration. Thus, how to capture the bias of data while running can be future work. Also, more types of distributions, like the frequency distribution [54], the Zipfian distribution [70], and the Levy distribution [7] can be used to simulate locality of reference and evaluate the performance of dynamic biased skip list.

Bibliography

[1] Lecture on splay trees. `http://software.ucv.ro/~mburicea/lab7ASD.pdf`.

[2] Red-black tree by emin. `http://web.mit.edu/~emin/www.old/source_code/red_black_tree/`.

[3] Georgy Adelson-Velsky and Evgenii Landis. An algorithm for the organization of information. In *Proceedings of the USSR Academy of Sciences*, volume 3, pages 1259–1263, 1962.

[4] Timo Aho, Tapio Elomaa, and Jussi Kujala. Reducing splaying by taking advantage of working sets. In *International Workshop on Experimental and Efficient Algorithms*, pages 1–13. Springer, 2008.

[5] Susanne Albers and Marek Karpinski. Randomized splay trees: theoretical and experimental results. *Information Processing Letters*, 81(4):213–221, 2002.

[6] Virgilio Almeida, Azer Bestavros, Mark Crovella, and Adriana De Oliveira. Characterizing reference locality in the www. In *Parallel and Distributed Information Systems, 1996., Fourth International Conference on*, pages 92–103. IEEE, 1996.

[7] Tim Althoff, Cristian Danescu-Niculescu-Mizil, and Dan Jurafsky. How to ask for a favor: A case study on the success of altruistic requests. In *ICWSM*, 2014.

[8] Cecilia R Aragon and Raimund G Seidel. Randomized search trees. In *Foundations of Computer Science, 1989., 30th Annual Symposium on*, pages 540–545. IEEE, 1989.

[9] Martin F Arlitt and Carey L Williamson. Internet web servers: Workload characterization and performance implications. *IEEE/ACM Transactions on networking*, 5(5):631–645, 1997.

[10] James Aspnes and Gauri Shah. Skip graphs. *Acm transactions on algorithms (talg)*, 3(4):37, 2007.

[11] Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Osama Khan, Lea Kissner, Zachary Peterson, and Dawn Song. Remote data checking using provable data possession. *ACM Transactions on Information and System Security (TISSEC)*, 14(1):12, 2011.

[12] Amitabha Bagchi, Adam L Buchsbaum, and Michael T Goodrich. Biased skip lists. *Algorithmica*, 42(1):31–48, 2005.

[13] Paul Barford and Mark Crovella. Generating representative web workloads for network and server performance evaluation. In *ACM SIGMETRICS Performance Evaluation Review*, volume 26, pages 151–160. ACM, 1998.

[14] Rudolf Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta informatica*, 1(4):290–306, 1972.

[15] Samuel W Bent, Daniel D Sleator, and Robert E Tarjan. Biased search trees. *SIAM Journal on Computing*, 14(3):545–568, 1985.

[16] Paul E Black. Dictionary of algorithms and data structures. *NISTIR*, 1998.

[17] Prosenjit Bose, Karim Douïeb, and Pat Morin. Skip lift: A probabilistic alternative to red–black trees. *Journal of Discrete Algorithms*, 14:13–20, 2012.

[18] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and zipf-like distributions: Evidence and implications. In *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 126–134. IEEE, 1999.

[19] Herv Brnnimann and Jyrki Katajainen. Efficiency of various forms of red-black trees. *CPH STL Report*, 2:2006, 2006.

[20] Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. 1994.

[21] Thomas Clouser, Mikhail Nesterenko, and Christian Scheideler. Tiara: A self-stabilizing deterministic skip list. In *Symposium on Self-Stabilizing Systems*, pages 124–140. Springer, 2008.

[22] Thomas H Cormen. *Introduction to algorithms*. MIT press, 2009.

[23] Robert Courts. Improving locality of reference in a garbage-collecting memory management system. *Communications of the ACM*, 31(9):1128–1138, 1988.

[24] Tyler Crain, Vincent Gramoli, and Michel Raynal. Brief announcement: a contention-friendly, non-blocking skip list. *Distributed Computing*, pages 423–424, 2012.

[25] Tyler Crain, Vincent Gramoli, and Michel Raynal. A speculation-friendly binary search tree. *Acm Sigplan Notices*, 47(8):161–170, 2012.

[26] Peter J. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, (1):64–84, 1980.

[27] Ian Dick, Alan Fekete, and Vincent Gramoli. A skip list for multicore. *Concurrency and Computation: Practice and Experience*, 29(4), 2017.

[28] Funda Ergun, S Cenk Sahinalp, Jonathan Sharp, and Rakesh Sinha. Biased dictionaries with fast insert/deletes. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 483–491. ACM, 2001.

[29] Funda Ergun, S Cenk Şahinalp, Jonathan Sharp, and Rakesh K Sinha. Biased skip lists for highly skewed access patterns. In *Workshop on Algorithm Engineering and Experimentation*, pages 216–229. Springer, 2001.

[30] C Chris Erway, Alptekin Küpçü, Charalampos Papamanthou, and Roberto Tamassia. Dynamic provable data possession. *ACM Transactions on Information and System Security (TISSEC)*, 17(4):15, 2015.

[31] David N Etim. Comparison of skip list algorithms to alternative data structures. *Int'l Journal of Computing, Communacations and Instrumentation*, 3(2):280–283, 2016.

[32] Rodrigo Fonseca, Virgılio Almeida, Mark Crovella, and Bruno Abrahao. On the intrinsic locality properties of web reference streams. In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, volume 1, pages 448–458. IEEE, 2003.

[33] Richard M Fujimoto. Time warp on a shared memory multiprocessor. Technical report, UTAH UNIV SALT LAKE CITY SCHOOL OF COMPUTING, 1989.

[34] Tingjian Ge and Stan Zdonik. A skip-list approach for efficiently processing forecasting queries. *Proceedings of the VLDB Endowment*, 1(1):984–995, 2008.

[35] Daniel Golovin. The b-skip-list: A simpler uniquely represented alternative to b-trees. *arXiv preprint arXiv:1005.0662*, 2010.

[36] Dennis Grinberg, Sivaramakrishnan Rajagopalan, Ramarathnam Venkatesan, and Victor K Wei. Splay trees for data compression. 1995.

[37] Dirk Grunwald, Benjamin Zorn, and Robert Henderson. Improving the cache locality of memory allocation. In *ACM SIGPLAN Notices*, volume 28, pages 177–186. ACM, 1993.

[38] Nicholas JA Harvey, Michael B Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. Skipnet: A scalable overlay network with practical locality properties. *networks*, 34(38), 2003.

[39] Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 207–216. ACM, 2008.

[40] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101. ACM, 2003.

[41] John Iacono. Alternatives to splay trees with o (log n) worst-case access times. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 516–522. Society for Industrial and Applied Mathematics, 2001.

[42] Predrag R Jelenkovi. Asymptotic approximation of the move-to-front search cost distribution and least-recently used caching fault probabilities. *Annals of Applied Probability*, pages 430–464, 1999.

[43] Predrag R Jelenkovi and Ana Radovanovi. Least-recently-used caching with dependent requests. *Theoretical computer science*, 326(1):293–327, 2004.

[44] Shudong Jin and Azer Bestavros. Greedydual web caching algorithm: exploiting the two sources of temporal locality in web request streams. *Computer Communications*, 24(2):174–183, 2001.

[45] Douglas W. Jones. Application of splay trees to data compression. *Communications of the ACM*, 31(8):996–1007, 1988.

[46] Richard WM Jones and Paul HJ Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Proceedings of the 3rd International Workshop on Automatic Debugging; 1997 (AADEBUG-97)*, number 001, pages 13–26. Linköping University Electronic Press, 1997.

[47] Mahmut Kandemir, J Ramanujam, and Alok Choudhary. Improving cache locality by a combination of loop and data transformations. *IEEE Transactions on Computers*, 48(2):159–167, 1999.

[48] Peter Kirschenhofer, Conrado Martínez, and Helmut Prodinger. Analysis of an optimized search algorithm for skip lists. *Theoretical Computer Science*, 144(1-2):199–220, 1995.

[49] Peter Kirschenhofer and Helmut Prodinger. The path length of random skip lists. *Acta Informatica*, 31(8):775–792, 1994.

[50] Paul C Kocher. On certificate revocation and validation. In *International Conference on Financial Cryptography*, pages 172–177. Springer, 1998.

[51] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data/wiki-Elec.html`.

[52] Justin Levandoski, David Lomet, Sudipta Sengupta, Adrian Birka, and Cristian Diaconu. Indexing on modern hardware: Hekaton and beyond. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 717–720. ACM, 2014.

[53] M. Lichman. UCI machine learning repository. `http://archive.ics.uci.edu/ml/datasets`, 2013.

[54] Alfred J Lotka. The frequency distribution of scientific productivity. *Journal of the Washington academy of sciences*, 16(12):317–323, 1926.

[55] Darko Marinov and Sarfraz Khurshid. Testera: A novel framework for automated testing of java programs. In *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, pages 22–31. IEEE, 2001.

[56] Julian McAuley. `http://jmcauley.ucsd.edu/data/amazon/`.

[57] Jason McDonald, Daniel Hoffman, and Paul Strooper. Programmatic testing of the standard template library containers. In *Automated Software Engineering, 1998. Proceedings. 13th IEEE International Conference on*, pages 147–156. IEEE, 1998.

[58] Alistair Moffat. Word-based text compression. *Software: Practice and Experience*, 19(2):185–198, 1989.

[59] J Ian Munro, Thomas Papadakis, and Robert Sedgewick. Deterministic skip lists. In *Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms*, pages 367–375. Society for Industrial and Applied Mathematics, 1992.

[60] Jürg Nievergelt and Edward M Reingold. Binary search trees of bounded balance. *SIAM journal on Computing*, 2(1):33–43, 1973.

[61] Rizal Mohd Nor, Mikhail Nesterenko, and Christian Scheideler. Corona: A stabilizing deterministic message-passing skip list. *Theoretical Computer Science*, 512:119–129, 2013.

[62] Chris Okasaki and Andy Gill. Fast mergeable integer maps. In *Workshop on ML*, pages 77–86, 1998.

[63] Chandandeep Singh Pabla. Completely fair scheduler. *Linux Journal*, 2009(184):4, 2009.

[64] Thomas Papadakis, J Ian Munro, and Patricio V Poblete. Analysis of the expected search cost in skip lists. In *Scandinavian Workshop on Algorithm Theory*, pages 160–172. Springer, 1990.

[65] Thomas Papadakis, J Ian Munro, and Patricio V Poblete. Average search and update costs in skip lists. *BIT Numerical Mathematics*, 32(2):316–332, 1992.

[66] Greg Pass, Abdur Chowdhury, and Cayley Torgeson. A picture of search. In *InfoScale*, volume 152, page 1, 2006.

[67] V. Paxson and S. Floyd. LBL trace data. `http://ita.ee.lbl.gov/html/contrib/LBL-TCP-3.html/`, 1995.

[68] V. Paxson and S. Floyd. LBL trace data. `http://ita.ee.lbl.gov/html/contrib/LBL-CONN-7.html/`, 1995.

[69] Ben Pfaff. Performance analysis of bsts in system software. *ACM SIGMETRICS Performance Evaluation Review*, 32(1):410–411, 2004.

[70] David MW Powers. Applications and explanations of zipf's law. In *Proceedings of the joint conferences on new methods in language processing and computational natural language learning*, pages 151–160. Association for Computational Linguistics, 1998.

[71] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[72] William Pugh. A skip list cookbook. Technical report, 1998.

[73] Liang Qiaoyu, Li Jianwei, and Xu Yubin. Performance analysis of data organization of the real-time memory database based on red-black tree. In *Computing, Control and Industrial Engineering (CCIE), 2010 International Conference on*, volume 1, pages 428–430. IEEE, 2010.

[74] Ronald Rivest. On self-organizing sequential search heuristics. *Communications of the ACM*, 19(2):63–67, 1976.

[75] John T Robinson and Murthy V Devarakonda. *Data cache management using frequency-based replacement*, volume 18. ACM, 1990.

[76] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM (JACM)*, 32(3):652–686, 1985.

[77] Janusz Sosnowski, Marcin Kubacki, and Henryk Krawczyk. Monitoring event logs within a cluster system. In *Complex Systems and Dependability*, pages 257–271. Springer, 2013.

[78] Svetlana Štrbac-Savić and Milo Tomašević. Comparative performance evaluation of the avl and red-black trees. In *Proceedings of the Fifth Balkan Conference in Informatics*, pages 14–19. ACM, 2012.

[79] Robert E Tarjan and Renato F Werneck. Dynamic trees in practice. *Journal of Experimental Algorithmics (JEA)*, 14:5, 2009.

[80] Clark D Thomborson and Belle W-Y Wei. Systolic implementations of a move-to-front text compressor. *ACM SIGARCH Computer Architecture News*, 19(1):53–60, 1991.

[81] Zouheir Trabelsi and Safaa Zeidan. Multilevel early packet filtering technique based on traffic statistics and splay trees for firewall performance improvement. In *Communications (ICC), 2012 IEEE International Conference on*, pages 1074–1078. IEEE, 2012.

[82] Dan Wang and Jiangchuan Liu. A dynamic skip list-based overlay for on-demand media streaming with vcr interactions. *IEEE Transactions on Parallel and Distributed Systems*, 19(4):503–514, 2008.

[83] Hugh E Williams, Justin Zobel, and Steffen Heinz. Self-adjusting trees in practice for large text collections. *Software: Practice and Experience*, 31(10):925–939, 2001.

[84] J. Caverlee Z. Cheng and K. Lee. Cheng-Caverlee-Lee September 2009 - January 2010 Twitter Scrape . `https://archive.org/details/twitter_cikm_2010/`, 2010.

[85] Ei Phyu Zaw and Ni Lar Thein. Improved live vm migration using lru and splay tree algorithm. *International Journal of Computer Science and Telecommunications*, 3(3):1–7, 2012.

Appendix A

More Experimental Results

A.1 Changing Class C_1 Sizes

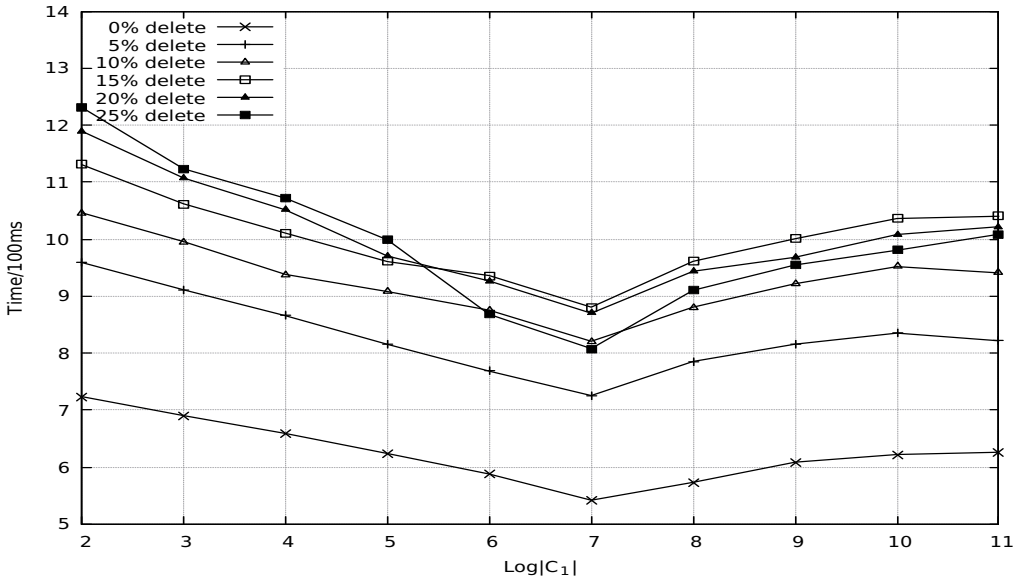


Figure A.1: Varying Class C_1 sizes on LBL trace data (long) (average rank 442) with varying deletion ratios

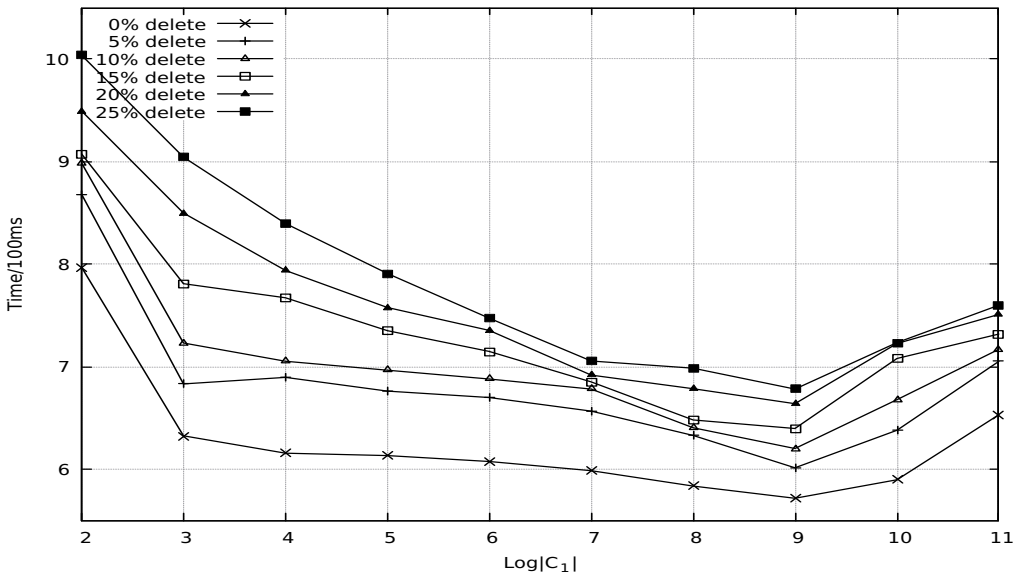


Figure A.2: Varying Class C_1 sizes on Query log data (average rank 3926) with varying deletion ratios

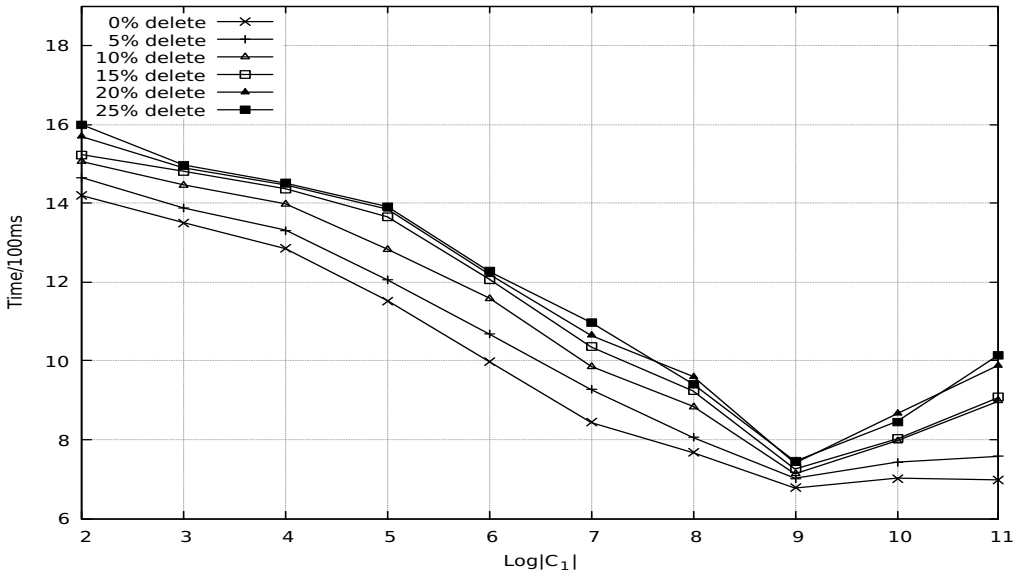


Figure A.3: Varying Class C_1 sizes on Pizza requests data (average rank 4165) with varying deletion ratios

A.2 Changing Value of t in H-DBSL

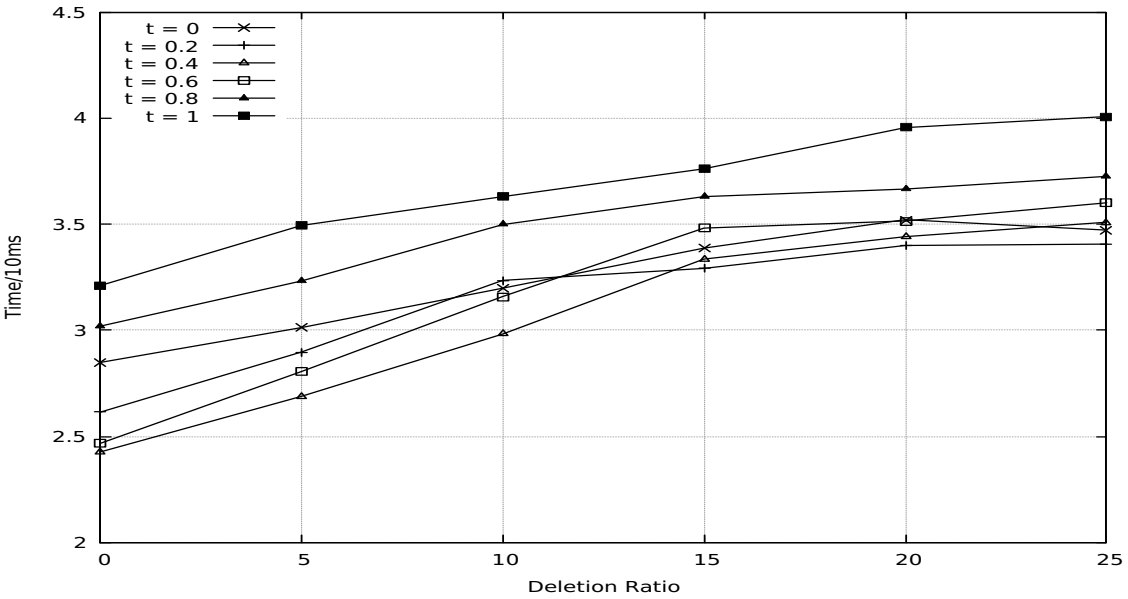


Figure A.4: Varying t on accesses to Amazon website data (average rank 283); Class C_1 default size is 128

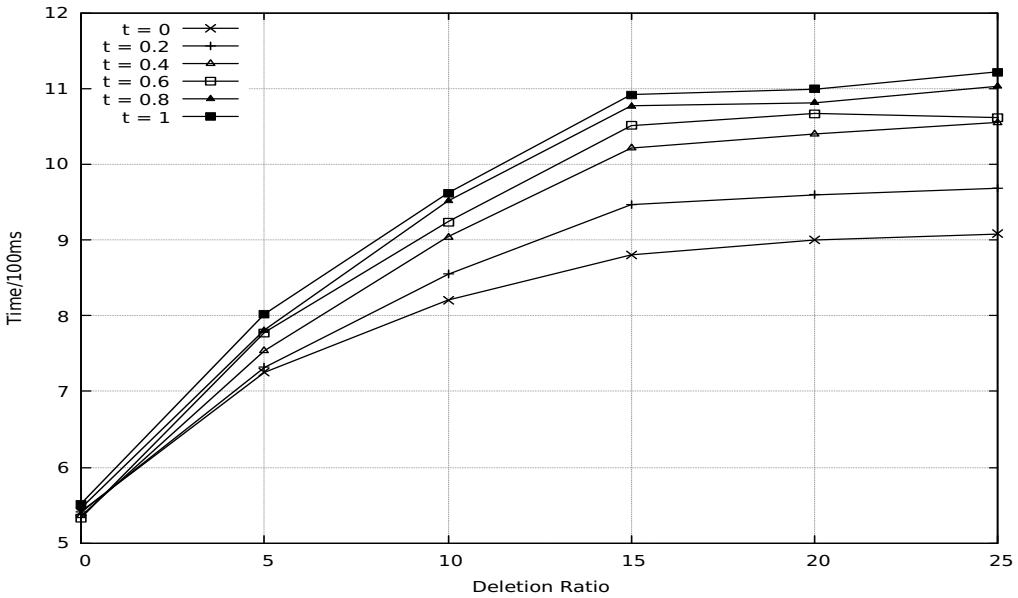


Figure A.5: Varying t on LBL trace data $\big($long$\big)$ $\big($average rank 442$\big)$; Class C_1 default size is 128

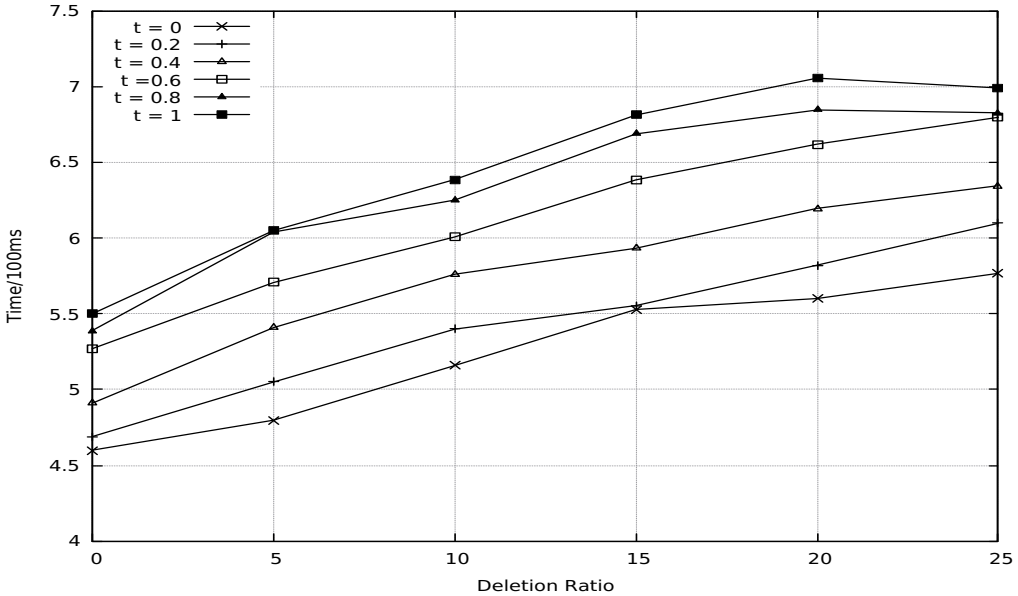


Figure A.6: Varying t on Twitter updates data $\big($average rank 515$\big)$; Class C_1 default size is 256

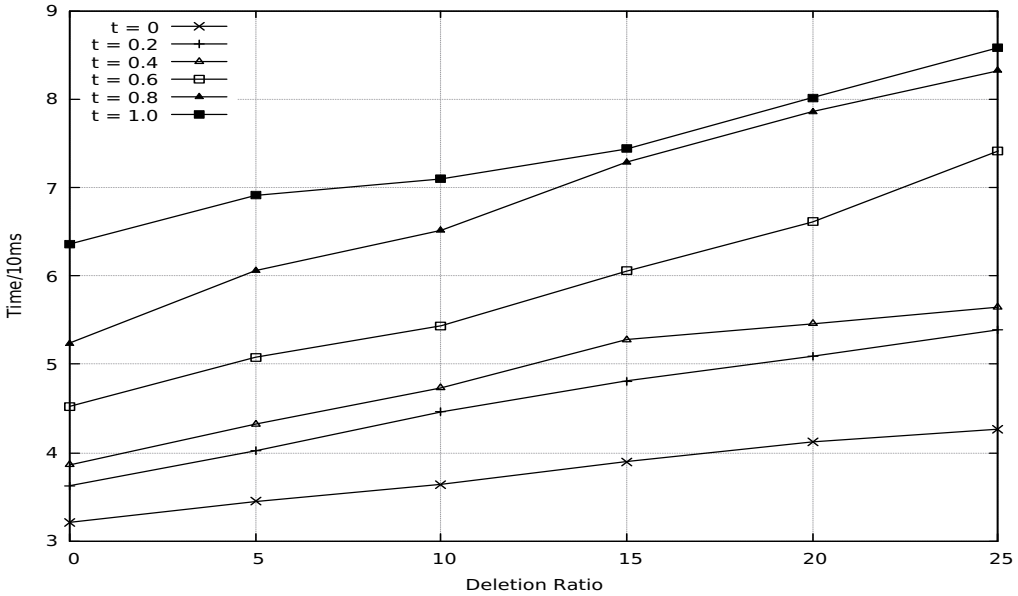


Figure A.7: Varying t on Amazon products reviews data $\big($average rank 1904$\big)$; Class C_1 default size is 256

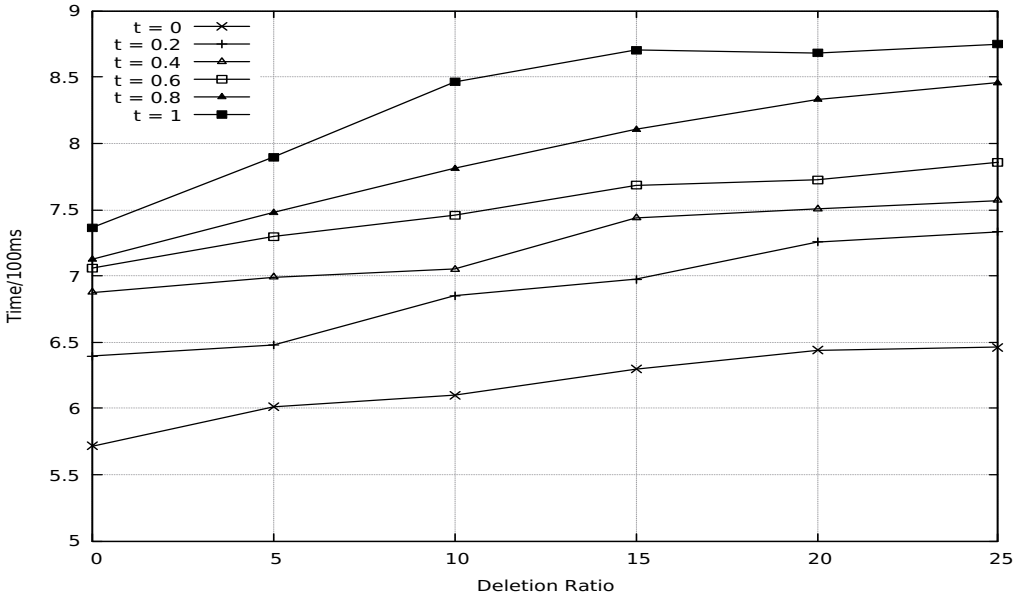


Figure A.8: Varying t on Query log data $\big($average rank 3926$\big)$; Class C_1 default size is 512

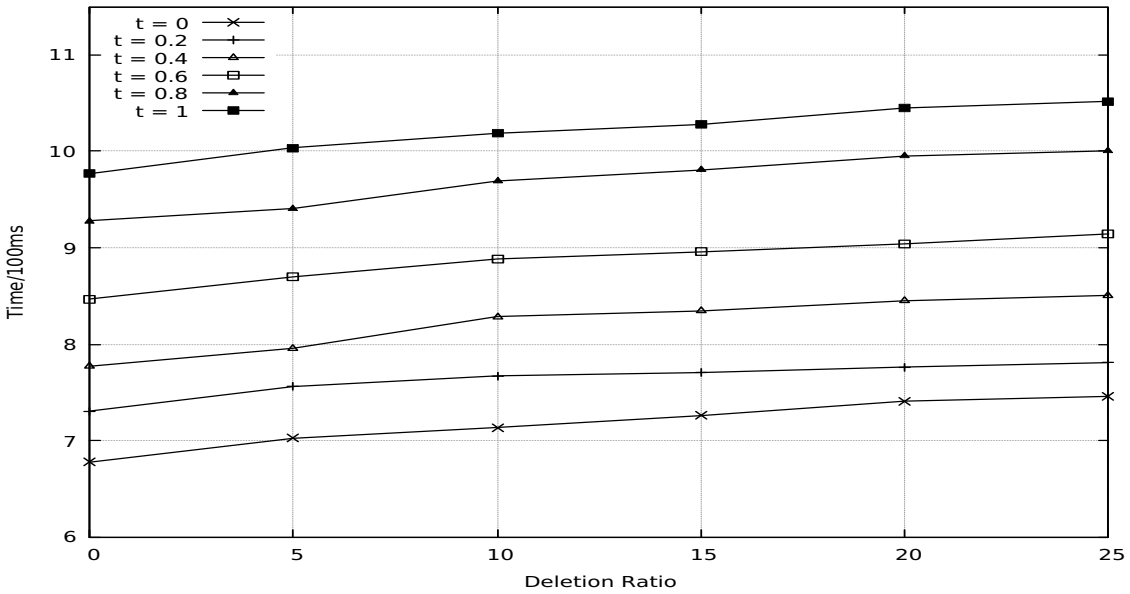


Figure A.9: Varying t on pizza requests data set $\big($average rank 4165$\big)$; Class C_1 default size is 512.

A.3 Comparison of Data Structures using Real-World Data

Table tables A.1 to A.6 provide the performances of each data structure under different deletion operation ratios. So that people can find the best solution for a particular data set with a particular sequence. The experiments using LBL trace data (short) (rank 19), Wiki adminship data (rank 33), Accesses to Amazon data (rank 283) and Amazon product review data (rank 1904) are executed for 100 times, LBL trace data (rank 442), Twitter updates data (rank 515), Query log data (rank 3926), Pizza requests (rank 4165) are executed for 10 times so that all the running times are measured with seconds and fit into one coordinate.

	Data Structures							
Data sets	MFL	SL	DBSL	H-DBSL	ST	R-ST	W-ST	RBT
LBL trace data (short)	1.30	10.14	6.45	6.33	9.93	8.42	10.94	10.53
Wiki adminship election	1.23	7.28	3.57	3.03	4.67	5.25	5.91	8.61
Accesses to Amazon website	3.71	4.13	2.85	2.55	3.60	3.30	3.08	4.28
LBL trace data (long)	2.47	6.01	5.41	5.23	6.63	6.45	4.03	6.20
Twitter updates	9.94	5.08	4.60	4.63	4.79	4.66	4.23	4.56
Amazon product reviews	13.28	1.59	3.21	3.42	1.21	1.11	0.99	1.03
Query log data	146.40	4.50	5.71	6.39	3.80	3.35	3.14	3.30
Pizza requests	25.48	3.03	6.78	8.00	3.21	3.01	2.87	2.02

Table A.1
Running time on each data structure when deletion ratio is 0

	Data Structures							
Data sets	MFL	SL	DBSL	H-DBSL	ST	R-ST	W-ST	RBT
LBL trace data (short)	4.35	12.80	8.09	7.85	10.48	8.93	11.65	13.20
Wiki adminship election	3.18	8.05	3.61	3.26	5.21	5.63	5.98	9.37
Accesses to Amazon website	6.06	5.27	3.01	2.96	4.02	3.45	3.37	5.27
LBL trace data (long)	8.42	6.73	7.25	7.23	7.31	6.30	4.00	6.73
Twitter updates	12.81	5.43	4.79	4.86	5.05	4.97	4.55	5.06
Amazon product reviews	14.64	1.80	3.44	3.61	1.24	1.14	1.04	1.16
Query log data	182.47	4.75	6.01	6.87	4.53	3.64	3.40	3.58
Pizza requests	25.91	3.30	7.02	8.56	3.24	3.03	2.81	2.05

Table A.2
Running time on each data structure when deletion ratio is 5%

	Data Structures							
Data sets	MFL	SL	DBSL	H-DBSL	ST	R-ST	W-ST	RBT
LBL trace data (short)	7.07	13.47	9.16	8.74	11.05	9.22	12.88	14.27
Wiki adminship election	4.80	8.871	3.90	3.31	5.75	5.90	6.06	10.36
Accesses to Amazon website	7.81	6.01	3.20	3.09	4.32	3.85	3.64	5.61
LBL trace data (long)	11.85	8.51	8.20	8.84	7.96	6.25	3.97	8.52
Twitter updates	14.73	5.77	5.15	5.30	5.26	5.11	4.71	5.47
Amazon product reviews	15.67	2.06	3.64	3.86	1.34	1.21	1.05	1.31
Query log data	24.62	4.98	6.10	7.05	4.88	3.99	3.70	3.94
Pizza requests	27.76	3.56	7.13	8.67	3.34	3.11	2.78	2.15

Table A.3
Running time on each data structure when deletion ratio is 10%

	Data Structures							
Data sets	MFL	SL	DBSL	H-DBSL	ST	R-ST	W-ST	RBT
LBL trace data (short)	9.45	14.42	10.09	9.83	11.24	10.10	13.36	14.97
Wiki adminship election	6.19	9.64	4.09	3.40	6.09	6.13	6.13	10.42
Accesses to Amazon website	9.46	6.89	3.39	3.15	4.53	3.98	3.88	5.81
LBL trace data (long)	19.10	9.29	8.80	9.07	8.33	6.27	3.94	9.93
Twitter updates	16.31	6.02	5.32	5.85	5.56	5.27	4.86	5.45
Amazon product reviews	18.92	2.49	3.89	4.11	1.39	1.23	1.11	1.41
Query log data	287.20	5.03	6.29	7.27	5.18	4.52	3.95	4.15
Pizza requests	26.71	3.57	7.25	8.70	3.39	3.17	2.70	2.18

Table A.4
Running time on each data structure when deletion ratio is 15%

	Data Structures							
Data sets	MFL	SL	DBSL	H-DBSL	ST	R-ST	W-ST	RBT
LBL trace data (short)	11.54	15.40	10.89	9.97	11.81	11.17	13.64	15.46
Wiki adminship election	7.30	9.94	4.13	3.66	6.21	6.45	6.18	10.46
Accesses to Amazon website	10.76	7.54	3.52	3.26	4.82	4.09	4.00	5.99
LBL trace data (long)	24.06	10.51	8.69	9.39	8.75	6.24	3.90	10.51
Twitter updates	17.60	6.16	5.40	5.62	5.59	5.32	4.99	5.81
Amazon product reviews	21.51	2.60	4.12	4.49	1.43	1.27	1.16	1.48
Query log data	322.95	5.18	6.43	7.56	5.41	4.70	4.02	4.30
Pizza requests	26.30	3.67	7.40	8.76	3.43	3.11	2.70	2.26

Table A.5
Running time on each data structure when deletion ratio is 20%

	Data Structures							
Data sets	MFL	SL	DBSL	H-DBSL	ST	R-ST	W-ST	RBT
LBL trace data (short)	13.18	16.24	10.91	10.02	12.40	11.64	13.88	16.32
Wiki adminship election	8.38	10.19	4.29	3.87	6.39	6.60	6.21	10.51
Accesses to Amazon website	11.69	7.66	3.47	3.38	4.90	4.34	4.11	6.15
LBL trace data (long)	28.17	11.02	8.08	9.08	8.90	6.22	3.88	11.02
Twitter updates	18.54	6.27	5.56	5.89	5.62	5.47	5.09	5.88
Amazon product reviews	23.37	2.72	4.26	4.69	1.48	1.32	1.21	1.53
Query log data	351.53	5.24	6.39	7.73	5.60	4.80	4.16	4.41
Pizza requests	25.48	3.03	6.77	8.30	3.21	3.00	2.87	2.02

Table A.6
Running time on each data structure when deletion ratio is 25%