The Complexity of Splay Trees and Skip Lists

Sayed Hassan Adelyar

Thesis presented in fulfilment
of the requirements for the degree of
Master of Science
at the University of the Western Cape

Supervisor: Reg Dodds

May 2008

Declaration

I, Sayed Hassan Adelyar, declare that this thesis "*The Complexity of Splay Trees and Skip Lists*" is my own work, that it has not been submitted before for any degree or assessment at any other university, and that all the sources I have used or quoted have been indicated and acknowledged by means of complete references.

Signature: [illegible] .... Date: 2008-05-01

Sayed Hassan Adelyar.

Abstract

Binary search trees (*BST*s) are important data structures which are widely used in various guises. Splay trees are a specific kind of binary search tree, one without explicit balancing. Skip lists use more space than *BST*s and are related to them in terms of much of their run-time behavior.

Even though binary search trees have been used for about half a century, there are still many open questions regarding their run-time performance and algorithmic complexity. In many instances, their worst-case, average-case, and best-case behaviors are unknown and need further research. Our analysis provides a basis for selecting more suitable data structures and algorithms for specific processes and applications.

We contrast the empirical behavior of splay trees and skip lists with their theoretical behavior. In particular we explore when splay trees outperform skip lists and *vice versa*.

The performance of a splay tree depends on the history of accesses to its elements. On the other hand, the performance of a skip list depends on an independent randomization of the height of links that lead to specific elements. Therefore, probabilistic methods are used to analyze the operation of splay trees and skip lists.

Our main results are that splay trees are faster for sorted insertion, where AVL trees are faster for random insertion. For searching, skip lists are faster than single class top-down splay trees, but two-class and multi-class top-down splay trees can behave better than skip lists.

Key words

Binary Search Trees, Balanced Trees, AVL Trees, Self-adjusting Trees, Bottom-up Splay Trees, Top-down Splay Trees, Skip Lists, Worst-case Time Bound, Amortized Time Bound, and Probabilistic Time Bound.

Acknowledgements

First of all I should thank the almighty and merciful God who gave me the ability to complete this work.

I am pleased to express my thanks to my respected supervisor Reg Dodds for his advice and support. I am deeply indebted to him for his help, valuable guidance and suggestions. Face to face and remote discussions through email and Skype with him were very helpful in bringing this thesis to fruition.

Many thanks to Jan Plane, Senior Instructor at the University of Maryland College Park for her advice throughout the work by way of weekly seminars.

I would like to acknowledge the Afghan eQuality Alliances (USAID) for their financial assistance.

Finally, I am thankful to my parents for their blessing and my family, especially my children Zobair, Hasina, and Faisal, for their understanding while I was busy.

Contents

List of Tables

List of Figures

Preface

Some of the notation used in this thesis is described here.

We often refer to the size of a data structure by referring to the number of its nodes as N, but also use `N`, typeset in a typewriter font to refer to a node `N`.

Most of the data structures used in this thesis have subfields, so we often use a *reference* or *pointer* `P` to the structure and then refer to its fields using simple dot-notation, such that `P.key` refers to the `key` field of the node pointed to by `P`. It means the same as the `Pascal` notation `P^.key` or the `C` notation `P->key`.

Steering clear of $\lg N$ for $\log_2 N$, we use $\log N$ for it.

Chapter 1

Introduction

1.1 Background and Motivation

Binary search trees (*BST*) are a fundamental data structure widely used for search and update operations and have attracted the attention of researchers in the last few decades. Many types of *BST* have been introduced over the years, including balanced *BST*s, such as *AVL* trees (Adel'son-Vel'skiĭ and Landis, 1962), symmetric binary *B* trees (Bayer, 1972), and B^+ trees (Bayer and McCreight, 1972), on which many file systems are based, and red-black trees, which are a variant of Bayer's symmetric binary *B* trees, introduced by Guibas and Sedgewick in 1978 and self-adjusting forms of *BST*s, such as splay trees (Sleator and Tarjan, 1983). Skip lists (Pugh, 1990) are an alternative representation for balanced trees. Skip lists use a probabilistic balancing method, rather than strictly enforcing balance.

A standard *BST* may degenerate into a linked list. In this case the worst-case performance is $O(N)$ instead of $O(\log N)$.[1] Balanced *BST*s have $O(\log N)$ worst case performance but after each insertion and deletion the balance must be checked and if necessary the tree must be restructured. Splay trees are a self-adjusting variant of *BST*, in which the newly accessed items are moved to the root of the tree. The next time the item is accessed the access is cheaper. Although they behave quite differently, splay trees and skip lists are similar to one other in the sense that neither follows strict balancing strategy but they both "self-balance" themselves.

Even though *BSTs* have been used for about a half century, many questions can be asked regarding their run time performance and algorithmic complexity. Even a simple question about these data structures may require an unexpected nontrivial analysis to answer. See (Jonassen and Knuth, 1978) and (Geldenhuys and van der Merwe, 2008). The most fundamental question which is still unsolved

[1]In this thesis $\log N$ should always be read as $\log_2 N$.

is: What is the asymptotically best *BST* data structure? In light of this there is much focus around splay trees (Demaine et al., 2007).

1.2 Research Problem

In this thesis we study and analyze the complexity of splay trees and skip lists and as a benchmark we compare them to a balanced variant of binary search trees, namely *AVL* trees.

Our research addresses the following questions:

1. What is the worst-case, average-case, and best-case behavior of splay trees?
2. How do splay trees compare with skip lists?
3. When do splay trees outperform skip lists and *vice versa*.
4. How do splay trees and skip lists compare with our benchmark for balanced binary search trees namely *AVL* trees?

We analyze the behavior of splay trees theoretically and empirically, and compare splay trees with skip lists and we identify the worst-case, average-case, and best case behaviors of splay trees and skip lists for search and update operations. We have identified applications for which splay trees outperform skip lists. AVL trees are used as a benchmark to compare how classic balanced binary search trees behave under the same circumstances.

1.3 Research Goals

Our research goals in more detail are to:

1. Analyze the performance of splay trees and skip lists theoretically and empirically,
2. Use the theoretical and empirical analysis for finding the worst-case, average-case, and best-case behavior of splay trees by applying our experiments consisting of both hand written simulations of small trees and timed computer runs of the same operations using much larger but similar data sets,
3. Compare the performance of splay trees with skip lists in order to identify the most suitable of these data structures for specific applications, and

4. Determine cases where splay trees outperform skip lists and *vice versa.*

1.4 Approach and Methodology

We initialized this research by studying and analyzing the behavior of splay trees. Our interest was stimulated by an early version of the paper by Geldenhuys and Van der Merwe, 2008. Their paper lead us to the idea of also investigating Bill Pugh's skip lists (Pugh, 1990) and of using the, by now, well understood *AVL* trees (Adel'son-Vel'skiĭ and Landis, 1962) as a benchmark to judge their performance. We initiated our theoretical analysis by, first, hand testing many different operations of standard balanced *BSTs*, splay trees, and skip lists.[2] We use data that was sorted in ascending and descending order and also in random order to determine the required number of comparisons needed. We used probabilistic methods to analyze the run time behavior of the skip lists. The probability of different cases for several generically different sequences of input was studied and analyzed. The worst-case, average-case, and best-case analyses for splay trees were performed for different sequences of inputs. We have not seen this kind of analysis in the literature.

The experiments were done on various machines, and the results were compared with the theoretical results. This sometimes proved to be more difficult than expected because of the unpredictable behavior of the available timers.

In these cases where the run time proved to differ from the theoretically expected time, the reason for the difference was explored. For the experiments we implemented the existing algorithms for various types of *BST*s, and then tabulated and evaluated their run time behavior. We used Java running on various machines under Linux or Sun's Solaris for doing our experiments. Each program was run for a few sequences of inputs, and for each sequence the run was repeated at least 30 times in order to attempt to achieve statistically meaningful results. When algorithms have logarithmic behavior the data sets have to be huge—in the order of millions—to achieve meaningful timings.

The results and statistics are programmatically generated, timed, and tabulated and graphed. Run time and theoretical time are statistically compared.

[2]See Appendix E on Page 136 for a brief description of these manual experiments.

1.5 Thesis Outline

This thesis contains seven chapters and an appendix with some definitions and proofs. In Chapter 2, we discuss various types of binary search trees (*BST*s) and their performance and we describe the performance of *AVL* trees, splay trees, and skip lists. Chapter 3 contains a literature review for the thesis. Chapter 4 describes our approaches and methodology for conducting the research and answering the thesis questions. Chapter 5 contains the theoretical analysis for the performance behaviors of *BST*s, skip lists, and splay trees where their behavior under insertion, deletion, and search operations is theoretically analyzed. Chapter 5 also contrasts splay trees and skip lists. In Chapter 6, we report our experimental results and compare these with the theory. The concluding Chapter 7 contains a brief overview of our main findings and interesting points and it details our results that we could not find in the literature and give some pointers to future research.

1.6 Summary

In this thesis we analyze the performance of splay trees and skip lists and contrast them with *AVL* trees. Standard *BST*s, *AVL* trees, splay trees, and skip lists are theoretically and experimentally analyzed for search and update operations. After comparing splay trees and skip lists we identify the cases in which splay trees would be more suitable than skip lists.

Chapter 2

Binary Search Trees

The binary search tree (*BST*) is a fundamental data structure, which combines the advantages of a sorted array and a linked list. We can search a balanced *BST* of N nodes, in $O(\log N)$ time which is as quick as doing binary search in a sorted array, and we can add and remove items as efficiently as with a linked list. *BST*s are known as *binary* search trees because each node holds up to two children and has the property that the key values of all the children in the left sub-tree of a node are *less than* the key of the node, and all the descendants in the right sub-tree of a node have keys that are *greater than or equal to* the node's key. In Figure 2.1

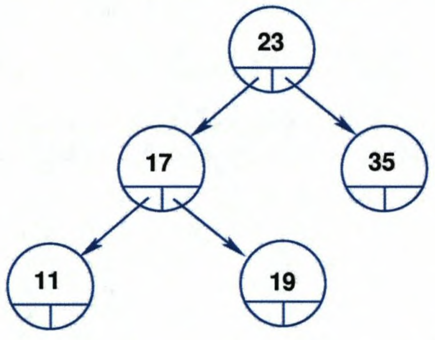


Figure 2.1: Order of nodes in a *BST*.

the `root` node has the key `23` and it has a left child with a key of `17` and a right child with key `35`. So for node `N` we have `N.left.key` $<$ `N.key` and `N.key` $\leq$ `N.right.key`. Once we have established that some tree is a BST and the actual values of the keys are irrelevant these need not be indicated. The root is generally assumed to be the uppermost node in the figure representing the tree. Figure 2.2 depicts a more abstract version of the tree in Figure 2.1.

*BST*s are widely used for search and update operations. Their organization allows for random and sequential access. Finding the minimum or maximum values are easy operations with *BST*s. They are dynamic data structures since memory for their nodes may be allocated and de-allocated during run time. In addition to the data and key, each node in a *BST* contains two links, which are called left and

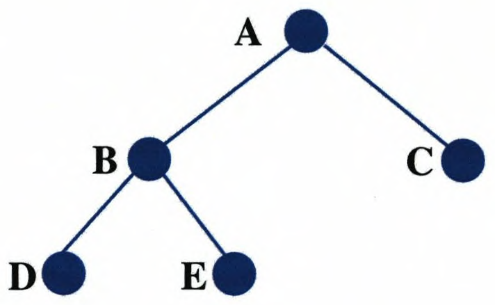


Figure 2.2: A more abstract depiction of a *BST*.

right links that may be null or point to a non null child of the node. Therefore, *BST*s with their extra pointers in each node, occupy one more pointer than a singly linked list.

*BST*s may be enhanced by adding a `parent` pointer which points to the parent of a node. This has the advantages of simplifying programming and often speeding up tree algorithms at the cost of a little wasted space. Pfaff, (2004), claims that using parent pointers is best if space is not at a premium.

The performance of *BST*s depends on the overall structure of the tree. *BST*s behave well when they are balanced. Balance means that the number of nodes in the left sub-tree is almost equal or exactly equal to the number of nodes in the right sub-tree. In other words, balance implies that the height of the tree is at a minimum. Balanced *BST*s allow us $O(\log N)$ worst-case search time—see the proof in Section 4.1.3, Pages 37–39. Balanced *BSTs* are suitable for applications in which search time must be minimized or in which the nodes are not necessarily processed in sequential order.

However, update operations such as insertion and deletion, can change the balance of the tree. Therefore, keeping the trees balanced complicates and slows the update operations of the tree. The order in which the data items are inserted into a *BST* affects the structure of the tree. Without balancing, if the data items are inserted in ascending or descending orders the tree becomes completely unbalanced, like a linked list. In addition to sorted insertion, there are other orders which also generate a completely unbalanced tree. In this case the worst-case performance is $O(N)$. Therefore, keeping the tree balanced, is an important issue when working with *BST*s.

There are many schemes to balance *BSTs* such as *AVL* trees, B^+ trees, red-black trees, their equivalent $(2, 3)$ trees and their generalization (n, m)-trees. Balanced trees use update algorithms which are more complicated than the standard *BST* algorithms and take longer times for insertion and deletion. However, the algorithm keeps the tree balanced in order to minimize the search time.

A balanced tree of N nodes is sometimes alternatively defined as a tree of height $\log N$.

We first consider *AVL* trees which we use as a benchmark to understand the performance of splay trees and skip lists.

2.1 *AVL* Trees

AVL trees, which are also called height-balanced trees, were first considered by the two Russian mathematicians, Adel'son-Vel'skii and Landis (1962). Each node is arranged such that the heights of its two subtrees never differ by more than 1. Each node in the tree holds a so-called *balance condition*, *balance criterion*, or *balance factor* that indicates if a node has subtrees of equal or different heights. After each insertion or deletion, the balance condition is checked and either the balance condition is altered or the tree is reconstructed by a single or double rotation. Duplicate keys are not allowed in an *AVL* tree. The balance criterion allows nodes with subtrees that differ by heights of at most 1. The tree in Figure 2.3 is not *perfectly* balanced but it is *AVL* balanced, since the left subtree of node `A` has height $h+2$ and its right subtree has height $h+1$, noting that each of the subtrees T_1, T_2, T_3 and T_4 is *AVL* balanced. Since these heights differ by at most 1, it is

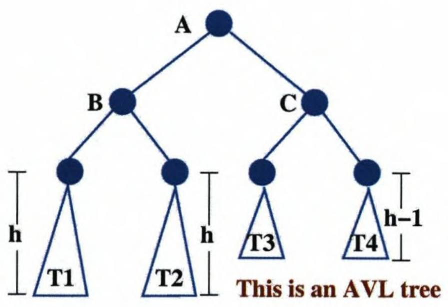


Figure 2.3: A balanced tree with the AVL property.

an *AVL* tree. The trees that appear in Figures 2.1 and 2.2 are also *AVL* trees. By appending the node `x` to the bottom of subtree T_1, in Figure 2.4, the height of the left subtree of node `A` becomes $h+3$ and now violates the *AVL* property. In

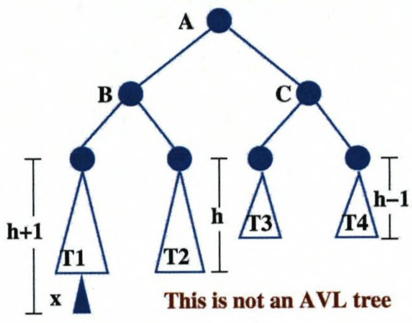


Figure 2.4: After adding **x** the tree has lost its AVL property.

Figure 2.5 a similar situation has arisen in the mirror of Figure 2.4, where an extra node **x** in the right subtree of `A` causes it to become 2 higher than the left subtree of `A`. We only need to assume that `A`, a pointer to the node where the imbalance

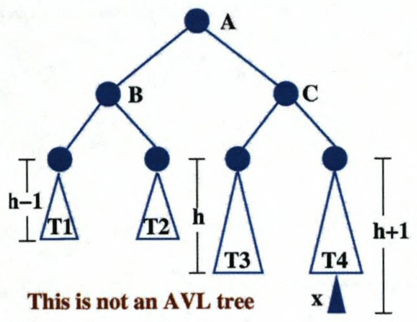


Figure 2.5: The mirror of Figure 2.4 is not an *AVL* tree.

occurs is known. So, given the pointer `A` to the point of imbalance, the following operations, or so-called *rotations*, turn the unbalanced tree in Figure 2.5 into a balanced one.

```
C = A.right;
A.right = C.left;
C.left = A;
C.parent = A.parent;
A.parent = C;
```

In Figure 2.6 the unbalanced tree of Figure 2.5 has been balanced.
AVL trees are *BST*s under the following conditions:

The heights of the left and right sub-trees of the each node differ by at most 1.
The left and right sub-trees of the root are also *AVL* trees.

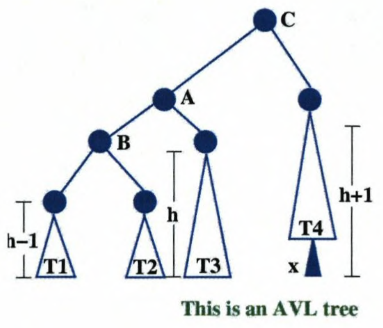


Figure 2.6: Figure 2.5 after balancing.

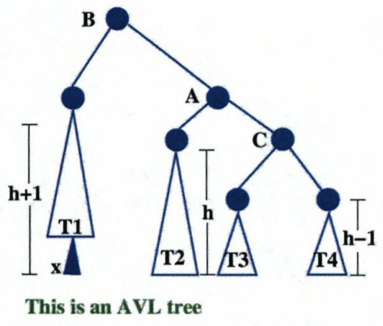


Figure 2.7: The balanced version of Figure 2.4.

In other words, for all nodes, `N`, in the tree, the difference in height of the left and right sub-trees of `N` is at most 1. These conditions ensure that the height of the tree is $O(LogN)$, where N is the number of elements in the tree (Foster, 1965). Suppose that, `N` is a node in an *AVL* tree then:

If `height(N.left) > height(N.right)`, then `N` is LEFT HIGH.
If `height(N.left) = height(N.right)`, then `N` is EQUAL HIGH.
If `height(N.left) < height(N.right)`, then `N` is RIGHT HIGH.

In addition to the data and references of the left and right sub-trees, each node also stores a balance factor. Every node must keep track of its balance factor. The balance factor for node `N` is a value such that:

If `N` is LEFT HIGH then the balance factor of `N` is -1.
If `N` is EQUAL HIGH then the balance factor of `N` is 0.
If `N` is RIGHT HIGH then the balance factor of `N` is 1.

Since the *AVL* tree is a *BST*, the search operation in an *AVL* tree uses the standard *BST* search algorithm. The guaranteed logarithmic height of an *AVL* tree ensures that searching in an *AVL* tree has a logarithmic worst-case time bound—see the proof in Section 4.1.3, Pages 37–39.

Insertion and deletion algorithms differ from their corresponding standard *BST* algorithms because they alter the structure of the tree and the resultant tree may not be an *AVL* tree and must sometimes be re-balanced.

2.1.1 *AVL* Tree Insertion

To insert a new item into an *AVL* tree, we first search the tree to find the position of the new item. Since duplicates are not allowed an appropriate error message is dispatched if the item has already been entered. If the search ends at an empty sub-tree, the new item is inserted there. After insertion, the resultant tree may not be an *AVL* tree. Thus we must restore the tree balance criteria. Only the nodes that are on the path from the insertion point to the root might have their balance changed. Therefore, we follow the same path, back to the root, which was followed when the new item was inserted into the *AVL* tree. The nodes on this path are visited and either their balance factor is changed, or we might have to reconstruct part of the tree. If we find a node whose new balance factor violates the *AVL* tree conditions—the so-called point of imbalance—we rebalance the tree by suitable rotation(s) (Foster, 1965).

Suppose that the node to be rebalanced is `N`. Since any node has at most two children and the height imbalance in an *AVL* tree requires that the height of the two sub-trees of `N` differs by more than 1, a violation may occur in any of the following cases:

An insertion into the:

(1) left sub-tree of the left child of `N`

(2) right sub-tree of the left child of `N`

(3) left sub-tree of the right child of `N`

(4) right sub-tree of the right child of `N`

Case (1) and case (4) are symmetric and in these two cases the insertion occurs on the outside, that is left–left or right–right respectively. In these cases the problem can be fixed by a *single* rotation. A single rotation changes the roles of the parent and child while maintaining the search order for the tree (Foster, 1965).

A single rotation can be left or right. If the rotation occurs at node `N` and it is a left rotation, then some nodes from the right sub-tree of N move to its left sub-tree. The root of the right sub-tree of N becomes the new root of the reconstructed sub-tree. Similarly, if it is a right rotation at node N, some nodes from the left sub-tree of N move to its right sub-tree and the root of the left sub-tree of N becomes the new root of the reconstructed sub-tree. Case (2) and case (3), in which the insertion occurs on the inside, that is, left–right or right–left, are complex and are repaired by double rotations (Manber, 1989, pp. 75–77). Figure 2.8 illustrates a tree requiring a double rotation and Figure 2.9 shows the result of a double rotation that restores the "*AVL*ness" of the tree. Assuming that we have a pointer to the

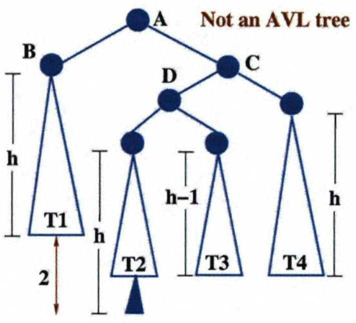


Figure 2.8: A tree requiring a double rotation.

imbalance point at node A, the following code does the rebalancing.

```
C = A.right;
D = C.left;
A.right = D.left;
D.left = A;
C.left = D.right;
D.right = C;
```

For the purpose of more clarity, we do not show the updates to the parent nodes. The double rotation is required, when the balance factor of the node where the tree is to be reconstructed, and the balance factor of the higher sub-tree are opposite. In this case first we rotate the tree at the lower node and then at the upper node. If the lower sub-tree is RIGHT HIGH, we make a left rotation, and if it is LEFT HIGH we make a right rotation. Secondly we rotate the tree at the upper node. If the tree rooted at the upper node is LEFT HIGH, we make a right rotation and if it is RIGHT HIGH, we make a left rotation (Malik and Nair, 2003).

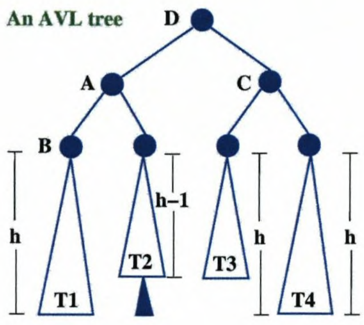


Figure 2.9: The tree in Figure 2.8 after a double rotation.

2.1.2 *AVL* Tree Deletion

To remove a node from an *AVL* tree, first we search the tree to find the node to be removed. If we find the node, we can remove it in the same way as we remove a node from a standard *BST*. The difference is that, after removing the node, the *BST* may not be an *AVL* tree. There may be an unbalanced node in the tree on the path from the parent of the deleted node to the root of the tree. In fact, there can be one such unbalanced node at most. Therefore, after deletion we have to restore the balance criteria for the tree. To do this, we follow the path from the parent of the deleted node back to the root node. We visit each node on this path, and sometimes we need to change only the balance factor, otherwise the tree is restructured at the point of imbalance. Suppose that `N` is a node on the path back to the root node. We check the current balance factor of `N`.

1. If the current balance factor of `N` is EQUAL HIGH, then the balance factor of `N` is changed according to whether the left sub-tree of `N` was shortened or the right sub-tree of `N` was shortened.

2. Suppose that the balance factor of `N` is not EQUAL HIGH and the taller sub-tree of `N` is shortened. The balance factor of `N` is changed to EQUAL HIGH.

3. Suppose that the balance factor of `N` is not EQUAL HIGH, the shorter sub-tree of `N` is shortened. Also suppose that `P` points to the root of the taller sub-tree of `N`.

   If the balance factor of `P`

(a) is EQUAL HIGH, a single rotation is required at `N`,

(b) is the same as `N`, a single rotation is required at `N`,

(c) and that of `N` are opposite. A double rotation is required at `N`, i.e. a single rotation at `P` and then another rotation at `N`.

(Malik and Nair, 2003).

2.2 Splay Trees

Splay trees are a self adjusting form of *BST*, invented by Daniel Sleator and Robert Tarjan (1983). Splay trees are used for many applications including searching, updating, and data compression (Grinberg and Rajagopalan, 2005). Linux kernels before 2.4.10 used *AVL* trees for keeping track of virtual memory areas. Later versions of the kernel use splay trees (Pfaff, 2004) and Windows NT also uses splay trees for its equivalent of virtual memory areas (Custer, 1993). Srinivasan et al. (2006) use splay trees for packet classification in high-speed routers and claim high performance gains in terms of space and time complexity over previous techniques. Contrary to other types of self-balancing trees, splay trees work well with nodes containing identical keys. Splay trees support all the operations of *BST*s and have the following *advantages*:

- They require[3] less memory space since they do not store extra balancing information.
- They are easy to program.
- They automatically adapt to the input sequence. This results in better performance than fixed trees, when the access pattern is non-uniform (Sleator and Tarjan, 1983).

  Splay trees have the following *disadvantages*:

- They require more adjustment than the other balanced *BST*s. It means more rotation and more swapping of children is needed during insertion, deletion, and search operations. It results in potentially expensive individual operations. Of course, this depends on the application. If rotations are

[3] *AVL* trees need two bits per node to represent the balance factor.

usually expensive, splay trees may be less efficient than other balanced *BST*s (Sleator and Tarjan, 1983).

- They have $O(N)$ worst case performance per operation. There are many search trees with $O(\log N)$ worst case times, such as *AVL* trees, (2-3)-trees, *B*-trees and red-black trees.

After each access to a node of a splay tree, the newly accessed node is moved to the root of the tree. When the node is accessed again, subsequent accesses are cheaper, if they occur soon enough. There are three methods for moving a node to the root of the tree:

1. The *simple* splay tree strategy,
2. The *bottom-up* splay tree strategy, and
3. The *top-down* splay tree strategy.

2.2.1 Simple Splay Trees

The easiest method for moving an item to the root of the tree is called the simple self-adjusting strategy, or rotate-to-root strategy. In this method the newly accessed item is continually rotated with its parent until it becomes the root of the tree. The tree is rearranged after each access. In this method, when the accessed item is moved to the root, some other items are moved further away from the root. Therefore, if the access order of the items is not clustered 90–10%, i.e. the accesses follow a *locality* rule where most accesses occur more or less in a 10% cluster and the rest outside of that cluster, it is possible for some bad accesses to occur. As a result, simple splay tree operations do not have logarithmic amortized behavior (Sleator and Tarjan, 1985).

2.2.2 Bottom-up Splay Trees

The bottom-up splay strategy also moves the accessed item to the root of the tree, but differently. This kind of move to root operation is known as *splaying*. The bottom-up splay strategy considers the item to be rotated, its parent, and its grandparent—if it exists. Single or double rotations are used during the splaying, depending on the position of the item to be splayed. A single rotation is required if the item to be splayed has only a parent but has no grandparent. This kind of rotation is called a *Zig* rotation (Sleator and Tarjan, 1985).

A double rotation is required if the node to be accessed has both, parent and grandparent. Double rotations can be a *Zig-zig* or *Zig-zag*. A *Zig-zig* rotation is

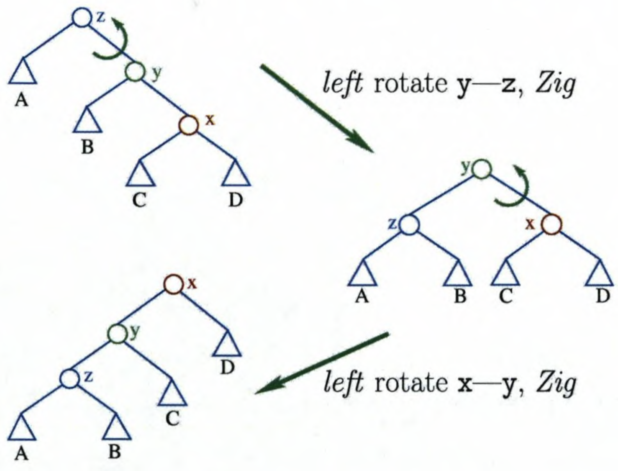


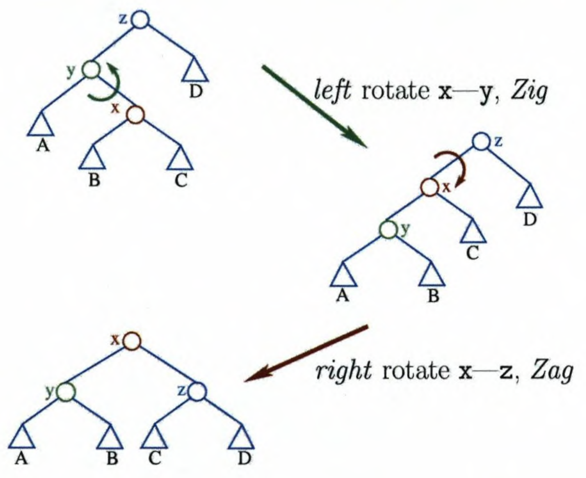
Figure 2.10: A left-left *Zig-zig*.

required if the item to be accessed and its parent, are both left children or both right children. In Figure 2.10 a *Zig-zig* of x is accomplished with two rotations: the first anti-clockwise rotation or left rotation is performed between its parent, y, and grandparent, z, i.e. y—z is left rotated with y as the center of rotation. Sleator and Tarjan call this a *Zig*. Next, the second rotation is performed between x and its parent, i.e. x—y is also left rotated around x—another *Zig* (Sleator and Tarjan, 1985).

A *Zig-zag* rotation is required if the item is a left child and its parent is a right child or vice versa. A *Zig-zag* is also performed in two rotations—one left and one right rotation or vice versa. The first rotation is between the item, x, and its parent, y, and the second rotation is between the item, x and its grandparent, z (Sleator and Tarjan, 1985).

The *Zig-zig* or *Zig-zag* process is repeated until the accessed item becomes the root of the tree.

To delete an item from a bottom-up splay tree, we first access the item. The access process moves the accessed item to the root of the tree. Deleting the item from the root of the tree, we get two sub-trees: *L* and *R*. At this stage there are two possibilities:



z
y
x
A
B
C
D
left rotate **x**—**y**, *Zig*
z
x
y
A
B
C
D
right rotate **x**—**z**, *Zag*
x
y
z
A
B
C
D

Figure 2.11: A left-right *Zig-zag.*

1. We find the *largest* item in L and splay that item to the root of L, and then make R the right sub-tree of L's root.

2. Alternately, we find the *smallest* item in R and splay that item to the root of R, and then make L the left sub-tree of R's root.

This is called the *join* operation, which joins the left and right sub-trees (Sleator and Tarjan, 1985).

2.2.3 Top-down Splay Trees

Bottom-up splaying uses two passes for splaying the item to the root of the tree. The first pass is required to access the item, and the second pass is used to splay the item to the root of the tree. Therefore, bottom-up splay trees need more comparisons and rotations for accessing and splaying the items to the root of the tree.

Top-down splay trees on the other hand, use a single pass for accessing and splaying the items to the root of the tree. In this way, top-down splay trees require less comparisons and rotations for accessing the items. Top-down splay trees are fast and maintain the amortized time bound (Sleator and Tarjan, 1985).

In a top-down splay, the tree is split into *three* sub-trees: a *left* sub-tree, L, *right* sub-tree, R, and a *middle* sub-tree, M. During the search for an item, we take the items that are on the access path and move them to the left sub-tree or right sub-tree depending on whether they are smaller or larger than the item for which we are searching. Initially the left and right sub-trees are empty and the middle tree consists of the entire tree (Sleator and Tarjan, 1985). At any point during the search for `x`, we have to follow the left or right link. When we follow the left link, then `x` and its right sub-tree are larger than the item which will become the root. Therefore, we put `x` and its right sub-tree into R. When we follow the right link then `x` and its left sub-tree are smaller then the item which will become the root. Therefore, we put `x` and its left sub-tree in another sub-tree, which we call its left sub-tree, L.

Descending the tree two levels at a time, we encounter three items, `x`, `y`, and `z`. `x` is the root of the middle tree, `y` is a child of `x`, and `z` is the grandchild of `x`. With these three items we consider three different cases: *Zig* case, *Zig-zig* case, and *Zig-zag* case. In the *Zig* case, `y` becomes the root of the middle tree, `x` and its sub-tree are attached to the L sub-tree or the R sub-tree, depending whether `x` is smaller or larger then `y`, respectively. If the case is *Zig-zig*, `z` become the root of the middle tree, and we rotate `y` around `x` and attach it as a right child of the largest value of sub-tree L or as a left child of the smallest value of sub-tree R. In a *Zig-zag* case, `z` is moved to the root of the middle tree, M sub-tree, the sub-trees `x` and `y` are moved to R and L sub-trees, respectively. Simplify the code, by changing the *Zig-zag* case to a *Zig* case. That is, instead of making `z` the root of the middle tree, `y` is moved to the root of the middle tree and `z` remains as a sub-tree of `y`. This simplification makes the *Zig-zag* case similar to the *Zig* case. It avoids the rotation process but causes more iterations in the splay process. When the value to be accessed is at the root of the middle tree, we make the left child of `x` a right child of the maximum item in the L sub-tree, and make the right child of `x` a left child of the minimum item in the R sub-tree. Finally we make L and R the left and right children of `x`, respectively.

2.3 Skip Lists

Skip lists, invented by William Pugh (1990), are an alternative data structure to balanced BSTs, but their structure is completely different. They are a suitable and often better alternative to balanced BSTs in many applications (Pugh, 1990). Skip lists use a probabilistic balancing method rather than strictly enforcing balance. Each node in a skip list contains a key, its data, and one or more pointers. The number of pointers, which is also called the node level, for each node is determined randomly. Each node has at least one pointer, and the first of them always points

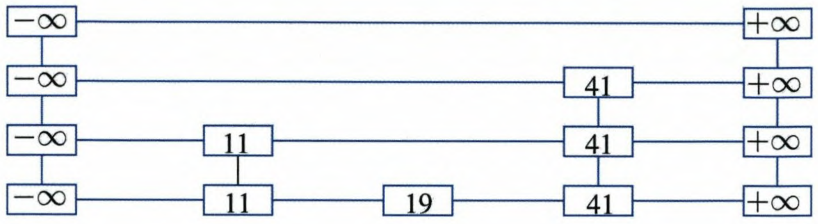


Figure 2.12: A skip list.

to the next node in the skip list. The additional pointers are used to skip one or more intermediate nodes. Each list has two extra nodes. The first node determines the beginning of the list and the last node determines the end of the list. The first node has a key smaller than any other node, and the last node has a key greater than any other node. These two nodes can conveniently be made $-\infty$ and $+\infty$ respectively.

Figure 2.12 gives Goodrich and Tamassia's (2004) depiction of a skip list in which the levels are terminated by an extra node which has a `null` out pointer and a key of $+\infty$ such that the value of the key does not need special treatment. Pugh (1990) draws the skip list slightly differently clearly showing the `null` pointers at the end of the list. In Figure 2.12 these null pointers can never be reached so the pointers need never be tested since the lists are terminated by a key sentinel, i.e. by $+\infty$ giving this representation a slight edge on Pugh's. Pugh's levels are terminated by a non sentinel key but the end node in the level has a `null` pointer and this is used to terminate searches in levels.

For most applications skip lists are easier and simpler to implement than balanced trees and self-adjusting trees. They are also very space efficient and Pugh 1990 points out that they can be configured easily to require an average of $1\frac{1}{3}$ pointers per element or even less and do not require balance or priority information to be stored with each node.

For many applications, skip lists are a more natural representation than balanced trees, and they lead to simpler algorithms. The simplicity of skip list algorithms makes them easier to implement in many applications. It is easier to balance a data structure probabilistically rather than explicitly maintaining its balance.

Skip lists have probabilistic time bounds, which means, any operations or sequence of operations can take longer than expected, although the probability of any operation taking significantly longer than expected is very low. We present our code in Appendix D on Page 133.

2.3.1 Skip List Structure

A skip list consists of a series of lists arranged horizontally in levels and vertically in towers. Each level contains about half the number of nodes than the previous level. Therefore, the total number of levels in a skip list is probably about $O(\log N)$, where N is the number of elements in the skip list. The structure of a skip list is determined only by the number of elements in the skip list and the results of consulting the random number generator. The sequence of operations that produced the current skip list does not matter. (Pugh, 1990).

Suppose that S is a skip list. Then S will contain a series of horizontal lists—levels—such as $S_0, S_1, S_2, \ldots, S_h$. Each list represents a level in the skip list. S_0 will contain all the elements in the skip list. S_1 probably contains about half of the elements of S_0, and list S_2 probably contains about half of the elements of S_1, and so on. In other words, S_0 contains all the elements, S_1 probably contains about $\frac{1}{2}$ of the elements, S_2 probably contains about $\frac{1}{4}$ of the elements, and S_h probably contains $\frac{1}{2^h}$ of the elements.(Pugh, 1990)

Generally, S_h will contain about $\frac{n}{2^h}$ elements, where h is the height of the list. The halving of the number of entries from one list to the next is not enforced as an explicit property of skip lists. Instead randomization is used. Each level contains special keys at the end points, such that the key of the starting element in every list is $-\infty$ and $+\infty$ is the key of the last element of every list. Level h, the highest level, contains only these special keys $-\infty$ and $+\infty$.

2.3.2 Skip List Algorithms

We can easily implement a skip list by means of a linked structure. We view a skip list as a two-dimensional collection of positions arranged horizontally into levels and vertically into towers. Each level is a list S_i and each tower contains positions storing the same entry across consecutive lists.

Each element in a skip list is represented by a node. The level of a node is chosen randomly when the node is inserted. We do not need to store the level of a node in the node. All levels of a skip list are terminated with `null`. A *new* list is initialized so that the level of the list is 1 and all forward pointers of the list's header point to `null`.

2.3.2.1 Search in a Skip List

Search for a key is started at the header of the top level, and continues moving forward, if the node key is smaller than the search key. If the node key is equal or greater than the search key, the search drops down one level and then continues forward. This process is continued until we find the target node or we are satisfied that the target node is not present in the skip list (Pugh, 1990).

For example, suppose that we want to search for a key `k`. We begin the search by setting a pointer `P` to the top most, left position in the skip list S, called the start position of S. We then perform the following steps, where `P.key` denotes the key of the entry at position `P`:

1. If the element of S below `P` is null, then the search terminates. Because we are at the bottom and have located the largest entry in S with a key less than or equal to the search key `k`. Otherwise, we drop down to the next lower level in the present tower by moving `P` down one level (Weiss, 1999).

2. Starting at position `P`, we move `P` forward until it is at the right-most position on the present level such that `P.key<=k`. We call this the forward-scan step. Note that such a position always exists, since each level contains the keys $+\infty$ and $-\infty$. In fact, after we perform the forward scan for this level, `P` may remain where it started. In any case, we then repeat the previous step (Pugh, 1990).

2.3.2.2 Inserting New Nodes into a Skip List

To insert a new item into a skip list, we first search the list to find a position for the new item. During the search, the references of items at which the search dropped down one level are saved in an array. When the position is found the item level is generated by a randomized level generator. Next the new item is created and it is entered into the skip list. This is done by assigning the forward references stored in the array (Pugh, 1990). Then the insertion algorithm for skip lists uses randomization to decide the height of the tower for the new entry. To do this we flip a coin. If the coin comes up tails, then we stop here. Else, we backtrack to the next higher level and insert `k` in this level at the appropriate position. We again flip a coin; if it comes up heads, we go to the next higher level and repeat. Thus, we continue to insert the new entry `k` in lists until we finally throw tails. We link together all the references to the new entry `k` created in this process to create the tower for the new entry. If an insertion generates a node with a level greater than the previous maximum level of the list, we update the maximum level of the list and initialize the appropriate portions of the update vector (Pugh, 1990).

A skip list S must maintain a reference to the start position—the top most, left position in S—as an instance variable, and must have a policy for any insertion that wishes to continue inserting a new entry past the top level of S. There are two possible actions we can take, both of which have their own advantages:

- One possibility is to restrict the top level h, to be kept at some fixed value that is a function of N, such as $\log N$ where N is the number of entries in the skip list.
- The other possible action is to let an insertion continue inserting a new position as long as it keeps getting heads returned from the random number generator. The probability that an insertion will go to a level that is more than $\log N$ is very low.

2.3.2.3 Deleting Elements from a Skip List

Deleting an item from a skip list is similar to insertion. To delete an item, first search for the item to be deleted. If it is found, update the references by reassigning the nodes that reference the node to be deleted, to the node that come after the node to be deleted. After each deletion, check if the maximum element of the list has been deleted. If so, decrease the list's maximum level (Pugh, 1990).

Chapter 3

Literature Review

Binary search trees (*BST*s) have an important data structure which is used in many computer applications. Their search operation has a complexity similar to that of the binary search algorithm in a sorted linear list, but it also allows fast insertions and deletions.

Each operation in a *BST* requires that the search begins at the root of the tree. The time needed to find a node depends on how many levels down the tree the node is situated. The number of levels depends on the total number of nodes and the structure of the tree. A balanced *BST* with N nodes, has the best search and update performance, and its worst-case search cost is logarithmic—$O(\log N)$ (Wright, 2006). Unfortunately, there is no guarantee that the tree is balanced. The update operations–insertion and deletion–tend to change the structure of the tree, and may unbalance the tree. If the tree is completely unbalanced, in the worst case, the access time becomes linear $O(N)$ (Baer and Schwab, 1977). Keeping the tree balanced requires restructuring the tree during insertion and deletion, which slow down the operations of insertion and deletion, and complicate their algorithms. There is a trade-off between the speed of updating the tree, and the speed of retrieving items from the tree (Foster, 1965). Different applications may prefer either fast update operations or fast search operations or both.

Some questions can be raised regarding the run time performance of *BST*s and their algorithm complexity. Even a simple question about these data structures may require an unexpected nontrivial analysis to answer (Jonassen and Knuth, 1978).

The balance of a *BST* depends on the key values and order in which the keys are entered into the tree. If the keys are inserted randomly, the tree will tend be more or less balanced. But, if the nodes are inserted strictly in order of the keys, say from smallest to largest or *vice versa*, then the tree degenerates into a linear list with a worst case search time of $O(N)$ (Baer and Schwab, 1977).

In a perfectly balanced full *BST*, the sub-trees of each node have exactly the same height. Other *BST*s are said to be *height balanced*, or *simply balanced*, if the sub-trees of each node in the tree differ in height by no more than 1. A balanced *BST* with N nodes has $O(\log N)$ performance in the worst-case, where unbalanced BST has $O(N)$ worst case performance (Jonassen and Knuth, 1978).

Several algorithms are used to implement balanced *BSTs*. Most of these algorithms are slightly more complicated than the standard *BST* algorithms. Because the operations of insertion and deletion can change the structure of the tree, they take longer than look-ups. However, a balanced *BST*'s update algorithm keeps the tree balanced and provides protection against unbalanced trees.

3.1 *AVL* Trees

AVL trees are the most well known balanced *BST*s (Adel'son-Vel'skiĭ and Landis, 1962); see also the article by (Baer and Schwab, 1977). They have the property that: for each node the height of its left sub-tree and the height of its right sub-tree differ by at most one. This property acts as a condition that keeps *AVL* trees balanced. The property must be checked after each insertion and deletion to ensure that the tree is still an *AVL* tree. When necessary the proper rotations are applied to the tree, thus complicating insertions and deletions.

In addition to the data and references of the left and right sub-trees, each node also stores a balance factor (Foster, 1965). Every node keeps track of its balance factor. The balance factor indicates the height of the left and right sub-tree of the node.

In the worst case, *AVL* trees require $\log(N+1)$ probes for retrieving an item from the tree (Foster, 1965). The worst-case search in an *AVL* tree is about $1.44 \log N$ comparisons, and the average search can require $1.05 \log N$ comparisons. If an application uses only insertion and search operations, then *AVL* trees are the preferred data structure (Nievergelt and Reingold, 1972).

The concept of *AVL* tree was generalized by Caxton Foster (Foster, 1973) to allow the imbalance of the tree by more than one before reconstructing the tree. That means, during the insertion and deletion the nodes left and right sub-trees can differ by more than one. In this way they permit a system designer several options which allow trade off between the access time and ease of construction. The ease of construction and speed of retrieval also depends on the application (Foster, 1973).

Foster found that by allowing imbalancing up to 4, the number of probes for retrieving increase by one, but the number of restructurings of the tree decrease from one every other item to only one for every 13.7 items (Foster, 1973).

3.2 Splay Trees

Splay trees are a self adjusting form of *BST*s. They were invented by Daniel Sleator and Robert Tarjan (Sleator and Tarjan, 1983). Splay trees are used for many applications including searching, updating, and data compression (Grinberg and Rajagopalan, 2005). They have been proposed to be used for packet classification to provide security, quality of service (QoS), monitoring and multimedia capabilities (Srinivasan et al., 2006). Contrary to other types of self-balancing trees, splay trees work well with nodes containing identical keys.

Even though balanced *BST*s have a $O(\log N)$ worst-case time bound per operation on an n-nodes tree, they are not as efficient as possible if the access pattern is non-uniform, and need extra space for storing balance information (Sleator and Tarjan, 1985). Balanced *BST*s have several limitations:

- They require a little extra space for storing balancing information.
- They are complicated to implement, making insertions and deletions expensive and potentially error-prone.
- Easy inputs need the same work as more difficult ones.

(Sleator and Tarjan, 1985)

On the other hand, the worst-case, average-case, and best-case performance for balanced *BST*s are essentially identical. This means, the performance of balanced *BST*s is improvable. For example, if we want to find an item such as `X` in a balanced *BST*, the find cost is logarithmic and the second find operation costs the same time as the first one. In splay trees, the second find operation for the same item is cheaper when it occurs soon enough. We would also expect that if we perform an access of `X`, `Y`, and `Z`, then a subsequent set of accesses for the same sequence would be cheaper in a splay tree than in an *AVL* tree (Sleator and Tarjan, 1985).

Splay trees support all the operations of *BST*s and have the following advantages:

1. They require slightly less storage space because, they do not store extra balancing information.

2. They are easy to program.

3. They adapt automatically to the input sequence, resulting in better performance than fixed trees, when the access pattern is skewed.

   (Sleator and Tarjan, 1985)

Splay trees have the following *disadvantages* :

1. They require more adjustment than balanced *BST*s, i.e. more rotations and more swapping of children is needed during insertion, deletion, and search operations. It results in potentially expensive individual operations. Of course, this depends on the application. If rotations are usually expensive, splay trees may be less efficient than other balanced *BST*s.

2. They have $O(N)$ worst case performance per operation. There are many search trees with $O(\log N)$ worst case times, such as *AVL* trees, (2-4)-trees, B-trees and red-black trees. In the worst case, the overall running time of a search, insertion, or deletion in a splay tree of height h is $O(h)$. This is because the node we splay might be the deepest node in the tree. Moreover, it is possible for h to be as large as N. Thus, from the worst-case point of view, a splay tree is not an attractive data structure. Splay trees perform well in an amortized sense, i.e. in a sequence of intermixed searches, insertions, and deletions, each operation on average takes logarithmic time.

   (Sleator and Tarjan, 1985)

Good performance of splay trees depends on the self-balancing, self-optimizing, in that frequently accessed nodes will move nearer to the root of the tree to be accessed more quickly. Inactive elements will be pushed away further from the root. This is the case in most applications, and is particularly useful for implementing caches. This is important because of the 90−10 rule. The 90−10 rule suggested by empirical studies states that, in practice 90% of the accesses are localized to 10% percent of the data items. The $90-10$ rule has been used for many years in disk input/output systems. A memory cache stores the contents of some of the disk blocks in fast memory. The hope is that when a disk access is required, the block can be found in the main memory cache and thus save the cost of an expensive disk access. Programs running in memory show this behavior. Peter Denning (1968) speaks of a *working set*. Only a relatively small proportion of an entire disk can be stored in memory. Even so, storing the most recently accessed disk blocks in the cache enables large improvements in performance because many of the same disk blocks are accessed over and over again. The same idea applies to splay trees. The most recently accessed nodes stay near the top of the tree.

Applications in which the total time for a sequence of operations is important and not the individual operation time, then splay trees become a better choice (Sleator and Tarjan, 1985).

The interesting point about a splay tree is that, it guarantees an amortized running time, for insertion, deletion, and search operations. This means, its bounds are amortized. The amortized time bound means that a specific operation may take longer but a sequence of operations is fast. In an N-node splay tree, all the standard search tree operations have an amortized time bound of $O(\log N)$ per operation. Thus splay trees are as efficient as balanced trees when the total running time is the measure of interest (Sleator and Tarjan, 1985). Moreover, designing a data structure with amortized performance, such as a splay tree, is simpler than designing the worst-case time performance (Iacono, 2001). However, amortized bounds are not always acceptable. Specifically, if a single bad operation becomes too time-consuming: $O(N)$ time for a single access may be acceptable as long as it does not happens too often. In particular, if any M operations take a total of $O(M \log N)$ worst-case time, giving an average of $\log N$, then the fact that some operations are expensive could become acceptable (Sleator and Tarjan, 1983).

In the best case the required number of comparisons for building a splay tree, from N sorted order data items, is $N - 1$ comparisons (Geldenhuys and van der Merwe, 2008). The worst case cost for building a splay tree with N items is $N(\frac{N}{4}+1)-2-\alpha$, where $\alpha = 0$ if N is even and $\alpha = \frac{1}{4}$ when N is odd (Geldenhuys and van der Merwe, 2008).

Considering the amortized time bound and ignoring the constant factor, splay trees are as efficient as both dynamic balanced trees and static optimal trees. They may have stronger optimality properties (Sleator and Tarjan, 1985). Since almost all uses of BSTs involve a sequence of operations rather than just a single operation, an amortized bound is generally as useful as a bound on each operation. On the other hand, data structures designed to achieve a certain amortized performance are often simpler to implement than the data structures with good worst-case performance. A situation in which this might not be true is in real-time applications, where it is important to have a low worst-case bound on the running time of each individual operation (Sleator and Tarjan, 1985).

Splay trees require more rotations than balanced *BST*s such as *AVL* trees. The cost of rotation depends on the application. Therefore, if the rotation cost is bigger then the self-adjusting tree's performance may be inefficient (Sleator and Tarjan, 1983). The cost of a rotation in a search tree, which we assume to be $O(1)$, depends upon the application. If rotations are unusually expensive, self-adjusting search trees may be inefficient. One worst-case issue with splay tree algorithms is that of sequentially accessing all the elements of the tree in sorted order. This will leave the tree completely unbalanced (Weiss, 1999).

The *delete minimum* in a minimizing heap, and *delete maximum* in a maximizing heap, are important priority queue operations. With splay trees, these operations become simple. To delete the minimum item, first we perform a find minimum operation. This operation brings the minimum item to the root, and by the *BST* property, there is no left child. Put the right child at the new root. Figure 3.1 illustrates the *delete minimum* operation. The initial tree whose root

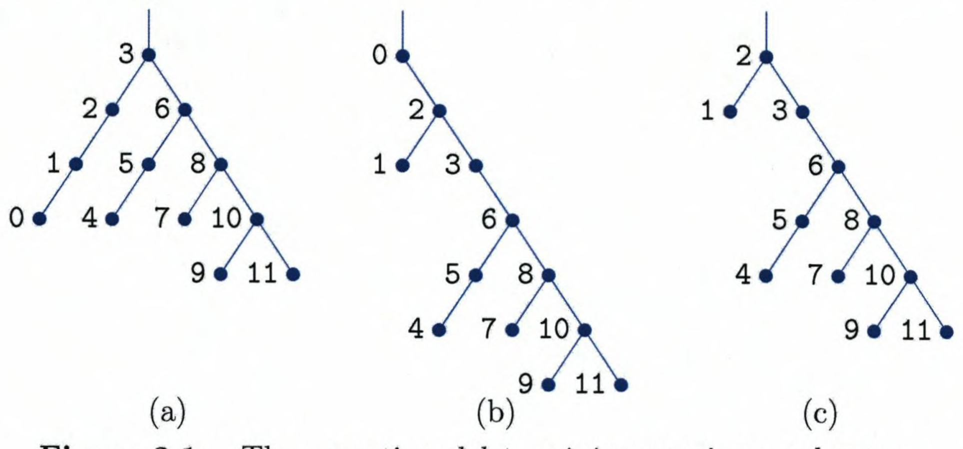


Figure 3.1: The operation *delete minimum* using a splay tree.

node has the key 3 is shown in Figure 3.1(a), then in Figure 3.1(b) the minimum node with key 0 has been splayed, and finally in Figure 3.1(c) the root containing 0 has been removed and the new root has been set pointing to the node with key 2. Similarly, *delete maximum* is implemented by finding the maximum node by following right pointers from the root of the tree until a leaf is reached, splaying this maximum node to the top and setting the new root to its left child.

The easiest way to move an item toward the root is by a simple rotation strategy. The accessed node is continually rotated with its parent until it reaches the root. After becoming the root, subsequent accesses to that node are cheaper. Even if some other nodes are accessed before it is re-accessed, that node is closer to the root and is found quickly. But, if the access order of nodes does not follow the $90-10$ rule, it is possible for some bad accesses to occur. In reality, rotating like this leads to badly behaved trees when every node in the tree is repeatedly accessed sequentially. A simple proof that this takes $\Theta(N^2)$ time is given in Appendix A on Page 128.

The second strategy for moving the accessed node to the root called the bottom-up splay strategy is more viable. Bottom-up splay strategy is similar to the simple rotate-to-root strategy, but the parent and grandparent are considered to decide what kind of rotations to apply. Movement of the accessed node in this way is called *splaying*. Splaying roughly halves the depth of every node along the access path (Sleator and Tarjan, 1985).

Bottom-up splay trees use *Zig*, *Zig-zig* and *Zig-zag* rotations for splaying a node to the root of the tree. See Section 2.2.2 on Page 14 for a full description. Top-down splay trees use the same three kinds of rotations differently. Top-down splay trees do not use a rotation in the *Zig* case. Instead, they use some movement of the nodes. *Zig-zig* and *Zig-zag* cases use only one rotation and some movement of the nodes (Sleator and Tarjan, 1985). Even the simplified *Zig-zag* case does not use rotation. It only uses movement of the nodes. Simplified *Zig-zag* avoids rotation and makes the code simpler, but increases the number of iterations (Weiss, 1999). For the details of top-down splaying see Section 2.2.3 on Page 16.

A more efficient method for moving the accessed node to the root of the tree is the *top-down* splay strategy. Top-down splaying uses a single pass for accessing and splaying the node. Therefore, the top-down splay procedure is faster in practice, and maintains the logarithmic amortized bound (Sleator and Tarjan, 1985).

3.2.1 Splay Tree Theorems

We have adapted the proofs by Daniel Sleator and Robert Tarjan 1985 for theorems concerning the run-time efficiency of splay trees.

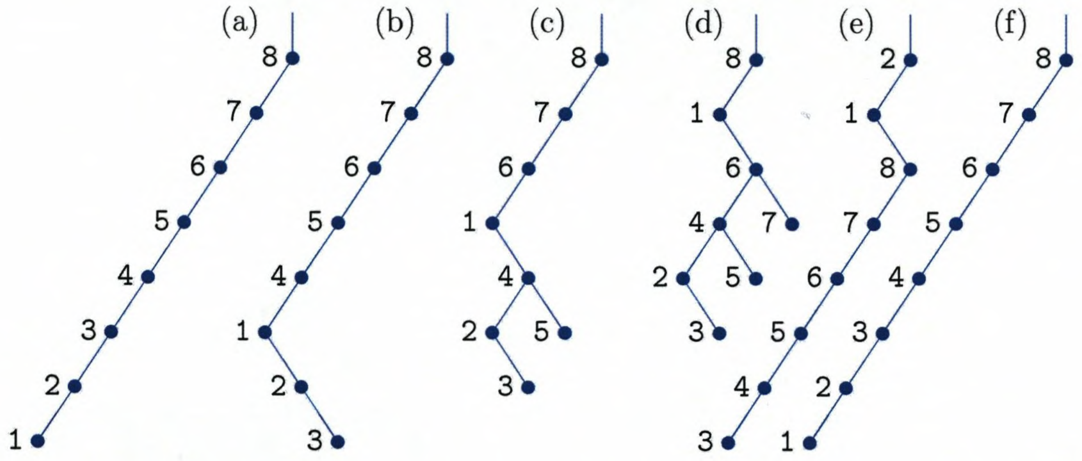


Figure 3.2: (a) A skew tree built by sequentially inserting [1..N] in $\Theta(N)$ time. (b) The tree after 1 has been zig-zigged once. (c) The tree after 1 has been zig-zigged twice. (d) The final tree after 1 has been zig-zigged a third time. (e) The resultant tree after 2 has been splayed to the root. (f) The final tree after each node has been accessed sequentially is the same as the initial tree.

Theorem 3.2.1 Balance Theorem *The total access time in a splay tree is* $O((M+N)\log N + M)$, *where* N *is the total number of nodes, and* M *is the total number of operations.*

Proof Let us assign a weight of $\frac{1}{N}$ to each item. Then the total weight $W = \sum_{r=1}^{N} \frac{1}{N} = 1$. The amortized access time is at most $3\log N + 1$ for any item. The net potential drop over the sequence is at most $N \log N$.
(Sleator and Tarjan, 1985). ■

Let $f(i)$ be the total number of times any item i is accessed. Splay trees have the same $O(\log f(i))$ runtime as optimal search trees. The only difference is that its bound is amortized, rather than worst case. Splay trees achieve this bound without any prior knowledge of the input distribution. This property is known as the *Static Optimality Theorem.*

Theorem 3.2.2 Static Optimality Theorem *If every item is accessed at least once, then the total access time is:*

$$O(M + \sum_{i=1}^{N} f(i) \log \frac{M}{f(i)})$$

Proof Assign a weight of $\frac{f(i)}{M}$ to item i. Then the total weight $W = 1$, the

amortized access time of item i is $O(\log \frac{M}{f(i)})$, and the net potential drop over the sequence is at most $\sum_{i=1}^{N} \log \frac{M}{f(i)}$.
(Sleator and Tarjan, 1985). ■

Assume that the items are numbered from 1 to N in symmetric order. Suppose that the the sequence of items is $i_1, i_2, \ldots, i_M$.

Theorem 3.2.3 ***Static Finger Theorem*** *For any fixed item, F, in the splay tree, the total access time is*

$$O(N \log N + M + \sum_{j=1}^{M} \log(|i_j - F| + 1))$$

Proof Assign a weight of $\frac{1}{(|i-F|+1)^2}$ to item i. Then $W \leq 2\sum_{k=1}^{\infty} \frac{1}{k^2} = O(1)$. Then, the amortized time of the jth access is $O(\log(|i_j - F| + 1))$. The net potential drop over the sequence is $O(N \log N)$, since the weight of each item is $\frac{1}{N^2}$.
(Sleator and Tarjan, 1985). ■

There are many open questions about the performance of splay trees. Further analysis and work are needed to analyze the worst-case, average-, and best- case behavior of splay trees (Geldenhuys and van der Merwe, 2008). Splay trees may follow a top-down, i.e. root insertion method or a bottom-up, i.e a leaf insertion method. In root insertion the newly formed splay tree is built as progress is made down the tree until the access is successful. Whereas in leaf insertion the splaying of the tree is started after the element is finally accessed. Leaf insertion thus needs two passes along the access path whereas leaf access traverses the access path once. In their analysis for root insertion and leaf insertion Geldenhuys and van der Merwe (2008) compare leaf insertion and root insertion for a sequence of inputs s such as $\{a_1, a_2, \ldots, a_n\}$ and the reverse of s such as $\{a_n, a_{n-1}, \ldots, a_1\}$. They also compare the average-case and worst-case for building a tree using root insertion, and conclude their study by providing experimental results that have provided the impetus for our investigation. They also mention many interesting questions that arise from considering their experimental results.(Geldenhuys and van der Merwe, 2008)

The analysis of worst-case, average, and best-case behaviors of root insertion in splay trees is still open and needs more work. We study and analyze the performance of splay trees experimentally and study theoretical aspects of splay tree behavior. In our analysis we will assume different possible cases and then compare the results.

3.3 Skip Lists

Skip lists—data structures that behave similarly to balanced binary trees—were discovered by Bill Pugh (1990).

Skip lists use a probabilistic balancing method, rather than strictly enforcing balance. They make random choices in arranging the entries in such a way that search and update times are $O(\log N)$ on average, where N is the number of entries in the list (Pugh, 1990). The average time depends on the use of a random number generator in the implementation of the insertion to help decide where to place the new entry. It does not depend on the probability distribution of the keys in the input. It means, no input sequence consistently produces the worst-case performance (Pugh, 1990). The insertion algorithm of skip lists can be easily modified to allow duplicate keys (Pugh, 1990).

For most applications skip lists are easier and faster to implement than balanced trees and self-adjusting trees (Pugh, 1990). They are also very space efficient and can be configured easily to require an average of 1⅓ percent pointers per element—or even less—and do not require balance or priority information to be stored with each node (Pugh, 1990). They have balance properties similar to search trees built by random insertion, but they do not require insertion to be random (Pugh, 1990).

The structure of a skip list is determined only by the number of elements in the skip list and the results of consulting the random number generator, and the sequence of operations that produced the current skip list does not matter (Pugh, 1990). The total number of levels in a skip list is probably about $O(\log N)$, where N is the number of elements in the skip list (Pugh, 1990). Items are randomly distributed in the levels and half of the items will move from level i to level $i + 1$ with high probability (Pugh, 1990).

Skip list implementation has roughly the same efficiency as the implementation of highly optimized, non-recursive balanced trees (Pugh, 1990). For uniform query distribution, skip lists are faster by a factor of 2–3, than recursive balanced trees or highly optimized splay trees (Pugh, 1990).

Although skip lists have bad worst-case performance, no input sequence *consistently* produces the worst-case performance (Pugh, 1990). It is very unlikely a skip list data structure will be significantly unbalanced. As an example, for a list of more than 250 elements, the chance that a search will take more than three times the expected time is less than one in a million (Pugh, 1990).

The time required to find an element is proportional to the length of the search path, which is determined by the pattern in which elements with different levels appear as we traverse the list (Pugh, 1990). Since a skip list has about $O(\log N)$ levels with high probability, and the expected amount of time spent scanning forward at any level i is $O(1)$, therefore, the expected running time for searching an entry on a skip list with N entries is $O(\log N)$ (Pugh, 1990). The probability of poor running times for successive operations on the same data structure are not independent, that means two successive searches for the same element will both take exactly the same time (Pugh, 1990).

The expected running time of the insertion algorithm on a skip list with N elements is $O(\log N)$ (Pugh, 1990). The running time is averaged over all possible outcome of the random numbers used when inserting entries. The expected running time for removal is also $O(\log N)$ (Pugh, 1990).

In worst-case performance, skip lists are not a superior data structure. In fact, if we do not officially prevent an insertion from continuing significantly past the current highest level, then the insertion algorithm can go into what is almost an infinite loop, however, since the probability of having a fair coin repeatedly come up heads forever is 0. If we terminate position insertion at the highest level h, then the worst case running time for performing the find, insert and remove operations in a skip list S with N entries and height h is $O(N + h)$.(Weiss, 1999) This worst-case performance occurs when the tower of every entry reaches level $h - 1$, where h is the height of the skip list. However, the probability for this to happen is very low. Judging from this worst case, we might conclude that the skip list structure is strictly inferior to the balanced *BST*s (Weiss, 1999).

Balanced trees–*AVL*, red-black trees–, self-adjusting trees–splay trees– and skip lists can be used for the same problems. A choice among them involves several factors such as, ease of implementation, type of bounds, constant factor, and performance on non-uniform distribution of queries (Pugh, 1990).

For many applications, skip lists are a more natural representation than balanced trees, and they lead to simpler algorithms. The simplicity of skip lists algorithms makes them easier to implement in many applications. It is easier to balance a data structure probabilistically than to maintain the balance explicitly (Pugh, 1990). The main advantage of using randomization in data structures and algorithm design is that the structures and methods that result, are usually simple and efficient (Pugh, 1990).

Balanced trees have worst-case time bounds, self-adjusting trees have amortized time bounds and skip lists have probabilistic time bounds. For skip lists, any operations or sequence of operations can take longer than expected, although the probability of any operation taking significantly longer than expected is very low. In some real time applications, we must be sure that an operation will be completed within a certain time bound. For such applications, self-adjusting trees may be undesirable since they can take significantly longer on an individual operation than expected. For example, in self-adjusting trees an individual search can take $O(N)$ time instead of $O(\log N)$ time. For real-time systems, skip lists may be usable if an adequate safety margin is provided (Pugh, 1990).

Constant factors can make a significant difference in the practical applications of an algorithm. Skip lists provide significant constant factor speed improvements over balanced trees and self-adjusting tree algorithms. Compared to self-adjusting trees, skip lists have low inherent complexity and low inherent overhead. Skip list algorithms are non-recursive and they are simple enough so that programmers can optimize them (Pugh, 1990).

Self-adjusting BSTs such as splay trees adjust to non-uniform query distribution, therefore, they are faster than skip list for skewed distribution. On the other hand skip lists are faster when processing uniform query distribution (Pugh, 1990).

The Table 3.1 on Page 34 by Bill Pugh (1990) compares the performance of implementations of skip lists, balanced trees, and self-adjusting trees. All implementations were optimized for efficiency.

Skip lists perform more comparisons than other methods. If real numbers are used as keys, skip lists were slower than the non-recursive AVL tree algorithms. Also search in a skip list was slightly slower than search in a 2–3 tree, but, insertion and deletion using the skip-list algorithms were still faster than using the recursive

Table 3.1: Comparison of methods with skip lists.

Implementation	Search Time	Insertion Time	Deletion Time
Skip Lists	0.051 ms. (1.0)	0.065 ms. (1.0)	0.059 ms. (1.0)
Non-recursive AVL trees	0.046 ms. (0.91)	0.10 ms. (1.55)	0.085 ms. (1.46)
Recursive 2–3 trees	0.054 ms. (1.05)	0.21 ms. (3.2)	0.21 ms. (3.65)
Top-down splaying	0.15 ms. (3.0)	0.16 ms. (2.5)	0.18 ms. (3.1)
Bottom-up splaying	0.49 ms. (9.6)	0.51 ms. (7.8)	0.53 ms. (9.0)

2–3 tree algorithms (Pugh, 1990).

Skip lists are faster than self-adjusting trees by a significant constant factor when queries are uniformly distributed. Self-adjusting trees are faster than skip lists only for highly skewed distributions. So, in an application where highly skewed distributions are expected, either self-adjusting trees or a skip list aided by a cache may be preferable (Pugh, 1990).

Chapter 4

Methodology

Our starting point for this research was conducting the literature review. We started our research by studying and analyzing the behaviors of binary search trees (*BST*s), particularly *splay trees* and *skip lists.* In this phase we collected enough information about *BST*s. Through our literature review we were able to grasp the history and development of various *BST*s.

Based on our study of the literature, we could formulate the theoretical analysis for *AVL* trees, splay trees, and skip lists. Our theoretical analysis is based on the brute-force approach. The insertion, deletion, and search operations for the above mentioned data structures were theoretically studied and analyzed. In order to generalize the theory, for each operation different inputs and tree structures were considered. For each operation the required number of comparisons, rotations—if necessary—, and the resultant tree structure were determined. To analyze the building operation of each data structure, sorted and unsorted insertions were considered.

Because of the involvement of randomization and probabilities in the operations of skip lists, we used simple probabilistic approaches to analyze the performance of skip lists. The probability of different cases for many sequences of inputs were studied and analyzed. Our results are based on the cases which have high probabilities.

The search and update operations of bottom-up and top-down splay trees are theoretically analyzed by using a brute-force approach. For each operation different data inputs and tree structures were considered. The best-case, average-case, and worst-case are calculated. The effect of the rotation types on the structure of the resultant tree is analyzed.

After completing the manual analysis we started our experimental analysis. In fact, in our research, completing the manual analysis before the experiments guided us to decide which experiments were needed. For our experimental analysis we implemented the existing algorithms for various *BST*s and skip lists.

After completing the experimental analysis we compare our results with the expected theoretical behavior. In the case of differences the reason is explained. We have compared statistically the run-time, R, with theoretical time, T. When comparing the empirical timings we scaled the data structures according to the theoretical bounds of the algorithms in order to get 'fair' comparisons. For example, if the theoretical bound for lookup in a balanced *BST* is $O(\log N)$ and the bound for a lookup in a *BST* that has degenerated into a linear list with bound $O(N)$ we will use $n = \log N$ elements where N is a large number when comparing two such methods. We expect the empirical times to differ by a constant. The experimental results and statistics are programmatically generated, timed, and tabulated.

All the above steps and analysis are used to analyze the worst-case, average-case, and best-case behavior of splay trees. It is also used for comparing splay trees with skip lists as well as for finding the cases in which splay trees outperform skip lists.

4.1 Analysis of Binary Search Trees

In a *BST*, the required time for finding an element depends on how many levels down the tree the element is situated. In other words, the required search time depends on the depth of the target element. The search operation begins at the root of the tree and descends the tree level by level to find the target node. For selecting the path to the right or left sub-tree, only one comparison is performed in each level.

4.1.1 *BST* Best Case Analysis

In any *BST*, the best case occurs when the target node is situated at the root of the tree. Therefore, trivially, the best case search time is $O(1)$.

4.1.2 *BST* Worst Case Analysis

The worst case occurs when the target node is situated at the bottom level of the tree. In a fully degenerate *BST*, where all the nodes have been inserted strictly in order, the nodes of the *BST* are arranged in a linear list. So the worst case search time occurs when searching for the node at the end of list giving a worst case search time of $O(N)$.

4.1.3 *BST* Average Case Analysis

We first show a deviation by calculating the time taken to access each node of a fully populated balanced *BST*. This analysis leads to the conclusion that the average search time is $O(\log N)$.

Non-probabilistic proof of average search time in a *BST*

In a binary search tree the target node can be located at any level of the tree. The relation between the total number of elements and the number of levels in a full *BST* is: Let N be the total number of *elements* and h be the *level number*,

Table 4.1: The relationship between the number of elements and the heights of each level: There are 2^{h-1} elements in level h and altogether $2^h - 1$ elements in an h level fully populated balanced *BST*, i.e. a *full BST*.

Level	*Number of elements per level*	*Number of elements down to this level*	
Level 1	2^0	2^0	$= 2^1 - 1$
Level 2	2^1	$2^0 + 2^1$	$= 2^2 - 1$
Level 3	2^2	$2^0 + 2^1 + 2^2$	$= 2^3 - 1$
...	...	...	
Level h	2^{h-1}	$2^0 + 2^1 + 2^2 + \ldots + 2^{h-1}$	$= 2^h - 1$

then:

$$N = 2^h - 1 \text{ and } h = log(N + 1)$$

The total number of elements can also be calculated as follows:

$$\begin{aligned} N &= 1 + 2 + \cdots + 2^{h-1} \\ &= \sum_{i=0}^{h-1} 2^i = 2^h - 1. \end{aligned}$$

To calculate the average search time in a full *BST*, divide the total search time for searching each of the elements in the tree once by the number of elements. The total time for searching all the elements in a *full*, balanced tree with h levels is:

$$T_h = 1 + 2 \cdot 2 + 3 \cdot 4 + \cdots + h \cdot 2^{h-1} = \sum_{i=1}^{h} i2^{i-1}.$$

In order to follow the standard derivation for the average case first consider Figure 4.1 which shows a few levels of a fully loaded balanced *BST*. Notice that there are 2^{h-1} nodes on level h and that the total number of nodes up to that level is $2^h - 1$. The number of *elements* N in a full balanced *BST* with h levels is given

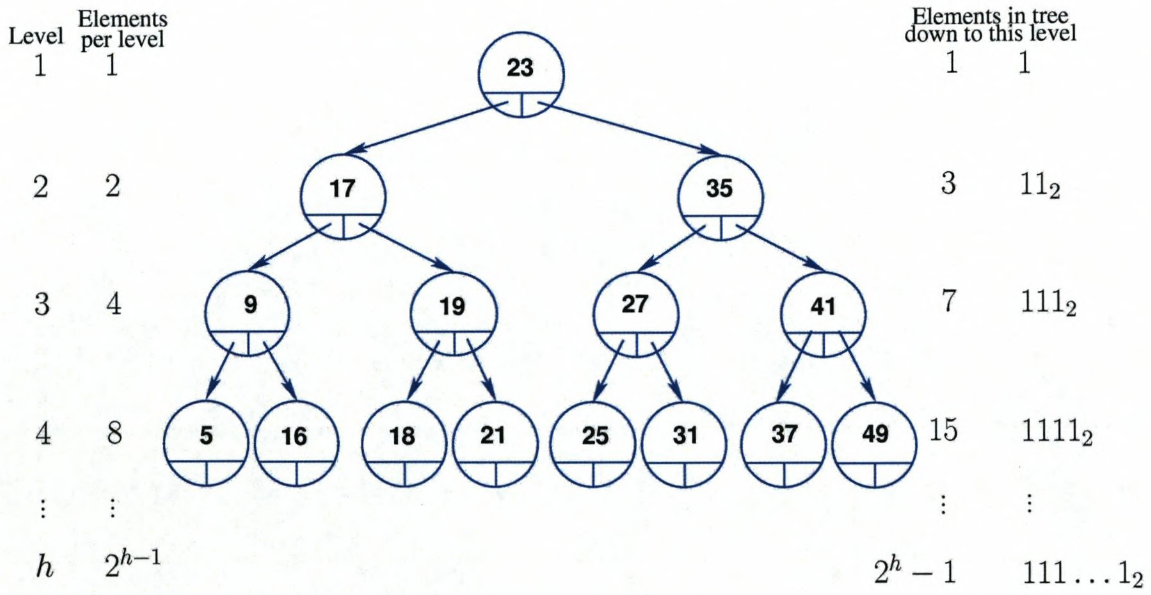


Figure 4.1: A full balanced *BST* with h levels has altogether $2^h - 1$ nodes.

by

$$\begin{aligned} N &= \sum_{i=1}^{h} 2^{i-1}, \\ &= 2^h - 1. \end{aligned}$$

To approximate the average access time we assume that the probability of accessing any one node is uniform. Going further we will approximate this by finding the total number of probes to probe each node once and divide this by N.

The total number of probes, T_h, needed for accessing each node once in a full balanced *BST* with h levels, is given by

$$T_h = \sum_{i=1}^{h} i2^{h-1}.$$

This is calculated as follows.

$$\begin{array}{rcl}
T_h & = & 1+2\cdot 2+3\cdot 4+ \quad \cdots \quad +h\cdot 2^{h-1}, \\
2T_h & = & 2+2\cdot 4+3\cdot 8+\cdots+(h-1)\cdot 2^{h-1} + h\cdot 2^h, \\
\hline
T_h & = & -(1+ \; 2+ \; 4+ \quad \cdots \quad +2^{h-1})+h\cdot 2^h, \\
 & = & -(2^h+1)+h2^h, \\
 & = & 2^h(h-1)+1.
\end{array}$$

where h is the number of levels in the *BST*. Substituting $N = 2^h - 1$, gives

$$T_h = (N+1)(\log(N+1)-1)+1$$

The average time, $T_{average}$, for finding a node in a full balanced *BST* is:

$$\begin{array}{rcl}
T_{average} & = & \frac{T_h}{N}, \\
 & = & \frac{(N+1)(\log(N+1)-1)+1}{N}, \\
 & = & (1+\frac{1}{N})(\log(N+1)-1)+\frac{1}{N}, \\
 & \approx & \log N, \text{ when } N \text{ is large.}
\end{array}$$

So the average access time for a full *BST* is $O(\log N)$. When we consider balanced *BST*s that are not fully populated it is clear that $(1+\frac{1}{N})(\log(N+1)-1)+\frac{1}{N}$ is an upper bound for their average search length.

The further observation that the majority of the nodes lie in the bottom level of a balanced tree ensures that this bound is reasonably close to the lower bound for the average number of probes required for access in a uniformly distributed balanced *BST*.

4.1.4 Probability of the Best case

A full and balanced binary search tree with N elements, has $\log(N+1)$ levels. In the best case the target node appears at the root of the tree. The probability of looking up this node is: $\frac{1}{N}$. So, the best case which has an access time of 1 occurs only $\frac{1}{N}$ of the time.

4.1.5 Probability of the Worst Case

Since in a complete *BST*, half of the elements are situated in the bottom level of the tree, the probability of the worst case occurring is: $\frac{1}{2}$.

4.2 Recursive Versus Iterative Implementation of a *BST*

Most of the operations in binary search trees necessitate descending from the root down the tree level by level. Half of the elements in a full *BST* are situated in the bottom level of the tree. Given this the target is situated at the bottom level, with a probability of $\frac{1}{2}$, and that it takes $O(\log N)$ to reach the target when it is at the bottom level, the number of comparisons for doing lookups in a *BST* is bounded by the height of the tree $O(\log N)$ even if the target is situated at the bottom level of the tree.

A *BST* has a pointer to the root node and each node in a *BST* has two extra references, namely pointers called `left` and `right` respectively. The search operation starts at the root of the tree. To do a lookup set the value of the `current` pointer to the value of the `root` using

```
current = root;
```

Next a comparison is made with the target key. If the target key, `id`, has been found, return `current`. If the target key is not present in the `current` node and is greater than its key, select the right sub-tree, otherwise select the left sub-tree to search further. The next comparison is made at the selected node.

```
if (id == root.key)
   return current;
else if (id>current.key)
   current = right;
else
   current = left;
```

The same three cases are repeatedly tested until the procedure terminates successfully and returns a pointer to the *target* node or returns a `null` pointer if the search was unsuccessful. The function procedure `lookup` accesses a tree called `T` and attempts to find a target node with the value contained in id is invoked as follows.

```
tree T;
   ...
   if (T.lookup(id) != null)
   ...
   else
      error(id + " not found");
```

The type of the 'node' is a subtree which we call 'tree'. We first show an *iterative* implementation of lookup

```
tree lookup (keytype id){
tree current = root;
   while (current != null){
      if (id == current.key)
         return current;
      else if (id > current.key)
         current = right;
      else
         current = left;
      }
   return current;
   }
```

We present the same algorithm *recursively.*

```
tree lookup (keytype id){
   if (root == null)
      return null;
   else if (id == root.key)
      return root;
   else if (id > root.key)
      return lookup(root.right id);
   else
      return lookup(root.left, id);
   }
```

On inspection of the code generated by the recursive version, we observe that it incurs some overhead not required by the iterative version. Whereas the pointer current is simply updated in the iterative version, a procedure call is executed with all its incumbent overhead including the usage of stack space for each new node inspected. Both procedures rely on the global value of a 'root', in the case of the iterative procedure root refers to the unique root of the tree T being traversed, and in the recursive procedure root refers to a pointer to the current node.

In Chapter 6 on Page 100 We show that the recursive procedures run at a constant time slower that the iterative ones.

Chapter 5

Theoretical Results

5.1 *AVL* Trees

The *AVL* tree uses balance conditions to ensure that the tree is almost balanced and its height is logarithmic. During the process of insertion and deletion, the balance condition is checked and either the balance criteria is changed or the tree is reconstructed by making the appropriate rotations. The tree can be reconstructed by a single *left* or *right rotation*, or it can be reconstructed by *double rotation* (Baer and Schwab, 1977).

Suppose that x is the node to be rebalanced. If the insertion occurs into the left sub-tree of the left child of x, or into the right sub-tree of the right child of x, then a single rotation between x and its child is required. If the insertion occurs into the right sub-tree of the left child of x, or into the left sub-tree of the right child of x, then a double rotation is required. A double rotation is two rotations: the first rotation is performed between x's grandchild and x's child. The second rotation is performed between x and its grandchild (Adel'son-Vel'skiĭ and Landis, 1962). The order in which the data is inserted into the tree can affect the type and the number of rotations.

5.1.1 Inserting Sorted Data into an *AVL* Tree

If the data is inserted in ascending or descending order, into an *AVL* tree, the balance is maintained by changing the balance criteria or making a single rotation. There is no need for double rotation at all. The rotation can be single left or single right depending on the order of the data. If the data is inserted in ascending order, the required rotation is left, but if the data is entered in descending order, the required rotation is right. In both cases, inserting the following items in order, does not need rotation: Items numbered 1, 2, 4, 8, ... If N sorted items are inserted into an *AVL* tree, then the required number of rotations is:

$$\text{Rotations} = N - (\log(N) + 1) \tag{5.1}$$

This means that only $(\log(N) + 1)$ cases do not need rotation. The probability of rotations (P_r) for ordered data is:

$$P_r = \frac{N - (\log(N) + 1)}{N} \tag{5.2}$$

where N is the number of items in the tree. The probability of cases which do not need rotation (P_f) is:

$$P_f = \frac{\log(N) + 1}{N}. \tag{5.3}$$

where N is the number of items in the tree.

Note: When calculating the log, using the above formula, ignore the decimal places.

In the best case there is no need for rotation. Worst case needs one rotation, and the average number of rotations is $\frac{N-(\log(N)+1)}{N} \approx 1$ rotation.

Table 5.1 shows our theoretical results for inserting ascending or descending sorted data into an *AVL* tree.

When sorted data items are inserted into an *AVL* tree, the tree is always balanced and the height of the tree is logarithmic. In the worst case the height of the tree is $1 + \log N$, in the best case the height of the tree is $\log(N + 1)$ and the average height is $0.5 + \log(N)$. The worst case is when item number 2^i is entered into the tree. The best case is when item number $2^i - 1$ is entered into the tree. We will see that, when the same number of unsorted data items are inserted into an *AVL* tree, the tree is balanced according to the *AVL* tree conditions, but it may not be a full and complete *BST*.

Table 5.2 shows our theoretical results for the number of nodes and number of levels, when sorted data is inserted into an *AVL* tree.

After inserting the items numbered 1, 3, 7, 15, 31, 63, 127, and so on, all the leaf nodes have equal height and the tree becomes a full *BST*. This means that, the tree is full if the number of items in the tree is $2^k - 1$, where k is any integer value from 0 to ∞.

Most of the rotations are performed on newly inserted nodes. Our manual

Table 5.1: Required number of rotations for inserting data in sorted order.

Number of Items	Total Rotations	Rotation probability
1	0	0.0000
2	0	0.0000
3	1	0.3333
4	1	0.2500
8	4	0.5000
16	11	0.6875
32	26	0.8125
64	57	0.8906
128	120	0.9375
256	247	0.9668
512	502	0.9805
1024	1013	0.9893
2048	2036	0.9941
4096	4083	0.9968
8192	8178	0.9982
16384	16369	0.9991
32768	32752	0.9995
65536	65519	0.9997
131072	131054	0.9998
262144	262125	0.9999
524288	524268	0.9999
1048576	1048555	0.9999

analysis shows that, after inserting the items numbered 3, 6, 12, 24, 48, 96, 192, 383, 768, 1536..., the rotation is performed around the root of the tree. Therefore, after inserting the above items, the process of rotation changes the root of the tree. When the root is changed, the number of items in the left sub-tree becomes $2m = 1$, where m is the number of items in the left sub-tree, before changing the root. In the same way, the number of items in the right sub-tree is halved. This is the case when the data is inserted in ascending order. The symmetric case is correct when the data is inserted in descending order. The above sequence of numbers shows that, the process of rotation changes the root of the tree, when the number of items is doubled.

Table 5.3 shows the old and the new root for the tree after inserting the above sequence of items:

According to our observations the total number of comparisons for inserting sorted data items into an AVL tree is: $\frac{N}{2} \log N + C$, where C is the total number of required comparisons for inserting $\frac{N}{2}$ items. The best case is when the first item is inserted into the tree. The worst case needs $1 + \log N$ comparisons. The average

Table 5.2: Number of items versus levels.

Number of Items	Number of Levels
1	1
2–3	2
4–7	3
8–15	4
16–31	5
32–63	6
64–127	7
128–255	8
256–511	9
512–1023	10
1024–2047	11
2048–4095	12
4096–8191	13
8192–16383	14
16384–32767	15
32768–65535	16
65536–131071	17
131072–262143	18
262144–524287	19
524288–1048575	20

Table 5.3: The old and the new root after doubling the number of items.

Number of Items	Old root	new root
3	1	2
6	2	4
12	4	8
24	8	16
48	16	32
96	32	64
192	64	128
383	128	256
768	256	512
1536	512	1024

number of comparisons is $\frac{N \log N + 2C}{2N}$. The average and worst cases are nearly the same because this BST has half of its items stored in bottom level—so that half of the elements appear in the worst case.

Table 5.4 shows the total, and the average number of comparisons, for inserting sorted data items into an AVL tree.

Table 5.4: Total and average number of comparisons for inserting sorted data.

Number of Items	Total Comparisons	Average Comparisons
2	1	0.5000
4	5	1.2500
8	17	2.1250
16	49	3.0625
32	129	4.0313
64	321	5.0156
128	769	6.0078
256	1793	7.0039
512	4097	8.0020
1024	9217	9.0010
2048	20481	10.0005
4096	45057	11.0002
8192	98305	12.0001
16384	212993	13.0001
32768	458753	14.0000
65536	983041	15.0000
131072	2097153	16.0000
262144	4456449	17.0000
524288	9437185	18.0000
1048576	19922945	19.0000

5.1.2 Inserting Unsorted Data into an *AVL* Tree

When unsorted data items are inserted into an *AVL* tree, the required rotation, for keeping the tree balanced, may be single left or single right or double. Our experimental analysis shows that, 25% of the rotation is left, 25% is right and 50% is double. The total cases which need rotation is 46% of N, where N is the number of items. Even though, the number of required rotations for inserting unsorted data is not fixed, but for all $N \geq 8$, it is less than the required number of rotations when the same number of items were inserted in a sorted order. For sorted insertion the percentage of rotations increases when the number of items increases, however, for unsorted insertion the percentage of rotations is always around 46%.

Table 5.5 compares the required number of rotations for sorted and unsorted insertion into an *AVL* tree.

When the sorted data items are inserted into an *AVL* tree, the tree will be full if the number of elements is 3, 7, 15, 31, 63,..., and so on. But this is not the case for unsorted data. For example, the following 7 elements will not generate a full binary search tree: 6, 3, 4, 15, 20, 10, 12. On the other hand, the insertion of the above sequence of unsorted items, does not require rotation.

Table 5.5: Required number of rotation for sorted and unsorted insertion.

Number of Items	Total Rotations for sorted data	Total Rotations for unsorted data
32	26	14
64	57	29
128	120	58
256	247	117
512	502	238
1024	1013	472
16384	16369	7591
32768	32752	15277
65536	65519	30539
262144	262125	122149
1048576	1048555	488218

The average number of rotations for inserting sorted data items into an *AVL* tree is $\frac{N-(\log(N)+1)}{N}$, where N is the number of items in the tree. It means, the number of required rotations depends on the number of items in the tree. But the exact number of required rotations for inserting unsorted data items into *AVL* tree depends on the number of items as well as the insertion order of the items. That means, the same number of items with different orders may need different number of rotations.

In the case of unsorted data the height of the tree is close to logarithmic. Our experimental analysis shows that the height of the tree will be $1 + \log N$ up to $4 + \log N$.

5.1.2.1 *AVL* Search Cost

The search is started from the root of the tree and descending one level at a time. Each level need only one comparisons. In the best case the search cost is $O(1)$ and in the worst case the search cost is equal to the number of levels in the tree. In the worst case the number of levels are equal to $4 + \log(N)$, therefore, the worst search time in *AVL* tree is $4 + \log(N)$.

In a full *AVL* tree, since each level contains 2^i items, where i is the number of levels. And each item in level i needs i comparison therefore, the total search cost for accessing all the items is:

$$\sum_{i=0}^{\log(N+1)-1} 2^i(i+1) \tag{5.4}$$

The average search cost is:

$$\frac{\sum_{i=0}^{\log(N+1)-1} 2^i(i+1)}{N} \tag{5.5}$$

5.2 Skip Lists

Skip lists are an alternative to balanced *BSTs*. They use a probabilistic balancing method, rather than strictly enforced balancing (Pugh, 1990). Skip lists with N elements has search and update times near to $O(\log N)$ on average. Skip list has a probabilistic time bound. The worst case behavior of skip lists is worse than that of linear lists, because of the extra overheads for handling the lists, but the probability for the worst case to occur is very low (Pugh, 1990). The time bounds for the average and worst cases depend on the random numbers generated and the number of items in the list. The time bound does not depend on the probability distribution of the keys in the input (Pugh, 1990). But, it still depends on the distribution of items in the upper levels.

5.2.1 Structure of a Skip Lists

A skip list is a series of lists in which the nodes are arranged horizontally in levels and vertically in towers. Each node contains a key, its data, and one or more pointers. The number of pointers, which is also called node level, for each node is determined randomly. Each node has at least one pointer, and the first of them always points to the next node in the skip list. From this point of view a skip list is like a linked list. The additional pointers are used to skip one or more intermediate nodes. Each list has two extra nodes. The first node determines the beginning of the list and the second node determines the end of the list. The first node has a key smaller than every node, and the last node has a key greater than every node. These two nodes can be $-\infty$ and $+\infty$ respectively. Other required properties for a skip list is probability, maximum level, and current overall level.

For the convenience of the reader Figure 5.1 duplicates the skiplist in Figure ??.

The order in which the data is inserted into a skip list does not affect the structure and performance of the skip list. The structure of the skip list is determined only by the number of elements in the skip list and the result of consulting the random number generator. In other words, the number of levels and the distri-

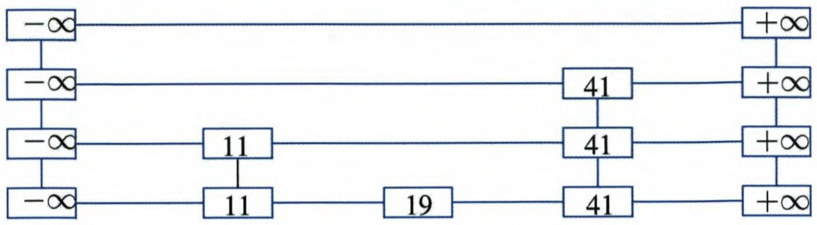


Figure 5.1: A skip list.

bution of items in the levels are determined only by the number of items and the process of randomization. Beside the number of levels, the distribution of items in the upper levels also affects the performance of the skip list. The average number of comparisons for the search and update operations depends on the number of levels and the distribution of items in the levels. The first or lowest level contains all the items. The items will be distributed randomly in the upper levels.

Theoretical analysis can be used to understand the possible number of levels and the distribution of elements *in the levels.* The analysis helps us to understand the performance of the skip list. The number of levels, the number of items in each level, and even the positions of items in the level affect the search and update performance and the number of comparisons for searching an item. The position of items, which reach to the top level, h, positions of elements which reach to the level $h - 1$, and so on, also affect the number of required comparisons, during the search and update operations.

The following are our observations for skip list performance. Our analysis uses simple probability. For finding the average and worst cases, we consider the situations which have higher probability than the situation with lower probability.

5.2.1.1 Probability for the Number of Levels

The number of levels is one of the factors that affects the performance of skip lists. The number of comparisons will increase or decrease according to the number of levels. The number of levels can be compared with the height of a *BST*. If the height of a *BST* is not logarithmic, then the number of comparisons for search and update operations increases.

For a balanced *BST*, we restrict the height of the tree to a minimum by changing the properties of the elements or by restructuring the tree through the process of rotation. But for a skip list the levels are generated randomly. Depending on the number of items in the skip list, more or less levels will be generated. When the number of items increase, the probability for having more levels also increase. It is also possible to have a different number of levels for the same number of items. We now analyze the probability for different numbers of levels.

Assume that, the probability for moving an item from level i to level $i+1$ is $\frac{1}{2}$. Further, assume that the keys of the items are uniformly distributed. Based on these assumptions, the probability for reaching each of these items, to level 1, level 2, ..., level h is as follows:

$$\begin{aligned} Level\ 1 &= \tfrac{1}{2}, \\ Level\ 2 &= \tfrac{1}{4}, \\ &\vdots \\ Level\ h &= \tfrac{1}{2^h}. \end{aligned}$$

Proof:

Let p_1 be the event of reaching an item to level 1, and p_2 be the event for reaching the same item to level 2. Then the probability of reaching the same item to level 2 is the intersection of p_1 and p_2:

$$P = p_1 \cdot p_2 = \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$$

Similarly we can prove the reachability of the same item to the higher levels.

Generally, the probability of reaching level h is:

$$P = p_1 \cdot p_2 \cdot \ldots \cdot p_h$$

Since the probability of reaching an item from level i to level $i+1$ is $\frac{1}{2}$, it follows that, the probability of reaching level h is:

Fact 1:

$$P = \frac{1}{2^h}.$$

If a skip list has N items, then the probability of reaching m items to level h, is:

Fact 2:

$$P = \binom{N}{m}\left(\frac{1}{2^h}\right)^N = \frac{N!}{m!(N-m)!}\left(\frac{1}{2^h}\right)^N.$$

Fact 3: Considering *Fact 1* and *Fact 2*, it follows that for all even N, half of the items will move from level i to level $i+1$ with high probability. And for all odd N, $\frac{N}{2} \pm \frac{1}{2}$ items will move from level i to level $i+1$. Both cases have equal possibility.

Fact 4: Consequently, *Fact 3* shows that the probability of having $\log N$ level is higher than having less or more than $\log N$ levels. Therefore, a skip list with N elements will has $1 + \log N$ levels, including the base level which contains all the items, with high probability.

We can also prove that the probability of having $\log N$ levels (except the base level) is higher than having $1 + \log N$, $2 + \log N \ldots$ levels.

Let P be the probability of having $\log N$ levels, then if at least one item reaches level $\log N$, the skip list will have $\log N$ levels. Let us find the probability of reaching at least one item to level $\log N$:

$p = (N-1)\frac{1}{2^{\log N}} = \frac{N}{2^{\log N}} - \frac{1}{2^{\log N}} = \frac{N-1}{N} \approx 1.$

The probability of reaching one of the items to level $\log N + 1$, and consequently having $\log N + 1$ levels is:

$P = N^{\log N+1} = \frac{N}{2^{\log N+1}} = \frac{N}{N2^1} = \frac{1}{2} \Rightarrow 50\%$

The probability of reaching one of the items to level $\log N + 2$, and consequently having $\log N + 2$ levels is:

$P{=}N^{\log N+2} = \frac{N}{2^{\log N+2}} = \frac{N}{N2^2} = \frac{1}{4} \Rightarrow 25\%$

The probability of reaching one of the items to level $\log N + 3$, and consequently having $\log N + 3$ levels is: $P{=}N^{\log N+3} = \frac{N}{2^{\log N+3}} = \frac{N}{N2^3} = \frac{1}{8} \Rightarrow 12.5\%$

Generally, the probability for reaching one of the items to level $\log N + m$ is: $P = N2^{\log N+m}$, where m is an integer.

This means that the probability for having $\log N$ levels is higher than the probability of having more than $\log N$ levels. In other words, the probability of having $O(\log N)$ levels is higher than having $O(\log N) \pm m$ levels.

Table 5.6 shows the probability of distribution of items in the levels, and consequently the probability of the number of levels.

Results:

1. The probability of having $\log N$ levels is higher than the probability of not having $\log N$ levels.

Table 5.6: Probability for the distribution of items over the levels.

Level	Positive Probability	Negative Probability
$\log n$	1.000000	0.0000000
$\log n + 1$	0.500000	0.5000000
$\log n + 2$	0.250000	0.7500000
$\log n + 3$	0.125000	0.8750000
$\log n + 4$	0.625000	0.9375000
$\log n + 5$	0.312500	0.9687500
$\log n + 6$	0.156250	0.9843750
$\log n + 7$	0.078125	0.9921875

2. The probability of having $1 + \log N$ levels is equal to the probability of not having $1 + \log N$ levels. If the number of levels is greater than $\log N$ than the search operations will not be logarithmic and the number of comparisons for search and update will increase. The probability of this happening is 50%. Therefore, if we do not stop the moving of elements to the level above $\log N$ then there is a 50% probability of moving one of the elements to level $1 + \log N$, and 25% for moving to level $2 + \log N$ and so on.

3. The probability of an element to reach level $7 + \log N$ is 0.78125%

   which is very low. This means that having $7 + \log N$ levels has 0.0078125 probability of all values of N.

5.2.1.2 Distribution of Items in Each Level

Another consideration about the structure of the skip list is the number of items in each level. The search and update operations are affected by the distribution of items in different levels. In the previous section we proved that half of the items will move from level i to level $i + 1$ with high probability. In this section we will see the effect of distribution of items in the levels.

Suppose that we have $\log N$ levels in a skip list. Our theoretical analysis show that, the skip list will behave well if we have only one item in the middle of the top level, which is level number $\log N$. In this case the performance of the skip list will be better than having more than one item in the top level. In the same way the number and distribution of items in the rest of levels also affect the performance of the skip lists. To analyze the performance of the skip list from this point of view we intend to find the probability of more than one item reaching level $\log N$.

- Let us see the probability of all the items reaching level $\log N$. If we have N items then the probability of all the items reaching level $\log N$ is the intersection of their probabilities:

  $$P = p(a_1) \cdot p(a_2) \cdot \ldots \cdot p(a_n)$$

  Since the probability for each item to reach level $\log N$ is $\frac{1}{2^{\log N}}$ then:

  $$P = (\frac{1}{2}^{\log N})^N$$

  The above result shows that the probability of all the items reaching level $\log N$ is very low. For example if there are 4 items then the probability of all the items reaching level 2 ($\log n$) is:

  $$(\frac{1}{2^{\log N}})^N = \frac{1}{256} = 0.0039 = 0.39\%$$

  If we have 8 items in a skip list, then the probability for reaching all the items to level $\log N$ which is level 3 is:

  $P = (\frac{1}{2^3})^8 = \frac{1}{16777216} = 0.00000005 = 0.000005\%.$

  $(2^{\log N})^N = \frac{1}{2^{N \log N}}.$

- The probability of all the items reaching level $\log N - 1$ is:

  $$P = (\frac{1}{\frac{N}{2}})^N = (\frac{2}{N})^N$$

  Therefore, the probability for this to happen, is very low.

  Table 5.7 shows the probability for different values of N.

- The probability of all the elements reaching level $\log N - 2$ is:

  $$P = (\frac{1}{\frac{N}{4}})^N = (\frac{4}{N})^N$$

  The probability for this to happen is very low.

Table 5.7: Probability of reaching all the items to level $N-1$.

Number of elements	Probability
2	N/A
4	6.25%
8	0.0%

Table 5.8 shows the probability for different value of N.

Table 5.8: Probability for reaching different values of N.

Number of elements	Probability
8	0.39%
16	0.0%

The probability of all the items reaching level $\log N$, $\log N - 1$ and $\log N - 2$ is very low and is close to zero. When the number of items increases, the chance of all the items reaching the upper levels decreases.

The probability of 2 elements reaching level $\log N$ is:

$\binom{N}{2} p^2 q^{N-2} = \frac{N!}{2!(N-2)!} \cdot (\frac{1}{2^{logN}})^N = \frac{N!}{2!(N-2)!} \cdot (\frac{1}{N})^N = \frac{n(N-1)}{2.N^N}$ Therefore, the probability for moving more than one item to level $\log N$ is $\frac{N(N-1)}{2.N^N}$.

Our experimental result also show that the distribution of items affect the performance of the skip list.

Results:

1. All the items reaching level $\log(N)$, $\log(N) - 1$, and $\log(N) - 2$, results in a worst case operation for the skip list. But the probability of all the items reaching the upper levels is very low and nearly zero. When the number of items increases in the skip list the probability for all of the items reaching the uppermost level decreases.

2. The probability of 2 items reaching level $\log N$ is also very low. Two items reaching to the top level causes the number of comparisons to increase during search and update operations.

5.2.1.3 Performance of the Skip List

The best performance of a skip list depends on the number of levels and the distribution of the keys in the levels. A skip list will behave like a balanced BST if the item which represents the middle of the bottom list reaches level $\log N$, and the items which represent the quarter of the bottom list reach to the level $\log(N) - 1$, and so on. In this case the search and update operations of the skip list will be identical to balanced BSTs such as AVL trees and Red-Black trees, with the ease of implementation. The worst-case performance will be $\log N$. But, the structure of the skip list is not determined by the input order of the data and also it is not reconstructed by some process such as rotations. The structure of the skip list is determined by the number of items and the distribution of the items in the levels. In the following sections, we identify the best, average and worst search performance of the skip list.

5.2.1.4 Skip List Insertion

To insert a new item into a skip list, we first search the list to find the place for the new item. During the search, the references of items at which the search dropped down one level are kept in an array. When the place is found the item level is retrieved from the random level algorithm. Next the new item is created and it is placed in the skip list. This is done by assigning the forward references stored in the array (Pugh, 1990).

Our analysis shows that the average number of comparisons for inserting a sequence of items into a skip list depends on:

- The number of levels,
- The distribution of items in the levels, and
- The order in which the data is inserted into the skip list.

The required number of comparisons for inserting a new item into a skip list is equal to the search comparisons, to find the place for the new item. In the next section, when we analyze the search operation for the skip lists, we will show that the number of levels and the distribution of items in the levels affect the search cost. Therefore, the number of levels and the distribution of items in the levels also affect the process of insertion. The order in which the data is inserted into a skip list, does not affect the average number of search comparisons. But, our analysis shows that the order in which the items are inserted into a skip list affects the average number of comparisons for inserting data into a skip list. Our analysis for inserting items into a skip list follows.

Our analysis shows that, when the data is inserted in descending order the average number of comparisons is equal to the number of levels in the skip list. Since the number of levels is logarithmic, the average number of comparisons is also logarithmic. In other words, when the data is inserted in descending order after each comparison the pointer is dropped down one level. This is exactly the same as with a *BST*. In a *BST* we need only one comparison at each level. Therefore, the worst case insertion cost for inserting descending ordered data is $1 + \log N$. But, when the data items are inserted in ascending order, the required number of comparisons at each level is not restricted to one comparison. But, the number of comparisons depends on the number of items traversed before reaching the newly inserted item's position. Since the data in the skip lists is in ascending order, some levels need more than one comparison, and the average number of comparisons is not logarithmic.

Suppose that we want to insert 3 items with keys 10, 20, and 30 into a skip list. With a high probability such a skip list will have two or three levels . In either case—2 levels or 3 levels—the descending ordered insertion requires less comparisons on average.

The average number of comparisons for inserting the above 3 items in descending order is 3.00 comparisons. But the average number of comparisons for inserting the same 3 items in ascending order is 3.78 comparisons. Even the descending insertion cost is better than random insertion cost. However, the random insertion cost is better than ascending ordered insertion. Our experimental results support this theory.

Table 5.9 shows our manual analysis for inserting 3 items in different orders. For each order, the average was calculated from 3 different distributions of items in the levels.

A similar manual analysis shows the same result, when seven items with keys $10, 20, 30, 40, 50, 60,$ and 70 are inserted into a skip list. The analysis shows that the insertion cost is lower if these seven items were inserted in descending order. The average insertion cost for descending insertion is 4.0000 where the average

Table 5.9: Manual analysis for inserting items in different order.

Insertion Order	Average Comparisons
10, 20, 30	3.78
10, 30, 20	3.67
20, 10, 30	3.44
20, 30, 10	3.33
30, 10, 20	3.33
30, 20, 10	3.00

insertion cost for ascending insertion is 5.7856 comparisons.

Table 5.10 shows our analysis for inserting seven items in three different orders. For each order, the average was calculated from four different distributions of items in the levels.

Table 5.10: Analysis of inserting seven items in different orders.

Insertion Order	Average Comparisons
10, 20, 30, 40, 50, 60, 70	5.79
40, 20, 60, 10, 50, 70, 30	5.21
70, 60, 50, 40, 30, 20, 10	4.00

Our empirical results support the above results. That is, the descending ordered insertion cost is less than the ascending and random ordered insertion cost.

5.2.1.5 Skip Lists Search Performance

Search for a key starts at the header of the top level, and continues moving forward, if the node key is smaller than the search key. If the node key is equal or greater than the search key, the search drops down one level and then continues forward. This process is continued until we find the target node or we show that the target node is not in the skip list (Pugh, 1990).

On average the cost of the search is logarithmic. We show that, the search cost for an item depends on the number of levels and the distribution of items in the levels. Our approach is to determine the best, average, and worst behaviors of the skip list from the cases which have higher probability. The following is our analysis for the search operation in a skip list.

Suppose that three items with keys 10, 20, and 30 are inserted into a skip list. For these three items, the skip list will have two or three levels with high probability. Consequently, a skip list with three items has two or three levels with equal probability. If one item is moved to level two then the probability for having a third level is not higher than the probability for not having the third level. But if two items were moved to level two, then with high probability one of these two items will move to level three. For both cases we have different distributions of items in the levels, which consequently affect the performance of the skip list.

Let us first consider Scenario 1 in which we have two levels.

Table 5.11 shows our analysis for Scenario 1:

Table 5.11: Analysis for Scenario 1.

Distribution of Items	Total Comparisons	Average Comparisons
10 reach to level 1	8	2.67
20 reach to level 1	6	2.00
30 reach to level 1	6	2.00
Overall Average	6.67	2.22

Results

- Table 5.11 shows that the distribution of items in the levels affects the performance of the skip list. The average number of comparisons for search increases or decreases depending on the distribution of items in the levels.

- Our analysis shows that, when the first item reaches level two, the average number of comparisons for searching an item is greater than when the second and third items reach to level two. The reason is when the middle item reaches the second level it keeps the distribution of items in balance, and when the last item reaches the second level, it eliminates some comparisons with the end pointer. As a result, the performance of the skip list is better if the items which are at the end of the skip list reach the top levels. But, if the items which are at the front of the skip list reach the higher levels, the performance of the skip list is lower.

- When the number of items is 3, the probability for having the above two best cases, in which the middle and last items reach to the top levels, is 0.67. But this probability decreases when the number of items increases.

- If these three items were inserted into a balanced *BST* such as *AVL* tree, then the average search cost is 1.67 comparisons. Whereas in the skip list the average search cost is 2.22 comparisons. Therefore, comparing skip lists with *AVL* trees, *AVL* trees need less comparisons for searching an item. When we compare the search efficiency between skip lists and *AVL* trees, we ignore the required number of rotations during *AVL* update operations. If no item reaches to level 1, the skip list behaves like a linked list. In this case the best case will be $O(1)$, the worst case is $O(N)$ and the average case is $\frac{N(N+1)}{2N}$. But the probability for such a worst case to happen is very low. A skip list with 3 items, has the probability of 0.125 (12.5%), for the worst case to occur. The probability decreases when the number of items increases. For example if the number of items is 8, then the probability for skip list to become like a linked list is 0.0039 (0.39%). Generally, the probability for a skip list to behave like a linked list is $\frac{1}{2^N}$. This shows that the probability decreases when the number of items increases.

Now let us consider the Scenario 2, which has 3 levels.

Table 5.12 shows our analysis for Scenario 2:

Table 5.12: Analysis for Scenario 2.

Distribution of Items	Total Comparisons	Average Comparisons
10 reach to level 3 and 20 to level 2	9	3.00
20 reach to level 3 and 10 to level 2	7	2.33
10 reach to level 3 and 30 to level 2	8	2.67
30 reach to level 3 and 10 to level 2	7	2.33
20 reach to level 3 and 30 to level 2	7	2.33
30 reach to level 3 and 20 to level 2	6	2.00
Overall Average	7.33	2.44

Results

- Our analysis for having 3 levels also shows that, the distribution of items in the levels affects the performance of the skip list. That is, the search cost depends on the distribution of items in the levels.

- Our analysis illustrates another interesting point: when the items which are at the front of the skip list reach the highest level, the average number of comparisons for searching the items increases. The best case is when the middle and the last items reach to the higher levels. Especially, when the last item reaches to the higher level h, and the middle item reach to level $h-1$. The reason is the same as it was for having two levels. That is, when the middle item reaches the top level, it keeps the items in balance, and when the last item reaches the second level, it eliminates some comparisons with the end pointer.

- The probability for having the above two best cases is 0.33, which is less than the probability for having the best case when we have two levels. Similarly, this probability decreases when the number of items increases.

- Compared with a balanced *BST* such as an *AVL* tree, the required number of comparisons for *AVL* tree is 1.67. But a skip list with 3 levels, require 2.44 comparisons on average.

- Our theoretical analysis shows that, a skip list with 3 items is more efficient if it has two levels rather than three levels. The average number of comparison for a search will increase by the factor of 1.099 if there are 3 levels. But, these two cases are equiprobable.

As a result, skip lists with 3 items have 2 or 3 levels with high probability, and the average number of search comparisons for these two cases, is 2.33 comparisons.

Now suppose that seven items with keys $10, 20, 30, 40, 50, 60$, and 70 are inserted into a skip list. For a skip list with seven items the high probability is for moving three or four items to level two. If three items were moved to level 2 then the high probability is for moving one or two items to level three. Their probability is equal. If four items were moved to level two then two items will move to level 3 with high probability. Considering both cases together, level three contains two items with high probability. Therefore, with high probability, one of these two items moves to level four. Consequently, when seven items are inserted into a skip list the number of levels has a high probability of reaching four. A skip list with four levels has different shapes depending on the distribution of items in the levels. From the probabilistic facts we know that, all the items have equal possibility for reaching the upper levels. This distribution affects the search performance of the skip list. We consider three scenarios:

1. A skip list with four levels in which four items are in level 2, two items are in level 3, and one item is in level 4.

2. A skip list with four levels, in which 3 items are in level two, two items are in level 3, and one item is in level 4.

3. A skip list with three levels, in which 3 items are in level two, one item is in level 3.

Table 5.13 shows our analysis for the distribution of items in the levels considering the Scenario 1:

Table 5.13: Distribution of items considering the Scenario 1.

Distribution of Items	Total Comparisons	Average Comparisons
40 = L4; 60 = L3; 10 and 30 = L2	29	4.14
40 = L4; 70 = L3; 20 and 50 = L2	26	3.71
40 = L4; 10 = L3; 20 and 60 = L2	28	4.00
40 = L4; 70 = L3; 30 and 60 = L2	25	3.57
40 = L4; 70 = L3; 50 and 60 = L2	28	4.00
40 = L4; 70 = L3; 20 and 60 = L2	25	3.57
40 = L4; 60 = L3; 20 and 50 = L2	26	3.71
10 = L4; 30 = L3; 20 and 60 = L2	33	4.71
10 = L4; 20 = L3; 30 and 40 = L2	42	6.00
70 = L4; 60 = L3; 40 and 50 = L2	25	3.57
10 = L4; 70 = L3; 20 and 60 = L2	34	4.86
10 = L4; 30 = L3; 50 and 60 = L2	34	4.86
10 = L4; 60 = L3; 20 and 70 = L2	34	4.86
Overall Average	29.92	4.27

Results

- The above table shows that, the distribution of items in the levels of a skip list with 7 items also affects the performance of the skip list. The required number of comparisons for searching depends on the distribution of items in the levels.

- The above table also shows that when the items at the middle and at the end of the skip list reach higher levels, then the performance will be better than, when the items which are in the beginning of the list reach to higher levels. The reason is the same as it was with a skip list with 3 items. When the middle item reaches the second level it keeps the distribution of the items in balance, and when the last item reaches the second level, it eliminates some comparisons with the end pointer.

- From the above table the average number of comparisons for searching an item in the skip list with seven items is 4.2746. We have 4 levels in the skip list. Therefore, the average number of comparisons is close to the number of levels as it was in the skip list with 3 items.
- Compared with *AVL* trees, if these seven items were inserted into an *AVL* tree, then the average search cost is 2.14 comparisons.

Now let us consider the Scenario 2, in which we have four levels but we have 3 items in level two, two items in level three, and one item in level four.

Table 5.14 shows our analysis considering the Scenario 2:

Table 5.14: Distribution of items considering the Scenario 2.

Distribution of Items	Total Comparisons	Average Comparisons
40 = L4; 20 = L3; 60 = L2	27	3.86
40 = L4; 60 = L3; 20 = L2	27	3.86
40 = L4; 10 = L3; 70 = L2	29	4.14
40 = L4; 70 = L3; 10 = L2	29	4.14
10 = L4; 40 = L3; 30 = L2	34	4.86
10 = L4; 20 = L3; 30 = L2	43	6.14
70 = L4; 60 = L3; 50 = L2	28	4.00
40 = L4; 70 = L3; 20 = L2	27	3.86
40 = L4; 70 = L3; 30 = L2	27	3.86
10 = L4; 30 = L3; 60 = L2	34	4.86
40 = L4; 70 = L3; 60 = L2	28	4.00
10 = L4; 30 = L3; 70 = L2	35	5.00
10 = L4; 70 = L3; 30 = L2	34	4.86
Overall Average	30.92	4.42

Results

- Table 5.14 also shows that, when the items at the middle and at the end of the skip list reach higher levels the performance is better than when the items which are at the beginning of the list reach to the higher levels.

- In Table 5.14 the average number of comparisons for searching an item in the skip list with seven items is 4.42, which is almost the same as with having 4 items in level two.

Now let us consider the third scenario, in which we have three levels. Three items are moved to level two and one item is moved to level three.

Table 5.15 shows our theoretical analysis for the Scenario 3:

Table 5.15: Distribution of items in a skip list considering Scenario 3.

Distribution of Items	Total Comparisons	Average Comparisons
40 = L3; 20 and 60 = L2	22	3.14
40 = L3; 10 and 70 = L2	24	3.43
70 = L3; 30 and 50 = L2	21	3.00
70 = L3; 40 and 60 = L2	22	3.14
10 = L3; 30 and 50 = L2	30	4.29
70 = L3; 40 and 60 = L2	22	3.14
70 = L3; 10 and 40 = L2	24	3.43
30 = L3; 10 and 70 = L2	25	3.57
10 = L3; 20 and 30 = L2	38	5.43
70 = L3; 50 and 60 = L2	24	3.43
Overall Average	25.20	3.60

Results:

- Table 5.15 also shows that when the items at the middle and at the end of the skip list reach to the higher levels the performance will be better than when the items which are in the beginning of the list reach to the higher levels.

- In Table 5.15 the average number of comparisons for searching an item in the skip list with seven items is 3.60. This is better than when the skip list has 4 levels. Although, the probability for having four levels is higher than having 3 levels.

- The average number of comparisons is still worse than for *AVL* trees.

Deleting an Item from a Skip List Deleting an item from a skip list is similar to inserting an item into a skip list. To delete an item, first we search the skip list to find the item to be deleted. If it is found, we update the references. To do this, reassign the nodes with references to the node to be deleted, to the node that comes after the node to be deleted. As a last step, we check to see if we have deleted the maximum element of the list and if so, we decrease the maximum level of the skip lists.

In the previous section we saw that the search cost depends on the number of levels and the distribution of items in the levels. However, the search cost is logarithmic on average. Therefore, deleting an item from a skip list needs $\log(N)$ comparisons for finding the item, plus updating the references and decreasing the maximum level, if we delete the node with maximum level.

5.3 Splay Trees

Splay trees are a self-adjusting form of binary search trees, in which the newly accessed items are moved to the root of the tree (Sleator and Tarjan, 1985). When the item is accessed second time, the second access is cheaper. There are three methods for moving an item to the root of the tree:

1. Simple Splay Trees
2. Bottom-up Splay Trees
3. Top-down Splay Trees

The easiest method for moving an item to the root of the tree is called the *simple self-adjusting strategy*, or *rotate-to-root strategy.* In this method the newly accessed item is continually rotated with its parent until it becomes the root of the tree (Sleator and Tarjan, 1985). The tree is rearranged after each access. In this method, when some items are moved to the root, some other items will be moved far from the root. Therefore, if the access order of the items do not follow the 90–10 rule, it is possible for some bad accesses to occur. As a result, simple splay tree operations will not have a logarithmic amortized time bound (Sleator and Tarjan, 1985). We do not analyze the performance of simple splay trees.

5.3.1 Bottom-up Splay Trees

The *bottom-up splay* strategy is similar to the simple rotate-to-root strategy, but it achieves a $O(\log N)$ amortized time bound. It also moves the accessed item to the root of the tree, but differently. This kind of move to root operation is known as splaying (Sleator and Tarjan, 1985). Bottom-up splay strategy considers the *item* to be rotated, its *parent*, and its *grandparent* (if exists). Single or double rotations are used during the splaying, depending on the position of the item to be splayed. A single rotation is required if the item to be splayed has only a parent but no grandparent. This kind of rotation is called *Zig* rotation (Sleator and Tarjan, 1985).

A double rotation is required if the node to be accessed has both, parent and grandparent. Double rotations can be a *Zig-zig* or *Zig-zag*. A *Zig-zig* rotation is

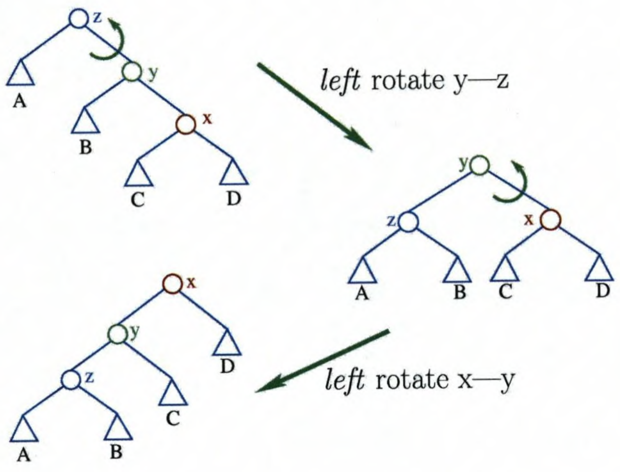


Figure 5.2: A left-left *Zig-zig*.

required if the item to be accessed and its parent, are both left children or both right children. In Figure 5.2 a *Zig-zig* of x is accomplished with two rotations: the first rotation is performed between its parent, y, and grandparent, z, and the second rotation is performed between x and its parent, y (Sleator and Tarjan, 1985).

A *Zig-zag* rotation is required if the item is a left child and its parent is a right child or vice versa. A *Zig-zag* is also performed in two rotations—one left and one right rotation or vice versa. The first rotation is between the item, x, and its parent, y, and the second rotation is between the item, x and its grandparent, z (Sleator and Tarjan, 1985).

These steps are repeated until the accessed item becomes the root of the tree.

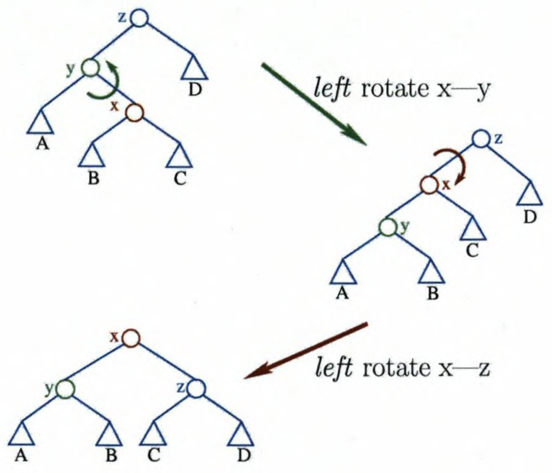


Figure 5.3: A right-left *Zig-zag.*

5.3.1.1 Analysis of Bottom-up Splay Trees

Bottom-up splay trees can be used for search and update operations. The performance of these operations depends on the arrangement of data items and the resultant tree structure. After each operation the structure of the tree is altered. Theoretical analysis can help us to understand the behavior of bottom-up splay trees. We analyze the following aspects of bottom-up splay trees:

- The insertion order of the data items,
- The effect of different operations on the structure and performance of the tree, and
- The cost of different operations.

5.3.1.2 Insertion Order of the Data Items

The order in which the data items are inserted into a bottom-up splay tree, affects the structure and performance of the bottom-up splay tree. Insertion of sorted and unsorted data items affects the behavior of the bottom-up splay trees. The number of comparisons will increase or decrease due to the order in which the data items are inserted. We first deal with in-order insertion.

Insertion of Sorted Data Items Let us identify N items with their keys, $\{k_i\}_1^N$. Suppose that items are ordered as follows $k_1 \leq k_2 \leq \ldots \leq k_{N-1} \leq k_N$, and they are entered in this order from smallest to largest. Entering k_1 into the splay tree puts it at the root with no comparison or rotation. Next entering k_2, puts it at the root with one comparison of k_2 and k_1, insertion of k_2 at the right child of k_1 and finally a rotation around k_2—k_1 which puts k_2 at the root and leaves k_1 as its left child. Inserting k_3, puts it at the root with one comparison of k_3 and k_2, insertion of k_3 at the right child of k_2 and finally a rotation around k_3—k_2 which puts k_3 at the root and leaves k_2 as its left child. Inserting k_4, puts it at the root with one comparison of k_4 and k_3, insertion of k_4 at the right child of k_3 and finally a rotation around k_4—k_3 which puts k_4 at the root and leaves k_3 as its left child. Inserting k_{N-1}, puts it at the root with one comparison of k_{N-1} and k_{N-2}, insertion of k_{N-1} at the right child of k_{N-2} and finally a rotation around k_{N-1}—k_{N-2} which puts k_{N-1} at the root and leaves k_{N-2} as its left child. Inserting k_N, puts it at the root with one comparison of k_N and k_{N-1}, insertion of k_N at the right child of k_{N-1} and finally a rotation around k_N—k_{N-1} which puts k_N at the root and leaves k_{N-1} as its left child.

Therefore, the total insertion cost for inserting N items into a bottom-up splay tree is $N-1$ comparisons and $N-1$ rotations. All the rotations are *Zig*. Ascending sorted items require left rotations, and descending sorted items require right rotations. The average insertion cost is $\frac{N-1}{N} \approx 1$ comparisons and rotations. The best, average and worst insertion cost are almost identical to each other. Insertion of sorted data into a bottom-up splay tree require less comparisons, when we compare it with standard *BST*, balanced *BST* such as AVL trees and red-black trees, and skip lists. This is due to:

- A single splay rotation after inserting each item, and
- the sorted order of the data items.

The single rotation after each insertion, keeps the insertion point for the next item close to the root of the tree. The sorted order of data items keeps the insertion of next item close to the previous item.

After inserting N sorted items into a bottom-up splay tree, the resultant tree becomes a linked list. This is exactly the same when sorted items are inserted into a standard *BST*. This causes the search operation to be expensive. In the worst case the search operation will need N comparisons. We will analyze this situation when discussing search operation of the bottom-up splay tree.

Results:

1. Inserting N sorted items into a bottom-up splay tree require one comparison and one rotation on average. The resultant tree behaves like a linked list.

2. Insertion of sorted data items into a bottom-up splay turn out to a degenerate trees as happens with standard BST. Insertion of sorted data items into a bottom-up splay tree needs $N-1$ rotations, where insertion of sorted data items into a standard BST does not need rotations. On the other hand, insertion of sorted data items into a bottom-up splay tree needs $N-1$ comparisons, where insertion of sorted data items into a standard BST needs $\frac{N(N+1)}{2}-1$ comparisons. On average, insertion of sorted data items into a bottom-up splay tree needs 1 comparison, where insertion of sorted data items into a standard BST need $\frac{N+1}{2}$ comparisons.

3. Insertion of sorted data items into an AVL tree yields a full BST. The number of required rotations is $N-\log(N+1)$, which is less than inserting the same number of items into a bottom-up splay tree. But the number of comparisons for inserting N items into an AVL tree is more than inserting the same number of items into a bottom-up splay tree.

4. If we repeatedly *insert* N sorted items into a bottom-up splay tree, the number of comparisons is $N-1$. But if we access any element during insertion, the number of comparisons increases.

5. For inserting N sorted items into a bottom-up splay tree the best-case, average-case and worst-cases are almost identical. The best case is when the first item is inserted in the bottom-up splay tree where we do not need comparisons and rotations. Insertion of each other element requires one comparison and one rotation.

6. If we need to insert sorted items and display them in reverse order we can use bottom-up splay trees.

Insertion of Unsorted Data Items When unsorted data items are inserted into a bottom-up splay tree, the required number of comparisons and rotations are more than inserting the same data items in ascending or descending orders. The total number of comparisons and rotations depends on the order of the data items. That means, different orders of data items need different numbers of comparisons and rotations. The resultant tree structure also depends on the order of the inserted data. Different orders of the data items will generate different structures for the tree. On the one hand inserting items in sorted order is very fast and need less comparisons, but it yields a degenerate tree. On the other hand inserting data in unsorted order is slower, but yields better balanced trees.

We analyze the effect of different unsorted orders on the required number of comparisons, and the resultant shape of the tree.

Suppose that we insert three items with keys 10, 20, and 30 into a bottom-up splay tree. There are 6 different orders for entering these three items into a bottom-up splay tree.

Table 5.16 shows the number of comparisons and rotations for inserting the data items in each order. As we can see the sorted insertion require less comparisons and rotations. Sorted data items always require a single rotation, where unsorted insertion require single, or double, or both types of rotation.

Table 5.16: Number of comparisons and rotations.

Order of Items	Total Comparisons	Zig Rotations	Zig-zig Rotations	Zig-zag Rotations	Total Rotations
10, 20, 30	2	2	0	0	2
10, 30, 20	3	1	0	1	3
20, 10, 30	3	1	1	0	3
20, 30, 10	3	1	1	0	3
30, 10, 20	3	1	0	1	3
30, 20, 10	2	2	0	0	2

Three items have 6 different orders. The two orders, in which 20 comes at the end, yields a full *BST*. These two orders need a *Zig-zag* rotation, where the other 4 orders do not require any *Zig-zag* rotations. An interesting point about *Zig-zag* rotations is that they tend to improve the balance of the tree.

Table 5.17 shows our results for inserting three items into a bottom-up splay tree, and the resultant tree shape.

Now let us see the insertion of seven items with the keys 10, 20, 30, 40, 50, 60 and 70, into a bottom-up splay tree. As with the insertion of three items into a bottom-up splay tree, the total number of comparisons and rotations depends

Table 5.17: The Resultant Tree Shape.

Order of Items	Resultant Tree Shape
10, 20, 30	Completely Unbalanced (Only Left Child)
10, 30, 20	Full *BST*
20, 10, 30	Completely Unbalanced (Only Left Child)
20, 30, 10	Completely Unbalanced (Only Right Child)
30, 10, 20	Full *BST*
30, 20, 10	Completely Unbalanced (Only Right Child)

on the order of the data items. However, all the unsorted orders, require more comparisons than sorted orders. The resultant tree structure also depends on the order of the inserted data. Different orders of the data items will generate different structures for the tree. As with three items, when 7 items are inserted in the sorted order, the number of comparisons is less than when the same items are inserted in unsorted orders. But, sorted insertion yields a degenerate tree, whereas insertion in unsorted order yields better balanced trees.

Table 5.18 shows the number of comparisons and rotations for different orders of the data items.

Table 5.18: Number of comparisons and rotations.

Order of Items	Total Comparisons	Zig Rotation	Zig-zig Rotation	Zig-zag Rotation	Total Rotation
10, 20, 30, 40, 50, 60, 70	6	6	0	0	6
40, 20, 60, 10, 30, 50, 70	15	3	3	3	15
40, 60, 20, 50, 70, 10, 30	17	3	5	2	17
40, 20, 10, 30, 60, 50, 70	11	3	2	2	11
70, 50, 30, 10, 60, 20, 40	14	4	1	4	14
40, 30, 20, 10, 50, 60, 70	9	5	2	0	9
40, 70, 10, 60, 20, 50, 30	15	5	1	4	15
10, 30, 50, 70, 20, 40, 60	13	5	1	3	13
10, 30, 20, 50, 70, 60, 40	11	3	1	3	11

Results

1. Our analysis shows that, when unsorted data is inserted into a bottom-up splay tree, the number of comparisons is equal to the number of rotations. The same holds when sorted data items are inserted into a bottom-up splay tree.

2. Insertion of sorted data items requires less comparisons and rotations—on average one comparison and one rotation—but the resultant tree is degenerate. On the other hand, insertion of unsorted data items require more comparisons and rotations but yields trees that are better balanced.

3. The average number of comparisons and rotations for inserting 3 items into a bottom-up splay tree is 0.8888 comparisons and rotations. Where the average number for inserting 7 items into a bottom-up splay tree is 1.7619 comparisons and rotations. The average number of comparisons and rotations increases when the number of items increase.

4. Sorted insertion needs only *Zig* rotations. Unsorted insertion may require *Zig*, *Zig-zig*, *Zig-zag* rotations, or all three types of rotations.

5. Our analysis shows that the *Zig-zag* rotations tend to improve the balance of the tree. For example if we insert the items in the following order: $70, 50, 30, 10, 60, 20,$ and 40 into a bottom-up splay tree the resultant tree will be a full balanced *BST*. The insertion process needs 4 *Zig*, 1 *Zig-zig* and 4 *Zig-zag* rotations. The 4 *Zig-zag* rotations tend to balance the tree. But, if we insert the same 7 items in the following order: $40, 50, 30, 60, 20, 70,$ and 10, which require 3 *Zig*, 5 *Zig-zig*, and no *Zig-zag* rotations, the resultant tree will be completely unbalanced, looking like a linked list. Therefore, the order which needs more *Zig-zag* rotations tends to balanced trees. On the other hand, if we insert the first order into an *AVL* tree, the insertion process requires 4 rotations and the resultant tree will not be a full balanced *BST*. Insertion of the second order into an *AVL* tree requires 2 rotation and yields a full *BST*. Therefore, the order which needs more *Zig-zag* rotations yields balanced trees.

6. Our analysis has shown that the number of required *Zig-zag* rotations increase if the item(s) which represent the root of the balanced *BST* come at the end of the orders. For example, with the three items such as 10, 20, and 30, if 20 is entered at the end, the tree will be balanced and require 1 *Zig-zag* rotation. All other 4 orders do not require *Zig-zag* rotations. In the same way if 4 items such as 10, 20, 30, and 40 were inserted into a bottom-up splay tree, the resultant tree will be balanced if 20 or 30 were inserted at the end.

Similar observations hold when 7 items are inserted into a bottom-up splay tree.

5.3.2 Effect of Operations on the Structure of a Bottom-up Splay Tree

Accesses to the items of a bottom-up splay tree change the structure of the tree. Each accessed item is splayed to the root of the tree. A number of rotations is required in order to splay the accessed item to the root of the tree. These rotations alter the structure of the tree.

5.3.2.1 Access to the Items

Accessing an item splays that item to the root of the tree. The structure of the tree is changed depending on the number of items accessed, their order of access, and the initial structure of the tree before any accesses. We analyze the effect of sequentially accessing all the items from the root to the leaf as well as from the leaf to the root of the tree.

5.3.2.2 Sequential Access to All Items from the Root to the Leaf

If all items of a degenerate bottom-up splay tree are accessed in sequential order from the root to the leaf of the tree, the structure of the tree is changed from a left degenerate tree to the right degenerate tree or vice versa. If the tree was a left-child-only tree, then the sequential access to all items will degenerate the tree to a right-child-only tree. But if the tree was a right-child-only tree, than the sequential access to all items will degenerate the tree to a left-child-only tree. The order of the items also change from ascending to descending order or vice versa.

Our analysis shows that, accessing all elements sequentially from the root to the leaf requires $2N - 1$ comparisons and $N - 1$ rotations. Therefore, the average number of comparisons is

$$\frac{2N-1}{N} \approx 2$$

and the average number of rotations is

$$\frac{N-1}{N} \approx 1.$$

All the rotations are *Zigs*. In the best case there is one comparison and no rotation. The worst case needs 2 comparisons and 1 rotation. On the average, there is

$\approx$ 2 comparison and 1 rotation. Here we can see that the average number of comparisons is 2 which is less than the average number of comparisons for the balanced *BST*. A degenerate bottom-up splay tree has a structure similar to a linked list, but accessing all items in a linked list requires $\frac{N+1}{2}$ comparisons on average, where accessing all the items in a degenerate bottom-up splay tree requires 2 comparisons on average. Our analysis shows that, bottom-up splay trees work very well with degenerate trees, if all the items are accessed from the root to the leaf. The total search cost is $2N-1$ comparisons and $N-1$ rotations.

5.3.2.3 Sequential Access to All Items from the Leaf to the Root

If all items of a degenerate bottom-up splay tree are accessed in sequential order from the leaf to the root of the tree, the structure of the tree is not changed. Of course, accessing each item will change the structure of the tree, but after accessing

Table 5.19: Comparisons and rotations for sequentially accessing all the elements.

Number of Items	Total Comparisons	Zig Rotation	Zig-zig Rotation	Zig-zag Rotation	Total Rotation
4	12	2	2	1	8
5	15	2	2	2	10
6	20	4	2	3	14
7	23	4	2	4	16
8	30	6	2	6	22
9	33	6	2	7	24
10	38	8	2	8	28
11	41	8	2	9	30
12	45	9	3	9	33
13	55	6	6	12	42
14	60	8	6	13	46
15	63	8	6	14	48
16	70	10	6	16	54

all the items we will have the same tree as before the access. Accessing all the items from the leaf to the root of the tree requires $N+R$ comparisons, where N is the number of items and R is the number of rotations. The total number of rotations is $C-N$ where C is the number of comparisons and N is the number of items.

Table 5.19 shows the number of comparisons and rotations for sequentially accessing all the elements from the leaf to the root.

5.3.2.4 Random Access to All Items of a Degenerate Bottom-up Splay Tree

The result of random access, to all items of a degenerate bottom-up splay tree differs from sequential access. But, random access to each item of a degenerate tree changes the shape of the tree to a better balanced *BST*. Some specific random orderings of access, may change a degenerate tree to a fully balanced *BST*.

Accessing all the items in random order requires $N + R$ comparisons, where N is the number of items and R is the number of rotations. The total number of rotations is $C - N$ where C is the number of comparisons and N is the number of items.

Suppose that a degenerate bottom-up splay tree contains seven items with the keys 10, 20, 30, 40, 50, 60 and 70. Our analysis shows that, after accessing all the items in random order, the tree shape is changed from a degenerate to some better shaped *BST*. Some orderings change the tree to a full and balanced *BST*.

Table 5.20 shows the result of some random access to all items of a degenerate bottom-up splay tree.

Table 5.20: Random access to all items of a degenerate bottom-up splay tree.

Access Order	Total Comparisons	Total Rotation	Resultant Tree
40, 70, 30, 60, 20, 50, 10	26	19	Not Linked List
10, 30, 50, 70, 20, 40, 60	34	27	*BST*
20, 40, 60, 10, 30, 50, 70	32	25	Not Linked List
10, 70, 20, 60, 30, 50, 40	28	21	Not Linked List
70, 50, 30, 10, 20, 60, 40	21	14	*BST*
10, 50, 70, 30, 20, 60, 40	31	24	BST
10, 70, 50, 30, 20, 60, 40	31	24	BST
50, 10, 70, 30, 60, 20, 40	29	22	Full *BST*
30, 70, 10, 50, 20, 60, 40	29	22	Full *BST*
30, 10, 50, 70, 20, 60, 40	32	25	Full *BST*
50, 70, 30, 10, 60, 20, 40	26	19	Full *BST*
10, 30, 50, 70, 20, 60, 40	34	27	Full *BST*
70, 50, 30, 10, 60, 20, 40	25	18	Full *BST*
50, 10, 30, 70, 60, 20, 40	27	20	Full *BST*
50, 30, 10, 70, 60, 20, 40	25	18	Full *BST*

The results of repeatedly inserting items in sorted order into a bottom-up splay tree are similar to those of a simple rotate-to-root splay tree. But accessing some nodes of a bottom-up splay tree improves the balance of the tree. This is one of the cases which illustrates that splay trees are not good for uniform operations.

If some random accesses are executed after inserting some nodes in sorted order the balance of the tree may improve. Suppose that items 10, 20, and 30 were inserted into a bottom-up splay tree in ascending or descending order. The resultant tree will be left-child-only if the items were inserted in ascending order, or it will be right-child-only if the items were inserted in descending order. Some random access orders will change the tree from linked list to a full balanced *BST*.

Table 5.21 shows the access order and the shape of the resultant tree.

Table 5.21: Access order and the shape of the resultant tree.

Access Order	Resulting Tree
10, 20, 30	Only left child tree (like a linked list)
20, 10, 30	Only left child tree (like a linked list)
30, 20, 10	Only right child tree (like a linked list)
20, 30, 10	Only right child tree (like a linked list)
10, 30, 20	Full *BST*
30, 10, 20	Full *BST*

Results:

- Two access orders with 20 at the end, generate a full *BST*. For these three items 20 is the root of the *BST*. It means if the leaf items are accessed first and the root is accessed last the resultant tree will be a full *BST*.
- If the access orders are left, root, right or root, left, right then the resultant tree will be a left-child-only tree.
- If the access orders are right, root, left or root, right, left then the resultant tree will be right-child-only tree.

Our elaboration shows that the above result is also true for more than 3 items. Suppose that the items 10, 20, 30, 40, 50, 60, and 70 were inserted into a bottom-up splay tree in ascending or descending order. The result will be a left-child-only tree if the items were inserted in ascending order, or it will be only right child if the items were inserted in descending order. After accessing all the elements the shape of the tree will change depending on the access order.

Table 5.22 shows the access order and the shape of the resultant tree.

Table 5.22: Access order and the resultant tree.

Access Order	Resulting Tree
10, 30, 50, 70, 20, 60, 40	Full *BST*
10, 50, 30, 70, 20, 60, 40	Full *BST*
10, 70, 30, 50, 20, 60, 40	Full *BST*
10, 30, 70, 50, 20, 60, 40	Full *BST*
30, 70, 10, 50, 20, 60, 40	Full *BST*
30, 10, 50, 70, 20, 60, 40	Full *BST*
50, 10, 70, 30, 60, 20, 40	Full *BST*
50, 70, 30, 10, 60, 20, 40	Full *BST*
50, 70, 10, 30, 60, 20, 40	Full *BST*
50, 30, 70, 10, 60, 20, 40	Full *BST*
50, 30, 10, 70, 60, 20, 40	Full *BST*
50, 10, 30, 70, 60, 20, 40	Full *BST*
50, 10, 30, 70, 20, 60, 40	Full *BST*
70, 50, 30, 10, 60, 20, 40	Full *BST*
10, 20, 30, 40, 50, 60, 70	Only left child tree
70, 60, 50, 40, 30, 20, 10	Only left child tree
10, 30, 20, 50, 70, 60, 40	Right balanced, left like linked list
10, 20, 50, 70, 30, 60, 40	Right balanced, left like linked list
70, 50, 30, 10, 20, 60, 40	Right balanced, left like linked list
10, 50, 70, 30, 20, 60, 40	Right balanced, left like linked list
10, 70, 50, 30, 20, 60, 40	Right balanced left like linked list
20, 40, 60, 10, 30, 50, 70	Near to linked list
40, 70, 30, 60, 20, 50, 10	Near to linked list

Results:

- If all items are accessed from the root to the leaf, the resultant tree will be a right-child-only tree like a linked list. In the same way, if all the items are accessed from the leaf to the root, the resultant tree will be a left-child-only tree like a linked list.
- If the leaf items are accessed first and then the items which are in the upper levels, and finally we access the root, the resultant tree will be a full *BST* or close to a full balanced *BST*.
- If the upper level items are accessed before the bottom level items, then the resultant tree will be close to the linked list shape.

Random Access to All Items of a Full Balanced Bottom-up Splay Tree
Random access to all items of a full balanced bottom-up splay tree changes the shape of the tree. Some access orders yield degenerate trees, and some other orders yield unbalanced trees. There are some orders which keep the tree, after accessing

all of its elements, full and balanced. That means, if all items of a full and balanced bottom-up splay tree are accessed, the resultant tree is again a full and balanced bottom-up splay tree.

An interesting observation is that, the access orders which yield a full and balanced bottom-up splay tree, also yield a full and balanced bottom-up splay tree, when all items of a degenerate bottom-up splay tree are accessed in the same order. Suppose that a bottom-up splay tree contains 7 items with keys 10, 20, 30, 40, 50, 60, and 70. Table 5.23 shows the access orders which yields a full balanced bottom-up splay tree when all the items are accessed in a full balanced bottom-up splay tree and degenerated bottom-up splay tree. Of course there are other orders

Table 5.23: Access orders and the resultant tree.

Access order	Total Comparison	Total Rotation	The Resultant Tree Shape
50, 10, 70, 30, 60, 20, 40	31	24	Full *BST*
30, 70, 10, 50, 20, 60, 40	31	24	Full *BST*
30, 10, 50, 70, 20, 60, 40	30	23	Full *BST*
50, 70, 30, 10, 60, 20, 40	30	23	Full *BST*
10, 30, 50, 70, 20, 60, 40	29	22	Full *BST*
70, 50, 30, 10, 60, 20, 40	29	22	Full *BST*
50, 10, 30, 70, 60, 20, 40	28	21	Full *BST*
50, 30, 10, 70, 60, 20, 40	29	22	Full *BST*

which also yield full *BST*s. But all of these orders have the root elements 20, 60, 40 at the end of the order.

Accessing all the items in random order from a full and balanced bottom-up splay tree requires $N + R$ comparisons, where N is the number of items and R is the number of rotations. The total number of rotations is $C - N$ where C is the number of comparisons and N is the number of items.

5.3.3 Bottom-up Splay Tree Deletion

To delete an item from a bottom-up splay tree, first we access the item. The process of access moves the accessed item to the root of the tree. Now deleting the item from the root of the tree, we get two sub-trees: L and R. At this stage we find the largest item in L and splay that item to the root of L, and then make R the right sub-tree of L's root. This is called the *join* operation, which joins the left and right sub-trees (Sleator and Tarjan, 1985).

The cost of deletion in a bottom-up splay tree is two splays. The first splay puts the item to be deleted at the root of the tree. The second splay is needed to splay the largest item in the left sub-tree to its root. For the deletion of some items we will not need the second pass splay rotation, if the largest item in the left sub-tree is already at the root of the left sub-tree. For some items we will not even need the first pass splay rotation. This depends on the structure of the tree, and the position of the item in the tree.

The number of comparisons for finding the item to be deleted, and finding the largest item in the left sub-tree, depends on the number of items and the structure of the tree. Analysis is helpful to understand the required number of comparisons for deleting an item from a bottom-up splay tree.

Suppose that we have a bottom-up splay tree with three items such as 10, 20, and 30. The number of comparisons and rotations for deleting an item depends on the structure of the tree, and on the position of the item to be deleted. Table 5.24 shows the number of comparisons and rotations for deleting an item from a bottom-up splay tree. In this analysis, the original tree is considered for deletion of each item.

Table 5.24: Comparisons and deletions when deleting in a bottom-up splay tree.

Tree Shape	Total Comparison	Total Rotation	Average Comparison	Average Rotation
Balanced	7	2	2.33	0.67
Only right children	8	3	2.67	1.00
Only left children	8	3	2.67	1.00
One right one left child	9	4	3.00	1.33
One left one right child	9	4	3.00	1.33

Results:

- The above table shows that, in a bottom-up splay tree the average number of comparisons and rotations for deleting the items increases if the tree is shaped like a linked list or near to a linked list.

- A bottom-up splay tree with 3 items, does not require rotations in the second pass, if the tree is balanced or has only left children or only right children. But the second pass of splay needs rotation if the tree is shaped with one left and one right child or one right and one left child.

Now let us consider the deletion of items in a bottom-up splay tree, which contains 7 items with the keys 10, 20, 30, 40, 50, 60 and 70. In our analysis we consider the different shapes of the bottom-up splay tree with the above 7 items. In each shape, when finding the average number of comparisons and rotations we consider the original tree, for the deletion of each item. The order in which the items are deleted is ascending order for each shape.

Table 5.25 shows the number of comparisons and rotations for the different shapes of the bottom-up splay tree with 7 items.

Table 5.25: The number of comparisons and rotations.

Tree Shape	Total Comparisons	Total Rotations	Average Comparisons	Average Rotations
Full Balanced *BST*	24	12	3.43	1.71
Linked List	31	21	4.43	3.00
BST but, not balanced	27	14	3.86	2.00
Balanced *BST*	25	12	3.57	1.71
BST but, not balanced	26	13	3.71	1.86
Near to linked list	29	16	4.14	2.29

Results:

- In a bottom-up splay tree with 7 items the average number of comparisons and rotations will increase if the tree is shaped like a linked list or near to a linked list. This is the same result as it was with 3 items.

- In a balanced bottom-up splay tree with 7 items, the average number of comparisons is 3.4285 and the average number of rotations is 1.7142. In a linked list shaped bottom-up splay tree with the same 7 items, the average number of comparisons is 4.4285, and the average number of rotations is 3.0000.

- Deletion from a left-child-only degenerate bottom-up splay tree with any number of items, does not require a rotation in the second pass. However, deletion from a right-child-only degenerate bottom-up splay tree, requires a number of *Zig* and *Zig-zig* rotations. But the number of rotations in the second pass is always less then the number of rotations in the first pass.

5.3.4 Top-down Splay Trees

Bottom-up splay trees use two passes for splaying the item to the root of the tree. The first pass is required to access the item, and the second pass is used to splay the item to the root of the tree. Therefore, bottom-up splay trees need more comparisons and rotations for accessing and splaying the items to the root of the tree.

Top-down splay trees on the other hand, use a single pass for accessing and splaying the items to the root of the tree. In this way, top-down splay trees require less comparisons and rotations for accessing the items. Top-down splay trees are fast and maintain the amortized time bound (Sleator and Tarjan, 1985).

In a top-down splay, the tree is split into three sub-trees: a left sub-tree, a middle sub-tree, and a right sub-tree. During the search for an item, we take the items which are on the access path and move them to the left sub-tree or right sub-tree depending on whether they are smaller or larger than the search item. Initially the left and right sub-trees are empty and the middle tree consists of the entire tree (Sleator and Tarjan, 1985). At any point during the search for `x`, we have to follow the left or right link. When we follow the left link, then `x` and its right sub-tree are larger than the item which will become the root. Therefore, we put `x` and its right sub-tree in a separate tree, which we call it right sub-tree (R). When we follow the right link then `x` and its left sub-tree are smaller than the item which will become the root. Therefore, we put `x` and its left sub-tree in another sub-tree, which we call the left sub-tree (L). Descending the tree two levels at a time, we encounter three items, `x`, `y`, and `z`. `x` is the root of the middle tree, `y` is a child of `x`, and `z` is the grandchild of `x`. With these three items we consider three different cases: the *Zig* case, the *Zig-zig* case, and the *Zig-zag* case.

In the *Zig* case, `y` becomes the root of the middle tree, `x` and its sub-tree are attached to sub-tree L or sub-tree R, depending whether `x` is smaller or larger then `y`, respectively. If the case is *Zig-zig*, `z` becomes the root of the middle tree, and we rotate `y` around `x` and attach it as a right child of the largest value of sub-tree L or as a left child of the smallest value of the sub-tree R. In a *Zig-zag* case, `z` is moved to the root of the middle tree, i.e. sub-tree M, the sub-trees `x` and `y` are moved to sub-trees R and L, respectively. To make the code simpler, the *Zig-zag* case is changed to a *Zig* case. That is, instead of making `z` the root of the middle tree, `y` is moved to the root of the middle tree and `z` remain as a sub-tree of `y`.

This simplification makes the *Zig-zag* case similar to the *Zig* case. It avoids the rotation process but causes more iterations in the splay process. When the value to be accessed is at the root of the middle tree, we make the left child of `x` as a right child of the maximum item in the sub-tree L, and make the right child of `x` a left child of the minimum item in the sub-tree R. Finally we make L and R the left and right children of `x`, respectively.

5.3.5 Analysis of Top-down Splay Trees

As with the bottom-up splay trees, we analyze the search and update operations of top-down splay trees to understand the structure and performance as well as the effect of different operations on the structure and performance of the top-down splay trees. In our analysis we consider the effects of:

- The order in which data items are inserted into a top-down splay tree,
- The effect of various operations on the structure of the tree, and
- The cost of operations.

5.3.5.1 The Order of the Data Items

The order in which the data items are inserted into a top-down splay tree, affects the structure and performance of the top-down splay tree. Sorted and unsorted insertion of data items changes the structure and performance of the top-down splay trees. The number of comparisons will increase or decrease due to the order in which the data items are inserted. Our observations for inserting sorted and unsorted data items into a top-down splay tree follows.

Insertion of Sorted Data Items When data items are inserted in sorted order into a top-down splay tree, one comparison is required for insertion of each item, except the first item, which does not need comparison. There is no need for rotation at all. When the new item is inserted into the middle tree, the previously inserted item with its sub-tree is attached to the left or right of the newly inserted item, depending on whether the new item is larger or smaller than the previous item, respectively. In other words, if the data items are inserted in ascending order, the previously inserted item with only its left sub-tree is attached to the left of the new item, and if the data items are inserted in descending order, the previously inserted item with only its right sub-tree is attached to the right of the new item.

Insertion of N sorted items into a top-down splay tree in ascending or descending order, requires $N - 1$ comparisons. Unlike the bottom-up splay tree, insertion of sorted data items into a top-down splay tree does not need rotations. Without rotation, the new item is moved to the root of the middle tree and the previous item with its sub-tree is attached to it. The difference between ascending and descending sorted data items is the resultant tree shape. If the data items are inserted in ascending order the resultant tree is a left-child-only tree with the larger keys close to the root of the tree. When the data items are inserted in descending order the resultant tree is a right-child-only tree with the smaller keys close to the root.

Suppose that we want to insert N items with their keys $\{k_i\}_1^N$ into a top-down splay tree. Let the items be entered in the following order $k_1 \leq k_2 \leq \ldots \leq k_{N-1} \leq k_N$ into a top-down splay tree. Entering k_1 into the splay tree puts it at the root with no comparisons or rotations. When we insert k_2, after one comparison we make k_1 (the previous item), the left child of k_2. In the same way when k_3 is entered, after one comparison k_2 is attached to the left of k_3. Finally when entering k_N, k_{N-1} is attached to the left of k_N. Therefore, insertion of each item requires one comparison and one attachment to its left. The total number of comparisons for inserting N sorted items into a top-down splay tree is $N - 1$. Therefore, the average number of comparisons is $\frac{N-1}{N} \approx 1$. The same method is applied when the items are ordered as $k_1 \geq k_2 \geq \ldots \geq k_{N-1} \geq k_N$. The only difference is that k_1 is attached to the right of k_2, k_2 is attached to the right of k_3, and k_{N-1} is attached to the right of k_N.

When *sorted* data items are inserted into a top-down splay tree the best, average, and worst insertion cost are *identical* to one another. We can also see that when inserting sorted data items into a top-down splay tree, the average number of comparisons is the same as it is for the bottom-up splay tree. But there is no need for rotations at all, whereas insertion of N sorted data items into a bottom-up splay tree needs $N-1$ rotations. This is due to the top-down splay tree algorithm in which, before inserting the new item, we turn the previous item and its sub-tree into a left or right child of the new item. Instead of rotation, the new item is put in the root of the middle tree and the previous item with its sub-tree is attached to it.

After inserting N sorted data items into a top-down splay tree, the resultant tree will be completely unbalanced, like a linked list. The resultant tree will be a left-child-only tree, if the items were inserted in ascending order, or it will be a right-child-only tree if the items were inserted in descending order. This causes the search operation for some items to be expensive. This is exactly the same as when the sorted items are inserted into a bottom-up splay tree or into a simple *BST*. But this differs from balanced *BST*s such as *AVL* trees and red-black trees. Inserting sorted data items into an *AVL* tree yields a balanced *BST*. However, for insertion of sorted data items, a top-down splay tree does not require rotations and puts the newly inserted item into the root of the tree. The number of required comparisons is also less than for inserting the same number of data items into an *AVL* tree.

Our analysis for inserting ascending or descending sorted data items, into a top-down splay tree, yields the following results.

Results:

- Insertion of N ascending or descending sorted data items, into a top-down splay tree, requires $N - 1$ comparisons. This is the same as inserting sorted data items into a bottom-up splay tree. This is despite that the bottom-up splay tree uses two passes, where top-down splay trees use a single pass, for inserting data items into the tree.

- When sorted data items are inserted into a top-down splay tree, the resultant tree shape is a left-child-only or a right-child-only tree. This is also the same as with the bottom-up splay tree and a simple *BST*.

- Insertion of sorted data items into a top-down splay tree does not require rotations, whereas insertion of sorted data items into a bottom-up splay tree requires $N - 1$ rotations.

- For inserting N sorted items, into a top-down splay tree, the average number of comparisons is one. The worst case also needs one comparison. The best case needs zero comparisons. Therefore, for inserting the sorted data items into a top-down splay tree, the best, average, and worst cases are identical. The best case is when the first item is inserted into the top-down splay tree, where we do not need comparisons. Insertion of each of the other items require one comparison.

5.3.5.2 Insertion of Unsorted Data Items

When *unsorted* data items are inserted into a top-down splay tree, the required number of comparisons is the same as when these unsorted items are inserted into a bottom-up splay tree. But, the number of required rotations for inserting unsorted data items into a top-down splay tree is less than inserting the same number of items into a bottom-up splay tree. Different orders of data items yield different numbers of comparisons and rotations. The resultant tree structure also depends on the order of the inserted data, but in most cases the insertion of unsorted data items into a top-down splay tree yields a degenerate tree.

Suppose that we insert three items with keys 10, 20, and 30 into a top-down splay tree. There are 6 different orders for entering these three items into a top-down splay tree.

Table 5.26 shows the number of comparisons and rotation for each order of the data items.

Table 5.26: Number of comparisons and rotations.

Order of Items	Total Comparisons	Total Rotations	Tree Shape
$10, 20, 30$	2	0	Linked List
$10, 30, 20$	3	0	Full *BST*
$20, 10, 30$	3	1	Linked List
$20, 30, 10$	3	1	Linked List
$30, 10, 20$	3	0	Full *BST*
$30, 20, 10$	2	0	Linked List

Now let us consider the insertion of seven items with the keys 10, 20, 30, 40, 50, 60, and 70, into a top-down splay tree. As with the insertion of three items, the total number of comparisons and rotations depends on the order of the data items. The resultant tree structure also depends on the order of the data items. However, comparing with bottom-up splay tree, for most of the orders, the resultant tree is an unbalanced or degenerate tree.

Table 5.27 shows the number of comparisons, rotations, and the resultant tree, for different orders of the data items.

Table 5.27: Sequential access from the root to the leaf.

Order of Items	Total Comparisons	Total Rotations	Tree Shape
10, 20, 30, 40, 50, 60, 70	6	0	Linked List
40, 20, 60, 10, 30, 50, 70	14	1	Linked List
40, 60, 20, 50, 70, 10, 30	17	6	*BST*
40, 20, 10, 30, 60, 50, 70	11	3	Linked List
70, 50, 30, 10, 60, 20, 40	14	4	Unbalanced *BST*
40, 30, 20, 10, 50, 60, 70	9	2	Unbalanced *BST*
40, 70, 10, 60, 20, 50, 30	15	5	Unbalanced *BST*
10, 30, 50, 70, 20, 60, 40	13	3	Unbalanced *BST*
10, 30, 20, 50, 70, 60, 40	11	2	Unbalanced *BST*
20, 60, 70, 30, 10, 50, 40	14	3	Unbalanced *BST*
60, 20, 10, 50, 70, 30, 40	14	4	Unbalanced *BST*

Results:

1. Our observations show that, the number of comparisons is almost the same as it was for bottom-up splay tree. But, insertion of unsorted data items requires less rotations. Top-down splay trees require rotation only in the *Zig-zig* case, which is only one rotation for each *Zig-zig* case. The *Zig* and *Zig-zag* cases don't require rotation. Instead, some movement of items between left, L, middle, M, and right ,R, and sub-trees are required.

2. After inserting the data items into a top-down splay tree, in most cases the resultant tree has degenerated to a linked list or almost to a linked list. Where insertion of the same items into a bottom-up splay tree yields a better balanced tree. For example, insertions of data in the orders 70, 50, 30, 10, 60, 20, 40 and 10, 30, 50, 70, 20, 60, 40 into a *top-down* splay tree results in an unbalanced *degenerate* tree, but inserting these two sequences of items in the same order into a *bottom-up* splay yields a full, *balanced BST*.

3. We observed another interesting point about the unsorted insertion into a top-down splay tree. Using the standard *Zig-zag* notation yields better balanced *BST*s, than using the simplified *Zig-zag* notation. The literature mentions that, the simplified *Zig-zag* notation simplifies the code but increases the number of iterations, where the *Zig-zag* complicates the code but decreases iterations (Weiss, 1999). This is true but, the simplified *Zig-zag* notation yields a more degenerate tree. For example, applying the simplified *Zig-zag*

notation for inserting the orders 20, 60, 70, 30, 10, 50, 40 and 60, 20, 10, 50, 70, 30, 40 into a top-down splay tree, results in a more degenerate tree than using the standard *Zig-zag* notation.

4. The average number of comparisons for inserting three items into a top-down splay tree is 0.78. The average number of rotations is 0.11. Where the average number of comparisons for inserting 7 items is 1.78 comparisons. The average number of rotations for inserting 7 items into a top-down splay tree is 0.43.

5.3.5.3 Effect of Operations on the Structure of Top-down Splay Trees

Search and update operations change the structure of the top-down splay tree. Every access to some item brings that item to the root of the tree. Consequently, every access changes the shape of the tree. The affect of each operation may be different. We will analyze the effect of different operations on the structure of the top-down splay trees. Our analysis considers the required number of comparisons and rotations, and the resultant tree structure.

Sequential Access to All Items from the Root to the Leaf If the data items are inserted in ascending order into a top-down splay tree, the resultant tree will be a left-child-only tree. After accessing all the elements sequentially from the root to the leaf, the structure of the tree is changed from a left-child-only tree to a right-child-only tree. Our analysis reveals that, the total number of comparisons for sequentially accessing all the items is $2N - 1$, where N is the number of items in the tree. Sequential access to all elements from the root to the leaf does not require any rotations. This is due to the movement of the accessed item to the root of the middle tree, tree M. While its left sub-tree, the old root, and its right sub-tree is moved to the right sub-tree, tree R. In this case the left sub-tree, tree L, is always empty.

Table 5.28 shows our observations for sequential access to all elements from the root to the leaf.

These results in Table 5.28 show that, when the number of items is increased by one, the number of comparisons is increased by two. That means that three items require 5 comparisons, 4 items require 7 comparisons, and so on. The average number of comparisons is $\frac{2N-1}{N} \approx 2$.

Table 5.28: Sequential access from the root to the leaf.

Number of Items	Total Comparisons	Total Rotation
3	5	0
4	7	0
5	9	0
6	11	0
7	13	0
8	15	0
15	29	0
31	61	0
63	125	0

We can conclude that after inserting N items in ascending order into a top-down splay tree, and than sequentially accessing all the items from the root to the leaf require 2 comparisons on average. This is the same as for the bottom-up splay tree. In the bottom-up splay tree we also need 2 comparisons on average for accessing all the items from the root to the leaf of the tree. But there is no need for rotation, when accessing all the items in a degenerate top-down splay tree. Where we need $N-1$ rotations to access all the items in a degenerate bottom-up splay tree. Bottom-up, and top-down splay trees will be degenerated, if the items were inserted in ascending or descending orders into the trees.

Inserting the data items in descending order into a top-down splay tree, and then accessing all the items from the root to the leaf of the tree, yields a symmetric result. After accessing all the elements sequentially from the root to the leaf, the structure of the tree is changed from a right-child-only tree to a left-child-only tree. The total number of comparisons for sequentially accessing all the elements is $2N-1$, where N is the number of items in the tree; there is no need for rotation at all.

Sequential Access to All Items from the Leaf to the Root After inserting the data items in ascending or descending orders, and then accessing all the items from the leaf to the root of the tree, the resultant tree shape is not changed. This is unlike accessing all the items from the root to the leaf. The number of comparisons is increased compared with accessing all the items from the root to the leaf. In some cases there is also need for rotation.

Table 5.29 shows our analysis for sequentially accessing all the items from the leaf to the root of the tree.

Table 5.29: Sequential access from the leaf to the root.

Number of Items	Total Comparisons	Total Rotations	Average Comparison	Average Rotation
3	7	1	2.33	0.33
4	11	1	2.75	0.25
5	15	2	3.00	0.40
6	19	3	3.16	0.50
7	23	4	3.28	0.57
8	28	4	3.50	0.50

Our experimental results show that the total number of comparisons for accessing all the items from the leaf to the root of a degenerate top-down splay tree is from $2N$ to $5.32N$ comparisons, where N is the number of items and N is between 2 and 1048576. Therefore, the average number of comparisons ranges from 2 to 5.32. The total number of required rotations ranges from 0 to $1.45N$ where N ranges between 1 and 1048576.

5.3.5.4 Random Access to All Items

For random access we will not consider the original structure of the tree for all items, as it was for sequential access. But, for each item we will consider the structure which was obtainied after accessing the previous item. Our theoretical analysis show that, random access to all items of a top-down splay tree, require different number of comparisons and rotations, depending on the access order. The resultant tree structure also depends on the order in which the items are accessed.

Suppose that three items with the keys 10, 30, and 20 were inserted into a top-down splay tree. The resultant tree will be a full balanced *BST*, with 20 in the root, and 10 and 30 as a left and right child respectively.

Table 5.30 shows the result of accessing all the items in various orders.

It means, after accessing the first item the tree shape will change, and we consider this shape for accessing the second item. In the same way when access to the second item changes the shape of the tree, we consider that change for accessing the third item. The best case is when the root, 20, is in the middle of the access order. In this case we need a total of 6 comparisons and there is no need for rotation. If the root is accessed first, the same number of comparisons is required, but there is also need for rotation. The worst case is when the root is at

Table 5.30: Access in various orders.

Access Order	Number of Comparisons	Number of Rotations
10, 20, 30	6	0
10, 30, 20	7	1
20, 10, 30	6	1
20, 30, 10	6	1
30, 10, 20	7	1
30, 20, 10	6	0

the end of the order. In the worst case we need 7 comparisons and 1 rotation for each order. But if we access these three items in different orders and considering the original shape of the tree for each access, the required number of comparisons will be 5 and there is no need for rotation. As a result, in top-down splay trees the access to items degenerate the tree. If we access all the items of a balanced *BST*, when considering the original shape of the tree for accessing each item, the number of comparisons and rotations decrease. But, if we consider the shape which was changed by accessing the previous item, the number of comparisons and rotations increase. On the one hand, top-down splay trees move the recently accessed items to the root of the tree, on the other hand it changes the structure of the tree from balanced to unbalanced. Let us consider a top-down splay tree which has 7 items with the keys 10, 20, 30, 40, 50, 60, and 70. Suppose that the tree which contains these items is a full balanced *BST*.

Table 5.31 shows the result of sequentially accessing all the elements in different orders.

Table 5.31: Sequential access in different orders.

Access Order	Number of Comparisons	Number of Rotations
10, 20, 30, 40, 50, 60, 70	17	1
40, 60, 20, 10, 50, 70, 30	22	3
70, 60, 50, 40, 30, 20, 10	17	1
20, 60, 10, 50, 30, 70, 40	25	7
60, 20, 50, 10, 70, 30, 40	25	7
10, 30, 50, 70, 20, 60, 40	24	6
70, 50, 30, 10, 60, 20, 40	24	6
30, 50, 10, 70, 20, 60, 40	28	8
50, 30, 70, 10, 60, 20, 40	29	7
30, 50, 40, 20, 60, 10, 70	25	6

Table 5.31 also shows that the best case is when we access the root in the middle (in-order). In this case we need 17 comparisons and 1 rotation. The worst case is when the root is at the end of the access order. In the worst case we need up to 29 comparisons and up to 8 rotations.

5.4 Comparing Splay Trees with Skip Lists

Skip lists are also widely used for search and update operations. They are related to splay trees in terms of most of their run time behaviors. For many non-uniform sequences of operations, splay trees perform better than other search trees, even when the specific pattern of the sequence is unknown (Sleator and Tarjan, 1985). Some operations in splay trees may take longer but a sequence of operations has a logarithmic time bound (Sleator and Tarjan, 1985). If we need a sequence of operations, and the importance is the total required time for all operations and not an individual operation, then the splay tree is a better choice. In such applications, a better goal is to reduce the amortized time of the operations, which is the average time (Sleator and Tarjan, 1985).

Skip lists on the other hand are a probabilistic data structure which use randomization for balancing. The operations of insertion and deletion change the structure of skip lists but searching does not change the structure of the skip list. Skip lists remain static after searching an item, therefore, the future search operation for the same item needs the same amount of time as the first search.

How do splay trees compare with skip lists? The effect of the search and update operations on splay trees differs from these operations on skip lists. We have analyzed these two data structures by (1) manually elaborating small data sets, (2) considering the theoretical aspects and (3) comparing timed computer runs empirically, the behavior of splay trees and skip lists. We consider the behaviors of splay trees and skip lists below.

Restructuring Movement of the accessed item to the root of the tree is an important feature of splay trees which distinguishes splay trees from skip lists and other *BST*s.

Skip lists and balanced *BST*s remain static when only search operations are done. But splay trees move the accessed item to the root of the tree. This movement of the accessed item to the root of the tree improves the efficiency of future search operations to the same item (Sleator and Tarjan, 1985).

Restructuring and Tree Building Our observations show that, the insertion of sorted data items into a bottom-up or top-down splay trees is affected by the restructuring feature of the splay tree. As a result of restructuring, insertion of sorted data items into a splay tree requires less comparisons than inserting the same number of items into a skip list and balanced *BST*s. The reason is that, after inserting each item, a restructuring rule is applied which moves the newly inserted item to the root of the tree so that the insertion point for the next item is likely to be closer to the root of the tree. Inserting N sorted data items into bottom-up or top-down splay trees requires $N - 1$ comparisons. Therefore, the average number of comparison is ≈ 1. Where the average number of comparisons for inserting N items into a skip list, in the best case, is $\log N + 1$ comparison. In fact, the required number of comparisons for inserting N items, into a skip list depends on:

1. The number of levels in the skip lists,
2. The distribution of items in the levels, and
3. The order in which the data items are inserted.

Though insertion of N sorted data items into a bottom-up splay tree requires $N - 1$ rotations, insertion of items into a skip lists does not require rotation. However, insertion of data items into a skip lists needs some initialization of pointer and random number generation. Insertion of sorted data items into top-down splay tree does not need rotation.

Our experiments affirm that the required time for inserting N sorted data items into a splay tree is less than inserting the same number of items into a skip list.

Restructuring and the Search Operation Our observations show that, the restructuring feature affects the search operation of splay trees. The obvious effect is on a degenerate tree. That means, splay trees work very well, with a degenerate tree if all the items are accessed from the root to the leaf of the tree. According to our observations, the total search cost is $2N-1$ comparisons and $N-1$ rotations. The average number of comparisons for searching is 2, and the average number of required rotations is one. Splay trees move the accessed item to the root of the tree, where skip list operations do not change the position of the items during the search operation. Instead, skip lists keep sorted lists of items. The average search cost for searching an item in a skip lists with 3 items is 2.33 comparisons, and the average search cost for searching an item in a skip list with 7 items is 4.10 comparisons. The averages are taken from the cases which have high probability. The average search cost in a degenerate splay tree with 3 items, when all the items are accessed from the root to the leaf of the tree, is $\frac{2N-1}{3} = 1.67$ comparisons. In a bottom-up or top-down splay tree with 7 items, if all the items are accessed from the root to the leaf, the average search cost is $\frac{2N-1}{7} = 1.86$ comparisons—see the proof in Section 5.3.5, page 86.

The average number of comparisons for accessing all the items from the root to the leaf of a degenerate bottom-up or top-down splay tree is 2 comparisons, which is less than the average number of comparison for accessing all the items in a skip list and balanced *BST*s. In a balanced *BST*, such as *AVL* tree, the average number of comparison is close to $\log N$.

Memory Requirements Splay trees are more storage-efficient because no balance information is stored. Rather than maintaining the balance, the tree is adjusted during each operation by splaying the accessed node to the root of the tree. The resultant tree behaves, in an amortized sense, as though it is balanced (Sleator and Tarjan, 1985).

Bottom-up splay trees require two passes for splaying an item to the root of the tree. The first pass is used to find the item, and the second pass splays the item to the root of the tree. The second pass traverses the path back to the root of the tree. This can be done by maintaining parent references, or by storing the access path on a stack. Top-down splay trees on the other hand use a single pass for accessing and splaying the item to the root of the tree. Top-down splaying is faster in practice and use only constant extra space (Sleator and Tarjan, 1985).

Skip lists are also very space efficient. They can easily be configured to require less memory because they do not need balance and priority information to be stored with each node (Pugh, 1990).

Ease of Implementation Splay trees are simpler to implement than *AVL* trees because there are fewer cases in the algorithms (Sleator and Tarjan, 1985).

The skip list is an alternative to the *BST*, which has a structure similar to linked lists and properties similar to self-balancing *BST*s. For many applications, it is easier to balance a data structure probabilistically, rather than strictly maintain a balanced data structure. Therefore, skip lists are a more natural representation for many applications and have a simpler algorithm (Pugh, 1990).

Skip list algorithms have low inherent constant-factor overheads. This makes them easy to implement. On the other hand self-adjusting trees re-arrange their structure after each operation. This causes a significant overhead on the implementation of self-adjusting trees. (Pugh, 1990).

Splay trees require more rotations than *AVL* trees. The cost of a rotation in a search tree, which we assume to be $O(1)$, depends upon the application. If rotations are expensive, self-adjusting search trees may be inefficient (Sleator and Tarjan, 1983).

When do splay trees outperform skip lists? Splay trees use an amortized time bound rather than worst case or probabilistic time bound. Any operation in splay trees may take longer but a sequence of operations is guaranteed to take logarithmic time on the average. Most of the time, a long sequence of operations improves the future operations of the splay trees.

Skip lists on the other hand have a probabilistic time bound. There are many cases which can cause worst case performance for the skip lists, but the probability of those cases occurring is very low.

Splay trees perform very well if the access patterns are non-random. Non-random access includes those that follow the 90–10 rule, as well as several special cases such as sequential, double-ended access, and apparently access patterns that are typical of priority queues during some types of event simulations. When access sequences are random and uniform, splay trees do not do as well as other balanced trees (Weiss, 1999).

There are many applications that can benefit from the properties of splay trees. The main features, which distinguish splay trees from the skip lists are listed below:

Locality Locality is the process of looking for the same item repeatedly in the list. Some applications need to look repeatedly for the same item in the list. A good example is a network router. A network router examines the IP address in each packet and decides onto which outgoing interface to route the incoming packets (Narlikar et al., 2000). For this purpose routers keep a list of IP addresses and their corresponding interfaces. For each packet the router checks the list to find the entry with the same IP address. If the entry was found the packet is routed from the related interface. If the entry was not found in the list, the packet is routed through the default route if it is configured; or the packet is dropped if the default route is not configured. On the other hand, in order to manage the size of the IP packets, the Internet system segments the message to multiple parts depending on the size of the messages. Then each segment is encapsulated in an IP packet. Therefore, if an IP packet arrives at the router, it is likely that multiple packets with the same destination IP address also arrive. The router has to check the destination IP address in the list for each packet. If splay trees are used the first check will bring the list entry which contains the IP address and its corresponding outgoing interface number, to the root of the tree. Then the subsequent checking will be easy which results very quick packet forwarding by the router.

Therefore, splay trees are an appropriate choice to use for storing a list of IP addresses and the corresponding outgoing interfaces (Narlikar et al., 2000). When the first packet arrives at the router the router searches for the IP address in the list and moves that address to the root of the tree. Skip lists and balanced BSTs are static data structures and they do not change their structure during search operations. Therefore, their performance is very slow in this situation compared with splay trees. Generally, if an item is used repeatedly or a small number of items are being heavily used, splay trees can be used to store these items near to the root.

Packet Classification Packet classification is the process of categorizing IP packets into different flows for suitable processing. Flows are specified by rules applied to incoming packets. A collection of rules is called a classifier. Each rule specifies a flow that a packet may belong to based on some criteria applied to the packet header. For example, denying all e-mail traffic to destination IP address `192.13.40.90`. Or send all voice-over-IP traffic via a separate ATM network.

Packet classification provides security, monitoring, quality of service (QoS) and multimedia capabilities (Srinivasan et al., 2006). To classify a packet as belonging to a particular flow or set of flows, routers and firewalls perform a search over pre-defined rules. Routers and firewalls use multiple fields in the packet, such as destination or source IP address, or application port number, as the search key.

Splay trees perform *better* than skip lists and balanced *BST*s for packet classification. Splay trees are faster than skip lists and balanced trees when they are used for packet classifications (Srinivasan et al., 2006).

Memory Caches A cache is a temporary storage area where frequently accessed data can be stored for rapid access, where the original data require longer access time, compared to the cost of reading the cache. When the data is stored in the cache, future use can be made by accessing the cached copy rather than re-fetching the original data, so that the average access time is shorter. A cache is very effective because many computer applications access patterns have locality of reference. The computer *CPU*, hard drive, web browsers, and web servers use a cache in order to increase the speed of previously accessed data.

When the cache client such as *CPU*, web browser, or operating system wishes to access data in its original place, it first checks the cache. If an entry can be found with a tag matching that of the desired data, the data in the entry is used instead. This situation is known as a cache hit. So, for example, a web browser program might check its local cache on disk to see if it has a local copy of the contents of a web page at a particular URL.

Splay trees can be used to implement caches.

Skewed Operations In some applications the distribution of required operations will be skewed rather than uniform. That means some operations will be performed repeatedly. Since splay trees adjust according to usage therefore, they are efficient to use in such applications. Splay trees perform better than a fixed tree when the access pattern is non-uniform (Sleator and Tarjan, 1985). On the other hand, if the distribution of operations is more uniform, and the items are all equally likely to be accessed, then a randomized data structure such as a skip list is preferable. Generally, if the set of frequently accessed elements is a small subset of the elements in the tree, splay trees perform better than skip lists and other

*BST*s.

Our observations show that uniform operations result in worst case splay trees. Some operations lead to best case splay trees, i.e. when the order of these operations is in ascending or descending order or sequentially accessing all the items of a degenerated splay tree from the root to the leaf of the tree. Another example is the insertion of sorted data items into a splay tree.

Individual operations within a sequence can be expensive, which may be a drawback in real-time applications (Sleator and Tarjan, 1985). In certain real-time applications, we must be sure that an operation will be completed within a certain time bound. For such applications, self-adjusting trees may be undesirable, since they can take significantly longer on an individual operation than expected time instead of $O(\log N)$ time. For real-time systems, skip lists may be usable if an adequate safety margin is provided (Pugh, 1990).

5.4.1 Theoretical Comparison of *AVL* Trees, Splay Trees, and Skip Lists

Table 5.32 shows the results of our theoretical comparison for inserting ascending ordered data items between *AVL* trees, splay trees, and skip lists.

Table 5.32: Comparing ascending insertion.

Operation Type	AVL Trees	Bottom-up Splay Trees	Top-down Splay Trees	Skip Lists
Best Comp.	0	0	0	1
Avg. Comp.	$\frac{N\log N+2C}{2N}$	≈ 1	1	$\geq \log N$ *
Worst Comp.	$1+\log N$	1	1	$O(N)$ *
Total Comp.	$\frac{N}{2}\log N + C$	N-1	N-1	*
Best Rot.	0	0	0	0
Avg. Rot.	$\frac{N-(\log N+1)}{N} \approx 1$	≈ 1	0	0
Worst Rot.	1	1	0	0
Total Rot.	$N-(\log N+1)$	N-1	0	0
Best Height	$\log(N+1)$	O(N)	O(N)	$\log N + 1$
Avg. Height	$0.5+\log N$	O(N)	O(N)	$\log N + 1$
Worst Height	$1+\log N$	O(N)	O(N)	$O(N)$ *
Tree Shape	Full *BST*	Degenerate	Degenerate	Balanced

Notes for Table 5.32

* For skip lists the number of required comparisons depends on the number of levels, distribution of items in the levels and the order in which the data items are inserted. Since skip lists have $\log N$ levels with high probability, therefore, the number of

comparisons is $\log N + d$, where d represents the distribution of items in the levels and the order in which the items are inserted into the skip list. Our theoretical and experimental analysis show that, if the items are inserted in ascending order then d can be $1 \leq d \leq N$. However, the probability for d to be N or even close to N is very low. The probability for d to be equal to N is $\frac{1}{2^N}$.

Table 5.33 shows the results of our theoretical comparison for inserting descending ordered data between *BST*, *AVL* tree, splay trees, and skip lists.

Table 5.33: Comparing descending insertion.

Operation Type	AVL Trees	Bottom-up Splay Trees	Top-down Splay Trees	Skip Lists
Best Comp.	0	0	0	1
Avg. Comp.	$\frac{N \log N + 2C}{2N}$	≈ 1	1	$1 + \log N$ *
Worst Comp.	$1 + \log N$	1	1	$O(N)$ *
Total Comp.	$\frac{N}{2} \log N + C$	N-1	N-1	*
Best Rot.	0	0	0	0
Avg. Rot.	$\frac{N-(\log N+1)}{N} \approx 1$	≈ 1	0	0
Worst Rot.	1	1	0	0
Total Rot.	$N - (\log N + 1)$	N-1	0	0
Best Height	$\log(N + 1)$	O(N)	O(N)	$\log N$
Avg. Height	$\log(N + 1)$	O(N)	O(N)	$\log N$
Worst Height	$1 + \log N$	O(N)	O(N)	$O(N)$ *
Tree Shape	Full *BST*	Degenerate	Degenerate	Balanced

Notes for Table 5.33

* For skip lists the number of required comparisons depends on the number of levels, distribution of items in the levels and the order of insertion. Since skip lists have $\log N$ levels with high probability, therefore, the number of comparisons is $\log N + d$, where d represents the distribution of items in the levels and the order in which the items are inserted into the skip list. Our theoretical and experimental analysis show that, if the items are inserted in descending order then d is 0.

Table 5.34 shows the results of our theoretical comparison for random insertion between standard *BST*, *AVL* tree, splay trees, and skip lists.

Notes for Table 5.34

* The required number of rotations for inserting unsorted data items into an *AVL* tree is not fixed but depends on the order in which the items are inserted into the tree, but, it is always less than inserting the same number of items in sorted order.

** When unsorted data items are inserted into a bottom-up splay tree, the required number of comparisons and rotations depend on the order of insertion, but, it is more than inserting the same number of items in sorted order. When unsorted data items are inserted into a bottom-up splay tree the number of comparisons is

Table 5.34: Comparing random insertion.

Operation Type	AVL Trees	Bottom-up Splay Trees	Top-down Splay Trees	Skip Lists
Best Comp.	1	**	***	****
Avg. Comp.	$\log N$	**	***	****
Worst Comp.	$\log N + 4$	**	***	****
Total Comp.		**	***	****
Best Rot.	0	**	***	****
Avg. Rot.	$46\%N$ *	**	***	****
Worst Rot.	$46\%N$ *	**	***	****
Total Rot.	$46\%N$ *	**	***	****
Best Height	$\log(N + 1)$	**	***	****
Avg. Height	$1 + \log N$	**	***	****
Worst Height	$4 + \log N$	**	***	****
Tree Shape	Balanced	**	***	****

always equal to the number of rotations. The structure of the tree also depends on the insertion order. Some orders yield degenerate trees, and some other orders yield full balanced *BST*s.

*** The number of comparisons and rotations for inserting unsorted data items, into a top-down splay tree also depends on the order of insertion. The number of comparisons is the same as for bottom-up splay trees, but the number of rotations is less than for a bottom-up splay tree.

**** For skip lists the number of comparisons depends on the number of levels, distribution of items in the levels and the order of insertion. Since skip lists have $\log N$ levels with high probability. The number of comparisons is $\log N + d$, where d represents the distribution of items in the levels and the order in which the items are inserted into the skip list. Our analysis shows that, if the items are inserted randomly then d can be $1 \leq d \leq N$. However, the probability for d to be N or even close to N is very low. The probability for d to be equal to N is $\frac{1}{2^N}$.

Table 5.35 shows the results of our theoretical comparison for accessing all the items from the root to the leaf.

Notes for Table 5.35

In the above table the required number of comparisons and rotations for bottom-up and top-down splay trees are according to the sequential access to a degenerate tree from the root to the leaf of the tree. The result of sequential access from the leaf to the root or random access will be different. The reason is, each access to the items of a splay tree changes the structure of the tree.

* For skip lists the number of required comparisons depends on the number of levels, distribution of items in the levels and the order of insertion. Since skip lists have $\log N$ levels with high probability, therefore, the number of comparisons is $\log N+d$, where d represents the distribution of items in the levels and the order in which the items are inserted into the skip list. Our analysis shows that, if the items are inserted in descending order then d is 0. But if the items are inserted in ascending

Table 5.35: Comparing sequential access to all items from root to the leaf.

Operation Type	*AVL* Trees	Bottom-up Splay Trees	Top-down Splay Trees	Skip Lists
Best Comp.	1	1	1	1
Avg. Comp.	$(\sum_{i=1}^{\log(n+1)-1} 2^i(i+1))/N$	≈ 2	≈ 2	$1+\log N*$
Worst Comp.	$4+\log N$	2	2	$O(N)$ *
Total Comp.	$\sum_{i=0}^{\log(n+1)-1} 2^i(i+1)$	$2N-1$	$2N-1$	*
Best Rot.	0	0	0	0
Avg. Rot.	0	≈ 1	0	0
Worst Rot.	0	1	0	0
Total Rot.	0	$N-1$	0	0
Best Height	$\log(N+1)$	$O(N)$	$O(N)$	$\log N$
Avg. Height	$\log N$	$O(N)$	$O(N)$	$\log N$
Worst Height	$1+\log N$	$O(N)$	$O(N)$	$O(N)$ *
Tree Shape	Full *BST*	Degenerate	Degenerate	Balanced

or random orders then d can be $1 \leq d \leq N$. However, the probability for d to be N or even close to N is very low. The probability for d to equal N is $\frac{1}{2^N}$.

Chapter 6

Experimental Results

This chapter describes our experimental results. The experiments include the insertion and search performance of *AVL* trees, splay trees and skip lists. The required number of comparisons and rotations for insertion and search operations of these data structures are programmatically calculated, tabulated, and the results are compared with the theory. The search performance and run time behaviors are practically tested and analyzed. For skip lists the effect of the number of levels and distribution of items in the levels are empirically tested and analyzed. For splay trees the search operations for one class keys and two class keys—applying the 90–10 rule—are analyzed.

The experiments test and verify the theoretical results by comparing search and insertion performance of *AVL* trees, splay trees and skip lists with each other.

All the experiments were done using Java on various Linux based machines and a Solaris based machine. The size of the data sets we used are powers of 2 so that the logarithms base 2 are obvious.

6.1 *AVL* Trees

6.1.1 Inserting Sorted Data into an *AVL* Tree

Our experimental results show that, if the data items are inserted in ascending or descending orders, into an *AVL* tree, then the required number of rotations for inserting N items is:

$$N - (\log N + 1)$$

So the *average* number of rotations is

$$\frac{N - (\log N + 1)}{N} \approx 1.$$

For sorted insertion our experimental results correspond exactly with the theoretical results.

Table 6.1 shows our experimental results for inserting *ascending* or *descending sorted data into an AVL tree.*

Table 6.1: Experimental results for inserting data in sorted order.

Number of Items	Total Rotations	Rotation probability
1	0	0.0000
2	0	0.0000
3	1	0.3333
4	1	0.2500
16	11	0.6875
64	57	0.8906
256	247	0.9648
1024	1013	0.9893
4096	4083	0.9968
16384	16369	0.9991
65536	65519	0.9997
262144	262125	0.9999
1048576	1048555	0.9999

When sorted data items are inserted into an *AVL* tree, the tree stays balanced because it is an *AVL* tree and thus the height of the tree is logarithmic. In the worst case the height of the tree is $1 + \log N$, in the best case the height of the tree is $\log(N + 1)$ and the average height of the tree is $0.5 \log N$. The worst case is when the item with key 2^i is entered into the tree, where i is an integer. The best case is when the item with key $2^i - 1$ is entered into the tree, where i is an integer.

Table 6.2 shows our experimental results for the number of nodes and number of levels, when *sorted data is inserted into an AVL tree.*

Table 6.2: Number of items versus levels.

Number of Items	Number of Levels
1	1
2–3	2
4–7	3
8–15	4
32–63	6
128–255	8
512–1023	10
2048–4095	12
8192–16383	14
32768–65535	16
131072–262143	18
524288–1048575	20

For inserting sorted data items into an *AVL* tree, our experimental results support the theory that, the height of the tree is almost exactly logarithmic.

Our experimental results support the theory for the required number of comparisons when sorted data items are inserted into *AVL* tree. That is, the total number of comparisons for inserting sorted data items into an *AVL* tree is: $\frac{N}{2}\log N + C$, where C is the total number of required comparisons for inserting $\frac{N}{2}$ items. The average number of comparison is $\frac{N\log N + 2C}{2N}$.

Table 6.3 shows our experimental results for the required number of comparisons when *sorted items are inserted into an AVL tree.*

Table 6.3: Experimental results for the number of comparisons.

Number of Items	Total Comparisons	Average Comparisons
2	1	0.5000
4	5	1.2500
8	17	2.1250
16	49	3.0625
64	321	5.0156
256	1793	7.0039
1024	9217	9.0010
4096	45057	11.0002
16384	212993	13.0001
65536	983041	15.0000
262144	4456449	17.0000
1048576	19922945	19.0000

6.1.1.1 Timing Insertion of Sorted Data into an *AVL* Tree

The run-time behavior of AVL trees was programmatically calculated and recorded. Table 6.4 shows the total time for 30 repetitions, average time for a single repetition, standard deviation, σ, and average time for a single insertion. Insertion of descending order yields to the same timing results.

Table 6.4: Time requirement for inserting ascending sorted data.

Number of Items	Total Time (s) for 30 Repetitions	Average Time (ms)	σ	Average Time μ secs.
16384	0.3491	11.64	2.1916	0.7102
32768	1.0582	35.27	4.0444	1.0764
65536	3.0558	101.86	7.8803	1.5542
131072	5.4712	182.37	20.8637	1.3914
262144	10.4017	346.72	40.3712	1.3226
524288	22.2717	742.39	77.4441	1.4160
1048576	43.6493	1454.98	160.6548	1.3876

6.1.2 Inserting Unsorted Data into an *AVL* Tree

When unsorted data items are inserted into an *AVL* tree, the required rotation, for keeping the tree balanced, may be single-left, single-right or double. Our experiments show that, 25% of the rotations are single-left, 25% are single-right and 50% are double. The total number of cases which need rotation is 46% of N, where N is the number of items. The number of required rotations for inserting unsorted data is not fixed, and depends on the number of items and the order in which the items are inserted into the tree. However, for all $N \geq 8$, it is less than the required number of rotations when the same number of items were inserted in a sorted order.

Table 6.5 shows the experimental results for inserting *unsorted data into an AVL tree.*

Table 6.5: Experimental result for unsorted data.

Number of Items	Left Rotations	Right Rotations	Double Rotations	Total Rotations	Rotation Probability
32	3	4	7	21	65.63
64	8	6	14	42	65.63
128	14	15	29	87	67.96
256	33	29	57	166	64.84
512	60	62	122	366	71.48
1024	113	122	239	713	69.62
16384	1914	1896	3832	11474	70.03
32768	3872	3817	7600	22889	69.85
65536	7713	7506	15191	45601	69.58
262144	30600	30645	60529	182303	69.54
1,000,000	116197	116447	233052	698748	69.87

In the case of unsorted data the height of the tree is close to logarithmic. Our experimental analysis shows that the height of the tree will be $1 + \log N$ up to $4 + \log N$.

Table 6.6 shows the experimental results for the height of the tree.

6.1.2.1 Time Requirement for Inserting Unsorted Data

Even though unsorted data requires less rotations than sorted data, the time requirement is more than that for sorted data. It means if we insert 1000000 unsorted items into an *AVL* tree, it will take slightly more time than if these 1000000 elements are inserted in a sorted order.

Table 6.7 shows the experimental timing result for inserting *unsorted data into an AVL tree.*

Table 6.6: Experimental results for the height of the tree.

Number of Items	Number of Levels
8	4
30	6
32	6
80	8
300	10
500	11
800	12
16500	17
65688	19
1000000	24

Table 6.7: Experimental results for inserting unsorted data.

Number of Items	Total Time (s) for 30 Repetitions	Average Time (ms)	σ	Average Time for an Insertion μ secs.
16384	0.6609	22.03	4.6795	1.3445
32768	1.3625	45.42	6.6924	1.3861
65536	3.5200	117.33	10.9296	1.7904
131072	6.6538	221.79	24.7740	1.6921
262144	13.2102	440.34	48.9074	1.6798
524288	26.9129	897.10	97.3714	1.7111
1048576	53.6187	1787.29	195.7715	1.7045

6.1.3 *AVL* Search Cost

In a full *AVL* tree each level contain 2^i items, where i is the level number, and each item in level i needs i comparison. Our experimental results show that the total search cost of accessing all the items of a full *AVL* tree is: $\sum_{i=0}^{\log(N+1)-1} 2^i(i+1)$ comparisons. The average search cost is: $\frac{\sum_{i=0}^{\log(N+1)-1} 2^i(i+1)}{N}$ comparisons.

Table 6.8 shows our experimental results for the total and average number of comparisons when *all the items of AVL tree are accessed.*

Table 6.9 shows the experimental timing for *searching all the items of AVL tree.*

6.1.4 Comparing Experimental and Theoretical Results

Our experimental results for the search and update operations of *AVL* tree are the same as our theoretical results. The number of comparisons and rotations for inserting sorted data items are exactly the same, in both our theoretical and experimental results. The number of levels for inserting sorted and unsorted data items are also the same in both our theoretical and experimental results.

Our theoretical results for inserting unsorted data items say that the exact

Table 6.8: Experimental results for searching an AVL tree.

Number of Items	Total Comparisons	Average Comparisons
3	5	1.66
7	17	2.42
15	49	3.26
31	129	4.16
63	321	5.09
127	769	6.05
255	1793	7.03
511	4097	8.01
1023	9217	9.00
2047	20481	10.00
4095	45057	11.00
8191	98305	12.00
16383	212993	13.00
32767	458753	14.00
65535	983041	15.00
131071	2097153	16.00
262143	4456449	17.00
524287	9437185	18.00
1048575	19922945	19.00

Table 6.9: Experimental results for searching every item of an AVL tree.

Number of Items	Total Time(s) for 30 Repetitions	Average Time (ms)	σ	Average Time for a Search μ secs.
16384	0.4925	16.42	2.4348	1.0010
32768	0.4933	16.44	3.9546	0.5018
65536	1.1829	39.43	5.0302	0.6017
131072	2.2974	76.58	9.0051	0.5842
262144	4.6136	153.79	17.0603	0.5866
524288	9.8212	327.37	33.7517	0.6244
1048576	20.3237	677.46	70.4748	0.6461

number of comparisons and rotations depends on the order of data items, but it is less than, when the same number of items are inserted in sorted order. In our experiments we calculated the number of comparisons and rotations for different orders of data items, and we obtained the same result as the theoretical results.

Our theoretical results show that the total and average search comparison are $\sum_{i=0}^{\log(N+1)-1} 2^i(i+1)$ and $\frac{\sum_{i=0}^{\log(N+1)-1} 2^i(i+1)}{N}$, respectively. Our experiments give the same results.

Therefore, regarding *AVL* trees, our theoretical and experimental results support each other and there is no inconsistency between our theoretical and experimental results.

6.2 Skip Lists

6.2.1 Experimental Results for the Number of Levels

Our experimental results support the theory that, the number of levels affects the performance of the skip list. The experiments show that the number of levels will be close to $\log N$ with high probability.

Table 6.10 shows our results for the *number of levels in a skip list.*

Table 6.10: Experimental results for the number of levels.

Number of Items	$\log N$	First Run	Second Run	Third Run	Fourth Run
256	8	7	9	8	7
512	9	9	9	10	9
1024	10	10	11	10	9
2048	11	10	10	11	10
4096	12	12	11	11	13
8192	13	11	12	14	13
16384	14	14	14	15	15
32768	15	17	14	17	15
65536	16	18	16	17	18
131072	17	17	16	21	18
262144	18	21	17	17	17
524288	19	16	18	19	19
1048576	20	21	20	18	19

These experiments were done on 13 different numbers of items in the range 256–1048576. For each number the program was run 4 times. The 52 cases are distributed as follows: 17 cases have $\log N$ levels, 16 cases $\log N+1$ levels, 10 cases have $\log N - 1$ levels. In the remaining 9 cases: 4 cases have $\log N + 2$ levels, 2 cases have $\log N - 2$ levels, 1 case has $\log N + 3$ levels, 1 case has $\log N - 3$ levels, and 1 case has $\log N + 4$ levels.

6.2.2 Distribution of Items in Each Level

Our experimental results support the theory that the distribution of items in the levels affects the performance of the skip lists.

Table 6.11 shows our experimental results for the *distribution of items in the levels in a skip list.* This table shows that, besides the number of levels, the distribution of items in the levels also affect the performance of the skip list. Considering the table, in some cases the skip list with $\log N$ levels has the best performance. In some other cases the skip list with $\log N - 1$ levels has the best performance. Skip lists with the same number of levels for the same number of items may perform differently. This confirms that, besides the number of levels, the distribution of items in the levels also affects the performance of the skip list.

Table 6.11: Experimental results for the distribution of items in the levels.

Number of Items	Number of Levels	Total Time (Sec)	Standard Deviation	Average Time for single insertion μ s
32768	17	0.0702	25.0465	2.1423
	14	0.0707	25.3567	2.1576
	17	0.0701	25.6426	2.1393
	15	0.0730	25.8887	2.2278
65536	18	0.1243	22.1226	1.8967
	16	0.1283	29.1463	1.9577
	17	0.1240	22.6280	1.8921
	18	0.1240	22.1521	1.8921
131072	17	0.2450	48.2721	1.8692
	16	0.2420	43.7013	1.8463
	21	0.2475	43.9928	1.8883
	18	0.2381	37.3447	1.8166
262144	21	0.4747	72.9647	1.8147
	17	0.4468	68.5452	1.7044
	17	0.4538	73.8355	1.7311
	17	0.4507	73.8310	1.7193
524288	16	0.8840	134.6113	1.6861
	18	0.8964	128.2645	1.7097
	19	0.8960	126.0836	1.7090
	19	0.9005	131.4925	1.7176
1048576	21	2.0304	290.5449	1.9363
	20	1.9879	288.4646	1.8958
	18	1.9608	290.4399	1.8700
	19	1.9655	283.8019	1.8744

6.2.3 Skip List Insertion

Our experimental results show that, the order in which the data is inserted into a skip list, does not affect the average search comparisons. But, it still affects the average number of comparisons for inserting data into a skip list. The descending ordered insertion cost is less than the ascending and random ordered insertion cost. Our theoretical analysis shows that the random insertion cost is less than the ascending insertion cost. But, our experimental results show that the ascending ordered insertion cost is less than the random insertion cost.

6.2.3.1 Timing Results

Table 6.12 shows our experimental results for *inserting data items in ascending order into a skip list.*

Table 6.13 shows our experimental results for inserting data items in descending order into a skip list.

Table 6.12: Timings for inserting ascending ordered data into a skip list.

Number of Items	Total Time(s) for 30 Repetition	Average Time(ms)	σ	Average for one insertion μ s
16384	0.9788	32.63	4.5485	1.9913
32768	1.7944	59.81	8.0291	1.8254
65536	4.0793	135.98	13.8803	2.0748
131072	7.1546	238.49	29.2792	1.8195
262144	14.4390	481.30	54.6476	1.8360
524288	28.9955	966.52	106.5134	1.8435
1048576	57.2318	1907.73	211.7464	1.8193

Table 6.13: Timings for inserting descending ordered data into a skip list.

Number of Items	Total Time(s) for 30 Repetition	Average Time(ms)	σ	Average for one insertion μ s
16384	0.8740	29.13	3.6925	1.7781
32768	1.6044	53.48	7.0246	1.6321
65536	3.7123	123.74	12.3199	1.8882
131072	6.3975	213.25	26.4979	1.6270
262144	12.3912	413.04	49.4233	1.5756
524288	25.4369	847.90	92.683	1.6172
1048576	52.0794	1735.98	187.1503	1.6556

Table 6.14 shows our experimental results for inserting data items in random order into a skip list.

Table 6.14: Timings for inserting random ordered data into a skip list.

Number of Items	Total Time(s) for 30 Repetition	Average Time(ms)	σ	Average for one insertion μ s
16384	1.1365	37.88	5.8565	2.3122
32768	2.3559	78.53	9.5268	2.3966
65536	3.7925	193.08	17.6536	2.9462
131072	11.5550	385.17	40.5972	2.9386
262144	25.2904	843.01	83.6645	3.2158
524288	50.9471	1698.24	180.0230	3.2391
1048576	105.9887	3532.96	368.2043	3.3693

6.2.4 Skip List Search Performance

The search cost of a skip list is logarithmic on average. However, our experimental results show that, the search cost for an item depends on the number of levels and the distribution of items in the levels.

Table 6.15 shows the experimental timing for *searching all the items of a skip list.*

Table 6.15: Experimental results for searching all the items of a skip list.

Number of Items	Total Time(s) for 30 Repetition	Average Time(ms)	σ	Average Time for a Search μ secs.
16384	0.1202	4.01	0.7769	0.2444
32768	0.2522	8.41	1.0870	0.2566
65536	0.5223	17.51	1.9315	0.2672
131072	1.1013	36.71	3.8413	0.2801
262144	2.2990	76.63	7.9509	0.2923
524288	4.8863	162.88	16.5520	0.3107
1048576	10.1815	339.38	35.0408	0.3237

6.2.5 Deleting an Item from a Skip List

Deleting items from a skip list is similar to inserting items into a skip list. However, the search cost is logarithmic on average, so deleting an item from a skip list needs $\log N$ comparisons for finding the item, plus the time for updating the references and decreasing the maximum level, when a node with a maximum level is deleted.

6.2.6 Comparing Experimental and Theoretical Results

Our experimental results support the theory that the number of levels and the distribution of items in the levels affect the performance of the skip list. Our experimental results show different performances for the same number of items and the same number of levels. This confirms the theory that, beside the number of levels, the distribution of items in the levels also affects the performance of the skip list.

For the search performance in the skip list our theoretical and experimental analysis show the same results. Our theoretical results show that the search time depends on the number of levels and distribution of items in the levels. Our experimental analysis show the same results.

6.3 Top-down Splay Trees

Our experimental results show that, the insertion order into a top-down splay tree, affects the performance of the top-down splay tree. Insertion of ordered data items into a top-down splay tree requires less comparisons than inserting the same number of items in unsorted order. We now present our experimental results for inserting sorted and unsorted data items into top-down splay trees.

6.3.1 Insertion of Sorted Data Items

Our experimental results show that, insertion of N sorted data items into a top-down splay tree in ascending or descending order, require $N - 1$ comparisons. Unlike the bottom-up splay tree, insertion of sorted data items into a top-down splay tree does not need any rotations.

Table 6.16 shows the experimental results for the *number of comparisons and rotations when ascending or descending ordered data items are inserted into a top-down splay tree.*

Table 6.16: Number of comparisons and rotations for inserting ordered items into a top-down splay tree.

Number of Items	Total Comparisons	Total Rotation
256	255	0
1024	1023	0
4096	4095	0
16384	16384	0
65536	65535	0
262144	262143	0
1048576	1048575	0

6.3.2 Insertion of Unsorted Data Items

Our experimental results show that, when unsorted data items are inserted into a top-down splay tree, the required number of comparisons is more than inserting the same number of items in sorted order. Unsorted insertion of data items also requires rotations.

Table 6.17 shows the number of comparisons and rotation when *random ordered data items are inserted into a top-down splay tree.*

Table 6.17: Number of comparisons and rotations for inserting random ordered items into a top-down splay tree.

Number of Items	Total Comparison	Average Comparison	Total Rotation	Average Rotation
256	2514	9.82	881	3.441406
1024	14104	13.77	4987	4.870117
4096	72741	17.75	25676	6.268555
16384	355772	21.71	125541	7.662415
65536	1685905	25.72	595441	9.085709
262144	7792344	29.72	2750522	10.492409
1048576	35360885	33.72	12485117	11.906735

6.3.2.1 Insertion Timing

Our experimental results show that insertion of sorted data items into a top-down splay tree require less time than inserting the same number of items in random order. The following section contain our experimental timing for inserting sorted and unsorted data items into top-down splay trees.

Table 6.18 shows the experimental results for inserting *strictly ascending data into a top-down splay tree.*

Table 6.18: Ascending ordered insertion into a top-down splay tree.

Number of Items	Total Time(s) for 30 Repetition	Average Time (ms)	σ	Average Time for an Insertion μ sec.
16384	0.1988	6.63	1.8027	0.4045
32768	0.7484	24.95	3.6285	0.7613
65536	1.9524	65.08	5.9745	0.9930
131072	3.3763	112.54	13.8605	0.8586
262144	6.9542	231.81	25.5942	0.8843
524288	12.3675	412.25	51.0229	0.7863
1048576	23.8297	794.32	104.9841	0.7575

Table 6.19 shows the experimental results for inserting *strictly descending data into a top-down splay tree.*

Table 6.19: Descending ordered insertion into a top-down splay tree.

Number of Items	Total Time(s) for 30 Repetition	Average Time (ms)	σ	Average Time for an Insertion μ sec.
16384	0.1997	6.66	2.4456	0.4062
32768	0.7126	23.75	3.8742	0.7249
65536	2.1447	71.49	6.0710	1.0908
131072	3.5875	119.58	14.9091	0.9123
262144	6.6248	220.83	27.3081	0.8424
524288	11.5643	385.48	50.1044	0.7352
1048576	23.2090	733.63	97.5527	0.7378

Table 6.20 shows the experimental results for inserting *randomly ordered data into a top-down splay tree.*

6.3.3 Searching a Top-down Splay Tree

Our experimental analysis includes two kinds of searching:

1. Splay Trees with One-class keys,

2. Splay Trees with Two-class keys.

Table 6.20: Randomly ordered insertion into a top-down splay tree.

Number of Items	Total Time(s) for 30 Repetition	Average Time (ms)	σ	Average Time for an Insertion μ sec.
16384	0.5275	17.58	3.4743	1.0732
32768	1.3701	45.67	5.7158	1.3938
65536	3.7491	124.97	10.3945	1.9069
131072	8.8066	293.55	25.8941	2.2396
262144	20.1351	671.17	61.4467	2.5603
524288	42.0433	1401.44	141.4106	2.6730
1048576	71.9864	2399.55	305.3005	2.2884

Splay trees with one-class keys consider the access to all items of the tree with equal probability. In this case, each element of the tree is accessed once. On the other hand, splay trees with two-class keys consider the 90–10 rules, which state that 90 percent of the accesses is to 10 percent of the elements. The 90–10 rule applies the concept of locality when using splay trees.

In order to compare the run-time behavior of one-class and two-class splay trees, we consider a degenerate top-down tree for both types of access.

6.3.3.1 Splay Tree with One-class keys

We first consider one-class splay trees, by accessing all the items from the root to the leaf of a degenerate top-down splay tree.

Number of Comparisons and Rotations Our experimental results show that the total search cost for accessing all the elements sequentially from the root to the leaf, is $2N-1$ comparisons, where N is the number of items in the tree. Sequential access to all elements from the root to the leaf does not require any rotations.

Table 6.21 shows the experimental results for the number of *comparisons and rotations when all the items are accessed from the root to the leaf in a degenerate top-down splay tree.*

Table 6.21: Number of comparisons and rotations for sequential search.

Number of Items	Total Comparisons	Average Comparisons	Total Rotations	Average Rotations
256	511	1.99	0	0.00
1024	2047	1.99	0	0.00
4096	8191	1.99	0	0.00
16384	32767	1.99	0	0.00
65536	131071	1.99	0	0.00
262144	524287	1.99	0	0.00
1048576	2097151	1.99	0	0.00

Our experimental results show that the total number of comparisons for accessing all the items from the leaf to the root of a degenerate top-down splay tree is from $2N$ to $5.32N$ comparisons, where N is the number of items and N is $2 \leq N \leq 1048576$. Therefore, the average number of comparisons is from 2 to 5.32. The total number of required rotation is from 0 to $1.45N$ where N is $1 \leq N \leq 1048576$.

Table 6.22 shows the experimental results for the number of required comparisons and rotation when *all the items are accessed from the leaf to the root of a degenerate top-down splay tree.*

Table 6.22: Number of comparisons and rotations for sequential search.

Number of Items	Total Comparisons	Average Comparisons	Total Rotations	Average Rotations
2	4	2.000000	0	0.00
8	28	3.500000	4	0.50
32	146	4.562500	33	1.03
128	646	5.046875	166	1.29
256	1320	5.156250	353	1.37
1024	5395	5.268555	1459	1.42
4096	21728	5.304688	5905	1.44
16384	87093	5.315735	23717	1.44
65536	348590	5.319061	94989	1.44
262144	1394617	5.320042	380136	1.45
1048576	5578631	5.320197	1521159	1.45

Table 6.23 shows the experimental results for the number of *comparisons and rotations when all the items of a top-down splay tree are accessed in random order.*

Table 6.23: Number of comparisons and rotations for random search.

Number of Items	Total Comparisons	Average Comparisons	Total Rotations	Average Rotations
2	4	2.0000	0	0.000000
8	32	4.0000	9	1.125000
32	218	6.8125	59	1.843750
128	1276	9.9687	383	2.992188
256	2841	11.0976	848	3.312500
1024	14471	14.1318	4426	4.322266
4096	69998	17.0893	21515	5.252686
16384	327812	20.0080	101798	6.213257
65536	1500352	22.8935	467011	7.126022
262144	6761574	25.7933	2108548	8.043472
1048576	30076736	28.6834	9398662	8.963263

6.3.3.2 Timing for Splay Trees with One-class Keys

Table 6.24 shows the experimental results for *searching all the items from the root to the leaf of a degenerate top-down splay tree.*

Table 6.24: Lookup timing in a top-down splay tree with one-class keys.

Number of Items	Total Time(s) for 30 Repetition	Average Time (ms)	σ	Average Time for a Lookup μ sec.
16384	0.1316	4.39	1.8833	0.2678
32768	0.3348	11.16	3.1994	0.3406
65536	0.3875	12.92	3.8564	0.1971
131072	0.8035	26.78	4.6884	0.2044
262144	1.4933	49.78	6.9413	0.1899
524288	2.9081	96.94	11.7476	0.1849
1048576	6.4100	213.67	25.5751	0.2038

Table 6.25 shows the experimental results for *searching all the items from the leaf to the root of a degenerate top-down splay tree.*

Table 6.25: Lookup timing in a top-down splay tree with one-class keys.

Number of Items	Total Time(s) for 30 Repetition	Average Time (ms)	σ	Average Time for a Lookup μ sec.
16384	0.2478	8.26	2.8874	0.5041
32768	0.6335	21.12	4.4086	0.6444
65536	1.0671	35.57	6.2412	0.5428
131072	2.1719	72.40	9.2178	0.5524
262144	4.3471	144.24	16.5074	0.5502
524288	8.9324	297.75	31.8676	0.5679
1048576	18.2683	608.94	66.3734	0.5807

Table 6.26 shows the experimental results for *searching all the items of a randomly built top-down splay tree in random order.*

Table 6.26: Lookup timing in a top-down splay tree with one-class keys.

Number of Items	Total Time(s) for 30 Repetition	Average Time (ms)	σ	Average Time for a Lookup μ sec.)
16384	0.2029	6.76	1.8646	0.4127
32768	0.3211	10.70	2.3778	0.3266
65536	0.7163	23.88	3.2072	0.3643
131072	1.5000	50.00	5.6029	0.3815
262144	3.3207	110.69	11.1846	0.4223
524288	6.4841	216.14	23.9056	0.4122
1048576	12.0755	402.52	54.9438	0.3839

6.3.3.3 Splay Tree with Two-class Keys

In our experiments for splay trees with two-class keys, we consider the keys of a degenerate top-down splay tree in two classes. Suppose that K represents all the keys in a splay tree. Let K_1 represent 10% of the keys, and let K_2 represent the other 90% of the keys. The probability for accessing a key from K_1 is 90%, and the probability for accessing a key from a K_2 is 10%.

Number of Comparisons and Rotations Table 6.27 shows the experimental results for the number of required comparisons and rotations when all the items of a top-down splay tree are *accessed in random order, using a splay tree with two-class keys.*

Table 6.27: Number of comparisons and rotations for random search in a top-down splay tree with two-class keys.

Number of Items	Total Comparisons	Average Comparisons	Total Rotations	Average Rotations
32	104	3.250000	19	0.593750
128	627	4.898438	156	1.218750
256	1624	6.343750	450	1.757813
1024	9350	9.130859	2784	2.718750
4096	48996	11.961914	14815	3.616943
16384	242422	14.796265	74237	4.531067
65536	1164299	17.765793	360257	5.497086
262144	5414563	20.654919	1681673	6.415073
1048576	24712052	23.567249	7696780	7.340221

6.3.3.4 Timing For Splay Trees with Two-class Keys

Table 6.28 shows the experimental results for *searching a top-down splay tree with two-class keys.*

Table 6.28: Lookup timing in a top-down splay tree with two-class keys.

Number of Items	Total Time (s)	Average Time (ms)	σ	Average Time for a Lookup μ sec.
16384	0.001	0.1	0.1054	0.0061
32768	0.002	0.2	0.2108	0.0061
65536	0.002	0.2	0.2108	0.0031
131072	0.003	0.3	0.3162	0.0023
262144	0.005	0.5	0.5270	0.0019
524288	0.007	0.7	0.7379	0.0013
1048576	1.242	124.2	130.9183	0.1184

6.3.4 Comparing Experimental and Theoretical Results

For inserting the sorted data items into a top-down splay tree our experimental results support our analysis. Our analysis and our experimental results show that the number of required comparisons for inserting sorted data items is $N-1$. Sorted insertion into a top-down splay tree does not require rotation.

Our experimental results for inserting unsorted items into a top-down splay tree is the same as our theoretical results.

Our theoretical and experimental results show that the required number of comparisons for accessing all the items in a degenerate top-down splay tree from the root to the leaf of the tree is $2N-1$ comparisons. There is no need for rotation when all the items of a degenerate top-down splay tree is accessed from the root to the leaf of the tree.

Our experimental results show that the required number of comparisons for accessing all the items from the leaf to the root of a degenerate top-down splay tree is from $2N$ to $5.32N$ comparisons, where N is $2 \leq N \leq 1048576$. And the number of required rotation is from 0 to $1.45N$ rotation where N is from $2 \leq N \leq 1048576$.

Table 6.29 *compares the timing for AVL trees, splay trees and skip lists.*

Table 6.29: Comparing the timing results for *AVL* trees, splay trees and skip lists.

Time for 10 Repetitions (s)	AVL Trees	Top-down Splay Trees	Skip Lists
Ascending Insertion	17.091	07.715	19.327
Descending Insertion	17.091	08.358	17.167
Random Insertion	18.600	27.301	35.456
Searching	6.895	4.690*	3.392

Notes for Table 6.29

* Unlike the other data structures, for splay trees the search time depends on the order in which the items are searched. The above mentioned time is for random search, in a randomly built top-down splay tree. But if we search a degenerate splay tree sequentially from the root to the leaf, then the total search time will be 1.9510 seconds. If we search a degenerate top-down splay tree from the leaf to the root of the tree, then the total search time will be 5.7920 seconds. For a top-down splay tree with two-class keys the total search time is 1.0850 seconds.

Table 6.29 shows that, top-down splay trees are better than *AVL* trees and skip lists for sorted insertion. For unsorted insertion *AVL* trees are better than splay trees and skip lists. For random search, skip lists are faster then *AVL* trees and splay trees. But splay trees with 90–10, two-class keys are faster than skip lists and *AVL*. Sequentially accessing all the items of a degenerate top-down splay trees from the root to the leaf of the tree is significantly faster than for *AVL* trees and skip lists.

Chapter 7

Conclusion

7.1 Overview

In this thesis we have studied the complexity of splay trees and skip lists. We presented theoretical and experimental analysis for splay trees and skip lists and identified their worst-case, average-case, and best-case behaviors. We have compared splay trees with skip lists, and then we identified where splay trees outperform skip lists.

Based on our studies we have analyzed the search and update operations of these data structures. Because of the involvement of randomization and probabilities in the Operation of skip lists, we used probabilistic approaches to analyze the performance of skip lists.

All the experiments were done on various machines, and the results were programmatically recorded. The experimental results were compared with the theoretical results. In the case of difference, the reason for the difference is explained. Where the theory seemed to be violated we attempted explanations of the aberrant behavior.

7.2 Summary of the Results

7.2.1 *AVL* Trees

Insertion of sorted data items into *AVL* trees, requires more rotations and comparisons than random insertion. Our theoretical and experimental results show that the total number of rotations for inserting N sorted data items into an *AVL* tree is:

$$N - (\log N + 1) \tag{7.1}$$

The average percentage of rotations for inserting N sorted items is 99.99%

of N, where N is equal to 1048576.

The total number of comparisons for inserting sorted data items into an AVL tree is:

$$\frac{N}{2} \log N + C,$$

where C is the total number of required comparisons for inserting $\frac{N}{2}$ items. The number of comparisons in the worst case is $1 + \log N$, and the average number of comparisons is

$$\frac{N \log N + 2C}{2N}.$$

Our experimental results show that, when unsorted data items are inserted into an AVL tree, the exact number of rotation depends on the insertion order of the data items, but, it is always less than the required number of rotations when the same number of items are inserted in a sorted order. The average percentage of rotations for random insertions into AVL tree is 46% of N.

When unsorted data is inserted into an AVL tree, the height of the tree is very close to logarithmic. Our experiments show that the height of the tree is between $1 + \log N$ and $4 + \log N$, for $2 \leq N \leq 1048576$.

Even though unsorted data items require less rotations than sorted data items, the insertion time for unsorted data is a little more than sorted insertion.

In a full AVL tree the total search cost for accessing all the items is:

$$\sum_{i=0}^{\log(N+1)-1} 2^i(i+1).$$

The average search cost is:

$$\frac{\sum_{i=0}^{\log(N+1)-1} 2^i(i+1)}{N}.$$

7.2.2 Skip Lists

Our theoretical analysis show that, in a skip list with N elements, for all even N, half of the items will move from level i to level $i+1$ with high probability. And for all odd N, $\frac{N}{2} \pm \frac{1}{2}$ items will move from level i to level $i+1$, with high probability. Therefore, the probability for having $\log N$ levels (except the base level) is higher than having $1 + \log N$, $2 + \log N \ldots$ levels. Our experimental result support our theoretical result.

Our analysis shows that the distribution of items in the levels of a skip list affects its the performance. The average number of comparisons increases if the items which are in the front of the list, reach the upper levels. But, the average number of comparisons decreases if the items which are in the middle and end of the skip list reach the upper levels. The reason is, when the middle items reach the upper levels it keeps the distribution of the items in balance, and when the items from the end of the skip list reach to the upper levels, it eliminates some comparisons with the end pointers.

Our analysis shows that the average number of comparisons for inserting 3 items into a skip list is 2.222, whereas the average number of comparisons for inserting the same three items into an AVL tree is 1.666 comparisons. The average number of comparisons for inserting 7 items into a skip list is 4.417 comparisons, whereas the average number of comparisons for inserting the same seven items into an AVL tree is 2.142 comparisons. Therefore, the cost of insertion into AVL trees is lower than for skip lists.

Our experimental results show that skip lists with the same number of items, and the same number of levels have different performance behaviors. This means that besides the number of levels, the distribution of items in the levels, and the location of high towers also affect the performance of the skip list.

Our analytical and experimental results conclude that, the insertion cost of skip lists does not depend only on the number of levels, and the distribution of items in the levels. It also depends on the order in which the data items are inserted into a skip list.

Our analysis shows that, when data items are inserted in descending order, the average number of comparisons equals the number of levels in the skip list. Therefore, the worst case insertion cost for inserting descending ordered data is $1 + \log N$. On the other hand if the data items are inserted in ascending order, then the required number of comparisons at each level is not restricted to one comparison. But, the number of comparisons also depends on the number of items which precede the newly inserted item. Since the data items in a skip list are in ascending order, some levels need more than one comparison, and the average number of comparisons is not logarithmic.

If we insert three items into a skip list in descending order, then the average number of comparisons is 3.00. But the average number of comparisons for inserting the same three items in ascending order is 3.78. Our analysis for inserting 7 items also shows that descending insertion requires less comparisons than ascending insertion.

Even the descending insertion cost is better than random insertion cost. However, the random insertion cost is better than ascending ordered insertion.

Our experimental results confirm the above observations. That is, for skip lists the descending ordered insertion cost is less than the ascending or random ordered insertion cost. Our observations show that the random insertion cost is less than the ascending insertion cost. But, our experimental results ANOMALOUSLY show that the ascending ordered insertion cost is less than the random insertion cost.

7.2.3 Splay Trees

Insertion of sorted data items, into a bottom-up or top-down splay trees require less comparisons, than inserting the same number of items into AVL trees, and skip lists. The total number of comparisons for inserting N sorted data items into a bottom-up or top-down splay trees is $N-1$ comparisons. Therefore, the average insertion needs $\frac{N-1}{N} \approx 1$ comparisons. The difference between bottom-up and top-down splay trees, for sorted insertion is the required number of rotations. For sorted insertions top-down splay trees do not require rotations, whereas a bottom-up splay trees needs $N-1$ rotations.

When the data items are inserted in a sorted order the top-down splay tree performs better than AVL trees, and skip lists in terms of comparisons and rotations. Bottom-up splay trees are also better, in terms of comparisons, than AVL trees and skip lists, but AVL trees are better than bottom-up splay trees in terms of rotations. For sorted insertion AVL trees need less rotations than bottom-up splay trees. Skip lists do not use rotation.

Insertion of sorted data items into splay trees yields a degenerate tree, where insertion into an AVL trees obviously preserves its balance.

When sorted data items are inserted into a bottom-up or top-down splay tree the best, average, and worst insertion cost are almost identical.

If we want to quickly insert sorted items and display them in reverse order, we can use top-down or bottom-up splay trees. Of course, a top-down splay tree is preferred.

When unsorted data items are inserted into bottom-up or top-down splay trees, the number of comparisons and rotations exceeds that for inserting the same number of items in ascending or descending orders. The total number of comparisons and rotations depends on the order of the data items. The resultant tree structure also depends on the order of the inserted data. However, top-down splay trees require less rotations than bottom-up splay trees. Our theoretical analysis shows that, for sorted and unsorted insertion, into a bottom-up splay tree, the number of comparisons is equal to the number of rotations, whereas for top-down splay trees, the number of comparisons exceeds the number of rotations.

The interesting point in our theoretical analysis is that, the *Zig-zag* rotations tend to improve the balance of a tree. For example if we insert the items in the following order: 70, 50, 30, 10, 60, 20, and 40, into a bottom-up splay tree, the resultant tree will be a full BST. The insertion process needs 4 *Zig*s, 1 *Zig-zig* and 4 *Zig-zag* rotations. The 4 *Zig-zag* rotations help the tree to be balanced. But, if we insert the same 7 items in the following order: 40, 50, 30, 60, 20, 70, and 10, which requires 3 *Zig*s, 5 *Zig-zig*s, and 0 *Zig-zag* rotations, the resultant tree turns out to be degenerate or a "completely unbalanced" tree. Therefore, the order which needs more *Zig-zag* rotations tends to yield balanced trees. The number of *Zig-zag* rotations increase when the data items are inserted in discrepancy order.

After inserting unsorted data items into a bottom-up or top-down splay tree, the resultant tree structure depends on the order of the data, but in most cases the insertion of unsorted data items into a top-down splay tree yields a degenerate tree. In terms of tree structure, bottom-up splay tree yield better balanced trees than top-down splay trees, when the same sequence of items is inserted. The reason is the *Zig-zag* rotation in the bottom-up splay tree. Top-down splay trees require rotations only in the *Zig-zig* case. For example, inserting the orders 70, 50, 30, 10, 60, 20, 40 and 10, 30, 50, 70, 20, 60, 40 into a top-down splay tree yields a completely unbalanced or degenerated tree. Whilst inserting the same sequences of items in the same orders into a bottom-up splay tree yields a full balanced BST. Again, the *Zig-zag* rotation tends to make the resultant bottom-up tree more balanced.

Our analysis shows another interesting point about unsorted insertion into a top-down splay tree. Using the standard *Zig-zag* approach yields better balanced BSTs, than using the simplified *Zig-zag* approach. The literature mentions that, the simplified *Zig-zag* approach simplifies the code but increases the number of iterations, where standard *Zig-zag* approach complicates the code and decreases the iterations (see Weiss, 1999, p. 619). This is true but, the simplified *Zig-zag* approach yields more degenerate trees. For example, using the simplified *Zig-zag* approach for inserting the orders 20, 60, 70, 30, 10, 50, 40 and 60, 20, 10, 50, 70, 30, 40 into a top-down splay tree, results more degenerated tree than using the standard *Zig-zag* approach.

If all the items of a degenerate bottom-up or top-down splay tree are accessed in sequential order, from the root to the leaf of the tree, the structure of the tree is changed from a left degenerate tree to a right degenerate tree, or vice versa. The order of the items will also change from ascending to descending order, or vice versa.

Accessing all the items of a degenerate bottom-up or top-down splay tree sequentially from the root to the leaf requires $2N - 1$ comparisons. Therefore, the average number of comparisons is

$$\frac{2N-1}{N}. \approx 2$$

A degenerate *top-down splay tree* does not require rotations, when all the items are accessed from the root to the leaf of the tree, where bottom-up splay tree needs $N - 1$ rotations for accessing all of its items from the root to the leaf of the tree. Therefore, the average number of rotations for accessing all the items of a degenerate *bottom-up splay tree* is

$$\frac{N-1}{N} \approx 1.$$

Degenerate bottom-up or top-down splay trees have a structure similar to a linked list. But, accessing all the items in a linked list requires $\frac{N+1}{2}$ comparisons on average, where accessing all the items in a degenerate splay trees requires 2 comparisons on average per item.

Accessing every item of a degenerate bottom-up splay tree from the leaf to the root of the tree requires $N + R$ comparisons, where N is the number of items and R is the total number of rotations. The total number of rotations is $C - N$ where C is the total number of comparisons, and N is the number of items.

Accessing every item of a degenerate top-down splay tree, from the leaf to the root of the tree needs from $2N$ to $5.32N$ comparisons, where $2 \leq N \leq 2^{20}$. The average number of comparisons is from 2 to 5.32. This means that the average number of comparisons in the worst case will be very close to 5.32, but does not exceed 5.32, for $2 \leq N \leq 2^{20}$. Accessing all the items of a degenerate top-down splay tree, also needs rotations. But, the number of rotations is always less that than for a bottom-up splay tree.

If all the items of a degenerate bottom-up or top-down splay tree are accessed in sequential order, from the *leaf* to the *root* of the tree, the structure of the tree is not changed. This is unlike the access to all items of a bottom-up or top-down splay trees from the *root* to the *leaf* of the tree.

Random access to all items of a degenerate bottom-up splay tree require $N+R$ comparisons, where N is the number of items and R is the number of rotations. The total number of rotations is $C-N$ where C is the number of comparisons and N is the number of items.

The result of random access, to all items of a degenerate bottom-up splay tree alters the structure of the tree to some better balanced *BST*. Some seemingly random access orderings can change the tree from its degenerate shape into a full balanced *BST*. When the leaf items are accessed first, and the root is accessed at the end, the resultant tree will be a full *BST*. But, if the root is accessed at the beginning or middle, then the resultant tree will be a degenerate or some kind of unbalanced tree. If the upper levels items are accessed before the bottom level items, than the resultant tree tends to take on a linked list shape.

The interesting point is that, the access orders which yield a full and balanced bottom-up splay tree, also yield a full and balanced bottom-up splay tree, when all items of a degenerated bottom-up splay tree are accessed in this order. All orders which yield a full and balanced *BST* have the root elements at the end of the order.

Random access to all items of a top-down splay tree requires a different number of comparisons and rotations, depending on the access order. The resultant tree structure also depends on the order in which the items are accessed. The number of comparisons and rotations decreases if the items which are in the root of the balanced *BST* are accessed in the middle. If the root items are accessed at the end, the number of comparisons and rotations increases.

7.3 Comparison Between *AVL* Trees, Splay Trees, and Skip Lists

Our theoretical and experimental results show that the insertion of sorted data items into a splay tree requires less comparisons than inserting the same number of items into a skip list and *AVL* trees. For ascending insertion *AVL* trees are faster than skip lists, but for descending insertion, skip lists perform better than *AVL* trees. For unsorted insertion *AVL* trees perform better than splay trees and skip lists. Splay trees are faster than skip lists when unsorted items are inserted.

The average number of comparisons for accessing all the items from the root to the leaf of a degenerate bottom-up or top-down splay tree is less than the average number of comparisons for accessing all the items in a skip list and balanced *BST*s. Our experimental result affirm that the average search cost for accessing all the items in a degenerate top-down splay tree is better than AVL tree and skip lists. Our experiments also show that the average search cost for accessing all the elements of an *AVL* tree is better than accessing all the items of a skip list.

Splay trees perform very well if the access patterns follow the 90–10 rule (see Weiss, 1999, p. 636). This is useful for applications which need to look repeatedly for the same item. Examples of such applications include routing tables, packet classifiers, caches for the *CPU*, hard drives, web browsers, and web servers (Narlikar et al., 2000) and (Srinivasan et al., 2006).

Splay trees perform better than search trees for non-uniform sequences of operations. When access sequences are random and uniform, splay trees do not do as well as other balanced trees. If the distribution of operations is more uniform, and the items are all equally likely to be accessed, then randomized data structure such as a skip list is preferable. There are some uniform operations which have a best case when splay trees are used if the order of these operations is in ascending or descending order. The example is sequentially accessing all the items of a degenerate splay tree from the root to the leaf of the tree. Another example is the insertion of sorted data items into a splay tree which has also the best case for building a data structure (Sleator and Tarjan, 1985).

7.4 Our Findings

We conclude by summarizing some key points of our research:

1. Our analysis and experimental results show that the total number of rotations for inserting N sorted data items into AVL tree is:

$$N - (\log N + 1)$$

The average percentage of rotations for inserting N sorted items is 99.99% of N, where $N = 2^{20}$. The average percentage of rotations for unsorted insertions into an AVL tree is 46% of N, where $32 \leq N \leq 2^{20}$.

2. The total number of comparisons for inserting sorted data items into an AVL tree is: $\frac{N}{2} \log N + C$, where C is the total number of required comparisons for inserting $\frac{N}{2}$ items.

3. In a full AVL tree the total search cost for accessing all the items is:

$$\sum_{i=0}^{\log(N+1)-1} 2^i(i+1)$$

and the average search cost is:

$$\frac{\sum_{i=0}^{\log(N+1)-1} 2^i(i+1)}{N}$$

4. For *skip lists*, our analysis shows that, the average number of comparisons increases if the items which are in the front of the list, reach the upper levels. But, the average number of comparisons decreases if the items which are in the middle and end of the skip list reach to upper levels.

5. Our theoretical and experimental results conclude that, the insertion cost of *skip list* does not depend only on the number of levels, and the distribution of items in the levels. It also depends on the order in which the data items are inserted into the skip lists. The descending order insertion is faster and require less comparisons than ascending and random insertion. The worst case insertion cost for descending insertion is $1 + \log N$.

6. The interesting point for the *bottom-up splay tree* is that, the *Zig-zag* rotations make the tree to be better balanced. The orders which need more

Zig-zag rotations tend to yield more balanced trees. The number of *Zig-zag* rotations increases, if the data items are inserted in random order.

7. In most cases the insertion of unsorted data items into a *top-down splay tree* yields a degenerate tree. In terms of tree structure, bottom-up splay trees yield better balanced trees than top-down splay trees when the same sequence of items are inserted. The reason is the appearance of *Zig-zag* rotations in the bottom-up splay tree.

8. Insertion into a top-down splay tree, using the standard *Zig-zag* approach yields better balanced *BST*s than using the simplified *Zig-zag* approach. The literature mentions that, the simplified *Zig-zag* approach simplifies the code but increases the number of iterations, whereas the standard *Zig-zag* approach complicates the code and decreases the iterations (see Weiss, 1999, p. 619). This is true but, the simplified *Zig-zag* approach yields more degenerate trees.

9. Accessing all the items of a degenerate bottom-up or top-down splay tree sequentially from the root to the leaf requires $2N-1$ comparisons. Therefore, the average number of comparison is

$$\frac{2N-1}{N} \approx 2.$$

The average number of rotations for accessing all the items of a degenerate bottom-up splay tree is

$$\frac{N-1}{N} \approx 1.$$

10. Accessing all the items of a degenerate bottom-up splay tree from the leaf to the root of the tree requires $N + R$ comparisons, where N is the number of items and R is the total number of rotations. The total number of rotations is $C - N$ where C is the total number of comparisons, and N is the number of items. Accessing all the items of a degenerate top-down splay tree, from the leaf to the root of the tree need from $2N$ to $5.32N$ comparisons, where $2 \leq N \leq 2^{20}$. The average number of comparisons is from 2 to 5.32 comparisons.

11. Random access to all items of a degenerate bottom-up splay tree requires $N + R$ comparisons, where N is the number of items and R is the number of rotations. The total number of rotations is $C - N$ where C is the number of comparisons and N is the number of items.

12. The result of random access, to all items of a degenerate bottom-up splay tree alters the structure of the tree to some more balanced *BST*. Our analysis shows that if the leaf items are accessed first, and the root is accessed at the end, the resultant tree will be a full *BST*. But, if the root is accessed at the beginning or middle, then the resultant tree will be degenerated or some kind of unbalanced tree.

13. The interesting point is that, the access orders which yield full and balanced bottom-up splay trees, also yield full and balanced bottom-up splay trees, when all the items of a degenerate bottom-up splay tree are accessed in the same order. All orders which yields full and balanced *BST*s have the root elements at the end of the order.

In summary, the thesis analyzed theoretically and practically, by means of manual and computational elaboration, the performance of splay trees and skip lists. We have compared the search and update operations of *AVL* trees, skip lists and splay trees. In our theoretical analysis the search and update operations of *AVL* trees, skip lists and splay trees were manually analyzed. The experiments practically analyzed the search and update performance of *AVL* trees, splay trees and skip lists. The required number of comparisons and rotations for search and update operations of these data structures are programmatically calculated, tabulated, and the results are compared with the theory.

We have compared the search and update performance of *AVL* trees, splay trees and skip lists. Based on the comparison we identified the cases where each data structure is desired or undesired. The exact random insertion and random search cost of skip list and splay trees still need further investigation.

Appendix A

Simple Rotate-to-root Splaying Can Take $\Theta(N^2)$ Time

There are arbitrarily long pathological sequences for which the amortized time is $\Theta(N)$. Suppose the keys in the tree are from the set $[1..N]$. Inserting these nodes in sequential order from 1 to N, is reasonably fast because the new node always gets attached to the previous root, so the average cost for a sequential insertion is $\Theta(1)$. The tree tree built initially possesses only left children as illustrated in Figure A.1(a). Figure A.1(b) Illustrates the result of rotating 1 once towards the

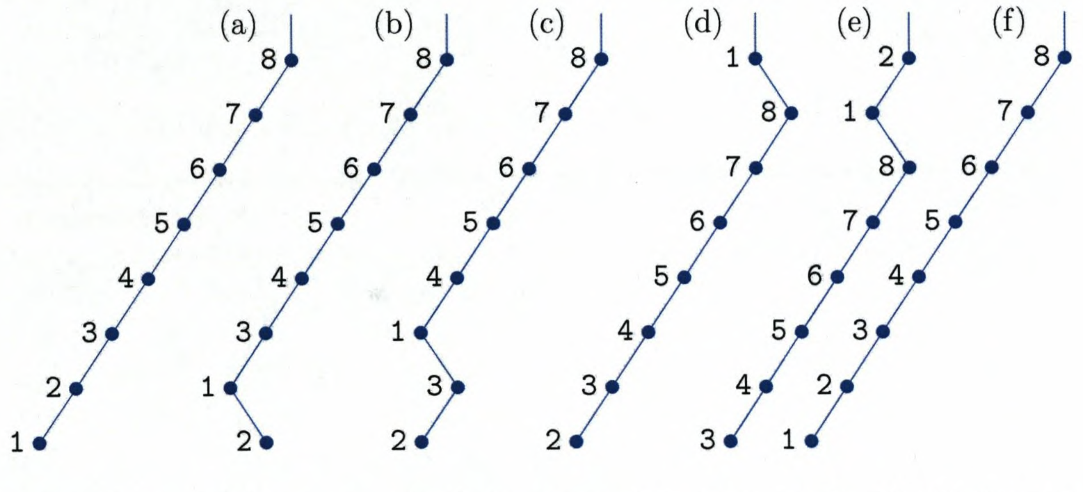


Figure A.1: (a) A skew tree built by sequentially inserting $[1..N]$. (b) The tree after 1 has been rotated once. (c) The tree after 1 has been rotated twice. (d) The final tree after 1 has been rotated $N-1$ times.(e) The resultant tree after 2 has been rotated to the root. (f) The final tree after each node has been accessed sequentially is the same as the initial tree.

root. Figure A.1(c) shows the result after accessing. Figure A.1(d) is the resulting tree after 1 has been accessed twice. In Figure A.1(e) 1 has reached its final position at the root and Figure A.1(f) shows that after sequentially accessing all the nodes once the resultant tree is exactly the same as the initial tree. Each of these accesses starts at the root and finds the node to be rotated-to-root using N comparisons, and then rotates the node to the root in $N-1$ rotations. Each such access thus costs $N(N-1)/2$ giving an average access time of $\Theta(N^2)$. Repeating accesses in the sequence $1, 2, \ldots, N$ an arbitrary number of times will cost an average of $\Theta(N^2)$ probes.

This makes it clear that trees using simple rotation-to-root, much like ordinary unbalanced BSTs, are prone to a worst case performance of $\Theta(N)$.

Appendix B

The Main Timing Code

The code used to time our skip list algorithms is given below. The code for running the other algorithms is similar. In this example we use the timing method `System.currentTimeMillis()`.

We later used the `System.nanoTime()` method which returns the time in nanoseconds. The `System.currentTimeMillis()`, gives the same results but returns milliseconds. Using `nanoTime()` gives the same values as `currentTimeMillis()` but gives three more so called significant digits. The Java SDK documentation states that the `System.nanoTime()` method provides nanosecond precision, but not necessarily nanosecond accuracy. Both methods are very crude and in order to attain meaningful times the data structures need to be run repeatedly—30 times—using up to a million data items. No guarantees are made about how frequently values change. Although the Java SDK documentation states that the `System.nanoTime()` method is guaranteed to be the most accurate timer available from the system, (Miller, 2005),

This method can only be used to measure elapsed time and is not related to any other notion of system or wall-clock time.

```
import java.text.*;
public class trySkiplist {
public static int logN = 30;
public static int N = 1<<logN; // (1<<23) + (1<<22);
public static final int numberOfClasses = 100;
public static void main(String args[]) {
node [] randomNode;
int classNo, classes[] = new int [numberOfClasses+1];
int repetition, repetitions = 10;
boolean debug = true;
final int random = 0, ascending = 1, descending = 2;
String modality[] = {"random", "ascending", "descending"};
DecimalFormat twoD = new DecimalFormat("0.00");
DecimalFormat fourD = new DecimalFormat("0.0000");
DecimalFormat eightD = new DecimalFormat("0.00000000");

   for (logN = 1; logN < 1<<30; logN++) {
System.out.println("log n = " + logN);
   int mode;
double maxTime = 0.0, minTime = 1<<30;
   double t[] = new double [repetitions + 1];

      N = 1<<logN;
      System.out.println("\n\n==================================================================\n"
 +  "java " + args[0] + " running " + N + " public nodes.");
      System.out.println("_______________________________");
   randomNode = new node [N+1];

      for (mode = descending; mode >= random; mode--) {
      double runTime = 0f, runTime2 = 0f, time;
   runTime = 0.0; runTime2 = 0.0;
maxTime = 0.0; minTime = 1<<20;

for (int i = 1; i <= numberOfClasses; i++) {
classes[i] = 0;
            }

         for (repetition = 1; repetition<=repetitions; repetition++) {
         skiplist S = new skiplist();
         double start, finish, duration;

S.setupListOfRandomNodes(mode, randomNode, N);

for (int i = 1; i <= numberOfClasses; i++) {
classes[i] = 0;
               }

         // start  ---------------------------------------------------------------------
//
          start = System.currentTimeMillis ();
            for (int i = 1; i <= N;  i++){
// System.out.println("insert node[" + i + "] = " + randomNode[i]);
               S.insert(randomNode[i]);
               }
            finish = System.currentTimeMillis ();

         // finish ---------------------------------------------------------------------

            //runTime = (finish - start)/1000;
            time = (double)(finish - start);
 t[repetition] = time;
            runTime += time;
            runTime2 += (time*time);
            if (time < minTime) minTime = time;
 if (time > maxTime) maxTime = time;
            // System.out.println("Skiplist S:\n" + S);

            System.out.println("Repetition " +repetition+ "  Run time = " +time+ "ms. for N = " +N
                              + " nodes. entered in " + modality[mode] + " order.");

            }
       double aveRuntime = runTime/repetitions;
       double stdDeviation = Math.sqrt(runTime2 - repetitions*aveRuntime*aveRuntime)/(repetitions-1);
 // Display a few nodes for visual verification
          System.out.println("_________________________________");
        System.out.printf("\n\nNodes in %s order\n", modality[mode]);
 int upto = 10; if (upto > N) upto = N;
 for (int i = 1; i <= upto; i ++){
 System.out.printf("%d", randomNode[i].key);
 if (i % upto != 0)
 System.out.printf(",  ");
 else
 System.out.printf("\n");
    }
          System.out.println("________________________________\n"
                            //+ "\nTotal time             = " + runTime
                            //+ "                 "
                            //+ "  per operation        = " + fourD.format(1000*runTime/N) + " mus."
    + "\nMinimum time           = " + minTime
                            + "                 "
                            + "  per operation        = " + fourD.format(1000*minTime/N) + " mus."
                            + "\nAverage time per run = " + twoD.format(aveRuntime) + " "
                                                        + '\u00B1' + " "
                                                        + fourD.format(stdDeviation) + "ms"
                            + "  per operation        = " + fourD.format(1000*aveRuntime/N) + " mus."
    + "\nMaximum time           = " + maxTime
                            + "                 "
                            + "  per operation        = " + fourD.format(1000*maxTime/N) + " mus."
                            + "\nStandard deviation   = " + fourD.format(stdDeviation)
                            + "\nN                    = " + N
                            + "\nRepetitions          = " + repetitions
                            + "\nAverage time         = " + fourD.format(1000*aveRuntime/N) + " mus."
                            + "\nper operation (aveT)\n"
                            + "\naveT/logN in mus.    = " + fourD.format(1000*aveRuntime/N/(double)logN)
                            + "\n________________________________\n\n");

  for (int i = 1; i <= repetitions; i++) {
  classNo = (int)((t[i] - minTime)/(maxTime - minTime)*numberOfClasses);
  classes[classNo]++;
                  }

  System.out.println();
  for (int i = 1; i <= numberOfClasses; i++) {
  double X = i/((double)numberOfClasses)*(maxTime - minTime) + minTime;
     if (classes[i] > 0)
  System.out.printf("%6.0f  %3d\n", X, classes[i]);
                  }

            }
         }
      }
   }
```

Appendix C

Code for rotation in a splay tree

```
rotate(tree x) {
// pre: x != null
tree y == x.parent;
if (y != null) // x is not the root
   if (x.key < y.key) { // right rotation
      y.left = x.right;
      x.right = y;
      }
   else { // x is right child:  left rotation
      y.right = x.left;
      x.left = y;
      }
   x.parent = y.parent;
   y.parent = x;
   if ((x.parent).key > x.key)
      (x.parent).left = x;
   else
      (x.parent).right = x;
      }

void splay(tree x) {
   if (x.parent != null) // y -- x
      if ((x.parent).parent != null) { // z -- y -- x
         if (x.key < (x.parent).key)
            if ((x.parent).key < ((x.parent).parent).key)
               rotate(x.parent);
            else // left-right: zig-zag
               rotate(x);
         else // right
            if ((x.parent).key < ((x.parent).parent).key)
               rotate(x.parent);
            else // right-left: zig-gig
               rotate(x);
         rotate(x);
         }
   }
```

Appendix D

Code for Skip Lists

```
public class skiplist {
public static final int random = 0, ascending = 1, descending =2;
public boolean debug = false;
public static int length = 0;
public static int N = 1<<20;
public static int i = 0;
node s = null;
int height = 0;
final int infinity = 2147483647;// 999; // 2147483647;

public skiplist() {
   s = new node(-infinity, "-infinity");    // {1} all pointers are set to null
   //  s.below = null;
   //  s.before = null;
   //  s.above = null;
node r = new node(infinity, "+infinity"); // {2} all pointers are set to null
   //  r.below = nulll;
   //  r.prev = null;                      // r.prev will later be set to s
   //  r.above = null;
   //  r.next = null;
/*
   s.setNext(r);                           // s.next = r;
   r.setPrev(s);                           // r.prev = s;
   s.setPrev(null);                        // s.prev = null;  // redundant
   */
   r = insertAfterAbove(s, null, r);         // {1} ----> {2}
node t = new node(-infinity, "-infinity");   // {3}
   t = insertAfterAbove(null, s, t);         // {3} \\\\V {1}
   s = t;
   t = new node(infinity, "+infinity");      // {4}
/*   t = insertAfterAbove(s, (s.getBelow()).getNext(), t);*/
   t = insertAfterAbove(s, r, t);             // {3} ----> {4}
   }

public boolean isEmpty(){
   return (length == 0);
   }

public boolean isFull(){
   return (length == N);
   }

/*
public node find (long k){
   p = this.search(long k);
   if (p.getKey() == k)
      return p;
   else
      return null;
   }*/

public node search(long k){
node p = s;
   if (p == null) {
      }
   else {
      if (p.below == null) {
         }
      while (p.below != null) {
         p = p.below;
                 while (p.next.key < k) {
                    p = p.next;
                    }
                 }
              }
           return p;
           }

public node new_search(long k){
node p = s;
   while (p.below != null) {
      p = p.below;
         while (p.next.key < k) {
            if (debug)
System.out.println("Going East from p == " + p);
            p = p.next;
            }
         }
   return p;
   }

public node insertAfterAbove(node p, node q, node r) {

   r.prev = p;

   if (p == null) {
      // r.setBelow(q);
      //    System.out.println("r.setBelow(q)");
      }
   else {
node psnext = p.next;
   if (psnext != null) {
      psnext.prev = r;
}
r.next = psnext;
      // r.setNext(p.getNext());
      //    System.out.println("r.setNext(p.getNext())");
      p.next = r;
      // r.setBelow(q);
      //    System.out.println("r.setBelow(q)");
      }
if (q != null) {
      q.above = r;
      }

   r.below = q;
   return r;
   }

public node insert(node item){
// Code based on Goodrich and Tamassia's (2005)
// Code Fragment 9.11, p. 402.
node e = null;
int i;
node p = search(item.key);
// if a null is returned then item is not in the list
node q = this.insertAfterAbove(p, null, item);

//   e = q.element()

   i = 0;
boolean throwHeads = (Math.random() >= 0.5);

   while (throwHeads) { // heads \in [0.5,1) or tails \in [0,0.5)
      i++;
      if (i >= height) {

         height++;
         node t = s.next;

         //     [-inf, --]------------------>[+inf, null]
         s = this.insertAfterAbove(null, s, new node (-infinity, "-inf-"));
node dummy;
         dummy = this.insertAfterAbove(s, t, new node (+infinity, "+inf+"));

         }

      while (p.above == null) {
         p = p.prev;
if (p == null)
System.out.println("Error in insert: p has become null");
         }

      p = p.above;

      e = new node(item.key, "...");

      q = insertAfterAbove(p, q, e);

      throwHeads = Math.random() >= 0.5;
      }
   length++;
   return e;
   }

public void display(){
   String disp = "Display skip list";

   System.out.println(disp);
   }

public long remove(){
// Dummy function
   return 0L;
   }

public double log2(double x){
   return Math.log(x)/Math.log(2);
   }

public int size(){
   return length;
   }

public void setupListOfRandomNodes (int mode, node randomNode[], int N){
int randomKey [] = new int [N+1];
// int [] randomKey = {4, 3, 12, 7, 9, 8, 1, 5, 10, 2, 11, 6};

   for (int i =1; i <= N; i++) {
      if (mode == descending)
randomKey[i] = N - i + 1;  // descending keys
else
randomKey[i] = i;  // random and ascending keys
      }

if (mode == random) {
      for (int i =1; i <= N; i++) {
      int r = (int) (Math.random() * N  + 1);
      int keep = randomKey[i];
         randomKey[i] = randomKey[r];
         randomKey[r] = keep;
         }
      }

int lineLength = 1;
   for (int i =1; i <= N; i++) {
      randomNode[i] = new node (randomKey[i], "...");
      // System.out.print("< " + randomKey[i] + " >  ");
      // if (lineLength++ % 11 == 0)
      //    System.out.println();
      }
   }

public String toString() {
   return "<< " + s + " >>";
   }

}
```

Appendix E

Our Manual Observations

In order to gain an understanding of the data structures we hand diagrammed many trees and skiplists considering insertion, search, and deletion operations. For each operation, different numbers of items in ascending, descending, and random order were considered.

E.1 *AVL* Trees

The insertion, search, and deletion operations of *AVL* trees were manually analyzed. For the insertion operation, we entered different number of items—from 1 to 32 items—in ascending, descending, and random orders into *AVL* tree. For each insertion we drew the corresponding *AVL* tree on paper. Then for each tree shape we manually counted the required number of comparisons and rotations. If all the cases yield the same result then the result was considered as our "theoretical" result. Otherwise, the process was continued by entering a new number of items to find a generic result. Similarly, the search and delete operations were manually analyzed by considering *AVL* trees with different numbers of items. For each case the required number of comparisons and rotations were manually counted.

E.2 Skip Lists

The search and update operations of the skip list were also analyzed similarly by entering different numbers of items into skip lists and the corresponding skip list was drawn on paper. For the insertion operation we manually drew many skip lists for different numbers of items considering all the possible shapes for the skip list. First we entered 3 items into a skip lists and we considered 36 different shapes for the skip list.

Then we counted the total and average number of comparisons for each shape. We derived our results from all the possible shapes. In the same way we entered 4 items into skip lists and considered all the possible shapes. We again calculated the total and average number of comparisons for each shape. Then in the same way we entered 7 items into skip lists and we considered all the possible shapes. We again calculated the total and average number of comparisons for each shape. Then we compared the results from arising from 3, 4 and 7 items. Using this manual analysis for skip lists we could draw some conclusions that are sometimes referred to in the thesis as our "theoretical" results or our "observations." Similarly, search and delete operations of skip lists were manually analyzed by searching and deleting of items from different skip lists. For each operation, the conclusions are based on considering all the possible cases.

E.3 Splay Trees

For splay trees we entered different numbers of items, again 3, 4, and 7 items, in different orders, and counted the total number of comparisons and rotations. We separately analyzed the bottom-up and top-down splay trees. First we entered 3 items in different orders into a bottom-up splay tree. We counted the total number of comparisons and rotations for each order manually. Then we entered 4 items in different orders into a bottom-up splay tree and we counted the total number of comparisons and rotations. We considered all the orders for these 4 items.

The result was the same as for inserting 3 items into a bottom-up splay tree. Then we entered 7 items in different orders into a bottom-up splay tree and we counted the total number of comparisons and rotations. We considered all the orders for these 7 items. The result was the same as for inserting 3 items and 4 items into a bottom-up splay tree. For the search operations of a bottom-up splay tree we considered a different number of items then we accessed all the items in different orders. We accessed all the items from the root to the leaf and from the leaf to the root. Then we accessed all the items of a bottom-up splay tree in different orders. Then we accessed all the items of a degenerated bottom-up splay tree with 7 items in different orders and we counted the total number of comparisons and rotations for each sequence.

Similarly, we entered 3, 4, and 7 items in different orders into a top-down splay tree. We counted the total number of comparisons and rotations for each order, by drawing the corresponding top-down splay tree on the paper.

We illustrate one the pages[4] generated in our manual elaboration of splay trees in Figure E.1.

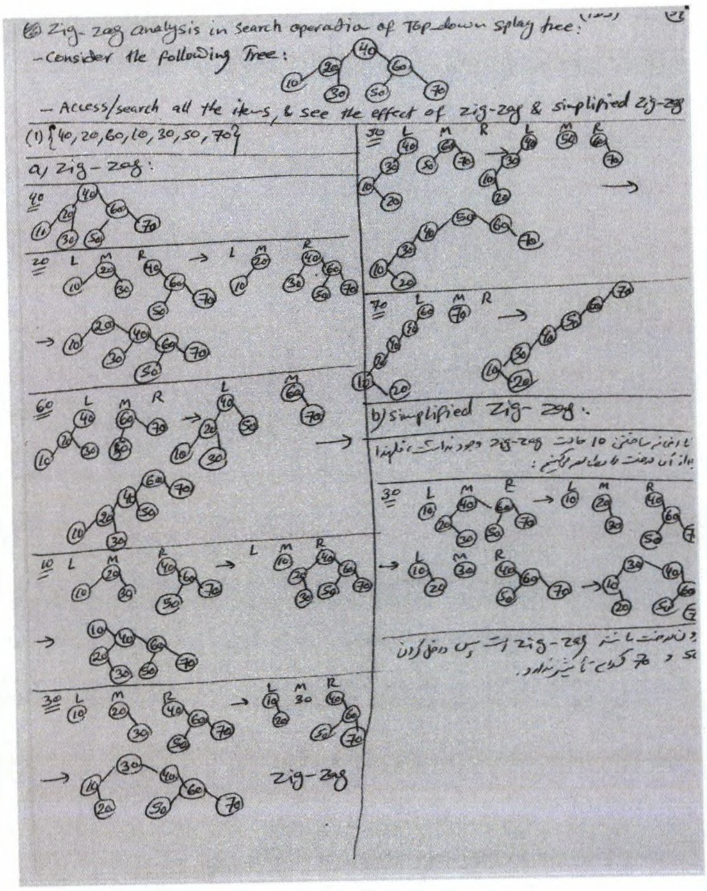

Zig-zag analysis in search operation of Top-down splay tree:
- Consider the following Tree:
- Access/search all the items, & see the effect of zig-zag & simplified zig-zag
(1) {40, 20, 60, 10, 30, 50, 70}
a) zig-zag:
b) simplified zig-zag:
zig-zag

Figure E.1: A page from the manually diagrammed *Zig-zag* analysis.

The final result was derived by considering different numbers of items, and for each number of items we test *all* the possible orders. For the search operations of a top-down splay tree we considered different numbers of items and accessed all the items in different orders. We accessed all the items from the root to the leaf and from the leaf to the root. Then we accessed all the items of a top-down splay tree in random order. We counted the total number of comparisons and rotations for each order and derived our final result considering different number of items and different orders for each number of items.

[4]We diagrammed about 200 such pages.

Bibliography

Georgiĭ Maksimovich Adel'son-Vel'skiĭ and Evgeniĭ Mikhaĭlovich Landis. An algorithm for the organization of information. *Soviet Math. Dokl.*, 3:1259–1262, 1962.

Arne Andersson. General balanced trees. *Journal of Algorithms*, 30:1–18, 1999.

Arne Andersson, Christian Icking, Rolf Klein, and Thomas Ottmann. Binary search trees of almost optimal height. *Acta Informatica*, 28, 1990.

Jean-Loup Baer and B. Schwab. A comparison of tree-balancing algorithms. *ACM*, 20(5):1–9, 1977.

Rudolf Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.

Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

James Michael Cannady. Balancing methods for binary search trees. *CACM*, pages 1–6, 1990.

Chris Wang Chengwen, Jonathan Derrynerry, and Daniel Dominic Sleator. $O(\log\log n)$-Competitive dynamic binary search trees. In *SODA'06*, pages 374–383, Miami, FL, 2006.

Thomas H Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1991.

Helen Custer. *Inside Windows NT*. Microcomputer Applications, Suisun City, CA, 1993.

Erik D. Demaine, Dion Harmon, John Iacono, and Mihail Pătraşcu. Dynamic optimality—almost. *SIAM Journal on Computing*, 37:240–251, May 2007.

Peter J. Denning. The working set model for program behavior. *CACM*, 11(5): 323–333, 1968.

Kurt W. Deschler and Elke A. Rundensteiner. Sustaining high volume inserts into large data pages. *ACM*, pages 1–8, 2001.

Caxton C. Foster. Information storage and retrieval using AVL tree. *ACM*, pages 192–205, 1965.

Caxton C. Foster. A generalization of AVL trees. *CACM*, 16(8):513–517, 1973.

Jaco Geldenhuys and Brink van der Merwe. Comparing leaf and root insertion. *To be published in SACJ*, pages 1–5, 2008.

Michael T. Goodrich and Robert Tamassia. *Data Structures and Algorithms in Java*. John Wiley and Sons, 3rd edition, 2004.

Michael T. Goodrich and Robert Tamassia. *Data Structures and Algorithms in Java*. John Wiley and Sons, 4th edition, 2005.

Dennis Grinberg and Sivaramakrishnan Rajagopalan. Splay trees for data compression. In *SODA'95*, pages 522–530, Miami, FL, 2005.

Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the Nineteenth Annual Symposium on Foundations of Computer Science*, pages 8–21, 1978.

John Iacono. Alternatives to splay trees with $O(\log n)$ worst-case access times. *SIAM Journal on Computing*, 31:516–522, 2001.

Wei Jiang, Cheng Ding, and Roland Cheng. Memory access analysis and optimization approaches on splay trees. *NPC*, 81:1–6, 2004.

Arne T. Jonassen and Donald E. Knuth. A trivial algorithm whose analysis isn't. *Journal of Computer and Sciences*, pages 301–322, 1978.

Edward G. Coffman Jr., Zhen Liu, and Arif Merchant, editors. *Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS 2004*, New York, NY, 2004.

Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison Wesley, 1968.

Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison Wesley, 1973.

Davender S. Malik and Premchand S. Nair. *Data Structures Using Java: AVL Trees*. Course Technology, 2003.

Udi Manber. *Introduction to Algorithms A Creative Approach*. Addison Wesley, 1989.

Robert Miller. A toolset for the analysis of concurrent algorithms. Master's thesis, Victoria University, Wellington, New Zealand, 2005.

Girija Narlikar, Lakshman, Y. N. Tin, and Kam Ho. Tabla: A client-based scheduling algorithm for web proxy clusters, 2000.

Jürg Nievergelt and M. Edward Reingold. Binary search trees of bounded balance. In *STOC '72: Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 137–142, Denver, Colorado, United States, 1972.

Otto Nurmi and Eljas Soisalon-Soininen. Uncoupling updating and rebalancing in chromatic binary search trees. *ACM*, pages 1–7, 1991.

Ben Pfaff. Performance analysis of BSTs in system software. In Edward G. Coffman Jr., Zhen Liu, and Arif Merchant, editors, *Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS 2004*, pages 410–411, 2004.

William Pugh. Skip lists: A probabilistic alternative to balanced trees. *CACM*, 33 (6):669–676, June 1990.

Eugene S. Schwartz. A dictionary for minimum redundancy encoding. *J. ACM*, 10(4):413–439, 1963.

Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary trees. In *STOC '83: Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 235–245, New York, NY, 1983.

Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.

Thanukrishnan Srinivasan, Muthusrinivasan Nivedita, and Vasudevan Mahadevan. Efficient packet classification using splay trees. *IJCSNS*, 6(5B):28–35, May 2006.

Anestis A. Toptsis. B^{**}-tree: a data organization method for high storage utilization. In *Computing and Information, Proceedings ICCI '93*, pages 277–281, 1993.

Mark Allen Weiss. *Data Structures and Problem Solving Using Java.* Addison-Wesley, 1999.

William E. Wright. An empirical evaluation of algorithms for dynamically maintaining binary search trees. *ACM*, pages 505–511, 2006.