Exploring the Duality Between Skip Lists and Binary Search Trees

Brian C. Dean
School of Computing
Clemson University
Clemson, SC
bcdean@cs.clemson.edu

Zachary H. Jones
School of Computing
Clemson University
Clemson, SC
zjones@cs.clemson.edu

ABSTRACT

Although skip lists were introduced as an alternative to balanced binary search trees (BSTs), we show that the skip list can be interpreted as a type of randomly-balanced BST whose simplicity and elegance is arguably on par with that of today's most popular BST balancing mechanisms. In this paper, we provide a clear, concise description and analysis of the "BST" interpretation of the skip list, and compare it to similar randomized BST balancing mechanisms. In addition, we show that any rotation-based BST balancing mechanism can be implemented in a simple fashion using a skip list.

1. INTRODUCTION

Since its introduction in 1989 [12, 13], the skip list has become a popular data structure for solving the dictionary problem, both in theory and in practice, and it is generally billed as a simpler alternative to balanced binary search trees (BSTs). All fundamental dictionary operations run in $O(\log n)$ time with high probability on an n-element skip list, the same guarantee offered by treaps [2] and randomized BSTs [7].

In this paper, we show that although the skip list may have been intended as a replacement for the balanced BST, one can interpret the skip list as a BST whose randomized balancing mechanism is perhaps on par with today's most popular randomized BST variants (e.g., [2, 7]) in terms of simplicity and elegance. We also show the reverse — that any rotation-based BST balancing mechanism can be implemented using a skip list. This "duality" allows us to transfer results from one representation to the other with ease. For example, by transforming, say, AVL trees [1], BB[α] trees [11], or red-black trees [3, 6], we obtain a deterministic skip list with $O(\log n)$ worst-case running time guarantees. By transforming the splay tree [14], we obtain an amortized skip list that inherits all of the nice properties of splay trees, such as static optimality (note that both deterministic [10] and statically optimal [5] skip lists have already been independently derived in the literature).

That the skip list can be interpreted as a type of randomly-balanced tree is not particularly surprising, and this has certainly not escaped the attention of other authors [2, 10, 9, 8]. However, essentially every tree interpretation of the skip list in the literature seems to focus entirely on casting the skip list as a randomly-balanced multiway branching tree (e.g., a randomized B-tree [4]). Messeguer [8] calls this structure the *skip tree*. Since there are several well-known ways to represent a multiway branching search tree as a BST (e.g., replace each multiway branching node with a miniature balanced BST, or replace (first child, next sibling) with (left child, right child) pointers), it is clear that the skip list can in principle also be represented by a BST. Munro et al. state precisely this in the concluding remarks of [10], without providing further details. To the best of our knowledge, however, there does not appear to be any explicit, detailed description in the literature of a BST representation of the skip list. We show that not only can the skip list be transformed into an equivalent BST in principle, but that this gives us a simple, natural, and elegant randomized balancing mechanism that is interesting in its own right as a purely BST-based result.

The remainder of this paper is structured as follows. In Section 2, we show how to transform between a skip list, a multiway branching search tree, and a BST. In Section 3, we describe BST analogs of the the *insert* and *delete* operations on a skip list. Finally, in Section 4 we show how to convert any rotation-based BST balancing mechanism into an equivalent skip list.

2. SKIP LISTS AND WEIGHTED TREES

As shown in Figure 1, there is a natural one-to-one transformation between the skip list, a multiway branching search tree with edge weights, and a BST with edge weights. To convert a skip list into a multiway branching search tree, we collect all elements appearing at the maximum height level into a root node, and then recursively fill in the subtrees between successive keys. Weights on edges indicate parent-child height differences, and so have negative values. In order to continue the transformation all the way to a BST, we apply the standard transformation from a multiway branching tree to a BST that replaces (first child, next sibling) pointers with (left child, right child) pointers.

We root both the multiway search tree and the BST from the right of a "dummy" root node, r, that corresponds to

ACMSE 2007, March 23-24, 2007, Winston-Salem, North Carolina, USA

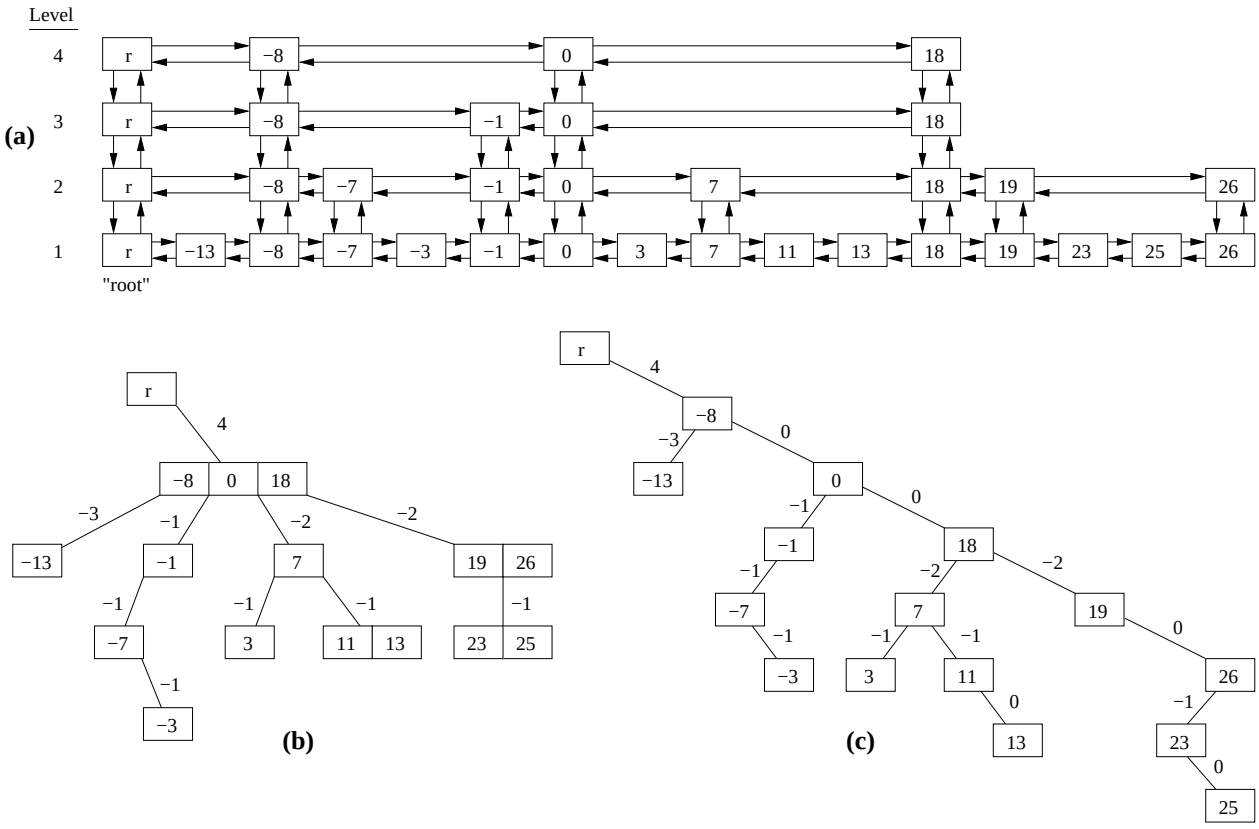

Figure 1: A sample skip list (a), shown as a weighted multiway branching search tree (b) and a weighted BST (c). In (b) and (c), the weight of an edge corresponds to a height difference in (a), except for the special edge connecting to the root, whose weight corresponds to the total height of the skip list in (a).

the dummy root node of the skip list. The weight of the edge connecting to the root is special — it specifies the height of the skip list, H, and it is the only edge in the tree having positive weight. All other edges have negative, or in the case of the BST, nonpositive weight. Note that we use 1-based indexing for heights, so the lowest level in a skip list has height 1. This will simplify our code slightly in the case of the BST, since it allows us more easily to distinguish zero-weight edges corresponding to zero height differences from the special edge connecting to the root. In either the multiway search tree or the BST, we can compute the skip list height $h(x)$ of any element x by summing up the edge weights on the path from x to the root.

Let us focus now on the BST representation of our skip list. Note that the standard method for searching in a skip list (follow right pointers until we would overshoot the element we seek, then move down instead) corresponds exactly to the standard BST search procedure. Since it is well known that search in a skip list runs in $O(\log n)$ time with high probability, we therefore immediately obtain an $O(\log n)$ high probability bound on the height of our equivalent BST as well as the running time of its search operation. Since our insertion and deletion procedures in the BST will correspond exactly to their analogs on the skip list, we will also be able to obtain an $O(\log n)$ high probability bound on their running times without the need for any additional analysis.

There is an inherent asymmetry in the BST generated by our transformation, since only right edges can have zero weight. Later, we will relax this restriction and consider what we call a *symmetric* BST that allows zero weights on left and right edges. The symmetric BST transforms into a somewhat non-proper skip list in which an element can belong to a particular level without actually being linked into the list at that level. As a result, we must be somewhat cautious when trying to carry over performance bounds from the skip list directly to the symmetric BST. However, we can still show $O(\log n)$ high probability bounds in this case by trivially modifying the standard skip list analysis in an appropriate fashion. Details are given in the appendix.

3. INSERTION AND DELETION IN THE BST ANALOG OF A SKIP LIST

Let us examine how operations on a skip list translate naturally into operations on the analogous BST. Both our insertion and deletion procedures on a BST will behave *exactly* as their skip list counterparts (according to the transformation above).

We begin with insertion. Recall that the standard insertion algorithm on a skip list inserts an element in its proper location in the level-1 list, then repeatedly (as long as a fair coin flip comes up heads) "promotes" the element by inserting it into higher levels. In the BST representation, we

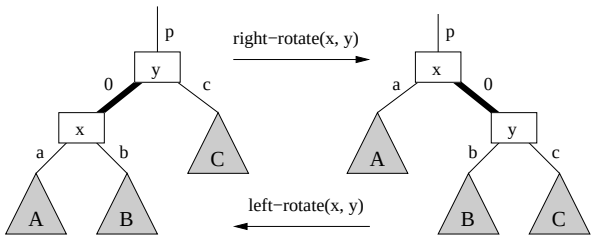

Figure 2: Local edge weights are not affected when rotating along an edge of zero weight. If $w(x, y)$ were non-zero, then we would need to adjust a and c accordingly after the rotation to preserve the skip list height $h(e)$ for elements $e \in A \cup C$.

follow the exact same approach: first insert the element at a leaf using the standard BST insertion algorithm, then repeatedly (as long as a fair coin flip comes up heads) promote the element.

As it turns out, when we view promotion of an element in the context of a BST, we end up with a very simple and natural procedure based on BST rotation. Consider the promotion of an element x with parent p, left child l, and right child r, and let $w(x, y)$ denote the weight of the edge from x to y in the BST. To reflect the fact that we move x one level higher in the skip list, we increment $w(x, p)$ while decrementing $w(x, l)$ and $w(x, r)$. If $w(x, p)$ happens to be zero initially, however, we repeatedly rotate x with its parent until $w(x, p)$ becomes non-zero, then we perform the increments and decrements. We can express the entire insertion operation in pseudocode as follows.

INSERT(T, x):

```
1  Insert x as a leaf in T (standard BST insert)
2  Set w(x, parent(x)) = H − h(parent(x))
3  while Random(0,1) = 0
4      Promote(x)
5  If w(x, parent(x)) = 0 and x is a left child
6      Rotate x with parent(x)
```

PROMOTE(x):

```
1  while w(x, parent(x)) = 0
2      Rotate x with parent(x)
3  Increment w(x, parent(x))
4  Decrement w(x, lchild(x)) and w(x, rchild(x))
```

Note that the while loop in the promotion routine is guaranteed to terminate before x reaches the root as a result of the positive weight assigned to the right edge leaving the root. Lines 5-6 in the *Insert* routine exist solely to ensure that we do not create any zero-weight left edges; in the case of a symmetric BST, these lines can be omitted. Finally, note that we only perform rotations along edges having zero weight (and this will be true for deletion as well). This is important, since it allows us to leave nearby edge weights unchanged, as shown in Figure 2.

Consider now the deletion of some element x with parent p. In a skip list, we delete x by unlinking it from every level in which it exists. As it turns out, the analog of this operation on a BST is again remarkably simple. If x has only 1 child c, we delete it simply by replacing x with c (adding $w(x, c)$ into $w(x, p)$, since the edges (x, c) and (x, p) are now

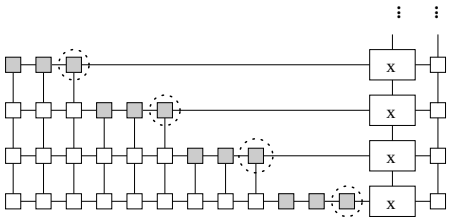

Figure 3: Elements affected by skip list deletion of x (circled) versus elements affected by BST deletion of x (shaded).

effectively merged together). If x has two children l and r, we repeatedly *demote* x until it has only one child, then delete it as before. Demotion is a symmetric operation to promotion, where we decrement $w(x, p)$ while incrementing $w(x, l)$ and $w(x, r)$. However, if one of $w(x, l)$ or $w(x, r)$ is zero (recall that $w(x, l)$ can only be zero in the symmetric case), we repeatedly rotate x downward with a child to which it is connected with a zero-weight edge. Pseudocode for the entire delete operation is as follows.

DELETE(x):

```
1  If x has no children, simply delete x
2  while x has two children
3      Demote(x)
4  Let c be the single child of x
5  w(x, parent(x)) = w(x, parent(x)) + w(x, c)
6  Replace x with c.
```

DEMOTE(x):

```
1  if w(x, rchild(x)) = 0
2      Rotate x with rchild(x)
3  Decrement w(x, parent(x))
4  Increment w(x, lchild(x)) and w(x, rchild(x))
5  while w(x, lchild(x)) = 0
6      Rotate x with lchild(x)
```

In the symmetric case, we replace the *Demote* operation with one that allows for the possibility of zero-weight left edges.

DEMOTE-SYMMETRIC(x):

```
1  while w(x, lchild(x)) = 0
2      Rotate x with lchild(x)
3  while w(x, rchild(x)) = 0
4      Rotate x with rchild(x)
5  Decrement w(x, parent(x))
6  Increment w(x, lchild(x)) and w(x, rchild(x))
```

The running time of both *Insert* and *Delete* is $O(\log n)$ with high probability, which we can easy argue by deferring to the well-established analysis of skip lists (for a detailed analysis of the symmetric skip list, see the appendix). The *Insert* operation is the most straightforward, since it corresponds exactly to insertion on a skip list. For *Delete*, we must be slightly more careful. It is true that *Delete*(x) restructures our BST to reflect *exactly* the unique BST we would obtain from a skip list with element x deleted. However, the work required to do this is slightly more in the case of the BST than it is for a skip list. As shown in Figure

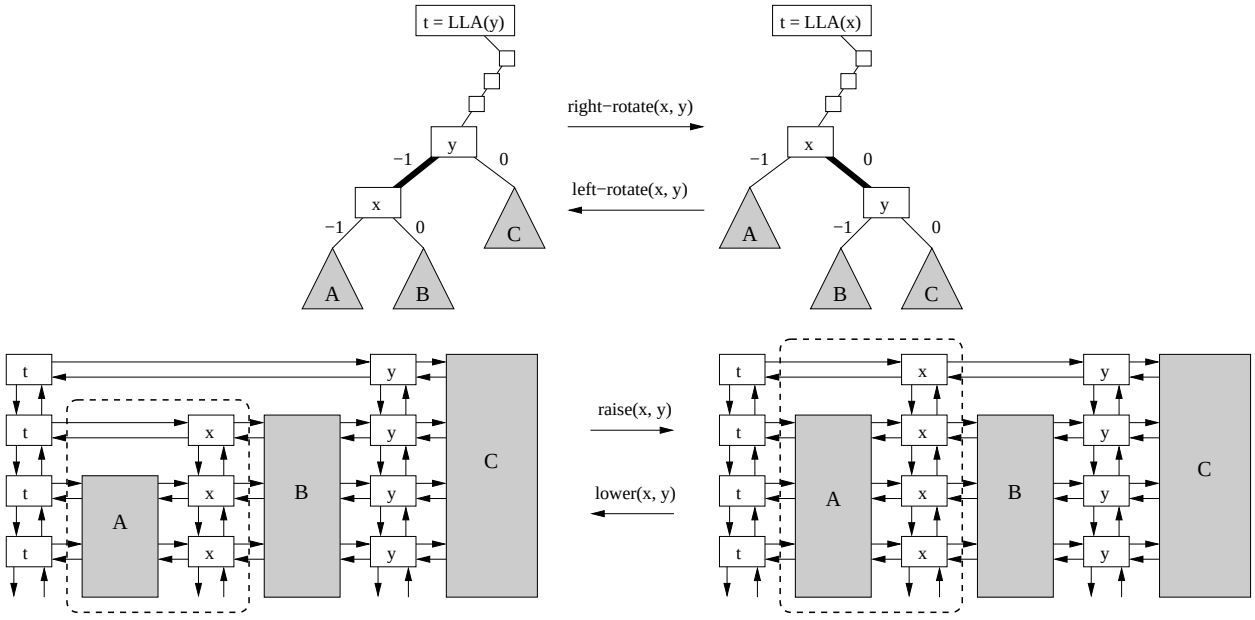

Figure 4: Illustration of how right and left rotation in a BST corresponds to raising and lowering a section of a skip list.

3, when we delete x we only need to modify the pointers attached to the circled elements to the left of x. In terms of the BST, however, *Delete*(x) ends up traversing all of the shaded elements. Fortunately, this is exactly the same set of elements we follow while searching for the predecessor of x in the skip list, which (just as with searching for any other element) requires only $O(\log n)$ time with high probability. In the symmetric case, the picture in Figure 3 becomes symmetric, with a "staircase" of shaded elements on both sides of x's column. In this case, one can bound the running time of *delete* by the time required to search for x's predecessor plus the amount of time required to search for x's successor but starting from the end of the skip list. In total, this is still $O(\log n)$ with high probability.

As we can see, implementation of the insertion and deletion operations on the BST analog of a skip list is quite straightforward, roughly comparable in complexity to heaps and randomized BSTs. In fact, one could argue that the BST interpretation is even preferable to that of a skip list in light of the fact that the BST always occupies $O(n)$ space in the worst case, rather than $O(n)$ space in expectation for the skip list.

4. THE BALANCED BST AS A SKIP LIST

Consider any BST balancing mechanism M based solely on rotations; this is sufficiently broad to cover almost every popular BST balancing mechanism. In this section, we show that M can be implemented in a simple and natural fashion in terms of a skip list. To do this, we use the same one-to-one transformation between skip lists and weighted BSTs described in the preceding section. Since this transformation requires an edge-weighted BST and we are starting with an unweighted BST, we define the weights of all left and right edges to be -1 and 0, respectively.

Note that it is reasonably easy to navigate through the resulting skip list as if we were navigating the BST. For example, let x be an element in the BST with left child l, right child r, and parent p. We denote by $S(x)$ the highest instance of x in the skip list.

- To move from $S(x)$ to $S(r)$, take one step to the right from $S(x)$.
- To move from $S(x)$ to $S(l)$, take one step left, one step down, and one step right from $S(x)$.
- To move from $S(x)$ to $S(p)$ if x is a right child of p, take one step left from $S(x)$ (we can tell if x is a right child of p if, after stepping left from $S(x)$, we cannot step upward).
- To move from $S(x)$ to $S(p)$ if x is a left child of p, take one step left, one step up, and one step right from $S(x)$.

It is also worth noting that the transformation described above always results in a skip list of size $O(n)$, regardless of the BST we start with. We can bound the size of the skip list by counting the number of pairs of elements that are linked together, since each pair of elements will be linked at most once. Consider now an element x in our BST. The only elements to which x can end up linked after the transformation are $parent(x)$, the elements along the right spine of x's left subtree, and the elements along the left spine of x's right subtree. By "charging" each link (x, y) to whichever of x and y has larger depth in the tree, we see that each element is charged at most twice.

Consider now a rotation along some edge (x, y) in our BST. As shown in Figure 4, a right rotation corresponds to raising up the contiguous range $(t, x]$ of the skip list by one unit in height, where $t = LLA(x) = LLA(y)$ denotes the *lowest left ancestor* of x. We can ensure that $LLA(x)$ always exists by making the root of our BST the

right child of a dummy root node, just as in the previous section. Similarly, a left rotation corresponds to the operation of lowering the contiguous section $(t, x]$ in the skip list, where again $t = LLA(x)$. Therefore, we can focus our attention on implementing the skip list operations *raise*(x, y) and *lower*(x, y), to be called whenever we would be calling *right-rotate*(x, y) and *left-rotate*(x, y) in the BST.

In general, one cannot hope to *raise* and *lower* an arbitrary range of elements in a skip list in $O(1)$ time, since there are simply too many pointers to reconnect at the boundaries of the range. However, in our case we are dealing with a range $(t, x]$ for which x has the tallest column height in the range, and the height of t's column is at least as large as that of x's column. For this special case, one can achieve an implementation of *raise* and *lower* in $O(1)$ time. To do this, we implicitly modify the rightward pointers emanating from column t and column x so they are offset by one unit of height. This is done by storing each column of the skip list (say, for element x) in an array $A_x[\cdot]$, where x is augmented with an offset $\Delta(x)$ that is applied to any index used to access A_x. That is, if we wish to follow the rightward pointer from $A_x[i]$, we actually follow the rightward pointer from $A_x[i + \Delta(x)]$. As a consequence of the special structure of the range $(t, x]$ we can use the $\Delta(t)$ and $\Delta(x)$ offsets to avoid explicitly modifying any leftward pointers. For example, when moving leftward along a pointer that lands us at $A_t[i]$, we instead choose to locate ourselves at $A_t[i - \Delta(t)]$. We can therefore implement *raise*(x, y) and *lower*(x, y) in $O(1)$ time by appropriately incrementing or decrementing $\Delta(t)$ and $\Delta(x)$. In the case of *raise*, we also need to add one new rightward pointer from the top of x's column across to y (as well as the corresponding leftward pointer from y; see Figure 4); for *lower*, these pointers are deleted.

Finally, we remark that BST balancing mechanisms involving rotations as well as the *join* operation (e.g., splay trees [14], randomized BSTs [7], where we delete an element x by replacing x by the join of its two subtrees) are also generally easy to accommodate. In this case, the process of deleting x by joining together its two subtrees is equivalent to rotating x downward until it falls off the bottom of the BST, so we can simulate the *join* operation using rotations.

5. ACKNOWLEDGMENTS

The authors would like to acknowledge Seth Gilbert and Erik Demaine for helpful discussions. They also thank the anonymous reviewers of this paper for their many detailed suggestions.

6. REFERENCES

[1] G. M. Adel'son-Vel'skiĭ and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1263, 1962.

[2] C. R. Aragon and R. Seidel. Randomized search trees. *Algorithmica*, 16:464–497, 1996.

[3] R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.

[4] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[5] V. Ciriani, P. Ferragina, F. Luccio, and S. Muthukrishnan. Static optimality theorem for external memory string access. In *Proceedings of the 43rd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 219–227, 2002.

[6] L. J. Guibas and R. Sedgewick. A diochromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 8–21, 1978.

[7] C. Martínez and S. Roura. Randomized binary search trees. *Journal of the ACM*, 45(2):288–323, 1998.

[8] X. Messeguer. Skip trees, an alternative data structure to skip lists in a concurrent approach. *Informatique Theorique et Applications*, 31(3):251–269, 1997.

[9] K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms*. Prentice-Hall, 1994.

[10] J. I. Munro, T. Papadakis, and R. Sedgewick. Deterministic skip lists. In *Proceedings of the 3rd annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 367–375, 1992.

[11] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. *SIAM Journal on Computing*, 2(1):33–43, 1973.

[12] W. Pugh. A skip list cookbook. Technical Report UMIACS-TR-89-72.1, University of Maryland Institute for Advanced Computer Studies, 1989.

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

[14] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

APPENDIX

A. THE SYMMETRIC SKIP LIST

In this section we analyze the performance of the symmetric skip list, the somewhat non-proper skip list obtained by transforming a BST with zero-weight left edges. We show that a trivial modification of the standard skip list analysis allows us to prove that insertion, deletion, and search in an n-element symmetric skip list all take $O(\log n)$ time with high probability (that is, with probability at least $1 - 1/n^k$ for any constant $k \geq 1$ of our choosing). In fact, the analysis that follows applies to both standard and symmetric skip lists.

For starters, the height of an n-element (symmetric) skip list is $O(\log n)$ with high probability — this follows from a union bound over all elements, since the height of a single generic element e is $O(\log n)$ with high probability: the probability that $h(e) > k \log n$ is exactly $1/n^k$ (we always use base-2 logarithms in this paper).

In both the symmetric and standard skip lists, the running time of insertion or deletion of an element e is bounded by the time required to search for e plus the height of the skip list. Hence, we need only show that search takes $O(\log n)$ time with high probability. Furthermore, since deletion of an element e leaves a (symmetric) skip list in the same state as if e had never been inserted to begin with, it suffices to analyze the running time of *search* in a (symmetric) skip list that has been freshly built by inserting n elements.

Consider now the operation of searching for a generic element e in a (symmetric) skip list of height at most $k \log n$. To do this, we examine the search path to e, as shown in

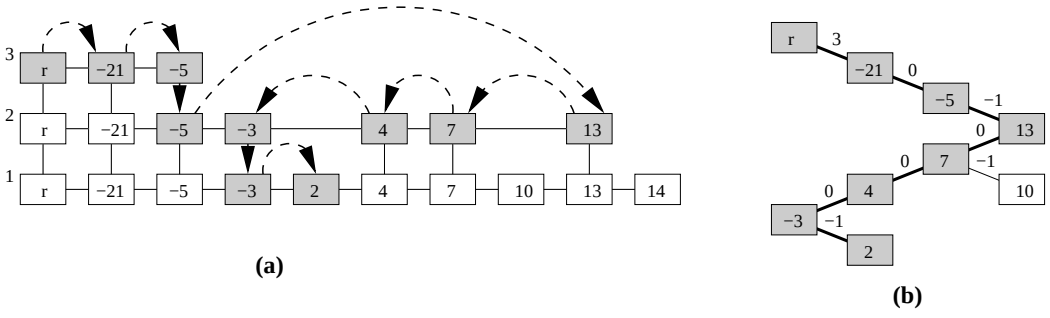

Figure 5: Searching for the element of value 2 in a "non-proper" skip list corresponding to a symmetric BST (a BST with zero-weight left edges). Note that the search path through the skip list can move left as well as right.

Figure 5. In a standard skip list, this search path only travels right and down, but in a symmetric skip list it may also move left. However, in this case the search path will never intersect itself; more formally, if we refer to a single (element, height) pair as a *node* in the skip list, then the search path through a symmetric skip list never visits the same node more than once. If we now follow the search path in reverse, we find ourselves moving upward at each node whenever possible (if the fair coin flip at that node came up heads), otherwise we move left or right (in the symmetric case). Let us focus now solely on the upward movement, since this is common to both cases.

Let the random variable X_a denote the total number of times we obtain heads when flipping a fair coins (so $E[x] = a/2$). Since we can write $X_a = Y_1 + \ldots + Y_a$ as a sum of independent "indicator" random variables Y_j (each taking the value 0 or 1 with probability 1/2), we can apply a standard variant of the Chernoff bound to bound the probability of X_a deviating significantly below its mean. For any $\beta \in [0, 1/2]$,

$$\begin{aligned} \Pr[X_a \leq \beta E[X_a]] &\leq e^{-(1-1/\beta)^2 E[X_a]/2} \\ &\leq e^{-E[X_a]/8} \\ &\leq e^{-a/16} \end{aligned}$$

Our skip list has height at most $k \log n$, so we can move upward along the reverse search path (i.e., flip heads) at most $k \log n$ times. Let T denote the length of our search path. Since each node flips an independent coin,

$$\begin{aligned} \Pr[T > 4k \log n] &\leq \Pr[X_{4k \log n} \leq k \log n] \\ &= \Pr[X_{4k \log n} \leq \frac{1}{2} E[X_{4k \log n}]] \\ &\leq e^{-(4k \log n)/16} \\ &= \frac{1}{n^{\Theta(k)}} \end{aligned}$$

which proves our desired high probability bound, since we are free to choose the constant k to be as large as we wish.