Chapter 43
Deterministic Skip Lists*

J. Ian Munro† Thomas Papadakis† Robert Sedgewick‡

Abstract

We explore techniques based on the notion of a skip list to guarantee logarithmic search, insert and delete costs. The basic idea is to insist that between any pair of elements above a given height are a small number of elements of precisely that height. The desired behaviour can be achieved by either using some extra space for pointers, or by adding the constraint that the physical sizes of the nodes be exponentially increasing. The first approach leads to simpler code, whereas the second is ideally suited to a buddy system of memory allocation. Our techniques are competitive in terms of time and space with balanced tree schemes, and, we feel, inherently simpler when taken from first principles.

1 Introduction

We address the classic problem of designing a data structure to support the operations of search, insert, delete and find_closest_value in logarithmic time in the worst case. There are several well-known solutions, the AVL tree [1], the B-tree [2] and its special case the 2-3 tree, the red-black tree [4], etc., all of which are balanced search trees, but which are rarely used for main memory applications in (non research) computational practice. Evidently, they seem too hard or complicated to implement for the "root mean square programmer". Our general question then becomes "how hard is it to understand and implement such a data structure?" or "how many neurons must fire in the head of the programmer to solve the dictionary problem in logarithmic time?" Intriguing as a lower bound for the latter form may be, we restrict ourselves to what we feel is a new upper bound.

We start with the skip list, that was recently introduced by Pugh [8] as an alternative to search trees. It is a very simple data structure; Lewis and Denenberg, for example, in their introductory data structures textbook [5] introduce the skip list in leading to the binary search tree. Unlike other data structures, the skip list is probabilistic in nature, that is, given a set of values to be stored in a skip list and the insertion sequence of these values, the shape of the skip list that will be formed is not determined, but depends on the outcomes of coin flips. It therefore makes sense to talk about the *average* cost for the search of the *mth* element in a skip list of n elements (where this average is taken over all outcomes of coin flips) and about the *average* cost for the search of the *average* element in a skip list of n elements (where this average is taken over all outcomes of coin flips and over all search keys). These costs are [8,7] $\log_{\frac{1}{p}} n + \frac{1-p}{p}\log_{\frac{1}{p}} m + \Theta(1)$ and $\frac{1}{p}\log_{\frac{1}{p}} n + \Theta(1)$ respectively, where p is the probability that the coin used to build the skip list decides to increase the height of the new node to be inserted. Similar results hold about the average insert and delete costs.

Despite the fact that the *average* insert and update costs for the skip lists are low, the *worst case* insert and update costs are technically unbounded as functions of n, and they remain linear even if the skip list is capped at some level, as suggested by Pugh [8]. Thus the balanced search tree schemes (AVL, red-black, etc.) remain the refuge of one who wants a firm logarithmic upper bound for any individual operation.

In this paper, we present several deterministic versions of the skip list that have a $\Theta(\lg n)$ worst case search cost. We begin by demonstrating our discipline in its simplest form, which requires a total of at most $2n$ pointers and has a $\Theta(\lg^2 n)$ worst case update cost. This update behaviour is improved to $\Theta(\lg n)$ with a total of at most $6n$ pointers in one version, or a total of at most $2.3n$ pointers in another. The $6n$ pointers version has the advantage of easier coding. The $2.3n$ version is ideally suited to an environment in which the buddy memory allocation system is employed (e.g. BSD Unix), since in this version, there is no increase in the storage requirement over the initial form having $\Theta(\lg^2 n)$ worst case update cost, as we are given this extra stor-

*This research was supported in part by the National Science and Engineering Research Council of Canada under grant No. A-8237, the Information Technology Research Centre of Ontario, and the National Science Foundation under Grant No. CCR-8918152

†Department of Computer Science, University of Waterloo, Waterloo, Ont N2L 3G1, Canada

‡Department of Computer Science, Princeton University, Princeton, NJ 08544, USA

age whether we want it or not.

Finally we present deterministic versions of skip lists in which updates are done in a top-down manner. Although these new skip lists are essentially simple extensions of the previous versions, their implementations are simpler, and if we become a bit cavalier with storage, we are able to achieve an implementation simpler than that of the 2-3-4 trees implemented directly or through the red-black formalism.

2 The starting point

A natural starting point is with the idea of a perfectly balanced skip list, namely one in which every kth node of height at least h is of height at least $h+1$ (for some fixed k). Although searches in balanced skip lists take logarithmic time, the insert and delete costs are prohibitive. We should therefore examine the consequences of relaxing a bit the strict requirement "*every* kth node".

Assuming that in a skip list of n elements there exists a 0th and a $(n+1)$st node of height 1 higher than the height of the skip list, we require that *between any two nodes of height h ($h > 1$) or higher, there exist either 1 or 2 nodes of height $h-1$*. We call this skip list a *1-2 skip list*. As we see from Fig. 1, there exists a one-to-one correspondence between 1-2 skip lists and 2-3 trees. We feel that, taken from first principles, the skip list view of the structure is simpler than the search tree view.

What is the number of (horizontal) pointers of a 1-2 skip list in the worst case? Clearly, the header has at most $\lfloor \lg(n+1) \rfloor$ pointers. In addition to these, there exist exactly n pointers at level 1, at most $\lfloor \frac{n-1}{2} \rfloor$ pointers at level 2, at most $\left\lfloor \frac{\lfloor \frac{n-1}{2} \rfloor - 1}{2} \right\rfloor = \lfloor \frac{n-(1+2)}{4} \rfloor$ pointers at level 3, and, in general, at most $\lfloor \frac{n-(1+2+4+\cdots+2^{i-2})}{2^{i-1}} \rfloor = \lfloor \frac{n+1}{2^{i-1}} \rfloor - 1$ pointers at level i. Therefore, the maximum number of pointers, including the pointers of the header, is $\sum_{i=1}^{\lfloor \lg(n+1) \rfloor} \lfloor \frac{n+1}{2^{i-1}} \rfloor$. This sum is well-known (see, for example, [3, pp. 113–114]), and so we get that the maximum number of pointers (including those of the header) in a 1-2 skip list of n elements is exactly

$$2n - \nu_2(n+1) + 1, \qquad \text{for all } n \geq 1,$$

where $\nu_2(n+1)$ is the numbers of 1's in the binary representation of $n+1$. Therefore, the maximum number of pointers never exceeds $2n$.

A search in a 1-2 skip list is performed in the same manner as in the probabilistic skip list. We may want to throw in an extra line of code to take advantage of the fact that after 2 horizontal steps at each level we are guaranteed to drop down a level, without looking at the 3rd element ahead at the same level. This will bring down the worst case number of key comparisons from $3\lfloor \lg(n+1) \rfloor$ to $2\lfloor \lg(n+1) \rfloor$.

An insertion in a 1-2 skip list is made by initially adding an element of height 1 in the appropriate spot. This may, of course, trigger the invalid configuration of 3 elements of height 1 in a row. This is easily rectified by letting the middle of these 3 elements grow to height 2. If this now results in 3 elements of height 2 in a row, we let the middle of these three elements grow to height 3, etc. For example, the insertion of element *15* in the skip list of Fig. 1(a) will cause *13* to grow from height 1 to height 2, that will cause *17* to grow from 2 to 3, and that will cause *17* again to grow from 3 to 4.

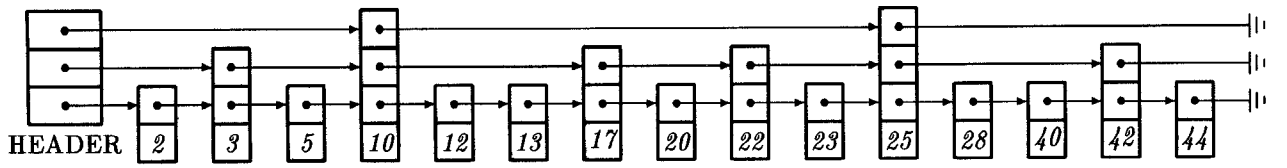
In general, the insertion algorithm described above may cause up to $\lfloor \lg(n+1) \rfloor$ elements to grow. In the 2-3 tree analogy, this is not a great problem, since each increase of an element's height takes constant time. However, in our setting, if we insist on implementing the sets of horizontal pointers of the nodes as arrays (rather than as linked lists) of pointers, then, when a node grows, we have to allocate space for a larger node, and we have to copy all pointers in and out of the old node into the new node. Therefore, the growth of a single node from height h to height $h+1$ requires the change of $\Theta(h)$ pointers. Since in the worst case all nodes of heights $1, 2, \ldots, \lfloor \lg(n+1) \rfloor$ will have to be raised, an insertion may take up to $\Theta(\lg^2 n)$ time.

A deletion of an element from a 1-2 skip list is done in a way completely analogous to the way it is done in the corresponding 2-3 tree. The deletion of *5* for example from the 1-2 skip list of Fig. 1(a), will cause *3* to shrink, and this will cause *10* to shrink and *17* to grow. In general, up to $\lfloor \lg(n+1) \rfloor$ elements may have to grow, and up to $\lfloor \lg(n+1) \rfloor$ may have to shrink, resulting thus in a $\Theta(\lg^2 n)$ worst case cost as well.

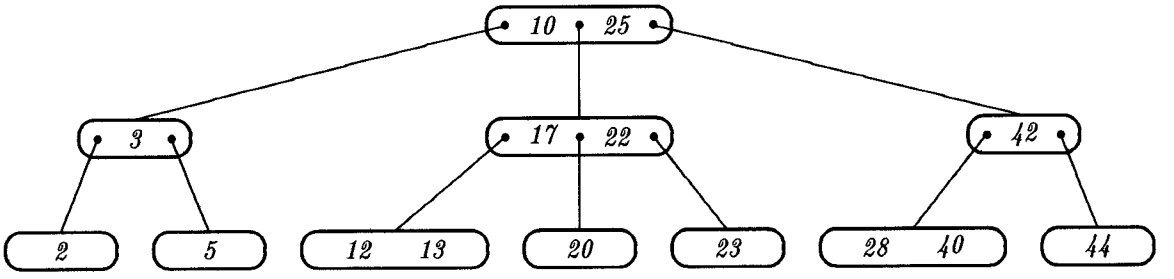
3 Achieving logarithmic worst case update costs

One solution to the problem of the $\Theta(\lg^2 n)$ worst case update costs encountered in the preceding section, would be to implement the set of horizontal pointers of each node as a linked list of pointers. This can almost triple the space requirement of our data structure; each horizontal pointer will be substituted by a right pointer, a down pointer, and a pointer to the key (or the key itself). Therefore, this linked list version of the 1-2 skip list will require up to 6n pointers in the worst case. We will return to this notion because of its simplicity.

If the above space overhead is considered unacceptable, we may implement the set of horizontal pointers of each node as an array of pointers, if we choose these



(a)



(b)

Figure 1: A 1-2 skip list and corresponding 2-3 tree

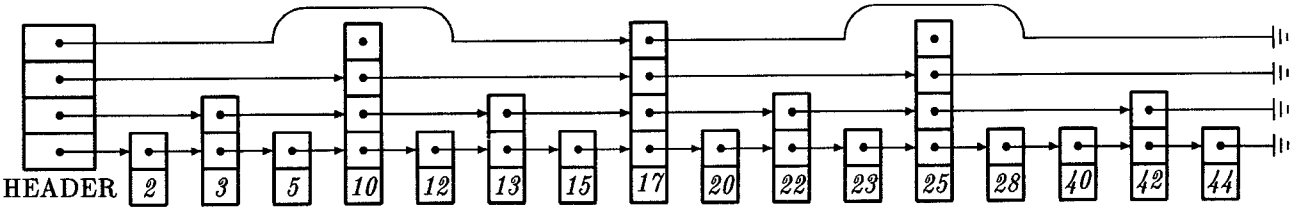


Figure 2: Array implementation of 1-2 skip list

arrays to have exponentially increasing physical heights. For simplicity, let's say that all nodes will have physical heights 2^i, for some $i = 0, 1, 2, 3, \ldots$. The logical heights of the nodes will never exceed their physical heights, and they will be equal to the heights of the nodes in the 1-2 skip list considered in the last section. We reserve the term *array implementation of the 1-2 skip list* for this variant of deterministic skip list. If we insert *15* into the skip list of Fig. 1(a), the resulting skip list under the just described scheme will be the one shown in Fig. 2.

It turns out that our new skip list does not increase the storage requirements by very much. It is not hard to see that the header has at most $2^{\lfloor \lg(\lfloor \lg(n+1) \rfloor - 1) \rfloor + 1}$ pointers, for $n \geq 3$. In addition to these, there exist exactly n pointers at level 1, at most $\lfloor \frac{n-1}{2} \rfloor$ pointers at level 2, at most $2 \left\lfloor \frac{n-(1+2)}{2^2} \right\rfloor$ pointers at levels 3 and 4, and, in general, at most $2^i \left\lfloor \frac{n-(1+2+2^2+2^{2^1-1})}{2^{2^i}} \right\rfloor = 2^i \left(\left\lfloor \frac{n+1}{2^{2^i}} \right\rfloor - 1 \right)$ pointers at levels $2^i+1, \ldots, 2^{i+1}$. Therefore, the maximum number of pointers is exactly

$$n + 1 + \sum_{i=0}^{\lfloor \lg(\lfloor \lg(n+1) \rfloor - 1) \rfloor} 2^i \left\lfloor \frac{n+1}{2^{2^i}} \right\rfloor, \qquad \text{, for all } n \geq 3,$$

and if we remove the floors and we define

$$\alpha_n \stackrel{\text{def}}{=} \begin{cases} \displaystyle\sum_{i=0}^{\lfloor \lg(\lfloor \lg(n+1) \rfloor - 1) \rfloor} 2^{i-2^i} & \text{for } n \geq 3 \\ 0 & \text{for } n = 1, 2 \end{cases}$$

we get the upper bound

$$(\alpha_n + 1)n + \alpha_n + 1, \qquad \text{for all } n \geq 1$$

for the maximum number of pointers in the array implementation of a 1-2 skip list of n elements. The sum defining α_n can be computed fast to a high accuracy, since it has only a few terms which grow very fast; we have $\alpha_3 = \cdots = \alpha_6 = .5$, $\alpha_7 = \cdots \alpha_{30} = 1$, $\alpha_{31} = \cdots \alpha_{510} = 1.25$, and $\alpha_n \simeq 1.281$ for $n \geq 511$. Hence, we never allocate more than $2.282n$ pointers.

When, doing an insertion, a node has to grow, we use the higher level pointer field, if one exists. If not, a new node has to be created, and all pointers in and out of the old node have to be copied to the new node. At this point, we achieve the $\Theta(\lg n)$ insertion cost. Consider the 2^i $(i > 1)$ nodes of logical heights $2^i+1, \ldots, 2^{i+1}$. All of them are of physical height 2^{i+1}. When the heights of all of them have to be increased by 1, only the node of logical height 2^{i+1} has to be copied into a new node; the remaining nodes will merely increase their logical heights, and their physical heights allow them to do so in constant time. Hence, the time required for an insertion, is proportional to the sum of the heights of the nodes that will be copied, that is, $\sum_{i=0}^{\lfloor \lg(\lfloor \lg(n+1) \rfloor - 1) \rfloor + 1} 2^i = \Theta(\lg n)$.

To delete an element, we apply the same technique, achieving thus a worst case $\Theta(\lg n)$ cost as well.

Having achieved logarithmic worst case update costs with the array implementation of the 1-2 skip list, which uses only $2.282n$ pointers, we might be wondering what are the advantages of the linked list implementation of the 1-2 skip list, which also achieves logarithmic worst case update costs, but which uses $6n$ pointers. The answer is *code simplicity*. In the linked list implementation, in order to raise or lower a skip list node we only have to change a few pointers. Of course, we may have to do that for all $\lfloor \lg(n+1) \rfloor$ levels, but any reasonable coder will include this piece of code in the insert/delete routines. On the other hand, in the array implementation, in order to change the height of a skip list node, we have to initiate a search for the key in that node, so that we can splice at the proper place and insert the new node. This will more likely be a separate procedure (admittedly, similar to the search routine), adding a time overhead when invoked.

We may now observe that the array implementation of the 1-2 skip list is serendipitous in the programming environment in which many of us find ourselves into, namely programming in C under Unix. In most of Berkeley-based versions of Unix (e.g. 4.3BSD), memory blocks less than 512 bytes are allocated by a (binary) buddy system. In particular, a request for memory of modest size results in the allocation of a number of bytes equal to the smallest power of 2 (reduced by 4 bytes used for administrative purposes) adequate for the request. It is therefore convenient to tune our structure by choosing physical element heights to be such that the entire node will fit exactly into the space allocated by the memory manager. For example, if we have 1-word keys, and if we also store the logical height of each skip list node in the node itself, then we should choose nodes of physical heights $2^i - 3$, for $i = 2, 3, \ldots$ (assuming that each pointer is 1 word = 4 bytes long).

We would like to emphasize at this point that the array implementation of our 1-2 skip list can be done in *any environment*. It is simply that in a buddy environment, if the array implementation is tuned as described, it doesn't require more space than its naive counterpart described in Section 2, and it improves the $\Theta(\lg^2 n)$ worst case update cost to $\Theta(\lg n)$. In a non-buddy environment, the array implementation may require up to 15% more pointers than its naive version, also achieving the $\Theta(\lg n)$ worst case update cost.

It is also worth noting that our array implementation can be streamlined somewhat. We can sidestep most of the $\Theta(\lg \lg n)$ calls to the storage manager, by simply keeping, along with the structure, $\Theta(\lg \lg n)$ free nodes, one of each size. When an insertion takes place, we initially request a free node of size 1 from our set of free nodes, and if some elements have to grow, we repeatedly request a new node one size larger and we release the old one. This leaves our set of free nodes with one node of each size except for the final node requested. A single call now to the memory manager rectifies the situation.

The $\Theta(\lg \lg n)$ calls to the storage manager can also be avoided if we choose to raise the elements' heights top-down, rather than bottom-up. We can do so, if, during our search, we record whether we drop from level h to level $h-1$ before the first, between the first and the second, or after the second element of height h. We can then determine which element has to grow to which height. Suppose that we find out that the highest element that has to grow is of height i and that it has to grow to height $i+j$. Then, the next lowest element that will grow, will grow to height exactly i. Thus, we can make only one call to the memory manager for an element of height $i+j$, since we may subsequently reuse the freed element of height i for the new element to grow to height i, etc.

4 Top-down 1-2-3 skip lists

The correspondence between 2-3 trees and 1-2 skip lists can be easily generalized. For any B-tree of order m, we can define a deterministic skip list that will allow $\lceil \frac{m}{2} \rceil - 1, \lceil \frac{m}{2} \rceil, \ldots, m-2, m-1$ nodes of the same height in a row. Fig. 3 shows a 2-3-4 tree and its corresponding skip list, that we will call *1-2-3 skip list* for obvious reasons.

As noted in [4], insertions in a 2-3-4 tree can be performed top-down, eliminating thus the need to maintain a stack with the search path. Adopting this approach, we insert an element in a 1-2-3 skip list by splitting any gap of size 3 into two gaps of size 1, when searching for the element to be inserted. We ensure in this way that the structure retains the gap invariant

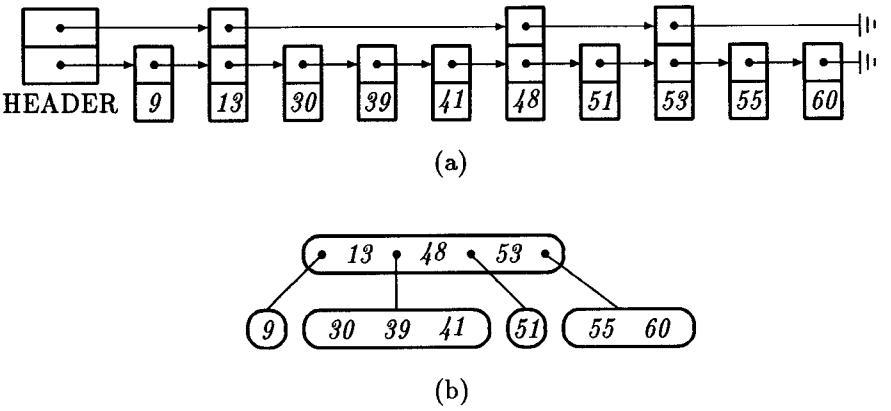


Figure 3: A 1-2-3 skip list and corresponding 2-3-4 tree

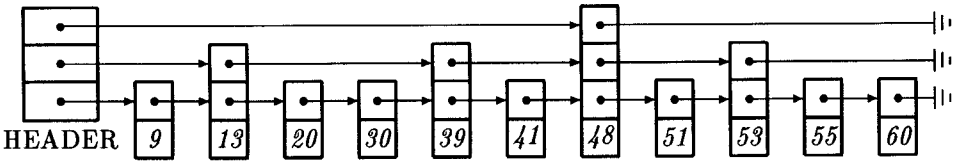


Figure 4: Top-down 1-2-3 skip list insertion

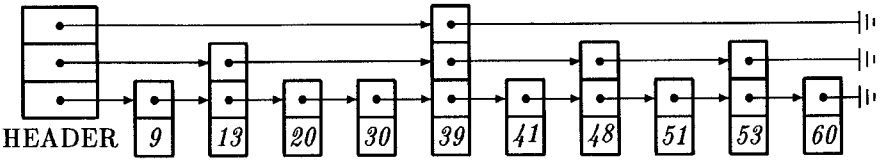


Figure 5: Top-down 1-2-3 skip list deletion

with or without the inserted element. To be more precise, we start our search at the header, and at level 1 higher than the height of the skip list. When we find the gap that we are going to drop, we look at the level below and if we see 3 nodes of the same height in a row, we raise the middle one; after that we drop down a level. When we reach the bottom level, we simply insert a new node of height 1. Since our algorithm allowed only gaps of sizes 1 and 2, the newly inserted element leaves us with a valid 1-2-3 skip list.

As an example, consider the case of inserting *20* in the skip list of Fig. 3(a). We start at level 3 of the header, we look at level 2 and we raise *48*, then we drop to level 2 of the header, we move to level 2 of *13*, we raise *39*, then we drop to level 1 of *13*, and we insert a new node of height 1 having the key *20*. The resulting skip list is shown in Fig. 4.

To delete an element from a 1-2-3 skip list, we work in a top-down manner as well. We want the search for the element that will be removed to have the side-effect that each gap is of legal — but above minimal — size as we pass through it. This is handled by either merging with a neighbour, or borrowing an element from that neighbour. More precisely, we start our search at the header and at level h equal to the height of the skip list. If we are going to drop down in the first gap and there is only 1 node of height $h-1$ there, then we lower the node of height h that we see ahead of us, and if there exist at least 2 nodes of height $h-1$ in the second gap, we raise the first of them. If we are not going to drop in the first gap, we find the $(i+1)$st (for some $i \geq 1$) gap that we are going to drop, and we also keep track of the ith gap. If there is only one node of height $h-1$ in the $(i+1)$st gap, then we lower the separator node

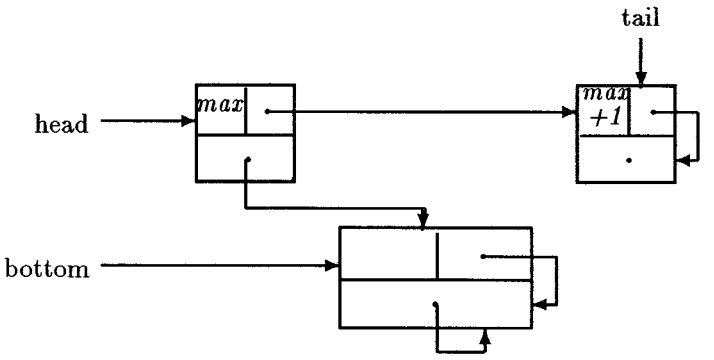


Figure 6: Linked list representation for the empty 1-2-3 skip list

of the ith and $(i+1)$st gaps, and if there exist at least 2 nodes of height $h-1$ in the ith gap, we raise the last of them. We then drop to level $h-1$. We continue in this way, until we reach the bottom level, where we remove the node of height 1 (if the key to be deleted is not in a node of height 1, we swap it with its neighbour of height 1, and we remove its neighbour). Since our algorithm didn't allow any gaps to be of size 1, what we are left with after the removal of the mode of height 1, is a valid 1-2-3 skip list.

As an example, consider the case of deleting *55* from the structure of Fig. 4. We start at level 3 of the header, and we lower *48* and raise *39*. We then continue from level 2 of *48*, we move to level 2 of *53*, then to level 1 of *53*, and we finally remove *55*. The resulting tree is shown in Fig. 5.

The top-down 1-2-3 skip list can be implemented by using either linked lists of pointers or arrays of pointers of exponential sizes for the skip list nodes. Both of these implementations are simpler than their counterparts for the 1-2 skip list because all of the work is done on one pass down through the structure, just as in [4]. Moreover, the linked list implementation of top-down 2-3-4 trees represented as skip lists is much simpler than the corresponding implementation using red-black trees and other balanced tree algorithms; such algorithms are notoriously complicated because of the numerous cases that arise involving single and double rotations on the left and the right.

To avoid having special cases in our code, we introduce a dummy `head` node and two sentinel nodes `bottom` and `tail`. Furthermore, to avoid a level of indirection, we also pull the keys back into the nodes in which the comparisons are actually made (see Fig. 6 and 7). Admittedly, this moves us to a representation between what one might view as a skip list and a binary tree.

Assuming the data type definition

```
struct node {int key; node *r, *d;}
```

where `r` is a right pointer and `d` is a down pointer, the function to search for key value `v` can be written as

```
node * Search(v)
int v;
{
   node * x;

   x = head;
   bottom->key = v;
   while (v != x->key)
      x = (v < x->key) ?  x->d :  x->r;
   return(x);
}
```

or as

```
node * Search(v)
int v;
{
  node * x;

  x = head;
  while (x != bottom) {
    while (v > x->key) x = x->r;
    if (x->d == bottom)
      return( (v == x->key) ?  x : bottom );
    x = x->d;
  }
}
```

These functions return a pointer to the node containing the key sought, or a pointer to `bottom` if the search was unsuccessful. Observe that the first version uses 3-way branches in each node, and it is identical to the search procedure for a binary search tree. The second version tests for equality only at the lowest level, and it is skip list in flavour.

If we choose the second version for the search code, then the code to insert a key `v` into a 1-2-3 skip list is just a matter of filling in a few gaps in the above code; we simply have to add a node in a horizontal link whenever necessary.

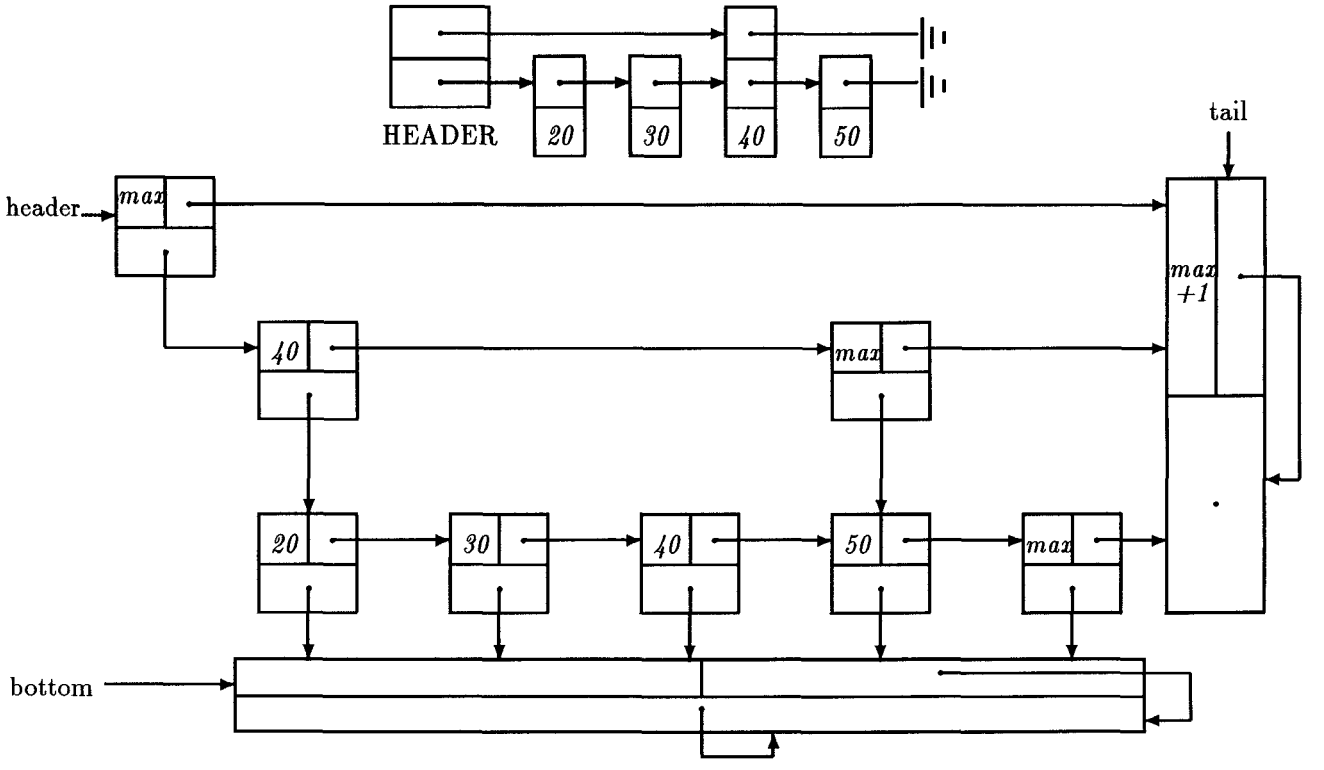


Figure 7: A 1-2-3 skip list and its corresponding linked list representation

```
int Insert(v)
int v;
{
 node *t, *x;

 x = head;
 bottom->key = v;
 while (x != bottom) {
  while (v > x->key) x = x->r;
  if ((x->d == bottom) && (v == x->key))
   return(0);
  if ((x->d == bottom)
    || (x->key == x->d->r->r->r->key)) {
   t = (node*)malloc(sizeof(struct node));
   t->r = x->r; t->d = x->d->r->r; x->r = t;
   t->key = x->key; x->key = x->d->r->key;
  }
  x = x->d;
 }
 if (head->r != tail) {
  t = (node*)malloc(sizeof(struct node));
  t->d = head; t->r = tail; t->key = maxkey;
  head = t;
 }
 return(1);
}
```

This function returns 1 if key value v was inserted in the skip list, and 0 if v was already in the skip list and it was not re-inserted.

We would like to emphasize here again that our purpose of giving the above rough C code is to show that this *fully debugged and working code* is much shorter and simpler than other published implementations of insertion for balanced trees. There are several tradeoffs in developing an implementation of the basic strategy that we have outlined. For example, one can maintain a field that counts the number of skipped nodes and thus not only simplify the test of when to split but also allow for trivial extension to higher-order skip lists. We are currently studying such implementation tradeoffs in detail.

Note also that, as discussed in [4], other standard balanced tree algorithms, such as AVL trees, can be transformed into simplified implementations through a skip list representation.

Similarly, the deletion algorithm is quite easy to code. Despite the fact that it is a bit longer than the insertion code, it is an even greater reduction in coding difficulty than is the case for insertion.

5 Conclusion

We have introduced several versions of deterministic skip lists, simple data structures with guaranteed logarithmic search and update costs. We observed (Fig. 1 and 3) that a skip list can be viewed as a multiway search tree in which the path length from the root to any leaf is the same; this path length corresponds to the height of the skip list, or to the number of vertical steps in the path for the search of any element in the skip list. As noted in [4], a multiway search tree, and hence a skip list, can be viewed as a binary search tree in which an element to the right, in the same node of the multiway tree, or in the same level of the skip list, can be viewed as a (possibly flagged) right child. While Pugh introduced the skip list as a probabilistic structure, we have moved away from that point, and we have focused on several representational distinctions that impact the ease of manipulation. One obvious distinction is that Pugh's skip list nodes have a key value and a varying number of pointers; our node may be implemented as linked lists of pointers. A search for an element in a skip list may encounter the same value several times (e.g. searching for *23* in the example of Fig. 1 involves two comparisons with *25*). These redundant comparisons, and indeed extra pointers, have the advantage of threading the structure, and also producing a representation in which splitting a multiway node (or merging two multiway nodes) in the tree framework becomes a much easier operation of allowing a linked list of horizontal pointers to grow (or shrink) by one. On the other hand, a standard rotation in a general search tree would require time proportional to the size of the entire subtree in the skip list setting.

The worst case number of comparisons in our 1-2 and 1-2-3 skip list is $2\lg n + \Theta(1)$ and $3\lg n + \Theta(1)$ respectively. The expected number of comparisons for both of them seem to be about $1.2\lg n + \Theta(1)$, better than the $1.9\lg n + \Theta(1)$ expected number of comparisons in the probabilistic skip list (tuned to minimize this value). Note though, that our data structures, unlike their probabilistic cousins, and like other balanced search tree schemes (AVL, red-black, etc.), are not history-independent.

The array implementations of our schemes require $2.282n$ pointers in the worst case, and the linked lists implementations require $6n$ pointers in the worst case. Probabilistic skip lists, minimizing the asymptotic expected search cost, require $1.582n$ pointers on average, and $n\lg n$ pointers in the worst case (when they are capped at level $\lg n$).

One might think that our deterministic skip lists require more space than balanced search trees. This is not the case though, for casual implementations in environments many of us program. Consider, for example, programming in C under Unix. For simplicity, assume that the keys are 1 word long each. Most Berkeley Unix-based systems allocate only powers of 2 number of bytes, and they keep 4 of these bytes for administrative purposes. So, to store 1 key, 2 pointers and 1 (or 2) bits per node, we have to allocate 8 words. Thus, AVL and red-black trees will always take $8n$ words. The linked list implementation of our skip lists will take at most $8n$ pointers in the worst case. The array implementation of our skip lists, even if we assume that we also store the height of each node in the node, will take 4 words for elements of height 1, 4 more words for the at most $\frac{1}{2}$ of the elements of heights 2 to 5, 8 more words for the at most $\frac{1}{32}$ of the elements of heights 6 to 13, etc. Hence, the array implementation of our skip lists will take $6.25n$ words in the worst case, which is less than 4/5 of the space always taken by the AVL and red-black trees. The probabilistic skip list will take $4.56n$ words on average, the array implementation of the 1-2 skip list about $5.77n$ ($4.74n$ if heights are not stored in nodes) words on the average, the array implementation of the 1-2-3 skip list about $5.49n$ ($4.54n$ if heights are not stored in nodes) words on the average, the linked list implementation of the 1-2 skip list about $6.58n$ words on the average, and the linked list implementation of the 1-2-3 skip list about $6.26n$ words on the average.

The "skip list" representation of binary trees can lead to substantially simpler implementations of standard abstract data structures such as 2-3 trees, top-down 2-3-4 trees, and AVL trees. Moreover, it provides a systematic way to view the "cross" pointers that are used in advanced data structures based on balanced trees [6] that should lead to substantially simpler implementations of a number of algorithms on such structures, and perhaps to the development of new methods.

References

[1] G. M. Adel'son-Vel'skii and E. M. Landis. "An algorithm for the Organization of Information". *Doklady Akademia Nauk SSSR*, vol. 146, 1962, pp. 263–266. English translation in *Soviet Mathematics Doklady*, vol. 3, pp. 1259–1263.

[2] R. Bayer and E. McCreight. "Organization and Maintenance of Large Ordered Indexes". *Acta Informatica*, vol. 1, 1972, pp. 173–189.

[3] R. L. Graham, D. E. Knuth and O. Patashnik. *Concrete Mathematics*. Addison-Wesley, 1989.

[4] L. Guibas and R. Sedgewick. "A dichromatic framework for balanced trees". *19th Annual Symposium in Foundations of Computer Science* IEEE Computer Society. Ann Arbor, Michigan, Oct. 1978, pp. 8–21.

[5] H. R. Lewis and L. Denenberg. *Data Structures and their Algorithms*. Harper Collins, 1991.

[6] K. Mehlhorn. *Data Structures and Algorithms*, vol. 1.

Springer-Verlag, 1984.

[7] T. Papadakis, J. I. Munro and P. Poblete. "Average Search and Update Costs in Skip Lists". *BIT*, to appear.

[8] W. Pugh. "Skip Lists: A Probabilistic Alternative to Balanced Trees". *Communications of the ACM*, vol. 33, no. 6, June 1990, pp. 668–676.