Skip Lifts: A Probabilistic Alternative to Red-Black Trees

Prosenjit Bose, Karim Douïeb, and Pat Morin*

School of Computer Science, Carleton University, Herzberg Building
1125 Colonel By Drive, Ottawa, Ontario, K1S 5B6 Canada
{jit,karim,morin}@cg.scs.carleton.ca
http://cg.scs.carleton.ca

Abstract. We present the *Skip lifts*, a randomized dictionary data structure inspired from the skip list [Pugh '90, Comm. of the ACM]. Similarly to the skip list, the skip lifts has the finger search property: Given a pointer to an arbitrary element f, searching for an element x takes expected $O(\log \delta)$ time where δ is the rank distance between the elements x and f. The skip lifts uses nodes of $O(1)$ worst-case size and it is one of the few efficient dictionary data structures that performs an $O(1)$ worst-case number of structural changes during an update operation. Given a pointer to the element to be removed from the skip lifts the deletion operation takes $O(1)$ worst-case time.

1 Introduction

The dictionary problem is fundamental in computer science. It asks for a data structure in the pointer machine model that stores a totally ordered set S of n elements and supports the operations search, insert and delete. A large number of data structures optimally solve this problem in $O(\log n)$ time per operation. Some of them guarantee an $O(1)$ worst-case number of structural changes (pointers/fields modifications) after an insertion or a deletion operation [12,19,11,13,10,6]. This is valuable when writing in memory is significantly slower than reading.

Typically the update operations, i.e., insert and delete, are performed in two phases: First, search for the position where the update has to take place. Second, perform the actual update and restore the balance of the structure. When the position where the new element has to be inserted or deleted is already known then the first phase of an update could be avoided. In general the first phase is considered to be part of the search operation. A dictionary that guarantees an $O(1)$ worst-case number of structural changes per update does not necessary quickly perform the second phase of the update. For example after inserting an new item in a red-black tree [12], $\Omega(\log n)$ steps may be required to find where the $O(1)$ number of rotations have to be performed in order to restore the balance. A lot of effort have been done to improve the worst-case time taken by the second

* Research partially supported by NSERC and MRI.

phase of the update: Levcopoulos and Overmars [13] presented the first search tree that takes $O(1)$ worst-case time for this second phase of the update. Later Fleischer [10] simplified this result. Brodal *et al.* [6] additionally guaranteed that such structure can also have the finger search property in worst-case. These structures are unfortunately quite complicated and not really practical.

In the other hand most randomized dictionaries are simple, practical and achieve the same performance than the result of Brodal *et al.* [6] in the expected sense. In worst case their performances are far from being optimal. Here we develop a simple randomized dictionary, called skip lifts inspired from the skip lists [18], that improves the worst-case performance of the second phase of the update operations. Namely we obtain a structure that has the finger search property in expectation and perform an $O(1)$ worst case number of structural change per update. Given a pointer to the element to be removed from the skip lifts the deletion operation takes $O(1)$ worst-case time.

In Section 1.1 we describe the original skip list dictionary. In Section 1.2 we mention some work related to the skip list dictionary. In Section 2 we introduce our new skip lifts data structure. In Section 3 we show how to enhance the skip lifts structure to allow a simple finger search process. Finally in Section 4 we give an overview of some classical randomized dictionary data structures. For each of them we briefly describe its construction and how the dictionary operations are performed. We show that in some situations $\Omega(n)$ structural changes are necessary to perform the update operations.

1.1 Skip list

The *skip list* of Pugh [18] has been introduced as a probabilistic alternative to balanced trees. It is a dictionary data structure storing a totally ordered set S of n elements that supports insertion, deletion and search operations in $O(\log n)$ expected time. Additionally the expected number of structural changes (pointer modifications) performed to the skip list during an update is $O(1)$. A skip list is built in levels, the bottom level (level 1) is a sorted linked list of all elements in S. The higher levels of the skip list are build iteratively. Each level is a sublist of the previous one where each element of a level is copied in the level above with (independent) probability p. The copies of an element are linked between adjacent levels (see Fig. 1.a).

The *height* $h(s)$ of an element s is defined as the highest level where s appears. The height $H(\mathcal{L})$ of a skip list $\mathcal{L}$ is defined as $\max_{s\in\mathcal{L}} h(s)$ and the *depth* $d(s)$ of s is $H(\mathcal{L}) - h(s)$. The expected height of a skip list is by definition $O(\log_{1/p} n)$. Adjacent elements on the same level are connected by their left and right pointers. The copies of a same element from two adjacent levels are connected by their up and down pointers.

Search: To search for a given element x in a skip list we start from the highest level of the sentinel element which has a key value $-\infty$. We follow the right pointers on a same level until we are about to overshoot the element x, i,e., until the element on the right has a key value strictly greater than x. Then we

go down one level and we iterate the process until x is found or when we have reached the lowest level (in this case we know that x is not in S and we have found its predecessor).

Updates: To insert an element x in a skip list we first determine its height in the structure. Then we start to search for x in the list to find the position where x has to be inserted. During the search we update the pointers of the copies of the elements that are adjacent to a newly created copy of x.

The deletion of an element x from a skip list is straightforward given the insertion process. We first search for x and we delete one by one all its copies while updating the pointers of the copies of elements that are adjacent to a copy of x.

1.2 Related work

Precise analysis of the expected search cost in a skip list have been extensively studied, we refer to the thesis of Papadakis for more information [17]. Several variants of the skip list have been considered: Munro *et al.* [16] developed a deterministic version of the skip list, based on B-trees [3], that performs each dictionary operation in worst case $O(\lg n)$ time. Under the assumption that the distribution of access probabilities is given, Martínez and Roura [14] developed an algorithm that minimizes the expected access time by either building an optimal static skip list in $O(n^2 \lg n)$ time or a nearly optimal one in $O(n)$ time. Bagchi *et al.* [2] developed the biased skip list, it manages a biased dictionary, i.e., an ordered set S of elements x associated with a weight $w(x)$ and performs search, insert, delete, join, split, finger search and reweight operations in worst case running times similar to those of biased search trees [4,9]. In the general case where access probabilities are unknown, Bose et al. [5] proves that for a class of skip lists that satisfy a weak balancing property, the working-set bound

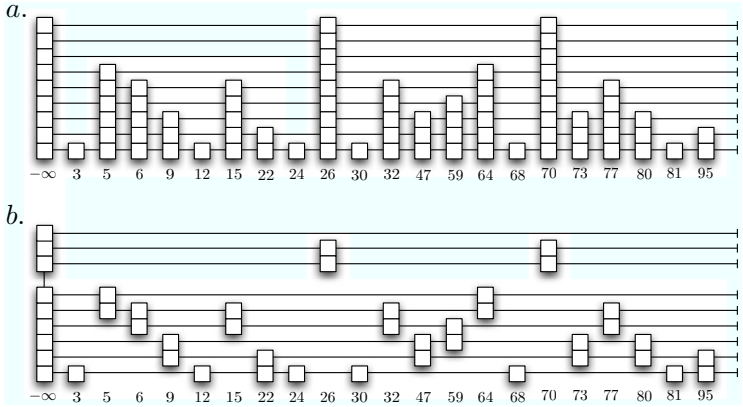

Fig. 1: a. Skip list, b. Skip lifts

is a lower bound on the time to access any sequence. Furthermore, they develop a deterministic self-adjusting skip list whose running time matches the working-set bound, thereby achieving dynamic optimality in this class (both in internal and external memory).

2 Skip lifts

The average number of extra information per element (number of copies) in a standard skip list [18] is constant. In the worst case this number can reach $\Omega(\log n)$. Hence the number of structural changes in a skip list during an update is $\Omega(\log n)$ in worst-case. Here we present a slight modification of the skip list data structure (as the title of the paper suggests it) which guarantees, in worst case, a constant number of extra information per element and a constant number of structural changes per update.

A skip lifts is a light version of the skip list where copies of elements have been removed from specific levels. A skip lifts only keeps the copies of an element in the two highest levels where it appears. Every other copy of an element is removed. The copies of the elements at the same level are connected with their left and right pointers. Additionally the two copies of an element are connected with their up and down pointers (see Figure 1.b). Each copy stores its height in an extra field height.

A level of the skip lifts is empty if no element of the set S appears in it. The copies of the sentinel element appearing in an empty level are deleted. The remaining copies of the sentinel element are connected with their up and down pointers. A copy of the sentinel element at height $+\infty$ is explicitly maintained, this copy is called the *header* of the skip lifts.

Search: To search for a given element x in a skip lifts we start at the header of the list. We follow the right pointers on the same level until we see that we are about to overshoot the element x, i,e., until the element on the right has a key value strictly greater than x. If it is possible we go down in the next non empty level. Otherwise we follow the left pointers until we find an element which allows us to go down in the next non empty level. Then we iterate the process until x is found or when we have reached the lowest level (in this case x is not in S and we know its predecessor). This procedure is described in detail in Algorithm 1.

Lemma 1. *Skip lifts support search operation in* $O\left(\frac{1}{p}\log_{1/p} n\right)$ *expected time.*

Proof. The expected lenght of the search path in a skip lifts $\mathcal{L}$ corresponds to the expected number of vertical steps plus the expected number of horizontal steps. The number of vertical steps performed during a search is upper bounded by the height $H(\mathcal{L})$ of the skip lifts which has an expected value of $\log_{1/p} n + \frac{1}{1-p}$. Indeed the expected height of a skip lifts corresponds exaclty to the expected height of a skip list [18].

Now we are going to bound the number of horizontal steps. In any level i of $\mathcal{L}$ only elements of height i and $i+1$ can appear with probability $1/(1+p)$ and

$p/(1+p)$, respectively. This means that from any position in level i the expected number of horizontal steps required to reach an element of height i is

$$s_1 \leq \sum_{j=1}^{\infty} j \left(\frac{p}{1+p}\right)^{j-1} \frac{1}{(1+p)} = 1+p.$$

Similarly the expected number of horizontal steps required to reach an element of height $i+1$ in level i is

$$s_2 \leq \sum_{j=1}^{\infty} j \left(\frac{1}{1+p}\right)^{j-1} \frac{p}{(1+p)} = \frac{1+p}{p}.$$

Consider $e(i,x)$ the element of height i that as the greatest key value smaller than x. The search path to an element x in $\mathcal{L}$ traverses all elements $e(i,x)$ with $h(x) \leq i \leq H(\mathcal{L})$. These are the only elements where the search path performs a down step. Between each of these $e(i,x)$ elements the search path traverses horizontally a certain number of other elements. On expectation this number differs depending on whether the path goes from left to right or right to left. If the path goes from right to left this expected number corresponds to s_1 otherwise it corresponds to s_2. The probability that the search path goes from left to right on level i is $1/(p+1)$. This corresponds to the probability of seeing $e(i,x)$ before $e(i+1,x)$ from the position of x on level i which also corresponds to the probability that an element of height i appears on level i. Respectively the probability that the search path goes from right to left on level i is $p/(p+1)$. Hence the expected number of horizontal steps performed between each element $e(i,x)$ is

$$p + 1\frac{p}{p+1} + \frac{1+p}{p}\frac{1}{1+p} = p + \frac{1}{p}.$$

Algorithm 1 Search(x)

```
c ⇐ header
pred ⇐ −∞
while c ≠ x and height[c]>1 do
   while down[c] = NIL do
      c ⇐ left[c]
   end while
   c ⇐ down[c]
   while right[c] ≠ NIL and right[c] ≤ x do
      c ⇐ right[c]
   end while
   if pred < c then
      pred ⇐ c
   end if
end while
return pred
```

The expected cost to access the fist element $e(H(\mathcal{L}), x)$ is smaller than the expected number of elements of height greater or equal to $\log_{1/p} n$ which is $1/p$. Thus total expected number of horizontal steps is upper bounded by

$$H(\mathcal{L}) \left(p + \frac{1}{p} \right) + \frac{1}{p}.$$

Therfore the expected length of a search path in a skip lifts is

$$H(\mathcal{L}) + H(\mathcal{L}) \left(p + \frac{1}{p} \right) + \frac{1}{p} = \frac{H(\mathcal{L}) + 1}{p} + (p+1)H(\mathcal{L}) = O\left(\frac{\log_{1/p} n}{p} \right).$$

□

Updates: To insert an element x in a skip lifts we first determine its height $h(x)$ in the structure. Then we start to search for x in the list to find the position where x has to be inserted, i,e,. its position in levels $h(x)$ and $h(x) - 1$. Once we found these positions the copies of the element x are inserted in the corresponding level. This is performed similarly as the insertion of an element in a standard doubly-linked list. If the level where the copy of x has to be insereted is empty then we create a new copy of the sentinel element and insert it the skip lifts (seeing all copies of the sentinel element as a doubly-linked list). This process is described in detail in Algorithm 2. We assume that x is not in the set S (otherwise we could simply search for x before perfoming the actual insert operation).

To delete an element x from a skip lifts we first search the two copies of x using the search operation decribed above. Once we found the copies of x we delete them from their corresponding level. This is performed similarly as the deletion of an element in a standard doubly-linked list. If the deletion of the copies of x creates an empty level, we remove the corresponding copy of the sentinel element. This process is described in detail in Algorithm 3.

Theorem 1. *The skip lifts support search, insert and delete operations in* $O\left(\frac{1}{p} \log_{1/p} n\right)$ *expected time and requires* $O(n)$ *worst-case space. The total amont of structural changes performed during an update is* $O(1)$ *in worst case.*

Algorithm 2 Insert(x)

```
c ⇐ header
h ⇐ randomLevel()
while height[c] ≥ h do
  while down[c] = NIL do
    c ⇐ left[c]
  end while
  if c = −∞ and height[down[c]] < h
  and h < height[c] then
    e ⇐ new element(−∞, h)
    down[e] ⇐ down[c]
    down[c] ⇐ e
  end if
  c ⇐ down[c]
  while right[c] ≠ NIL and right[c] ≤ x
  do
    c ⇐ right[c]
  end while
  if height[c] = h then
    right[x] ⇐ right[c]
    left[x] ⇐ c
    left[right[c]] ⇐ x
    right[c] ⇐ x
    x ⇐ down[x]
    if x ≠ NIL then
      h ⇐ h − 1
    end if
  end if
end while
```

Algorithm 3 Delete(x)

```
c ⇐ header
while height[c] > 1 do
  while down[c] = NIL do
    c ⇐ left[c]
  end while
  c ⇐ down[c]
  while right[c] ≠ NIL and right[c] ≤ x do
    c ⇐ right[c]
  end while
  while c = x do
    right[left[x]] ⇐ right[x]
    if right[x] ≠ NIL then
      left[right[x]] ⇐ left[x]
    else if left[x] = −∞ then
      down[up[left[x]]] ⇐ down[left[x]]
      if down[left[x]] ≠ NIL then
        up[down[left[x]]] ⇐ up[left[x]]
      end if
      delete left[x]
    end if
    c ⇐ down[c]
    delete(up[c])
  end while
end while
```

3 Finger Search

A data structure satisfies the finger search property if searching for an element x given a pointer, called *finger*, to an arbitrary element f requires logarithmic time in the rank distance between x and f in the set of ordered elements. It is possible to describe a finger search operation on the skip lifts (as described in the previous section) but it is a bit complicate. Instead we show how to enhance the skip lifts structure in order to simplify the description of the finger search. The *enhanced skip lifts* are maintaining an extra copy of each element at the bottom level. This copy is linked to the lowest other copy of the corresponding element with the up and down pointers.

We can search for an element x in a enhanced skip lifts starting at the bottom copy of any element f to which we are given an initial pointer. Assume without loss of generality that the key value of the element x is greater than the one of f (the opposite case is symmetric). The finger search can be decomposed into an

up phase and a *down phase*. The up phase behaves as the inverse of the search operation described in Alg. 1 and the down phase is similar to Alg. 1.

We start the search from the bottom copy of f then from any current position we follow the left pointers on the same level until the element on the left has a key value strictly smaller than f. If it is possible we go one level up (if the up pointer jump over more than one level then we are not taking it). Otherwise we follow the right pointers until we find an element which allows us to go one level up or when the element on the right has a key value greater than x (this last case corresponds to the end of the up phase). The down phase consists, from the current position, of following the right pointers on the same level until the element on the right has a key value strictly greater than x. If it is possible we go down by one level (if the down pointer jump over more than one level then we are not taking it). Otherwise we follow the left pointers until we find an element which allows us to go down by one level. Then we iterate the process until x is found or when we have reached the lowest level (in this case x is not in S and we know its predecessor).

Theorem 2. *Finger searching for an element x given a finger pointing to an arbitrary element f in an enhanced skip lifts takes $O\left(\frac{1}{p}\log_{1/p}\delta\right)$ time where δ is the rank distance between the finger and the search element x.*

Proof. The search path traverses only elements that are between f and x in the skip lifts. The sublist between f and x contains δ elements by definition. Thus the expected height of this sublist is $O(\log_{1/p}\delta)$. In each level we perform $O(1/p)$ expected steps since this corresponds to the expected number of steps needed to find an element of height i or $i+1$ from any position on level i. Therefore the total length of the search path is $O\left(\frac{1}{p}\log_{1/p}\delta\right)$. □

4 Overview of randomized dictionaries

We present an overview of some classical randomized dictionary data structures. For each of them we briefly describe its construction and how search, insertion and deletion operations are performed. It is easy to realize that the structural changes performed during an update operation can in some situations involve $\Omega(n)$ elements of the structure. Of course those situations are very unlikely to happen but are not impossible. This means that the skip lifts are probably the first efficient randomized dictionary that guarantees a $O(1)$ number of structural changes per update.

4.1 Modified skip list

A *modified skip list* is a variant of the skip list introduced by Cho and Sahni [8] that uses nodes of constant worst-case size (containing $O(1)$ pointers). The modified skip list structure is a skip list where all copies of an element are deleted except for its highest copy. Thus an element x only appears on the level $h(x)$.

Each element x has three pointers: $\mathsf{right}[x]$, $\mathsf{left}[x]$ and $\mathsf{down}[x]$. The pointers $\mathsf{right}[x]$ and $\mathsf{left}[x]$ point to the elements on level $h(x)$ to the right and the left of x, respectively. The pointer $\mathsf{down}[x]$ points to the element on level $h(x) - 1$ that has the smallest key value greater than x. Two sentinel elements with key value $-\infty$ and ∞ are maintained, a copy of these elements appear in every level. The down pointer of a copy of a sentinel element points to the copy of itself on the level below (see Fig. 2).

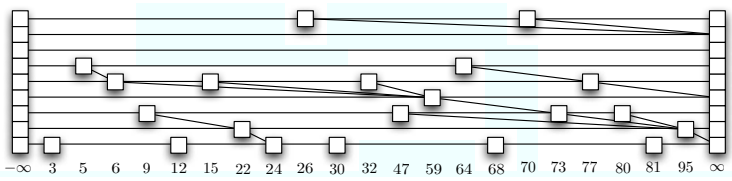

Fig. 2: Modified skip list

Search: To search for a given element x in a modified skip list we start from the highest level of the sentinel element with key value $-\infty$. We follow the right pointers on a same level until the element on the right has a key value strictly greater than x. From this point we follow the left pointer then we immediately go down one level by following the down pointer from this left element. The process is iterated until x is found or when we have reached in the lowest level (in this case we know that x is not in S and we have found its predecessor).

Updates: The insert and delete operations require to search the position of x in the list. When inserting an element x in a modified skip list only one copy is created in the level $h(x)$ and the down pointer of x is set to the element in level $h(x) - 1$ that has the smallest key value greater then x. When deleting an element x from a modified skip list we have to update the down pointer of all the elements from level $h(x) + 1$ that are pointing to x by setting them to the element on the right of x.

A degenerate situation would be when all elements in the structure have height 2 except for the very last one (with the greatest key value). Deleting the last element would force the modification of the down pointer all elements, implying an $\Omega(n)$ number of structural changes in the structure. A similar situation can occurs if we insert an element just before the last one.

4.2 Treap

A *treap* is a randomized data structure introduced by Aragon and Seidel [1]. It is structured as a binary search tree structure, so the left and the right subtrees of any node only contain elements of smaller or greater key value, respectively.

Each element of S is given a random priority. The treap is built such that the root is the minimum-priority node and the priority of any non-root node must be greater than or equal to the priority of its parent (heap-ordering property).

Search: To search for a given element x, we use the standard binary search algorithm in a binary search tree independently of the priorities.
Updates: To insert a new elements x into the treap, we first generate a random priority for x. We perform a search for x in the treap. If $x \in S$ we do nothing otherwise we make x a child of the last element visited during the search. Then x is rotated up as long as its priority is smaller than the priority of its parent or when x becomes the new root.

To delete a node x from the treap three cases are considered. If x is a leaf, we simply remove it. If x has a single child, we remove x from the treap and make the child of x the new child of its the parent (or make the child of x the root if x had no parent). Finally, if x has two children, swap its position in the treap with its predecessor, resulting in one of the previously discuss cases. In this final case, the swap may violate the heap-ordering property, so additional rotations may need to be performed to restore it.

A degenerate situation would be when the tree is a path of n elements. Inserting an elements at the end of the path with a given priority that is smaller than any priority in the tree would bring the new inserted element at the root. This is performed by a sequence of $\Omega(n)$ rotations, i,e,. an $\Omega(n)$ number of structural changes in the tree. A similar situation could occurs when deleting an element.

4.3 Randomized binary search tree

A *Randomized binary search tree* is another dictionary data structure developed by Martínez and Roura [15] prior to the treap. Each subtrees of a random search tree is itself a random search tree. The root of such a tree is chosen uniformly at random among the elements of S, i.e., with probability $1/n$. The remaining of the tree is defined iteratively.

Search: To search for a given element x, we use the standard binary search algorithm in a binary search tree.
Updates: To insert a new elements x into a random search tree T we proceed as follow: With probability $1/(|T|+1)$ the element x has to be the root of the new tree. In this case the tree T is split at x and the two obtained subtrees are attached as the children of x. Otherwise we iterate the process on the left (right) subtree if x is smaller (greater) than the key value of the root.

To delete an element x from a random search tree T, we search for it in T. Once it is found we remove it and we replace the subtree rooted at x by a newly created subtree obtained by joining the left and right subtree of x (this joining procedure is fully described in [15]).

A degenerate situation would be when the tree is a path of n elements so that the key of the elements from the root to the leaf are alternatively greater

and smaller x. Assume we insert a new element with key value x, it could be that x has to be inserted has the root of the tree. In this case we split the tree at x which requires an $\Omega(n)$ number of structural changes in the tree. The inverse situation could occurs when deleting an element.

4.4 Jumplist

A *jumplist* of Brönnimann *et al.* [7] is a randomized data structure inspired from the randomized tree. It is an linked list data structure ordered by key value whose nodes are endowed with an additional pointer, the *jump pointer* (see Fig. 3). An element x of a jumplist has a $\mathsf{next}[x]$ pointer which points to the immediate successor of x in S. Additionally an element has a $\mathsf{jump}[x]$ pointer which points to an element further on the list to the right of x. The jumplist is constructed as follow: The element j pointed by the jump pointer of the header of the list is chosen uniformly at random among the elements in the list. This assignment divides the list into two independent sublists that are build recursively using to the same random process. This construction ensures that the jump pointers do not cross.

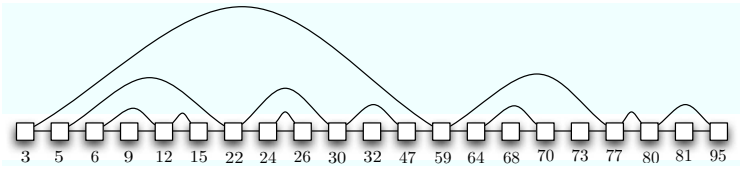

Fig. 3: Jumplist

Search: The jumplist are based on the *jump-and-walk* strategy: whenever possible use the jump pointer to speed up the search, and walk along the list otherwise. So to search for an element x we use the jump pointer until we are about to overshot x in this case we follow the next pointer. We iterate this process until we found the element x or when the next pointer lead us to an element with greater key value than x (in this case we know that x is not in S and we have found its predecessor).
Updates: To insert an element x in a jumplist J we proceed as follow: With probability $1/|J|$ the element x has to be the element pointed by the jump pointer of the header of the list. In this case the whole list is rebuilt from crash. Otherwise x is inserted in one of its sublists. In the case where x has to be inserted as the new header of a sublist, a process that do not rebuild the sublist from crash is called to maintain the randomness of the structure.

Since an insertion could yield to the reconstruction of the entire jumplist, it is obvious that this operation requires an $\Omega(n)$ number of structural changes in the list.

References

1. C. R. Aragon and R. Seidel. Randomized search trees. In *Algorithmica*, volume 16, pages 464–497, 1996.
2. A. Bagchi, A. L. Buchsbaum, and M. T. Goodrich. Biased skip lists. In *Algorithmica*, volume 42(1), pages 31–48, 2005.
3. R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. In *Acta Informatica*, volume 1, pages 173–189, 1972.
4. S. W. Bent, D. Sleator, and R. Tarjan. Biased search trees. In *SIAM Journal on Computing*, volume 14(3), pages 545–568, 1985.
5. P. Bose, K. Douïeb, and S. Langerman. Dynamic optimality for skip lists and b-trees. In *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms (SODA '08)*, pages 1106–1114, 2008.
6. G. S. Brodal, G. Lagogiannis, C. Makris, A. K. Tsakalidis, and K. Tsichlas. Optimal finger search trees in the pointer machine. *J. Comput. Syst. Sci.*, 67(2):381–418, 2003.
7. H. Brönnimann, F. Cazals, and M. Durand. Randomized jumplists : A jump-and-walk dictionary data structure. In *Proceedings of the 12th Annual Symposium on Theoretical Aspects of Computer Science (STACS '03)*, volume 2607 of *LNCS*, pages 283–294, 2003.
8. S. Cho and S. Sahni. Weight-biased leftist trees and modified skip lists. *J. Exp. Algorithmics*, 3:2, 1998.
9. J. Feigenbaum and R. Tarjan. Two new kinds of biased search trees. In *Bell System Technical Journal*, volume 62(10), pages 3139–3158, 1983.
10. R. Fleischer. A simple balanced search tree with $O(1)$ worst-case update time. *International Journal of Foundations of Computer Science*, 7:137–149, 1996.
11. B. Haeupler, S. Sen, and R. E. Tarjan. Rank-balanced trees. In *11th International Symposium on Algorithms and Data Structures (WADS '09)*, pages 351–362, 2009.
12. R. S. Leonidas J. Guibas. A dichromatic framework for balanced trees. In *Proc. 19th IEEE Symp. on Foundations of Computer Science*, pages 8–21, 1978.
13. C. Levcopoulos and M. Overmars. A balanced search tree with $O(1)$ worst-case update time. *Acta Informatica*, 26(3):269–277, 1988.
14. C. Martinez and S. Roura. Optimal and nearly optimal static weighted skip lists. Technical report, LSI-95-34-R, Dept. Llenguatges i Sistemes Informàtics (Universitat Politècnica de Catalunya), 1995.
15. C. Martínez and S. Roura. Randomized binary search trees. *J. ACM*, 45(2):288–323, 1998.
16. I. Munro, T. Papadakis, and R. Sedgewick. Deterministic skip lists. In *Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms*, pages 367–375, 1992.
17. T. Papadakis. *Skip lists and probabilistic analysis of algorithms*. PhD thesis, University of Waterloo, Department of Computer Science and Faculty of Mathematics, 1993. (Available as Tech. Report CS-93-28).
18. W. Pugh. Skip lists: a probabilistic alternative to balanced trees. In *Communications of the ACM*, volume 33(6), pages 668–676, 1990.
19. R. E. Tarjan. Updating a Balanced Search Tree in $O(1)$ Rotations. *Inf. Process. Lett.*, 16(5):253–257, 1983.