

Randomized Jumplists: A Jump-and-Walk Dictionary Data Structure ^{*}

Hervé Brönnimann, Frédéric Cazals, and Marianne Durand

¹ Polytechnic University, CIS, Six Metrotech, Brooklyn NY 11201, USA;
`hbr@poly.edu`

² INRIA Sophia-Antipolis, Projet Prisme, F-06902 Sophia-Antipolis, France;
`Frederic.Cazals@sophia.inria.fr`

³ INRIA Rocquencourt, Projet Algo, F-78153 Le Chesnay, France;
`Marianne.Durand@inria.fr`

Abstract. This paper presents a data structure providing the usual dictionary operations, i.e. CONTAINS, INSERT, DELETE. This data structure named *Jumplist* is a linked list whose nodes are endowed with an additional pointer, the so-called jump pointer. Algorithms on jumplists are based on the *jump-and-walk* strategy: whenever possible use to the jump pointer to speed up the search, and walk along the list otherwise. The main features of jumplists are the following. They perform within a constant factor of binary search trees. Randomization makes their dynamic maintenance easy. Jumplists are a compact data structure since they provide rank-based operations and forward iterators at a cost of three pointers/integers per node. Jumplists are trivially built in linear time from sorted linked lists.

Keywords. Dictionary data structures. Searching and sorting. Randomization. Asymptotic analysis.

1 Introduction

Dictionaries, Binary Search Trees (BST) and alternatives. Dictionaries and related data structures have a long standing history in theoretical computer science. These data structures were originally designed so as to organize pieces of information and provide efficient storage and access functions. But in addition to the standard CONTAINS, INSERT and DELETE operations, several other functionalities were soon felt necessary.

In order to accommodate divide-and-conquer algorithms, split and merge operations are mandated. For priority queues, access to the minimum and/or maximum must be supported. For applications involving order statistics, rank-based operations must be provided. Additionally, the data structure may be

^{*} Extended abstract. Due to space limitations, all proofs have been omitted. Refer to [1] for the full paper.

requested to incorporate knowledge about the data processed —e.g. random or sorted keys.

This variety of constraints lead to the development of a large number of data structures. The very first ones were randomly grown BST [15, 14] as well as deterministic balanced BST [26, 14, 7]. Since then, solutions more geared to provably good amortized or randomized performance were proposed. Splay trees [24], treaps [5], skip lists [18] and more recently randomized BST [16] fall into this category. An important feature of the randomized data structures—in particular randomized BST and skip lists—is their ease of implementation. Skip lists are particularly illustrative since the deterministic versions are substantially more difficult to code [17].

This paper presents a data structure providing the usual dictionary operations, i.e. CONTAINS, INSERT and DELETE. This data structure named *jumplist* is an ordered list whose nodes are endowed with an additional pointer, the so-called jump pointer. Algorithms on jumplists are based on the *jump-and-walk* strategy: whenever possible use to the jump pointer to speed-up the search, and walk along the list otherwise.

Like skip lists, we use jump pointers to speed-up searches. Similarly to skip lists too, the profile of the data structure does not depend on the ordering of the keys processed. Instead, a jumplist depends upon random tosses independent from the keys processed. Unlike skip lists, however, jumplists do not have a hierarchy of jump pointers. In particular, every node contains exactly two pointers, so the storage required is linear—as opposed to expected linear storage for skip lists. The data structures closest to jumplists in addition to skip lists are randomized BST. Similarly to skip lists or jumplists, the profile of randomized BST is independent from the sequence of keys processed.

The main features of jumplists are the following. Their performance are within a constant factor of optimal BST. Randomization makes their dynamic maintenance very easy. Jumplists are a compact data structure since they provide rank-based operations and forward iterators at a cost of three pointers/integers per node. Jumplists are trivially built in linear time from sorted linked lists.

Overview. This paper is organized as follows. The jumplist data structure and its search algorithms are described in Section 2. Section 3 analyzes the expected performance of the data structure. Dynamic maintenance of jumplists is presented in Section 4.

2 Jumplist: data structure and searching algorithms

2.1 The data structure

The jumplist is stored as a singly or doubly connected list, with $\text{next}[x]$ pointing forward and $\text{prev}[x]$ backward. The reverse pointers are not needed, except for backward traversal. If bidirectional traversal is not supported, they can be omitted from the presentation. The list is circularly connected, and the successor

(resp. predecessor) of the header is the first (resp. last) element of the list, or the header in either case if the list is empty. See Figure 1. Following standard implementation technique, the list always has a node $\text{header}[L]$ which contains no value, called its *header*, and which comes before all the other nodes. This facilitates insertions in a singly-linked list.

To each node is associated a *key*, and we assume that the list is sorted with respect to the keys. It is convenient to treat the header as the first node of the jumplist and give it a key of $-\infty$, especially for expressing the invariants and in the proofs. In this way, there is always a node with key less than k , for any k . We will be careful to state our algorithms such that the header key is never referenced. We may introduce for each node x an interval $[\text{key}[x], \text{key}[\text{next}[x]]]$ which corresponds to the keys that are *not* present in the jumplist. Thus if n represents the number of nodes, there are $n - 1$ keys. When all $n - 1$ keys are distinct, there are n intervals defined by the keys and by the header.

We denote by $x \prec y$ the relation induced by the order of the list (beginning at the first element and ending at the header). If all nodes have distinct keys, this relation is the same as that induced by the keys: $x \prec y$ iff $\text{key}[x] < \text{key}[y]$, and $x \preceq y$ iff $\text{key}[x] \leq \text{key}[y]$. When some keys are identical, it is inefficient to test whether $x \prec y$ when $\text{key}[x] = \text{key}[y]$ (one must basically traverse the list).

Traversal of jumplists, unlike BST, is extremely simple: simply follow the list pointers.

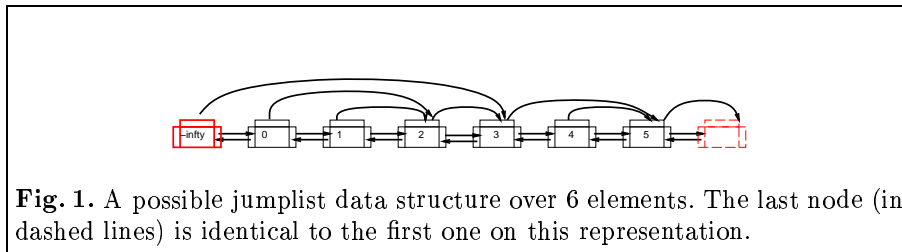


Fig. 1. A possible jumplist data structure over 6 elements. The last node (in dashed lines) is identical to the first one on this representation.

In addition to the list pointers, each node also has a pointer $\text{jump}[x]$ which points to a successor of x in the list. We refer to the pair $(x, \text{jump}[x])$ as an *arch*, and to the arch starting at $\text{header}[L]$ as the *fundamental arch*. As mentioned earlier, the jump pointers have to satisfy $x \prec \text{jump}[x]$ for every node x not equal to the last node, as well as the non-crossing condition: for any pair of nodes x, y , we cannot have $x \prec y \prec \text{jump}[x] \prec \text{jump}[y]$. Thus if $x \prec y$, then either $y \prec \text{jump}[y] \prec \text{jump}[x]$ (*strictly nested*), $\text{jump}[x] \preceq y \prec \text{jump}[y]$ (*semi-disjoint*), or $y \prec \text{jump}[y] = \text{jump}[x]$ if $\text{jump}[y] = \text{next}[y]$ (*exceptional pointer*). In order to close the loop, we also require that the jump pointer of the last element points back to the header. This last pointer is also called exceptional.

Remark. Exceptional pointers are necessary because otherwise we could not set the jump pointers of $y = \text{prev}[\text{jump}[x]]$, for nodes x such that $\text{next}[x] \prec \text{jump}[x]$. The value $\text{jump}[y] = \text{next}[y]$ is the only one that does not break a non-strictly nested condition. We could also get rid of exceptional pointers by

putting $\text{jump}[x] = \text{NIL}$ in that case. This actually complicates the algorithms which have to guard against null jump pointers. Since it will be clear during the discussion of the search algorithm that these exceptional jump pointers are never followed anyway, it only hurts to set them to NIL . Therefore, we choose to set them to $\text{next}[x]$ in the data structure, so that the jump pointers are never NIL . This also has the advantage that the search can be started from anywhere, not just from the header, without changing the search algorithm, because in that case the exceptional jump pointers are followed automatically.

Remark. The list could be singly or doubly linked, although the version we present here is singly linked and is sufficient to express all of our algorithms. Moreover, the predecessor can be searched efficiently in a singly-linked jumplist, unlike singly-linked lists. All in all, we therefore have the following invariants:

- I1** (LIST) L is a singly- or doubly-connected list, with $\text{header}[L]$ a special node whose key is $-\infty$, $\text{key}[x] \leq \text{key}[\text{next}[x]]$ for every node $x \neq \text{header}[L]$.
- I2** (Jump forward) $x \prec \text{jump}[x]$ for every node, except the last node y for which $\text{jump}[y] = \text{next}[y] = \text{header}[L]$.
- I3** (Non-crossing jumps) for any two nodes $x \prec y$, either $x \prec y \prec \text{jump}[y] \prec \text{jump}[x]$, or $\text{jump}[x] \preceq y$, or $\text{jump}[y] = \text{jump}[x] = \text{next}[y]$.

As shown in [1] it is possible to verify the invariant in $O(n)$ time.

Let C be a jumplist node and let J and N be the nodes pointed by its jump and next pointers. The jumplist rooted at C will be denoted by the triple (C, J, N) or just by C if there is no ambiguity. The jumplists pointed by J and N are called the *jump* and *next sublists*.

2.2 Randomized jumplists

Following the randomized BST of Martinez and Roura [16], we define a randomized jumplist as a jumplist in which the jump pointer of the header takes any value in the list, and the jump and next sublists recursively have the randomized property.

In order to construct randomized jumplists, we need to augment the nodes with a size information. Each jumplist node is endowed with two fields jsize and nsize corresponding to the sizes of the two sublists. Thus $\text{jsize}[C] = \text{size}(J)$ and $\text{nsize}[C] = \text{size}(N)$, and for a jumplist rooted at C , we have

$$\text{size}(C) = 1 + \text{jsize}[C] + \text{nsize}[C].$$

Remark. The size of the jumplist therefore counts the header. If only the number of keys stored in the jumplist is desired, then subtract one.

2.3 Searching

As already pointed out, the basic search algorithm is *jump-and-walk*: follow the jump pointers if you can, otherwise continue with the linear search. Note that if

two arches $(x, \text{jump}[x])$ and $(y, \text{jump}[y])$ crossed, i.e. $x \prec y \prec \text{jump}[x] \prec \text{jump}[y]$, the second one would never be followed, since to reach y the key would be less than that of $\text{jump}[x]$, and by transitivity less than $\text{jump}[y]$. Note also that in order to be useful, arches should neither be too long (or they would never be used) nor too short (or else they would not speed up the search).

Several searching strategies are available depending on which pointers are tested first. Two search algorithms are presented on Figure 2. Our search algorithm is JUMPLIST-FIND-LAST-LESS-THAN-OR-EQUAL, and it returns the node y after which a key k should be inserted to preserve the list ordering. (Note that if there are many keys equal to k , this will return the last such key; inserting after that key preserves the list ordering as well, and the equal keys will be stored in the the order of their insertions). With this, testing if a key is present is easy: simply check whether $\text{key}[y] = k$.

To evaluate both strategies, let us compute the average and worst-case costs of accessing —at random with uniform probability— one of the n keys of a jumplist. If the jump and next pointers are tested in this order, accessing all the elements of the jump and next sublists, as well as the header respectively requires $n - i + 1$, $2(i - 2)$ and 3 comparisons (there is an extra comparison for testing if $\text{key}[y] = k$), if the jump pointer points to the i th element. This leads to an average of $(n + i)/n$ and a worst-case of 3. If the next and jump pointers are tested in this order, accessing any element always requires exactly 2 comparisons. We shall resort to the first solution which has a better average case.

<pre> JUMPLIST-FIND-LAST-LESS-THAN-OR- EQUAL(k) 1: $y \leftarrow \text{header}[L]$ 2: \triangleleft <i>Current node, never more than k</i> 3: while $\text{next}[y] \neq \text{header}[L]$ do 4: if $\text{key}[\text{jump}[y]] \leq k$ then 5: $y \leftarrow \text{jump}[y]$ 6: else if $\text{key}[\text{next}[y]] \leq k$ then 7: $y \leftarrow \text{next}[y]$ 8: else 9: return y 10: return y </pre>	<pre> JUMPLIST-FIND-LAST-LESS-THAN-OR- EQUAL(k) 1: $y \leftarrow \text{header}[L]$ 2: \triangleleft <i>Current node, never more than k</i> 3: while $\text{next}[y] \neq \text{header}[L]$ do 4: if $\text{key}[\text{next}[y]] \leq k$ then 5: if $\text{key}[\text{jump}[y]] \leq k$ then 6: $y \leftarrow \text{jump}[y]$ 7: else 8: $y \leftarrow \text{next}[y]$ 9: else 10: return y 11: return y </pre>
--	--

Fig. 2. Two implementation of the search algorithm (left) with $1 + \frac{i}{n}$ comparisons per node on average. (right) with exactly 2 comparisons per node.

2.4 Finding predecessor

If we change the inequalities in the algorithm JUMPLIST-FIND-LAST-LESS-THAN-OR-EQUAL to strict inequalities, then we obtain an algorithm JUMPLIST-FIND-LAST-LESS-THAN which can be used to find the predecessor of a node x with key k in the same time as a search. Thus, unlike linked lists, jumplists allow to trade off storage (one prev pointer) for predecessor access (constant vs. logarithmic time). One optimization for predecessor-finding is that, as soon as a node y such that $\text{jump}[y] = x$ is found, no more comparisons are needed: simply follow the jump pointers until $\text{next}[y] = x$.

2.5 Correspondence with Binary Search trees

From a structural point of view and if one forgets the labels of the nodes of a jumplist —i.e. the keys, there is a straightforward bijection between a jumplist and a binary tree: the jump and next sublists respectively correspond to the left and right subtrees. The jump sublist however is never empty, so that the number of unlabeled jumplists of a given size is expected to be less than the n -th Catalan number.

If one compares the key stored into a jumplist node with those of the jump and next sublists, a jumplist is organized as a heap-ordered binary tree and not a BST —the root points to two larger keys. Stated differently, the keys are stored not in the binary search tree order, but in preorder. But this bijection does not help much for insertion and deletion algorithms since there is no equivalent of rotation in jumplists. Rotations as performed for BST are actually impossible since the head node must contain the smallest value.

As is known in the folklore, BST can be searched with at most one comparison per node, at the cost of one extra comparison on the average at the end of the search (simply remember the last potential equal node along the path—for which the strict comparison fails, and test it when reaching a leaf). Unfortunately, the same trick cannot work for jumplists: consider a jumplist storing the keys $[1..n]$ for which $\text{jump}[i] = n + 1 - i$ for each $i < (n + 1)/2$. For the keys in $[1..(n + 1)/2]$, no jump pointer will be followed, so storing the last node during the search for which a jump pointer failed will not disambiguate between these keys, and the comparisons with the next pointers seem necessary in that case.

3 Expected performances of randomized jumplists

3.1 Internal path length, expected number of comparisons, jumplists profiles

We start the analysis of jumplists with the Internal Path Length (IPL) statistic. Similarly to BST, the IPL is defined as the sum of the depths of the internal nodes of the structure, the depth of a node being the number of pointers traversed from the header of the list. The analysis uses the so-called level polynomial and its associated bivariate generating function.

Definition 1. Let $s_{n,k}$ denote the expected number of nodes at depth k from the root in a randomized jumplist of size n . The level polynomial is defined by $S_n(u) = \sum_{k \geq 0} s_{n,k} u^k$. The associated bivariate generating function is defined by $S(z, u) = \sum_{n \geq 0} S_n(u) z^n$.

The expected value of the IPL is given by $S'_n(1)$, and can easily be extracted from $S(z, u)$. The bivariate generating function can also be used to study the distribution of the nodes' depths.

Internal Path Length. Let γ stand for the Euler constant and Ei denote the exponential integral function [2]. The following shows that the leading term of IPL for jumplists matches that of randomized BST [25, 15]:

Theorem 1. *The expected internal path length of a jumplist of size n is asymptotically equivalent to*

$$2n \ln n + n(-3 - 2Ei(1, 1) + e^{-1}) + 2 \ln n - 2Ei(1, 1) + e^{-1} + 3 + o(1).$$

Profile of a jumplist. Theorem 1 shows that the expected depth of a node is $2 \ln n$. We can actually be more precise and exhibit the corresponding distribution. This distribution is Gaussian, and its variance matches that of BST [15]:

Theorem 2. *The random variable X_n defined by $P(X_n = k) = s_{n,k}/S_n(1)$ is asymptotically Gaussian, with average $2 \ln n$ and variance $2 \ln n$.*

Expected number of comparisons along a search. The previous analysis investigates the number of pointers traversed from the root to the nodes of a jumplist. But the search cost of a key requires a more careful analysis since, according to the search algorithm JUMPLIST-FIND-LAST-LESS-THAN-OR-EQUAL of Figure 2(left), accessing the nodes of the jump and next sublists requires one and two comparisons, while accessing the root requires three comparisons. This accounts for the following

Definition 2. *Consider a jumplist C with n nodes, i.e. $n - 1$ user-defined keys together with the sentinel. The internal search length (external search length) $ISL(C)$ ($ESL(C)$) is defined as the total number of comparisons performed by all possible successful (unsuccessful) searches, i.e. needed to access the $n - 1$ user-defined keys (n intervals).*

Since the only difference between a successful and an unsuccessful search is that the former cannot reach the sentinel i.e. the node whose key is $-\infty$, and that accessing the first node of a jumplist requires three comparisons —recall that the jump and next pointers are checked first, we have:

Proposition 1. *The external and internal search lengths satisfy $ESL = ISL + 3$.*

Interestingly, ESL and ISL are within a constant as opposed to BST where one has $EPL = IPL + 2n$ with n the number of internal nodes. Using a variation of the analysis performed for the internal path length we get:

Theorem 3. *The expected number of comparisons ESL_n performed when accessing all the nodes of a jumplist of size n is asymptotically equivalent to*

$$ESL_n \sim 3n \ln n + n(-3 - 3Ei(1, 1) + 3e^{-1}) + 3 \ln n - 3Ei(1, 1) + 3e^{-1} + 9/2 + o(1).$$

It is not hard to see that algorithm JUMPLIST-FIND-LAST-LESS-THAN of Section 2.4 has an expected cost of $ESL_n + n + o(n)$, since in addition to the search, it follows all the jump pointers of the next sublist to arrive at the predecessor. The number of comparisons is the same, however, with the optimization mentioned at the end of Section 2.4.

4 Insertion and deletion algorithms

So far, we have shown that the performance of randomized jumlists are as good as those of randomized BST. We now show how to maintain the randomized property upon insertions and deletions. This maintenance will make the performance of jumlists identical for random keys or sorted keys—a property similar to that of randomized Binary Search Trees [16].

4.1 Creating a jumplist from a sorted linked list

Constructing a jumplist from a list is very simple: we only have to choose the jump pointer of the header, and recursively build the next and jump sublists. We use a recursive function $REBALANCEINTERVAL(L, x, n)$, which creates a randomized jumplist for the n elements of L starting at x , $next[x]$, \dots , $z = next^{n-1}[x]$. The element $y = next^n[x]$ acts as a sentinel (the last element z jumps to it, but $jump[y]$ is not set). It is this element y which is returned. Hence:

```
JUMPLISTFROMLIST( $L$ )
1:  $y \leftarrow header[L]$ 
2:  $n \leftarrow 1 + nsize[y] + jsize[y]$ 
3:  $REBALANCEINTERVAL(y, n)$ 
 $REBALANCEINTERVAL(x, n)$ 
1: while  $n > 1$  do
2:    $m \leftarrow$  random number in  $[2, n]$ 
3:    $y \leftarrow next[x]$ 
4:    $jump[x] \leftarrow$ 
5:      $REBALANCEINTERVAL(y, m-2)$ 
6:    $x \leftarrow jump[x]$ 
7:    $n \leftarrow n - m + 1$ 
8: return  $x$ 
```

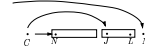


Fig. 3. Insertion: notation.

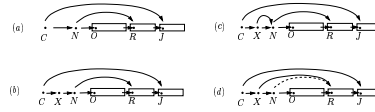


Fig. 4. Usurping arches

4.2 Maintaining the randomness property upon an insertion

Suppose as depicted on Figure 3 that we aim at inserting the key x into the jumplist (C, J, N) , and let X be the jumplist node to be allocated in order to accommodate x . The general pattern of the insertion algorithm is the search algorithm of Figure 2(left) since we need to figure out the position of x . But on the other hand we have to maintain the randomness property. We now describe algorithm JUMPLISTINSERT. Refer to Figure 3 for the notation, and to [1] for the pseudo-code.

When inserted into the list rooted at C , the node X containing x becomes a candidate as the endpoint of the fundamental arch starting at C . Assume that $\text{jsize}[C] + \text{nsiz}[C] = n - 1$. Since x is inserted into (C, J, N) , X has to be made the fundamental arch of $C \cup X$ with probability $1/n$. (Notice that if C is the end of the list, this probability is one so that we create a length one arch between the last item of the list and X , and exit.) With probability $1 - 1/n$ the fundamental arch of C does not change. If x is more than the keys of J or N , we recursively insert into the jump or next sublist. If not, x has to be inserted right after C , that is X becomes the successor of C and the randomness property must be restored for the jumplist rooted at X . To summarize, the recursive algorithm JUMPLISTINSERT stops when one of the following events occur:

- Case 1: x is inserted in the list rooted at C , and $[C, X]$ becomes the new fundamental arch. The randomness property of the list rooted at C , that is $C \cup X$, has to be restored.
- Case 2: x is inserted right after C . The randomness property of the list rooted at X that is $X \cup N$ has to be restored.

Following this discussion, we have the following

Proposition 2. *Algorithm JUMPLISTINSERT maintains the randomness property of the jumplist under insertion of any key.*

We proceed with the complexity of algorithm JUMPLISTINSERT. The reorganization to be performed in cases 1 and 2 consists of maintaining the randomness property of a jumplist whose root and size are known. This can be done using the algorithm JUMPLISTFROMLIST of section 4.1. Alas, algorithm JUMPLISTFROMLIST has linear complexity and the following observation shows that applying JUMPLISTFROMLIST to cases 1 and 2 is not optimal.

Observation 4 *Let C be a randomized jumplist of size n and suppose that x has to be inserted right after C . The expected number N_n of keys involved in the restoration of the randomness property of $C \cup X$ satisfies $N_n \sim n/2$.*

4.3 Insertion algorithm

As just observed, inserting a key after the header of the list may require restoring the randomness property over a linear number of terms. We show that this can

be done at a logarithmic cost. The intuition is that instead of computing from scratch a new randomized jumplist on $X \cup N$, and since by induction N is a randomized jumplist, one can reuse the arches of N in order to maintain randomness. More precisely, we shall make X usurp N and proceed recursively.

Algorithm USURPARCHES runs as follows. Due to the lack of space, we also omit the pseudo-code.

As depicted on Figure 4(a,b), assume that x is inserted after C and that prior to this insertion C has a successor N which is the root of the jumplist (N, O, R) . Let n be the size of N . With probability $1/n$ we just create the length one arch $[X, N]$ —Figure 4(c). If not and with probability $1 - 1/n$, the arch of X has to be chosen as any of the nodes in the jumplist rooted at N . But since by induction hypothesis (N, O, R) is a randomized jumplist, X can usurp the arch of N —Figure 4(d). The next sublist of X then has to be re-organized. We do so recursively by having N usurp its successor.

Following the previous discussion we have the following

Proposition 3. *Algorithm USURPARCHES maintains the randomness property of the jumplist.*

Analyzing the complexity of algorithm USURPARCHES requires counting the number of jump pointers updates along the process. We have:

Proposition 4. *Let X be a jumplist node whose next sublist (N, O, R) has size n . The expected number S_n of jump pointers updates during the recursive usurping strategy starting at X satisfies $S_n \sim \ln n$.*

We are now ready to prove the main result of this section:

Theorem 5. *Algorithm JUMPLISTINSERT using algorithm JUMPLISTFROMLIST for Case 1 and algorithm USURPARCHES for Case 2 returns a randomized jumplist. Moreover, its complexity is $O(\log n)$.*

4.4 Deletion algorithm

We show in this section that removing a key from a jumplist is exactly the symmetric operation of the insertion. Assume key x has to be removed from a jumplist C . To begin with, the node containing x has to be located. To do so, we search for x as usual following the pattern of the search algorithm of Figure 2(left). To be more precise, we actually seek the node C such that $\text{key}[\text{jump}[C]] = x$ or $\text{key}[\text{next}[C]] = x$. The actual removal of the node X containing x distinguishes between the following two situations.

Removing X with $\text{key}[\text{jump}[C]] = x$. First, X is removed from the linked list. If we assume a singly connected linked list is used, the removal of X requires the knowledge of its predecessor, but this can be obtained by using the algorithm JUMPLIST-PREDECESSOR(X) of section 2. Second, the system of arches starting at X needs to be recomputed. To do so, we again use the function JUMPLISTFROMLIST.

Removing X with $\text{key}[\text{next}[C]] = x$. Here too, we first remove X and second maintain the randomness property of the next sublist of C . The former operation is trivial. For the second one, first observe that if we have a length one arch, i.e. $\text{jump}[X] = \text{next}[X]$, the situation is trivial too. Consider the situation where this is not the case. To create a random arch rooted at N , we recursively unwind the usurping operation described for the insertion algorithm. The operations performed are exactly the opposite of those described for the usurping algorithm, and the expected complexity is also logarithmic.

5 Conclusion

In this paper, we have presented a data structure called jumplist, which is inspired by skip lists and by randomized binary search trees, and which shares many of their properties.

There are a few advantages to randomized jumplists over randomized BST and treaps, the main one being the low storage used (if only forward traversal is needed; note that BST require parent pointers in order to provide the successor operation), and that the traversal is very simple (simply follow the underlying list, in $O(1)$ worst-case time per element). Moreover, it is conceivable to avoid storing the sublist sizes, but the insertion is more involved and does not have the randomness property. It is an open challenge to design deterministic jumplists. Determinism would likely make the data structure faster, and can probably be achieved by weight balancing (since the sublist sizes are known). Yet we have not carried it to its conclusion. Another challenge is to identify a splay operation on jumplists (in the manner of splay trees) in order to provide good amortized performance.

The main advantage to jumplists over skip lists is the size requirement: the jumplist stores exactly one jump pointer per node, whereas this number is not constant for skip lists (although the total expected storage remains linear).

Thus, jumplists provide an alternative to the classical dictionary data structures, and like skip lists, they have the potential to extend for higher-dimensional search structures in computational geometry [9]. Anecdotically, this is the reason we started to investigate jumplists. We plan to continue our research in this direction.

Acknowledgments. Philippe Flajolet, Marc Glisse and Bruno Salvy are acknowledged for discussions on the topic.

References

1. Hervé Brönnimann and Frédéric Cazals and Marianne Durand. Randomized jumplists: A jump-and-walk dictionary data structure. Research Report RR-xxxx, INRIA, 2003.
2. M. Abramowitz and I. A. Stegun. *Handbook of Mathematical Functions*. Dover, 1973. A reprint of the tenth National Bureau of Standards edition, 1964.

3. N. Amenta, S. Choi, T. K. Dey, and N. Leekha. A simple algorithm for homeomorphic surface reconstruction. In *Proc. 16th Annu. ACM Sympos. Comput. Geom.*, pages 213–222, 2000.
4. Nina Amenta and Marshall Bern. Surface reconstruction by Voronoi filtering. *Discrete Comput. Geom.*, 22(4):481–504, 1999.
5. C. Aragon and R. Seidel. Randomized search trees. In *Proc. 30th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 540–545, 1989.
6. Jean-Daniel Boissonnat and Frédéric Cazals. Smooth surface reconstruction via natural neighbour interpolation of distance functions. In *Proc. 16th Annu. ACM Sympos. Comput. Geom.*, pages 223–232, 2000.
7. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.
8. Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, Germany, 2nd edition, 2000.
9. Olivier Devillers. Improved incremental randomized Delaunay triangulation. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 106–115, 1998.
10. P. Flajolet and A. M. Odlyzko. Singularity analysis of generating functions. *SIAM J. Disc. Math.*, 3(2):216–240, 1990.
11. G. H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures*. Addison-Wesley, 1991.
12. D. H. Greene and D. E. Knuth. *Mathematics for the analysis of algorithms*. Birkhäuser, Boston, 1982.
13. H.-K. Hwang. *Théorèmes limites pour les structures combinatoires et les fonctions arithmétiques*. PhD thesis, École Polytechnique, Palaiseau, France, December 1994.
14. Donald E. Knuth. *The Art of Computer Programming, Vol 3., 2nd Edition*. Addison-Wesley, 1998.
15. H.M. Mahmoud. *Evolution of random search trees*. Wiley, 1992.
16. C. Martínez and S. Roura. Randomized binary search trees. *J. Assoc. Comput. Mach.*, 45(2), 1998.
17. J.I. Munro, T. Papadakis, and R. Sedgewick. Deterministic skip lists. In *SODA*, Orlando, Florida, United States, 1992.
18. W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.
19. S. Roura. An improved master theorem for divide-and-conquer recurrences. *J. Assoc. Comput. Mach.*, 48(2), 2001.
20. R. Sedgewick. *Algorithms in C++, Parts 1-4: Fundamentals, Data Structure, Sorting, Searching*. Addison-Wesley, third edition, 1998.
21. R. Sedgewick and P. Flajolet. *An Introduction to the Analysis of algorithms*. Addison-Wesley, 1996.
22. R. Sedgewick and P. Flajolet. Analytic combinatorics—symbolic combinatorics. To appear, 2002.
23. R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16:464–497, 1996.
24. D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.
25. J.S. Vitter and P. Flajolet. Average-case analysis of algorithms and data structures. In J. van Leeuwen, editor, *Algorithms and Complexity*, volume A of *Handbook of Theoretical Computer Science*, pages 432–524. Elsevier, Amsterdam, 1990.
26. N. Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall, 1975.