Randomized Jumplists—A Jump-and-Walk Dictionary Data Structure

Hervé Brönnimann* Frédéric Cazals† Marianne Durand‡

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 very easy. Jumplists are a compact data structure. (To the best of our knowledge, jumplists are the only data structure providing rank-based operations and iterators at a cost of 12 bytes per node.) Jumplists are trivially built in linear time from sorted linked lists.

In addition to the presentation of jumplists and the probabilistic analysis of their expected performance, the paper presents a detailed experimental study of jumplists against Red-Black trees, randomized BST, and splay trees.

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 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 [10, 9] as well as deterministic balanced BST [19, 9, 3]. Since then, solutions more geared to provably good amortized or randomized performance were proposed. Splay trees [17], treaps [2], skip lists [12] and more recently randomized BST [11] fall into this category. An important feature of the randomized data structures —in particular randomized BST and skip lists— is their ease of implementation.

This paper presents a one-dimensional 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 are a *flat* data structure, in the sense that there is no hierarchy. This means in particular that we use deterministic storage. A jumplist for n user defined keys requires $n+1$ nodes i.e. $2(n+1)$ pointers. (As we shall see later, the extra node is a sentinel.) 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.

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

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

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

Their performance are within a constant factor of optimal BST. Randomization makes their dynamic maintenance very easy. Jumplists are a compact data structure. (To the best of our knowledge, jumplists are the only data structure providing rank-based operations and iterators at a cost of 12 bytes per node.) Jumplists are trivially built in linear time from sorted linked lists. At last, it is expected that jumplists will prove to be versatile and will adapt to geometric problems. It should indeed be recalled that elaborating upon skip lists resulted in the fastest incremental Delaunay triangulation algorithm known to date [4].

Overview. This paper is organized as follows. Section 2 overviews some important properties of BST. The jumplist data structure and its search algorithms are described in Section 3. Section 4 analyzes the expected performance of the data structure. Dynamic maintenance of jumplists is presented in Section 5. Implementation details and experimental results are presented in Section 6.

2 Searching a Binary Search Tree

This section overviews some important properties of search algorithms for Binary Search Trees (BST).

2.1 BST and path lengths

The performance of BST (and especially random BST) are usually analyzed in terms of Internal and External Path Lengths —IPL and EPL, see [10, 15]. Although it is usually assumed that the cost of a successful search (unsuccessful) is measured by IPL (EPL), we point out here than in order for this to hold, some care must be devoted to the implementation of the search functions.

To see why, first recall that the depth of a node in a BST is the number of pointers traversed to reach it from the root of the tree. Also recall that IPL (EPL) is defined the sum of the depths of the internal (external) nodes. When searching for a key k from a node x of the tree, three situations arise: $k < \text{key}[x], k = \text{key}[x]$, or $k > \text{key}[x]$. Since tests are binary operators, choosing from these three situations requires two comparisons in the worst-case. For example, one can choose from the left subtree, the root or the right subtree by first checking whether $k < \text{key}[x]$, and then in case of failure whether $k = \text{key}[x]$. Algorithms BST-Search provided by most textbooks [3, 6, 10, 14] proceed this way. But the number of comparisons along a search might differ from the path length which counts a single operation per node.

The exact relationship between IPL and the actual number of comparisons performed depend on the profile of the tree. Consider for example a random BST traversed by the search algorithm just outlined. It is then easily checked that the expected cost of a search (successful or not) is equivalent to $3 \ln n$, while IPL/n and $EPL/(n+1)$ are equivalent to $2 \ln n$. (This validates the informal reasoning that the search algorithm as implemented above makes 1.5 comparisons per node on the average.)

A subtle modification of BST-Search, however, reconciles the path lengths and the number of comparisons:

Observation 1 *One can search a BST with at most one comparison per node, plus one extra final comparison.*

Let's say, for instance, that the algorithm tests whether $k > \text{key}[x]$: if this comparison fails, then the search goes into the left subtree; if $k = \text{key}[x]$, then every comparison in the left subtree will succeed, and the search will end up in the leaf which is the predecessor of x. Hence it suffices to remember the last node during the search for which the comparison fails, and to perform the equality test only once at the end of the search. This lengthens the search a bit, but not by much (the number of nodes visited for a random key is increased by exactly one on the average), and is made up for by the smaller number of comparisons performed (one per node visited, plus one final equality test). It appears that it is this version that is analyzed in [18, Thm. 4.13].

2.2 BST and rank-based algorithms

Data structures enabling the retrieval of the k-th order value are called order-statistic trees. If the nodes of a BST are endowed with an integer value coding the size of the tree, a OS-Select can easily be implemented [3].

For random BST [16], the cost of searching the k-th element in a tree of size n is $H_k + H_{n-k}$ —with $H_k = \sum_{i=1}^{k} 1/i$ the k-th harmonic number. This cost function is symmetric and equivalent to $2 \ln n$ when $k = O(n)$ and $n - k = O(n)$.

3 Jumplist: data structure and searching algorithms

3.1 The data structure

The jumplist is stored as a singly or doubly connected list, with next$[x]$ pointing forward and 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 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 $[\mathrm{key}[x], \mathrm{key}[\mathrm{next}[x]]]$ 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 $\mathrm{key}[x] < \mathrm{key}[y]$, and $x \preceq y$ iff $\mathrm{key}[x] \leq \mathrm{key}[y]$. When some keys are identical, it is inefficient to test whether $x \prec y$ when $\mathrm{key}[x] = \mathrm{key}[y]$ (one must basically traverse the list).

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

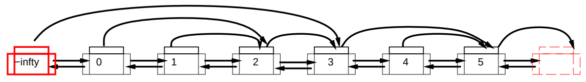

Figure 1: A possible jumplist data structure over 6 elements. The last node (in dashed lines) is identical to the first one on this representation.

In addition to the list pointers, each node also has a pointer jump$[x]$ which points to a successor of x in the list. We refer to the pair $(x, \mathrm{jump}[x])$ as an *arch*, and to the arch starting at header$[L]$ as the *fundamental arch*. As mentioned earlier, the jump pointers have to satisfy $x \prec \mathrm{jump}[x]$ 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 $x \prec y \prec \mathrm{jump}[x] \prec \mathrm{jump}[y]$. Thus if $x \prec y$, then either $y \prec \mathrm{jump}[y] \prec \mathrm{jump}[x]$ (*strictly nested*), $\mathrm{jump}[x] \preceq y \prec \mathrm{jump}[y]$ (*semi-disjoint*), or $y \prec \mathrm{jump}[y] = \mathrm{jump}[x]$ if $\mathrm{jump}[y] = \mathrm{next}[y]$ (*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 $y = \mathrm{prev}[\mathrm{jump}[x]]$, for nodes x such that $\mathrm{next}[x] \prec \mathrm{jump}[x]$. The value $\mathrm{jump}[y] = \mathrm{next}[y]$ is the only one that does not break a non-strictly nested condition. We could also get rid of exceptional pointers by putting $\mathrm{jump}[x] = \textsc{nil}$ in that case. This has the advantage of making the connection with binary trees explicit (see next section), but also complicates the algorithms which would 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 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 header$[L]$ a special node whose key is $-\infty$, $\mathrm{key}[x] \leq \mathrm{key}[\mathrm{next}[x]]$ for every node $x \neq \mathrm{header}[L]$.

I2 (Jump forward) $x \prec \mathrm{jump}[x]$ for every node, except the last node y for which $\mathrm{jump}[y] = \mathrm{next}[y] = \mathrm{header}[L]$.

I3 (Non-crossing jumps) for any two nodes $x \prec y$, either $x \prec y \prec \mathrm{jump}[y] \prec \mathrm{jump}[x]$, or $\mathrm{jump}[x] \preceq y$, or $\mathrm{jump}[y] = \mathrm{jump}[x] = \mathrm{next}[y]$.

Although we will not detail the algorithm here, it is possible to verify the invariant in $O(n)$ time by a simple recursive traversal.

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.*

3.2 Randomized jumplists

Following the randomized BST of Martinez and Roura [11], 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.

3.3 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 ith 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.

3.4 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 $\text{jump}[y]$ is found, no more comparisons are needed: simply follow the jump pointers until $\text{next}[y] = x$.

3.5 Rank-based algorithms

Since we have information about the jsize and nsize of each node, our jumplist behaves like an order-statistic tree [3] and can be used to efficiently retrieve the k-th element by its rank k, $1 \le k \le n$. The algorithm Jumplist-Nth-Element(C, k) returns the k-th element contained in the sublist of C (provided $1 \le k \le \text{size}(C)$). This algorithm will be analyzed in section 4.

```
Jumplist-Nth-Element($k$)
1: $y \leftarrow \text{header}[L]$, $n \leftarrow 1 + \text{nsize}[y] + \text{jsize}[y]$
2: $\triangleleft$ $k$ must be $1 \le k \le n$
3: while $k > 1$ do
4:     if $k \le 1 + \text{nsize}[y]$ then
5:         $y \leftarrow \text{jump}[y]$, $k \leftarrow k - 1 - \text{nsize}[k]$
6:     else
7:         $y \leftarrow \text{next}[y]$, $k \leftarrow k - 1$
8: return $y$
```

3.6 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.

At last a comment is in order regarding searching performances. As observed in the previous section, BST can be searched with at most one compar-

```
JUMPLIST-FIND-LAST-LESS-THAN-OR-EQUAL($k$)
 1: $y \leftarrow$ header[$L$]
 2: ◁ Current node, never more than $k$
 3: while next[$y$] $\neq$ header[$L$] do
 4:     if key[jump[$y$]] $\leq k$ then
 5:         $y \leftarrow$ jump[$y$]
 6:     else if key[next[$y$]] $\leq k$ then
 7:         $y \leftarrow$ next[$y$]
 8:     else
 9:         return $y$
10: return $y$
```

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

Figure 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.

ison per node. Unfortunately, the same trick cannot work for jumplists: consider a jumplist storing the keys $[1..n]$ for which 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.

We shall prove however that the number of comparisons along a jumplist search is equivalent to $3 \ln n$, i.e. the same value than for BST if CONTAINS is implemented without the refinement discussed.

4 Searching a jumplist: expected performances

This section investigates the expected performance of randomized jumplists.

4.1 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 2 *Let s_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 $S_n(u) = \sum_{k \geq 0} s_k\, u^k$. 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 [1]. The following shows that the leading term of IPL for jumplists matches that of randomized BST [18, 10]:

Theorem 3 *The expected internal path length of a jumplist of size n is asymptotically equivalent to*

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

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

Proof. Consider a jumplist of size n. Since the jump sublist has size k, $1 \leq k \leq n-1$, with probability $\frac{1}{n-1}$, the sequence $S_n(u)$ satisfy the recurrence equation

$$S_n(u) = 1 + \frac{u}{n-1} \left(\sum_{k=1}^{n-1} S_k(u) + S_{n-k-1}(u) \right). \quad (1)$$

The generating function $S(z,u)$ satisfies the differential equation (directly translated from the recurrence) with respect to the variable z:

$$S'(z,u) - S(z,u) \left(\frac{1}{z} + u \frac{1+z}{1-z} \right) = \frac{z}{(1-z)^2}. \quad (2)$$

This differential equation is solved with the variation of constant method and we get:

$$S(z,u) = \frac{z e^{-uz}}{(1-z)^{2u}} \left(1 + \int_0^z (1-t)^{2(u-1)} e^{ut} dt \right). \quad (3)$$

To get the expected IPL, S is first differentiated with respect to u then evaluated at $u = 1$. The coefficient of z^n is obtained by using the dictionary [5]. □

Expected number of comparisons along a search. A variation of the previous analysis provides the expected number of comparisons required by the algorithm Jumplist-Find-Last-Less-Than-Or-Equal of Figure 2(left):

Theorem 4 *The expected number of comparisons* JLC_n *performed when searching all the keys of a jumplist of size* n *is asymptotically equivalent to*

$$\mathrm{JLC}_n \sim 3n\ln n + n(-3 - 3Ei(1,1) + 3e^{-1})$$

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

Proof. Algorithm Jumplist-Find-Last-Less-Than-Or-Equal checks sequentially the jump pointer, the next pointer, and the root of the jumplist. The costs of accessing the keys of the jump and next sublists are therefore 1 and 2 comparisons, while that of accessing the root is 3 comparisons. The corresponding level polynomial is

$$S_n(u) = u^3 + \frac{1}{n-1}\left(\sum_{k=1}^{n-1} uS_k(u) + u^2 S_{n-k-1}(u)\right).$$

Computing $S(z,u)$ and extracting the coefficients as done for IPL completes the proof. □

It is not hard to see that algorithm Jumplist-Find-Last-Less-Than of Section 3.4 has an expected cost of $\mathrm{JLC}_n + n + o(n)$, 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 3.4.

Profile of a jumplist. Theorem 3 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 [10]:

Theorem 5 *The random variable* X_n *defined by* $\mathrm{P}(X_n = k) = S_{n,k}/S_n(1)$ *is asymptotically Gaussian, with average* $2\ln n$ *and variance* $2\ln n$.

Proof. The asymptotic expansion of the generating function S in a neighborhood of $u = 1$ gives information about the limit law of the profile of the jumplist. As

$$S_n(u) \sim \frac{e^{-u}(\lambda(u)+1)}{\Gamma(2u)} n^{2u-1} + O(n^{2u-2}), \qquad (4)$$

uniformly in a neighborhood of $u = 1$, with $\lambda(u) = \int_0^1 (1-t)^{2(u-1)} e^{ut} dt$, using the quasi-power theorem [8] completes the proof. □

4.2 Searching the k-th element

Consider algorithm Jumplist-Nth-Element from section 3. Its cost is the number of pointers traversed when searching for the k-th element of a jumplist of size n.

Lemma 6 *Let* $F_{n,k}$ *denote the expected number of pointers traversed in a randomized jumplist of size* n *when searching the* k*-th element with algorithm* Jumplist-Nth-Element. *The bivariate generating function* $F(z,u) = \sum_{n,k} F_{n,k} u^k z^n$ *is given by*

$$F(z,u) = \frac{ze^{-uz}}{(1-z)^u(1-zu)} \int_0^z (1-t)^u (1-tu) e^{ut}$$

$$\frac{u^2}{1-u}\left(\frac{1}{(1-t)^2} - \frac{u}{(1-tu)^2}\right) dt.$$

Proof. Assume that we want to retrieve the k-th element in a jumplist of size n whose fundamental arch is $[1,i]$. Depending on the relative values of k and i, the k-th node has to be sought in the jump or next sublists, whence the following recurrence for $F_{n,k}$:

$$\begin{aligned} F_{n,1} &= 0, \\ F_{n,k} &= 1 + \frac{1}{n-1}\sum_{i=2}^{k-1} F_{n-i+1,k-i+1} \\ &+ \frac{1}{n-1}\sum_{i=k+1}^{n} F_{i-2,k-1}. \end{aligned}$$

Using the bivariate generating function $F(z,u)$, this recurrence translates into the following differential equation

$$zF'_z(z,u) - F(z,u)\left(1 - \frac{z^2u}{1-z} + \frac{zu}{1-zu}\right)$$

$$= \frac{z^2u^2}{(1-u)(1-z)^2} - \frac{u^3z^2}{(1-u)(1-zu)^2}.$$

This equation is solved using the variation of the constant method, which yields the result. □

The classical techniques of singularity analysis can not be applied here to extract the coefficient of z^n (and then of u^k), because there are two singularities, located in 1 and $1/u$ that cannot be separated as u is meant to tend to 1. A simple example of this problem is the function $F(z,u) = \frac{1}{1-z}\frac{1}{1-zu}$. Its expansion is very simple, $\sum_n \sum_{k\le n} z^n u^k = \sum_n \frac{1-u^{n+1}}{1-u} z^n$, but an attempt to misuse singularity analysis would lead to results like $[z^n]F(z,u) = u^n$, if u is assumed > 1, $[z^n]F(z,u) = n$ if $u = 1$ and $[z^n]F(z,u) = 1/(1-u)$ otherwise. The results are not coherent in a neighborhood of $u = 1$. Trying to extract first the coefficient of u^k and then of z^n would lead to other incorrect results.

The method used to obtain the asymptotic of the coefficients $F_{n,k}$ is a technical study of the expansion of the generating function. This study leads to the theorem:

Theorem 7 *Asymptotically, the cost of accessing the k-th element in a list of size n, $F_{n,k}$ is equivalent to* $2\ln n$ *if* $3\ln n \leq k \leq n - 3\ln n$.

The proof of this theorem is a systematic study of the behavior of the coefficients of all the generating functions factors of $F(z,u)$. Everything is based on the fact that the Stirling coefficients of the first kind $\left[{n \atop k}\right]$ (as denoted in [7]) are very concentrated around their mean value. More precisely the distribution of the coefficients $\left[{n \atop k}\right]_{0\leq k\leq n}$ is Gaussian when n tends to infinity [16]. This property is not sufficient when k is either too small or too large. The precise statements and proof are fairly technical and omitted in this abstract.

When $k \leq 3\ln n$, the cost of the k-th element $F_{n,k}$ is bounded by k. So we expect that the cost is at first linear (as all the jump pointers are useless for the first elements), and then tends to be constant at $2\ln n$. When $k \geq n - 3\ln n$, we expect that the cost behaves better than if there was a stopover around $n - 3\ln n$, that would give $2\ln n + 2\ln\ln n$ as an upper bound for the cost. In fact experiments seem to show that the cost is slightly decreasing when k becomes close to n.

This result should be compared against its party for BST as recalled in section 2. The complexity is essentially the same, yet better for $k < 3\ln n$. [1]

5 Insertion and deletion algorithms

So far, we have shown that the performances of randomized jumplists 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 performances of jumplists identical for random keys or sorted keys —a property similar to that of randomized Binary Search Trees [11].

[1]In all fairness, it is possible to implement OS-Select in BST with the same time complexity, but the procedure is then markedly more complex: it involves maintaining a pointer to the min element, and starting the search there, while always going to parent$[x]$ from the current node x if $k >$ size$[x]$. The same idea can be used for starting the search at a node of known rank.

5.1 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$[x]$, ..., $z =$ 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: RebalanceInterval($y, 1 +$ nsize[$y$] $+$ jsize[$y$])

RebalanceInterval($x, n$)
 1: while $n > 1$ do
 2:     $m \leftarrow$ random number in $[2, n]$
 3:     jump[$x$] $\leftarrow$RebalanceInterval(next[$x$], $m - 2$)
 4:     $x \leftarrow$ jump[$x$]
 5:     $n \leftarrow n - m + 1$
 6: return $x$
```

Consider a jumplist of size n and assume the fundamental arch is $[1, m]$ with $m = \lfloor\frac{n+1}{2}\rfloor + 1$. The sizes of the jump and next sublists are $n_{jump} = n - m + 1 = \lfloor n/2\rfloor$ and $n_{next} = m - 2 = \lceil n/2\rceil - 1$. Such a jumplist is called *perfectly balanced* since the sizes n_{jump} and n_{next} differ by at most one. Algorithm RebalanceInterval can easily be modified so as to return a balanced jumplist by having m be set to $\lfloor\frac{n+1}{2}\rfloor + 1$ instead of being chosen at random.

5.2 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.

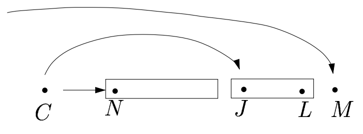

Figure 3: Insertion: notation.

We now describe algorithm JumplistInsert. Due to the lack of space, we omit the pseudo-code. Refer to Figure 3 for the notation.

When inserted into the list rooted at C, the node X containing x becomes a candidate as the endpoint of the fundamental arch stating at C. Assume that $\mathrm{jsize}[C] + \mathrm{nsize}[C] = n - 1$. 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 8 *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 5.1. Alas, algorithm JumplistFromList has linear complexity and the following observation shows that applying Jumplist-FromList to cases 1 and 2 is not optimal.

Observation 9 *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 $N_n \sim n/2$.*

Proof. With probability $1/n$, Case 1 applies and we rebuild the whole list. With probability $1-1/n$, Case 2 applies and insertion is performed after C. Whence an expected number of keys

$$N_n = \frac{1}{n}n + \left(1 - \frac{1}{n}\right)\frac{1}{n-1}\sum_{i=2}^{n}(i-2),$$

which solves to $N_n \sim n/2$. □

5.3 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 10 *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 11 *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 $S_n \sim \ln n$.*

Proof. With probability $1/n$ we just create the length one arch $[X, N]$, whence an expected cost of $1/n$. With probability $1 - 1/n$, X usurps N —one jump pointer update, and the recursive algorithm proceeds. The expected cost of a recursive call has to be computed wrt the position of the fundamental arch, whence the recurrence

$$\begin{aligned} S_n &= \frac{1}{n}1 + \left(1 - \frac{1}{n}\right)\frac{1}{n-1}\sum_{i=2}^{n}(1 + S_{i-2}) \\ &= 1 + \left(1 - \frac{1}{n}\right)\frac{1}{n-1}\sum_{i=2}^{n}S_{i-2}. \end{aligned}$$

The equivalent follows from the application of the continuous master theorem of [13]. □

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

Theorem 12 *Algorithm* JUMPLISTINSERT *using algorithm* JUMPLISTFROMLIST *for Case 1 and algorithm* USURPARCHES *for Case 2 returns a randomized jumplist. Moreover, its complexity is* $O(\log n)$.

Proof. The correctness stems from propositions 8 and 10.

For the complexity, let C_n denote the complexity of JUMPLISTINSERT and assume for the induction that C_n is $O(\log n)$. Upon insertion of X and following the instructions of algorithm JUMPLISTINSERT, we have to examine the following situations:
–Case 1: jumplist rebuilt. Since algorithm JUMPLISTFROMLIST is used to restore the randomness property, the cost incurred is $O(n)$.
–Recursive insertion: x is inserted recursively into the jump or next sublist. Since since we do not make any assumption on x, we actually do not know which situation arises. But by induction hypothesis and letting i be the position of the fundamental arch, the corresponding costs are $\log(n-i+1)$ or $\log(i-2)$.
–Case 2: x inserted after C. If i denotes the position of the fundamental arch and since algorithm USURPARCHES is used, the corresponding cost is $O(\log(i-1))$.

Let $1_{x\in J}$ ($1_{x\in N}$) be the boolean variable whose value is one if x is inserted into the jump (next) sublist, and zero otherwise. Define $1_{x\in C}$ similarly. Weighing the three events just listed the $1/n$ and $1-1/n$ probabilities yields the following recurrence

$$\begin{aligned} C_n &= \frac{1}{n}O(n) + \left(1-\frac{1}{n}\right)\frac{1}{n-1} \\ &\quad \sum_{i=2}^{n}\big[1_{x\in J}\ O(\log(n-i+1) \\ &\qquad +1_{x\in N}\ O(\log(i-2)) \\ &\qquad +1_{x\in C}\ O(\log(i-1))\big]. \end{aligned}$$

An upper bound on the operand of the sum is $\max(O(\log(n-i+1), O(\log(i-1))$, from which the induction is worked out easily. □

5.4 Deletion algorithm

We show in this section how to remove keys from jumplists. 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

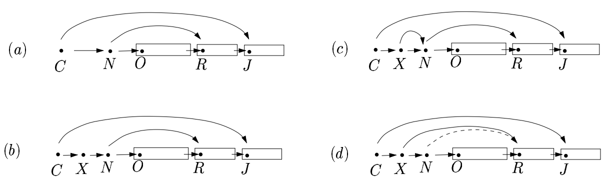

Figure 4: Usurping arches

C such that key[jump[C]] $= x$ or key[next[C]] $= x$. The actual removal of the node X containing x distinguishes between the following two situations.

Removing X with key[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 JUMPLISTPREDECESSOR(X) of section 3. Second, the system of arches starting at X needs to be recomputed. To do so, we again use the function JUMPLISTFROMLIST.

Removing X with key[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. jump[X] $=$ 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. Since the operations to be performed are exactly the opposite of those described for the usurping algorithm, we omit them in this version. The expected complexity is also logarithmic.

6 Experimental results

There are many variants and choices to make when implementing tree-like data structures. Sedgewick's book [14] includes a nice discussion of the possible search structures, and concludes by saying that red/black trees are the best suited for a general-purpose library implementation. Indeed, red/black trees are the structure of choice in many implementations of the C++ STL `set` and `map` structures. We based our implementation of red/black trees on the publically available SGI STL implementation [], which is based on [3] and is highly optimized. In addition, the search procedure uses the observation 1 from section 2. For augmented red/black treees and splay trees, we simply maintain the subtree sizes. For randomized BST and compact jumplists, we use a trick or Roura [11] and store only one of the two substructure sizes (and a bit to remember which): this avoids the problem of incrementing sizes when the key is alreay present and no insertion will occur — we simply flip the bit if necessary to store the size of the substructure that does not contain the key to be inserted.

We perform some experiments to verify the claims about the performance of the three kinds of data structures (jumplists, randomized and red/black BST). The complete experimental setup is described in the appendix. Moreover, the code will be available online from the first author's web page. In order to make sure that the random generator works correctly for randomized trees and jumplists, we gather the number of links followed by the search procedure, as well as the number of comparisons performed. We observe indeed that the values we get are in accordance to the expected values derived in Section 4. The complete results and data structures are given in the appendix. A more thorough comparison of dictionary data structures including the results here will be submitted to ALENEX. Here we summarize only the main points which can be derived from those results.

First, the node size ranges between 3 and 5 pointers (assuming an integral size takes the same storage as a pointer), not including the storage needed to store the information. The only data structure to provide 3-pointer nodes and rank-based operations is the singly-linked jumplist in which only one of the sublist sizes is stored (and a bit compacted in the parent pointer). If the value is a pointer and the key is dereferenced from it, this means a savings of 20% in storage compared to randomized binary trees or augmented red/black trees, and 33% compared to treaps or similar data structures.

It turns out that maintaining only one of the size fields in jumplists saves one pointer of storage (same applies to randomized BST) and also *improves* performance (probably because the memory cache can be better utilized). Likewise, going from doubly-linked to singly-linked jumplists increases performance by a little bit (less than 5%), while decrease storage by 25%.

We measure the impact of compacting the bit color into the parent pointer for red/black trees. We used the parent pointer because it is only needed for traversal and not in the search algorithm, hence is the best choice for compaction. It turns out that the impact on all the operations is less than 3% in the running time. Given that the savings is a whole 4 bytes (due to alignment issues in modern architectures), it is surprising that this technique is not applied more often.

We also measure the impact of maintaining the subtree sizes, by augmenting the red/black tree. We augment the red/black tree using the procedure of [3]. In particular, since we maintain the whole subtree size, we need to do two passes for insertion, one to check if the element is not already present, and the other for the insertion proper. We note that insertions are slowed down by about 25%. These findings are confirmed by splay trees as well. For fairness, it is the augmented variants of red/black and splay trees that ought to be compared to the randomized dictionaries. The timings mentioned below follow that suggestion.

Our next observation is that traversal costs are negligible in all three structures, compared to insertions and deletions. Nevertheless, in an application where traversal accounts for a substantial part of the running time, jumplists should have a clear advantage over BST (of any kind). This does not appear to be the case, however. For random order, the nodes are allocated in a completely different order as they are traversed. Thus the bad performance of the memory cache completely spoils the advantage of jumplists. For increasing and decreasing insertion orders, however, we still don't see a difference, and this time we cannot yet explain why.

For random keys, the insertion and deletion performances of jumplists are about three times slower than those of red/black trees, and 50% slower than randomized BST. The query performances are comparable, though, with a difference of only 30% over red/black trees and randomized search trees. Splay lists are comparable to randomized trees both in storage and in insertion time, but since searching also splays the tree, searching on all the keys is about twice slower. Thus, splay trees present an improvement over randomized search trees only if the access pattern presents a strong coherence.

For sorted keys, the picture is different. First, locality of reference means that the insertion is much faster (up to 25%). We also observe that the insertion and deletion costs for red/black trees behave similarly (up to 45%). However, the search time of jumplists benefits greatly from the locality of reference: it becomes less than half of the search time of red/black trees, which was already improved by 50%. Exceptionally, splay trees are extremely fast for insertion (always at the root) when keys are sorted, but the search then performs quite a bit of splaying, and ends up being as slow as the insertion would otherwise be.

For reverse sorted keys, however, there is no marked improvement over the random order in either the red/black tree (a side effect of a bias in the insertion?) or the jumplists (as expected). At the moment, though, we don't fully understand the real impact of caching and other systems phenomena for reverse keys. Further profiling seems appropriate.

We verify experimentally that the randomized search tree is relatively immune to the insertion order, up to a small improvement of about 5% in the insertion time when keys are sorted or reverse sorted.

Profiling jumplists shows that insertion and deletions are expensive due to calls to the JUMPLISTFROMLIST function. An appealing improvement of JUMPLISTFROMLIST would be to use a recycling or usurping strategy instead of rebuilding the randomized jumplist from scratch.

7 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. Jumplists are in bijection to binary trees with keys stored in preorder, but there are minor differences as explained in this paper.

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 exciting 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 exciting 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 [4]. 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] M. Abramowitz and I. A. Stegun. *Handbook of Mathematical Functions.* Dover, 1973. A reprint of the tenth National Bureau of Standards edition, 1964.

[2] C. Aragon and R. Seidel. Randomized search trees. In *Proc. 30th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 540–545, 1989.

[3] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

[4] Olivier Devillers. Improved incremental randomized Delaunay triangulation. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 106–115, 1998.

[5] P. Flajolet and A. M. Odlyzko. Singularity analysis of generating functions. *SIAM J. Disc. Math.*, 3(2):216–240, 1990.

[6] G. H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures.* Addison-Wesley, 1991.

[7] R.L. Graham, D.E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science.* Addison Wesley, second edition, 1994.

[8] H.-K. Hwang. *Théorèmes limites pour les structures combinatoires et les fonctions arithmetiques.* PhD thesis, École Polytechnique, Palaiseau, France, December 1994.

[9] Donald E. Knuth. *The Art of Computer Programming, Vol 3., @nd Edition.* Addison-Wesley, 1998.

[10] H.M. Mahmoud. *Evolution of random search trees.* Wiley, 1992.

[11] C. Martínez and S. Roura. Randomized binary search trees. *J. Assoc. Comput. Mach.*, 45(2), 1998.

[12] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

[13] S. Roura. An improved master theorem for divide-and-conquer recurrences. *J. Assoc. Comput. Mach.*, 48(2), 2001.

[14] R. Sedgewick. *Algorithms in C++, Parts 1-4: Fundamentals, Data Structure, Sorting, Searching.* Addison-Wesley, third edition, 1998.

[15] R. Sedgewick and P. Flajolet. *An Introduction to the Analysis of algorithms.* Addison-Wesley, 1996.

[16] R. Sedgewick and P. Flajolet. Analytic combinatorics—symbolic combinatorics. To appear, 2002.

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

[18] 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.

[19] N. Wirth. *Algorithms + Data Structures = Programs.* Prentice-Hall, 1975.

A Implementation details

There are many variants and choices to make when implementing tree-like data structures. Sedgewick's book [14] includes a nice discussion of the possible search structures, and concludes by saying that red/black trees are the best suited for a general-purpose library implementation. Indeed, red/black trees are the structure of choice in many implementations of the C++ STL `set` and `map` structures. We based our implementation of red/black trees on the publically available SGI STL implementation [], which is based on [3] and is highly optimized. In addition, the search procedure uses the observation 1 from section 2.

We offer two additional optimizations: One is a highly optimized pool memory allocator for nodes, which is taken from the Boost library Boost.alloc.[2] The other is to compact the color bit into the parent pointer: since a node is aligned to memory boundaries, the last two bits are necessarily 0, and can be used for other purposes (they are masked out when accessing the pointer).

We implemented jumplists in the same framework, and compared them against red/black trees, as well as randomized search trees [11]. The search procedure was implemented using the superior one-comparison-per-node paradigm in all BST, and using the algorithm of Figure 2(left) for jumplists. When maintaining the subtree sizes, a problem occurs during insertion in a set: a node may be inserted or not if its key is already present, depending on whether the set is a multiset or not; traditional implementations perform a search to determine this, then a second traversal to update the size fields. For both jumplists and randomized search trees, we use the following trick of Martinez & Roura: instead of maintaining the size of the substructure rooted at a node, we maintain the size of one of its two children and a bit to indicate which (for jumplist, the bit differentiates between the jsize and the nsize; for BST, between left and right subtree size). When inserting an element, the size field is flipped if necessary to always store the size of the substructure that is not recursed into, and the size is maintained during the search by subtraction. This allows to perform a single traversal prior to insertion, whether the node will be inserted or not. The size field contains both an integer and a bit in a compact format.

The issue of single/double linkage notwithstanding, a minimal jumplist node thus takes two pointers and size field, in addition to the node's key and data. With compaction, a (either randomized or red/black) BST node takes three pointers and a size field (optional for red/black trees). Note that if traversal through iterator is provided, as demanded by the C++ concept of container, the parent pointer is necessary as it is used in the algorithm Tree-Successor. If bidirectional iterators are provided, then jumplist nodes also need to store a third pointer as well.[3]

Given size fields, it is possible to implement rank-based operations for all three types of dictionaries, thereby providing random-access iterators with logarithmic time.[4]

We implemented our code in C++, following best practice and using aggressive optimizations. Consequently, we verified that our implementation of red/black trees is more memory-efficient and about as fast as the C++ standard library implementation which it was based on. The implementation is freely available (consult `http://photon.poly.edu/~hbr`) and we plan to release it eventually in the form of a Boost library.

B Complete experimental results

In this section, we offer the complete experimental results, in the form of three tables detailing the running times for the following operations:

1. Insert a million elements in a certain order.
2. Traverse the data structure in order a hundred times.
3. Search for all the elements in the same order as inserted.
4. Gather the number of links followed by the search procedure of **3** divided by $n \ln n$ (this gives a multiplicative constant that should match the results of Section 4), as well as the number of comparisons performed (per node visited).
5. Perform a hundred thousand searches (either successful, or unsuccesful), picked at random without replacement from a pool of keys present or not present.
6. Destroy the data structure.

The data structures tested are summarized in Ta-

[2]The boost repository (`www.boost.org`) is a compendium of peer-reviewed very high quality libraries, and is generally very highly regarded among programmers in C++. Its libraries range from simple utilities, fixes for buggy compilers, and more involved domain-specific solutions. In particular, it has a very complete random generator library, and a graph library which represents the state of the art.

[3]Therefore jumplists are not likely to be used for implementing a standard-conforming C++ `set`, as the concept requires bidirectional iterators. Their advantage lies mainly in situations when forward traversals only are required.

[4]Technically speaking, the C++ standard mandates that all operations on random-access iterators take amortized constant time. It is an interesting open problem to try and meet this guarantee with jumplist iterators.

ble 1. We assume a setup in which a pointer has the same number of bytes as an integer representing the size (as indeed they should), and this number is 4 bytes. Because of alignment issues, a `bool` field with other pointer fields will require 4 bytes for storage as well.

The results for a random insertion order are presented in Table 2, for an increasing insertion order in Table 3, and for a decreasing insertion order in Table 4. Because of time limitations, we could not implement treaps, but we expect they will figure in the full version of the paper. As a rule, all the functionality gathering information about path length and number of comparisons is given in separate functions in order to keep the search functions free of side effects. Thus the search times as quoted do not include the time taken to gather the statistics.

Name	Type	Description	Node size	Bidirectional
`std::set`	Det.	C++ set, internally red/black tree.	16 bytes	YES
`rb_tree`	Det.	Custom red/black trees, derived from the SGI STL `std::set` implementation, augmented to gather path length information.	16 bytes	YES
`rb_cp_tree`	Det.	Same as previous entry, augmented to gather path length information. The color bit is compacted in the parent pointer.	12 bytes	YES
`rb_aug_tree`	Det.	Same as `rb_tree`, augmented with subtree size information.	20 bytes	YES
`rb_aug_cp_tree`	Det.	Same as `rb_cp_tree`, augmented with size information. The color bit is compacted in the parent pointer.	16 bytes	YES
`rand_tree`	Rand.	Randomized BST, storing the size of only one of the two subtree sizes and with a bit to remember which (compacted in the parent pointer).	16 bytes	YES
`treap`	Rand.	Treaps, storing a randomly chosen priority.	16 bytes	YES
`splay_tree`	Amort.	Splay trees.	12 bytes	YES
`splay_aug_tree`	Amort.	Splay trees, augmented with subtree size information.	16 bytes	YES
`jl`	Rand.	Doubly linked randomized jumplists, storing both sublist sizes.	20 bytes	YES
`jl_fw`	Rand.	Singly linked randomized jumplists, storing both sublist sizes.	16 bytes	NO
`jl_cp`	Rand.	Same as `jl`, storing only one of the two sublist sizes, and a bit to remember which (compacted in the parent pointer).	16 bytes	YES
`jl_fw_cp`	Rand.	Combination of `jl_fw` and `jl_cp`.	12 bytes	NO

Table 1: Data structures tested in our experiments.

Data structure	**1**	**2**	**3**	**4**(ptr.)	**4**(comp.)	**5**(succ.)	**5**(unsucc.)	**6**
`std::set`	3.18s	0.43s	2.58s			0.28s	0.27s	0.35s
`rb_tree`	3.25s	0.37s	2.69s	1.023	1.049	0.29s	0.27s	0.37s
`rb_cp_tree`	3.33s	0.36s	2.57s	1.022	1.049	0.28s	0.27s	0.41s
`rb_aug_tree`	4.11s	0.36s	2.83s	1.022	1.049	0.3s	0.29s	0.38s
`rb_cp_aug_tree`	4.12s	0.37s	2.84s	1.026	1.049	0.3s	0.29s	0.38s
`rand_tree`	9.55s	0.36s	3.06s	1.005	1.05	0.33s	0.32s	0.37s
`splay_tree`	6.28s	0.34s	7.66s	1.388	1.036	0.66s	0.63s	0.48s
`splay_aug_tree`	9.62s	0.34s	10.16s	1.388	1.036	1.15s	1.06s	0.33s
`jl`	14.45s	0.43s	4.28s	0.9514	1.625	0.45s	0.44s	0.02s
`jl_fw`	13.91s	0.43s	4.18s	0.9495	1.626	0.47s	0.43s	0.12s
`jl_cp`	14.13s	0.43s	4.23s	0.9862	1.603	0.44s	0.44s	0.11s
`jl_fw_cp`	14.02s	0.43s	4.15s	0.9992	1.595	0.43s	0.44s	0.13s

Table 2: Random insertion order.

Data structure	**1**	**2**	**3**	**4**(ptr.)	**4**(comp.)	**5**(succ.)	**5**(unsucc.)	**6**
`std::set`	1.99s	0.44s	1.25s			0.18s	0.16s	0.17s
`rb_tree`	1.98s	0.37s	1.32s	1.02	1.049	0.18s	0.16s	0.21s
`rb_cp_tree`	2.7s	0.36s	1.44s	1.02	1.049	0.18s	0.17s	0.19s
`rb_aug_tree`	3.92s	0.37s	1.55s	1.02	1.049	0.2s	0.18s	0.21s
`rb_cp_aug_tree`	3.72s	0.37s	1.37s	1.02	1.049	0.18s	0.17s	0.23s
`rand_tree`	9.05s	0.36s	3.02s	1.008	1.05	0.36s	0.31s	0.36s
`splay_tree`	0.55s	0.31s	1.15s	89.47	1.001	0.9s	0.26s	0.24s
`splay_aug_tree`	0.64s	0.33s	1.57s	89.47	1.001	0.99s	0.23s	0.25s
`jl`	10.68s	0.43s	0.45s	0.9505	1.595	0.09s	0.08s	0.15s
`jl_fw`	10.33s	0.43s	0.44s	0.9505	1.595	0.1s	0.08s	0.14s
`jl_cp`	10.35s	0.44s	0.45s	0.9505	1.595	0.09s	0.08s	0.16s
`jl_fw_cp`	9.87s	0.43s	0.43s	0.9505	1.595	0.09s	0.07s	0.16s

Table 3: Increasing insertion order.

Data structure	**1**	**2**	**3**	**4**(ptr.)	**4**(comp.)	**5**(succ.)	**5**(unsucc.)	**6**
`std::set`	3.18s	0.43s	2.56s			0.27s	0.26s	0.36s
`rb_tree`	3.24s	0.36s	2.69s	1.022	1.049	0.29s	0.27s	0.37s
`rb_cp_tree`	3.36s	0.37s	2.56s	1.021	1.049	0.31s	0.26s	0.41s
`rb_aug_tree`	4.1s	0.37s	2.82s	1.025	1.049	0.3s	0.29s	0.38s
`rb_cp_aug_tree`	4.11s	0.37s	2.83s	1.022	1.049	0.3s	0.29s	0.38s
`rand_tree`	9.05s	0.36s	3.02s	1.008	1.05	0.36s	0.31s	0.36s
`splay_tree`	6.09s	0.33s	7.1s	89.47	1.001	0.89s	0.74s	0.4s
`splay_aug_tree`	9.51s	0.35s	10.82s	89.47	1.001	1.26s	1.15s	0.5s
`jl`	14.4s	0.43s	4.36s	1.012	1.587	0.45s	0.47s	0.01s
`jl_fw`	14.33s	0.43s	4.14s	0.9388	1.633	0.44s	0.42s	0.01s
`jl_cp`	13.92s	0.43s	4.13s	0.9446	1.63	0.43s	0.42s	0.01s
`jl_fw_cp`	13.94s	0.43s	4.07s	0.9491	1.626	0.45s	0.44s	0.16s

Table 4: Decreasing insertion order.