Deterministic Jumplists

Amr Elmasry

Department of Computer Engineering and Systems
Alexandria University, Egypt
elmasry@alexeng.edu.eg

Abstract. We give a deterministic version of the randomized jumplists recently introduced by Brönnimann et al. The new structure supports searches in worst-case logarithmic time, insertions and deletions in amortized logarithmic time (versus expected logarithmic time for the randomized version), and the successor of an element can be accessed in worst-case constant time (similar to the randomized version). As for the randomized version, our deterministic counterpart does not involve duplicate indexing information or extra pointers to access the successor, in contrast with other known dictionary structures. A Jumplist is basically an ordered linked list with an extra jump pointer per node to speed up the search, using a total of two pointers and two integers per node. All dictionary operations are implemented in an easy, efficient and compact way.

ACM CCS Categories and Subject Descriptors: E.1 [Data Structures]: Lists, stacks, and queues; E.2 [Data Storage Representations]: Linked representations; F.2.2 [Analysis of Algorithms and Problem Complexity]: sorting and searching

Key words: Data structures, dictionaries, weight-balanced trees, linked lists, amortized analysis

1. Introduction

One of the basic data structures in the history of computer science is the dictionary structure. Several search tree structures have been introduced in the literature. Hibbard [1962] determined how the average search time behaves if trees are left to grow at random. To improve the search time, one looks for trees which satisfy three conflicting requirements: they must be close to being balanced so that the search time would be short, they should be restructured easily when they have become too unbalanced, and this restructuring should be required only rarely. The fact that the depth is a crucial parameter for the behavior of search trees led to the development of balanced trees where the depth is guaranteed to be $O(\log n)$. There are two major types of balanced trees, the height- and the weight-balanced trees. In the first class the height of subtrees is balanced; it includes AVL trees [Adelson-Velskii and Landis 1962], B-trees [Bayer and McCreight 1972], red-black trees [Guibas and Sedgewick 1978], half-balanced trees [Olivie 1982] and balanced binary trees [Tarjan 1983]. In the second class the weight of the
subtrees is balanced; it includes $BB[\alpha]$ trees [Nievergelt and Reingold 1973] and the internal-path trees [Gonnet 1983].

Solutions geared to provably good amortized or randomized performance were proposed; splay trees [Sleator and Tarjan 1985], treaps [Aragon and Seidel 1989], skip lists [Pugh 1990], and randomized $BST$ [Martinez and Roura 1998] fall into this category.

To support the SUCCESSOR operation in $O(1)$ time, for the above search structures, one of two solutions can be implemented: The first is to build an index structure (internal nodes) over a linked list of the original data items (external nodes) by duplicating the search information in these data items. The second is to maintain an extra pointer with every node that points to its successor (threading). Both implementations require more storage, which does not only account for extra memory but also contribute to complicating different dictionary operations while maintaining the balancing criteria.

Recently, randomized jumplists were introduced [Brönnimann et al. 2003]. A jumplist is an ordered list with an extra pointer called the jump pointer, accounting for two pointers per node. The basic idea is to employ the jump pointer to speed up the search whenever possible, or to sequentially access the list otherwise. Randomized jumplists support the operations SEARCH, INSERT and DELETE in expected logarithmic time, and the SUCCESSOR operation in constant time. Thus, jumplists provide an alternative to the classical dictionary data structures.

In this paper, we introduce a deterministic version of the randomized jumplists, achieving worst-case logarithmic time for the SEARCH operation, and amortized logarithmic time per INSERT and DELETE, and still constant SUCCESSOR time. Similar to $BB[\alpha]$-trees [Nievergelt and Reingold 1973], it relies on a parameter which can be used to compromise between short search time and frequency of restructuring. Hence, the deterministic version appears to be more flexible (by using the parameter). Contrary to that of skip lists [Munro et al. 1992], the deterministic version of the jumplists appears to be more natural.

2. The data structure

As introduced by Brönnimann et al., the jumplist is a linked list that has two pointers per node, the next pointer and the jump pointer. Every node has a key, according to which the list is ordered following the next pointers. In other words, $next[x]$ points to the node whose key is the successor of $key[x]$ with respect to this order. Throughout the paper, we use the successor of $x$ to denote the node that is pointed to by $next[x]$. We denote by $x \prec y$ the relation induced by the order of the list, indicating that $key[x]$ precedes $key[y]$ in this order.

The next pointer of the last node of a jumplist points to null. Within the implementation, to indicate that the jump pointer of a node is null, we make this jump pointer point to the node itself, i.e. $jump[x] = x$. To facilitate the implementation of the operations of jumplists, a header node
header is used. The successor of the header is the first node of the jumplist. To allow insertions and deletions at the beginning of the jumplist, we make \( \text{jump[header]} \) point to the first node of the jumplist, i.e. \( \text{jump[header]} = \text{next[header]} \), or to null if the jumplist is empty.

Every node \( x \) in a jumplist corresponds to a sublist, as defined by following the next pointers. This sublist consists of: the node \( x \), the next sublist of \( x \), and the jump sublist of \( x \). The first node of the next sublist of \( x \) is the node pointed to by \( \text{next}[x] \), and its last node is a node \( y \) where \( \text{next}[y] = \text{jump}[x] \) (the case \( \text{next}[x] = \text{jump}[x] \) implies that the next sublist of \( x \) is empty). The first node of the jump sublist of \( x \) is the node pointed to by \( \text{jump}[x] \), and its last node is the node with the first null jump pointer while following jump pointers starting from \( x \) (the case \( \text{jump}[x] = x \), implies that both the next sublist and the jump sublist of \( x \) are empty). Recursively, the sublist corresponding to the node pointed to by \( \text{next}[x] \) is the next sublist of \( x \), and the sublist corresponding to the node pointed to by \( \text{jump}[x] \) is the jump sublist of \( x \). Hence, another way to reach the last node of the next sublist of \( x \), if it is not empty, is to follow jump pointers starting from the node pointed to by \( \text{next}[x] \) until reaching a node with a null jump pointer. The following three conditions should be satisfied:

- The jump condition: For any node \( x \), \( x \prec \text{jump}[x] \) or \( \text{jump}[x] = x \) (null pointer indicating that \( x \) is the last node of a sublist).
- The non-crossing condition: For any pair of nodes \( x \) and \( y \) such that \( x \prec y \) and \( \text{jump}[x] \neq y \), either \( \text{jump}[y] \prec \text{jump}[x] \) (nested), or \( \text{jump}[x] \prec y \) (disjoint).
- The access condition: For any node \( y \), if \( \text{jump}[y] = y \) and \( y \) is not the last node of the jumplist, then there exists a node \( x \) where \( x \prec y \) such that \( \text{jump}[x] = \text{next}[y] \).

(Here, we modify the original conditions in [Brönnimann et al. 2003] by allowing the jump pointer of a node to point to the node itself. This prevents the possibility of the exceptional pointers in the non-crossing conditions. We also add the access condition.)

The nodes of a jumplist also contain size information. Each node maintains two integers, \( ncount \) and \( jcount \) corresponding to the sizes of the next and jump sublists, respectively. Let \( \text{count}[x] \) be the size of the sublist of \( x \), then

\[
\text{count}[x] = 1 + ncount[x] + jcount[x].
\]

If \( \text{jump}[x] = x \), then \( ncount[x] = jcount[x] = 0 \). If \( \text{jump}[x] = \text{next}[x] \), then \( ncount[x] = 0 \).

Our main goal is to maintain, for every node, the ratio of the size of the next sublist and the jump sublist within a specified constant range. More specifically, we want to have

\[
\alpha \leq \frac{ncount[x] + 1}{\text{count}[x] + 1} \leq 1 - \alpha,
\]

where \( \alpha \) is the range.
for some constant $0 < \alpha < \frac{1}{2}$. We call this condition the balancing condition. Notice the similarity between these conditions and the balancing conditions of the weight-balanced trees $BB[\alpha]$. If upon the insertion or deletion of a node to the jumplist the balancing condition is not fulfilled for some nodes (we call these nodes bad nodes), then the smallest sublist that contains all the bad nodes is rebuilt to form, what we call, a perfect jumplist. The following theorem follows as a result of the balancing condition.

**Theorem 1.** The number of recursive levels of the sublists of a jumplist that has $n$ nodes is at most

$$\frac{\log_2 (n + 1) - 1}{\log_2 \frac{1}{1-\alpha}}.$$**

**Proof.** The proof is by induction on $n$. If $n = 1$, the number of recursive levels is 0 and the base case follows. Let $l$ be the size of the next sublist of the jumplist. As a result of the balancing condition, $\frac{l+1}{n+1} \leq 1 - \alpha$. Using the induction hypothesis, then the number of recursive levels of this next sublist is at most $\frac{\log_2 (l+1) - 1}{\log_2 \frac{1}{1-\alpha}}$. A similar argument for the jump sublist of the jumplist can be shown. Therefore, the number of recursive levels of the jumplist is at most

$$\frac{\log_2 (l + 1) - 1}{\log_2 \frac{1}{1-\alpha}} + 1 \leq \frac{\log_2 (n + 1)(1 - \alpha) - 1}{\log_2 \frac{1}{1-\alpha}} + 1 = \frac{\log_2 (n + 1) - 1}{\log_2 \frac{1}{1-\alpha}}.$$\[\square\]

The following theorem directly follows from a corresponding theorem of Nievergelt and Wong [1973].

**Theorem 2.** The average number of recursive levels of the sublists of a jumplist that has $n$ nodes is

$$\frac{1}{H(\alpha)} \left(1 + \frac{1}{n}\right) \log_2 (n + 1) - 2,$$

where

$$H(\alpha) = -\alpha \log_2 \alpha - (1 - \alpha) \log_2 (1 - \alpha).$$

**Corollary 1.** Using $\alpha = 1 - 1/\sqrt{2}$, then the number of recursive sublists of the jumplist is at most $2 \log_2 (n + 1)$, and on average $1.15 \log_2 (n + 1)$.

### 2.1 Searching

The search algorithm is introduced in Brönnimann et al. [2003]. The main idea of the search is to try to use the jump pointers, if possible, to skip several nodes; otherwise to use the next pointers and sequentially access the list. To speed up the search, the jump pointers should neither be too long nor too short. The search for a predecessor of a value $k$ continues while the
keys of the nodes are still smaller than \( k \), until a node is reached that either has a null jump pointer or has a successor whose key value is larger than or equal to \( k \).

Note that the correctness of the search algorithm relies on the fact that, for any node \( x \) where \( \text{jump}[x] \neq x \), \( \text{key}[x] < \text{key}[\text{jump}[x]] \) (that is why the jump condition is meaningful). Also, if \( x \prec y \prec \text{jump}[x] \prec \text{jump}[y] \), then \( \text{jump}[y] \) would have never been used (that is why the non-crossing condition is meaningful). Finally, if the jump pointer of a node is null and there is no other jump pointer pointing to the successor of this node, such a successor would have never been reached (that is why the access condition is meaningful).

The pseudo-code to find the node with the largest key value that is smaller than \( k \) is given below.

```plaintext
Find-Predecessor(k)
{
    x ← header; found = 0;
    while (jump[x] \neq x && found = 0)
    {
        if (key[jump[x]] < k)
            x ← jump[x];
        else if (key[next[x]] < k)
            x ← next[x];
        else
            found ← 1;
    }
    return x;
}
```

The number of comparisons needed to search for a value is at most two comparisons per level (one with the next node and the other with the jump node). Using Theorem 1, the total number of such comparisons is at most

\[
\frac{2 \log_2 (n + 1) - 2}{\log_2 \frac{1}{1-\alpha}}.
\]

To access the predecessor of a node, we use the value of the key of that node in the above procedure requiring logarithmic time. Alternatively, we may maintain a back pointer with every node to access the predecessor in constant time. To search for some value \( k \), the predecessor node is first identified, and the value of the key of its successor is checked and reported if it is equal to \( k \). If the inequalities in the given pseudo-code are changed from \( < \) to \( \leq \), the last node that has a value less or equal to \( k \) is identified. We adopt this variation in the insertion algorithm; we allow duplicate keys and the equal keys will be stored the same order as that of their insertions.
2.2 Building a perfect jumplist

A jumplist is said to be perfect, if for all nodes, the size of the next sublist equals the size of the jump sublist (up to a difference of 1). More precisely, for every node $x$, $n\text{count}[x] = \lfloor \frac{\text{count}[x] - 1}{2} \rfloor$ and $j\text{count}[x] = \lceil \frac{\text{count}[x] - 1}{2} \rceil$.

It follows that the number of levels of the recursive sublists of a perfect jumplist, that has $n$ nodes, is at most $\lceil \log_2 (n + 1) \rceil$. A perfect jumplist of size 23 is shown in Figure 1.

To construct a perfect jumplist from a sorted linked list, we only need to update the jump pointers. We use a recursive function that recursively builds the next sublist of the first node, returns a pointer to the successor of the last node of this next sublist, uses this pointer to set the jump pointer of the first node, then recursively builds its jump sublist.

Note that the jump condition, the non-crossing condition, and the access condition are maintained by this procedure. It is crucial that this function builds a perfect jumplist in $O(n)$ time, where $n$ is the number of elements of the jumplist. The detailed pseudo-code is given below.

```
Build-Perfect-Jumplist(x, n)
{
    while (n > 1)
    {
        m ← $\lfloor \frac{n-1}{2} \rfloor$;
        n ← n − m − 1;
        n\text{count}[x] ← m;
        j\text{count}[x] ← n;
        y ← next[x];
        if (m = 0) jump[x] ← y;
        else jump[x] ← Build-Perfect-Jumplist(y, m);
        x ← jump[x];
    }
    jump[x] ← x;
    n\text{count}[x] ← 0;
    j\text{count}[x] ← 0;
    return next[x];
}
```
2.3 Insertion

The insertion algorithm follows the same pattern as in Brönnimann et al. [2003]. The main difference is that, where their algorithm would randomly rebuild the jumplist of a node on the path to the new node being inserted, we use the balance criterion instead. In more details, the insertion algorithm starts with searching for the node $x$ prior to the insertion point (as illustrated by the search algorithm). The sizes of the corresponding sublists are incremented during this search process, indicating a new node that will be inserted in these sublists. The new node $y$ is constructed by initializing $ncount[y]$ and $jcount[y]$ to 0 and $jump[y]$ to null, then $y$ is inserted after $x$. If $jump[x] = x$, indicating that $x$ is the last node of a sublist, the insertion procedure ends by making $jump[x]$ point to $y$. If $y$ is the only node in the next sublist of $x$, the insertion terminates. Otherwise, $jump[y]$ inherits the value of the jump pointer of the successor of $y$. The successor of $y$ inherits the jump pointer of its successor, and so on. We repeat doing so until reaching a node $y$ whose next sublist has at most one node. If the next sublist of $y$ is empty, we make $jump[y]$ point to $y$. If the next sublist of $y$ contains one node, we make $jump[y]$ point to this node.

Note that the jump condition, the non-crossing condition, and the access condition are maintained by the insertion procedure. The work done by the insertion procedure after searching for the node $x$ is bounded by the number of levels of the sublist of $x$, i.e. $O(\log count[x])$. Hence, the overall insertion cost in a jumplist of size $n$, other than the rebalancing cost, is $O(\log n)$.

Accompanying the search for the insertion point, the balancing condition is checked for every node that we access. As soon as a node is reached, whose balancing condition is violated (for the ease of implementation, we allow a node to remain violating until the next insertion that accesses this node), the sublist of this node is perfectly rebuilt using the procedure $Build-Perfect-Jumplist$. Since the insertion continues within this sublist, the rebuilding will be done for at most one sublist per insertion. The detailed pseudo-code is given below.
Insert\((k)\)
\[
\begin{array}{l}
x \leftarrow \text{header}; \ y \leftarrow \text{New}(k); \ found \leftarrow 0; \\
\text{while} \ (\text{jump}[x] \neq x \ \& \& \ found = 0) \\
\quad \{ \\
\quad \quad \text{if} \ (\frac{\text{ncount}[x]+1}{\text{count}[x]+1} < \alpha \ \& \ |1 - \alpha < \frac{\text{ncount}[x]+1}{\text{count}[x]+1}) \\
\quad \quad \quad \text{Build-Perfect-Jumplist} \ (x, \text{count}[x]); \\
\quad \quad \text{if} \ (\text{key}[\text{jump}[x]] \leq k) \\
\quad \quad \quad jcount[x] \leftarrow jcount[x] + 1; \ x \leftarrow \text{jump}[x]; \\
\quad \quad \text{else if} \ (\text{key}[\text{next}[x]] \leq k) \\
\quad \quad \quad ncount[x] \leftarrow ncount[x] + 1; \ x \leftarrow \text{next}[x]; \\
\quad \quad \text{else} \ found \leftarrow 1; \\
\quad \}\quad \text{next}[y] \leftarrow \text{next}[x]; \ \text{next}[x] \leftarrow y; \\
\quad \text{if} \ (\text{jump}[x] = x) \\
\quad \quad \text{jump}[x] \leftarrow y; \ jcount[x] \leftarrow 1; \\
\quad \text{else} \\
\quad \quad \{ \\
\quad \quad \quad \text{if} \ (\text{jump}[x] \neq \text{next}[y]) \\
\quad \quad \quad \{ \\
\quad \quad \quad \quad \text{ncount}[y] \leftarrow \text{ncount}[x]; \\
\quad \quad \quad \quad \text{while} \ (\text{ncount}[y] > 1) \\
\quad \quad \quad \quad \quad \{ \\
\quad \quad \quad \quad \quad \quad \text{jump}[y] \leftarrow \text{jump}[\text{next}[y]]; \ jcount[y] \leftarrow jcount[\text{next}[y]]; \\
\quad \quad \quad \quad \quad \quad \text{ncount}[y] \leftarrow \text{ncount}[\text{next}[y]] + 1; \ y \leftarrow \text{next}[y]; \\
\quad \quad \quad \quad \quad \}\quad \text{if} \ (\text{ncount}[y] = 0) \\
\quad \quad \quad \quad \quad \quad \text{jump}[y] \leftarrow y; \ jcount[y] \leftarrow 0; \\
\quad \quad \quad \quad \quad \text{else} \\
\quad \quad \quad \quad \quad \quad \text{jump}[y] \leftarrow \text{next}[y]; \ jcount[y] \leftarrow 1; \ \text{ncount}[y] \leftarrow 0; \\
\quad \quad \quad \quad \}\quad \text{if} \ (x \neq \text{header}) \ \text{ncount}[x] \leftarrow \text{ncount}[x] + 1; \\
\quad \}\quad \}
\end{array}
\]

\textbf{2.4 Deletion}

The deletion algorithm follows the same pattern as in Brönnimann et al. [2003]. The main difference is that, where their algorithm would rebuild the jumplist of a node on the path to the node being deleted with a given probability, we use the balance criterion instead. Without loss of generality, we assume that the value to be deleted exists in the jumplist. Otherwise, we can first search for this value, then perform the deletion only if the element exists. The deletion algorithm starts by locating the predecessor of the node \(x\) that we want to delete. The sizes of the corresponding sublists are decremented during this search process, indicating that a node will be deleted from these sublists. If \(x\) was reached via a jump pointer \(\text{jump}[s]\), the value of this jump pointer is set to the successor of \(x\) (A special case is
when the jump sublist of $s$ only contains $x$; To maintain the access condition in this case, this jump pointer is either set to null if the next sublist of $s$ is empty, or otherwise we make it point to the successor of $s$). Let $y$ be the successor of $x$. The next pointer of the predecessor of $x$ is adjusted to point to $y$. If the next sublist of $x$ is empty, the deletion procedure terminates. Otherwise, the jump pointer of $y$ inherits the value of $\text{jump}[x]$, the jump pointer of the successor of $y$ inherits the value of $\text{jump}[y]$, and so on. We repeat doing so until reaching a node whose jump pointer points to itself or its successor. The detailed pseudo-code is given below.

Delete($k$)
{  $x \leftarrow \text{header}$;
  while ($\text{key}[x] \neq k$)
  {  $s \leftarrow x$;
      if ($\frac{\text{ncount}[x]+1}{\text{count}[x]+1} < \alpha \ | \ | 1 - \alpha < \frac{\text{ncount}[x]+1}{\text{count}[x]+1}$)
          Build-Perfect-Jumplist($x$, $\text{count}[x]$);
      if ($\text{key}[\text{jump}[x]] \leq k$)
          $\text{jcount}[x] \leftarrow \text{jcount}[x] - 1$; $x \leftarrow \text{jump}[x]$;
      else if ($\text{key}[\text{next}[x]] \leq k$)
          $\text{ncount}[x] \leftarrow \text{ncount}[x] - 1$; $x \leftarrow \text{next}[x]$;
  }
  if ($\text{jump}[s] = x$)
  {  if ($\text{jump}[x] = x$)
      {  if ($\text{next}[s] = x$) $\text{jump}[s] \leftarrow s$ else $\text{jump}[s] \leftarrow \text{next}[s]$;
          $\text{jcount}[s] \leftarrow \text{ncount}[s]$; $\text{ncount}[s] \leftarrow 0$;
      }
      else $\text{jump}[s] \leftarrow \text{next}[x]$;
      if ($\text{next}[s] \neq x$)
          $s \leftarrow \text{next}[s]$; while ($\text{jump}[s] \neq s$) $s \leftarrow \text{jump}[s]$;
  }
  $\text{next}[s] \leftarrow \text{next}[x]$;
  if ($\text{ncount}[x] \neq 0$)
  {  $y \leftarrow \text{next}[x]$; $t1 \leftarrow \text{jump}[x]$; $jc1 \leftarrow \text{jcount}[x]$;
      while ($\text{ncount}[y] \neq 0$)
      {  $t2 \leftarrow \text{jump}[y]$; $jc2 \leftarrow \text{jcount}[y]$;
          $\text{jump}[y] \leftarrow t1$; $\text{ncount}[y] \leftarrow \text{ncount}[y] + jc2$; $\text{jcount}(y) \leftarrow jc1$;
          $t1 \leftarrow t2$; $jc1 \leftarrow jc2$; $y \leftarrow \text{next}[y]$;
      }
      $\text{jump}[y] \leftarrow t1$; $\text{ncount}[y] \leftarrow \text{jcount}[y]$; $\text{jcount}[y] \leftarrow jc1$;
  }
  Release($x$);  
}
Note that the jump condition, the non-crossing condition, and the access condition are maintained by the deletion procedure. The work done by the deletion procedure after searching for the node \( x \) is bounded by the number of levels of the sublist of \( x \), i.e. \( O(\log \text{count}[x]) \). Hence, the overall deletion cost in a jumplist of size \( n \), other than the rebalancing cost, is \( O(\log n) \).

Accompanying the search for the element to be deleted, the balancing condition is checked for every node that we reach. As soon as a node is reached, whose balancing condition is violated (for the ease of implementation, we allow a node to remain violating until the next deletion that accesses this node), the sublist of this node is perfectly rebuilt using the procedure \textit{Build-Perfect-Jumplist}. Since the deletion continues within this sublist, the rebuilding will be done for at most one sublist per deletion.

**Lemma 1.** The cost of rebalancing the sublists of a jumplist, involved in building perfect sublists, is bounded by the order of the number of comparisons done during the insertions and deletions.

**Proof.** Consider the sublist of \( x \), and let \( n \) be \( \text{count}[x] \) immediately after calling \textit{Build-Perfect-Jumplist} on this sublist. To be rebuilt again, several updates (insertions and deletions) should have taken place in this sublist. Let \( n_u \) be the number of such updates. Each of these \( n_u \) updates involves a comparison with \( \text{key}[x] \). The value of \( n_u \) is minimized when all the updates are deletions and all of them are performed in only one of the next and the jump sublists of \( x \). Hence, \( n_u \) should satisfy

\[
\frac{\frac{n}{\alpha}}{n - n_u + 1} > 1 - \frac{1}{\alpha} \Rightarrow n_u > \frac{\frac{1}{\alpha}}{1 - \frac{1}{\alpha}} - \frac{\alpha}{1 - \alpha}.
\]

For \( 0 < \alpha < \frac{1}{2} \), the value of \( n_u \) is a constant factor of \( n \). Since the cost of this rebalancing is \( O(n + n_u) \), we charge a constant value for each of the \( n_u \) comparisons that involves \( \text{key}[x] \).

The next main theorem is a consequence of Theorem 1 and Lemma 1.

**Theorem 3.** For an \( n \)-node jumplist, the worst-case cost of \textsc{search} and \textsc{successor} operations is \( O(\log n) \) and \( O(1) \) respectively, and the amortized cost of \textsc{insert} and \textsc{delete} operations is \( O(\log n) \).

3. Correspondence with binary trees

There is a one-to-one function between a jump list and a binary tree: the next and jump sublists respectively correspond to the left and right subtrees of the binary tree [Brönnimann et al. 2003]. If a node has no left subtree, its next node is its right child. The next node of a leaf \( l \) (except for the rightmost leaf) is the right child of the lowest such ancestor of \( l \), the left child of which is also an ancestor of \( l \) (or \( l \) itself). A leaf has a jump pointer pointing to itself (null pointer). In the corresponding binary tree, a node
that has an empty right subtree must be a leaf. This is a direct consequence of the access condition for jumplists. Hence, the number of jumplists of a given size $n$ is less than the $n$-th Catalan number. The jump and non-crossing conditions follow naturally as a result of this correspondence.

It follows that the sequence of keys, resulting from a preorder traversal of the nodes of a tree corresponding to a jumplist, is ordered. That is why we turned to globally rebuilding the jumplists, as the rotations performed by height-balanced search trees seem impossible to be achieved in such case within the required time bounds.

In view of this correspondence, the search algorithm for a predecessor of a given value $k$ is performed as follows. Starting from the root, when $k$ is found to be larger than the value of a node, it is compared with its right child first. If $k$ is larger, we recursively traverse the right subtree. Otherwise, $k$ is compared with the value of the left child. If $k$ is larger, we recursively traverse the left subtree. The search continues until the required node is found, or until a leaf is reached.

The insertion algorithm of a node $y$ is done as follows. A search is performed to identify a node $x$, the predecessor of $y$. If $x$ was a leaf, the insertion terminates after making $y$ the right child of $x$. Otherwise, $y$ is inserted as a left child of $x$. If $y$ is a leaf, the insertion terminates. Otherwise, the right subtree of the left child of $y$ becomes the right subtree of $y$. In turn, the left child of $y$ inherits the right subtree of its left child. This process is repeated until a leaf node is reached. If this leaf is a left child with no right sibling, the insertion terminates after making it a right child.

In the deletion algorithm, the node $x$ to be deleted is first identified. If $x$ is a right child that is a leaf, the left subtree of its parent becomes its right subtree and the deletion terminates with the removal of $x$. If $x$ has no left subtree, its right subtree replaces the subtree of $x$ after the removal of $x$, and the deletion terminates. Otherwise, the right subtree of $x$ becomes the right subtree of its left child $y$. The right subtree of $y$ becomes the right subtree of the left child of $y$. The process is repeated until a node that has no left subtree is reached. The right subtree of this node becomes its left subtree before it inherits the right subtree of its parent as its right subtree. Finally, the subtree of $y$ replaces the subtree of $x$ after the removal of $x$.

4. Application - Fortune’s sweep-line algorithm for building Voronoi diagrams

Among the plenty of applications where the jumplists can be used efficiently, we implemented Fortune’s sweep-line algorithm to build a Voronoi diagram [De Berg et al. 1997].

As defined by Fortune’s algorithm, the beach line is a set of arcs, which is $x$-monotone, each corresponding to a site. When we move the sweep line vertically until reaching the next site, to update the beach line, a search is required to locate the arc of the beach line lying above the new site. To
facilitate searches as well as insertions and deletions, the arcs of the beach line are represented as a dictionary structure. During the search process, once we reach an arc, we need to locate its x-coordinate successor in order to evaluate the breakpoint resulting from the intersection of these two arcs.

The first dictionary structure we used is a red-black tree, where each node represents a site. The sites are stored in accordance with the x-coordinate values of their arcs with respect to the current position of the sweep line. The drawback of this implementation is that, during the search, the successor of each node is to be identified. If we insist on not using an extra pointer with every node pointing to its successor, the search time complexity increases for an overall bound of $O(n \log^2 n)$.

In the second implementation (which was the standard used before introducing the jumplists [De Berg et al. 1997]), the beach line is represented by a red-black tree as before, while the sites corresponding to the arcs are stored only in the leaves. The internal nodes of the tree represent the breakpoints on the beach line. A breakpoint is stored at an internal node by an ordered tuple of sites $(p_i, p_j)$, where $p_i$ defines the arc left of the breakpoint and $p_j$ defines the arc to the right. Given a new site, a search in the tree takes place to locate the arc of the beach line lying above this site. At an internal node, the x-coordinate of the new site is compared with the x-coordinate of the breakpoint, which can be computed in constant time from the tuple of sites and the position of the sweep line. The drawback of this implementation is that each site is stored twice in the internal nodes. Other than using extra storage, the implementation of such trees becomes complicated while performing rotations. This is a result of the possibility that the sites in the internal nodes may change with these rotations.

The other two dictionary structures we implemented are the two versions of the jumplists, the randomized and the deterministic versions.

We generated uniformly-distributed independent random values to represent the data points (sites); one random value for each of the two coordinates of each site. Throughout the experiments, we changed the number of sites $n$, and took the average of 100 sample runs for each $n$. To achieve good performance for the deterministic jumplists, the balancing parameter $\alpha$ is fixed as 0.3. The algorithm is implemented using Borland C ++ builder V6. A PC with Intel Pentium III 550 MHZ and 128 MB RAM is used.

The results of the experiments are in Table 1. It shows the average CPU time in seconds for each of the four suggested implementations. The results indicate the superiority of the jumplists over red-black trees in this application. The red-black trees with no successor pointers are the worse, while augmenting the internal nodes of the red-black trees with pairs of successive sites would improve their performance. On the contrary, jumplists (both the randomized and the deterministic versions) showed very efficient performance with less storage, in addition to their simplicity and ease of implementation. The deterministic jumplists are slightly better than their randomized counterpart.
Finally, we point out that using the jumplists in other applications that do not intensively involve the successor operation would not be as efficient, and other dictionary structures may be a better choice. It would be interesting to see more detailed results for different dictionary operations while implementing such applications. Since the experimental data is random, the balancing operations are not frequently performed. It would also be interesting to perform other tests with other data sets that involve more balancing requirements. Another possibility is to compare the jumplists with other dictionary structures, like weight-balanced trees, that do not keep the balance all the time as in the case of red-black trees.

5. Conclusion

In this paper, we introduced a deterministic version of the jumplists; a dictionary structure that supports, in addition to the regular dictionary operations, the successor operation in constant time. A jumplist has only two pointers per node, that makes the structure efficient and easy to implement.

In comparison with the randomized version, we think that the deterministic version is more natural and easier to explain. The deterministic version also avoids generating random bits, and is more controllable by using a balancing parameter.

For searching, the number of key comparisons is somewhat higher than for balanced binary search trees. For search-intensive applications this could be a dominating efficiency drawback. Still, for applications that require a dynamic dictionary structure using many successor operations, we demonstrated that jumplists would be very efficient.

It is still challenging to find out whether it is possible or not to efficiently achieve a worst-case logarithmic time per update and constant successor time, while using only two pointers per node with no duplicate information.

Acknowledgements

I would like to thank Hanady Taha, Heba El-Allaf, and Shady Elbassuoni for implementing the jumplist’s procedures.
References


