DIMACS Technical Report 2003-29
December 2004

Deterministic Jumplists

by

Amr Elmasry
Computer Science Department, Alexandria University
Alexandria, Egypt
elmasry@cs.rutgers.edu

ABSTRACT

A deterministic version of the randomized Jumplists of Brönnimann et al. is introduced. This structure supports searches, insertions and deletions in amortized logarithmic time (versus expected logarithmic time for the randomized version), while the successor of an element can be accessed in constant time. As with the randomized version, our deterministic counterpart does not involve duplicate indexing information or extra pointers to access the successor, in contrast with other tree structures. Instead, it relies on a linked list with an extra jump pointer per node to speed up the search, for a total of two pointers and two integers per node. All dictionary operations are implemented in an easy, efficient and compact way.

1 Introduction

One of the basic data structures in the history of computer science is the dictionary structure. Several search tree structures were introduced in the literature. Hibbard [7] 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 [1], B trees [3], red-black trees [6], half-balanced trees [13] and balanced binary trees [16]. In the second class the weight of the subtrees is balanced; it includes the $BB[\alpha]$ trees [11] and the internal-path trees [5]. Solutions more geared to provably good amortized or randomized performance were proposed. Splay trees [15], treaps [2], skip lists [14] and randomized BST [9] fall into this category.

To support the SUCCESSOR operation in $O(1)$ time, one of two solutions are 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 that do not only account for extra memory, but also contribute to complicating different dictionary operations that maintain the balancing criteria.

Recently, randomized jumplists was introduced [4]. A jumplist is an ordered list with an extra pointer called the jump pointer. In other words, every node contains two pointers. 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 supports the operations SEARCH, INSERT and DELETE in expected logarithmic time, and the SUCCESSOR operation in constant time [4]. Thus, jumplists provide an alternative to the classical dictionary data structures.

In this paper, we introduce a deterministic version of the randomized jumplists, achieving amortized logarithmic time for the operations SEARCH, INSERT and DELETE, and sill constant SUCCESSOR time. Similar to $BB[\alpha]$-trees [11], it involves a parameter which can be used to compromise between short search time and frequency of restructuring. Hence, the deterministic version is more flexible (by using the parameter). Unlike that for skip lists [10], the deterministic version of the jumplists also appears to be more natural.

2 The data structure

As introduced by Brönnimann et al. [4], the jumplist is a linked list, with *next[x]* a pointer pointing to the next element of x, and another pointer per node called *jump[x]*. Every node has a *key*, according to which the list is ordered. We denote by $x \prec y$ the relation induced by the order of the list. In other words, $x \prec y$ indicates that node x comes before node y.

To facilitate the implementation of the jumplists' operations, a header node *header[L]* is used. The successor of the header is the first element of the list. Every node x in a jumplist corresponds to a sublist. This sublist recursively contains: the node x, the sublist of $next[x]$, and the sublist of $jump[x]$. The next sublist of x starts with $next[x]$ and ends with the element y where $next[y] = jump[x]$. The jump sublist of x starts with $jump[x]$ and contains the rest of the elements of the sublist of x. The following three conditions should be satisfied:

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

(Here, we modify the original conditions in [4] 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 to the set of constraints.)

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 $count[x]$ be the size of the sublist of x, then

$$count[x] = 1 + ncount[x] + jcount[x].$$

If $jump[x] = x$ (null pointer), then $ncount[x] = jcount[x] = 0$. If $jump[x] = next[x]$, 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

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

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 addition 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 largest such sublist (the one whose sublist contains all the other 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 recusive 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 $\frac{\log_2{(l+1)}-1}{\log_2{1/(1-\alpha)}}$. 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

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

□

The following theorem can be deduced from a corresponding theorem of Nievergelt and Wong [12].

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

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

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 $1.05 \log_2{(n+1)}$.*

Searching

The searching algorithm is introduced in [4]. The main idea of the search is to try to use the jump pointer if possible to skip several nodes, otherwise to use the next pointer and sequentially access the list. The search continues as long as the node keys are still smaller than the given value, until a node x is reached, where the jump pointer of x is pointing to x. Note that if $x \prec y \prec jump[x] \prec jump[y]$, then jump[y] would never be used (that is why the non-crossing condition is meaningful).

To speed up the search, the jump pointers should neither be too long nor too short. The pseudo-code to find the node whose value is smaller than some value k is given below.

```
Find-Predecessor(k)
{
    x ← header[L];
    while (jump[x] ≠ x)
    {
        if (k > key[jump[x]])
            x ← jump[x];
        else if (k > key[next[x]])
            x ← next[x];
        else return x;
    }
    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 certain node, we either use the value of that node in the above procedure requiring logarithmic time, or have a back pointer with every node to access the predecessor in constant time. To search for some value k, the predecessor node x is first identified, and the value of $next[x]$ is checked and reported if it matches the value k. If the inequalities in the given pseudo-code are changed from $>$ to $\geq$, the last node that has a value less or equal to k is

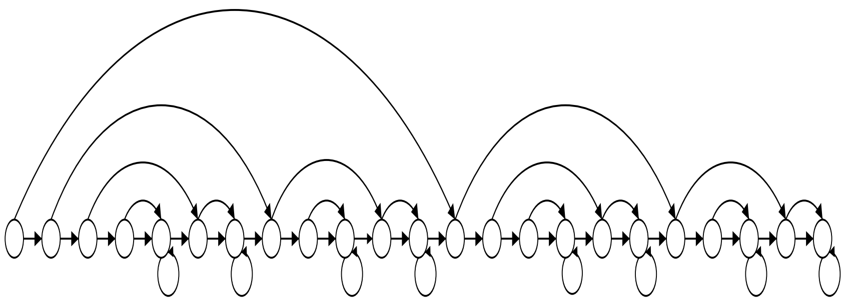

Figure 1: A perfect jumplist with 23 elements

identified. If we insert a new node after this identified predecessor, the equal keys will be stored the same order as that of their insertions.

Building a perfect jumplist

A jumplist is said to be perfect, if for all the 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, $ncount[x] = \lfloor \frac{count[x]-1}{2} \rfloor$ and $jcount[x] = \lceil \frac{count[x]-1}{2} \rceil$. 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 header node, returns a pointer to the successor of the tail of this next sublist, uses this pointer to set the jump pointer of the header node, then recursively builds the jump sublist of the header node. The detailed pseudo-code is given below, and is a simple variation of the randomly built jumplist in [4].

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. A longer iterative procedure to build a perfect jumplist is omitted from this version.

```
Build-Perfect-Jumplist(x, n)
{
    while (n > 1)
    {
        m ← ⌊(n−1)/2⌋;
        n ← n − m − 1;
        ncount[x] ← m;
        jcount[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;
    ncount[x] ← 0;
    jcount[x] ← 0;
    return next[x];
}
```

Insertion

The insertion algorithm follows the same pattern as in [4]. 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 searching algorithm). The counts 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 inserted after x. If the jump pointer of x was pointing to x, the insertion procedure ends by making this pointer point to y. If y is the only node in the next sublist of x, the insertion terminates. Otherwise, the jump pointer of y inherits the value of the jump pointer of its successor. The successor of y inherits the jump pointer of its successor, and so on. We repeat doing so until reaching a node whose next sublist has at most one element. If we reached a node whose jump pointer points to its successor, we make this pointer point to the node itself. If we reached a node whose next sublist has one element, we make the jump pointer of this node point to this element. 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 cost of the insertion procedure in a jumplist of size n, other than the re-balancing, is $O(\log n)$.

```
Insert (k)
{
    x ← header[L]; y ← New-Node(k);
    while (jump[x] ≠ x)
    {
        if (1 − α < (ncount[x]+1)/(count[x]+1) < α)
            Build-Perfect-Jumplist (x, count[x]);
        if (key[jump[x]] ≤ k)
            jcount[x] ← jcount[x] + 1;  x ← jump[x];
        else
        {
            ncount[x] ← ncount[x] + 1;
            if (key[next[x]] ≤ k) x ← next[x]  else break;
        }
    }
    next[y] ← next[x];  next[x] ← y;
    if (jump[x] = x)
        jump[x] ← y;  jcount[x] ← 1;
    else if (jump[x] ≠ next[y])
    {
        while (ncount[y] > 1)
        {
            jump[y] ← jump[next[y]];  jcount[y] ← jcount[next[y]];
            ncount[y] ← ncount[next[y]] + 1;  y ← next[y];
        }
        if (jump[y] = next[y])
            jump[y] ← y;  jcount[y] ← 0;
        else
            jump[y] ← next[y];  jcount[y] ← 1;  ncount[y] ← 0;
    }
}
```

Accompanying the search for the insertion point, the balancing condition is checked for every node that we reach. As soon as a node is reached, whose balancing condition is violated, the sublist of this node is perfectly rebuilt using the procedure *Build-Perfect-Jumplist*. Since the search continues within this sublist, the rebuilding will be done for at most one sublist per insertion. The detailed pseudo-code is given above.

Deletion

The deletion algorithm follows the same pattern as in [4]. 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 probablity, we use the balance criterion instead. In more details, we assume without loss of generality that the value to be deleted exists in the jumplist. Otherwise, we can first search for this value, then execute the deletion if the element exists. The deletion algorithm starts by locating the predecessor of the node x that we want to delete. The counts 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 $jump[s]$, 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 s if x is the successor of s, or otherwise it is set to the successor of s.) Let y be the successor of x. The next pointer of the predecessor of x will point to y. The jump pointer of y inherits the jump pointer of x. The jump pointer of the successor of y inherits the jump pointer of y, and so on. We repeat doing so until reaching a node whose jump pointer points to itself or its successor. Note that the jump condition, the non-crossing condition, and the access condition are maintained by the deletion procedure.

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, the sublist of this node is perfectly rebuilt using the procedure *Build-Perfect-Jumplist*. Since the search continues within this sublist, the rebuilding will be done for at most one sublist per deletion. The detailed pseudo-code is given below.

```
Delete(k)
{
    x ← header[L];
    while (key[x]! = k)
    {
        s ← x;
        if (1 − α < (ncount[x]+1)/(count[x]+1) < α)
            Build-Perfect-Jumplist(x, count[x]);
        if (key[jump[x]] ≤ k)
            jcount[x] ← jcount[x] − 1; x ← jump[x];
        else if (key[next[x]] ≤ k)
            ncount[x] ← ncount[x] − 1; x ← next[x];
    }
    if (jump[s] = x)
    {
        if (jump[x] = x)
            if (next[s] = x) jump[s] ← s else jump[s] ← next[s];
            jcount[s] ← ncount[s]; ncount[s] ← 0;
        else jump[s] ← next[x];
        if (next[s] ≠ x)
            s ← next[s]; while (jump[s] ≠ s) s ← jump[s];
    }
    next[s] ← next[x];
    if (ncount[x] ≠ 0)
    {
        y ← next[x]; t1 ← jump[x]; jc1 ← jcount[x];
        while (ncount[y] ≠ 0)
        {
            t2 ← jump[y]; jc2 ← jcount[y];
            jump[y] ← t1; ncount[y] ← ncount[y] + jc2; jcount(y) ← jc1;
            t1 ← t2; jc1 ← jc2; y ← next[y];
        }
        jump[y] ← t1; ncount[y] ← jcount[y]; jcount[y] ← jc1;
    }
    delete x;
}
```

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 *count*$[x]$ immediately after calling *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 $key[x]$. The value of n_u is minimized when all the updates are deletions and all of them are performed in either the next or the jump sublists of x. Hence n_u should satisfy

$$\frac{\frac{n}{2}+1}{n-n_u+1} > 1-\alpha \implies n_u > \frac{\frac{1}{2}-\alpha}{1-\alpha}n - \frac{\alpha}{1-\alpha}.$$

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 $key[x]$. □

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

Theorem 3 *For an-n node jumplist, the worst-case cost of the CONTAINS and SUCCESSOR operations is $O(\log n)$ and $O(1)$ respectively, and the amortized cost of the INSERT and 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, given in [4]: the next and jump sublists respectively correspond to the left and right subtrees of the binary tree. The next node of a leaf l, of the corresponding binary tree, is the right child r of the least common ancestor of l and r (except for the rightmost leaf). 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 jumplist is a heap-ordered binary tree. The value of the root is smaller than or equal to the value of its descendents. The values of the nodes of the left subtree are smaller than or equal to the values of the nodes of the right subtree. That is why we turned to rebuilding the jumplists, as the rotations performed by balanced search trees seem impossible with junplists.

In view of this correspondence, the search algorithm is performed as follows. The value we are looking for k is compared with the right child of the root first. If k is larger we recursively traverse the right subtree. Otherwise, the value of k is compared with the value of the left child. 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 the position, in the tree, of the predecessor node of the insertion point. The node y is then inserted in this position. If y is inserted as a leaf of the tree, the insertion terminates. Otherwise, the right subtree of the left child of y becomes the right subtree of y. The left child of y inherits the right subtree of its left child as well. The process is repeated until a leaf node is reached. If as a result of the insertion a node will have a left child and no right children, this left child is made the right child (maintaining the access condition).

In the deletion algorithm, the node x to be deleted is identified. If x is a right child that is a leaf, the left subtree of its parent becomes the right subtree (maintaining the access condition) and the deletion terminates with the removal of x. 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.

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

Among the tons of applications where the jumplists can be used efficiently, we mention this application. Using Fortune's sweep-line algorithm to build a Voronoi diagram [8], the beach line is represented by a balanced binary search tree. The leaves of this tree correspond to the arcs of the beach line, which is x-monotone. Each leaf stores the site that defines the arc it represents. 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 parabola left of the breakpoint and p_j defines the parabola 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 decision about which branch of the tree to take depends on the result of this comparison. Unlike the jumplists, using any of the known balanced tree implementations, the internal nodes involve duplicating each site twice. 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 as a result of these rotations. On the contrary, jumplists showed a very efficient performance with less storage, in addition to its simplicity and ease of implementation.

Acknowledgement

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

References

[1] G. Adelson-Velskii and E. Landis. *On an information organization algorithm.* Doklady Akademia Nauk SSSR, 146(1962), 263-266.

[2] C. Aragon and R. Seidel. *Randomized search trees.* 30th IEEE Symp. on Foundations Of Computer Science. (1989), 540-545.

[3] R. Bayer and E. McCreight. *Organisation and maintenance of large ordered indices.* Acta Inform. 1 (1972), 173-189.

[4] H. Brönnimann, F. Cazals and M. Durand. *Randomized jumplists: A jump-and-walk dictionary data structure.* 20th Symp. On Theor. Aspects of Computer Science. In LNCS 2607 (2003), 283-294.

[5] G. Gonnet. *Balancing binary trees by internal path reduction.* Comm. ACM 26(12) (1983), 1074-1081.

[6] L. Guibas and R. Sedgewick. *A dichromatic framework for balanced trees.* 17th IEEE Symp. on Foundations Of Computer Science. (1978), 8-21.

[7] T. Hibbard. *Some combinatorial properties of certain trees.* J. Assoc. Comput. Mach. (1962) 13-28.

[8] M. De Berg, M. Kreveld, M. Overmars and O. Shwarzkopf. *Computational geometry-algorithms and applications.* Springer-Verlag, (1997).

[9] C. Martinez and S. Roura. *Randomized binary search trees.* J. Assoc. Comput. Mac. 45(2) (1998).

[10] J. Munro, T. Papadakis and R. Sedgewick. *Deterministic skip lists.* Symp. On Discrete Algorithms. (1992), 367-375.

[11] I. Nievergelt and E. Reingold . *Binary search trees of bounded balance.* SIAM J. Comput. 2 (1973), 33-43.

[12] J. Nievergelt and C. Wong. *Upper bounds for the total path length of binary trees.* J. Assoc. Comput. Mach. (1973), 1-6.

[13] H. Olivie. *A new class of balanced search trees: half balanced binary search tree.* RAIRO Inform. Theor. 16(1) (1982), 51-71.

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

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

[16] R. Tarjan. *Updating a balanced search tree in $O(1)$ rotations.* Inform. Process. Lett. 16 (183), 253-257.