Predecessor Queries in Dynamic Integer Sets

Gerth Stølting Brodal*

BRICS**, Department of Computer Science, University of Aarhus
Ny Munkegade, DK-8000 Århus C, Denmark
gerth@brics.dk

Abstract. We consider the problem of maintaining a set of n integers in the range $0..2^w - 1$ under the operations of insertion, deletion, predecessor queries, minimum queries and maximum queries on a unit cost RAM with word size w bits. Let $f(n)$ be an arbitrary nondecreasing smooth function satisfying $\log\log n \leq f(n) \leq \sqrt{\log n}$. A data structure is presented supporting insertions and deletions in worst case $O(f(n))$ time, predecessor queries in worst case $O((\log n)/f(n))$ time and minimum and maximum queries in worst case constant time. The required space is $O(n2^{\epsilon w})$ for an arbitrary constant $\epsilon > 0$. The RAM operations used are addition, arbitrary left and right bit shifts and bit-wise boolean operations. The data structure is the first supporting predecessor queries in worst case $O(\log n/\log\log n)$ time while having worst case $O(\log\log n)$ update time.

1 Introduction

We consider the problem of maintaining a set S of size n under the operations:

INSERT(e) inserts element e into S,
DELETE(e) deletes element e from S,
PRED(e) returns the largest element $\leq e$ in S, and
FINDMIN/FINDMAX returns the minimum/maximum element in S.

In the comparison model INSERT, DELETE and PRED can be supported in worst case $O(\log n)$ time and FINDMIN and FINDMAX in worst case constant time by a balanced search tree, say an (a, b)-tree [8]. For the comparison model a tradeoff between the operations has been shown by Brodal *et al.* [6]. The tradeoff shown in [6] is that if INSERT and DELETE take worst case $O(t(n))$ time then FINDMIN (and FINDMAX) requires at least worst case $n/2^{O(t(n))}$ time. Because predecessor queries can be used to answer member queries, minimum queries and maximum queries, PRED requires worst case $\max\{\Omega(\log n), n/2^{O(t(n))}\}$ time. For the sake of completeness we mention that matching upper bounds can be

* Supported by the Danish Natural Science Research Council (Grant No. 9400044). Partially supported by the ESPRIT Long Term Research Program of the EU under contract #20244 (ALCOM-IT).

** Basic Research in Computer Science, a Centre of the Danish National Research Foundation.

achieved by a $(2,4)$-tree of depth at most $t(n)$ where each leaf stores $\Theta(n/2^{t(n)})$ elements, provided DELETE takes a pointer to the element to be deleted.

In the following we consider the problem on a unit cost RAM with word size w bits allowing addition, arbitrary left and right bit shifts and bit-wise boolean operations on words in constant time. Miltersen [10] refers to this model as a *Practical RAM*. We assume the elements are integers in the range $0..2^w - 1$. A tradeoff similar to the one for the comparison model [6] is not known for a Practical RAM.

A data structure of van Emde Boas *et al.* [15, 16] supports the operations INSERT, DELETE, PRED, FINDMIN and FINDMAX on a Practical RAM in worst case $O(\log w)$ time. For word size $\log^{O(1)} n$ this implies an $O(\log\log n)$ time implementation.

Thorup [14] recently presented a priority queue supporting INSERT and EXTRACTMIN in worst case $O(\log\log n)$ time independently of the word size w. Thorup notes that by tabulating the multiplicity of each of the inserted elements the construction supports DELETE in amortized $O(\log\log n)$ time by skipping extracted integers of multiplicity zero. The data structure of Thorup does not support predecessor queries but Thorup mentions that an $\Omega(\log^{1/3-o(1)} n)$ lower bound for PRED can be extracted from [9, 11]. The space requirement of Thorup's data structure is $O(n2^{\epsilon w})$ (if the time bounds are amortized the space requirement is $O(n + 2^{\epsilon w})$).

Andersson [2] has presented a Practical RAM implementation supporting insertions, deletions and predecessor queries in worst case $O(\sqrt{\log n})$ time and minimum and maximum queries in worst case constant time. The space requirement of Andersson's data structure is $O(n + 2^{\epsilon w})$. Several data structures can achieve the same time bounds as Andersson [2], but they all require constant time multiplication [3, 7, 13].

The main result of this paper is Theorem 1 stated below. The theorem requires the notion of *smooth* functions. Overmars [12] defines a nondecreasing function f to be smooth if and only if $f(O(n)) = O(f(n))$.

Theorem 1. *Let $f(n)$ be a nondecreasing smooth function satisfying $\log\log n \leq f(n) \leq \sqrt{\log n}$. On a Practical RAM a data structure exists supporting* INSERT *and* DELETE *in worst case $O(f(n))$ time,* PRED *in worst case $O((\log n)/f(n))$ time and* FINDMIN *and* FINDMAX *in worst case constant time, where n is the number of integers stored. The space required is $O(n2^{\epsilon w})$ for any constant $\epsilon > 0$.*

If $f(n) = \log\log n$ we achieve the result of Thorup but in the worst case sense, i.e. we can support INSERT, EXTRACTMIN and DELETE in worst case $O(\log\log n)$ time. We can support PRED queries in worst case $O(\log n/\log\log n)$ time. The data structure is the first allowing predecessor queries in $O(\log n/\log\log n)$ time while having $O(\log\log n)$ update time. If $f(n) = \sqrt{\log n}$, we achieve time bounds matching those of Andersson [2].

The basic idea of our construction is to apply the data structure of van Emde Boas *et al.* [15, 16] for $O(f(n))$ levels and then switch to a packed search tree of height $O(\log n/f(n))$. This is very similar to the data structure of Andersson [2].

But where Andersson uses $O(\log n/f(n))$ time to update his packed B-tree, we only need $O(f(n))$ time. The idea we apply to achieve this speedup is to add *buffers* of delayed insertions and deletions to the search tree, such that we can work on several insertions concurrently by using the word parallelism of the Practical RAM. The idea of adding buffers to a search tree has in the context of designing I/O efficient data structures been applied by Arge [4].

Throughout this paper we w.l.o.g. assume DELETE only deletes integers actually contained in the set and INSERT never inserts an already inserted integer. This can be satisfied by tabulating the multiplicity of each inserted integer.

In the description of our data structure we in the following assume n is a constant such that the current number of integers in the set is $\Theta(n)$. This can be satisfied by using the general dynamization technique described by Overmars [12], which requires $f(n)$ to be smooth. In Sect. 2 if we write $\log^5 n \leq k$, we actually mean that k is a function of n, but because we assume n to be a constant k is also assumed to be a constant.

In Sect. 2 we describe our packed search trees with buffers. In Sect. 3 we describe how to perform queries in a packed search tree and in Sect. 4 how to update a packed search tree. In Sect. 5 we combine the packed search trees with a range reduction based on the data structure of van Emde Boas *et al.* [15, 16] to achieve the result stated in Theorem 1. Section 6 contains some concluding remarks and lists some open problems.

2 Packed search trees with buffers

In this and the following two sections we describe how to maintain a set of integers of w/k bits each, for k satisfying $\log^5 n \leq k \leq w/\log n$. The bounds we achieve are:

Lemma 2. *Let k satisfy $\log^5 n \leq k \leq w/\log n$. If the integers to be stored are of w/k bits each then on a Practical RAM* INSERT *and* DELETE *can be supported in worst case $O(\log k)$ time,* PRED *in worst case $O(\log k + \log n/\log k)$ time and* FINDMIN *and* FINDMAX *in worst case constant time. The space required is $O(n)$.*

The basic idea is to store $O(k)$ integers in each word and to use the word parallelism of the Practical RAM to work on $O(k)$ integers in parallel in constant time. In the following we w.l.o.g. assume that we can apply Practical RAM operations to a list of $O(k)$ integers stored in $O(1)$ words in worst cast constant time. Together with each integer we store a *test bit*, as in [1, 2, 14]. An integer together with the associated test bit is denoted a *field*. Figure 1 illustrates the structure of a list of maximum capacity k containing $\ell \leq k$ integers $x_1, \ldots, x_\ell$. A field containing the integer x_i has a test bit equal to zero. The remaining $k - \ell$ empty fields store the integer zero and a test bit equal to one.

Essential to the data structure to be described is the following lemma due to Albers and Hagerup [1].

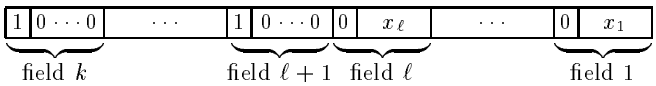

Fig. 1. The structure of a list of maximum capacity k, containing integers $x_1, \ldots, x_\ell$.

Lemma 3 Albers and Hagerup. *On a Practical RAM two sorted lists each of at most $O(k)$ integers stored in $O(1)$ words can be merged into a single sorted list stored in $O(1)$ words in $O(\log k)$ time.*

Albers and Hagerup's proof of Lemma 3 is a description of how to implement the bitonic merging algorithm of Batcher [5] in a constant number of words on the Practical RAM. The algorithm of Albers and Hagerup does not handle partial full lists as defined (all test bits are assumed to be zero), but it is straightforward to modify their algorithm to do so, by considering an integer's test bit as the integer's most significant bit. A related lemma we need for our construction is the following:

Lemma 4. *Let k satisfy $k \leq w/\log n$. Let A and B be two sorted and repetition free lists each of at most $O(k)$ integers stored in $O(1)$ words on a Practical RAM. Then the sorted list $A \setminus B$ can be computed and stored in $O(1)$ words in $O(\log k)$ time.*

Proof. Let C be the list consisting of A merged with B twice. By Lemma 3 the merging can be done in worst case $O(\log k)$ time. By removing all integers appearing at least twice from C we get $A \setminus B$. In the following we outline how to eliminate these repetitions from C. Tedious implementation details are omitted.

First a mask is constructed corresponding to the integers only appearing once in C. This can be done in worst case constant time by performing the comparisons between neighbor integers in C by subtraction like the mask construction described in [1]. The integers appearing only once in C are compressed to form a single list as follows. First a prefix sum computation is performed to calculate how many fields each integer has to be shifted to the right. This can be done in $O(\log k)$ time by using the constructed mask. Notice that each of the calculated values is an integer in the range $0, \ldots, |A| + 2|B|$, implying that each field is required to contain at least $O(\log k)$ bits. Finally we perform $O(\log k)$ iterations where we in the i'th iteration move all integers x_j, 2^i fields to the right if the binary representation of the number of fields x_j has to be shifted has the i'th bit set. A similar approach has been applied in [1] to reverse a list of integers. □

The main component of our data structure is a search tree T where all leaves have equal depth and all internal nodes have degree at least one and at most $\Delta \leq k/\log^4 n$. Each leaf v stores a sorted list I_v of between $k/2$ and k integers. With each internal node v of degree $d(v)$ we store $d(v)-1$ keys to guide searches. The $d(v)$ pointers to the children of v can be packed into a single word because

they require at most $d(v)\log n \leq w$ bits, provided that the number of nodes is less than n.

This part of the data structure is quite similar to the packed B-tree described by Andersson [2]. To achieve faster update times for INSERT and DELETE than Andersson, we add buffers of delayed INSERT and DELETE operations to each internal node of the tree.

With each internal node v we maintain a buffer I_v containing a sorted list of integers to be inserted into the leaves of the subtree T_v rooted at v, and a buffer D_v containing a sorted list of integers to be deleted from T_v. We maintain the invariants that I_v and D_v are disjoint and repetition free, and that

$$\max\{|I_v|, |D_v|\} < \Delta \log n \quad . \tag{1}$$

The set S_v of integers stored in a subtree T_v can recursively be defined as

$$S_v = \begin{cases} I_v & \text{if } v \text{ is a leaf,} \\ I_v \cup ((\bigcup_{w \text{ a child of } v} S_w) \setminus D_v) & \text{otherwise.} \end{cases} \tag{2}$$

Finally we maintain two nonempty global buffers of integers L and R each of size $O(k)$ to be able to answer minimum and maximum queries in constant time. The integers in L are less than all other integers stored, and the integers in R are greater than all other integers stored.

Let h denote the height of T. In Sect. 4 we show how to guarantee that $h = O(\log n / \log k)$, implying that the number of nodes is $O(hn/k) = O(n)$.

3 Queries in packed search trees

By explicitly remembering the minimum integer in L and the maximum integer in R it is trivial to implement FINDMIN and FINDMAX in worst case constant time. A PRED(e) query can be answered as follows. If $e \leq \max(L)$ then the predecessor of e is contained in L and can be found in worst case $O(\log k)$ time by standard techniques. If $\min(R) \leq e$ then the predecessor of e is contained in R. Otherwise we have to search for the predecessor of e in T.

We first perform a search for e in the search tree T. The implementation of the search for e in T is identical to how Andersson searches in a packed B-tree [2]. We refer to [2] for details. Let λ be the leaf reached and $w_1, \ldots, w_{h-1}$ be the internal nodes on the path from the root to λ. Define $w_h = \lambda$. Because we have introduced buffers at each internal node of T the predecessor of e does not necessarily have to be stored in I_λ but can also be contained in one of the insert buffers I_{w_i}. An integer $a \in I_{w_i}$ can only be a predecessor of e if it has not been deleted by a delayed delete operation, i.e. $a \notin D_{w_j}$ for $1 \leq j < i$. It seems necessary to *flush* all buffers I_{w_i} and D_{w_i} for integers which should be inserted in or deleted from I_λ to be able to find the predecessor of e. If dom_λ denotes the interval of integers spanned by the leaf λ, the buffers I_{w_i} and D_{w_i} can be

flushed for elements in dom_λ by the following sequence of operations:

$$\begin{aligned} I_{w_{i+1}} &\leftarrow I_{w_{i+1}} \setminus (D_{w_i} \cap \mathrm{dom}_\lambda) \cup (I_{w_i} \cap \mathrm{dom}_\lambda) \setminus D_{w_{i+1}} \ , \\ D_{w_{i+1}} &\leftarrow D_{w_{i+1}} \setminus (I_{w_i} \cap \mathrm{dom}_\lambda) \cup (D_{w_i} \cap \mathrm{dom}_\lambda) \setminus I_{w_{i+1}} \ , \\ I_{w_i} &\leftarrow I_{w_i} \setminus \mathrm{dom}_\lambda \ , \\ D_{w_i} &\leftarrow D_{w_i} \setminus \mathrm{dom}_\lambda \ . \end{aligned}$$

Let $\hat{I}_\lambda$ denote the value of I_λ after flushing all buffers I_{w_i} and D_{w_i} for integers in the range dom_λ. From (2) it follows that $\hat{I}_\lambda$ can also be computed directly by the expression

$$\hat{I}_\lambda = \mathrm{dom}_\lambda \cap (((\cdots((I_\lambda \setminus D_{w_{h-1}}) \cup I_{w_{h-1}}) \cdots) \setminus D_{w_1}) \cup I_{w_1}) \ . \tag{3}$$

Based on Lemmas 3 and 4 we can compute this expression in $O(h \log k)$ time. This is unfortunately $O(\log n)$ for the tree height $h = \log n / \log k$. In the following we outline how to find the predecessor of e in $\hat{I}_\lambda$ without actually computing $\hat{I}_\lambda$ in $O(\log k + \log n / \log k)$ time.

Let I'_{w_i} be $I_{w_i} \cap \mathrm{dom}_\lambda \cap]\infty, e]$ for $i = 1, \ldots, h$. An alternative expression to compute the predecessor of e in $\hat{I}_\lambda$ is

$$\max \bigcup_{i=1,\ldots,h} (I'_{w_i} \setminus \bigcup_{j=1,\ldots,i-1} D_{w_j}) \ . \tag{4}$$

Because $|\bigcup_{j=1,\ldots,h-1} D_{w_j}| < \Delta \log^2 n$ we can w.l.o.g. assume $|I'_{w_h}| \leq \Delta \log^2 n$ in (4) by restricting our attention to the $\Delta \log^2 n$ largest integers in I'_{w_h}, i.e. all sets involved in (4) have size at most $\Delta \log^2 n$. The steps we perform to compute (4) are the following. All implementation details are omitted.

- First all buffers I_{w_i} and D_{w_i} for $i < h$ are inserted into a single word $\mathcal{W}$ where the contents of $\mathcal{W}$ is considered as $2h-2$ independent lists each of maximum capacity $\Delta \log^2 n$. This can be done in $O(h) = O(\log n / \log k)$ time.
- Using the word parallelism of the Practical RAM we now for all I_{w_i} compute I'_{w_i}. This can be done in $O(\log k)$ time if $\min(\mathrm{dom}_\lambda)$ is known. The integer $\min(\mathrm{dom}_\lambda)$ can be computed in the search phase determining the leaf λ. $\mathcal{W}$ now contains I'_{w_i} and D_{w_i} for $i < h$.
- The value of I'_{w_h} is computed (satisfying $|I'_{w_h}| \leq \Delta \log^2 n$) and appended to $\mathcal{W}$. This can be done in $O(\log k)$ time. The contents of $\mathcal{W}$ is now

$$I'_{w_h} D_{w_{h-1}} I'_{w_{h-1}} \cdots D_{w_1} I'_{w_1} \ .$$

- Let $\mathcal{W}_I = (I'_{w_h})^{h-1} \cdots (I'_{w_1})^{h-1}$ and $\mathcal{W}_D = (D_{w_{h-1}} \cdots D_{w_1})^h$. See Fig. 2. The number of fields required in each word is $h(h-1)\Delta \log^2 n \leq \Delta \log^4 n \leq k$. The two words can be constructed from $\mathcal{W}$ in $O(\log k)$ time.

$\mathcal{W}_I$	I'_{w_h}	$\cdots$	I'_{w_h}	I'_{w_h}	$\cdots$	I'_{w_1}	$\cdots$	I'_{w_1}	I'_{w_1}
$\mathcal{W}_D$	$D_{w_{h-1}}$	$\cdots$	D_{w_2}	D_{w_1}	$\cdots$	$D_{w_{h-1}}$	$\cdots$	D_{w_2}	D_{w_1}
$\mathcal{W}_M$	$M_{h,h-1}$	$\cdots$	$M_{h,2}$	$M_{h,1}$	$\cdots$	$M_{1,h-1}$	$\cdots$	$M_{1,2}$	$M_{1,1}$

Fig. 2. The structure of the words $\mathcal{W}_I$, $\mathcal{W}_D$ and $\mathcal{W}_M$.

- From $\mathcal{W}_I$ and $\mathcal{W}_D$ we now construct $h(h-1)$ masks $M_{i,j}$ such that $M_{i,j}$ is a mask for the fields of I'_{w_i} which are not contained in D_{w_j}. See Fig. 2. The construction of a mask $M_{i,j}$ from the two list I'_{w_i} and D_{w_j} is very similar to the proof of Lemma 4 and can be done as follows in $O(\log k)$ time.
  First I is merged with D twice (we omit the subscripts while outlining the mask construction). Let C be the resulting list. From C construct in constant time a mask C' that contains ones in the fields in which C stores an integer only appearing once in C and zero in all other fields. By removing all fields from C having *exactly* one identical neighbor we can recover I from C. By removing the corresponding fields from C' we get the required mask M. As an example assume $I = (7,5,4,3,1)$ and $D = (6,5,2)$. Then $C = (7,\underline{6},\underline{6},\underline{5},5,\underline{5},4,3,\underline{2},\underline{2},1)$, $C' = (1,\underline{0},\underline{0},\underline{0},0,\underline{0},1,1,\underline{0},\underline{0},1)$ and $M = (1,0,1,1,1)$ where underlined fields are the fields in C having exactly one identical neighbor.
- We now compute masks $M_i = \bigwedge_{j=1,\ldots,i-1} M_{i,j}$ for all i. By applying M_i to I'_{w_i} we get $I'_{w_i} \setminus \bigcup_{j=1,\ldots,i-1} D_{w_j}$. This can be done in $O(\log k)$ time from $\mathcal{W}_M$ and $\mathcal{W}_I$.
- Finally we in $O(\log k)$ time compute (4) as the maximum over all the integers in the sets computed in the previous step. Notice that it can easily be checked if e has a predecessor in $\hat{I}_\lambda$ by checking if all the sets computed in the previous step are empty.

We conclude that the predecessor of e in $\hat{I}_\lambda$ can be found in $O(\log k + h) = O(\log k + \log n/\log k)$ time.

If e does not have a predecessor in $\hat{I}_\lambda$ there are two cases to consider. The first is if there exists a leaf $\bar{\lambda}$ to the left of λ. Then the predecessor of e is the largest integer in $\hat{I}_{\bar{\lambda}}$. Notice that $\hat{I}_{\bar{\lambda}}$ is nonempty because $|\bigcup_{j=1,\ldots,h-1} D_{\bar{w}_j}| < |I_{\bar{\lambda}}|$. If λ is the leftmost leaf the predecessor of e is the largest integer in L. We conclude that PRED queries can be answered in worst case $O(\log k + \log n/\log k)$ time on a Practical RAM.

4 Updating packed search trees

In the following we describe how to perform INSERT and DELETE updates. We first give a solution achieving the claimed time bounds in the amortized sense. The amortized solution is then converted into a worst case solution by standard techniques.

We first consider INSERT(e). If $e < \max(L)$ we insert e into L in $\log k$ time, remove the maximum from L such that $|L|$ remains unchanged, and let e become the removed integer. If $\min(R) < e$ we insert e in R, remove the minimum from R, and let e become the removed integer.

Let r denote the root of T. If $e \in D_r$, remove e from D_r in worst case $O(\log k)$ time, i.e. INSERT(e) cancels a delayed DELETE(e) operation. Otherwise insert e into I_r.

If $|I_r| < \Delta \log n$ this concludes the INSERT operation. Otherwise there must exist a child w of r such that $\log n$ integers can be moved from I_r to the subtree rooted at w. The child w and the $\log n$ integers X to be moved can be found by a binary search using the search keys stored at r in worst case $O(\log k)$ time. We omit the details of the binary search in I_r. We first remove the set of integers X from I_r such that $|I_r| < \Delta \log n$. We next remove all integers in $X \cap D_w$ from X and from D_w in $O(\log k)$ time by Lemma 4, i.e. we let delayed deletions be cancel out by delayed insertions. The remaining integers in X are merged into I_w in $O(\log k)$ time. Notice that I_w and D_w are disjoint after the merging and that if w is an internal node then $|I_w| < (\Delta + 1) \log n$.

If $|I_w| \geq \Delta \log n$ and w is not a leaf we recursively apply the above to I_w. If w is a leaf and $|I_w| \leq k$ we are done. The only problem remaining is if w is a leaf and $k < |I_w| \leq k + \log n \leq 2k$. In this case we split the leaf w into two leaves each containing between $k/2$ and k integers, and update the search keys and child pointers stored at the parent of w. If the parent p of w now has $\Delta + 1$ children we split p into two nodes of degree $\geq \Delta/2$ while distributing the buffers I_p and D_p among the two nodes w.r.t. the new search key. The details of how to split a node is described in [2]. If the parent of p gets degree $\Delta + 1$ we recursively split the parent of p.

The implementation of inserting e in T takes worst case $O(h \log k)$ time. Because the number of leaves is $O(n)$ and that T is similar to a B-tree if we only consider insertions we get that the height of T is $h = O(\log n / \log \Delta) = O(\log n / \log(k / \log^4 n)) = O(\log n / \log k)$ because $k \geq \log^5 n$. It follows that the worst case insertion time in T is $O(\log n)$. But because we remove $\log n$ integers from I_r every time $|I_r| = \Delta \log n$ we spend at most worst case $O(\log n)$ time once for every $\log n$ insertion. All other insertions require worst case $O(\log k)$ time. We conclude that the amortized insertion time is $O(\log k)$.

We now describe how to implement DELETE(e) in amortized $O(\log k)$ time. If e is contained in L we remove e from L. If L is nonempty after having removed e we are done. If L becomes empty we proceed as follows. Let λ be the leftmost leaf of T. The basic idea is to let L become $\hat{I}_\lambda$. We do this as follows. First we flush all buffers along the leftmost path in the tree for integers contained in dom_λ. Based on (3) this can be done in $O(h \log k)$ time. We can now assume $(I_w \cup D_w) \cap \text{dom}_\lambda = \emptyset$ for all nodes w on the leftmost path and that $I_\lambda = \hat{I}_\lambda$. We can now assign L the set I_λ and remove the leaf λ. If the parent p of λ gets degree zero we recursively remove p. Notice that if p gets degree zero then I_p and D_p are both empty. Because the total size of the of insertion and deletion buffers on the leftmost path is bounded by $h\Delta \log n \leq k / \log^2 n$ it follows that

$\log n \leq k/2 - k/\log^2 n \leq |L| \leq k + k/\log^2 n$. It follows that L cannot become empty throughout the next $\log n$ DELETE operations. The case $e \in R$ is handled symmetrically by letting λ be the rightmost leaf.

If $e \notin L \cup R$ we insert e in D_r provided $e \notin I_r$. If $e \in I_r$ we remove e from I_r in $O(\log k)$ time and are done. If $|D_r| \geq \Delta \log n$ we can move $\log n$ integers X from D_r to a child w of r. If w is an internal node we first remove $X \cap I_w$ from X and I_w, i.e. delayed insertions cancels delayed insertions, and then inserts the remaining elements in X into D_w. If $|D_w| \geq \Delta \log n$ we recursively move $\log n$ integers from D_w to a child of w. If w is a leaf λ we just remove the integers X from I_λ. If $|I_\lambda| \geq k/2$ we are done. Otherwise let $\bar{\lambda}$ denote the leaf to the right or left of λ (If $\bar{\lambda}$ does not exist the set only contains $O(k)$ integers and the problem is easy to handle. In the following we w.l.o.g. assume $\bar{\lambda}$ exists). We first flush all buffers on the paths from the root r to λ and $\bar{\lambda}$ such that the buffers do not contain elements from $\mathrm{dom}_\lambda \cup \mathrm{dom}_{\bar{\lambda}}$. This can be done in $O(h \log n)$ time as previously described. From

$$k/2 + k/2 - \log n - 2h\Delta \log n \leq |I_\lambda \cup I_{\bar{\lambda}}| \leq k/2 + k - 1 + 2h\Delta \log n$$

it follows that $k/2 \leq |I_\lambda \cup I_{\bar{\lambda}}| \leq 2k$. There are two cases to consider. If $|\lambda + \bar{\lambda}| \geq k$ we redistribute I_λ and $I_{\bar{\lambda}}$ such that they both have size at least $k/2$ and at most k. Because all buffers on the path from λ ($\bar{\lambda}$) to the root intersect empty with $\mathrm{dom}_\lambda \cup \mathrm{dom}_{\bar{\lambda}}$ we in addition only need to update the search key stored at the nearest common ancestor of λ and $\bar{\lambda}$ in T which separates dom_λ and $\mathrm{dom}_{\bar{\lambda}}$. This can be done in $O(h + \log k)$ time. The second case is if $|\lambda + \bar{\lambda}| < k$. We then move the integers in I_λ to $I_{\bar{\lambda}}$ and remove the leaf λ as described previously. The total worst case time for a deletion becomes $O(h \log k) = O(\log n)$. But again the amortized time is $O(\log k)$ because L and R become empty for at most every $\log n$'th DELETE operation, and because D_r becomes full for at most every $\log n$'th DELETE operation.

In the previous description of DELETE we assumed the height of T is $h = O(\log n / \log k)$. We argued that this was true if only INSERT operations were performed because then our search tree is similar to a B-tree. It is easy to see that if only $O(n)$ leaves have been remove, then the height of T is still $h = O(\log n / \log k)$. One way to see this is by assuming that all removed nodes still resist in T. Then T has at most $O(n)$ leaves and each internal node has degree at least $\Delta/2$, which implies the claimed height. By rebuilding T completely such that all internal nodes have degree $\Theta(\Delta)$ for every n'th DELETE operation we can guarantee that at most n leaves have been removed since T was rebuild the last time. The rebuilding of T can easily be done in $O(n \log k)$ time implying that the amortized time for DELETE only increases by $O(\log k)$.

We conclude that INSERT and DELETE can be implemented in amortized $O(\log k)$ time. The space required is $O(n)$ because each node can be stored in $O(1)$ words.

To convert the amortized time bounds into worst case time bounds we apply the standard technique of incrementally performing a worst case expensive operation over the following sequence of operations by moving the expensive

operation into a shadow process that is executed in a quasi-parallel fashion with the main algorithm. The rebuilding of T when $O(n)$ DELETE operations have been performed can be handled by the general dynamization technique of Overmars [12] in worst case $O(\log k)$ time per operation. For details refer to [12]. What remains to be described is how to handle the cases when L or R becomes empty and when I_r or D_r becomes full. The basic idea is to handle these cases by simply avoiding them. Below we outline the necessary changes to the amortized solution.

The idea is to allow I_r and D_r to have size $\Delta \log n + O(\log n)$ and to divide the sequence of INSERT and DELETE operations into phases of $\log n/4$ operations. In each phase we perform one of the transformations below to T incrementally over the $\log n/4$ operations of the phase by performing worst case $O(1)$ work per INSERT or DELETE operation. We cyclic choose which transformation to perform, such that for each $\log n$'th operation each transformation has been performed at least once. Each of the transformations can be implemented in worst case $O(\log n)$ time as described in the amortized solution.

- If $|L| < k$ at the start of the phase and λ denotes the leftmost leaf of T we incrementally merge L with $\hat{I}_\lambda$ and remove the leaf λ. It follows that L always has size at least $k - O(\log n) > 0$.
- The second transformation similarly guarantees that $|R| > 0$ by merging R with $\hat{I}_\lambda$ where λ is rightmost leaf of T if $|R| < k$.
- If $|I_r| \geq \Delta \log n$ at the start of the phase we incrementally remove $\log n$ integers from I_r. It follows that the size of I_r is bounded by $\Delta \log n + O(\log n) = O(k)$.
- The last transformation similarly guarantees that the size of D_r is bounded by $\Delta \log n + O(\log n)$ by removing $\log n$ integers from D_r if $|D_r| \geq \Delta \log n$.

This finishes our description of how to achieve the bounds stated in Lemma 2.

5 Range reduction

To prove Theorem 1 we combine Lemma 2 with a range reduction based on a data structure of van Emde Boas *et al.* [15, 16]. This is similar to the data structure of Andersson [2], and for details we refer to [2]. We w.l.o.g. assume $w \geq 2^{f(n)} \log n$.

The idea is to use the topmost $f(n)$ levels of the data structure of van Emde Boas *et al.* and then switch to our packed search trees. If $f(n) \geq 5 \log\log n$ the integers we need to store are of $w/2^{f(n)} \leq w/\log^5 n$ bits each and Lemma 2 applies for $k = 2^{f(n)}$. By explicitly remembering the minimum and maximum integer stored FINDMIN and FINDMAX are trivial to support in worst case constant time. The remaining time bounds follow from Lemma 2. The space bound of $O(n2^{\varepsilon w})$ follows from storing the arrays at each of the $O(n)$ nodes in the data structure of van Emde Boas *et al.* as a trie of degree $2^{\varepsilon w}$.

6 Conclusion

We have presented the first data structure for a Practical RAM allowing the update operations INSERT and DELETE in worst case $O(\log\log n)$ time while answering PRED queries in worst case $O(\log n/\log\log n)$ time. An interesting open problem is if it is possible to support INSERT and DELETE in worst case $O(\log\log n)$ time and PRED in worst case $O(\sqrt{\log n})$ time. The general open problem is to find a tradeoff between the update time and the time for predecessor queries on a Practical RAM.

Acknowledgments

The author thanks Theis Rauhe, Thore Husfeldt and Peter Bro Miltersen for encouraging discussions, and the referees for comments.

References

1. Susanne Albers and Torben Hagerup. Improved parallel integer sorting without concurrent writing. In *Proc. 3rd ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 463–472, 1992.
2. Arne Andersson. Sublogarithmic searching without multiplications. In *Proc. 36th Ann. Symp. on Foundations of Computer Science (FOCS)*, pages 655–663, 1995.
3. Arne Andersson. Faster deterministic sorting and searching in linear space. In *Proc. 37th Ann. Symp. on Foundations of Computer Science (FOCS)*, pages 135–141, 1996.
4. Lars Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proc. 4th Workshop on Algorithms and Data Structures (WADS)*, volume 955 of *Lecture Notes in Computer Science*, pages 334–345. Springer Verlag, Berlin, 1995.
5. Kenneth E. Batcher. Sorting networks and their applications. In *Proc. AFIPS Spring Joint Computer Conference, 32*, pages 307–314, 1968.
6. Gerth Stølting Brodal, Shiva Chaudhuri, and Jaikumar Radhakrishnan. The randomized complexity of maintaining the minimum. In *Proc. 5th Scandinavian Workshop on Algorithm Theory (SWAT)*, volume 1097 of *Lecture Notes in Computer Science*, pages 4–15. Springer Verlag, Berlin, 1996.
7. Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47:424–436, 1993.
8. Scott Huddleston and Kurt Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.
9. Peter Bro Miltersen. Lower bounds for Union-Split-Find related problems on random access machines. In *Proc. 26th Ann. ACM Symp. on Theory of Computing (STOC)*, pages 625–634, 1994.
10. Peter Bro Miltersen. Lower bounds for static dictionaries on RAMs with bit operations but no multiplications. In *Proc. 23rd Int. Colloquium on Automata, Languages and Programming (ICALP)*, volume 1099 of *Lecture Notes in Computer Science*, pages 442–453. Springer Verlag, Berlin, 1996.

11. Peter Bro Miltersen, Noam Nisan, Shmuel Safra, and Avi Wigderson. On data structures and asymmetric communication complexity. In *Proc. 27th Ann. ACM Symp. on Theory of Computing (STOC)*, pages 103–111, 1995.
12. Mark H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 1983.
13. Rajeev Raman. Priority queues: Small, monotone and trans-dichotomous. In *ESA '96, Algorithms*, volume 1136 of *Lecture Notes in Computer Science*, pages 121–137. Springer Verlag, Berlin, 1996.
14. Mikkel Thorup. On RAM priority queues. In *Proc. 7th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 59–67, 1996.
15. Peter van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6:80–82, 1977.
16. Peter van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

This article was processed using the LaTeX macro package with LLNCS style