

# Word-based RAM Priority Queues

a semester project by *Che-Bin Liu*

This report is aimed at the word-based RAM priority queues. Suppose we have a  $w$ -bit RAM and want to design priority queues that only handle integer keys represented by  $w$  bits. We denote the universe as  $U = \{1, 2, \dots, u\}$ <sup>1</sup> where  $u = 2^w$ . A priority queue  $S$  that we consider supports the following operations:

- $\text{MIN}(S)$ : Return the element of  $S$  with the smallest key.
- $\text{INSERT}(S, x)$ : Insert the element  $x$  into  $S$ .
- $\text{DELETE}(S, x)$ : Delete the element  $x$  from  $S$ .
- $\text{EXTRACTMIN}(S)$ : Delete and return the element of  $S$  with the smallest key.

A widely known and popular data structure to support priority queues is the *heap* data structure. A heap supports operations of priority queues in  $O(\log n)$  time where  $n$  is the number of elements in a queue. However, with a unit-cost RAM model, the priority queue operations can be significantly speeded up. In the literature, this was first proposed by van Emde Boas [3, 4] whose data structure supported priority queue operations in  $O(\log \log u)$  worst-case time. Fredman and Willard's Fusion Trees [5, 6, 7] gave these operations an  $O(\sqrt{\log n})$  time bound. Andersson's search trees [2] also provided  $O(\sqrt{\log n})$  time bound but only used  $\text{AC}^0$  operations. And finally Thorup proposed an  $O(\log \log n)$  algorithm [8] which is known so far as the fastest data structure supporting priority queues.

In this report, we start with the van Emde Boas Trees. We then talk about Thorup's algorithm. At the end, we compare four different data structures supporting priority queue operations.

## 1 van Emde Boas Trees

The most important result of this data structure is that it supports priority queue operations in  $O(\log \log u)$  time. If  $u$  is polynomial in  $n$  (i.e.  $u = O(n^c)$ ), then we have  $O(\log \log n)$  per operation which is an exponential speedup. van Emde Boas et al. [4] provided some different ways to achieve this time bound. We here will talk about one of them.

Consider that we divide the universe  $U$  into  $\sqrt{u}$  blocks of size  $\sqrt{u}$ . So the  $i$ th ( $i = 1, 2, \dots, \sqrt{u}$ ) block consists the integer set  $\{(i-1)\sqrt{u} + 1, (i-1)\sqrt{u} + 2, \dots, i\sqrt{u}\}$ . Each block is also recursively divided in the same way until the block size becomes 1. This process can be viewed as recursive bit-decomposition. For an integer  $x$ , we denote its higher half bits as  $x_h$  and its lower half bits as  $x_l$ . Therefore,  $x_h$  is used to index the block (a subtree

<sup>1</sup>The set of  $w$ -bit integers should be  $\{0, 1, \dots, 2^w - 1\}$  in the binary representation. However, without loss of generality, we use  $\{1, 2, \dots, 2^w\}$  through this report to simply the notations.

containing integers with  $x_h$  in the corresponding bits) and  $x_l$  is the corresponding key in the indexed subtree. This data structure is called a van Emde Boas tree or a van Emde Boas queue.

To summarize, for a  $w$ -bit vEB tree  $S$ , we maintain the minimum value in  $S$  ( $\min(S)$ ), the number of items in  $S$  ( $n(S)$ ), a  $w/2$ -bit vEB tree  $H$  on higher  $w/2$  bits  $x_h$ . For each  $x_h \in H$ , we also maintain a  $w/2$ -bit vEB tree  $L(x_h)$  on lower  $w/2$  bits  $x_l$ . In addition, we need a look-up table  $T$  (e.g. an array of size  $2^{w/2}$ ) to index lower  $w/2$ -bit vEB trees ( $T(x_h) = L(x_h)$ ).

$\text{MIN}(S)$  can be done in  $O(1)$  by simply returning the minimum value in the data structure.  $\text{INSERT}(S, x)$  and  $\text{EXTRACTMIN}(S)$  of this implementation are shown as follows.  $\text{DELETE}(S, x)$  can be done similarly. And note that the minimum value is never inserted. It is simply recorded as the minimum.

```

INSERT(S, x) :
  if n(S) = 0
    min(S) ← x
  else
    if x < min(S)
      min(S) ↔ x (interchange)
    end
    (xh, xl) ← x (decompose)
    if n(T(xh)) > 0 (xh ∈ H)
      INSERT(T(xh), xl)
    else
      INSERT(H, xh)
      INSERT(T(xh), xl) (O(1) time)
    end
  end
  n(S) ← n(S) + 1

```

```

EXTRACTMIN(S)
  if n(S) = 0
    return NULL
  end
  (mh, ml) ← min(S) (decompose)
  if n(T(mh)) = 1
    EXTRACTMIN(H)
    x'h ← min(H)
    x'l ← min(T(x'h))
  else
    EXTRACTMIN(T(mh))
    x'h ← mh
    x'l ← min(T(mh))
  end
  n(S) ← n(S) - 1
  min(S) ← (x'h, x'l)
  return (mh, ml)

```

## 1.1 Time and Space Bounds

In this implementation, either  $\text{INSERT}$  or  $\text{EXTRACTMIN}$  has only one recursive call in the respective function, and the other operations can all be done in constant time. So the time  $T(u)$  is given by the recurrence

$$T(u) = T(\sqrt{u}) + O(1)$$

or

$$T(w) = T(w/2) + O(1)$$

where  $u$  is the number in universe and  $w$  is the number of bits. According to Master's Theorem, we have  $T(w) = O(\log w)$ . Therefore, the algorithm runs in  $O(\log \log u)$  time.

Since a  $w$ -bit vEB tree ( $w = \log u$ ) has  $\sqrt{u}$  lower  $w/2$ -bit vEB trees, one higher  $w/2$ -bit vEB tree, and a look-up table of size  $O(\sqrt{u})$ , the space  $S(u)$  occupied by the data structure is given by the recurrence

$$S(u) = (\sqrt{u} + 1)S(\sqrt{u}) + O(\sqrt{u}).$$

From this recurrence, we show that this vEB data structure uses  $\Theta(u)$  space.

**Proof:** We prove this by mathematical induction.

- $S(u) \leq c_1u - c_2$ :

Assume by induction that  $S(k) \leq c_1k - c_2$  for all  $k < u$ . We have

$$\begin{aligned} S(u) &\leq (\sqrt{u} + 1)(c_1\sqrt{u} - c_2) + O(\sqrt{u}) \\ &= c_1u + c_1\sqrt{u} - c_2\sqrt{u} - c_2 + O(\sqrt{u}) \\ &= c_1u - c_2 + (c_1 - c_2 + O(1))\sqrt{u} \\ &\leq c_1u - c_2 \end{aligned}$$

- $S(u) \geq cu$ :

Assume by induction that  $S(k) \geq ck$  for all  $k < u$ . We have

$$\begin{aligned} S(u) &\geq (\sqrt{u} + 1)c\sqrt{u} + O(\sqrt{u}) \\ &= cu + c\sqrt{u} + O(\sqrt{u}) \\ &\geq cu \end{aligned}$$

Therefore, we have  $S(u) = \Theta(u)$ . □

## 1.2 Improved Space Bound for Stratified Trees

van Emde Boas also proposed a stratified tree data structure [4] which supports priority-queue operations in  $O(\log \log u)$  worst-case processing time per instruction on a RAM. But its storage requires  $O(u \log \log u)$  RAM-words. Here, we will not go over details of the stratified tree. However, we will show that its space bound can be improved to linear space  $O(u)$  [3].

Suppose we have a data structure  $Q(u)$  supporting priority queue operations in space  $O(u)$  with  $O(\log u)$  processing time (e.g. heaps). We also denote van Emde Boas Trees as  $P(u)$ . We now show how to obtain a new data structure  $R(u)$  such that  $R(u)$  supports priority-queue operations in space  $O(u)$  with  $O(\log \log u)$  time. Assume  $u = p \cdot q$ . The idea is essentially to decompose the universe  $U$  into  $p$  clusters of size  $q$ . So the  $i$ th cluster consists the integers  $\{(i-1)q+1, (i-1)q+2, \dots, iq\}$ . We form a two-level priority queue in which the first level is implemented by  $P(p)$  and the second level is implemented by  $Q_i(q)$  ( $i = 1, 2, \dots, p$ ). Any query of element  $x$  should first consult its cluster membership  $C(x)$  on the first level. The  $\text{INSERT}(R, x)$  operation needs to insert  $C(x)$  into queue  $P$  if  $C(x)$  does not exist in  $P$ , and then insert  $x$  into queue  $Q_{C(x)}$ . The  $\text{DELETE}(R, x)$  operation needs to

delete  $C(x)$  from queue  $P$  if the corresponding queue  $Q_{C(x)}$  is empty after removing  $x$ . The  $\text{MIN}(R)$  operation needs to find the cluster  $c$  by  $\text{MIN}(P)$ , and find the the element of  $R$  with the smallest key by  $\text{MIN}(Q_c)$ .

**Proof:** Assume  $u = k \cdot \lceil \log \log k \rceil$  so that  $p = k$  and  $q = \lceil \log \log k \rceil$ . Obviously, the membership  $C(x)$  can be computed in a constant time. Therefore, we consider the time and space bound of  $R(u)$  required on operations of  $P(p)$  and  $Q_i(q)$ .

- Time:

Each operation on  $R(u)$  needs one operation on  $P(p)$  and one operation on  $Q_i(q)$ . Therefore, the time bound for  $R(u)$  is

$$\begin{aligned} O(\log \log p) + O(\log q) &= O(\log \log k) + O(\log \lceil \log \log k \rceil) \\ &= O(\log \log k) \\ &= O(\log \log u) \end{aligned}$$

- Space:

The space needed for  $R(u)$  is the space needed for one  $P(p)$  and  $p$   $Q_i(q)$ . Therefore, the space bound for  $R(u)$  is

$$\begin{aligned} O(p \log \log p) + pO(q) &= O(k \log \log k) + kO(\lceil \log \log k \rceil) \\ &= O(k \log \log k) \\ &= O(u) \end{aligned}$$

□

Note that an unsorted linked list can also be used for the second-level data structure  $Q_i(q)$  ( $O(q)$  in time and space) to achieve the same bounds.

### 1.3 Improved Space Bound by Perfect Hashing

To save the needed space, another alternative is to utilize a hash table  $Hash$  to index lower  $w/2$ -bit vEB trees [9]. It uses  $x_h$  as the key and  $L(x_h)$  as the value in the hash table ( $Hash(x_h) = L(x_h)$ ). Note that implementing a hash table to be truly constant time is tricky (e.g. multiplication is not a constant time bit operation). It requires perfect hashing which can be found in the literature. The algorithms are shown as follows.

```

MAKEVEB(x) :
  allocate memory for S
  min(S) ←→ x
  n(S) ← 1
  H ← NULL
  Initialize Hash as empty (O(1) time)
  return S

```

```

INSERT( $S, x$ ) :
  if  $S = \text{NULL}$ 
     $S \leftarrow \text{MAKEVEB}(x)$ 
  else
    if  $x < \text{min}(S)$ 
       $\text{min}(S) \longleftrightarrow x$  (interchange)
    end
     $(x_h, x_l) \leftarrow x$  (decompose)
    if  $\text{Hash}(x_h) \neq \text{NULL}$  ( $x_h \in H$ )
      INSERT( $\text{Hash}(x_h), x_l$ )
    else
      INSERT( $H, x_h$ )
       $L(x_h) \leftarrow \text{MAKEVEB}(x_l)$ 
       $\text{Hash}(x_h) \leftarrow L(x_h)$ 
    end
     $n(S) \leftarrow n(S) + 1$ 
  end

```

```

EXTRACTMIN( $S$ )
  if  $n(S) = 0$ 
    return NULL
  end
   $(m_h, m_l) \leftarrow \text{min}(S)$  (decompose)
  if  $n(\text{Hash}(m_h)) = 1$ 
    remove  $m_h$  from  $\text{Hash}$ 
    EXTRACTMIN( $H$ )
     $x'_h \leftarrow \text{min}(H)$ 
     $x'_l \leftarrow \text{min}(\text{Hash}(x'_h))$ 
  else
    EXTRACTMIN( $\text{Hash}(m_h)$ )
     $x'_h \leftarrow m_h$ 
     $x'_l \leftarrow \text{min}(\text{Hash}(m_h))$ 
  end
   $n(S) \leftarrow n(S) - 1$ 
  if  $n(S) = 0$ 
    free memory of  $S$ 
  else
     $\text{min}(S) \leftarrow (x'_h, x'_l)$ 
  end
  return  $(m_h, m_l)$ 

```

To process each level of recursion in constant time for INSERT, we have to create a vEB tree in constant time, i.e. without recursively creating vEB trees. Therefore, MAKEVEB( $x$ ) simply records the value  $x$  as the minimum value of the newly created vEB structure. The key here is that the minimum value is never inserted and each time of insertion we only expand the data structure for the larger element between current value and old minimum value.

Next, we show that the vEB data structure with hashing uses  $O(n)$  space.

**Proof:** Consider INSERT( $S, x$ ). Note that we only record the minimum value when creating new vEB data structure and do not recursively construct the data structure to the bottom. The hash table also increases the space by a constant factor (amortizing over the constant cost of each newly created data structure). Therefore, we have the space bound  $O(n)$ .  $\square$

## 2 $O(\log \log n)$ Priority Queues

Thorup [8] proposed a data structure supporting INSERT and EXTRACTMIN operations of a priority queue in  $O(\log \log n)$  time. The beauty of this is that the time bound is given by the number of elements  $n$  in the queue instead of the size of the universe  $u$ . The idea came from Albers and Hagerup's list merging in  $O(1)$  words [1]. We here will start with introducing list merging and finally extend to Thorup's  $O(\log \log n)$  priority queues.

**Lemma 1.** *We can merge two sorted lists, each has at most  $k$  keys stored in a single word, into a single sorted list stored in two words in time  $O(\log k)$ .*

**Proof:** Here, a sorted word with at most  $k$  keys means a word of the form  $\{x_{k-1}, \dots, x_0\}$ , where  $x_0 \leq x_1 \leq \dots \leq x_{k-1}$ . To merge two sorted words in  $O(\log k)$  time, we need the bitonic sorting algorithm. A sequence of integers is said to be *bitonic* if it is the concatenation of a nondecreasing sequence and a nonincreasing sequence, or obtained from such a sequence via a cyclic shift. For example, the sequence 5, 7, 6, 3, 1, 2, 4 is bitonic, but 2, 5, 3, 1, 4, 6 is not.

The bitonic sorting algorithm requires input to be a bitonic sequence  $x_0, \dots, x_{h-1}$  where  $h$  is a power of 2. The algorithm works as follows.

```

BITONICSORT( $x$ ) :
   $h \leftarrow |x|$ 
  if  $h = 1$ 
    return  $x$ 
  end
  for  $i = 0$  to  $h/2 - 1$ 
     $m_i \leftarrow \min(x_i, x_{i+h/2})$ 
     $M_i \leftarrow \max(x_i, x_{i+h/2})$ 
  end
   $X_1 \leftarrow \{m_0, \dots, m_{h/2-1}\}$ 
   $X_2 \leftarrow \{M_0, \dots, M_{h/2-1}\}$ 
  return BITONICSORT( $X_1$ ) + BITONICSORT( $X_2$ ) (concatenation)

```

The key of the bitonic sorting is that  $\{m_i\}$  and  $\{M_i\}$  are bitonic, and  $m_i \leq M_j$  for all  $i, j \in \{0, \dots, h/2 - 1\}$ .

To merge two sorted words  $X = \{x_{k-1}, \dots, x_0\}$  and  $Y = \{y_{k-1}, \dots, y_0\}$ , we generate a new sequence  $Z = \{y_0, \dots, y_{k-1}, x_{k-1}, \dots, x_0\}$ . Note that  $Z$  is bitonic. We can then modify the bitonic sorting algorithm to sort the merged word  $Z$ . The detail implementation is described in [1]. It is obvious that merging two sorted words of at most  $k$  keys requires  $O(\log k)$  time.  $\square$

**Corollary 2.** For  $n \geq k$ , given two sorted lists of  $n$  ( $w/k$ )-bit integers, spread over  $n/k$  words, we can merge them into  $2n/k$  words in time  $O(n/k \cdot \log k)$ .

**Proof:** Note that  $w$  is the word size. So we have  $k$  sorted integers in each word. Therefore, we can use Lemma 1 to derive an algorithm to achieve this time bound.

```

MERGESORTEDLISTS( $x, y$ ) :
   $z \leftarrow \text{NULL}$ 
  while  $|x| > 0$  and  $|y| > 0$  ( $|\cdot|$  means the length in words)
    Pop first word of  $x$  and  $y$ , namely  $x_0$  and  $y_0$  respectively
    Merge  $x_0$  and  $y_0$  using Lemma 1, obtaining  $w_0w_1$ 
     $z \leftarrow z + w_0$ 
    if  $\max(w_1) \in x_0$ 
       $x \leftarrow w_1 + x$ 
    else
       $y \leftarrow w_1 + y$ 
    end
  end
   $z \leftarrow z + x + y$  (either  $x$  or  $y$  is empty)

```

After merging,  $w_1$  is the second word containing larger  $k$  keys. We push  $w_1$  back to the list from which its largest key came from. This makes sure the list is still in sorted order. It is clear that this algorithm runs almost  $2n/k - 1$  iterations of Lemma 1. Thus, the merge can be done in  $O(n/k \cdot \log k)$  time.  $\square$

**Theorem 3.** *There is a priority queue with up to  $n$  keys, supporting operations INSERT and EXTRACTMIN of  $(w/\log n)$ -bit integers in  $O(\log \log n)$  time.*

**Proof:** Note that the complexity here is in terms of the size of universe. Later, we will further reduce this dependency to the current number of elements in the priority queue.

Consider INSERT. We have a sorted list  $B_0$  with at most  $\log n$  keys stored in a single word. If  $B_0$  is not full, by Lemma 1, we can merge a single new key into  $B_0$  in time  $O(\log \log n)$ . We also have buffers  $B_i$  ( $0 < i \leq \log n - \log \log n$ ) which may contain 0, 1 or 2 sorted lists with  $2^{i-1} \log n$  keys. Each key distributes over  $2^{i-1}$  words. So the maximum capacity of  $B_i$  is  $2^i$  words. The INSERT operation is organized in rounds of length  $\log n$  which is the time it needs to fill  $B_0$ . The algorithm is shown below. When the process starts, all  $B_i$  are empty and the global counter  $i = \log n - 1$ .

```

INSERT( $x$ ) :
  Insert  $x$  into  $B_0$ 
  if  $i = 0$ 
    Move the list of  $B_0$  to  $B_1$ 
  else if  $0 < i \leq \log n - \log \log n$  and  $B_i$  contains 2 lists
    Make one merge step on these two lists
    if the merging is thereby completed
      Move the resulting list to  $B_{i+1}$ 
    end
  end
end
 $i \leftarrow i - 1 \pmod{\log n}$ 

```

Note that no buffer is visited if  $i > \log n - \log \log n$ . The key point is that each round allows us to visit each buffer  $B_i$  exactly once. It is clear that the INSERT runs correctly in  $O(\log \log n)$  time.

Consider EXTRACTMIN. We can use a standard heap  $M$  with the smallest key of each  $B_i$ . So the number of items in the heap is no more than  $\log n$ , and operations on  $M$  can be done in  $O(\log \log n)$  time. When performing EXTRACTMIN, we use EXTRACTMIN operation of  $M$  to find the smallest key  $x$  and delete  $x$  from  $M$ . We also need to delete  $x$  from the list  $B_i$  which it came from. We then need to use INSERT of  $M$  to insert the new smallest key in  $B_i$  to  $M$ . These can all be done in  $O(\log \log n)$  time.

Note that INSERT also needs to properly maintain  $M$ . First, INSERT may change the smallest key of  $B_0$ . In this case, we need a deletion and an insertion on  $M$ . Second, when moving  $B_i$  to  $B_{i+1}$ , we need to delete the larger smallest keys of  $B_i$  and  $B_{i+1}$  from  $M$ . Since these operations can all be done in  $O(\log \log n)$  time, this does not affect the time bound of INSERT.  $\square$

**Theorem 4.** *There is a RAM priority queue supporting INSERT and EXTRACTMIN in  $O(\log \log n)$  worst case time, where  $n$  is the current number of keys in the queue.*

**Proof:** To break down arbitrary  $w$ -bit integers to short integers, we use the recursive range reduction technique which we introduce in van Emde Boas trees (see Section 1). This range reduction technique is basically to divide a  $b$ -bit integer into a higher  $b/2$ -bit integer and a lower  $b/2$ -bit integer. So if we recursively apply this range reduction for  $\log \log n$  times, we reduce our problem to dealing with  $(w/\log n)$ -bit integers. This range reduction requires  $O(\log \log n)$  time. By Theorem 3, these can also be done in  $O(\log \log n)$  time. Therefore, the total time is  $O(\log \log n)$  per operation.

To make the complexities depends on the current number of keys in the queue, we will work on a current queue  $Q$  with the current keys, and on two auxiliary queues  $Q^-$  and  $Q^+$ . The sizes of  $Q^-, Q, Q^+$  are  $2^i, 2^{i+1}, 2^{i+2}$  for some  $i$ . Also,  $Q = Q^- \cup Q^+$ . The system is started with  $Q$  half full. Consider any query  $q$  on  $Q$ . We perform the following procedure instead.

```

If  $Q$  is more than half full and  $Q^- \neq \emptyset$ 
   $x \leftarrow \text{EXTRACTMIN}(Q^-)$ 
   $\text{INSERT}(Q^+, x)$ 
  perform query  $q$ 
  if  $Q$  is full ( $Q^- = \emptyset$  and  $Q^+ = Q$ )
     $(Q^-, Q, Q^+) \leftarrow (Q, Q^+, \emptyset)$ 
  end
else if  $Q$  is less than half full and  $Q^+ \neq \emptyset$ 
   $x \leftarrow \text{EXTRACTMIN}(Q^+)$ 
   $\text{INSERT}(Q^-, x)$ 
   $\text{INSERT}(Q^-, x)$ 
  perform query  $q$ 
  if  $Q$  is 1/4 full ( $Q^- = Q$  and  $Q^+ = \emptyset$ )
     $(Q^-, Q, Q^+) \leftarrow (\emptyset, Q^-, Q)$ 
  end
end
end

```

Clearly, this standard doubling / halving does not change the complexities. This completes the proof.  $\square$

Thorup also notes that by tabulating the multiplicity, DELETE can be done  $O(\log \log n)$  amortized time.

This algorithm requires look-up tables with constant access time. With hash tables, the algorithm needs linear space  $O(n)$ .

### 3 Comparison between RAM Priority Queues

There are two other important word-based RAM data structures, namely Fredman and Willard's fusion trees [5, 6, 7] and Andersson's  $O(\sqrt{\log n})$  search trees [2]. Although both of them achieve the same time bounds, the fusion trees need constant time multiplication which is not a  $\text{AC}^0$  operation. Without this assumption, fusion trees may not be faster than van Emde Boas trees.

Table 1 summarizes the time bounds of priority queue operations with different algorithms. Among these operations, MAX and EXTRACTMAX can be implemented similarly to MIN and EXTRACTMIN with each data structure. However, Thorup's data structure cannot support SUCCESSOR and PREDECESSOR. The definitions of these two operations are as follows.

- $\text{SUCCESSOR}(S, x)$ : Return the smallest element in  $S$  larger than  $x$ .
- $\text{PREDECESSOR}(S, x)$ : Return the largest element in  $S$  smaller than  $x$ .

**Table 1.** Time bounds of different RAM priority queues

Operation	van Emde Boas [3, 4]	Fredman et al. [5, 6, 7]*	Andersson [2]	Thorup [8]
MIN	$O(1)$	$O(1)$	$O(1)$	$O(1)$
MAX	$O(1)$	$O(1)$	$O(1)$	$O(1)$
INSERT	$O(\log \log u)$	$O(\sqrt{\log n})$	$O(\sqrt{\log n})$	$O(\log \log n)$
DELETE	$O(\log \log u)$	$O(\sqrt{\log n})$	$O(\sqrt{\log n})$	$O(\log \log n)**$
EXTRACTMIN	$O(\log \log u)$	$O(\sqrt{\log n})$	$O(\sqrt{\log n})$	$O(\log \log n)$
EXTRACTMAX	$O(\log \log u)$	$O(\sqrt{\log n})$	$O(\sqrt{\log n})$	$O(\log \log n)$
SUCCESSOR	$O(\log \log u)$	$O(\sqrt{\log n})$	$O(\sqrt{\log n})$	N/A
PREDECESSOR	$O(\log \log u)$	$O(\sqrt{\log n})$	$O(\sqrt{\log n})$	N/A

\* Requires constant time multiplication

\*\* Amortized time

At the end, we show how vEB trees support SUCCESSOR operation. PREDECESSOR can be done similarly.

```

SUCCESSOR( $S, x$ ) :
  if  $x < \min(S)$ 
    return  $\min(S)$ 
  end
  ( $x_h, x_l$ )  $\leftarrow x$  (decompose)
  if  $\text{Hash}(x_h) \neq \text{NULL}$  and  $x_l < \max(\text{Hash}(x_h))$ 
     $i \leftarrow \text{Successor}(x_l, \text{Hash}(x_h))$ 
    return ( $x_h, i$ )
  else
     $j \leftarrow \text{Successor}(x_h, H)$ 
    return ( $j, \min(\text{Hash}(j))$ )
  end

```

Since SUCCESSOR has only one recursive call and other operations can be done in constant time, the time bound is given by the recurrence  $T(u) = T(\sqrt{u}) + O(1)$ . Therefore, vEB trees support SUCCESSOR in  $O(\log \log u)$  time.

## References

- [1] S. Albers and T. Hagerup. Improved parallel integer sorting without concurrent writing. *3rd ACM-SIAM Symposium on Discrete Algorithms*, pp. 463–472. 1992.
- [2] A. Andersson. Sublogarithmic searching without multiplications. *36th Annual Symposium on Foundations of Computer Science*, pp. 655–663. 1995.
- [3] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters* 6:80–82, 1977.
- [4] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory* 10:99–127, 1977.
- [5] F. W. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences* 47(3):424–436, 1993.
- [6] F. W. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences* 48(3):533–551, 1994.
- [7] M. L. Fredman and D. E. Willard. BLASTING through the information theoretic barrier with FUSION TREES. *22nd Annual ACM Symposium on Theory of Computing*, pp. 1–7. 1990.
- [8] M. Thorup. On RAM priority queues. *7th ACM-SIAM Symposium on Discrete Algorithms*, pp. 59–67. 1996.
- [9] D. E. Willard. Log-logarithmic worst-case range queries are possible in space  $\theta(n)$ . *Information Processing Letters* 17(2):81–84, 1983.