University of Ljubljana
Institute of Mathematics, Physics and Mechanics
Department of Mathematics
Jadranska 19, 1000 Ljubljana, Slovenia

Preprint series, Vol. 38 (2000), 717

WORST CASE CONSTANT TIME PRIORITY QUEUE

Andrej Brodnik Svante Carlsson
Johan Karlsson J. Ian Munro

ISSN 1318-4865

August 28, 2000

Ljubljana, August 28, 2000

Worst Case Constant Time Priority Queue

Andrej Brodnik [*] [†] Svante Carlsson [†] Johan Karlsson [†]
J. Ian Munro [‡]

Abstract

We present a new data structure of size $3M + o(M)$ bits for solving the *"discrete priority queue" problem.* When this data structure is used in combination with a new memory topology it provides an O(1) worst case time solution.

1 Introduction

In this paper we reexamine the well known "discrete priority queue" problem of van Emde Boas et al [12]. Operating over the bounded universe of integers $\mathcal{M} = [0, .., M-1]$, the usual operations of *insert* and *extractmin* are supported, as are the additional operations of extracting any value and finding $Predecessor(e)$ and $Successor(e)$. These operations determine, respectively, the largest element present that is less than e, and the smallest greater than e. The problem is referred to by Mehlhorn et al. [7] as the union-split-find problem. Under this terminology, one thinks of $[0, .., M-1]$ as being partitioned into subranges that can be further subdivided or merged, and that one can ask for the subrange containing a given value. More formally we define the data type as:

Definition 1 *The* discrete extended priority queue problem *is to maintain a set, $\mathcal{N}$ of size N with elements drawn from an ordered bounded universe $\mathcal{M} = [0..M-1]$, and support the following operations:*

[*]Department of Theoretical Computer Science, Institute of Mathematics, Physics, and Mechanics, Ljubljana, Slovenia

[†]Department of Computer Science and Electrical Engineering, Luleå University of Technology, Luleå, Sweden, (`Andrej.Brodnik`, `Svante.Carlsson`, `Johan.Karlsson@SM.LUTh.SE`)

[‡]Department of Computer Science, University of Waterloo, Ontario, Canada, (`IMunro@daisy.UWaterloo.CA`)

`Insert(`e`)`	: $\mathcal{N} := \mathcal{N} \cup \{e\}$
`Delete(`e`)`	: $\mathcal{N} := \mathcal{N} \backslash \{e\}$
`Member(`e`)`	: *Find whether* $e \in \mathcal{N}$
`Min`	: *Compute the smallest element of* $\mathcal{N}$
`Max`	: *Compute the largest element of* $\mathcal{N}$
`DeleteMin`	: *Delete the smallest element of* $\mathcal{N}$
`DeleteMax`	: *Delete the largest element of* $\mathcal{N}$
`Predecessor(`e`)`	: *Compute the largest element of* $\mathcal{N} < e$
`Successor(`e`)`	: *Compute the smallest element of* $\mathcal{N} > e$

We will refer to an element e's predecessor as its *left neighbour* and its successor as its *right neighbour*. When talking about the *neighbours* of e we mean the left and the right neighbour. We let m denote $\lg M$.

Lower Bounds and some Matching Upper Bounds

Under the pointer machine model (cf. [10]) Mehlhorn et al. proved an amortized lower bound of $\Omega(\lg\lg M)$ [7] for our problem. Recently, Beame and Fich [2] gave a lower bound of $\Omega(\sqrt{\lg N / \lg\lg N})$ under the communication game model (cf. [8,14]) which also apply to the cell probe (cf. [15]) and RAM (cf. [11]) models. They also gave a matching upper bound for the static version of our problem. Andersson and Thorup [1] gave a data structure and an algorithm with a matching worst case time for the dynamic version.

Our Model of Computation

Our model is based on the RAM model of computation which includes branching and the arithmetic operations addition and subtraction. We will also need bitwise Boolean operations and multiplication/division (cf. MBRAM [11]). However, we do not want the model to be unrealistic and therefore we restrict the model to only use bounded registers. The registers we use are at least m bits wide; i.e. a memory locations can store at least m-bit values and all operations are defined for arguments with at least m bits. This model of computation is implemented by any standard computer today.

Operations to search for *Least (Most) Significant Bit (LSB, MSB)* can be implemented to run in $O(1)$ time in our model, using a technique called *Word-Size-Parallelism* [3]. Hence, we let these operations be defined in the model as well.

A final, and crucial, aspect of our model of computation, is the notion of a word of memory. Under the standard model, a word is a sequence of bits and each bit is in only one word. We will consider a model in which a single bit may be in several different words. The notion of a "random access machine with byte overlap" (*RAMBO*).

2 Stratified Trees

The basis of the $\lg\lg M$ solution of van Emde Boas et al. [12] is a structure they called a *stratified tree*. Indeed, the stratified tree is often referred to informally as the "van Emde Boas" structure. We stick with the name originally given.

Definition 2 *A* stratified tree *is a complete binary tree on $\mathcal{M}$ in which:*

- *Each leaf representing an element of $\mathcal{N}$ is tagged and contains pointers to its predecessor and successor;*
- *Nodes can be active (tagged) and if so they contain:*
  - *pointers to the smallest and largest active node or leaf in each subtree.*
  - *an indication if there exist branching nodes on the path from the node to the root or not.*

A simple boolean flag is used to tag a node. To find neighbours and insert/delete elements in a stratified tree, a binary search is performed on the path from the leaf to the root to find the proper internal node or leaf (see [12] for details). This yields a $O(\lg\lg M)$ worst case time for all the discrete extended priority queue operations, which matches the lower bound of Mehlhorn et al.

We cannot lower the worst case time to constant time for the operations in any of the above models of computation. Therefore, we appeal to a new, but implementable model.

3 The Split Tagged Tree

In a complete binary tree that has leaves for every element in a universe $\mathcal{M}$ (cf. *trie*) we define:

Definition 3 *An internal node as a* splitting node *if there is at least one tagged leaf in each of its subtrees.*

The splitting node ν as a left (right) splitting node *of e if e is a leaf in the left (right) subtree of ν. The first left (right) splitting node on a path from e to the root is the* lowest left (right) splitting node *of e.*

The splitting nodes are the only internal nodes of the *Split Tagged Tree* (*STT*) that store additional information. In detail:

Definition 4 *A* Split Tagged Tree *is a complete binary tree on $\mathcal{M}$ in which:*

- *Each leaf representing an element of $\mathcal{N}$ is tagged;*
- *A splitting node is tagged and has pointers to the leaf representing the largest (smallest) element x (y) in its left (right) subtree, where $x \in \mathcal{N}$ ($y \in \mathcal{N}$); and*

- *There is a special supernode, placed on top of the tree. It is tagged if the set contains at least one element. If the supernode is tagged it contains pointers to the leaves representing the minimum and maximum elements.*

The supernode is both a left and a right splitting node for all the elements in the set. Hence, all elements, even minimum and maximum, have both left and right splitting nodes. This somehow simplifies the operations on the STT because the supernode has a role similar to the sentinel in linked lists.

Since leaves represent elements of $\mathcal{M}$, we will refer to the leaves and the corresponding elements interchangeably.

The leaves only need a tag and hence are stored in a boolean array `leaves` of `STT` in Algorithm 1. Therefore the pointers to the leaves are simple indices into the array. On the other hand, the internal nodes and the supernode are represented by the structure `STT_node`. They are stored in standard heap order in an array `nodes` of `STT` where the supernode is stored at the location 0. The fields `left` and `right` of the supernode point at the maximum and the minimum elements respectively.

```
typedef struct {
  boolean tag;                                        /* Tag */
  int left;                   /* Largest element in left subtree */
  int right;                 /* Smallest element in right subtree */
} STT_node;
typedef struct {
  STT_node nodes[M];                    /* Array of internal nodes */
  boolean leaves[M];                          /* Array of leaf tags */
} STT;
```

Algorithm 1: An efficient representation of the STT.

A key feature of the stratified tree is that nodes are tagged in a manner such that the lowest ancestor of a tagged leaf can be found by a binary search. This property does not need to hold for the STT as we will use a novel memory architecture to permit constant time search and still find the analogous nodes. The STT does, however, have a number of crucial properties.

Lemma 1 *Let $e \in \mathcal{N}$ and let ν_l be its lowest left (right) splitting node. Then, ν_l is the only of e's left (right) splitting nodes that points at e.*

Proof: The proof is given in terms of the left splitting node. From Definition 4 we know that an element f is pointed at by a splitting node if f is the largest element in the left subtree of the node.

Assume that an element $z \in \mathcal{N}$ exists such that $z \neq e$ is the largest element in the left subtree of ν_l. Since both e and z are in the left subtree of ν_l, the lowest common ancestor ν of e and z, is also in the left subtree of ν_l.

By Definition 3 ν is a splitting node. Moreover, ν is a left splitting node of e since e is in the left subtree of ν which contradicts our initial assumption that ν_l is e's lowest left splitting node.

Finally, by a counting argument and the first part of this lemma we see that if the element e is pointed at by two left splitting nodes there exists an element $z \in \mathcal{N}$, such that z is not pointed at, which contradicts the first part of this lemma. $\mathcal{QED}$

Lemma 1 implies that in the STT there is a constant number of nodes that have pointers to an element $e \in \mathcal{N}$. Moreover, the nodes containing the pointers in question are the lowest left and lowest right splitting nodes of e. We proceed by showing how to find the neighbours of e:

Lemma 2 *Let ν_l and ν_r be e's lowest left and lowest right splitting nodes respectively, and let $x \in \mathcal{N}$ be the left (right) neighbour of e, hence $x < e$ $(e < x)$. Then, if $e \in \mathcal{N}$ a pointer at ν_r (ν_l) refer to x; and if $e \notin \mathcal{N}$ then either a pointer in ν_l or in ν_r points to x.*

Proof: Again we focus on the left neighbour. If $e \in \mathcal{N}$, this lemma follows from Definitions 3 and 4 and Lemma 1. If $e \notin \mathcal{N}$, let $y \in \mathcal{N}$ be e's right neighbour, hence $x < e < y$, and let ν be the lowest common ancestor of x and y. By definition, ν is also a splitting node, and since $x < e < y$ it is a splitting node on the path from e to the root. Since x and y are neighbours in $\mathcal{N}$, x is the largest element in the left subtree of ν and y is the smallest element in the right subtree of ν. Consequently, ν has a pointer to x, which is e's left neighbour. To prove that ν is either ν_l or ν_r assume ν is a left splitting node of e but not ν_l. Then by Lemma 1 we know that ν has to be below ν_l since ν has a pointer to x, which contradict our initial assumption that ν_l is the lowest left splitting node of e. If ν is a right splitting node of e the proof is symmetrical. $\mathcal{QED}$

Finally, we show that at the insertion of an element e, it is sufficient to find either the lowest left or the lowest right splitting node of e, to decide where the new splitting node shall be (see Figure 1). We state the lemma in terms of the left splitting node:

Lemma 3 *Let ν_l be the lowest left splitting node of $e \notin \mathcal{N}$ and let $z \in \mathcal{N}$ be the largest element in the left subtree of ν_l. If $z < e$ then no element $v \in \mathcal{N}$ exists in the right subtree of the lowest common ancestor ν of z and e and; if $e < z$ then no element $v \in \mathcal{N}$ exists in the left subtree of the lowest common ancestor ν of e and z.*

Proof: If $z < e$, assume an element $v \in \mathcal{N}$, such that v is in the right subtree of ν, exists, hence $z < v$. However, since all three elements are in the left subtree of ν_l this contradicts the assumption that z is the largest element $\in \mathcal{N}$ in the subtree. If $e < z$, assume an element $v \in \mathcal{N}$ exists in the left subtree of ν, then by Definition 3 is ν a splitting node. Since all of e, z, v are in the left subtree of ν_l, ν has to be in the left subtree of ν_l.

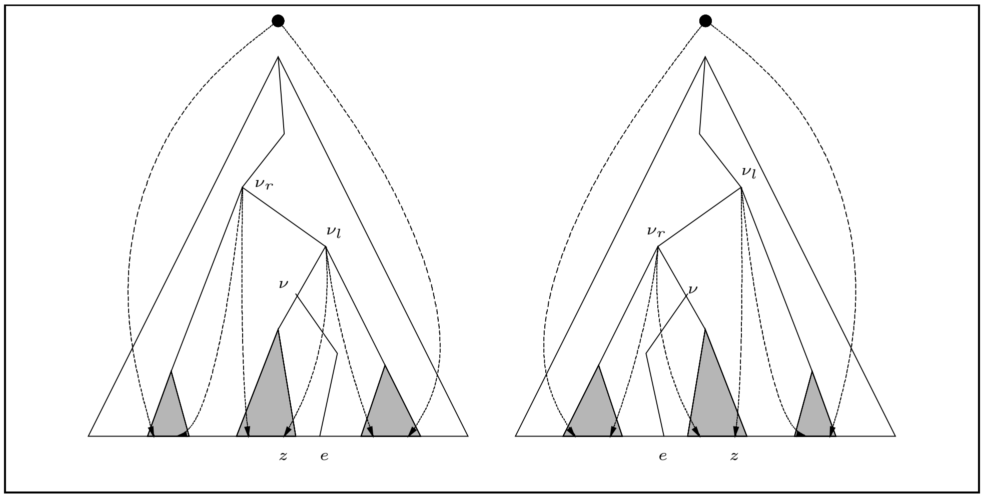

Figure 1: The two scenarios before insertion of e.

Hence ν is lower than ν_l and ν is a left splitting node of e contradicting the assumption about ν_l. $\mathcal{QED}$

We proceed to describe how to answer queries and perform updates in the STT. Queries about neighbours can be answered using Lemma 2. By a counting argument at an *insertion* of element $e \notin \mathcal{N}$, exactly one internal node becomes a splitting node and one leaf gets tagged. The leaf is the leaf corresponding to e. To find the new splitting node we get the element z pointed at by the lowest left splitting node ν_l of e. If $z < e$ then, by Lemma 3, the lowest common ancestor ν of e and z should be e's new lowest right splitting node with one pointer to z and one to e, and ν_l should be updated to point at e instead of z. If $e < z$ then, by Lemma 3, the lowest common ancestor ν of e and z should be e's new lowest left splitting node with one pointer to e and one to e's right neighbour found in e's lowest right splitting node ν_r. ν_r should be updated to point at e instead of e's right neighbour.

By a similar reasoning we see that *deletion* is done by removing the tags in the leaf and in the lower of the left and right lowest splitting nodes, and by updating the pointers in the other node.

Since only a constant number of nodes need to be updated, the time to update (and also to search) the STT is asymptotically bounded by the time to traverse the tree. The tree is of height $\lg M + 1$, which implies that the time to update and search the tree is $O(\lg M)$ in our and pointer machine models.

Yggdrasil implementation of RAMBO

We have worked our way from the stratified tree, in which the lowest tagged node can be found in $O(\lg \lg M)$ time, to a perhaps simpler situation that would appear to require a sequential scan of the nodes up a path. At this point we resort to a change in a model and consequently an improvement in the hardware to achieve a constant time solution.

The RAMBO model of computation is a RAM model in which, in one part of the memory, registers can share bits; i.e. bytes overlap (cf. RAMBO introduced by Fredman et al. [6] and further described by Brodnik [4]). One particular implementation of the RAMBO called *Yggdrasil* can help us to solve our problem (see [9]).

In the Yggdrasil memory layout, registers overlap as paths from root to leaf in a complete binary tree (see Figure 2). In particular, we think of the bits $B_{k(k=1,..,M-1)}$ as being enumerated in standard heap order. (The root is B_1, the children of B_i are B_{2i} and B_{2i+1}, and the leaves are bits $B_{M/2}$ through B_{M-1}.) The most significant bit of any register is the root bit, B_1. (By convention, we call this bit $\lg M - 1$, and so, bit 0 is the least significant.) The bits of register i correspond to those along the path from the root to leaf i (i.e. bit $M/2 + i$). This means:

$$\begin{aligned} reg[i].bit[j] &= B_k \qquad \text{where} \\ k &= (i \; div \; 2^j) + 2^{m-j-1} \qquad (1) \end{aligned}$$

We now store the tags of the STT in the Yggdrasil memory `reg` (see

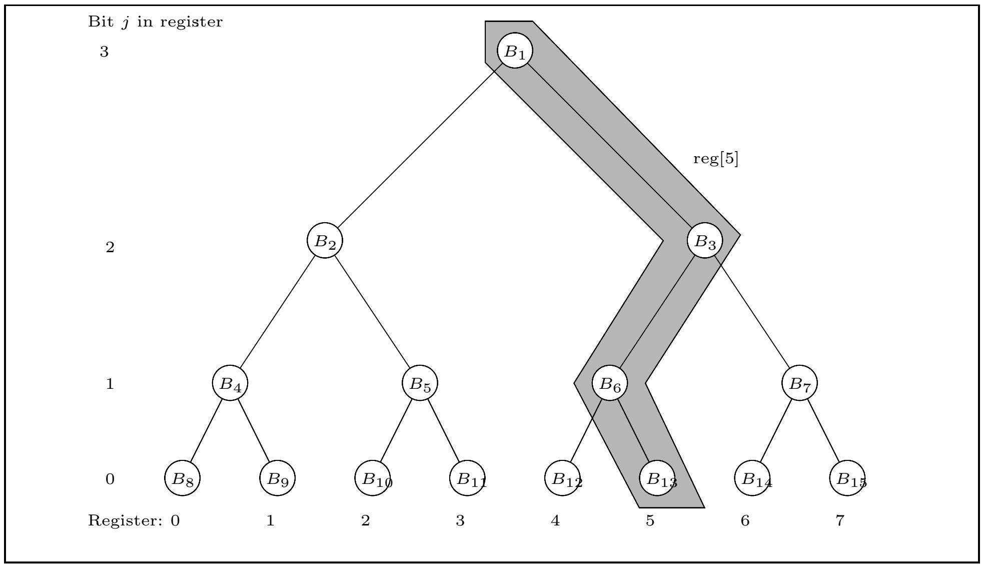

Figure 2: Overlapped memory *Yggdrasil*.

Algorithm 2), the tree of internal pointers in an array `nodes`, while the tags of the leaves are stored in the boolean array `leaves`.

To find the lowest splitting node ν_k of the element e we read the register `reg[e div 2]`, find the least significant set bit j and compute k using Equation 1. Since the path from the root to e can be deduced from the binary representation of e we can use in the computation e or $\overline{e}$ to mask the register value and get in constant time the lowest right or left splitting node respectively.

```
typedef struct {
  int left;                    /* Largest element in left subtree */
  int right;                 /* Smallest element in right subtree */
} STT_node;
typedef struct {
  Yggdrasil reg[M/2];          /* Register with byte overlapping */
  STT_node nodes[M];               /* Array of internal nodes */
  boolean leaves[M];                    /* Array of leaf tags */
} STT;
```

Algorithm 2: Representation of the STT with Yggdrasil memory

The lowest common ancestor ν_k of the elements e and f is the last common node on the paths from the root to e and f respectively. Hence, the node ν_k corresponds to the least significant common bit j in the binary representations of e and f. Let $i = e$ div 2 then Equation 1 gives

$$\begin{aligned} k &= ((e \textit{ div } 2) \textit{ div } 2^j) + 2^{m-j-1} \\ &= (e \textit{ div } 2^{j+1}) + 2^{m-j-1} \end{aligned} \tag{2}$$

The least significant common bit j is next to, toward the top, the most significant non common bit of e and f, i.e.:

$$j = \mathrm{MSB}(e \,\mathrm{XOR}\, f) + 1 \tag{3}$$

Now we can find both the lowest splitting nodes of an element and the lowest common ancestor of two elements in constant time. Following pointers and updating those and the tags can also be done in constant time. Hence, given Yggdrasil memory, updates and queries can be performed in constant time. The Yggdrasil memory has been developed in hardware by *Priqueue AB*, as a SDRAM memory module according to the PC100 standard [5].

The STT contains quite some redundant information – the leaves can be removed since the leaf is tagged if and only if it is pointed at by an internal node. Next, the information in the internal nodes can be stored using pointers of variable length. This reduces the space requirements from $2M \lg M + M + O(\lg M)$ bits to $2M + O(\lg M)$ bits of conventional memory for the discrete extended priority queue problem. However, reducing the size increases the constant for the time. We can further reduce the space by using perfect hashing to store tagged internal nodes similarly to Willard's approach for stratified trees [13]. This gives a space requirement of $O(N \lg M)$ bits of ordinary memory and M bits of Yggdrasil memory.

The above discussion leads to our main result:

Theorem 1 *Using the* Split-Tagged-Tree *together with the* Yggdrasil *memory we can solve the discrete extended priority queue problem in $O(1)$ worst case time per operation using only $2M + O(\lg M)$ bits of ordinary memory* *and M bits of Yggdrasil memory. This can be reduced to $O(N \lg M)$ bits of ordinary memory and M bits of Yggdrasil memory but the time is then degraded to $O(1)$ per operation with a high probability.*

To solve our problem with support for either predecessor or successor, but not both, we can simply remove the pointer to one or the other element in the nodes. This saves half of the ordinary memory and it reduces the constant for updates times.

References

[1] A. Andersson and M. Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. Manuscript to apper in STOC2000.

[2] P. Beame and F. E. Fich. Optimal bounds for the predecessor problem. In *Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing*, pages 295–304, New York, May 1–4 1999. ACM Press.

[3] A. Brodnik. Computation of the least significant set bit. In *Proceedings Electrotechnical and Computer Science Conference*, volume B, pages 7–10, Portorož, Slovenia, 1993.

[4] A. Brodnik. *Searching in Constant Time and Minimum Space (*MINIMÆ RES MAGNI MOMENTI SUNT*)*. PhD thesis, University of Waterloo, Waterloo, Ontario, Canada, 1995. (Also published as technical report CS-95-41.).

[5] A. Brodnik, J. Karlsson, R. Leben, M. Miletić, M. Špegel, and A. Trost. Design of high performance memory module on PC100. In *Proceedings Electrotechnical and Computer Science Conference*, Slovenia, 1999.

[6] M. L. Fredman and M. E. Saks. The cell probe complexity of dynamic data structures. In *21st ACM Symposium on Theory of Computing*, pages 345–354, 1989.

[7] K. Mehlhorn, S. Näher, and H. Alt. A lower bound on the complexity of the union-split-find problem. *SIAM Journal on Computing*, 17(6):1093–1102, 1988.

[8] P. B. Miltersen. Lower bounds for Union-Split-Find related problems on random access machines. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on the Theory of Computing*, pages 625–634, Montréal, Québec, Canada, 23–25 May 1994.

[9] Encyclopedia Mythica. Yggdrasil. `http://www.pantheon.org/mythica/articles/y/yggdrasil.html`.

[10] A. Schönhage. Storage modifications machines. *SIAM Journal on Computing*, 9(3):490–508, August 1980.

[11] P. van Emde Boas. Machine models and simulations. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, pages 3–66. Elsevier/MIT Press, Amsterdam, 1990.

[12] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[13] D. E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17(2):81–84, 24 August 1983.

[14] A. Yao. Some complexity questions related to distributive computing. In *Proceedings of the 11th Annual ACM Symposium on Theory of Computing*, pages 209–213, 1979.

[15] A. Yao. Should tables be sorted? *Journal of the ACM*, 28(3):614–628, July 1981.