Queaps

John Iacono[1] and Stefan Langerman[2⋆]

[1] Dept. of Computer and Information Science, Polytechnic University
jiacono@poly.edu
[2] School of Computer Science, McGill University
sl@cgm.cs.mcgill.ca

Abstract. We present a new priority queue data structure, the *queap*, that executes insertion in $O(1)$ amortized time and extract-min in $O(\log(k+2))$ amortized time if there are k items that have been in the heap longer than the item to be extracted. Thus if the operations on the queap are first-in first-out, as on a queue, each operation will execute in constant time. This idea of trying to make operations on the least recently accessed items fast, which we call the queueish property, is a natural complement to the working set property of certain data structures, such as splay trees and pairing heaps, where operations on the most recently accessed data execute quickly. However, we show that the queueish property is in some sense more difficult than the working set property by demonstrating that it is impossible to create a queueish binary search tree, but that many search data structures can be made almost queueish with a $O(\log \log n)$ amortized extra cost per operation.

1 Introduction

Self adjusting data structures are those structures that use a simple restructuring heuristic to implement their operations, and forgo any use of explicit balance information. Examples of self adjusting structures include the splay trees[14], pairing heaps[6] and skew heaps[15]. These structures share many of the per operation runtimes as the corresponding best comparison based structures in the amortized sense, while using less memory and being significantly easier to code. For example, splay trees execute k insert, delete, and search operations on a tree of size at most n in time $O(k \log n)$, which is the same as for AVL trees[1], or any of the other balanced binary search trees.

However, self adjusting data structures do more than just match the existing performance of their non-self adjusting counterparts; they are able to execute many seemingly natural classes of searches in time much faster than in $O(\log n)$ time. Currently, there exists no complete analysis of splay trees, instead their runtime is characterized by several different theorems, none of which imply all of the others: The static optimality theorem, the static finger theorem, the working set theorem (all from [14]), the sequential access theorem [16, 17], the dynamic finger theorem [2, 3] and the key-independent optimality theorem [11]. Each of

⋆ Research is supported by grants from MITACS, FCAR and CRM.

these theorems targets a seemingly natural class of sequences of searches, and shows how these desirable classes of sequences run in $o(\log n)$ amortized time per operation. These theorems have been joined by a number of conjectures about the runtime of splay trees: most notably the dynamic optimality conjecture [14] and the unified conjecture [10].

The splay tree theorems and conjectures, although intended to merely bound the runtime of splay trees, can be viewed as characterizations of distribution sensitive behavior that can be used in a much broader context. These properties thus deserve study in their own right, as studying these properties gives us insight into the possibilities and limitations of classic comparison based data structures. Recent work has established the relationships amongst the known properties, thus establishing a hierarchy of properties [10]. New data structures that are, in some ways, advantageous to splay trees have been engineered from several of the properties. Still other work has applied the properties, which were originally formulated for use in analyzing dictionaries, to heaps [9], planar point location [7] and point searching [4].

The properties that splay trees have, and are conjectured to have, show that splay trees execute many "natural sequences" quickly. In this paper we ask the intentionally vague question "Do splay trees execute all natural sequences quickly?" The answer is no, as we show that there is a class of sequences of operations that can be executed on a queue in constant time per operation that requires $\Omega(\log n)$ to execute on splay trees, and, in fact any binary search tree.

We introduce a new property, which we call the *queueish* property. As this property is a variant of the working set property of splay trees, we present the working set property first: The working set property of splay trees states that the time to search item x is $O(\lg w(x))$[1], where $w(x)$ is the number of distinct items that have been accessed (an access is a search or an insert) since x's last access. Informally, in a data structure with the working set property, accesses to items recently accessed are faster than accesses to items that have not been accessed in a while. The queueish property states the complementary idea that in any structure with this property, an access to x is fast if x is one of the least recently accessed items. Formally, a data structure is said to be queueish if the time to search item x is $O(\lg q(x))$, where $q(x) = n - 1 - w(x)$ is the number of distinct items that have *not* been accessed since x's last access, and n is the number of elements in the data structure. Any structure with the queueish property can execute any repeated sequence of n distinct search operations in constant time per operation. The queueish property is seemingly as natural as the working set property, yet we prove in section 5 that splay trees, and in fact any binary search tree structure cannot have the queueish property. More precisely, we show that an amortized lower bound of $\Omega(\log n)$ per access on a binary search tree still holds even when $q(x) = 0$ for every element accessed. We do however provide a general transformation in Section 4 that turns many data structures, including any standard dynamic dictionary and a point location data structure, into one with the queueish property at a cost of an additional $O(\lg \lg n)$ time per

[1] To ease the notation, we denote $\lg x = \log_2 x$ for $x \geq 2$, and $\lg 0 = \lg 1 = 1$.

operation. This transformation produces a data structure which is not a binary search tree, and so does not contradict our previous statement.

We also present a positive result for priority queues. We say that a priority queue has the queueish property if the amortized time to insert is $O(1)$ and the amortized time to extract-min x is $O(\lg q(x))$ where $q(x)$ is the number of items that have been in the priority queue longer than x. A similar definition for the working set property can be made. It has been shown that pairing heaps, a form of self-adjusting heap, have the working set property [9]. In section 2, we describe a priority queue data structure with the queueish property, which we call the queap. Queaps might prove useful in some situations. For example if a queap is used for sorting an array, the running time will be $O(n \lg(I/n))$ where I is the number of inversions in the array to sort. More generally, queaps could be advantageous for implementing discrete event simulations such as *future event set* algorithms. In such algorithms, one wishes to simulate some complex system, such as the replacement of broken light bulbs in a building. The algorithm maintains a list of future events (e.g. the breaking time of every currently used light bulb), and at every step extracts the next occurring event in the list (e.g. the first light bulb to break), processes the event (replace the light bulb) and inserts some eventual new events (the time at which the newly replaced light bulb breaks), according to some probability distribution. If the distribution is such that the new events are inserted within the k last occurring events in the list (deterministically or on average), then finding the next event will only require $O(\lg k)$ amortized time (deterministically or on average) for a queap, while insertions will take worst case and amortized constant time. For a survey on discrete event simulation, see, for example, [5, Chapter XIV.5].

The paper proceeds as follows: Section 2 provides a description of queaps, and Section 3 provides the analysis of queaps. A general transformation that turns many data structures into a queueish one with an additional cost of $O(\lg \lg n)$ per operation is found in Section 4. Section 5 provides the proof that no standard binary search tree structure can be queueish.

2 Queaps

Our data structure supports the operations:

$New(Q)$**:** Creates a new empty queap.
$Insert(Q, x)$**:** inserts element x into Q.
$Minimum(Q)$**:** returns a pointer to the element in Q whose key is minimum.
$Delete(Q, x)$**:** deletes element x from Q.
$Delete\_Min(Q)$**:** returns a pointer to the element in Q whose key is minimum, and removes that element from Q.

With both delete operations running in amortized $O(\lg q(x))$ time for an element x and all other operations running in worst case (and amortized) constant time.

Consider the list $x_1, \ldots, x_n$ of elements currently in the data structure, in the order in which they were inserted. These elements will be split into two groups: The *old* set $x_1, \ldots, x_k$ and the *new* set $x_{k+1}, \ldots, x_n$ for some k, $1 \leq k \leq n$.

The old set will be stored in the leaves of a $(2,4)$-tree T, from left to right in the order $x_1, \ldots, x_k$, preceded by a dummy leaf x_0 with infinite key value. A $(2,4)$-tree is a balanced tree structure in which all internal nodes have degree between 2 and 4, and all leaves have same depth [8, 13]. In such trees, insertions and deletions of leaves can be performed in amortized constant time per operation. Note that we are using the structural properties of $(2,4)$-trees, and we are not using them as binary search trees. Each internal node v of T contains h_v, a pointer to the leaf with the minimum key value in its subtree (denoted T_v). Furthermore, with each node v on the path from the root to the leftmost leaf will be stored the pointer c_v to the leaf with the smallest key in $T - T_v$. For these nodes, the value h_v will not be maintained. We also keep a pointer to the leftmost leaf x_0 in T. Note that c_{x_0} contains a pointer to the leaf with the minimum key value in T.

The new keys $x_{k+1}, \ldots, x_n$ will be stored in a linked list L in that order, and a pointer to the element with minimum key value in L will be kept in a variable $minL$. Here is a description of the operations:

$New(Q)$**:** Create a new empty tree T and list L. $k \leftarrow 0$, $n \leftarrow 0$.
$Insert(Q, x)$**:** Insert x to the right of L, update $minL$. $n \leftarrow n + 1$.
$Minimum(Q)$**:** If $key(minL) < key(c_{x_0})$, return $minL$ else return c_{x_0}.
$Delete(Q, x)$**:** If $x \in L$, insert all the elements of L in T and L becomes empty, update h_v for all nodes v whose children are new or have been modified, up to the root of T. Walk down from the root to x_0, updating the c_v values. $k \leftarrow n$.
Delete x from T, walk in T on the path from x to x_0, correcting h_v values, and then c_v values. $n \leftarrow n - 1$, $k \leftarrow k - 1$.
$Delete\_Min(Q)$**:** $min \leftarrow Minimum(Q)$. $Delete(Q, min)$. Return min.

3 Running time analysis

We will analyze the amortized cost of the operations using the potential method. The potential function for a queap $Q = (T, L)$ will be $\phi(Q) = c|L|$ for some constant c to be specified later. If the actual cost of an operation is t, the amortized cost for that operation is defined to be $\tilde{t} = t + \phi(Q') - \phi(Q)$ where Q and Q' correspond to the data structure before and after the operation, respectively. Since $\phi(Q) \geq 0$ for any Q, and is zero for an empty data structure Q, we know that the total amortized cost of any sequence of operations is an upper bound on the actual cost of that sequence.

Note that we are using $(2,4)$-trees, a data structure whose cost is already amortized, as a substructure. In the following analysis, we will use the amortized costs for insertions and deletions on those trees. This would be equivalent to using the actual costs of these operations and adding the potential function of the $(2,4)$-trees to our $\phi(Q)$. We now describe the amortized cost for each operation.

Insert(Q,x)**:** The actual cost of this operation is $O(1)$. Since the size of the list L grows by one, the potential increases by c, which is a constant, and so the amortized cost of this operation is $O(1)+c=O(1)$.

Minimum(Q)**:** This operation doesn't modify the data structure, so its amortized cost is equal to its actual cost, $O(1)$.

Delete(Q,x)**:** If the deleted leaf is in T, then the cost of the delete operation in the tree is $O(1)$. We then still have to correct the h_v and c_v pointers on the path from x to x_0. Let r be the highest node on that path. If r is k levels above the leaves, then the subtree of the left child of r contains at least 2^{k-1} leaves, all of which have been inserted before x. Thus $q(x) > 2^{k-1}$ and $k = O(\lg q(x))$ and so is the cost of this operation. Since this doesn't modify the potential function, the amortized cost is also $O(\lg q(x))$.

If $x \in L$, we first need to insert all the elements of L in the tree T. This has a cost of $a|L|$ for some constant a (amortized over $(2,4)$-tree operations). Once these insertions are done, the update of the h_v and c_v pointers for all nodes that are new or have been structurally modified is certainly no more than the very cost of the insertions, and so the total time spent so far is bounded by $2a|L|$.

At this point, if the root of T has not been modified, we still need to continue correcting h_v values for nodes v whose children's h values have been modified, up to the root of T. This costs at most $O(\lg|T|)$. We now walk from the root of T down to x_0, correcting c_v values, for a cost of $O(\lg|T|)$. The total spent so far is now no more than $2a|L| + O(\lg|T|)$, but note that if x was in L, then all elements in T have been inserted before x and $q(x) > |T|$. So we have spent $2a|L| + O(\lg q(x))$ and have caused a drop in potential of $c|L|$ since all elements of the list have been removed from it. So if $c > 2a$, the amortized cost for merging T and L is $O(\lg q(x))$. We still have to delete x from the updated tree, but this can be done in $O(\lg q(x))$ as seen above.

Delete_Min(Q)**:** The amortized cost of this operation is just the sum of the costs of the operations it uses, i.e. $O(\lg q(x))$.

4 Weakly queueish data structures

We here present a simple scheme to transform data structures for searching problems into an amortized queuish equivalent with an additive $O(\log\log n)$ access cost. A *searching problem* consists of maintaining a set A of objects with respect to some binary relation $\Box[x,a]$. A *query* $Q(x,A)$ will return some object $a \in A$ for which $\Box[x,a]$ is true, if such an object exists. For ease of notation we will write $q(x)$ for $q(a)$ if a is the object returned by $Q(x,A)$. It is assumed that the $\Box$ operator is computable in $O(1)$ time.

Theorem 1. *Let $D(A)$ be a data structure for solving a searching problem with query operation Q on a set A with n elements. If $D(B')$ for any $B' \subset B$, $|B| = O(|B'|^2)$ can be constructed in $O(|B'|)$ time given $D(B)$, and supports $O(\lg |B'|)$ query times, then $D(A)$ can be transformed into a data structure supporting $O(\lg q(x) + \lg \lg n)$ amortized query times.*

Proof. The data structure consists of a series of data structures $D_1, D_2, ..., D_k$ and a series of queues $Q_1, Q_2, ..., Q_k$. The concatenation of all these queues contains all the elements in increasing order of the last access times on these elements, i.e. Q_1 contains the least recently accessed items and Q_k contains the most recently accessed items. The queues are implemented as doubly linked lists, and we will refer to the most recently accessed items in Q_i as being in the *head* of Q_i, and to its least recently accessed items as being in the *tail* of Q_i.

All elements of $Q_1 \cup Q_2 \cup \ldots \cup Q_i$ are present in D_i but D_i might contain some extra elements. Pointers are maintained between each element in D_i and its corresponding element in one of the queues even if that queue is not Q_i. Every element in Q_i contains the number i of the queue it is in. For $i < k$, $2^{(2^i - 1)} < |Q_i| \leq 2^{(2^i)}$. D_k will contain all the elements present in the whole data structure, and $k = \lfloor \lg \lg n \rfloor$. This will ensure that $|Q_k| > 2^{(2^k - 1)}$. Let $m_i = 2^{(2^i)} - |Q_i|$. For this data structure we will use the potential function

$$\phi = c \sum_{i=1}^{k-1} m_i (\lfloor \lg \lg n \rfloor - i)$$

At creation, $|Q_i| = 2^{(2^i)}$ for all $i < k$, and so $\phi = 0$. Every time an element x is looked up, we query it sequentially in $D_1, D_2, \ldots$ until it is found in D_i. We then know the queue Q_j that contains x and $j \geq i$. Remove x from Q_j and insert it in the head of Q_k. The actual cost of this operation is $O(2^i) = O(\lg q(x))$, and the increase in potential is $c(\lfloor \lg \lg n \rfloor - j)$, so the amortized cost so far is $O(\lg q(x) + \lg \lg n)$, but $|Q_j|$ might have just become too small. If that happens, we remove $r = 2^{(2^j)} - |Q_j| \geq 2^{(2^j - 1)}$ elements from the tail of Q_{j+1} to insert them at the head of Q_j, and we reconstruct D_j from $\cup_{\ell=1}^{j} Q_\ell$. The actual cost of this operation is no more than ar for some constant a. But the potential has now decreased by $cr(\lfloor \lg \lg n \rfloor - j)$ for structure j and increased by $cr(\lfloor \lg \lg n \rfloor - (j+1))$ for structure $j + 1$, and so altogether, it has decreased by cr. So if $c > a$, this reconstructing operation has a null amortized cost. We now might still have to reconstruct the structure $j + 1$ and so on, but all this is also done for free, and so the amortized query time for x is $O(\lg q(x) + \lg \lg n)$. □

Note that if the original data structure allows $O(1)$ time deletions when supplied with a pointer to the element (e.g. marking the deleted elements), then the transformed data structure could support amortized $O(\lg \lg n)$ time deletions by simultaneously deleting elements from all D_i whenever they are deleted from the structure, and reconstructing the structure whenever some constant fraction of the elements have been deleted. If insertions are supported in the original data structure, they can be supported in the transformed structure with the same amortized times by always inserting in D_k and rebuilding the structure whenever n grows by some constant factor.

Corollary 1. *Given a totally ordered set A of n elements, it is possible to maintain them in a dictionnary data structure supporting $O(\lg q(x) + \lg\lg n)$ query times to access element $x \in A$.*

Proof. Let $D(A)$ be an array keeping the elements of A in sorted order. Given $D(B)$ and a set $B' \subset B$, the elements of B' given as unordered indices in $D(B)$ and $|B| = O(|B'|^2)$, it is possible to construct $D(B')$ in $O(|B'|)$ time using radix sorting. □

Corollary 2. *Given a set of n disjoint triangles in R^2, it is possible to maintain them in a point location data structure so that given a point $p \in R$, the triangle Δ containing p, if it exists, can be found in $O(\lg q(\Delta) + \lg\lg n)$.*

Proof. The data structure $D(A)$ for a set A of triangles will be the combination of the point location data structure of Kirkpatrick [12], and an array containing the vertices of the triangles in A in sorted order of their x coordinate. Given $D(B)$ and a set $B' \subset B$, and $|B| = O(|B'|^2)$, it is possible as in the previous corollary to reconstruct the sorted array of vertices for $D(B')$. Then, knowing the sorted order of the vertices for B', the point-location data structure of Kirkpatrick can be constructed in $O(|B'|)$ time. □

It is interesting to note that no data structure is known that approaches the working set running times for a point location data structure.

5 No queueish binary search trees

In this section we show that no dictionary implemented as a standard binary search tree structure can be queueish. By a standard binary search tree, we mean one that can traverse one edge in the tree or perform one rotation in unit cost. More precisely, we will consider an initial binary search tree T_0 containing n elements and a sequence s of nodes of T_0, and denote by $\chi(s, T_0)$ the minimum cost of accessing the nodes in the sequence s, starting from the tree T_0, where the cost for traversing an edge and the cost for performing a rotation in the tree is 1. We can assume without loss of generality that the nodes in T_0 are indexed in symmetric order by integers from 1 to n. Wilber [18] proves:

Theorem 2 (Wilber [18]). *There exists a sequence s^* of n distinct key values such that for any initial search tree T_0 containing n elements, $\chi(s^*, T_0) = \Omega(n \lg n)$.*

Or in other words, the amortized cost per operation for sequence s^* on any standard binary search tree structure is $\Omega(\lg n)$. To strengthen the statement of our theorem, we introduce a weaker version of the queueish property: a dictionnary data structure is said to be *weakly queueish* if a search to item x runs in amortized time $O(\lg q(x)) + o(\lg n)$. We can now prove our theorem:

Theorem 3. *No standard binary search tree structure can be queueish or even weakly queueish.*

Proof. Let $\hat{s}$ be the sequence formed by repeating the sequence s^* $\lg n$ times. By Wilber's theorem, $\chi(\hat{s}, T_0) = \Omega(n(\lg n)^2)$ for any T_0. On the other hand, any queueish or weakly queueish dictionnary data structure will run the first occurence of s^* in $O(n \lg n)$ since $q(x) \leq n$ by definition, and for the remainder of the sequence, $q(x) = 0$ and so the total cost for accessing the sequence $\hat{s}$ on a queueish data structure would be $O(n \lg n)$, and $o(n(\lg n)^2)$ on a weakly queueish data structure. This implies that a queueish or even a weakly queueish dictionnary data structure cannot be implemented as a binary search tree. □

For example, this theorem implies that the data structure presented in the previous section, which is weakly queueish, could not be implemented as a binary search tree and keep the same running times.

References

1. G. M. Adel'son-Vel'skii and E. M. Landis. An algorithm for the organization of information. *Soviet. Math.*, 3:1259–1262, 1962.
2. R. Cole. On the dynamic finger conjecture for splay trees. Part II: the proof. *SIAM J. Computing*, 30(1):44–85, 2000.
3. R. Cole, B. Mishra, J. Schmidt, and A. Siegel. On the dynamic finger conjecture for splay trees. Part I: splay sorting log n-block sequences. *SIAM J. Computing*, 30(1):1–43, 2000.
4. E. Demaine, J. Iacono, and S. Langerman. Proximate point searching. In *Proc. 14th Canad. Conf. on Computational Geometry*, pages 1–4, 2002.
5. L. Devroye. *Nonuniform random variate generation.* Springer-Verlag, New York, 1986.
6. M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1:111–129, 1986.
7. M. T. Goodrich, M. Orletsky, and K. Ramaiyer. Methods for achieving fast query times in point location data structures. In *Proc. 8th ACM-SIAM Sympos. Discrete Algorithms*, pages 757–766, 1997.
8. S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Inform.*, 17:157–184, 1982.
9. J. Iacono. New upper bounds for pairing heaps. In *Scandinavian Workshop on Algorithm Theory (LNCS 1851)*, pages 32–45, 2000.
10. J. Iacono. Alternatives to splay trees with $O(\log n)$ worst-case access times. In *Symposium on Discrete Algorithms*, pages 516–522, 2001.
11. J. Iacono. Key independent optimality. In *International Symp. on Algorithms and Computation*, 2002. To Appear.
12. D. G. Kirkpatrick. Optimum search in planar subdivisions. *SIAM J. Comput.*, 12(1):28–35, 1983.
13. D. Maier and S. C. Salveter. Hysterical B-trees. *Inform. Process. Lett.*, 12:199–202, 1981.
14. D. D. Sleator and R. E. Tarjan. Self-adjusting binary trees. *JACM*, 32:652–686, 1985.

15. D. D. Sleator and R. E. Tarjan. Self-adjusting heaps. *SIAM Journal of Computing*, 15:52–69, 1986.
16. R. Sundar. *Amoritzed Complexity of Data Structures.* PhD thesis, New York University, 1991.
17. R. E. Tarjan. Sequential access in splay trees takes linear time. *Combinatorica*, 5:367–378, 1985.
18. R. Wilber. Lower bounds for accessing binary search trees with rotations. *SIAM J. Computing*, 18(1):56–67, 1989.