Theoretical Informatics and Applications
Informatique Théorique et Applications

BIPARTITE BINOMIAL HEAPS

Amr Elmasry[1], Claus Jensen[2] and Jyrki Katajainen[3]

Abstract. We describe a heap data structure that supports Minimum, Insert, and Borrow at $O(1)$ worst-case cost, Delete at $O(\lg n)$ worst-case cost including at most $\lg n + O(1)$ element comparisons, and Union at $O(\lg n)$ worst-case cost including at most $\lg n + O(\lg \lg n)$ element comparisons, where n denotes the (total) number of elements stored in the data structure(s) prior to the operation. As the resulting data structure consists of two components that are different variants of binomial heaps, we call it a bipartite binomial heap. Compared to its counterpart, a multipartite binomial heap, the new structure is simpler and mergeable, still retaining the efficiency of the other operations.

AMS Subject Classification. 68P05, 68W01, 68W40.

1. Introduction

In this study we focus on the problem of constructing worst-case-efficient heaps. We consider heaps that are composed of *nodes*, each node storing an *element* in addition to pointers to other nodes and, if needed, some other data. We use n (m) to denote the number of nodes stored in the larger (smaller) of the manipulated data structure(s) prior to the operation in question, and $\lg n$ as a shorthand for $\log_2(\max\{2, n\})$. Furthermore, we use the term *cost* to denote the sum of the machine instructions executed and element comparisons performed. In particular, we assume that the nodes are constructed and destroyed outside the data

Keywords and phrases: Data structures, heaps, numeral systems, comparison complexity.

[1] Department of Computer Engineering and Systems, Alexandria University, Egypt; e-mail: `elmasry@alexu.edu.eg`

[2] The Royal Library, Copenhagen, Denmark; e-mail: `cjen@kb.dk`

[3] Department of Computer Science, University of Copenhagen, Denmark; e-mail: `jyrki@di.ku.dk`

structure. Consequently, in our analysis we ignore the costs attributed to memory management. As our model of computation, we use the word RAM [14]. It would be possible, with some modifications, to realize our data structures on a pointer machine. However, when using the word-RAM model, the data structures and the operations are simpler and easier to describe.

A *queue* is a data structure where new elements can be injected to one or both ends, and existing elements can be ejected from the end(s). So a queue has two doors, and sometimes one or both of these doors can be folded in both directions. Metaphorically, a *priority queue* has three doors: a front door where new elements come in, a back door where elements can get out, and a roof door through which an element with the highest priority—without loss of generality, a minimum element—can be accessed and removed. In particular, all three doors can only be folded in one direction. A *double-ended priority queue* has the fourth door, a floor hatch, through which an element with the lowest priority can be accessed and removed. In fact, the priority queues presented in this paper are more general; they are *addressable*, so that any given element can be extracted, not only those accessible via the roof and back doors. Hence, we use the term *priority heap*, or simply *heap*, rather than the term *priority queue*.

In technical words, a *mergeable heap* is a data structure which consists of a collection of nodes and supports, *inter alia*, the operations:

Minimum($\mathcal{H}$): Return a pointer to the node in heap $\mathcal{H}$ whose element is minimum.

Insert($\mathcal{H}, p$): Insert a node, already storing an element, referenced by pointer p into heap $\mathcal{H}$.

Borrow($\mathcal{H}$): Extract an unspecified node from heap $\mathcal{H}$ and return a pointer to that node.

Delete($\mathcal{H}, p$): Extract the node referenced by pointer p from heap $\mathcal{H}$.

Union($\mathcal{H}_1, \mathcal{H}_2$): Move the nodes from heaps $\mathcal{H}_1$ and $\mathcal{H}_2$ into a new heap and return a reference to it. After the operation $\mathcal{H}_1$ and $\mathcal{H}_2$ are both empty.

Of these operations, Borrow is non-standard, but its importance has been demonstrated in several earlier papers, see *e.g.* [2,5,8,9,15]. Most notably, in [9], we used several (single-ended) priority heaps to implement a double-ended priority heap. When moving elements from one priority heap to another, efficient Borrow and Insert operations were essential.

Naturally, a full program-library interface of a mergeable heap also provides other operations, but in our discussion we focus on the aforementioned operations. In most cases, the implementations of operations like Construct (which creates an empty heap), Destroy (which dismisses an empty heap), and Size (which returns the number of nodes stored in the given heap) are straightforward. A heap is *efficiently mergeable* if the cost of Union is sublinear in the number of nodes stored in the two involved heaps. The deletion of the node containing the minimum element can be accomplished by invoking Minimum followed by Delete that uses the pointer returned by Minimum as its argument. We emphasize that in this study we make no attempt to efficiently support the Decrease operation (which replaces the element at the given node with a smaller value).

From the comparison-based lower bound for sorting (see, for example, [4, Section 8.1]), it follows that Delete has to perform at least $\lg n - O(1)$ element comparisons, if Minimum and Insert only perform $O(1)$ element comparisons. Furthermore, it is known [1] that, if Union is supported at $o(n)$ worst-case cost, then Delete cannot be supported at $o(\lg n)$ worst-case cost.

A tree is said to be *heap-ordered* if its root stores the minimum element and the same is recursively true for all the subtrees (if any) of the root. A binomial heap, introduced by Vuillemin [17], consists of a collection of heap-ordered binomial trees of size 2^i, for some integer $i \geq 0$, with at most one tree of any particular size. In a binary representation of the counter n for the number of nodes stored, a 1-bit at position i indicates that there is a binomial tree of size 2^i present. Then the data structure emulates a binary-number increment when carrying out Insert, and a binary-number addition when carrying out Union. As already pointed out by Brown [2], Insert can be performed at constant worst-case cost if one relies on the redundant regular binary numbers [3] instead of the standard binary numbers. We describe different implementations of binomial heaps in Section 2.

A few years ago, we introduced multipartite binomial heaps [8] and proved that they can support Minimum, Insert, and Borrow at $O(1)$ worst-case cost, and Delete at logarithmic worst-case cost including at most $\lg n + O(1)$ element comparisons. It was a long-standing open problem how to achieve these bounds, optimal up to the constant additive terms, for the number of element comparisons performed by these four operations. A brief review of this data structure is provided in Section 3.

Our main contribution in this paper is to make multipartite binomial heaps simpler and efficiently mergeable. In their original form [8], multipartite binomial heaps are not efficiently mergeable. The main reason for this is that, to support Borrow at $O(1)$ worst-case cost, the perfect structure of one of the trees is broken. In our simplification we reduce the number of components constituting the heap from three to two. We clearly distinguish these components, and keep the structure of binomial trees inside the components intact. We call the resulting data structure a bipartite binomial heap. The key idea behind the simplification is to let the two components support Borrow directly, instead of having a separate component for borrowing. Our data structure supports Minimum, Insert and Borrow at constant worst-case cost, Delete at logarithmic worst-case cost including at most $\lg n + O(1)$ element comparisons, *i.e.* as efficiently as a multipartite binomial heap, and Union at logarithmic worst-case cost including at most $\lg n + O(\lg \lg n)$ element comparisons. We describe this data structure in Section 4.

Many other heaps are known to achieve the same—or even better—asymptotic bounds as the bipartite binomial heaps described in this paper. However, our treatment is tuned for improving the constant factors involved. In Table 1 we list the comparison complexity of heap operations for some known heaps versus the results proved in this paper.

TABLE 1. The worst-case comparison complexity of heap operations for a bipartite binomial heap and its competitors. Here n (m) denotes the number of elements stored in (the smaller of) the manipulated data structure(s) prior to the operation in question. All data structures support MINIMUM at $O(1)$ worst-case cost involving no element comparisons. The "–" sign means that the operation was not discussed in the original source.

Data structure	INSERT	BORROW	UNION	DELETE
Binomial heap [17]	$\lg n + 1$	–	$\lg n + 1$	$2 \lg n$
Run-relaxed heap[a] [5]	$O(1)$	0[b]	$\lg m + O(1)$	$2 \lg n$
Fat heap[a] [13,15]	$O(1)$	$O(1)$	$1.27 \lg m$	$2.53 \lg n$
Multipartite binomial heap [8]	$O(1)$	0	–	$\lg n + O(1)$
Two-tier relaxed heap[a] [10]	$O(1)$	$O(1)$	$5 \lg m + O(\lg\lg m)$	$\lg n + O(\lg\lg n)$
Fast mergeable heap [11]	$O(1)$	–	$O(1)$	$2 \lg n + O(1)$
Optimal priority heap[a] [12]	$O(1)$	$O(1)$	$O(1)$	$\approx 70 \lg n$
Run-relaxed weak heap[a] [7]	$O(1)$	0	–	$2 \lg n + O(1)$
Bipartite binomial heap	$O(1)$	0	$\lg n + O(\lg\lg n)$	$\lg n + O(1)$

[a]Without DECREASE even though it can be supported at $O(1)$ worst-case cost.
[b]The worst-case cost is logarithmic.

A bipartite binomial heap has two components: a buffer and a main store. The purpose of the buffer is to accommodate insertions. To support INSERT at $O(1)$ worst-case cost, the buffer is implemented as a binomial heap that emulates the redundant regular binary system. To support DELETE efficiently, the size of the buffer is limited to $O(\lg n)$. When the buffer becomes too big, a binomial tree is extracted from it and incrementally merged with the main store by a background process. The main store is implemented as a binomial heap that emulates the standard binary system, and additionally it maintains prefix-minimum pointers for the roots of all binomial trees. The use of prefix-minimum pointers was the key ingredient in the comparison-optimized implementation of DELETE for multipartite binomial heaps [8]. To optimize UNION with respect to the number of element comparisons, we show how to efficiently add two numbers represented using the redundant regular binary system. As another byproduct we show how to efficiently merge two binomial heaps, each accompanied with the prefix-minimum pointers, while maintaining the prefix-minimum pointers for the resulting heap.

In the following sections, when describing our results, we start with the original structure of a binomial heap (see [17] or [4, Exercise 19-2]) and show how to refine this data structure stepwise until we eventually end up with the data structures guaranteeing the claimed bounds.

☞ *Take notice* To make it easier for the reader to observe that a particular piece of information is important, we have put a corresponding note at the margin.

2. Three Variants of Binomial Heaps

The basic building block of a binomial heap is a heap-ordered binomial tree. Vuillemin [17] mentioned other alternatives that could be used instead like a perfect tournament tree and a perfect weak heap (the latter data structure was named as such several years later [6]). Other variations have been proposed like relaxed heaps [5] (that allow heap-order violations) and fat heaps [15] (that use trinomial trees, not binomial trees). In this section we explain how the standard binomial heaps (Section 2.1) can be modified to get the improved bounds. For this study, two variants will be relevant: One that can support INSERT at $O(1)$ worst-case cost (Section 2.2) and another that reduces the number of element comparisons performed by DELETE from $2 \lg n$ to about $\lg n$ (Section 2.3). Most of this material is folklore; the main reason for including it here is to make the paper self-contained. However, be aware that some of our improvements are non-standard.

2.1. Standard binomial heaps

A *binomial tree* of rank 0 consists of a single node storing one element; for an integer $k > 0$, a binomial tree of rank k comprises a node and its k binomial subtrees of rank 0, 1, ..., $k-1$ attached to that node in this order. Among those, we call the root of the subtree of rank 0 the *smallest child* and that of rank $k-1$ the *largest child*. The size of a binomial tree is a power of two, and the rank of a binomial tree of size 2^k is k. In a computer realization of a binomial tree, we rely on a non-standard representation (*cf.* [4, Section 10.4]). The children of a node are maintained in a circular, doubly-linked list, called the *child list*, and each node has a pointer to its largest child. The largest child of a node has a pointer back to its parent, but for the other children the parent pointers are not used. This will give a constant-cost access to both ends of a child list and, when two lists are concatenated, only one parent pointer needs to be updated. A non-existing parent or child is indicated with a null pointer.

☜ *Child list*
☜ *Parent pointers*

For heap-ordered binomial trees the element stored at a node is not larger than the elements stored at the children of that node. If two heap-ordered binomial trees have the same rank, they can be linked together by making the root that stores the non-smaller element the largest child of the other root. We call this linking of trees a *join*. Observe that a join is possible even if the ranks of the two trees are not the same, but the resulting structure is no more binomial. A join involves a single element comparison and has $O(1)$ worst-case cost. We call the reverse of a join, where the largest child of the root is detached from a tree, a *split*. A split involves no element comparisons and has $O(1)$ worst-case cost.

A *binomial heap* [17] is a collection of heap-ordered binomial trees. Let the binary representation of $n > 0$, the number of nodes in the data structure, be $\langle b_0, b_1, \ldots, b_{\ell-1} \rangle$, where b_0 is the least significant bit and $b_{\ell-1} = 1$ the most significant bit. A binomial heap of size n has a binomial tree of rank j if, and only if, $b_j = 1$. We call the sequence $\langle b_0, b_1, \ldots, b_{\ell-1} \rangle$ the *rank sequence* of the roots. Using the standard notation for regular expressions, the rank sequence respects the pattern $\varepsilon \mid (0 \mid 1)^\star 1$. In a computer realization of a binomial heap, we can reuse the sibling pointers of the roots to keep the roots in a circular, doubly-linked list, called the *root list*. The roots appear on the root list in increasing rank order. In accordance, the rank of a root can be deduced from the rank sequence (that can be kept in one word) and need not be stored in the nodes.

The UNION operation resembles an addition of two binary numbers. When adding two 1-bits at position k, in a binomial heap a join of two trees of rank k is performed. Based on this connection and the fact that the length of the binary representation of n is bounded by $\lg n + 1$, it follows that for two binomial heaps of sizes m and n, $m \leq n$, UNION involves at most $\lg n + 1$ element comparisons and has $O(\lg n)$ worst-case cost. Now INSERT can be viewed as a special case of UNION where an integer is increased by one. Hence, INSERT involves at most $\lg n + 1$ element comparisons and has $O(\lg n)$ worst-case cost as well. Since one of the roots contains the minimum, MINIMUM involves at most $\lg n$ element comparisons and has $O(\lg n)$ worst-case cost. Also DELETE can be reduced to UNION. A root can be deleted by unlinking its subtrees and merging them with the remaining trees. To delete a non-root node, that node is repeatedly swapped with its parent until it becomes a root, and the root is deleted as above. Observe that even though we can get to the parent only via the largest child and we can stop the traversal first after reaching the root of the largest tree, the length of the path traversed is still logarithmic. Thus, DELETE involves at most $\lg n$ element comparisons and has $O(\lg n)$ worst-case cost.

☞ *Better* DELETE

Another way of implementing DELETE is to utilize borrowing [2,5,8]. To realize BORROW, we detach the root of the smallest tree, concatenate its child list with the rest of the root list, and return a pointer to the detached root. Clearly, a constant number of pointers need to be updated by this procedure and BORROW has $O(1)$ worst-case cost. To realize DELETE of a node that is different from the borrowed node, we first detach the node to be deleted. Starting with the borrowed node and the subtrees of the deleted node, by repeatedly joining the smallest two subtrees, we end up with a combined tree that is of the same size as the subtree rooted at the deleted node. The root of the combined tree is attached in place of the deleted node. To maintain the heap order, the value at the root of the combined tree is compared with that of its parent and the two nodes are swapped if necessary. If the nodes are swapped, we compare the value of the new parent with its parent, and repeat the process until the heap order is restored. The salient feature of borrow-based DELETE is that it only breaks the structure of the smallest binomial tree, even though BORROW can add several new trees to the root list.

To speed up MINIMUM, a simple idea [17] is to maintain a pointer to the node storing the minimum. Hereafter MINIMUM has $O(1)$ worst-case cost and performs no element comparisons. INSERT and UNION would need one additional element comparison to keep the minimum pointer up to date. Also BORROW must be adjusted since we do not want to update the minimum pointer if it points to the root of the smallest tree. Let x be the root of the smallest tree, y the single-node child of x if any, and z the root of the second-smallest tree if any. If the minimum is at x, the manoeuvre performed is as follows.

☞ *Fast* BORROW

(1) If y exists: Swap x and y, detach y from the root list and its children, concatenate the child list of y with the rest of the root list, and borrow y.
(2) If y does not exist and z exists: Swap x and z, detach z from the root list, and borrow z.
(3) If neither y nor z exists: Make the root list empty, set the minimum pointer to null, and borrow x.

The consequence of speeding up Minimum is more significant for Delete since, when the current minimum is deleted, the work normally done by Minimum, *i.e.* the scan over all roots, has to be done in Delete to update the minimum pointer. This increases the bound on the number of element comparisons for Delete from $\lg n$ to $2 \lg n$.

A detailed description of binomial heaps, their properties and operations (including the implementation details), can be found in many textbooks on algorithms and data structures; Vuillemin's paper [17] is also a recommended reading.

2.2. Binomial heaps with a powerful numeral system

As a consequence of keeping the rank sequence of the roots in the form of a standard binary number, Insert could have logarithmic cost due to carry propagation. It is well known that, if we repeatedly increase a binary counter by one starting from zero, at most two bit flips are done per increment in the amortized sense (see, for example, [4, Chapter 17]). By relying on a more powerful numeral system, this amortized bound can be achieved in the worst case. For example, we could use a redundant regular binary system [3]. For such a system, any non-empty string complies with the regular expression $(0 \mid 1 \mid 01^{\star}2)^{\star}(1 \mid 01^{\star}2)$. Hereby, there can be up to two binomial trees per rank. Let d_j be the jth digit within the rank sequence of the roots. The basic primitive used by the operations is a *fix* in which, if $d_j = 2$, we set $d_j \leftarrow 0$ and $d_{j+1} \leftarrow d_{j+1} + 1$. Note that a fix does not change the value of a number. For a binomial heap, a fix corresponds to a join.

☜ *Regular binary system*

In a computer realization of the rank sequence, we let each node store its rank explicitly and use a stack to record where in the rank sequence there is a 2. More precisely, the stack stores pointers to the positions in the root list where there are two consecutive binomial trees of the same rank. We use a stack since, when several joins are possible, preference is given to smaller ranks. Hence, the two trees to be joined, if any, can be found by accessing the top of the stack.

When the redundant regular binary system is in use, in connection with each Insert, a new node is added to the collection of trees, and one 2 is processed forward by a fix, meaning that a single join is executed. For a proof that this suffices to keep the representation regular, an interested reader is referred to [3]. It may also be necessary to update the minimum pointer. Therefore, Insert involves at most two element comparisons and has $O(1)$ worst-case cost.

In Borrow, after enforcing that the minimum pointer is not pointing to the node being borrowed, we detach the root of the smallest tree and move its subtrees to the collection of binomial trees. This action retains the regularity of the number representation and does not introduce any new 2's. Still, it is necessary to pop the top of the stack if the first non-zero digit of the rank sequence was a 2. In borrow-based DELETE, the number representation does not change after borrowing.

Tuned UNION

For UNION a more careful treatment is needed. Assume that we are given two regular strings of digits representing two counters. To add the counters, we process the digits of the shorter string, one by one starting from the least-significant digit, and update the digits of the longer string accordingly. Let d_i (d'_i) denote the digit at position i in the shorter (longer) string. Assume that we are processing position $i \geq 0$, and that d'_j is the first 2, if any, in the longer string where $j > i$.

(1) If $d_i = 0$: Do nothing.
(2) If $d_i = 1$:
    (a) If $d'_i = 0$, or $d'_i = 1$ and is a digit within a $01^\star$ substring: Increase d'_i by one and fix d'_j if it exists.
    (b) If $d'_i = 2$: Fix d'_i then increase it by one.
    (c) Otherwise: Increase d'_i by one and then fix it.
(3) If $d_i = 2$: Repeat the actions of the previous case twice.

To distinguish whether $d'_i = 1$ is a digit within a $01^\star$ substring or not, since we sequentially traverse the string least-significant digits first, we can remember the last digit that is not 1 before the current digit. It is not difficult, using a case-by-case analysis, to see that this algorithm computes the correct sum and that the resulting string is regular. The stack of pointers must be concomitantly updated to refer to the 2's in the resulting rank sequence.

By regularity, the sum of the digits for a string of length ℓ is at most ℓ. From the description of the algorithm, it directly follows that if ℓ is the length of the shorter string, the algorithm performs at most ℓ fixes each corresponding to one join. In a binomial heap of size m, the number of binomial trees is bounded by $\lg m + 1$. It follows that, for two binomial heaps of sizes m and n, $m \leq n$, UNION involves at most $\lg m + 1$ element comparisons and has $O(\lg m)$ worst-case cost.

2.3. Binomial heaps with prefix-minimum pointers

Prefix-minimum pointers

Starting with the standard binomial heaps, our objective is to implement DELETE involving at most $\lg n$ element comparisons such that MINIMUM requires no element comparisons. To do this, we maintain *prefix-minimum pointers* [8] for the roots of the binomial trees. The prefix-minimum pointer of a root of rank k points to the root with the smallest value among the roots of rank j for all $j \leq k$. That is, when the prefix-minimum pointers are available, we know for each root which of the roots of the smaller trees contains the minimum element in this prefix. In particular, the prefix-minimum pointer of the root of the largest tree points to the current overall minimum. The prefix-minimum pointer of a root either points to itself or to the same node as the prefix-minimum of the root of the next smaller tree; this requires one element comparison per pointer update.

To support all other operations efficiently, except INSERT, we rely on a compact representation of the prefix-minimum pointers. Let $\mathcal{I} = \{r_1, r_2, \ldots, r_t\}$ be the set of the ranks of all roots. Now we introduce a minimal set of disjoint closed intervals, that we call *stairs*, having the following two properties:

Stairs

- The union of these intervals covers the whole set $\mathcal{I}$.
- For each interval $[r_j \mathinner{..} r_k]$ the prefix-minimum pointer of every root, whose rank is in this interval, points to the root of rank r_j.

That is, the prefix-minimum pointers of all roots at the same stair point to the root whose rank is indicated by the left endpoint of this stair. Observe that only Borrow will gain from this compact representation, whereas every Insert, Delete, or Union has to partially recompute the list of stairs.

Fast Borrow

We implement Borrow carefully so that we neither perform any element comparisons nor invalidate the prefix-minimum pointers represented by the stairs. The target is that the worst-case cost of Borrow would still be $O(1)$. Let x be the root of the smallest tree, y the smallest child of x if any, and z the root of the second-smallest tree if any.

(1) If y exists: Swap x and y, detach y from the root list and its children, concatenate the child list of y and the rest of the root list, and borrow y. Assume that the rank of x is r_j. Now replace the first stair $[r_j \mathinner{..} r_k]$ with $[0 \mathinner{..} r_j - 1]$ if $r_j = r_k$, or with $[0 \mathinner{..} r_k]$ otherwise. In both cases, the corresponding prefix-minimum pointer is set to point to x.

(2) If y does not exist, z exists, and the prefix-minimum pointer of z points to x: Swap x and z, detach z from the root list, and borrow z. Assume that the rank of z is r_j. Now replace the first stair $[0 \mathinner{..} r_k]$ with $[r_j \mathinner{..} r_k]$. The prefix-minimum pointer associated with this stair should still refer to x.

(3) If y does not exist, z exists, and the prefix-minimum pointer of z points to itself: Borrow x and remove the first stair.

(4) If neither y nor z exists: Borrow x and make the root and stair lists empty.

We rely on borrow-based Delete. Let k be the rank of the binomial tree that contains the deleted node. The number of element comparisons involved when deleting the node and fixing the heap order is at most k. We then have to update the prefix-minimum pointers. The key idea is that we only need at most $\lg n - k$ element comparisons to recompute the prefix-minimum pointers for the roots of the larger trees. It follows that Delete involves at most $\lg n$ element comparisons.

One way to implement Insert is to rely on repeated joins. Once there are no more joins to perform, we update the prefix-minimum pointers by scanning them sequentially. Hence, Insert involves at most $\lg n + 1$ element comparisons and has $O(\lg n)$ worst-case cost. Another comparison-optimized way is to use binary search to find the first root whose element is smaller than that in the given node. When this position is known, joins can be performed and the prefix-minimum pointers can be updated without any further element comparisons. This reduces the number of element comparisons performed per Insert to at most $\lg \lg n + 1$, even though Insert still has $O(\lg n)$ worst-case cost.

Tuned Insert

A straightforward implementation of Union would be to perform all the possible joins and then update all the prefix-minimum pointers, for a total of at most $2 \lg n$ element comparisons. To improve this bound to $\lg n + 1$, we implement Union

Tuned Union

more carefully. The main idea is to sequentially consider trees from the smaller to the larger ranks such that for each rank we only consume one element comparison for either performing a join or updating a prefix-minimum pointer, but not both. Assume that we have already merged and updated the prefix-minimum pointers of the roots of the trees whose ranks are less than k. There is at most one tree of rank k from each heap and at most one more tree of rank k that results from the previous joins. If in total there is one tree of rank k, we only update the prefix-minimum pointer of its root. If in total there are two trees of rank k, we only join the two trees, resulting in no trees of rank k in the combined heap. If there are three trees of rank k, we only need to perform a join but not a prefix-minimum pointer update for the leftover tree of rank k; the details of this case are as follows. Let x be the root of the tree of rank k from the heap that contains the minimum element among all smaller trees handled so far, let y be the root of the tree of rank k that results from the previous joins, and let z be the root of the third tree of rank k. There are two cases that can be distinguished by a pointer comparison:

(1) If the prefix-minimum pointer of x is pointing to y: Let the prefix-minimum pointer of y point to itself, and join the trees rooted at x and z.
(2) Otherwise: Keep the prefix-minimum pointer of x as it is, and join the trees rooted at y and z.

It is not difficult to see that the above algorithm correctly maintains the prefix-minimum pointers for the roots of the merged heap.

3. Multipartite Binomial Heaps

In this section we review the operational principles of multipartite binomial heaps [8], which were designed to reduce the number of element comparisons performed by Delete to $\lg n + O(1)$ while keeping the worst-case cost of Minimum, Insert, and Borrow a constant. A multipartite binomial heap has three components (assuming that the floating tree is part of the main store):

Buffer: This is a binomial heap, with a minimum pointer, relying on the redundant regular binary system (as in Section 2.2). The buffer stores $O(\sqrt{n})$ elements and is responsible for handling insertions.

Main store: This is a binomial heap augmented with prefix-minimum pointers (as in Section 2.3). A big portion of the n elements is stored in this part of the data structure.

Floating tree: This is a single binomial tree. It is needed to regulate the traffic between the buffer and the main store, as it is necessary to move elements to the main store when the buffer overflows.

Reservoir: This is a single tree, initially a binomial tree, but it gradually forfeits its perfect structure while nodes are borrowed or deleted. The reservoir is never larger than the main store.

In the original description [8], the buffer and the main store were tightly coupled. We view the components as separate heaps and the floating tree as an annex to

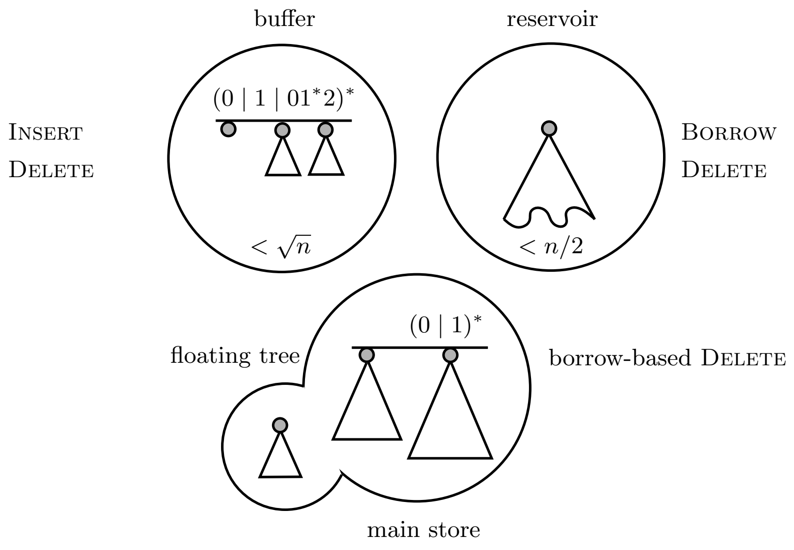

Figure 1. A multipartite binomial heap illustrated in abstract form.

the main store. An illustration of the data structure, together with the operations supported by the different components, is given in Figure 1.

When the buffer becomes too big, one of the largest two trees or half of the largest tree is moved to the main store. Analogously, when the reservoir becomes empty, one of the largest trees or half a tree is moved from the buffer (or from the main store if the buffer is empty) to the reservoir. This relocation should avoid invalidating any of the pointers pointing to the roots. Moreover, since a split of a binomial tree can be carried out at constant worst-case cost, these two operations do not introduce any significant overhead to the heap operations. When a tree is moved to the main store, its transplantation has constant worst-case cost, but the merge of this floating tree into the main store would have logarithmic worst-case cost. Hence, this merge has to be done incrementally keeping both components functional throughout the process. As proved in [8], an important point is that a logarithmic number of new insertions—within which this incremental work will be performed—are to be executed before the buffer overflows again. In consequence, there will always be at most one floating tree in the data structure. As Delete is borrow-based for the floating tree, its structure remains binomial until it is integrated with the main store.

In [8], the reservoir is the most problematic component; it was introduced to support fast Borrow (in Sections 2.2 and 2.3 we showed how to efficiently perform this operation for the buffer and the main store). To execute Borrow from the reservoir, the smallest child of the root is detached and a pointer to it is returned. Subsequently, the child list of this borrowed node is appended to the child list of the root. If the reservoir contains only a single node, this node is borrowed and an underflow operation is applied which refills the reservoir with a subtree from the buffer (or from the main store if the buffer is empty). In DELETE for the reservoir, the given node is repeatedly swapped with its parent until it becomes the root, the root is then detached from its children, and the children are joined from the smaller to the larger. Here we join trees that are not of the same size and not even binomial. Still, as proved in [8], the reservoir can accommodate BORROW and DELETE within the required bounds. Since the binomial structure of the reservoir is broken, it is not known how to merge two reservoirs efficiently; this is the reason why we needed a different implementation to make the data structure efficiently mergeable.

Abnormal joins

Let $\mathcal{H}$ be a multipartite binomial heap, and let B, R, S, and T be its buffer, reservoir, main store, and floating tree, respectively. A subtle detail here is that every root should know in which component it lies. This owner information is needed for deciding which type of DELETE to invoke. When a tree is moved from one component to another, due to a buffer overflow or reservoir underflow, the owner information at the root can be updated accordingly. In joins it is equally easy to update this information.

Owners

Let us now consider how the heap operations are implemented for $\mathcal{H}$ by employing the corresponding operations for the individual components.

MINIMUM($\mathcal{H}$)**:** Since the overall minimum can be in any of the components, we maintain a pointer to the overall minimum among the four minima. Subsequently, minimum finding involves no element comparisons and the additional overhead introduced to the other operations is only a constant.

INSERT($\mathcal{H}, p$)**:** Invoke INSERT(B, p). If B becomes too big, cut one of the largest two trees or half of the largest tree from B such that the minimum pointer is not invalidated and make the cut tree the floating tree T for S. If possible, perform one step of the incremental process merging T into S.

BORROW($\mathcal{H}$)**:** If the reservoir happens to be empty, move one of the largest two trees or half a tree from B (or from S if B is empty) to R such that the minimum pointer (any of the prefix-minimum pointers) is not invalidated. After this, invoke BORROW(R) and return the borrowed node.

DELETE($\mathcal{H}, p$)**:** Starting from the node pointed to by p, traverse the nodes until reaching the root of the tree where p is located. Consult in which component this root lies; in accordance, invoke DELETE(B, p), DELETE(R, p), DELETE(S, p), or DELETE(T, p).

As to the performance of these operations, the claimed bounds directly follow from the bounds derived for the individual components. Since the size of the buffer is limited to $O(\sqrt{n})$, matching the bound for the DELETE operations for the other components, DELETE(B, p) will perform at most $\lg n + O(1)$ element comparisons. For a more detailed description and analysis of the data structure, we refer to [8].

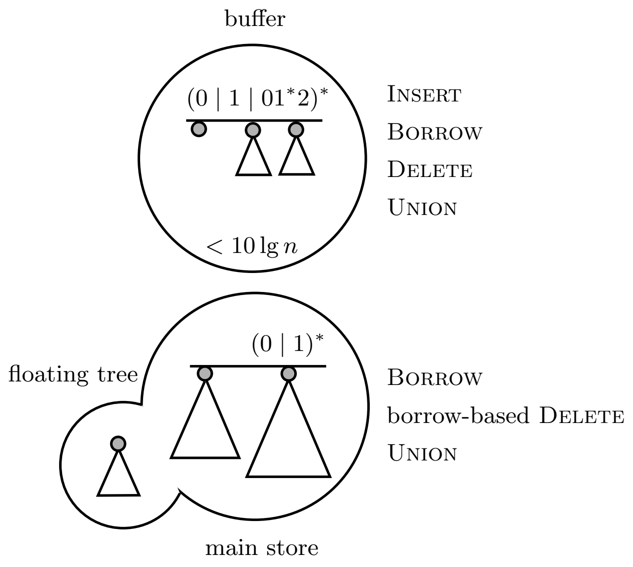

FIGURE 2. A bipartite binomial heap illustrated in abstract form.

4. Bipartite Binomial Heaps

Now we are ready to describe a simplified version of multipartite binomial heaps. We can summarize the changes made as follows:

- As described, we endow the buffer and the main store to support BORROW at $O(1)$ worst-case cost. This makes the reservoir obsolete.
- We retain the binomial structure of the building blocks used in the two components. This makes the data structure efficiently mergeable.
- We limit the size of the buffer to $O(\lg n)$. This helps us to improve the constant factor in the leading term for the bound on the number of element comparisons performed by UNION.

Because of the first two properties we name the resulting data structure a *bipartite binomial heap*. In its basic form a bipartite binomial heap has a buffer, a main store, and possibly a floating tree as an annex to the main store. These components are implemented as in a multipartite binomial heap: The buffer is a binomial heap, with a minimum pointer, relying on the redundant regular binary system; the main store is a binomial heap augmented with prefix-minimum pointers; and the floating tree is a single binomial tree. An illustration of the data structure is given in Figure 2. Let $\mathcal{H}$ be a bipartite binomial heap, and let B, S, and T be its buffer, main store, and floating tree, respectively. Next we consider how the different heap operations are implemented for $\mathcal{H}$.

We still maintain a pointer to the smallest among the minima of B, S, and T. Therefore, MINIMUM($\mathcal{H}$) just returns the node pointed to by this pointer.

Clearly, MINIMUM involves no element comparisons and has $O(1)$ worst-case cost. At the end of every INSERT, DELETE, and UNION, the minimum pointer is updated whenever necessary, involving a constant amount of extra work per operation.

In INSERT$(\mathcal{H}, p)$, the node pointed to by p is inserted into the buffer by invoking INSERT(B, p). Let n_B be the size of B and n_S that of S, *i.e.* $n_B + n_S = n$. When limiting n_B to $O(\lg n)$, we maintain an invariant that the buffer always contains at most one tree of rank $\lceil \lg \lg n_S \rceil + 1$ and no trees of higher rank. Immediately, when an insertion creates two trees of rank $\lceil \lg \lg n_S \rceil + 1$, an overflow occurs and one of the largest trees is moved to the main store as a floating tree. Naturally, we avoid selecting a tree if the minimum pointer points to it. As before, this floating tree is incrementally merged with the trees in the main store. The work is distributed evenly among the forthcoming $\lceil \lg n_S \rceil + 1$ INSERT operations such that each INSERT will perform a constant amount of extra work. Each step of the incremental process will go forward one rank. The process can be in three different states: Either it moves a floating tree forward since its rank is larger than the rank of the current tree, it joins the floating tree and a tree of the same size forming a larger floating tree, or it updates a prefix-minimum pointer after the floating tree has reached its final location.

☞ *Redesigned overflow*

INSERT has $O(1)$ worst-case cost and may involve four element comparisons (one for a join inside the buffer, one to possibly update the minimum pointer of the buffer, one to update the overall minimum pointer if necessary, and one to advance the incremental merge in the main store if necessary).

In BORROW$(\mathcal{H})$, if the buffer is non-empty, we invoke BORROW(B), else we invoke BORROW(S). As explained before, BORROW involves no element comparisons and has $O(1)$ worst-case cost. Observe that, if a node from the buffer is borrowed, this is advantageous for the incremental merge since it will have more time to finish before the next overflow occurs. Formally, the correctness is proved in the following theorem.

Theorem 4.1. *At most one floating tree exists at any given point of time.*

Proof. Let $\ell = \lceil \lg \lg n_S \rceil + 1$ just prior to an overflow. When one of the two trees of rank ℓ has been removed from the buffer, all the digits in the rank sequence of the roots are either 0's or 1's. In particular, the digit at position $\ell - 1$ must be a 0. Hence, the buffer can contain at most $2^\ell + \sum_{i=0}^{\ell-2} 2^i$ nodes, which is at most $2^\ell + 2^{\ell-1} - 1$. To produce another tree of rank ℓ, $2^{\ell-1} + 1$ new nodes must be inserted into the structure. Due to the choice of ℓ, this is at least $\lceil \lg n_S \rceil + 1$.

Let n'_S denote the size of the main store when handling this specific overflow. Since $n_S < n'_S$, the buffer can accommodate even more than $\lceil \lg n_S \rceil + 1$ insertions before it will contain two trees of rank $\lceil \lg \lg n'_S \rceil + 1$. To sum up, it must be the case that the previous incremental merge in the main store has finished before a new incremental process starts. When the buffer is non-empty, the decrease in size always happens in it, which means that two trees of rank $\lceil \lg \lg n'_S \rceil + 1$ can never be created because of borrowings and deletions. When the buffer is empty, the main store may decrease in size, but in this case our invariant is trivially true. □

As for a multipartite binomial heap, Delete is performed in the component where the given node lies. So, the roots should still know their owners. In the main store and floating tree, Delete is borrow-based.

- In the buffer, the worst-case cost of Delete is only $O(\lg\lg n)$ due to its logarithmic size. By setting the limit $10 \lg n$ on the size of the buffer, the number of element comparisons performed is at most $2 \lg(10 \lg n)$, which is at most $\lg n$ for $n \geq 2^{16}$.
- Assuming that the rank of the floating tree (if any) is k, borrow-based Delete in there involves at most k element comparisons and has $O(k)$ worst-case cost. Depending on the current state of the incremental merging process there are two cases. First, if the floating tree has not yet reached its final location, there is no need to do anything else. Second, if the floating tree has reached its final location and some of the prefix-minimum pointers have already been updated, Delete can recompute all the prefix-minimum pointers maintained for the roots of rank k and higher, and the incremental process can be terminated. This recomputation requires $\lg n - k$ element comparisons, and the amount of work done is proportional to this. Thus, for the floating tree the worst-case cost of Delete is $O(\lg n)$.
- Since the main store is a binomial heap using prefix-minimum pointers, it supports Delete at $O(\lg n)$ worst-case cost involving at most $\lg n$ element comparisons.

Taking into account the additional two element comparisons possibly needed for updating the pointer to the overall minimum, at most $\lg n + 2$ element comparisons are performed per Delete, provided that n is large enough.

Lastly, let us consider the Union operation. Assume that the two bipartite binomial heaps to be merged are $\mathcal{H}_1$ and $\mathcal{H}_2$. In Union$(\mathcal{H}_1, \mathcal{H}_2)$, the two buffers are merged as described in Section 2.2, and the two main stores are merged as described in Section 2.3. The merge of the buffers may produce up to two new overflow trees. Also, the already existing two floating trees, if any, need a special handling. To get rid of (up to three and leave one of) these floating and overflow trees, we insert them one by one into the main store that resulted from the merging. We insert these extraneous trees using binary search in the same way as we inserted a single node into a binomial heap that uses prefix-minimum pointers.

The worst-case cost of merging the buffers is $O(\lg\lg m)$ including at most $\lg\lg m + 3$ element comparisons. The merging of the two main stores has $O(\lg n)$ worst-case cost and involves at most $\lg n + 1$ element comparisons. The worst-case cost of handling the extraneous trees is $O(\lg n)$, but this involves at most $3 \lg\lg n + 3$ element comparisons. The update of the pointer to the overall minimum involves one element comparison. Thus, in total, at most $\lg n + 4 \lg\lg n + 8$ element comparisons are performed per Union.

Table 2. The worst-case comparison complexity of heap operations after each refinement. Here n (m) denotes the number of elements stored in (the smaller of) the manipulated data structure(s) prior to the operation in question. All structures support Minimum at $O(1)$ worst-case cost involving no element comparisons. The improvements over known bounds are highlighted.

Data structure	Insert	Borrow	Union	Delete
Binomial heap				
• minimum pointer [17]	$\lg n + 1$	0	$\lg n + 1$	$2 \lg n$
• powerful numeral system [2]	2	0	$\lg m + 1$	$2 \lg n$
• prefix-minimum pointers [8]	$\lg\lg n + 1$ [a]	0	$\lg n + 1$	$\lg n$
Bipartite binomial heap	4	0	$\lg n + 4 \lg\lg n + 8$	$\lg n + 2$

[a]The worst-case cost is logarithmic.

5. Concluding Remarks

We showed that, starting from a textbook version of a binomial heap, via data-structural transformations, it was possible to obtain a state-of-the-art mergeable heap that is asymptotically optimal and nearly constant-factor optimal with respect to the number of element comparisons for the considered heap operations. The improvements achieved by stepwise refinements are summarized in Table 2. Although we described the transformations using binomial trees as the building blocks, the transformations are quite general; some of them only require that the building blocks can be efficiently joined.

As for the comparison complexity of heap operations, the following open questions still remain.

- By applying data-structural bootstrapping [16, Chapter 10], which moves the cost of Union to Delete, we would be able to achieve $2 \lg n + O(\lg\lg n)$ element comparisons per Delete and $O(1)$ worst-case cost per Union. The basic idea of bootstrapping is to allow heaps to store both elements and heaps. This bound is a bit weaker than the best known upper bound $2 \lg n + O(1)$ [11] on the number of element comparisons performed by Delete, when Union has $O(1)$ worst-case cost. Furthermore, there is a gap between the lower bound $\lg n - O(1)$ and this best known upper bound. Is it possible to achieve a bound of $\lg n + O(1)$ element comparisons per Delete when Minimum, Insert, and Union have constant worst-case cost?
- There seems to be a trade-off between the worst-case cost of Delete and that of Borrow. In [11], the bound $2 \lg n + O(1)$ on the number of element comparisons per Delete was achieved, but the guaranteed worst-case cost of Borrow is logarithmic. For a bipartite binomial heap, using bootstrapping [16, Chapter 10], the bound $2 \lg n + O(\lg\lg n)$ per

DELETE would be achievable and the worst-case cost of BORROW would still be $O(1)$. What is the best possible bound on the number of element comparisons performed by DELETE when MINIMUM, INSERT, BORROW, and UNION are required to have constant worst-case cost?

- Consider extending the operation repertoire with DECREASE, which is defined as follows:

  DECREASE($\mathcal{H}, p, v$)**:** Replace the element stored at the node referenced by pointer p in heap $\mathcal{H}$ with element v, assuming that the new element is not larger than the old element.

  What is the number of element comparisons performed by DELETE when all other heap operations (MINIMUM, INSERT, BORROW, DECREASE, and UNION) are required to have $O(1)$ worst-case cost? At the time of writing this paper, the best known upper bound is around $70 \lg n$ element comparisons per DELETE [12].

REFERENCES

[1] G. S. Brodal, Fast meldable priority queues, *Proc. of the 4th International Workshop on Algorithms and Data Structures*, *Lecture Notes in Comput. Sci.* **955**, Springer, Berlin/Heidelberg (1995), 282–290.

[2] M. R. Brown, Implementation and analysis of binomial queue algorithms, *SIAM J. Comput.* **7**, 3 (1978), 298–319.

[3] M. J. Clancy and D. E. Knuth, A programming and problem-solving seminar, Tech. Rep. **STAN-CS-77-606**, Dept. Comput. Sci., Stanford Univ., Stanford (1977).

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd Edition, The MIT Press, Cambridge (2009).

[5] J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan, Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation, *Commun. ACM* **31**, 11 (1988), 1343–1354.

[6] R. D. Dutton, Weak-heap sort, *BIT* **33**, 3 (1993), 372–381.

[7] S. Edelkamp, A. Elmasry, and J. Katajainen, The weak-heap data structure: Variants and applications, *J. Discrete Algorithms* **16** (2012), 187–205.

[8] A. Elmasry, C. Jensen, and J. Katajainen, Multipartite priority queues, *ACM Trans. Algorithms* **5**, 1 (2008), Article 14.

[9] A. Elmasry, C. Jensen, and J. Katajainen, Two new methods for constructing double-ended priority queues from priority queues, *Computing* **83**, 4 (2008), 193–204.

[10] A. Elmasry, C. Jensen, and J. Katajainen, Two-tier relaxed heaps, *Acta Inform.* **45**, 3 (2008), 193–210.

[11] A. Elmasry, C. Jensen, and J. Katajainen, Strictly-regular number system and data structures, *Proc. of the 12th Scandinavian Symposium and Workshops on Algorithm Theory*, *Lecture Notes in Comput. Sci.* **6139**, Springer, Berlin/Heidelberg (2010), 26–37.

[12] A. Elmasry and J. Katajainen, Worst-case optimal priority queues via extended regular counters, *Proc. of the 7th International Computer Science Symposium in Russia*, *Lecture Notes in Comput. Sci.* **7353**, Springer, Berlin/Heidelberg (2012), 130–142.

[13] A. Elmasry and J. Katajainen, Fat heaps without regular counters, *Discrete Math. Algorithms Appl.* **5**, 2 (2013), Article 1360006.

[14] T. Hagerup, Sorting and searching on the word RAM, *Proc. of the 15th Annual Symposium on Theoretical Aspects of Computer Science*, *Lecture Notes in Comput. Sci.* **1373**, Springer, Berlin/Heidelberg (1998), 366–398.

[15] H. Kaplan, N. Shafrir, and R. E. Tarjan, Meldable heaps and Boolean union-find, *Proc. of the 34th Annual ACM Symposium on Theory of Computing*, ACM, New York (2002), 573–582.

[16] C. Okasaki, *Purely Functional Data Structures*, Cambridge University Press, Cambridge (1998).

[17] J. Vuillemin, A data structure for manipulating priority queues, *Commun. ACM* **21**, 4 (1978), 309–315.