Journal of Undergraduate Research at Minnesota State University, Mankato

Volume 10 Article 9

2010

Verification of Costless Merge Pairing Heaps

Joshua Vander Hook
MInnesota State University, Mankato

Verification of Costless Merge Pairing Heaps

Joshua Vander Hook, Dr. Dean Kelley

August 9, 2010

Abstract

Most algorithms' performance is limited by the data structures they use. Internal algorithms then decide the performance of the data structure. This cycle continues until fundamental results, verified by analysis and experiment, prevent further improvement. In this paper I examine one specific example of this. The focus of this work is primarily on a new variant of the pairing heap. I will review the new implementation, compare its theoretical performance, and discuss my original contribution: the first preliminary data on its experimental performance. It is instructive to provide some background information, followed by a formal definition of heaps in 1.1. I also provide a brief overview of existing literature on the design of these data structures in 1.2 and discuss the methods for evaluating these types of structures in 1.3. Full details about the implementation of a pairing heap can be found in 2.2. Ongoing research has produced a variety of different types of heaps, which will be briefly discussed.

1 Background

A heap is a type of set. The motivation behind heaps was to minimize the cost of retrieving the minimum element (min heaps) or maximum element (max heaps) from the set. Both types of heaps are analogous. Their only difference is how the elements of the set are compared. For the sake of readability we will consider only min heaps.

All heaps provide the following template methods:

insert(h,x) Add x to the heap h.

removemin(h) Remove and return the minimum element from the heap h.

The term heap is from Williams, who also described the heap sort[32]. Alternatively, heaps are also known as priority queues thanks to Donald Knuth[19]. To further confuse terminology, some differentiate between heaps that can be merged, and those that cannot[31]. For the purposes of this paper, read "heap" as merge-able heaps. Consider a priority queue a non-merge-able heap. To be a merge-able heap, a third method is supplied:

merge(h_1, h_2) Merge the two item-disjoint sets $h_1$and h_2, return the result and destroy the original heaps.

The development of heaps was driven by real-world optimization problems. Heaps are important research topics because they form the backbone for a number of commonly used algorithms. Thus, an improvement to heaps improves the performance of many existing solutions.

Table 1 shows some of the algorithms which are based on heaps, and an example of their current-day use.

Table 1: Heap-based algorithms

Heap-Based Algorithm	Designed for:	Example of use
Minimum Spanning Tree (MST)	Graph Optimization	Wireless Sensor Networks
Link-State Routing	Network Traffic	OSPF Routing Protocol
Dijkstra's Shortest Path	Graph Traversal	Google Maps
Earliest Deadline First Scheduling	Operating Systems	Industrial Control

1.1 Formal Definition of Heaps

Formally, a heap is any data structure that honors the heap-ordering property[5]. This states (again, for a min heap) that for any node (or position) having key k and children $\{c_1, c_2, \ldots, c_n\}$,

$$k = min(\{c_1, c_2, \ldots, c_n\} \cup \{k\}).$$

Or simply, that the element with the minimum key is the root of the subtree. The heap property makes no claims about the ordering of the children. This is important because it allows some flexibility. Exploiting this internal flexibility provides us with efficient heaps.

Both the collections in Figure 1 are min heaps. In practice, tree-based heaps are often used (left), because of their lower overall cost for expanding and contracting to fit arbitrarily-sized collections[5]. In light of this, our current use of the term "heap" can be expanded to mean a merge-able heap-ordered endogenous[1] tree.

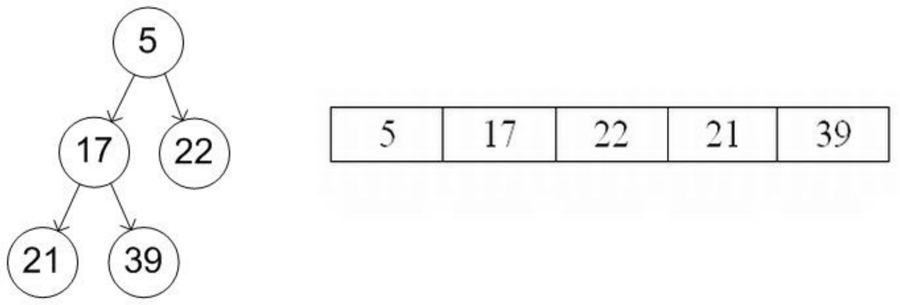


Figure 1: Two equivalent heap-ordered structures: a binary tree (left) and array (right). The numbers shown are the key values associated with the heap positions. The array-based heap says that for any position i its children are in $2i$ and $2i + 1$.

1.2 Heaps to Date

Heaps were described first by J. Williams[32]. Williams' data structure was conceptually organized as a tree, but was physically stored as a contiguous array of keys. This organization (pictured in Figure 1) was dubbed the implicit

[1]Lit. "Growing from within." It means we do not distinguish node key values from the nodes actual.

heap for this reason. However, this was essentially the first binary heap, and was formally described by Floyd[9].

Despite their origin as a sorting mechanism, the real motivations behind continuing heap development were graph-theoretic. Prior to the growth of research into data structures, network optimization algorithms were already maturing. A solution to the minimum spanning tree (MST) problem dates to 1926 and a study of power line routing[15]. The algorithm presented was improved upon and simplified by Kruskal[20] and Prim[23]. Their work produced three algorithms which use graphs in the form of adjacency lists or matrices. However, for sparse graphs (those with few connections relative to the number of nodes), the heap is a time and memory-saving solution to these problems. The MST problem is quite fascinating, with a wide array of practical and theoretical uses including approximation algorithms for the famous Traveling Salesman Problem. An excellent history can be found in [15]. These algorithms have found widespread use in networking and other areas, which added to the importance of this research[29].

Following the discovery of heap-based solutions to networking problems, research began in earnest on defining efficient heaps. The results gave us the binomial heap[4] (an important predecessor to the pairing heap), leftist heap[6], and others. These heaps are usually tree-based. The study of tree-based structures is beyond the scope of this (or perhaps any single) paper. However some relevant details should be mentioned.

Trees as data structures seem to emerge around 1962 with [16]. These structures were expanded to include better performance from balancing and rigidity[1]. Knuth[18] described an optimal binary search tree using prior knowledge of access probabilities. It earned the name Optimal because it altered the internal structure of the tree to provide the minimum access times to all elements over the entire sequence of accesses. It is worth mentioning that this was fourteen years before amortized analysis, which generalized the analysis method of averaging over a sequence.

Knuth's optimal binary search tree worked well with apriori access probabilities, but for practical applications it may be better to construct the tree using the actual access sequence. In 1978 Allan and Murno worked to create a self-adjusting binary search tree that was competitive[2] with Knuth's optimal[2]. The self-adjusting binary search tree was refined by Sleator and Tarjan, and is known as the splay tree[25]. This used a splay technique (path shortening) to simplify previous work in dynamic data structures[26]. In this paper (refined in [27]), they presented a heap version, known as the skew heap, which they regarded as a self-adjusting version of the leftist heap. It could be classified as a heap-ordered binary tree. This was an interesting step, as it abandoned the

[2]meaning: within a constant multiplier of optimal

rigid internal structure that was common (and still is). According to their work, the skew heap out performed most existing heap implementations.

Sleator and Tarjan took these results and applied similar modifications to the binomial heap. They called the result the pairing heap[12]. At the time, they were only able to prove $O(logn)$ bounds for each operation. The pairing heap would emerge as a top performer in subsequent analysis and experiments. The question of complete analysis is still open, though the bounds have been tightened[22].

The following year, the Fibonacci heap was developed, which provided excellent theoretical bounds[13]. To date, these performances seem to be the gold standard for heaps of this type. However, the implementation of a Fibonacci heap is complex, due again to rigid internal structure. Furthermore, experimental results by Moret and Shapiro have shown the pairing heap to be the structure of choice for the MST problem[21]. This, coupled with other experimental results which favor the pairing heap tend to preclude Fibonacci Heaps from common use[7, 11, 17, 21, 22, 24, 28].

The ultimate goal remains to find a heap with minimal complexity that performs as well as the Fibonacci heap. However, the pairing heap has a long-standing reputation as an excellent performer with relatively simple implementation.

Fitting with the existing literature, we can classify the pairing heap as a d-ary self-adjusting heap-ordered tree. In 2.2 we give a greater detail of it's implementation, while full details can be found in[12], [10], [11] and [22]. Two variants of this structure are the subject of our experimental results.

1.3 On Performance

When we speak of performance of a data structure, we address two things-its theoretical performance and its experimental performance. A good theoretical bounds for a data structure tells us how it will perform under the worst case, and often how it will perform on average. These measures are good validation of a design, and can provide insights into the use of a data structure.

1.3.1 Theoretical Results

There is a theoretical bounds for heaps. Michael Ben-Or showed that for n items, they can be sorted in at best $O(nlogn)$ comparisons[3]. Some exceptions to this rule exist if there are insights into the information that can be exploited, but in general the bounds holds. Using a heap we could perform *n insert* operations, followed by *n delete-min* operations to sort any arbitrary data. For the lower bounds on sorting to hold, one of those operations must be at least $O(logn)$ in complexity.

Theoretical results are concerned with the worst-case performance per operation. This type of analysis dominates computer science, because one can guarantee that a particular data structure or algorithm will perform no worse than some bounds. However, it is occasionally more interesting to evaluate the average performance over some worst-case *sequence* of operations. Note that this is still a theoretical result; the analysis simply defines what may be the worst case sequence and attempts to provide a bounds for its operating cost. This type of analysis is particularly insightful for self-organizing data structures, or any structure in which varying amounts of work are done during subsequent calls to the same method. This analysis is called Amortized Analysis, and was first described in [30].

There are two types of amortized analysis, and both are used to analyze pairing heaps. Most common is the potential method. It states that the amortized cost of the operations is equal to the change in potential of the structure plus the actual cost of the operation. Intuitively, the potential of a structure could be thought of as the energy of the structure. A change in energy implies that work was done, and thus, our work had unintended (or, more accurately, unaccounted-for) side effects which must be tallied.

Formally, the amortized time a of operation i is shown by $a_i = t_i + \varphi_i - \varphi_{i-1}$ where φ is the potential of the structure after operation i and t is the actual time. Thus, the total cost of a sequence of operations can be given by:

$$\sum_{i=1}^{m} t_i = \sum_{i=1}^{m} (a_i - \varphi_i + \varphi_{i-1}) = \varphi_0 - \varphi_m + \sum_{i=1}^{m} a_i$$

If it can be arranged that φ begins at zero and remains non-negative, then the amortized cost provides an upper bound to the actual run time over the given sequence[30].

$$\sum_{i=1}^{m} t_i \leq \sum_{i=1}^{m} a_i$$

Another method of amortized analysis is called the *accounting method*. It allocates a certain number of credits to each data item, and charges these credits for operations. The analysis can be made more complicated by introducing saving credits (which averages forward in time) and borrowing. This is merely an

Table 2: Theoretical Heap Performances.

	insert	merge	decrease	remove-min	delete
Skew Heap	$O(logn)$	$O(logn)$	$O(logn)$	$O(logn)$	N/A
Leftist Heap	$O(logn)$	$O(logn)$	$O(logn)$	$O(logn)$	N/A
Binomial Heap	$\Theta(logn)$	$\Theta(logn)$	$\Theta(logn)$	$\Theta(logn)$	N/A
Pairing Heap	$O(1)$	$O(1)$	$\Omega(loglogn)$	$O(logn)$	$O(logn)$
Fibonacci Heap	$O(1)$	$O(1)$	$O(1)$	$O(logn)$	$O(logn)$

analysis tool, and is not reflected in the final implementation of the structure. There is no CPU operation for "collect credit."

Table 2 provides some known bounds for various types of heaps and their operations.

1.3.2 Experimental Results

Given that two competing structures may have similar theoretical results (as is often the case), experimental verification is needed before a structure can be accepted. All things being equal, the structure with better experimental results will probably be the most widely used.

There are several ways to verify the performance of heaps experimentally. The most obvious is to use it to sort data. While this is appropriate for any heap, it is not very enlightening given the specific applications for our type of heaps. That is, sorting does not make use of the unique operations that the pairing heap provides.

The majority of performance tests from existing literature fit into one of two types. In the first, the hold method[14, 17], a heap was used to simulate a fixed-size queue of events. In the second type, an actual graph is constructed, and the heap is used to find a minimum spanning tree or shortest path. Unfortunately, the first tests suffer from the same problem as generic sort tests. Stasko and Vitter defined a test similar to the hold method which included the *decrease* operation[28]. Their tests are used as a model for the tests used in this paper.

A third type of testing was described in [24], which used a markov model to define the queue operations. This provides some inspiration for our test methods, with minor adjustments.

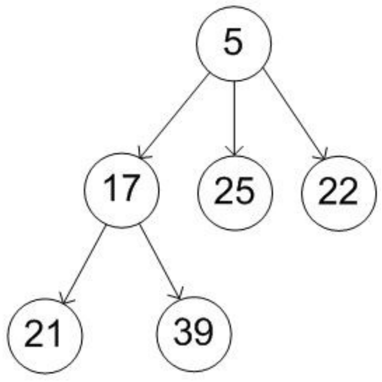


Figure 2: A Pairing Heap

2 The Pairing Heap

2.1 Description

By applying the heap property to only half of a node's children, and stating that any node can have any number of children, we have defined a half-ordered multi-way tree. The pairing heap is one such example of this structure. An example pairing heap is shown in Figure 2. Notice that the siblings of a node are not necessarily heap-ordered. Each layer of the tree could be thought of as an unsorted collection of heaps, and allows us to make some intuitive analysis in 2.4. Internally, the pairing heap uses child-sibling representation, and can be displayed as a binary tree, with the right children as siblings, and the left children as greater nodes.

The following operations are provided by pairing heaps:

makeheap(h) Initialize the heap h.

insert(h,x) Insert the value x into the heap h.

findmin(h) Return the minimum value of h.

delete(h,x) Delete the element x from the heap.

deletemin(h) Delete and return the minimum element from the heap

decrease(h,x,Δ) Decrease x in the heap h by Δ. update the heap to reflect the new minimum.

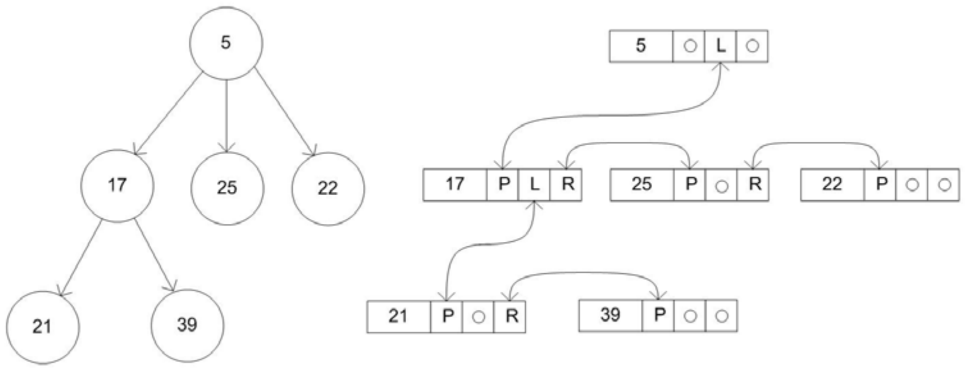


Figure 3: A multiway tree and its child-sibling representation

2.2 Implementation

A pairing heap is implemented as a structure containing a pointer to the root node, a size type to denote size, and nothing else.

2.2.1 Internal Representation

As shown in Figure 2 and Figure 3 any node can have an arbitrary number of children. While a true d-ary tree uses arrays for each evel of nodes, a pairing heap performs better because of its tree-based internal structure.

To implement this organization effectively, the child-sibling representation is used. Figure 3 is an example of a multi-way tree presented in child-sibling representation. Each node has a pointer to its left child, next sibling, and parent. In pairing heaps, the heap-ordering property is satisfied because each node is smaller than any of its distinct left children. In addition, each node has one data field and one bit to denote if the child is left or right of its parent.

2.3 Psuedo-code examples

The first five methods are simple, and shared by both the pairing heap and the costless-merge pairing heap. For brevity, in all algorithms we omit verification of parameters (such as null reference checking).

2.3.1 Shared methods.

Algorithm 1 separate

Separate: separate the node and the node's sub-tree from the parent tree. The node retains its left branch, essentially becoming a heap of its own.

```
seperate (n)
        temp = n.right
        if (isLeft(n))
                n.parent.left = temp
        else
                n.parent.right = temp
        n.right = null
        n.parent = null
```

Algorithm 2 Linkchild

Linkchild: make n the child of m. if either of the nodes is null, it is acceptable to simply return the non-null node

```
linkchild (m, n)
        n.parent = m
        temp = m.left
        m.left = n
        n.right = temp
```

Algorithm 3 Make Heap

Makeheap has little work to do, simply initializing the pointers to some default value.

```
makeheap(h)
        h.root = null
```

Algorithm 4 Merge

Merge is used by most of the other operations. It is performed simply by comparing the values of the root data, and linking the tree with the greater value to the lesser's left branch.

```
merge(m,n)
        if (m.data < n.data)
                linkchild(m,n)
                return m
        else
                linkchild(n,m)
                return n
```

Algorithm 5 Insert

Insert can be accomplished by creating a new heap with *x* as the root, and calling *merge(h, x)*. The new heap's node can be returned so that future operations on the node can be performed.

```
insert(h,x)
        makeheap(h2)
        h2.root = new node(x)
        ret = h2.root
        h = merge(h,h2)
        return ret //to allow future operations on the node inserted
```

Algorithm 6 FindMin

Findmin simply returns the data field from the root of the tree.

```
findmin(h)
        return h.root.data
```

2.3.2 Distinct Pairing Heap Methods

The remaining methods are more difficult. Since these heaps do not support (quick) searching, an explicit pointer to the node must be passed in (hopefully stored after insertion). Both of these operations perform a subtle but important step of pruning the internal tree. By separating the sub-tree and merging it into the root, the tree is made more shallow. This "flattening" effect is actually undesirable (see 2.4). However, it is preferred to the binary heap's *sift up* because of its good experimental performance.

Algorithm 7 Delete

```
delete(h,n)
        seperate(n)
        h2 = deletemin(n) //see section 2.3.3
        return merge(h,h2) //merge the tree back in, minus n
```

Algorithm 8 Decrease

```
decrease(h,n,d)
        n.data = n.data − d
        h2 = seperate(n)
        return merge(h,h2)
```

2.3.3 Delete-Min

Delete-min is where the real work takes place. This method has been redesigned no less than six times in the search for better performance[12, 8, 17, 7, 28].

To delete the minimum node (the root) we first seperate the root from the tree, then the minimum child is selected from the root's child list and is made the new root. All other children are made the child of the new root. However, this operation must be executed carefully. If the root is allowed to accumulate too many children relative to the size of the heap (the heap is too shallow), the performance of subsequent delete-min operations will degrade (since delete-min must search the first-level children for a new root). Thus the delete-min operation does a small amount of extra work to restore some tree structure to the heap.

Delete-min can be verbalized as such:

1 remove the root

2 merge the children into N distinct heaps

3 merge the child heaps into one

Because *merge*, *decrease*, and *insert* all link sub-trees to the root, the structure tends to flatten out over time. So the question now becomes, what is the appropriate value of N (and possibly: what is the appropriate structure for those trees).

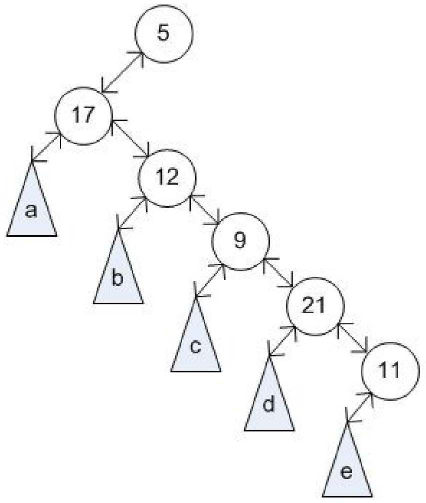
The most efficient way to do this was described as an alternate method in [12] and verified in [28]. It was called the Two-Pass Pairing Heap, and its method of *delete-min* is described as such:

1 remove the root

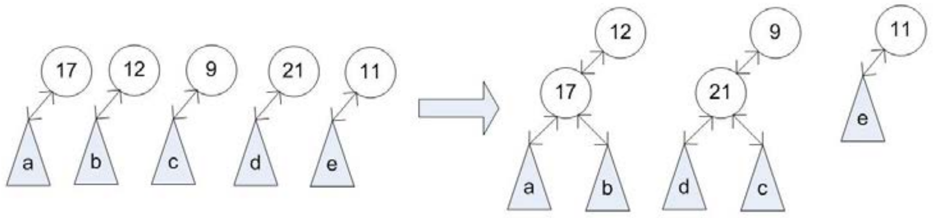
2 merge every other child with its next sibling, forming a list of new heaps

3 merge these children in reverse order, from last to first.

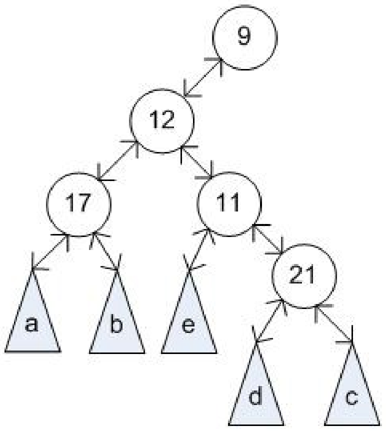
Figure 4 shows an example of this operation, and the procedure is detailed in Algorithm 9. A recursive implementation of this algorithm is arguably cleaner, but runs the risk of blowing the call stack for huge heap sizes. This problem was encountered during testing.



(a) The original heap



(b) After root deletion and subsequent pairing



(c) After merging from right to left

Figure 4: Delete-min Two Pass

Algorithm 9 Pairing Heap Delete-min Two pass

```
//n = first child of deleted root
prev = null
newroot = null
do
        if (n.right != null) //siblings remain
                y = n.right
                z = y.right
                y.parent = null; n.right = null
                z.parent = null; y.right = null
                newroot = merge(n,y)
                newroot.parent = prev
                prev = newroot
                n = z
        else
                n.parent = prev
                prev = root = n
                n = null
while (n!=null)

prev=newroot.parent
newroot.parent = null

while(prev!=null)
        y = prev.parent
        newroot=merge(newroot,prev)
        prev = y
return newroot
```

2.3.4 Variations

A number of variations of the pairing heap exist. Most of the variation is concentrated in the method of creating the new heap after a *delete-min*. The multipass variant was the original method described in[12].

1 After a delete-min, enqueue the children, merging the first two and enqueing the result.

2 Repeats this until only one heap remains in the queue.

But the two-pass seems to perform the best in experimental testing. Therefore, it is this variant that serves as the standard for the costless-merge tests.

2.4 Prior Analysis and Results

As a self-organizing data structure, it is difficult to arrive at a complete analysis of all of the pairing heap's operations. While a rigorous reproduction of the analysis is omitted, some intuitive results are critical to understanding how the pairing heap performs well despite a lack of internal structure. A full analysis can be found in [12].

Recently, Fredman showed that the Pairing Heap does not support $O(1)$ decrease cost for all sequences[11]. He showed that there exist a sequence of operations such that the cost for *decrease* is $\Omega(loglogn)$. However, sub-logarithmic amortized time was proven for the general case by Pettie[22]. Again, we omit reproducing the majority of the results, but provide an intuitive example.

Consider the pairing heap as a *d-ary* tree. Any time a delete-min operation is done on a pairing heap, the list of d children must be searched for the minimum node to promote. Since the children are essentially an unsorted list, the time to search this list is $O(d)$. Thus, it is important to keep the heap deep (a tall, narrow tree), which minimizes the number of children attached to the root. This is exactly the opposite motivation for the splay tree, which sought to bring common elements (and subsequently their immediate neighbors) up with each access. Therefore, *merge*, *insert*, and *decrease* operations actually reduce the performance of *delete-min*, and thus may reduce the overall performance of the heap over time. The constant number of pointer assignments made in these operations do not fully reflect the effect they have on the structure.

The flattening effect might explain why the multi-pass method was preferred in the original publication. The multi-pass method would produce a taller tree..

However, this method is essentially an extra imposition of structure. The two-pass method of combining children probably makes fewer comparisons overall, and therefore produces better experimental performance.

Finally, there is a fine line to walk when designing an experiment for a pairing heap. Following the lead of [22], we consider that the cost of *decrease* (at somewhere around sub-logarithmic), is probably dwarfed by the $\Theta(n\log n)$ cost paid for *n insert* and *delete-min* pairs. The ratio of *decrease* to *delete-min* has been referred to as the density of the sequence.

3 Recent Developments - The Costless Merge Pairing Heap

3.1 Description

In [7], a new variant of paring heap was described. Unlike previous variants this new structure was different in its implementation of *decrease*. The costless merge pairing heap (CMPH), exploits the fact that a heap only has to return the min node, it doesn't necessarily have to keep the min node in any particular location. Thus, whenever a node is decreased, its new value is compared to the current minimum. If it is less, the min pointer is updated to point to the decreased node. No further work is done. Asymptotically, this is the same as the PH2p's method at $O(1)$. However, since only one comparison and one pointer is updated, it has the potential to add savings over time. It is exactly this "over time" argument that is central to the analysis of these data structures.

One of the advantages of a CMPH is that it maintains a separate list of decreased nodes to avoid moving a large number of subtrees up to the root. This list is later merged during a special operation called *cleanup.* Keeping the decreased nodes separate may reduce the impact to *delete-min*, and reduce sensitivity to the density of the operations sequence.

Finally, *cleanup* combined with the normal two-pass delete-min produces a taller tree after each delete-min, provided the operations sequence was dense enough.

The CMPH provides the following operations.

makeheap(h) Initialize the heap h.

insert(h,x) Insert the value x into the heap h.

findmin(h) Return the minimum value of h.

deletemin(h) Delete and return the minimum element from the heap

decrease(h,x,Δ) Decrease x in the heap h by Δ. Update the heap to reflect the new minimum.

Delete is omitted, because it cannot be efficiently implemented.

3.2 Implementation and Intuitions

A summary of the analysis is presented to show the motivations for the optional implementations. More details can be had in [7].

The implementation of the CMPH is similar to the PH, except for the *deletemin* and *decrease* operations. However, an additional pointer assignment is made during each *insert, merge,* and *decrease*. For brevity, only the new *decrease* is shown, but each of these operations would have to update the *min* pointer if the new node is less than the root. Additionally, the *findmin* operation simply returns the data pointed to by *min*.

Algorithm 10 CMPH Decrease

The CMPH maintains an internal list of decreased nodes. In this example, d is the Δvalue, n is the node to be decreased, and h is the heap.

```
decrease(h, n, d)
        n.data = n.data −d
        if (n!=h.root)
                h.list.add(n)
                if (n.data< h.root.data)
                        h.min = n
```

17

Algorithm 11 Costless-Merge Pairing Heap cleanup()

Here, h is the CMPH, which contains the fields *root*, *decreased* (the list of decreased nodes) and *min* (a pointer to the current minimum node).

```
Let n = log(decreased.length)
if (n<2)
        na = new node[decreased.length]
else
        node[] na = new node[n]
pos = 0
while (decreased.size > 0)
        x = decreased.pop
        p = x.parent
        y = x.left
        z = x.right
        x.parent = null
        yr = null
        if (y!=null)
                yr = y.right
                y.parent = p
                if (isleft(x))
                        p.left = y
                else
                        p.right = y
                y.right = z
                if (z!=null)
                        z.parent = y
        else
                //y == null
                if (isleft(x))
                        p.left = z
                else
                        p.right = z
                if (z!=null)
                        z.parent = p
        x.left = yr
        if (yr != null)
                yr.parent = x
        na[pos] = x
        pos++

        if (pos==n)
                sort(na,0,n) //mergesort entire array
                while( --pos > 0)
                        linkchild(na[pos -1],na[pos])
                root = merge(root,build[pos])

if (pos>0)
        sort(na,0,pos)
        while( --pos > 0)
                linkchild(na[pos -1],na[pos])
        root = merge(root,build[pos])
```

Algorithm 12 Costless-Merge Pairing Heap delete-min

```
cleanup()
ret = root //assign return value
min = root = delete-min() //do standard two pass delete-min
return ret
```

From Algorithm 10 it is apparent why the CMPH does not support an arbitrary *delete.* To avoid corrupting the heap during a *deletemin* operation, the list of decreased nodes would have to be searched and the reference to the deleted node removed. This is too expensive, and so the method was not implemented.

Two other things are apparent upon inspection. First, Algorithm 11 makes the assumption that sequential nodes will be distinct. This is required because the algorithm does not make distinctions between the node to be joined to, and the node joining. If a node was decreased twice during operation, the node would be in the list twice. Depending on the order of pointer assignments, this would result in either losing a reference to the node's parent or sub-tree.[3] This may not happen commonly during the use of the data structure, but the possibility is there.

To solve this problem, the decreased list would again have to be checked for membership of the newly decreased node. This would add execution cost to the *decrease* method. However, to prevent conflict with the original design, data sets were scrubbed of matching *decrease* operations that occurred before a *deletemin* (and thus before each *cleanup*) prior to execution[4].

Futhermore, the use of a min pointer addes a comparison to each operation, and the *cleanup* operation adds overhead to both *merge* and *deletemin.* However, all of these things conspire to decrease the overall cost of a sequence of operations, and these effects would be absorbed by the benefits[5].

In [7], Elamsary showed that the amortized cost of *cleanup* is $O(logn)$.

4 Experimental Validation

The main purpose of this paper is to compare the CMPH to the two-pass pairing heap (2pPH). To accomplish this goal, they both had to be implemented. The

[3]Actually, it makes the assumption that all nodes in the list will be distinct. Performing the link-join phase on a node twice will not have devastating effects. However, separating the pool into sub-pools of size log(n) makes it more likely that only sequential nodes will cause the problem.

[4]For heaps of sizes in the hundreds of thousands of nodes, with randomly selected nodes for decrease, this was a time-consuming process. In fact, this scrubbing took the majority of the CPU time before each test.

[5]In theory

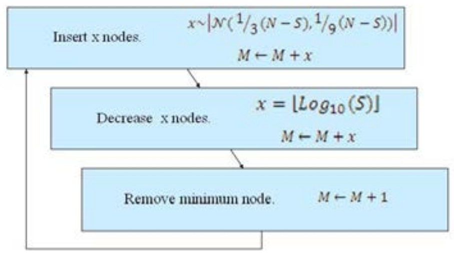
2pPH was implemented as described in 2.2. The CMPH was implemented as described above.

4.1 Methods

Programs were written to generate sequences of operations, in an effort to simulate a real-world application. These data sets were created according to the algorithm in 13. To remove the effect of the testing environment from the results (garbage collection, system load, and other random effects), the tests were run in random order and up to 100 times for each data set. The total system execution time and thread execution time were recorded for both data structures. Finally, they were tabulated, compared, and statistical methods were used to determine the better performer.

Algorithm 13 Generate Data set

For parameters N = maximum heap size, M = desired operations, S = current size (initially zero).



Note that a data set is actually a list of operations and values. The testing program could be thought of as an interpreter which simply executed the programs that were generated from Algorithm 13.

4.2 Results

Over random operations sequences of the same size and composition, the Costless-merge pairing heap underperformed. P-values were less than .01 for similar-sized data sets, and were as small as .00147. Thus, the difference is significant, with CMPH run times per given data set averaging higher than PH2p run times for the same set.

All tests were run on an idle 3.2 GHz Pentium IV with 2 GB RAM with Windows XP Pro. JVM options were -Xmx1024 and -Xms1024.

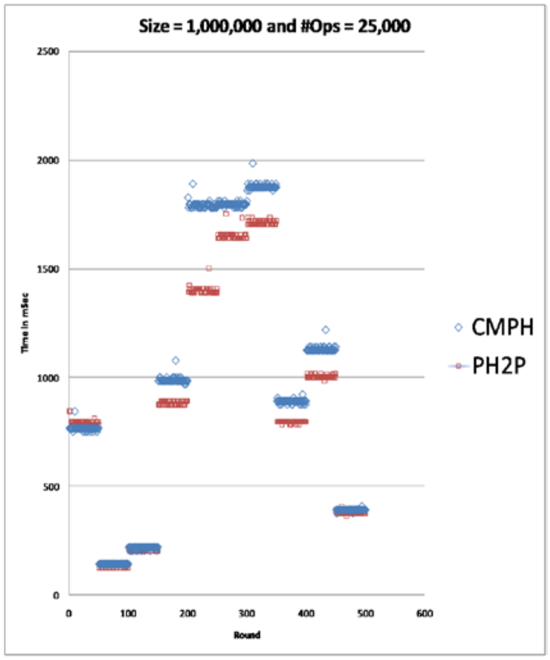


Figure 5: Sample Data set

Ten sequential test results are shown in Figure 5, while the bulk of the data sorted by run time, is shown in Figure 6 on the following page. A test of means was done on run times from each sample operations set using PSPP. No analysis was done on the aggregate data set, as the large variance between data sets would render the results inconclusive. Figure 6 shows what appears to be an exponential increase in run time. However, it should be noted that this merely reflects the composition of the tests, not the asymptotic runtime of the data structures.

5 Conclusions

From 6 on the next page it is apparent that the CMPH under performs. However, careful observation of the chart shows a portion where the CMPH seems to out perform the 2pPH. It should be mentioned that analyzing these specific entries is the equivalent of asking, "Which data structure has lower run time in this range of run times on this system, while ignoring the composition and size of the operations sequence?" The validity of this question is suspect, but the analysis was done anyway (to satisfy curiosity) and revealed no significant

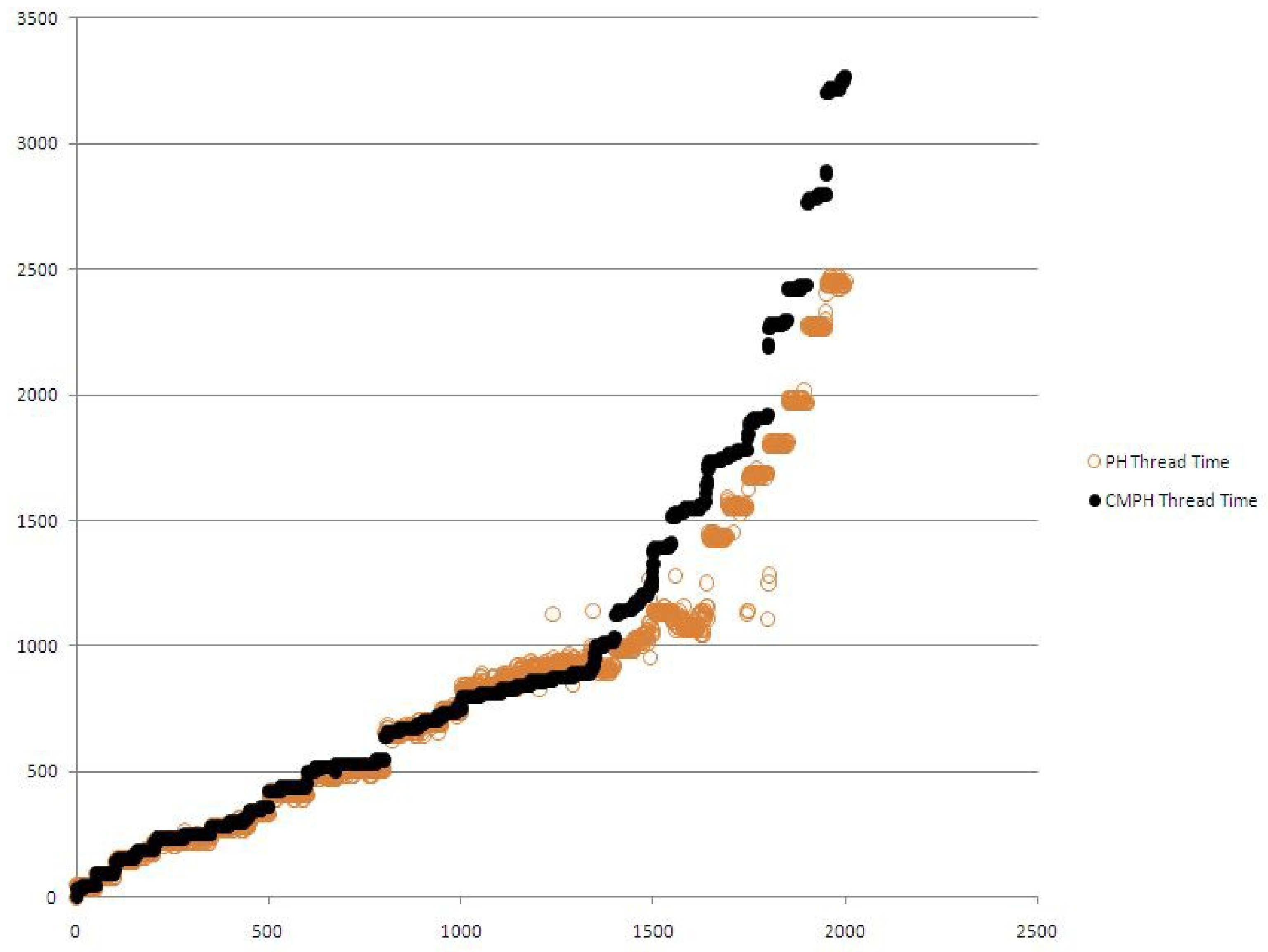


Figure 6: Aggregate Test Results (Runtime vs Test Number)

difference between the means. Where the data was conclusive, it showed the Pairing Heap out performing.

This data could be considered preliminary, and other types of tests could be run. For instance the algorithm used to generate operations could be altered to produce different ratios of *decrease* to *deletemin*. This may yield different performance curves. It is possible that with a higher percentage of *decrease* operations, the overhead of managing the decreased trees separate from the child list of the root might pay off. However, in [11]Fredman showed that the cost of m pairing heap operations with n *decrease* operations has an amortized cost of $O(m\log_{\frac{2m}{n}} n)$. Given the bounds shown in 2.4, and the right value for m as mentioned in [7], a constant *decrease* cost could be should provide additional benefit to both structures.

As an aside, given the motivations for simplicity that made (and keep) the Pairing Heap popular, the Costless-Merge Pairing Heap seems unnecessarily complex, unless some interesting performance increase could be shown.

References

[1] M. AdelsonVelskii and E.M. Landis. An algorithm for the organization of information,. 2006.

[2] Brian Allen and Ian Munro. Self-organizing binary search trees. *J. ACM*, 25(4):526–535, 1978.

[3] Michael Ben-Or. Lower bounds for algebraic computation trees. In *STOC '83: Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 80–86, New York, NY, USA, 1983. ACM.

[4] Mark R. Brown. Implementation and analysis of binomial queue algorithms. *SIAM Journal on Computing*, 7(3):298–319, 1978.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd revised edition edition, September 2001.

[6] Clark Allan Crane. *Linear lists and priority queues as balanced binary trees*. PhD thesis, Stanford, CA, USA, 1972.

[7] Amr Elmasry. Pairing heaps with costless meld. *CoRR*, abs/0903.4130, 2009.

[8] Amr Elmasry. Pairing heaps with o(log log n) decrease cost. In *SODA '09: Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 471–476, Philadelphia, PA, USA, 2009. Society for Industrial and Applied Mathematics.

[9] Robert W. Floyd. Algorithm 245: Treesort. *Commun. ACM*, 7(12):701, 1964.

[10] Michael L. Fredman. Information theoretic implications for pairing heaps. In *STOC '98: Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 319–326, New York, NY, USA, 1998. ACM.

[11] Michael L. Fredman. On the efficiency of pairing heaps and related data structures. *J. ACM*, 46(4):473–501, 1999.

[12] Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert E. Tarjan. The pairing heap: a new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.

[13] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.

[14] Gaston H. Gonnet. Heaps applied to event driven mechanisms. *Commun. ACM*, 19(7):417–418, 1976.

[15] R. L. Graham and Pavol Hell. On the history of the minimum spanning tree problem. *IEEE Ann. Hist. Comput.*, 7(1):43–57, 1985.

[16] Kenneth E. Iverson. *A programming language*. John Wiley & Sons, Inc., New York, NY, USA, 1962.

[17] Douglas W. Jones. An empirical comparison of priority-queue and event-set implementations. *Commun. ACM*, 29(4):300–311, 1986.

[18] D E KNUTH. Optimal binary search trees. In *Acta If: I (197 I*, pages 14–25.

[19] Donald E. Knuth. *The art of computer programming, volume 1 (3rd ed.): fundamental algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[20] Jr. Kruskal, Joseph B. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[21] Bernard M. E. Moret and Henry D. Shapiro. An empirical analysis of algorithms for constructing a minimum spanning tree. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 400–411. Springer, 1991.

[22] Seth Pettie. Towards a final analysis of pairing heaps. *Foundations of Computer Science, Annual IEEE Symposium on*, 0:174–183, 2005.

[23] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technology Journal*, 36:1389–1401, 1957.

[24] Robert Rönngren and Rassul Ayani. A comparative study of parallel and sequential priority queue algorithms. *ACM Trans. Model. Comput. Simul.*, 7(2):157–209, 1997.

[25] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary trees. In *STOC '83: Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 235–245, New York, NY, USA, 1983. ACM.

[26] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.

[27] Daniel Dominic Sleator and Robert Endre Tarjan. Self adjusting heaps. *SIAM J. Comput.*, 15(1):52–69, 1986.

[28] John T. Stasko and Jeffrey Scott Vitter. Pairing heaps: experiments and analysis. *Commun. ACM*, 30(3):234–249, 1987.

[29] Robert Endre Tarjan. *Data structures and network algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1983.

[30] Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.

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

[32] J. W. J. Williams. Algorithm 232: Heapsort. *Communication of the ACM*, 7:347–348, 1964.

Dr. Dean Kelley received his Ph.D in Computer Science from the University of Minnesota. He would say, “Dean Kelley was born, but hasn’t died.” He seems to be interested in data structures, particularly self-optimizing data structures. His students recognize him as an excellent teacher and near-master of the “chalk talk.”

Joshua Vander Hook is a senior in the Computer Science deptartment. He is interested in data structures, algorithms, and the study of artificial intelligence, specifically motion and goal planning for autonomous robotics. He intends to attend the University of Minnesota for graduate studies.