Efficient Merge and Insert Operations for Binary Heaps and Trees

Christopher Lee Kuszmaul[*†]

1 Summary

Binary heaps and binary search trees merge efficiently. We introduce a new amortized analysis that allows us to prove the cost of merging either binary heaps or balanced binary trees is $O(1)$, in the amortized sense. The standard set of other operations (create, insert, delete, extract minimum, in the case of binary heaps, and balanced binary trees, as well as a search operation for balanced binary trees) remain with a cost of $O(\log n)$.

For binary heaps implemented as arrays, we show a new merge algorithm that has a single operation cost for merging two heaps, a and b, of $O(|a| + \min(\log|b| \log\log|b|, \log|a| \log|b|))$. This is an improvement over $O(|a| + \log|a| \log|b|)$ [11].

The cost of the new merge is so low that it can be used in a new structure which we call shadow heaps, to implement the insert operation to a tunable efficiency. Shadow heaps support the insert operation for simple priority queues in an amortized time of $O(f(n))$ and other operations in time $O((\log n \log\log n)/f(n))$, where $1 \leq f(n) \leq \log\log n$.

More generally, the results here show that any data structure with operations that change its size by at most one, with the exception of a merge (aka meld) operation, can efficiently amortize the cost of the merge under conditions that are true for most implementations of binary heaps and search trees.

2 Introduction

A binary heap is a tree structure where a given node satisfies the *heap property*, the key for a node is smaller than the key of the children of the node. A binary heap can be implemented as a contiguous array, A where

[*]MRJ Technology Solutions
[†]fyodorr@nas.nasa.gov

we define the root as $A[1]$, the left child of $A[i]$ as $A[2i]$ and the right child as $A[2i+1]$.

A binary search tree satisfies the *search tree property* where a given node's key is larger than that of its left child and smaller than that of its right child. A binary search tree is virtually always implemented using pointers, but may have a variety of constraints on how the tree is composed. In this paper we concern ourselves with balanced binary search trees, which satisfy the additional requirements[3].

Binary heaps and binary search trees are data structures that allow the efficient implementation of certain operations that are valuable in a variety of applications [4].

Binary heaps efficiently support:

- `create`(H): Create an empty structure called H.
- `insert`(x, H): Insert key x into H.
- `delete`(p, H): Delete key pointed at by p from H.
- `extract_min`(H): Remove and return the key in H with smallest value.

Binary search trees, in addition to the above, efficiently support:

- `search`(k, H): Locate an element of H with a key value of k.
- `next`(p, H): Given a pointer p to a key in H, find the next larger key in H.
- `prev`(p, H): Given a pointer p to a key in H, find the next smaller key in H.

Each of the operations can be supported efficiently ($O(\log |H|)$ steps) by the proper choice of implementation of a binary heap, or else by a binary search tree. Now consider the *merge* operation.

- Merge(H_1, H_2): Return a structure with the contents of H_1 and H_2 combined, destroying H_1 and H_2.

Merge is not supported efficiently for binary heaps implemented using arrays, nor for binary search trees. Note that the *join* operation seen in [12], only supports binary search trees where the maximum key in one is less than the minimum key in the other.

However, as we will show, merge is supported efficiently in the amortized sense. In the remainder of this paper we discuss the memory allocation issues in choosing between a pointer based and an array based heap (section 3) what we mean by amortized analysis (section 4), and the requirements for a merge to be efficient in an amortized sense (section 5). We will also see in section 5 that binary heaps implemented as arrays and balanced binary search trees are efficient in the amortized sense for merging. We will also introduce in section 6 a new merge algorithm for binary heaps implemented as arrays, with an efficiency that exceeds that found in [11]. We call this new merge the 'median shadow merge'. Next, we will show how to implement insert using merge by making a slight modification to the binary heap data structure (section 7). This insert has a very low amortized cost, which can be balanced against the efficiency of the other operations that heaps support. We discuss the tradeoffs between the cost of insertion and other operations in section 8.

3 On memory Allocation

An alternative to the array based implementation of a binary heap is to employ a pointer based method, where a node stores the memory addresses of its children. The advantage of this alternative is flexibility of storage of the heap, which makes it easier to prove efficiency. Also, a pointer based heap avoids deallocation and reallocation of the scale called for by the array based method, and so may avoid fragmentation problems.

The array based method uses less memory (as a lower bound), and the memory used for each heap is contiguous, each of these features can improve performance on cache based computer architectures. The array based method also tends to free large regions of memory during deallocation, rather than isolated words as may happen in the deallocation of a single node of a pointer based heap. As such, in some scenarios, the array based method may produce less fragmentation than the pointer based method.

Which method is more efficient with respect to memory allocation thus remains an open question beyond the scope of this paper. Certainly, we know that without garbage collection, the proportion of memory that can be wasted is no more than $O(\log n)$, [15] and with garbage collection, the proportion can be made arbitrarily small [1]. In any case, it is certainly worth while to find the best possible uses of array based method as well as the pointer based method.

4 On Amortized Analysis

To show that the amortized cost of a given set of operations is $O(f(n))$, we must show that the *worst case* cost of an entire sequence of n operations never exceeds $O(nf(n))$. For simple data structures, this can be done directly by considering all possible combinations of operations, and identifying the most expensive sequence. Usually, such an analysis shows that expensive operations must be preceded by a large number of inexpensive operations. However the complex interrelationships between the operations considered in this paper make it difficult to prove that expensive operations are inherently infrequent.

We use *potential functions* to simplify the analysis. A potential function's value depends on parameters that describe the state of a data structure. The *change* in the value of the potential function after an operation corresponds to (and in some sense offsets) the cost of the operation in question. If an inexpensive operation results in an increase of the potential function, but the increase of the potential function is within a constant factor of the actual cost of the operation, then the amortized cost of the operation is unchanged. Meanwhile, if an expensive operation results in a decrease of the potential function that offsets the cost of the expensive operation, then the amortized cost of the expensive operation may be small. For such an analysis to remain valid, the potential function must stay nonnegative, and begin with a value of zero. For more on amortized analysis and its origins see [13].

5 How to get an efficient merge

Consider the following potential function:

$\Psi = (\sum_{i=0}^{N} |H_i|)(\log(\sum_{i=0}^{N} |H_i|)) - \sum_{i=0}^{N} |H_i| \log |H_i|$

Where H_i is the i'th heap in the set of all heaps that have been created, and there are N heaps. On operations that change the size of a heap by one or less, Ψ changes by $O(\log N)$. Thus the amortized cost of all binary heap operations other than merge is $O(\log N)$. A merge of two heaps, a and b results in a change of Ψ equal to

$(|a| + |b|) \log(|a| + |b|) - |a| \log |a| - |b| \log |b|$.

Any merge with an actual cost within a constant factor of this has an amortized cost of zero. We use the series expansion of an increment for log and can conclude the change of Ψ is at least $|a| \log(|b| \,/\, |a|)$. Throughout this paper we will assume $|a| \leq |b|$.

For binary heaps implemented as arrays, [11] has established a time to

merge heap a and heap b in $O(|a| + \log|a| \log|b|)$ steps. Since this is less than the drop of Ψ due to the merge, the amortized cost for a merge is $O(1)$.

For balanced binary search trees, [3] has established a time to merge tree a with tree b in $O(|a| \log(|b| / |a|))$ steps, which is within a constant factor of the change of Ψ, so again the amortized cost for the merge is $O(1)$.

6 The shadow merge algorithm

Now we present a new algorithm for merging binary heaps implemented as arrays. The cost of this merge is $O(|a| + \min(\log|b| \log\log|b|, \log|b| \log|a|))$.

The classic heapify algorithm takes an unordered array and makes it into a heap in time proportional to the size of the array [4]. A naive merge would concatenate two heaps to be merged, and then run the make heap operation found in [4] on the resulting array.

```
naive_merge(a,b)
  c = concatenate(b,a);
  return(make_heap(c));
```

The concatenate can be viewed as a sequence of table inserts, which has an amortized cost of $O(|a|)$ [4], so the cost of the naive merge is $O(|b|)$, since the make heap dominates the cost of the naive merge. As observed in [11], this naive merge does not take advantage of the inherent structure of the heaps that exist prior to the merge.

Now consider c, the concatenation of b and a, as seen in the naive merge code. If we view c as a heap, the only nodes in c that may not satisfy the heap property (or whose descendents may not satisfy the heap property) are the ancestors of the final $|a|$ nodes. These ancestors are what we refer to as the 'shadow' of a.

There are not very many nodes in the shadow. We will show that the number of such nodes is $O(|a| + \log(|b|))$

The concatenated a occupies indices in c in the range $[|b| + 1 : |b| + |a|]$ inclusive. The parents of these nodes occupy the range $[(|b| + 1)/2 : (|b| + |a|)/2]$ where division rounds downward. The parents of those nodes are computed similarly until the root node of the heap is reached. The number of nodes then, that may have a non heap as a descendent is $O(\sum_{i=0}^{\log(|a|+|b|)} 2 + (|a|)/2^i)$ (Except for at most two isolated nodes at each end of the range,

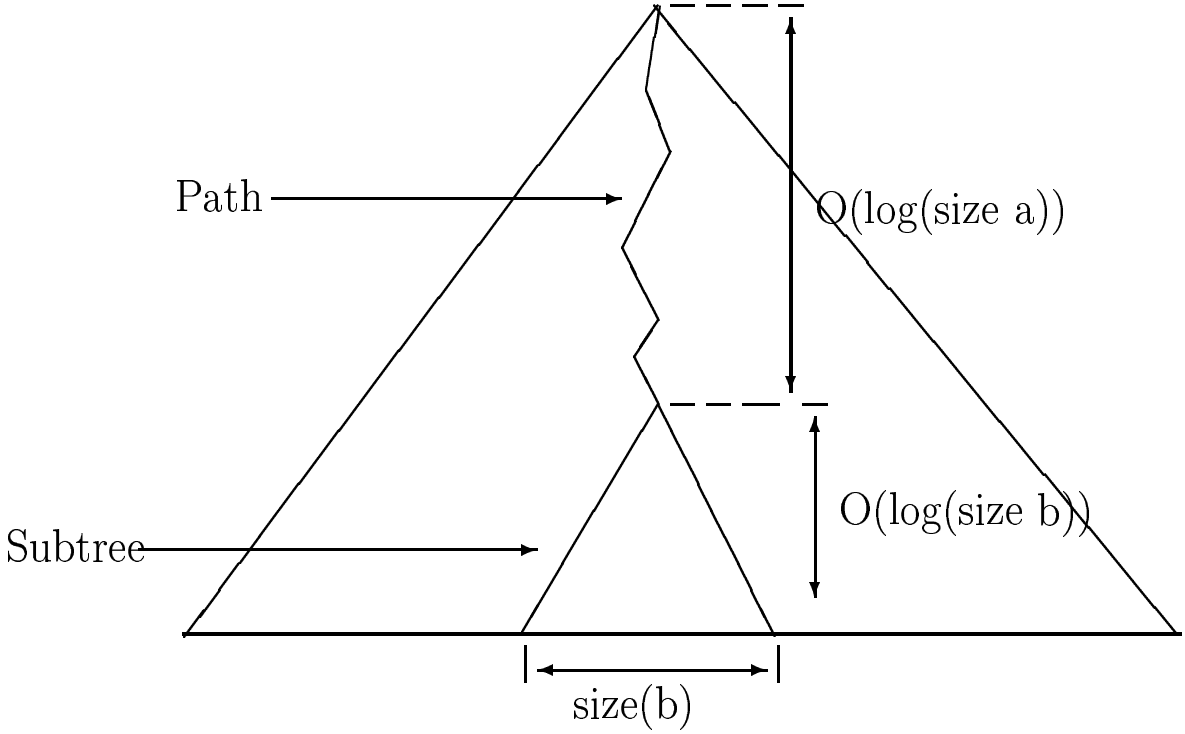


Figure 1: The concatenated heap, and the two parts of its shadow: the path and the subtree. The large triangle represents the original heap, b. The leaf nodes at the base of the small triangle represent the heap to be merged, a, and so there are $|a|$ nodes at the base of the small triangle. The small triangle and the zig zagging line leading to the top of the large triangle represent the ancestors of the leaf nodes at the base of the small triangle - the shadow of a.

every node that is a potential ancestor of a non heap has a similar sibling). This sum is $O(|a| + \log(|b|))$ QED.

Let us divide the shadow into two parts. The first part, which we will call the path, is the set of nodes such that the number of nodes at any given depth in heap c is at most 2. The second part, which we will call the subtree, is the remainder of the shadow, and is characterized by the fact that the number of nodes in the shadow at a depth d is at most 3/4 the number of nodes at a depth $d + 1$. The size of the subtree is $O(|a|)$ while the size of the path is $O(\log(|b|))$.

The shadow merge algorithm involves two steps. First it extracts the smallest |path| nodes from the shadow, in sorted order. These nodes replace

the nodes that originally are in the path. Because the nodes are sorted, we can guarantee that for any two nodes with index i and j in the path, if $i < j$ then $c[i] < c[j]$. This ensures that the heap property is preserved in the path, and that the largest element in the path is smaller or equal to the smallest element in the subtree. The second step is to ensure the heap property is satisfied in the subtree - this ensures that the entire array c is a heap.

6.1 Extract and Sort |path| nodes

Because the path has at most two nodes at a given level in b, it can be transformed into a sorted array in $O(\log |b|)$ steps. The subtree can be transformed into a single heap (still isolated from b) in $O(|a|)$ steps. The task is then reduced to replacing any elements in a sorted array that are larger than any elements in a heap, and to keep the array sorted.

If the sorted array is larger in size than the heap, then we begin at the smallest element, e, in the sorted array. e is inserted into the heap, and the smallest element of the heap then replaces e. This step can be optimized if e is smaller than the smallest element in the heap.

If the sorted array is smaller in size than the heap, then we extract the |path| smallest elements of the heap, using the standard algorithm on page 187 of [4]. We then sort this result and combine it with the sorted array to obtain the desired result.

```
combine(sortedarray, heap)
1  if(size(sortedarray) > size(heap))
2    for(i = 1; i <= size(sortedarray); i = i+1)
3         insert(sortedarray[i], heap);
4         sortedarray[i] = extract_min(heap);
5  else
6    heaptop = extract_k_smallest(heap, size(sortedarray));
7    sort(heaptop);
8    sortedarray = merge_sorted_lists(heaptop, sortedarray);
```

Lines 1,and 5 obviously have cost $O(1)$. The loop specified at line 2 iterates $O(\log |b|)$ times, and each of lines 3 and 4 in the loop require $O(\log |a|)$ steps. Thus, if the heap is smaller than the sorted array, the cost of the

combine is $O((\log |b|)(\log |a|))$. This case leads to the same complexity as is found in [11].

If the heap is larger than the sorted array, then we extract the smallest $O(\log |b|)$ elements from the heap in $O(|a|)$ steps (line 6). We then sort these elements in $O(\log |b| \log \log |b|)$ steps (line 7). Finally we merge the sorted lists (line 8) in $O(\log |b|)$ steps. This case thus has a cost of $O(|a| + \log |b| \log \log |b|)$. A practical code would probably avoid this case unless the heap was much larger than the sorted array, but the inclusion of this case allows us to establish a more efficient asymptotic performance for insertion using merge.

All that now remains is to enforce the heap property on the remainder of the shadow — the subtree.

6.2 Heapify the subtree

To heapify the subtree we operate very similarly as the standard build heap algorithm. We use the heapify subroutine of [4].

```
subtreeheapify(heap, start,  end)
  if start + 3 > end, return
  For each parent of each node in the range [start:end]:
      heapify that parent node
  subtreeheapify(heap, start/2, end/2)
```

Subtreeheapify makes it the case that every node in the subtree satisfies the heap property, provided the set of nodes in the range passed to it are roots of heaps. The termination condition guarantees that every node in the subtree is heapified.

The proof that subtreeheapify is correct and runs in time $O(end - start)$ is virtually identical to the corresponding proof for the build heap algorithm in [4]. The time to build a heap out of the subtree is $O(|a|)$ steps, and is achieved by calling:

```
subtreeheapify(c, size(b), size(b) + size(a));
```

After having concatenated a and b into c.

7 Shadow Heaps

Our new data structure, which we dub a shadow heap, is an array that satisfies the heap property for the first *heapsize* indices, and does not for

the next *tablesize* indices. This can be viewed as a heap adjacent to an unordered table. On insertion, the new element is placed into the table. If the table is too large, we make it into a heap and merge it with the rest of the structure.

```
insert(key, structure)
   table_insert(key, structure.table);
   if (tablesize > threshold)
      make_heap(structure.table);
      merge(structure.heap, structure.table);
```

The other operations besides insert are modified in an obvious way, with an added cost proportional to *tablesize*. The value of *threshold* is modified after each operation to conform to some function of *heapsize*. In the case of the insert, the larger *threshold* is, the lower the amortized cost. In the next section, we calculate this cost in detail.

8 The amortized efficiency of shadow heaps

We will show the amortized efficiency of creation, insertion, deletion, deletion of the minimum, finding the minimum, reducing the value of a key, and merging two shadow heaps.

For each shadow heap,H, we employ the following potential function.

$\Psi(H) = \text{tablesize}(1+\min(\log|H|\log\log|H|, \log|H|\log\text{tablesize})/\text{threshold})$

When tablesize = threshold, Ψ equals the cost of a merge of a heap of size $|H|$ with a heap of size threshold. For purposes of our analysis, we assume that *threshold* is a 'well behaved' function of $|H|$ — in particular, since *threshold* is smaller than $|H|$, we assume that any change in $|H|$ results in no larger a change in *threshold*.

- On creation, Ψ does not change, so the amortized cost is the actual cost: $O(1)$.
- On insertion, we have two cases to consider: one when the merge is employed, and the other where it is not. If the merge is done, the actual cost is exactly equal to the drop in the potential function, so the amortized cost is $O(1)$. If the merge is not done, then the actual cost is $O(1)$, while the potential function increases by $O(1 + \min(\log|H|\log\log|H|, \log|H|\log\text{tablesize})/\text{threshold})$.

- On deletion, deletion of the minimum, and finding the minimum the actual cost is $O(\text{tablesize} + \log |H|)$, and the potential function will increase only if the value of *threshold* shrinks. But *threshold* will shrink at most by one on these operations, and since *tablesize* is at most as large as *threshold*, the potential function will grow by $\Omega(\log |H|)$, so the actual cost is the amortized cost.

- To reduce the value of a key (where we know where the node for the key is), the potential function does not change. The amortized cost is thus the actual cost of $O(\log |H|)$.

- To merge two shadow heaps, a and b we can insert the $|a|$ elements of a into b, with the corresponding amortized cost.

We can see that the amortized cost of these operations depends on threshold. The interesting range is when $\log |H| \leq \text{threshold} \leq \log |H| \log\log |H|$. A merge of heap a and heap b costs $O(|a|)$ insertions, the cost of an insert is $O(\log |H| \log\log |H| \,/\, \text{threshold})$, and the other operations cost $O(\text{threshold})$. As such, one can select the efficiency of the insert operation to match its frequency, so in some applications one can have insertion cost of $O(1)$ while retaining a cost of $O(\log |H|)$ for the other operations.

9 Summary and Conclusion

Shadow heaps can support insert operations with an efficiency between $O(1)$ and $O(\log\log |H|)$, where the product of the cost of an insert and the cost of other priority queue operations is $O(\log |H| \log\log |H|)$. This efficient insert may be applicable to variations of Prim's algorithm, or in the derivation of an efficient operation that reduces the key of a node in a heap.

In general, simple data structures like binary heaps implemented as arrays, and balanced binary search trees merge efficiently, in the amortized sense. We feel splay trees also can also merge efficiently using a recursive algorithm that splits one tree using the root of the other, but we cannot prove it. Further, we suspect that a reduce_key operation can be made to perform in O(1) steps using shadow heaps, or similar simple structures.

References

[1] Y. Bekkers, J Cohen editors International Workshop on Memory Management number 637 in Lecture Notes in Computer Science, St. Malo, France, September 1992 Springer-Verlag.

[2] M. R. Brown, 'Implementation and Analysis of Binomial Queue Algorithms'. *SIAM Journal on Computing* 7(3):298-319, 1978

[3] M.R. Brown, R.E. Tarjan, 'A Fast Merging Algorithm'. *Journal of the ACM*, 26(2):211-226, 1979

[4] T. Cormen, C. Leiserson, R. Rivest *Introduction to Algorithms* MIT Press, Cambridge Massachusetts.

[5] R. W. Floyd. Algorithm 245 (TREESORT). *Communications of the ACM*, 7:701, 1964.

[6] M. L. Fredman, R. E. Tarjan. 'Fibonacci heaps and their uses in improved network

optimization algorithms.' *Journal of the ACM*, 34(3):596-615, 1987.

[7] D.W. Jones, 'An Empirical-Comparison of Priority-Queue and Event-Set Implementations'. *Communications of the ACM*, 29(4):300-311, 1986

[8] C.L. Kuszmaul, *Amortized Analysis of Binary Heaps*, Masters Thesis, Department of Computer Science, University of Illinois, 1995

[9] C.L. Kuszmaul, *Splay Heaps Merge Efficiently*, Submitted to IEEE Transactions on Computing, May 1997.

[10] R.C. Prim. 'Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389-1401, 1957.

[11] J. Sack, T. Strothotte. 'An Algorithm for Merging Heaps'. *Acta Informatica* 22(2):171-186, 1985

[12] D.D. Sleator, R.E. Tarjan. 'Self-Adjusting Binary Search Trees'. *Journal of the ACM*, 32(3):652-686, 1985

[13] R. E. Tarjan. 'Amortized computational complexity'. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306-318, 1985.

[14] J. W. J. Williams. Algorithm 232 (heapsort). *Communications of the ACM*, 7:378-348, 1964.

[15] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic Storage Allocation: A Survey and Critical Review Proceedings. 1995 International Workshop on Memory Management, Kinrose Scotland, UK, September 27-29 1995, Springer-Verlag LNCS.