An Amortized Analysis of Binary Heaps and Binomial Heaps

Natthapong Jungteerapanich

April 17, 2015

1 Amortized analysis of binary heaps

Here we describe an amortized analysis of three important operations on binary min heaps, namely, INSERT, EXTRACT-MIN, and FIND-MIN. In our amortized analysis, we assume that, starting from an empty heap, a sequence of these three operations are performed. We then propose and prove an amortized cost for each of these operations. We demonstrate the analysis using both the accounting method and the potential method.

Cost model. To simplify our analysis, when we analyze the time complexity of each operation, instead of measuring the running time (in seconds or in number of steps) the operation takes, we shall measure the number of times the following *basic operations* are executed:

- a new node is created;
- two nodes are exchanged positions in the heap.

We stipulate that each execution of one of the basic operations above costs 1 *credit*. Note that you may choose the basic operations other than those two above. The reason we select these two basic operations is because the running time of the three heap operations that we are analyzing is linearly proportional to the number of times these two basic operations are executed.

Actual costs. Let's analyse the cost in credits of the operations INSERT, EXTRACT-MIN, and FIND-MIN.

- INSERT(H, x): This involves creating a new node, appending it as the last node of the heap, and performing the bubble-up operation to maintain heap-orderedness. Creating a new node costs 1 credit. Suppose the INSERT operation is carried out on a heap H with n nodes, where $n \geq 0$. After adding the new node, the tree height would be $\lfloor lg(n+1) \rfloor$. Consequently, the bubble-up operation requires at most $\lfloor lg(n+1) \rfloor$ exchanges of nodes in the heap. Therefore, the cost of INSERT is no greater than $1 + \lfloor lg(n+1) \rfloor$ credits.

- EXTRACT-MIN(H): This involves exchanging the root node of heap H with the last node of the heap, then removing the last node, and then performing the sink-down operation to maintain heap-orderedness. Suppose before this operation, H contains n nodes (where $n > 0$). Exchanging the root node with the last node costs 1 credit. The height of the heap after removing the last node is $\lfloor lg(n-1) \rfloor$. Consequently, the sink-down operation requires at most $\lfloor lg(n-1) \rfloor$ exchanges of nodes in the heap. Therefore, the cost of EXTRACT-MIN is no greater than $1 + \lfloor lg(n-1) \rfloor$ credits.

- FIND-MIN(H): For this operation, we only need to return the value stored in the root of H. No basic operation is executed. Therefore, the cost is 0 credits.

1.1 Accounting method

Recall that by the accounting method, we assign an amortized cost to each operation so that, in any sequence of operations, the sum of the actual costs of the operations is no greater than the sum of the amortized costs of the operations in the sequence. This is to ensure that the amortized costs assigned do indeed provide an upper bound on the actual cost of any sequence of operations.

Precisely, consider a sequence of operations on heaps, starting from an empty heap H_0:

$$H_0 \xrightarrow{f_1} H_1 \xrightarrow{f_2} H_2 \xrightarrow{f_3} \ldots$$

Suppose the actual cost of each operation f_i (where $i \geq 1$) on heap H_{i-1} is c_i, and the amortized cost assigned to this operation is $\hat{c}_i$. Then the following constraint must hold for any $k \geq 1$:

$$\sum_{i=1}^{k} c_i \leq \sum_{i=1}^{k} \hat{c}_i,$$

or, equivalently,

$$\sum_{i=1}^{k} \hat{c}_i - \sum_{i=1}^{k} c_i \geq 0.$$

An intuitive way to understand this method is as follows. For each operation on the data structure, we charge an amortized cost $\hat{c}$. If $\hat{c}$ is greater than the actual cost c of the operation, we imagine that the extra credits are stored within the data structure and can be used at some later time. If the actual cost c exceeds the amortized cost $\hat{c}$, we will have to use the credits previously stored in the data structure to cover the cost of the operation. The term $\sum_{i=1}^{k} \hat{c}_i - \sum_{i=1}^{k} c_i$ in the constraint above can be seen as the total amount of credits that are stored in the data structure after k operations.

In our analysis of binary heaps below, we imagine that the extra credits after each operation are stored in the nodes of the heap. To ensure that the accounting constraint above is satisfied, we will charge an amortized cost to each operation so that the actual cost of the operation is covered (possibly using the credits previously stored in the heap) while maintaining the following invariant:

Each node x in the heap stores at least depth(x) credits.

depth(x) denotes the depth of node x in the tree, defined recursively as follows:

- the depth of the root node is 0;
- the depth of a non-root node is 1+ the depth of its parent node.

Clearly, since the depth of any node is non-negative, the above invariant guarantees that the amount of credits stored in the heap never becomes negative.

The table below summarizes the amortized cost we assign to each operation.

Operation	Upper bound on actual cost	Amortized cost
INSERT(H, x)	$1 + \lfloor lg(n+1) \rfloor$	$1 + 2\lfloor lg(n+1) \rfloor = O(log(n))$
EXTRACT-MIN(H)	$1 + \lfloor lg(n-1) \rfloor$	$1 = O(1)$
FIND-MIN(H)	0	$0 = O(1)$

Below is an explanation of why these amortized costs are sufficient to cover the actual costs and maintain the invariant.

- INSERT(H, x): We assign the amortized cost of $1 + 2\lfloor lg(n+1) \rfloor$, where n is the number of nodes in heap H. This covers the actual cost of inserting the node, with the leftover credits of at least $\lfloor lg(n+1) \rfloor$ which we store in the new node. Clearly, after insertion, the depth of the new node will be no greater than the height, which is $\lfloor lg(n+1) \rfloor$. Therefore, the invariant is maintained.

- EXTRACT-MIN(H): We assign the amortized cost of 1 to this operation. Let's see why this is sufficient. The actual cost of this operation is no greater than $1 + \lfloor lg(n-1) \rfloor$. Since we remove the last node from the heap, we obtain at least $\lfloor lg(n) \rfloor$ credits that were stored in the node. Together with the amortized cost of 1, we have at least $1 + \lfloor lg(n) \rfloor$ credits available to perform this operation, which can clearly cover the cost. Since we are not adding any new node, it is clear that the invariant is still maintained after this operation.

- FIND-MIN(H): We can assign the amortized cost of 0 for this operation as the actual cost of this operation is 0. Since we are not adding any new node, it is obvious that the invariant is still maintained.

1.2 Potential method

In the potential method, instead of directly assigning an amortized cost to each operation, a function on the data-structure instances, called a *potential function*, is defined. Precisely, a potential function Φ maps a real value to each data-structure instance such that:

- $\Phi(D_0) = 0$, where D_0 is an *initial data-structure instance* (such as the empty heap, the empty list, etc.);

- $\Phi(D) \geq 0$, for any data-structure instance D.

Then the amortized cost of an operation f on a data-structure instance D is defined as the actual cost of the operation plus the increase of the potential of the data-structure instance after the operation is applied. Precisely, suppose c is the actual cost of the operation f on a data-structure instance D and D' is the data structure after applying f. Then the amortized cost $\hat{c}$ is given by

$$\hat{c} = c + (\Phi(D') - \Phi(D)).$$

It can be easily shown that the amortized cost defined in this way will provide an upper bound on the actual cost of a sequence of operations. Consider a sequence of operations applied to an initial data structure D_0:

$$D_0 \xrightarrow{f_1} D_1 \xrightarrow{f_2} D_2 \xrightarrow{f_3} ...$$

Suppose the actual cost of each operation f_i on the data-structure instance D_{i-1} is c_i and $\hat{c}_i$ is the amortized cost of this operation defined above. Then, for any $k \geq 1$,

$$\begin{aligned}
\sum_{i=1}^{k} \hat{c}_i &= \sum_{i=1}^{k}(c_i + (\Phi(D_i) - \Phi(D_{i-1}))) \\
&= (\sum_{i=1}^{k} c_i) + (\sum_{i=1}^{k}(\Phi(D_i) - \Phi(D_{i-1}))) \\
&= (\sum_{i=1}^{k} c_i) + (\Phi(D_k) - \Phi(D_0)) \\
&\geq \sum_{i=1}^{k} c_i \qquad \text{since } \Phi(D_k) \geq 0 \text{ and } \Phi(D_0) = 0.
\end{aligned}$$

For our analysis of binary heaps, we define the potential function as follows:

$$\Phi(H) = \sum_{\text{node } x \text{ in } H} \text{depth}(x).$$

We consider the amortized cost of each operation. For each operation considered below, H and H' are the heaps before and after applying the operation, respectively. n is the number of nodes in heap H. c and $\hat{c}$ are the actual cost and the amortized cost of the operation, respectively.

- INSERT(H, x): The actual cost of this operation is at most $1 + \lfloor lg(n+1) \rfloor$, where n is the number of nodes in heap H. This operation adds a new node, appended as the last node of the heap. Thus, the potential of the heap increases by the depth of the new node, which is $\lfloor lg(n+1) \rfloor$. Precisely,

$$\begin{aligned}
\hat{c} &= c + (\Phi(H') - \Phi(H)) \\
&\leq (1 + \lfloor lg(n+1) \rfloor) + \lfloor lg(n+1) \rfloor \\
&= 1 + 2 \lfloor lg(n+1) \rfloor .
\end{aligned}$$

  Therefore, the amortized cost of inserting an element into a binary heap with n nodes is $O(log(n))$.

- EXTRACT-MIN(H): The actual cost of this operation is at most $1 + \lfloor lg(n-1) \rfloor$. After applying this operation, the last node is moved to replace the root. Hence, the potential of the heap decreases by the depth of the last node, which is $\lfloor lg(n) \rfloor$. Precisely,

$$\begin{aligned} \hat{c} &= c + (\Phi(H') - \Phi(H)) \\ &\le (1 + \lfloor lg(n-1) \rfloor) - \lfloor lg(n) \rfloor \\ &\le 1 \end{aligned}$$

  Therefore, the amortized cost of extracting a minimum element from a binary heap is $O(1)$.

- FIND-MIN(H): The actual cost of this operation is 0 and the heap is unchanged. Hence,

$$\begin{aligned} \hat{c} &= c + (\Phi(H') - \Phi(H)) \\ &= 0 + 0 = 0 \end{aligned}$$

  Therefore, the amortized cost of finding a minimum element in a binary heap is $O(1)$.

2 Amortized analysis of binomial heaps

Next we look at an amortized analysis of binomial heaps. In particular, we shall analyze the amortized running time of these operations: MAKE-HEAP, MELD, INSERT, EXTRACT-MIN, and DECREASE-KEY. We leave the analysis of other operations as exercises to the reader. In our amortized analysis, we consider a sequence of *sets of binomial heaps*:

$$S_0 \xrightarrow{f_1} S_1 \xrightarrow{f_2} S_2 \xrightarrow{f_3} ...,$$

where S_0 is the empty set. From each set S_{i-1} of binomial heaps,

- if f_i is the MAKE-HEAP operation, a new empty heap is created and added to the set;
- if f_i is a MELD operation, two heaps are removed from the set S_{i-1} and their union is added to the set;
- if f_i is an INSERT operation, a new element is inserted to one of the heaps in the set;
- if f_i is an EXTRACT-MIN operation, the minimum element is extracted from one of the heaps in the set S_{i-1};
- if f_i is a DECREASE-KEY operation, the key of a node in one of the heaps in the set S_{i-1} is decremented to the given value.

As before, we demonstrate the analysis using both the accounting method and the potential method.

Cost model. In our analysis, we shall measure the number of times the following *basic operations* are executed:

- a new empty heap is created;
- a new node or a new binomial tree is created;
- two nodes in a binomial heap are exchanged positions;
- two binomial trees in a binomial heap are merged to form a new binomial tree.

Each execution of one of the basic operations above costs 1 credit.

Actual costs. Let's analyse the cost in credits of the operations MAKE-HEAP, MELD, INSERT, EXTRACT-MIN, and DECREASE-KEY.

- MAKE-HEAP(): This involves creating a new empty heap. Hence the actual cost is 1.

- MELD(H_1, H_2): This involves a sequence of binomial tree merging, analogous to binary addition. Suppose H_1 and H_2 contain n_1 and n_2 nodes, respectively. Then, from the basic properties of binomial trees and binomial heaps, we know that H_1 contains at most $\lfloor lg(n_1) \rfloor + 1$ binomial trees, and similarly for H_2. The number of trees merging involved in melding the two heaps, which is the cost of this operation, is at most $\lfloor lg(n_1) \rfloor + \lfloor lg(n_2) \rfloor + 1$.

- INSERT(H, x): This operation consists of the following steps:

  1. Make a new empty heap H_0;
  2. Insert x into H_0;
  3. Meld H and H_0.

  Steps 1 and 2 each costs 1 credit. The number of tree merging in the melding operation in Step 3 is at most $\lfloor lg(n) \rfloor + 1$, where n is the number of nodes in heap H. Therefore, the total cost of this operation is at most $\lfloor lg(n) \rfloor + 3$.

- EXTRACT-MIN(H): This operation consists of the following steps:

  1. Remove a binomial tree T in heap H with a minimum root (this can be easily done if we assume the heap always keeps a pointer to a binomial tree with a minimum root); call the resulting heap H';
  2. Remove the root of T, thus obtaining a bunch of k smaller binomial trees $T_1, T_2, ..., T_k$, where k is the order of the binomial tree T;
  3. Form a new heap H_T from the binomial trees $T_1, T_2, ..., T_k$;
  4. Meld H' with H_T.

  Steps 1-2 cost nothing. Step 3 involves creating a new heap, thus costing 1 credit. Step 4 involves melding the heap H', which contains $n - n_T$ nodes (where n is the number nodes in H and n_T is the number of nodes in tree T), with the heap H_T which contains $n_T - 1$ nodes. Thus the number of tree merging required is at most $\lfloor lg(n - n_T) \rfloor + \lfloor lg(n_T - 1) \rfloor + 1 \leq 2 \lfloor lg(n-1) \rfloor + 1$. Therefore, the total cost of this operation is at most $2 \lfloor lg(n-1) \rfloor + 2$.

- DECREASE-KEY(H, x, k): This involves decreasing the key of the given node x in heap H to k. After decreasing the key of x, a bubble-up operation is required to maintain heap-orderedness. The number of node exchanges required is bounded by the height of heap H, which is at most $\lfloor lg(n) \rfloor$, where n is the number of nodes of H. Thus, the cost of this operation is at most $\lfloor lg(n) \rfloor$.

2.1 Accounting method

We shall charge an amortized cost to each operation to cover the actual cost of the operation (also possibly using the credits previously stored in the heap) and maintain the following invariant:

Each binomial tree in each binomial heap stores at least 1 credit.

The table below summarizes the amortized cost we assign to each operation.

Operation	Upper bound on actual cost	Amortized cost
MAKE-HEAP()	1	$1 = O(1)$
MELD(H_1, H_2)	$\lfloor lg(n_1) \rfloor + \lfloor lg(n_2) \rfloor + 1$	$0 = O(1)$
INSERT(H, x)	$\lfloor lg(n) \rfloor + 3$	$3 = O(1)$
EXTRACT-MIN(H)	$2 \lfloor lg(n-1) \rfloor + 2$	$\lfloor lg(n) \rfloor = O(log(n))$
DECREASE-KEY(H,x,k)	$\lfloor lg(n) \rfloor$	$\lfloor lg(n) \rfloor = O(log(n))$

Below is an explanation of why these amortized costs are sufficient to cover the actual costs and maintain the invariant.

- MAKE-HEAP(): We assign the amortized cost to be equal to the actual cost, which is 1 credit. The invariant still holds as the new heap created is empty.

- MELD(H_1, H_2): We assign the amortized cost of 0. Recall that the cost of melding arises from merging binomial trees. Since each binomial tree stores 1 credit, when merging two trees, the credit from one tree covers the cost of merging and the credit from the other tree is passed on to stored in the new tree resulted from merging. Therefore, the cost of melding is covered by the stored credits alone and the invariant is maintained without requiring any additional charge.

- INSERT(H, x): We assign the amortized cost of 3. Of these 3 credits, one credit covers the creation of a new heap; another credit covers the creation of a new binomial tree for the new element; and the last credit is stored in the new binomial tree. The melding of the new heap and heap H can be done without requiring additional credits.

- EXTRACT-MIN(H): We assign the amortized cost of $\lfloor lg(n) \rfloor$ to this operation, where n is the number of nodes in heap H. Recall that the EXTRACT-MIN operation involves taking out a binomial tree T with a minimum root. We get one credit that was stored in the tree T. Then we remove the root from T to obtain $T_1, T_2, ..., T_k$ smaller binomial trees, where k is the order of the binomial tree T, which is at most $\lfloor lg(n) \rfloor$. We need to store 1 credit to each of these trees, thus requiring at most $\lfloor lg(n) \rfloor$ credits. Another one credit is required to create a new heap H_T to contain the trees $T_1, T_2, ..., T_k$. The last step which involves melding the new heap H_T with the old heap does not require extra credits. Therefore, the net credits required are at most $\lfloor lg(n) \rfloor + 1 - 1 = \lfloor lg(n) \rfloor$ credits, which can be covered by the amortized charge.

- DECREASE-KEY(H,x,k): We assign the amortized cost of $\lfloor lg(n) \rfloor$ to this operation, where n is the number of nodes in heap H. This is sufficient to cover the cost of the bubble-up operation required. Since the changes made by this operation are internal within a tree, the invariant still holds.

2.2 Potential method

To apply the potential method, we define the potential function as follows: for any set S of binomial heaps,

$$\Phi(S) = \sum_{\text{heap } H \text{ in } S} \text{number of binomial trees in } H.$$

We consider the amortized cost of each operation. For each operation considered below, S and S' are the sets of heaps before and after applying the operation, respectively. c and $\hat{c}$ are the actual cost and the amortized cost of the operation, respectively.

- MAKE-HEAP(): The actual cost of this operation is 0. As the number of trees is unchanged, the potential is unchanged by this operation. Hence,

$$\begin{aligned}\hat{c} &= c + (\Phi(S') - \Phi(S)) \\ &= 0 + 0 = 0.\end{aligned}$$

  Therefore, the amortized cost of creating an empty heap is $0 = O(1)$.

- MELD(H_1, H_2): The actual cost of this operation equals to the number of mergings of binomial trees. Since each tree merge reduces the number of binomial trees by 1. Consequently,

$$\begin{aligned}\hat{c} &= c + (\Phi(S') - \Phi(S)) \\ &= \text{number of tree mergings} + ((\Phi(S) - \text{number of tree mergings}) - \Phi(S)) \\ &= 0.\end{aligned}$$

  Therefore, the amortized cost of melding two binomial heaps is $0 = O(1)$.

- INSERT(H, x): This operation can be broken into two steps:

  1. Create a new heap H' and a new binomial tree in H' containing a single node for x;
  2. Meld H with H'.

  The actual cost of Step 1 is 2 and the potential of the set of heaps increases by 1, since one new binomial tree is created. Therefore, the amortized cost of Step 1 is 3. Step 2 involves melding two heaps, which, as explained above, has the amortized cost of 0. Consequently, the amortized cost of inserting a new node to a heap is $3 = O(1)$.

- EXTRACT-MIN(H): We breake this operation into 2 steps:

  1. Remove a binomial tree T in heap H with a minimum root; call the resulting heap H'; remove the root of T, obtaining smaller binomial trees $T_1, T_2, ..., T_k$; and form a new heap H_T from the binomial trees $T_1, T_2, ..., T_k$;
  2. Meld H' with H_T.

  The actual cost of Step 1 is 1 (from creating a new heap). Since one binomial tree, namely T, is removed and k new binomial trees are created, where k is the order of tree T, which is at most $\lfloor lg(n) \rfloor$, where n is the number of nodes in H. Hence, the potential of the set of heaps increases by at most $-1 + k \leq \lfloor lg(n) \rfloor - 1$. Consequently, the amortized cost of Step 1 is at most $1 + \lfloor lg(n) \rfloor - 1 = \lfloor lg(n) \rfloor$. Step 2 involves melding two heaps, which, as explained above, has the amortized cost of 0. Therefore, the amortized cost of extrating a minimum element from a heap H with n nodes is at most $\lfloor lg(n) \rfloor = O(log(n))$.

- DECREASE-KEY(H,x,k): The actual cost of this operation is at most $\lfloor lg(n) \rfloor$, where n is the number of nodes in H. Since the number of binomial trees is unchanged, the potential difference is 0. Hence, the amortized cost of this operation equals to its actual cost, which is at most $\lfloor lg(n) \rfloor = O(log(n))$.