A Tale of Two Trees: New Analysis for AVL Tree and Binary Heap

Russel L. Villacarlos*  Jaime M. Samaniego†  Arian J. Jacildo‡  Maria Art Antonette D. Clariño§

Abstract

In this paper, we provide new insights and analysis for the two elementary tree-based data structures – the AVL tree and binary heap. We presented two simple properties that gives a more direct way of relating the size of an AVL tree and the Fibonacci recurrence to establish the AVL tree’s logarithmic height. We then give a potential function-based analysis of the bottom-up heap construction to get a simpler and tight bound for its worst-case running-time.

1 Introduction

The AVL tree [AL62] and binary (max-)heap [Wil64; Cor+09] are arguably the most elementary tree data structures in the literature. The AVL tree is the first binary search tree data structure with guaranteed logarithmic height. Proving its logarithmic height relies on bounding \( N(h) \), the minimum number of nodes needed to construct an AVL tree with height \( h \). A tree of minimum size must have subtrees of different height and of minimum size thus, \( N(h) = N(h - 1) + N(h - 2) + 1 \).

The structure of the recurrence suggests that \( N(h) \) is related to the Fibonacci recurrence. Indeed, it is provable via induction that \( N(h) = F(h + 2) - 1 \), where \( F(i) \) is the \( i \)th Fibonacci number. Since it is known that \( F(i) \geq \phi^{i-2} \), where \( \phi = (1 + \sqrt{5})/2 \) is the golden ratio, it follows that \( h = O(\log_\phi n) \) with \( n \) being the number of elements in the tree.

There has been renewed interest in AVL tree with the development of new AVL variants – rank-balanced tree [HST09] and ravl tree [ST10], and the analysis of AVL tree performance with respect to the number of rotations [ALT16]. Very recently, an open problem posed in [GV20] asks for an alternative analysis of the height of AVL tree using a potential function. In this paper, we partially address this problem by presenting an alternative analysis. Although not based on a potential function argument, our analysis offers new insights on properties of AVL trees.

The binary heap is a simple data structure primarily used for the implementation of the heapsort algorithm [Wil64] and priority queue [Cor+09]. An important algorithm for binary heap, introduced by Floyd [Flo64], efficiently constructs a heap from an \( n \) element array. The algorithm, which we refer to as build-heap, treats the input array as an ordinary binary tree then applies, bottom-up, the sift-down operation to each subtree. This sift-down operation transforms a subtree into a heap by moving down the root element, via exchanges with a child, until it is in its proper position in the subtree.

*Department of Information Technology, Cavite State University. Email: rlvillacarlos@cvsu.edu.ph. Research conducted at University of the Philippines - Los Baños with support from the Accelerated Science and Technology Human Resource Development Program of the Department of Science and Technology.
†Institute of Computer Science, University of the Philippines - Los Baños. Email: jmsamaniego2@up.edu.ph.
‡Institute of Computer Science, University of the Philippines - Los Baños. Email: ajjacildo@up.edu.ph.
§Institute of Computer Science, University of the Philippines - Los Baños. Email: mdclarino@up.edu.ph.
The running-time of build-heap can be obtained by summing-up the time taken by the individual sift-down operations. A sift-down, in the worst-case, takes \( O(h) \) time, where \( h \) is the height of the subtree. This is since the root element can go down the very bottom of the subtree. There are \( \lceil n/2^{(h+1)} \rceil \) subtrees of height \( h \) and the maximum height of a subtree is \( \log_2 n \). Therefore, the running time can be described by the sum \( \sum_{h=0}^{\log_2 n} h \cdot \lceil n/2^{(h+1)} \rceil \), which is at most \( 2n \).

The analysis of build-heap presented above is a form of aggregation commonly used in amortized analysis [Cor+09]. Apart from aggregation, another common technique used is the potential method. The recent result in [GV20] used this potential method to obtain a simple analysis of the Euclidean algorithm. We take a similar approach in this paper for a simpler analysis of the running-time of build-heap.

### 1.1 Our Contributions

In this paper, we provide alternative analysis for the height of AVL tree and the running time of the build-heap algorithm. In our analysis of AVL tree we present two simple properties of AVL tree and used them to directly prove that \( N(h) = F(h+2) - 1 \). The first property shows that an AVL tree with \( N(h) \) elements can be constructed from a tree with \( N(h-1) \) elements by only adding leaves. This property implies that if we construct an AVL Tree of height \( h \) from an initially empty tree, then the nodes of the final tree can be divided into groups based on the period they were added as leaves. We note that this property is a more explicit formulation of the result in [ALT16], that proves that an \( n \)-node AVL tree can be constructed using \( n \) inserts.

The second property shows that if \( N_L(h) \) is the number of leaves of the minimum sized AVL tree of height \( h \), then \( N_L(h) = F(h) \). This property gives a more direct connection between the AVL tree and the Fibonacci numbers. If we then consider the grouping of nodes earlier, \( N(h) \) is the sum of the first \( h \) Fibonacci numbers. Since the sum of the first \( i \) Fibonacci number is known to be equal to \( F(i+2) - 1 \), the bound on \( N(h) \) follows.

For our analysis of build-heap, the key idea is the heap-merging interpretation of the algorithm. First, we treat the array as a forest containing \( n \) heaps. We then view the sift-down operation as a merging operation to build a larger heap from smaller heaps in the forest. Finally, the build-heap then becomes a sequence of merge that produces a single heap. This new interpretation allows us to use a very simple potential function in terms of the levels of heaps in the forest. We show that the time taken by a sift-down operation is proportional to potential loss in the process.

### 2 Analysis of AVL Tree

Let \( T_h \) be the AVL tree of height \( h \) with minimum number of nodes. \( T_h \) can be viewed recursively as a tree containing a root \( r \) with two, possibly empty, subtrees \( T_{h-1} \) and \( T_{h-2} \).

Let \( N(h) \) be the number of nodes and \( N_L(h) \) be the number of leaves of \( T_h \), respectively. We now establish the relation between the sizes of \( T_h \) and \( T_{h-1} \).

**Lemma 2.1.** For \( h \geq 1 \), \( N(h) = N(h-1) + N_L(h) \).

**Proof.** The proof relies on the construction of \( T_h \) from \( T_{h-1} \). Using the recursive view of \( T_{h-1} \), it follows that we must increase the height of the subtrees of \( T_{h-1} \) that contains a leaf as a child. These are the \( T_0 \) and \( T_1 \) subtrees. A \( T_1 \) subtree is any height 1 subtree while \( T_0 \) is any height 0 subtree that is not a subtree of some \( T_1 \). Note that \( T_0 \) contains only a leaf while \( T_1 \) contains an internal node with one leaf. Increasing the height of any \( T_0 \) effectively replaces its leaf with a \( T_1' \) subtree. For the case of a \( T_1 \) subtree, its leaf is replaced by two subtrees – \( T_0' \) and \( T_1' \). Essentially, the process
replaces all the leaves of \( T_{h-1} \) with internal nodes from all the \( T'_1 \) subtrees then introduces new leaves from the \( T'_0 \) and \( T'_1 \) subtrees. In effect, we can form a bijection between the nodes of \( T_{h-1} \) and the internal nodes of \( T_h \). Therefore, adding the number leaves of \( T_h \) and the size of \( T_{h-1} \) gives the size of \( T_h \).

Lemma 1, when applied repeatedly, suggests that an AVL tree \( T_h \) can be constructed incrementally beginning with some smaller tree. We now show that the number of leaves of \( T_h \) is strongly related to the Fibonacci numbers.

**Lemma 2.2.** Let \( F(i) \) be the \( i^{th} \) Fibonacci number, then \( N_L(h) = F(h) \), for \( h \geq 0 \).

*Proof.* We show that \( N_L(h) \) follows the Fibonacci recurrence. For \( h \leq 1 \), direct construction of \( T_0 \) and \( T_1 \) shows that \( N_L(0) = F(0) \) and \( N_L(1) = F(1) \). We now show that for \( h > 1 \), \( N_L(h) = N_L(h-1) + N_L(h-2) \). From the proof of Lemma 1, the leaves of \( T_h \) are the leaves created after transforming all \( T_0 \) and \( T_1 \) subtrees of \( T_{h-1} \). Let \( x'_1 \) be the leaf added to \( T_h \) after replacing the leaf of \( T_0 \) with a \( T'_1 \) subtree. Let \( y'_1 \) be the leaves added to \( T_h \) after replacing the leaf of \( T_1 \) by \( T'_0 \) and \( T'_1 \) subtrees. Observe that we can form a bijection between the leaves of \( T_{h-1} \) and the \( x' \) leaves of \( T_h \). Therefore, counting all the \( x' \) gives \( N_L(h-1) \). Also, a bijection can be formed between the roots of each \( T_1 \) and the \( y' \) leaves of \( T_h \). These roots correspond to the leaves of \( T_{h-2} \) since all height 0 nodes (leaves) in \( T_{h-2} \) becomes height 1 nodes (roots of \( T_1 \) subtrees) in \( T_{h-1} \). Thus, counting all \( y' \) gives \( N_L(h-2) \). Since the leaves of \( T_h \) are the \( x' \) and \( y' \) combined, \( N_L(h) = N_L(h-1) + N_L(h-2) \) and the statement of the lemma follows.

We can now easily prove the bound on the size of \( T_h \).

**Corollary 2.1.** \( N(h) = F(h+2) - 1 \).

*Proof.* Applying Lemma 1 and Lemma 2 repeatedly we have \( N(h) = N(0) + \sum_{i=1}^{h} N_L(i) \). Since \( N(0) = N_L(0) = F(0) \), \( N(h) = \sum_{i=0}^{h} F(i) \). The claim follows given that \( \sum_{i=0}^{h} F(i) = F(i + 2) - 1 \).

**Theorem 2.1.** AVL tree has logarithmic height.

*Proof.* From Corollary 2.1, \( N(h) = F(h+2) - 1 \). Since \( F(i) \geq \phi^{(i-2)} \) we have \( N(h) \geq \phi^{h} - 1 \). Taking the logarithm of both sides and letting \( n = N(h) \), it follows that the height of an AVL tree is at most \( \log_\phi(n + 1) \).

### 3 Analysis of Build-Heap

In our analysis, we shall treat the array as a forest of \( n \) trees. That is, each element of the array is a root of a distinct tree. Since each tree contains only one node, they can be considered as heaps. A sift-down can then be interpreted as merging of heaps to produce one larger heap. Thus, the build-heap algorithm is simply a sequence of merges to convert the entire forest into a single heap.

Let the level of a heap be equal to the height of the heap plus one. We let the level of an empty heap be 0. Under this definition, the forest initially contains \( n \) heaps with level 1. Further, a sift-down merges one heap of level 1, called the *parent heap*, with two heaps with level at most \( l \), called *child heaps*, to produce a child heap of level \( l + 1 \). Initially, the parent heaps are those elements belonging to the upper-half of the array, while the remaining half are the child heaps.

We now prove the running-time of build-heap using the potential method:

**Theorem 3.1.** The worst-case running-time of build-heap is \( O(n) \).
Proof. Let the potential be the total levels of all heaps in the forest. Since the forest initially contains \( n \) heaps of level 1, \( \Phi_0 = n \). After merging all the heaps, the height of the final heap is \( \log_2 n \), thus \( \Phi_m = \log_2 n + 1 \). During a sift-down, if the child heaps have levels \( l \) and \( l - 1 \), then in the worst case, the potential will decrease by \( 2l \) and then increase by \( l + 1 \). This is so, since the sift-down will remove three heaps, which has a total level of \( 2l \), and then replace them with a heap of level \( l + 1 \). For the case where the child heaps both have \( l \) levels, the decrease in potential is \( 2l + 1 \).

The actual cost, \( a_i \), of the \( i \)th sift-down operation is at most \( l \) since in the worst case, the root of the parent heap will be compared to at most \( l \) nodes from one of the child heaps. The amortized cost, \( \tilde{a}_i \), of a sift-down at time \( i \) is:

\[
  \begin{align*}
  \text{Case 1: Child heaps have different levels:} & \quad \tilde{a}_i = a_i + \Phi_i - \Phi_{i-1} = l - 2l + (l + 1) = 1 \\
  \text{Case 2: Child heaps have same level:} & \quad \tilde{a}_i = a_i + \Phi_i - \Phi_{i-1} = l - (2l + 1) + (l + 1) = 0
  \end{align*}
\]

Since there are only at most \( n/2 \) parent heaps, the number of sift-down operations, \( m \), is at most \( n/2 \). The amortized cost of a sift-down is at most 1 therefore, the total amortized cost of the sequence of sift-down is \( n/2 \). The worst-case running time of the bottom-up heap construction is the total actual cost:

\[
  \sum_{i=0}^{m} a_i = \sum_{i=0}^{m} \tilde{a}_i + \sum_{i=1}^{m} (\Phi_{i-1} - \Phi_i) \\
  \leq n/2 + \Phi_0 - \Phi_{n/2} \\
  = n/2 + n - \log_2 n - 1 \\
  \leq n/2 + n \\
  = 1.5n \\
  = O(n)
\]

References

[AL62] George M Adel’son-Vel’skii and Evgenii Mikhailovich Landis. “An algorithm for organization of information”. In: Doklady Akademii Nauk. Vol. 146. 2. Russian Academy of Sciences. 1962, pp. 263–266.

[ALT16] Mahdi Amani, Kevin A Lai, and Robert E Tarjan. “Amortized rotation cost in AVL trees”. In: Information Processing Letters 116.5 (2016), pp. 327–330.

[Cor+09] Thomas H. Cormen et al. Introduction to Algorithms, Third Edition. 3rd. The MIT Press, 2009. isbn: 0262033844.

[Flo64] Robert W. Floyd. “Algorithm 245: Treesort”. In: Commun. ACM 7.12 (Dec. 1964), p. 701. issn: 0001-0782. doi: 10.1145/355588.365103. url: https://doi.org/10.1145/355588.365103.

[GV20] Bruno Grenet and Ilya Volkovich. “One (more) line on the most Ancient Algorithm in History”. In: Symposium on Simplicity in Algorithms. SIAM. 2020, pp. 15–17.

[HST09] Bernhard Haeupler, Siddhartha Sen, and Robert E Tarjan. “Rank-balanced trees”. In: Workshop on Algorithms and Data Structures. Springer. 2009, pp. 351–362.
[ST10] Siddhartha Sen and Robert E Tarjan. “Deletion without rebalancing in balanced binary trees”. In: Proceedings of the twenty-first annual ACM-SIAM Symposium on Discrete Algorithms. SIAM. 2010, pp. 1490–1499.

[Wil64] J. W. J. Williams. “Algorithm 232: Heapsort”. In: Communications of the ACM 7.6 (1964), pp. 347–348.