On the Optimality of a Family of Binary Trees
Techical Report TR-20111012-1

Dana Vrajitoru and William Knight
Indiana University South Bend
Department of Computer and Information Sciences

Abstract

In this technical report we present an analysis of the complexity of a class of algorithms. These algorithms recursively explore a binary tree and need to make two recursive calls for one of the subtrees and only one for the other. We derive the complexity of these algorithms in the worst and in the best case and show the tree structures for which these cases happen.

1 The Problem

Let us consider a traversal function for an arbitrary binary tree. Most of these functions are recursive, although an iterative version is not too difficult to implement with the use of a stack. The object of this technical report, though, is those functions that are recursive.

For the remainder of the paper we'll consider the classic C++ implementation of a tree node as follows:

```
template <class otype>
struct node
{
  otype datum;
  node *left, *right;
};
```

When a recursive function makes a ***simple traversal*** of a binary tree in which the body of the traversal function contains exactly two recursive calls, one on the pointer to the left subtree and one on the pointer to the right, and all other parts of each call, exclusive of the recursive calls, require time bounded by constants, then the execution time for traversal of a tree with n nodes is roughly proportional to the total number of calls (initial and recursive) that are made. In this case that will be $1+2n$ (the call on the pointer to the root of the tree and one call on each of the $2n$ pointers in the tree), so the execution time is $\Theta(n)$. The analysis would apply, for example, to the function in Figure 1 that traverses the tree to calculate its height.

Figure 2 shows a differently coded version of the function that calculates the height of a binary tree. Note that the code is a little simpler (shorter) than the code in the version in Figure 1. The code in Figure

```
int height (node_ptr p)  // p  is a pointer to a binary tree
{
  if (p == NULL)
    return -1;   // The base case of an empty binary tree.
  int left_height  = height (p->left);
  int right_height = height (p->right);
  if (left_height <= right_height)
    return 1 + right_height;
  else
    return 1 + left_height;
}
```

Figure 1: The height of a binary tree

2 is *not* a "simple traversal" of the kind described above. Here is the reason: when recursive calls are made, exactly one of the recursive calls is *repeated*. Clearly then the total number of calls (initial and recursive) is not just $2n + 1$, where n is the number of nodes in the tree. We shall try to figure out the total number of calls that could be made when the second version of `height` is called on a tree T with n nodes.

```
int height (node_ptr p)  // p  is a pointer to a binary tree
{
  if (p == NULL)
    return -1;   // The base case of an empty binary tree.
  if (height(p->left) <= height(p->right))
    return 1 + height(p->right);
  else
    return 1 + height(p->left);
}
```

Figure 2: Inefficient version of the function height

At first sight it would seem that this is not a very useful problem to study because we can easily correct the fact that this function performs two recursive calls on one of the subtrees. We can store the result of the function in a local variable and use it instead of the second recursive call, as shown in Figure 1. Even if this is the case indeed, it would still be useful to know just "how bad" the complexity of the function can get from a simple change.

The second motivation is that just as the function in Figure 1 is representative of a whole class of traversal functions for binary trees, the analysis for the function in Figure 2 can also be applied to a whole class of functions. Some of these can be optimized with the method used for the function `height`, but some of them might require operations making the second recursive call on the same subtree necessary.

An example of such a problem would be modifying the `datum` in each of the nodes situated in the taller subtree of any node. One traversal is necessary to determine the height of the subtrees. A second traversal is necessary for the subtree of larger height to increment its datum values.

2 Complexity Function

Let $K(T)$ denote the total number of calls (initial and recursive) made when the second height function is called on a binary tree T, and let L_T and R_T denote the left and right subtrees of T. Then we can write

$$K(T) = \begin{cases} 1 \quad \text{if } T = \emptyset \text{ (i.e. } n = 0) \\ 1 + K(L_T) + K(R_T) + K(\text{the taller of } L_T \text{ and } R_T) \text{ otherwise} \end{cases}$$

$$= \begin{cases} 1 \quad \text{if } T \text{ is empty} \\ 1 + K(L_T) + 2\,K(R_T) \quad \text{if } R_T \text{ is at least as tall as } L_T \text{ and } T \neq \phi \\ 1 + 2\,K(L_T) + K(R_T) \quad \text{otherwise} \end{cases}$$

Theorem 2.1. *For a tree with n nodes, the function K has complexity $\Theta(2^n)$ in the worst case.*

Proof. For non-empty trees with n nodes, we can maximize the value of $K(T)$ by making the taller of L_T and R_T contain as many nodes as possible, so that the last term will add as much as possible to the value of $K(T)$. This involves putting all the nodes except the root into one of the two subtrees, and doing the same at every level below the root. This results in a tree that has maximum possible height $n - 1$. Suppose, for example we make every node (except the root) the right child of its parent. Let $F(n)$ denote $K(T)$ for this kind of tree T with n nodes (that is, $F(n)$ denotes the total number of calls that will be made on such a tree). Then our equations above can be turned into a recurrence problem of the form

$$F(0) = 1, \quad F(n) = 1 + F(0) + 2F(n-1) = 2F(n-1) + 2. \tag{1}$$

This problem is easy to solve for $F(n)$, and the solution is $\Theta(2^n)$. That is, the second version of the `height` function has catastrophically bad execution time on degenerate binary trees of maximal height. This is the worst possible case for that algorithm. ■

Having identified the worst case for $K(T)$, let's now try to find the best case. Suppose we are given a positive integer n and asked which among all binary trees T with n nodes minimize $K(T)$.

Definition 2.2. *A* K-optimal tree *of size* n *is a binary tree* T *with* n *nodes that minimizes the value of* K *among all trees with* n *nodes.*

Based on what we have just seen with trees that maximize $K(T)$, it is reasonable to conjecture that the way to build a K-optimal tree of size is to make it as short as possible.

Perhaps, one might guess, a binary tree is K-optimal if and only if it is *compact*, meaning that all of its levels except for the last one contain all the nodes that they can contain. As it turns out, however, many compact trees are not K-optimal, and many K-optimal trees are not compact.

The following lemma will allow us to simplify our search for K-optimal binary trees by restricting the shapes of the trees that need to be examined.

Lemma 2.3. *Let* T *be a binary tree. For any node in* T*, if the left subtree is taller than the right subtree, then the two subtrees can be interchanged without changing the value of the function* K*.*

Proof. This is easy to see by examining the code in the second height function. ■

Lemma 2.3 tells us that given any binary tree T, we can go through the tree and interchange the subtrees of every node whose left subtree is taller than its right subtree to produce a tree T' for which $K(T') = K(T)$. Thus in searching for K-optimal binary trees, we can restrict our search to those trees in which every node has a left subtree of height less than or equal to the height of its right subtree. We will call such trees *right-heavy*. Note that a binary tree is right-heavy if and only if every one of its subtrees is right-heavy.

For convenience in discussing how to modify a binary tree T to decrease the value of $K(T)$, let's label each node N in a tree with the number of calls to the second height function that will be made on the pointer to N, and label each empty subtree E (sometimes called an "external node") with the number of calls on the null pointer that indicates that E is empty. Figure 3 shows a tree labeled using this system. The K value of this tree is obtained by adding up all the numeric labels in the tree. In this example the K value is 118. We will also refer to the sum of the labels in a subtree as the *weight* of the subtree. Because the tree in Figure 3 is right heavy, the duplicate recursive call in the second height function will always be made on the right subtree, never on the left. As a result, for each (internal) node N in the tree, the left child of N always has the same label as N, while the right child always has a label that's twice the label on N. This explains why all the labels in Figure 3 will be integer powers of 2. Actually, this is true for all binary trees.

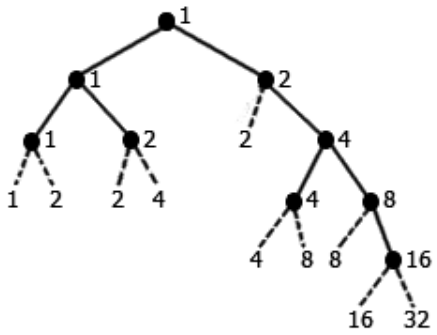

Figure 3: An example of right-heavy tree with labeled nodes. The dashed lines indicate null pointers.

Suppose A and N are nodes in a binary tree; if A is an ancestor of N, and if N is reached from A by following only right pointers, then N is a "right descendant" of A, and A is a "right ancestor" of N.

Lemma 2.4. *Let T be a right-heavy binary tree, and let L be a leaf of T. Then L can be removed without changing the label of any other node if and only if L satisfies one of the following conditions:*

a) *L is the only node in T;*

b) *L is a left child of its parent;*

c) *L is a right child of its parent, and for each right ancestor A of L, the left subtree of A is strictly shorter than its right subtree. (Figure 4 shows an example of a right leaf, in solid black color, that can be removed without changing the label on any other node in the tree.)*

Proof. A simple observation tells us that the leaf L can be removed from T without changing the label of any other node in T if and only if the remaining tree is right-heavy after L is removed. Thus our strategy for proving the Lemma will be as follows: we'll prove that each of the three conditions (a), (b), and (c) separately implies that when L is removed from T the remaining tree is right-heavy; then we'll prove that if all three conditions are false, the remaining tree is not right-heavy after L is removed from T.

First, suppose the leaf L is the only node in T. Then removing L from T leaves the empty tree, which is vacuously right-heavy. (Note that the two empty subtrees of L are also removed from T and their labels disappear, but the label on the empty subtree that replaced L is the same as the label on L.)

Second, suppose the leaf L is the left child of some node P. Since T is right-heavy, P must have a non-empty right subtree. It is now easy to see that if L is removed from T the remaining tree is right-heavy.

Now suppose the leaf L is the right child of some node P, and that for each right ancestor A of L, the left subtree of A is strictly shorter than its right subtree. Then in particular, the left subtree of P must be shorter than the right subtree, which consists of just L, so the left subtree of P is empty. Removing L

will leave the subtree rooted at P right-heavy (both subtrees are empty), and no other subtrees in T are changed. Now go up one level to the parent P' of P. If the leaf L is a right descendant of P', then the left subtree of P' is shorter than its right subtree (before L is removed). Thus if L is removed, the resulting tree rooted at P' will be right-heavy (its left subtree will be no taller than its right). Continue up the tree in a similar manner to each of the right ancestors of L.

If we reach the root of the tree in this manner, then the proof of this case is finished. If we reach a left parent of a right ancestor of L, call it B, then the right subtree of B must be of the same height or taller than its left subtree from which L is removed. After removing L, not only the left subtree of B is now shorter than the right subtree, but the height of the subtree with root B does not change on the whole. This height was initially equal to 1 plus the height of the right subtree, so it doesn't change by removing L from the left side. This also implies that the labels of any node outside of the subtree with root B will not change either.

Finally, suppose that all three conditions (a), (b), and (c) of the Lemma are false, which means that the leaf L is the right child of some node in T and at least one right ancestor of L has left and right subtrees of equal height (the left can't be strictly taller because T is right-heavy). We must prove that when L is removed from the tree, the remaining tree is not right-heavy. Begin by finding the right ancestor of L that's closest to L. Call that ancestor A. If A is the parent of L, its left subtree must contain exactly one node, and if L is removed, the remaining tree will no longer be right-heavy at A. If A is farther up the tree, then each of its right descendants except L must have a left subtree that's shorter than its right subtree. Thus, when L is removed, the height of A's right subtree will decrease, making it strictly shorter than A's left subtree. This means that the remaining tree is no longer right-heavy at A. ■

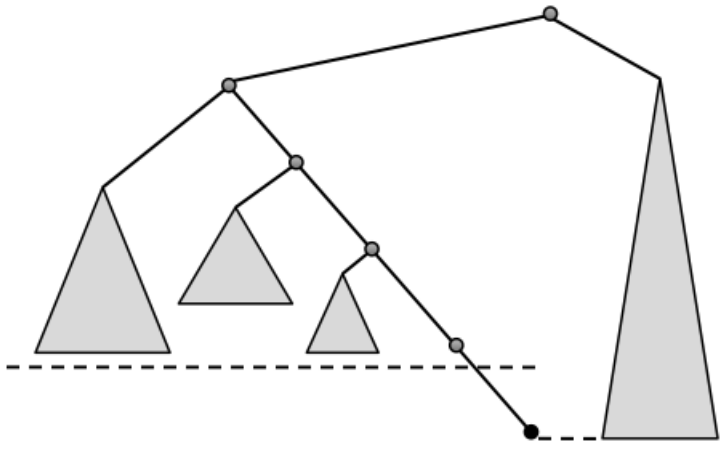

Figure 4: A right leaf that can be removed without changing the labels in the tree

Corollary 2.5. *Let T be a right-heavy binary tree. We can add a new leaf L to the tree without changing the label of any other node if and only if L and T satisfy one of the following conditions:*

a) *T is empty before inserting L;*

b) *L is added as a left child of any node that has a right child;*

c) *L is added as the right-most leaf in the tree or in a place such that the first ancestor of L that is not a right ancestor has a right subtree of height strictly greater than the height of the left subtree before adding L.*

Proof. This is a direct consequence of Lemma 2.4. ■

Theorem 2.6. *The K function is strictly monotone over the number of nodes on the set of K-optimal trees. In other words, if T_m and T_n are two K-optimal trees with number of nodes equal to m and n respectively, where $m < n$, then $K(T_m) < K(T_n)$.*

Proof. It suffices to prove the statement in the theorem for $m = n - 1$. Let T_n be a K-optimal tree with n nodes. Without loss of generality, we can assume that T_n is right-heavy.

Let us locate the left-most leaf. To find this node, we need to follow the left-most path in the tree starting from the root. If we end up with a node that is not a leaf, then we take one step to the right. We repeat this procedure until we find a leaf. There are 3 possible situations that we need to consider, as shown in Figure 5. Note that in this figure, the labels of the empty subtrees are not shown for a better clarity of the illustration.

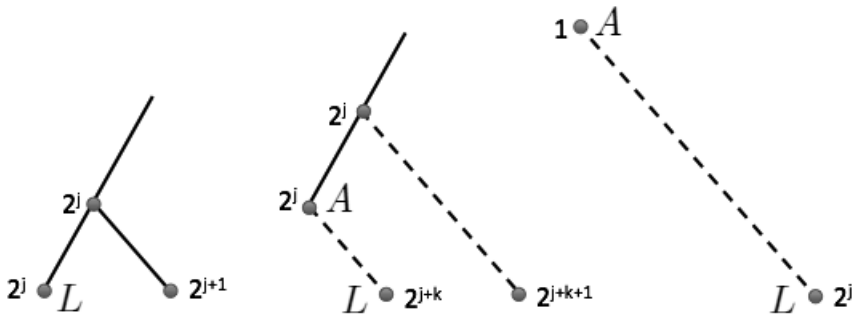

Figure 5: Possible placement of the left-most leaf, denoted by L

Suppose this leaf, call it L, is at the end of a left branch (see the left-most case in Figure 5). Since T_n is right-heavy, Lemma 2.4, case (b), tells us that we can remove L from T_n without changing any of the labels on the other internal nodes of the tree. This produces a right-heavy tree with $n - 1$ nodes and

strictly smaller K value, because the labels that were on the two empty subtrees (external nodes) of L have disappeared from the tree. This smaller tree may not be optimal among all binary trees with $n-1$ nodes, in which case there is some K-optimal tree T_{n-1} with even smaller K value. Thus all K-optimal trees with $n-1$ nodes have smaller K-value than $K(T_n)$.

Now suppose the leaf L is a right child. Let A be its highest right ancestor in T_n. (In the most extreme case, A is the root of T_n and L is the only leaf in T_n, as shown in the right-most case in Figure 5.) Then each of the right ancestors of L must have an empty left subtree, for otherwise in our search for L we would have gone left from some right ancestor of L instead of descending all the way to L. By Lemma 2.4 we can remove L without changing any of the other labels in T_n, leaving a right-heavy tree with smaller K-value. As in the preceding paragraph, this proves that K-optimal trees with $n-1$ nodes have smaller K-value than $K(T_n)$. ■

3 Two Special Cases

Definition 3.1. *A* perfect binary tree *is one where all the levels contain all the nodes that they can hold.*

A perfect tree of height h has a number of nodes $n = 2^{h+1} - 1$. We can reverse this to express $h = \lg(n+1) - 1 = \Theta(\lg(n))$.

Theorem 3.2. *The function K has a complexity of $\Theta(n^{\lg(3)})$ on perfect trees, where n is the number of nodes in the tree.*

Proof. For a perfect tree of height $h \geq 0$, the two subtrees are perfect trees of height $h-1$. If we denote by κ the value of the function K on a perfect tree of height h, we can write the sum of labels on these trees as

$$\kappa(h) = 1 + 3\kappa(h-1), \quad \kappa(0) = 4.$$

A particular solution for this recurrence relation will be a constant, let's say α. Substituting in the recurrence relation we get $\alpha = 1 + 3\,\alpha$, from which we can deduce that $\alpha = -1/2$.

By adding to this particular solution, the general solution to the corresponding homogeneous recurrence relation, we get the solution $\kappa(h) = \beta\, 3^h - \frac{1}{2}$. By substituting $\kappa(0) = 4$, we get $\beta = 9/2$, and so

$$\kappa(h) = \frac{9}{2}3^h - \frac{1}{2} = \Theta(3^h).$$

Let us denote by P_n a perfect binary tree with n nodes. Using the relationship between n and h, we can now express the same sum of labels as a function of the number of nodes, getting us back to the function K itself:

$$K(P_n) = \Theta(3^{\lg(n)}) = \Theta(n^{\lg(3)}).$$

Even though most perfect trees turn out not to be K-optimal, knowing what their sum of labels is and knowing that the K-optimal function is monotonic gives us an upper bound for the minimal complexity for a given number of nodes. ■

Corollary 3.3. *The height of a K-optimal tree with n nodes cannot be larger than $c + \lg(3)\lg(n)$, where c is a constant.*

Proof. A K-optimal tree with n nodes and height h must have one longest path where the label of every node is an increasing power of 2, going from 1 for the root to 2^h for the leaf. We have to add to this the labels on the empty subtrees of the leaf, which are 2^h and 2^{h+1}. If we add up all these labels and the labels of the empty subtrees we obtain the sum $2^{h+2} - 1 + 2^h$. This sum is less than or equal to the K-value of this K-optimal tree with n nodes, which, by monotonicity, is less than or equal to the K-value of the smallest perfect tree of a number of nodes $m \geq n$. If g is the height of this perfect tree, then its number of nodes is $m = 2^{g+1} - 1$. If we choose the smallest of these trees, then $2^g - 1 < n \leq 2^{g+1} - 1$, which implies $2^g \leq n < 2^{g+1}$, so $g \leq \lg(n) < g + 1$. Since the number g is an integer, by the properties of the function floor we have $g = \lfloor \lg(n) \rfloor$.

Thus, the height of this perfect tree is equal to $\lfloor \lg(n) \rfloor$ and its number of nodes is $m = 2^{\lfloor \lg(n) \rfloor + 1} - 1 \leq 2n - 1$. By Theorem 3.2, this implies that,

$$2^{h+2} - 1 + 2^h \leq a\, m^{\lg(3)} \leq a(2n-1)^{\lg(3)} < a(2n)^{\lg(3)} = 3an^{\lg(3)}$$

for some constant a. From this we can write

$$5\, 2^h \leq 3an^{\lg(3)} \quad \Rightarrow \quad h \leq \lg(3a/5) + \lg(3)\lg(n)$$

and the quantity $\lg(3a/5)$ is the constant c in the corollary. ■

Lemma 3.4. *The sum of labels on level k of a perfect binary tree is equal to 3^k.*

Proof. We will use induction on the level number to prove this lemma.

Base case. The root of the tree has a label of 1 and is on the level 0, and $1 = 3^0$.

Inductive step. Let us suppose that the sum of labels on level k is equal to 3^k. Let us consider an arbitrary node on this level. The node has two children on the next level: the left one with the same label as it, and the right one with a label equal to twice the size. Thus, the sum of the labels of the children is equal to 3 times the label of the node. Since the tree is perfect and this property is true for each of its nodes on level k, then the sum of the labels on level $k+1$ will be equal to $3 * 3^k = 3^{k+1}$. ■

Lemma 3.5. *The number of nodes on level k of a perfect binary tree that have labels equal to 2^j, where $0 \leq j \leq k$, is equal to $C(k, j)$, where C(k,j) denotes the number of combinations of k things taken j at a time.*

Proof. We will prove this lemma by induction over k using a property of the combinations. It is well known that the combinations function has the following property:

$$C(m, p) = C(m-1, p) + C(m-1, p-1).$$

Let us denote by $C_t(k, j)$ the count of nodes with label equal to 2^j on level k. We'll prove that C_t is identical with the function C.

Base case. For $k = 0$ we only have one node, so $C_t(0, 0) = 1 = C(0, 0)$.

Inductive step. For an arbitrary k and j, there are two types of nodes with label 2^j on level k. The first type are left children of their parents and their labels are identical to those of their parents. The count of such nodes is $C_t(k-1, j) = C(k-1, j)$ (by the inductive step). The second type of nodes are right children of their parents. These nodes have labels that are the double of the labels of their parents. So every node of label 2^{j-1} on level $k-1$ will have a right child of label 2^j on level k. Thus, the count of such nodes on level k is equal to $C_t(k-1, j-1) = C(k-1, j-1)$ (by the inductive step).

By summing up the count of nodes that are left children and those that are right children, we have that

$$C_t(k, j) = C(k-1, j) + C(k-1, j-1) = C(k, j). \quad ■$$

Theorem 3.6. *A perfect binary tree of height $h \geq 16$ is not K-optimal.*

Proof. Let T be a perfect binary tree of height $h \geq 16$. Our strategy will be to show that we can find

another binary tree, say T', with the same number of nodes as T but a smaller K-value. This will prove that T is not K-optimal. T' will be constructed by removing $h+2$ of the leaves of T and re-attaching them elsewhere, as shown in Figure 6. Now let's look at how to do the removals.

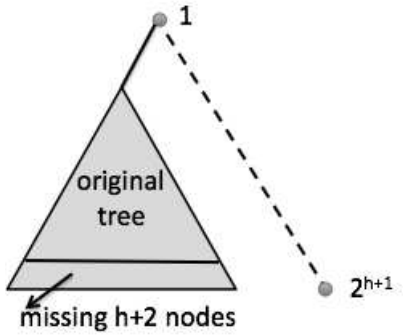

Figure 6: Tree of smaller weight built from a perfect tree

The next-to-last level (level $h-1$) of our perfect tree T contains 2^{h-1} nodes, each with a label that's a power of 2. By Lemma 3.5, there are $C(h-1,h-2)$ labels of the form 2^{h-2}. Note that $C(h-1,h-2)=h-1$. By Lemma 2.4, the left child of each of these $h-1$ nodes can be removed from T without changing any of the labels on the remaining nodes. Consider any one of these left children, call it L. Then L has the same label 2^{h-2} as its parent, and it also has two empty subtrees, one having label 2^{h-2} and the other having label 2^{h-1}. When L is removed from the tree, it leaves behind one empty subtree with label 2^{h-2}. Thus two labels, namely 2^{h-2} and 2^{h-1}, have disappeared from the tree. The net effect, then, of removing L is to decrease the sum of labels in T by $2^{h-2}+2^{h-1}=3*2^{h-2}$. When we do this for all $h-1$ of these left leaves with label 2^{h-2}, we have decreased the total weight (i.e., sum of labels) of T by $3(h-1)2^{h-2}$.

So far we've removed h-1 leaves from T. We need to remove 3 more (to get a total of $h+2$). So now look at the nodes in level $h-1$ that have label 2^{h-3}. There are $C(h-1,h-3)$ such nodes, and when $h\geq 6$, this number exceeds 3. So pick any three of these nodes and remove their left children. Each child removed reduces the weight of the tree by $3*2^{h-3}$ by the same reasoning as we used in the preceding paragraph. Thus the total decrease in the weight of the tree is $9*2^{h-3}$ when these 3 nodes are removed.

Now we've removed $h+2$ nodes from T with a total decrease in weight of $3*(h-2)2^{h-2}+9*2^{h-3}$. We are going to re-attach them as shown in Figure 7. That is, we make one of them the root of the new tree T'; we let what remains of T be the left subtree of T'; and we make all the other removed nodes into right descendants of the new root. This will not change any of the labels in what remains of the original tree, but it will add new labels on the re-attached nodes and their empty subtrees. The nodes themselves have

labels 1, 2, 2^2, ..., 2^{h+1}. Their empty subtrees have labels 2, 2^2, 2^3, ..., 2^{h+2}. The total weight that has been added by the re-attachment of the nodes is therefore $3(2^{h+2} - 1)$.

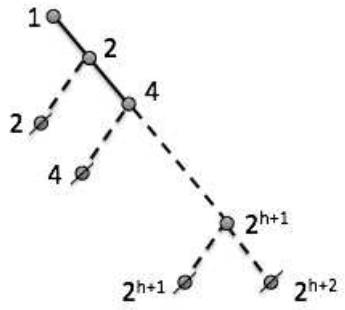

Figure 7: Labels on the added path

Now we need to prove that the weight we subtracted is greater than the weight we added. That is, we need to verify that

$$3(h-1)2^{h-2} + 9 * 2^{h-3} > 3(2^{h+2} - 1).$$

Canceling a 3 from each side gives us the equivalent inequality

$$(h-1)2^{h-2} + 3 * 2^{h-3} > 2^{h+2} - 1.$$

For integers p and q the inequality $p > q - 1$ is equivalent to $p \geq q$, so the inequality we want to verify is equivalent to

$$(h-1)2^{h-2} + 3 * 2^{h-3} \geq 2^{h+2},$$

which in turn is equivalent to

$$2(h-1)2^{h-3} + 3 * 2^{h-3} \geq 2^5 * 2^{h-3}.$$

This can be simplified to

$$2(h-1) + 3 \geq 32,$$

which, since h is an integer, simplifies to $h \geq 16$. ■

Note. A slightly more complex proof allows us to lower the threshold in Theorem 3.6 to 12.

Definition 3.7. *A binary tree T with n nodes is a* **size-balanced** *tree if and only if its left and right subtrees contain exactly $\lfloor (n-1)/2 \rfloor$ and $\lceil (n-1)/2 \rceil$ nodes respectively, and a similar partition of the descendents occurs at every node in the tree.*

Two examples of size-balanced trees are shown in Figure 8. Note that for every node in a size-balanced binary tree, the subtree rooted at that node is, by our definition, size-balanced. Note also that for each positive integer n there is only one possible shape for a size-balanced tree with n nodes.

Theorem 3.8. *The function K on a size-balanced tree with n nodes has a complexity that is $\Theta(n^{\lg(3)})$.*

Proof. Let $S(n)$ denote the value of $K(T)$ when T is the size-balanced tree containing n nodes. That is, $S(n)$ is the total number of times that the height function in Figure 2 is called when the first call is to the size-balanced tree of n nodes.

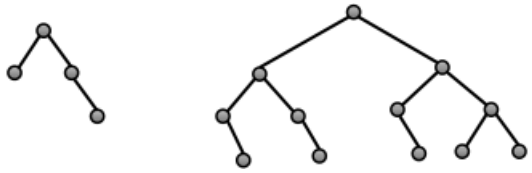

Figure 8: Two size-balanced trees, with 4 and 12 nodes respectively

It is easy to prove by induction that if T_k and T_{k+1} are size-balanced trees having k and $k+1$ nodes respectively, then the height of T_k will be less than or equal to the height of T_{k+1}. This means that at every node in a size-balanced tree, the height of the left subtree of that node will be less than or equal to the height of the right subtree of that same node. This makes size-balanced trees right-heavy. The height function in Figure 2 is written in such a way that for every call to the function on a size-balanced tree there will be one call on the pointer to the left subtree and two calls on the pointer to the right subtree. Thus we can write the following recurrence relation for $S(n)$:

$$S(n) = 1 + S\left(\left\lfloor \frac{n-1}{2} \right\rfloor\right) + 2S\left(\left\lceil \frac{n-1}{2} \right\rceil\right), \quad \text{which is valid for all } n \geq 1.$$

The initial value is $S(0) = 1$. Unfortunately, this is a difficult recurrence relation to solve exactly. We can, however, use the recurrence relation and induction to prove the inequality

$$S\left(\left\lfloor \frac{n-1}{2} \right\rfloor\right) \leq S\left(\left\lceil \frac{n-1}{2} \right\rceil\right), \quad \text{which is valid for all } n \geq 1.$$

This inequality can be combined with the recurrence relation for $S(n)$ to produce two recurrence inequalities:

$$S(n) \leq 1 + 3S\left(\left\lceil \frac{n-1}{2} \right\rceil\right) \quad \text{and} \quad S(n) \geq 1 + 3S\left(\left\lfloor \frac{n-1}{2} \right\rfloor\right) \quad \text{for all } n \geq 1.$$

These inequalities together with the initial value $S(0) = 1$ imply that

$$S(n) \leq \frac{3^{\lfloor \lg(n) \rfloor + 2} - 1}{2} \quad \text{and } S(n) \geq \frac{3^{\lfloor \lg(n+1) \rfloor + 1} - 1}{2},$$

which imply that $S(n) = \Theta(n^{\lg(3)})$. Since $\lg(3) \approx 1.585$, it follows that the growth rate of $S(n)$ is only a little greater than $\Theta(n\sqrt{n})$. Finally, remember that size-balanced trees are not necessarily K-optimal trees, and thus a K-optimal tree T with n nodes will satisfy $K(T) \leq S(n)$. From this it follows that $K(T) = O(n^{\lg(3)})$, where n denotes the number of nodes in T. ■

With the perfect trees (Theorem 3.2) we have seen an example of a class of trees for which the complexity of the function K is $\Theta(n^{\lg(3)})$ but only for a number of nodes equal to a power of 2 minus 1. Theorem 3.8 now gives us an example of a class of trees where the function K has a complexity that is $\Theta(n^{\lg(3)})$ for any arbitrary number of nodes n.

4 Best Case Complexity

Theorem 4.1. *For K-optimal binary trees T_n with n nodes, $K(T_n) = \Theta\left(n^{\lg(3)}\right)$.*

Suppose we want to build a K-optimal binary tree with a prescribed number of nodes n. We shall show how the majority of the nodes must be inserted so as to minimize the sum of labels. This will allow us to show that the K-optimal n-node tree we are building must have a sum of labels that's at least $A(n^{\lg(3)})$ for some number A independent of n. Since Theorem 3.8 implies that the sum of labels in a K-optimal tree with n nodes can be at most $B(n^{\lg(3)})$ for some constant B, we will have proved Theorem 4.1.

So suppose we are given some positive integer n. In building a K-optimal n-node tree, we can without loss of generality require that it be right-heavy (see Lemma 2.3). Then the longest branch in the tree will be the one that extends along the right edge of the tree. Its lowest node will be at level h, where h is the height of the tree. By Corollary 3.3, h will have to satisfy $\lfloor \lg(n) \rfloor \leq h \leq c + \lg(3) \lg(n)$ for a constant c. Thus h is $\Theta(\log(n))$. We can start with $h = \lfloor \lg(n) \rfloor$, then attach additional nodes to this longest branch if they are needed late in the construction. When n is large, we will have used only a small fraction of the prescribed n nodes during construction of this right-most branch. We will still have many nodes left over to insert into the optimal tree we are building. Finally, note that the longest branch will have $h + 1$ nodes, with labels 2^0, 2^1, 2^2, …, 2^h. Their sum is $2^{h+1} - 1$.

Let us add nodes to this branch in the order of labels, following Corollary 2.5. Note that it is not always

possible to add the node of lowest label, and oftentimes we need to add a right leaf of higher label before we can add a left one of lower label.

The first node that we can add is the left child of the root, of label 1, as shown in Figure 9 left. Then we can add all 3 nodes in the empty spots on level 2 of the tree, as shown in the second tree in Figure 9. At this point, there are 3 spots available for nodes of label 4, and that is the lowest label that can be added, as shown in the third tree in Figure 9. The left-most node of label 4 would allow us to add 3 nodes of labels lower than 4. The one to its right would allow only the addition of one node of label 2. The right-most node of label 4 does not open any other spots on the same level.

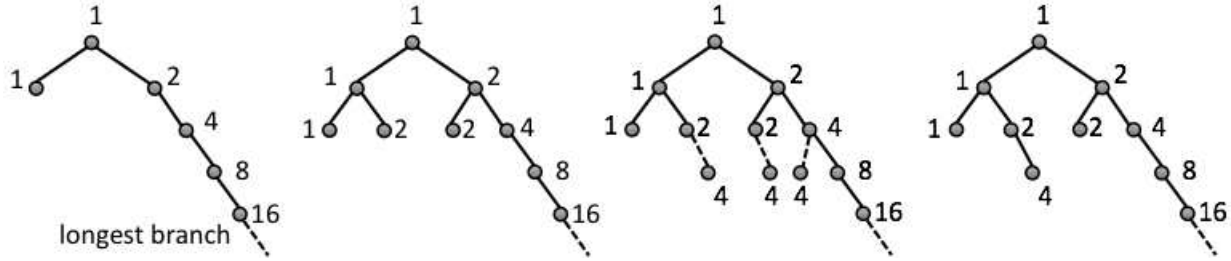

Figure 9: Incremental level addition in a K-optimal tree

It stands to reason that we should insert the left-most label 4 first, as shown in the right-most tree in Figure 9. After this insertion there are two spots at which a label 2 can be added. The left-most one allows us to add a node of label 1, while the other one doesn't. Thus we would insert the left-most 2, followed by a 1. Then we can insert the other 2 into level 3, as shown in Figure 10.

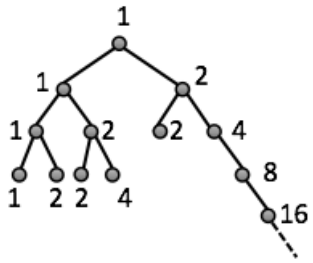

Figure 10: Incremental level addition in a K-optimal tree

Continuing a few more steps the same way, we notice that a structure emerges from the process, shown in Figure 11. We shall call it the *skeleton structure*. At every step in a new level, these nodes represent the ones that would open the most spots of lower labels out of all available spots of optimal label. This figure does not show all the nodes added on a level before the next one is started, but rather the initial structure that the rest of the nodes are added on. In fact, the first few levels in the tree are filled up completely

by the procedure. At some point it can become less expensive to start adding nodes on the next level down rather than continuing to complete all the upper levels. Theorem 3.6 indicates the level where this situation occurs.

The skeleton structure of the K-optimal tree we will construct will consist of the right-most branch of height h, the right-most branch of the left subtree, the right-most branch of the left subtree of the left subtree, and so on down the tree. Let's use g to denote the height of the left subtree, so that $g \leq h-1$. It follows that $g = O(log(n))$.

Note that the skeleton structure without the longest branch contains the first new nodes added to every new level. By trimming the whole tree at the level g, we only cut off $h-g$ number of nodes on the right-most branch, and their number is at most $h = \Theta(\log(n))$. Thus, this subtree of height g will contain at least $n-h+g$ nodes, and this number is asymptotic to n. Thus, $g \geq \lfloor \lg(n) \rfloor$ for n large enough. In general, $g = \Theta(\log(n))$. For the remaining of the proof, let us consider the skeleton structure to be trimmed at the level g.

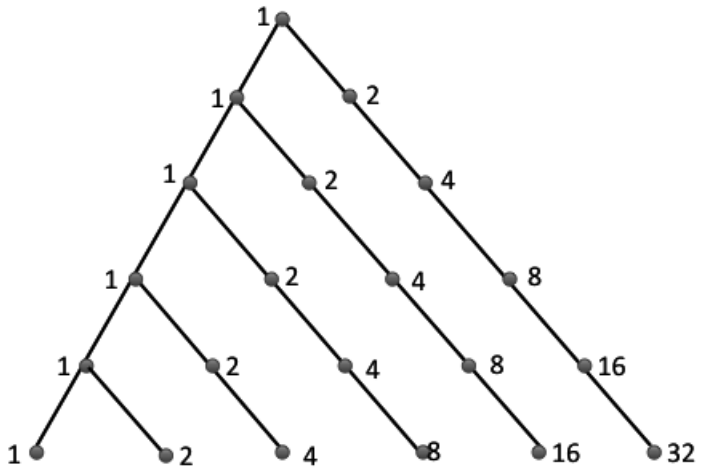

Figure 11: The skeleton structure for a tree of height 5

Let us now examine the contribution of the skeleton structure trimmed to level g in terms of number of nodes and sum of labels. The *number of nodes* in this structure is calculated by noting that it is composed of $g+1$ paths, starting from one composed of $g+1$ nodes and decreasing by 1 every time. So we have

$$\sum_{i=0}^{g} i = \frac{(g+1)(g+2)}{2} = \Theta((\log(n))^2).$$

The sum of labels can be computed by observing that on each of these paths, we start with a label equal

to 1, and then continue by incremental powers of 2 up to the length of the path. The sum of the labels on a path of length i is computed just like we did for the right-most branch, and is equal to $2^{i+1} - 1$. Thus, we can compute the total *sum of labels* as

$$\sum_{i=0}^{g}(2^{i+1} - 1) = 2^{g+2} - 2 - (g + 1) = 2^{g+2} - g - 3 = \Theta(n).$$

We can see that this skeleton structure contributes only $\Theta(n)$ to the sum of labels in the tree, which will not change its overall complexity, but it also uses only $\Theta((\log(n))^2)$ of the n nodes.

Minimal Node Placement. For the next part of the proof, we shall place the remainder of the nodes in this structure in order starting from the empty places of lowest possible label going up. These nodes are naturally placed in the tree while the skeleton structure is being built up, but for the purpose of the calculation, it is easier to consider them separately.

A simple observation is that the empty spots of lowest labels available right now are the left children of all the nodes labeled 2. For all of them, a branch on the right side is present, so we can add them without any changes to the labels in the tree. There are $g - 1$ such empty spots available, because the first of them is on level 2, as shown in Figure 12 left.

Next, by the same reasoning, we can add $g - 2$ left children of label 4. At the same time, we can add a right child of label 4 to every node added at the previous step with label 2, except for the lowest one. That is, we can add $g - 2$ right children, each having label 4, as shown in the $i = 2$ column of Figure 12. In addition, we can also add the $g - 2$ left children of the same parents. None of these additions causes any changes in the labels of the original nodes in Figure 11.

We can thus proceed in several steps, at each iteration adding nodes with labels going from 2 up to a power of 2 incrementing at every step. Let us examine one more step before we draw a general conclusion.

For the third step, we can add $g - 3$ nodes of label $8 = 2^3$. Next to this, we can add a complete third level to $g-3$ perfect subtrees added at the very first step, that have a root labeled 2, and a second complete level to $g - 3$ perfect subtrees of root labeled 4. This continues to grow the perfect subtrees started at the previous levels. The sum of labels on a level of a perfect tree is equal to a power of 3, but this quantity must also be multiplied by the label of the root in our case. Table 1 summarizes the nodes we have added and their total weight for the 3 steps we've examined so far. Figure 12 also illustrates this explanation.

From this table we can generalize that for the iteration number i we will have groups of nodes that can

Table 1: Nodes of lowest weight that can be added to the skeleton structure

Iteration	# Nodes	Weight
$i = 1$	$g - 1$	$2(g-1) = 2^1 \cdot 3^0(g-1)$
$i = 2$	$g - 2$	$4(g-2) = 2^2 \cdot 3^0(g-2)$
	$2(g-2)$	$6(g-2) = 2^1 \cdot 3^1(g-2)$
$i = 3$	$2^0(g-3)$	$8(g-3) = 2^3 \cdot 3^0(g-3)$
	$2^1(g-3)$	$2^2 \cdot 3^1(g-3)$
	$2^2(g-3)$	$2^1 \cdot 3^2(g-3)$

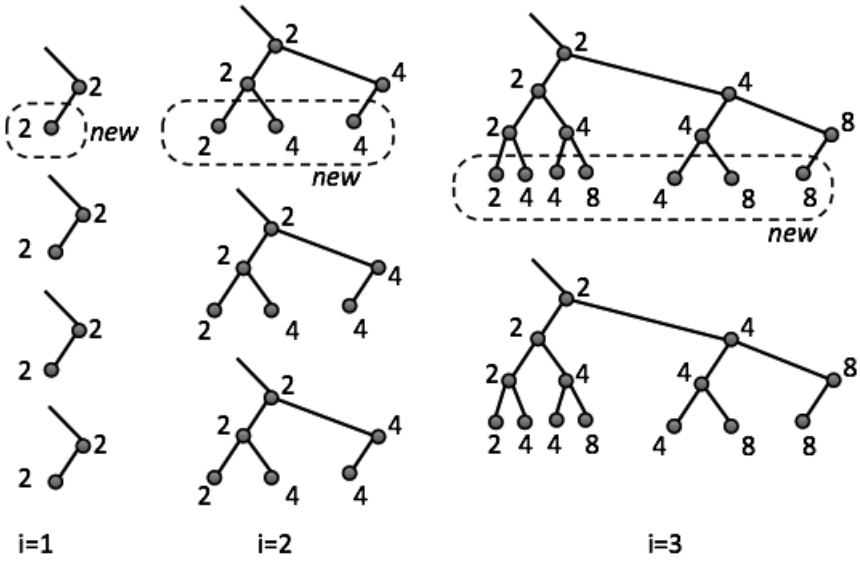

Figure 12: Nodes added to the skeleton structure in 3 steps for a tree of height 5

be added, with a count of $g - i$ groups in each category. For each category we will be adding the level k of a perfect tree that has a root labeled 2^{i-k}. The number of nodes in each such group is 2^k. The weight of each group is $2^{i-k} \cdot 3^k$.

Let us assume that to fill up the tree with the remainder of the nodes up to n, we need m such operations, and maybe another incomplete step after that. We can ignore that step for now, since it will not change the overall complexity. To find out what the total sum of labels is, we need to find a way to express m as a function of g or n.

The total number of nodes added at step i is $\sum_{k=0}^{i-1} 2^k(g-i) = (g-i)(2^i-1)$. If we add m such steps, then the total number of nodes that we've added is $\sum_{i=1}^{m}(g-i)(2^i-1)$. We need to find m such that this sum is approximately equal to $2^g - (g+1)(g+2)/2$, which is n from which we subtract the nodes in the skeleton structure. This is assuming that $g \approx \lg(n)$ and later we will address the case where g is approximately equal to a constant times $\lg(n)$, constant less than or equal to $\lg(3)$.

The total weight added in the step number i is

$$\sum_{k=0}^{i-1}(g-i)2^{i-k}3^k = 2(g-i)\sum_{k=0}^{i-1}2^{(i-1)-k}3^k = 2(g-i)2^{i-1}\sum_{k=0}^{i-1}\frac{3^k}{2^k} = 2^i(g-i)\sum_{k=0}^{i-1}\left(\frac{3}{2}\right)^k$$

We can use the formula $\sum_{k=0}^{i-1} x^k = \frac{x^i-1}{x-1}$ to compute the sum as

$$2^i(g-i)\frac{(3/2)^i-1}{(3/2)-1} = 2^i(g-i)\frac{3^i-2^i}{2^i}\frac{2}{3-2} = 2(g-i)(3^i-2^i)$$

To compute the number of nodes, we will need the following known sum, valid for all positive integers p and real numbers $t \neq 1$,

$$1+2t+3t^2+\ldots+pt^{p-1} = \sum_{i=1}^{p} it^{i-1} = \frac{1+pt^{p+1}-(p+1)t^p}{(t-1)^2}$$

We can rewrite our sum as

$$\sum_{i=1}^{m}(g-i)(2^i-1) = 2^g\sum_{i=1}^{m}(g-i)\frac{1}{2^{g-i+1}} - \sum_{i=1}^{m}(g-i) \quad .$$

By making the change of variable in both sums $j = g-i$, we have

$$2^g\sum_{j=g-m}^{g-1} j\frac{1}{2^{j+1}} - \sum_{j=g-m}^{g-1} j = 2^{g-2}\sum_{j=g-m}^{g-1} j\frac{1}{2^{j-1}} - \frac{(m-1)(2g-m-1)}{2}$$

Let us compute the sum in the last expression separately.

$$\sum_{j=g-m}^{g-1} j\frac{1}{2^{j-1}} = \sum_{j=1}^{g-1} j\frac{1}{2^{j-1}} - \sum_{j=1}^{g-m-1} j\frac{1}{2^{j-1}} =$$

$$\frac{1+(g-1)(1/2)^g - g(1/2)^{g-1}}{(1/2-1)^2} - \frac{1+(g-m-1)(1/2)^{g-m} - (g-m)(1/2)^{g-m-1}}{(1/2-1)^2}$$

The two fractions have common denominator $1/4$, so we combine the numerators. The leading 1s cancel each other. We can factor out $1/2^g$ from the remaining terms to obtain

$$\frac{4}{2^g}((g-1)-2g-(g-m-1)2^m+(g-m)2^{m+1}) = \frac{1}{2^{g-2}}((g-1)-2g-(g-m-1)2^m+(g-m)2^{m+1})$$

$$= \frac{1}{2^{g-2}}(2^m(g-m+1)-g-1).$$

By replacing it back into the original formula, the number of nodes is equal to

$$2^m(g-m+1)-g-1-\frac{(m-1)(2g-m-1)}{2} = \Theta(2^m(g-m)).$$

Given the similarity between the two sums, we obtain that the total weight of the nodes in the tree is

$$\Theta((3^m-2^m)(g-m)) = \Theta(3^m(g-m)).$$

Coming back to the question of expressing m as a function of g, if we write

$$(g-m+1)2^m = 2^g \Leftrightarrow g-m+1 = 2^{g-m}$$

and then introduce $r = g-m$, we have the equation $r+1 = 2^r$ which has the solutions $r = 0$ and $r = 1$. Figure 13 shows the graph of the function $2^x - x$ in the interval $[-1, 3]$.

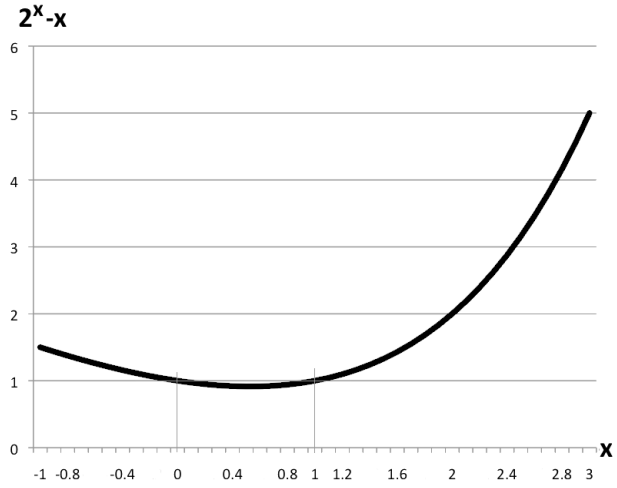

Figure 13: The graph of the function $2^x - x$

The first solution would mean that the tree is almost perfect, and we have proved before that perfect trees are not K-optimal. So we can conclude that $m = g-1$. Considering that the last level of the skeleton structure itself may be incomplete, this means that for g large enough, only 1 or 2 levels beyond the last may not be complete in the tree trimmed at the level g.

To examine the relationship between m and g further, let us assume that $g \approx d\lg(n)$, where $1 \leq d \leq$

$\lg(3) \approx 1.585$. Then we can write $n \approx 2^{g/d}$. Going back to the formula computing the number of nodes in the tree, we have

$$2^m(g - m + 1) \approx 2^{g/d}$$

from which we can write

$$g - m + 1 \approx 2^{g/d-m} = 2^{g-m+(g/d)-g} = \frac{2^{g-m}}{2^{g(d-1)/d}}.$$

Again, making the substitution $x = g - m$, we get

$$2^{g(d-1)/d} \approx \frac{2^x}{x+1}.$$

Remembering that $g \approx d\lg(n)$, we can write

$$n^{d(d-1)/d} = n^{d-1} \approx \frac{2^x}{x+1} \quad \text{or} \quad n \approx \left(\frac{2^x}{x+1}\right)^{1/(d-1)},$$

where $0 \leq d - 1 \leq 0.585$.

Let us write $f(y) = \frac{2^y}{y+1}$ and start with the observation that this function is monotone ascending for $y \geq 1$. Let us examine the hypothesis that $f(b\lg(n)) > f(x)$ for some constant b to be defined later. The hypothesis is true if and only if

$$f(b\lg(n)) = \frac{2^{b\lg(n)}}{b\lg(n)+1} = \frac{n^b}{b\lg(n)+1} > f(x) \approx n^{d-1}$$

which is equivalent to

$$\frac{n^b}{b\lg(n)+1} > n^{d-1} \Leftrightarrow n^{b-d+1} > b\lg(n) + 1.$$

Since a positive power of n grows faster than the logarithm in any base of n, we can say that the inequality above is true for any constant $b > d - 1$. So we can choose a constant b, $d - 1 < b < d$, such that $f(x) < f(b\lg(n))$. By the monotonicity of the function, this implies that $x < b\lg(n)$, which means that $g - m < b\lg(n)$, and considering that $g \approx d\lg(n)$, we can say that $(d - b)\lg(n) < m \leq \lg(3)\lg(n)$, from which we can conclude that $m = \Theta(log(n))$.

Coming back to the formula computing the weight as $\Theta(3^m(g - m))$, based on the result that $m =$

$\Theta(log(n))$, we can conclude that the complexity of the function is minimal in the case where the value of $g-m$ is a constant, and that this complexity is indeed $\Theta(n^{\lg(3)})$ in this case. While this does not necessarily mean that $g - m = 1$, the difference between the two numbers must be a constant.

Now we can examine how many nodes we can have on the longest branch in the tree beyond the level of the skeleton structure. One node can be expected, for example in those cases where a perfect tree is K-optimal for small values of n, and a new node is added to it. If more nodes are present on the same branch, those node will have labels incrementing exponentially and larger than any empty spots still available on lower levels. They can easily be moved higher in the tree to decrease the total weight. Thus, we can deduce that $g = h$ or $g = h - 1$.

The weight of the tree, and thus the complexity of the K function, is the order of $\Theta(3^h) = \Theta(3^{\lg(n)}) = \Theta(n^{\lg(3)})$. ■

It is interesting to note that this is also the order of complexity of the function K on perfect trees and on size-balanced trees, even though neither of them is K-optimal in general.

5 Conclusion

In this paper we have studied the complexity of a special class of recursive functions traversing binary trees. We started with a recurrence relation describing this complexity in the general case. We continued with a simple analysis of the worst case complexity, which turned out to be exponential. Next, we showed two particular types of trees that give us a complexity of $\Theta(n^{\lg(3)})$.

Finally, after discussing a few more properties of the K-optimal trees that minimize the complexity function over the trees with a given number of nodes, we showed a constructive method to build these trees. In the process we have also shown that the complexity of the function on these trees is also $\Theta(n^{\lg(3)})$, which concludes the study of this function.

We can conclude from this analysis that any method that allows us to avoid repeating recursive calls will significantly improve the complexity of a function in all the cases.