Complexity of an efficient implementation of ADT DisjointSets

ADT DisjointSets (aka. Partition or Union-Find) is important in many algorithms, starting with MWST (Kruskal) through Depth Determination. Its operations are:

- **MakeSet**: Element -> Set
- **Find**: Element -> Set
- **Union**: Set x Set -> Set

Sets are implemented as trees of elements, with each non-root element $x$ having a unique other element as parent, $p(x)$; for the root $y$, $p(y) = y$. The root of the tree representing a set holds information about the set, e.g., the set’s name, size, rank.

Elements are assigned *ranks* which are defined as 0 for leaves and are incremented by 1 for the root of the result of **Union** of two sets rooted in elements of equal rank. When **Union** is applied to two sets represented by trees rooted at $x$ and $y$, respectively, with $r(x) \leq r(y)$, the result is the tree rooted at $y$, $p(x)$ assigned the value of $y$. This rule is called **Union by rank** and guarantees logarithmic bound on the values of rank, which is also the (tight) upper bound on the height of the tree (the length of a longest path to the root.) With other words, the number $n_r$ of nodes in a tree rooted at a node of rank $r$ is at least $2^r$. (You know the name of the tree with the least nodes: what is it?)

The main and somewhat unexpected reason for an efficient implementation of **Find** is the *path compression*, where all elements on the find-path gain a new value of their parent — the set’s root. (New for all but the penultimate element on the path, the root’s child.) From these implementations of **Union** and **Find** it follows that ranks are strictly increasing along any find-path.

While it’s obvious that **Union** (by rank) can be implemented in constant time, the worst case time complexity of **Find** is equally obviously logarithmic. Amortized over all operations, however, **Find** proves to be almost constant. (A separate construction of a bad case shows that it is *not* constant.)

We can fairly easily show that the *iterated logarithm*, $\log^* n$, bounds the amortized complexity of **Find**. (Defining $\log^{(i)} n$ to be the result of taking logarithm $i$ times, $\log^* n$ equals to the smallest $i$ such that $\log^{(i)} n \leq 1$.)

The *accounting scheme* used in the proof distributes the charges for traversing parent pointers along the find-path. For that we need the concept of *rank groups*, the group $g$ being the interval of rank values $B(g - 1) \leq r < B(g)$. (Like HAL and IBM, $B$ could remind you of the diagonal Ackermann function $A$.)
The constant time computation represented by traversing a pointer $p(x)$ is charged to the element $x$ if and only if $r(x)$ and $r(p(x))$ belong to the same rank group and $p(x)$ is not the root. Otherwise, the traversal is charged to the $\text{Find}$ operation.

Before we determine the values of $B(g)$, we note that we have to trade-off ("balance") the charges against $\text{Find}$ and charges against the $n$ elements. Therefore, we need to design relatively few rank groups, and those groups with more elements should have fewer ranks.

For nodes in a generic rank group $g$ we have maximum charges $B(g) - B(g - 1)$ levied against each of $\sum_{B(g-1) \leq r < B(g)} n/2^r$ nodes of rank in rank group $g$ ($n/2^r$ is the maximum possible number of nodes with rank $r$). After factoring out $n/2^{B(g-1)}$ the total number of charges for nodes in this rank group is bound by

$$n(B(g) - B(g - 1))/2^{B(g-1)} \sum_{B(g-1) \leq r < B(g)} 1/2^{r-B(g-1)}$$

Thus, after summing over all rank groups $g$, we have the total charges against all nodes bound from above by

$$n \sum_{g} B(g)/2^{B(g-1)} \sum_{i \geq 0} 1/2^{i} = 2n \sum_{g \geq 1} B(g)/2^{B(g-1)}$$

The terms of the summation will be made equal to 1 if we define $B(g) = 2^{B(g-1)}$, for $g > 0$ and $B(0) = 0$. $B(g - 1) < \log n$ (the maximum rank) and the summation equals the number of rank groups (the inverse of $B(.)$) which is the iterated logarithm $\log^* n$. Thus the amortized charge per node to be $O(\log^*(\log n))$, the number of rank groups, which is also equal to the charges against a $\text{Find}$ operation. This shows an upper bound on the amortized time complexity of $\text{Find}$ to be $O(\log^* n)$.

What I like about this simple proof is that it balances on the one hand charges against elements and $\text{Find}$s, and on the other contributions to these charges from elements with ranks in different rank groups ("charge more to few and less to many").