Chapter 8

Disjoint Sets

In order to motivate the topic of this chapter, let us consider the following problem. We want to design an algorithm to schedule a set of jobs on a single server. Each job requires one unit of execution time and has its own deadline. We must assign a job with deadline \( d \) to some time slot \( t \), where \( 1 \leq t \leq d \). Furthermore, no two jobs can be assigned to the same time slot. If we can’t find a time slot for some jobs, we simply won’t schedule them.

One way to construct such a schedule is to assign each job in turn to the latest available time slot prior to its deadline, provided there is such a time slot. The challenge here is to find an efficient way of locating the latest available time slot prior to the deadline.

One way to think about this problem is to partition the time slots into \textit{disjoint sets} — i.e., a collection of sets such that no two sets have any element in common. In this case, each set will contain a nonempty range of time slots such that the first has not been assigned to a job, but all the rest have been assigned to jobs. In order to be able to handle the case in which time slot 1 has been assigned a job, we will also include a time slot 0, which we will consider to be always available.

Suppose, for example, that we have scheduled jobs in time slots 1, 2, 5, 7, and 8. Each set must have a single available time slot, which must be the smallest time slot in that set; thus, the elements 0, 3, 4, 6, and all elements greater than 8 must be in different sets and must each be the smallest element of its set. If 10 is the latest deadline, our disjoint sets will therefore be \{0, 1, 2\}, \{3\}, \{4, 5\}, \{6, 7, 8\}, \{9\}, and \{10\}. If we then wish to schedule a job with deadline 8, we need to find the latest available time slot prior to 8. This is simply the first time slot in the set containing 8 — namely, 6. Thus, in order to find this time slot, we need to be able to
determine which set contains the deadline 8, and what is the first time slot in that set. When we then schedule the job at time slot 6, the set \{6, 7, 8\} no longer contains an available time slot. We therefore need to merge the set \{6, 7, 8\} with the set containing 5, namely, \{4, 5\}.

The operations of finding the set containing a given element and merging two sets are typical of many algorithms that manipulate disjoint sets. The operation of finding the smallest element of a given set is not as commonly needed, so we will ignore this operation for now; however, as we will see shortly, it is not hard to use an array to keep track of this information. Furthermore, we often need to manipulate objects other than Nats; however, we can always store these objects in an array and use their indices as the elements of the disjoint sets. For this reason, we will simplify matters by assuming that the elements of the disjoint sets are the Nats 0..\(n - 1\). In general, the individual sets will be allowed to contain non-consecutive integers.

The DisjointSets ADT, shown in Figure 8.1, specifies the data structure we need. Each of the sets contains an element that is distinguished as its representative. The Find operation simply returns that representative. Thus, if two calls to Find return the same result, we know that both elements belong to the same set. The Merge operation takes two representatives, combines the sets identified by these elements, and returns the resulting set’s representative. In this chapter, we will consider how the DisjointSets ADT can be implemented efficiently. Before we do this, however, let us take a closer look at how the DisjointSets ADT can be used to implement the scheduling algorithm outlined above.

### 8.1 Using DisjointSets in Scheduling

We will use an instance of the DisjointSets ADT to maintain the disjoint sets in the scheduling algorithm outlined above. In order to find the time slot in which to schedule a job, we first need to find the time interval containing the last time slot prior to the job’s deadline. We can use the Find operation for this purpose. Assuming that we can obtain the available time slot in a given partition, and assuming this time slot is not 0, we have the time slot \(i\) in which to schedule the job. We then need to combine this partition with the one immediately preceding it. We can find the preceding partition with Find\((i - 1)\). We can then combine the two partitions using Merge.

We need one additional data structure in order to be able to find the available slot in an interval, given the representative of that interval. For this
Figure 8.1 The DisjointSets ADT

Precondition: $n$ is a positive Nat.
Postcondition: Constructs a DisjointSets object in which each element in $0..n-1$ forms a singleton set and is the representative of that set.

DisjointSets($n$)

Precondition: $k$ is a Nat less than the size of the universe of elements.
Postcondition: Returns the representative of the partition containing $k$.

DisjointSets.Find($k$)

Precondition: $i$ and $j$ are representatives of two different partitions.
Postcondition: Merges the partitions containing $i$ and $j$ into a new partition and returns the representative of the new partition.

DisjointSets.Merge($i, j$)

Purpose, we can use an array $avail[0..n]$ such that if $j$ is the representative of an interval, then $avail[j]$ is the available time slot in that interval. Initially, $avail[i] = i$ for all $i$. Suppose we schedule a job at time $i$. We then merge the interval containing $i$ with the preceding interval. Let $j$ be the representative of the preceding interval prior to the merger. Then $avail[j]$ is the available element in the resulting interval. If $k$ is the value returned by the call to Merge, then we can update $avail$ by assigning to $avail[k]$ the value $avail[j]$. Because no other representatives change, no other updates are needed. The entire algorithm is shown in Figure 8.2.

8.2 A Tree-Based Implementation

In this section, we will consider a tree-based implementation of the DisjointSets ADT, as illustrated in Figure 8.3. Each partition will be represented by a tree. The nodes of the tree will be the elements of the partition. The element at the root of the tree will be the representative of that partition. Because we will need to find the root from an arbitrary node in the tree, the children will maintain references to their parents, rather than vice versa. Note that because parents do not need to reference children, a node can have arbitrarily many children.

Given a value $k$, we need to be able to find the parent of the node
CHAPTER 8. DISJOINT SETS

Figure 8.2 Scheduling algorithm using DisjointSets

Precondition: deadlines[1..m] contains positive Nats no larger than n.
Postcondition: Returns an array sched[1..n] of Nats no larger than m such that if sched[i] > 0, then i ≤ deadlines[sched[i]].

Scheduler(deadlines[1..m], n)

    sched ← new Array[1..n]; avail ← new Array[0..n]
    intervals ← new DisjointSets(n + 1); avail[0] ← 0
    // Invariant: For 1 ≤ j < i, sched[j] = 0 and avail[j] = j.
    for i ← 1 to n
        sched[i] ← 0; avail[i] ← i
        // Invariant: For 1 ≤ s ≤ n, sched[s] ≤ m and if sched[s] > 0, then
        // s ≤ deadlines[sched[s]]. Each partition in intervals contains a range
        // of natural numbers u, u + 1, . . . u + d such that sched[u] = 0,
        // sched[u + v] > 0 for 1 ≤ v ≤ d, and for the representative w of this
        // partition, avail[w] = u.
        for i ← 1 to m
            k ← intervals.Find(deadlines[i]); t ← avail[k]
            if t ≠ 0
                sched[t] ← i; j ← intervals.Find(t − 1)
                k ← intervals.Merge(j, k); avail[k] ← avail[j]
        return sched

representing k. In order to accommodate this functionality, we will use an array parent[0..n − 1] to represent the trees. Specifically, parent[k] will give the parent of k in its tree, or if k is the root, parent[k] will be k. Thus, parent will be the only representation variable. Our structural invariant will be that for 0 ≤ i < n:

- 0 ≤ parent[i] < n; and
- there is a finite sequence parent[i], parent[parent[i]], ..., k such that parent[k] = k.

Note that this invariant implies that the values in the universe are grouped into trees, where the root k of a tree is denoted by parent[k] = k.

The Merge and Find operations are now straightforward. Merge simply makes one tree a child of the root of the other, and Find follows the
Figure 8.3 A tree-based implementation of DisjointSets.

parent references until the root is reached. The full implementation is shown in Figure 8.4.

Clearly, the constructor operates in $\Theta(n)$ time, and Merge operates in $\Theta(1)$ time. The number of iterations of the while loop in Find is the depth of $k$, which in the worst case is the height of the tree. Clearly, the height is at most $n - 1$. Unfortunately, this height can be achieved by the sequence

$$\text{Merge}(0, 1), \text{Merge}(1, 2), \ldots, \text{Merge}(n - 2, n - 1).$$

Thus, the worst-case running time for Find is in $\Theta(n)$.

Because the constructor will only be executed once for each structure, $\Theta(n)$ is not a bad running time. However, we will probably need to execute Find repeatedly. We would therefore like to improve its performance. In the next section we will examine a simple way to do this.

8.3 A Short Tree Implementation

Because the worst-case running time for Find is proportional to the height of the tree, we can improve the worst-case performance by controlling heights of the trees. In order to accomplish this, when we merge two trees, we will always make the tree with smaller height the child of the root of the other tree. If both trees have the same height, we arbitrarily choose one as the child. Note that by using this technique, the only way we can increase the height of a tree is to merge it with another tree of the same height. We will show that as a result, the heights of the trees are always at most logarithmic in their number of nodes.
CHAPTER 8. DISJOINT SETS

Figure 8.4  

TreeDisjointSets implementation of DisjointSets

Structural Invariant: For $0 \leq i < n$, $0 \leq \text{parent}[i] < n$ and there is a finite sequence $\text{parent}[i], \text{parent}[\text{parent}[i]], \ldots, k$ such that $\text{parent}[k] = k$.

TreeDisjointSets($n$)

$\text{parent} \leftarrow \text{new} \ \text{Array}[0..n - 1]$

// Invariant: For $0 \leq j < i$, $\text{parent}[j] = j$.

for $i \leftarrow 0$ to $n - 1$

\[ \text{parent}[i] \leftarrow i \]

TreeDisjointSets.Find($k$)

\[ i \leftarrow k \]

// Invariant: $i$ is an ancestor of $k$.

while $\text{parent}[i] \neq i$

\[ i \leftarrow \text{parent}[i] \]

return $i$

TreeDisjointSets.Merge($i, j$)

if $i = j$ or $i \neq \text{parent}[i]$ or $j \neq \text{parent}[j]$

error

else

\[ \text{parent}[i] \leftarrow j \]

return $j$

As we did in analyzing the heights of AVL trees (Section 6.2), let us compute the minimum number of nodes required to achieve a tree of height $h$, assuming we always make the tree with smaller height the child. Let $f(h)$ give this number. Then $f(0) = 1$. In order to build a tree with height $h > 0$ using the fewest nodes, we must merge two trees of height $h - 1$, each having the fewest nodes possible. Thus, each of the two merged trees must have $f(h - 1)$ nodes. The total number of nodes is given by the recurrence

\[ f(h) = 2f(h - 1). \]

It is easily seen that $f(h) = 2^h$, so that $h = \log f(h)$. Thus, if $k$ is the
number of nodes in a tree of height $h$, we have

$$h = \lg f(h) \leq \lg k.$$  

We conclude that if the universe contains $n$ elements, then no tree has a height greater than $\lg n$.

In order to be able to merge trees in this way, we need to keep track of the height of each tree. For this purpose, we include an additional representation variable $height[0..n−1]$. Our structural invariant will then be that for $0 \leq i < n$, $height[i]$ is the maximum of 0 and $height[j] + 1$ for all $j \neq i$ such that $parent[j] = i$, and that $0 \leq parent[i] < n$. Note that because $height[parent[i]] > height[i]$ whenever $parent[i] \neq i$, each value $i$ must have an ancestor $j$ such that $parent[j] = j$; thus, the elements of the universe must be grouped into trees rooted at nodes with $parent[i] = i$.

The constructor and `Merge` operation for this implementation are shown in Figure 8.5. The `Find` operation is implemented exactly as `TreeDisjointSets.Find` in Figure 8.4.

As in the previous implementation, the constructor runs in $\Theta(n)$ time, and `Merge` runs in $\Theta(1)$ time. Based on the above discussion, `Find` runs in $\Theta(\lg n)$ time.

### 8.4 * Path Compression*

In order to improve the performance of a `Find`, we would like to decrease the distance from a node to the root as much as possible. An effective way of accomplishing this is to modify the `Find` so that it changes the parent of every node it encounters to be the root of its tree. This technique is called *path compression*. We implement this technique using a recursive internal function, `Compress`. If the given node $k$ is not the root, `Compress` first performs a recursive `Compress` on $k$’s parent. If we treat the distance between a node and the root as the size of a call to `Find`, we see that such a recursive call is valid. Furthermore, it compresses the path from $k$’s parent to the root and returns the root. It can therefore complete its task by making $k$ a child of the root and returning the root. The resulting `Find` algorithm is shown in Figure 8.6.

The rest of the implementation is the same as for `ShortDisjointSets`; however, because path compression can decrease the height of a node $i$ without updating $height[i]$, this value is no longer guaranteed to give the height of node $i$. For this reason, we will change the name of this array
CHAPTER 8. DISJOINT SETS

Figure 8.5 ShortDisjointSets implementation of DisjointSets

**Structural Invariant:** For $0 \leq i < n$, $\text{height}[i]$ is the maximum of 0 and $\text{height}[j] + 1$ for all $j \neq i$ such that $\text{parent}[j] = i$, and $0 \leq \text{parent}[i] < n$.

```plaintext
ShortDisjointSets(n)
    parent ← new Array[0..n - 1]; height ← new Array[0..n - 1]
    for i ← 0 to n - 1
        parent[i] ← i; height[i] ← 0

ShortDisjointSets.Merge(i, j)
    if i = j or i ≠ parent[i] or j ≠ parent[j]
        error
    else if height[i] ≤ height[j]
        parent[i] ← j
        if height[i] = height[j]
            height[j] ← height[j] + 1
        return j
    else
        parent[j] ← i
    return i
```

to rank and weaken the structural invariant so that rank[i] is at least the height of i. The precise meaning of rank[i] is now rather elusive, other than the fact that it gives an upper bound on the height of i.

Clearly, the running time of CompressedDisjointSets.Find is in $O(h)$, where $h$ is the height of the tree. Furthermore, the height of a tree in this implementation can certainly be no larger than it would have been if no path compression had been done; hence, the upper bound of $O(\log n)$ shown for ShortDisjointSets.Find also holds for CompressedDisjointSets.Find. Because we can still construct a tree with height in $\Theta(\log n)$ with a sequence of Merges, we conclude that CompressedDisjointSets.Find runs in $\Theta(\log n)$ time in the worst case. Clearly, CompressedDisjointSets.Merge runs in $\Theta(1)$ time, so that in the worst case, the asymptotic performance of CompressedDisjointSets is identical to that of ShortDisjointSets.

On the other hand, because each path compression has the tendency
Figure 8.6 The Find algorithm for the CompressedDisjointSets implementation of DisjointSets

**Structural Invariant:** For 0 \leq i < n, 0 \leq parent[i] < n and there is a finite sequence \( parent[i], parent[parent[i]], \ldots, k \) such that \( parent[k] = k \).

```plaintext
CompressedDisjointSets.Find(k)
    return Compress(k)
```

// Internal function

**Precondition:** \( k \) is a Nat less than the size of the universe of elements, and the structural invariant holds.

**Postcondition:** Returns the representative of the partition containing \( k \), and makes each other element that was an ancestor of \( k \) a child of the representative. The structural invariant holds.

```plaintext
CompressedDisjointSets.Compress(k)
    if parent[k] \neq k
        parent[k] \leftarrow Compress(parent[k])
    return parent[k]
```

to improve the performance of subsequent Finds, we might suspect that the amortized performance of this structure is improved. Indeed, we will show that the amortized performance of the Merge and Find operations for CompressedDisjointSets is “almost” constant.

In order to perform an amortized analysis, we need to assign actual costs to the operations. The running time of \( \text{Find}(i) \) is proportional to the number of nodes on the path from \( i \) to the root of its tree. Another way to say this is that the running time is proportional to the number of locations of the parent array that are accessed. Let us therefore define the actual cost of an operation to be the number of locations accessed in the parent array. The actual cost of Merge is therefore 2.

The key to a good amortized analysis is finding a good potential function. Let \( S \) be the set of all possible states of some CompressedDisjointSets with size \( n \). We need a function \( \Phi : S \rightarrow \mathbb{R}^{\geq 0} \) such that the initial state is mapped to 0. To arrive at this function, we will define, for a given state \( s \in S \), a potential for each node; i.e., we will define a function \( \phi_s : U \rightarrow \mathbb{N} \),
where \( U = \{ i \in \mathbb{N} \mid i < n \} \). We will then define the potential function to be the sum of the potentials of all of the nodes:

\[
\Phi(s) = \sum_{i=0}^{n-1} \phi_s(i).
\]

Let \( rank_s \) and \( parent_s \) denote the values of the \textit{rank} and \textit{parent} arrays in state \( s \). We will define our potential function based on these values. Let \( \epsilon \) denote the initial state. Thus, for \( 0 \leq i < n \), \( rank_\epsilon[i] = 0 \). In order for \( \Phi \) to be a valid potential function, we need \( \Phi(\epsilon) = 0 \). To accomplish this, we let \( \phi_s(i) = 0 \) if \( rank_s[i] = 0 \) for \( 0 \leq i < n \) and any state \( s \). Note that a node can only obtain a nonzero rank when a \texttt{Merge} makes it the parent of another node; thus, \( rank_s[i] = 0 \) iff \( i \) is a leaf.

We have two operations we need to consider as we define \( \phi_s(i) \) for non-leaf \( i \). \texttt{Merge} is a cheap operation, having an actual cost of 2, whereas \texttt{Find} is more expensive in the worst case. We therefore need to amortize the cost of an expensive \texttt{Find} over preceding \texttt{Merges}. This means that we need the potential function to increase by some amount — say \( \alpha(n) \), where \( \alpha \) is some appropriate function — for at least some of the \texttt{Merges}. The \texttt{Merge} operation only operates on roots, so let us focus our attention there. When a \texttt{Merge} is performed, the rank of one node — a root — may increase by 1, but otherwise, no ranks increase (see Figure 8.5, and recall that the \textit{rank} array replaces the \textit{height} array in this algorithm). Therefore, let us define \( \phi_s(i) \) to be \( \alpha(n) \cdot rank_s[i] \) if \( i \) is a root (i.e., if \( parent_s[i] = i \)).

Note that the above definitions are consistent with each other: if \( i \) is both a leaf and a root, its rank is 0, and hence its potential is 0 by either of the above definitions. Furthermore, if we can ensure that a \texttt{Merge} causes the potential of no node other than the root of the resulting tree to increase, we will have a bound of \( \alpha(n) + 2 \) on the amortized cost of \texttt{Merge} with respect to \( \Phi \). We still need to define the potentials for nodes that are neither leaves nor roots in such a way that an expensive \texttt{Find} causes \( \Phi \) to decrease enough to offset much of the actual cost.

Consider the effect of \texttt{Find}(\( j \)) on a node \( i \) that is neither a leaf nor a root. \( i \)'s rank doesn't change, but its parent may. In particular, it may receive a parent with different rank than its original parent. It is not hard to show as a structural invariant that if \( parent[i] \neq i \), then \( rank[parent[i]] > rank[i] \). We would therefore like the potential of \( i \) to decrease, generally speaking, as \( rank[parent[i]] \) increases. Furthermore, in order that its potential does not increase when a \texttt{Merge} changes it from a root to a non-root, we should have \( \phi_s(i) \leq \alpha(n) \cdot rank_s[i] \) for all states \( s \) and nodes \( i \).
As a first approximation to a definition of $\phi_s(i)$, where $i$ is neither a leaf nor a root, suppose we let $\phi_s(i) = (\alpha(n) - f(s,i)) \text{rank}_s[i]$, where $f$ is some function that depends on the ranks of $i$ and its parent in $s$ and obeys the constraint

$$0 \leq f(s,i) < \alpha(n).$$

If $f$ does not decrease when $i$ receives a parent of higher rank, $\phi_s$ would satisfy the constraints outlined above. However, it would suffer the disadvantage that its value is always a multiple of $\text{rank}_s[i]$. In order to give us more control over how much the potential function changes, we would like for this function to have a larger range when $\text{rank}_s[i]$ is fixed, as it is when $i$ is not a root. Therefore, we wish to find, for each state $s$, a function

$$\phi_s(i) = (\alpha(n) - f(s,i)) \text{rank}_s[i] - g(s,i),$$

where $f$ is as above and $g$ is some function that depends on the ranks of $i$ and its parent in $s$ and obeys the constraint

$$0 < g(s,i) \leq \text{rank}_s[i].$$

Let us consider the nodes examined during the operation FIND($j$) on some state $s$. These nodes are the ancestors of $j$. Let us restrict our attention to those ancestors that have nonzero rank and are not the root. (Note that as a result, we ignore at most two ancestors.) Let $A_k$ be a function describing the relationship between the ranks of $i$ and its parent for any non-root $i$, where $k = f(s,i)$; i.e., $\text{rank}_s[parent_s[i]] \geq A_{f(s,i)}(\text{rank}_s[i])$. When the path compression is done, the parent of $i$ may have the same rank (if $i$ had already been a child of the root), or it may be very little more than the rank of $i$’s original parent. However, suppose that in the original state $s$, $i$ has a proper ancestor $i'$ other than the root such that $f(s,i') = f(s,i) = k$. Because $i'$ is not a root, the rank of $i'$’s new parent is at least the rank of the original parent of $i'$. Therefore, if $A_k$ is nondecreasing, we have

$$\text{rank}_{s'}[parent_{s'}[i']] \geq A_k(\text{rank}_s[i']) \geq A_k(\text{rank}_s[parent_s[i]]).$$

Thus, each time a path compression moves a non-leaf $i$ having a proper ancestor $i'$ (other than the root) with $f(s,i) = f(s,i') = k$, the rank of $i'$’s parent increases by at least the application of the function $A_k$. Furthermore, because $0 \leq f(s,i) < \alpha(n)$, there can be at most $\alpha(n)$ nodes $i$ on the path.
from $j$ to the root, other than leaves or the root, that do not have a proper ancestor $i'$ with $f(s, i') = f(s, i)$. Thus, if we can decrease the potential of each $i$ having a proper ancestor $i'$ with $f(s, i') = f(s, i)$, without increasing any potentials, then we will have a bound of $\alpha(n) + 2$ on the amortized cost of a FIND with respect to $\Phi$.

The behavior described above gives us some insight into how we might define $f$, $g$, and each $A_k$. First, we would like $g$ to give the maximum number of times $A_{f(s, i)}$ can be applied to the rank of $i$ without exceeding the rank of $i$’s parent. Thus, if $i$ has a proper ancestor $i'$ with $f(s, i') = f(s, i)$, and if $f(s', i) = f(s, i)$, then $g(s', i) > g(s, i)$. As a result, the potential of $i$ decreases. In order to keep $g(s, i)$ within the proper range, we should define $f(s, i)$ and $A_k$ so that if we apply $A_{f(s, i)}$ more than $\text{rank}_s(i)$ times to $\text{rank}_s(i)$, we must attain a value of at least $A_{f(s, i)} + 1$.

We now define:

$$A_k(n) = \begin{cases} 
  n + 1 & \text{if } k = 0 \\
  A_{k-1}(n) & \text{if } k \geq 1.
\end{cases}$$

We can then define, for each node $i$ that is neither a leaf nor a root,

$$f(s, i) = \max\{k \mid \text{rank}_s[\text{parent}_s[i]] \geq A_k(\text{rank}_s[i])\}$$

and

$$g(s, i) = \max\{k \mid \text{rank}_s[\text{parent}_s[i]] \geq A_{f(s, i)}(\text{rank}_s[i])\}.$$
We have shown that without path compression, the height of a tree never exceeds $\lg n$; hence, with path compression, the rank of a node never exceeds $\lg n$. It therefore suffices to define

$$\alpha(n) = \min\{k \mid A_k(1) > \lg n\}.$$

As the subscript $k$ increases, $A_k(1)$ increases very rapidly. We leave it as an exercise to show that

$$A_4(1) \geq 2^{2^{\cdot \cdot \cdot 2}},$$

where there are 2051 2s on the right-hand side. It is hard to comprehend just how large this value is, for if the right-hand side contained only six 2s, the number of bits required to store it would be $2^{65536} + 1$. By contrast, the number of elementary particles in the universe is currently estimated to be no more than about $2^{300}$. Hence, there is not nearly enough matter in the universe to store $A_4(1)$ in binary. Because $\alpha(n) \leq 4$ for all $n < 2^{A_4(1)}$, we can see that $\alpha$ grows very slowly.

To summarize, we define our potential function $\Phi$ so that

$$\Phi(s) = \sum_{i=0}^{n-1} \phi_s(i),$$

where

$$\phi_s(i) = \begin{cases} 
0 & \text{if } rank_s[i] = 0 \\
\alpha(n) \cdot \text{rank}_s[i] & \text{if } \text{parent}_s[i] = i \\
(\alpha(n) - f(s, i)) \cdot \text{rank}_s[i] - g(s, i) & \text{otherwise},
\end{cases}$$

for $\alpha$, $f$, and $g$ as defined above. Before we can complete the amortized analysis, we need to show that both $f$ and $g$ satisfy the properties outlined in the discussion above.

**Lemma 8.1** Let $s$ be a state of a CompressedDisjointSets of size $n$, and let $0 \leq i < n$ such that $\text{parent}_s[i] \neq i$ and $\text{rank}_s[i] > 0$. Then

$$0 \leq f(s, i) < \alpha(n).$$

**Proof:** First, because the rank of the parent of $i$ is strictly larger than that of $i$, we have

$$\text{rank}_s[\text{parent}_s[i]] \geq \text{rank}_s[i] + 1 = A_0(\text{rank}_s[i]),$$

where $A_0(\cdot)$ is the inverse of the Ackermann function.
from the definition of \( A_0 \). Thus, from the definition of \( f \), \( f(s,i) \geq 0 \).

It is not hard to show that each \( A_k \) is nondecreasing — we leave the details as an exercise. Using this fact, along with the definition of \( \alpha \) and the fact that no rank exceeds \( \lg n \), we have

\[
A_{\alpha(n)}(\text{rank}_s[i]) \geq A_{\alpha(n)}(1) \\
> \lg n \\
\geq \text{rank}_s[\text{parent}_s[i]].
\]

Thus, from the definition of \( f \), \( f(s,i) < \alpha(n) \). \(\square\)

**Lemma 8.2** Let \( s \) be a state of a CompressedDisjointSets of size \( n \), and let \( 0 \leq i < n \) such that \( \text{parent}_s[i] \neq i \) and \( \text{rank}_s[i] > 0 \). Then

\[
0 < g(s,i) \leq \text{rank}_s[i].
\]

**Proof:** First, from the definition of \( f \), we have

\[
\text{rank}_s[\text{parent}_s[i]] \geq A_{f(s,i)}(\text{rank}_s[i]) \\
= A_{f(s,i)}^{(1)}(\text{rank}_s[i]).
\]

Thus, from the definition of \( g \), \( g(s,i) > 0 \).

Now from the definitions of \( A \) and \( f \), we have

\[
A_{f(s,i)}^{(\text{rank}_s[i]+1)}(\text{rank}_s[i]) = A_{f(s,i)+1}(\text{rank}_s[i]) \\
> \text{rank}_s[\text{parent}_s[i]].
\]

Thus, \( g(s,i) \leq \text{rank}_s[i] \). \(\square\)

We are now ready to show that the amortized costs of \texttt{Merge} and \texttt{Find} are in \( O(\alpha(n)) \).

**Theorem 8.3** With respect to \( \Phi \), the amortized cost of \texttt{Merge} on a CompressedDisjointSets of size \( n \) is in \( O(\alpha(n)) \).

**Proof:** Suppose we do \texttt{Merge}(i,j) in state \( s \), yielding state \( s' \). Without loss of generality, assume \( j \) is made the parent of \( i \). Then \( i \) is the only node whose parent changes, and \( j \) is the only node whose rank may change;
hence, the potentials for all other nodes remain unchanged. The change in potential for node $i$ is given by
\[
\phi_{s'}(i) - \phi_s(i) = (\alpha(n) - f(s', i))\text{rank}_{s'}[i] - g(s', i) - \alpha(n)\text{rank}_s[i]
\]
\[
< \alpha(n)(\text{rank}_{s'}[i] - \text{rank}_s[i])
\]
\[
= 0.
\]
The change in potential for node $j$ is given by
\[
\phi_{s'}(j) - \phi_s(j) = \alpha(n)\text{rank}_{s'}[j] - \alpha(n)\text{rank}_s[j]
\]
\[
\leq \alpha(n),
\]
because the rank of $j$ can increase by at most 1. The change in $\Phi$ is therefore less than $\alpha(n)$. Because the actual cost is 2, the amortized cost is less than $\alpha(n) + 2 \in O(\alpha(n))$. □

**Theorem 8.4** With respect to $\Phi$, the amortized cost of \textsc{Find} on a \textsc{CompressedDisjointSets} of size $n$ is in $O(\alpha(n))$.

**Proof:** Suppose we perform \textsc{Find}(j) on state $s$. Let $s'$ be the resulting state. Suppose there are $d$ nodes on the path from $j$ to the root in $s$. Then the actual cost of the operation is $d$. We will show that as a result of this operation, at least $d - \alpha(n) - 2$ nodes decrease in potential, and no nodes increase in potential. As a result, we will have shown the amortized cost to be at most $\alpha(n) + 2 \in O(\alpha(n))$.

First, we will show that no potentials increase as a result of \textsc{Find}(j). Because the \textsc{Find} operation does not change any ranks and does not change which nodes are roots, no leaves or roots can change potential. The potential for any other node $i$ can change only due to changes in $f$ and $g$. Because $i$ cannot receive a parent with a smaller rank as a result of path compression, $f(s', i) \geq f(s, i)$. If $f(s', i) = f(s, i)$, then clearly $g(s', i) \geq g(s, i)$. In this case, the potential does not increase. If, on the other hand, $f(s', i) > f(s, i)$, from Lemma 8.2 and the fact that path compression leaves all ranks unchanged, $g(s, i) - g(s', i) < \text{rank}_s[i]$. Then
\[
\phi_{s'}(i) - \phi_s(i) = (\alpha(n) - f(s', i))\text{rank}_{s'}[i] - g(s', i) - (\alpha(n) - f(s, i))\text{rank}_s[i] + g(s, i)
\]
\[
< (f(s, i) - f(s', i))\text{rank}_s[i] + \text{rank}_s[i]
\]
\[
\leq 0.
\]
In this case, the potential of \( i \) decreases.

The only nodes whose parents change are ancestors of \( j \), and no ranks change. Hence, the only nodes whose potentials change are ancestors of \( j \). The ancestors of \( j \) include a root and at most one leaf. For the other ancestors \( i \), from Lemma 8.1, there can be at most \( \alpha(n) \) distinct values for \( f(s, i) \). For a given value \( k \), each node \( i \) with \( f(s, i) = k \) except the one nearest the root has a proper ancestor \( i' \) with \( f(s, i') = k \). We will show that all of these nodes — i.e., at least \( d - \alpha(n) - 2 \) of the \( d \) ancestors of \( j \) — decrease in potential as a result of the \texttt{Find}.

Let \( i \) be an ancestor of \( j \) in \( s \) such that \( i \) is neither a leaf nor a root and such that for some proper ancestor \( i' \) of \( i \) other than the root, \( f(s, i) = f(s, i') = k \). We have already shown that if \( f(s', i) > f(s, i) \), the potential of \( i \) decreases. Therefore, suppose \( f(s', i) = f(s, i) = k \). Then

\[
\begin{align*}
\text{rank}_{s'}[\text{parent}_{s'}[i]] & \geq \text{rank}_s[\text{parent}_s[i]] \\
& \geq A_k(\text{rank}_s[i']) \quad (\text{definition of } f) \\
& \geq A_k(\text{rank}_s[\text{parent}_s[i]]) \\
& \geq A_k(A_k^{g(s,i)}(\text{rank}_s[i])) \quad (\text{definition of } g) \\
& = A_k^{g(s,i)+1}(\text{rank}_s[i]) \\
& = A_k^{g(s,i)+1}(\text{rank}_{s'}[i]).
\end{align*}
\]

Because \( f(s', i) = k, g(s', i) > g(s, i) \), so that \( \phi_{s'}(i) < \phi_s(i) \).

The above theorems show that the amortized running times of \texttt{Merge} and \texttt{Find} are in \( O(\alpha(n)) \). However, \( \alpha \) appears to be a somewhat contrived function. We have argued intuitively that \( \alpha \) increases very slowly, but we have not formally compared it with any better-known slow-growing function like \( \lg \) or \( \lg \lg \). We address this issue more formally in the Exercises. For now, we will simply state that the collection of functions \( A_k \) form a variation of \textit{Ackermann’s function}, and that \( \alpha \) is one way of defining its inverse. There have actually been several different 2- or 3-variable functions that have been called Ackermann’s function, and all grow at roughly the same rapid rate.

### 8.5 Summary

Tree-based implementations of disjoint sets provide very efficient \texttt{Merge} and \texttt{Find} operations, particularly when path compression is used. The worst-case running times for these operations are in \( \Theta(1) \) and \( \Theta(\lg n) \), respectively, for both \texttt{ShortDisjointSets} and \texttt{CompressedDisjointSets}.
Figure 8.7 Comparison of running times of the DisjointSets operations for various implementations

<table>
<thead>
<tr>
<th></th>
<th>Find</th>
</tr>
</thead>
<tbody>
<tr>
<td>TreeDisjointSets</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>ShortDisjointSets</td>
<td>$\Theta(\log n)$</td>
</tr>
<tr>
<td>CompressedDisjointSets</td>
<td>$O(\alpha(n))$</td>
</tr>
</tbody>
</table>

Notes:

- $n$ is the number of elements in the universe of the sets.
- The constructor runs in $\Theta(n)$ worst-case time for each implementation.
- The Merge operation runs in $\Theta(1)$ worst-case time for each operation.
- Unless otherwise noted, all running times are worst-case.

The latter implementation yields nearly constant amortized running time. A summary of the running times of the operations for the different implementations is shown in Figure 8.7. As we will see in later chapters, these structures are very useful in the design of efficient algorithms.

8.6 Exercises

Exercise 8.1 Draw the trees that result from the following sequence of operations:

```java
    t ← new TreeDisjointSets(8)
    t.Merge(0, 1)
    t.Merge(t.Find(1), 2)
    t.Merge(3, 4)
    t.Merge(5, 6)
    t.Merge(t.Find(3), t.Find(6))
    t.Merge(t.Find(3), t.Find(0))
```

Exercise 8.2 Repeat Exercise 8.1 using a ShortDisjointSets implementation.
CHAPTER 8. DISJOINT SETS

Exercise 8.3 Repeat Exercise 8.1 using a COMPRESSEDDISJOINTSETS implementation.

Exercise 8.4 Prove that SCHEDULE, shown in Figure 8.2, meets its specification.

Exercise 8.5 Prove that an algorithm that returns an array sched[1..n] containing all 0s meets the specification of SCHEDULE (Figure 8.2).

Exercise 8.6 Analyze the worst-case running time of SCHEDULE (Figure 8.2) assuming the TREEDISJOINTSETS implementation of DISJOINTSETS. Your analysis should be in terms of n, and you may assume that m ≤ n. Express your result as simply as possible using Θ-notaiton.

Exercise 8.7 Repeat Exercise 8.6 assuming the SHORTDISJOINTSETS implementation of DISJOINTSETS.

Exercise 8.8 Repeat Exercise 8.7 assuming the COMPRESSEDDISJOINTSETS implementation of DISJOINTSETS. Express the best result you can as simply as possible using big-O notation.

Exercise 8.9 Prove that TREEDISJOINTSETS, shown in Figure 8.4, meets the DISJOINTSETS specification, given in Figure 8.1.

Exercise 8.10 Prove that SHORTDISJOINTSETS, shown in Figure 8.5, meets the DISJOINTSETS specification, given in Figure 8.1.

Exercise 8.11 Prove that COMPRESSEDDISJOINTSETS, described in Section 8.4, meets the DISJOINTSETS specification, given in Figure 8.1. Use as the structural invariant that for 0 ≤ i < n,

- if rank[i] = 0 then there is no j, such that 0 ≤ j < n, j ≠ i, and parent[j] = i; and
- if rank[i] > 0, then
  - there is some j such that 0 ≤ j < n, j ≠ i, and parent[j] = i; and
  - rank[i] > max{rank[j] | 0 ≤ j < n, j ≠ i, parent[j] = i}.
Exercise 8.12 Suppose that we modify ShortDisjointSets so that in the Merge operation we make the tree with fewer nodes a child of the root of the other tree (choosing arbitrarily if both trees have the same number of nodes). Prove by induction on \( k \) that any tree with \( k > 0 \) nodes formed in this way will have height at most \( \lg k \).

Exercise 8.13 Suppose that we modify TreeDisjointSets so that in the Merge operation we flip a fair coin to determine which node will be the new root. Analyze the worst-case expected running time of Find for such an implementation. Express your answer as simply as possible using \( \Theta \)-notation. In showing the lower bound, describe a sequence of operations for an arbitrarily large universe of elements such that the last Find is expected to require the stated running time.

Exercise 8.14 Prove by induction on \( i \) that \( A_0^{(i)}(n) = n + i \), so that \( A_1(n) = 2n + 1 \).

Exercise 8.15 Using the result of Exercise 8.14, prove by induction on \( i \) that \( A_1^{(i)}(n) = 2^i(n + 1) - 1 \), so that \( A_2(n) = 2^{n+1}(n + 1) - 1 \).

Exercise 8.16 Prove by induction on \( k \) that each \( A_k \) is nondecreasing.

Exercise 8.17 Using the results of Exercises 8.15 and 8.16, prove by induction on \( i \) that

\[
A_2^{(i)}(n) \geq 2^{2 \cdot 2n},
\]

where the right-hand side has \( i \) 2s.

Exercise 8.18 Using the result of Exercise 8.15, evaluate \( A_3(1) \).

Exercise 8.19 Using the result of Exercises 8.17 and 8.18, show that

\[
A_4(1) \geq 2^{2 \cdot 2},
\]

where the right-hand side has 2051 2s.

Exercise 8.20 For the following, you may use the results of Exercises 8.15 and 8.16.

a. Prove by induction on \( i \) that for each \( i \in \mathbb{N} \),

\[
\lg^{(i)}(n) \geq \min\{ k \mid A_2^{(i)}(k) \geq n \}.
\]
b. Prove by induction on \( k \) that for \( k \geq 4 \), \( A_k(1) \geq A_2^{(k)}(k) \).

c. Using the results of parts a and b, prove that for each \( i \in \mathbb{N} \), there is an \( n_i \in \mathbb{N} \) such that whenever \( n \geq n_i \), \( \alpha(n) \leq \lg^{(i)}(n) \).

d. Using the result of part c, prove that for each \( i \in \mathbb{N} \), \( \alpha(i) \in o(\lg^{(i)}(n)) \).

* Exercise 8.21 Let

\[
\lg^* n = \min\{k \mid \lg^{(k)} n \leq 1\}.
\]

Prove that \( \alpha(n) \in o(\lg^* n) \).

8.7 Chapter Notes

The TreeDisjointSets implementation of DisjointSets is due to Galler and Fisher [45]. The improvement of Section 8.3 is presented by Hopcroft and Ullman [63], who credit McIlroy and Morris with having implemented it.

The improvement using path compression is credited to Tritter by Knuth [76]. The amortized analysis of this structure yielding results similar to those presented here was done by Tarjan [103, 104]. The analysis given here is based on the presentation by Cormen, et al. [25], which is based on a proof due to Kozen [82].

Exercise 8.12 is from Brassard and Bratley [18].