Original notes by Matos Gilberto and Patchanee Ittarat.

1 Maintaining Disjoint Sets

Read Chapter 22 for Disjoint Sets Data Structure and Chapter 24 for Kruskal's Minimum Spanning Tree Algorithm [?].

I assume everyone is familiar with Kruskal's MST algorithm. This algorithm is the nicest one to motivate the study of the disjoint set data structure. There are other applications for the data structure as well. We will not go into the details behind the implementation of the data structure, since the book gives a very nice description of the UNION, MAKESET and FIND operations.

1.1 Disjoint set operations:

- Makeset(x) : $A \leftarrow Makeset(x) \equiv A = \{x\}$.
- Find(y) : given y, find to which set y belongs.
- Union(A, B) : $C \leftarrow Union(A, B) \equiv C = A \cup B$.

Observe that the Union operation can be specified either by two sets or by two elements (in the latter case, we simply perform a union of the sets the elements belong to).

The name of a set is given by the element stored at the root of the set (one can use some other naming convention too). This scheme works since the sets are disjoint.

1.2 Data structure:

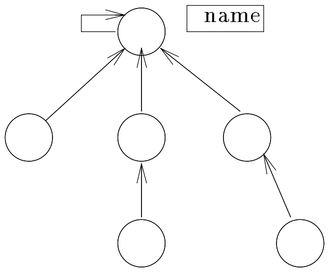

Figure 1: Data Structure

We will maintain the sets as rooted trees. Performing the UNION operation is done by linking one tree onto another. To perform a FIND, we traverse the path from the element to the root.

To improve total time we always hook the shallower tree to the deeper tree (this will be done by keeping track of ranks and not the exact depth of the tree).

Ques: Why will these improve the running time?

Ans: Since the amount of work depends on the height of a tree.

We use the concept of the "rank" of a node. The rank of a vertex denotes an upper bound on the depth of the sub-tree rooted at that node.

$rank(x) = 0$ for $\{x\}$

Union(A,B) - see Fig. 2.

If $rank(a) < rank(b)$, $rank(c) = rank(b)$.

If $rank(a) = rank(b)$, $rank(c) = rank(b) + 1$.

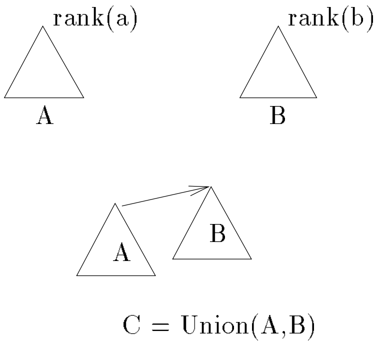

Figure 2: Union

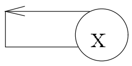

Figure 3: Rank 0 node

1.3 Union by rank

Union by rank guarantees "at most $O(\log n)$ of depth".

Lemma 1.1 *A node with rank k has at least 2^k descendants.*

Proof:
[By induction on size of tree]
$rank(x) = 0 \Rightarrow x$ has one descendant (itself).
The rank and the number of descendants of any node are changed only by the Union operation, so let's consider a Union operation in Fig. 2.

Case 1 *$rank(a) < rank(b)$:*

$$\begin{aligned} rank(c) &= rank(b) \\ desc(c) &\geq desc(b) \\ &\geq 2^{rank(b)} \end{aligned}$$

Case 2 *$rank(a) = rank(b)$:*

$$\begin{aligned} rank(c) &= rank(b)+1 \\ desc(a) &\geq 2^{rank(a)} \quad and \\ desc(b) &\geq 2^{rank(b)} \\ desc(c) &= node(a)+node(b) \\ &\geq 2^{rank(a)}+2^{rank(b)} \\ &\geq 2^{rank(b)+1} \end{aligned}$$

□

In the Find operation, we can make a tree shallower by path compression. During path compression, we take all the nodes on the path to the root and make them all point to the root (to speed up future find operations).

The UNION operation is done by hooking the smaller rank vertex to the higher rank vertex, and the FIND operation performs the path compression while we traverse the path from the element to the root. Path compression takes two passes – first to find the root, and second time to change the pointers of all nodes on the path to point to the root. After the find operation all the nodes point directly to the root of the tree.

1.4 Upper Bounds on the Disjoint-Set Union Operations

Simply recall that each vertex has a rank (initially the rank of each vertex is 0) and this rank is incremented by 1 whenever we perform a union of two sets that have the same rank. We only worry about the time for the FIND operation (since the UNION operation takes constant time). The parent of a vertex always has higher rank than the vertex itself. This is true for all nodes except for the root (which is its own parent).

The following theorem gives two bounds for the total time taken to perform m find operations. (A union takes constant time.)

Theorem 1.2 *m operations (including makeset, find and union) take total time $O(m \log^* n)$ or $O(m + n \log n)$, where $\log^* n = \{min(i) | \log^{(i)} n \leq 1\}$.*

$$\begin{aligned} \log^* 16 &= 3, \ \textit{and} \\ \log^* 2^{16} &= 4 \end{aligned}$$

To prove this, we need some observations:

1. Rank of a node starts at 0 and goes up as long as the node is a root. Once a node becomes a non-root the rank does not change.
2. $Rank(p(x))$ is non-decreasing. In fact, each time the parent changes the rank of the parent must increase.
3. $Rank(p(x)) > rank(x)$.

The rank of any vertex is lesser than or equal $\log_2 n$, where n is the number of elements.

First lets see the $O(m + n \log n)$ bound (its easier to prove). This bound is clearly not great when $m = O(n)$.

The cost of a single find is charged to 2 accounts

1. The find pays for the cost of the root and its child.
2. A bill is given to every node whose parent changes (in other words, all other nodes on the find path).

Note that every node that has been issued a bill in this operation becomes the child of the root, and won't be issued any more bills until its parent becomes a child of some other root in a union operation. Note also that one node can be issued a bill at most $\log_2 n$ times, because every time a node gets a bill its parent's rank goes up, and rank is bounded by $\log_2 n$. The sum of bills issued to all nodes will be upper bounded by $n \log_2 n$

We will refine this billing policy shortly.

Lemma 1.3 *There are at most $\frac{n}{2^r}$ nodes of rank r.*

Proof:

When the node gets rank r it has $\geq 2^r$ descendants, and it is the root of some tree. After some union operations this node may no longer be root, and may start to lose descendants, but its rank will not change. Assume that every descendant of a root node gets a timestamp of r when the root first increases its rank to r. Once a vertex gets stamped, it will never be stamped again since its new roots will have rank strictly more than r. Thus for every node of rank r there are at least 2^r nodes with a timestamp of r. Since there are n nodes in all, we can never create more than $\frac{n}{2^r}$ vertices with rank r. □

We introduce the fast growing function F which is defined as follows.

1. $F(0) = 1$
2. $F(i) = 2^{F(i-1)}$

1.5 Concept of Blocks

If a node has rank r, it belongs to block $B(\log^* r)$.

- $B(0)$ contains nodes of rank 0 and 1.
- $B(1)$ contains nodes of rank 2.
- $B(2)$ contains nodes of rank 3 and 4.
- $B(3)$ contains nodes of rank 5 through 16.
- $B(4)$ contains nodes of rank 17 through 65536.

Since the rank of a node is at most $\log_2 n$ where n is the number of elements in the set, the number of blocks necessary to put all the elements of the sets is bounded by $\log^*(\log n)$ which is $\log^* n - 1$. So blocks from $B(0)$ to $B(\log^* n - 1)$ will be used.

The find operation goes the same way as before, but the billing policy is different. The find operation pays for the work done for the root and its immediate child, and it also pays for all the nodes which are not in the same block as their parents. All of these nodes are children of some other nodes, so their ranks will not change and they are bound to stay in the same block until the end of computation. If a node is in the same block as its parent it will be billed for the work done in the find operation. As before find operation pays for the work done on the root and its child. Number of nodes whose parents are in different blocks is limited to $\log^* n - 1$, so the cost of the find operation is upper bounded by $\log^* n - 1 + 2$.

After the first time a node is in the different block from its parent, it is always going to be the case since the rank of the parent only increases. This means that the find operation is going to pay for the work on that node every time. So any node will be billed for the find operations a certain number of times, and after that all subsequent finds will pay for their work on the element. We need to find an upper bound for the number of times a node is going to be billed for the find operation.

Consider the block with index i; it contains nodes with the rank in the interval from $F(i-1)+1$ to $F(i)$. The number of nodes in this block is upper bounded by the possible number of nodes in each of the ranks. There are at most $n/2^r$ nodes of rank r, so this is a sum of a geometric series, whose value is

$$\sum_{r=F(i-1)+1}^{F(i)} \frac{n}{2^r} \leq \frac{n}{2^{F(i-1)}} = \frac{n}{F(i)}$$

Notice that this is the only place where we make use of the exact definition of function F.

After every find operation a node changes to a parent with a higher rank, and since there are only $F(i) - F(i-1)$ different ranks in the block, this bounds the number of bills a node can ever get. Since the block $B(i)$ contains at most $n/F(i)$ nodes, all the nodes in $B(i)$ can be billed at most n times (its the product of the number of nodes in $B(i)$ and the upper bound on the bills a single node may get). Since there are at most $\log^* n$ blocks the total cost for these find operations is bounded by $n \log^* n$.

This is still not the tight bound on the number of operations, because Tarjan has proved that there is an even tighter bound which is proportional to the $O(m\alpha(m,n))$ for m union, makeset and and find operations, where α is the inverse ackerman function whose value is lower than 4 for all practical applications. This is quite difficult to prove (see Tarjan's book). There is also a corresponding tight lower bound on *any* implementation of the disjoint set data structure on the pointer machine model.