Disjoint Sets

Today’s Outline

• Data Structures for Disjoint Sets

Operations

We want to support the following operations:

• Make-Set(x): creates a new set whose only member (and representative) is x
• Union(x,y): unites the sets that contain x and y (call them \(S_x\) and \(S_y\)) into a new set that is \(S_x \cup S_y\). The new set is added to the data structure while \(S_x\) and \(S_y\) are deleted. The representative of the new set is any member of the set.
• Find-Set(x): Returns a pointer to the representative of the (unique) set containing x

A disjoint set data structure maintains a collection \(\{S_1, S_2, \ldots, S_k\}\) of disjoint dynamic sets
Each set is identified by a representative which is a member of that set
Let’s call the members of the sets objects.
Simple Union

Make-Set(x){
    parent(x) = x;
    size(x) = 1;
}

Simple-Union(x,y){
    xRep = Find-Set(x);
    yRep = Find-Set(y);
    if (size(xRep)) > size(yRep)){
        parent(yRep) = xRep;
    }else{
        parent(xRep) = yRep;
    }
    size(yRep) = size(yRep) + size(xRep);
}

Analysis

- We showed in last class that the heights of all trees are no more than logarithmic in the number of nodes in the tree
- Thus all of these operations take $O(\log n)$ time
- Q: Can we do better?
- A: Yes we can do much better in an amortized sense.

Shallow Threaded Trees

- One good idea is to just have every object keep a pointer to the leader of it's set
- In other words, each set is represented by a tree of depth 1
- Then Make-Set and Find-Set are completely trivial, and they both take $O(1)$ time
- Q: What about the Union operation?

Union

- To do a union, we need to set all the leader pointers of one set to point to the leader of the other set
- To do this, we need a way to visit all the nodes in one of the sets
- We can do this easily by "threading" a linked list through each set starting with the sets leaders
- The threads of two sets can be merged by the Union algorithm in constant time
The Code

Make-Set(x){
    leader(x) = x;
    next(x) = NULL;
}

Find-Set(x){
    return leader(x);
}

Union(x,y){
    xRep = Find-Set(x);
    yRep = Find-Set(y);
    leader(y) = xRep;
    while(next(y)! = NULL){
        y = next(y);
        leader(y) = xRep;
    }
    next(y) = next(xRep);
    next(xRep) = yRep;
}

Example

Merging two sets stored as threaded trees. Bold arrows point to leaders; lighter arrows form the threads. Shaded nodes have a new leader.

Analysis

• Worst case time of Union is a constant times the size of the larger set
• So if we merge a one-element set with a \( n \) element set, the run time can be \( \Theta(n) \)
• In the worst case, it's easy to see that \( n \) operations can take \( \Theta(n^2) \) time for this alg
Problem

- The main problem here is that in the worst case, we always get unlucky and choose to update the leader pointers of the larger set
- Instead let’s purposefully choose to update the leader pointers of the smaller set
- To do this, we will need to keep track of the sizes of all the sets

The Code

```
Make-Weighted-Set(x){
leader(x) = x;
next(x) = NULL;
size(x) = 1;
}
```

```
Weighted-Union(x,y){
xRep = Find-Set(x);
yRep = Find-Set(y)
if(size(xRep)>size(yRep){
    Union(xRep,yRep);
    size(xRep) = size(xRep) + size(yRep);
}else{
    Union(yRep,xRep);
    size(yRep) = size(xRep) + size(yRep);
}
}
```

Analysis

- The Weighted-Union algorithm still takes $\Theta(n)$ time to merge two $n$ element sets
- However in an amortized sense, it is more efficient
- Intuitively, in order to merge two large sets, we need to perform a large number of cheap Weighted-Unions
- We will show that a sequence of $n$ Make-Weighted-Set operations and $m$ Weighted-Union operations takes $O(m+n \log n)$ time in the worst case.
Proof

- Whenever the leader of an object $x$ is changed by a call to Weighted-Union, the size of the set containing $x$ increases by a factor of at least 2
- Thus if the leader of $x$ has changed $k$ times, the set containing $x$ has at least $2^k$ members
- After the sequence of operations ends, the largest set has at most $n$ members
- Thus the leader of any object $x$ has changed at most $\lceil \log n \rceil$ times

- We’ve just shown that for $n$ calls to Make-Weighted-Set and $m$ calls to Weighted-Union, that total cost for updating leader pointers is $O(n \log n)$
- We know that other than the work needed to update these leader pointers, each call to one of our functions does only constant work
- Thus total amount of work is $O(n \log n + m)$
- Thus each Weighted-Union call has amortized cost of $O(\log n)$

Side Note: We’ve just used the aggregate method of amortized analysis

Analysis

- Let $n$ be the number of calls to Make-Weighted-Set and $m$ be the number of calls to Weighted-Union
- We’ve shown that each of the objects that are not in singleton sets had at most $O(\log n)$ leader changes
- Thus, the total amount of work done in updating the leader pointers is $O(n \log n)$

- Using Simple-Union, $Find$ takes logarithmic worst case time and everything else is constant
- Using Weighted-Union, $Union$ takes logarithmic amortized time and everything else is constant
- A third method allows us to get both of these operations in almost constant amortized time
Path Compression

• We start with the unthreaded tree representation (from Simple-Union)
• Key Observation is that in any Find operation, once we get the leader of an object \( x \), we can speed up future Find’s by redirecting \( x \)’s parent pointer directly to that leader
• We can also change the parent pointers of all ancestors of \( x \) all the way up to the root (We’ll do this using recursion)
• This modification to Find is called path compression

Example

Path compression during Find(\( c \)). Shaded nodes have a new parent.

PC-Find Code

```
PC-Find(x)
{
    if(x!=Parent(x)){
        Parent(x) = PC-Find(Parent(x));
    }
    return Parent(x);
}
```

Rank

• For ease of analysis, instead of keeping track of the size of each of the trees, we will keep track of the rank
• Each node will have an associated rank
• This rank will give an estimate of the log of the number of elements in the set
Code

PC-MakeSet(x) {
    parent(x) = x;
    rank(x) = 0;
}

PC-Union(x, y) {
    xRep = PC-Find(x);
    yRep = PC-Find(y);
    if (rank(xRep) > rank(yRep))
        parent(yRep) = xRep;
    else{
        parent(xRep) = yRep;
        if (rank(xRep) == rank(yRep))
            rank(yRep)++;
    }
}

Rank Facts

Can also say that there are at most $n/2^r$ objects with rank $r$.

- When the rank of a set leader $x$ changes from $r - 1$ to $r$, mark all nodes in that set. At least $2^r$ nodes are marked and each of these marked nodes will always have rank less than $r$.
- There are $n$ nodes total and any object with rank $r$ marks $2^r$ of them.
- Thus there can be at most $n/2^r$ objects of rank $r$.

Blocks

- We will also partition the objects into several numbered blocks.
- $x$ is assigned to block number $\log^*(\text{rank}(x))$.
- Intuitively, $\log^* n$ is the number of times you need to hit the log button on your calculator, after entering $n$, before you get 1.
- In other words $x$ is in block $b$ if
  $$2 \uparrow\uparrow (b - 1) < \text{rank}(x) \leq 2 \uparrow\uparrow b,$$
  where $\uparrow\uparrow$ is defined as in the next slide.

- If an object $x$ is not the set leader, then the rank of $x$ is strictly less than the rank of its parent.
- For a set $X$, size($X$) $\geq 2^{\text{rank(leader}(X))}$ (can show using induction).
- Since there are $n$ objects, the highest possible rank is $O(\log n)$.
- Only set leaders can change their rank.
Definition

\(2 \uparrow\uparrow b\) is the tower function

\[2 \uparrow\uparrow b = \begin{cases} 1 & \text{if } b = 0 \\ 2^{2 \uparrow\uparrow (b-1)} & \text{if } b > 0 \end{cases}\]

Number of Blocks

- Every object has a rank between 0 and \(|\log n|\)
- So the blocks numbers range from 0 to \(\log^* |\log n| = \log^*(n) - 1\)
- Hence there are \(\log^* n\) blocks

Number Objects in Block \(b\)

- Since there are at most \(n/2^r\) objects with any rank \(r\), the total number of objects in block \(b\) is at most

\[
\sum_{r=2 \uparrow\uparrow (b-1)+1}^{2 \uparrow\uparrow b} \frac{n}{2^r} < \sum_{r=2 \uparrow\uparrow (b-1)+1}^{\infty} \frac{n}{2^r} = \frac{n}{2^{2 \uparrow\uparrow (b-1)}} = \frac{n}{2 \uparrow\uparrow b}.
\]

Theorem

- Theorem: If we use both PC-Find and PC-Union (i.e. Path Compression and Weighted Union), the worst-case running time of a sequence of \(m\) operations, \(n\) of which are MakeSet operations, is \(O(m \log^* n)\)
- Each PC-MakeSet and PC-Union operation takes constant time, so we need only show that any sequence of \(m\) PC-Find operations require \(O(m \log^* n)\) time in the worst case
- We will use a kind of accounting method to show this
Proof

- The cost of PC-Find($x_0$) is proportional to the number of nodes on the path from $x_0$ up to its leader.
- Each object $x_0, x_1, x_2, \ldots, x_l$ on the path from $x_0$ to its leader will pay a 1 tax into one of several bank accounts.
- After all the Find operations are done, the total amount of money in these accounts will give us the total running time.

Taxation

- The leader $x_l$ pays into the leader account.
- The child of the leader $x_{l-1}$ pays into the child account.
- Any other object $x_i$ in a different block from its parent $x_{i+1}$ pays into the block account.
- Any other object $x_i$ in the same block as its parent $x_{i+1}$ pays into the path account.

Example

- During any Find operation, one dollar is paid into the leader account.
- At most one dollar is paid into the child account.
- At most one dollar is paid into the block account for each of the $\log^* n$ blocks.
- Thus when the sequence of $m$ operations ends, these accounts share a total of at most $2m + m \log^* n$ dollars.
The only remaining difficulty is the Path account.

Consider an object $x_i$ in block $b$ that pays into the path account.

This object is not a set leader so its rank can never change.

The parent of $x_i$ is also not a set leader, so after path compression, $x_i$ gets a new parent, $x_l$, whose rank is strictly larger than its old parent $x_{i+1}$.

Since $\text{rank}(\text{parent}(x))$ is always increasing, parent of $x_i$ must eventually be in a different block than $x_i$, after which $x_i$ will never pay into the path account.

Thus $x_i$ pays into the path account at most once for every rank in block $b$, or less than $2 \uparrow \uparrow b$ times total.

Since block $b$ contains less than $n/(2 \uparrow \uparrow b)$ objects, and each of these objects contributes less than $2 \uparrow \uparrow b$ dollars, the total number of dollars contributed by objects in block $b$ is less than $n$ dollars to the path account.

There are $\log^* n$ blocks so the path account receives less than $n \log^* n$ dollars total.

Thus the total amount of money in all four accounts is less than $2m + m \log^* n + n \log^* n = O(m \log^* n)$, and this bounds the total running time of the $m$ operations.

We can now say that each call to PC-Find has amortized cost $O(\log^* n)$, which is significantly better than the worst case cost of $O(\log n)$.

The book shows that PC-Find has amortized cost of $O(A(n))$ where $A(n)$ is an even slower growing function than $\log^* n$. 