SNS COLLEGE OF ENGINEERING

Kurumbapalayam (Po), Coimbatore – 641 107

AN AUTONOMOUS INSTITUTION

Accredited by NAAC-UGC with 'A' Grade

Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

Department of Computer science and Engineering (Internet of things with Block chain technology and cyber security)

DAA Notes

Unit-1

Disjoint sets

Disjoint sets in mathematics are two sets that don't have any element in common. Sets can contain any number of elements, and those elements can be of any type. We can have a set of cars, a set of integers, a set of colors, etc. Sets also have various operations that can be performed on them, such as union, intersection, difference, etc.

Disjoint Set Operations in DAA

There are 2 major disjoint set operations in daa:

Union/Merge - this is used to merge 2 subsets into a single subset.

Find - This is used to find which subset a particular value belongs to.

We will take a look at both of them. To make things easier, we will change our sets from integers to Cities.

Union/Merge

Suppose, in our example - a new express train started from Delhi to Agra. This would mean that all the cities connected to Delhi are now connected to all the cities connected to Agra.

This simply means that we can merge the subsets of Agra and Delhi into a single subset. It will look like this.

One key issue here is which out of Agra and Delhi will be the new Parent? For now, we will just choose any of them, as it will be sufficient for our purposes. Let's take Delhi for now.

Further optimizations can be made to this by choosing the parent based on which subset has a larger number of values.

This is the new state of our Disjoint Set Data Structure. Delhi is the parent of Delhi, Gurgaon, Noida, Agra, Vrindavan, and Mathura.

This would mean that we would have to change the parent of Agra, Vrindavan, and Mathura (basically, all the children of Agra), to Delhi. This would be linear in time. If we perform the union operation m times, and each union takes O(n) times, we would get a quadratic O(mn) time complexity. This is not optimized enough.

Instead, we structure it like this:

Earlier, Agra was its own parent. Now, Delhi is the parent of Agra.

Now, you might be wondering - The parent of Vrindavan is still Agra. It should have been Delhi now.

This is where the next operation helps us - the find operation.

Find

The find operation helps us find the parent of a node. As we saw above, the direct parent of a node might not be its actual (logical) parent. E.g., the logical parent of Vrindavan should be Delhi in the above example. But its direct parent is Agra.

So, how do we find the actual parent?

The find operation helps us to find the actual parent. In pseudocode, the find operation looks like this:

```
find(node):
if (parent(node)==node) return node;
else return find(parent(node));
```

We don't return the direct parent of the node. We keep going up the tree of parents until we find a node that is its own parent.

Let's understand this via our example.

Suppose we have to find the parent of Mathura.
Check the direct parent of Mathura. It's Agra. Is Agra == Mathura?
The answer is false. So, now we call the find operation on Agra.
Check the direct parent of Agra. It's Delhi. Is Delhi ==Agra?
The answer is false. So now we call the find operation on Delhi.
Check the direct parent of Delhi. It's Delhi. Is Delhi==Delhi.
The answer is true! So, our final answer for the parent of Mathura is Delhi.
This way, we keep going "up" the tree until we reach a root element. A root element is a node with its own parent - in our case, Delhi.

Let's try to find the parent of K in this example. (A and D are their own parents)
ParentOf(K) is I. Is I == K?
False. So, we move upward, now using I.
ParentOf(I) is C. Is C== I?
False. Move upward using C.
ParentOf(C) is A. Is A==C?
False. Move upward using A.
ParentOf(A) is A. Is A==A?
True! Our final answer is A.