Disjoint Set

Sunday, 5 November 2023 10:24 AM

Introduction of Disjoint Set

— called disjoint sets if they don't have any elements in common, intersection of sets is NULL.
— supports following operations:
  — Adding new sets to disjoint set
  — Merging disjoint sets to a single disjoint set using Union operation
  — Find representative of a disjoint set using Find operation
  — Check if two sets are disjoint or not

Data structures used are:

Array — An array of integers is called parent[].
The ith item of parent[] array is parent of ith item.

Tree — It is a disjoint set. If two ele in same tree then they are in same disjoint set.
— root node is called representative of set.

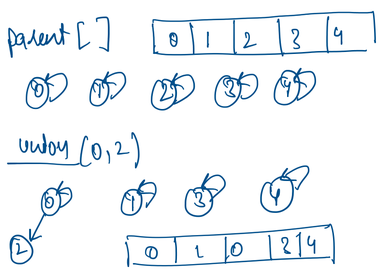
Operations

Find — implemented recursively traversing the parent array until we hit a node that is parent of itself.
O(n) in worst case.

Union — two ele as input and find the representatives of their sets using Find operation
finally put either one of tree (representing set) under root node of tree.
inefficient approach and could lead to a tree of length O(n) in worst case.

```
n=5
parent = [i for i in range(n)]
def find(x):
    if parent[x]==x:
        return x
    return find(parent[x])
def union(x,y):
    x_rep = find(x)
    y_rep = find(x)
    if x_rep==y_rep:
        return
    parent[y_rep] = x_rep
```

parent[] | 0 | 1 | 2 | 3 | 4 |



Optimization (union by rank / size and path compression)

efficiency depends on which tree is getting attached to other. 2 ways to get this done.
Union by Rank → considers height of tree as the factor and
Union by Size → " size " " " " " while attaching one tree to another.
↳ (this method with path compression gives complexity time of nearly constant)

Path Compression

— speeds up by compressing the height of tree.
— achieved by inserting a small caching mechanism to Find operation.
— O(log n) on average per call

Union by Rank

— we need another array, rank[], same as parent[] size.
— if i is representative of a set, rank[i] is height of tree representating the set.
— in union, it does not matter which of two trees is moved under the other.
— if we are uniting two trees (or sets), lets call them left and right, then it depends on rank of left and rank of right:
  — if rank of left < rank of right, then it's best to move left under right, because that won't change rank of right.
  — if ranks are equal, doesn't matter which tree goes under other, but rank of result will be one greater than rank of trees.

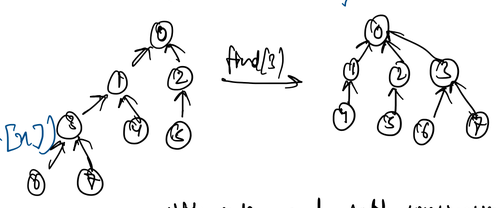
Union by Size

— we need new array, size[], same as parent's size.
— if i is representative of a set, size[i] is no. of elements in tree.
— Uniting two trees (or sets),
  — if size of left < size of right, best to move left under right and increase right size by size of left.
  — if equal, it doesn't matter which tree go under other.
— O(log n) without path compression

Path compression

we make parent of all nodes (on the path from given node to root) as root.

```
def find(x):
    if x==parent[x]:
        return x
    parent[x] = find(parent[x])
    return parent[x]
```



Union by rank

```
n=5
parent = [i for i in range(n)]
rank = [0 for i in range(n)]
def union(x,y):
    x_rep = find(x)
    y_rep = find(y)
    if x_rep==y_rep:
        return
    if rank[x_rep] < rank[y_rep]:
        parent[x_rep] = y_rep
    elif rank[x_rep] > rank[y_rep]:
        parent[y_rep] = x_rep
    else:
        parent[y_rep] = x_rep
        rank[y_rep] += 1
```

With union and path compression

Time for m operations on n elements:
$O(m\,\alpha(n))$ where $\alpha(n) \le 4$
↳ inverse Ackermann function