CSCB63 – Design and Analysis of Data Structures

Anya Tafliovich[1]

[1] based on notes by Anna Bretscher and Albert Lai

disjoint sets

Operations:

- `make-set(`x`)`: create a set that contains x
- `find-set(`x`)`: return the set that contains x
- `union(`S_1`, `S_2`)`: merge sets S_1 and S_2, or
- `union(`x_1`, `x_2`)`: merge set that contains x_1 and set that contains x_2

Where have we seen this?

linked lists implementation

- each set is a linked list
- x.set is a pointer to x's owning linked list
- `find-set(`x`)` is:
- `union(`S_1`, `S_2`)` is merging two linked lists
- choose to always move the smaller list into the larger one

What is the **amortised** complexity of `union`?

linked lists implementation: complexity

What is the **amortised** complexity of `union`?

Consider a sequence of k operations `make-set`, `find-set`, and `union`, with n operations `make-set`.

- the longest a list can be is:
- operations `make-set` and `find-set` are:
- operation `union` is:
  - in the best case:
  - in the worst case:
  - in the worst case:
- then how many such updates can we do?
  - each x`.set` field is updated at most:
  - there are ______`.set` fields
  - total number of updates at most:

linked lists implementation: complexity

What is the **amortised** complexity of `union`?

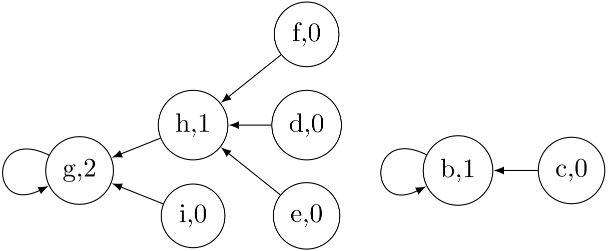
Consider a sequence of k operations `make-set`, `find-set`, and `union`, with n operations `make-set`.

Total time at most:

Amortised time:

forest implementation

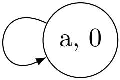
- each set is a tree
- pointers from children to parents
- root points to itself
- each node stores `rank`
  - an upper bound on the height of the tree rooted at that node



forest implementation: `make-set`

- `make-set` creates a single-node tree

```
make-set(x):
0. root := new node(value=x, rank=0)
1. root.parent := root
2. return root
```



forest implementation: union

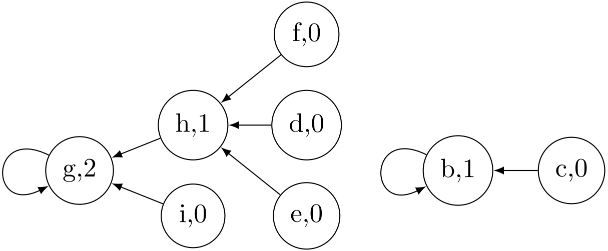
- union makes root of shorter tree a child of root of taller tree

```
union(node1, node2):
  link(find-set(node1), find-set(node2))

link(root1, root2):
  if root1.rank > root2.rank:
    root2.parent := root1
  else:
    root1.parent := root2
    if root1.rank = root2.rank:
      root2.rank++
```

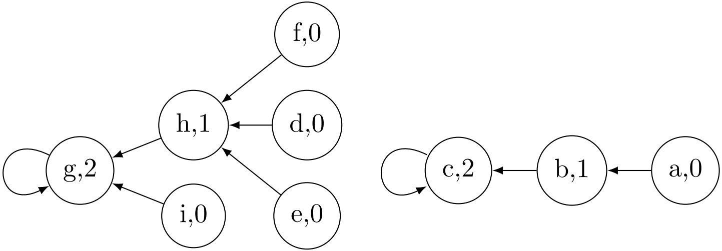
forest implementation: `union`

- `union` makes root of shorter tree a child of root of taller tree



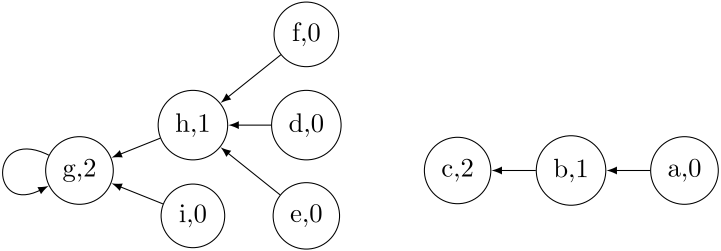
forest implementation: `union`

- `union` makes root of shorter tree a child of root of taller tree



forest implementation: `union`

- `union` makes root of shorter tree a child of root of taller tree

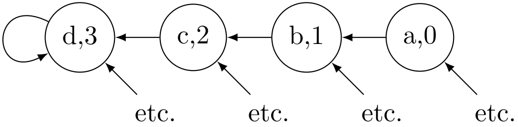


Called union by rank

forest implementation: `find-set`

- `find-set` follows links to root

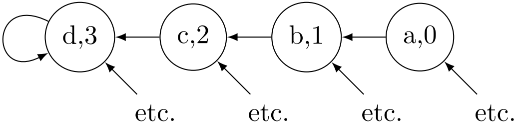
```
find-set(node):
0. if node.parent != node:
1.   return find-set(node.parent)
2. return node
```



forest implementation: `find-set`

- better: path compression
- `find-set` updates parent link directly to root
- `ranks` are not updated

```
find-set(node):
0. if node.parent != node:
1.   node.parent := find-set(node.parent)
2. return node.parent
```



forest implementation: complexity

- The best disjoint set implementation is forests using union-by-rank and path compression.
- What is the worst-case sequence complexity?
- Can show worst-case time for a sequence of k operations with n `make-sets`, is $\mathcal{O}(k\alpha(n)) \in \mathcal{O}(k \log^* n)$
- Here $\log^* n$ is the number of times that you need to apply log to n until the answer is < 1
- The function α grows very, very slowly, virtually a constant.
- Amortised time of disjoint set operations is $\mathcal{O}(\alpha(n))$.
- Full proof outside the scope of this course.
- Note this means the best implementation of Kruskal's algorithm has complexity $\mathcal{O}(|E| \log |E| + |E|\alpha(|V|))$ time.