Disjoint Sets

Partition

A Partition (disjoint division) of a set S is a collection of non-empty subsets A_i, for i = 1, ..., k, which are disjoint and together make up S:

$$A_i \neq \varnothing \text{ for all } i$$

$$A_i \cap A_j = \varnothing \text{ for } i \neq j$$

$$A_1 \cup A_2 \cup \ldots\ldots \cup A_k = S$$

Example:

{a, b, e}, {f }, {c, d, g, h} is a partition of {a, b, c, d, e, f , g, h}

Disjoint Sets operations

Disjoint Sets (Partition) as a Data Structure? The following is a collection of operations that have proven relevant in applications (applications discussed later in the course):

MAKE-SET(*x*):
Create {x} as a set.

UNION(*x, y*):
Merge {a, b, c, ..., x} and {h, i, j, ..., y} into {a, b, c, ..., x, h, i, j, ..., y}.

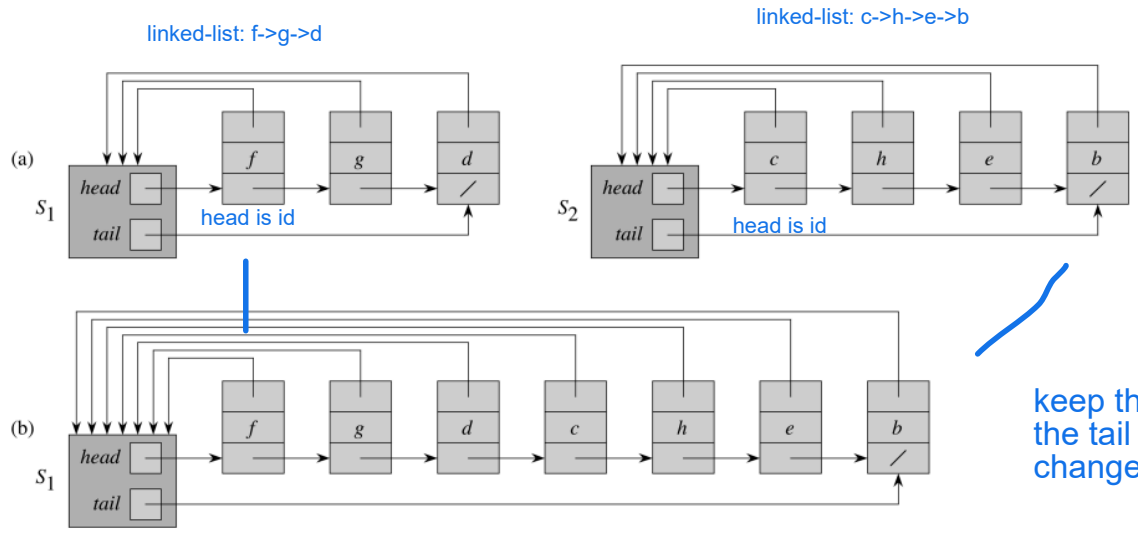
FIND-SET(*x*):
Return an ID for the set containing x.

Note: We have no specific requirements for the ID. It simply needs to be the same for all x in the same set so that we can check whether two elements x and y belong to the same set.

Data structure for Disjoint Sets via linked lists

linked list = contains data, and a pointer to the next element

Each set is a linked list of elements; the ID of the set is the first element in the list:



keep the first (head) and keep the tail of the last linked list (tail). change pointer to head for (old) s2

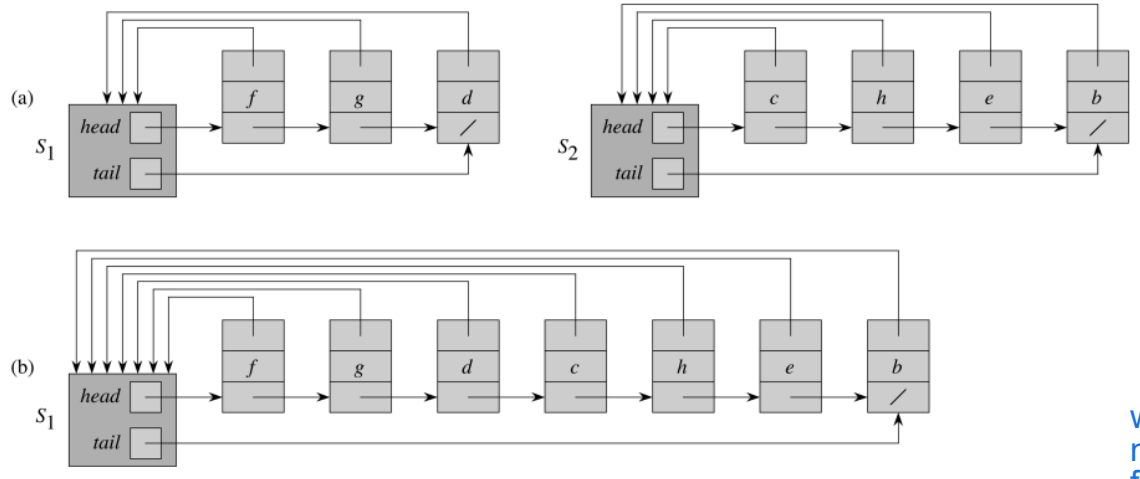
Returns a representative (or ID) of the set that element x belongs to

- FIND-SET(x): return (via header-pointer) the first element in the list.
- MAKE-SET(x): create a new list. O(1)
- UNION(x, y): Merge the lists, keep one header, and change all header pointers in the other list.

we need the tail of a linked list to now where to merge/union

Data structure for Disjoint Sets via linked lists

Running Time (n is the number of elements, i.e., the number of Make-Set operations performed): $O(1)$



worst case runtime could be n^2 for combinging the functions. fx if the list are euqally long

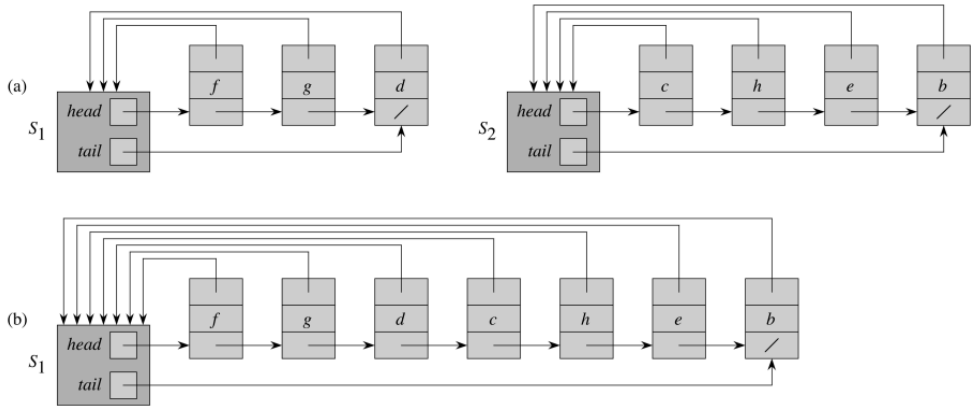
- FIND-SET(x): return (via header-pointer) the first element in the list: $O(1)$
- MAKE-SET(x): create a new list: $O(1)$
- UNION(x, y): merge the lists, keep one header, and change all header pointers in the other list: $O(n)$ worst case all elements (n) need new pointers

Naive analysis: n MAKE-SET, up to $n-1$ UNION, and m FIND-SET costs $O(m+n^2)$.

becuase you merge two sets into 1

Data structure for Disjoint Sets via linked lists

Skriv tekst her



- FIND-SET(x): return (via header-pointer) the first element in the list: $O(1)$ for simply the header (but if you search for an element than O(n))
- MAKE-SET(x): create a new list: $O(1)$ when i make a set of 1 elment, if more elements then O(n)
- UNION(x, y): merge the lists, keeping the header of the longer list, and change all header pointers in the shorter list: $O(n)$ (more optimal)

x,y in union does, not have to be the head, it can also be an element, and we can still merge the correct sets

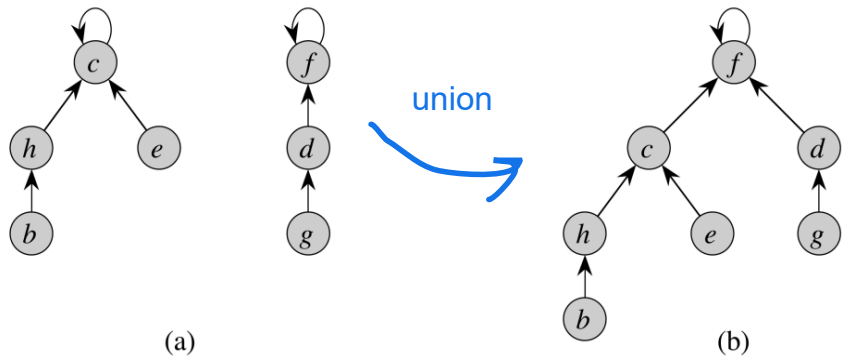
Note that now the following holds: a node can only change its header pointer log n times, since the size of its set increases by at least a factor of two each time: (1. $2^k \leq n \Leftrightarrow k \leq \log n$).

So for better analysis: n MAKE-SET, up to $n-1$ UNION, and m FIND-SET costs $O(m+ n \log n)$.

Data structure for Disjoint Sets via trees

not linked-list any more
(not binary trees, can have many childs)

Each set is a tree with elements in nodes; the root is the ID of the set:



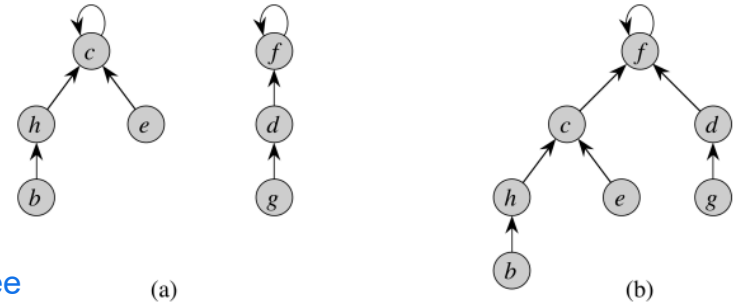
each node point to it's parent

- FIND-SET(x): Traverse to the root.
- MAKE-SET(x): create a new tree.
- UNION(x, y): make the root of one tree a child of the root of the other tree.

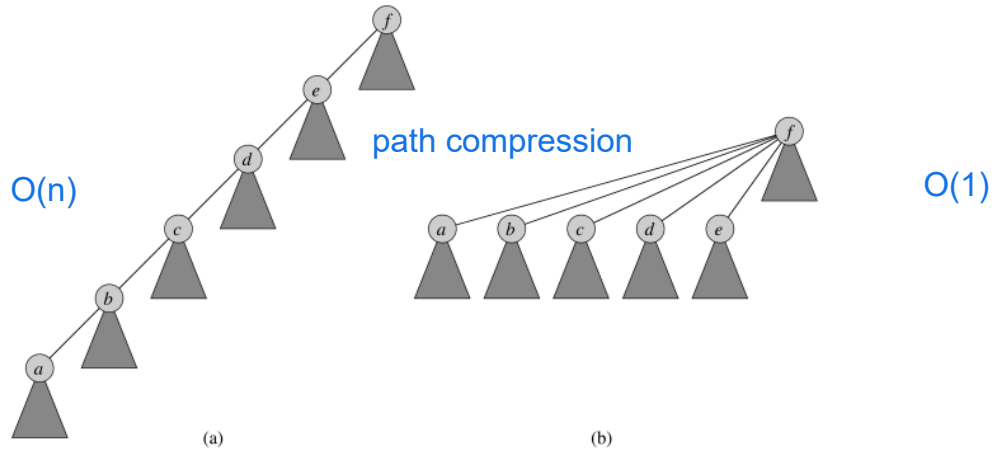
and see of they have the same root, if not they are not in the same set

Data structure for Disjoint Sets via trees

rank = 2



Rank is the longest path from the bottom node, to the top root.
max rank is the log to the elements in the tree



O(n)

O(1)

path compression keeps the rank, even though the height is different

Union by Rank and Path Compression (see textbook section 19.3 [3rd edition: section 21.3] or the code below for the definition).

Union by rank = tree with smallest height under the bigger tree
if the rank is equal in both tree, union and increase the rank by +1

result in runtime O(n)

path compression =when transversion the tree from bottom node to root,rearrange so the nodes point to the root.

This effectively "flattens" the tree by connecting nodes directly to the root, making future FIND-SET operations for these nodes (and others on the path) much faster.

Data structure for Disjoint Sets via trees

Pseudocode (with union by rank and path compression) is simple:

```
MAKE-SET(x)
  x.p = x
  x.rank = 0
```

```
UNION(x, y)
  LINK(FIND-SET(x), FIND-SET(y))
```

first findset (find the root), than link

```
FIND-SET(x)
  if x ≠ x.p
      x.p = FIND-SET(x.p)
  return x.p
```

```
LINK(x, y)
  if x.rank > y.rank
      y.p = x
  else x.p = y
      // If equal ranks, choose y as parent and increment its rank.
      if x.rank == y.rank
          y.rank = y.rank + 1
```

operation on roots