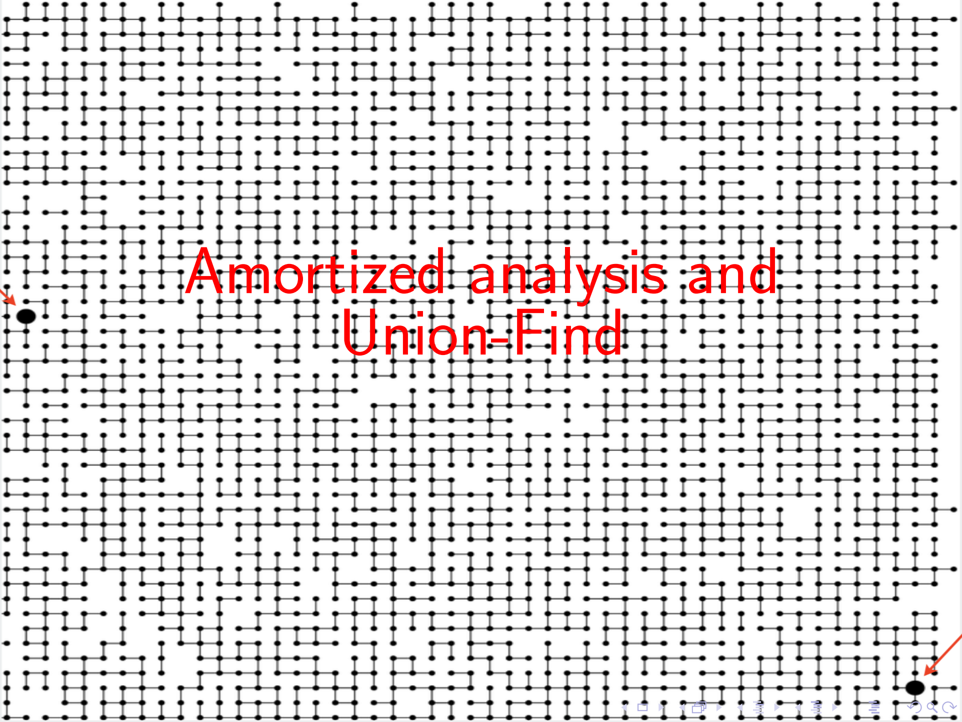
Amortized analysis and Union-Find



Analyzing operations on data structure

(See for ex. Sect. 17-1 to 17.3 in CLRS)

Given a Data Structure we want to analyze the cost of basic operations on the DS, for example

- Create and initialized a DS with size n.
- Introduce an element in a position of the DS.
- Read and return an element in the DS.

Worst-case analysis: Determine worst-case running time of a data structure operation as function of its input size.

Amortized analysis: a strategy for analyzing a sequence of operations on a DS, to show that the "average" cost per operation is small, even though a single operation within the sequence might be expensive.

Amortized analysis

- An amortized analysis guarantees the average performance of each operation in the worst case.
- The easier way to think about amortized analysis is to consider the number of steps averaged over all the sequence of operations.
- Amortization gives us a procedure to do an average-case analysis without using any probability.
- If the series of operations is not specified, we can assume to be a series of operations from the DS, starting with an empty structure at time 0.

Binary Counter

(See page 454 in CLRS).

- Binary Counter: We have a k-bit binary counter $A[0 \dots k-1]$, where $A[0]$ is the least significative bit, and $A[k-1]$ most significant bit.
- Let v the value represented by the counter A, then $v = \sum_{i=0}^{k-1} A[i]2^i$.
- At the the beginning the initial counting is set to 0. At each step we increment by $1 \mod 2^k$ the counter.
- Cost of **Increment** is the total number of bits flipped.
- The goal is to estimate the cost of a sequence of increments on a k-bit binary counter, i.e. $v = n$.

Binary Counter

Table: Ex. for $|A| = k = 4$

value v	A	cost
0	0000	0
1	0001	1
2	0010	3
3	0011	4
4	0100	7
5	0101	8
6	0110	10
7	0111	11
8	1000	15
9	1001	16

```
Increment (A, k)
i = 1
while i < k and A[i] = 1 do
    A[i] = 0
    i = i + 1
end while
if i < k then
    A[i] = 1
end if
```

Rough analysis: As each call could flip k bits, to do n **Increment** operations has cost $O(nk)$.

Amortized analysis: Aggregate method

For any n, show a sequence of n operations takes worst case time $T(n)$. Then the amortized cost per operation is $T(n)/n$.

For the BINARY COUNTER, to get a value of n,we have that:
Bit $A[0]$ flips n times.
Bit $A[1]$ flips $\lfloor n/2 \rfloor$ times.
Bit $A[2]$ flips $\lfloor n/2^2 \rfloor$ times.
...
Bit $A[i]$ flips $\lfloor n/2^i \rfloor$ times.
Therefore, the total cost $= \sum_{i=0}^{k-1} \lfloor n/2^i \rfloor < n \sum_{i=0}^{\infty} 1/2^i = 2n$.

Starting from $v = 0$, a sequence of n **Increment** operations has cost $O(n)$. The average cost per operation is $O(1)$.

Amortized analysis: Accounting method

- The difference of the accounting method with the aggregate method is that in the accounting we assign differing charges to different operations, with some operations charged more or less than the real cost.
- Assign to each different operation i a credit $\hat{c}_i$ Ƀ.
- Notation In operation i, c_i denotes the real cost and $\hat{c}_i$ denotes the credit and d_i is the resulting balance after implementing the operation (Ƀ).
- The cumulative $\hat{c}_i$ is denote as the amortized cost.
- Notice that for the most expensive operations (a new more significant 1), when $i = 2^x$ we must use the credit to amortize the costs of reseting to 0 the positions to the right.
- At each step i, the amortized cost $>$ real cost, the difference d_i, is saved as credit, to be used later.

Binary Counter: Accounting method

- Each bit flip ($0 \to 1$ or $1 \to 0$) has cost 1
- Charge $\hat{c}_i = 2$Ƀ when flipping bit i from $0 \to 1$, use the 2Ƀ:
  - $c_i = 1$Ƀ pays for the $0 \to 1$ flipping of the i-th. bit,
  - and $B_i = 1$Ƀ is saved for flipping i back to 0.

v	$A[2]$	$\hat{c}_2$	d_2	$A[1]$	$\hat{c}_1$	d_1	$A[0]$	$\hat{c}_0$	d_0
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1	2	1
2	0	0	0	1	2	1	0	0	0
3	0	0	0	1	0	1	1	2	1
4	1	2	1	0	0	0	0	0	0
5	1	0	1	0	0	0	1	2	1
6	1	0	1	1	2	1	0	0	0
7	1	0	1	1	0	1	1	2	1

Amortized cost of BINARY COUNTER using accounting

For each **Increment** operation

- Cost of resetting bits to 0 is paid by credit.
- There is at most 1 flip $0 \to 1$, so the amortized cost is ≤ 2Ƀ.

As the total actual cost of n operations is upper bounded by sum of the amortized costs, then

Starting from the zero counter, a sequence of n **Increment** operations have a cost of $\leq 2n$, i.e. **Increment** need to flip at most $2n$ bits.

Dynamic table problem

- Tables are implemented as vectors with contiguous memory area to store objects.
- In many applications, it is not known in advance the size of the table, it may happen we run out of space in the middle of an application and dynamically we must increase the size. (Data streaming, online algorithms, size of hashing table)
- Each time the number of elements in a table is larger than the current size, we have to construct a new table with double size, copy the contains into the new space, delete the old table.
- Problem: We want to store n streaming keys in a table of size m. The goal is to make the m as small as possible, but large enough so that it won't overflow.

Dynamic tables

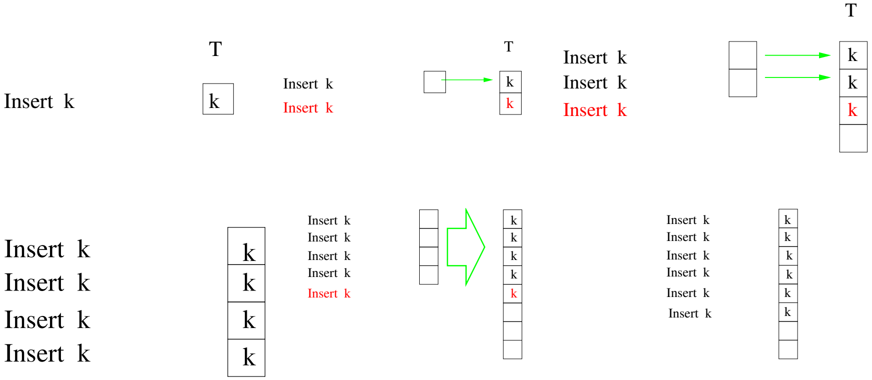
- We don't know in advance how many objects will be stored in the table.
- We start with table of size 2, and double if at the i-th. operation the table overflows: i.e. $i = 2^x + 1$.
- Whenever the table overflows:
  1. Allocate a new array of double size.
  2. Move all items from the old table into the new one.
  3. Free storage.
- Goal: To get $\Theta(1)$ amortized time per operation

Table insertion and expansion

Table Insertion (T, x)
$\text{elem}[T] = \text{size}[T] = 0.$
Create T: $\text{elem}[T] = 0, \text{size}[T] = 2$
if $\text{elem}[T] = \text{size}[T]$ **then**
  create T' with $\text{size}[T'] = 2\text{size}[T]$
  copy all elements in T into T'
  free T and rename T' as T
end if
insert x into T
$\text{elem}[T] = \text{elem}[T] + 1$

Example

Consider a sequence of insertions from the one-slot, empty table:



Cost of insertion and doubling

- Count only elementary insertions and copying, since all other costs together are constant per call.
- Charge 1 per elementary insertion.
- Let c_i be the cost of the ith. operation:

$$c_i = \begin{cases} 1 & \text{if } T \text{ does't expand,} \\ i & \text{if we double.} \end{cases}$$

- Therefore if n operations are performed, the worst case for insertions could have cost $= \Theta(n)$, then the cost of doing n insertions is $n \times \Theta(n) = \Theta(n^2)$. WRONG!. That is an over-counting, because we do not expand the table at each of the n insertion operations.

Aggregate method

Recall: For any n, show a sequence of n operations takes worst case time $T(n)$. Then the amortized cost per operation is $T(n)/n$.

$$c_i = \begin{cases} i & \text{if } i-1 = 2^x, \\ 1 & \text{otherwise.} \end{cases}$$

$$\begin{aligned} T(n) = \sum_{i=0}^{n} c_i \leq n + \sum_{j=1}^{\lfloor \lg n \rfloor} 2^j \\ = n + \frac{2^{\lfloor \lg n+1 \rfloor} - 1}{2-1} < n + 2n = 3n. \end{aligned}$$

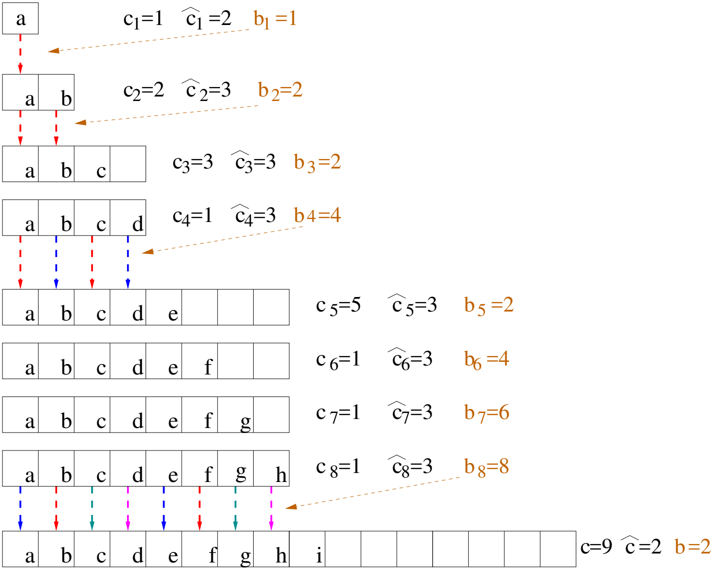
Thus the amortized cost is $3n/n = 3$.

i	1	2	3	4	5	6	7	8	9	10
$\lvert A \rvert$	2	2	4	4	8	8	8	8	16	16
$c(i)$	1	1	3	1	5	1	1	1	9	1

Dynamic tables: Accounting method

- We want to show that for all sequence of $m = f(n)$ operations, the sequence of $\hat{c}_i$ is an UB to the sequence of c_i,
$$\forall m > 0, \sum_{j=1}^{m} \hat{c}_j \geq \sum_{j=1}^{m} c_j.$$
- The goal is to keep the UB $\hat{c}_i$ as close to the real cost, as possible.
- If $i = 2^x + 1$, c_i is the expense of copying the previous elements and writing the new element, otherwise c_i is the expense of incorporating the new element.
- Charge $\hat{c}_1 = \hat{c}_2 = 2$Ƀ, for each other insertion $i > 2$, $\hat{c}_i = 3$Ƀ.
- At each operation $i > 2$, from $\hat{c}_i = 3$Ƀ, 1Ƀ pays for the insertion and if $i = 2^x + 1$, each register pays 1Ƀ for copying itself and and 1Ƀ to copy one of the earlier registers that have 0 Ƀ.

Example


a
$c_1=1$ $\widehat{c}_1=2$ $b_1=1$
a b
$c_2=2$ $\widehat{c}_2=3$ $b_2=2$
a b c
$c_3=3$ $\widehat{c}_3=3$ $b_3=2$
a b c d
$c_4=1$ $\widehat{c}_4=3$ $b_4=4$
a b c d e
$c_5=5$ $\widehat{c}_5=3$ $b_5=2$
a b c d e f
$c_6=1$ $\widehat{c}_6=3$ $b_6=4$
a b c d e f g
$c_7=1$ $\widehat{c}_7=3$ $b_7=6$
a b c d e f g h
$c_8=1$ $\widehat{c}_8=3$ $b_8=8$
a b c d e f g h i
$c=9$ $\widehat{c}=2$ $b=2$

Disjoint Set Union-Find

B. Galler, M. Fisher: An improved equivalence algorithm. ACM Comm., 1964; R.Tarjan 1979-1985.
See for example Ch. 21 of CLRS

- Union-Find is a data structure to maintain any collection of dynamic disjoint sets.
- Union-Find is one of the most elegant data structures in the algorithmic toolkit.
- Union-Find makes possible to design almost linear time algorithms for problems that otherwise would be unfeasible.
- Union-Find is a first introduction to an active research field in algorithmic; Self organizing data structures.

Some applications of Union-Find

- Kruskal's algorithm for MST.
- Dynamic graph connectivity in very large networks.
- Cycle detection in undirected graph.
- Random maze generation and exploration.
- Percolation.
- Strategies for games: Hex and Go.
- Least common ancestor.
- Compiling equivalence statements.
- Equivalence of finite state automata.

Partition and equivalent relations

Remember a **partition** of an n element set S is collection $\{S_1, \ldots, S_k\}$ of subsets s.t.:

$$\forall S_i \subseteq S; \cup_{i=1}^{k} S_i = S; \forall S_i, S_j \text{ then } S_i \cap S_j = \emptyset$$

.
Recall also that a partition implies an equivalence relation:

$$\forall x, y \in S, x \equiv y \text{ iff } x \in S_i \& y \in S_i.$$

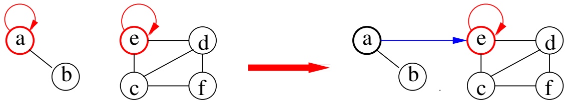
The collection $\{S_1, \ldots, S_k\}$ are the equivalence classes of the equivalence relation.

Union-Find

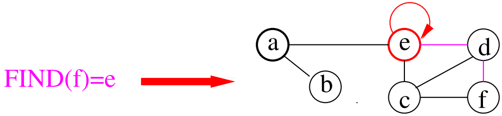
Union-Find is a data structure that supports three operations on partitions of a set:

MAKESET (x): creates a new set containing the single element x.

UNION (x, y): Merge the sets containing x and y, by using their union.



FIND (x): Return the representative of the set containing x.



Warning about UNION operation

Warning: For any $x, y \in S$ we can do UNION(x, y), x, y do not need to be representatives, but from the point of view to study the complexity of the different implementations, and separate the complexity of the operations UNION and FIND, we would consider that if x and/or y are not representatives then

$$\text{UNION }(x, y) = \text{UNION }(\text{FIND}(x), \text{FIND}(y)).$$

Union-Find Data Structure: The basic working

Given a set S of size n, construct a data structure that maintains a collection $\{S_1, \ldots, S_k\}$ of disjoint dynamic sets, each set identified by a *representative*,.

We have n initial elements a set S, we start by applying n times MAKESET to have n single element sets.

After, we want to implement a sequence of m UNION and FIND operations on the initial sets, using the minimum number of steps.

Union find is a nice data structure, for dynamic settings, where membership evolves with time.

Graph connectivity

- Consider friendship network, which is a social network on S people, where each person is represented by a node x, and the set of (undirected edges) is defined: $x, y \in S$, $(x, y) \in E$ if x and y are friends Those networks could be very large.
- Let $G = (S, E)$ represent the friendship network.
- An interested problem is social modeling is identifying connected components in $G = (S, E)$.
- A connected component is a maximal set of vertices that are connected.
- That is, a connected component is a $C \subset S$ such that:
  - For every $x, y \in C$ either $(x, y) \in E(C)$ or there is a path $x \rightsquigarrow y$.
  - No $x \in C$ has friends outside of C.

Testing if two elements are in the same connected component (FIND)

Given $G = (S, E)$ decomposed in their connected components, $C_1, \ldots, C_n$, and two $x, y \in S$.

SAME-COMPO (u, v)
if FIND (u) = FIND (v) **then**
  return true
else
  return false
end if

Find the larger connecter components in a graph: (UNION)

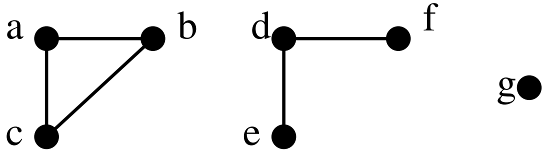
Given a set S of n people with a set $E = \{(x, y)\}$ of m pairs of friends from S, let $G = (S, E)$ be the friendship network.
The output is for each person $x \in S$, the representative of the connected component to which x belongs.

CONNECTED-COMP. $G = (S, E)$
for each $x \in S$ **do**
  MAKESET (x)
end for
for each $(x, y) \in E$ **do**
  if FIND $(x) \neq$ FIND (y) **then**
    UNION (x, y)
  end if
end for

We will see that using Union-Find, we would implement the algorithm in effectively linear time.

Graph connectivity: Example

Given $G = (S, E)$ with $S = \{a, b, c, d, e, f, g\}$, and $E = \{(a, b), (a, c), (c, b), (de), (d, f)\}$,



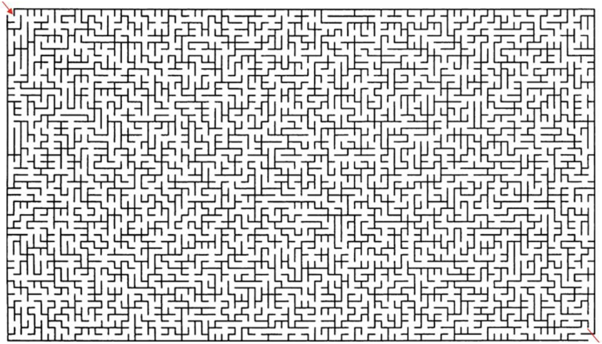
{a}, {b}, {c}, {d}, {e}, {f }, {g},

{a}, {b c}, {e , d }, {f }, {g },

{ a b c }, {e d f}, {g }

Dynamic Connectivity

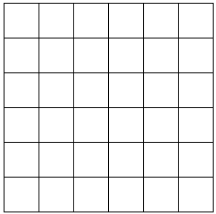
It can be used to mode many different problems with different kinds of objects:

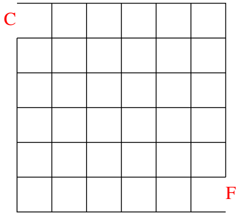


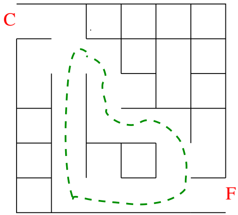
Building Mazes

- Start from a grid G with n cells and E wall between the cells
- Build a maze by erasing walls
- Define an start C and an end F
- Iterate picking random walls and deleting them, taking care. of not picking a wall separating adjacent vertices, which are in the same component: This is to avoid cycles.
- Do not delete boundary edges (except for C and F).
- Every cell should be reachable from every other cell.
- The paths should define a tree.

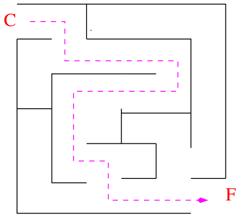
Building paths in Mazes







Cycle



Building Mazes: Union-find

Given a grid $G = (V, E)$, where V are the n cells and the m valls $E = \{(x, y) \mid x, y \in V\}$, together with a C and a F, we want to output a $R \subset E$, such that removing R produces a valid maze.

1. Consider the set V of cells and number them $[n]$
2. Consider the set of walls $E = \{(i, j)\}$ between adjacent cells $i, j \in V$
   - 2.1 For every cell i MAKESET$(i) = \{i\}$
   - 2.2 Iterate until having a path $C \rightsquigarrow F$
   - 2.3 Take a wall $(i, j) \in E$
   - 2.4 If FIND(i) $\neq$FIND(j) Erase the wall make UNION(FIND(i),FIND(j))
   - 2.5 Enditerate

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

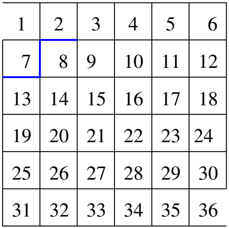
Building Mazes: Algorithm

Recall $|V| = n, |E| = m$:

MAZE $(V, E), C, F, \ R = \emptyset$
for each $x \in V$ **do**
  MAKESET (x)
end for
while $|R| < n - 1$ **do**
  Select uniformly at random $(y, z) \in E$
  Remove (y, z) from E
  if $x =$ FIND $(y) \neq w =$ FIND (y) **then**
    UNION (x, w)
    $R = R \cup \{(y, z)\}$
  end if
end while

Example Mazes: First iterations

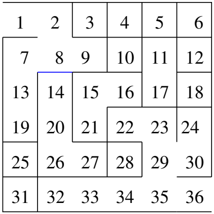
After MAKESET: $V = \{\{1\}, \{2\}, \ldots, \{36\}\}$
Edge $(7, 13)$: $V = \{\{1\}, \{2\}, \ldots, \{7, 13\}, \ldots \{36\}\}$
Edge $(2, 8)$: $V = \{\{1\}, \{2, 8\}, \ldots, \{7, 13\}, \ldots \{36\}\}$
Edge $(7, 8)$: $V = \{\{1\}, \{2, 7, 8, 13\}, \ldots, \{36\}\}$

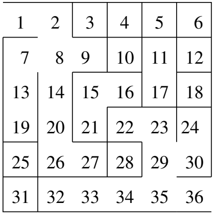


Example Mazes: Intermediate

Start
$V = \{\{1, 2, 7, 8, 9, 13, 19\}, \{3\}, \{4\}, \ldots, \{11, 17\}, \{14, 20, 26, 27\},$
$\{15, 16, 21\}, \{22, 23, 24, 29, 30, 32, 33, 34, 35, 36\}\}$

choose $(8, 14)$ $\Rightarrow$ FIND(8)=7$\neq$ 20= FIND (14) $\Rightarrow$ UNION(7, 14)
New $V = \{\{1, 2, 7, 8, 9, 13, 14, 19, 20, 26, 27\}, \ldots\}$

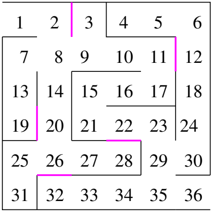




Example Mazes: Intermediate

Final configuration:
$V = \{1, 2, \ldots, {\color{red}7}, 8, \ldots, 36\}$

At the given pattern, the edges $\{(2,3), (11,12), (19,20), (26,32)\}$, should not be selected



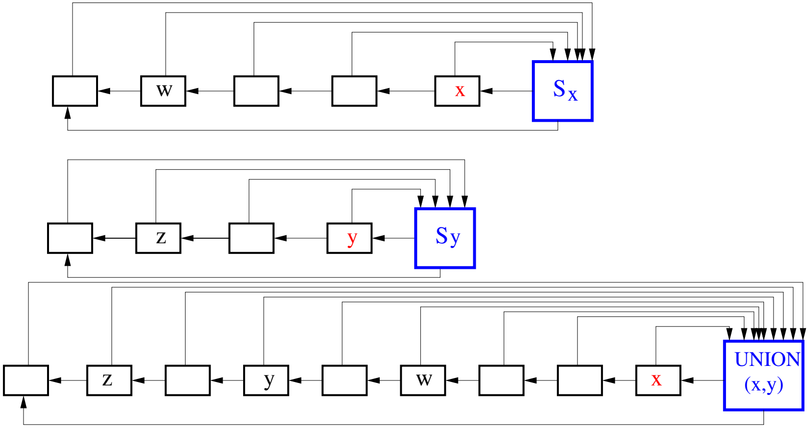
How do we implement union find?

Union-Find implementation: Linked list

Given $\{S_1, \ldots, S_k\}$,

• Each set S_i represented by a linked list, with a header

• The *representative* of S_i is defined to be the element at the head of the list representing the set.



Union-find: Linked-list

For the implementation and complexity of union-find, see for ex. Sec. 5.1.4 of Dasgupta, Papadimitriou, Vazirani.

Given sets S_x and S_y, we use the following 3 operations:

- MAKESET (x): Initializes x as a lone list. Worst time $\Theta(1)$.
- UNION (z, w): Find the representative y, and point to the tail of the list S_x implementing $S_x \cup S_y$.
- FIND (z): Goes from z to the head of S_x and then to the representative of the set. Worst case $O(1)$.

Complexity of UF by linked-lists: Worst case analysis

We have a set of n elements $\{x_1, \dots, x_n\}$ and m applications of MAKESET and UNION. Notice $m = 2n - 1$:

1. We start with $x_1, x_2, \dots, x_n$, do n MAKESET. Total cost $O(n)$,
2. do UNION(x_1, x_2), UNION(x_2, x_3), ..., UNION(x_{n-1}, x_n),

In worst case the cost is $n + \sum_{i=1}^{n-1} i = \Theta(n^2)$.

Amortized analysis: A quick view

Using the aggregate amortized analysis we get that the average cost of each operation is $\Theta(n^2)/(2n-1) \sim \Theta(n)$.

Union-find: Linked-list

We use the following heuristic to implement UNION (z, w):
Append the smallest list to the larger one.
This implies that all pointers in the shortest set must be updated.

- MAKESET (x): Initializes x as a lone list. Worst time $\Theta(1)$.
- UNION (z, w): Find the representative y, and point to the tail of the list S_x implementing $S_x \cup S_y$. Worst time $O(\min\{|S_x| + |S_y\}|)$. As it could be $|S_x| = |S_y| = n/2$, then UNION needs $O(n)$ steps.
- FIND (z): Goes from z to the head of S_x and then to the representative of the set. Worst case $O(1)$.

Complexity of UF by linked-lists: Amortized analysis

Theorem *Using the linked-list implementation of UF, with the max length heuristic for* UNION, *a sequence of m* MAKESET, UNION, FIND *operations, n of which are* MAKESET, *takes $O(n + m \lg n)$ steps.*
Moreover, the amortized running time of each UNION *operation is $O(\log n)$, and the amortized running time for each* MAKESET *and* FIND *is $\Theta(1)$.*

Complexity of UF by linked-lists: Proof of the Theorem

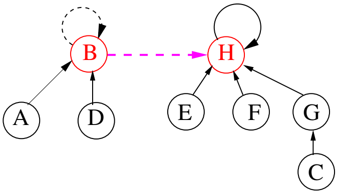
Proof

- If we have n elements, the number of *UNION* operations is $\leq n - 1$.
- Each time we do UNION(x, y), we double the size of the smaller set,
- therefore any element x will update its pointers a maximum of $\lg n$ times,
- therefore the total cost of updating pointers due to the application of *UNION* is $O(n \lg n)$.
- Therefore, the total time for the entire m-sequence is $O(n + m \lg n)$.
- As number of UNIONS is $\leq n - 1$, The amortized cost for each UNION operation is $O(n \lg n/n) = O(\lg n)$.
- The amortized cost for each MAKESET and FIND operation is $\Theta(1)$. $\square$

Union-Find implementation: Link by size of forest of Trees

- Stating from initial n singleton trees, represent each set as a tree of elements, where the root contain the representative of the tree.
- As a DS, this representation of a tree has only parent links, and does not provide a way to access the children of a given node.

• MAKESET (x): $\Theta(1)$
• FIND (z): find the root of the tree containing z. $\Theta(\text{height})$
• UNION (x, y): make the root of the tree with less elements point to the root of the tree with more elements.



Complexity of Link by size

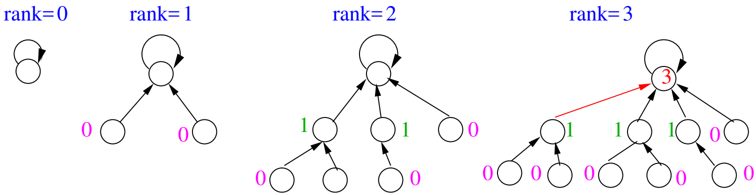
- The cost of MAKESET(x) is $\Theta(1)$ and the cost of FIND(z) is $O(\lg n)$
- Notice that for x, y representatives of S_x and S_y, the cost of UNION (x, y) is $\Theta(1)$.
  If z, w are not representatives, to implement UNION (w, z) we must first do FIND(z) and FIND(w).

Doing a sequence of m MAKESET, FIND, UNION *operations, starting from n elements, using the link-by-size tree implementation of UF structure, takes $O(n + m \lg n)$ steps.*

So the complexity of UF is the same for both implementations: Link by size of forest of Trees and linked-list with the proposed heuristic.

First Heuristic: Link by rank

- Define the rank $r(x)$ as height of subtree rooted at x.
- Notice It is possible for two trees that the one with less nodes has higher root than the one with less nodes.
- Any singleton element has rank=0.
- Inductively as we join trees, only the rank of the root increases.

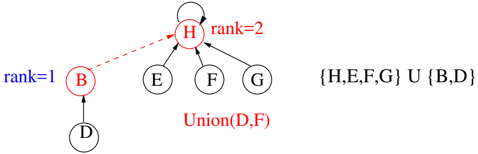


Trees: Link by rank

Union rule: Link the root of smaller rank tree to the root of a tree with larger rank.

In case the roots of both trees have the same rank, choose arbitrarily and increase +1 the rank of the winer.

Except for the root, a node does not change rank during the process



• UNION (x, y): is $\Theta(1)$ if x, y are roots, otherwise we have to apply first climbs to the roots of the tree containing FIND(x) and FIND(y), which takes Θ(height) steps.

Link by rank

Maintain an integer rank for each node, initially 0. Link root of smaller rank to root of larger rank; if a tie, increase rank of new root by 1.

Let $p(z)$ be the parent of (z) in the forest, and let $r(z)$ be the rank of (z). Assume x and y are the roots of the trees.

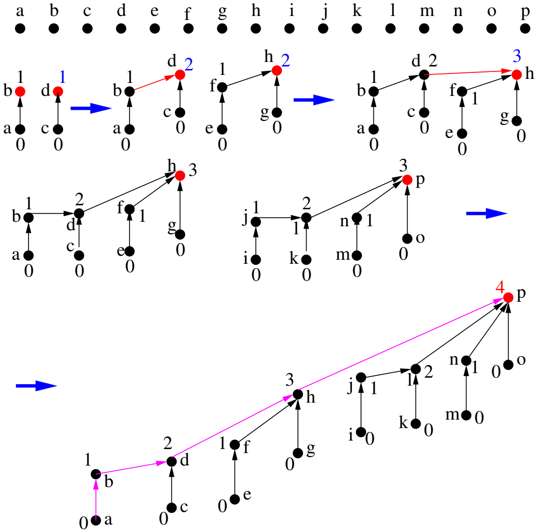
MAKESET (x)
$p(x) = x$
$r(x) = 0$

FIND (z)
while $z \neq p(z)$ **do**
  $z = p(z)$
end while

UNION (x, y)
if $x = y$ **then**
  STOP
else if $r(x) > r(y)$ **then**
  $p(y) = x$
else if $r(x) < r(y)$ **then**
  $p(x) = y$
else
  $p(x) = y$
  $r(y) = r(y) + 1$
end if

Worst case example

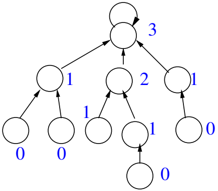
Worst case example that we can have $r(\text{root}) = \lg n$.



Properties of forests constructed by using Link by rank

P1.- If x is not a root then $r(x) < r(p(x))$

P2.- If $p(x)$ changes then $r(p(x))$ increases



P3.- For any node x, let $N(x)$ be the number of nodes in the subtree rooted at x. If $r(x) = k$, then $N(x) \geq 2^k$.

Proof (Induction on k) True for $k = 0$, if true for k then a node of rank k results from the merging of 2 nodes with rank $k - 1$, so $2^{k-1} + 2^{k-1} = 2^k$. □

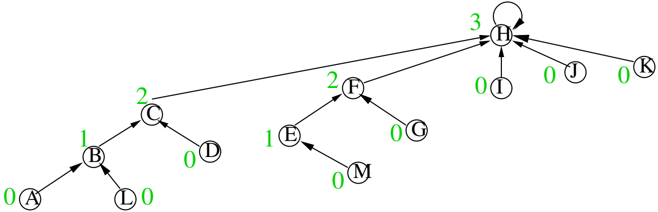
Properties of Link by rank

P4.- The highest rank of a root is $\leq \lfloor \lg n \rfloor$

Proof Follows P1 and P3. $\square$

P5.- For any $r \geq 0$, there are $\leq n/(2^r)$ nodes with rank r.

Proof By (P4) a root x with $r(x) = k$ has $\geq 2^k$ descendants.
Any non-root node y with $r(y) = k'$ has $\geq 2^{k'}$ descendants.
As it is a tree, different nodes with rank $= k$ can't have common descendants. $\square$



rank 0=7; rank 1=2; rank 2=2; rank 3=1

$n/2^k$ rank 0<13; rank1 <6; rank 2<3 rank 3=1

Complexity of Link by rank

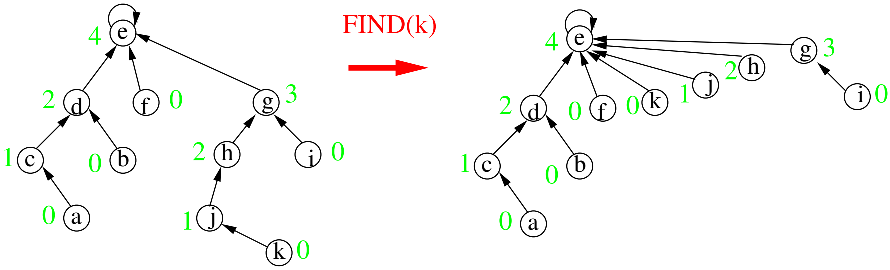
As for FIND, the number of steps is bounded by the height of the tree, then

Lemma *Using link-by-rank, each application of* FIND *takes* $O(\lg n)$ *steps.*

Lemma *Starting from an empty data structure with n elements, the performance of union-find implemented using link-by-rank, for any n* MAKESET *and any intermixed sequence of m* FIND *and* UNION *operations is* $O(n + m \lg n)$ *steps.*

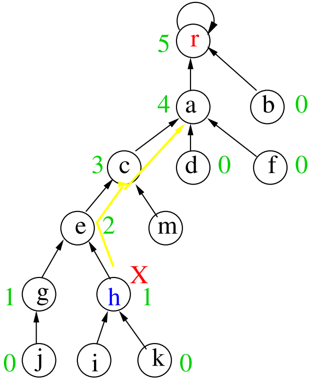
Final Heuristic: Path compression with link-by-rank

To improve the $O(\log n)$ amortized cost per FIND in union-bound with link-by-rank, we keep the trees "as flat" as possible.
We use the path compression: At each use of FIND(x) we follow all the path $\{y_i\}$ of nodes from x to the root r change the pointers of all the $\{y_i\}$ to point to r.



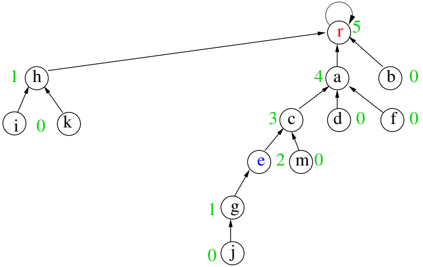
Path compression: Function

FIND (x)
if $x \neq p(x)$ **then**
  $p(x) =$ **FIND** $p(x)$
  return $p(x)$
end if



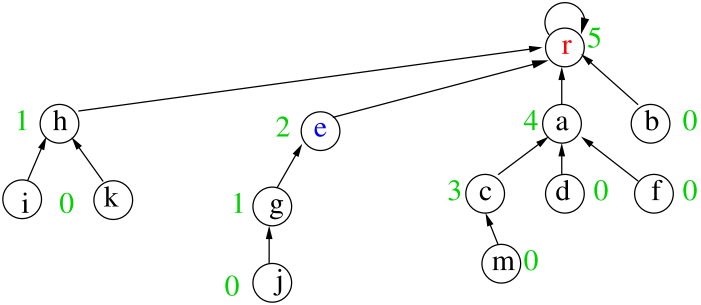
Path compression: Function

FIND (x)
if $x \neq p(x)$ **then**
  $p(x) =$ **FIND** $(p(x))$
  return $p(x)$
end if



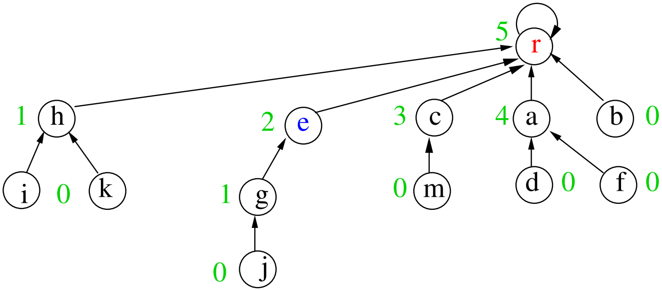
Path compression: Function

FIND (x)
if $x \neq p(x)$ **then**
  $p(x) =$ **FIND** $(p(x))$
  return $p(x)$
end if



Path compression: Function

FIND (x)
if $x \neq p(x)$ **then**
  $p(x) =$ **FIND** $(p(x))$
  return $p(x)$
end if



Union-Find: Link by rank with path compression

This implementation of the data structure is the one that reduces the complexity of making a sequence of m UNION and FIND operations.

Key Observation: Path compression does not create new roots, change ranks, or move elements from one tree to the another

As a corollary, the properties of Link by rank also hold for this implementation.

Notice, the FIND operations only affect the inside nodes, while the UNION operations only affect roots.

Thus, compression has no effect on UNION operations.

The iterated logarithm

The iterated logarithm $\lg^*(n)$ is defined: $\lg^*(n) = k$ if k is the smallest integer such that $\lg^k n = \Theta(1)$.

Recall: $\lg^k n = \underbrace{\lg(\lg(\lg(\cdots(\lg n)\cdots)))}_{k-\text{times}}$.

The formal definition:

$$\lg^*(n) = \begin{cases} 0 & \text{if } n \leq 1, \\ 1 & \text{if } n = 2, \\ 1 + \lg^*(\lg(n)) & \text{if } n > 2. \end{cases}$$

n	$\lg^* n$
1	0
2	1
[3,4]	2
[5,16]	3
[17,65536]	4
[65537,10^{19728}]	5

Why for all practical purposes $\lg^*(n) \leq 5$

We have that if $\lg^*(n) \leq 5$, then for all $n \leq 2^{2^{16}} \sim 10^{19728}$.

Intuitive argument: If we consider that in any computer, each memory bit has size ≥ 1 atoms, using the canonical estimation that the number of atoms in the universe is $\sim 10^{83}$, we can conclude the size of any computer memory is $< 10^{83}$. Therefore it is impossible with today technology, that you can manipulate sets with size 10^{83}

Therefore, we can consider that for any n, $\lg^*(n) \leq 5$.

Main result

The following result was due to J. Hopcroft and J. Ullman: SIAMJC 1973

Theorem

Starting from an empty data structure with n disjoint single sets, link-by-rank with path compression performs any intermixed sequence of m FIND *and* UNION *operations in* $O(m \lg^* n)$ *steps.*

The proof uses an aggregate amortized analysis argument: look at the sequence of FIND and UNION operations from an empty and determine the average time per operation. The amortized costs turns to be $\lg^* n$ (basically constant) instead of $\lg n$.

Recall: Properties 1-6

1. P-1 If x is not a root then $\text{rank}(x) < \text{rank}(\text{parent}(x))$
2. P-2 If $r(x)$ changes then it is a root and $r(x)$ increases.
3. P-3 A root with rank k has $N \geq 2^k$ nodes in its tree.
4. P-4 If there are n elements, their rank values are $\leq \lfloor \lg n \rfloor$.
5. P-5 For any integer $k \geq 0$ there are $\leq n/2^k$ nodes with rank k.
6. P-6 New The ranks of all nodes are unchanged by path compression, those values can not longer be interpreted as tree heights.

Analysis

- Divide non-zero ranks into the following intervals:

$$\underbrace{\{1\}}_{k=0}, \underbrace{\{2\}}_{k=1}, \underbrace{\{3,4\}}_{k=2}, \underbrace{\{5,\ldots,16\}}_{k=3}, \underbrace{\{65537,\ldots,2^{65536}\}}_{k=4}, \ldots \underbrace{\{k+1,\ldots,2^k\}}_{k\text{group}}.$$

- By P-4, every non-zero rank in a Union-find implemented with link by rank with path compression falls within the first $\lg^* n$ intervals.
- We give a credit of 2^k Ƀ to any non-root node with rank within $\{k+1,\ldots,2^k\}$
- By P-5, the number of nodes with rank $\geq k+1$ is $\leq \frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \cdots \leq \frac{n}{2^k}$, then nodes in interval k need $\leq n$ Ƀ.
- As there are $\leq \lg^* n$ intervals:
  The number of Ƀ given to all nodes is $\leq n \lg^* n$.

Summary

Given from empty data structure with n disjoint single sets, on which there is a sequence of m FIND and UNION operations, the following table gives summary of the time-complexity for the different implementations of union-bound:

Implementation	cost
Linked list	$O(n + m \lg n)$
Link by size	$O(n + m \lg n)$
Link by rank	$O(n + m \lg n)$
Rank + path compression	$O(n + m \lg^* n) \simeq O(n + m)$

Notice UF in inherently sequential method, so parallel would not help that much.