CSE 5311 Notes 8: Disjoint Sets

CLRS, Chapter 21

Problem: For an equivalence relation:

1. Determine if two elements are equivalent (FIND), and

2. Allows merging (UNION) of equivalence classes.

Naive implementation - indicate subset for each element

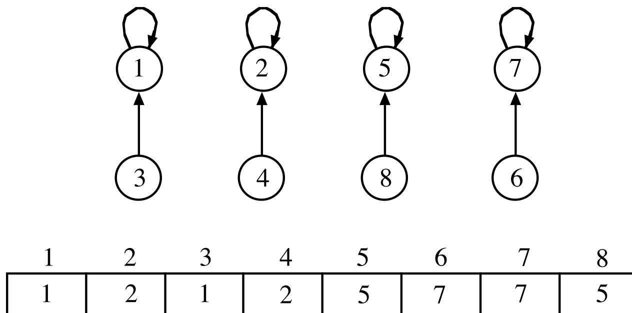
1	2	3	4	5	6	7	8
1	2	1	2	3	4	4	3

Represents equivalence relation:

{1, 3} {2, 4} {5, 8} {6, 7}

UNION takes O(n) time - can do much better!!!!

Galler-Fischer Representation - Use trees (in an array) with just parent pointers



1	2	3	4	5	6	7	8
1	2	1	2	5	7	7	5

Trade-off:

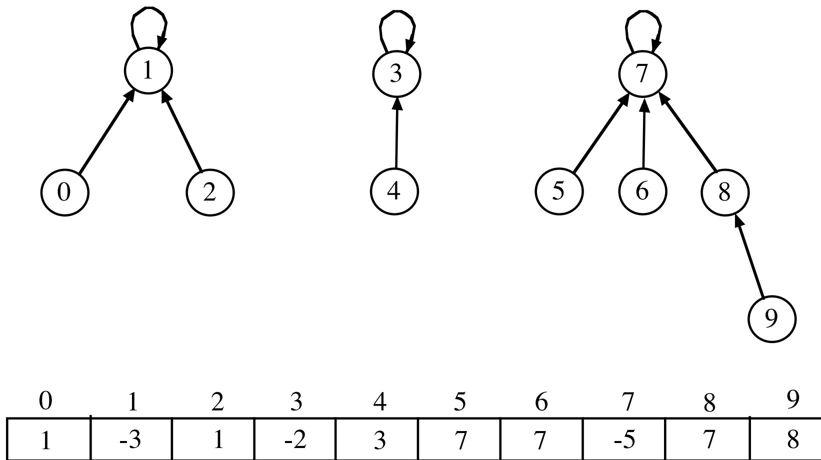
Increase in time to check equivalence (FIND)

vs.

Simplicity in merging (UNION) - redirect one root to another, then apply heuristics to reduce depth

UNION-BY-WEIGHT (size)

Keep subtree size in root. If integer tables, then negative value for pointer indicates that the root's size (negated) is stored.



0	1	2	3	4	5	6	7	8	9
1	-3	1	-2	3	7	7	-5	7	8

Theorem: For any node x with height $h(T_x)$ in union-by-weight and size $s(T_x)$, $2^{h(T_x)} \le s(T_x)$.

Proof: By induction

Single node (as initialized):

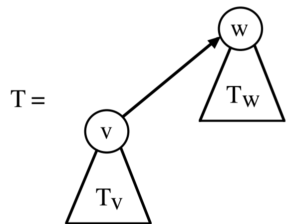
$s(T_x) = 1$ and $h(T_x) = 0$

So, $2^{h(T_x)} = 2^0 \le s(T_x)$

Property holds before union and holds afterwards:

Suppose T_v and T_w are to be unioned. WOLOG, $s(T_v) \le s(T_w)$.

$2^{h(T_v)} \le s(T_v)$ and $2^{h(T_w)} \le s(T_w)$



Show that $2^{h(T)} \le s(T)$:

$$2^{h(T)} = 2^{\max(1+h(T_v),h(T_w))} = \max\left(2^{1+h(T_v)}, 2^{h(T_w)}\right)$$

$\le \max(2s(T_v), s(T_w))$ $2^{h(T_v)} \le s(T_v)$ and $2^{h(T_w)} \le s(T_w)$

$\le \max(s(T), s(T_w))$ $s(T_v) \le s(T_w)$ and $s(T_v) + s(T_w) = s(T)$

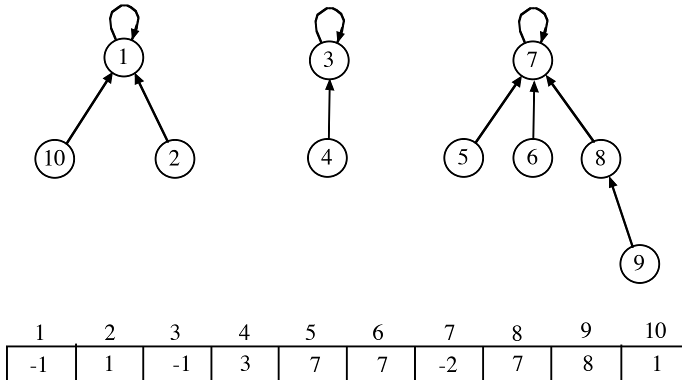
$\le \max(s(T), s(T)) = s(T)$ $s(T_w) \le s(T)$

Corollary: $h(T) \le \log s(T)$

So, FINDs under union-by-weight take O(log n)

UNION-BY-RANK (height)

Keep subtree rank (height) in root.



1	2	3	4	5	6	7	8	9	10
-1	1	-1	3	7	7	-2	7	8	1

Theorem: For any node x with rank $r(T_x)$ in union-by-rank and size $s(T_x)$, $2^{r(T_x)} \le s(T_x)$

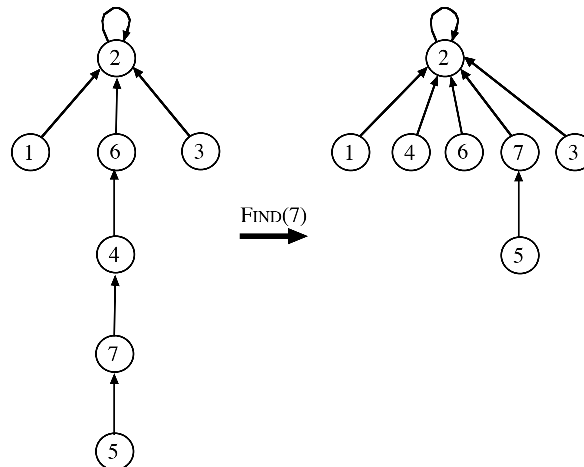
Proof: Very similar to union-by-weight.

Corollary: $r(T) \le \log s(T)$

So, FINDs under union-by-rank take O(log n)

PATH COMPRESSION

After a FIND reaches a tree's root, a second pass along the path makes every node point directly to the root.



Can easily combine with union-by-weight or union-by-rank.

Under union-by-rank, path compression causes each rank to be just an upper bound on the height.

In addition, the amortized cost of FIND and UNION will be *nearly* constant (inverse of extremely fast-growing function).

APPLICATIONS

1. Kruskal's Minimum Spanning Tree

    Sort edges in ascending order.

    Place each vertex in its own set.

    Process each edge {x, y} in sorted order:

    ```
    a=FIND(x)
    b=FIND(y)
    if a ≠ b
        UNION(a,b)
        Include {x, y} in MST
    ```

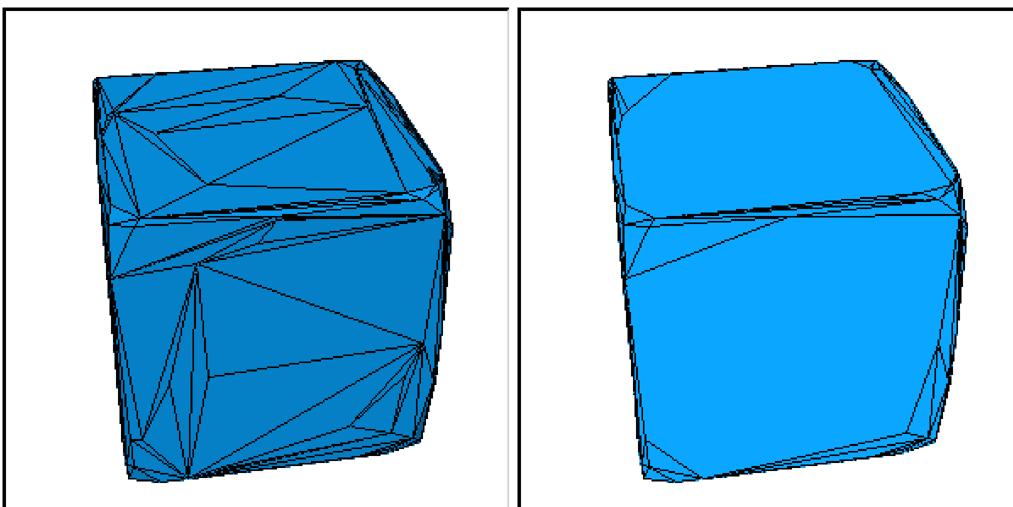
2. Many parallel algorithms use similar ideas. (See books by Ja Ja or Reif)

3. Connected components.

4. First-order unification / logic programming (ACM Computing Surveys 21:1, March 1989, fig. 4)

5. Off-line least common ancestors (CLRS, p. 521)

6. Find co-planar triangles in 3-d convex hull (Spring 2005 CSE 5392):



CSE 5311 Notes 9: Hashing

CLRS, Chapter 11

Review: 11.2: Chaining - related to perfect hashing method

11.3: Hash functions, skim universal hashing

11.4: Open addressing

COLLISION HANDLING BY OPEN ADDRESSING

Saves space when records are small and chaining would waste a large fraction of space for links.

Collisions are handled by using a *probe sequence* for each key – a permutation of the table's subscripts.

Hash function is h(key, i) where i is the number of reprobe attempts tried.

Two special key values (or flags) are used: *never-used* (-1) and *recycled* (-2). Searches stop on *never-used*, but continue on *recycled*.

Linear Probing - h(key, i) = (key + i) % m

Properties:

1. Probe sequences eventually hit all slots.

2. Probe sequences wrap back to beginning of table.

3. Exhibits lots of *primary clustering* (the end of a probe sequence coincides with another probe sequence):

   $i_0\ i_1\ i_2\ i_3\ i_4 \ldots i_j\ i_{j+1} \ldots$
   $i_j\ i_{j+1}\ i_{j+2} \ldots$

4. There are only m probe sequences.

5. Exhibits lots of *secondary clustering*: if two keys have the same initial probe, then their probe sequences are the same.

What about using h(key, i) = (key + 2*i) % 101 or h(key, i) = (key + 50*i) % 1000?

Suppose all keys are *equally likely* to be accessed. Is there a best order for inserting keys?

Insert keys: 101, 171, 102, 103, 104, 105, 106

Slot	Key
0	
1	
2	
3	
4	
5	
6	

Slot	Key
0	
1	
2	
3	
4	
5	
6	

Double Hashing – $h(key, i) = (h_1(key) + i*h_2(key)) \% m$

Properties:

1. Probe sequences will hit all slots only if m is prime.
2. $m*(m-1)$ probe sequences.
3. Eliminates most clustering.

Hash Functions:

$h_1 = key \% m$

a. $h_2 = 1 + key \% (m-1)$

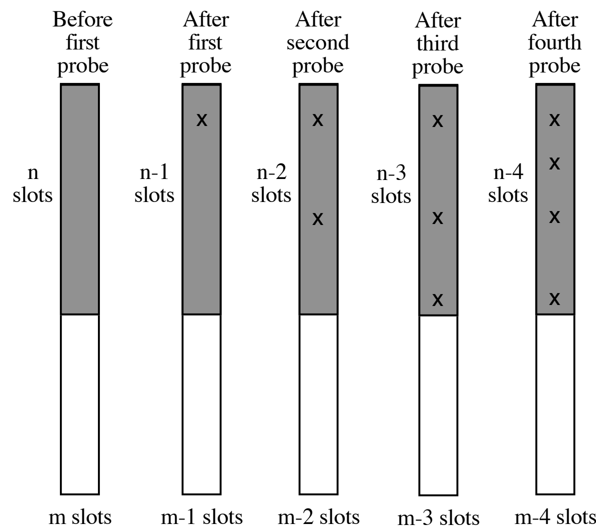
b. $h_2 = 1 + (key/m) \% (m-1)$

c. Use last few bits of key as h_2, but must avoid zero.

UPPER BOUNDS ON EXPECTED PERFORMANCE FOR OPEN ADDRESSING

Double hashing comes very close to these results, but analysis assumes that hash function provides all m! permutations of subscripts.

1. Unsuccessful search with load factor of $\alpha = \frac{n}{m}$. Each successive probe has the effect of decreasing table size and number of slots in use by one.



a. Probability that all searches have a first probe $\quad 1$

b. Probability that search goes on to a second probe $\quad \alpha = \frac{n}{m}$

c. Probability that search goes on to a third probe $\quad \alpha \frac{n-1}{m-1} < \alpha \frac{n}{m} < \alpha^2$

d. Probability that search goes on to a fourth probe $\quad \alpha \frac{n-1}{m-1} \frac{n-2}{m-2} < \alpha^2 \frac{n-2}{m-2} < \alpha^3$

. . .

Suppose the table is large. Sum the probabilities for probes to get upper bound on expected number of probes:

$$\sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha} \quad \text{(much worse than chaining)}$$

2. Inserting a key with load factor α

   a. Exactly like unsuccessful search

   b. $\frac{1}{1-\alpha}$ probes

3. Successful search

   a. Searching for a key takes as many probes as inserting *that particular key*.

   b. Each inserted key increases the load factor, so the inserted key number i + 1 is expected to take no more than

$$\frac{1}{1-\frac{i}{m}} = \frac{m}{m-i} \text{ probes}$$

c. Find expected probes for n consecutively inserted keys (each key is equally likely to be requested):

$$\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} = \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} \qquad \text{Sum is } \frac{1}{m}+\frac{1}{m-1}+\ldots+\frac{1}{m-n+1}$$

$$= \frac{m}{n}\sum_{i=m-n+1}^{m}\frac{1}{i}$$

$$\le \frac{m}{n}\int_{m-n}^{m}\frac{1}{x}dx \qquad \text{Upper bound on sum for decreasing function. CLRS, p. 1067 (A.12)}$$

$$= \frac{m}{n}\left(\ln m - \ln(m-n)\right) = \frac{1}{\alpha}\ln\frac{m}{m-n} = \frac{1}{\alpha}\ln\frac{1}{1-\alpha} = -\frac{1}{\alpha}\ln(1-\alpha)$$

BRENT'S REHASH - On-the-fly reorganization of a double hash table

During insertion, moves no more than one other key to avoid usual penalty on recently inserted keys.

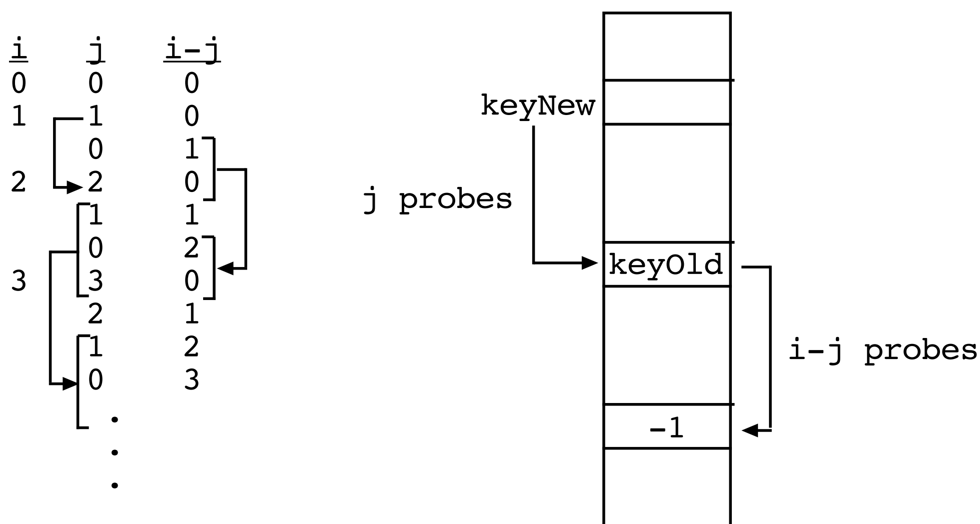
Expected probes for successful search ≤ 2.5. (Assumes uniform access probabilities.)

Insertion is more expensive, but typically needs only three accesses per key to balance.

Unsuccessful search performance is the same.

```
void insert (int keyNew, int r[])
{
int i, ii, inc, init, j, jj, keyOld;

init = hashfunction(keyNew);
inc = increment(keyNew);
for (i=0;i<=TABSIZE;i++)
{
  printf("trying to add just %d to total of probe lengths\n",i+1);
  for (j=i;j>=0;j--)
  {
    jj = (init + inc * j) % TABSIZE;
    keyOld = r[jj];
    ii = (jj + increment(keyOld) * (i - j)) % TABSIZE;
    printf("i=%d j=%d jj=%d ii=%d\n",i,j,jj,ii);
    if (r[ii] == (-1))
    {
      r[ii] = keyOld;
      r[jj] = keyNew;
      n++;
      return;
    }
  }
}
}
```



PERFECT HASHING (CLRS 11.5)

Static key set

Obtain O(1) hashing (“no collisions”) using:

1. Preprocessing (constructing hash functions)

   and/or

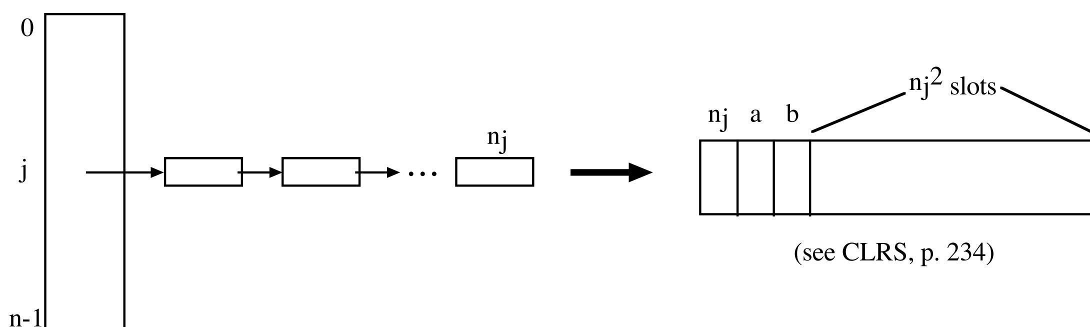
2. Extra space (makes success more likely) - want cn, where c is small

Many informal approaches - typical application is table of reserved words in a compiler.

11.5 approach:

1. Suppose n keys and $m = n^2$ slots. Randomly assigning keys to slots gives prob. < 0.5 of any collisions.

2. Use two-level structure (in one array):



(see CLRS, p. 234)

$\sum E\left[n_j^2\right] < 2n$, but there are three other values for $n_j > 1$ and one other value when $n_j \le 1$.

Brent’s method - about 1.8 million keys

```
0.909999 l.f. double=2.646082 brent=1.858524 CPU 23.436005
0.919999 l.f. double=2.745343 brent=1.893388 CPU 24.563019
0.929999 l.f. double=2.859402 brent=1.932261 CPU 25.894732
0.939999 l.f. double=2.992969 brent=1.976763 CPU 27.540789
0.949999 l.f. double=3.153377 brent=2.029128 CPU 29.725117
0.959999 l.f. double=3.352963 brent=2.093167 CPU 32.841255
0.969999 l.f. double=3.614963 brent=2.177511 CPU 38.089577
Retrievals took 8.176472 secs
Worst case probes is 90
Probe counts:
Number of keys using 1 probes is 932302
Number of keys using 2 probes is 449570
Number of keys using 3 probes is 201999
Number of keys using 4 probes is 96512
Number of keys using 5 probes is 49356
Number of keys using 6 probes is 26496
Number of keys using 7 probes is 14911
Number of keys using 8 probes is 9126
Number of keys using 9 probes is 5610
Number of keys using 10 probes is 3728
Number of keys using 11 probes is 2595
Number of keys using 12 probes is 1774
Number of keys using 13 probes is 1234
Number of keys using 14 probes is 878
Number of keys using 15 probes is 738
Number of keys using 16 probes is 507
Number of keys using 17 probes is 439
Number of keys using 18 probes is 356
Number of keys using 19 probes is 282
Number of keys using 20 probes is 234
Number of keys using 21 probes is 198
Number of keys using 22 probes is 164
Number of keys using 23 probes is 139
Number of keys using 24 probes is 119
Number of keys using 25 probes is 92
Number of keys using 26 probes is 65
Number of keys using 27 probes is 75
Number of keys using 28 probes is 73
Number of keys using 29 probes is 52
Number of keys using >=30 probes is 378
```

CLRS Perfect Hashing - 2 million keys

```
malloc'ed 24000000 bytes
realloc for 25277908 more bytes
final structure will use 16.638954 bytes per key
Subarray statistics:
Number with 0 keys is 734741
Number with 1 keys is 737974
Number with 2 keys is 366899
Number with 3 keys is 122234
Number with 4 keys is 30648
Number with 5 keys is 6287
Number with 6 keys is 1041
Number with 7 keys is 160
Number with 8 keys is 13
Number with 9 keys is 1
Number with 10 keys is 2
Number with 11 keys is 0
Number with 12 keys is 0
Number with 13 keys is 0
Number with >=14 keys is 0
Time to build perfect hash structure 48.047688
Time to retrieve each key once 17.372749
```

RIVEST'S OPTIMAL HASHING

Application of the assignment problem (weighted bipartite matching)

m rows

n columns (m ≤ n)

Positive weights

Choose an entry in each row of M such that:

No column has two entries chosen.

The *sum* of the chosen entries is minimized.

Solved using the *Hungarian method.*

Minimizing *expected* number of probes by *placement* of keys to be retrieved by a specific open addressing technique:

Static set of m keys for table with n slots. Key i has probability P_i.

Follow probe sequence for key i: $S_{i1}, S_{i2}, \cdots, S_{im}$.

Assign weight jP_i to $M_{iS_{ij}}$. (Remaining entries get ∞.)

Solve resulting assignment problem (e.g. Hungarian method in $O\left(m^2 n\right)$ time).

If M_{ij} is chosen, then store key i at slot j.

Minimizing *worst-case* number of probes:

No probabilities needed.

Similar approach to expected case, but now assign m^j to $M_{iS_{ij}}$.

Any matching that uses a smaller maximum j must have a smaller total weight:

$$m^j + \text{other costs} > m \bullet m^{j-1}$$

Wide range of magnitudes will be inconvenient in using Hungarian method, but result will not skip over elements of a probe sequence.

Also possible to use binary search on instances of (unweighted) bipartite matching:

```
low = 1
high = m                    // Can also hash all m keys to get a better upper bound
while low ≤ high
        k = (low + high)/2
        Generate instance of bipartite matching that connects each key to k-prefix of that key's
                probe sequence.
        Attempt to find matching with m edges.
        if successful
                high = k - 1
                Save this matching since it may be the final solution.
        else
                low = k + 1
< low is the worst-case number of probes needed>
Iteratively patch solution so that no elements of any probe sequence are skipped over.
```

Generating instance of bipartite matching as maximum flow problem (unit capacity edges):