Operations on Dynamic sets

- Hash functions map items from a very large set to a smaller set (storage location)
- Each item has a key (an integer)
- Randomness is expected in the hash function
- Collision occurs when the mapping for a given key returns an occupied slot
- Collision resolution is done through chaining
- Rehashing can be done to avoid collision till finding an unoccupied location
- Symbol tables in assemblers and compilers are implemented as hash tables

Operations on hash table

- Insert, delete and search
- Problem size depends on
  - Number of slots in the hash table, m
  - Number of items stored in the hash table, n
  - Load factor = α = n/m
- Chaining needs traversal along the chain if slot is occupied => $\alpha > 1$ is possible
- Open addressing needs to free slot for insertion upon deletion of an item => $\alpha \leq 1$

Search time complexity - Chaining

- Search time can be bounded for both success and failure
- In case of failure:
  - Time to hash the key = $O(1)$
  - Time to find the key = $O(\alpha)$
  - Total time = $O(\alpha+1)$
- In case of success:
  - Let the item be i-th one to be inserted
  - Load factor at that time was $(i-1)/m$
  - Chain traversal needed is of length $(i-1)$
  - Expected no of elements searched out of n elements
  - $= (1/n)\ \Sigma\ (1+ (i-1)/m)$ averaged over no of currently stored
  - $= 1+ (1/_{nm})\ (1/2)\ (n(n-1)) = O(1+\alpha)$

Types of hash functions

- Division method $h(k) = k \bmod m$, m is prime
- Multiplication method $h(k)=floor(m(kA \bmod 1))$ – A being {0,1} $A=(\sqrt{5} - 1)/2 = 0.618$
- Collision probability depends on randomness of h
- Linear probing: $h(k,i)=(h'(k)+i) \bmod m$
  – may cause primary clustering
- Quadratic probing: $h(k,i) = (h'(k)+c_1i+c_2i^2) \bmod m$
  – may cause secondary clustering
- Double hashing: $h(k,i) = (h_1(k) + i\, h_2(k)) \bmod m$
- Uniform hashing: unoccupied slots are equally likely to be selected => $h(k,i)=(h_1(k) + i\, h_2(k) + i^2\, h_3(k)+\ldots) \bmod m$

Unsuccessful Search-Open addressing

- P_i = prob{exactly i probes access occupied slots} for i = 0,1,2,..,n. P_i=0 for i > n
- Q_i = prob{at least i probes access occupied slots} assuming uniform distribution, will be

  $= (n/m)(n-1/m-1)\ldots(n-i+1/m-i+1) \leq (n/m)^i \leq \alpha^i$
- Now, $\sum_{i=0} iP(x=i) = \sum_{i=0} i\,(Pr(x \geq i) - Pr(x \geq i+1))$

  $= \sum_{i=1} (Pr(x \geq i) = \sum_{i=1} Q_i$
- Expected no of probes for unsuccessful search=

  $1+\sum iP_i \leq 1+ \sum_{i=1} Q_i \leq \sum_{i=0} \alpha^i \leq 1/(1-\alpha)$
- Expected no of probes = 2 for α=0.5 and =10 for α=0.9

Successful Search-Open addressing

- When element was inserted as (i+1)-th element, load factor was $\alpha = i/m$
- It took $1/_{(1-i/m)}$ no of unsuccessful probes
- Expected no of probes for successful search is sum over all such i when n slots are occupied

  $1/_n \sum 1/_{(1-i/m)} = (m/n) \sum 1/_{(m-i)} = 1/_\alpha \, (H_m - H_{m-n})$
- $H_i = \sum_{j=1}^{j=i} (1/j)$ bounded by $\ln(i) \leq H_i \leq 1+\ln(i)$
- no of probes $\leq 1/_\alpha \, (1+ \ln(m) - \ln(m-n)) = 1/_\alpha + 1/_\alpha \, (\ln 1/_{1-\alpha})$
- Expected no of probes ≤ 3.387, $\alpha=0.5$ and ≤ 3.67, $\alpha=0.9$
- No of probes not increasing rapidly as table fills up.

Data structures for disjoint sets

- S={S_1,S_2,..,S_k) disjoint dynamic sets having representative elements
- MAKE_SET(x) creates new set with only one member pointed to by x and x cannot belong to any other set – disjoint.
- UNION (x,y) unites two dynamic sets S_x and S_y containing x and y into a new set U , with new representative.
- FIND_SET (x) returns pointer to the representative of the set containing x.

Connected Components

```
for each vertex v ε V[G]
    do MAKE_SET(v)
for each edge (u,v) ε E[G]
    if FIND_SET(u) ≠ FIND_SET(v)
            UNION(v,u)
End

SAME_COMPONENT(u,v)
    if FIND_SET(u) == FIND_SET(v)
            then return TRUE
            else return FALSE
End
```

Parameters for analyzing running times of operations

- The no of MAKE_SET operations is n
- total no of operations is m
- Each UNION operation reduces no of sets by one, so that after n-1 such operations, only one set remains.
- Therefore, m cannot exceed n-1 and m ≥ n since MAKE_SET operations are included in m.

- Then q=m-n operations requires incremental no of updations O(q^2) with i-th UNION operation requiring O(i).
- Total time is O(m^2) i.e. average O(m) per operation
- Some heuristics that update representative of smaller list to that of larger list can reduce this complexity.

Linked list representation

- An element x is always on the smaller set if its representative pointer is updated.
- Hence first time, resulting set has at least two members, next time at least 4 and so on.
- For any $k \leq n$ after pointer of x is updated $\lg(k)$ times, resulting set has at least k members.
- Hence a pointer of an object is updated at most $\lg(n)$ times. For n objects, this is $O(n \lg n)$.
- Since there are $O(m)$ MAKE and FIND operations taking $O(1)$ time each, total time for entire operation is $O(m+n \lg n)$

Disjoint set forest

- MAKE-SET creates trees with just one node
- FIND-SET chases parent pointers upto the root
- UNION causes root of one tree to point to other
- Heuristics – union by rank, path compression
- Rank- approximates the logarithm of the subtree size, it is also the upper bound on height of the node
- Root with smaller rank is made to point to the one with larger rank
- Path compression – makes each node on find path point directly to the root – rank not affected by this

Disjoint Set Forest Algorithms

```
MAKE-SET(x)
    p[x]=x
    rank[x]=0

FIND-SET(x)
    if x ≠ p[x]
        p[x] = FIND-SET(p[x])
    return p[x]

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

```
LINK(x,y)

    if rank[x] > rank[y]
        p[y]=x
    else
        p[x]=y

    if rank[x]==rank[y]
        rank[y]++
```

Properties of rank

- P1- follows from definition rank[x] ≤ rank[p[x]] hence, Subtree rooted at p[x] is larger.
- P2- Let size(x) be no of nodes in the tree rooted at x. For all tree roots x, size(x) ≥ $2^{rank[x]}$
- P3- For any integer r ≥ 0, at most $n/2^r$ nodes of rank r exists.
- P4- Every node has a rank at most floor (lg n)

Property on size of tree rooted at x

- This can be proved by induction on no of LINK operations.
- Before first LINK on x, this is TRUE since rank[x]=0.
- Let rank, size before LINK be *rank, size* and after LINK it becomes *rank', size'*.
- In operation LINK(x,y) let rank[x] < rank[y].
- Node y is root of tree formed through LINK and we have size'(y)=size(x)+size(y) $\geq 2^{rank[x]} + 2^{rank[y]}$
- Which gives size'(y) $\geq 2^{rank[y]} \geq 2^{rank'[y]}$ [no rank changes other than y]
- When rank[x] = rank[y], size'(y) $\geq 2.\ 2^{rank[y]} = 2^{rank[y]+1} = 2^{rank'[y]}$
- **Hence by induction, For all tree roots x, size(x) $\geq 2^{rank[x]}$**

Counting nodes within a rank r

- When rank r is assigned to x, attach a label x to all nodes of the tree rooted at x.
- At least 2^r nodes are labeled each time. When root changes for x, rank of root is at least r+1. Hence no new node is labeled x for this.
- Each node is therefore labeled at most once. There being n nodes in all, at most n labeled nodes with at least 2^r labels assigned for each node of rank r.
- If there are more than $n/2^r$ nodes of rank r, then more than $(n/2^r)\, 2^r$ i.e. more than n nodes would be labeled by a node of rank r, which is a contradiction.

Maximum rank possible for a node

- Let $r > \lg n$, then there are at most $n/2^r < 1$ node of rank r.
- But this is impossible as rank is integer.
- **Every node has a rank at most floor (lg n)**

Dividing ranks into rank groups

- Rank 0, 1 → rank group 1; Rank 2,2^2-1→ rank group 2
- Rank 4 to 2^2^2-1 (15) → rank group 3
- Rank 16 to 2^2^2^2-1 (255) → rank group 4
- Rank F(g) to $2^{F(g)}$ - 1 → rank group g
- F(g) =2^2^2…g times … ^2 so that G(n) = $lg^*(n)$ take lg till n reduces to 1.
- This puts rank r into group $lg^*(r)$ for r=0,1,…, floor(lg n)
- Highest group no will be $lg^*(lg\ n) = lg^*(n)$ -1.
- Then j-th group has ranks {F(j-1)+1, F(j-1)+2, …, F(j)}

Time complexity for transitions

- Two cost types: within group and transition to higher rank group.
- **In Transition cost, t**here can be $lg^*(n)$ +1 transitions in all for each FIND-SET operation. Once a node has parent in a different group, it can no longer come back to previous group because of heuristics.
- For m FIND-SET operations, total cost of transitions is thus $m(lg^*(n) +1)$

Cost within group

- No of nodes are given by
  - $N(g) \leq \sum (n/2^r) = (n/2^{F(g-1)+1}) \sum (1/2^r)$
- The sum running from r=0 to F(g) – F(g-1)+1 can be changed to an infinite series sum for large g.
- So, $N(g) < n / F(g)$
- For g=0, $N(0) = 3n / 2F(0)$ Hence $N(g) \leq 3n / 2F(g)$ for all $g \geq 0$
- Hence considering all rank groups, denoting by P(n) the total cost within groups, can be obtained

Cost within group

- Multiplying no-of-nodes by no-of-ranks and summing from g=0 to $\lg^*(n)$-1
- P(n) ≤ ∑ [3n / 2 F(g)] [F(g) – F(g-1) – 1] ≤ ∑ (3n/2) since F(g) >> F(g-1) for large g.
- Hence P(n) ≤ n $\lg^*(n)$ so that
  - T(n) = m ($\lg^*(n)$ + 1)

Total time complexity

- Total cost is therefore (since $m \geq n$)
  $O(m\ (\lg^*(n) + 1) + n \lg^*(n)) = O(m\ (\lg^* n)$
- There are O(n) MAKE-SET and LINK or UNION operations with O(1) time each.
- Total time complexity stays at $O(m\ (\lg^* n))$
- Time per operation is therefore $O\ (\lg^* n)$ – amortized complexity

MST Lemma

- $G = (V, E)$ be weighted connected graph
- U is a strict subset of V i.e. nodes in G
- T *is* a promising subset of edges in E such that no edge in T leaves U
- e is a least cost edge that leaves U
- Then the set of edges $T' = T \cup \{e\}$ is promising.

MST - Kruskal's Algorithm

- J.B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem *Proceedings of the American Mathematical Society*, Volume 7, pp. 48-50, 1956.
- Complexity is $O(e\log e)$ where e is the number of edges. Can be made even more efficient by a proper choice of data structures.
- At the end of the algorithm, we will be left with a single component that comprises all the vertices and this component will be an MST for G.

MST - Kruskal

```
Let G = (V, E) be the given graph, with | V| = n
{
    Start with a graph T = (V,) consisting of only the
    vertices of G and no edges;
/* This can be viewed as n connected components, each vertex being one
   connected component */
   Arrange E in the order of increasing costs;  → GREEDY
   for (i = 1, in - 1, i + +)  → DISJOINT SETS
   {
       Select the next smallest cost edge;
       if (the edge connects two different connected components)
           add the edge to T;
   }
}
```

Kruskal – matroids and disjoint sets

```
MST-Kruskal (G,w)
    A ← Φ
    FOR each vertex v in V[G]
       DO MAKE-SET(v)
    Sort the edges of E in order of non-decreasing w
    FOR each edge (u,v) in E in sorted order DO
       IF FIND-SET(U) ≠ FIND-SET(v) THEN
                A = A U {(u,v)}
                UNION (u,v)
    RETURN A
```

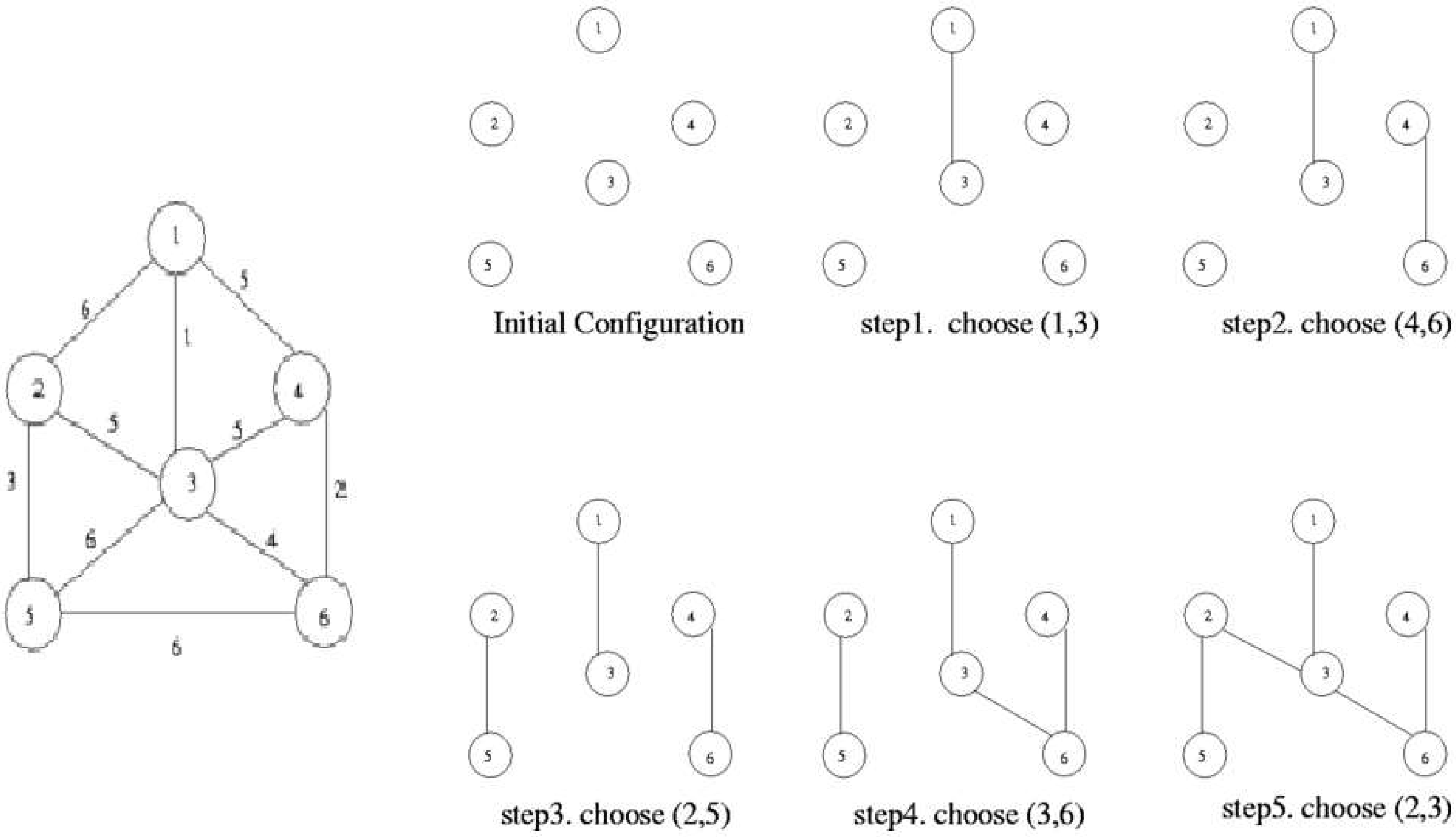
Theorem: Kruskal algorithm finds MST

- **Proof:** Let G = (V, E) be a weighted, connected graph. Let T be the edge set that is grown in Kruskal's algorithm. The proof is by mathematical induction on the number of edges in T.
  - We show that if T is promising at any stage of the algorithm, then it is still promising when a new edge is added to it in Kruskal's algorithm
  - When the algorithm terminates, it will happen that T gives a solution to the problem and hence an MST.
- **Basis:** $T = \Phi$ is promising since a weighted connected graph always has at least one MST.
- **Induction Step:** Let T be promising just before adding a new edge $e = (u, v)$. The edges T divide the nodes of G into one or more connected components. u and v will be in different components. Let U be the set of nodes in the component that includes u.

Theorem: Kruskal algorithm finds MST

- Note that
  - U is a strict subset of V
  - T is a promising set of edges such that no edge in T leaves U (since an edge T either has both ends in U or has neither end in U)
  - e is a least cost edge that leaves U (since Kruskal's algorithm, being greedy, would have chosen e only after examining edges shorter than e)
- The above three conditions are precisely like in the MST Lemma and hence we can conclude that the T $\{e\}$ is also promising. When the algorithm stops, T gives not merely a spanning tree but a minimal spanning tree since it is promising.

Kruskal - illustration



Running Time of Kruskal's Algorithm

- The total time for performing all the merge and find depends on the method used.
- $O(e \log e)$ without path compression
- $O(e(\lg^* e))$ with the path compression

Prim’s algorithm - MST

R.C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, Volume 36, pp. 1389-1401, 1957.

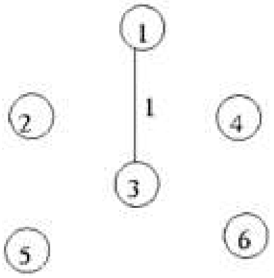
```
{       T = Φ;
        U = { 1 };
        while (U≠V)
        {
           let (u, v) be the lowest cost edge
           such that u ε U and v ε V - U;
            T = T ∪ {(u, v)}
            U = U  ∪ {v}
        }
}
```

- At each step, we can scan lowcost to find the vertex in *V* - *U* that is closest to *U*.
- Then we update lowcost and closest taking into account the new addition to *U*.
- Complexity: $O(n^2)$

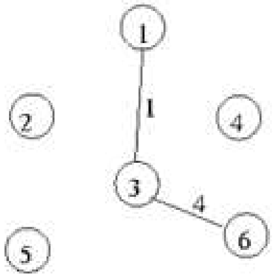
Proof of Correctness-Prim's Algorithm

- Let G = (V,E) be a weighted, connected graph. Let T be the edge set that is grown in Prim's algorithm. The proof is by mathematical induction on the number of edges in T and using the MST Lemma.
- **Basis**: The empty set is promising since a connected, weighted graph always has at least one MST.
- **Induction Step:** Assume that T is promising just before the algorithm adds a new edge e = (u,v). Let U be the set of nodes grown in Prim's algorithm. Then all three conditions in the MST Lemma are satisfied and therefore T U e is also promising.
- When the algorithm stops, U includes all vertices of the graph and hence T is a spanning tree. Since T is also promising, it will be a MST.

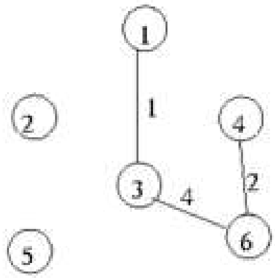
Prim’s algorithm - illustration



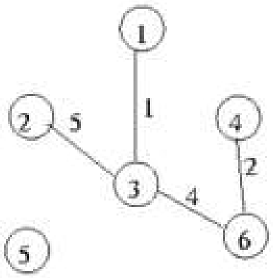
Iteration 1. U = {1}



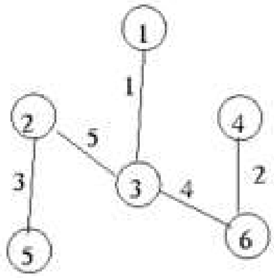
Iteration 2. U = {1,3}



Iteration 3. U = {1,3,6}



Iteration 4. U = {1,3,6,4}



Iteration 5. U = {1,3,6,4,2}

Difference between Prim and Kruskal

Prim's algorithm

- Initialize with a node
- Graph has to be connected
- Always keep a connected component, look at all edges from the current component to other vertices and find the smallest among them - then add the neighbouring vertex to the component, increasing size by 1.

Kruskal's algorithm

- Initialize with an edge
- Work on disconnected graph
- do not keep one connected component but a forest. At each stage, look at the globally smallest edge that does not create a cycle in the current forest. Such an edge has to necessarily merge two trees in the current forest into one.

Difference between Prim and Kruskal

Prim's algorithm

- In N-1 steps, every vertex would be merged to the current one if we have a connected graph.
- Next edge shall be the cheapest edge in the current vertex.
- Prim's algorithm is found to run faster in dense graphs with more number of edges than vertices

Kruskal's algorithm

- Since you start with N single-vertex trees, in N-1 steps, they would all merge into one if the graph was connected.
- Choose the cheapest edge, but it may not be in the current vertex.
- Kruskal's algorithm is found to run faster in sparse graphs.