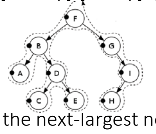
Runway Reservation System
- Single runway
- Reservations for landings: maintain a set of future landing times (R), a new request to land at time t, add t to R if no other landings are scheduled within < 3 mins from t, when a plane lands, remove its reservation from the set

Binary Search Trees
- Each node x has key[x]
- Pointers: left[x], right[x], parent[x]
- Property: Key[left[x]] < key[x] & key[x] < key[right[x]]
- Tree Traversals: O(n)

Preorder: FBADCEGIH
Inorder: ABCDEFHI
Postorder: ACEDBHIGF



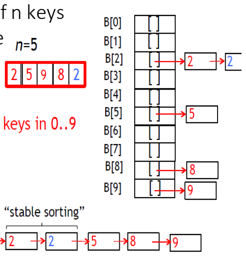
When you del a root of a BST, the next-largest node replaces that root.
Next-larger(x): finds the next element after element x

```
if right[x] ≠ NIL then
   return findmin(right[x])
else
   y = parent[x]
   while y≠NIL and x==right[y]
   x = y
   y = parent[y]
   return y
```

insert(k) (with <3mins check)
find(k) (finds node containing k)
findmin(x) (finds min of tree rooted at x)
deletemin() (finds min of tree and delete it)
All are worst case: O(n)

Comparison sorts: Ω(nlogn)
A decision tree can model the execution of ANY comparison sort
One tree for each input size n, a path from root to the leaves is the trace of comparisons the algo perform.
Worst-case running time = height of tree
Proof: Tree must contain >= n! leaves => n! possible answers
A height-h binary tree has <= 2^h leaves
Thus $n! \leq L_n \leq 2^{h_n} \leftrightarrow \log(n!) \leq h_n \rightarrow h_n = \Omega(nlog\ n)$

Counting Sort
Given array A[0],...,A[n-1] of n keys to be sorted. The n keys are integers in {0,1,...k-1}
B = array of k empty lists (linked/python lists) – O(k)



for j in range(n): – O(n)
B[A[j]].append(A[j])
output = []
for i in range(k): – O(n+k)
output.extend(B[i])
Total time: Θ(n+k) = Θ(n) if k=O(n)

Radix Sort
- Sort on least significant digit/alphabet first
- Given n integers, each int <= M, in base k

Each int has d = $log_k(M)$ digits, a digit is in {0,1,...,k-1}
- Assume counting sort is *auxiliary stable* sort

Counting sort: we need
$\theta(n+k)$ per digit $\Rightarrow \theta((n+k)d) \Rightarrow \theta((n+k)log_k M)$
$\theta(nlog_n M)$ if k=n $\Rightarrow \theta(nc)$ if $M \leq n^c$ *for some* $c > 0$
- if range M of possible values grows at most proportionally with size of problem (n), use counting sort
- if M grows even faster but $O(n^c)$ for some c>1, use radix sort (choose optimal base)

Hash Tables:
Data structure that supports 'dictionary operations'
Insert O(1) | Search O(n) worst, O(1+α) avg | Del O(1) *if have pointer to obj
Initialise array w null, insert(x) -> A[x.key]=x, delete(x) -> A[x.key]=null, search(x) -> A[x.key] != null return T/F. All O(1) operations
Not practicable at times: not possible to store obj in array if set of obj very large, dk the keys in advance or keys are not int (eg. names)
Solution: Hash function (maps all possible keys K onto {0,...,l-1}

Collisions: two keys are hashed onto the same value: h(key1)=h(key2)
Hash table is an array of size m, each element in the table is a linked list (chaining)
Worst case search: O(n) one entry has all n elements, essentially searching a linked list
Average case: every entry have α=n/m objects(search list, apply hash function & random access to slot)
O(1+α) = O(1) if α = O(1) ie m= Ω(n)
Chained-hash-insert(A,x): insert x at head of linked list A - O(1) compute h(x.key) + 1 insert
Chained-hash-delete(A,x): assume we have pointer to x. Go to x in A[h(x.key)] using address, link predecessor and successor of x in linked list, delete x from list A[h(x.key)] using pointer - O(1)
Hashing for non-numerical input: go by some radix
Simple uniform hashing assumption: any given element is equally likely to hash into any of the m slots, indep of where any other element has hashed to.

Hash functions:
- Division method: h(k) = k mod m

practical when m is prime but not close to power of 2 or 10 (then just depending on low bits/digits)
- Multiplication method:

h(k) = floor(m (kA mod 1)) where kA mod 1 = kA - floor(kA) = fractional part of kA
Constant A where 0<A<1.
Pro: value of m is not critical (can be 2^p)
Con: slower than division method

Resizing a Hash Table:
Complexity: O(n+m+m') m for visiting every table entry in old table, n for processing every object, m' for making new table
Resize by 1: m'=m+1 => rebuild table every 1 insert: cost= θ (1+2+...+n)= O(n^2)
Resize by adding a constant: m' = m+c whenever n/m hits a threshold t overall O(n^2)
Resize by doubling: m'=2m whenever n/m hits t
Cost of n inserts: θ(1+2+4+...+n) = θ(n)
Overall resize complexity = **O(Nfinal)** *works for x4/x8 also m'=m*r where r>1
Amortized cost over Nfinal insertions= **O(1)** for search (if SUH holds)
If n>m, we can find ≥ N/l keys that collide for any hash function

Open Addressing: one array entry = one element but can only fill table with at most n=m elements before table doubling
Linear Probing: $h(k,i) = (h'(k) + i)\ mod\ m$
Quadratic Probing:
$$h(k,i) = (h'(k) + c_1 i + c_2 i^2)\ mod\ m$$
Double Hashing:
$$h(k,i) = \left(h_1(k) + ih_2(k)\right)\ mod\ m$$
Use the first mod then compute the second mod if m is different
EG h'(k) = k % n
(h'(k) + ci + ki^2) % m
(here u needa compute the first mod first!!)

AVL: Augment every node in BST with its height
- Invariant: for each node x, heights of its left child & right child differ by at most 1

(maximal height diff between any 2 leaves is O(logn), every pair of child subtrees differ by 1)
Let n_h be min num of nodes of an AVL tree of height h
$$n_h \geq 1 + n_{h-1} + n_{h-2} > 2n_{h-2} > 2 \cdot 2n_{h-4} > \cdots > 2^{h/2}$$
$$\Rightarrow h < 2\log(n_h) \leq 2\log(n) | \ h = O(\log(n))$$
Let mh be max num of nodes, $m_h \leq 2^h$
$\log(m_h) \leq h\ ; h = \Omega(\log n)$ => h= Θ(log n)
- All operations take O(log n), can maintain balanced BST using O(log n) time per insertion

Uniform Hashing Assumption: *Probe sequence* of each key is equally likely to be any of m! permutations of {0,...,m-1}.
P(first probe finding empty slot)= 1-α
Search: 1/(1−α) probe steps on average for unsuccessful search | 1/α log (1/(1−α)) probe steps on average for successful search
Insertion: 1/(1−α) probe steps on avg
if α is small (much < 1), 1/(1-α) is close to 1.
α should be < 1, use doubling before n = m.

Open Addressing vs Chaining:
- OA: Better cache perf, no pointers to off regions needed when objects are "small" eg int/float
- Chaining: Less sensitive to hash function choices, similar results even when you choose diff hash fn. Less sensitive to high load factors, OA needs α to be small.

Cuckoo Hashing:
- use 2 hash functions, key stored in either T[h1(k)] or T[h2(k)]. Just need to look at at most 2 places – O(1).

Insertion: if T[h1(k)] empty, store. else if T[h2(k)] empty, store. Else, store in T[h1(k)] and move key that was there to its other location.
If α<1, insertion succeeds with high prob.
If Insertion loops: rehash entire table/ x2 table size.
Search is O(1) worst case, Insertion is O(n) worst case, O(1) avg

Representation of Graph:
1. Adjacency lists (of neighbours of each vertex) Array A of |V| linked list, for each v in vertices, A[v] stores neighbours
Directed graph only stores outgoing neigh, undirected stores edge in 2 places
2. Implicit Representation (as neighbour fn)
Don't store graph at all, implement Adj(u) that returns list of neigh/edges of u, requires no space
3. Incidence lists (of edges from each vertex)
For each vertex v, list its edges: Array A of |V| linked list, for each v in vertices, A[v] store edges. Directed graph only stores outgoing edges, undirected store edge in 2 places
4. Adjacency Matrix (of which pairs are adj)
nxn matrix A = (aij); aij =1 if (i,j) is edge

Space: Adj list: one list node per edge: space is θ(n+m) | Adj matrix: space θ(n^2), better only for v dense graph where m is near n^2 (m can be n(n-1))
Time: Add edge: both O(1)|Check edge: mat O(1), Adj list O(n)|Visit all neigh of u: adj list O(neigh), mat θ(n)|Remove edge: find+add

BFS (Array+Adj list):

```
BFS(s,Adj):
 level={s:0}
 parent={s:None}
 i=1
 frontier=[s]
 while frontier:
  next=[]
  for u in frontier:
   for v in Adj[u]:
    if v not in level:
     level[v]=i
     parent[v]=u
     next.append(v)
  frontier=next
  i+=1
```

Replace queue by stack -> DFS

BFS (Queue+Adj list):

```
BFS(G,s):
 for each vertex:
   u.color=WHITE
   u.d=∞ ; u.parent=NIL
 s.color=GRAY
 s.d=0
 s.parent=NIL
 Q=Ø
 ENQUEUE(Q,s)
 while Q != Ø:
   u=DEQUEUE(Q)
   for each v in Adj[u]:
     v.color=GRAY; v.d=u.d+1
     v.parent=u
     ENQUEUE(Q,v)
   u.color=BLACK
```

Shortest paths from s to v:
Length=level[v]; is ∞ (if not reachable from s)
follow v->parent[v]->...->s

Depth-First Search (depth up to V-1, complete graph)
- No 2 vertices will have same start/end time
- Type of edge, u -> v: (no forward in undir)

back: v.d < u.d < u.f < v.f   tree: first time v is visited
forward: u.d < v.d < v.f < u.f
cross: v.d < v.f < u.d < u.f
Graph has a cycle iff DFS has a back edge

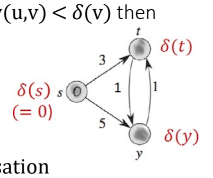
```
DFS(G):
  for each vertex u:
    u.color=WHITE; u.parent=NIL
  time=0
  for each vertex u
    if u.colour==WHITE
      DFS-VISIT(G,u)
```

```
DFS-VISIT(G,u):
  time+=1
  u.d=time // u was white
  u.color=GRAY
  for v in Adj[u]: //explore edge
    if v.color==WHITE:
      v.parent=u
      DFS-VISIT(G,v)
  u.colour=BLACK //finished
  time+=1
```

Θ(n+m)
For both directed&undirected

Topological Sort: (SSSD For DAG)
1. Run DFS(G), compute finishing times of nodes θ(n+m)
2. Output nodes in decreasing order of finishing times (can get to θ(n))
after u.color=black, store finishtime[time]=u.
then for (j=2n;j>0;j--) {if (finishtime[j] not empty) output finishtime[j]}

Single Source Shortest Path (hav weights, diff from BFS)
Shortest path not necessarily unique but value function is unique
Subpaths of shortest paths are also shortest paths
$\delta(Vo,Vj) + \delta(Vj,Vn) = \delta(Vo,Vn)$
Triangle inequality: $\delta(s,t) <= \delta(s,u) + \delta(u,t)$
Negative-weight cycle: shortest path cannot be defined
Positive-weight cycle: never occurs in any shortest path
Relax an edge(u,v): if $\delta(u)+w(u,v) < \delta(v)$ then
$\delta(v):= \delta(u)+w(u,v)$; v.π:=u
Bellman Eqns: $\delta(s)=0$;
$\delta(t)=\min\{\delta(s)+3,\delta(y)+1\}$;
$\delta(y)=\min\{\delta(s)+5,\delta(t)+1)\}$



Bellman-Ford
for v in V: | initialisation θ(|V|)
  v.d=∞; v.π=nil
s.d=0
do n-1 times: | θ(|V||E|)
  for each edge(u,v) in E:
    relax(u,v)
for each edge(u,v) in E: | detect neg cyc θ(|E|)
  if v.d > u.d + w(u,v)
    negative cycle
no neg cycle

Overall complexity=θ(|V||E|)
Dense graphs: $|E| \sim |V|^2$
Sparse graphs: |E| ~ |V|

★ if no neg cyc, at termination for all v: v.d=$\delta(s,v)$
★ Can be exponentially many relaxations hence order matters

Dijkstra's Algo
Applies to graphs with only NONNEGATIVE WEIGHTS
Uses relaxation to improve estimated dist of non-established vertices via newly promoted vertex; terminates when all vertices have established distances
Overall complexity=$\theta(|V|^2)$
for v in V: | initialisation θ(|V|)
  v.d=∞; v.π=nil
s.d=0; S=∅
while S≠V: | $|V|(|V|+|V|)$ / $O(|V|^2 + |E|)$ - while loop thru all v - calculate min + loop thru all neigh (max. no of neigh for v = |V| - 1)
  u=v with v.d≤w.d for all w not in S
  S=S∪{u}
  for each edge(u,v) in E, v not in S:
    relax(u,v)
If priority queues used for storing & retrieving v.d, O(|V|log(|V|)+|E|)

Properties of hash functions:
- One-way: given a hash z, can't find x that created this hash: h(x)=z
- Collision-resistance: infeasible to find x & x' st h(x)=h(x') (dk x,x')
- Target-collision-resistance: given some x, infeasible to find x' st h(x')=h(x) (know x, dk x')
- Hash maps 2 close keys x,x' to very different locations

Universal Hashing:
- choose a function h at random from a function class H={h1,...,hr}
- H is universal collection of hash fn if for any fixed pair of keys k1≠k2, chance of collision between k1 & k2 is 1/m is we choose h randomly from H.
- For h drawn randomly from uniform distribution over a universal collection of hash functions, we have O(1+α) insertion time.

Hashing n items to hash table of size k:
- E(num items hashed to any one location) =
- P(pos i will be empty when you hash 1 item) =
- P(2 pos will be mepty when you hash 1 item) =
- E(num empty locations) =
- E(num of collisions) = n - E(occupied) = n - (k - E(empty locations))

★ Let P be shortest path from vertex s to t. If weight of each edge in graph increased by one, P will still be shortest path from s to t. False. Let w(s,v1),w(v1,v2),w(v2,t)=1 and w(s,t)=4 | if weight x2, P is still shortest (linear transformation)
★ Changing RELAX function to update if d[v]≥d[u]+w(u,v) will change correctness of Bellman-Ford outputs d & π. Parent pointers may not lead back to source node if a zero-length cycle exists.
★ If weighted directed graph G is known to have no shortest paths longer than k edges, then it suffices to run BF for only k passes. True. ith iteration finds shortest path in G of i or fewer edges by path relaxation property.
★ If priority queue in Dijkstra is implemented using sorted linked list, running time θ(V(V+E)lgV). Array take θ(V) to insert a node, V nodes to insert, θ(E) calls to decrease key can take θ(V) time.
★ Implement Dijkstra using priority queue, requires O(V) to initialise, worst-case f(V,E) for each extractmin op & worst-case g(V,E) time for each decreasekey op. Whats worst-case to run Dijkstra? O(V.f(V,E) + E.g(V,E))
★ For Dijkstra: non-negative weights. To transform w∈(0,1], let w'=-log(w)
★ Given a directed graph with exactly one neg weight edge & no negweight cycles. Give an algo to find shortest dist from s to all other vertices that has same running time as Dijkstra. Let neg weight edge be (u,v). Remove the edge and run Dijkstra from s. check if δ(s,u)+w(u,v)<δ(s,v). If not, we are done. If yes, run Dijkstra from v, for any node t, its shortest dist from s will be min{δ(s,t),δ(s,u)+w(u,v)+δ(v,t)}
★ Dijkstra can be implemented using AVL tree instead of heap for priority queue and still run in O((V+E)lgV) time. True, both decreasekey and extract min for AVL can be done in O(lgV) time.
★ In Bellman-Ford, if we have 2 consec edges on a shortest path: (u,v) & (v,w). Edge (v,w) is guaranteed to be relaxed within O(1) relaxations of (u,v) being relaxed. False, BF runs repeatedly through all edges in some arbitrary order, so 2 edges may be visited θ(E) steps from each other.
★ Multi-source shortest path distance: Add dummy vertex s*, add zero-weight edges from s* to all s∈S. Run Dijkstra from s*.
★ Performing a left rotation and then a right rotation on same node will not change the tree structure. False. To undo left rotation on x, must do a right rotation on y (initial right child of x)

Double Hashing: Two numbers are relatively prime if no number larger than 1 can divide both numbers. Eg 12 &13. Value h2(k) must be relatively prime to m for entire hash table to be searched. Convenient way is for m to be power of 2, design h2 to be always odd.
- If a key is deleted, sometimes we may not be able to find a key after it, mark deleted keys with "del" instead of "nil".
- A node in a BST has rank k if precisely k other keys in the BST are smaller. So, if you order all the BST nodes according to their keys, then each node with rank k will take k-th place.
- DFS store as much memory on stack as is required for longest root to leaf path in tree. O(h). BFS queue every node at fixed depth before visiting next depth. Θ(n)
- find x' st h(x)=h(x')
- Dowan person to reverse hash to find out x. Hash x+c where c is a large random number to remember.
- dk x, try to find x' st h(x')=h(x)
a b c d e f g h i j k l m n o p q r s t u v w x y z

★Maximum height of any AVL tree with 8 nodes is 3. (Assume that height of a tree with single node is 0). Minimum no. of nodes = N(h) = N(h-1) + N(h-2) + 1; N(3)=N(2)+N(1)+1= 4+2+1=7
★ Any binary search tree can be brought into AVL balance by performing a sequence of rotations. True. O(n) rotations suffice if start at leaves and work bottom-to-top
★ Given an array of n numbers (keys) in sorted order, an AVL tree on those keys can be built in O(n) time. True. Using divide-and-conquer, repeatedly find middle element k, create node N containing k as key, perform recursion on remaining array elements to create left and right subtrees of N. AVL pointers can be added in a bottom-up fashion in O(n) time.
★If you know the numbers stored in a BST, you know structure of the tree, you can determine the value stored in each node. True. You can do an in-order walk of the tree, ordering the nodes from smallest to largest key. You can then match them with the values
★ While inserting an element into a BST, we will pass the element's predecessor and sucessor (if they exist). True. Predecessor of a node is either max element of left subtree or one of its ancestors. A newly inserted node has no descendants; hence predecessor must be one of its ancestor traversed during insertion procedure.
★ Merge 2 (unbalanced) BST with n elements into a balanced BST. Do in-order walk of both trees concurrently, at each step, compare 2 tree elements and add smaller one into list L, before calling that element's successor. When finish walking both trees, L contain sorted list of elements from both trees, taking O(n+n)=O(n) time. From sorted list, set root as middle element of list, let first half be left subtree, and do this recursively. Takes O(n+n)=O(n) time. Overall=O(n)
★ Inserting into an AVL tree only need at most 2 rotations to fix imbalance, O(1) rotations
★ Both collision resolution by chaining & OA may result in worst-case search time for one of n keys being θ(n). True, all keys could hash to same bucket/same probe sequence.
★ Hashing is generally used for its average case behavior, not its worst-case behavior.
★ A non-uniform hash function is expected to produce worse performance for a hash table than a uniform hash function because it is more likely to result in collisions, leading to slower lookup times
★ Good values for hash fns: relatively prime to size of table, not a power of the other.
★ If you store each chain using AVL tree instead of linked list & α=1, expected running time of search is O(1+lgα)=O(1). Worst case, all operations cost O(1+lgn)
★ With n^2 keys & hash table of size n, greatest number of distinct keys the table can hold: chaining-n^2, probing-n
★ Running time of radix sort is effectively indpt of whether input is alr sorted. True
★ Radix sort works in O(n) time for all inputs when values are int ranging from 1 to n^2. Counting sort expects 1 to n^2 values => O(n^2)
★ We can sort 7 numbers with 10 comparisons. False. To sort 7 num, binary tree must have 7! leaves. No. of leaves of a complete binary tree of height 10 is 2^10 which is not enough
★ A set of n integers whose values are in range [0,n^8) can be sorted in O(n). True, use radix sort with radix size n. Each invocation of counting sort takes O(n+n)=O(n). Each element has 8 "digits", total time for radix sort is O(8n)=O(n).
★ DFS forest may contain different no. of trees & edges depending on starting vertex and order vertices are searched. EG. a->b if start at a, (a,b) is tree edge. If start at b, a&b are separate trees & (a,b) becomes cross edge.
★ If a DFS of a graph contains at least one back edge, any other DFS on same graph will also contain at least one back edge. True, backedge iff graph contains cycle
★ DFS on directed graph, remove all back edges, resulting graph is acyclic. True, removing any back edges doesn't change traversal order of DFS.
★ Any DFS forest of an undir graph contains same num of trees. True, each connected component will be a single tree in a DFS.
★ Analyse a DAG using DFS, choosing search order st all edges in analysis will be cross edges. > Search nodes in reverse order by topological sort, guarantees that every edge traversed will point to a node that is already finished.
★ Every DAG has only one topological ordering of its vertices. False, {(1,2),(1,3)} can be sorted either [1,2,3] or [1,3,2]
★ You can have multiple back edges but possible to remove 1 edge that destorys all cycles. EG. {(1,2),(2,1),(2,3),(3,1)}, remove (1,2) disrupts both cycles.
★ Give an O(V+E) algo to remove all cycles in a directed graph. Do a DFS, as you traverse, check the edge goes to a node that has been seen but not finished, if so, store in a set. After the DFS, remove all edges in this set.
★ If a DFS on a connected undir graph produces n back edges then a BFS on same graph produce n cross edges. True, BFS on undir graph has no forward/back edges, while DFS on undir graph has no forward/cross edges.
★ The depth of any DFS tree rooted at a vertex is at least as much as depth of any BFS tree rooted at same vertex. True, since BFS finds paths using fewest num of edges.
★ There is no edge in undir graph that jumps more than 1 level of any BFS tree. True, if such an edge existed, it would provide a shorter path to some node than the path found by BFS.
★ BFS takes O(V+E) irrespective of whether presented with adj list or adj matrix. False, with adj matrix, visiting each vertex takes O(V) as we must check all N possible outgoing edges in matrix. Thus, BFS take O(V^2) time using matrix. | DFS using matrix also O(V^2)

```
DFS(G):
  for each vertex u:
    u.color=WHITE; u.parent=NIL
  time=0
  for each vertex u
    if u.colour==WHITE
      DFS-VISIT(G,u)
```

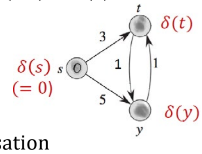
```
DFS-VISIT(G,u):
  time+=1
  u.d=time // u was white
  u.color=GRAY
  for v in Adj[u]: //explore edge
    if v.color==WHITE:
      v.parent=u
      DFS-VISIT(G,v)
  u.colour=BLACK //finished
  time+=1
```

Θ(n+m)
For both directed&undirected

Topological Sort: (SSSD For DAG)
1. Run DFS(G), compute finishing times of nodes θ(n+m)
2. Output nodes in decreasing order of finishing times (can get to θ(n))

after u.color=black, store finishtime[time]=u.
then for (j=2n;j>0;j--) {if (finishtime[j] not empty) output finishtime[j]}

Single Source Shortest Path (hav weights, diff from BFS)
Shortest path not necessarily unique but value function is unique
Subpaths of shortest paths are also shortest paths
$\delta(Vo,Vj) + \delta(Vj,Vn) = \delta(Vo,Vn)$
Triangle inequality: $\delta(s,t) <= \delta(s,u) + \delta(u,t)$
Negative-weight cycle: shortest path cannot be defined
Positive-weight cycle: never occurs in any shortest path
Relax an edge(u,v): if $\delta(u)+w(u,v) < \delta(v)$ then $\delta(v):= \delta(u)+w(u,v)$; v.π:=u
Bellman Eqns: $\delta(s)=0$;
$\delta(t)=\min\{\delta(s)+3,\delta(y)+1\}$;
$\delta(y)=\min\{\delta(s)+5,\delta(t)+1\}$



Bellman-Ford

```
for v in V:                    | initialisation
  v.d=∞; v.π=nil               | θ(|V|)
s.d=0
do n-1 times:
  for each edge(u,v) in E:     | θ(|V||E|)
    relax(u,v)
for each edge(u,v) in E:       | detect neg cyc
  if v.d > u.d + w(u,v)        | θ(|E|)
    negative cycle
no neg cycle
```

Overall complexity=θ(|V||E|)
Dense graphs: $|E| \sim |V|^2$
Sparse graphs: |E| ~ |V|

★ if no neg cyc, at termination for all v: v.d=δ(s,v)
★ Can be exponentially many relaxations hence order matters

Dijkstra's Algo
Applies to graphs with only NONNEGATIVE WEIGHTS
Uses relaxation to improve estimated dist of non-established vertices via newly promoted vertex; terminates when all vertices have established distances

Overall complexity=$\theta(|V|^2)$

```
for v in V:                                  | initialisation
  v.d=∞; v.π=nil                             | θ(|V|)
s.d=0; S=∅
while S≠V:                                   | |V|(|V|+|V|) / O(|V|^2 + |E|)
  u=v with v.d≤w.d for all w not in S        | - while loop thru all v
  S=S∪{u}                                    | - calculate min + loop
  for each edge(u,v) in E, v not in S:       | thru all neigh
    relax(u,v)                               | (max. no of neigh for v = |V| - 1)
```

If priority queues used for storing & retrieving v.d, O(|V|log(|V|)+|E|)

Properties of hash functions:
- One-way: given a hash z, can't find x that created this hash: h(x)=z
- Collision-resistance: infeasible to find x & x' st h(x)=h(x') (dk x,x')
- Target-collision-resistance: given some x, infeasible to find x' st h(x')=h(x) (know x, dk x')
- Hash maps 2 close keys x,x' to very different locations

Universal Hashing:
- choose a function h at random from a function class H={h1,...,hr}
- H is universal collection of hash fn if for any fixed pair of keys k1≠k2, chance of collision between k1 & k2 is 1/m is we choose h randomly from H.
- For h drawn randomly from uniform distribution over a universal collection of hash functions, we have O(1+α) insertion time.

Hashing n items to hash table of size k:
- E(num items hashed to any one location) =
- P(pos i will be empty when you hash 1 item) =
- P(2 pos will be mepty when you hash 1 item) =
- E(num empty locations) =
- E(num of collisions) = n - E(occupied) = n - (k - E(empty locations))

Double Hashing: Two numbers are relatively prime if no number larger than 1 can divide both numbers. Eg 12 &13. Value h2(k) must be relatively prime to m for entire hash table to be searched. Convenient way is for m to be power of 2, design h2 to be always odd.
- If a key is deleted, sometimes we may not be able to find a key after it, mark deleted keys with "del" instead of "nil".
- A node in a BST has rank k if precisely k other keys in the BST are smaller. So, if you order all the BST nodes according to their keys, then each node with rank k will take k-th place.
- DFS store as much memory on stack as is required for longest root to leaf path in tree. O(h). BFS queue every node at fixed depth before visiting next depth. Θ(n)
- find x' st h(x)=h(x')
- Dowan person to reverse hash to find out x. Hash x+c where c is a large random number to remember.
- dk x, try to find x' st h(x')=h(x)

a b c d e f g h i j k l m n o p q r s t u v w x y z

★Maximum height of any AVL tree with 8 nodes is 3. (Assume that height of a tree with single node is 0). Minimum no. of nodes = N(h) = N(h-1) + N(h-2) + 1; N(3)=N(2)+N(1)+1= 4+2+1=7
★ Any binary search tree can be brought into AVL balance by performing a sequence of rotations. True. O(n) rotations suffice if start at leaves and work bottom-to-top
★ Given an array of n numbers (keys) in sorted order, an AVL tree on those keys can be built in O(n) time. True. Using divide-and-conquer, repeatedly find middle element k, create node N containing k as key, perform recursion on remaining array elements to create left and right subtrees of N. AVL pointers can be added in a bottom-up fashion in O(n) time.
★If you know the numbers stored in a BST, you know structure of the tree, you can determine the value stored in each node. True. You can do an in-order walk of the tree, ordering the nodes from smallest to largest key. You can then match them with the values
★ While inserting an element into a BST, we will pass the element's predecessor and sucessor (if they exist). True. Predecessor of a node is either max element of left subtree or one of its ancestors. A newly inserted node has no descendants; hence predecessor must be one of its ancestor traversed during insertion procedure.
★ Merge 2 (unbalanced) BST with n elements into a balanced BST. Do in-order walk of both trees concurrently, at each step, compare 2 tree elements and add smaller one into list L, before calling that element's successor. When finish walking both trees, L contain sorted list of elements from both trees, taking O(n+n)=O(n) time. From sorted list, set root as middle element of list, let first half be left subtree, and do this recursively. Takes O(n+n)=O(n) time. Overall=O(n)
★ Inserting into an AVL tree only need at most 2 rotations to fix imbalance, O(1) rotations
★ Both collision resolution by chaining & OA may result in worst-case search time for one of n keys being θ(n). True, all keys could hash to same bucket/same probe sequence.
★ Hashing is generally used for its average case behavior, not its worst-case behavior.
★ A non-uniform hash function is expected to produce worse performance for a hash table than a uniform hash function because it is more likely to result in collisions, leading to slower lookup times
★ Good values for hash fns: relatively prime to size of table, not a power of the other.
★ If you store each chain using AVL tree instead of linked list & α=1, expected running time of search is O(1+lgα)=O(1). Worst case, all operations cost O(1+lgn)
★ With n^2 keys & hash table of size n, greatest number of distinct keys the table can hold: chaining-n^2, probing-n
★ Running time of radix sort is effectively indpt of whether input is alr sorted. True
★ Radix sort works in O(n) time for all inputs when values are int ranging from 1 to n^2. Counting sort expects 1 to n^2 values => O(n^2)
★ We can sort 7 numbers with 10 comparisons. False. To sort 7 num, binary tree must have 7! leaves. No. of leaves of a complete binary tree of height 10 is 2^10 which is not enough
★ A set of n integers whose values are in range [0,n^8) can be sorted in O(n). True, use radix sort with radix size n. Each invocation of counting sort takes O(n+n)=O(n). Each element has 8 "digits", total time for radix sort is O(8n)=O(n).
★ DFS forest may contain different no. of trees & edges depending on starting vertex and order vertices are searched. EG. a->b if start at a, (a,b) is tree edge. If start at b, a&b are separate trees & (a,b) becomes cross edge.
★ If a DFS of a graph contains at least one back edge, any other DFS on same graph will also contain at least one back edge. True, backedge iff graph contains cycle
★ DFS on directed graph, remove all back edges, resulting graph is acyclic. True, removing any back edges doesn't change traversal order of DFS.
★ Any DFS forest of an undir graph contains same num of trees. True, each connected component will be a single tree in a DFS.
★ Analyse a DAG using DFS, choosing search order st all edges in analysis will be cross edges. > Search nodes in reverse order by topological sort, guarantees that every edge traversed will point to a node that is already finished.
★ Every DAG has only one topological ordering of its vertices. False, {(1,2),(1,3)} can be sorted either [1,2,3] or [1,3,2]
★ You can have multiple back edges but possible to remove 1 edge that destorys all cycles. EG. {(1,2),(2,1),(2,3),(3,1)}, remove (1,2) disrupts both cycles.
★ Give an O(V+E) algo to remove all cycles in a directed graph. Do a DFS, as you traverse, check the edge goes to a node that has been seen but not finished, if so, store in a set. After the DFS, remove all edges in this set.
★ If a DFS on a connected undir graph produces n back edges then a BFS on same graph produce n cross edges. True, BFS on undir graph has no forward/back edges, while DFS on undir graph has no forward/cross edges.
★ The depth of any DFS tree rooted at a vertex is at least as much as depth of any BFS tree rooted at same vertex. True, since BFS finds paths using fewest num of edges.
★ There is no edge in undir graph that jumps more than 1 level of any BFS tree. True, if such an edge existed, it would provide a shorter path to some node than the path found by BFS.
★ BFS takes O(V+E) irrespective of whether presented with adj list or adj matrix. False, with adj matrix, visiting each vertex takes O(V) as we must check all N possible outgoing edges in matrix. Thus, BFS take O(V^2) time using matrix. | DFS using matrix also O(V^2)
★ Let P be shortest path from vertex s to t. If weight of each edge in graph increased by one, P will still be shortest path from s to t. False. Let w(s,v1),w(v1,v2),w(v2,t)=1 and w(s,t)=4 | if weight x2, P is still shortest (linear transformation)
★ Changing RELAX function to update if d[v]≥d[u]+w(u,v) will change correctness of Bellman-Ford outputs d & π. Parent pointers may not lead back to source node if a zero-length cycle exists.
★ If weighted directed graph G is known to have no shortest paths longer than k edges, then it suffices to run BF for only k passes. True. ith iteration finds shortest path in G of i or fewer edges by path relaxation property.
★ If priority queue in Dijkstra is implemented using sorted linked list, running time θ(V(V+E)). Array take θ(V) to insert a node, V nodes to insert, θ(E) calls to decrease key can take θ(V) time.
★ Implement Dijkstra using priority queue, requires O(V) to initialise, worst-case f(V,E) for each extractmin op & worst-case g(V,E) time for each decreasekey op. Whats worst-case to run Dijkstra? O(V.f(V,E) + E.g(V,E))
★ For Dijkstra: non-negative weights. To transform w∈(0,1], let w'=-log(w)
★ Given a directed graph with exactly one neg weight edge & no negweight cycles. Give an algo to find shortest dist from s to all other vertices that has same running time as Dijkstra. Let neg weight edge be (u,v). Remove the edge and run Dijkstra from s. check if δ(s,u)+w(u,v)<δ(s,v). If not, we are done. If yes, run Dijkstra from v, for any node t, its shortest dist from s will be min{δ(s,t),δ(s,u)+w(u,v)+δ(v,t)}
★ Dijkstra can be implemented using AVL tree instead of heap for priority queue and still run in O((V+E)lgV) time. True, both decreasekey and extract min for AVL can be done in O(lgV) time.
★ In Bellman-Ford, if we have 2 consec edges on a shortest path: (u,v) & (v,w). Edge (v,w) is guaranteed to be relaxed within O(1) relaxations of (u,v) being relaxed. False, BF runs repeatedly through all edges in some arbitrary order, so 2 edges may be visited θ(E) steps from each other.
★ Multi-source shortest path distance: Add dummy vertex s*, add zero-weight edges from s* to all s∈S. Run Dijkstra from s*.
★ Performing a left rotation and then a right rotation on same node will not change the tree structure. False. To undo left rotation on x, must do a right rotation on y (initial right child of x)