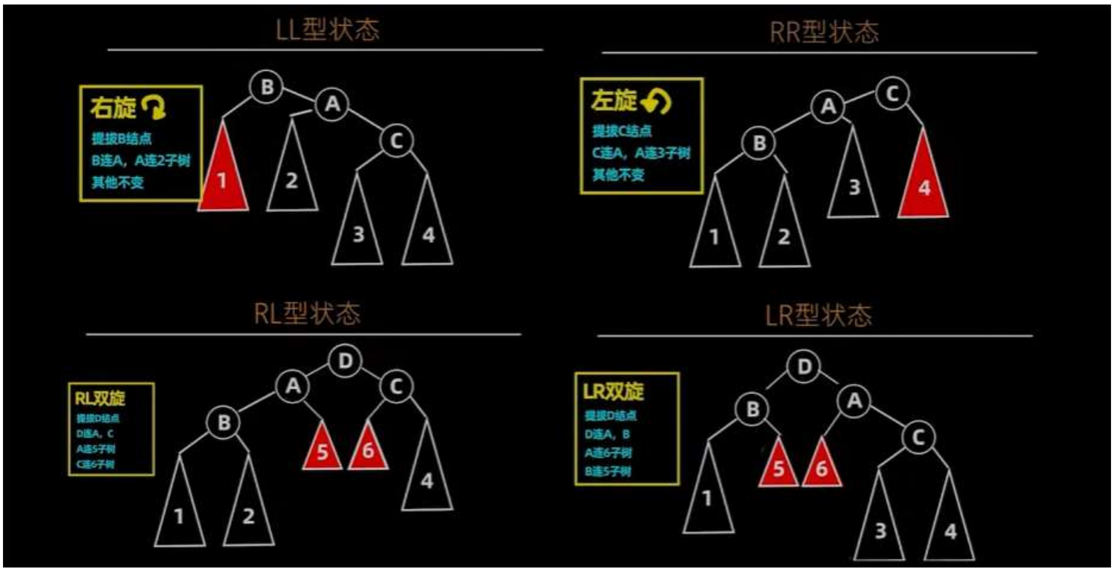
CSC3100: Data Structure

Final Review

Baicheng Chen
SDS, CUHK-Shenzhen
baichengchen@link.cuhk.edu.cn

AVL Tree

1. Motivation: keep the BST completed => balanced BST
   An AVL tree is a self-balancing BST:
   (1) For every node in the tree, the height of the left subtree differs from the height of the right subtree by at most 1. (balance factor: height(left subtree) – height(right subtree)) => balance factor of every node is between -1 and 1.
   (2) If at any time they differ by more than 1, rebalancing is done to restore this property.
2. Insertion: 4 cases => LL, RR, LR, RL



3. Deletion: 查询、删除（无子树/一个子树，两个子树→前驱后继转化）、调整（突出）
4. Pros and cons of AVL trees:
   Pros:
   (1) Search is O(logn) since AVL trees are always balanced
   (2) Insertions and deletions also cost O(logn) time
   (3) The height balancing adds no more than a constant factor to the speed of insertion
   Cons:
   (1) Difficult to program & debug; more space for balance factor
   (2) Asymptotically faster but rebalancing costs time
   (3) Most large searches are done in database systems on disk and use other structures (e.g., B-trees)

Heap

1. Priority queue
2. Binary heap: insert and delete both in O(logn)
   (1) Structure property:

A heap is a complete binary tree
A complete binary tree of height h has between 2^h and $2^{h+1} - 1$ nodes
The height of a complete binary tree = $\lfloor logN \rfloor$

(2) Heap order property: The value at any node should be smaller than (or equal to) all of its descendants (guarantee that the node with the minimum value is at the root)

3. Insert: (1) creating a hole, (2) bubbling the hole up (worst case O(logn) – the new element is percolating up all the way to the root)
4. deleteMin: worst case O(logn)
   (1) delete the root
   (2) fill the root with the last node X
   (3) percolate X down
5. Binary heap construction:
   (1) Naïve way: repeatedly insert nodes one by one, O(nlogn)
   (2) Faster way: N successive appends at the end of the array, each taking O(1), so the tree is unordered: O(n)
   for (i = n/2; i > 0; i --) percolateDown(i);
6. Min-heap, Max-heap
7. HeapSort: sorting using a max-heap
   (1) Create a max-heap H with a capacity of arr. length + 1
   (2) Repeatedly delete the max element from the max-heap until it become empty

Hashing

1. Motivation: searching (keys matching a given search key + key-value pair)
2. Applications: Keeping track of customer account information at a bank, keeping track of reservations on flights, search engines, applications need a lot of queries.
3. First solution: direct addressing
   一一对应
   Limitation: The universe of the keys is usually very large

	Insert	Search
direct addressing (keys are the indexes)	O(1)	O(1)
ordered array (keys are not indexes)	O(N)	O(logN)
ordered linked list	O(N)	O(N)
unordered array (keys are not indexes)	O(N)	O(N)
unordered linked list	O(1)	O(N)
binary search tree	O(logN)	O(logN)

4. Main idea: In hashing, the element is stored in slot h(k), i.e., T[h[k]], where h is a hash function.
5. Collision: two keys hash to the same slot
   Chaining: we place all elements that hash to the same slot into the same linked list
   ⇨ Query time is O(1)
6. Open addressing: all elements are stored in the hash table; for insertion, we examine, or

probe, the hash table until an empty slot is found to put the key.

Probe? Linear probing and double hashing

(1) Linear probing: 不断+1，直到找到空位 h(k, i) = (h(k)+i)%m from i=0 to i=m-1

(2) Double hashing: two hash functions

Double hashing

- We have an additional hash function $h' > 0$
- Insertion: we probe $h(k, i) = \big(h(k) + i \cdot h'(k)\big)\%m$ one by one for i from 0 to $m-1$ until an empty slot is found
- Search: we search $h(k, i)$ for i from 0 to $m-1$ until one of the following happens:
  - $T[h(k, i)]$ has the record with key equal to k
  - $T[h(k, i)]$ is empty, then no record contains key k in the hash table

Query time is O(1).

7. Pros of chaining:

   (1) Less sensitive to hash functions and load factors (α can be larger than 1), while open addressing requires to avoid long probes, and its load factor α < 1

   (2) Support deletion, while open addressing is difficult to support deletion

   Pros of open addressing: Usually much faster than chaining

8. What is a good hash function? => uniform hashing property: each key is equally likely to hash to any of the m slots, independent of where other keys will hash to

   Division, universal hashing:

   Division is effective in practice without any theoretical guarantee on O(1) query time

   Universal hashing provides theoretical guarantees on O(1) query time

9. Division: h(k) = k mod m = k % m

   If m = 10^p, then h(k) only uses the lowest-order p digits of the key value k

   Option of m: choose a prime number not close to the power of 2 or 10 (m=701)

10. Universal hashing

- Let $\mathcal{H}$ be a family of hash functions from $[U]$ to $[m]$
- $\mathcal{H}$ is called universal if the following condition holds:

> Let k_1, k_2 be two distinct integers from $[U]$. By picking a function $h \in \mathcal{H}$ uniformly at random, we guarantee that
> $$\Pr[h(k_1) = h(k_2)] \le \frac{1}{m}$$

  - Then, we choose one from $\mathcal{H}$ uniformly at random and use it as the hash function h for all operations

Construct a universal family H of hash function from [U] to [m]

(1) Pick a prime number p (p > U)

(2) For every $\alpha \in \{1, 2, \ldots, p-1\}$, and every $b \in \{0, 1, 2, \ldots, p-1\}$ => p * (p - 1) functions

$$h_{a,b}(k) = \big((a \cdot k + b)\ mod\ p\big) mod\ m$$

Graph

BFS & DFS

1. Definition: G (V, E), V nodes and E edges.
2. Degree: the number of neighbors of a node v => d(v)

   Connected: there is a path from every vertex to every other vertex
3. Graph representation:

   (1) Adjacency list: O (n + m)

   (2) Adjacency matrix: A[u][v] = 1 if (u, v)∈E, O (n^2)

   Undirected graph: A is symmetric; Directed: A is not symmetric

   (3) Comparison:

- Adjacency list:
  - Space: $O(n+m)$, save space if the graph is sparse, i.e., $m \ll n^2$
  - Check the existence of an edge (u,v): $O(k)$ time where k is the number of neighbors of u
  - Retrieve the neighbors of a node: $O(k)$ time
  - Add/delete a node: $O(n)$
  - Add/delete an edge: $O(k)$

- Adjacency matrix:
  - Space consumption: $O(n^2)$
  - Check the existence of an edge (u,v): $O(1)$ time
  - Retrieve the neighbors of a node: $O(n)$ time
  - Add/delete a node: $O(n^2)$, (create a new matrix)
  - Add/delete an edge: $O(1)$

4. BFS: queue

- At the beginning, color all nodes to be white
- Create a queue Q, enqueue the source s to Q, and color the source to be gray (meaning s is in the queue)
- Repeat the following until queue Q is empty
  - Dequeue from Q, let the node be v
  - For every out-neighbor u of v that is still white
    - Enqueue u into Q, and color u to gray (to indicate u is in queue)
  - Color v to be black (meaning v has finished)

Time complexity: O (n + m) [with adjacency list representation]

5. DFS: stack

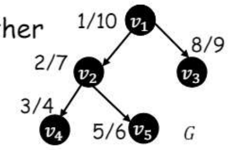
- Initialization:
  - At the beginning, color all nodes to be white
  - Create a stack S, push the source s to S, and color the source to be gray (meaning s is in the stack)

- Repeat the following until S is empty
  - Get the top node, denoted as v, on stack S, **do not** pop v
  - If v still has white out-neighbors
    - Let u be such a white out-neighbor of v
    - Push u to S, and color u to gray
  - Otherwise (v has no white out-neighbors)
    - Pop v and color it as black (meaning that v has finished)

Time complexity: O (n + m) [with adjacency list representation]

Properties:

- Let $u.d$ and $u.f$ to indicate their first discovery time and their finish time, respectively, and denote $I(u)$ as the interval $[u.d, u.f]$
- We will only have three cases for two nodes u and v
  - $I(u) \subset I(v)$, u is the descendant of v
  - $I(v) \subset I(u)$, v is the descendant of u
  - $I(u) \cap I(v) = \emptyset$, neither one is the descendant of the other
  - Example:
    - $I(v_2)$: $[2,7]$, $I(v_4) = [3,4]$
      - v_4 is a descendant of v_2 in the DFS tree



- We can check if a node u is a descendant of another node v in $O(1)$ time
  - If there is no such property, we need to retrieve the path

Minimum Spanning Tree

1. Spanning tree: A tree which contains all the vertices of the graph.

   MST: spanning tree with minimum sum of weights => not unique; no cycles

2. Growing an MST:

   (1) Generic approach: Grow a set A of edges, incrementally add “safe” edges to A.

   Finding “safe” edges:

   Cut (S, V-S) => partition into two disjoint sets;

   “safe” edge: (u, v) is a light edge crossing (S, V-S)

   (2) Prim:

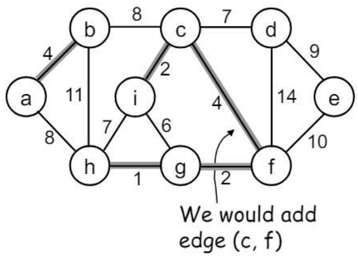
   - The edges in set A always <u>form a single tree</u>
   - Starts from an arbitrary “root”: V_A = {a}
   - At each step:
     - Find a light edge crossing (V_A, V - V_A)
     - Add this edge to A
     - Repeat until the tree spans all vertices

   Find light edges quickly? Priority queue Q.

   Time: O (mlogn) => adjacency list; O (mlogn + n^2) => matrix

   (3) Kruskal:

- Start with each vertex being its own component
- Repeatedly merge two components into one by choosing the **light** edge that connects them



- Which components to consider at each iteration?
  - Scan the set of edges in monotonically increasing order by weight

- A naïve method: O (mn)
- Using labels: a label means a connected component: O (mlogm)

Graph shortest path

1. Remark:

   (1) Shortest paths cannot contain cycles
2. A simple case: unweighted graph => BFS! O (E + V)

   A simple algorithm

   1. Mark the starting vertex, s
   2. Find and mark all unmarked vertices adjacent to s
   3. Find and mark all unmarked vertices adjacent to the marked vertices
   4. Repeat Step 3 until all vertices are marked
3. Dijkstra: greedy algorithm => solving a problem by stages by doing what appears to be the best thing at each stage

   - Select a vertex u, which has the smallest d_u among all the unknown vertices, and declare that the shortest path from s to u is known
   - For each adjacent vertex, v, update $d_v = d_u + c_{u,v}$ if this new value for d_v is an improvement

   Running time: O (mlogn) for min-heap implementation

- Given a directed graph G=(V,E) where each edge (u, v) has an associated value r(u,v), which is a real number in the range $0 \le r(u,v) \le 1$ that represents the reliability of a communication channel from vertex u to vertex v
  - We interpret r(u,v) as the probability that the channel from u to v will not fail, and we assume that these probabilities are independent
  - Give an efficient algorithm to find the most reliable path between two given vertices

4. All pairs shortest path: find the shortest distance/path between every pair of vertices of a graph
5. A naïve method: run a single-source shortest path algorithm for each vertex => O (mnlogn)
6. Floyd: O (n^3) => dense graph is faster

- Main idea of Floyd's algorithm
  - One way is to restrict the paths to only include vertices from a restricted subset
  - Initially, the subset is empty
  - Then, it is incrementally increased until it includes all the vertices

- Since

$$D^{(k)}[i, j] = D^{(k-1)}[i, j] \text{ or}$$
$$D^{(k)}[i, j] = D^{(k-1)}[i, k] + D^{(k-1)}[k, j]$$

We conclude:

$$D^{(k)}[i, j] = \min\{ D^{(k-1)}[i, j], D^{(k-1)}[i, k] + D^{(k-1)}[k, j] \}$$

```
Floyd
D ← W    // initialize D array to W[ ]
P ← 0    // initialize P array to [0]
for k ← 1 to n
    do for i ← 1 to n
        do for j ← 1 to n
            if (D[ i, j ] > D[ i, k ] + D[ k, j ] )
                then  D[ i, j ] ← D[ i, k ] + D[ k, j ]
                      P[ i, j ] ← k;
```

O(|V|^3)

Can only use a single D matrix to implement.

7. Dijkstra's algorithm cannot handle a graph that has negative weights but no negative cycles
8. Bellman-Ford: allow negative edge weights (无负环) – can detect negative cycles

- Idea:
  - Each edge is relaxed |V| - 1 times by making |V| - 1 passes over the whole edge set
  - Any path will contain at most |V| - 1 edges

Relaxation: if d[v] > d[u] + w (u, v): d[v] = d[u] + w (u, v);

Steps: pass 1 edge/ 2/ 3/ 4

Detecting negative cycles (perform extra test after V – 1 iterations):

```
for each edge (u, v) ∈ E
    do if d[v] > d[u] + w(u, v)
        then return FALSE
return TRUE
```

Time: O(mn)　轮数最多 n - 1 轮，如果第 n 轮是仍然存在松弛操作→负环

If there is no negative cycle, then after |V| - 1 iterations, d values will not be updated or can't be lowered any more, and d values store the measure of the shortest path

9. Topological sort: no cycle! A Directed acyclic graph (DAG) has at least one topological ordering.
对于图中每条边(x, y), x 在 A 中都出现在 y 之前，则称 A 是该图的一个拓扑序列

- An ordering of all vertices in a directed acyclic graph, such that if there is a path from v_i to v_j, then v_j appears after v_i in the ordering
- If there is no path between v_i and v_j, then any order between them is fine

- A simple algorithm
  - Compute the indegree of all vertices from the adjacency information of the graph
  - Find any vertex with no incoming edges
  - Print this vertex, and remove it, and its edges
  - Apply this strategy to the rest of the graph

1）在图中找到所有入度为0的点
2）把所有入度为0的点在图中删掉，重点是删掉影响！继续找到入度为0的点并删掉影响
3）直到所有点都被删掉，依次删除的顺序就是正确的拓扑排序结果
4）如果无法把所有的点都删掉，说明有向图里有环

Running time: O (n^2)

- An improved algorithm: O(|E|+|V|) time
  - Keep all the unassigned vertices of indegree 0 in a queue
  - While queue is not empty,
    - Remove a vertex in the queue
    - Decrease the indegrees of all adjacent vertices
    - If the indegree of an adjacent vertex is 0, enqueue the vertex

DAG and SCC computation

1. Directed acyclic graph (DAG), Strongly connected component (SCC)
2. DAG: a directed graph that contains no cycles
   Check: DFS!

- To apply to the entire graph:
  - Randomly generate a permutation of the nodes and repeat the following until there is no white node
    - Pick the first white node *s* in the permutation and do DFS (during DFS, we will color nodes, and record timestamps)

Let <u, v> be an edge in G, it can be classified into three types:
(1) Forward edge: if u is an ancestor of v in one of the DFS-trees
(2) Backward edge: if u is a descendant of v in one of the DFS-trees
(3) Cross edge: if none of the above happens
How to judge?

Interval $I(u)$ of node u is $[u.d, u.f]$, where $u.d$ is the first discovery time and $u.f$ is the finish time

- We will only have three cases for two nodes u and v
  - $I(u) \subset I(v)$, u is the descendant of v backward edge
  - $I(v) \subset I(u)$, v is the descendant of u forward edge
  - $I(u) \cap I(v) = \emptyset$, neither one is the descendant of the other.

Given the DFS result on G, then G contains a cycle iff there is a backward edge.

- Step 1: Do DFS traversal on graph G
  - Time complexity: $O(n+m)$ (permutation can be done in $O(n)$)
- Step 2: Classify edges according to the interval of each node derived with DFS
  - Time complexity: $O(m)$
- Step 3: If there exists a backward edge, G contains a cycle, otherwise, G is a DAG
- Total time complexity: $O(n+m)$

3. Connected (undirected) Graphs
4. SSC

How to solve SCC problem?

- A naïve approach: $O(n^2(n+m))$

```
For each i,j in nodes:
    If i is reachable from j and vice versa
        Then i and j are in the same SCC
```

- Another approach: $O(n(n+m))$

```
Array of bool reachable
For each i in nodes:
    DFS and put the visited array inside reachable of i
For each i,j in nodes:
    If reachable[i][j] and reachable[j][i]
        Then i,j are in the same SCC.
```

Also, 3 algorithms with linear time.

- Kosaraju-Sharir algorithm [1]
  - Run DFS on G, and get a post order
  - Run DFS on G^T and output SCCs
- Path-based algorithm [2]
  - A single DFS with sub-path contraction
- Tarjan's algorithm [3]
  - A single DFS; Each SCC corresponds to a sub-tree

K-S algorithm:

Fact: the transpose graph (the direction of every edge reversed) has exactly the same SCCs as the original graph

Input: a directed graph $G=(V, E)$

Output: all the SCCs of G

1. Run DFS on G, during which we compute the first discovery time and finish time of each vertex
2. Build the transpose graph $G^T=(V, E^T)$
3. Run DFS on G^T, by considering the vertices' finish time in descending order
4. Output the vertex set in each DFS traversal as an SCC

Time: O (n + m)