8.13 – 9.13

8.14-8.15 复习和总结373的所有ppt

8.16-8.17 复习154 sql部分

8.18-8.19 做完Pokedex, 加到resume里

8.20-9.12 每天7道题

***Induction***

1. The thing you want to prove (e.g sum of integers from 1 to n is equal to n\*(n+1)/2)
2. The base case
3. The assumption step
4. The induction step

**Abstract Data Type**

***Array***

1. Declaration:

* *Type*[ ] a = new *type*[length] (e.g. int[ ] s = int[10])

1. Frequently Used Methods:

* **.**length
* *array*[index]
* Arrays.toString(*array*)

***Stack(Last-in-first-out)***

1. Declaration:

* Stack s = new Stack();
* Stack<*type*> s = new Stack<*type*>();

1. Frequently Used Methods:

* Inherited methods from Vector class: *isEmpty( ), Equals(e), size( ), hashCode()*
* *peek( ):* Looks at the object at the top of this stack without removing it
* *pop( )*: Removes and returns the object at the top of the stack
* *push(e)*: Pushes an item onto the top of the stack

1. Features:

* Can be self-implemented use *Array* or *LinkedList*

***Queue(First-in-first-out)***

1. Declaration:

* Queue q = new LinkedList();
* Queue<*type*>q = new LinkedList<*type*>();
* Queue q = new PriorityQueue();
* Queue<*type*> q = new PriorityQueue<*type*>();
* (And many kinds of Queues, can be checked on Java documentation)

1. Frequently Used Methods:

* Inherited methods from Collection class*: isEmpty(), Equals(e), size(), hashCode()*
* *add(e)*: Inserts the element into the queue if it’s possible and return true; return false otherwise
* *offer(e)*: Inserts the element into the queue if it’s possible
* *peek()*:Retrieves but not remove the element at the head of the queue
* *poll()/remove()*: Retrieves and removes the element at the head of the queue

1. Features:

* Can be self-implemented use *Array* or *LinkedList*

***List***

1. Declaration:

List<type> li = new ArrayList<type>();

List<type> li = new LinkedList<type>();

List<type> li = new Vector<type>();

1. Frequently Used Methods:

* .add(*e*);
* .size();

***LinkedList***

1. Declaration

* List l1 = new LinkedList();

1. Frequently Used Methods

* **.**value;
* **.**next;

1. Features

* Consists of list nodes

***ArrayList***

1. Declaration

* List<*type*> al = new ArrayList<*type*>();
* ArrayList<*type*> al = new ArrayList<*type*>();

1. Frequently Used Methods

* get(*index*);
* size();
* add(*value*);
* add(*index, value*);
* remove(*index*);
* contains(*value*);
* set(*index, value*);
* **.**equals(object);

1. Features

* Initially use Array to store value, but unlike an Array, ArrayList take care of the size of the storage space

***Map***

1. Declaration:

Map m = new HashMap();

Map<*keyType*, *valueType*> m = new HashMap<*keyType, valueType*>();

Map n = new TreeMap();

Map<*keyType, valueType*> n = new TreeMap<*KeyType, valueType*>();

1. Frequently Used Methods:

* clear(): remove all the mapping in the map
* get(*key*): returns the value to which the specified key is mapped
* put(*key, value*): Associates the specified value with the specified key
* size(): returns the number of the key-value pairs in the map
* isEmpty(): returns the Boolean
* remove(*key*):removes the mapping for a key

***Binary Tree***

1. Representation: Left, right, left.value, right.value
2. **Tree Traversal**: (The word “order” represents the root order)

* Pre-Order: root, left, right

Void preOrderTraversal(Node n) {

If (n != null) {

Process(n.value);  
 inOrderTraversal(n.left);

inOrderTraversal(n.right);

}

}

* In-Order: left, root, right
* Post-Order: left, right, root

***Binary Search Tree (BST)***

1. Representation: Left, right, left.value, right.value
2. Structure Property:

* Each node has less than 2 children
* All keys in the left subtree smaller than the nodes key
* All keys in the right subtree larget than the nodes key

1. Features:

* A binary search tree is a type of binary tree, but not all binary trees are BST

1. Related Algorithms:

* **Insertion:**

1. If current node is null, put the inserting value here
2. If the inserting value is larger than the current node, go to left node; otherwise go to right node

* **Deletion：**

1. If the node being deleted has no child or only one child, just delete it and replace the position with its child
2. If the node being deleted has two children, find the maximum node in the left subtree or minimum node in the right subtree of the node that being deleted; Put either of the “min” or “max” value into the deleted node position

* **Building:**

1. Depends on the input order, worst case(n^2), best case(nlogn)

***AVL Tree***

1. The AVL balance condition: Left and right subtree of every node have heights differing by at most 1
2. Related Algorithms:

* **Rotation:**

1. Find the deepest unbalanced node, whose left subtree’s height and right subtree’s height has a difference more than 1 (empty node count as -1 height, leaf node count as 0 height)
2. Decides which kind of unbalanced situation it is depending on the grandchild of the node, same direction (left- left or right-right) or different directions (left-right or right-left)
3. If it is same direction, then ***single rotation*** is enough:
4. Move left (child) of unbalanced node into parent position
5. Parent node becomes the right (left) child
6. Other subtrees move in the only way BST allows
7. If it is different directions, then use ***double rotation***:
8. Rotate the problematic child and the problematic grandchild
9. Rotate the node itself with the new child

* **Insertion:**

1. Insert the new node as in a BST
2. After insertion in a subtree, detect height imbalance and perform a rotation to restore balance at the node
3. Pros and Cons of AVL trees

* Pros: All operations are logarithmic worst-case because trees are always balanced
* Cons: more space for height field

***Heap***

1. Features:

* A heap is a tree, but not BST
* Heap property: The priority of every node is less than its parent; there is no relationship like left subtree or right subtree in BST

1. Operations basic ideas:

* ***findmin:***return root.data
* ***deleteMin:*** return root.data, then move the right-most in the last row to the root, then percolate down to restore heap property
* ***Insert****:* Put the new node in the next position on the bottom row, percolate up to restore heap property
* ***Percolate down:***

1. Keep comparing priority with both children
2. If the priority of the node is less than either of the child, swap with the most important child (smallest child in this case) and go down one level
3. Done if the node’s priority is greater than both of the child, or reached a leaf node

* ***Percolate Up:***

1. Put data in the new location
2. If parent is less important than the node, swap with parent and continue
3. Done if the parent is more important than the node, or reached root

***PriorityQueue***

1. Features:

* PriorityQueue holds **comparable**-data type
* Typically has two fields, the priority and the data
* Must implement a comparable method if the object is not naturally comparable
* We choose binary heap to implement this function

1. Frequently Used Methods:

* add(*E e*)/offer(E e): Insert specific element into the priorityQueue
* peek(): retrieves but not removes the head of this queue
* poll(): retrieves and removes the head of this queue
* size();
* isEmpty();

***Disjoint sets***

1. Features:

* A set is a collection of elements (no-repeats)
* Two sets are said to be disjoint if there have no element in common

***The Union-Find ADT***

1. Features:

* The union-find ADT keeps track of a set of elements **partitioned** into a number of disjoint subsets

1. Operations:

* **Create:** given an unchanging set S, create an initial partition of a set

1. Typically each item in its own subset: {a}, {b}, {c}…
2. Give each subset a “name” by choosing a representative element

* **Find**: takes an element of S and returns the representative element subset it is in
* **Union**: takes two subsets and makes one larger subsets

1. A different partition with one fewer set
2. Choice of representative element up to implementation
3. Building a maze use union-find:

* Partition the maze into disjoint sets

1. Two cells in the same set if they are connected
2. Initially every cell is in its own subset

* If removing an edge would connect two different subsets

1. Then remove the edge and ***union*** the subsets
2. Else leave the edge because removing it makes a cycle

* The algorithm:

1. P = disjoint sets of connected cells, initially each cell in its own 1-element set
2. E = set of edges not yet processed, initially all edges
3. M = set of edges kept in maze, initially empty

While P has more than one set {

Pick a random edge (x,y) to remove from E;

u = find(x);

v = find(y);

if (u==v) //Same subset, do not remove edge, do not create cycle

add (x,y) to M;

else // Connect subsets, do not put edge in M

Union(u,v);

}

Add remaining members of E to M, and then output M as the maze

* Implementation of Union-find:

1. Use “up tree” data structure
2. No limit on branching factor
3. Reference from children to parent
4. **find(x)**
5. Assume we have O(1) access to each node (Use an array where index i holds node i)
6. Start at x and follow parent pointers to root
7. Return the root
8. **Improvement:** path compression: As part of **find**, change each encountered node’s parent to point directly to root
9. **Union(x,y)**
10. Assume **x** and **y** are roots, else find the roots of their trees
11. Assume distinct trees (else do nothing)
12. Change root of one to have parent be the root of the other
13. If the sets are continuous numbers, use an array of length called up
14. If the sets are not continuous numbers, could have a separate dictionary to map elements (key) to numbers (values)
15. **Improvement:** union-by-size: connect smaller tree to larger tree

***B-Trees***

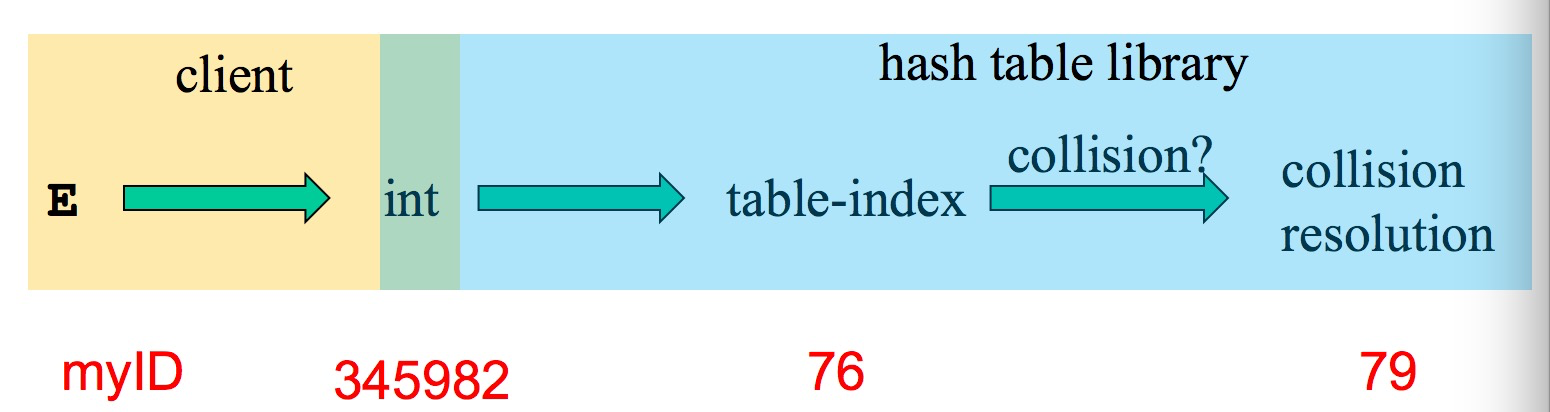
* Features:

1. B+-trees are multi-way search trees that are kept somewhat shallow to limit disk accesses
2. B+-trees are often used in implementation for the database; the data are stored only in leaf nodes
3. Children of each “internal” node are between the “keys” in that node
4. All leaves are at the same depth

***HashTable***

* Features:

1. Need the stored element to be hashable and comparable
2. Usually use an array to store data, array index as table index



* Collision Avoidance:

1. Large table size tends to help but not always
2. Pick table size to be prime
3. Separate Chaining: All keys that map to the same location are kept in a list
4. Probing Chaining: try the next spot that are available
5. Linear Probing: bad, could produce primary clustering
6. Quadratic Probing: i^th probe: (h(key) + i^2) % tableSize

* Use double hashing to avoid secondary clustering: use two hash functions for hashing e.g. index = (h(i) + i\*g(i)) % tableSize

1. Rehashing: extends the tableSize

***Spanning Trees***

* Definition: Given a connected undirected graph G=(v,e), find minimum subset of edges such that G is still connected
* Features:

1. A spanning tree connect all the nodes with as few edges as possible
2. Any solution to this problem is a tree
3. Solution not unique unless original graph was already a tree

* Two Approaches:

1. Do a graph traversal, keeping track of edges that form a tree
2. Iterate through edges, add to output any edge that not create a cycle (To detect cycle, use union-find with disjoint-sets)

**Algorithm**

***Binary Search***

1. Where to use: to find certain value in a sorted array
2. Algorithm:

* Begin with the interval covering the whole array, evenly divided the array into two parts
* If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half; Otherwise narrow it to the upper half
* Repeatedly check until the value is found or the interval is empty

1. Compare the target value with the middle element
2. If x matches with middle element we return the mid index
3. Else if x is greater than the mid element, then recur to the right half interval
4. Else recur for the left half interval

1. Efficiency: O(logn)

***Topological sorting:***

1. Where to Use: In some situations, we need to find out the linear path in a Directed Acyclic Graph (DAG), in which the relative hierarchy or order of the elements are dependent to each other
2. Algorithm:
3. Label (“mark”) each vertex with its in-degree
4. While there are vertices not yet output:

* Choose a vertex v with in-degree of 0
* Output v and mark it removed
* For each vertex **u** adjacent to **v**, decrement the in-degree of **u** by 1

1. Efficiency:

***DFS(Iteration)***

1. Mark the root node as start
2. For each node **u** adjacent to start
3. If u is not marked
4. DFS(**u**)

* Exactly what we called a “pre-order” traversal for trees
* The marking is because we want arbitrary graph and we want to process each node exactly once

***DFS(Stack)***

1. Initialize a stack **s** and push the root node as start
2. Mark start as visited
3. While **s** is not empty
4. Pop the next node in **s**
5. For each node **u** adjacent to next
6. If **u** is not marked, mark **u** and push onto **s**

* DFS use more space than BFS in finding a path

***BFS(Queue)***

1. Initialize a queue **q** and enqueue the root node as start
2. Mark start as visited
3. While **q** is not empty
4. Dequeue the next node in q
5. For each node **u** adjacent to next
6. If **u** is not marked, mark **u** and enqueue onto **q**

* A “level-order” traversal
* Always find a shortest solution if one exists and there is enough memory

***Iterative Depth First Search (IDFS)***

1. Iterative deepening:
2. Try DFS but disallow recursions more than **K** levels deep
3. If that fails, increment **K** and start the entire search over

* Advantages: Like BFS, can find the shortest pat. Like DFS, use less space

***Dijkstra’s Algorithm***

1. Initially, the start node has cost 0, known, and other nodes have cost to and unknown
2. While there are unknown nodes in the graph
3. Select the unknown node v with the lowest cost
4. Mark v as known
5. For each edge v connected to u with weight w
6. cost1 = v.cost + w
7. cost2 = u.cost
8. If cost1 is less than cost2, assign u.cost with value of cost1, u.path equal to v

* Where to use: for weighted graph (directed or undirected) with no negative-weight edges to find the shortest path from single given start node to all other nodes in terms of the weights on the edges
* Basic idea is BFS
* An example of greedy algorithm: at each step the locally best choice is made
* The path solution is globally optimal
* Could be improved by using a priority queue holding all unknown nodes, sorted by cost. (But support **decreaseKey**() operation)

***Prim’s Algorithm***

1. Grow a tree by picking a vertex from unknown set that has the unknown set that has the smallest cost. Here, cost is the cost of the edge that connects the vertex to the known set. (Pick the vertex with the smallest cost that connect “known” to “unknown”)
2. Prim's VS Dijkastra’s

* Dijkastra’s picked the unknown vertex with the smallest cost; where cost = distance to the *source*;
* Prim’s picked the unknown vertex with the smallest cost;

Where cost = distance from this vertex to the *known set*;

* Otherwise identical;

1. Algorithm
2. For each node v, set v.cost = infinity, and v.known = false;
3. Choose any node v
4. Mark v as known
5. For each edge (u,v) with weight w, set u.cost = w and u.prev = v
6. While there are unknown nodes in the graph
7. Select the unknown node v with the lowest cost
8. Mark v as known and add (v, v.prev) to output
9. For each edge (v,u) with weight w,

If (w < u.cost) {

u.cost = w;

u.prev = v;

}

***Kruskal’s Algorithm***

1. Grow a forest out of edges that do not create a cycle. Pick an edge with the smallest weight
2. Algorithm:
3. Sort edges by weight (min-heap)
4. Each node in its own set(up-trees)
5. While output size < |V| - 1 {
6. Consider next smallest edge (u, v)
7. If **find(u)** and **find(v)** indicate u and v are in different sets

* Consider next smallest edge(u,v);
* If **find(u)** and **find(v)** indicate u and v are in different sets

Output(u,v);

**Union(find(u), find(v))**

}

1. Features:

* Iterate through edges using union-find for cycle detection and **deleteMin** to get the next edge
* Not better worst-case asymptotically, but often stop long before considering all edges

***Insertion Sort***

1. Idea: at step k, put the k^th element in the correct position among the first k element

***Selection Sort***

1. Idea: at step k, find the smallest element among the not-yet-sorted elements and put it at the position k

***Heap Sort***

1. Idea: Insert each array[i] into a separate heap, or better yet use ***buildHeap***

For (i = 0; i < arr.length; i++) {

arr[i] = deleteMin();

}

***Merge Sort***

1. Idea: Sort the left half of the elements (recursively), sort the right half of the elements (recursively), merge the two sorted halves into a sorted whole
2. Algorithm

* To sort array from position **lo** to position **hi**

-If range is 1-element long, it is already sorted! (Base case)

-Else:

Sort from **lo** to (**hi + lo**) / 2

Sort from (**hi + lo**) / 2 to **hi**

Merge the two halves together

1. Features:

* Merge sort is the basis of massing sorting because its relatively low random disk access

***Quick Sort***

1. Idea: Pick a “pivot” element; divide the element into less than pivot and greater than pivot; Sort the two divisions (recursively on each)
2. Features:

* Recursively chop into two pieces
* Instead of doing all the work as we merge together, we will do all the work as we recursively split into halves
* Unlike merge sort, does not need auxiliary space
* Often believed to be faster than merge sort – It does fewer copies and more comparisons, so it depends on the relative cost of these two operations

1. Algorithm:

* Pick a pivot element
* Partition all the data into:

1. The element less than the pivot
2. The pivot
3. The elements greater than the pivot

* Recursively sort I. and III.
* How to pick the pivot? Any choice is correct: data will end up sorted, but better want two partitions to be about equal in size since it will make the algorithm faster

1. Implementation:

* Partitioning:

- Swap pivot with **arr[lo];**

**-** Use two pointers **i** and **j**, starting at **lo + 1** and **hi** **- 1**

- While (i < j)

if (arr[j] > pivot ) j--;

else if (arr[i] < pivot) i++;

else swap arr[i] with arr[j]

- Swap pivot with **arr[i] \***

***Bucket Sort (Bin Sort)***

1. Idea: If all values to be sorted are known to be integers between 1 and K (or any small range)

- Put each element in its proper bucket (bin), the array index represents the integer and the value in the index is the count of the occurrence of this integer

- If data is only integers, no need to store more than a count of how many times bucket has been used

-Output result via linear pass through array of buckets

***Radix Sort***

1. Idea: Used in a number system

- Bucket sort on one digit at a time

* Number of bucket = radix
* Starting with least significant digit
* Keeping sort stable

- Do one pass per digit