HashMap (time complexity, hash->array, hash->arrayList)

Single List/Double linked list (Iterator, remove, insert)

Stack/Queue/PriorityQueue /Heap

Tree

(BST, pre-order, inorder, post order, dfs, bfs, depth, balanced tree, full tree)

Bit manipulation

(most significant bit, least significant bit)

Search – Binary search

Sort

(merge sort (Divid and Conquer), insertion sort)

Graph

(vector graph, non-vector graph, dfs/bfs iterate)

Backtrack

(base condition, recursion)

DP

(数组型，序列型，划分型， 背包， 贪心， 博弈， 除博弈从后向前找规律)

Math

(坐标， 距离， 向量， 斜率， 多边形)

Two pointers

(sliding windows)

Union find

Java built in functions

(comparator, sort, binary search)

**Data Structures and Operations**

**Charcter**:

Character.isDigit()

**String:**

charAt/substring(startIndex, endIndex)

**StringBuilder:**

Append/charAt/deleteCharAt/reverse/insert(I, c)/indexOf(string/c)/setCharAt(I, c)

**ArrayList:**

Insert(index, value), add(), size(), remove()//o(n) time

**Single Linked List:**

**Delete:**

**public** ListNode removeElements(ListNode head, **int** val) {

**if**(head==**null**) **return** **null**;

ListNode dummy = **new** ListNode(0);

dummy.next = head;

ListNode pre = dummy;

ListNode cur = head;

**while**(cur!=**null**){

**if**(cur.val == val){

pre.next = cur.next;

}**else**{

pre = pre.next;

}

cur = cur.next;

}

**return** dummy.next;

}

203. Remove linked list elements

**Reverse:**

Method1: recursive (on space, on time)

**public** ListNode reverseList(ListNode head) {

**return** reverseList(head, **null**);

}

**private** ListNode reverseList(ListNode head, ListNode newHead){

**if**(head==**null**) **return** newHead;

ListNode next = head.next;

head.next = newHead;

**return** reverseList(next, head);

}

Method2: (o(1) space, o(n) time)

**public** ListNode reverseList(ListNode head) {

**if**(head == **null**) **return** **null**;

ListNode newHead = **null**;

**while**(head!=**null**){

ListNode next = head.next;

head.next = newHead;

newHead = head;

head = next;

}

**return** newHead;

}

reverse ([m,n]):

用一个dummy node 来记录开始以及“前一个”

**public** ListNode reverseBetween(ListNode head, **int** m, **int** n) {

ListNode dummy = **new** ListNode(0);

dummy.next = head;

ListNode pre = dummy;

//1->2->3->4->5

**for**(**int** i=0;i<m-1;i++){

pre = pre.next;

}

ListNode start = pre.next;

ListNode then = start.next;

**for**(**int** i=0;i<n-m;i++){

start.next = then.next;

then.next = pre.next;

pre.next = then;

then = start.next;

}

**return** dummy.next;

}

92.Reverse Linked List II

**Merge**

(k sorted lists)

**public** ListNode mergeKLists1(List<ListNode> lists) {

**if**(lists==**null** || lists.size()==0) **return** **null**;

ListNode Dummy = **new** ListNode(0);

PriorityQueue<ListNode> pq = **new** PriorityQueue<>(lists.size(),

**new** Comparator<ListNode>(){

@Override

**public** **int** compare(ListNode l1, ListNode l2){

**return** l1.val - l2.val;

}

});

**for**(ListNode l:lists){

**if**(l!=**null**)

pq.offer(l);

}

ListNode head = Dummy;

**while**(!pq.isEmpty()){

ListNode node = pq.poll();

**if**(node.next!=**null**){

pq.offer(node.next);

}

head.next = node;

head = head.next;

}

**return** Dummy.next;

}

Sort: (insertation sort)

**public** ListNode insertionSortList(ListNode head) {

**if**(head == **null**) **return** head;

ListNode dummy = **new** ListNode(0);

//dummy.next = head;

ListNode pre = dummy;//start of the sorted list

ListNode cur = head;//node need to be moved

**while**(cur!=**null**){

ListNode next = cur.next;//next node will be inserted

**while**(pre.next!=**null** && pre.next.val<cur.val){

//find place to insert

pre = pre.next;

}

//insert cur to pre and pre.next

cur.next = pre.next;

pre.next = cur;

cur = next;

pre = dummy;

}

**return** dummy.next;

}

147. insertation sort

**Double Linked List:**

**Stack:**

**Queue:**

**PriorityQueue:**

**Heap/Deque:**

**HashSet:**

keySet() / values()/ add/ put/ getOrDefault

**TreeSet:**

Same with HashSet, only contains unique value.

Operations:

addAll/ add/ Ceiling closest >= / floor closest <= / higher closest > / lower closest < / headset ( o, inclusive) <(=) smaller objects / tailSet (0, inclusive(true/fase)) >(=) larger objects/first/ last

QS: Max Sum of Rectangle No Larger Than K – 363

//find the max subarray no more than k ->O(nlogn) time

TreeSet<Integer> set = **new** TreeSet<>();

set.add(0);

**int** curSum =0;

**for**(**int** l=0;l<m;l++){

curSum += col[l];

Integer ceiling = set.ceiling(curSum-k);

//System.out.println(curSum + ", c:" + ceiling);

**if**(ceiling!=**null**){

maxSum = Math.*max*(maxSum, curSum-ceiling);

}

set.add(curSum);

}

**Tree:**

Iterator:

Remove node:

Insert node:

Bst:

Trie Tree:

**Graph:**

**Undirect Graph**

**Direct Graph**

Graph storage:

Graph iterator (shortest path):

Weight Graph:

Java Build in functions:

Sort:

**Algorithms (use data structure wisely):**

Sort:

Two pointers/Sliding Windows:

Recursive/Backtracking/ DFS:

BFS:

Dynamic programming (back page/gready):

Union Find:

KMP:

Virtual indexing:

**Math:**

1. 多边形 (Convex 凸/ concave 凹 polygon)

Check cross product/anti-clock/clock wise

//check the second line is clock wise or anti-clock wise

**private** **int** check(Point p1, Point p2, Point p3){

**return** (p2.y-p1.y)\*(p3.x-p2.x)-(p3.y-p2.y)\*(p2.x-p1.x);

}

If(check()>0) clock wise

QS: Erect the Fence l-578/ Convex Polygon l-469 (Hash, Math)

Reservoir sampling:

**Topic:**

**Palindrome:**

**judge palindrome:**

**find all palindrome array in a string:**