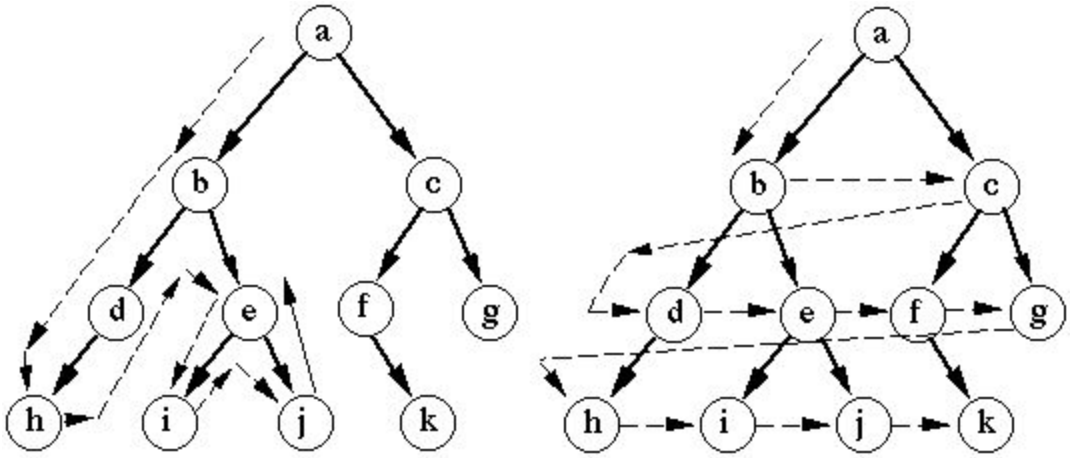
Chapter 1 设计问题

根据题目要求，找到最合适的数据结构来处理或优化，常见数据结构：Graph，Stack，Queue，Tire，LinkedList

Graph

Graph Traverse 算法
BFS - Breadth-First Search
DFS - Depth-First Search



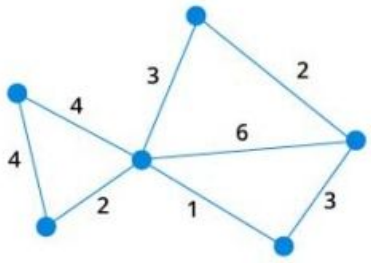
https://github.com/tinkerpop/gremlin/wiki/Depth-First-vs.-Breadth-First

Dijkstra's algorithm 解法

参考
https://www.programiz.com/dsa/dijkstra-algorithm

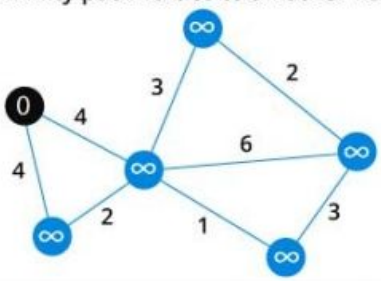
1

Start with a weighted graph



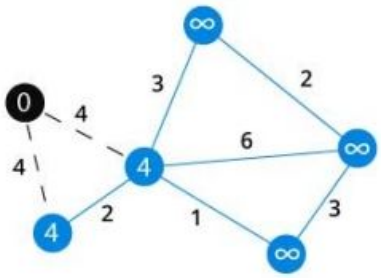
2

Choose a starting vertex and assign infinity path values to all other vertices



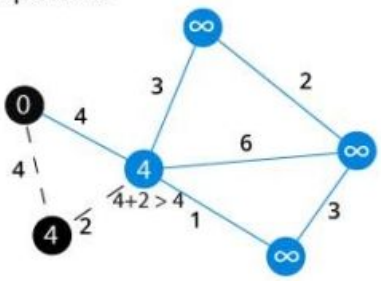
3

Go to each vertex adjacent to this vertex and update its path length



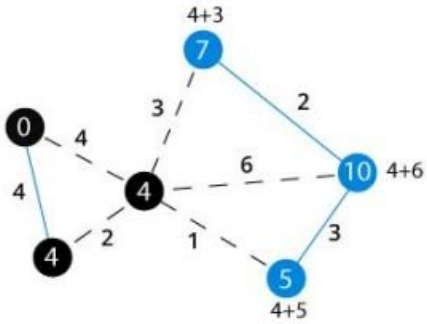
4

If the path length of adjacent vertex is lesser than new path length, don't update it.



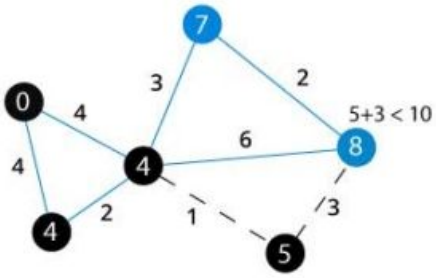
5

Avoid updating path lengths of already visited vertices



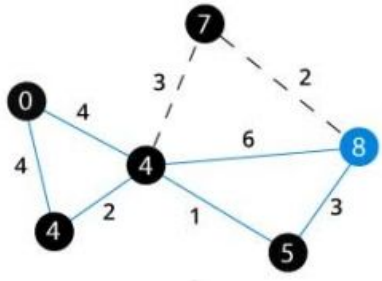
6

After each iteration, we pick the unvisited vertex with least path length. So we chose 5 before 7



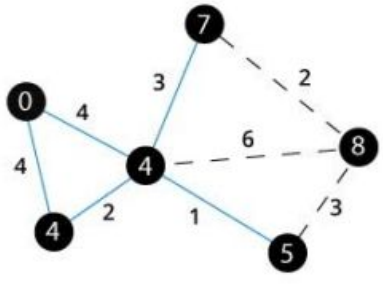
7

Notice how the rightmost vertex has its path length updated twice



8

Repeat until all the vertices have been visited



Number of Islands

Given a 2d grid map of '1's (land) and '0's (water), count the number of islands. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Example 1:

```
Input:
11110
11010
11000
00001
```

Output: 1

Example 2:

```
Input:
11000
11000
00100
00011
```

Output: 3

```
public int numIslands(char[][] grid) {
    int count=0;
    for(int i=0;i<grid.length;i++)
        for(int j=0;j<grid[0].length;j++){
            if(grid[i][j]=='1'){
                dfsFill(grid,i,j);
                count++;
            }
        }
    return count;
}
private void dfsFill(char[][] grid,int i, int j){
    if(i>=0 && j>=0 && i<grid.length && j<grid[0].length&&grid[i][j]=='1'){
        grid[i][j]='0';
        dfsFill(grid, i + 1, j);
        dfsFill(grid, i - 1, j);
        dfsFill(grid, i, j + 1);
        dfsFill(grid, i, j - 1);
    }
}
```

Max Area of Island

Given a non-empty 2D array grid of 0's and 1's, an island is a group of 1's (representing land) connected 4-directionally (horizontal or vertical.) You may assume all four edges of the grid are surrounded by water.

Find the maximum area of an island in the given 2D array. (If there is no island, the maximum area is 0.)
Example 1:

```
[[0,0,1,0,0,0,0,1,0,0,0,0,0],
 [0,0,0,0,0,0,0,1,1,1,0,0,0],
 [0,1,1,0,1,0,0,0,0,0,0,0,0],
 [0,1,0,0,1,1,0,0,1,0,1,0,0],
 [0,1,0,0,1,1,0,0,1,1,1,0,0],
 [0,0,0,0,0,0,0,0,0,0,1,0,0],
 [0,0,0,0,0,0,0,1,1,1,0,0,0],
 [0,0,0,0,0,0,0,1,1,0,0,0,0]]
```

Given the above grid, return 6. Note the answer is not 11, because the island must be connected 4-directionally.
Example 2:

```
[[0,0,0,0,0,0,0,0]]
```

Given the above grid, return 0.
Note: The length of each dimension in the given grid does not exceed 50.

```
public int maxAreaOfIsland(int[][] grid) {
    int max_area = 0;
    for(int i = 0; i < grid.length; i++)
        for(int j = 0; j < grid[0].length; j++)
            if(grid[i][j] == 1) max_area = Math.max(max_area, AreaOfIsland(grid, i,
j));
    return max_area;
}

public int AreaOfIsland(int[][] grid, int i, int j){
    if( i >= 0 && i < grid.length && j >= 0 && j < grid[0].length && grid[i][j] == 1){
        grid[i][j] = 0;
        return 1 + AreaOfIsland(grid, i+1, j) + AreaOfIsland(grid, i-1, j) + AreaOfIsland(grid, i, j-1)
+ AreaOfIsland(grid, i, j+1);
    }
     return 0;
}
```

Surrounded Regions
Given a 2D board containing 'X' and 'O' (the letter O), capture all regions surrounded by 'X'.

A region is captured by flipping all 'O's into 'X's in that surrounded region.

Example:

```
X X X X
X O O X
X X O X
X O X X
```

After running your function, the board should be:

```
X X X X
X X X X
X X X X
X O X X
```

Explanation:

Surrounded regions shouldn't be on the border, which means that any 'O' on the border of the board are not flipped to 'X'. Any 'O' that is not on the border and it is not connected to an 'O' on the border will be flipped to 'X'. Two cells are connected if they are adjacent cells connected horizontally or vertically.

```
class Point {
 int x;
 int y;
 Point(int x, int y) {
  this.x = x;
  this.y = y;
 }
}


public void solve(char[][] board) {
 if (board == null || board.length == 0)
  return;

 int rows = board.length, columns = board[0].length;
 int[][] direction = { { -1, 0 }, { 1, 0 }, { 0, 1 }, { 0, -1 } };

 for (int i = 0; i < rows; i++) {
  for (int j = 0; j < columns; j++) {
   if ((i == 0 || i == rows - 1 || j == 0 || j == columns - 1) && board[i][j] ==
'O') {
    Queue <Point> queue = new LinkedList<>();
    board[i][j] = 'B';
    queue.offer(new Point(i, j));

    while (!queue.isEmpty()) {
     Point point = queue.poll();
     for (int k = 0; k < 4; k++) {
```

```
      int x = direction[k][0] + point.x;
      int y = direction[k][1] + point.y;
      if (x >= 0 && x < rows && y >= 0 && y < columns && board[x][y] == 'O') {
       board[x][y] = 'B';
       queue.offer(new Point(x, y));
      }
     }
    }
   }
  }
 }

 for (int i = 0; i < rows; i++) {
  for (int j = 0; j < columns; j++) {
   if (board[i][j] == 'B')
    board[i][j] = 'O';
   else if (board[i][j] == 'O')
    board[i][j] = 'X';
  }
 }

}
```

Friend Circles

There are N students in a class. Some of them are friends, while some are not. Their friendship is transitive in nature. For example, if A is a direct friend of B, and B is a direct friend of C, then A is an indirect friend of C. And we defined a friend circle is a group of students who are direct or indirect friends.

Given a N*N matrix M representing the friend relationship between students in the class. If M[i][j] = 1, then the ith and jth students are direct friends with each other, otherwise not. And you have to output the total number of friend circles among all the students.

Example 1:

```
Input:
[[1,1,0],
 [1,1,0],
 [0,0,1]]
Output: 2

Explanation:
The 0th and 1st students are direct friends, so they are in a friend circle.
The 2nd student himself is in a friend circle. So return 2.
```

Example 2:

```
Input:
```

```
[[1,1,0],
 [1,1,1],
 [0,1,1]]
Output: 1

Explanation:
The 0th and 1st students are direct friends, the 1st and 2nd students are direct
friends, so the 0th and 2nd students are indirect friends. All of them are in the
same friend circle, so return 1.
```

Note:
N is in range [1,200].
M[i][i] = 1 for all students.
If M[i][j] = 1, then M[j][i] = 1.

```
public class Solution {
    public void dfs(int[][] M, int[] visited, int i) {
        for (int j = 0; j < M.length; j++) {
            if (M[i][j] == 1 && visited[j] == 0) {
                visited[j] = 1;
                dfs(M, visited, j);
            }
        }
    }
    public int findCircleNum(int[][] M) {
        int[] visited = new int[M.length];
        int count = 0;
        for (int i = 0; i < M.length; i++) {
            if (visited[i] == 0) {
                dfs(M, visited, i);
                count++;
            }
        }
        return count;
    }
}
```

Flood Fill
An image is represented by a 2-D array of integers, each integer representing the pixel value of the image (from 0 to 65535).

Given a coordinate (sr, sc) representing the starting pixel (row and column) of the flood fill, and a pixel value newColor, "flood fill" the image.

To perform a "flood fill", consider the starting pixel, plus any pixels connected 4-directionally to the starting pixel of the same color as the starting pixel, plus any pixels connected 4-directionally to those pixels (also

with the same color as the starting pixel), and so on. Replace the color of all of the aforementioned pixels with the newColor.

At the end, return the modified image.
Example 1:

```
Input:
image = [[1,1,1],[1,1,0],[1,0,1]] sr = 1, sc = 1, newColor = 2
Output: [[2,2,2],[2,2,0],[2,0,1]]

Explanation:
From the center of the image (with position (sr, sc) = (1, 1)), all pixels
connected by a path of the same color as the starting pixel are colored with the
new color. Note the bottom corner is not colored 2, because it is not
4-directionally connected to the starting pixel.
```

Note:
The length of image and image[0] will be in the range [1, 50].
The given starting pixel will satisfy 0 <= sr < image.length and 0 <= sc < image[0].length.
The value of each color in image[i][j] and newColor will be an integer in [0, 65535].

```
class Solution {
    public int[][] floodFill(int[][] image, int sr, int sc, int newColor) {
        if (image[sr][sc] == newColor) return image;
        fill(image, sr, sc, image[sr][sc], newColor);
        return image;
    }

    private void fill(int[][] image, int sr, int sc, int color, int newColor) {
        if (sr < 0 || sr >= image.length || sc < 0 || sc >= image[0].length ||
image[sr][sc] != color) return;
        image[sr][sc] = newColor;
        fill(image, sr + 1, sc, color, newColor);
        fill(image, sr - 1, sc, color, newColor);
        fill(image, sr, sc + 1, color, newColor);
        fill(image, sr, sc - 1, color, newColor);
    }
}
```

Making A Large Island
In a 2D grid of 0s and 1s, we change at most one 0 to a 1. After, what is the size of the largest island? (An island is a 4-directionally connected group of 1s).

Example 1:

```
Input: [[1, 0], [0, 1]]
Output: 3
```

```
Explanation: Change one 0 to 1 and connect two 1s, then we get an island with area
= 3.
```

Example 2:

```
Input: [[1, 1], [1, 0]]
Output: 4
Explanation: Change the 0 to 1 and make the island bigger, only one island with
area = 4.
```

Example 3:

```
Input: [[1, 1], [1, 1]]
Output: 4
Explanation: Can't change any 0 to 1, only one island with area = 4.
```

Notes:
1 <= grid.length = grid[0].length <= 50.
0 <= grid[i][j] <= 1.

```
class Solution {
    public int largestIsland(int[][] grid) {
        int max = 0, m = grid.length, n = grid[0].length;
        boolean hasZero = false; //To check if there is any zero in the grid
        for(int i = 0; i < grid.length; i++){
            for(int j = 0;j<grid[0].length;j++){
                if(grid[i][j] == 0){
                    grid[i][j] = 1;
                    max = Math.max(max,dfs(i,j,grid,new boolean[m][n]));
                    if(max == m*n) return max;
                    grid[i][j] = 0;
                    hasZero = true;
                }
            }
        }
        return hasZero?max:m*n;
    }
    private int dfs(int i, int j, int[][] grid,boolean[][] visited){
        if(i < 0 || j < 0 || i >= grid.length || j >= grid[0].length || grid[i][j]
== 0||visited[i][j]) return 0;
        visited[i][j] = true;
        int result =
1+dfs(i-1,j,grid,visited)+dfs(i+1,j,grid,visited)+dfs(i,j+1,grid,visited)+dfs(i,j-1
,grid,visited);
        return result;
    }
```

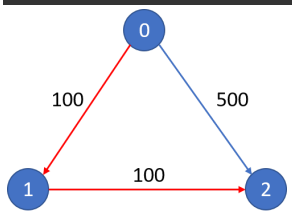
```
}
```

[Cheapest Flights Within K Stops]
There are n cities connected by m flights. Each fight starts from **city u** and arrives at **v** with a **price w**.

Now given all the cities and fights, together with starting city **src** and the destination **dst**, your task is to find the cheapest price from src to dst with up to **k stops**. If there is no such route, output -1.

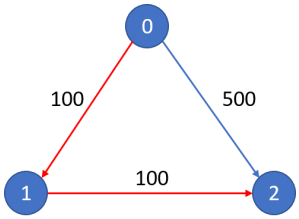
Examples:

```
Input:
n = 3, edges = [[0,1,100],[1,2,100],[0,2,500]]
src = 0, dst = 2, k = 1
Output: 200
```



The cheapest price from city 0 to city 2 with at most 1 stop costs 200, as marked red in the picture.

```
Input:
n = 3, edges = [[0,1,100],[1,2,100],[0,2,500]]
src = 0, dst = 2, k = 0
Output: 500
```



The cheapest price from city 0 to city 2 with at most 0 stop costs 500, as marked blue in the picture.

Note:

- The number of nodes n will be in range [1, 100], with nodes labeled from 0 to n - 1.
- The size of flights will be in range [0, n * (n - 1) / 2].
- The format of each flight will be (src, dst, price).
- The price of each flight will be in the range [1, 10000].
- k is in the range of [0, n - 1].
- There will not be any duplicated flights or self cycles.

```
public int findCheapestPrice(int n, int[][] flights, int src, int dst, int k) {
    // 建立 city s 到 city t 的price map
    Map<Integer, Map<Integer, Integer>> prices = new HashMap<>();
```

```
    for (int[] f : flights) {
        if (!prices.containsKey(f[0])) prices.put(f[0], new HashMap<>());
        prices.get(f[0]).put(f[1], f[2]);
    }

    // 用Dijkstra's algorithm来寻找最短路径
    Queue<int[]> pq = new PriorityQueue<>((a, b) -> (Integer.compare(a[0], b[0])));
    pq.add(new int[] {0, src, k + 1});
    while (!pq.isEmpty()) {
        int[] top = pq.remove();
        int price = top[0];
        int city = top[1];
        int stops = top[2];

        if (city == dst) return price;
        if (stops > 0) {
            Map<Integer, Integer> adj = prices.getOrDefault(city, new HashMap<>());
            for (int a : adj.keySet()) {
                pq.add(new int[] {price + adj.get(a), a, stops - 1});
            }
        }
    }
    return -1;
}
```

Stack

[Basic Calculator II]
Implement a basic calculator to evaluate a simple expression string.

The expression string contains only non-negative integers, +, -, *, / operators and empty spaces . The integer division should truncate toward zero.

Examples:

```
Input: "3+2*2"
Output: 7

Input: " 3/2 "
Output: 1

Input: " 3+5 / 2 "
Output: 5
```

Stack 解法
总共5个情况需要考虑

```java
public int calculate(String s) {
    if(s==null || s.length() ==0) return 0;
    Stack<Integer> stack = new Stack<Integer>();
    int num = 0;
    char sign = '+';

    for(int i=0 ; i < s.length(); i++){
        if(Character.isDigit(s.charAt(i))){
            num = num*10+s.charAt(i)-'0';
        }

        if((!Character.isDigit(s.charAt(i)) && ' '!=s.charAt(i)) ||
i==s.length()-1){
            if(sign=='-'){
                stack.push(-num);
            }
            if(sign=='+'){
                stack.push(num);
            }
            if(sign=='*'){
                stack.push(stack.pop()*num);
            }
            if(sign=='/'){
                stack.push(stack.pop()/num);
            }
            sign = s.charAt(i);
            num = 0;
        }
    }

    int re = 0;
    for(int i : stack){
        re += i;
    }
    return re;
}
```

[Basic Calculator]
Implement a basic calculator to evaluate a simple expression string.

The expression string may contain open (and closing parentheses), the plus + or minus sign -, non-negative integers and empty spaces.

Examples:

```
Input: "1 + 1"
Output: 2
```

```
Input: " 2-1 + 2 "
Output: 3

Input: "(1+(4+5+2)-3)+(6+8)"
Output: 23
```

Stack 解法
总共5个情况需要考虑

```
public int calculate(String s) {
    Stack<Integer> stack = new Stack<Integer>();
    int result = 0;
    int number = 0;
    int sign = 1;
    for(int i = 0; i < s.length(); i++){
        char c = s.charAt(i);
        if(Character.isDigit(c)){
            number = 10 * number + (int)(c - '0');
        }else if(c == '+'){
            result += sign * number;
            number = 0;
            sign = 1;
        }else if(c == '-'){
            result += sign * number;
            number = 0;
            sign = -1;
        }else if(c == '('){
            //we push the result first, then sign;
            stack.push(result);
            stack.push(sign);
            //reset the sign and result for the value in the parenthesis
            sign = 1;
            result = 0;
        }else if(c == ')'){
            result += sign * number;
            number = 0;
            result *= stack.pop(); //stack.pop() is the sign before the parenthesis
            result += stack.pop(); //stack.pop() now is the result calculated
before the parenthesis
        }
    }
    if(number != 0) result += sign * number;
    return result;
}
```

LinkedList

[Moving Average from Data Stream]
Given a stream of integers and a window size, calculate the moving average of all integers in the sliding window.

```
MovingAverage m = new MovingAverage(3);
m.next(1) = 1
m.next(10) = (1 + 10) / 2
m.next(3) = (1 + 10 + 3) / 3
m.next(5) = (10 + 3 + 5) / 3
```

使用LinkedList来优化存储和搜索

```
public class MovingAverage {
    LinkedList<Integer> queue;
    int size;
    double avg;

    public MovingAverage(int size) {
        this.queue = new LinkedList<Integer>();
        this.size = size;
    }

    public double next(int val) {
        if(queue.size()<this.size){
            queue.offer(val);
            int sum=0;
            for(int i: queue){
                sum+=i;
            }
            avg = (double)sum/queue.size();

            return avg;
        }else{
            int head = queue.poll();
            double minus = (double)head/this.size;
            queue.offer(val);
            double add = (double)val/this.size;
            avg = avg + add - minus;
            return avg;
        }
    }
}
```

LRU Cache

Design and implement a data structure for Least Recently Used (LRU) cache. It should support the following operations: get and put.

get(key) - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.
put(key, value) - Set or insert the value if the key is not already present. When the cache reached its capacity, it should invalidate the least recently used item before inserting a new item.

The cache is initialized with a positive capacity.

Follow up:
Could you do both operations in O(1) time complexity?

Example:

```
LRUCache cache = new LRUCache( 2 /* capacity */ );

cache.put(1, 1);
cache.put(2, 2);
cache.get(1);       // returns 1
cache.put(3, 3);    // evicts key 2
cache.get(2);       // returns -1 (not found)
cache.put(4, 4);    // evicts key 1
cache.get(1);       // returns -1 (not found)
cache.get(3);       // returns 3
cache.get(4);       // returns 4
```

Map + LinkedList
因为 LinkedList 的 Node 查询时间慢，所以我们用 Map 来 improve 查询时间；LinkedList 的添加删除都是 O(1) 时间。每次对某个节点的访问时，我们需要把该 Node 放置最前 - 表示 most recently used，那对应的最后的 Node 就是 least recently used 了。

```
public class LRUCache {
    class DLinkedNode {
        int key;
        int value;
        DLinkedNode pre;
        DLinkedNode post;
    }

    private void addNode(DLinkedNode node) {
        node.pre = head;
        node.post = head.post;

        head.post.pre = node;
        head.post = node;
    }
```

```
private void removeNode(DLinkedNode node) {
    DLinkedNode pre = node.pre;
    DLinkedNode post = node.post;

    pre.post = post;
    post.pre = pre;
}

private void moveToHead(DLinkedNode node) {
    this.removeNode(node);
    this.addNode(node);
}

private DLinkedNode popTail() {
    DLinkedNode res = tail.pre;
    this.removeNode(res);
    return res;
}
```

```
private Hashtable < Integer, DLinkedNode > cache = new Hashtable < Integer, DLinkedNode > ();
private int count;
private int capacity;
private DLinkedNode head, tail;

public LRUCache(int capacity) {
    this.count = 0;
    this.capacity = capacity;

    head = new DLinkedNode();
    head.pre = null;

    tail = new DLinkedNode();
    tail.post = null;

    head.post = tail;
    tail.pre = head;
}

public int get(int key) {
    DLinkedNode node = cache.get(key);
    if (node == null) {
        return -1; // should raise exception here.
    }

    this.moveToHead(node);
```

```
        return node.value;
    }


    public void put(int key, int value) {
        DLinkedNode node = cache.get(key);

        if (node == null) {

            DLinkedNode newNode = new DLinkedNode();
            newNode.key = key;
            newNode.value = value;

            this.cache.put(key, newNode);
            this.addNode(newNode);

            ++count;

            if (count > capacity) {
                // pop the tail
                DLinkedNode tail = this.popTail();
                this.cache.remove(tail.key);
                --count;
            }
        } else {
            // update the value.
            node.value = value;
            this.moveToHead(node);
        }
    }
}
```

Map

LFU Cache
Design and implement a data structure for Least Frequently Used (LFU) cache. It should support the following operations: get and put.

get(key) - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.
put(key, value) - Set or insert the value if the key is not already present. When the cache reaches its capacity, it should invalidate the least frequently used item before inserting a new item. For the purpose of this problem, when there is a tie (i.e., two or more keys that have the same frequency), the least recently used key would be evicted.

Note that the number of times an item is used is the number of calls to the get and put functions for that item since it was inserted. This number is set to zero when the item is removed.

Follow up:
Could you do both operations in O(1) time complexity?

Example:

```
LFUCache cache = new LFUCache( 2 /* capacity */ );

cache.put(1, 1);
cache.put(2, 2);
cache.get(1);       // returns 1
cache.put(3, 3);    // evicts key 2
cache.get(2);       // returns -1 (not found)
cache.get(3);       // returns 3.
cache.put(4, 4);    // evicts key 1.
cache.get(1);       // returns -1 (not found)
cache.get(3);       // returns 3
cache.get(4);       // returns 4
```

通过三个 map 来 maintain 这个 LFU: key <-> {value, freq}; key <-> list iterator; freq <-> list of keys

```
class LFUCache {
    int cap;
    int size;
    int minFreq;
    unordered_map<int, pair<int, int>> m; //key to {value,freq};
    unordered_map<int, list<int>::iterator> mIter; //key to list iterator;
    unordered_map<int, list<int>>  fm;  //freq to key list;
public:
    LFUCache(int capacity) {
        cap=capacity;
        size=0;
    }

    int get(int key) {
        if(m.count(key)==0) return -1;

        fm[m[key].second].erase(mIter[key]);
        m[key].second++;
        fm[m[key].second].push_back(key);
        mIter[key]=--fm[m[key].second].end();

        if(fm[minFreq].size()==0 )
              minFreq++;

        return m[key].first;
    }
```

```
    void put(int key, int value) {
        if(cap<=0) return;

        int storedValue=get(key);
        if(storedValue!=-1)
        {
            m[key].first=value;
            return;
        }

        if(size>=cap )
        {
            m.erase( fm[minFreq].front() );
            mIter.erase( fm[minFreq].front() );
            fm[minFreq].pop_front();
            size--;
        }

        m[key]={value, 1};
        fm[1].push_back(key);
        mIter[key]=--fm[1].end();
        minFreq=1;
        size++;
    }
};
```

Priority queue

[Find Median from Data Stream]

Median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value. So the median is the mean of the two middle value.

For example,
[2,3,4], the median is 3
[2,3], the median is (2 + 3) / 2 = 2.5

Design a data structure that supports the following two operations:

- void addNum(int num) - Add a integer number from the data stream to the data structure.
- double findMedian() - Return the median of all elements so far.

```
addNum(1)
addNum(2)
findMedian() -> 1.5
addNum(3)
findMedian() -> 2
```

思路：max heap + min heap

```
class MedianFinder {
    priority_queue<int> lo;                              // max heap
    priority_queue<int, vector<int>, greater<int>> hi;   // min heap

public:
    // Adds a number into the data structure.
    void addNum(int num)
    {
        lo.push(num);                                    // Add to max heap
        hi.push(lo.top());                               // balancing step
        lo.pop();

        if (lo.size() < hi.size()) {                     // maintain size property
            lo.push(hi.top());
            hi.pop();
        }
    }

    // Returns the median of current data stream
    double findMedian() {
        return lo.size() > hi.size() ? (double) lo.top() : (lo.top() + hi.top()) * 0.5;
    }
};
```

类似题目：
https://leetcode.com/tag/design/

Find Median from Data Stream
https://leetcode.com/problems/backspace-string-compare/