Data Structure and Algorithms

why do we need to learn DSA?

For interview, for job, for money. It is also the basic knowledge of computer science. After this class, I can learn computer vision, Neural network and so on.

Pseudocode

What is pseudocode?

How to write pseudocode?

Examples

Practices

Lecture 1

What is an algorithm?

It is any well defined computational procedure that take any values as input and produces some values as output.

Examples

Sorting

- Problem
  - Input:

    sequence of n numbers $(a_1,a_2,\cdots,a_n)$
  - Output:

    Output: a permutation $(a'_1,a'_2,\cdots,a'_n)$ of the input instance such that $a'_1 \le a'_2 \le \cdots \le a'_n$

Two fundamental issues

correctness

- given correct input, show us correct output

efficiency

- To compare algorithms mainly in terms of running time but also in terms of other factors (e.g., memory requirement, programmer's efforts)

steps to analyze and design an algorithm

Formally define a problem

Clearly describe an algorithm

Prove correctness of the algorithm

Analyze the efficiency of the algorithm

How do we compare algorithms?

Express running time as a function of the input size n (i.e., f(n))

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Compare different functions corresponding to running time

Such an analysis is dependent of machine time, programming style.

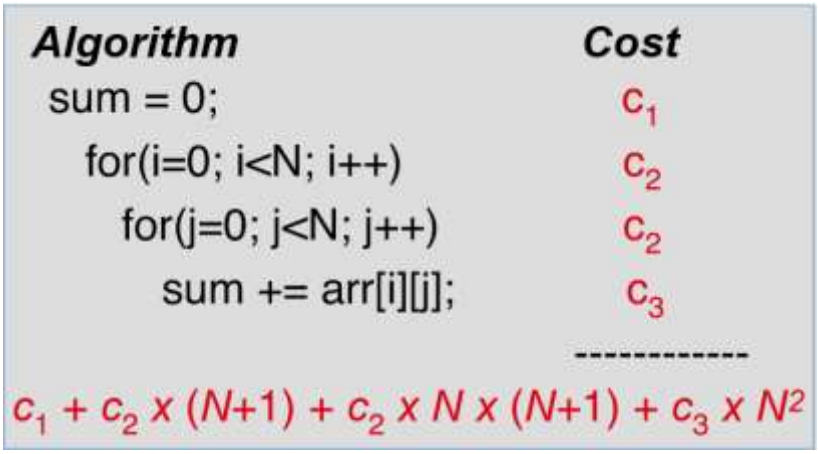
Random-access machine(RAM)

A computation model

- All memory equally expensive to access. We assume write to memory and read from the memory, will use the same time
- No concurrent operations. We assume no parallel happens, the program will execute step by step
- All reasonable instructions take unit time. Except function calls

Example

- Associate a "cost" with each statement
- Find the "total cost by finding the total number of times each statement is executed
- We need to understand why it is N+1. When we execute "for loop", it always need to judgement. if it is true, then go to loop. if not, then exit. one more judgement, that's why N+1



Types of analysis

Worst case

- Provides an upper bound on running time.
- An absolute guarantee that the algorithm would not run longer

Best case

- Provides a lower bound on running time
- Input is the one for which the algorithm runs the fastest

Average case

- Provides a prediction about the running time
- Very useful, but what is average?
  - Random input
  - Real life inputs

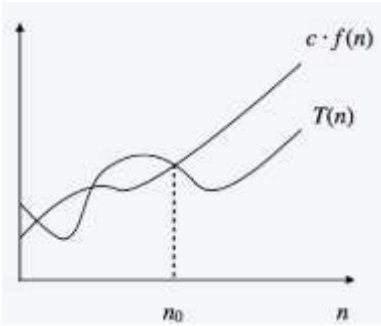
Comparison

$$Lower\ Bound \le Running\ Time \le Upper\ Bound$$

Asymptotic notation

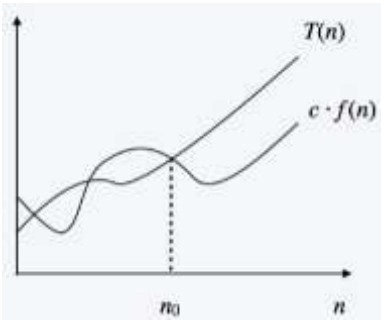
The notation is used to describe the running time of an algorithm

O notation: asymptotic "less than",就是小于等于的意思。这怎么记呢？当别人的东西比你的好时，比你大时，在别人炫耀时，就 oh~回复一下，表示敷衍



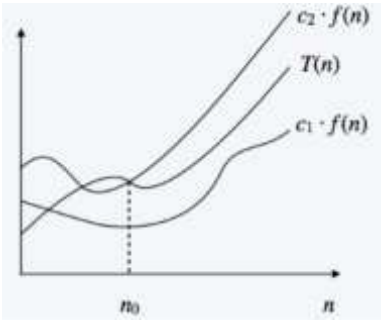
- f(n) = O(g(n)) implies : f(n) ≤ g(n)

Ω notation: asymptotic "greater than"，就是大于等于的意思。怎么记呢？就这么想，这个读音相当于 OMG，有点想要炫耀的意思，oh,my god，我的比你的大诶



- f(n) =Ω (g(n)) implies : f(n) ≥ g(n)

θ notation asymptotic "equality"，就是相等的意思



- f(n) = θ(g(n)) implies : f(n) = g(n)

Little-o，有 little 的，就没有等于，就纯小于<

- f(n) = o(g(n)) implies: f(n) < g(n)

Little-ω,有 little 的，就没有等于，就纯大于>

- f(n) = ω(g(n)) implies: f(n) > g(n)

Asymptotic exercise

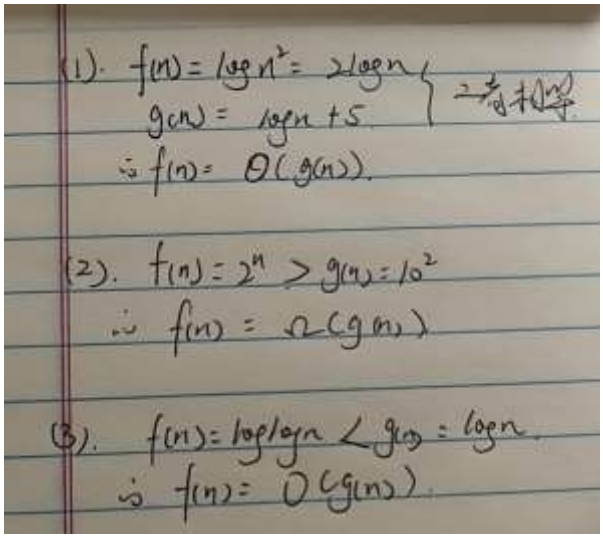
题目

For a pair of functions f(n) and g(n), their relative complexity relation can be:
A. f(n) is O(g(n))
B. f(n) is Ω(g(n))
C. f(n) = Θ(g(n))

For the following pairs, please write the corresponding relation (A, B or C) in the box next to each pair:

(1) $f(n)=\log n^2; g(n)=\log n+5$
(2) $f(n) = 2^n; g(n) = 10n^2$
(3) $f(n) = \log\log n; g(n) = \log n$

solution



Asymptotic properties

Theorem

For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. ■

Transitivity

$$
\begin{array}{llllll}
f(n) = \Theta(g(n)) & \text{and} & g(n) = \Theta(h(n)) & \text{imply} & f(n) = \Theta(h(n)), \\
f(n) = O(g(n)) & \text{and} & g(n) = O(h(n)) & \text{imply} & f(n) = O(h(n)), \\
f(n) = \Omega(g(n)) & \text{and} & g(n) = \Omega(h(n)) & \text{imply} & f(n) = \Omega(h(n)), \\
f(n) = o(g(n)) & \text{and} & g(n) = o(h(n)) & \text{imply} & f(n) = o(h(n)), \\
f(n) = \omega(g(n)) & \text{and} & g(n) = \omega(h(n)) & \text{imply} & f(n) = \omega(h(n)).
\end{array}
$$

Reflexivity

$$
\begin{aligned}
f(n) &= \Theta(f(n)) , \\
f(n) &= O(f(n)) , \\
f(n) &= \Omega(f(n)) .
\end{aligned}
$$

Symmetry

$$f(n) = \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n)) .$$

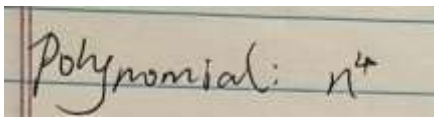
Transpose symmetry

$$
\begin{aligned}
f(n) &= O(g(n)) \text{ if and only if } g(n) = \Omega(f(n)) , \\
f(n) &= o(g(n)) \text{ if and only if } g(n) = \omega(f(n)) .
\end{aligned}
$$

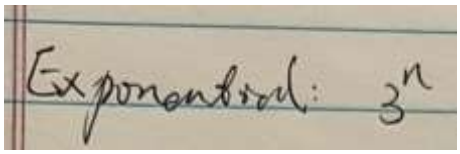
lecture 2

common functions

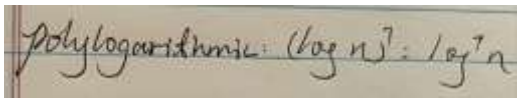
Polynomial: Powers of n, such as n^4



Exponential: A constant (not 1) raised to the power n, such as 3^n



Polylogarithmic: powers of log n, such as (log n)^7 , we will usually write this as log^7n



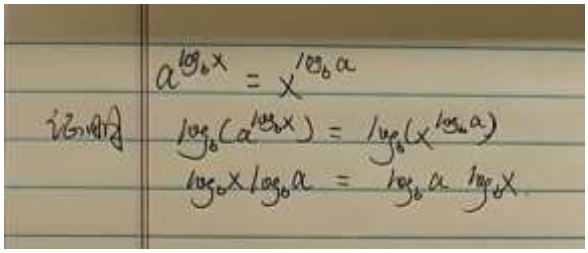
Logarithmic Functions

Formulas

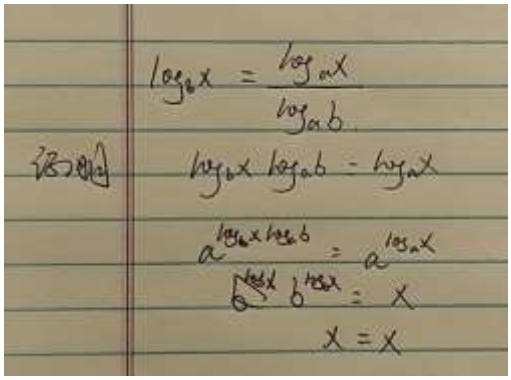
$$\lg^k n = (\lg n)^k$$
$$\lg\lg n = \lg(\lg n)$$

$$\log x^y = y \log x$$
$$\log xy = \log x + \log y$$
$$\log \frac{x}{y} = \log x - \log y$$
$$a^{\log_b x} = x^{\log_b a}$$
$$\log_b x = \frac{\log_a x}{\log_a b}$$

- 一些证明过程（1）



- 一些证明过程（2）

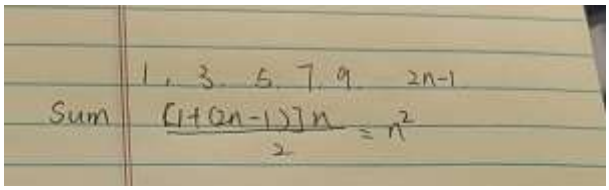


Common Summations

Arithmetic Series

$$\sum_{k=1}^{n} k = 1 + 2 + \ldots + n = \frac{n(n+1)}{2}$$

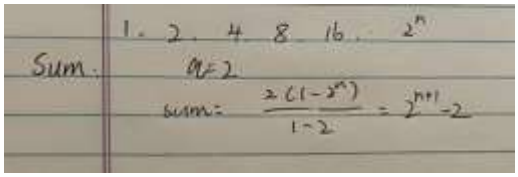
- Example



Geometric Series

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 + \ldots + x^n = \frac{x^{n+1} - 1}{x - 1} (x \neq 1)$$

- Example



- special case, when |q| < 1

$$S_n = \frac{a_1 \times (1 - q^n)}{1 - q} = \frac{a_1 - a_n q}{1 - q} = \frac{a_n q - a_1}{q - 1}, (q \neq 1)$$

$$S_\infty = \frac{a_1}{1 - q} (|q| < 1, n \to \infty)$$

harmonic Series

$$\sum_{k=1}^{n} \frac{1}{k} = 1 + \frac{1}{2} + \ldots + \frac{1}{n} \approx \ln n$$

other important formulas

$$\sum_{k=1}^{n} \log k \approx n \log n - n$$

$$\sum_{k=1}^{n} k^p = 1^p + 2^p + \ldots + n^p \approx \frac{1}{p+1} n^{p+1}$$

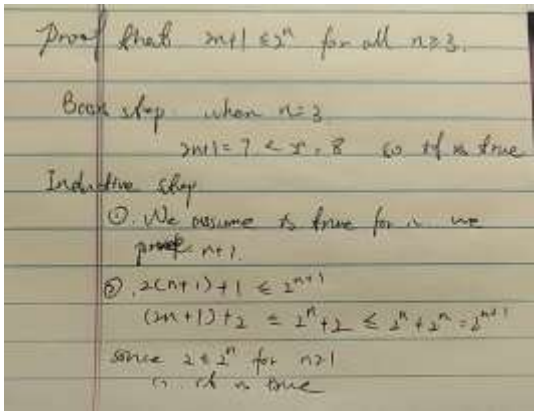
Mathematical Induction

A powerful, rigorous technique for proving that a statement S(n) is true for every natural number n, no matter how large n is

Proof

- Basis step
  - prove that the statement is true for base case (e.g. , n=1),就是说最初始的情况是正确的
- Inductive step
  - assume that statement is true for n (or all integers ≤ n), and then prove that statement is true for n+1. 就是说我们把 n 的那个表达式认为是正确的，然后去证明 n+1 的那个方程式也是正确的就行

Example



- 一般的方法就是在原来函数的基础上，两边加减乘除同一个数，然后进行比较

Recursive Algorithms

A recursive algorithms is an algorithm which calls itself with "smaller" (or simpler) input values, and which obtains the result for the current input, by applying simple operations to the returned value for the smaller input

recursive algorithms 有一个很重要的点是要找到退出的条件

Examples

- Factorial
- binary search

Recurrences and Running Time

- Examples
  - Recursive algorithm that loops through the input to eliminate one item.对于这个公式的理解，就是每次过一次 loop，这个数列都会少一个，所以是 T(n-1)，后面那个 n 是什么意思呢，这代表它的 size 是 n 个，因为每次都要 loop 这个 array 什么的，所以就得+n

$$T(n) = T(n-1) + n$$

  - Recursive algorithm that halves the input in one step，这个就+c 了，意思就是说，因为他每次都分一半，其实这个就是 binary search，每次之找 1 个，所以就是加个 C

$$T(n) = T(n/2) + c$$

  - Recursive algorithm that halves the input but must examine every item in the input，你看，这个+n 了，为什么呢？因为啊，每次都要 examine 这个 input 里面的每一个数，假设这个 input 有 n 个数，那不就得+n 吗

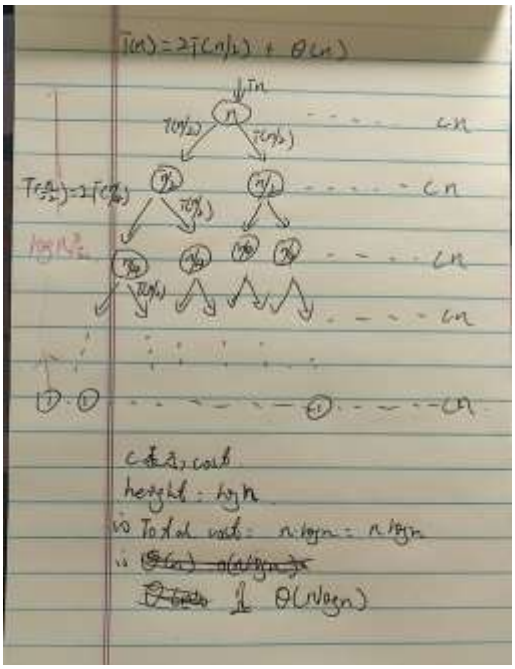
$$T(n) = T(n/2) + n$$

- Recursive algorithm that splits the input into 2 halves and does a constant amount of other work

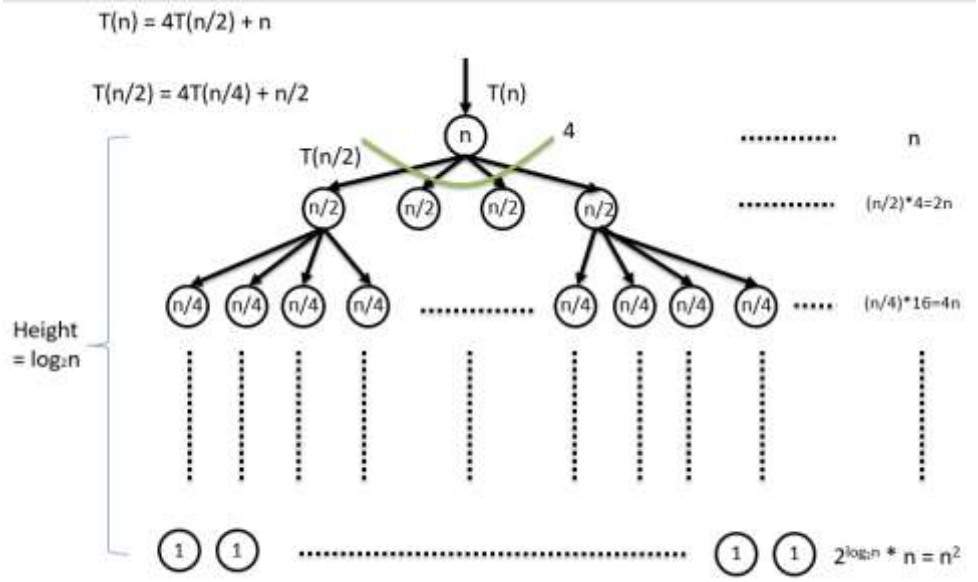
$$T(n) = 2T(n/2) + 1$$

Methods for solving Recurrences

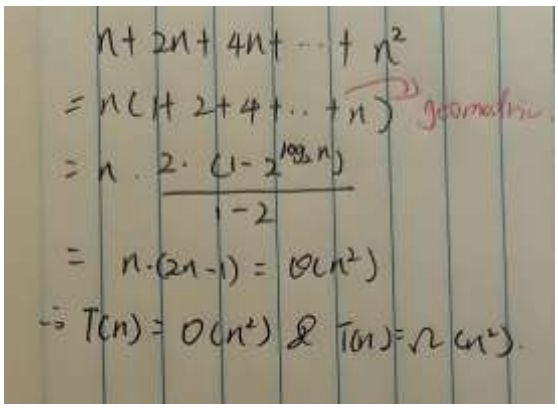
- Iteration method
  - steps
    - Sum up the costs of all levels
    - Each node represents the cost incurred at various levels of recursion
    - 简单来说，就是画树，画完之后把每一层加起来就行了
  - Examples
    - 这题就是算 T(n) = 2T(n/2) + θ(n),这东西应该怎么看呢？很简单，首先看 θ(n)，这代表它的 input size 是 n，然后每次要分层画树啊，然后就分成两边的树，就是 2T(n)，以此下去，直至分到 1 为止，每一层的 cost 怎么计算呢？就是很简单的加起来，把里面的值加起来就行了，也就是（n/2）+ (n/2)=n，就 ok 啦。然后算出高度来，然后加起来就行啦。这题的高度是 logn，然后每一层的 cost 又是一样的，所以就是 θ(nlogn)



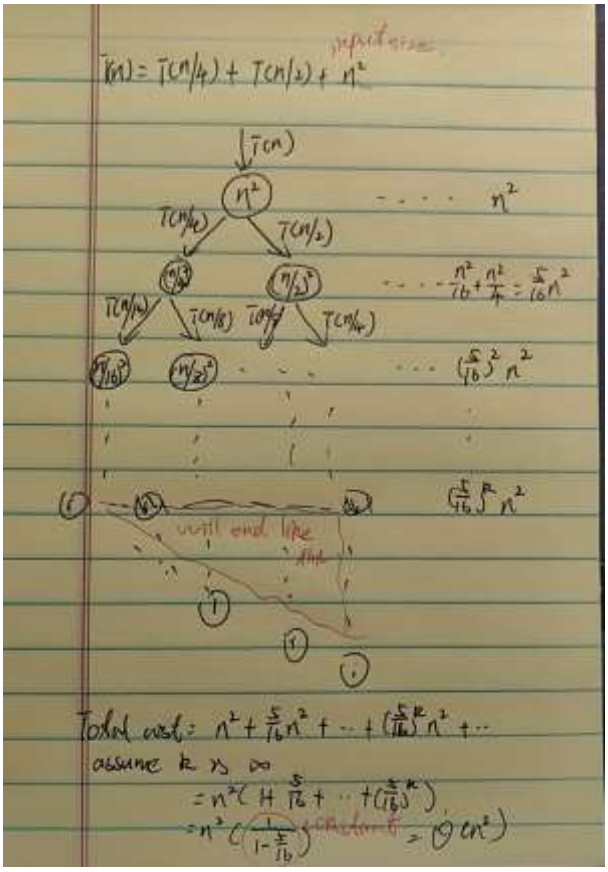
    - 这个 Example 就是 T(n) = 4T(n/2) + n. 我们一定要弄清楚他的高度，别弄错了，其次就是把所有的 cost 加起来。我们一定要明白那个圈圈里面代表的是什么意思。在方程式中，那个 n 代表的就是 input size，也就是总共有多少个。然后圈圈里面的数就是根据那个方程式，每一个步骤算出来的那个 n，你看 T(n) 和 T(n/2)，结尾的那个数就代表 T(n) 和 T(n/2) 还有多少数



- 相加的时候一定要小心，等比等差的求和公式中，那个 n 是代表有多少项，千万别弄错了



- T(n) = T(n/4) + T(n/2) + n^2，这个 example 要注意的就是最后以类似三角形似地结束，因为除数比较大的会先到 1，除数较小的会后到 1，最后我们直接把 n 想成∞，这样 k 的值也会∞，这样我们就可以使用等比数列的求和公式



- Substitution method
  - steps
    - Guess a solution
    - Use induction to prove that the solution works
  - Examples
- Master method
  - 这个方法就比较有意思了，很简单，就是比较大小即可，没难度，但要满足一些条件

需要满足的条件

$T(n) = aT(n/b) + f(n)$

Where $a \geq 1$, $b>1$, and $f(n) >0$

- 那谁跟谁比较大小呢？

f(n) 要跟这个数相比 $n^{\log_b a}$

- 分情况讨论
  - case 1
    - 如果 f(n)比较小

若 $f(n) = O\left(n^{\log_b a - \varepsilon}\right), \varepsilon > 0$, 那么 $T(n) = \Theta\left(n^{\log_b a}\right)$

  - case 2

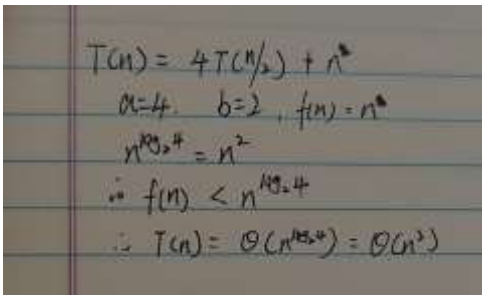
- 如果 f(n)与那个数相等

若 $f(n)=\Theta\left(n^{\log_b a}\right)$，那么 $T(n)=\Theta\left(n^{\log_b a}\log n\right)$

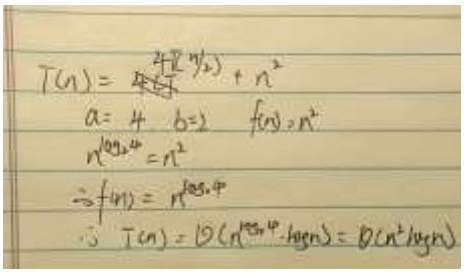
- case 3
  - 如果 f(n) 比较大

(3) 若 $f(n)=\Omega\left(n^{\log_b a+\varepsilon}\right)$，$\varepsilon>0$，且对于某个常数 $c<1$ 和所有充分大的 n 有 $af\left(\frac{n}{b}\right)\le cf(n)$，那么 $T(n)=\Theta(f(n))$.

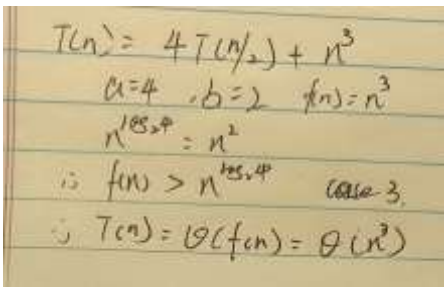
- example
  - T(n) = 4T(n/2) + n，比较之后，这个是属于 case 1 的，就是 f(n) 比较小的那个



  - T(n) = 4T(n/2) + n^2，比较之后，这个是属于 case 2 的，就是两者相等的那个

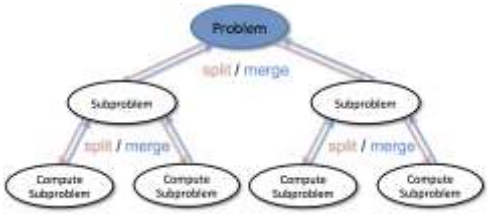


  - T(n) = 4T(n/2) + n^3，比较之后，这个是属于 case 3 的，就是 f(n) 比较大的那个



lecture 3

Divide and Conquer



steps

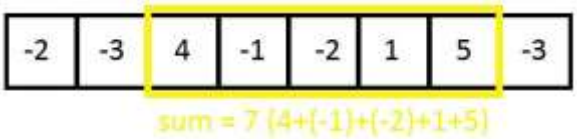
- Divide the problem into a number of sub-problems
- Conquer the sub-problems
  - Solve the sub-problems recursively
  - sub-problem size small enough
  - solve the problems in straightforward manner
- Combine the solutions of the sub-problems

Examples

- fibonnacci number

- binary search

  - 它的运行前提是这个 array 已经 sorted 的
- Maximum subarray



  - 我们用分治法来解，非常简单，首先得明白思路。就直接把一个数组分成三个部分，然后递归运行即可。我们分别计算左边部分，右边部分，还有就是横跨左右的，听起来跟废话一样。分的过程跟那个 mergeSort 都是差不多的意思，这个难点就在于理解横跨那部分即可
    - 如果数组就只有一个数值，直接返回那个数即可。这里为什么要说，因为怕你忘记这个情况
    - 分组

```
int mid = (left + right)/2;          //有没有感觉很熟悉
                                     //接下来就得左右分组了
maxSubArr(arr, left, mid);           //左边
maxSubArr(arr, mid+1, right);        //右边

                                     //分完组之后呢？
```

    - 求和计算，就像是 MergeSort 里的分开之后就要比较然后 merge。进行递归之后，也就会求出左边最大和，右边最大和了

```
int leftVal = arr[mid];
for(int i = mid; i>=1;i--)
        t += arr[i];
        leftVal = max(t, leftVal);

int rightVal = arr[mid+1];
for(int i = mid+1; i<=r;;i++)
        t += arr[i];
        rightVal = max(t, rightVal);
```

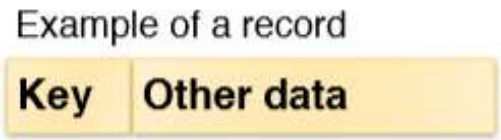
```
/*我们为什么要从mid开始呢？因为
横跨的话,一定有左边最后一个元素，
也一定有右边的第一个元素，对吧。
然后我们就干脆左边取最后一个元
素，然后连续往前加，如果连续往
前加的和大于最后一个元素，取最
大值。如果没有，那么就返回左边
最后一个值。对于右边，那就是取
第一个值，连续往右加，然后同上，
这样我们就会有三个值*/
```

    - 我们弄出左边的最大和，右边的最大和，与中间的最大和，在这三个数中求最大值，就解决啦
- Runtime analysis
    - 其实这个就跟 mergeSort 一样的，都是分开一半，可以得出 T(n) = 2T(n/2) + O(n)
    - 然后用 master theory 得出时间复杂度为 O(nlogn)

The Sorting problem

Structure of Data

- The numbers to be sorted are part of collection of data called a record



- Each record contains a key, which is the value to be sorted
- Noted that when the key must be arranged, the data associated with the key must also be rearranged (time consuming!)
- pointer can be used instead (space consuming!)

Why study sorting algorithms

- Most fundamental problem in algorithm
- widely encountered in practice
- Rich set of classical sorting algorithms using different techniques
- A variety of situations that we can encounter
    - Do we have randomly ordered keys?
    - Are all keys distinct?
    - How large is the set of keys to be ordered?
    - Need guaranteed performance?
- Certain algorithms are better suited to certain situations

Some definitions about sorting

- Internal sort
    - The data to be sorted is all stored in the computer's main memory. 一般来说当数据量小时，可以在内存中存储时，就是 internal sort
        - Bubble sort

    - Insertion sort
    - Quick sort
    - Heap sprt
- External sort
    - Some of the data to be sorted might be stored in some external, slower, device。当数据量大时，内存中存不下，就得借助硬盘
        - External merge sort
- In place sort
    - The amount of extra space required to sort the data is constant with the input size. 举个例子来说明，就是在 sorting 的过程中，你原先有一个 array，然后这个过程就发生在这个 array 中，不用去创造额外的一个 array。

Stability

- A stable sort preserves relatives order of records with equal keys. 举个例子一目了然，就以学生的 GPA 按升序来排序

StuID	GPA
001	3.5
002	3.7
003	3.5
004	3.7
005	3.6
006	3.5

Stable		Unstable	
StuID	GPA	StuID	GPA
001	3.5	003	3.5
003	3.5	006	3.5
006	3.5	001	3.5
005	3.6	005	3.6
002	3.7	004	3.7
004	3.7	002	3.7

各种 sort

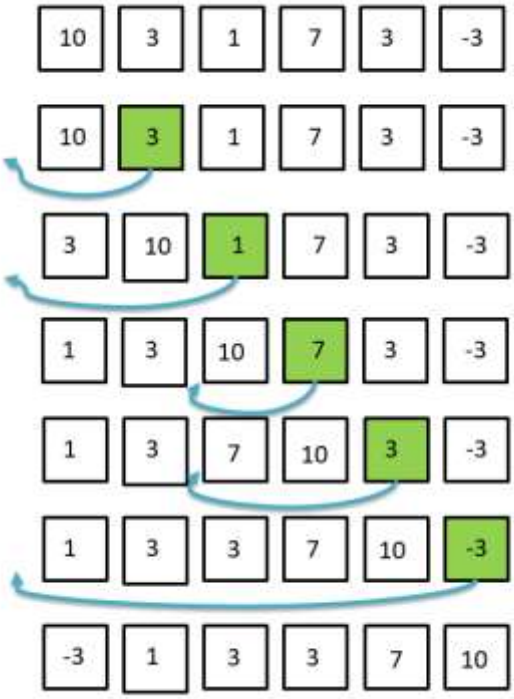
- Insertion sort
    - Basic idea
        - 就像打扑克牌一样，把值放在合适的位置

Insertion sort

A = arr[10, 3, 1, 7, 3, -3]

我们手里有个10，然后3跟10进行对比，很明显，10>3，于是就把3插在10前面。数列就变成了arr[3, 10, 1, 7, 3, -3]。接下来就是1跟10比，很明显，10>1，就把1插在10前面。此时，前面还有数，就接着比，很明显3>1，于是就把1插在3前面，于是arr[1, 3, 10, 7, 3, -3]。接下来我们抽到的牌是7，就跟10比，很明显，10>7，就把7插在10的前面。因为前面还有牌，咱就接着比，没想到，前面的很小，于是arr[1, 3, 7, 10, 3, -3]。接下来我们抽到的牌是3，又一个一个地往前比。没想到前面有一个跟它一样大的，也是3。E=(´o｀*)))唉，停住了，老子不往前走，找到同伴了。于是arr[1, 3, 3, 7, 10, -3]。最后一张牌了，抽到的是-3，照旧，往前比，没想到，一个个这个大，就只能插在最前面了。于是Arr[-3, 1, 3, 3, 7, 10]。没牌了，也就排完了

- 让我们来看看图的解释



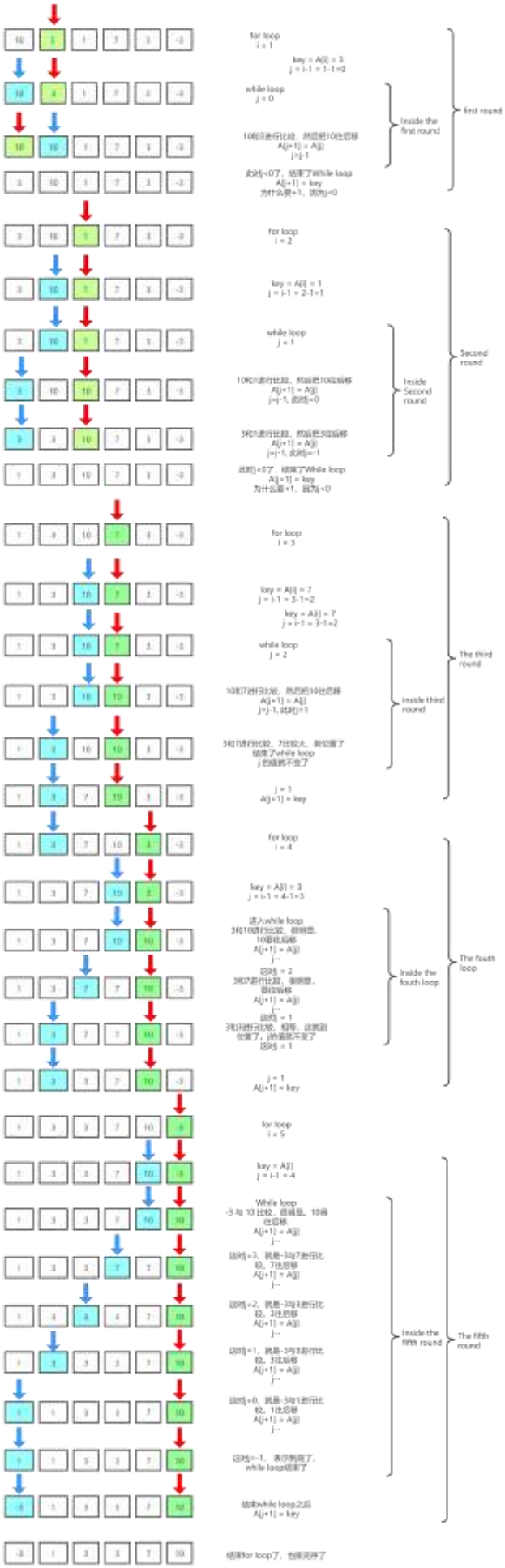
- Pseudocode
  - 要去理解

```
for i=1 to A.length
    key = A[i];
    j = i-1;
    while(j>=0 && A[j] > key)
        A[j+1] = A[j];
        j--;
    A[j+1] = key;
```

- Insertion sort step by step
  - 这是代码运行的结果

```
Inside the 1 round[10, 10, 1, 7, 3, -3]
the 1 round: [3, 10, 1, 7, 3, -3]
Inside the 2 round[3, 10, 10, 7, 3, -3]
Inside the 2 round[3, 3, 10, 7, 3, -3]
the 2 round: [1, 3, 10, 7, 3, -3]
Inside the 3 round[1, 3, 10, 10, 3, -3]
the 3 round: [1, 3, 7, 10, 3, -3]
Inside the 4 round[1, 3, 7, 10, 10, -3]
Inside the 4 round[1, 3, 7, 7, 10, -3]
the 4 round: [1, 3, 3, 7, 10, -3]
Inside the 5 round[1, 3, 3, 7, 10, 10]
Inside the 5 round[1, 3, 3, 7, 7, 10]
Inside the 5 round[1, 3, 3, 3, 7, 10]
Inside the 5 round[1, 3, 3, 3, 7, 10]
Inside the 5 round[1, 1, 3, 3, 7, 10]
the 5 round: [-3, 1, 3, 3, 7, 10]
[-3, 1, 3, 3, 7, 10]
```

- 图解


for loop
i = 1
key = A[i] = 3
j = i-1 = 1-1=0
while loop
j = 0
10和3进行比较，然后把10往后移
A[j+1] = A[j]
j=j-1
此时j<0了，结束了while loop
A[j+1] = key
为什么要+1，因为j=0
Inside the first round
first round
for loop
i = 2
key = A[i] = 1
j = i-1 = 2-1=1
while loop
j = 1
10和1进行比较，然后把10往后移
A[j+1] = A[j]
j=j-1, 此时j=0
3和1进行比较，然后把3往后移
A[j+1] = A[j]
j=j-1, 此时j=-1
此时j<0了，结束了While loop
A[j+1] = key
为什么要+1，因为j=0
Inside Second round
Second round
for loop
i = 3
key = A[i] = 7
j = i-1 = 3-1=2
key = A[i] = 7
j = i-1 = 3-1=2
while loop
j = 2
10和7进行比较，然后把10往后移
A[j+1] = A[j]
j=j-1, 此时j=1
3和7进行比较，7比较大，则位置了
结束了while loop
j 的值就不变了
j = 1
A[j+1] = key
Inside third round
The third round
for loop
i = 4
key = A[i] = 3
j = i-1 = 4-1=3
进入while loop
3和10进行比较，很明显，
10要往后移
A[j+1] = A[j]
j--
此时j = 2
3和7进行比较，很明显，
很往后移
A[j+1] = A[j]
j--
此时j = 1
3和3进行比较，相等，这就到
位置了，j的值就不变了
此时j = 1
j = 1
A[j+1] = key
Inside the fouth loop
The fouth loop
for loop
i = 5
key = A[i]
j = i-1 = 4
While loop
-3 与 10 比较，很明显，10得
往后移
A[j+1] = A[j]
j--
此时j=3，就是-3与7进行比
较，7往后移
A[j+1] = A[j]
j--
此时j=2，就是-3与3进行比
较，3往后移
A[j+1] = A[j]
j--
此时j=1，就是-3与3进行比
较，3往后移
A[j+1] = A[j]
j--
此时j=0，就是-3与1进行比
较，1往后移
A[j+1] = A[j]
j--
此时j=-1，表示到底了，
while loop结束了
结束while loop之后
A[j+1] = key
结束for loop了，也得完序了
Inside the fifth round
The fifth round

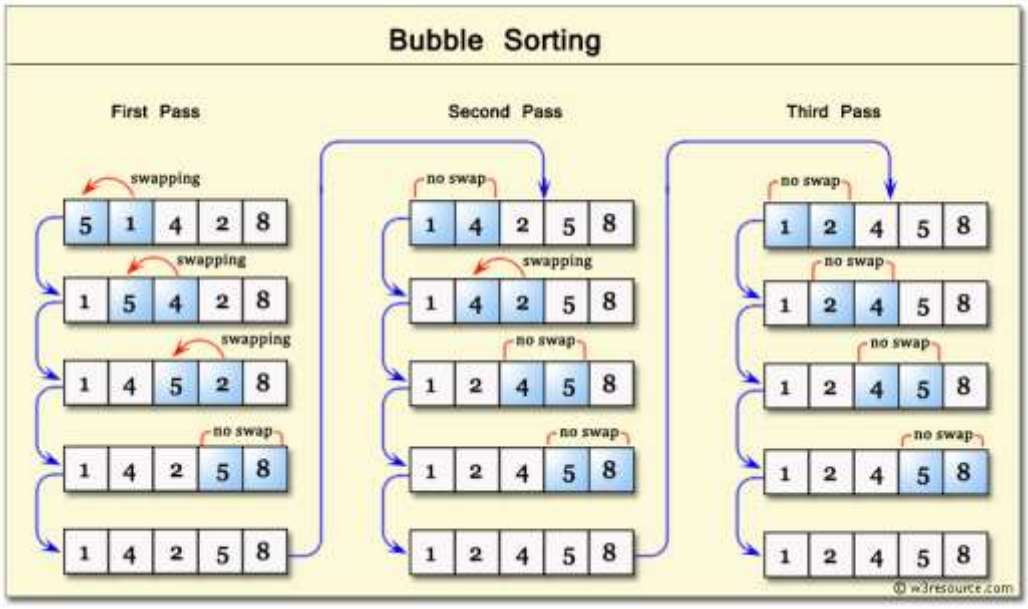
- Proving Loop Invariants
- Run time analysis
  - Best case
    - When the array is almost sorted or sorted, in this case, it does not need to compare a lot. It means it will not go to the while loop.
    - The Running time will be O(n), it is linear time
  - Worst case
    - When the array is reverse sorted. In this case, it has to compare all numbers, every time, it needs to go to while loop
    - It has (n^2)/2 comparisons and exchanges
    - The Running time is O(n^2)

- Bubble sort
  - Basic idea

Bubble sort

这玩意吧，就像泡泡一样，如果是按升序来说的话，就是小的数从尾部不断地往上冒泡，噗噗

  - 让我们来看看图的解释



  - Pseudocode

```
for i to A.length:
    for j = A.length-1 downto 1:
        if A[j]<A[j-1]
            temp = A[j]
            A[j]   = A[j-1]
            A[j-1] = temp
```

Swap

  - Bubble sort step by step

- 这是代码运行结果，没错，就是这么又臭又长

A = {10, 3, 1, 7, 3, -3}

```
Inside 0 round:
[10, 3, 1, 7, -3, 3]
Inside 0 round:
[10, 3, 1, -3, 7, 3]
Inside 0 round:
[10, 3, -3, 1, 7, 3]
Inside 0 round:
[10, -3, 3, 1, 7, 3]
Inside 0 round:
[-3, 10, 3, 1, 7, 3]
it is 0 round:
[-3, 10, 3, 1, 7, 3]
```

```
Inside 1 round:
[-3, 10, 3, 1, 3, 7]
Inside 1 round:
[-3, 10, 3, 1, 3, 7]
Inside 1 round:
[-3, 10, 1, 3, 3, 7]
Inside 1 round:
[-3, 1, 10, 3, 3, 7]
Inside 1 round:
[-3, 1, 10, 3, 3, 7]
it is 1 round:
[-3, 1, 10, 3, 3, 7]
```

```
Inside 2 round:
[-3, 1, 10, 3, 3, 7]
Inside 2 round:
[-3, 1, 10, 3, 3, 7]
Inside 2 round:
[-3, 1, 3, 10, 3, 7]
Inside 2 round:
[-3, 1, 3, 10, 3, 7]
Inside 2 round:
[-3, 1, 3, 10, 3, 7]
it is 2 round:
[-3, 1, 3, 10, 3, 7]
```

```
Inside 3 round:
[-3, 1, 3, 10, 3, 7]
Inside 3 round:
[-3, 1, 3, 3, 10, 7]
Inside 3 round:
[-3, 1, 3, 3, 10, 7]
Inside 3 round:
[-3, 1, 3, 3, 10, 7]
Inside 3 round:
[-3, 1, 3, 3, 10, 7]
it is 3 round:
[-3, 1, 3, 3, 10, 7]
```

```
Inside 4 round:
[-3, 1, 3, 3, 7, 10]
Inside 4 round:
[-3, 1, 3, 3, 7, 10]
Inside 4 round:
[-3, 1, 3, 3, 7, 10]
Inside 4 round:
[-3, 1, 3, 3, 7, 10]
Inside 4 round:
[-3, 1, 3, 3, 7, 10]
it is 4 round:
[-3, 1, 3, 3, 7, 10]
```

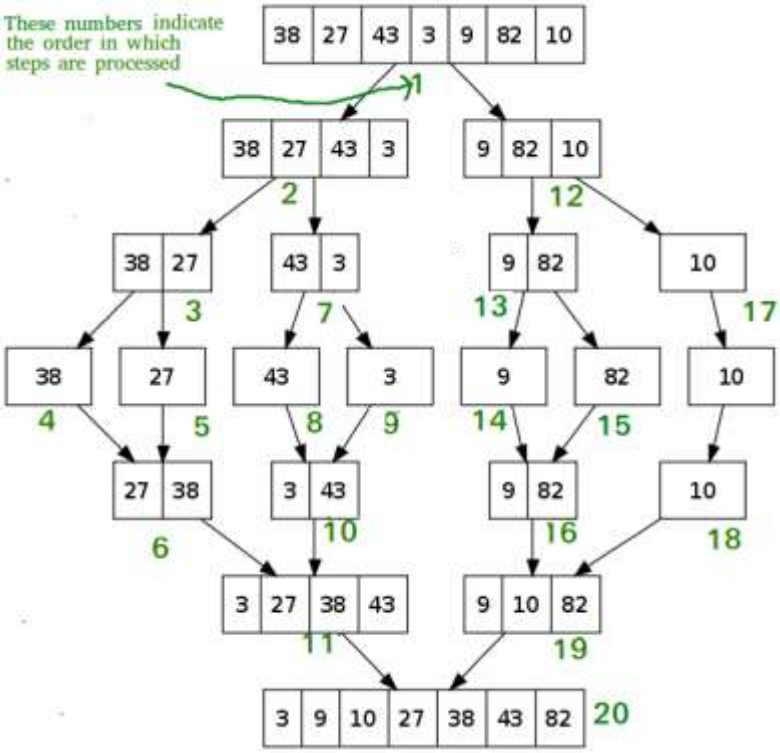
```
Inside 5 round:
[-3, 1, 3, 3, 7, 10]
Inside 5 round:
[-3, 1, 3, 3, 7, 10]
Inside 5 round:
[-3, 1, 3, 3, 7, 10]
Inside 5 round:
[-3, 1, 3, 3, 7, 10]
Inside 5 round:
[-3, 1, 3, 3, 7, 10]
it is 5 round:
[-3, 1, 3, 3, 7, 10]
```

    - 图解，我就不画了，理解就行了
  - Run time analysis
    - It is always O(n^2)
- Merge sort
  - Basic idea

这玩意吧，用的是Divide and conquer的思想

就是把一串数组不断分开，分开后就解决每一个小问题
解决后就把每个小问题组合起来，然后大问题也就解决啦

    - 来看看图解



- Pseudocode。 这个有点复杂，我们得分开讲
  - Merge
    - 就是数组分开之后，我们就得对数进行比较

- 进行数比较的 loop

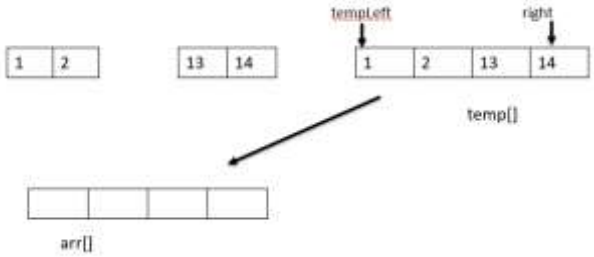
```
While loop => i<=mid and j<=right
        //进行数的比较
        if(arr[i] <= arr[j]) ①
                //把数放进临时数组
                temp[t++] = arr[i++]
        else ②
                temp[t++] = arr[j++]
```

- 当其中一组的数全部完成之后，就得把另一组剩下的数放进临时数组中，也就是 temp[]

- 把剩余的数全部放进 temp 里

```
左边剩余
While loop ⟶ i <= mid
        temp[t++] = arr[i++]

右边剩余
While loop ⟶ j <= right
        temp[t++] = arr[j++]
```

- 比较难理解的一步，就是把 temp 里的数又拷贝回 arr 里

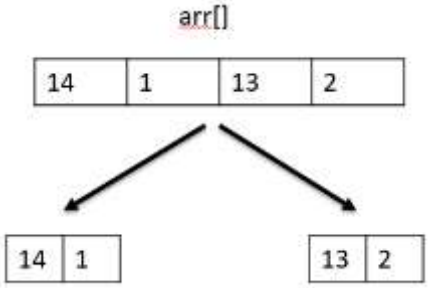


- 循环放回去

```
t =0;
int tempLeft = left

While loop => tempLeft<= right
        arr[tempLeft++] = temp[t++]
```

- Conquer

- Mergesort
  - Mergesort，最难的在于 merge，divide 就很简单了，就纯粹的分开而已



- Mergesort

```
if(left < right)
    int mid = (left + right)/2; //找到中间，分开他们
    mergesort(arr[], left, mid, temp[]); //左边接着分开
    mergesort(arr[], mid+1, right, temp[]); //右边接着分开
    merge(arr[], left, mid, right, temp[]);  //合起来
```

- Divide

- Merge sort step by step
- Running time analysis

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n/2) + cn & \text{if } n > 1 \end{cases}$$

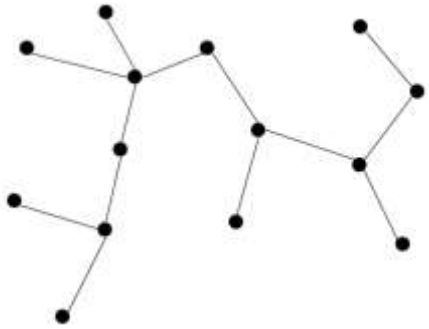
- $T(n) = \theta(n\log n)$

- The running time is good but it requires extra space temp[], so, it does not sort in place

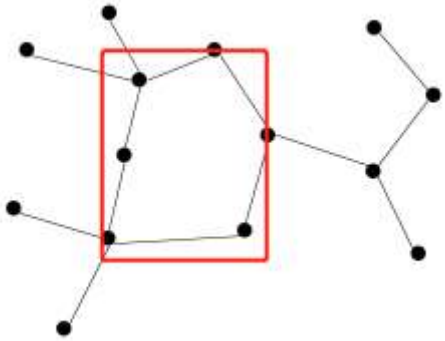
lecture 4

some concepts

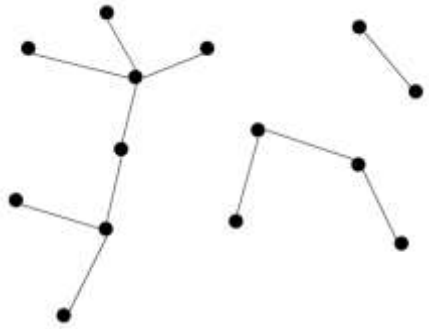
- Tree: Connected, acyclic, undirected graph



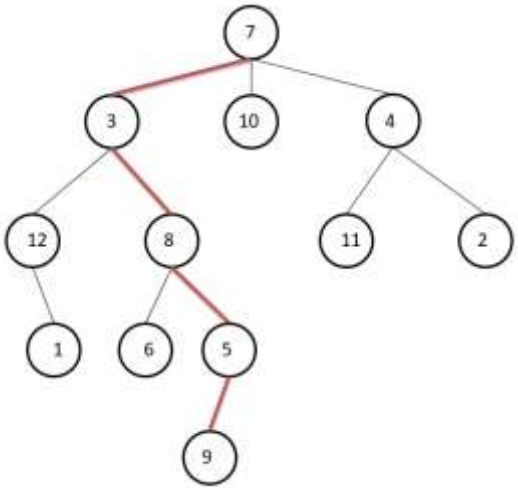
  - This one is not a tree



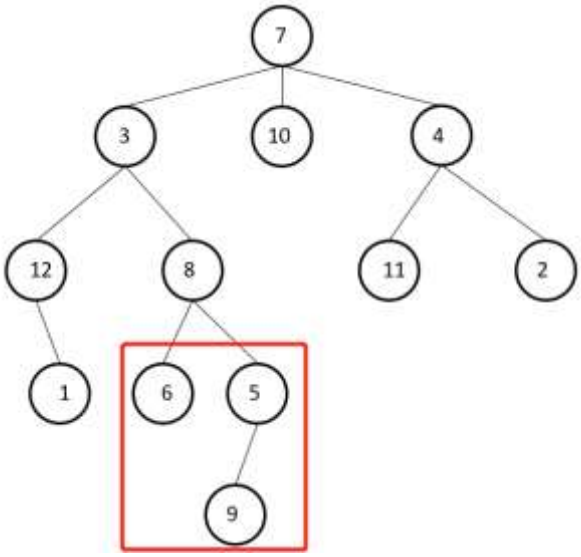
- Forest: acyclic, undirected graph, possibly disconnected



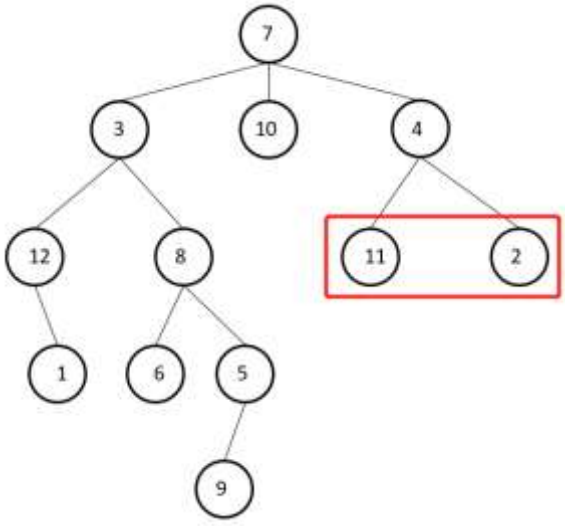
- Rooted Tree: a free tree with special root mode
  - Ancestor of node x: any node on the path from root to x. 假设问 9 的 ancestor，那么看 path，就是 5, 8, 3, 7



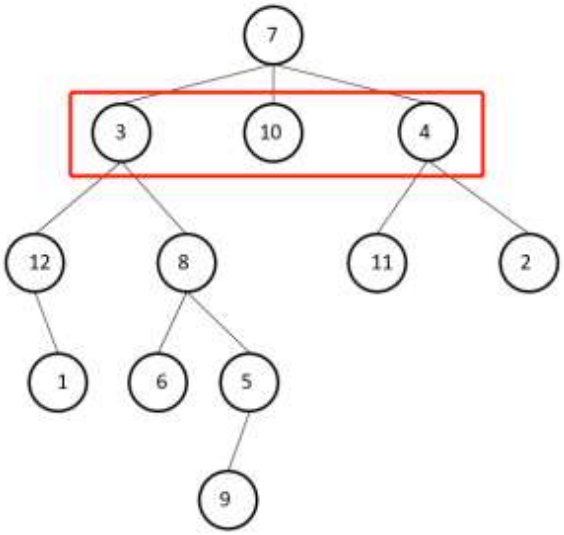
- Descendant of node x: any node with x as its ancestor. 这玩意就是后代的意思。我们假设问 8 的后代，那么就是 6, 5, 9



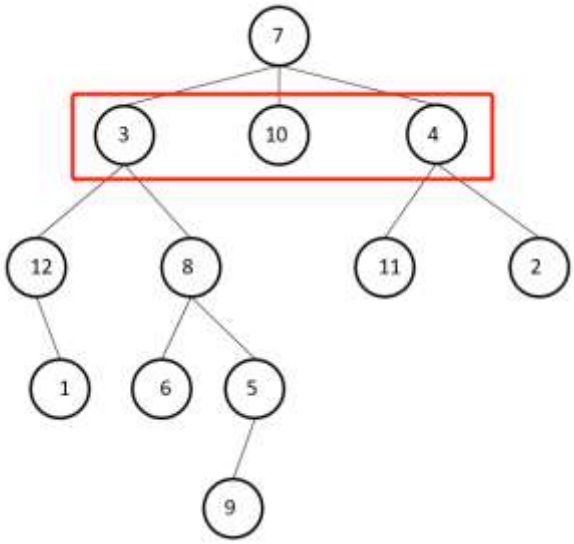
- Parent of node x: node immediately before x on path from root. 就是那个点的前一个就是。 比如说，我们问 11， 或者 2 的 parent 是哪个？那就是 4



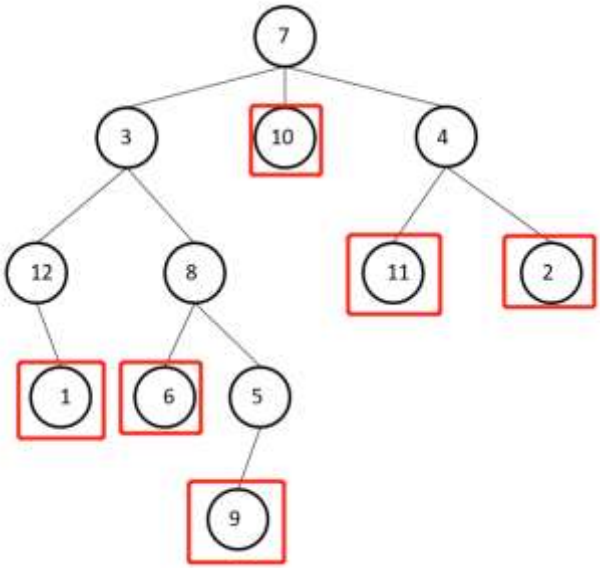
- Child of node x: any node with x as its parent. 7 的孩子有谁，就是 3, 10， 4



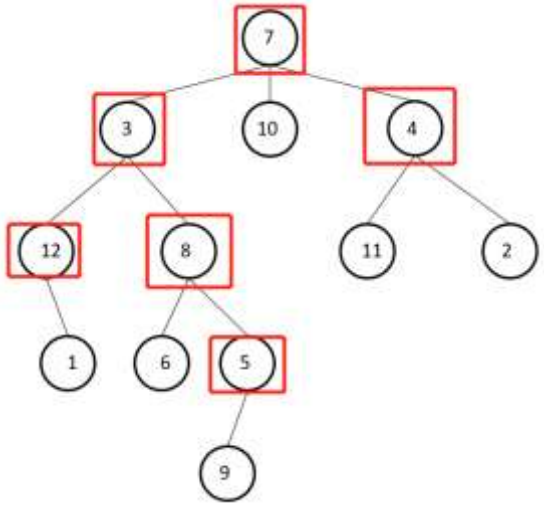
- siblings of node x: nodes sharing parent with x. 就比如说 3， 10, 4 就是 siblings



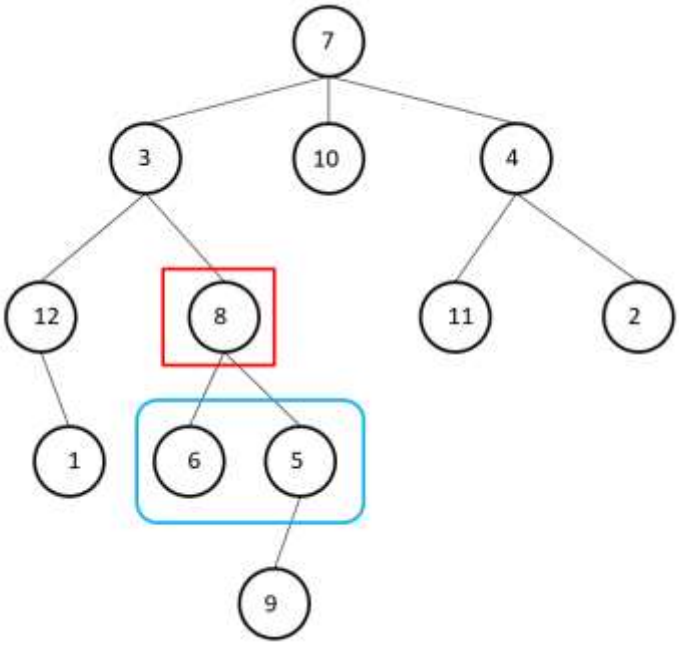
- leaf/ external node： without child. 为什么是 external 呢?就是孩子都生不出来，就干脆赶出去，就是 external 了



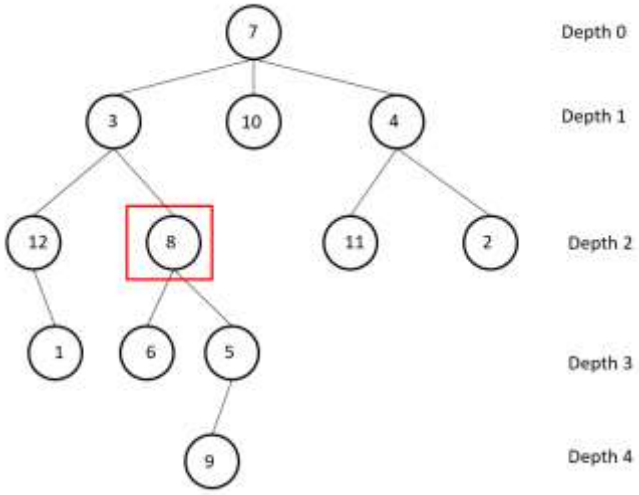
- internal node： with at least one child.生的出来的肯定是家族的一部分啊，那就是 internal 啊



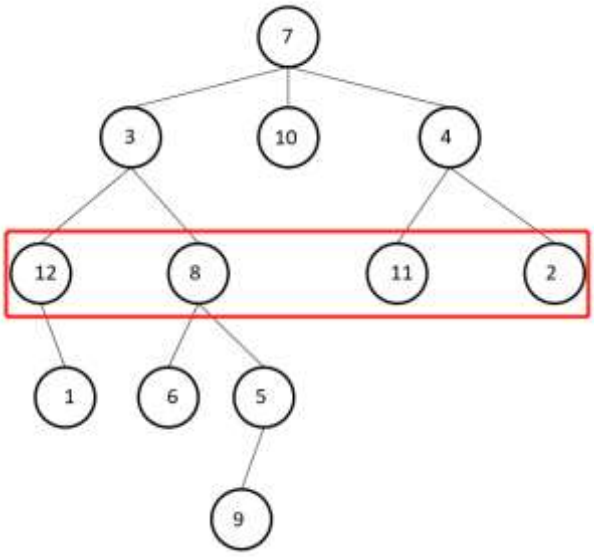
- tree concepts
  - degree of x: number of children。就举这个例子，8 的 degree 就是 2，因为有两个孩子



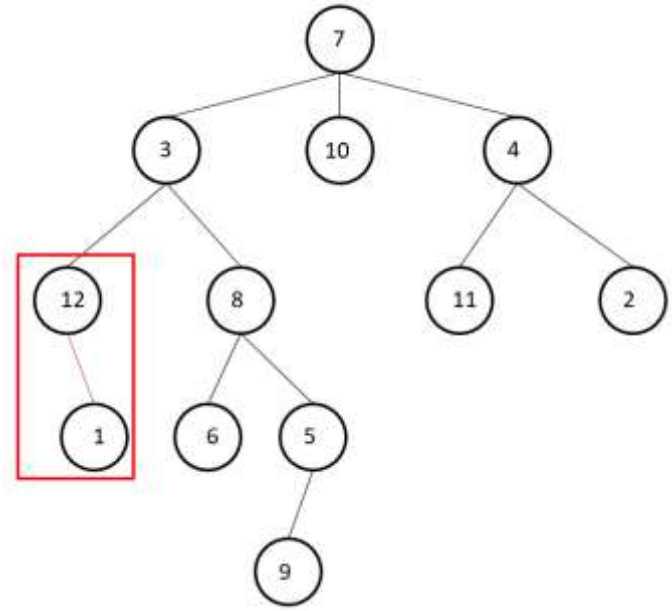
  - depth of x: length of the simple path from root to x. 你看 8 的 depth 就是 2.我们要知道 depth 是从 0 开始的



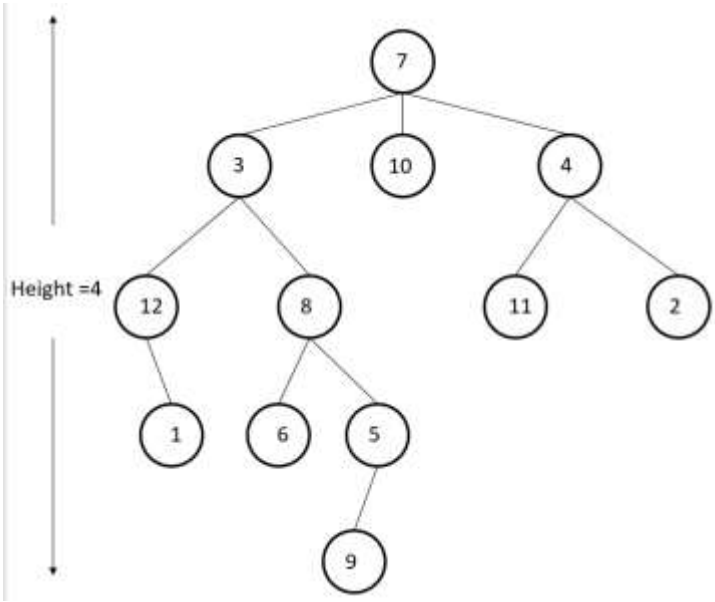
- level of a tree: all nodes at the same depth.就比如说在 depth 2，它的 level 就是 4，因为有 4 个 nodes。再比如说在 depth 3， 它的 level 就是 3，因为只有 3 个 nodes



- height of x: length of the longs simple path from x downward to some leaf node. 这个就是往下算 path。举例。12 的 height 就是 1. 它的高度就是 1，因为只有一条 path



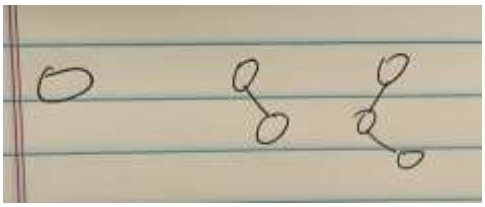
- height of a tree. 算最长的那个 path 就行



- ordered Tree： rooted tree in which children of each node are ordered. 也就是排好序的 tree

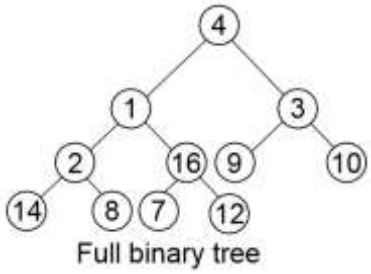
special types of trees

- Binary tree
  - contains no node
  - root node, left subtree right subtree
  - 这三个例子都是 binary tree



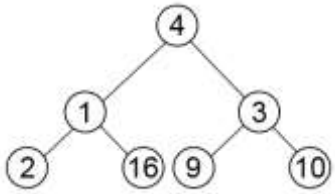
- Full binary tree
  - a is either a leaf or has degree exactly 2

- 这个例子就是 full binary tree. 要么没有孩子，要么孩子有两个



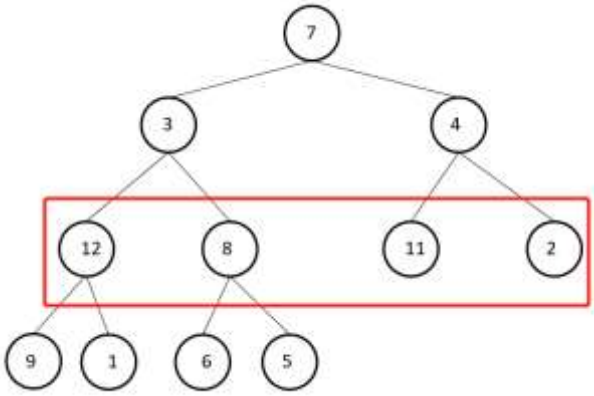
Full binary tree

- complete binary tree
  - A binary tree in which all leaves are on the same level and all internal nodes have degree 2
  - 孩子全满的那种



binary tree useful properties

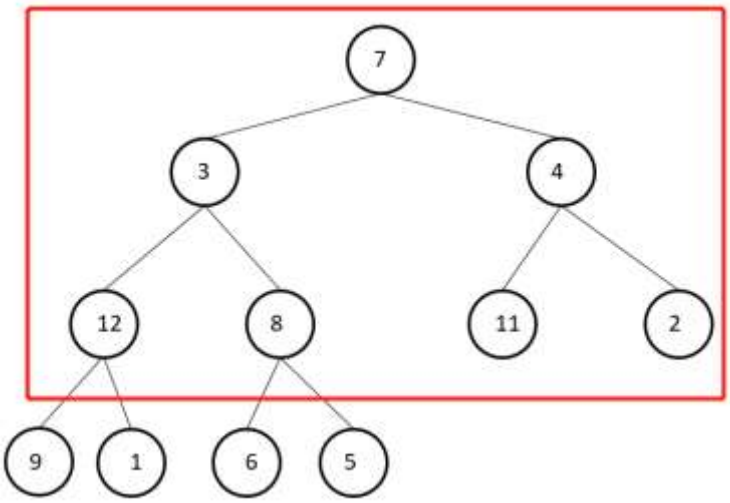
- 最多有 2^l 个 nodes 在 level l of a binary tree
  - 在这个 depth，它的 level 最多有 2^2=4nodes



- A binary tree with depth d has at most 2^(d+1) -1 nodes.证明也简单，就是算每个 level 的 nodes 相加求和即可

$$n \le \sum_{l=0}^{d} 2^l = \frac{2^{d+1}-1}{2-1} = 2^{d+1}-1$$

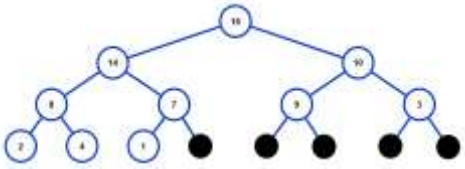
  - 在 depth 2， 最多有 2^3 -1 个 nodes， 也就是 7 个 nodes



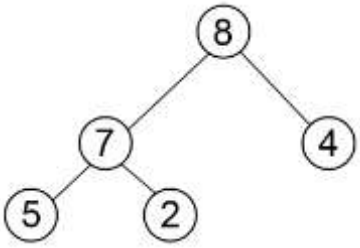
- A binary tree with n nodes has depth at least lgn

Data structure——heap

A heap can be seen as a complete binary tree。那些原本没有 nodes，就用 null pointers 填充



two properties



- structural property: all levels are full, except possibly the last one, which is filled from left to right. 也就是说从左到右填充
- order property: for any node x: parent(x) ≥ x . 这个 property 说明， root 就是最大的元素

假设我们有个数组

- 这样我们就会有

heap types

- max heap: (largest element at root), have the max-heap property. it means for all nodes i, excluding the root: A[parent(i)]≥A[i]
- min heap (smallest element at root), have the min-heap property。 it means for all nodes i, excluding the root: A[parent(i)]≤A[i]

Adding/Deleting nodes

- New nodes are always inserted at the bottom level (from left to right)
- Nodes are removed from the bottom level (from right to left)

Data structure——priority Queues

我们都知道队列的特点就是先进先出

但是优先队列，也就是有优先级。优先队列的存在，对 heap 是有很大的帮助的。

它就是设置优先级。不再是先进先出，优先级高的先出来

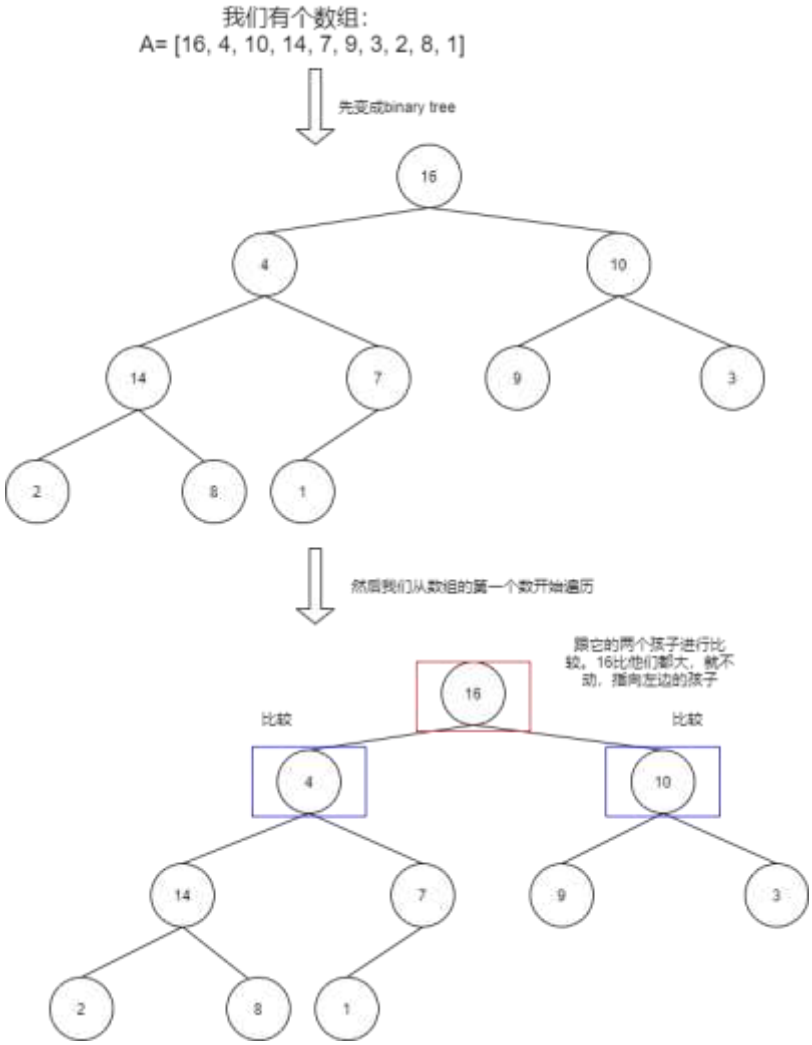
每个 element 都 associated with a value (priority)

The key with the highest (or lowest) priority is extracted first

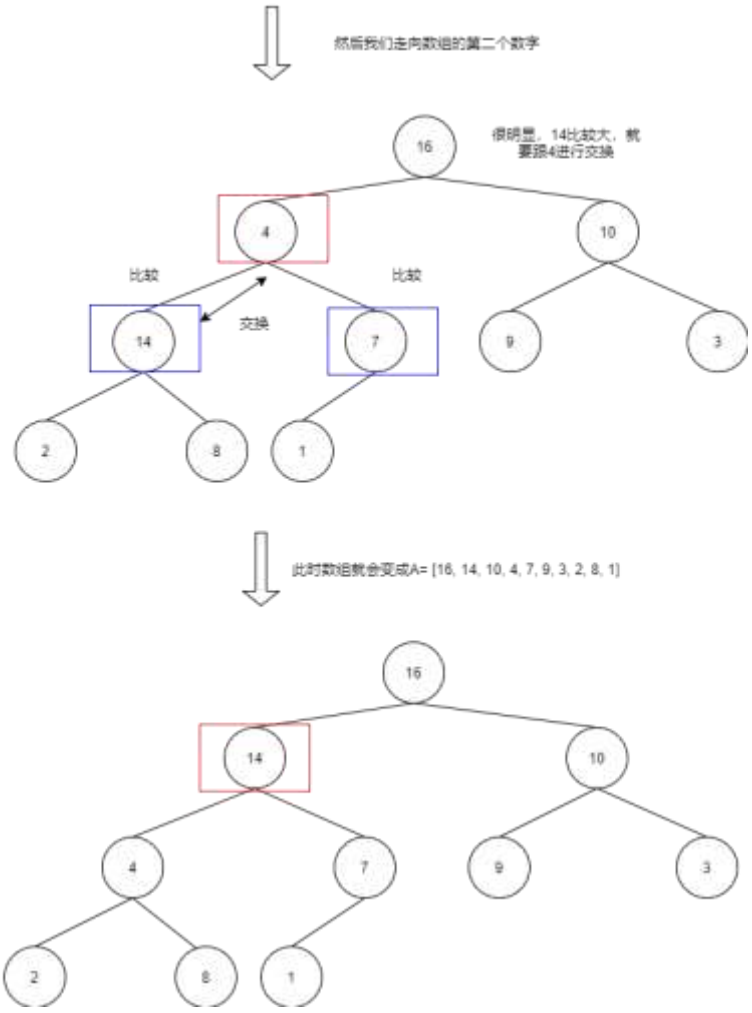
Sort

Heap sort

- Max-heapify
  - basic idea
    - 我们先假设孩子的值比较大
    - 这个时候就不算 max heap 了，就要进行交换
    - 此时，可以把较大的值往上移，就是与父母的值进行交换
    - 直到 parent value > children values
  - 举个简单的例子
    - 我们有个数组，从第一个数开始遍历



- 我们再走向下一个数字，我们就一直遍历，直到遍历到 7 停止就可以，因为再往下走就没有意义了，于是我们就有公式遍历到 i=（A.length）/2



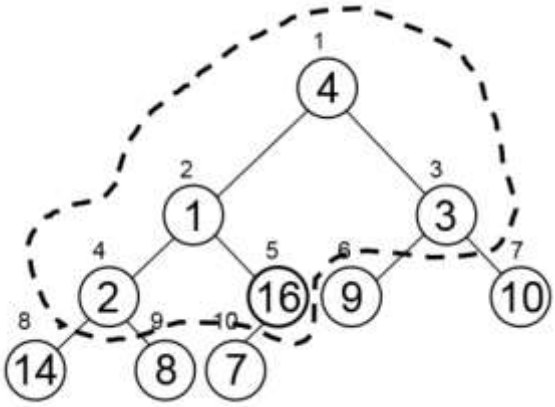
- 再接着往下遍历，直到 max-heap 即止

- pseudocode
  - 用递归实现

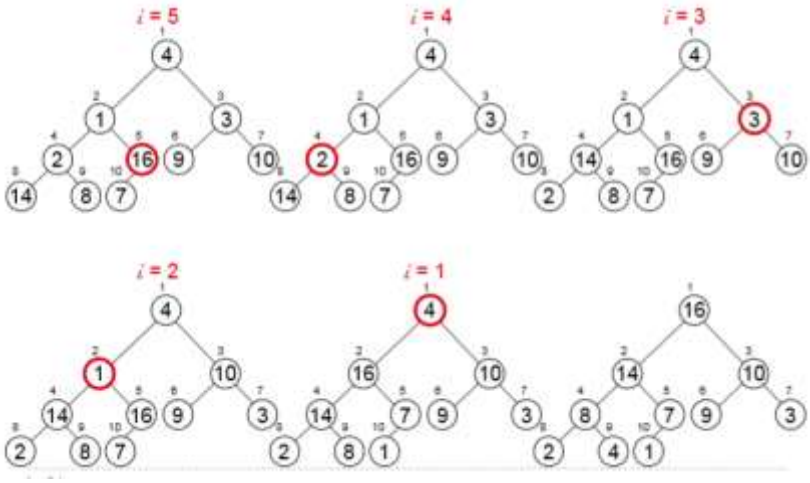
```
Max-heapify

Max-heapify (A, i)  //i starts from root.
        l = left(i), r = right(i) //l is the index of left child, r is index of right child
        if(l<=n && A[l]>A[i])
                largestIndex = l  //记录index
        if(r<=n && A[r]>A[i])
                largestIndex = r  //记录index
        if(largest != i)
                swap(A, A[largest], A[i]) //两数进行交换
                Max-heapify(A, largest) //递归调用
```

  - 所以它的 runtime 就是 O(lgn)

- Build max-heap
  - We can build a heap in a bottom-up manner by running Max-heapify on successive subarrays
  - 这玩意就是循环使用 Max-heapify。使每一个节点都是 max-heap。上面已经提到，遍历到 i = A.length/2 就可以了。 因为再往下就都是 leaves 了



- 举例，很简单
  - 就是不断地调用 max-heapify 而已

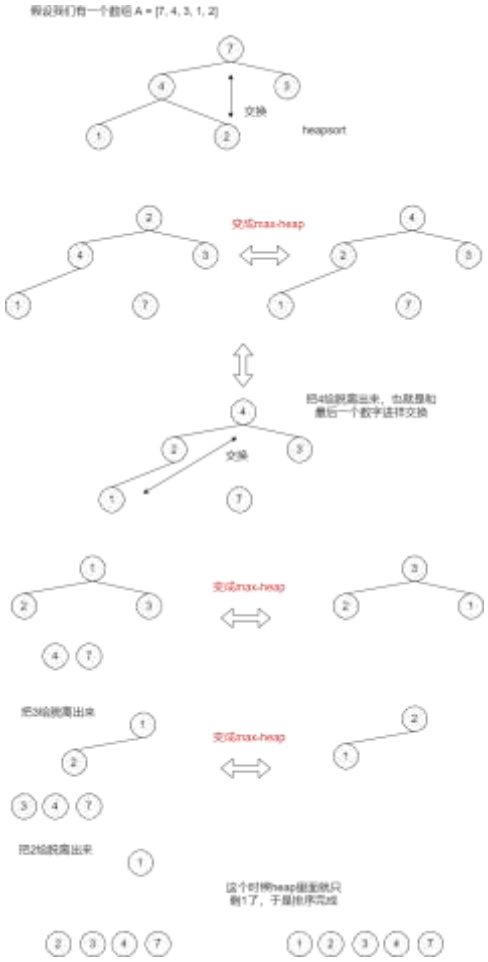


- pseudocode
  - 难度不大

```
刚刚已经提到，遍历到 i= A.length/2就可以， floor(A[n/2])
Build_Max_heap(A, i)
        n = A.length/2
        for n downTo 1
                Max_heapify(A, i)
```

- runtime analysis
  - upper bound is O(nlgn)
  - tight bound is O(n). 所以 Build max_heap 的时间复杂度就是 O(n)

- heapSort
  - basic idea
    - Build a max-heap from an anrray
    - swap the root (the maximum element) with the last element in the array
    - "discard" the last node by decreasing the heap size
    - Call Max-heapify on the new root
    - Repeat this process until onw node remains
  - Example
    - heap sort 的过程

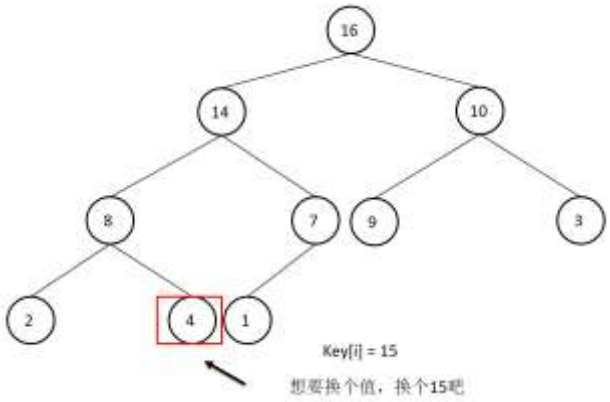


- pseudocode
  - 过程很好理解

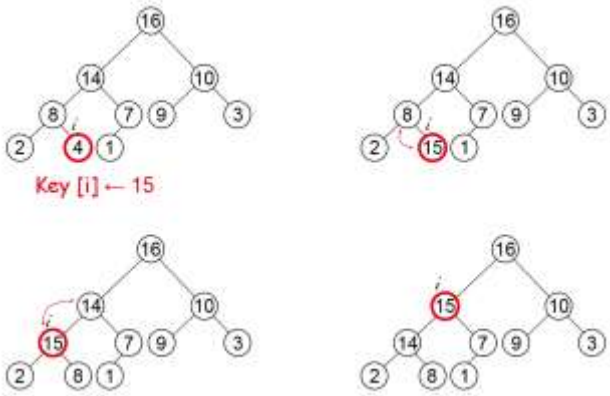
```
Heapsort

Build max heap;  //把数组输入后，建一个maxheap
for i=A.length downTo 2 //直到数组的第二个数即止
        exchange A[i] and A[1]  //这个时候root已经是最大了。于是和最后一个数进行交换
        Max-heapify(A, i-1)  //交换完之后，再建一个maxheap
```

- runtime analysis
  - 时间复杂度是 O(nlgn)
  - 因为 Build max heap 的时间复杂度是 O(n)
  - 有一个 for loop，里面有个 Max-heapify， 这样时间复杂度就是 O(nlgn)

- heap-increase-key
  - 这个东西的作用其实就是改变原有数组某个 element value，但是插入的值要比原来那个值大
  - basic idea
    - increment the key of A[i] to its new value.改变那个值。记得哦，要比原来的值大
    - if the max-heap property does not hold anymore, traverse a path toward the root to find the proper place for a newly increased key。改变之后要看是不是 max-heap 结构，如果不是，调一调
  - example
    - 换个值

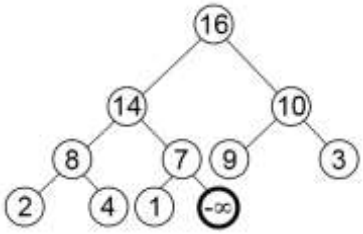


- 换了之后就看看是不是还符合 max-heap，不符合就调调

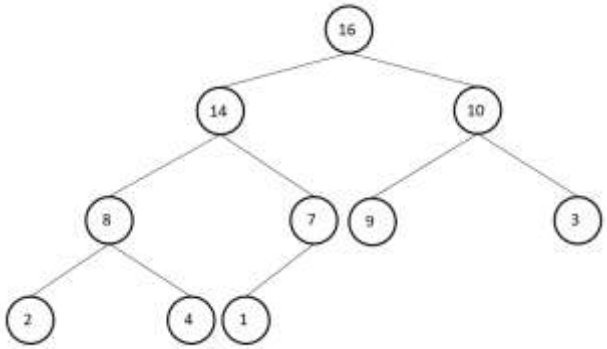


- pseudocode
  - Subtopic 1
- runtime analysis
  - 时间复杂度是 O(lgn)

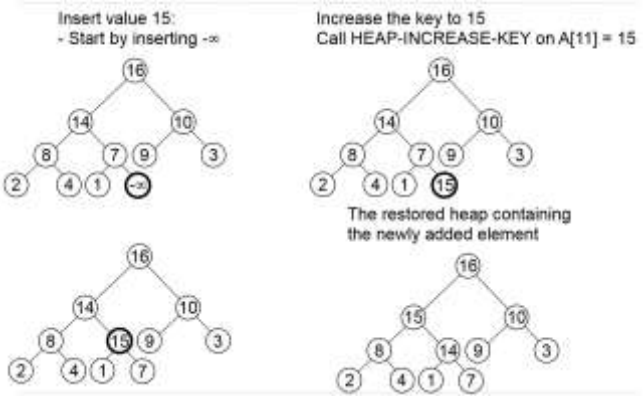
- Max-Heap-insert
  - 就是插入一个新值到 max-heap 中
  - basic idea
    - Expand the max-heap with a new element whose key is -∞. 也就是说在 heap 后面再加一个 - ∞



    - Calls Heap-increase-key to set the key of the new node to its correct value and maintain the max-heap property. 然后我们再 call heap-increase-key 函数，把新插入的这个数值放在合适的位置来 maintain max-heap
  - example
    - 假设我们有这么一个原始数组，已经是 max-heap 了



- 这个就是整体的 insert 过程，很简单，也就是 insert 在最后一位，然后调到合适的过程即可

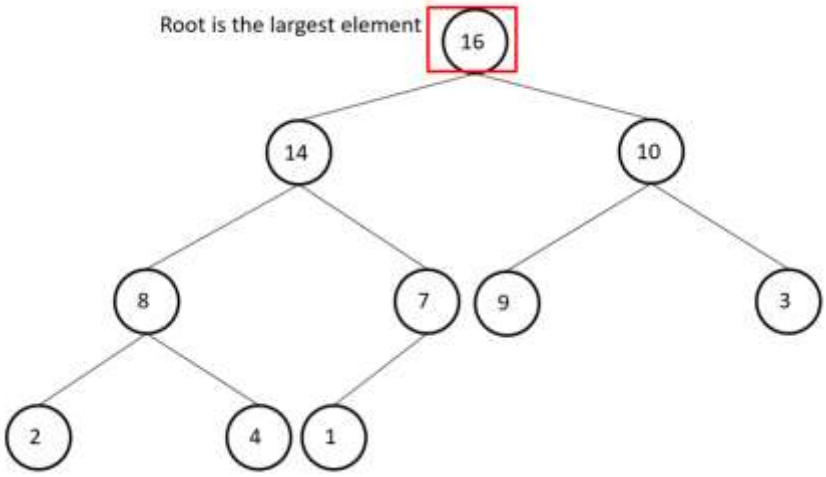


- pseudocode
  - 不难理解，就要要扩大原有的数组大小，然后插入，调用函数，调位置

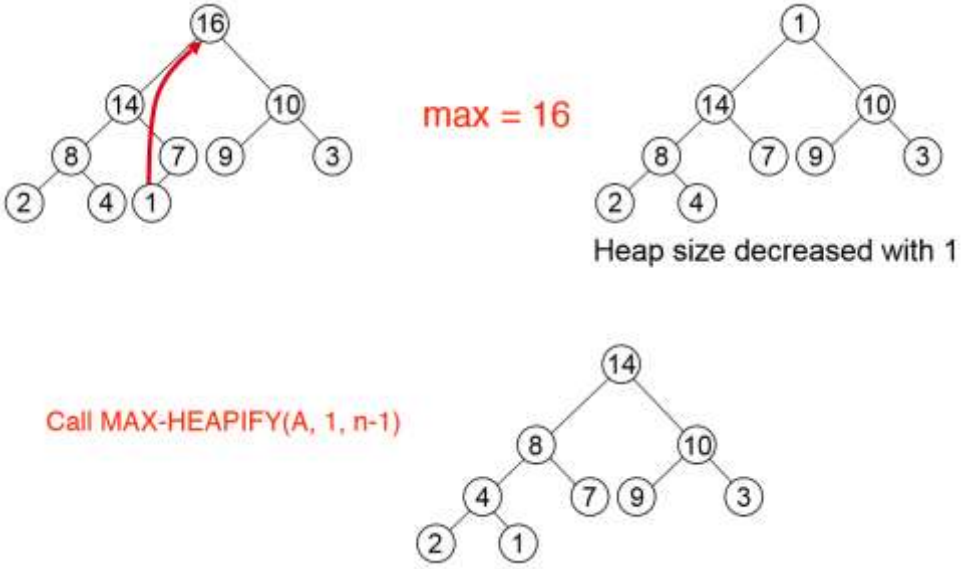
```
Max-heap-insert

Max_heap_insert(A, key, n) //A就是数组，key就是要插入的value，n就是原有数组长度
        heap_size -> A[n+1] //要把原有数组的长度变为n+1
        A[n+1]<- - ∞  //最后一个位置插入 - ∞
        heap_increase_key(A, n+1, key) //调用函数，插入，调到合适的位置
```

- runtime analysis
  - 这个时间复杂度就是看 heap_increase_key 的时间复杂度， 也就是 O(lgn)

- Heap-extract-Max
  - 这也就是要取 heap 里面最大的数，也就是 return max value 然后把那个 max value 去掉
  - basic idea
    - exchange the root element with the last element
    - decrease the size of the heap by 1 element
    - call max-heapify on the new root, on a heap of size n-1
    - root 就是最大值



- example
  - 一定要记得调回 max-heap



- pseudocode
  - 过程很简单，就是 call max-heapify

```
heap-extract-max
heap-extract-max(A, n) //给出数组，数组长度

        if n<1 //如果数组里没有数，就return error
              then error "heap underflow"
        max is A[1] //最大值就是root，也就是heap的第一个数
        A[1] <- A[n] //和最后一个值进行交换
        max-heapify(A, n-1) //call max-heapify, 忽略掉最后一个数
        retun max
```

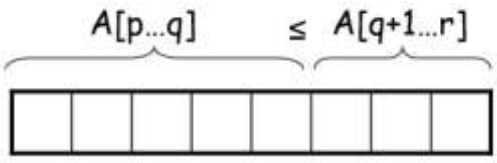
- runtime analysis
  - 就是 max-heapify 的时间复杂度，为 O(lgn)

- summary

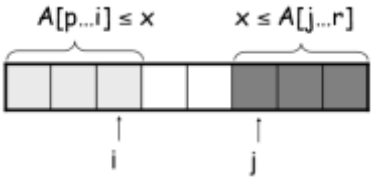
MAX-HEAPIFY	O(lgn)
BUILD-MAX-HEAP	O(n)
HEAP-SORT	O(nlgn)
MAX-HEAP-INSERT	O(lgn)
HEAP-EXTRACT-MAX	O(lgn)
HEAP-INCREASE-KEY	O(lgn)
HEAP-MAXIMUM	O(1)

Quick Sort

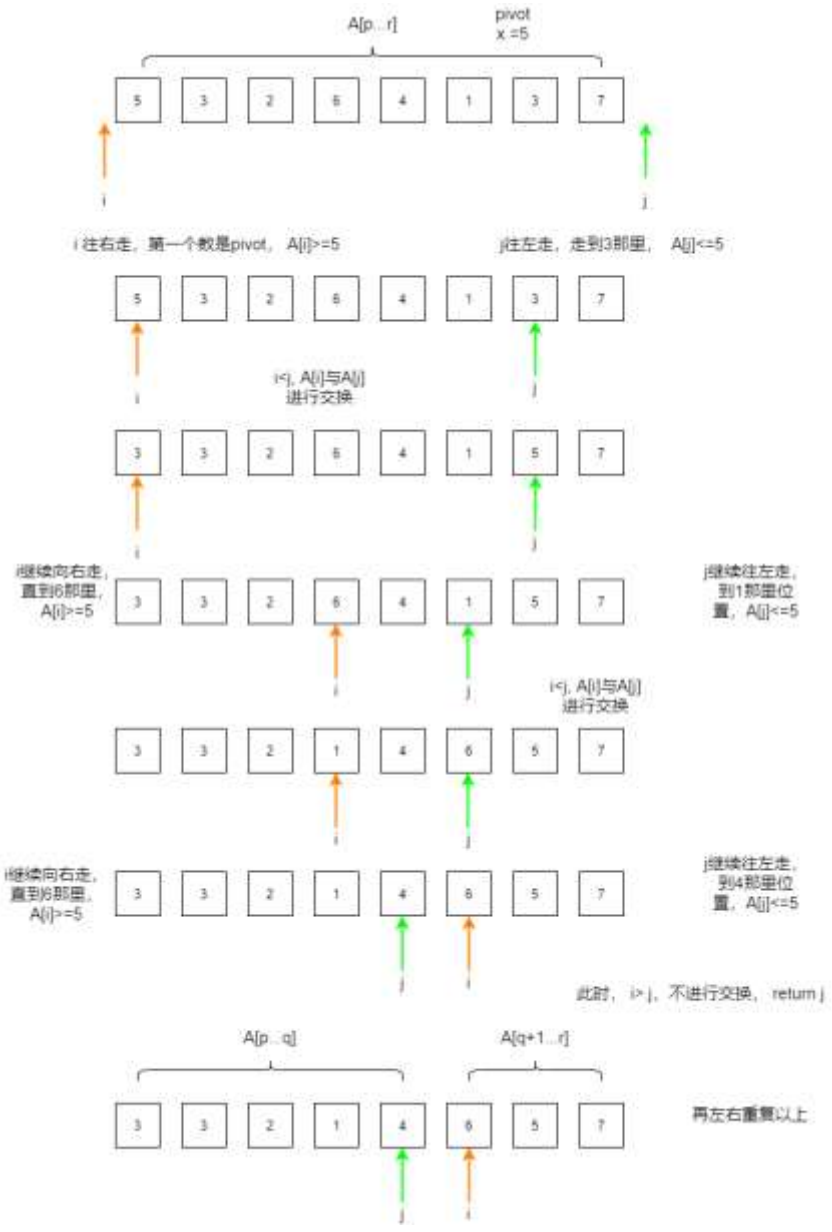
- 使用 divide and conquer 的方法
  - divide
    - partition the array A into 2 subarrays A[p...q] and A[q+1...r], such that each element of A[p...q] is smaller than or equal to each element in A[q+1...r] . 这玩意吧，就是找个中间值，然后分两半，左边的全部都≤中间值，右边的都≥中间值



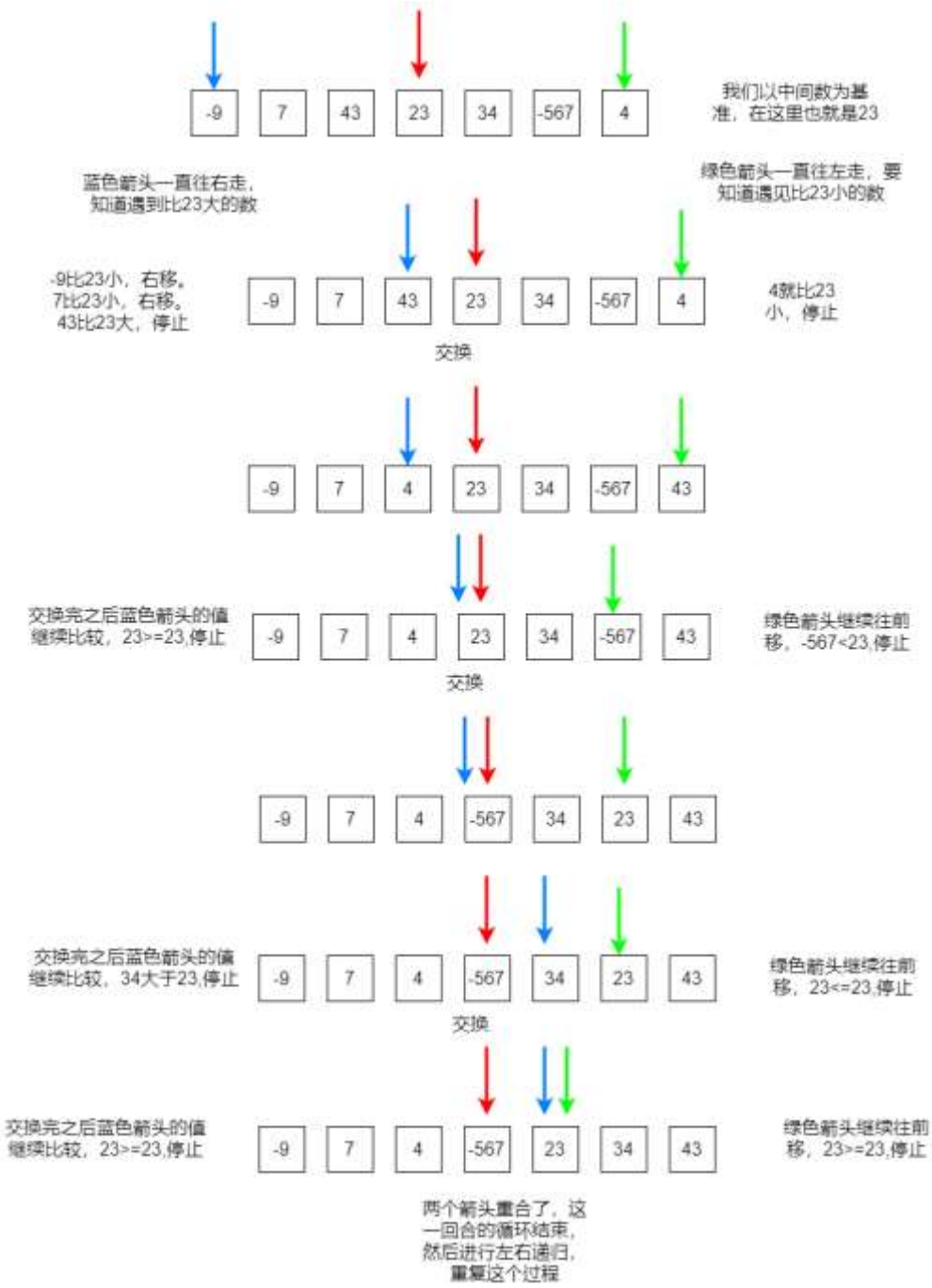
    - need to find the index q to partition the array。最重要的就是找个这个index
  - conquer
    - recursively sort A[p...q] and A[q+1...r] by calls to Quicksort。递归地call Quicksort
  - pseudocode
- Partition
  - basic idea
    - Quicksort 里面最重要的核心就在这里
    - rearranges the subarray in place
    - end result
      - two subarrays
      - all values in first subarray ≤ all values in second
    - returns the index of the "pivot" element separating the two subarrays
  - Hoare partition
    - basic idea
      - select a pivot element x around which to partition. 就是说选一个pivot 值。有许多种选法啦。
        - starts from both ends



- grows two regions
  - A[p...i] ≤ x
  - A[j...r] ≥ x
- exmaple
  - 图解。很好理解，就是左右分开，然后进行递归，左右重复。这个是选最左边的值为 pivot



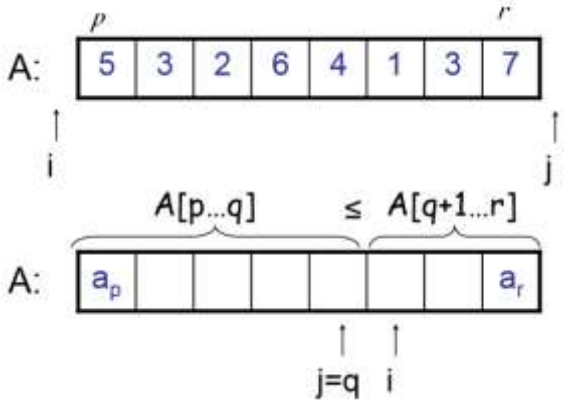
  - 这个是选取中间值为 pivot



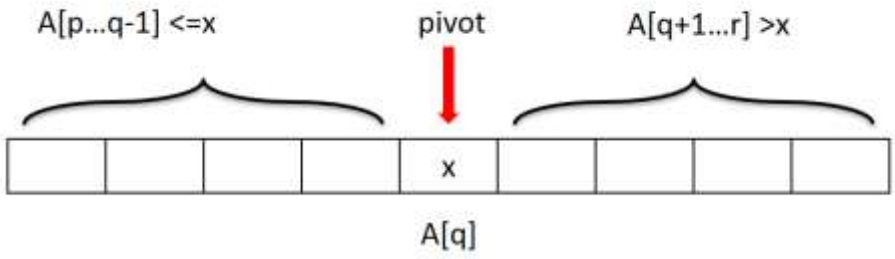
- pasudocode
  - 过程也挺简单，就是左右找，找到之后进行交换

```
Partition (A, p, r)
        x <- A[p]
        i <- p-1
        j <- r+1
        while (true)
                do repeat j <- j-1
                        until A[j]<=x
                do repeat i <- i+1
                        until A[i] >=x
                if (i<j)
                        then exchange A[i] <->A[j]
                else return j
```

  - 图解

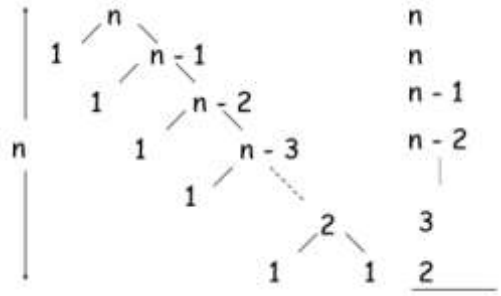


- runtime analysis
  - 时间复杂度为 O(n),就是遍历每一个数
- Another partition: Lomuto's Partition
  - basic idea
    - given an array A, partition the array into the following subarrays
      - A pivot element x = A[q]
      - subarray A[p...q-1] such that each element of A[p...q-1] is smaller than or equal to x (pivot)
      - subarray A[q+1...r], such that each element of A[p...q+1] is strictly great than x (the pivot)
    - The pivot is not included in any of the two subarrays

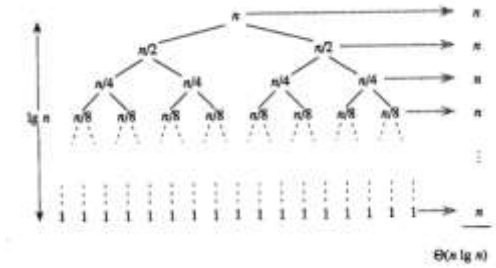


在Lomuto's method 中，pivot值是不包括在任何subarray中的，但是Hoare method，pivot值是包括在subarray中的

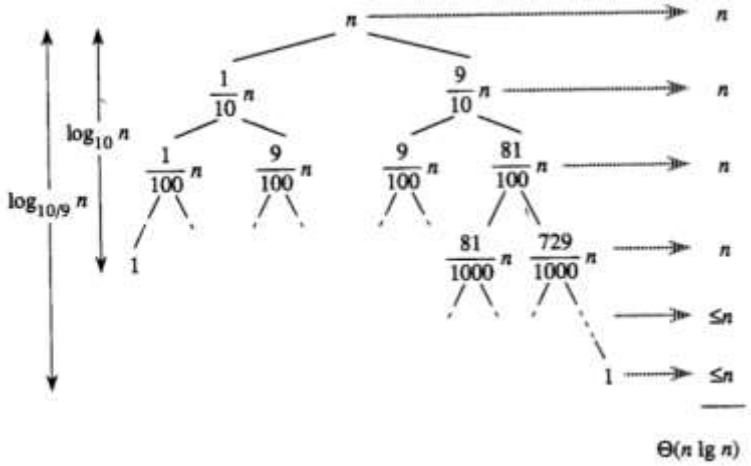
- Analyzing Quicksort
  - basic staff
    - sorts in place
    - it is not stable
    - sorts O(nlgn) in the average case
    - sorts O(n^2) in the worst case
      - but it does not happen often
  - what will be the worst case for quick sort?
    - when partition is always unbalanced
    - when the input is already sorted
    - when it is extremely unbalanced. 求和就会是 O(n^2)



- what will be the best case for the algorithm?
  - when partition is perfectly balanced
  - when it is perfectly balanced



- 我们再来举个例子，每次 partition 都是 9-1 分，这样看起来已经很 unbalanced 了，对吧



- 写出方程式是这样的

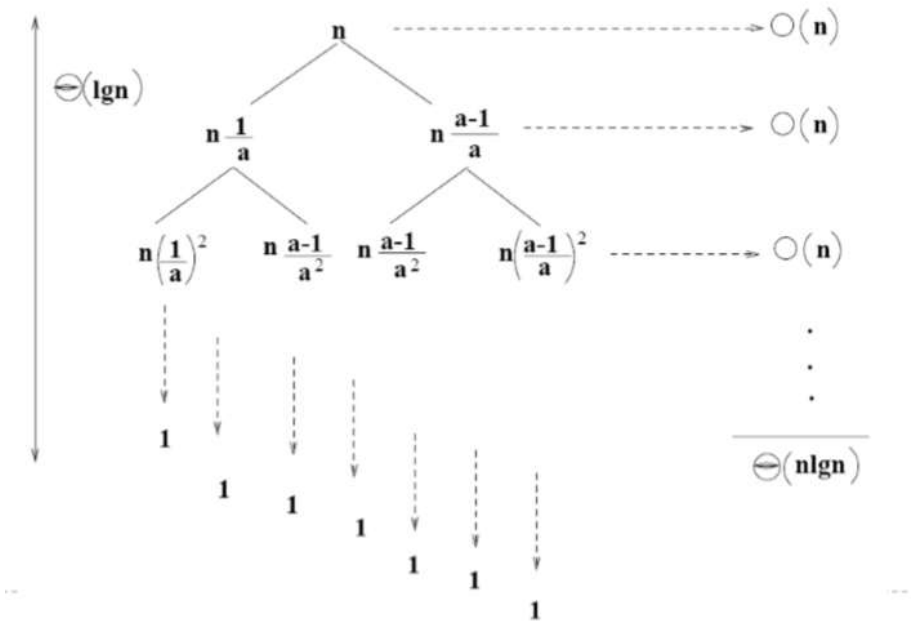
$T(n) = T(9n/10) + T(n/10) + n$

- 进行求和，时间复杂度也是 O(nlgn)，所以，worst case 的可能性很低

- How does partition affect performance?

- 我们记住一个理论就行。就是左右的比例是常数的，那么时间复杂度为 O(nlgn)
  - consider the (1 : n-1) splitting
    - 1/(n-1)不是常数，最坏情况产生了
  - consider the (n/2 : n/2) splitting
    - (n/2)/(n/2) = 1，是常数，那么时间复杂度就是 O(nlgn)
  - consider the (9n/10 : n/10) splitting
    - (9n/10)/(n/10)=9，是常数

- consider the ((a-1)n/a : n/a) splitting



  - ((a-1)n/a)/(n/a) = a-1, it is constant.
- average case
  - All permutations of the input numbers are equally likely
  - on a random input array, we will have a mix of well balanced and unbalanced splits
  - Good and bad splits are randomly distributed across throughout the tree
  - the runtime is O(nlgn)

lecture 5

Randomized Quick Sort

counting sort

计数排序，这种排序方法没有比较，但是数字必须在一定的范围之内．必须在[0...r], 有负数出现都不行

counting sort is stable

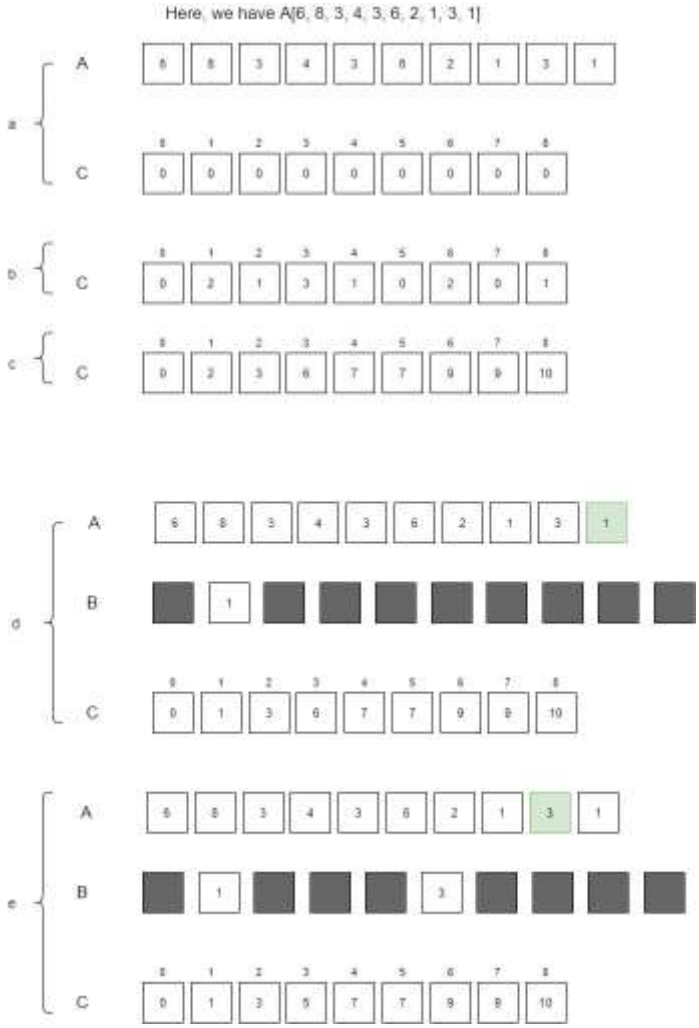
counting sort is not in place sort. 因为得另外开辟新数组

basic idea

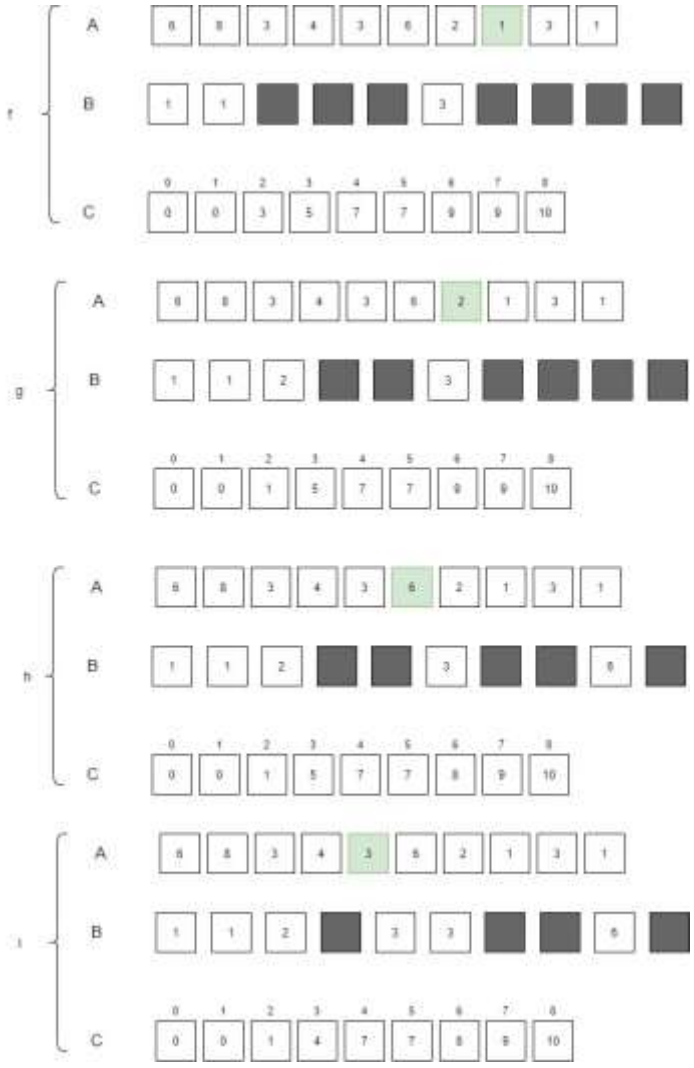
- for each element x, find the number of elements <= x
- place x into its correct position in the output array

example

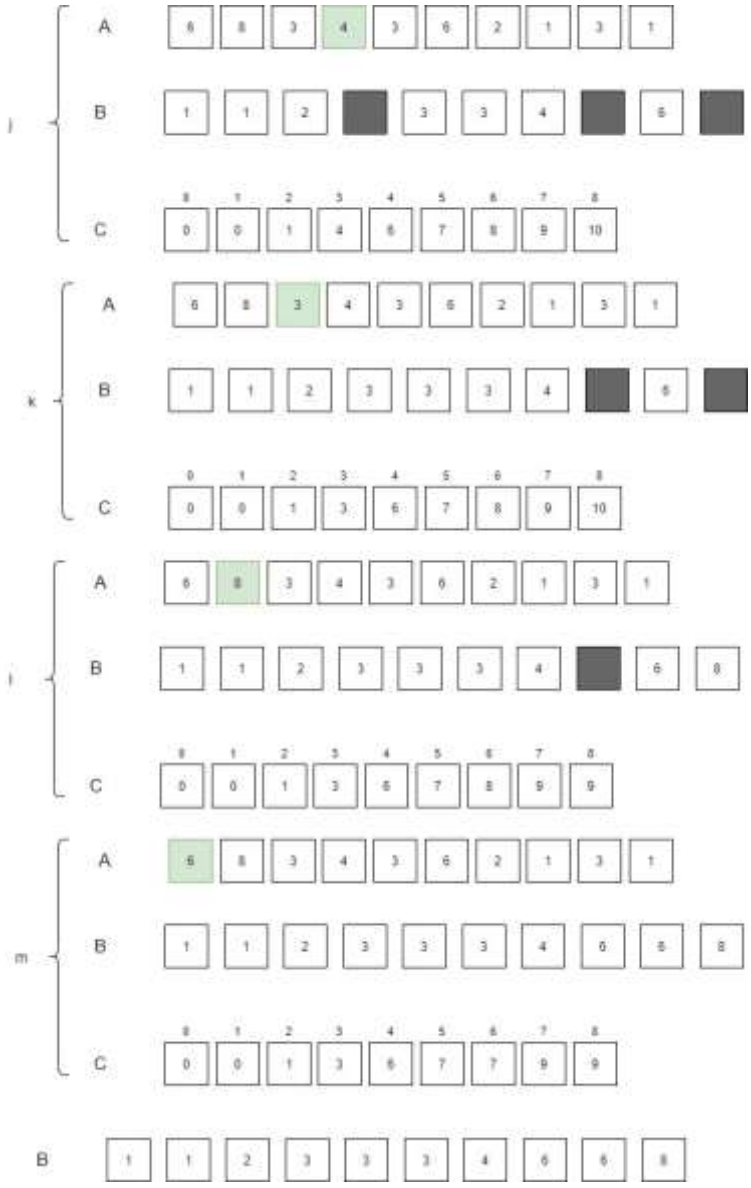
- counting sort 其实很简单，给出一个数组，就是看看它里面的各个元素分别有多少个



- 知道有多少个之后就从数组的最后一个元素走起，一一摆放在正确的位置



- 对数组的每个元素都进行操作，记得要 update 记录数组元素个数的那个数组，每过一个就要减一



pseudocode

- 很简单，用 4 个 for loop 是可以搞定的

```
Counting sort

For i to r  //自己定r的范围，也就是看数的大小，定个范围
        C[i] = 0  //先全部初始化为0

For i to n  //记录每个元素的个数
        C[A[i]] +=1

For i =1 to r  //记录每个元素应该插入的位置
        C[A[i]] = C[A[i]] + C[A[i-1]]

For i = n down to 1  //开始sort
        B[C[A[i]]] = A[i]
        C[A[i]] -= 1   //记得要update 位置
```

runtime analysis

- 因为是 4 个 for loop， 所以时间复杂度为 O(n)
- 但是会占用很多空间，因为得另外开辟两个数组

Radix sort

basic idea

- 通过键值的各个位的值，将要排序的元素分配置某些桶中，达到排序的作用
- 就是个位数，百位数，千位数
- 这是效率高的稳定性排序法

example

pseudocode

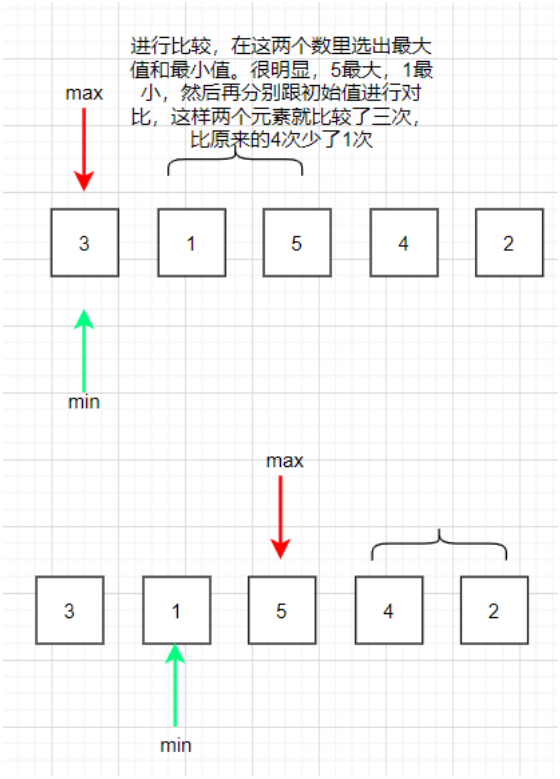
Sorting Lower Bound

Order statistics & selection

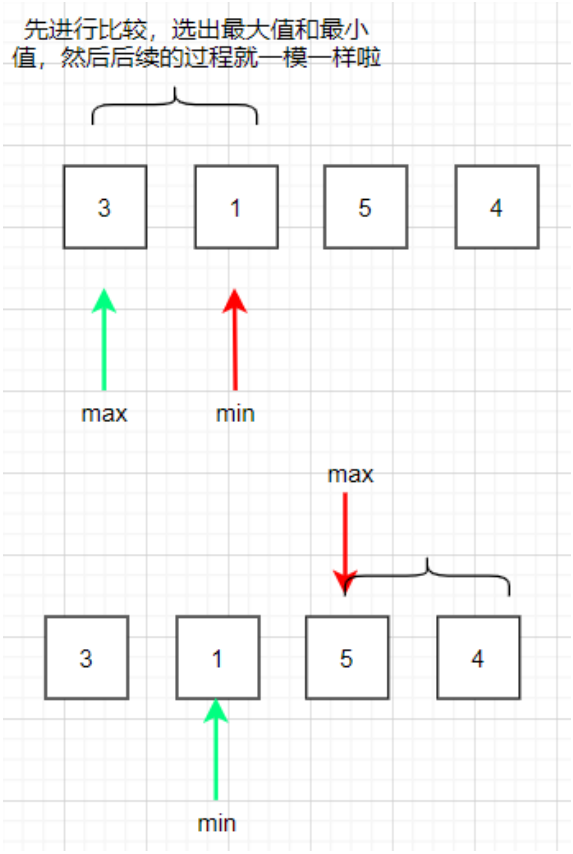
order statistic

- 对于 order statistic 的理解就是排在第几位的数。一般里的应用就是说找第几个数举个例子来说，在已经排好序（升序）的数组中，第一个就是最小的数，最后一个就是最大的数

- 但是对于 median，有点不太一样哦
  - 奇数，只有一个 median，那就是（n+1）/2
  - 偶数，有两个 medians，一个是 low median （n/2），一个是 upper median ((n/2)+1)，但是，我们日常生活中，对于偶数的 median，一般就是认为是 low median
- how many comparisons are needed to find the minimum element in a set? the maximum? 未排好的数组中
  - 这样我们就一个一个比较呗，就比（n-1）次
- 那么我们怎么同时找出 minimum 和 Maximum value 呢?
  - 正常来说，我们找最小值比较 （n-1）次，找最大值，又得比较（n-1）次，这样就总共比较 2n-2 次
  - 有一个方法，不用比较这多次，但这种得分情况讨论，但区别也就是设置初始值不一样，仅此而已
    - 奇数



    - 偶数

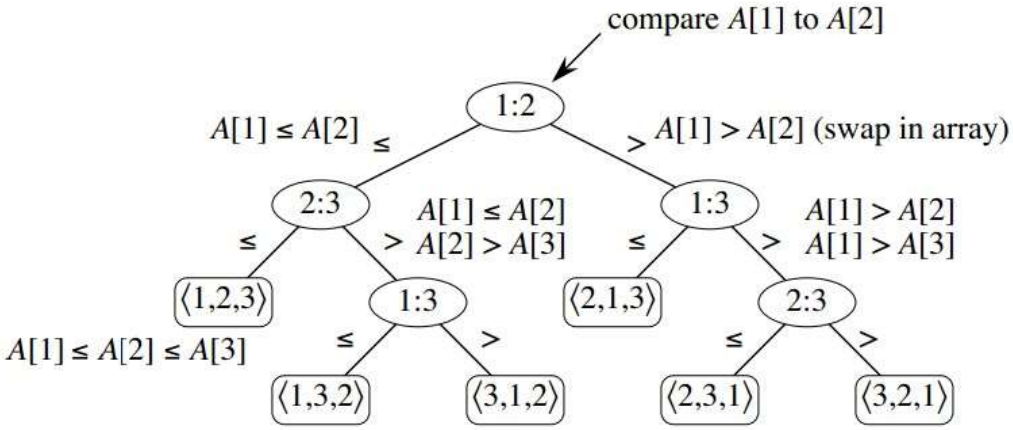


- runtime analysis
  - 美两个元素，就比较三次，时间为 θ(3n/2),比原来的稍微快一点
- finding order statistic: the selection problem
  - A more interesting problem is selection: finding the i-th smallest element of a set.也就是在一串没有排序好的数组中，找到第 i-th 小的数。解决这个问题的方法就是 randomized slelection
  - randomized selection
    - basic idea
      - use partition() from quicksort
    - example
    - pseudocode

Decision Trees

就是说比较排序（凡是要进行数的比较的排序算法）可以被抽象地视为决策树

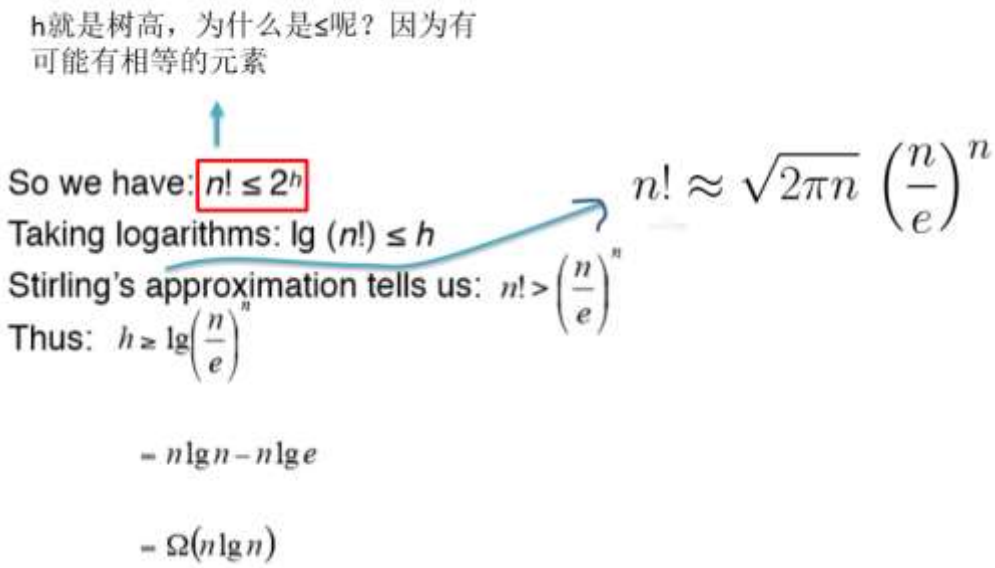
我们来举个例子，假设有一组三个元素的序列，我们就用 1， 2， 3 来表示，我们要将他们进行排序，就可以有这么一种决策树



- 我们可以发现，这些 leaf 表示的是这三个元素的所有可能排列。只要找到正确的排列，我们就能找到它排序的路径

现在我们假设有 N 个 elements，那么 how many leaves must be there? 答案是 N!，这是排列组合知识

- 现在我们来证明：Any decision tree that sorts n elements has height Ω(n lg n)



sort summary

	Best	Average	Worst	inplace	Stable
Bubble	O(n)sorted	O(n^2)	O(n^2)	Yes	Yes
Quick	O(nlogn)	O(nlogn)	O(nlogn)(unblance)	Yes	Noo
Heap	O(nlogn)	O(nlogn)	O(nlogn)	Yes	No
Insertion	O(n)	O(n^2)	O(n^2)	Yes	Yes
Counting	O(n)			No	Yes
Merge	O(nlogn)	O(nlogn)	O(nlogn)	No	yes
radix	O(nlogn)	O(nlogn)	O(nlogn)	no	yes

sort stable or not

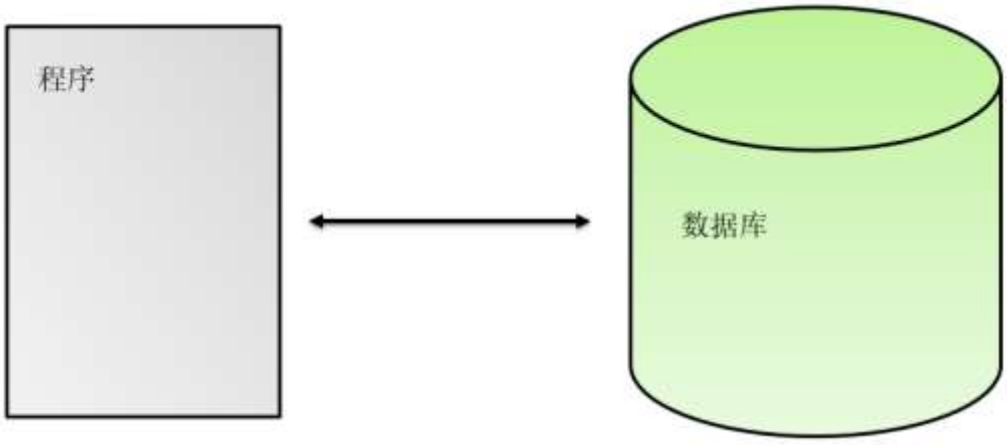
- stable
  - bubble sort
  - insertion sort
  - merge sort
  - radix sort
- unstable
  - selection sort
  - quick sort
  - shell sort
  - heap sort

lecture 6

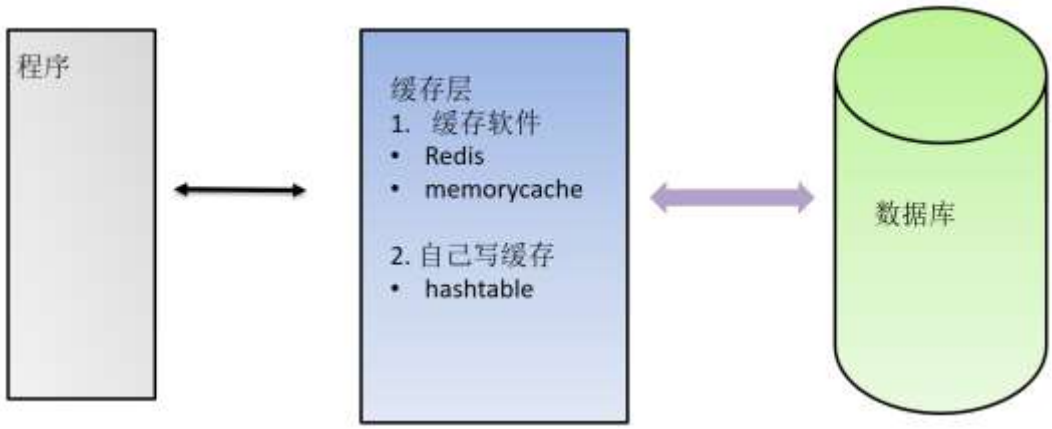
Hash Tables

Why do we need hashtable?

- 就是我们正常来说，我们取数据，一般是通过访问数据库得到的数据，但是有些数据是经常用的，而访问数据库却得去查每个数据，这样很浪费时间



- 然后我们就加入缓存层，把经常用的数据放在缓存层，这样就可以加快查询速度



Basic idea

- use a function h to compute the slot for each key
- store the element in slot h(k)
- 就是根据 key value 而直接进行访问的数据结构。也就是说，通过关键码值映射到表中的一个位置来访问记录，来加快查找速度
- hashtable 一般有两种实现方式
  - 数组+链表
  - 数组 + 二叉树

A has function h transforms a key into an index in a hash table T[0...m-1]

$$h : U \rightarrow \{0, 1, \ldots, m - 1\}$$

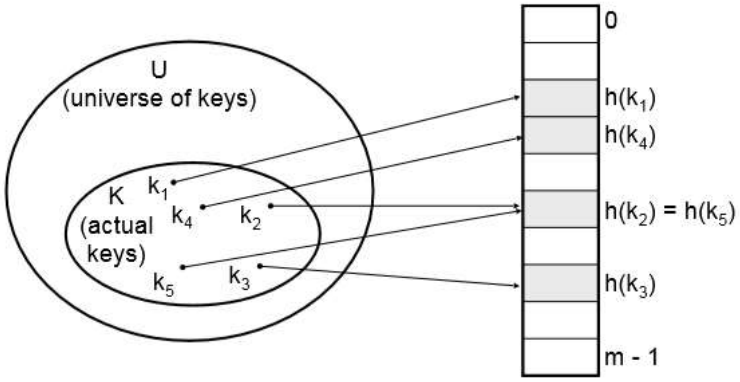
- Here, we can say that k hashes to slot h(k)

Advantage

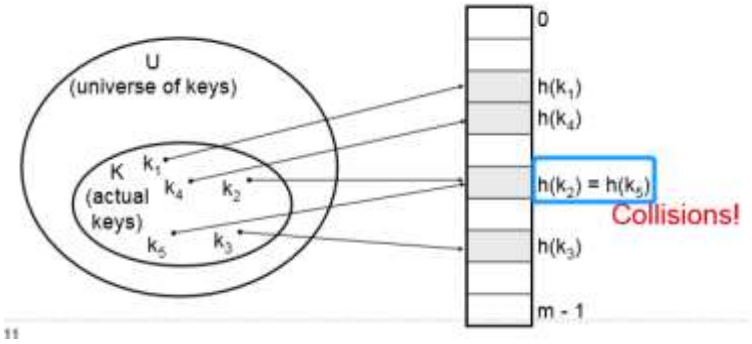
- Reduce the range of array indices handled: m instead of |U|
- storage is also reduced
- 省空间，查询速度快

example

- 先大概看一下，后面详细讲



- 有冲突了



- 很简单的例子来说，就是，给出员工信息，像，id，name, position, salary 等等。因为 hash table 最快的就是查询，所以想通过输入 id 来查询员工信息

Collisions

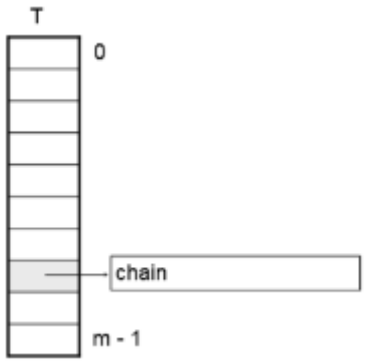
- Two or more keys hash to the same slot
- For a given set K of keys
  - if $|K| \leq m$, collisions may or may not happen, depending on the hash function
  - if $|k| > m$, collision will definitely happen (i.e. there must be at least two keys that have the same hash value)
- Avoiding collisions completely is hard, even with a good hash function
  - 这个时候我们得先考虑 table size
  - 太小不行，因为这样数组后面的链表会太长，增加了 search 时间
  - 太大不行，浪费空间
  - 通常就选总 elements 数的 五分之一或者是十分之一
  - 而且，到了 linked list 那里，数据并不是有序的的

Handing Collisions

- Chaining
  - Basic idea
    - put all elements that hash to the same slot into a linked list
    - slot j contains a pointer to the head of the list of all elements that hash to j
- Open addressing
  - Linear probing
  - Quadratic probing
  - Double hashing

Analysis of hashing with chaining

- worst case
  - All n keys hash to the same slot.也就是冲突很大，每次都是到同一个 slot



    - worst case time to search is θ(n), plus time to compute the hash function
- average case
    - it depends on how well the hash function distributed the n keys among the m slots
    - Subtopic 2

Hash functions

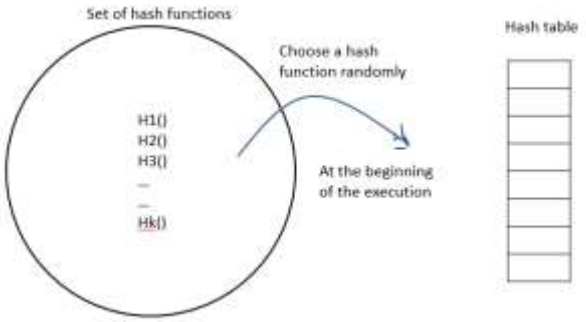
- A hash function transforms a key into a tale address
- what makes a good hash function?
    - easy to compute
    - approximates a random function: for every input, every output is equally likely (simple uniform hashing). But in practice, it is very hard to achieve
- Good approaches for hash functions
    - Minimize the chance that closely related keys hash to the same slot
        - strings such as pt and pts should hash to different slot
    - Derive a hash value that is independent from any patterns that may exist in the distribution of the keys
- Several methods for hash functions
    - the division method
        - basic idea
            - Map a key into one the m slots by taking the remainder of k divided by m
            - h(k) = k mod m, m 的选取，可能就是质数，像 11,13，但是不要靠近一个数，这个数就是 power of 2
        - advantage
            - Fast, requires only one operation
        - disadvantage
            - certain values of m are bad like power of 2, non-prime numbers
    - The multiplication method
        - basic idea
            - Multiply key k by a constant A, where 0 < A < 1
            - Extract the fractional part of (kA)
            - Multiply the fractional part by m
            - take the floor of the result， 必须要知道， (kA mod 1)是取小数部分，举个例子，3.55 % 1 = 0.55， 3 % 1 = 0。还有一个要清楚的是 floor，表示向下取整，比如，floor(4.99) = 4

$$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor = \lfloor m\ (k\,A \bmod 1) \rfloor$$

fractional part of kA = kA - ⌊kA⌋

- disadvantage
  - slower than division method
- advantage
  - value of m is not critical, eg., typically 2^p

- Universal hashing
  - In practice, keys are not randomly distributed
  - Any fixed hash function, adversary may construct a key sequence so that the search time in θ(n)
  - basic idea
    - select a hash function at random, from a designed class of functions at the beginning of the execution



  - designing a universal class of hash functions
    - choose a prime number p large enough so that every possible key k is in the range [0...p-1]

$$Z_p = \{0, 1, \ldots, p - 1\} \text{ and } Z_p^* = \{1, \ldots, p - 1\}$$

    - Define the following hash function

$$h_{a,b}(k) = ((ak + b) \bmod p) \bmod m,$$
$$\forall\ a \in Z_p^* \text{ and } b \in Z_p$$

    - The family of all such hash functions is this one. a, b will be chosen randomly at the beginning of execution

$$H_{p,m} = \{h_{a,b} : a \in Z_p^* \text{ and } b \in Z_p\}$$

    - example

$p = 17,\ m = 6$

$h_{a,b}(k) = ((ak + b) \bmod p) \bmod m$

$h_{3,4}(8) = ((3 \cdot 8 + 4) \bmod 17) \bmod 6$

$= (28 \bmod 17) \bmod 6$

$= 11 \bmod 6$

$= 5$

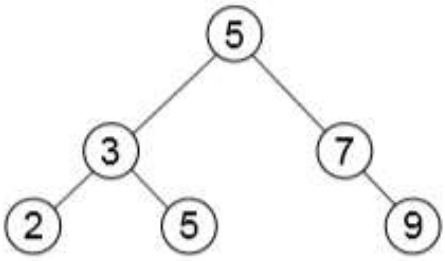
  - Advantage

- Universal hashing provides good results on average performance, independently of the keys to be stored
- guarantees that no input will always elicit the worst-case performance
- poor performance occurs only when the random choice returns an inefficient hash function - this has small probability

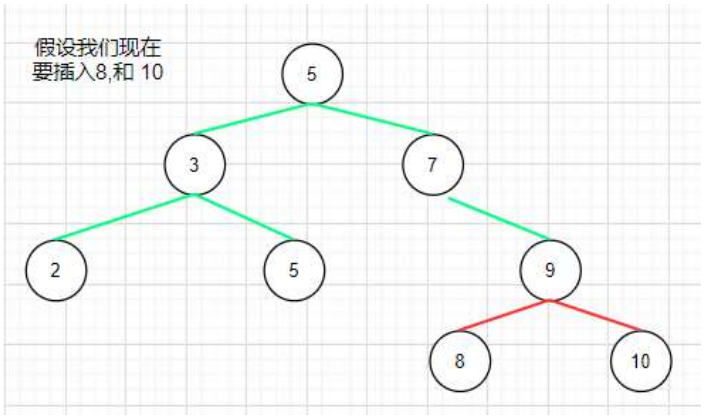
Binary Search Tree

property

- if y is in left subtree of x, then we can get key[y]<= key[x]
- if y is right subtree of x, then we can get key[y]>= key[x]
- 观察他们的大小

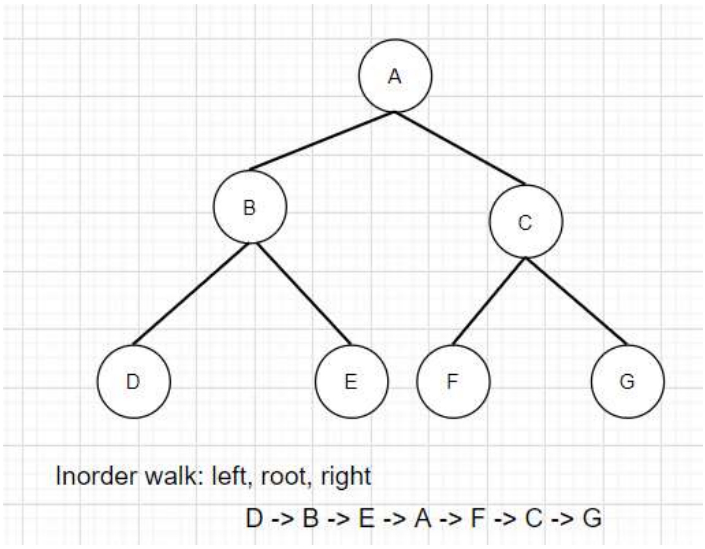


- 假设我们要插入 8 和 10



Traversing a binary search tree

- inorder tree walk
  - root is printed between the values of its left and right
  - example

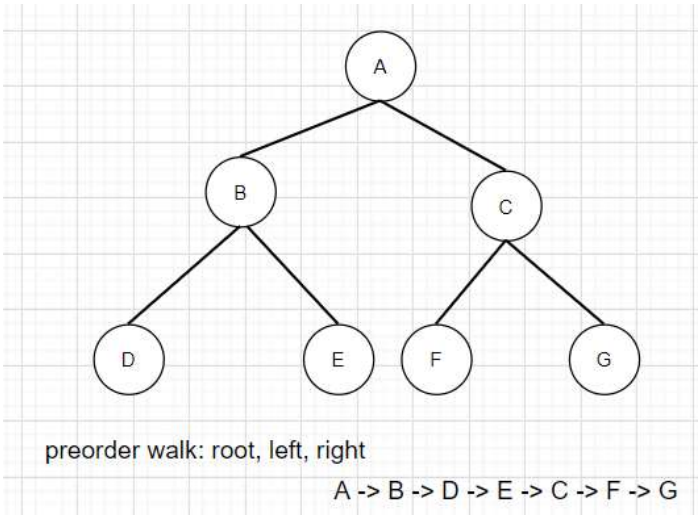


- pseudocode: 很简单，很容易理解，都是通过递归来实现的

```
Inorder_walk (x)
        if x ≠ NIL  //NIL就是值为空的意思
                Inorder_walk(left[x])
                print nodes
                Inorder_walk(right[x])
                print nodes
```

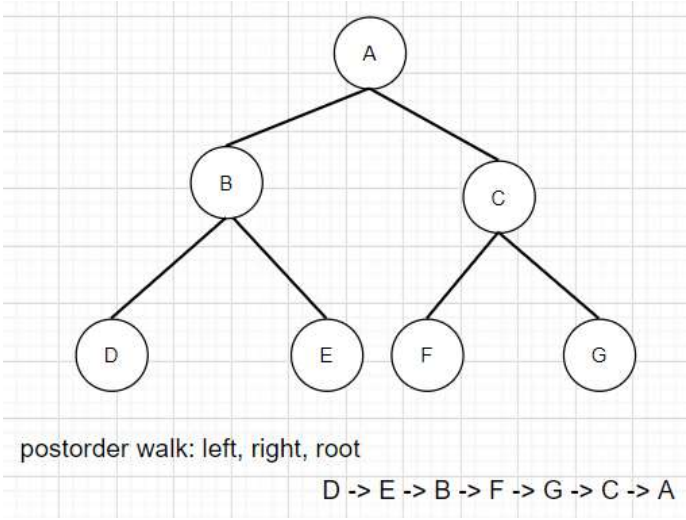
- runtime analysis
  - 每一个点都会遍历到，所以是 O(n)

- preorder tree walk
  - root printed first: root, left right
  - example



  - pseudocode 同上

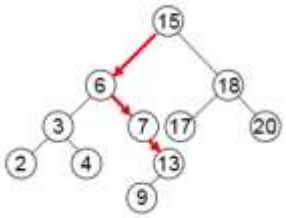
- postorder tree walk
  - root printed last: left, right, root
  - example



- pseudocode 同上

Binary search trees

- support many operations
  - search
    - basic idea
      - starting at the root, trace down a path by comparing k with the key of the current node, k 就是我们要找的数
        - if the keys are equal: we have found the key
        - if k < key[x], search in the left subtree of x
        - if k > key[x], search in the right subtree of x
        - 找不到， return NIL
    - example

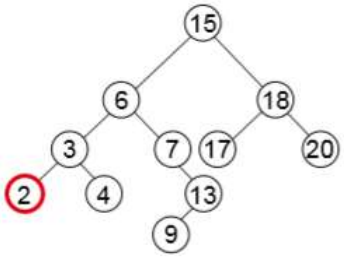


- Search for key 13:
  - 15 → 6 → 7 → 13

    - pseudocode

```
Binary search (x, k)
If x = NIL or k = key[x]
        return x
If k <key[x]
        return Binary search(left[x], k)
Else
        return Binary search(right[x], k)
```

  - minimum
    - 就是找最小值嘛，一直往左走就行
    - example

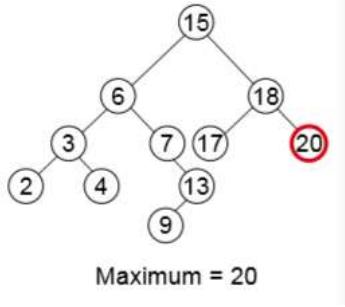


Minimum = 2

- pseudocode

```
Tree minimum(x)
While left[x] !=NIL
        do x <- left[x]
Return x
```

- maximum
  - 这个就是一直往右走
  - example



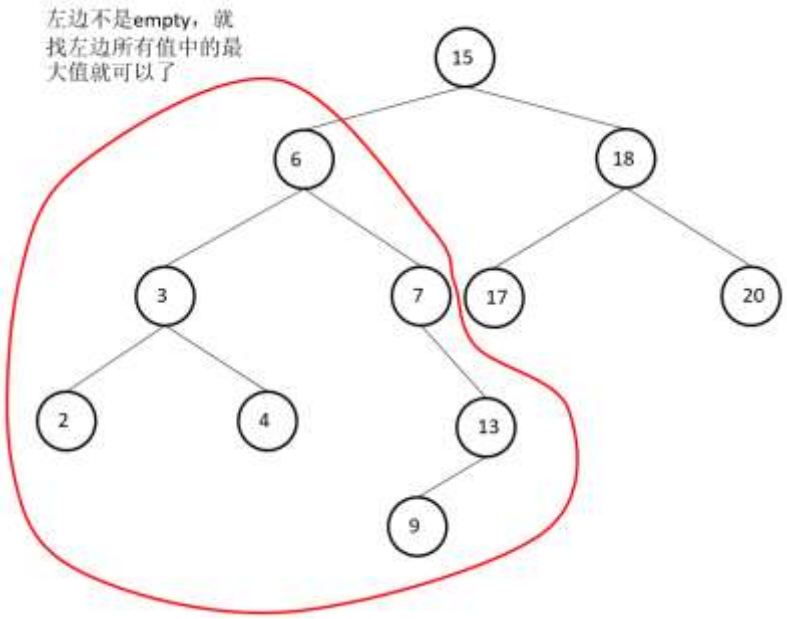
Maximum = 20

  - pseudocode

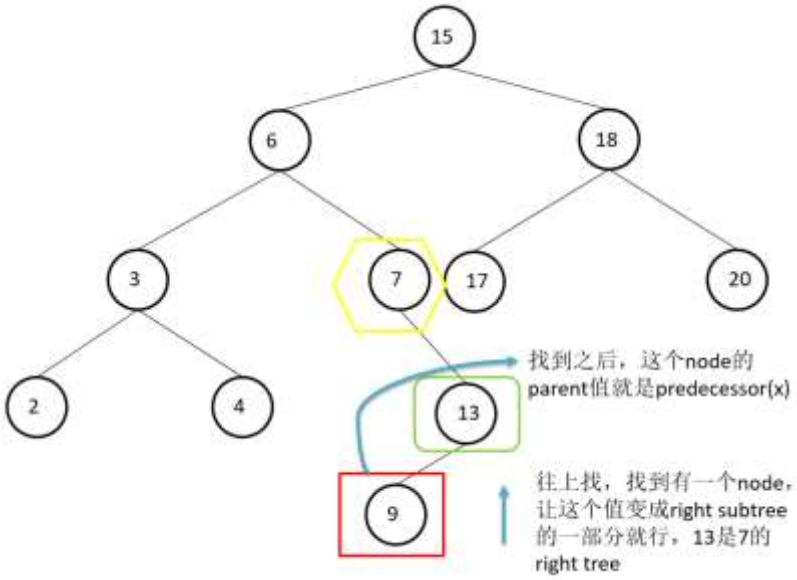
```
Tree maximum(x)
While right[x] !=NIL
        do x <- right[x]
Return x
```

- predecessor
  - 不是很明白为什么要有这种东西，它的定义是 predecessor (x) = y, such that key[y] is the biggest key < key[x]
  - 怎么找呢？分情况
    - left（x）is non empty
      - predecessor (x) = the maximum in left (x)
    - left (x) is empty
      - go up the tree until the current node is a right child: predecessor(x) is the parent of the current node
      - if you cannot go further (and you reached the root); x is the smallest element

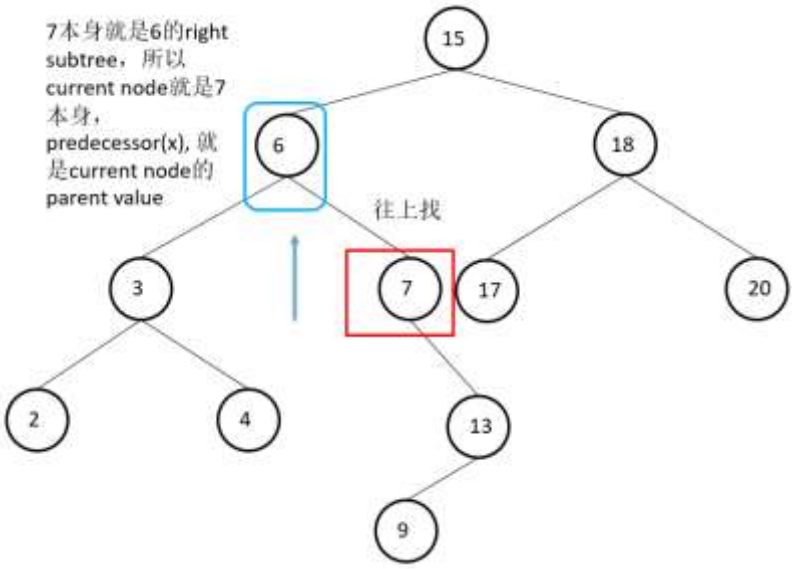
- example
  - predecessor(15) = 13



  - predecessor(9) = 7

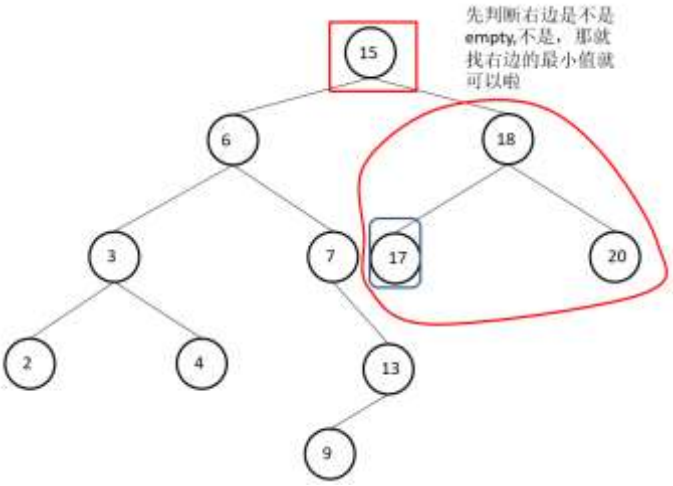


  - predecessor (7) = 6

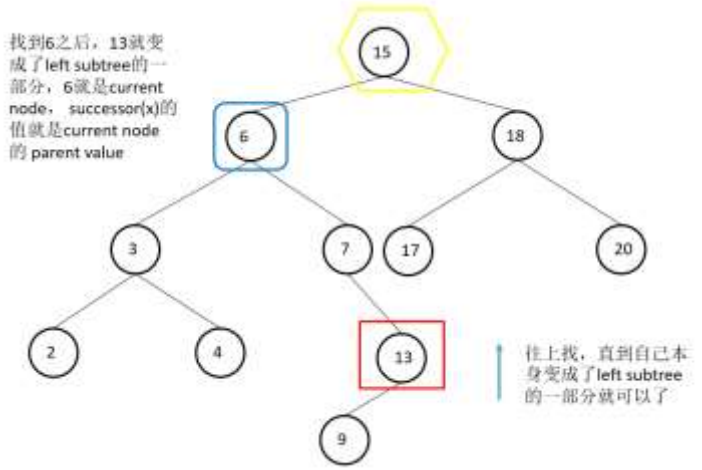


- pseudocode

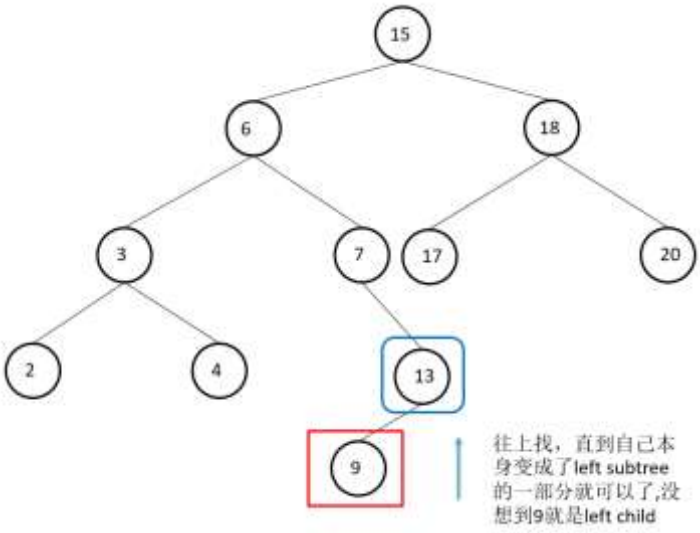
- successor
  - 也依旧不太明白为什么要有这个？这个东西的定义就是，successor (x) = y, such that key [y] is the smallest key > key [x]
  - 怎么找呢？得分情况
    - right(x) is non empty
      - successor(x) = the minimum in right (x)
    - right(x) is empty
      - go up the tree until the current node is a left child: successor(x) is the parent of the current node. If you cannot go further (and you reached the root): x is the largest element
    - example
      - successor(15) = 17



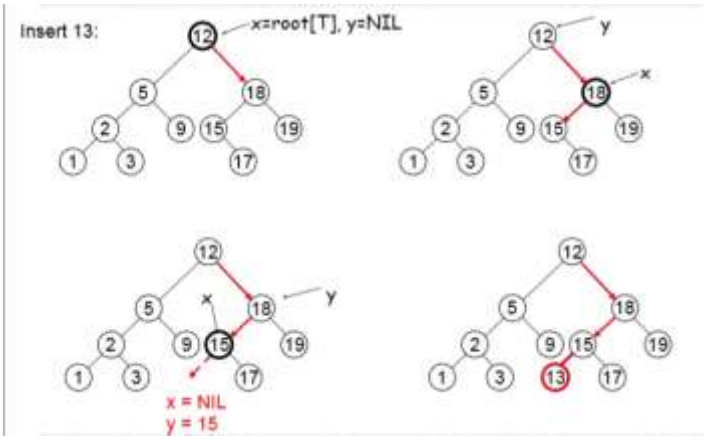
      - successor (13) = 15



- successor (9) = 13

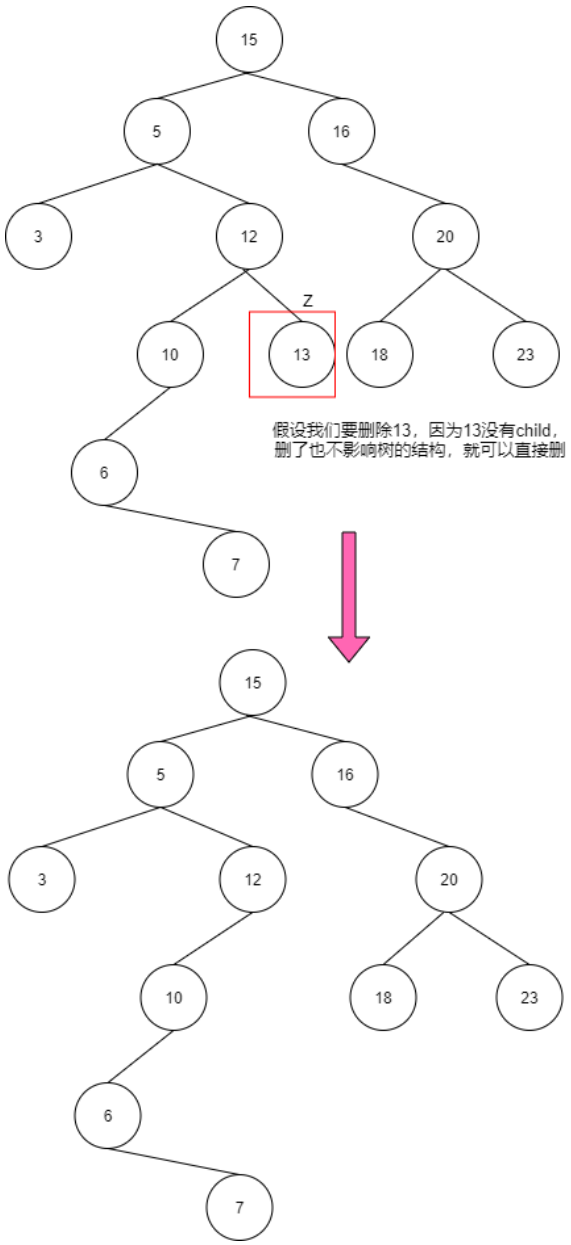


  - pseudocode
- insert
  - basic idea
    - if key[x] < v, move to the right child of x, else move to the left child of x
    - when x is NIL, we found the correct position
    - let y be parent of x, if v < key[y] insert the new node as y's left child else insert it as y's right child
    - beginning at the root, go down the tree and maintain
      - pointer x: traces the downward path (current node)
      - pointer y: parent of x ("trailing pointer")
  - example

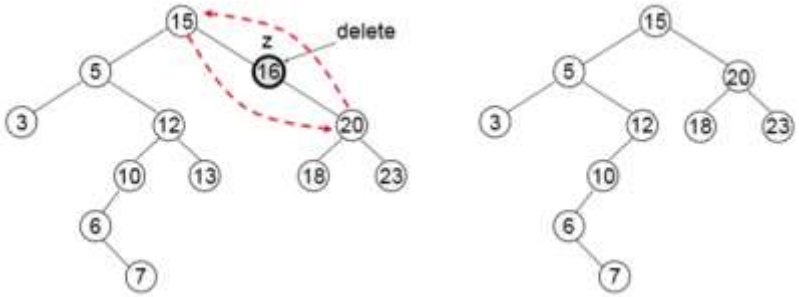


- pseudocode
- runtime analysis

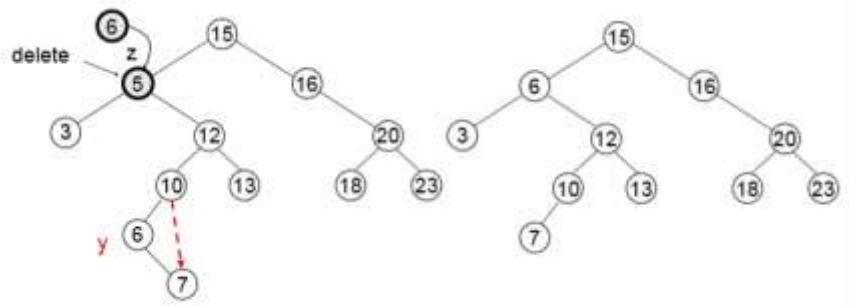
- delete
  - 这个就比较复杂了，得分好多种情况讨论
    - z has no child



    - z has one child，这种情况往上接就可以了



- z has two children
  - if a node in a binary search tree has two children, then its successor has no left child and its predecessor has no right child.
  - 很简单，也就是先去找 z 的 successor 就行了，也就是它的 right subtree 的最小值，然后替换到 z 的 value，z 的 successor 要么没有 child，要么最多只有一个，如果有一个 child，就直接往上接



  - code
- Running time of basic operations on binary search tress
  - average is θ(logn), the expected height of the tree is logn
  - worst case is θ(n), the tree is a linear chain of n nodes (very unbalanced)

lecture 7

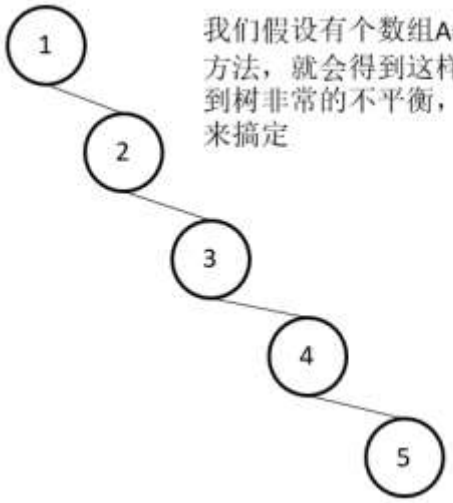
AVL (Adelson-Velskii and Landis) tree

为啥需要这玩意呢？就是 Binary search tree 的时间复杂度都是取决于它的 depth，也就是树的高度（height），我们的目标就是让树的 height 变得尽可能小，做法就是让树变得更加 balance，我们有三种做法，AVL tree，red-black tree，还有 B-trees

Operations on binary search trees:

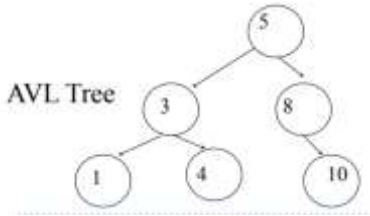
SEARCH	O(h)
PREDECESSOR	O(h)
SUCCESOR	O(h)
MINIMUM	O(h)
MAXIMUM	O(h)
INSERT	O(h)
DELETE	O(h)

- 举例，这种不平衡的树，对插入影响不大，但是对查询影响大

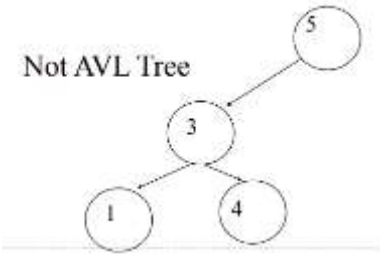


AVL - Good but not perfect balance

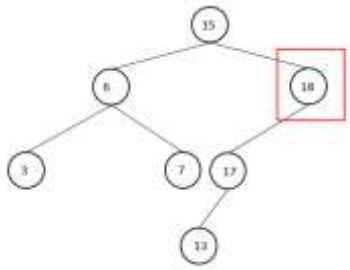
- AVL trees are height balanced binary search trees where the height of the two subtrees of a node differs by at most one
- Balance factor of a node: height (left subtree) - height (right subtree)
  - The balance factor of every node may be -1, 0, 1
  - all leaves will have balance factor of 0
- An AVL tree has balance factor calculated at every node
  - For every node, heights of left and right subtree can differ by no more than 1. 这也就是说对于每一个节点来说，它的左右两个子树的高度差的绝对值不超过 1 。
  - example
    - 这是 AVL tree



    - 这就不是了



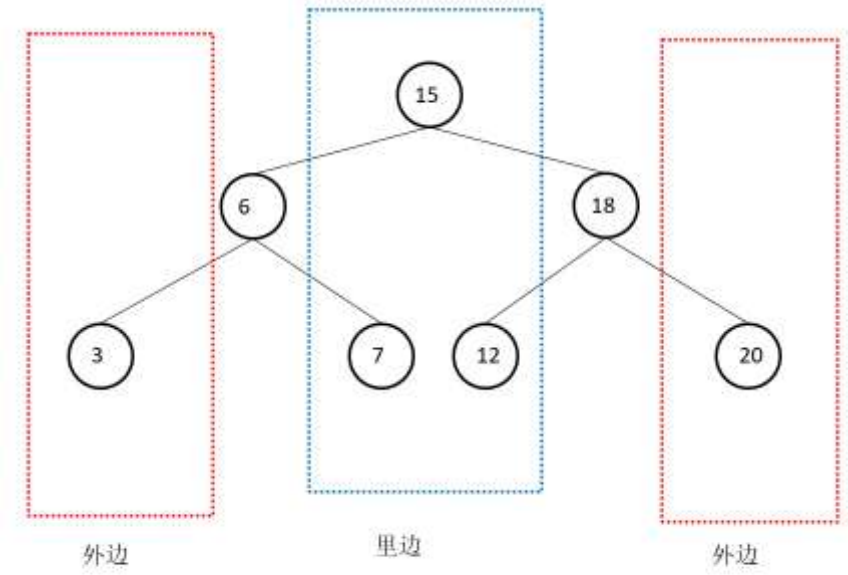
    - 这个也不是 AVL 树，你看红色框框的那个节点，它的左右 subtree 的差值就为 2 了



那怎么实现呢？通过 rotation

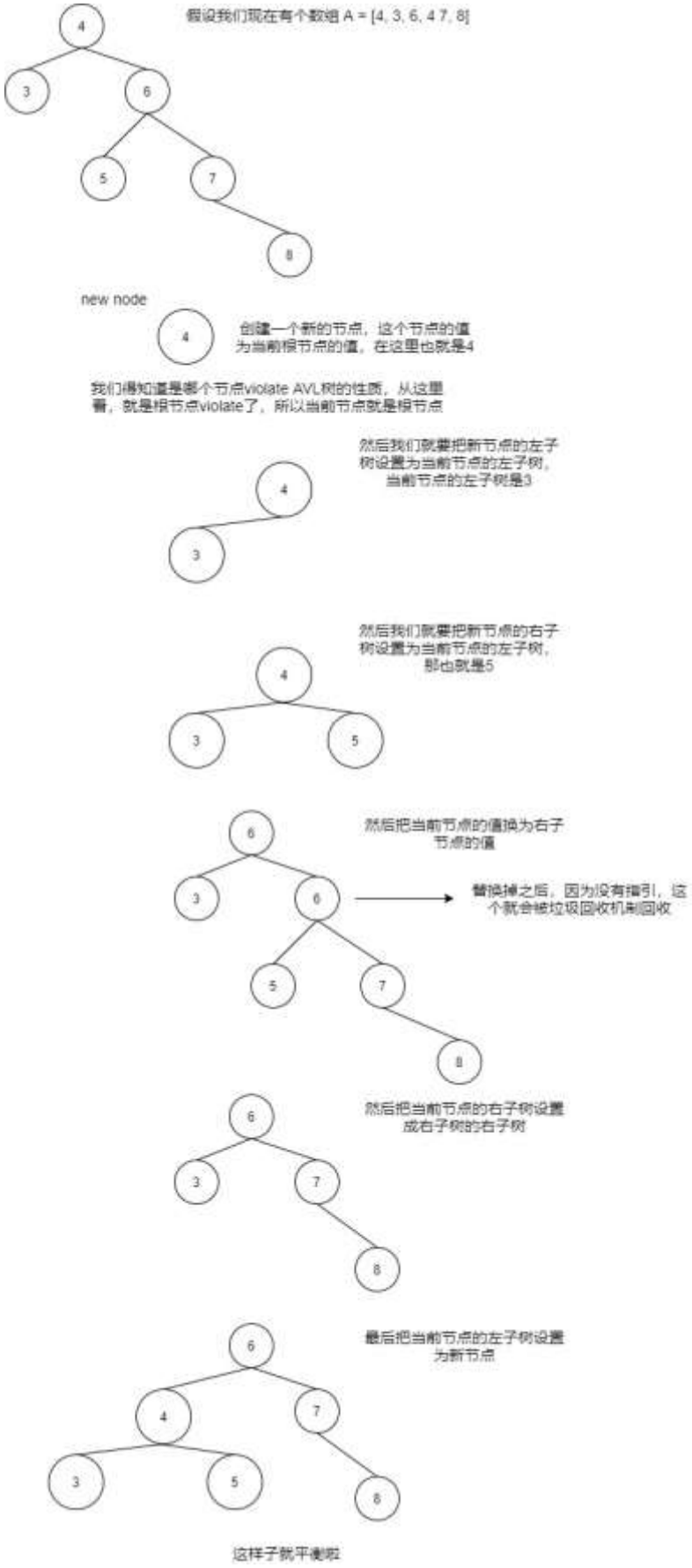
- 就是在我们插入或者删除一个节点的时候，我们就会改变平衡二叉树的结构，

  本来 balance factor 是 1 的，因为每次插入或者删除一个节点时，造成 balance factor 就正好为 2，因为仅仅只是插入或删除了一个节点
- single rotation
  - 什么情况是 single rotation 呢？



当外边的高度减去里边的高度>1时，就用single rotation

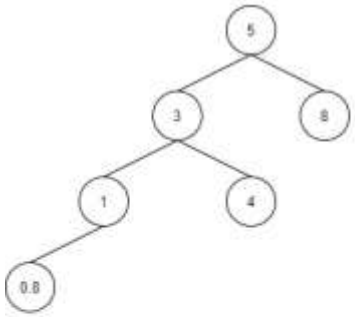
  - Insertion in the left subtree of the left child of U
  - Insertion in the right subtree of the right child of U
- left rotate
  - 当最右外边的比较长的时候就用 left rotate
  - 创建一个新的节点 newnode，newnode 的值就是当前根节点的值
  - 把 newnode 的左子树设置为当前节点的左子树，那哪个是当前节点呢？就是那个节点的 balance factor > 1 的那个节点
  - 把 newnode 的右子树设置为当前节点的右子树的左子树
  - 把当前节点的值换为右子节点的值
  - 把当前节点的右子树设置为右子树的右子树
  - 把当前节点的左子树设置为新节点
  - 这个就是 left rotate 的每一步



- 代码实现
- 时间复杂度

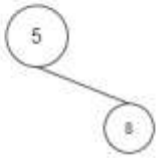
- right rotate
  - 当最左外边的长度较长的时候就用 right rotate

- 创建一个新的节点，newnode，设置这个 newnode 值为当前根节点的值
- 把新节点的右子树设置为当前节点的右子树
- 把新节点的左子树设置为当前节点的左子树的右子树
- 把当前节点的值换为左子节点的值
- 把当前节点的左子树设置成左子树的左子树
- 把当前节点的右子树设置为新节点
- 这就是 right rotate 的每一步

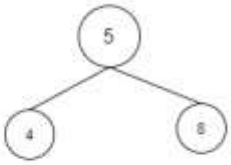


首先我们得先看是哪一个节点violate balance factor，很明显是根节点违反了，对吧，所以当前节点就是根节点

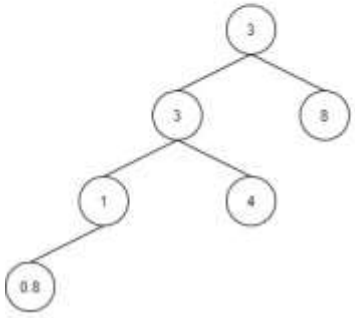
创建一个新的节点，newnode，这个newnode的值要等于当前根节点的值



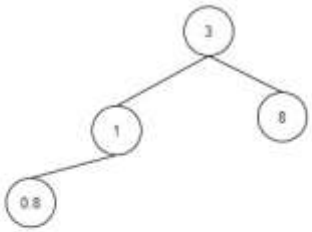
把新节点的右子树设置为当前节点的右子树，当前节点的右子树是数字8



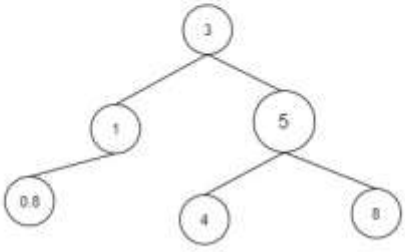
把新节点的左子树设置为当前节点的左子树的右子树，当前节点的左子树的右子树是数字4



把当前节点的值换成左子节点的值



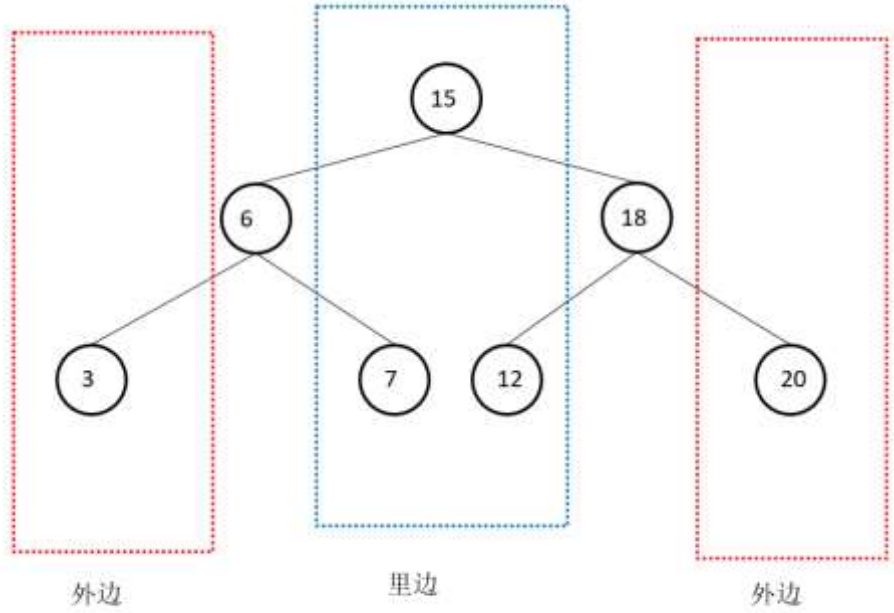
把当前节点的左子树设置成成左子树的左子树



把当前节点的右子树设置成为新节点

完成

    - 代码实现
    - 时间复杂度
- double rotation
    - 什么情况是 double rotation 呢？当中间长的时候就得用 double rotation



当里边的高度减去外边的高度>1时，就用double rotation

    - Insertion in the left subtree of the right child of U
    - Insertion in the right subtree of the left child of U
    - 实现步骤
        - 先左后右或者先右后左

time complexity

- for a AVL tree, the insertion, deletion, and search is O(logn)
- worst case tree height for AVL tree with n nodes is θ（logn）
- A single restructure/rotation is O(1)

red-black tree

B-trees

Graph Basics

Graphs

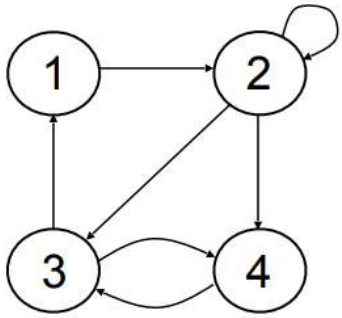
why do we need graph

- 线性表局限于一个直接前驱和一个直接后继的关系
- 树叶只能有一个直接前驱，也就是父节点

- 当我们需要表示多对多的关系时，就用到了图

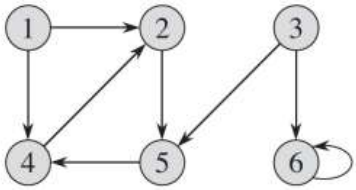
Definition

- A set of vertices (nodes) with edges (links) between them
- G = (V, E) - graph
- V = set of vertices
- E = set of edges = subset of V x V
- Thus |E| = O(|V|^2)
- example

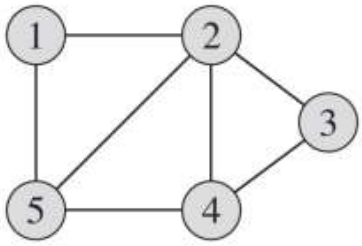


Directed Graphs (Digraph) VS. Undirected Graph

- Directed Graphs (digraphs) (ordered pairs of vertices)



  - in-degree of v: # of edges entering v
  - out-degree of v: # of edges leaving v
  - v is adjacent to u if there is an edge (u,v)
- Undirected Graphs (unordered pairs of vertices)



  - degree of v: # of edges incident on v
  - v is adjacent to u and u is adjacent to v is if there is an edge between v and u

More Graph variations

- A weighted graph associates weights with either the edges or the vertices. 比如说，

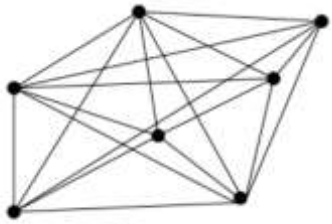
  a road map: edges might be weighted with distance

- A multigraph allows multiple edges between the same pair of vertices. 比如说，

  the call graph in a program (a function can get called from multiple points in another function)

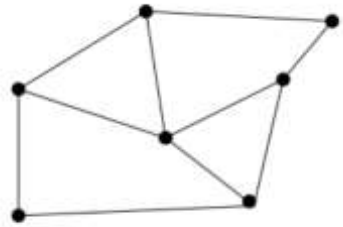
Sparse VS. Dense Graphs

- We will typically express running times in terms of |E| and |V|
  - if |E| ≈ |V|^2 ,the graph is dense. It has a quadratic number of edges



dense

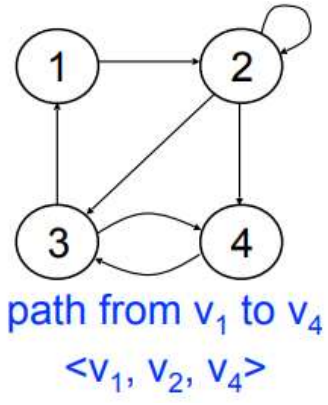
  - if |E| ≈ |V|, the graph is sparse. Linear in size, only a small fraction of the possible number of vertex pairs actually have edges defined between them



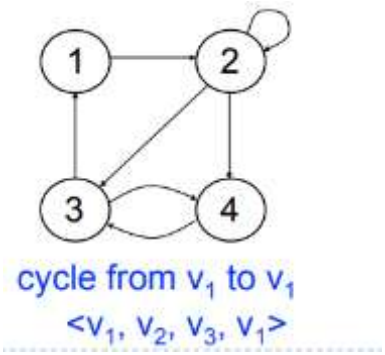
sparse

Terminology

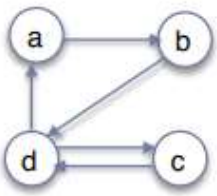
- Complete graph
  - A graph with an edge between each pair of vertices. 就是每个 node 都能互相连接上
- subgraph
  - A graph (V', E') such that V'∈ V and E' ∈ E
- Path from v to w
  - A sequence of vertices < V0, v1,....., Vk> such that V0 = V and Vk = w
  - example



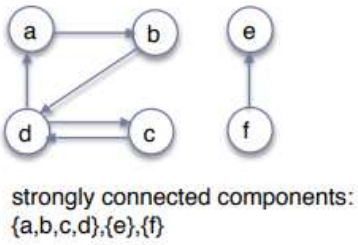
- Length of a path
  - Number of edges along the path
- w is reachable from v
  - if there is a path from v to w
- simple path
  - all the vertices in the path are distinct
- Cycles
  - A path <V0, V1, ...Vk> forms a cycle if V0 = Vk and K ≥ 2
  - example



- Acyclic graph
  - A graph without any cycles
- special case： Tree
- Strongly connected VS Connected
  - Directed Graphs
    - Strongly connected
      - 任意两个 vertices 都能互相连通
      - example

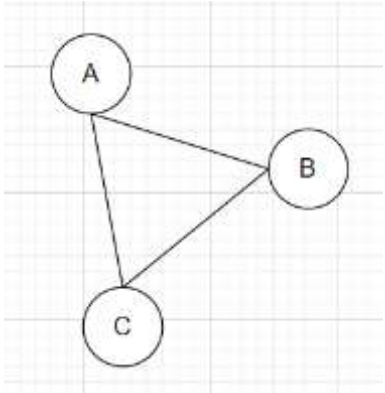


    - Strongly connected components
      - 所有的能够彼此连通的 subgraphs
      - exmaple

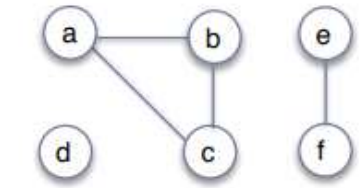


  - Undirected Graphs
    - connected
      - 任意两点都连通

- example



- connected components
  - 所有的能够 connected 的 subgraph
  - example



connected components:
{a,b,c},{d},{e,f}

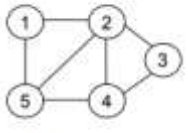
Representing Grpahs

- 假设有|V| 个 vertices，那么，这个矩阵的大小为 V * V，怎么进行表示呢？

  如果之间有连通，值为 1，如果没有连通，值为 0。我们就可以根据这个写一个矩阵，称为邻接矩阵，adjacency matrix，邻接矩阵就是用二维数组进行表示的

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$$

- example，这是 undirected 的



Undirected graph

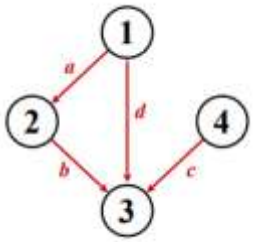
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

For undirected graphs, matrix A is symmetric:

$a_{ij} = a_{ji}$

$A = A^T$

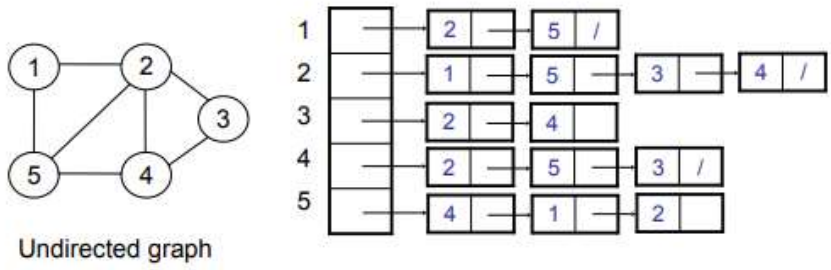
- example，这是 directed 的



directed graph

	1	2	3	4
1	0	1	1	0
2	0	0	1	0
3	0	0	0	0
4	0	0	1	0

- properties of adjacency Matric representation
  - Memory required
    - θ(V^2), independent on the number of edges in G
  - Preferred when
    - The graph is dense: |E| is close to |V|^2
    - We need to quickly determine if there is an edge between two vertices
  - Time to determine if (u,v) ∈ E: θ（1）
  - Disadvantage： no quick way to determine the vertices adjacent to a vertex
  - Time to list all vertices adjacent to u is θ（v）
- Graph adjacency list
  - Adjacency list representation of G = (V, E)
    - An array of |V| lists, one for each vertex in V
    - Each list Adj[u] contains all the vertices v that are adjacent to u (i.e., there is an edge from u to v)
    - can be used to both directed and undirected graphs
    - 邻接表的实现只关心存在的边，不关心不存在的边，因此没有空间浪费。而邻接矩阵，用 0 和 1 进行表示，不管有没有边都要进行表示，就会造成一定的空间浪费
  - example，数组+链表



Undirected graph

  - properties of Adjacency list representation
    - Sum of "lengths" of all adjacency lists
      - Directed graph: |E|
        - edge(u,v) appears only once (i.e., in the list of u)
      - Undirected graph：2|E|
        - edge（u，v）appears twice (i.e., in the lists of both u and v)
    - memory required: θ（V+E）
    - preferred when
      - The graph is sparse: |E| << |V|^2
      - we need to quickly determine the nodes adjacent to a given node

- disadvantage
  - no quick way to determine whether there is an edge between node u and v
- Time to determine if (u,v) ∈ E is： θ（degree(u)）
- Time to list all vertices adjacent to u:θ（degree(u)）

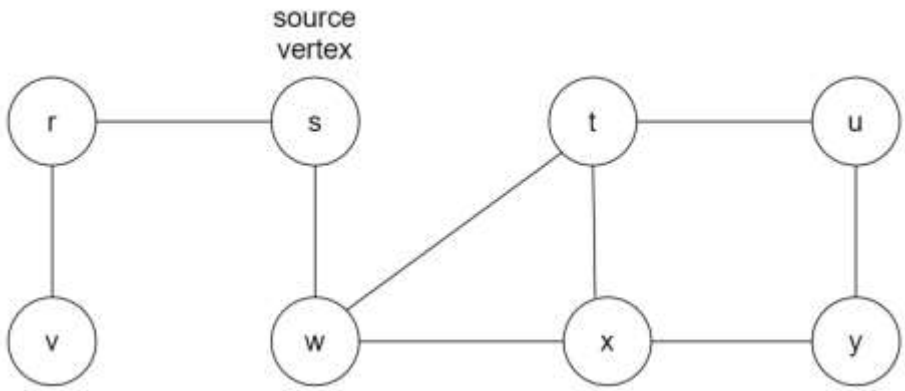
Graph search

Given: a graph G = (V,E), directed or undirected

- In general, given a vertex s, we want to locate some vertex t, it means find a path in G
- We want to visit all vertices in a "local" organized manner

Breadth-First Search (BFS)

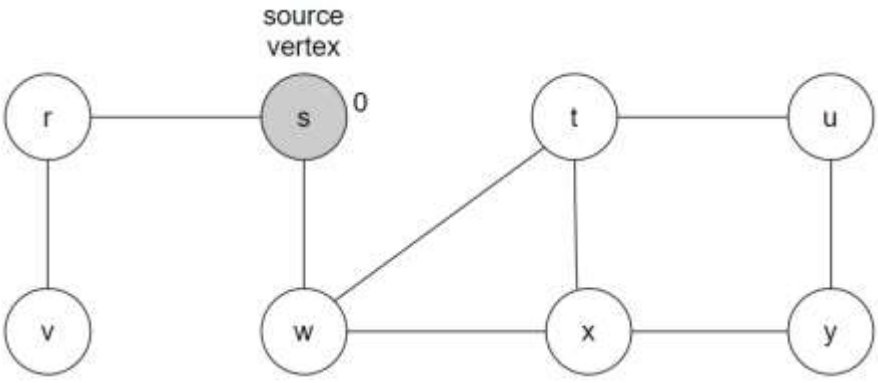
- BFS 过程
  - 这就类似于一个分层搜索的过程，BFS 需要用到一个队列来保存访问过的节点的顺序，一遍按照这个顺序来访问这些节点的邻接节点。
- BFS 图解
  - 在这里呢，教授用黑白灰三种颜色来表示，若是白色，表示节点没有被访问。若是灰色，表示已经被访问，但是还在继续向下探索。若是黑色，表示正式结束
  - 假设我们现在有这么一幅图，我们从 S 开始



  - 这便是这幅图的邻接矩阵

	S	R	T	U	V	W	X	Y
S	0	1	0	0	0	1	0	0
R	1	0	0	0	1	0	0	0
T	0	0	0	1	0	1	1	0
U	0	0	1	0	0	0	0	1
V	0	1	0	0	0	0	0	0
W	1	0	1	0	0	0	1	0
X	0	0	1	0	0	1	0	1
Y	0	0	0	1	0	0	1	0

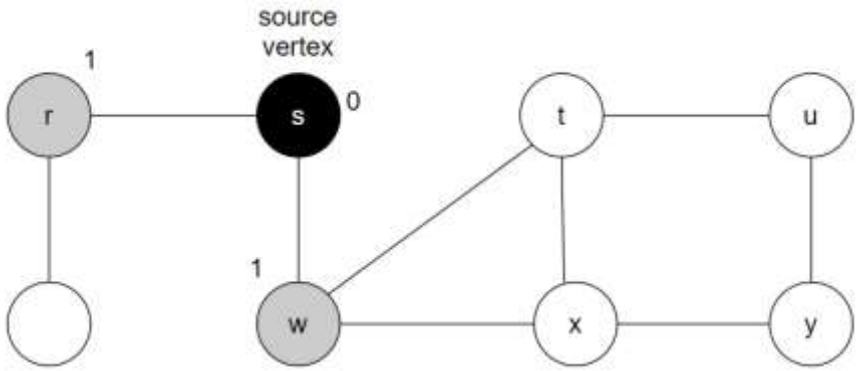
- S 被访问，变为灰色，接下来就得去邻接矩阵那里看看有哪些点是可以走的



- 我们发现，R 和 W 都是可以走的，于是我们就按字母顺序进入队列

	S	R	T	U	V	W	X	Y
S	0	1	0	0	0	1	0	0
R	1	0	0	0	1	0	0	0
T	0	0	0	1	0	1	1	0
U	0	0	1	0	0	0	0	1
V	0	1	0	0	0	0	0	0
W	1	0	1	0	0	0	1	0
X	0	0	1	0	0	1	0	1
Y	0	0	0	1	0	0	1	0

- 因为 S 已经把 R，W 扩展了，S 的下一个 level 的节点就已经没有了，就变成了黑色。R, w 进队列，接着就看看队列的头元素是不是可以继续扩展

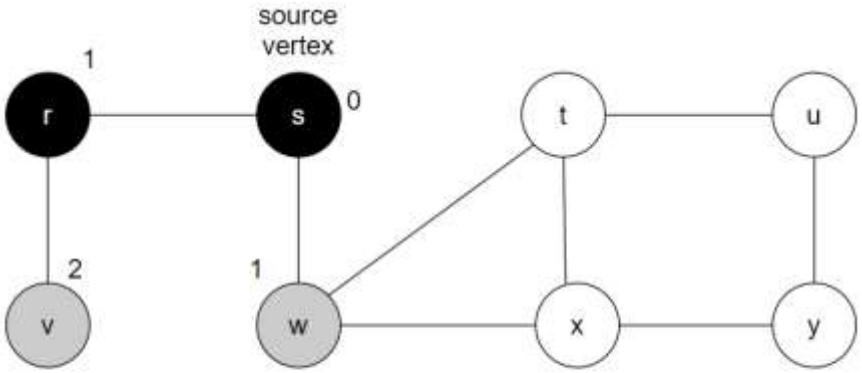


Queue: r, w　　S出队列，r, w进队列

- 我们发现 R 可以到 S 和 V，但是 S 已经被访问过啦，于是乎队列进 v，把 R 弹出来

	S	R	T	U	V	W	X	Y
S	0	1	0	0	0	1	0	0
R	1	0	0	0	1	0	0	0
T	0	0	0	1	0	1	1	0
U	0	0	1	0	0	0	0	1
V	0	1	0	0	0	0	0	0
W	1	0	1	0	0	0	1	0
X	0	0	1	0	0	1	0	1
Y	0	0	0	1	0	0	1	0

- 因为 R 和 W 是同一层的，所以还得继续探索 W 节点

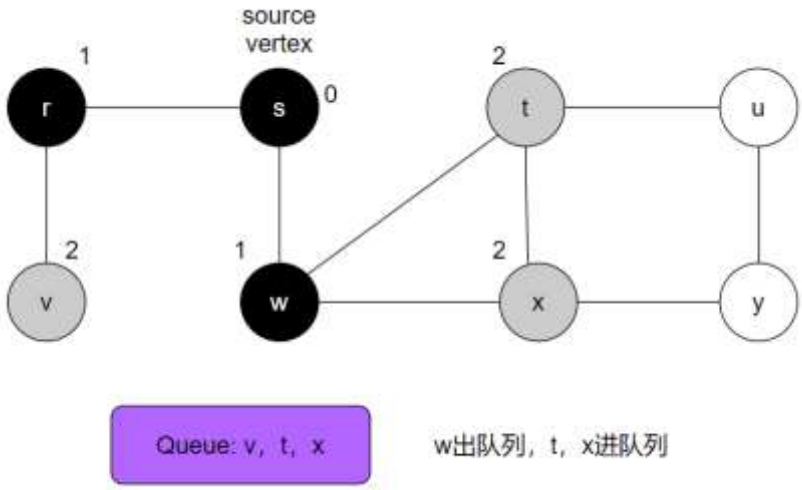


Queue: w, v　　r出队列，v进队列

- 我们发现 T， X 节点是可以走的

	S	R	T	U	V	W	X	Y
S	0	1	0	0	0	1	0	0
R	1	0	0	0	1	0	0	0
T	0	0	0	1	0	1	1	0
U	0	0	1	0	0	0	0	1
V	0	1	0	0	0	0	0	0
W	1	0	1	0	0	0	1	0
X	0	0	1	0	0	1	0	1
Y	0	0	0	1	0	0	1	0

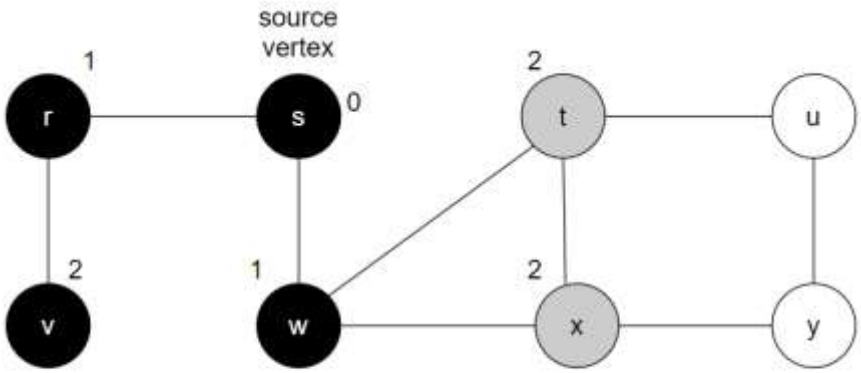
- 访问 t, x,变为灰色，因为 w 已经被探索结束了，所以就变成了黑色。接着我们又继续从队列的头元素开始探索



- 我们发现 R 这个点已经被访问啦，于是 V 已经无路可走了

	S	R	T	U	V	W	X	Y
S	0	1	0	0	0	1	0	0
R	1	0	0	0	1	0	0	0
T	0	0	0	1	0	1	1	0
U	0	0	1	0	0	0	0	1
V	0	i	0	0	0	0	0	0
W	1	0	1	0	0	0	1	0
X	0	0	1	0	0	1	0	1
Y	0	0	0	1	0	0	1	0

- 因为 v 已经没有其他路可以走了，于是就变成黑色的了，接着，我们就看队列的头部元素，也就是 t，看看 t 可以去哪里

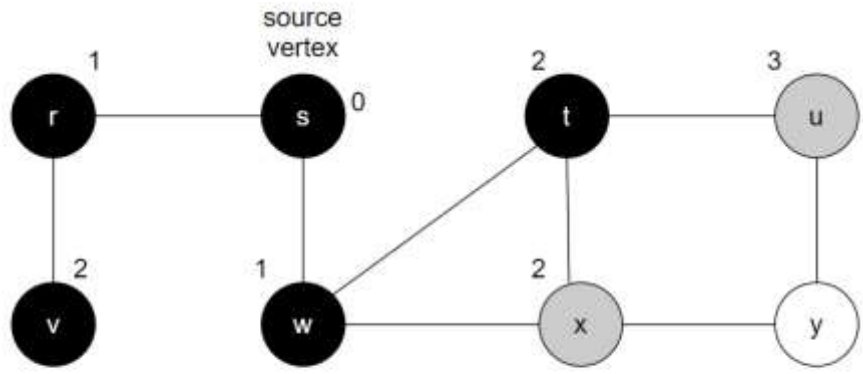


Queue: t，x　　v出队列

- 我们可以看到 T 是可以到 u，w，x 的，但是 w 和已经被访问了，然后 U 是没有被访问的，于是就把 U 加入队列

	S	R	T	U	V	W	X	Y
S	0	1	0	0	0	1	0	0
R	1	0	0	0	1	0	0	0
T	0	0	0	1	0	1	1	0
U	0	0	1	0	0	0	0	1
V	0	1	0	0	0	0	0	0
W	1	0	1	0	0	0	1	0
X	0	0	1	0	0	1	0	1
Y	0	0	0	1	0	0	1	0

- t 已经探索结束了，变为黑色，继续从队列的头元素探索，也就是 x

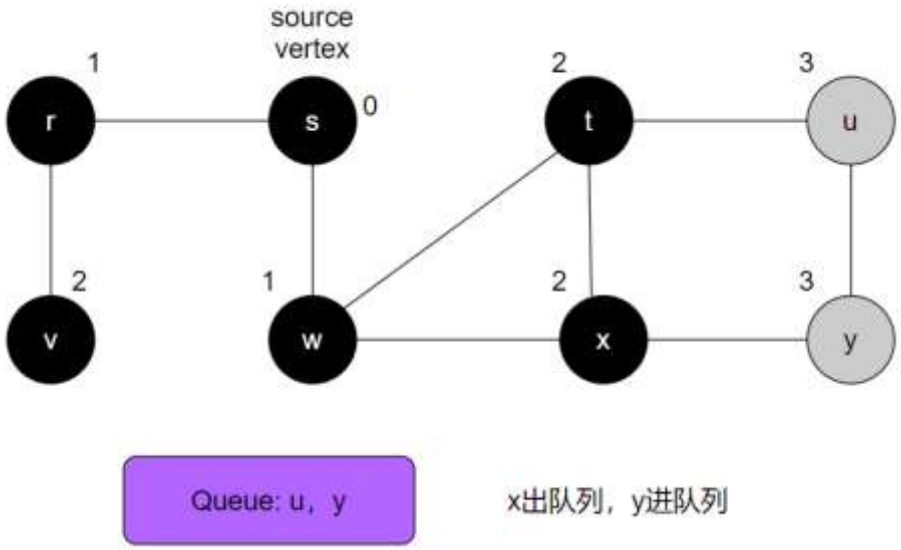


Queue: x，u　　t出队列，u进队列

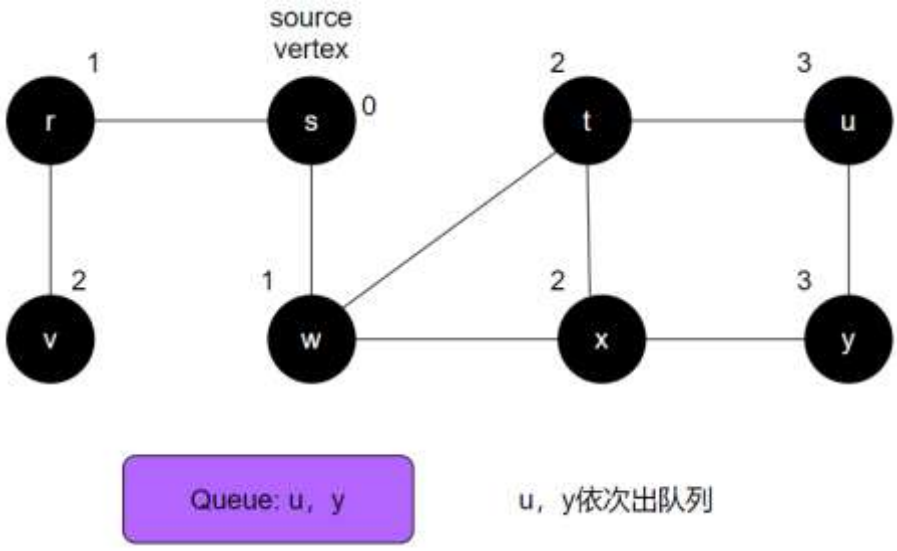
- x 可以到 y

	S	R	T	U	V	W	X	Y
S	0	1	0	0	0	1	0	0
R	1	0	0	0	1	0	0	0
T	0	0	0	1	0	1	1	0
U	0	0	1	0	0	0	0	1
V	0	1	0	0	0	0	0	0
W	1	0	1	0	0	0	1	0
X	0	0	1	0	0	1	0	1
Y	0	0	0	1	0	0	1	0

- 现在，x 也已经变成黑色的了，现在我们就只剩下 u 和 y 了，他们也没路可以走了，按照队列的顺利依次弹出



- 依次出队列

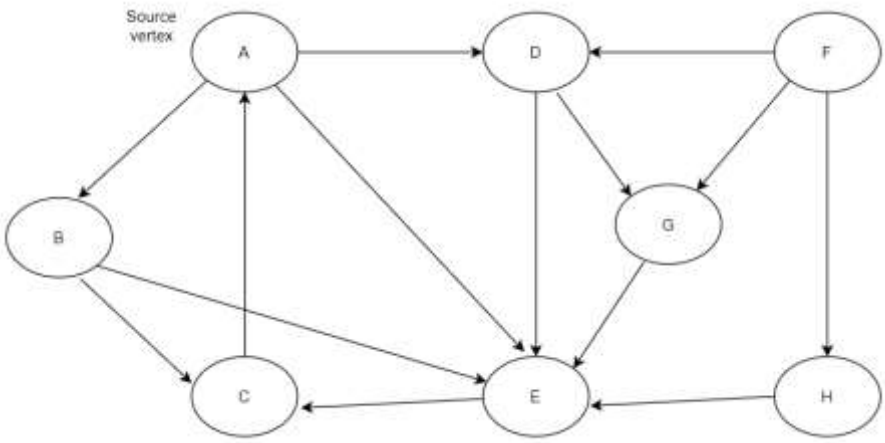


- 最终，我们得到的顺序就是 S->r->w->v->t->x->u->y

- BFS 性质
    - 我们能用 BFS 来算最短路径，前提就是没有权重，也就最少边。
        - shortest-path distance δ（s , v）= minimum number of edges from s to v, or ∞ if v not reachable from s. 也就是这个公式 d(v) = δ(s, v)
    - 然后用 BFS 来算最短路径的话，时间复杂度是 O(V+E)，因为得走 V 个 vertex 和走 E 个 edges。空间复杂度是 O(V)，就是用 queue 来存储节点
    - 其次就是离 source vertex 近的节点一定比离得远的节点先被访问到
    - Complexity of BFS on a dense graph is θ(E)

Depth-First Search (DFS)

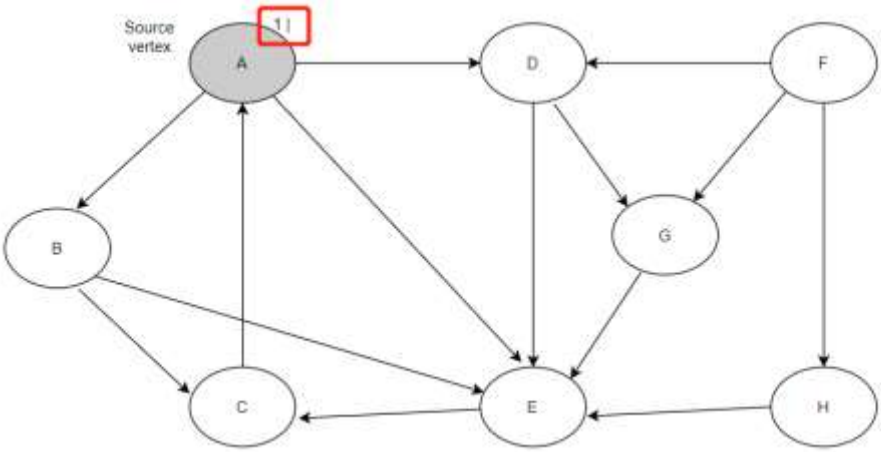
- DFS 过程
    - 从初始访问节点出发，初始访问节点可能有多个邻接节点，DFS 的策略就是先访问第一个邻接节点，那怎么判断这个邻接节点呢？我们的数据不是用连接矩阵或者是邻接表表示吗，我们就用这个来遍历，从而知道下一个邻接点在哪。
    - 找到邻接点之后，我们就把这个邻接点当做是初始节点，（也就表示可以用递归），我们也就再接着访问他的第一个邻接节点。也就是说每次都在访问完当前节点后首先访问当前节点的第一个邻接节点
    - 当没有下一个节点的时候，就需要 backtrack 回上一个节点，继续探索。 backtrack 的意思就是从哪来的回哪去
    - DFS 就是纵向挖掘深入，而不是对一个节点的所有临街节点进行横向访问
- DFS 图解
    - 在这里呢，教授用黑白灰三种颜色来表示，若是白色，表示节点没有被访问。若是灰色，表示已经被访问，但是还在继续向下探索。若是黑色，表示正式结束
    - 假设我们有这么一张图，而且是有向图，白色表示还没有被访问



        - 邻接矩阵

	A	B	C	D	E	F	G	H
A	0	1	0	1	1	0	0	0
B	0	0	1	0	1	0	0	0
C	1	0	0	0	0	0	0	0
D	0	0	0	0	1	0	1	0
E	0	0	1	0	0	0	0	0
F	0	0	0	1	0	0	1	1
G	0	0	0	0	1	0	0	0
H	0	0	0	0	1	0	0	0

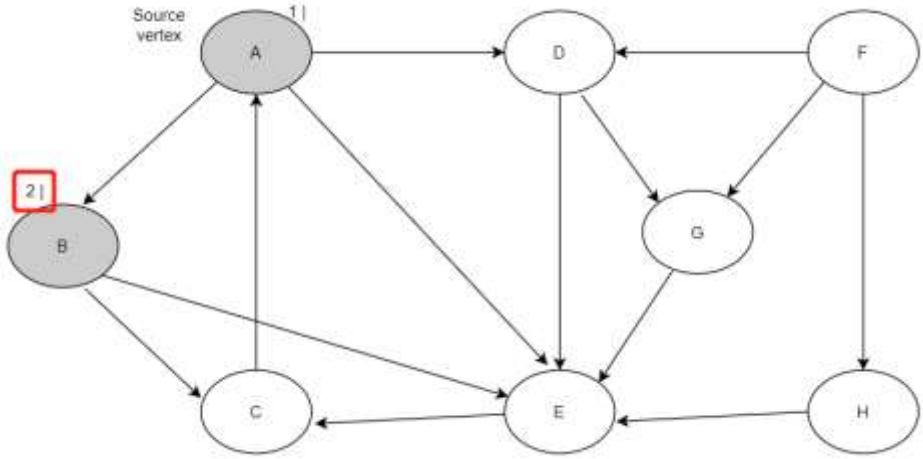
- 我们就先去访问 A 节点，红色框框表示的是顺序，接下来我们就去看邻接矩阵能到达哪一个节点，也就是看下一个节点时哪里



- 从邻接矩阵中看到，我们可以去访问 B

	A	B	C	D	E	F	G	H
A	0	1	0	1	1	0	0	0
B	0	0	1	0	1	0	0	0
C	1	0	0	0	0	0	0	0
D	0	0	0	0	1	0	1	0
E	0	0	1	0	0	0	0	0
F	0	0	0	1	0	0	1	1
G	0	0	0	0	1	0	0	0
H	0	0	0	0	1	0	0	0

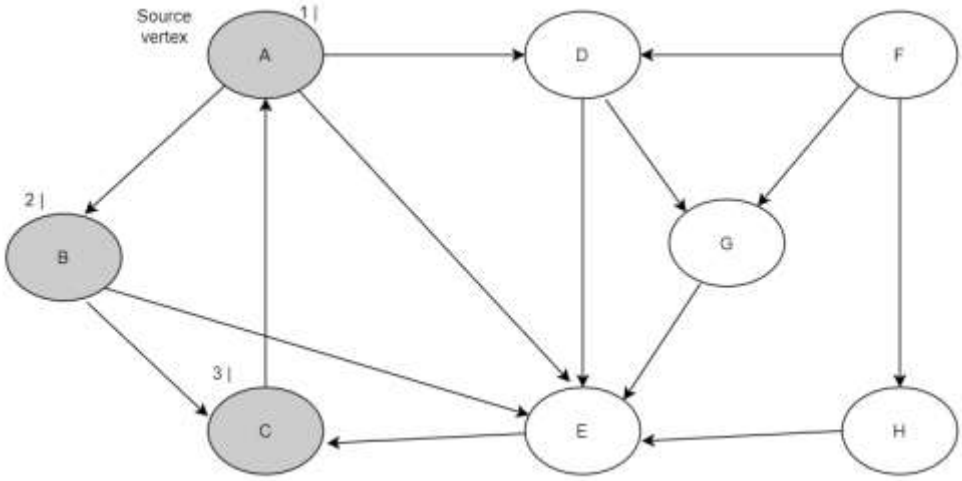
- 我们就访问到了 B 节点，B 节点要变成灰色，继续看邻接矩阵



- 我们可以访问 C

	A	B	C	D	E	F	G	H
A	0	1	0	1	1	0	0	0
B	0	0	1	0	1	0	0	0
C	1	0	0	0	0	0	0	0
D	0	0	0	0	1	0	1	0
E	0	0	1	0	0	0	0	0
F	0	0	0	1	0	0	1	1
G	0	0	0	0	1	0	0	0
H	0	0	0	0	1	0	0	0

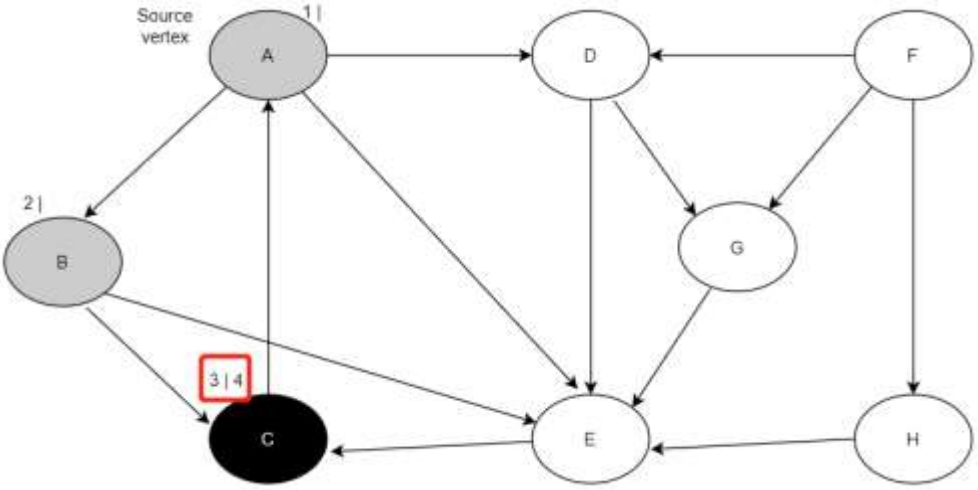
- 访问了 C, C 变成了灰色，继续看邻接矩阵



- 发现 C 可以去 A，但是 A 的颜色是灰色，表示已经访问过了，此时该怎么办呢？我们就要 backtrack，退回一步，上一个访问的节点是 B，我们就要去看看 B 还可以去哪里。因为 C 已经哪里都不能访问了，就要变成黑色的了，表示已经访问完毕

	A	B	C	D	E	F	G	H
A	0	1	0	1	1	0	0	0
B	0	0	1	0	1	0	0	0
C	1	0	0	0	0	0	0	0
D	0	0	0	0	1	0	1	0
E	0	0	1	0	0	0	0	0
F	0	0	0	1	0	0	1	1
G	0	0	0	0	1	0	0	0
H	0	0	0	0	1	0	0	0

- C 就已经变成了黑色，此时要退回去了

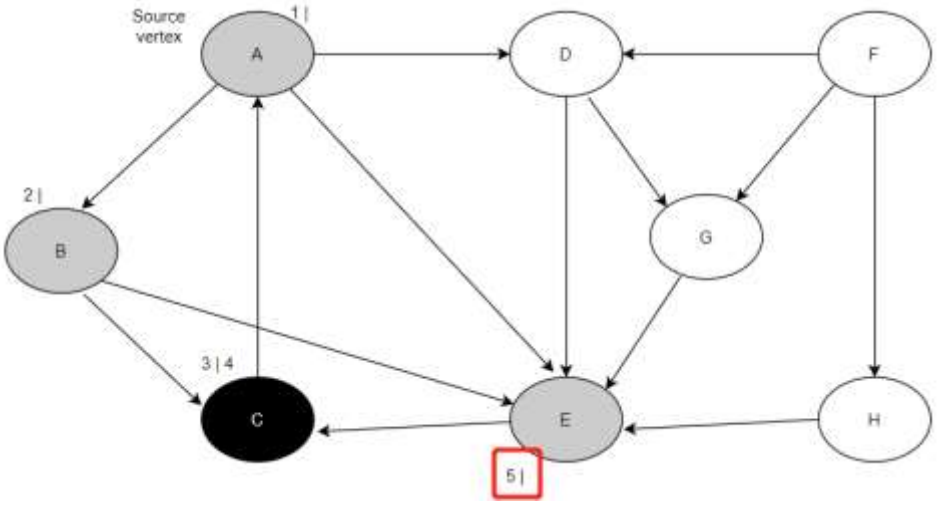


- backtrack 回去，看看 B 还能往哪里走，看到 B 可以去到 E

backtrack

	A	B	C	D	E	F	G	H
A	0	1	0	1	1	0	0	0
B	0	0	1	0	1	0	0	0
C	1	0	0	0	0	0	0	0
D	0	0	0	0	1	0	1	0
E	0	0	1	0	0	0	0	0
F	0	0	0	1	0	0	1	1
G	0	0	0	0	1	0	0	0
H	0	0	0	0	1	0	0	0

- E 的节点就要变成灰色，表示已经访问，接着看邻接矩阵

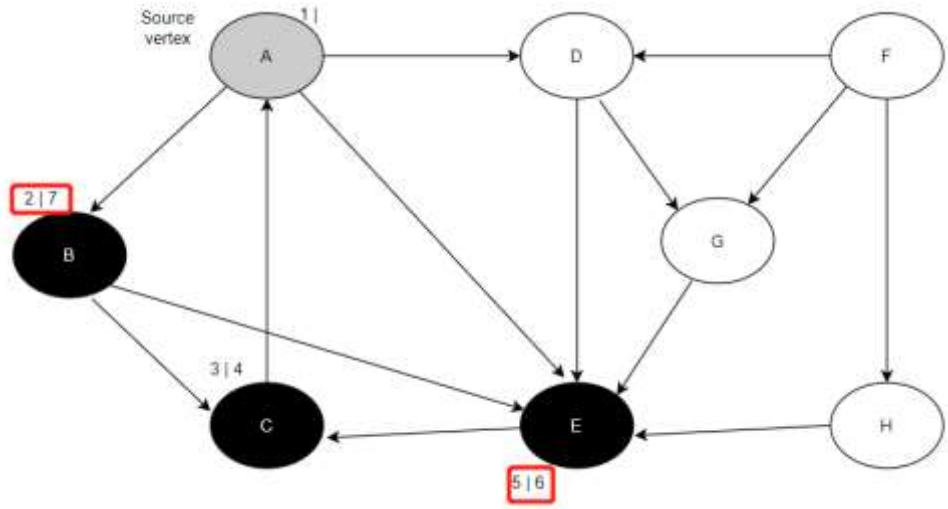


- 发现 E 可以去到 C，但是 C 是黑色的，表示已经访问过了，而且已经结束了，只能回去。B 的节点，其他地方也去不了了，又只能 backtrack 回 A，看看 A 能去哪里

backtrack

	A	B	C	D	E	F	G	H
A	0	1	0	1	1	0	0	0
B	0	0	1	0	1	0	0	0
C	1	0	0	0	0	0	0	0
D	0	0	0	0	1	0	1	0
E	0	0	1	0	0	0	0	0
F	0	0	0	1	0	0	1	1
G	0	0	0	0	1	0	0	0
H	0	0	0	0	1	0	0	0

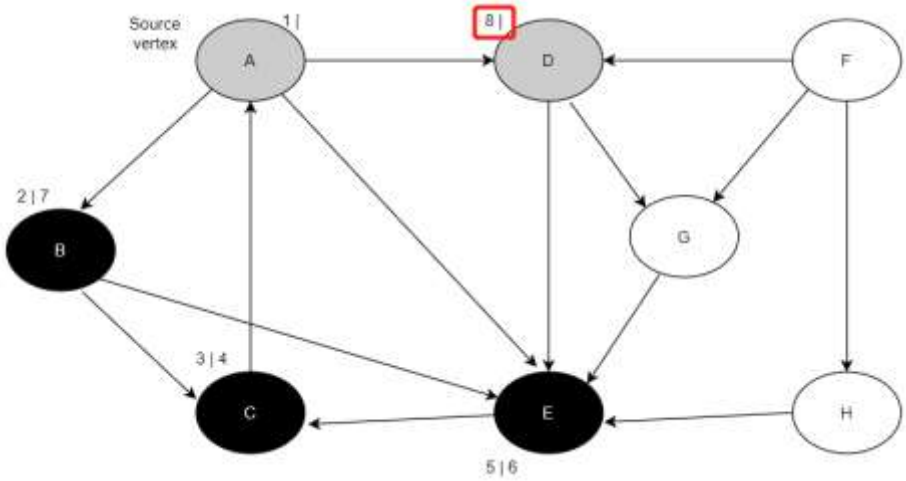
- E 和 B 的节点就要变黑啦，因为哪里也去不了啦，记得 update 这个序号，接下来就看看邻接矩阵，看看 A 能去哪里



- A 能去 D

	A	B	C	D	E	F	G	H
A	0	1	0	1	1	0	0	0
B	0	0	1	0	1	0	0	0
C	1	0	0	0	0	0	0	0
D	0	0	0	0	1	0	1	0
E	0	0	1	0	0	0	0	0
F	0	0	0	1	0	0	1	1
G	0	0	0	0	1	0	0	0
H	0	0	0	0	1	0	0	0

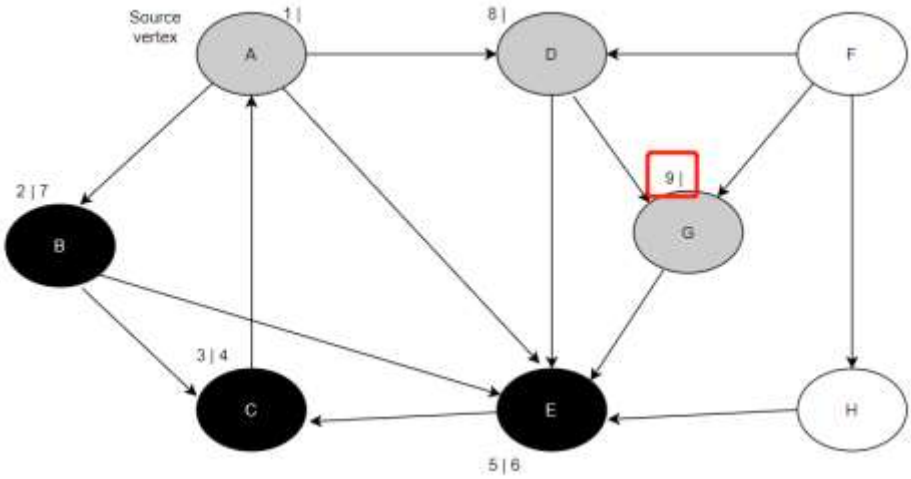
- 访问到 D，接下来就看看 D 能去哪里



- D 能访问到 E，但是 E 已经访问过啦，于是就去了 G

	A	B	C	D	E	F	G	H
A	0	1	0	1	1	0	0	0
B	0	0	1	0	1	0	0	0
C	1	0	0	0	0	0	0	0
D	0	0	0	0	1	0	1	0
E	0	0	1	0	0	0	0	0
F	0	0	0	1	0	0	1	1
G	0	0	0	0	1	0	0	0
H	0	0	0	0	1	0	0	0

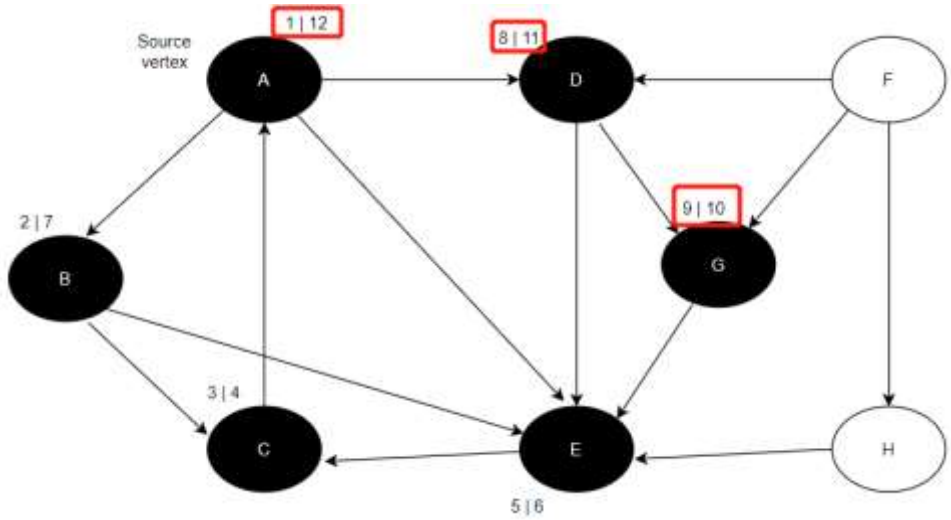
- 访问到 G，G 就变成了灰色，然后看看邻接矩阵



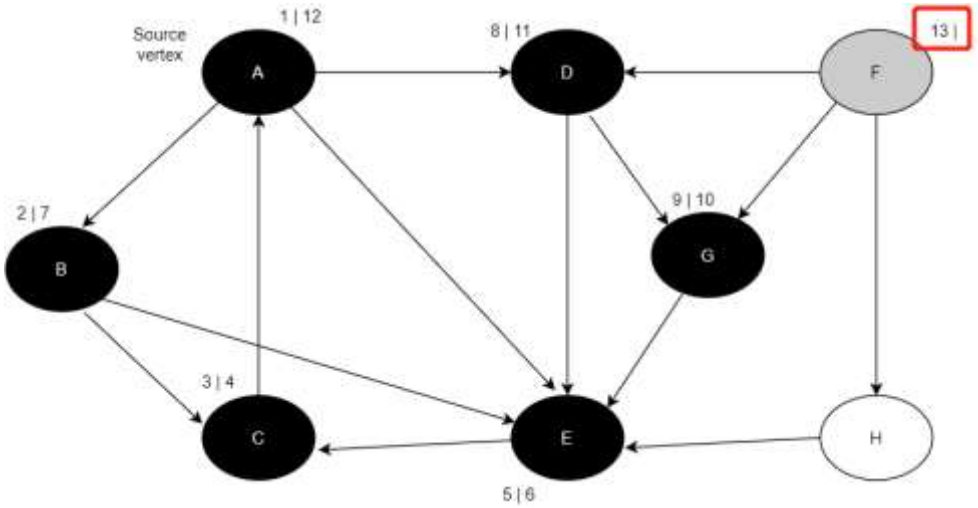
- G 能访问到 E，但是 E 已经被访问，于是 backtrack 到 D，但是 D 也已经没有节点可以访问了，就再次回到 A，接下来就看看 A 能去哪

		A	B	C	D	E	F	G	H
	A	0	1	0	1	1	0	0	0
backtrack	B	0	0	1	0	1	0	0	0
	C	1	0	0	0	0	0	0	0
	D	0	0	0	0	1	0	1	0
backtrack	E	0	0	1	0	0	0	0	0
	F	0	0	0	1	0	0	1	1
	G	0	0	0	0	1	0	0	0
	H	0	0	0	0	1	0	0	0

- G,D,A 都不能再访问其他节点了，但是又还有两个节点怎么办？我们这里是按照字母顺序的，于是我们就选 F 点再次开始遍历



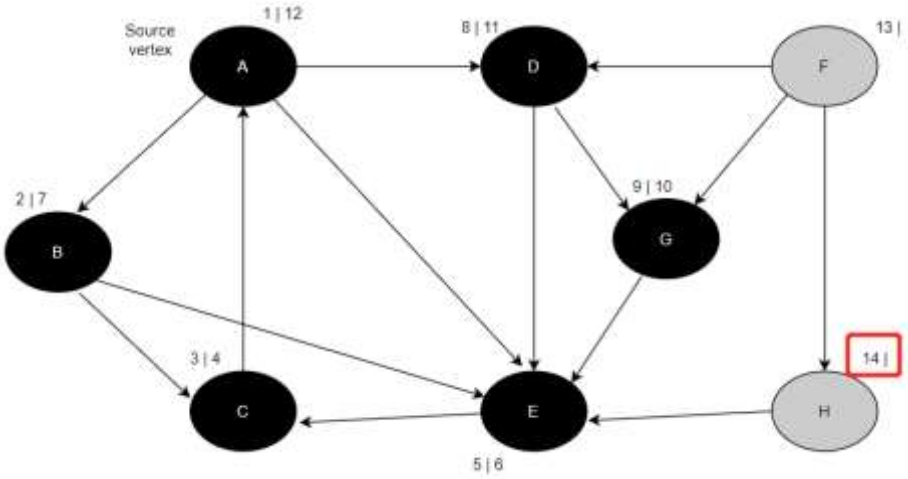
- 现在就再去看看邻接矩阵，看看能去哪



- F 能到 D，G，但是 D, G 都已经被访问了，所以就只能去 H 啦

	A	B	C	D	E	F	G	H
A	0	1	0	1	1	0	0	0
B	0	0	1	0	1	0	0	0
C	1	0	0	0	0	0	0	0
D	0	0	0	0	1	0	1	0
E	0	0	1	0	0	0	0	0
F	0	0	0	1	0	0	1	1
G	0	0	0	0	1	0	0	0
H	0	0	0	0	1	0	0	0

- 访问完 H，就看看 H 能去哪里

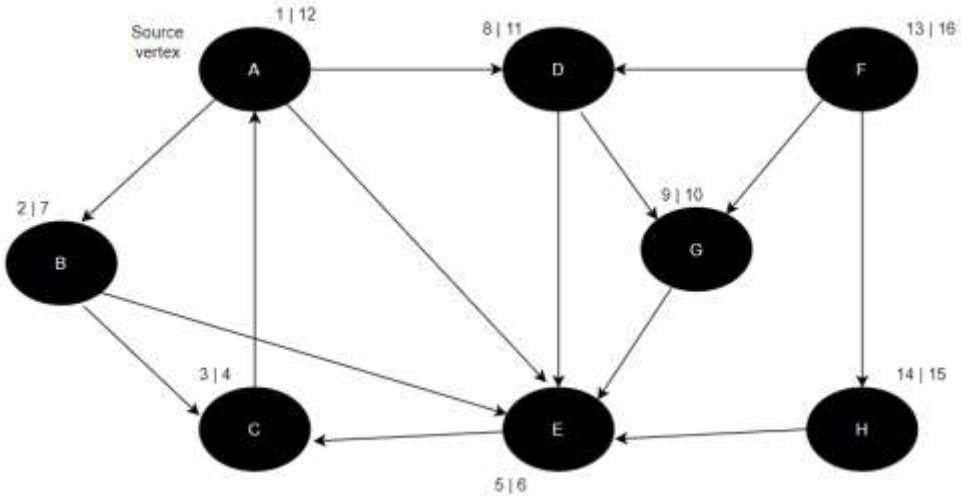


- H 能到 E，但是 E 已经被访问了，就只能 backtrack 回 F 啦

backtrack

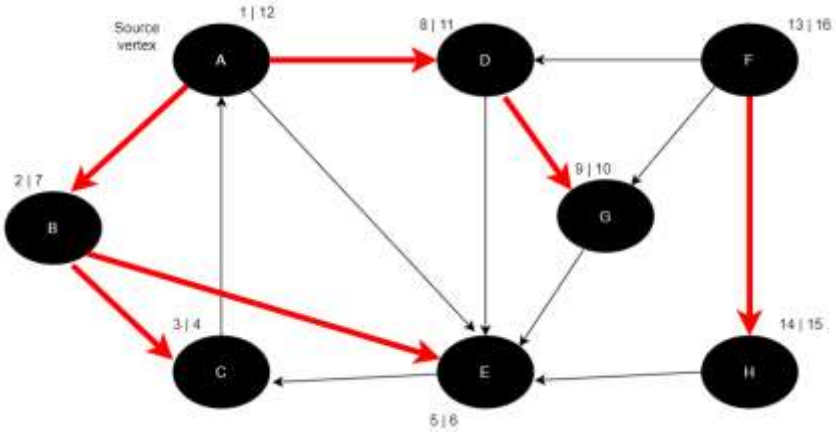
	A	B	C	D	E	F	G	H
A	0	1	0	1	1	0	0	0
B	0	0	1	0	1	0	0	0
C	1	0	0	0	0	0	0	0
D	0	0	0	0	1	0	1	0
E	0	0	1	0	0	0	0	0
F	0	0	0	1	0	0	1	1
G	0	0	0	0	1	0	0	0
H	0	0	0	0	1	0	0	0

- 回到 F，发现 F 也去不了哪里了，所有的节点都被访问啦，于是 DFS 就结束啦

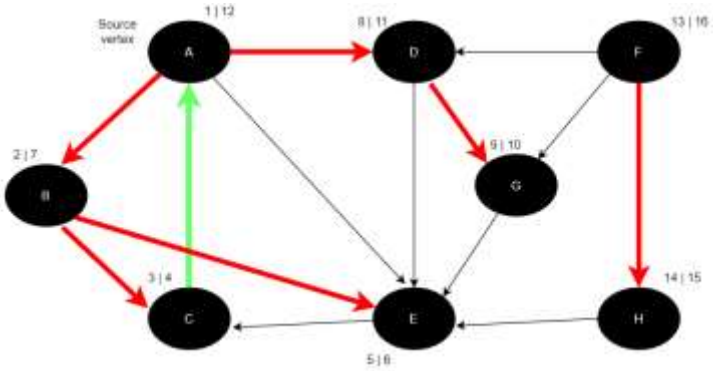


- 我们可以根据上面的数字顺序写出它的访问过程:
A->B->C->E->D->G->F->H

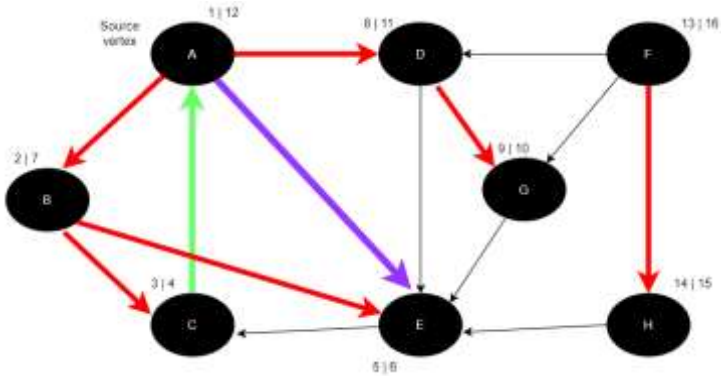
- Kinds of Edges: Edges 的种类
  - Tree edge
    - 就是按着 DFS 的所有访问顺序，凡是遇到白色的点，也就是没有被访问的点的那条边就是 tree edge
    - example，我们仔细想想，这些红色边，从 A->B, B 是白色的。B->C，C 是白色的。B->E，E 是白色的。A->D，D 是白色的。D->G，G 是白色的，F->H，H 是白色的。我们想想 A->E 那条线为什么不是？因为按照 DFS 的顺序，E 节点已经被访问了，所以 A->E 那条线就不是。



- Back edge
  - from descendent to ancestor，就是从后辈到祖宗辈的线，back edge 不可以是 tree edge，这里用绿色表示

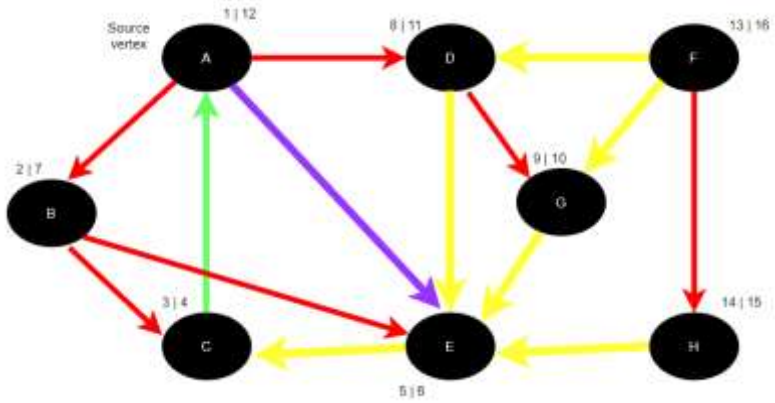


  - 这里是因为 A->B->C，按照 DFS 的遍历顺序，C 是 A 的后代，但是图里，C->A，也就是 C 可以到 A，所以 C->A 的那条线就是 back edge，关于 back edge，可以看看那些循环线，比如说 A->B->C->A,就是一个循环线
- forward edge
  - from ancestor to descendent. 就是从祖宗辈到后辈，这条线就是从灰色的点到黑色的点，（按照 DFS 的顺序）线不可以是 tree edges。这里用紫色表示



  - 在 DFS 访问的时候，A 是灰色的，但是 E 已经变成黑色的了，所以那条线就是 forward edge
- cross edge
  - between two nodes w/o ancestor-descendant relation in a depth first tree or two nodes in two different depth-first trees，说白了，就是除开 tree

edges，back edges, forward edges，其他的边都是 cross edges，这里就用黄色表示



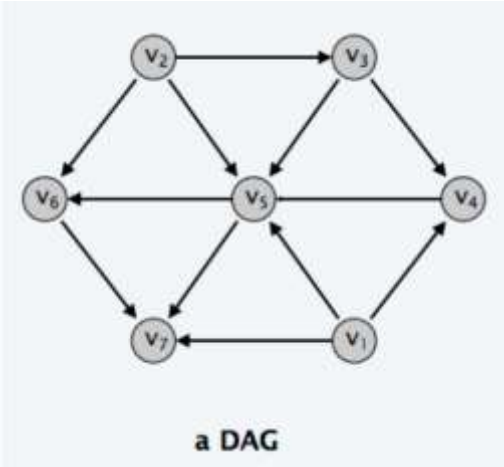
- 三个重要理论

  - In a depth-first search of an undirected graph G, every edge of G is either a tree edge or a back edge. 也就说是在 DFS 的无向图中，每条边不是 tree edge 就是 back edge。也就是说没有 forward edges 和 cross edges
  - An undirected graph is acyclic if a DFS yields no back edges. 如果无向图中，DFS 中没有 back edges，表明没有自循环，也就是 acyclic。反过来也成立，就是如果这幅图是 acyclic，说明没有 back edges
  - A directed graph is acyclic if a DFS yields no back edges. 在有向图中，如果 DFS 没有产生 back edges，说明是 acyclic 的

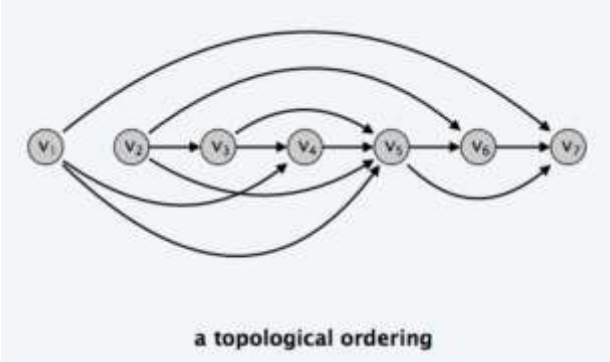
Topological sorting

Directed Acyclic Graph，有向无环图

- A DAG is a directed graph that contains no directed cycles. 这就表示有向无环图中是没有任何自循环的



- A topological order / topological sort of a DAG, G= (V,E) is an ordering of its nodes as v1, v2,..., vn so that for every edge (vi, vj) we have i<j



- precedence constraints
  - DAG 应用于许多地方，应用的限制就是又先后顺序的，where there are precedence or ordering constraints。
  - example
    - 举例就是说：if there are a series of tasks to be performed, and certain tasks must precede other tasks. 就是说需要先完成一些事情才能去做下一件事情，是有个顺序的，比如说大学选课，都是有前置课程的，那么，这个就是有顺序的，这个就是 DAG
    - 还有一个例子就是做工程，先做什么，后做什么，这个顺序很重要。
  - precedence constraints
    - Edge (vi, vj) means task vi must occur before vj

compute topological sort

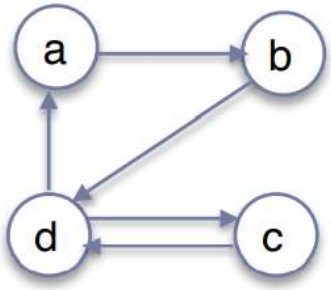
- we can use DFS
  - call DFS to compute finishing times f(v) for each vertex v
  - as each vertex is finished, insert it into the front of a linked list
  - return the linked list of vertices
- the time complexity is θ（V+E）

example，leetcode 207 和 210

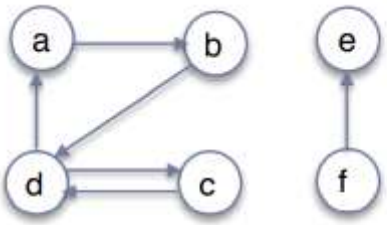
Strongly connected V.S. Connected

Directed Graphs

- Strongly connected
  - every two vertices are reachable from each other. 也就是说任意两点都是可以点联通的
  - 比如说在通信应用上，判断是否两两连通
  - example

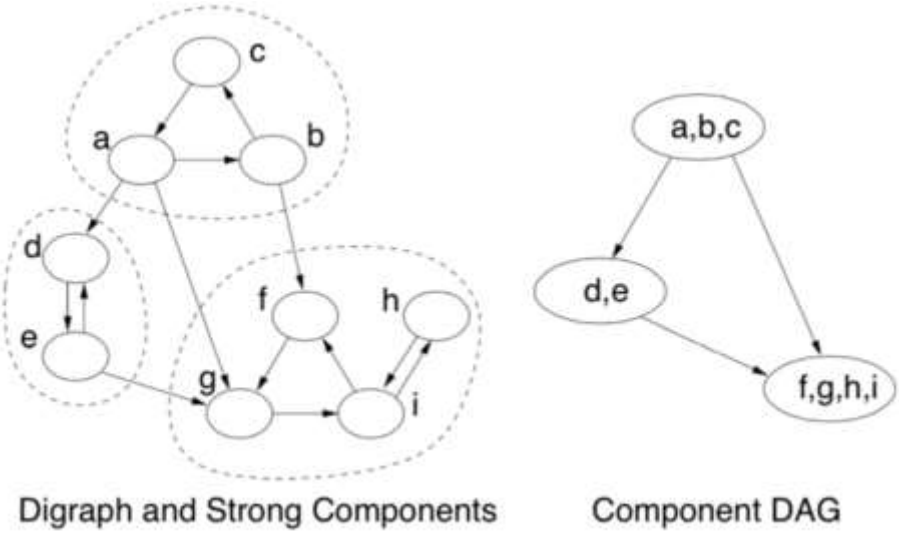


- strongly connected componets
  - all possible strongly connected subgraphs
  - example

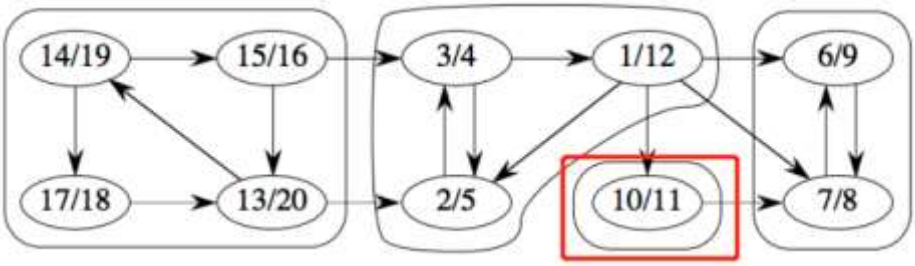


strongly connected components:
{a,b,c,d},{e},{f}

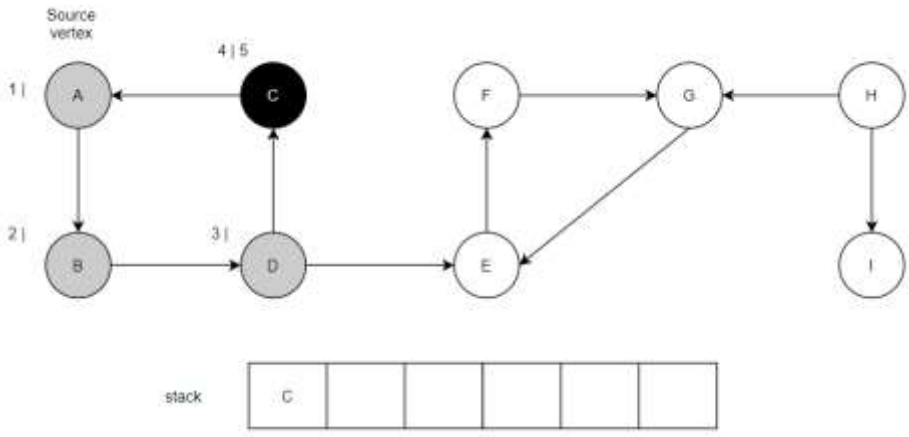
- Component DAG，跟上面的 DAG 不一样啊，不要弄混淆了
  - computing the strongly connected components of a digraph
  - Partition the vertices of the digraph into subsets such that the induced subgraph of each subset is strongly connected. 也就是说把 graph 进行分配，然后把每一组能够两两连通的部分作为一个 subgraph
  - Merging the vertices in each strong component into a single super vertex, and joint two super-vertices (A, B) if and only if there are vertices u ∈ A and v ∈ B such that (u,v) ∈ E。 也就是说再把这个几个大 subgraph 给连接起来
  - The resulting digraph, called the component digraph, is necessarily acyclic. We may refer it as the component DAG. 连接起来后，我们管这个叫 component DAG
  - example



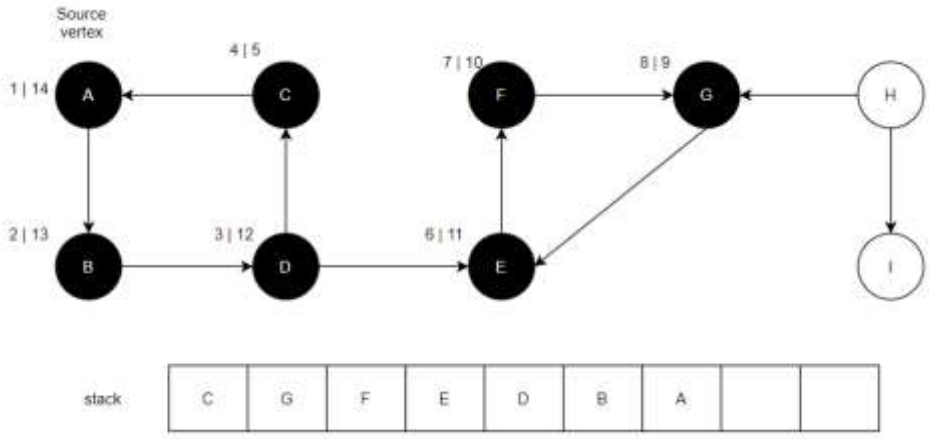
- another example，落单的也要算进去



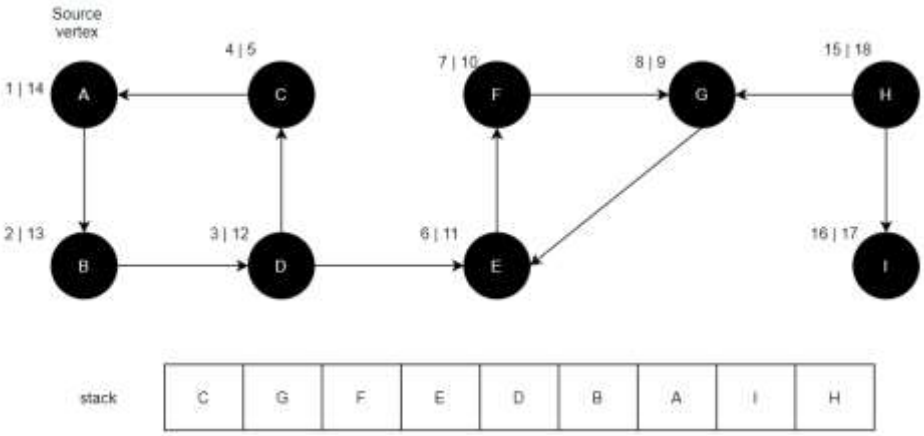
- Finding Strongly connected component
  - Kosaraju's algorithm
    - 基本 idea
      - Call DFS（G）to compute finishing times f(u) for each vertex
      - compute G^T (reversing all edges of G)，也就是要把这个图给 reverse 一下
      - call DFS (G^T), but in the main loop of DFS, consider the vertices in order of decreasing f(u) ，然后倒序计算，可以借助 stack
      - output the vertices of each tree in the depth-first forest we got above
    - 算法图解
      - call DFS(), visit 节点，然后 visit 到不能 visit 的时候就入栈



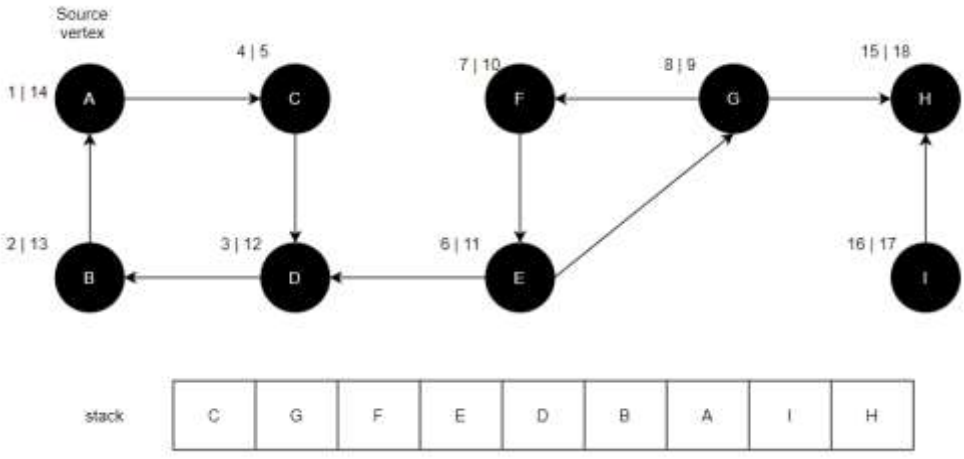
- 都是 visit 到不能 visit 为止，然后入栈，还有两个节点，按字母顺序，从 H 开始



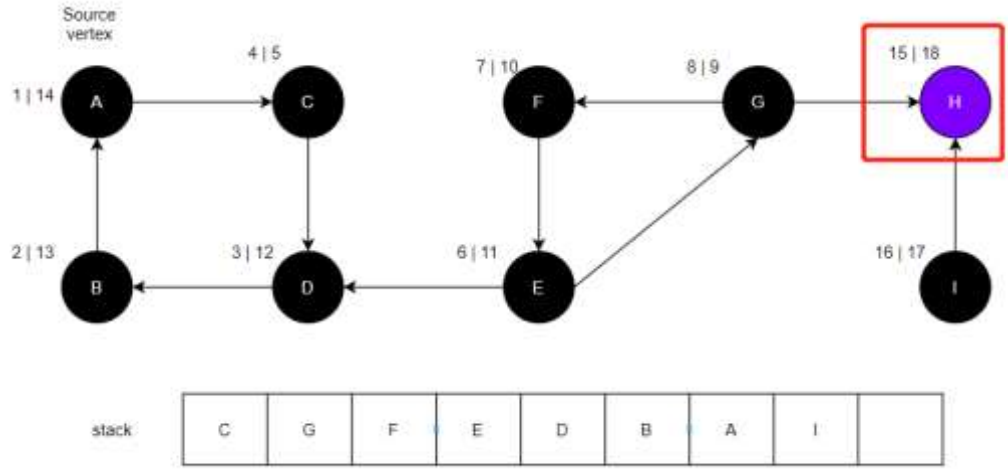
- 最后全部入栈了。接下来呢，按照算法，得把所有的边的方向都给反过来，再做一次 DFS（G^T）如果是强连通的，那么就还是强连通。反过来之后，就从最后一个 finish time 倒序回做



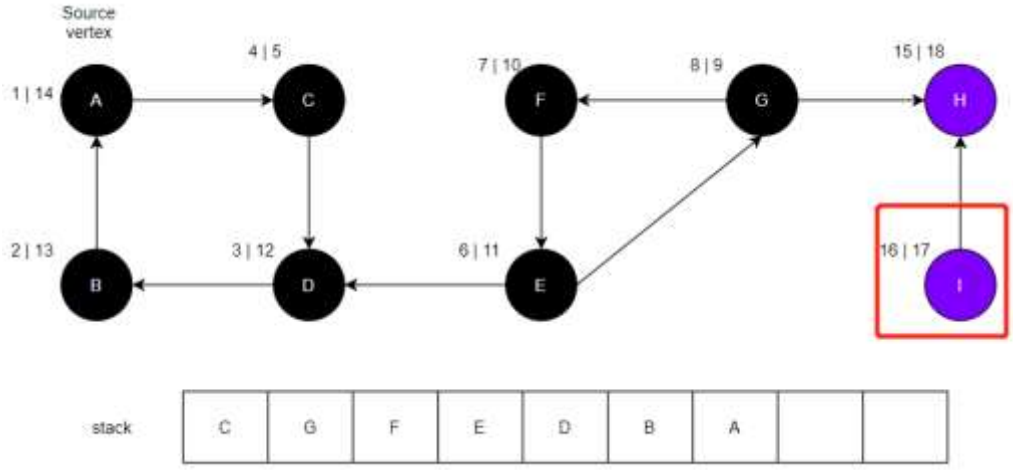
- 把全部箭头都反过来之后，从栈顶元素开始，也就是 H



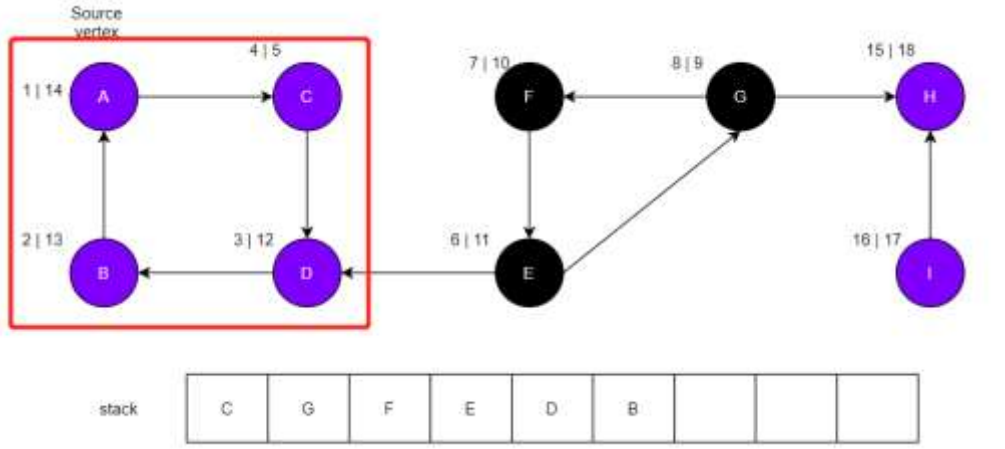
- 反过来之后，发现 H 去不了别的点了，于是 H 就是一个 strongly connected component，紧接着我们再去看栈顶元素，也就是 I



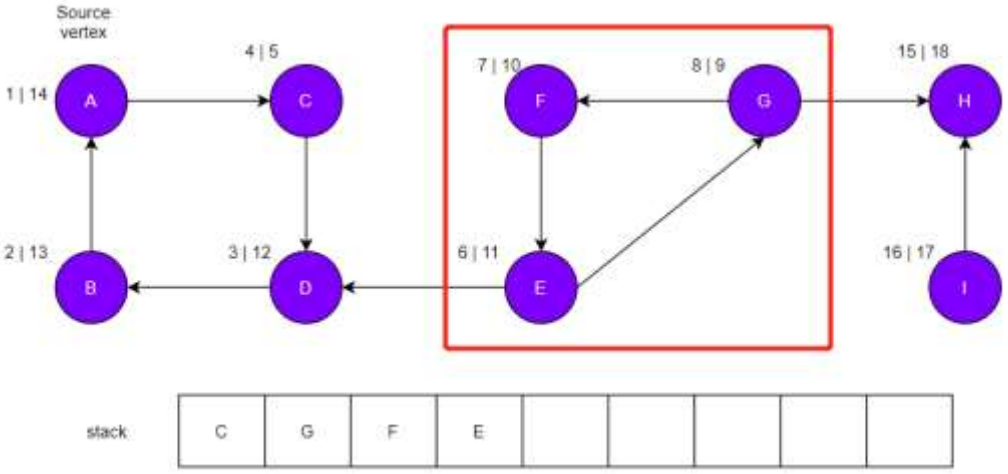
• 我们发现 I 又哪里都去不了了，这说明又是一个 strongly connected component，接着再看栈顶元素，也就是 A



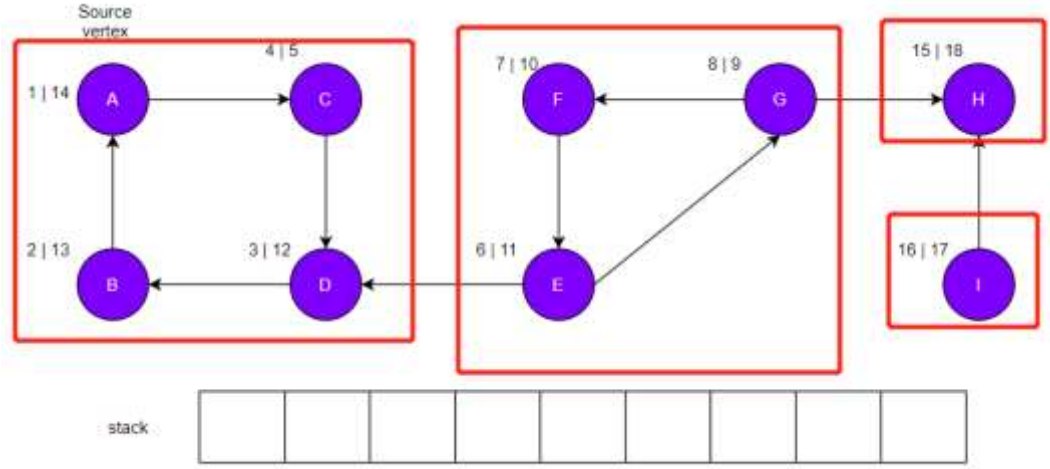
• 发现 A->C->D->B 可以连通，说明又是一个 strongly connected component，接着再看栈顶元素，也就是 B，但是 B 已经被 visited 了，就再接着看下一个，一直到 E



• 于是我们又发现了一个 strongly connected component，然后再查栈里元素，都被 visited 了，就全弹出来了，算法结束

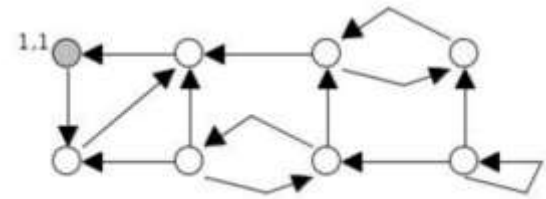


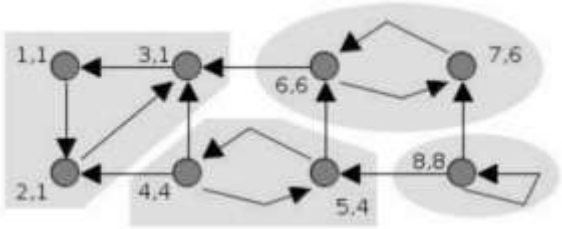
- 最终结果，返回 4 个 strongly connected components



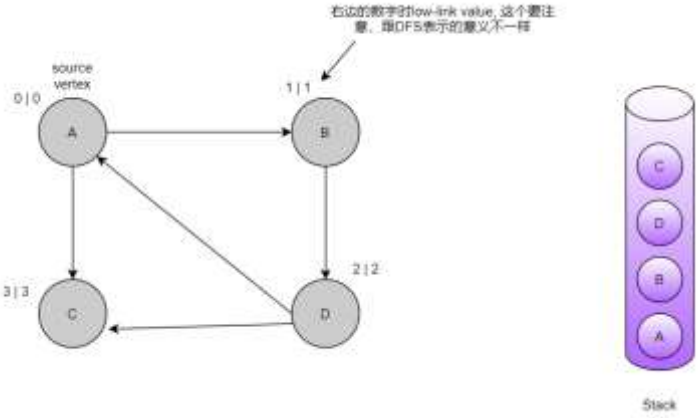
- 时间复杂度是 O(V+E)

- Tarjan's algorithm
  - this algorithm will be based on the following facts
    - DFS search produces a DFS tree/forest
    - Strongly connected components form subtrees of the DFS tree
    - if we can find head of such subtree, we can print/store all the nodes in that subtree (including head) and that will be one SCC
    - there is no back edge from one SCC to another
  - 算法过程
    - the vertices are indexed as they are traversed by DFS procedure
    - while returning from the recursion of DFS, every vertex v gets assigned a vertex L as a representative
    - L is a vertex with the least index that can be reached from v
    - Nodes with the same representative assigned are located in the same strongly connected component
    - example，也就是说把同一个 SCC 的都分为一个 index，然后那个圈的就是同个 SCC

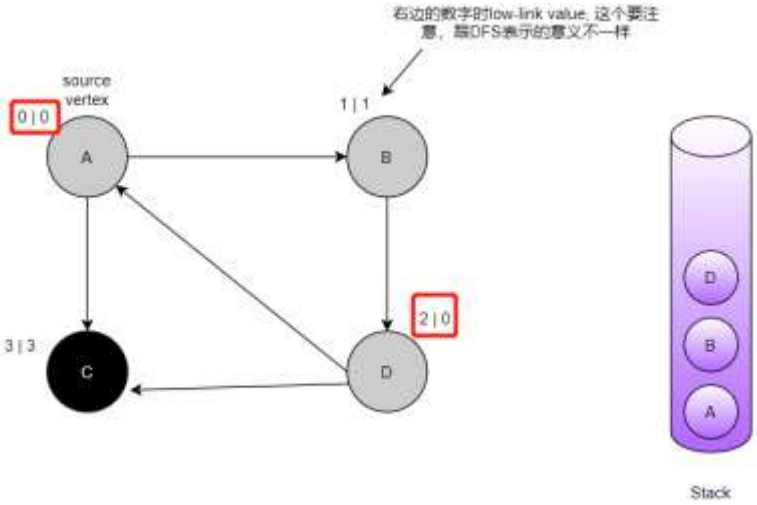




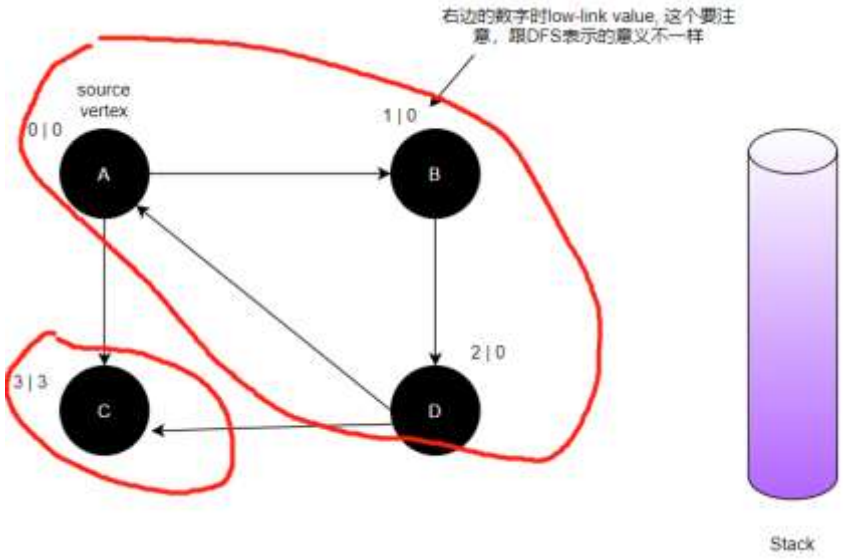
- 过程图解
  - 也是按照 DFS，但是数字表达的意义不一样了，我们到了 D 之后，假设先去探索 C，发现 C 已经不能再探索了，但是又没有在 stack 里找到相同元素，就只能返回



  - 这时，我们把 C 从 stack 里 poll 出来，然后返回去 check，发现 D 能去到 A，然而 A 又在 stack 里，说明能形成一个环，就把他的 low-link value 改成跟 A 一样的 low-link value



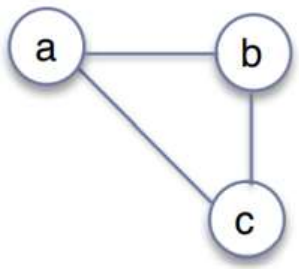
  - 最后我们去查相同的 low-link value，相同的就是一组 SCC，别忘了，落单的也是一组，找到之后算法完成



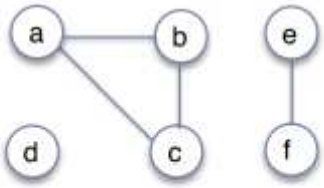
- 时间复杂度
  - O(V + E)

undirected graphs

- connected
  - every pair of vertices are connected by a path
  - example



- connected components
  - all possible connected subgraphs
  - example



connected components:
{a,b,c},{d},{e,f}

Dynamic programming

DP

DP 非常重要，面试也非常重要，应用也非常广

- Bioinformatics

- control theory
- computer science, AI
- 等等

DP 就是 break up a problem into a series of overlapping subproblems, and build up solutions to larger and larger subproblems. 也就是说把大问题细小化，然后去解决小问题

看完我们会发现有点像 divide and conquer 诶，也是把问题细分，然后解决这个小问题, 但其实有一个最大的区别, 也就是 divide and conquer 里面的小问题是 independent 的，不彼此依赖的。这里解决的 sub-problem 也是一样的

DP 细分后的小问题是彼此依赖的, 也就是说现在的结果要之前的结果完成后才能有现在的结果，这里解决的 sub-problem 是跟 main problem 一样的，只不过是规模会小一点而已，main problem 必须要先得到 sub-problem 的解决，依赖之前的结果。解决完之后，会把结果存进 table 里

DP 可以解决的问题

- optimization problems.就是最优化问题
    - 像问，smallest，largest,最少的，最多的，类似这种情况的
    - "find XX with the smallest/largest YY" (for two strings, find a longest common subsequence; for a set of strings with frequencies, find the best way to organize them into a search tree to minimize the expected cost; etc)
- 计数问题
    - 就是问有多少种方式什么的
- 求存在性
    - 就像是取石子游戏，先取是否必胜之类的
    - 能否选出 k 个数据的和等于 sum

DP algorithm

Characterize the structure of an optimal solution. 也就是说我们首先要清楚最优解的结构是怎样的

recursively define the value of an optimal solution。然后递归地去使用，解决之前的 sub-problem

compute the value of an optimal solution typically in a bottom-up fashion. 把每一个结果都计算出来

construct an optimal solution from computed information (not always necessary)。 得到

最优解

DP 解决问题的步骤

确定状态

- 看最后一步
- 子问题是什么

转移方程 state expression

- 根据子问题得出的方程

初始条件和边界情况

计算顺序

Rod cutting problem

Longest common subsequence

Greedy Algorithm

greedy algorithm

贪心算法就是在当时，在当时的时刻做最好的选择，先不担心将来的情况

关键就是 make a locally optimal choice and it will lead to globally optimal solution. 也就是说一直做最好的选择，最终结果就会是最好的

但是贪心算法也不总是会得到最好的结果，所以，还得看题目

当题目同时符合以下两种情况

- greedy choice property
  - a globally optimal solution can be arrived at by making a locally optimal. 我们要证明的就是 a greedy choice at each step yields a globally optimal solution
- optimal substructure property
  - A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems. 也就是说它的子问题有最优解

DP V.S greedy

- DP
  - 最优子结构

        - 重叠子问题
            - 就是子问题求得还是跟原问题是一样的，只不过过变小了
    - greedy
        - 最优子结构
            - 每一步贪心选完后会留下子问题，子问题的最优解和贪心选出来的解可以凑成整个问题的最优解
        - 贪心选择
            - 每一次贪心选出来的一定是最优解的一部分

An activity selection problem

Subtopic 1

Knapsack problem

Huffman codes

Huffman coding

- coding is used for data compression
- Binary character code: character is represented by a unique binary string
    - fixed length code (block code)
    - variable-length code
        - frequent characters: short codeword
            - a: 000, b: 001, …, f: 101
            - ace: 000 010 100
        - infrequent characters: long codewords
    - fixed V.S. frequent

	a	b	c	d	e	f	cost / 100 characters
Frequency	45	13	12	16	9	5	
Fixed–length codeword	000	001	010	011	100	101	300
Variable–length codeword	0	101	100	111	1101	1100	224

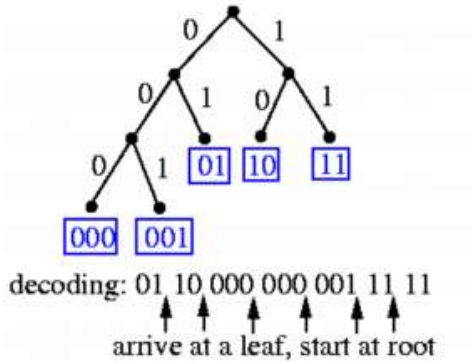
prefix codes

- one code per input symbol
- no code is a prefix of another
- 也就是说这样编码满足前缀编码，即字符的编码都不会其他字符编码的前缀，不会造成匹配的多义性，这是无损压缩处理
- 非常容易 decode

方法

- 就是使用 binary tree

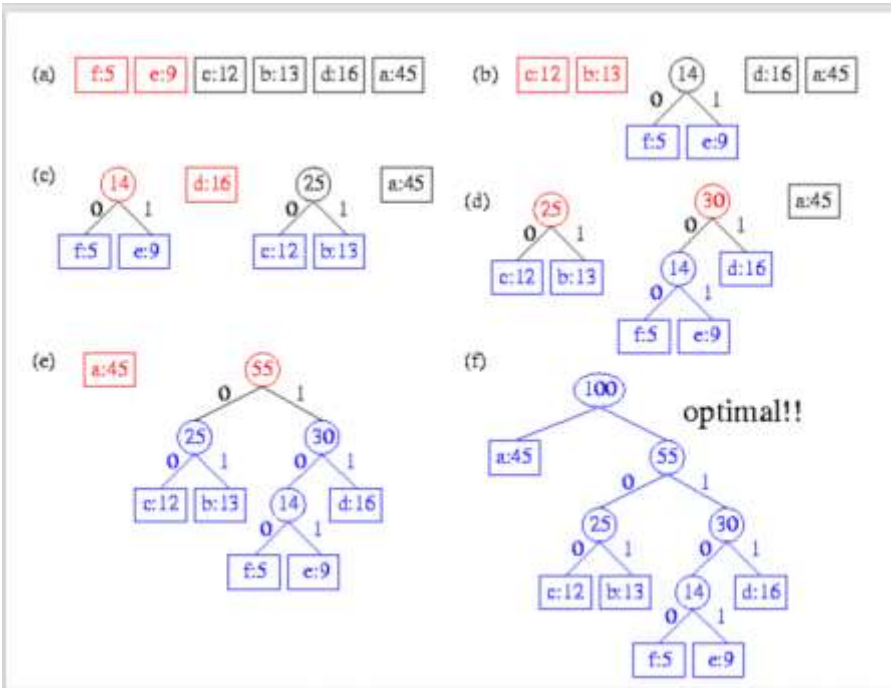
- 左边代表是 0
- 右边代表是 1
- 就是这么个过程



算法过程

- 我们把频率作为节点数据，首先我们按照从小到大进行排序，选出根节点最小的两棵二叉树
- 组成一棵新的二叉树，这时，又有了一个新的根节点数值，然后又跟下个数据进行对比，看看怎么排，最后生成的就是 Huffman tree

算法图解



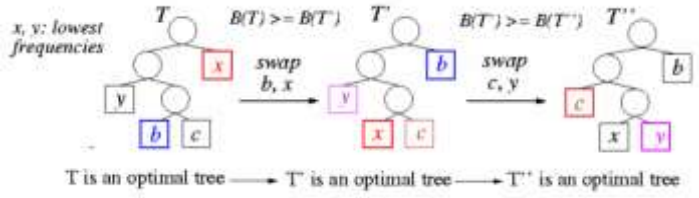
pseudocode

```
Huffman(C)
1. n = |C|
2. Q = C
3. for i = 1 to n -1
4.     Allocate a new node z
5.     z.left = x = Extract-Min(Q)
6.     z.right = y = Extract-Min(Q)
7.     z.freq = x.freq + y.freq
8.     Insert(Q, z)
9. return Extract-Min(Q)  //return the root of the tree
```

time complexity: O(nlgn)

Huffman algorithm is greedy algorithm

- Greedy choice
  - 每一步都是最优解



  - x, y 是最低频的，他们都会有相同的 length，不同的是仅仅是数据的最后一位， the last bit
- optimal substructure
  - 最优子结构：每次贪心选走两个后，剩下的最优编码和贪心选走的那两个的最优解就是原问题的最优编码

Optimal substructure: Let T be a full binary tree for an optimal prefix code over C. Let z be the parent of two leaf characters x and y. If z.freq = x.freq + y.freq, tree T' = T - {x, y} represents an optimal prefix code for C'= C - {x, y} U {z}.

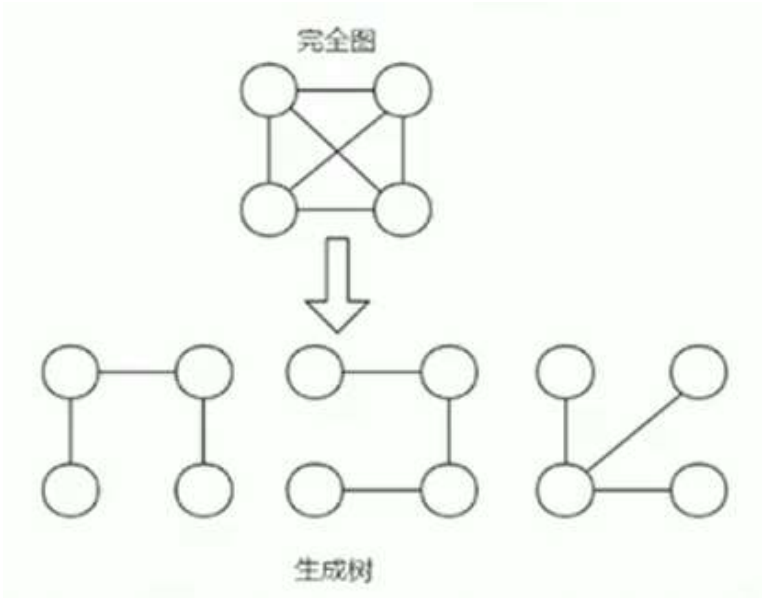
B(T) = B(T')+x.freq+y.freq
($d_T(x)=d_T(y)=d_T(z)+1$)

If T' is not optimal, find T'' s.t. B(T'')<B(T')
z is in C' => z is a leaf of T''
Add x, y as z's children to form T'''
=> B(T''') = B(T'')+x.freq+y.freq
< B(T')+x.freq+y.freq
= B(t)
Contradiction!!

Minimum Spanning tree problem. 最小生成树

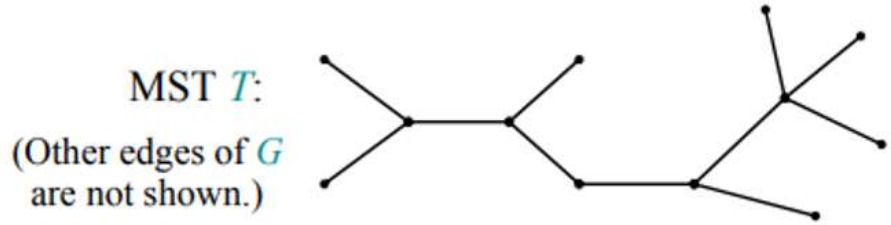
The subset of edges that connected all vertices in the graph, and has minimum total weight. 首先是一个 subgraph，其次权重和要最小

- 有很多的生成树，但是 MST 是要找权重总和最小的

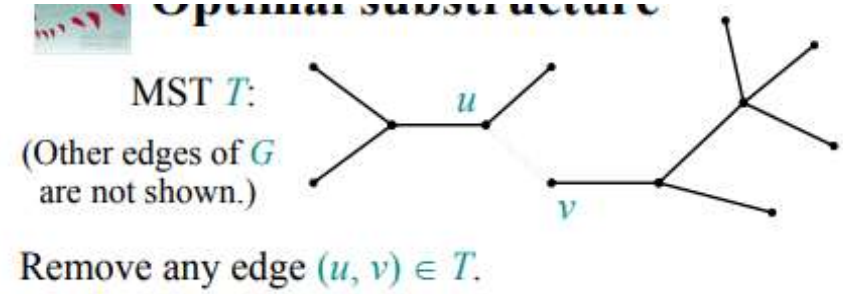


两种算法

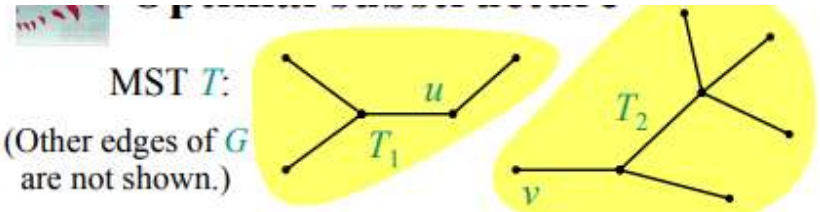
- Prim's algorithm
  - 思想也就是贪心算法的一种实现，因为是找最小权重，于是我们就去找 local minimum，找到所有的 local minimum 加起来，就是 global minimum
  - optimal substructure，最优子结构
    - 假设有一幅图，其他的边没有画出来，假设这个就是 MST，如果要符合贪心算法，那么我们就要去证明这东西有最优子结构



    - 我们随意地去除一条边，这里我们要清楚，那条边是（u，v）

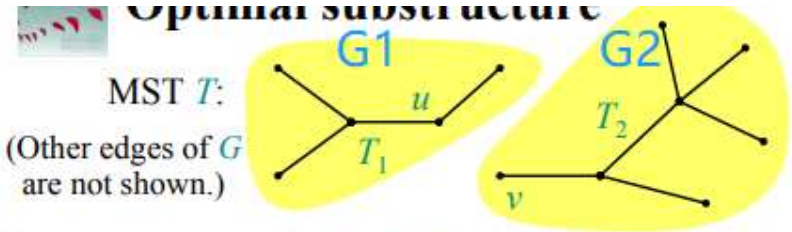


    - 这样就被分成了两部分，也就是 T1 和 T2



Remove any edge $(u, v) \in T$. Then, T is partitioned into two subtrees T_1 and T_2.

- 于是，这里就有一个理论，被分割后，就有了两个最小 MST，这个理论就是 subtree T1 是 G1 的 MST，刚刚说还有其他边没有画出来，但是现在画出来的就是 MST，所以，被分割后依旧是 MST



Remove any edge $(u, v) \in T$. Then, T is partitioned into two subtrees T_1 and T_2.

Theorem. The subtree T_1 is an MST of $G_1 = (V_1, E_1)$, the subgraph of G ***induced*** by the vertices of T_1:

$$V_1 = \text{vertices of } T_1,$$
$$E_1 = \{(x, y) \in E : x, y \in V_1\}.$$

Similarly for T_2.

- 证明这个最优子结构，也就是说如果还存在 T'更小，那么原来的公式就不成立了，证明结束

Proof. Cut and paste:

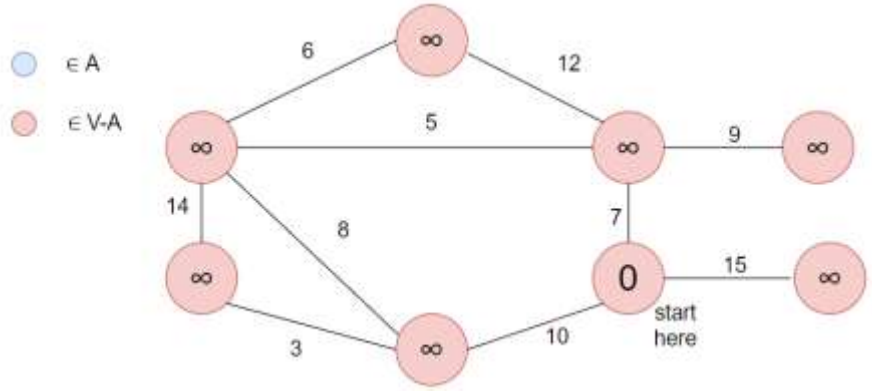
$$w(T) = w(u, v) + w(T_1) + w(T_2).$$

If T_1' were a lower-weight spanning tree than T_1 for G_1, then $T' = \{(u, v)\} \cup T_1' \cup T_2$ would be a lower-weight spanning tree than T for G. □

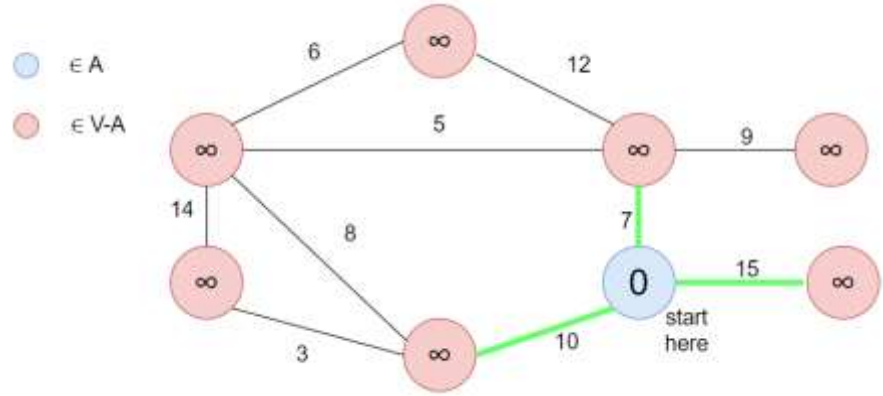
- 算法图解
  - prim 的基本思想，看起来这么复杂，说白了，就是选两个顶点之间的最小 weight，这个就是整句话的核心

Theorem. Let T be the MST of $G = (V, E)$, and let $A \subseteq V$. Suppose that $(u, v) \in E$ is the least-weight edge connecting A to $V - A$. Then, $(u, v) \in T$.

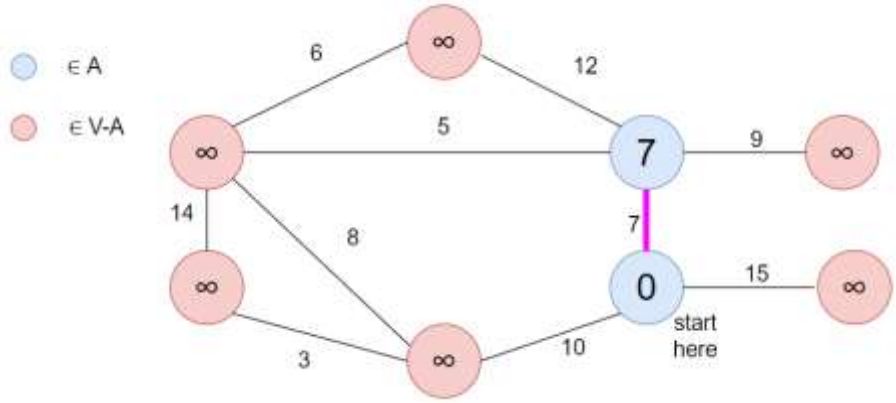
- 红色表示还没访问的点，如果变成了蓝色，说明可以进入 queue，来表示 MST，我们可以随意选一个点开始



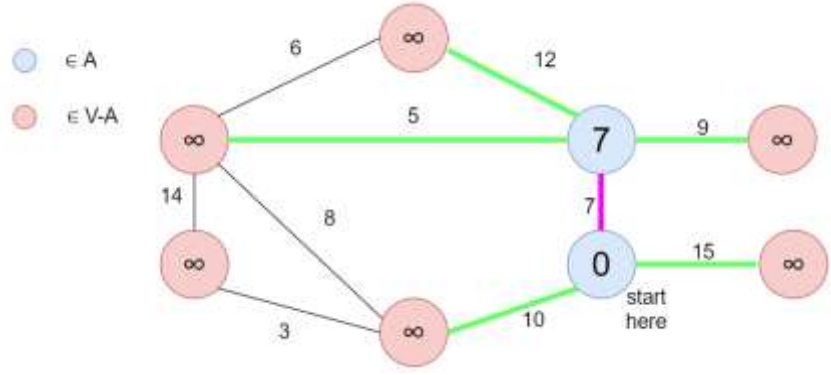
- 因为我们会有邻接矩阵，于是，我们就去看看从出发点开始的邻接点，哪条边的 weight 是最小的，然后记录这条最小的边



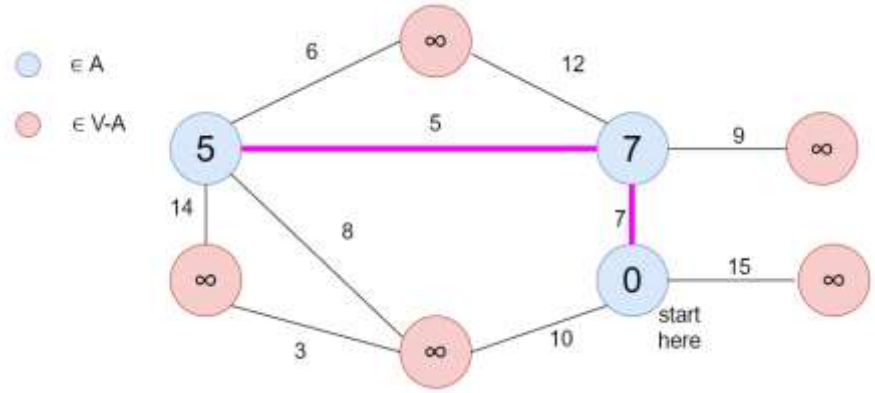
- 我们把最小的边个留下来，记录这个数值，比如这个最小值是 7，我们就把邻接节点的 value 暂记为 7，因为我们现在有了两个节点，然后分别从这两个节点出发，选取最小的边



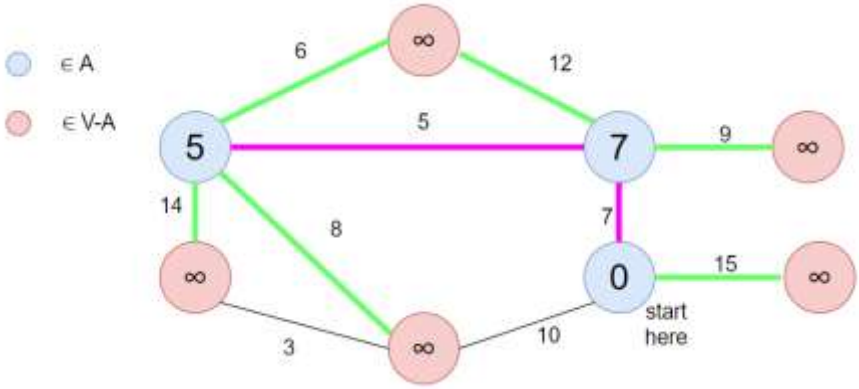
- 两条边都要进行探索，要去寻找最小的边



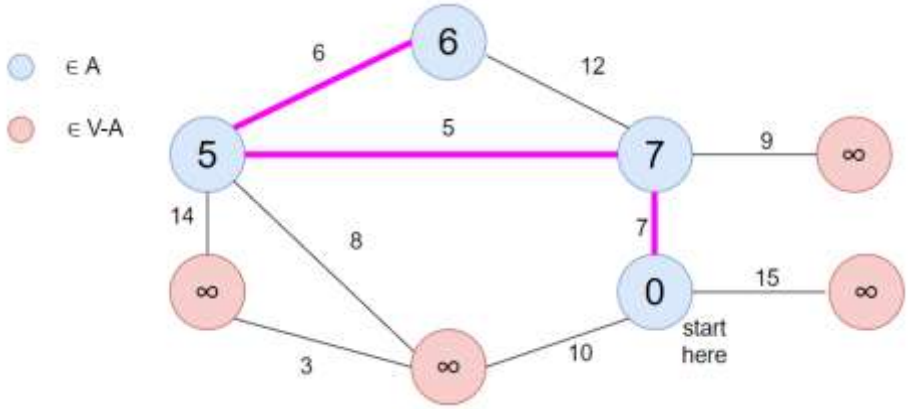
- 现在，我们有了三个节点了，重复以上步骤，三个节点都要去找邻接点，取最小的边，然后再记录



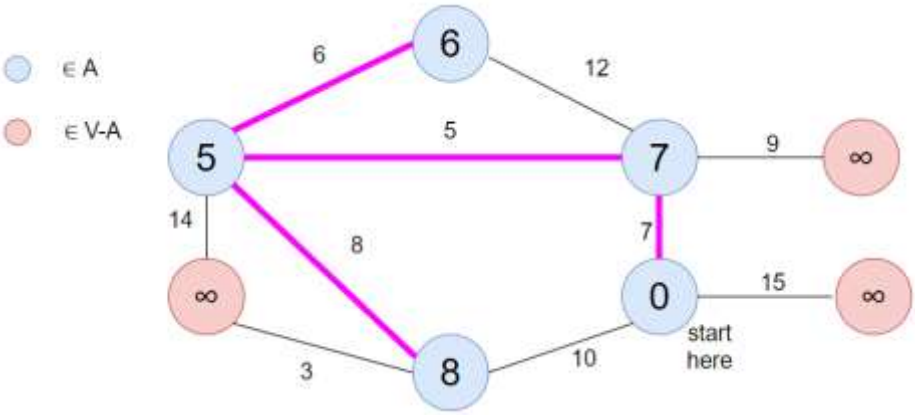
- 取最小的边，这里是 6



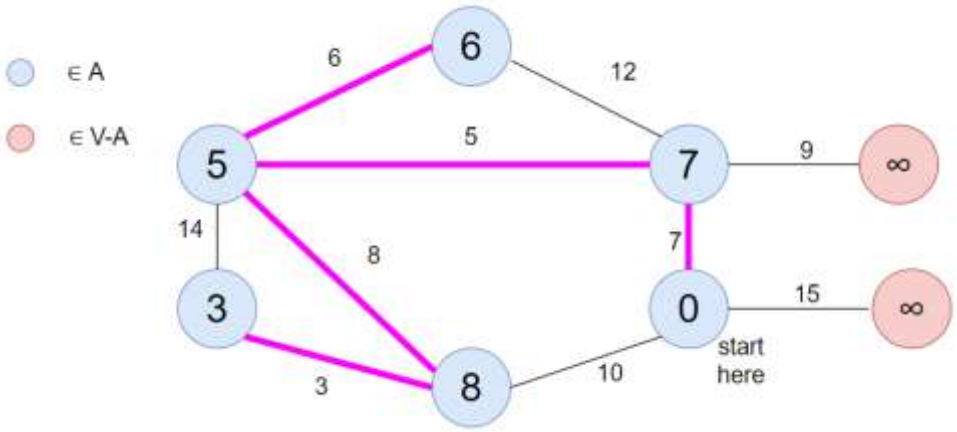
- 接下来就是记录这个节点



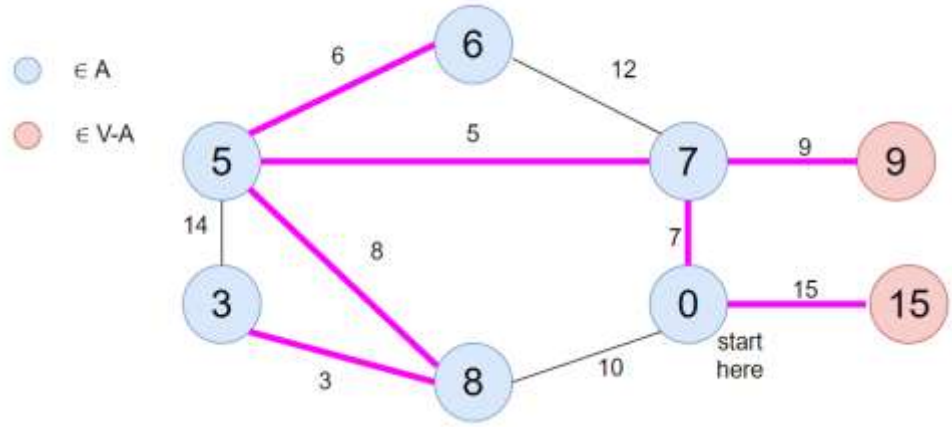
- 再继续重复这个步骤，我们会得到 8 这个节点



- 再接下来是 3，最后 9，和 15 也是一样



- 最终我们得到紫色路径就是 MST



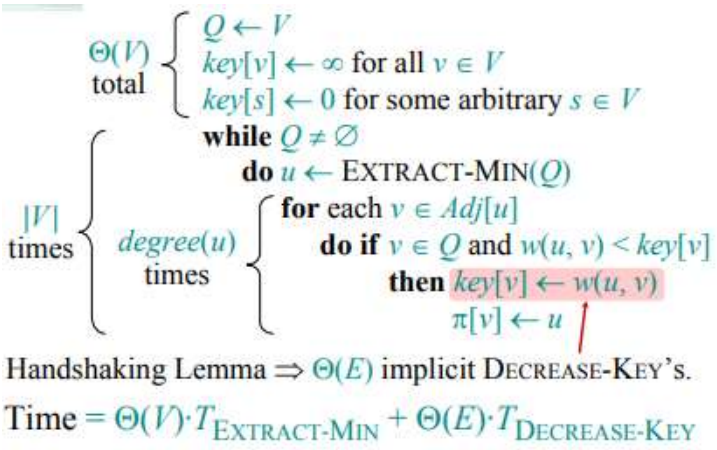
- pseudocode

IDEA: Maintain $V - A$ as a priority queue Q. Key each vertex in Q with the weight of the least-weight edge connecting it to a vertex in A.

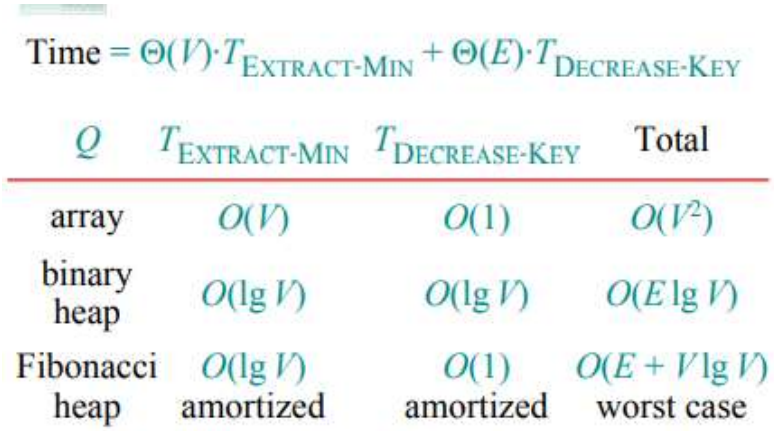
$Q \leftarrow V$
$key[v] \leftarrow \infty$ for all $v \in V$
$key[s] \leftarrow 0$ for some arbitrary $s \in V$
while $Q \neq \varnothing$
  do $u \leftarrow$ EXTRACT-MIN(Q)
    for each $v \in Adj[u]$
      do if $v \in Q$ and $w(u, v) < key[v]$
        then $key[v] \leftarrow w(u, v)$ ▷ DECREASE-KEY
          $\pi[v] \leftarrow u$

At the end, $\{(v, \pi[v])\}$ forms the MST.

- 时间复杂度
  - 分析

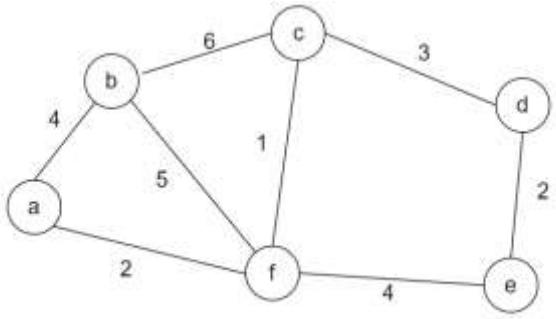


- 也取决于我们用什么方法来做这个 extract min

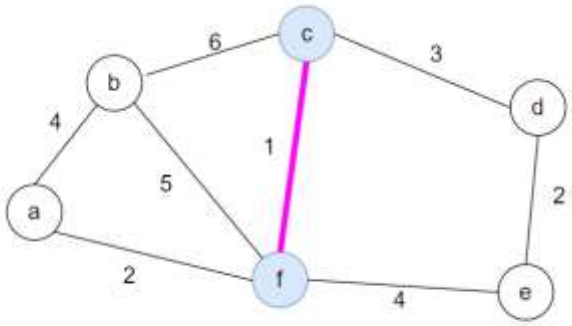


Q	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total
array	$O(V)$	$O(1)$	$O(V^2)$
binary heap	$O(\lg V)$	$O(\lg V)$	$O(E \lg V)$
Fibonacci heap	$O(\lg V)$ amortized	$O(1)$ amortized	$O(E + V \lg V)$ worst case

- Kruskal's algorithm
  - 基本 idea
    - 假设我们有 n 个节点，然后按照权值从小到大进行排序，我们根据排序选出 n-1 条边并保证这些节点与边不会构成回路
    - 所以这个算法要解决的问题就是
      - 边的权值大小的排序
      - 怎么判断会不会构成回路
        - 这里运用 disjoint set,也就是并查集
        - 我们记录顶点在“最小生成树”的终点，某个顶点的终点是“最小生成树中与他连通的最大顶点”，然后每次需要将一条边添加到最小生成树中，接着判断该边的两个顶点的终点是否重合，如果重合，表示构成回路
  - 算法图解
    - 假设我们有这么一幅图，我们要先把边的权值进行排序，选出最小权值的边，然后看看这条边连接了哪两个点

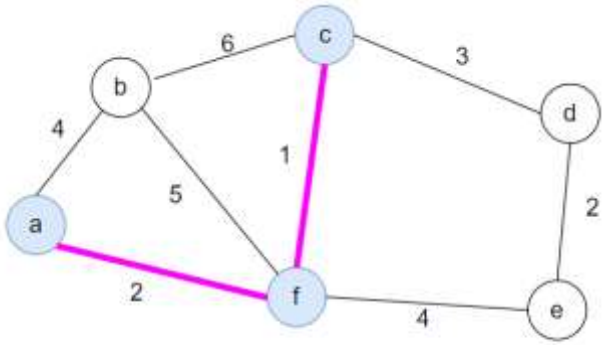


- 我们会看到 1 是最小的，然后连接的点是 c,f,接着再去看下一个最小的权值



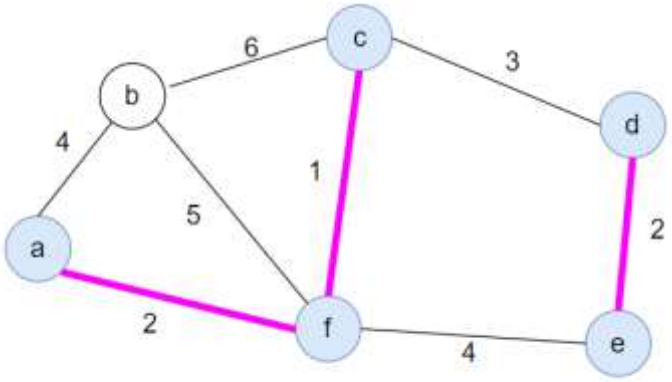
A = {{c,f}}

- 如果遇到相同的，我们就按照字母顺序加入，接着再去找下一个最小的权值



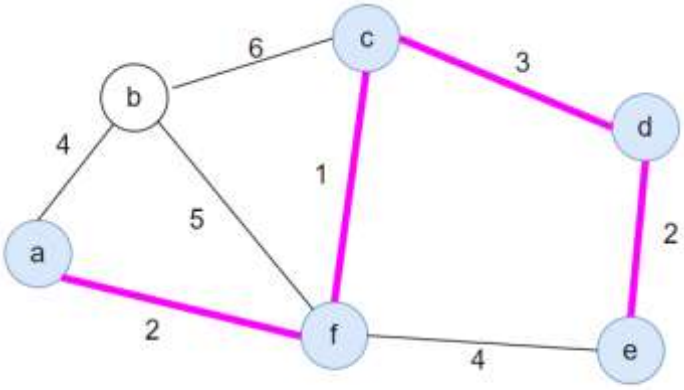
A = {{c,f},{a,f}}

- 我们又找到了 2，接着找



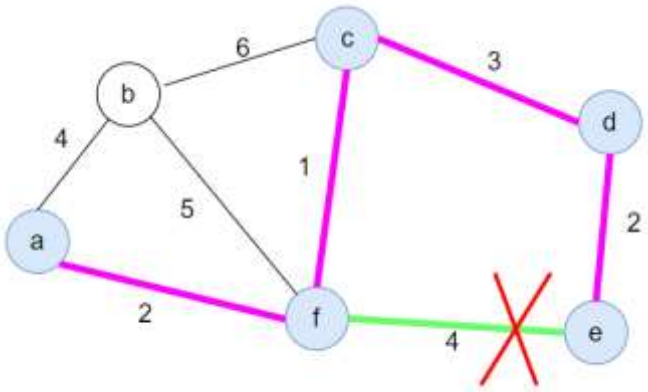
A = {{c,f},{a,f},{d,e}}

- 接下来是 3，再接着找



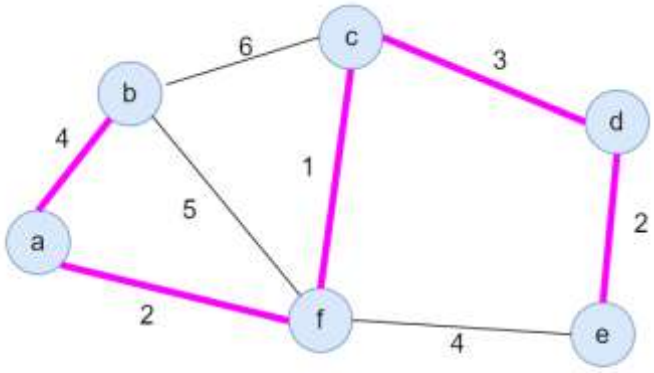
A = {{c,f},{a,f},{d,e},{c,d}}

- 然后现在出现了两个 4，但是图中的不能选，因为会构成回路，某个顶点的终点是“在最小生成树中与它连通的最大的顶点”，这里，最大的顶点也就是 e，这样一来，所有顶点的终点都指向 e，所以不行



A = {{c,f},{a,f},{d,e},{c,d}}

- 这样我们就把所有节点都找齐啦



A = {{c,f},{a,f},{d,e},{c,d},{a,b}}

- pseudocode

```
Kruskal(V, E)

A = ∅
foreach v ∈ V:
   Make-disjoint-set(v)
Sort E by weight increasingly
foreach (v1, v2) ∈ E:
   if Find(v1) ≠ Find(v2):
      A = A ∪ {(v1, v2)}
      Union(v1, v2)
return A
```

- 时间复杂度
  - O（ElogV）

- 即使有边的权重是负数，prim 和 kruskal 都没问题，都能解决

shortest paths

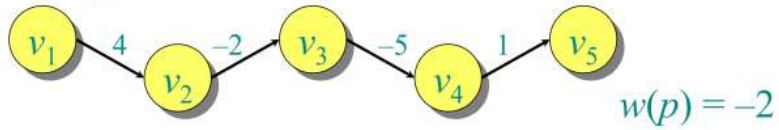
paths in graphs

其实这个东西讲的就是节点与节点之间的权重加起来

Consider a digraph $G = (V, E)$ with edge-weight function $w : E \to \mathbb{R}$. The ***weight*** of path $p = v_1 \to v_2 \to \cdots \to v_k$ is defined to be

$$w(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1}).$$

Example:



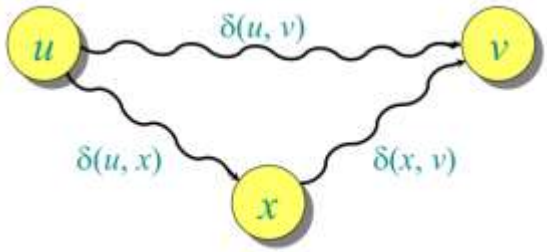
$w(p) = -2$

shortest paths

- 最短路径就是从 u 到 v ，加起来的权重要最小
- δ（u, v）= min {w(p)}, p 就是那条 path
- δ（u，v）= ∞，说明此路不通

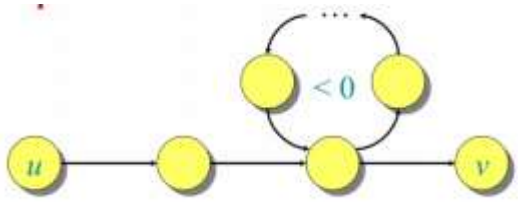
triangle inequality

- 就是两边之和大于第三边。δ（u,v） ≤δ（u,x）+δ（x,v）



Negative-weight cycle

- if a graph G contains a negative-weight cycle, then some shortest paths may not exist



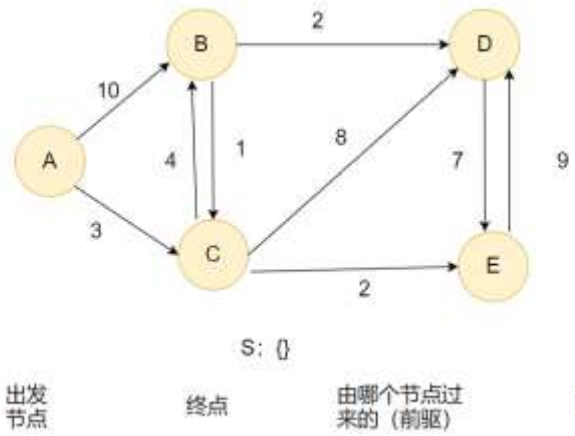
single source shortest paths，这个 single source 的意思就是从一个 node 到另一个 node 的最短距离

problem

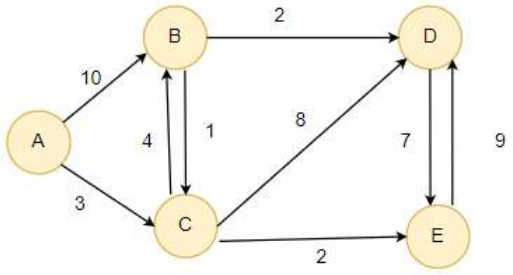
- From a given source vertex s ∈ V， find the shortest-path weights δ（s,v）for all v ∈V. If all edge weights w(u,v) are non-negative, all shortest-path weights must exist.
- idea: greedy,也就是可以采用贪心算法
  - maintain a set S of vertices whose shortest-path distances from s are known. 也就是说用一个 set 来记录节点，且要记录最短距离
  - At each step add to S the vertex v ∈ V-S whose distance estimate from s is minimal. 从某个节点出发后，如果有更小的距离，就要把原来的距离变为更小的
  - update the distance estimates of vertices adjacent to v

Dijkstra algorithm

- Shortest paths and relaxation
  - Dijkstra 算法的基本思想就是用一个数组 d[v] 来记录那个最短路径
    - 算法得一个一个节点地走
    - 走到一个节点的时候，d[v]就要记录 s-v 的最短距离，就是原先我们 d[v]有一个距离，然后我们到下一个的时候，要加上原来的距离，所以这个值应该要比原来的大或者是相等
    - 把起始点 d[v]设置为 0，其他的节点就要设置成∞
    - 走到每一个节点的时候，都要去 update 这个 d[v]的值，然后这个 update 的过程叫做 relaxation。这么无聊，update 都要取个名。这个过程就是看到更小的值，我们就用这个最小的
- 算法图解
  - 假设我们有这么一幅图，然后我们要记录这些信息



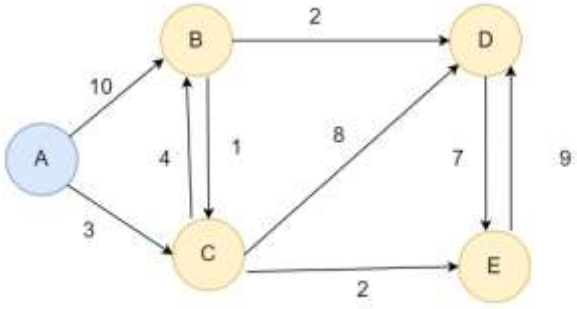
  - 我们出 A 点出发，看看去其他点的最短距离，自己是起始点，就要设为 0，其他的设为∞，然后算法开始，我们选取最小的距离，然后以那个为节点，看看到其他点的距离



S: {}

出发节点	终点	由哪个节点过来的（前驱）	Dist
A	A	A	0
	B		∞
	C		∞
	D		∞
	E		∞

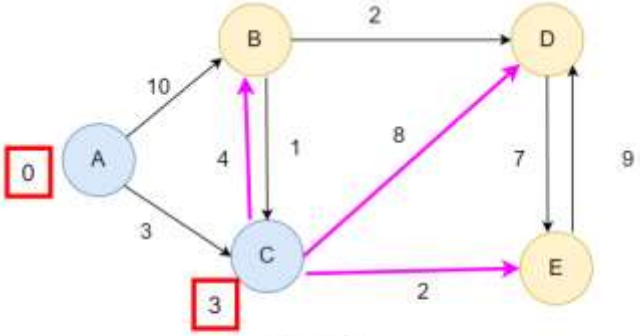
- 然后我们就要更新距离，A 能到 B 和 C，此时，我们要选取最小的距离的点作为下一个节点，重复 A 的动作，这里，C 最小



S: {A,}

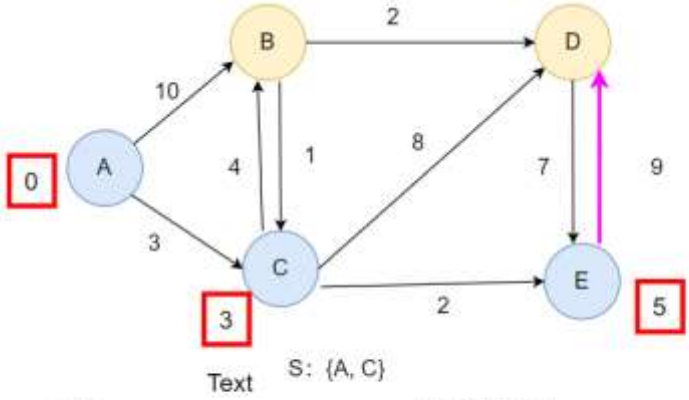
出发节点	终点	由哪个节点过来的（前驱）	Dist
A	A	A	0
	B	A	10
	C	A	3
	D		∞
	E		∞

- 我们发现 C 能到 B，D，E，原来 B 那一行的 dist 是 10，但是从 C 过去之后，距离会更短，变为了 7，就要 update 为 7。我们看到在图中，vertex 旁边有一个红色框框的数字，那个代表着如果是从这个点经过的，就要加上这个数字。就好比 B 的 dist，3+4=7. C 也能到其他顶点，更新距离。再选上最小的那个顶点，这里是 E



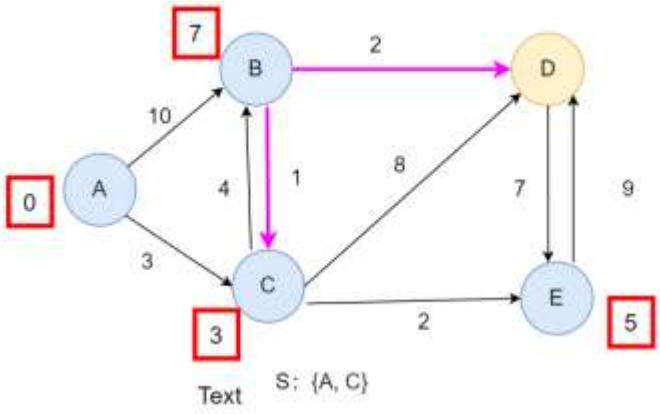
出发节点	终点	由哪个节点过来的（前驱）	Dist
A	A	A	0
	B	C	7
	C	A	3
	D	C	11
	E	C	5

- E 只能到 D，从 E 到 D 的距离为 5+9=14，我们得去对比原来的数据，原来是 11，比较小，我们就 keep 小的数值。接着我们再选一个节点，我们看到 C, E 都在 set S 中了，就要除去这两个，看看那个数值最小，这时是 B



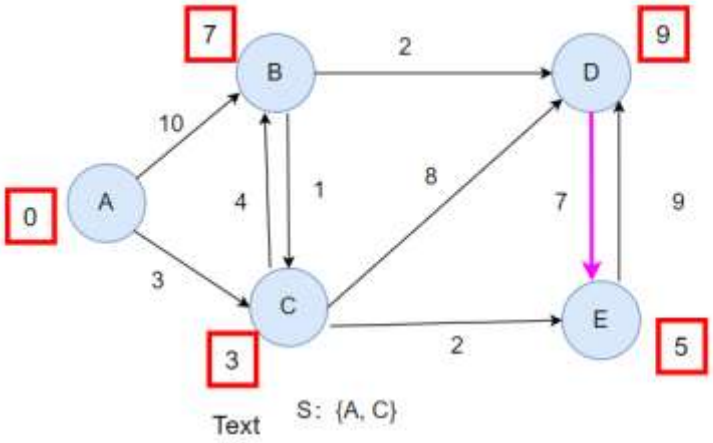
出发节点	终点	由哪个节点过来的（前驱）	Dist
A	A	A	0
	B	C	7
	C	A	3
	D	C	11
	E	C	5

- 我们看到 B 可以到 C 和 D，但是 C 已经在 set 中了，就不用理了，然后，我们看到 B-D 是 9，比原来的 11 要小，就要进行更新，这时，我们再选一个最小的，发现只有 D 节点可以选了，于是，我们就从 D 出发



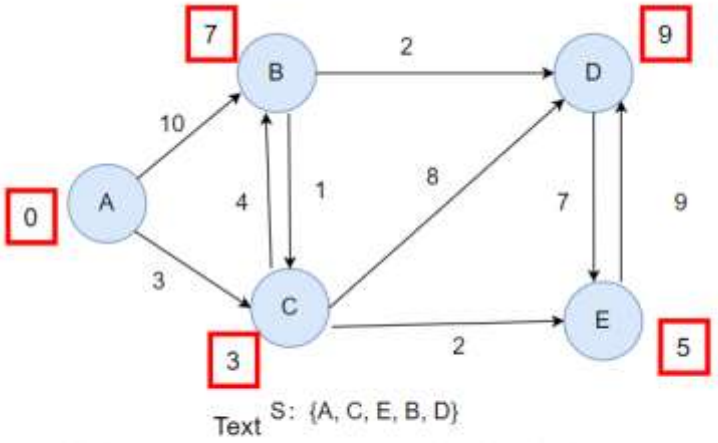
出发节点	终点	由哪个节点过来的（前驱）	Dist
A	A	A	0
	B	C	7
	C	A	3
	D	B	9
	E	C	5

- D 只能到 E，但是距离没有比原来的小，就不用理，现在，我们已经得到 A 到其他节点的最短距离了



出发节点	终点	由哪个节点过来的（前驱）	Dist
A	A	A	0
	B	C	7
	C	A	3
	D	B	9
	E	C	5

- 得到结果，结束算法



Text S: {A, C, E, B, D}

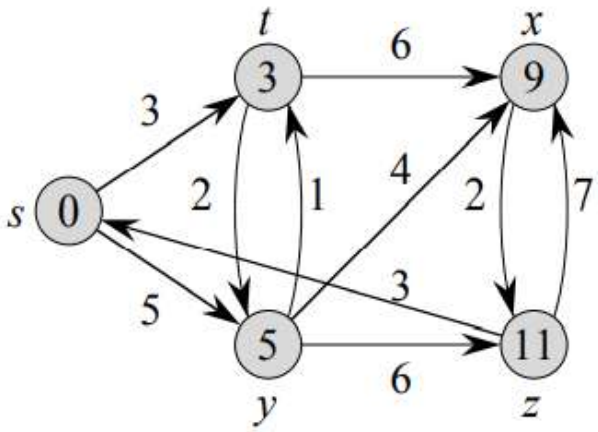
出发节点	终点	由哪个节点过来的（前驱）	Dist
A	A	A	0
	B	C	7
	C	A	3
	D	B	9
	E	C	5

- 要学会写 D values and π matrix
  - example
    - problem

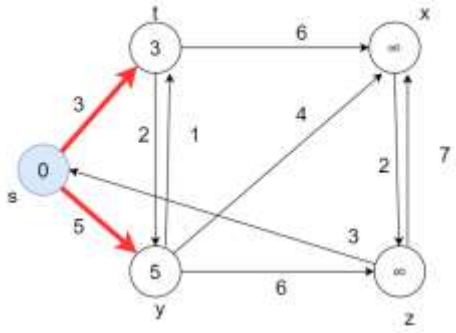
24.3-1

Run Dijkstra's algorithm on the directed graph of Figure 24.2, first using vertex s as the source and then using vertex z as the source. In the style of Figure 24.6, show the d and π values and the vertices in set S after each iteration of the **while** loop.

    - 图



    - solution
      - use s as source vertex
      - 第一步，先 init。第二步把所有的节点都放到 queue 里，然后选取节点值最小的，我们首先选到 s，看看 s 能到哪，然后更新距离，只要符合 relaxation step 的就可以更新



Set: s,t,x,y,z

d

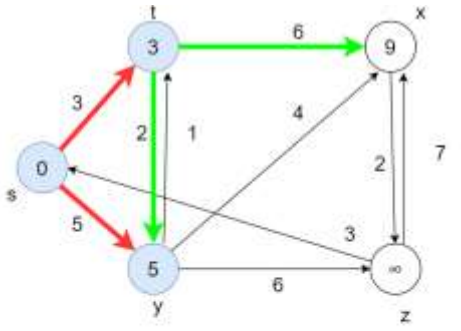
	s	t	x	y	z
	0	∞	∞	∞	∞
	0	3	∞	5	∞

- 更新对应位置

π

	s	t	x	y	z
	NIL	NIL	NIL	NIL	NIL
	NIL	s	NIL	s	NIL

- 第三步，现在，节点有 s, x, y,我们要选节点值最小的，如果选过了，就直接下一个，现在最小的是 t



Set: s,t,x,y,z

d

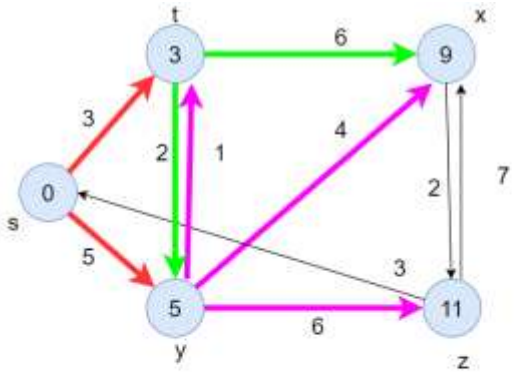
	s	t	x	y	z
	0	∞	∞	∞	∞
	0	3	∞	5	∞
	0	3	9	5	∞

- 更新对应位置

π

	s	t	x	y	z
	NIL	NIL	NIL	NIL	NIL
	NIL	s	NIL	s	NIL
	NIL	s	t	s	NIL

- 第四步，现在有 y，x，我们要选择最小值的，这里也就是 y



Set: s,t,

d

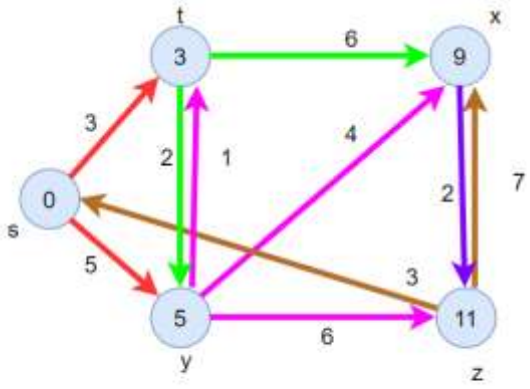
	s	t	x	y	z
init	0	∞	∞	∞	∞
pass1	0	3	∞	5	∞
pass2	0	3	9	5	∞
pass3	0	3	9	5	11

- 更新对应位置

π

	s	t	x	y	z
	NIL	NIL	NIL	NIL	NIL
	NIL	s	NIL	s	NIL
	NIL	s	t	s	NIL
	NIL	s	t	s	y

- 第五第六步，也是以此类推



Set: {

d

	s	t	x	y	z
init	0	∞	∞	∞	∞
pass1	0	3	∞	5	∞
pass2	0	3	9	5	∞
pass3	0	3	9	5	11
pass4	0	3	9	5	11
pass5	0	3	9	5	11

- 然后更新位置

π

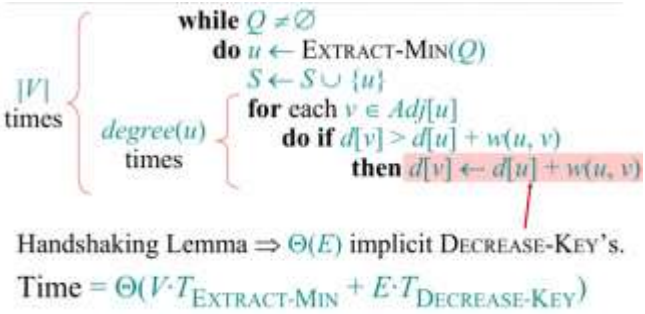
	s	t	x	y	z
	NIL	NIL	NIL	NIL	NIL
	NIL	s	NIL	s	NIL
	NIL	s	t	s	NIL
	NIL	s	t	s	y
	NIL	s	t	s	y
	NIL	s	t	s	y

- use z as source vertex

- pseudocode

```
d[s] ← 0
for each v ∈ V − {s}
    do d[v] ← ∞
S ← ∅
Q ← V          ▷ Q is a priority queue maintaining V − S
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each v ∈ Adj[u]
           do if d[v] > d[u] + w(u, v)          relaxation
              then d[v] ← d[u] + w(u, v)        step
                   ↑
              Implicit DECREASE-KEY
```

- 时间复杂度
  - 跟 prim 是一样的时间复杂度

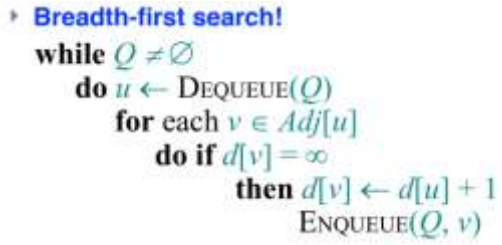


- 这个也是跟 prim 一样的

Time = $\Theta(V) \cdot T_{\text{EXTRACT-MIN}} + \Theta(E) \cdot T_{\text{DECREASE-KEY}}$

Q	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total
array	$O(V)$	$O(1)$	$O(V^2)$
binary heap	$O(\lg V)$	$O(\lg V)$	$O(E \lg V)$
Fibonacci heap	$O(\lg V)$ amortized	$O(1)$ amortized	$O(E + V \lg V)$ worst case

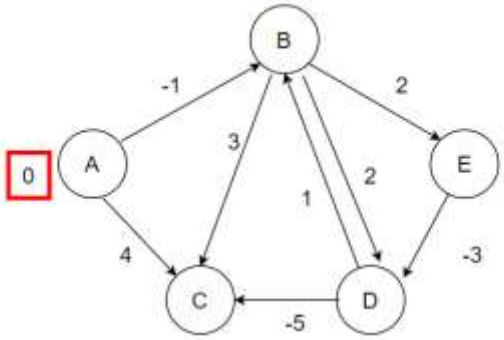
- unweighted graphs
  - 就是说假设所有的边的权重都是 1，can we improve Dijkstra？说白了，就是用 BFS



  - 这样子时间复杂度就是 O(V+E)
  - BFS 得出来的结果就是最短路径

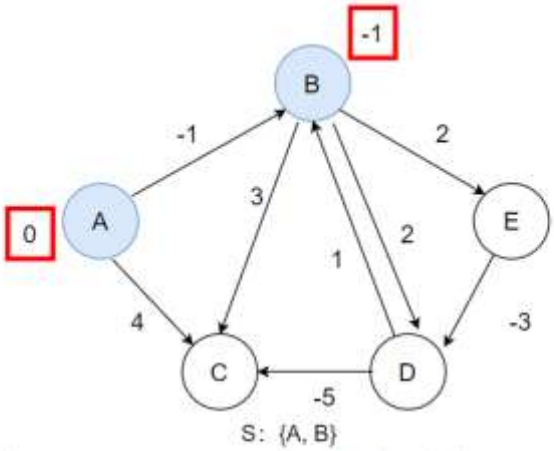
Bellman-ford algorithm

- 这个算法是用在当某些边是负数的时候，因为如果存在负数，那么就形成 negative weight cycle，如果用 Dijkstra 算法就会一直到负无穷。其实，路径存在负数也是正常的，当在金融经济的时候，就会出现这种情况
- 基本 idea
  - 就是先去走一遍所有节点，跟上面一样，做 update，也就是 relaxation，得到一组数值。然后再重复走一遍，如果都是正数，或者没有负权重的环，那么得到的结果应该是一样的，如果有 relaxation 的发生，说明有负权重环，return false
- 算法图解
  - 假设我们有这么一幅图，从节点 A 开始，记住，我们的 relaxation 是 d[v] > d[u] + w[u,v]. 我们看看 A 能去哪，A 能去 B 和 C



出发节点	终点	由哪个节点过来的（前驱）	Dist
A	A	A	0
	B		∞
	C		∞
	D		∞
	E		∞

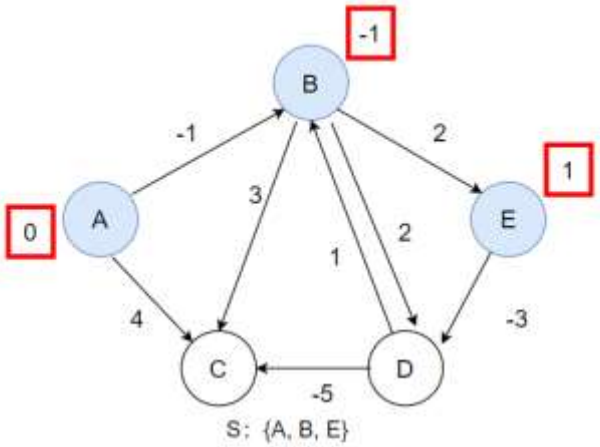
- A 能到 B 和 C，现在，我们要选择最小的，于是我们选了 B，因为 B 是-1，但是没有做 relaxation，因为公式表明后者要比前者大。现在，我们看看 B 能去哪，B 能去到 E 和 D



S：{A, B}

出发节点	终点	由哪个节点过来的（前驱）	Dist
A	A	A	0
	B	A	-1
	C	A	4
	D		∞
	E		∞

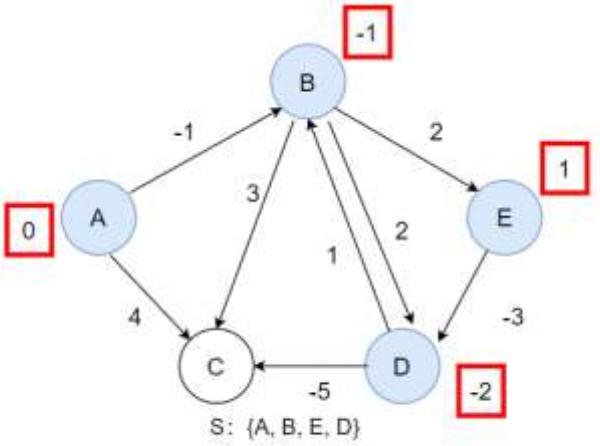
- 最后，D， E 的距离相同，我们就选 E，然后再看看 E 能去哪，E 能去 D



S：{A, B, E}

出发节点	终点	由哪个节点过来的（前驱）	Dist
A	A	A	0
	B	A	-1
	C	A	4
	D	B	1
	E	B	1

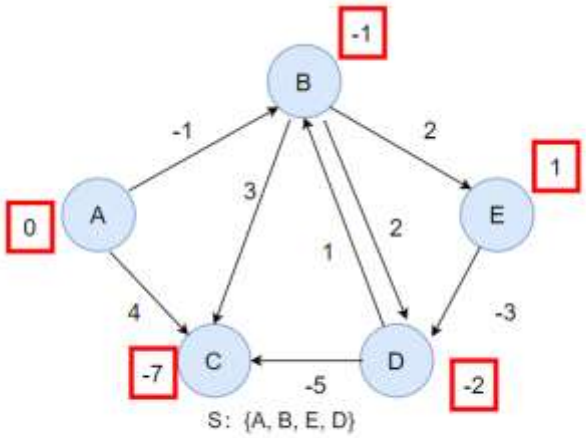
- 继续看看 D 能去哪，D 能去 C，B，但是只有 C 的距离可变小



S：{A, B, E, D}

出发节点	终点	由哪个节点过来的（前驱）	Dist
A	A	A	0
	B	A	-1
	C	A	4
	D	E	-2
	E	B	1

- 这是节点全部都已经走完了



出发节点	终点	由哪个节点过来的（前驱）	Dist
A	A	A	0
	B	A	-1
	C	D	-7
	D	E	-2
	E	B	1

- 现在，我们再走一次，从 A 节点开始，如果没有出现负权重和，那么值应该是不变的，如果变了，说明有负权重和，就要 return false，这里很明显，再走一遍依旧是一样的数值，如果 D-C 那条边反过来了，那么数值就会有变，不信可以试一试
- practice
  - 题目

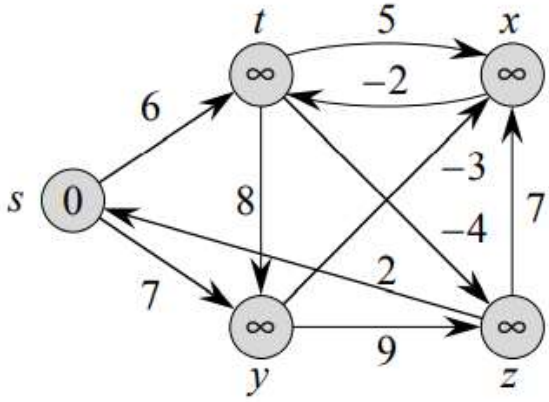
24.1-1

Run the Bellman-Ford algorithm on the directed graph of Figure 24.4, using vertex z as the source. In each pass, relax edges in the same order as in the figure, and show the d and π values after each pass. Now, change the weight of edge (z, x) to 4 and run the algorithm again, using s as the source.

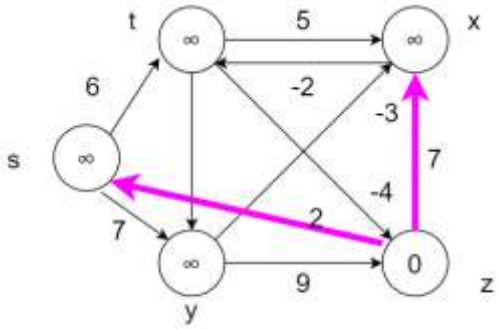
    - the order

$$(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y).$$

  - 图



- solution
  - when Z is the source vertex
    - d and π value
    - 第一步就是 init
    - 第二步就是看看我所选的点能到哪，然后就可以更新距离矩阵，也就是 d value



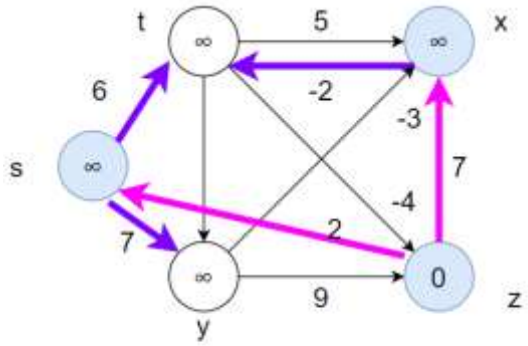
d	s	t	x	y	z
init	∞	∞	∞	∞	0
pass1	2		7		0
pass2					
pass3					
pass4					

    - 我们就在对应位置更新，这个 π 就是前驱节点，也就是说看它是从哪个点过来的

π

d	s	t	x	y	z
	NIL	NIL	NIL	NIL	NIL
pass1	z	NIL	z	NIL	NIL
pass2					
pass3					
pass4					

- 第三步就是，我们上一步选到了 x， s，我们就得看看 x 和 s 能到哪，选最短路径，比如说，x 能到 t，s 也能到 t，但是我们选最小的



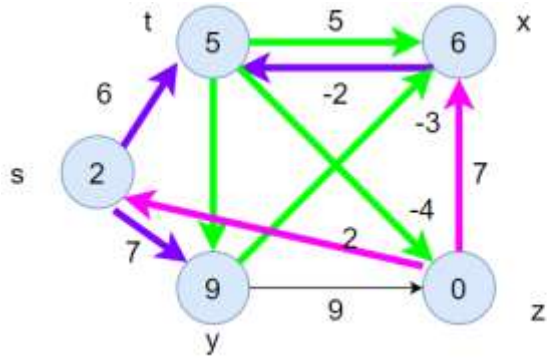
d	s	t	x	y	z
init	∞	∞	∞	∞	0
pass1	2	∞	7	∞	0
pass2	2	5	7	9	0
pass3					
pass4					

- 对应位置更新前驱节点

π

d	s	t	x	y	z
	NIL	NIL	NIL	NIL	NIL
pass1	z	NIL	z	NIL	NIL
pass2	z	x	z	s	NIL
pass3					
pass4					

- 第四步，上一步，我们选到了 t，y，这一步，就得看看 t，y 能到哪，我们发现可以到 t 可以到 y 和 x，y 可以到 x，我们就看看距离，比较一下，发现，从 y 到 x 可以更短，原来是 7，现在是 6，更新



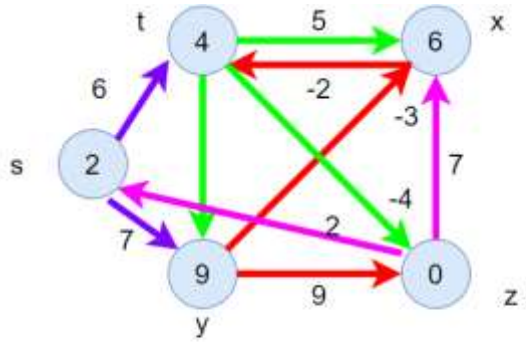
d	s	t	x	y	z
init	∞	∞	∞	∞	0
pass1	2	∞	7	∞	0
pass2	2	5	7	9	0
pass3	2	5	6	9	0
pass4					

- 对应位置更新

π

d	s	t	x	y	z
	NIL	NIL	NIL	NIL	NIL
pass1	z	NIL	z	NIL	NIL
pass2	z	x	z	s	NIL
pass3	z	x	y	s	NIL
pass4					

- 第五步，刚刚，我们选到了 x 和 y，那我们就看看他们能去哪，还能怎么更新，我们发现从 x 到 t 可以变得更短，原来是 5，现在变成了 4，更新。到这里，算法结束，不再继续



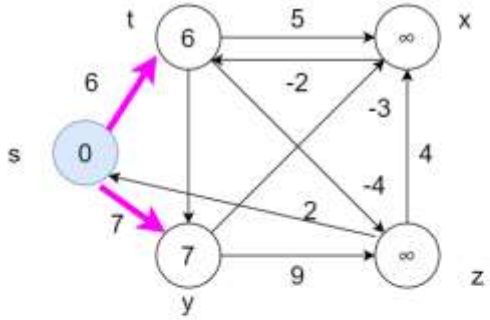
d	s	t	x	y	z
init	∞	∞	∞	∞	0
pass1	2	∞	7	∞	0
pass2	2	5	7	9	0
pass3	2	5	6	9	0
pass4	2	4	6	9	0

- 对应位置更新

π

d	s	t	x	y	z
	NIL	NIL	NIL	NIL	NIL
pass1	z	NIL	z	NIL	NIL
pass2	z	x	z	s	NIL
pass3	z	x	y	s	NIL
pass4	z	x	y	s	NIL

- when the weight change to 4 on edge (z, x), and s is the source vertex
  - d and π value
  - 第一步，先 init。第二步，看看从 source vertex 能到哪，我们能去到 t, y,更新 t，y 的 value



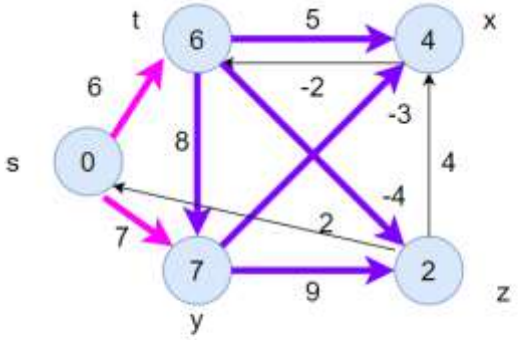
d	s	t	x	y	z
init	0	∞	∞	∞	∞
pass1	0	6	∞	7	∞
pass2					
pass3					
pass4					

- 对应位置更新

π

d	s	t	x	y	z
	NIL	NIL	NIL	NIL	NIL
pass1	NIL	s	NIL	s	NIL
pass2					
pass3					
pass4					

- 第三步，看看 t，y 能到哪，然后选取最小值



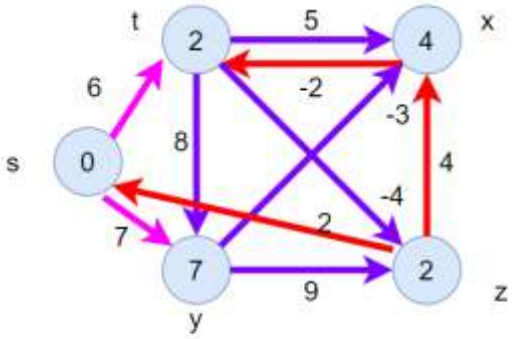
d	s	t	x	y	z
init	0	∞	∞	∞	∞
pass1	0	6	∞	7	∞
pass2	0	6	4	7	2
pass3					
pass4					

- 对应位置更新

π

d	s	t	x	y	z
	NIL	NIL	NIL	NIL	NIL
pass1	NIL	s	NIL	s	NIL
pass2	NIL	s	y	s	t
pass3					
pass4					

- 第四步，刚刚到了 x，z，现在就看看 x，z 能到哪，再选取最短路径，更新，我们发现 x-t 是可以更新的，于是更新



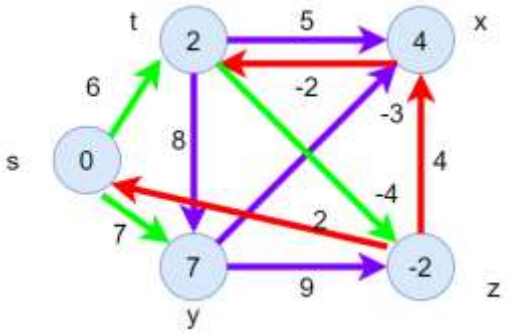
d	s	t	x	y	z
init	0	∞	∞	∞	∞
pass1	0	6	∞	7	∞
pass2	0	6	4	7	2
pass3	0	2	4	7	2
pass4					

- 对应位置更新

π

d	s	t	x	y	z
	NIL	NIL	NIL	NIL	NIL
pass1	NIL	s	NIL	s	NIL
pass2	NIL	s	y	s	t
pass3	NIL	x	y	s	t
pass4					

- 第五步，刚刚到了 t，s，那我们就看看他们能分别到哪，再继续更新距离，我们发现 x-z 是可以更新的，算法结束



d	s	t	x	y	z
init	0	∞	∞	∞	∞
pass1	0	6	∞	7	∞
pass2	0	6	4	7	2
pass3	0	2	4	7	2
pass4	0	2	4	7	-2

- 对应位置更新

π

d	s	t	x	y	z
	NIL	NIL	NIL	NIL	NIL
pass1	NIL	s	NIL	s	NIL
pass2	NIL	s	y	s	t
pass3	NIL	x	y	s	t
pass4	NIL	x	y	s	t

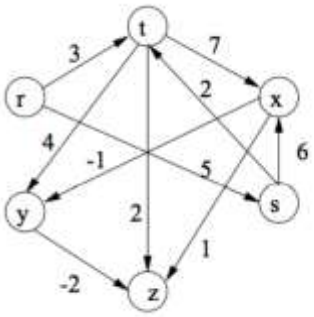
- 我们发现，z-x，它还可以再缩短，说明存在权值和为负的圈，说明这个就有问题，所以最终的结果会 return false

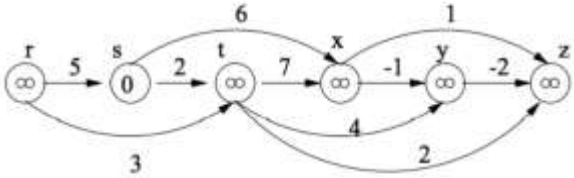
- pseudocode

```
d[s] ← 0
for each v ∈ V − {s}                    } initialization
    do d[v] ← ∞

for i ← 1 to |V| − 1
    do for each edge (u, v) ∈ E
        do if d[v] > d[u] + w(u, v)     } relaxation step
            then d[v] ← d[u] + w(u, v)
for each edge (u, v) ∈ E
    do if d[v] > d[u] + w(u, v)
        then report that a negative-weight cycle exists
                                        Time = O(VE).
```

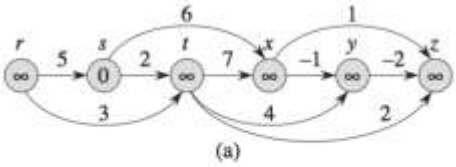
- 时间复杂度是 O(VE)，比起 Dijkstra with binary heap，时间大了一点

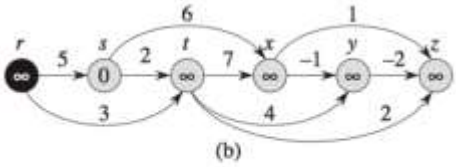
- shortest paths in DAG
  - topological sort + one pass Bellman-Ford
  - 为什么做一次 Bellman-ford 就可以了呢？因为 DAG'没有环啊，即使有负数也不影响，所以做一次 Bellman-ford 就可以了
  - 算法过程跟上面是一样的，只不过多了 topological sort
    - topological sort

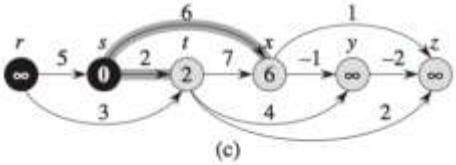


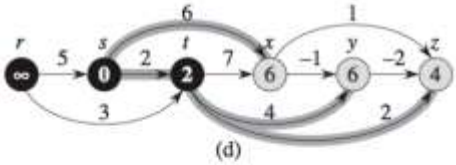


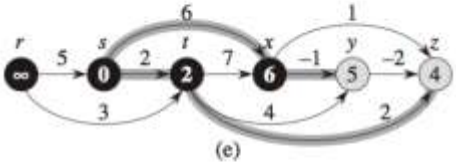
    - 然后最短路径

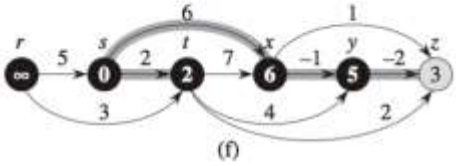


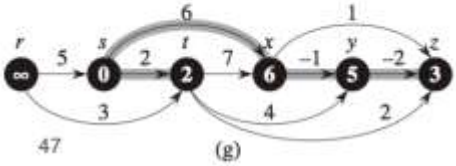




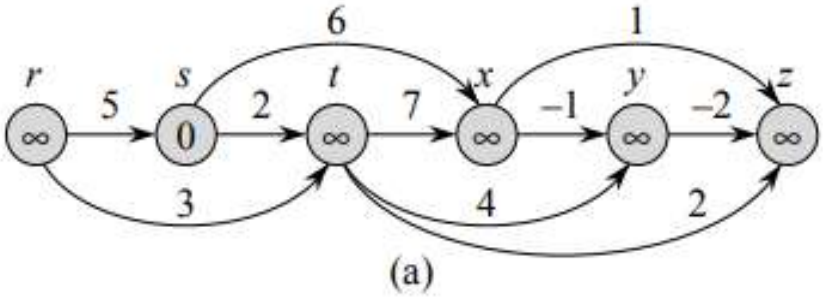








    - example，here, we use r as source vertex instead of s, so we init r s 0 and s will be ∞



(a)

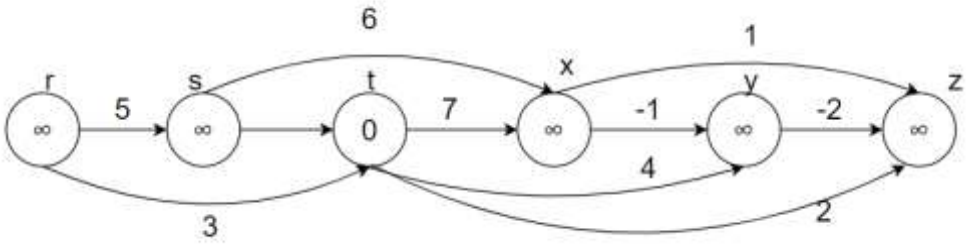
- 因为是拓扑排序，之前的边是到达不了的

- practice
  - problem，我们把题目改一下，用 t as the source

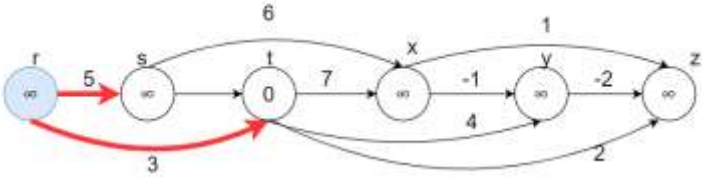
24.2-1

Run DAG-SHORTEST-PATHS on the directed graph of Figure 24.5, using vertex *r* as the source.

  - 图



  - solution
    - d value and π value
    - 第一步， init。第二步，看看 r 能去哪，这里，我们是以 t 为 source 的，任何在 t 前面的都是∞，因为这是 DAG



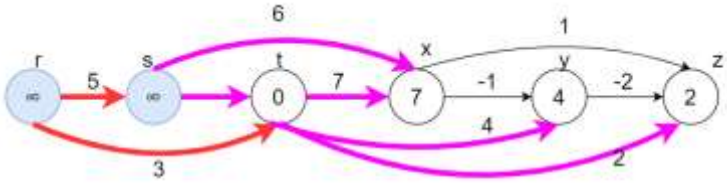
	r	s	t	x	y	z
init	∞	∞	0	∞	∞	∞
pass1	∞	∞	0	∞	∞	∞
pass2						
pass3						
pass4						
pass5						

    - 对应位置更新 π

π

	r	s	t	x	y	z
init	NIL	NIL	NIL	NIL	NIL	NIL
pass1	NIL	NIL	NIL	NIL	NIL	NIL
pass2						
pass3						
pass4						
pass5						

- 第三步，刚刚到了



	r	s	t	x	y	z
init	∞	∞	0	∞	∞	∞
pass1	∞	∞	0	∞	∞	∞
pass2	∞	∞	0	7	4	2
pass3						
pass4						
pass5						

- 更新对应位置

π

	r	s	t	x	y	z
init	NIL	NIL	NIL	NIL	NIL	NIL
pass1	NIL	NIL	NIL	NIL	NIL	NIL
pass2	NIL	NIL	NIL	t	t	t
pass3						
pass4						
pass5						

- 第四第五以此类推

- pseudocode

```
DAG-SHORTEST-PATHS(G, w, s)
1  topologically sort the vertices of G
2  INITIALIZE-SINGLE-SOURCE(G, s)
3  for each vertex u, taken in topologically sorted order
4       for each vertex v ∈ G.Adj[u]
5            RELAX(u, v, w)
```

- 时间复杂度是 O(V+E)

all pairs shortest paths

Given a weighted digraph G= (V,E) with weight function w; E-> R(R is the set of real numbers), determine the length of the shortest path (i.e., distance) between all pairs of vertices in G. 也就是给一个有权重的无向图，求各个顶点的最短距离

- 注意区别，不是求一个顶点到其他顶点的最短距离，而是各个顶点到其他顶点的最短距离
- 这是用其他算法的时间复杂度，我们需要更好的，就是 floyd-warshall algorithm
  - Nonnegative edge weights: |V| times of Dijkstra's algorithm: $O(VE + V^2 \lg V)$
  - Unweighted graphs: |V| times of BFS: $O(VE + V^2)$
  - General case: Bellman-Ford algorithm: $O(V^2E)$.
    - Dense graph (V^2 edges) $\Rightarrow \Theta(V^4)$ time in the worst case.
  - Better approach? Floyd-Warshall algorithm

Input and Output formats

- we assume that V = {1,2,3,...,n}
- Assume n vertices will be represented by n*n matrix

$$w_{ij} = \begin{cases} 0 & \text{if } i = j, \\ w(i,j) & \text{if } i \neq j \text{ and } (i,j) \in E, \\ \infty & \text{if } i \neq j \text{ and } (i,j) \notin E. \end{cases}$$

- output format: an n*n matrix D = [dij] where dij is the length of the shortest path from vertex i to j.

Floyd-Warshall Algorithm

- Dynamic programming approach
  - The Floyd-warshall algorithm considers the intermediate vertices of a shortest path. 也就是说这个算法会用到一些过度的节点
  - 这个就是过渡节点的定义

Definition: The vertices $v_2, v_3, \ldots, v_{l-1}$ are called the *intermediate vertices* of the path $p = \langle v_1, v_2, \ldots, v_{l-1}, v_l \rangle$.

- Decomposition with intermediate vertices
  - 这也就是定义从 i 到 j 的最短路径，假设它可能会经过一些过渡节点，那么，这就是 DP 的思想，可以把问题缩小范围。

Let $d_{ij}^{(k)}$ be the length of the shortest path from i to j such that *all* intermediate vertices on the path (if any) are in set $\{1, 2, \ldots, k\}$.

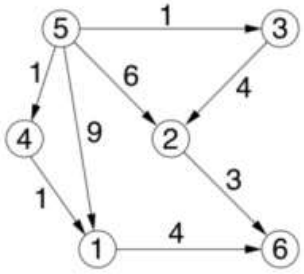
$d_{ij}^{(0)}$ is set to be w_{ij}, i.e., no intermediate vertex.
Let $D^{(k)}$ be the $n \times n$ matrix $[d_{ij}^{(k)}]$.

Claim: $d_{ij}^{(n)}$ is the distance from i to j. So our aim is to compute $D^{(n)}$.

Subproblems: compute $D^{(k)}$ for $k = 0, 1, \cdots, n$.

- exmaple

Example: how the value of $d_{5,6}^{(k)}$ changes as k varies



$d_{5,6}^{(0)}$ = INF (no path)
$d_{5,6}^{(1)}$ = 13 (5,1,6)
$d_{5,6}^{(2)}$ = 9 (5,2,6)
$d_{5,6}^{(3)}$ = 8 (5,3,2,6)
$d_{5,6}^{(4)}$ = 6 (5,4,1,6)

  - 我们要去理解这个例子，k 的取值范围是 k=0,1,..n 的，所以，就有一个，两个等等过渡节点

- Structure of shortest paths
  - 观察 1
    - A Shortest path does not contain the same vertex twice. 也就是说最短路径不会包含相同的节点，要是包含了，那就有环了
  - 观察 2
    - for a shortest path from i to j such that any intermediate vertices on the path are chosen from the set{1,2,...,k} there are two possibilities:
      - K is not a vertex on the path

        The shortest such path has length $d_{ij}^{(k-1)}$.

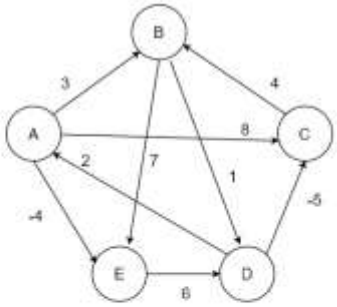
      - K is a vertex on the path

        The shortest such path has length $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$.

    - 对这玩意的理解就从 i 到 j 的路径，一种是 i 到 k，这时会有一个路径 dik，再接着是 k 到 j，这时会有一个路径 dkj，所以，其中一种可能性就是把这两个距离加起来。还有一种可能性就是不经过 k， i 到 j 有其他路径，就是 dij
  - 于是，我们可以得出状态方程

$$d_{ij}^{(k)} = \min\left\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right\}.$$

- 算法图解
  - 我们要借助两个二维矩阵，一个用来记录距离，一个用来记录从哪个节点到哪个节点
  - 假设我们现在有这么一幅图，根据图是可以写出两个矩阵的，一个是距离矩阵，表示从某个顶点到某个顶点之间的距离，还有一个是顶点矩阵



距离矩阵

D0=

	A	B	C	D	E
A	0	3	8	∞	-4
B	∞	0	∞	1	7
C	∞	4	0	∞	∞
D	2	∞	-5	0	∞
E	∞	∞	∞	6	0

顶点矩阵

π0 =

	A	B	C	D	E
A	NIL	A	A	NIL	A
B	NIL	NIL	NIL	B	B
C	NIL	C	NIL	NIL	NIL
D	D	NIL	D	NIL	NIL
E	NIL	NIL	NIL	E	NIL

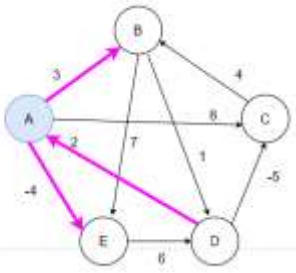
  - 此时，我们把 A 节点加进来了，那么，我们就看看以 A 为中心的节点，就是说，看看其他节点通过 A 能不能到达其他节点，或者是路劲更短，就需要进行遍历。我们从图中看到，箭头指向 A 的只有 D，所以，我们就只看 D 那一行就可以了，A 又指向 B 和 E，于是我们主要看 B 和 E 列，以及 D 那一行我们看到有两个数值是可以改变的，就更新距离矩阵，还有顶点矩阵

D1

	A	B	C	D	E
A	0	3	8	∞	-4
B	∞	0	∞	1	7
C	∞	4	0	∞	∞
D	2	5	-5	0	-2
E	∞	∞	∞	6	0

π1 =

	A	B	C	D	E
A	NIL	A	A	NIL	A
B	NIL	NIL	NIL	B	B
C	NIL	C	NIL	NIL	NIL
D	D	A	D	NIL	A
E	NIL	NIL	NIL	E	NIL



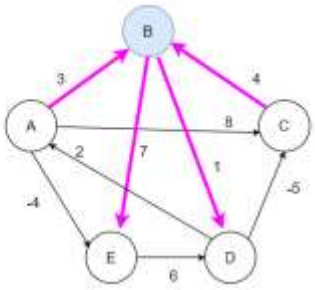
  - 此时，我们就看下一个顶点，也就是 B，以 B 为中心，看看其他点到任意一点的距离，我们注意到只有 A 和 C 是指向 B 的，所以重点关注 A 点和 C 点，也就是 A 和 C 这两行，然后 B 又指向 D 和 E，所以，也要看 DE 两列我们看到有三个是可以更新的，就要进行更新

D2

	A	B	C	D	E
A	0	3	8	4	-4
B	∞	0	∞	1	7
C	∞	4	0	5	11
D	2	5	-5	0	-2
E	∞	∞	∞	6	0

π2 =

	A	B	C	D	E
A	NIL	A	A	B	A
B	NIL	NIL	NIL	B	B
C	NIL	C	NIL	B	B
D	D	A	D	NIL	A
E	NIL	NIL	NIL	E	NIL



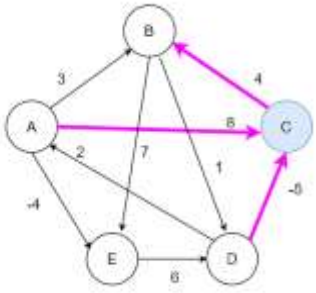
• 此时，我们就到了 C 顶点了，我们注意到 C 只有 A 和 D 指向 C，所以就关注 A，D 两行，C 又指向 B，所以要关注 B 列，A-B 的距离原来是 3，如果通过 C 就变成了 12，我们取小的，所以就不变。D 到 B 原来是 5 的，现在是-1，更新两个矩阵

D3

	A	B	C	D	E
A	0	3	8	4	-4
B	∞	0	∞	1	7
C	∞	4	0	5	11
D	2	-1	-5	0	-2
E	∞	∞	∞	6	0

π3=

	A	B	C	D	E
A	NIL	A	A	B	A
B	NIL	NIL	NIL	B	B
C	NIL	C	NIL	B	B
D	D	C	D	NIL	A
E	NIL	NIL	NIL	E	NIL



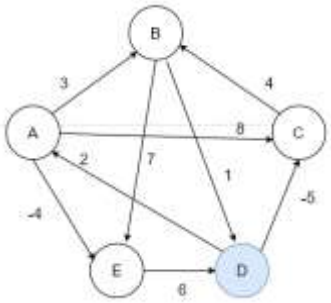
• 当 D 节点的加入，就变得复杂起来了，因为现在 ABCD 都在图里了，这说明连接节点可能是一个，可能是两个，但是 D 一定在其中，现在 B 和 E 是指向 D 的，B 和 E 和两行是重点，D 又指向 A 和 C，AC 两列也是重点

D3

	A	B	C	D	E
A	0	3	-1	4	-4
B	3	0	-4	1	-1
C	7	4	0	5	3
D	2	-1	-5	0	-2
E	8	5	1	6	0

π3=

	A	B	C	D	E
A	NIL	A	D	B	A
B	D	NIL	D	B	A
C	D	C	NIL	B	A
D	D	C	D	NIL	A
E	D	C	D	E	NIL



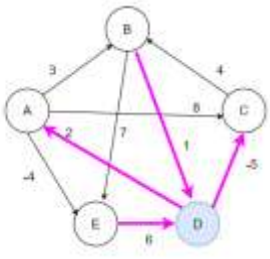
• 我们先来看刚刚说的重点行列，先重点去里交叉点，看能改变什么

D4

	A	B	C	D	E
A	0	3	-1	4	-4
B	3	0	-4	1	-1
C	7	4	0	5	3
D	2	-1	-5	0	-2
E	8	5	1	6	0

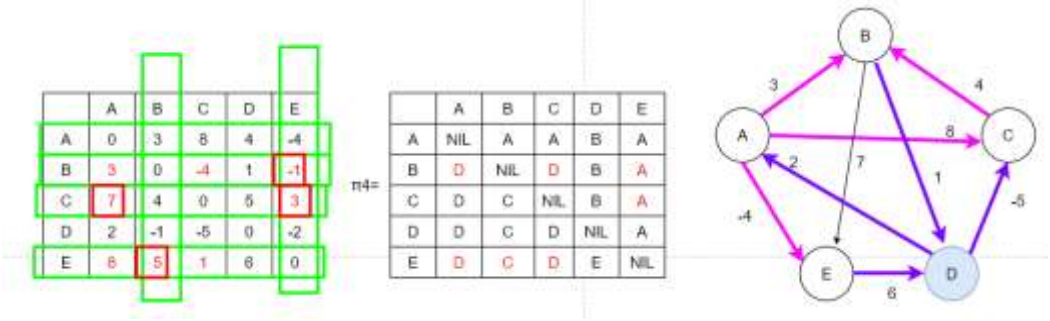
π4=

	A	B	C	D	E
A	NIL	A	D	B	A
B	D	NIL	D	B	A
C	D	C	NIL	B	A
D	D	C	D	NIL	A
E	D	C	D	E	NIL

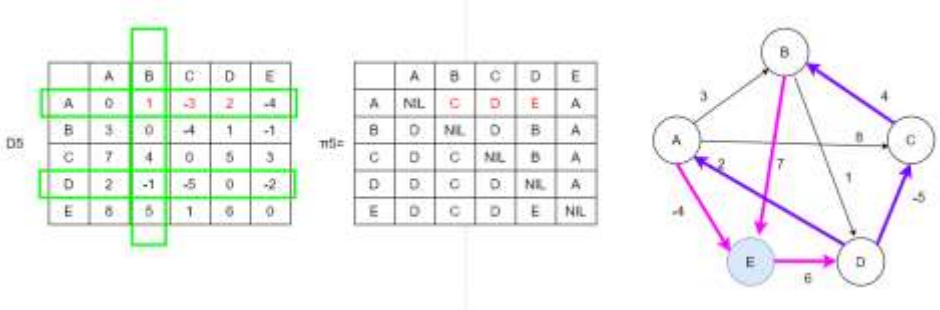


• 现在，图中加入了 BDA，BDC，EDA，EDC，然后我们还得去看看，通过这四个新的节点，我们得看每个节点的头结点和尾节点，看什么节点是能到头结点的，这里也就是 B 和 E，然后再看看尾节点能到哪，也就是 A 和 C 能到哪

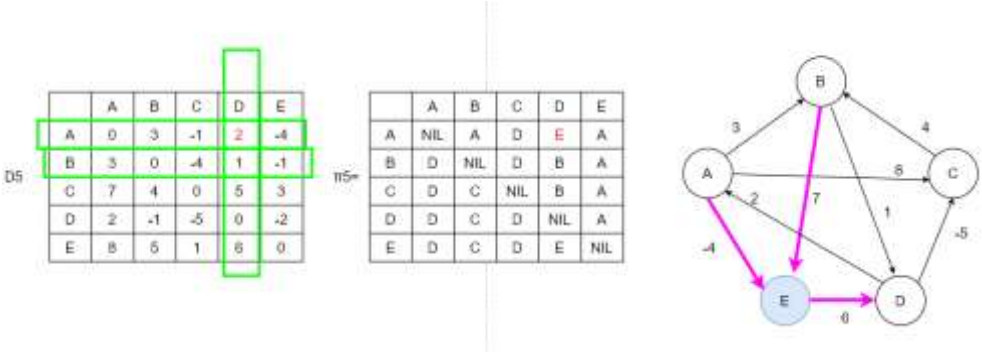
- 现在，我们就把范围给圈出来，B 和 E 行是一定要放进去的，因为是头结点，接下来就看看什么能到 B，什么能到 E，也就是 C 和 A，要把 AC 行给圈出来。其次再看看尾节点，A 和 C 能到哪，给圈出来，然后把圈出来的元素进行遍历，找出最短距离



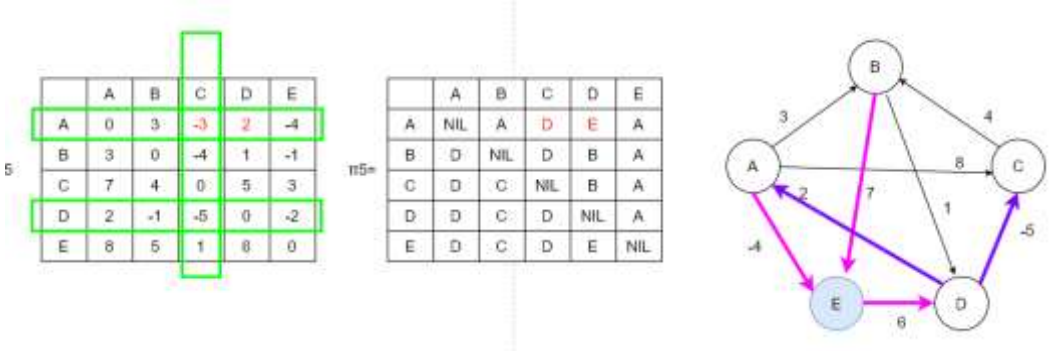
- 最终得到的结果，遍历完全部节点，算法结束



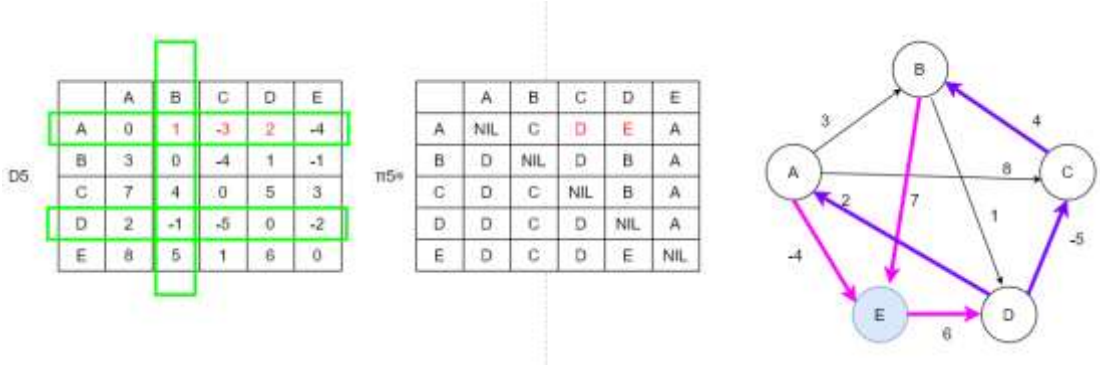
- 我们去 check 圈出来的点，发现就只有一个可以改的，于是，我们现在有了新的节点 AED，接下来就看什么能到 A，D 又能去哪



- 把头结点也圈进去，然后看看有什么可以改的，现又有一个新节点，也就是 AEDC



- 看看什么是可以到 A 的，看看 C 能去哪，看看哪个是可以改的



- pseudocode

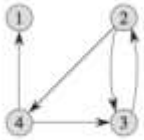
```
Floyd-Warshall(w, n)
{ for i = 1 to n do                    initialize
    for j = 1 to n do
    { d[i, j] = w[i, j];
      pred[i, j] = nil;
    }

  for k = 1 to n do                    dynamic programming
    for i = 1 to n do
      for j = 1 to n do
        if (d[i, k] + d[k, j] < d[i, j])
              {d[i, j] = d[i, k] + d[k, j];
               pred[i, j] = k;}
  return d[1..n, 1..n];
}
```

- 时间复杂度是 O(n^3)
- Extracting the shortest paths
  - we use pred[i,j] to extract the final path
    - 一旦我们找到从 i 到 j 的最短路径，而且还通过过渡点 k 的话，那么 pred[i,j] = k
    - 如果没有通过过渡点 k，那么 pred[i,j] = nil
    - 所以，如果得到的结果是 nil，说明是直达，如果不是，就轮番输出 k
  - pseudocode

```
Path(i, j)
{
  if (pred[i, j] = nil)   single edge
    output (i, j);
  else          compute the two parts of the path
  {
    Path(i, pred[i, j]);
    Path(pred[i, j], j);
  }
}
```

- Transitive closure of a Directed Graph

- 也就是说，在有向图中，不给权重，问你能不能到达其他节点，0 表示不能，1 表示可以，做法也是一样的，一个一个节点地加进来，然后看看能不能抵达
- example



$$T^{(0)}=\begin{pmatrix}1&0&0&0\\0&1&1&1\\0&1&1&0\\1&0&1&1\end{pmatrix}\quad T^{(1)}=\begin{pmatrix}1&0&0&0\\0&1&1&1\\0&1&1&0\\1&0&1&1\end{pmatrix}\quad T^{(2)}=\begin{pmatrix}1&0&0&0\\0&1&1&1\\0&1&1&1\\1&0&1&1\end{pmatrix}$$

$$T^{(3)}=\begin{pmatrix}1&0&0&0\\0&1&1&1\\0&1&1&1\\1&1&1&1\end{pmatrix}\quad T^{(4)}=\begin{pmatrix}1&0&0&0\\1&1&1&1\\1&1&1&1\\1&1&1&1\end{pmatrix}$$

NP

NP complete problem

我们已经学过如何设计一些有效的算法，像是 divide and conquer，dp, greedy algorithms 等等。

what will happen if we cannot find an efficient algorithm? 其实证明一个问题有解还相对简单, 但是要证明一个问题无解很难, 因为无法证明不存在的东西。而 NP complete problems 讲的就是这一类的东西

NP-complete 说的东西非常多

- it is not known if the problem have "efficient" solutions. 就是说有解的问题往往不是很出名
- it is known that any one of the NP-complete problems has an efficient solution then all of the NP-complete problems have efficient solutions. 如果任意一个 NP-complete problems 有解，那么，所有的 NP-problems 都有解
- 很多花时间在这上面但都失败了，要证明的东西就是 P = NP

optimization and decision problems

Decision problems

- given an input and a question regarding a problem, determine if the answer is yes or no
- 就是 yes 还是 no 的问题

optimization problems

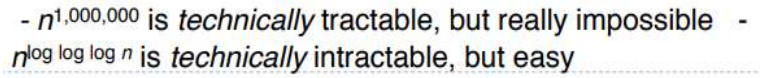
- find a solution with the "best" value

- optimization problems can be cast as decision problems that are easier to study
  - Subtopic 1

Class of "P" problem

Class P consists of decision problems that are solvable in polynomial time

- 也就是 O(n^k), for some constant k，比如说：O(n^2), O(n^3), O(1), O(nlgn)
- 这些就是 non-polynomial time: O(2^n), O(n^n), O(n!)
- 但也并不是说 non-polynomial algorithms always worse than polynomial algorithms



  - 所谓 tractable，就是 problems in P，也就是 solved in polynomial time
  - 所谓 intractable or undecidable，就是 problems not in P
    - can be solved in reasonable time only for small inputs, as they grow large, we are unable to solve them in reasonable time
    - or, cannot be solved at all
    - example: halting problem
    - Intractable problems: can be classified in various categories based on their degree of difficulty
      - NP
      - NP-complete
      - NP-hard

Nondeterministic and NP algorithms

Nondeterministic algorithm = two stage procedure

- Nondeterministic stage (guessing): generate randomly an arbitrary string that can be thought of as a candidate solution ("certificate")
- Deterministic stage (verification): take the certificate and the instance to the problem and returns YES if the certificate represents a solution

NP algorithms (Nondeterministic polynomial): verification stage is polynomial

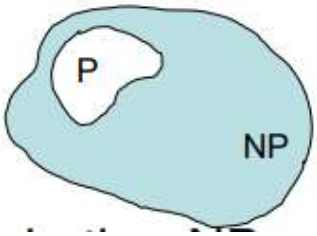
class of "NP" problems

Class NP consists of problems that could be solved by NP algorithms，那也就是说，verifiable in polynomial time. 注意注意， NP 并不是代表 “non-polynomial”

example: Hamiltonian cycle

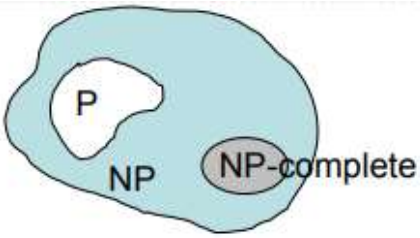
Is P=NP?

Any problem in P is also in NP: P ∈ NP



NP-completeness

NP-complete problems are defined as the hardest problems in NP: an interesting class of problems whose status in unknown,也就是未知



Most practical problems turn out to be either P or NP-complete

polynomial reductions

- Given two problems A, B, we say that A is polynomially **reducible** to B ($A \leq_p B$) if:
  - There exists a function *f* that converts the input of A to inputs of B in polynomial time
  - A(i) = YES ⇔ B(f(i)) = YES

B is NP-complete if

A problem B is **NP-complete** if:

(1) $B \in NP$

(2) $A \leq_p B$ for all $A \in NP$

- 如果 B∉ NP，只符合第二个 property 的话，那么就是 NP-hard

No polynomial time algorithm has been discovered for an NP-complete problem. 没有一个 polynomial time 算法是 NP-complete problem

NP-naming convention

NP-complete

- means problem that are 'complete' in NP, i.e. the most difficult to solve in NP

NP-hard

- stand for 'at least' as hard as NP but not necessarily in NP

NP-easy

- stand for 'at most' as hard as NP but not necessarily in NP

NP-equivalent

- means equally difficult as NP but not necessarily in NP

implication of reduction

- If $A \leq_p B$ and $B \in P$, then $A \in P$
- If $A \leq_p B$ and $A \notin P$, then $B \notin P$
- If $A \leq_p B$ and A is NP-Complete, B is NP-Hard. In addition, if $B \in NP \Rightarrow B$ is NP-Complete

Main Topic 3