CSC 263 Algorithm + Data Structure.

JAN 9, 2017.

Complexity: Bubble Sort: n^2

$T(n)$ = maximum # of steps that algorithm A takes on inputs of size n.

$t_A(x)$ = # of steps the algorithm A takes on input x.

$T(n) = \max\{t_A(x) \mid \text{Input } x \text{ of size } n\}$

$T(n): \mathbb{N} \to \mathbb{N}$
(integer to integer)

[Worst Case Running Time]

Example: SAY: $n^2 \le T(n) \le n^3$

To show the worst case $T(n) \le n^3$:

想像一下，存在一个 S: set of integers.
有个最大数值 max(S); constant C.
假如想证明 $\max(S) \ge c$,
只需要证明 S 的集中一项大于 C 即可.

$\Rightarrow \exists e \in S,\ e \ge c$

反之，想证明 $\max(S) \le c$，则需要证明 S 里的每一项都小于 C.

$\Rightarrow \forall e \in S,\ e \le c$.

考虑 $T(n) \le n^3$:

$\max\{t_A(x) \mid \text{input } x \text{ of size } n\} \le n^3$

$\Leftrightarrow$ for every input x of size n, $t_A(x) \le n^3$

$\Leftrightarrow$ for every input of size n, the algorithm A takes at ~~least~~ MOST n^3 steps.

同理，考虑 $n^2 \le T(n)$

$\max\{t_A(x) \text{ input } x \text{ of size } n\} \ge n^2$

$\Leftrightarrow$ for some input x of size n, $t_A(x) \ge n^2$.

$\Leftrightarrow$ for some input of size n, the algorithm A takes at LEAST n^2 steps.

假设： $n^2 \leq T(n) \leq n^3$

可以通过与某 constant 相乘来改变原等式.

$$\Rightarrow \frac{n^2}{\sqrt{129} \times \pi} \leq T(n) \leq 1739\, n^3$$

$\uparrow$ $\Omega(n^2)$ 　　$\uparrow$ $O(n^3)$

$T(n)$ is $O(g(n))$:

a) Intuitively : $T(n) \lesssim g(n)$ | · Within some constant factor & · for larger n(s).

b) Precisely : $\exists\, c>0$, $\exists\, n_0>0$, $\forall n \geq n_0$:

$$T(n) \leq c \cdot g(n)$$

$\Rightarrow$ for every input size of n, $n>0$, algorithm takes at most $c\,g(n)$ steps.

$T(n)$ is $\Omega(g(n))$:

a) Intuitively : $T(n) \gtrsim g(n)$ | · within some constant factor & · for larger n(s)

b) Precisely : $\exists\, c>0$, $\exists\, n_0>0$, $\forall n > n_0$:

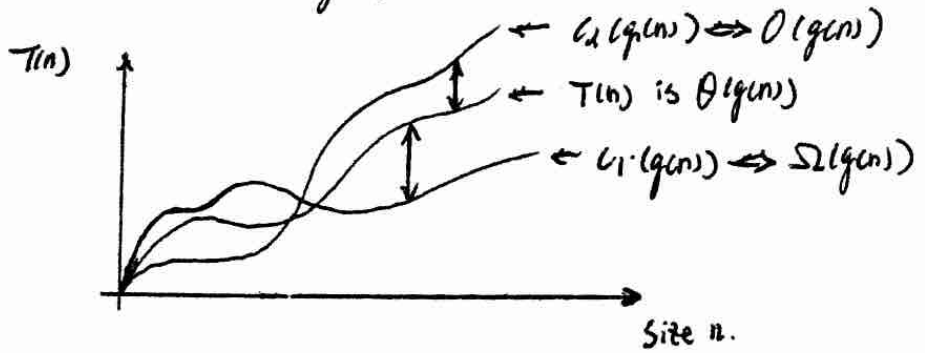
$$T(n) \geq c \cdot g(n)$$

$\Rightarrow$ for some input of size n, $n>0$, algorithm takes at least $c \cdot g(n)$ steps.

$T(n)$ is $\Theta(g(n))$:iff :

a) $T(n)$ is $O(g(n))$ (bounded above)

b) $T(n)$ is $\Omega(g(n))$ (bounded below) } Eg. constants factor.



JAN 11, 2017

Abstract Data Type (ADT)

- Describes an object and its operation (Priority Queue)

Data structure: a specific implementation of an ADT.

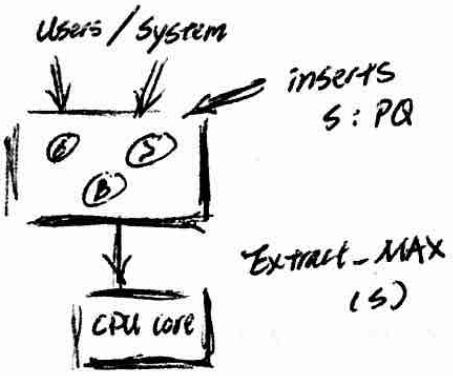
PQ (priority queue) Objects: set S of elements with 'keys' (priority) that can be compared.

PQ operations:

① insert (S, x): insert x in S $S \leftarrow S \cup \{x\}$.

② MAX (S): return an element with highest priority in S $\} x \leftarrow MAX(S)$.

$S \leftarrow S - \{x\}$



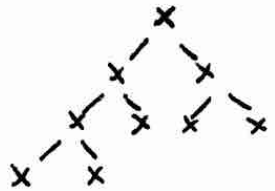
Examples of PQ data structures

	W.C. Running Time	Insert	Extract_MAX.
①	Unsorted Linked-List	O(1) (simply insert at the end)	O(n)
②	Sorted Linked-List	O(n)	O(1) (remove the top one)
③	We want HEAPS (算法导论第六章)	O(log n)	O(log n)

HEAPS :

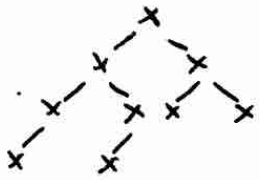
HEAP Shape : A complete Binary Tree (CBT)

Example : A CBT with $n = 9$ nodes.



fill from left to right.

Example : Not complete binary tree.



MAX-HEAP : The n elements of S go into a CBT with n nodes s.t. : MAX-HEAP property holds.

MAX-HEAP property : priority of each node $\geq$ priority of its children.

Assume $S = \{3, 4, 5, 7, 7, 9, 9, 12, 17\}$

注意：CBT 的排列顺序并不唯一。

在 Array 中表示：

HEAPSIZE = 9.

A :

①	②	③	④	⑤	⑥	⑦	⑧	⑨	⑩	⑪
17	9	12	7	7	9	5	3	4		

规律：

LEFT child of A[i] is A[2i]

RIGHT child of A[i] is A[2i+1]

Parent of A[i] is A[⌊i/2⌋]

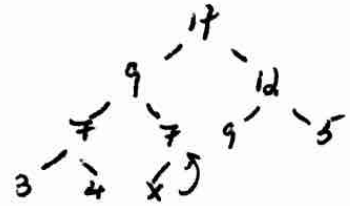
HEAP Operations:

① Maintain CBT shape

② Maintain HEAP property.

假如要 INSERT，例如 INSERT (5, x).

CBT 与 Array 将作出以下改变.



⇒ 17, 9, 12, 7, 7, 9, 5, 3, 4, x

HEAPSIZE = 10.

x 将与其 parent node 进行比较，如果小于 parent node，就保持现在位置，反之，与 parent node 互换，从而满足 HEAP property.

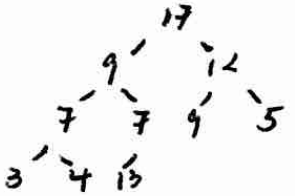
★ W.C. Running Time:

INSERT → $\Theta(\log n)$, 于 constant C.

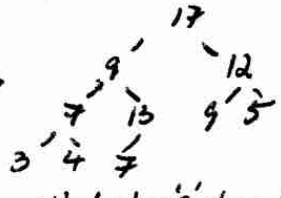
$O(\log n)$: EVERY insert takes at MOST $c \cdot \log n$ steps (height of CBT with n nodes is $\lfloor \log_2 n \rfloor \Rightarrow \log n$ ∴ 有 $\log n$ 次互换，最多!

$\Omega(\log n)$: SOME insert ($x > A[1]$) takes at LEAST $c \cdot \log n$ steps.

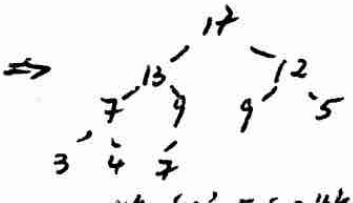
例子: INSERT (5, 13)



将 '13' 与 '7' 作比较，13 > 7 ∴ 与 '7' 互换



将 '13' 与 '9' 作比较 13 > 9 ∴ 与 '9' 互换



将 '13' 与 '17' 作比较 13 < 17 ∴ 保持现在位置.

① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩

17, 9, 12, 7, 7, 9, 5, 3, 4, 13

比较，13 > 7，互换

⇒ 17, 9, 12, 7, 13, 9, 5, 3, 4, 7

比较，13 > 9，互换

⇒ 17, 13, 12, 7, 9, 9, 5, 3, 4, 7

比较，13 < 17，不变.

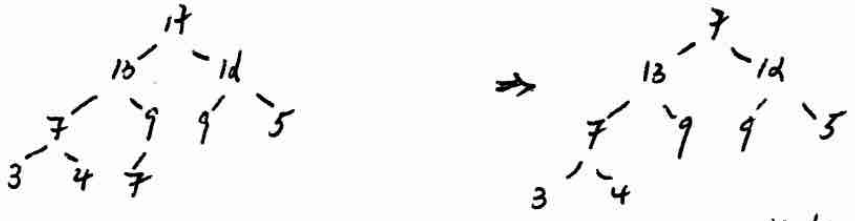
Parent of A[10] ⇒ A[⌊10/2⌋] = A[5]

Parent of A[5] ⇒ A[⌊5/2⌋] = A[2]

Parent of A[2] ⇒ A[⌊2/2⌋] = A[1]

EXTRACT_MAX(S):

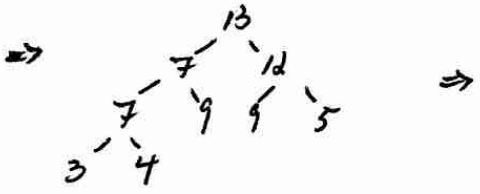
假如从CBT中拿出最大的值，CBT和ARRAY将作出以下改变：



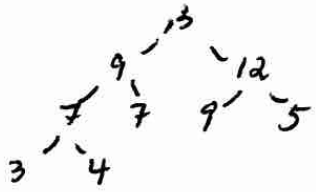
=> 去掉根后，将最底层最后的node放到根位置，随后调整位置以满足priority.

顺序：先形成CBT shape，再调整位置满足priority.

将7分别与13和12，其子node进行比较，并与最大子node互换



将7与7和9进行比较，并与9互换位置



HEAPSIZE = 10

17, 13, 12, 7, 9, 9, 5, 3, 4, 7

去掉A[1]，并将ARRAY最后一位放置在A[1]上.

=> 7, 13, 12, 7, 9, 9, 5, 3, 4 || 17 HEAPSIZE = 9

L.C of A[1] = A[1x2] = A[2]
R.C of A[1] = A[1x2+1] = A[3]
将A[1]与A[2], A[3]作比较.

=> 13, 7, 12, 7, 9, 9, 5, 3, 4 || 17 HEAPSIZE = 9

L.C of A[2] = A[2x2] = A[4]
R.C of A[2] = A[2x2+1] = A[5]
将A[2]与A[4], A[5]作比较.

=> 13, 9, 12, 7, 7, 9, 5, 3, 4 || 17 HEAPSIZE = 9.

想像EXTRACT_MAX(S) n times，将会得到一个sorted list
那边就是HEAP SORT.: 17, 13, 12, 9, ------

Jan 14. 2017

总结整理：

ADT: Abstract Data Type (如：stack [LIFO])

需要解决的4问题：

① What data is stored ?

② What operation are supported ? ⇒ Interface.

ADT为大方向，所以应该首先考虑如何把实际问题转换成ADT.

DATA STRUCTURE：

转换成ADT后再选择适合的数据类型将其实现.

如：LINKED-LIST (PUSH、POP)

定义：

Worst-Case 在某个size里，最差的一个.

$T(n)$: $\max(t(x) \mid \text{Input } x \text{ of size } n)$

Best-Case 在某个size里，最优的一个.

$T(n)$: $\min(t(x) \mid \text{Input } x \text{ of size } n)$.

分别求 W.C 与 B.C UPPER BOUND → 比较区别.
LOWER BOUND

W.C.: $\forall$ input，$\exists$ input $\leq cn$ (所有) → 上限

$\exists$ input，$\exists$ input $\geq cn$ (存在一个) → 下限

B.C.: $\exists$ input，$\exists$ input $\leq cn$ (存在一个) → 上限

$\forall$ input，$\exists$ input $\geq cn$ (所有) → 下限

Jan 16 2017.

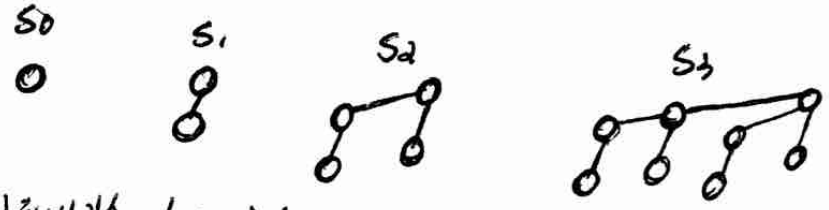
BINOMIAL HEAPS (B.H)

ADT	DS	INSERT	MIN	EXTRACT MIN	MERGE : Union(Q_1,Q_2)
PQs	HEAPS (CLRS6)	✓	✓	✓	×
MERGEABLE PQs	BINOMIAL QS	✓	✓	✓	✓
			O(log n)		

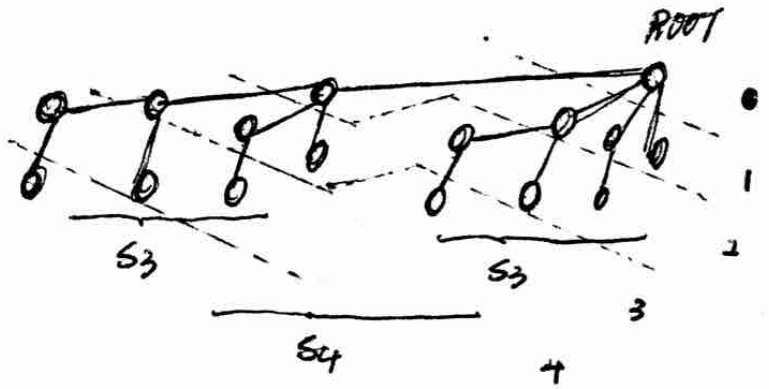
SHAPE OF BH.

BINOMIAL TREE "S_K" : $S_0 = 0$

$S_K =$ 



我们可以将一个 S_K 树
看成由两个 S_{K-1} 树组成

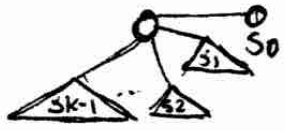


d (DEPTH)	# of NODES
0	$1 = \binom{4}{0}$
1	$4 = \binom{4}{1}$
2	$6 = \binom{4}{2}$
3	$4 = \binom{4}{3}$
4	$1 = \binom{4}{4}$

定义： $\Rightarrow \binom{K}{i}$

DEPTH : # of hops to the top

HEIGHT : # of hops to the leaf.

也可以看成单独一个 NODE
分别与 $S_0, S_1, \ldots, S_{K-1}$ $\Rightarrow$ 

S_K	# OF NODES	HEIGHT (LOG OF # OF NODES)
S_0	1	0
S_1	2	1
S_2	4	2
S_3	8	3
S_K	2^K	K

BINOMIAL FOREST.

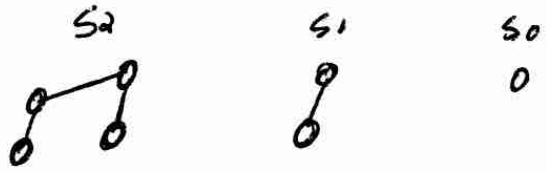
BINOMIAL FOREST WITH n NODES.

F_n is a sequence of S_k trees with strictly increasing k. and a total of n nodes.

例子：一个有 7 个 NODES 的 BF.

$n = 7 = \langle 111 \rangle_2 = 2^2 + 2^1 + 2^0 = 4 + 2 + 1$

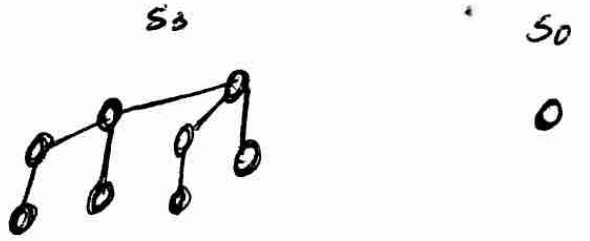
$\Rightarrow$ 由 S_2, S_1, S_0 组成.



例子：一个有 9 个 NODES 的 BF.

$n = 9 = \langle 1001 \rangle_2 = 2^3 + 2^0 = 8 + 1$

$\therefore \Rightarrow$ 由 S_3 和 S_0 组成



A BF F_n with $n = \langle b_k, b_{k-1}, \ldots, b_1, b_0 \rangle$ NODES. (BITS)

$F_n = \langle$ all S_j such that $b_j = 1 \rangle$ $\quad l = \lfloor \log_2 n \rfloor$

LET $\alpha(n)$ = # of 1's in the binary representation of n

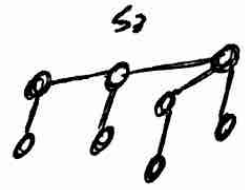
a) F_n has $\alpha(n)$ trees

b) F_n has $n - \alpha(n)$ edges.

例子：一个有9个 NODES 的 BT

$n=9=(1001)_2=2^3+2^0=8+1$

由 S_3 和 S_0 组成



$n=9$ → NODE 的数量

$\alpha(n)=2$ ⇒ 由多少棵数对组成

$n-\alpha(n)=7$ ⇒ Edges 数量

MIN - BINOMIAL HEAP 定义

A min binomial heap of n keys is a binomial forest of n nodes F_n such that:

a) each node of F_n stores one key.

b) each binomial tree S_K in the F_n is min heap ordered.

例子 n=7 (BINOMIAL HEAP)

$S=\{10, 13, 1, 3, 8, 18, 十\}$

S_0 S_1 S_2

很容易得出 S_0 ⇒ (10)

S_1 由 13 和 1 组成.

根据 min heap property 对 13 和 1 进行比较可得

S_1 ⇒

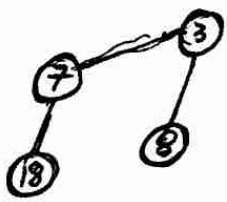
S_2 由 3, 8, 18, 十

可拆分为两组 S_1 ⇒ {3,8}, {18, 十}.

根据 min heap property 对 {3.8}, {18, 十} 进行比较.

再将其 ROOT 进行比较，组合成 S_2.

⇒



BINOMIAL TREE MERGE. JAN 18 2017

MIN, EXTRACT_MIN, INSERT
+UNION(Q1, Q2)

PRIORITY Qs

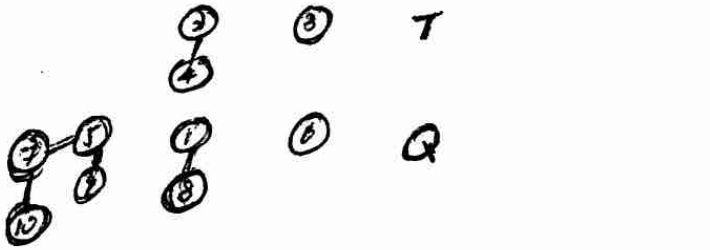
IN (O log n) times.

假设 BINOMIAL TREE T:

BINOMIAL TREE Q:

MERGE T & E.

第一步:

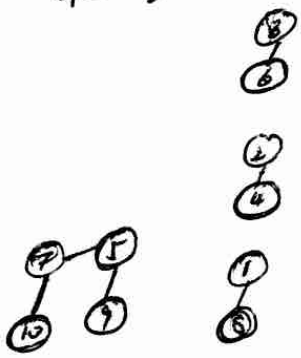


⇒ ③ 和 ⑥ MERGE.

根据 MIN HEAP 性质

可得  5, 并 Carry 到下一位.

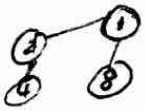
第二步



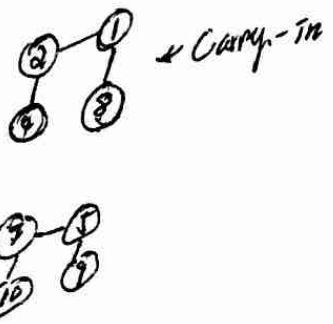
← Carry-in 进位.

X

⇒  和  MERGE

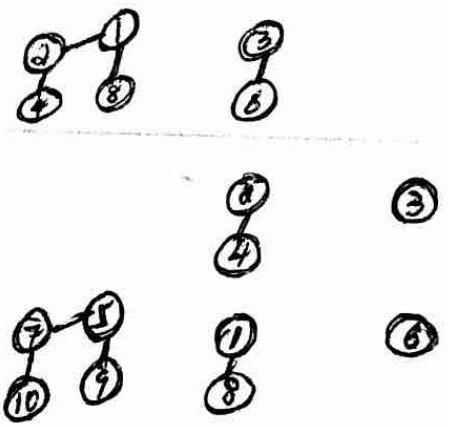
可得  5d 并 Carry 到下一位

第三步



← Carry-in

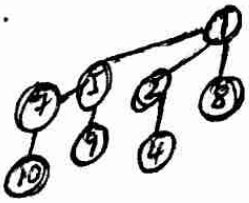
第四步 整合结果



X

⇒  和 MERGE.

CARRY 到下一位.

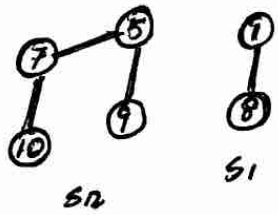


X

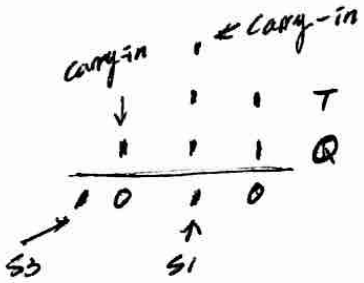
X

以上一个例子

T:  → <1, 1>

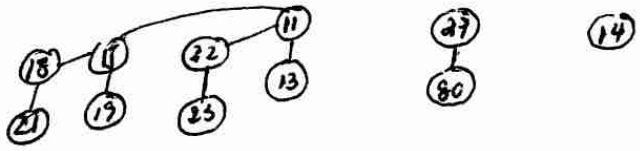
Q:  → <1, 1, 1>

可以看作二进制运算：



与上一个例子的结果吻合.

DELETE_MIN(T)

例子： T = 

Delete_MIN(T):

① Scan roots to find smallest element

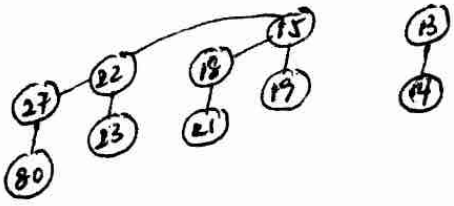
② Delete element

③ Merge resulting BQ's

Carry: 

$T_1 = T - S_3$ 

\+ $T_2 = S_3 - \{x\}$ 

$T \leftarrow UNION(T_1, T_2)$ 

T: INSERT(T, x)
UNION(T, Q)

假设 T ⇒ $S_k, \ldots, S_1, S_0$ ← carry

$1 \quad x$

$1, 0, 0, \ldots, 0, 0, 0$

⇒ log n to insert single element. w.c.

$S_k \ldots S_0$ ← even

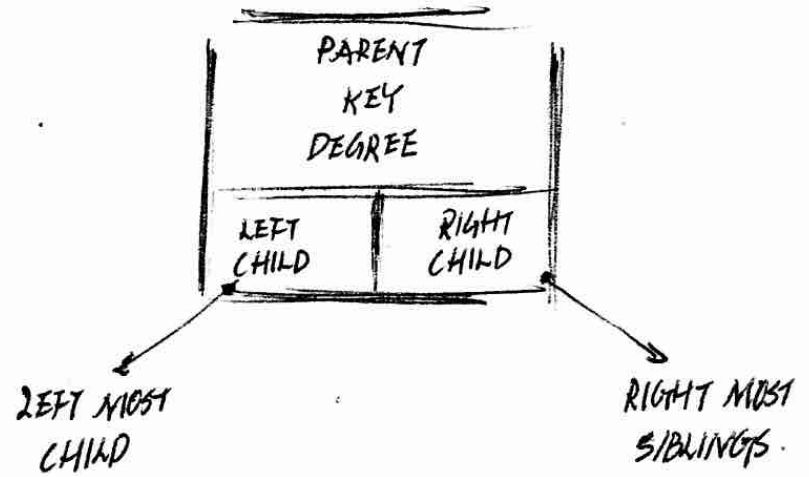
$1, 1, \ldots, 1, 1 \quad x \quad T$

$1 \quad x$

$1, 1, \ldots, 1, 1, 1$

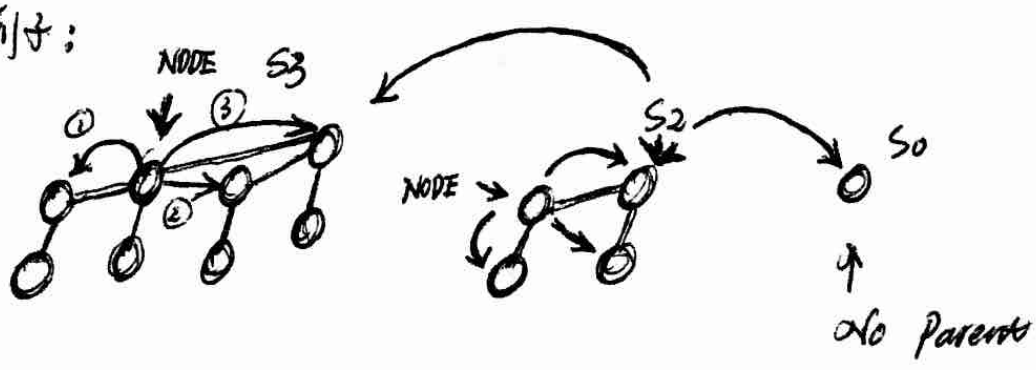
⇒ 0 comparison, b.c.

NODE 的结构



使用这样的结构是为了保证在 DELETE 或其他情况下，不用使胖树结构瓦散。

例子：



EVERYTHING IS CONNECTED

DICTIONARY ADT : SET S WITH KEYS

OPERATIONS : - SEARCH(x)
- INSERT(x)
- DELETE(x)

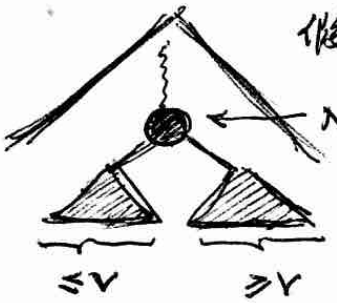
} x is a KEY.

我们的目标 complexity 为 $O(\lg n)$

BST : BINARY TREE (二叉树) T with BST properties.

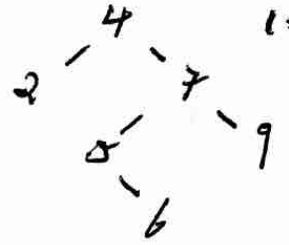
BST PROPERTIES.

FOR ALL NODES →



INORDER TRVERSAL ⇒ 顺序：左中右

例子：假设 SET S = {2, 4, 5, 6, 7, 9}.



(排序不唯一)

逆时针旋转 →

操作：

SEARCH 5 : 跟 root 比较，决定向左还是向右.

INSERT 8 : 向上，比较，决定左右.

COMPLEXITY : HEIGHT OF THE TREE.

DELETE(x)

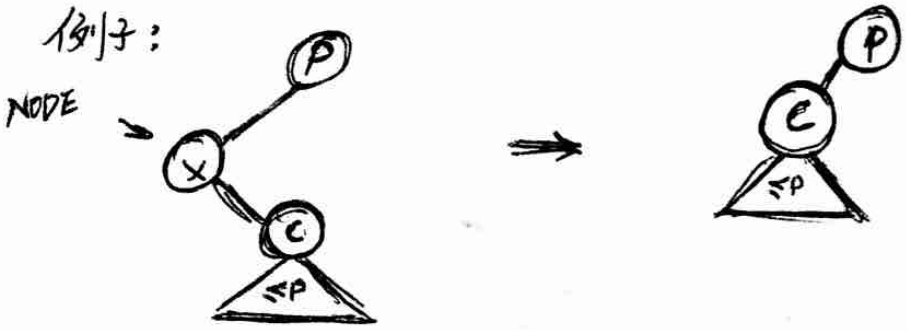
① 第一种情况：x is a leaf

直接移除。

② 第二种情况：x has only one child

用其子将其替换。

例子：

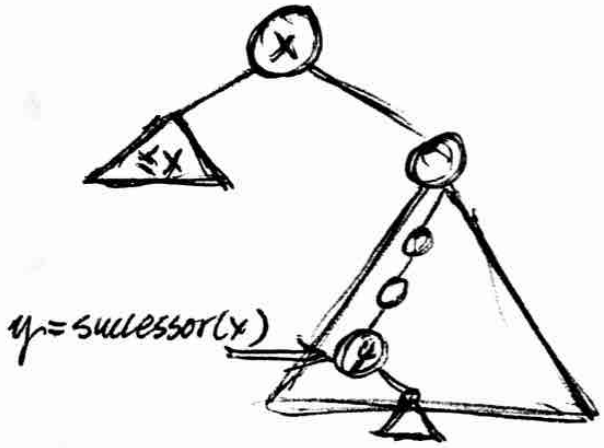


③ 第三种情况：x has two children

ⓐ 找 $y = succ(x)$

ⓑ 用 y 将 x 替换

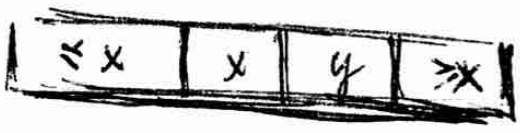
ⓒ 将原来 y 删除。



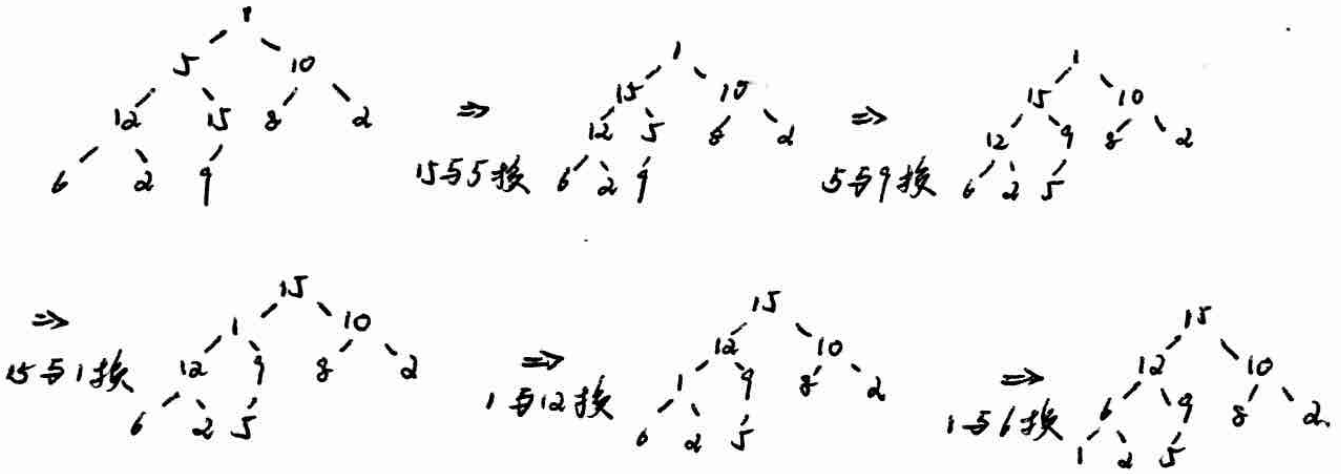
$y = successor(x)$

右 subtree 最小的 NODE。

y 最多有一个子 node (右边)



做设下：HEAPIFY过程演示



PRECONDITION:

the binary trees rooted at LEFT(i) and RIGHT(i) are heaps.

POSTCONDITION:

the subtree rooted at index i is a heap.

代码：

```
MAX_HEAPIFY(A, i)
① l = LEFT(i)
② R = RIGHT(i)
③ if L ≤ A.heap-size and A[L] > A[i]
④       largest = L
⑤ else largest = i
⑥ if R ≤ A.heap-size and A[R] > A[largest]
⑦       largest = R
⑧ if largest ≠ i
⑨       exchange A[i] with A[largest]
⑩       MAX_HEAPIFY(A, largest)
```

How many time would MAX_HEAPIFY be call.

Upper Bound:

at most c·h steps where h is the height of the subtree starting at i.

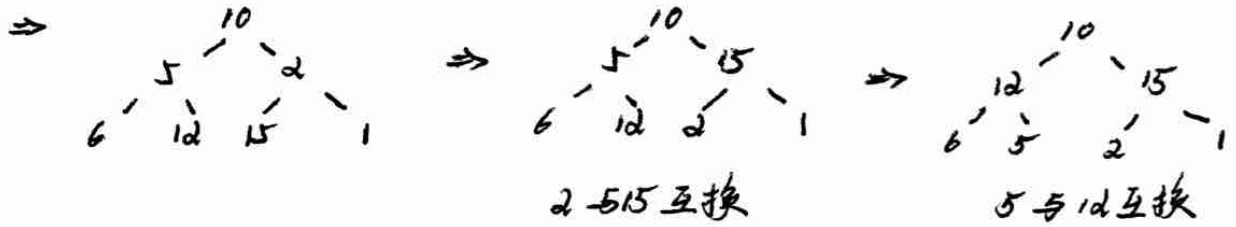
⇒ at most log n (height is ⌊lg n⌋)

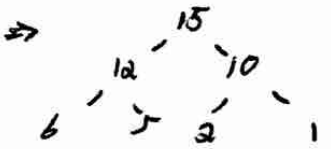
$c \cdot h \leq c \lg(n)$

$\Rightarrow O(\lg(n))$

MAKING AN ARRAY INTO A HEAP

例子: [10, 5, 2, 6, 12, 15, 1]






```
BUILD-MAX-HEAP(A)
    heap_size = length(A)
    for i = ⌊length(A)/2⌋ down to 1
        MAX-HEAPIFY(A, i)
```

$\frac{n}{2}$ iterations

$\frac{n}{2} \cdot \log n$

定义:

depth(x): length of longest path from root to x.

height(x): length of longest path from x to a leaf.

At depth of d: 2^d nodes

Height at depth d = SAY 'h' is height of the heap
= h-d.

$\Rightarrow O(n \log n)$ [DEPENDS ON THE HEIGHT].

代码: HEAPSORT(A)

```
BUILD-MAX-HEAP(A)
for i = length(A) down to 2.
    exchange A[1] with A[i]
    heap_size[A] -= 1
    MAX-HEAPIFY(A, 1)
```

ACTUAL TIMES OF CALLING MAX-HEAPIFY.

$$\text{cost} \le \sum_{d=0}^{h-1} \underbrace{2^d (h-d)}_{\text{all nodes}}$$

LET $i = h-d$

$\Rightarrow d = h-i$

$$\Rightarrow \sum_{i=1}^{h} 2^{h-i} \cdot i$$

$$= \sum_{i=1}^{h} 2^h \cdot 2^{-i} \cdot i$$

$$= 2^h \sum_{i=1}^{h} \frac{i}{2^i} \le 2^h \sum_{i=0}^{\infty} \frac{i}{2^i}$$

$$\le n \sum_{i=0}^{\infty} \frac{i}{2^i}$$

$$= n \frac{\frac{1}{2}}{(1-\frac{1}{2})^2}$$

$$= 2n$$

$\therefore O(n \log_2(n))$

$\therefore O(n)$, $\Omega(n)$

$\Rightarrow \Theta(n)$

FORMULA:

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1}-1}{x-1}$$

$|x| < 1$, $n \to \infty$

$$\Rightarrow \sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

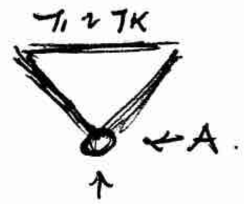
DERIVITIVE ↓

$$\sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2}$$

HEAP 的运用 (大致思路)

① How to find K-largest integer in a text file full of int numbers.

选择使用 min-heap，size = K.
把前 K 个东西放进去 ⇒



根据 min-heap 性质，A，min-heap 的根是最小的 element.

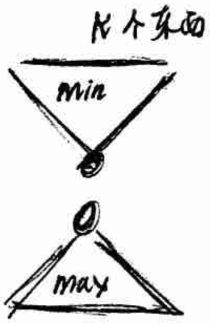
步骤：

取 T_{k+1}，如果 T_{k+1} < A，什么都不做，如果 T_{k+1} > A，EXTRACT A，把 T_{k+1} INSERT 进去.

⇒ $O(n \log k)$ (WORST CASE) EXTRACT_MIN & INSERT

② Suppose I have a lot of numbers, K-th largest item during any time of the process

典型的面试题

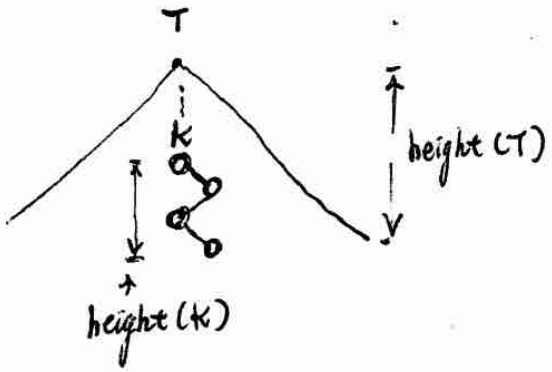


往 min 放，满了放 MAX.

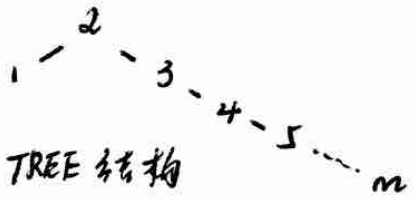
Jan 23 . 2017.

ADT	OPS	DATA STRUC.
DICTIONARY	• INSERT • DELETE • SEARCH	• BSTS (C2RS i2) • BALANCED TREES. • AVL TREES • 2~3 TREES • RED-BLACK-TREES.

假设 BINARY TREE

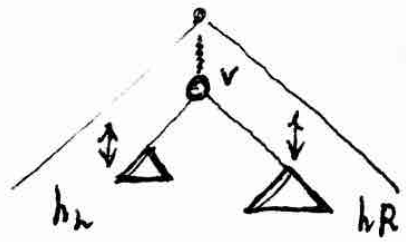


做设一个 BINARY TREE



可见 BINARY TREE 结构
罗重失衡.

BALANCE FACTOR (BF)



定义: HEIGHT(•) = 0 (T is a single node)
HEIGHT( ) = -1 (T is an empty tree)

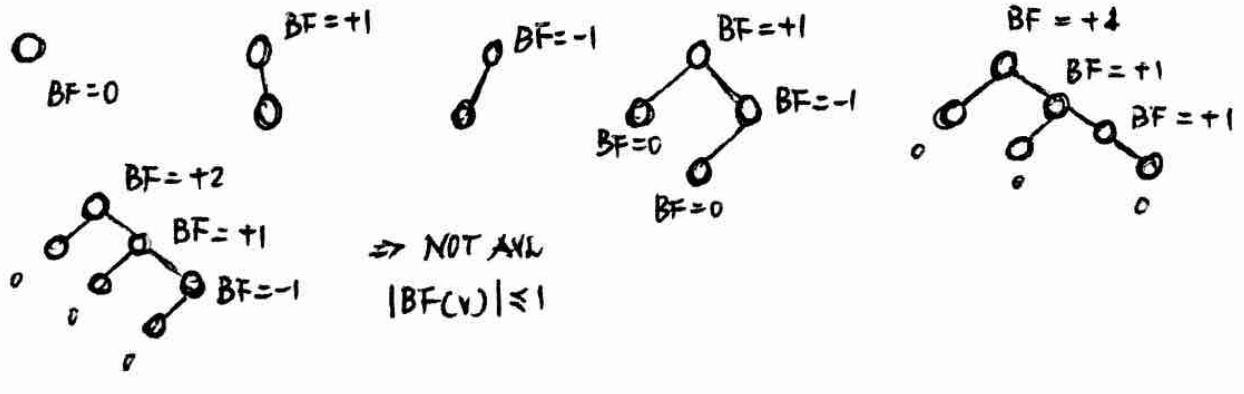
定义: DIFFERENCE IN HEIGHT IS BALANCE FACTOR

$BF(v) = h_R - h_L$ 右子树的高与左子树的高之差.

$\Rightarrow$ AVL TREE $\equiv$ BST TREE T
where $\forall v \in T$, $|BF(v)| \leq 1$
$\Rightarrow (-1, 0, 1)$

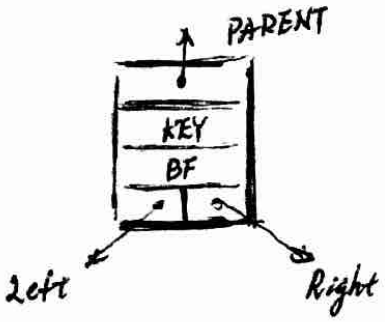
AN AVL TREE with n nodes,
has height $\theta(\lg n)$.
精准来说: $\lceil h(T) < 1.44 \log_2(n+2) \rceil$

AVL TREE 例子:



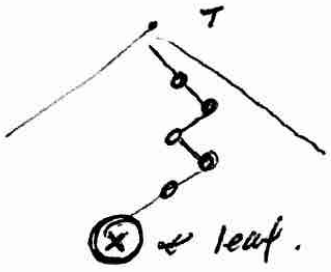
$\Rightarrow$ NOT AVL
$|BF(v)| \leq 1$

NODE 长这样



AVR satisfied binary tree properties.

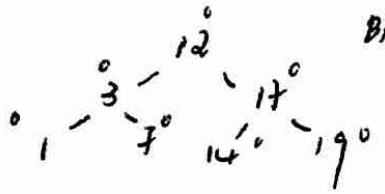
OPERATION: INSERT(T,x)



假设 ARRAY 为 [1, 3, 7, 12, 14, 17, 19]

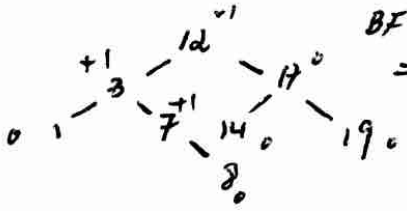
PERFECT BALANCE.

可以建 BINARY TREE:



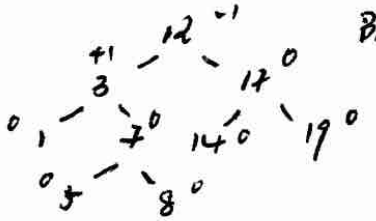
BF=0 => AVR TREE 成立.

INSERT(T,8)



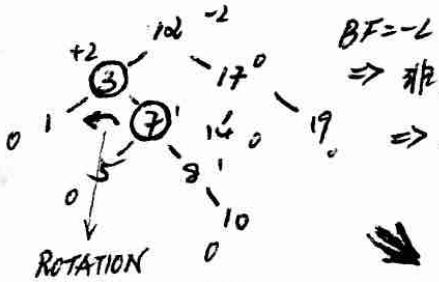
BF=-1 => AVR TREE 成立.

INSERT(T,5)



BF=-1 => AVR TREE 成立.

INSERT(T,10)



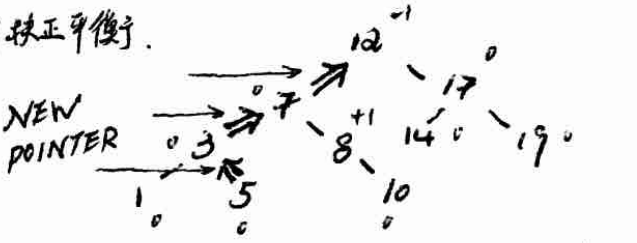
BF=-2 => 非AVL TREE => RIGHT HEAVY

AFTER ROTATION ⟹

a) is rebalanced

b) is the same (h=2) as before insertion

c) binary tree property is maintained.

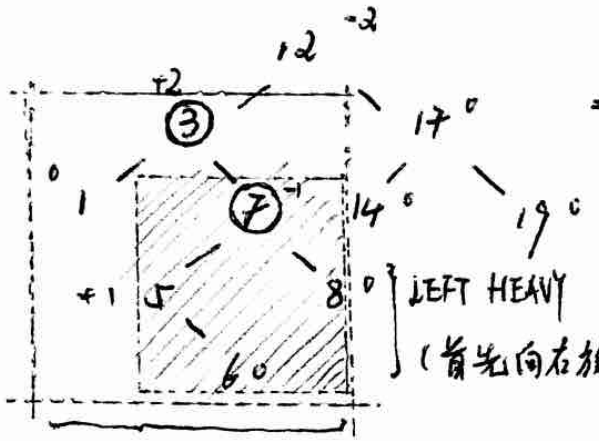


=> AVL TREE 成立

BF=-1

该例子为 Single Rotation 后的形成的 AVL TREE.

INSERT(T, 6)



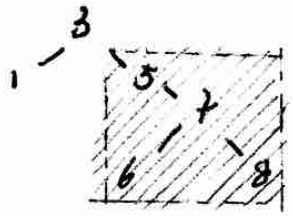
BF = -2

⇒ 非 AVL TREE

须要通过 ROTATION 平衡.

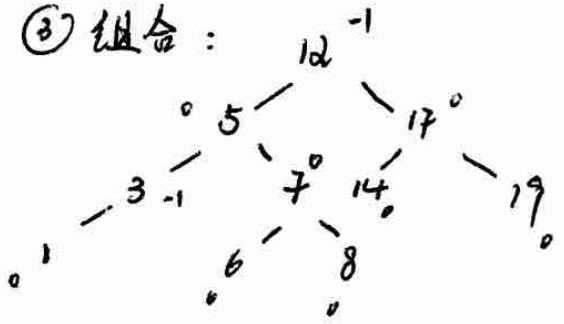
观察：在 ③ NODE 上，其树属于 RIGHT HEAVY，可反观 ⑦ NODE，其树属于 LEFT HEAVY. 所以须要通过两次不同方向的 ROTATION 进行平衡。

① 首次向右旋转后可得：



② 第二次向左旋转后可得：

③ 组合：



BF = -1

⇒ AVL TREE 成立.

总结：AFTER DOUBLE ROTATION: LEFT SUBTREE OF Id.

a) is rebalanced

b) is the same ($h=2$) as before insertion of 6.

c) binary search tree property is maintained.

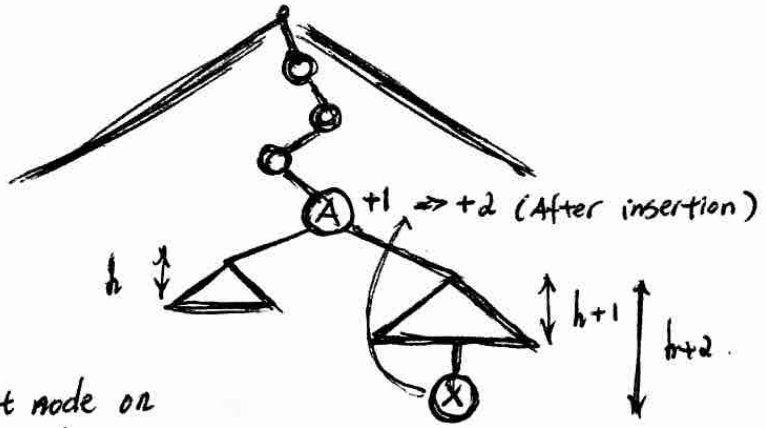
Jan 25 2017

AVL TREE 性质回顾：

- Special Binary Search Tree
- Balance Factor : $|BF| \leq 1$

  Difference between height of right sub-tree and height of left sub-tree.

General Case : Insertion of x



假设：

Let (A) be the first node on the path from (X) to the root of the AVL tree T that loses its balance, i.e. BF(A) becomes +2 or -2 after X is inserted.

通过 Rotation 将 AVL TREE 平衡

平衡后可以确保：

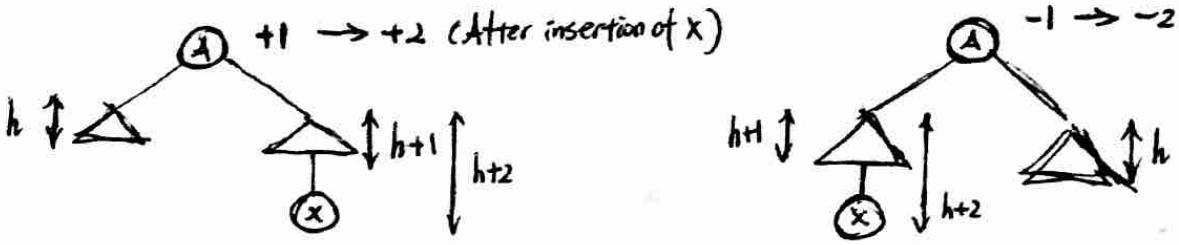
a) preserves the binary search tree order

b) rebalances the AVL Tree.

c) restores the height to what it was before the insertion of x (h+2)

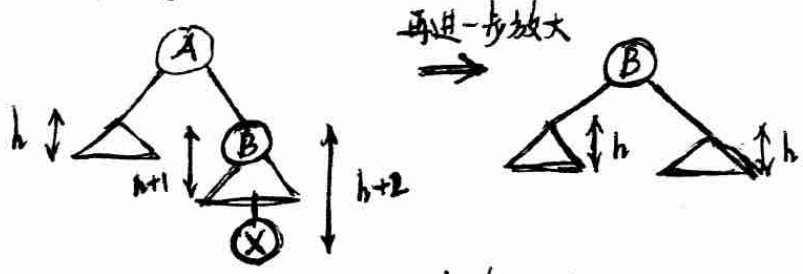
分 CASE 讨论：

CASE 1 ：



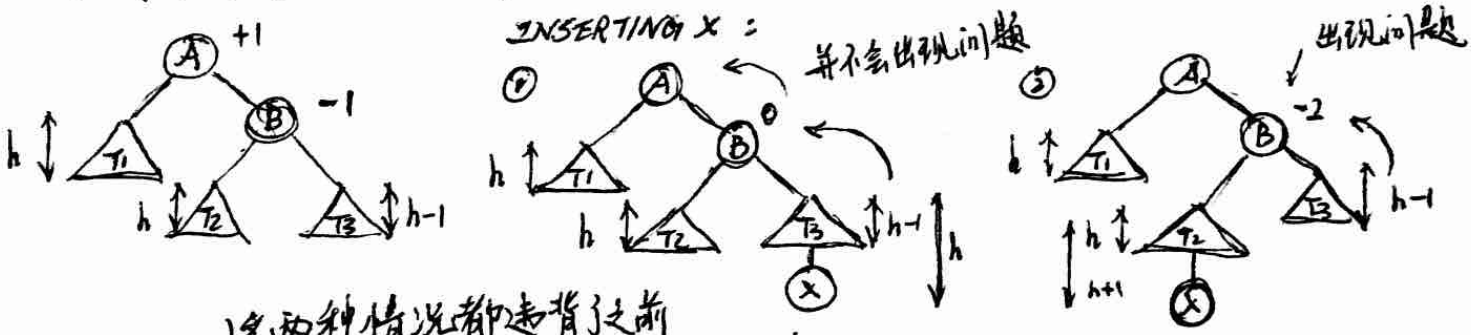
这两种情况是完全 symmetric，可以
综合成一种情况讨论. 以下，我们用
左边的情况进行讨论。

放大左边 sub-tree：



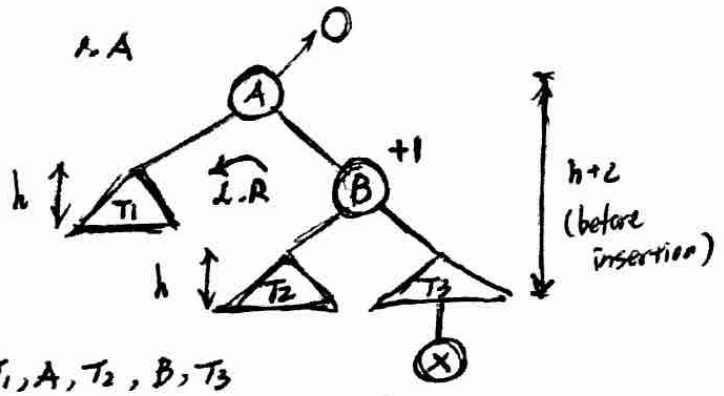
为什么 B 的 sub-tree 的
高度是一致的？

证明 B 两边 sub-tree 的高度一致：

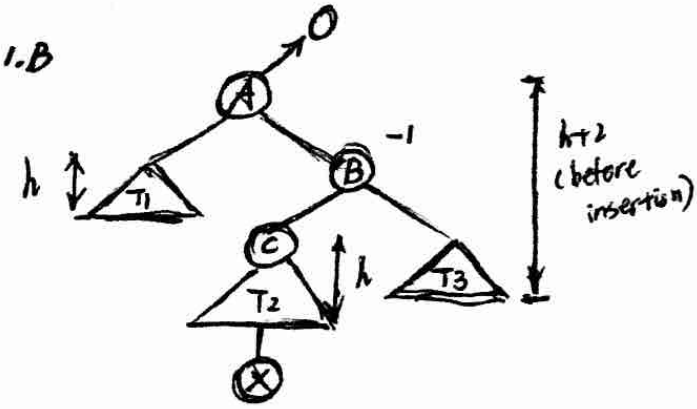


这两种情况都违背了之前
说 A 是第一个 lose balance 的 node
所以，根据 contradiction，B 两边 sub-tree 的高度一致。

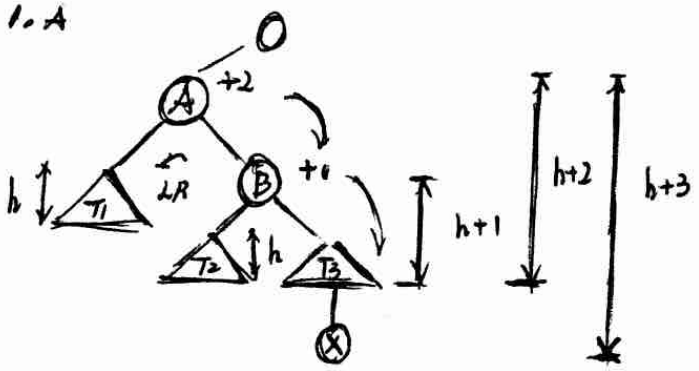
根据 CASE 1，可分两种情况



T1, A, T2, B, T3
representation as Array.



1.A

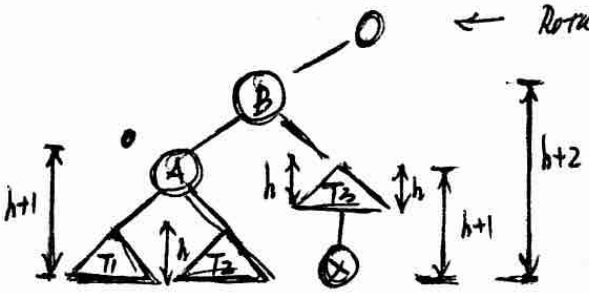


LEANING TO THE RIGHT

通过逆时针旋转FIX.

AFTER ROTATION.

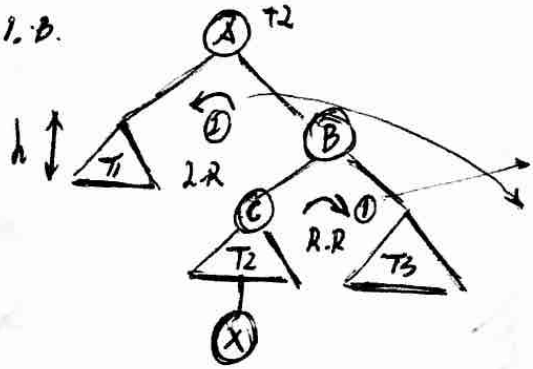
← Rotation 后该 node 的 BF 也不会改变.



T1, A, T2, B, T3

- Preserved binary search tree order.
- rebalances AVL Tree
- restores the height to what it was before => h+2.

1.B.



须要通过两次 rotation 来平衡.

首先顺时针

然后逆时针.

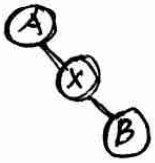
1.B 可分两个大 Sub-case，其中一个 sub-case 再分两个小 case

1.B
- ① 1.B.α
- ② 1.B.β
  - ③ 1.B.β.①
  - ④ 1.B.β.②

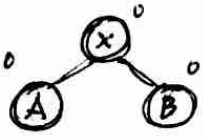
1.B.α : $h = -1 \Rightarrow T_1, T_2, T_3$ are null trees.

=> ARRAY: A, X, B.

第一次 ROTATION:



第二次 ROTATION:

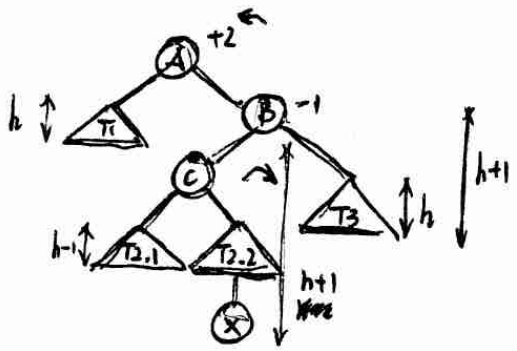


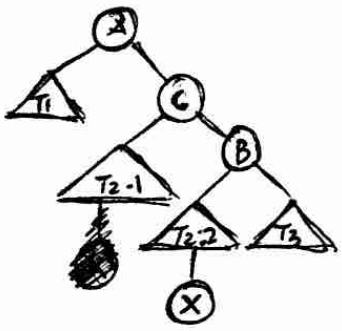
h=1 => ARRAY: A, X, B.

- Order preserved
- restored the height.

1.B.β

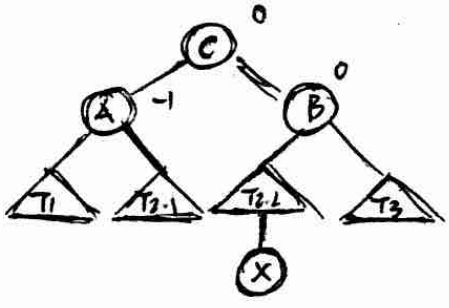
1.B.β.①





第一次 Rotation

顺时针旋转

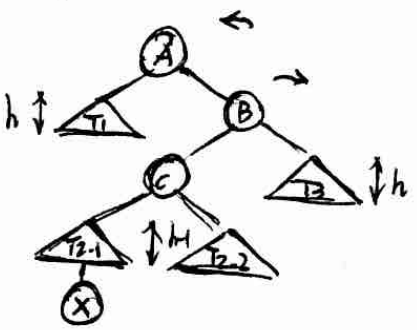


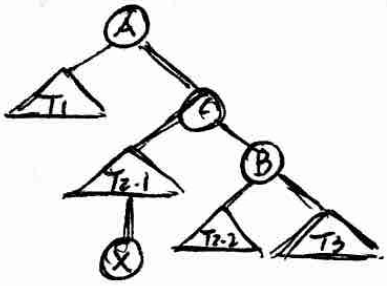
第二次 Rotation

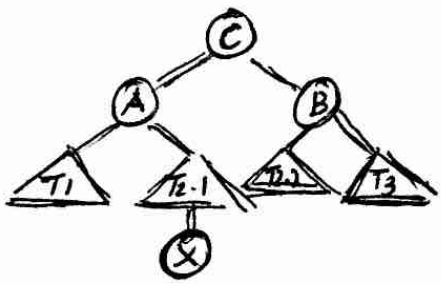
逆时针旋转

- ORDER PRESERVED
- restored height

1.B.β.②





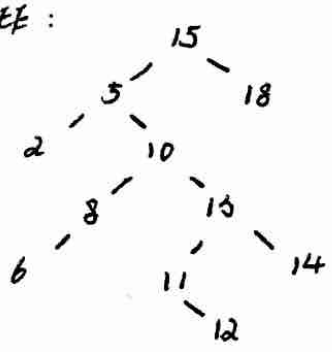


ARRAY: $T_1, A, T_{2.1}, C, T_{2.2}, B, T_3$

Tue Jan 24 2017

Delete Node:

假设 BINARY TREE:



CASES:

- 0 children: just delete it.
- 1 children: replace with child then delete it.
- 2 children: find successor
  - find x's successor z
  - replace x's content with z's contents
  - delete z

树状结构:

successor(x)

z = right_child(x)

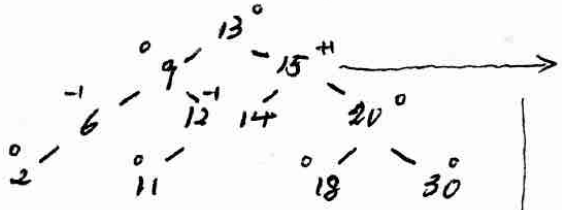
while left_child(z)

z = left_child(z)

return z.

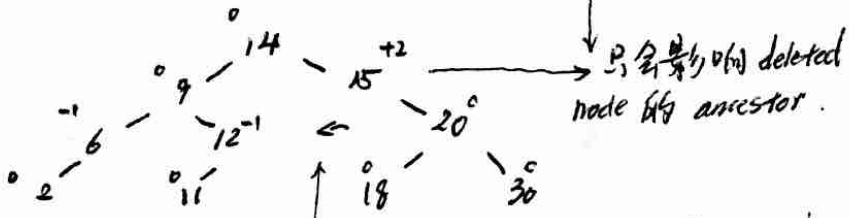
例子: 假设 BINARY TREE (AVL TREE) T.

DELETE(T, 13)



Find Successor => 14

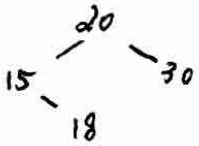
replace 13 with 14. delete 14.



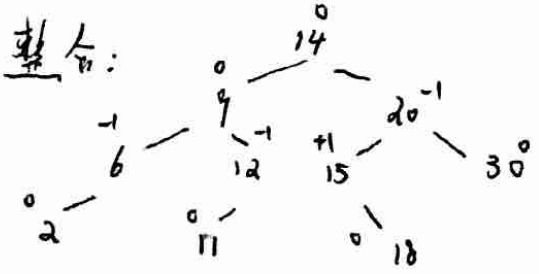
只会影响到 deleted node 的 ancestor.

LEFT ROTATION

=> 进行 rebalance:

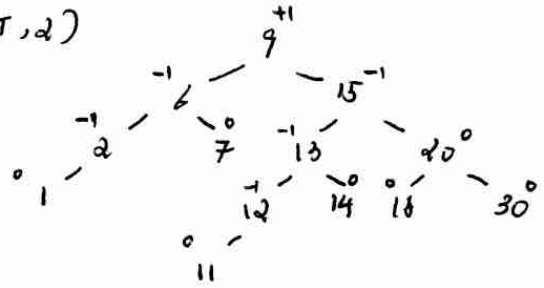


最后整合:



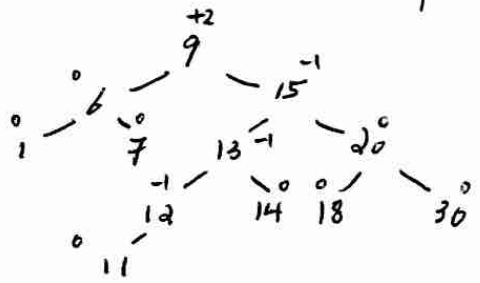
例子：假设 AVL TREE T.

DELETE (T, 2)



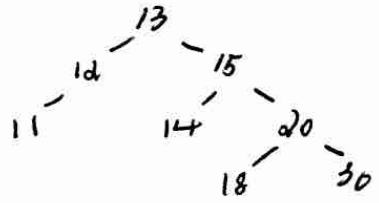
2 原子 1 children 的情况：replace with child, then delete it.

⇒

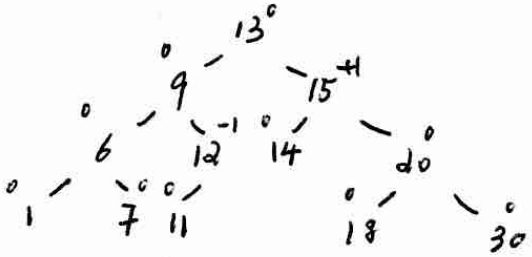


- if a node is unbalanced and right-heavy and its right child is left heavy, then we need to do a double rotation.

第一个 ROTATION：



第二个 ROTATION：



Jan 30 2017

AUGMENTING DATA STRUCTURES.

- It is often useful to modify a standard data structure to perform additional operations.
- To do so, we often need to store and efficiently maintain additional information.

Example Today: Dynamic order statistics CRLS 14.

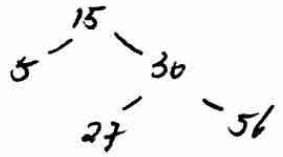
Want to maintain set S of elements with KEYS. and do the following operations efficiently:

- Insert, delete, search ["dictionary"]
- insert(k): find element of rank k.
- rank(k): given a pointer to element k, find the rank of k.

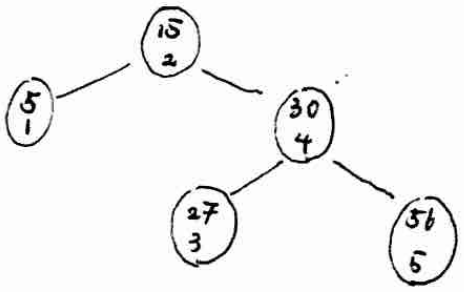
↓ in sorted order (position in sorted order)

例子: 假设 S = {5, 15, 27, 30, 56}.

Select(4) = 30
Rank(15) = 2
Rank(30) = 4



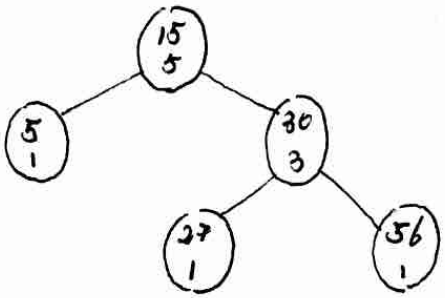
⇒ Add a field in the node



But updating the rank is too much work.

At Each node x, keep size(x) ≡ number of nodes in subtree rooted at x.

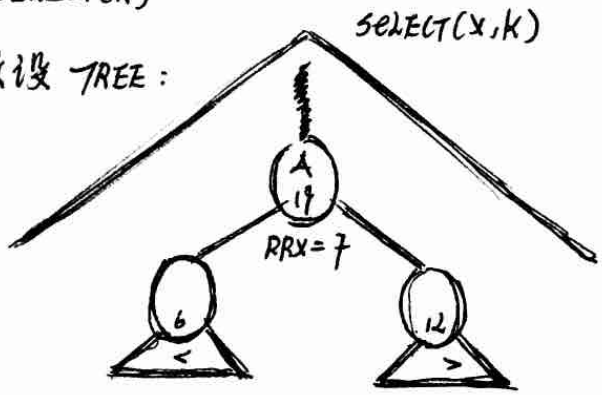
x - subtree



SIZE(x) = SIZE(LEFT(x)) + SIZE(RIGHT(x)) + 1

SELECT(k)

假设 TREE:



RELATIVE RANKING OF x-SUBTREE

RRX = SIZE(LEFT(x)) + 1 = 6 + 1 = 7

① SELECT(x, k)

RRX ← SIZE(LEFT(x)) + 1

IF k = RRX Then return x.

If k < RRX -> select(x, 3)

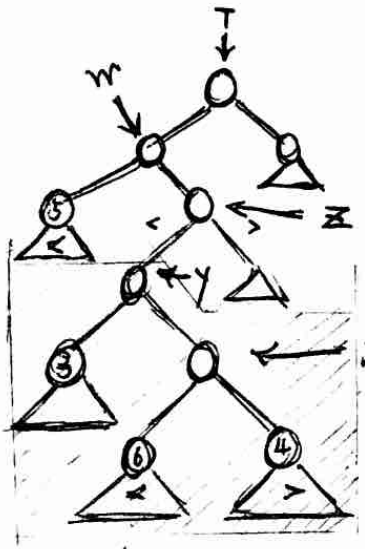
then select(left(x), k)

② SELECT(x, 11)

If k > RRX. SELECT(RIGHT(x), k-RRX)

O(log(n))

假设 Tree T:



X (RRX = SIZE(LF(x)) + 1)

6 + 1 = 7

∴ RRX ≡ 7

↑ 在 x 的 subtree 下，随意的每个都小于 8

∴ going up from the left, rank of x doesn't change.

RRX in w-subtree, going up from the right. → rank(x) + 6.

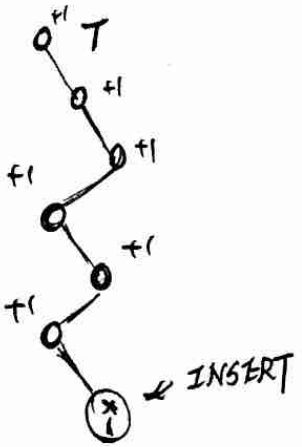
If k = rrx then return x

If k < rrx then select (left(x), k)

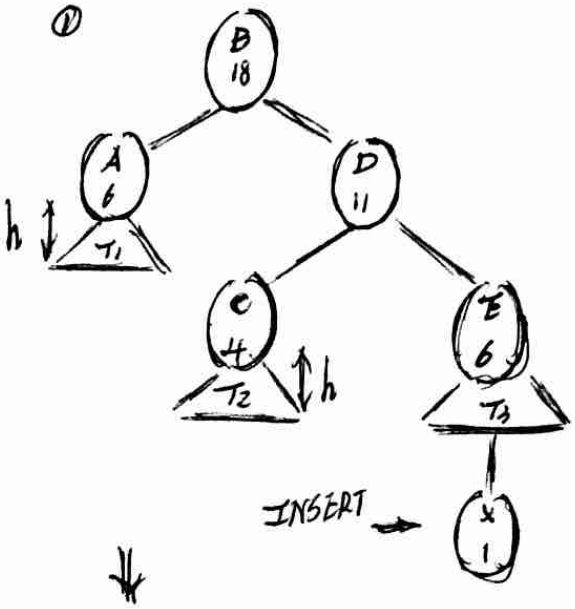
If k > rrx then select (right(x), k - rrx)

Maintain Size(x) Fields.

- INSERT(T, x)

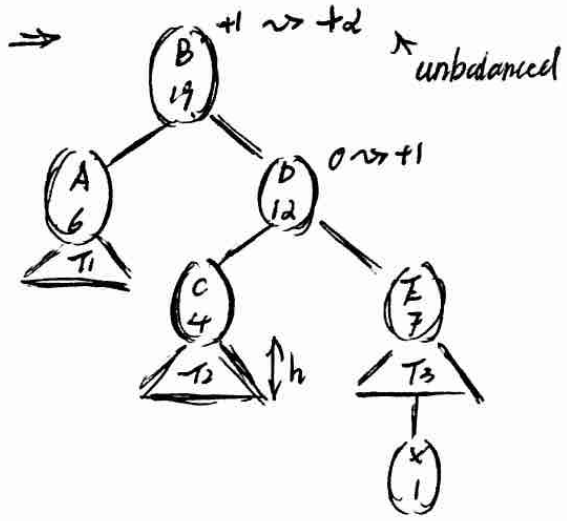


①

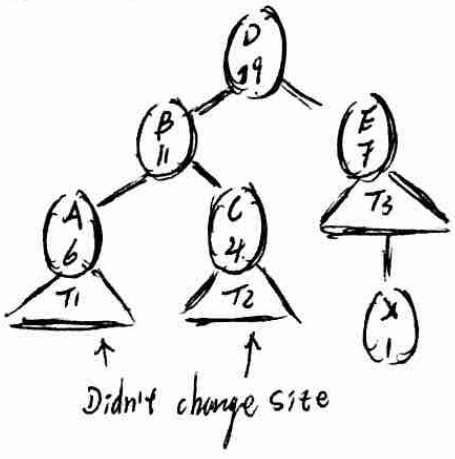


从右面insert，所以B的ranking有改动

②



ROTATION:



BLOOM FILTER 布隆过滤算法.

一种多哈希函数映射的快速查找算法，应用于快速判断某个元素是否属于合集，但是并不严格要求100%正确的场合。

若要完全明白 BLOOM FILTER，
要先弄明白 HASHING —— 哈希函数.

HASHING 散列法 / 哈希法

key 对应 value.

作用：知道一个东西是否存在.

操作：通过哈希函数，给的相关ID编号.

课堂笔记：

假设 set S of M elements with keys taken from universe $U = \{0, 1, 2, \ldots, u\}$

操作 OPERATIONS: INSERT. DELETE, SEARCH.

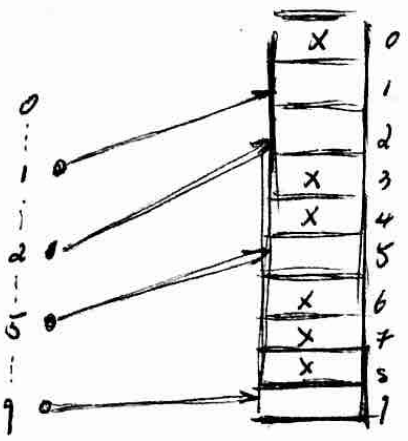
① |u| is small:

- Store S in a direct access table:
  T[0, 1, 2, ..., u]
- Store element with key k in T[k]

例子：$u = \{0, 1, 2, \ldots, 9\}$

T[0...9]

$S = \{2, 5, 9\}$



② |u| is very large

例子：keys is any 64-bit integers

⇒使用 Hash Table 储存数.

n = # keys stored in dictionary

$n << |u|$

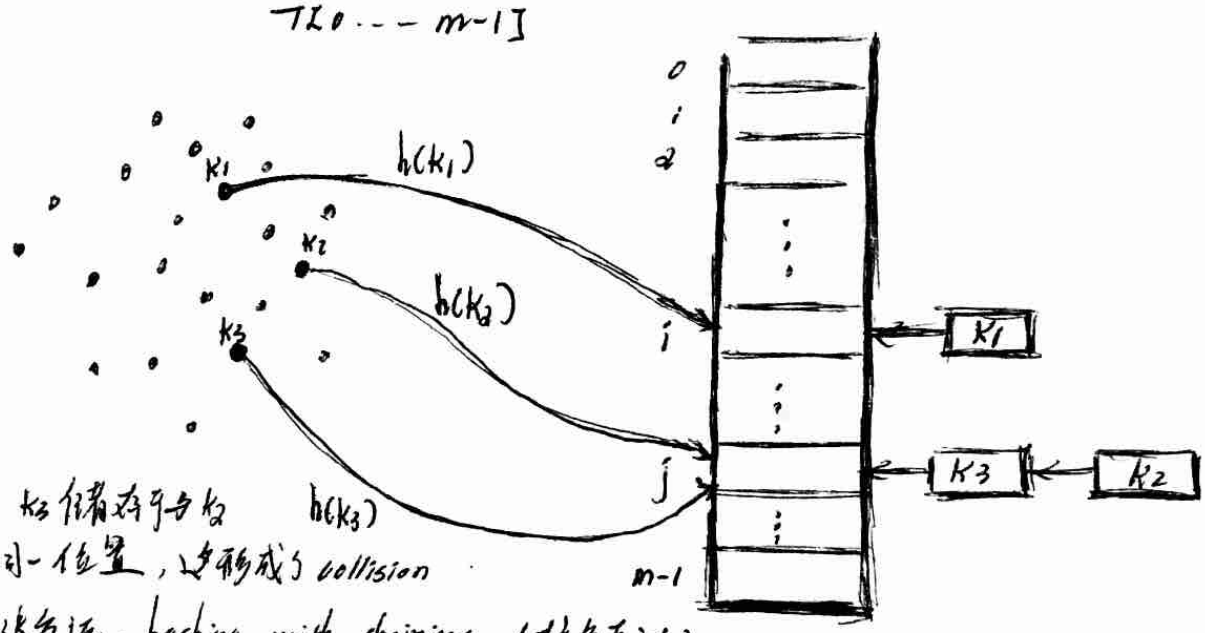
m = size of hash table ($m = \Theta(n)$)

Hash Func: T[0 ··· m-1]

$h: u \rightsquigarrow \{0, 1, \cdots, m-1\}$

$k \overset{h}{\rightsquigarrow} h(k) = i$ ⇒ 通过 hash function 映射特定编号.

如图：（连锁拉链法）



解决方法：hashing with chaining（拉链法）

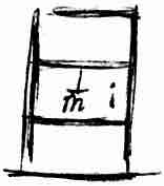
- Search(k) after entering n keys in T:

  search for k in the list that store at T[h(k)]

SUHA: Simple Uniform Hashing Assumption

- Assume that any key k is equally likely to hash into any of the m slots of T independently of where other keys hash to.

INSERT
DELETE ⇒ constant time.

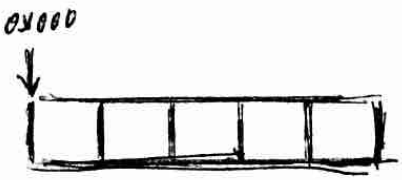
prob. 

$PROB[h(k) = i] = \frac{1}{m}$ for each i $0 \le i \le m-1$
(Probability of storing in one slot)

After inserting n keys into empty T, the expected length of the chain at slot i is $\alpha = \frac{n}{m}$ ← utilized of table
← table size

[LOAD FACTOR OF TABLE]

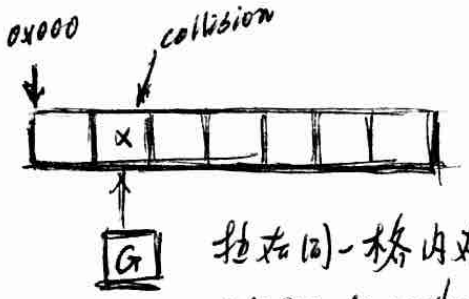
总结：散列哈希法，通过散列算法，变换成固定长度的输出，该输出值就是散列值。这种转换是一种压缩映射，也就是，散列值的空间通常远小于输入的空间，不同的输入可能会散列成相同的输出，而不可能从散列值来确定唯一的输入值。简单的说就是一种将任意长度的消息压缩到某一固定长度的消息摘要的函数。



做法 function
h_1(value): offset.

1 放 0x004
open addressing hash table
把页帧原封不动复制到 memory.
1 对应 0x004, 2 对应 0x008 --
风险：有可能用到不该用的内存，因为 open addressing 没有 limit.

解决方法：局限内存范围.



指右侧一格内存上.
pointer to another data structure
常规来说是 Linked-List.

假如 n 个指右侧一格内存上，找某一个东西 $\Rightarrow O(n)$
假如指针增多，其性能就会下降.

方法，类似 AVL TREE，将 node 平均分布以部优化的二叉树。

n items
m slots $\Rightarrow$ n 个东西平均放到 m 个 slots 上。

根据 SUHA: Simple Uniform Hash Assumption.
$\Rightarrow$ Uniform Distribution.

$E(x) = \sum_i prob(x) \cdot x \Rightarrow \frac{1}{m}$

$O(\frac{n}{m})$ 去找一个东西.

Bloom Filter 布隆过滤算法

一个具有良好的空间和时间效率的二进制向量数据结构.
用来检测一个元素是否为集合中的一个成员.

课堂笔记:

"Probabilistic Dictionary" BF.

- Maintain "finger prints" of a set S of elements.

Operations: INSERT(BF, x): $S \leftarrow S \cup \{x\}$

[enters finger print of x in BF]

SEARCH(BF, x) → "NO" → $x \notin S$

→ "YES" → probably $x \in S$

(FALSE POSITIVE)

FALSE POSITIVE: 假阳性; 误报

FALSE NEGATIVE: 假阴性; 漏报

SEARCH(x) ANSWER	$x \in S$	$x \notin S$
"YES"	100%	1% (Probability of False Probability)
"NO"	0%	99%

+ and − :

+ Space Efficient
+ Fast
+ No False Negative
− False Positive Possible.

BLOOM FILTER IMPLEMENTATION:

Store "Finger Prints" in A.

① ARRAY B[0, ..., m-1] of m bits.
(Initially all 0's)

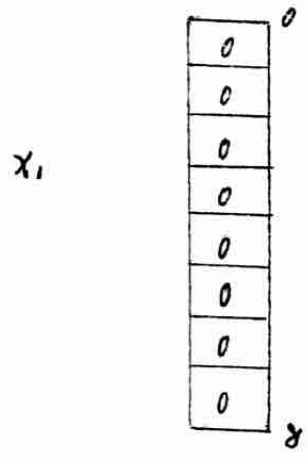
② t independent hash func:

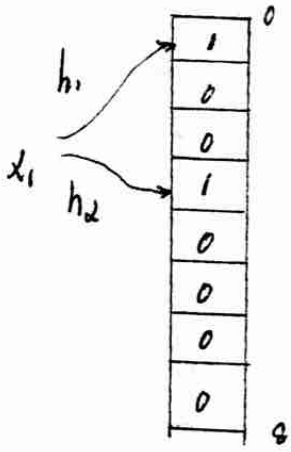
$h_1(), h_2(), \ldots, h_t()$

$h_i : U \to \{0, 1, \ldots, m-1\}$

h_i satisfies SUHA: $PROB[h_i(x)] = \frac{1}{m}$

Example: BF[0, ..., 8] $t = 2$ $h_1(), h_2()$

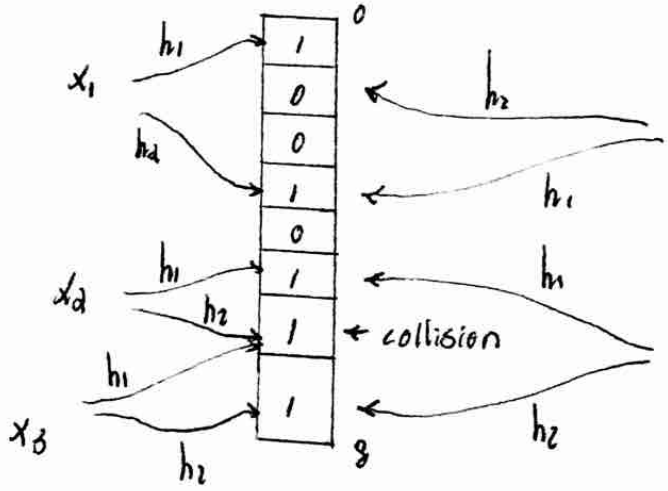




• ENTER $x_1, x_2, \ldots, x_m$ into a BF[0, ..., m-1] using t independent hash functions $h_1, h_2, \ldots, h_t$ that satisfy SUHA.

• SEARCH FOR $x \neq x_1, x_2, \ldots, x_m$

• PROB[FALSE POSITIVE] = PROB[SEARCH(x) = "Yes"]

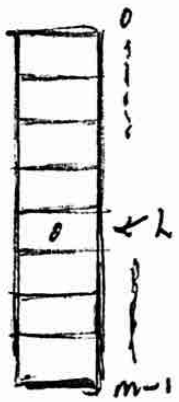


x SEARCH(x) = "No"
hash x with x_1

x' SEARCH(x') = "Yes"
Matches
$x' \neq x_1, x_2, x_3$
⇒ False Positive 误报.

Enter $x_1, x_2, \ldots, x_m$ into a $BF[0, \ldots m-1]$.
using t independent hash functions. $h_1, h_2, \ldots, h_t$ that satisfy SUHA.

分析概率.



第一次:

保持为'0'的概率: $(1-\frac{1}{m})$

变为'1'的概率: $\frac{1}{m}$ ↑ m个slot

假设 L, $0 \le L \le m-1$

- PROB[AFTER INSERTING x_1, x_2, x_n, $BF[L]=0$]

$= (1-\frac{1}{m})^{nt}$ ← t个hash.

↑ n elements

$\approx e^{-\frac{nt}{m}}$ $(1-x) = e^{-x}$.

- PROB[AFTER $x_1, x_2, \ldots, x_n$. $BF[L]=1$]

$= 1-((1-\frac{1}{m})^{nt})$

$= 1- e^{-\frac{nt}{m}}$

$y \neq x_1, x_2, \ldots, x_n$

PROB[FALSE POSITIVE] = PROB[SEARCH(y) = "YES"]

$=$ PROB[$BF(h_1(y))=1 \wedge BF(h_2(y))=1 \wedge \cdots \wedge BF(h_t(y))=1$]

$= \prod_{i=1}^{t}$ PROB[$BF(h_i(y))=1$]

$= (1-e^{-\frac{nt}{m}})^t$

In this example.

$\frac{m}{n} = \frac{\#\text{ BITS IN TABLE}}{\#\text{ of fingerprints}} \approx 8$ BITS.

$t = (\ln 2)\frac{m}{n}$

PROB[FALSE POS] $\approx 0.62^{\frac{m}{n}}$

$\frac{m}{n} = 8 \rightarrow t \approx 5.45$.

概率.

Outcome : describes fully the result of an experiment.

Example : roll a dice → 1, 2, 3, 4, 5, 6.

Sample Space S : set of all outcomes of an experiment.

Example : roll a dice → {1, 2, 3, 4, 5, 6}.

toss two coins → {HH, HT, TH, TT}.

EVENT A : one or more possible outcomes a subset of the sample space.

probability distribution : a mapping from every outcome in S to a real number such that.

1) $\sum_{s \in S} P(s) = 1$

2) $P(s) \geq 0 \quad \forall s \in S$.

Probability of an event $P(A)$

$P(A) = \sum_{s \in A} P(s)$ Ex $P(\text{even}) = P(2) + P(4) + P(6) = \frac{1}{2}$.

Random Variable X.

a function from the sample space to the real numbers.

$X(1) = 12$ $X(3) = 8$ $X(5) = 8$

$X(2) = 12$ $X(4) = 8$ $X(6) = 18$.

Expected Value. $E(X) = \sum_{o \in S} X(o) P(o)$

$$E(X) = \sum_{x \in X} x P(X = x)$$

$$= 12 P(X=12) + 8 P(X=8) + 18 P(X=18)$$

$$= 12\left(\frac{1}{3}\right) + 8\left(\frac{1}{2}\right) + 18\left(\frac{1}{6}\right) = 11.$$

$E[X+Y] = E[X] + E[Y]$

Expected Running Time:

Let A be an algorithm

S_n = the sample space of all inputs of size n.

$t_n(x)$ = the number of steps A takes an input x.

$$E[t_n(x)] = \sum_{x \in S_n} t_n(x) P(x)$$

Example: a linked list with 8 elements with distinct keys

what is the expected time complexity of SEARCH (k, L)?

Assumptions:

1) the input k is random and has the probability distribution.

For $1 \le i \le 8$ $P(k$ is the ith element of the list$) = \frac{1}{16}$

$P(k$ is not in the list$) = \frac{1}{2}$.

2) the time to execute Search (k, L) is $2i$ if k is the ith element of the list, 17 if k is not in the list.

=> 转换条件可得: $S_n = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$

$$T_n(j) = 2i \quad 1 \le j \le 8$$

$$T_n(j) = 17 \quad j = 9$$

$$E[T_n] = \sum_{j=1}^{8} \frac{1}{16} 2j + 17 \cdot \left(\frac{1}{2}\right)$$

$$= \frac{2}{16} \sum_{j=1}^{8} j + \frac{17}{2}$$

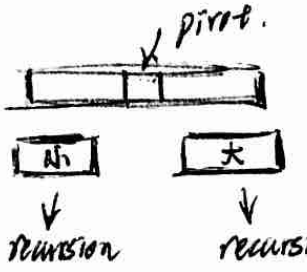
求和公式: $\frac{1}{2} n(a_1 + a_n)$.

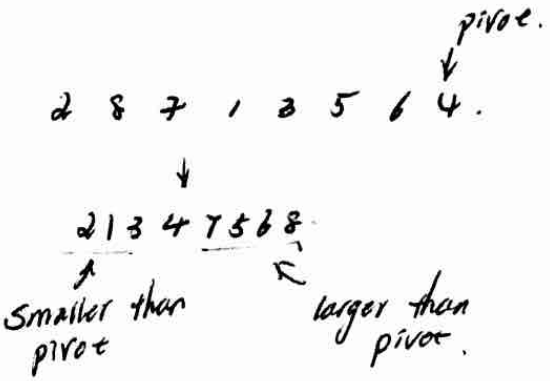
$$= \frac{2}{16} \left(\frac{(8)(9)}{2} + \frac{17}{2}\right)$$

$$= \frac{26}{2}$$

$$= 13.$$

Quick Sort. 快速排序.





Recursively partition the sub-arrays before and after the pivot.

Worst-case Analysis of Quick Sort.

$T(n)$: the total number of comparisons made.

Claim 1. Each element in A can be chosen as pivot at most once.

- A pivot never goes into a sub-array on which a recursive call is made.

Claim 2. Elements are only compared to pivots.

Claim 3. Any pair (a,b) in A, they are compared with each other at most once.

∴ the total number of comparisons is no more than the total number of pairs.

$$T(n) \leq \binom{n}{2} = \frac{n(n-1)}{2} \qquad T(n) \in O(n^2)$$

The worst input: is an already sorted array (picking last as pivot)

Choose pivot $A[n]$, then $n-1$ comparisons

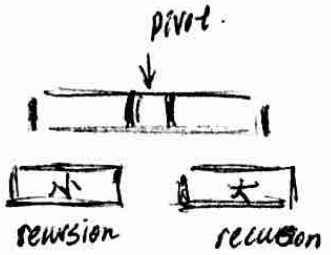
Recurse to subarray, pivot $A[n-1]$, then $n-2$ comparison.

Recursive to subarray, pivot $A[n-2]$, then $n-3$ comparison.

∴ total $(n-1)+(n-2)+\cdots+1 = \frac{n(n-1)}{2}$.

$$T(n) \geq \frac{n(n-1)}{2} \qquad T(n) \in \Omega(n^2)$$

$$\therefore T(n) \in \theta(n^2)$$



i, j 如果分别在 pivot 两边，i j 不会进行比较。

只有 i 和 j 其中一个选为 pivot 才会进行比较。

概率 $\frac{1}{j-i+1} \times 2$ (i & j)

Worst Case，每两个都要相互比较一次 (n^2)

从小到大选 pivot.

$T_{ij} = \begin{cases} 1 & \text{会比较} \\ 0 & \text{不会比较} \end{cases}$ ⇒ indicater variable.

So the total number of compurisons:

$$T = \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij} \quad \text{(sum over all possible pairs)}$$

$$E[X] = E\left[\sum_{i=1}^{n} \sum_{j=i+1}^{n} T_{ij}\right]$$

$$= \sum_{i=1}^{n} \sum_{j=i+1}^{n} E[T_{ij}] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \text{Prob (i and j are compared)}$$

i 和 j 比较的概率.

Claim 只有第一个选中的 pivot 是 i 或 j 才会进行比较.

Pr (i and j are compared)

= Prob (i or j is the first among Z_{ij} chosen as pivot)

$= \frac{2}{j-i+1}$ 一共有 $j-i+1$ 个东西在 Z_{ij} 里.

每一个都有相同概率被选作第一个 pivot.

$$= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \in O(n \log n)$$

RQS(S) randomized Quick Sort.

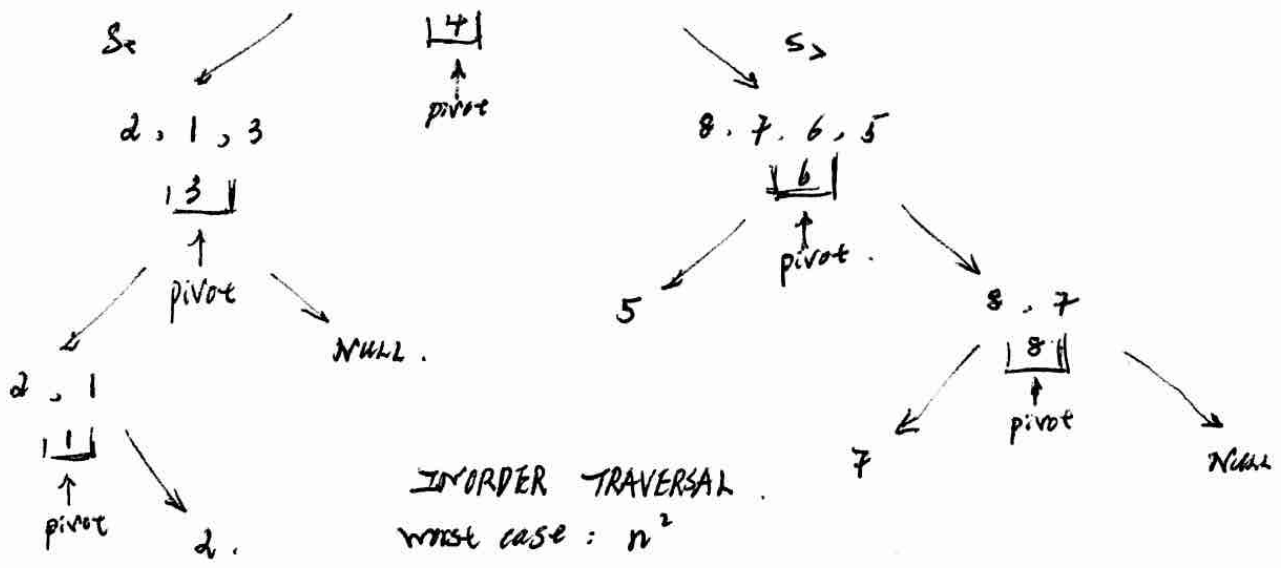
- If S is empty then return
- If $|S|=1$ then output key of S.
- If $|S|>1$ then
  - ① select a pivot p in S uniformly at random.
  - ② by comparing p to every other elements (key) of S.
    split S into;

$$S_< = \{s \in S \mid s<p\}$$

$$S_> = \{s \in S \mid s>p\}$$

  - ③ RQS($S_<$); output p; RQS($S_>$)
    return → sorted.

例子: S: 2, 8, 7, 1, 3, 4, 6, 5 $n=8$.



INORDER TRAVERSAL.

worst case: n^2

- LET $z_1 < z_2 < \cdots < z_i < \cdots < z_j < \cdots < z_n$.

- two keys are compared if only if one of them is pivot, the other keys either go left or right. (pivot remaind at the same level).

- let $C_{ij} = \begin{cases} 1 & \text{if RQS(s) compares } z_i \text{ and } z_j \\ 0 & \text{otherwise} \end{cases}$ → indicating variable.

$C = \sum_{i<j} C_{ij}$ # of comparison.

$C = \sum_{1 \le i \le j \le n} C_{ij}$ Total # of comparison done by RQS.

$E(C) = E\left(\sum_{i<j} C_{ij}\right) = \sum_{i<j} E(C_{ij})$

Expected Value

$E[C_{ij}] = 1 \cdot \text{PROB}[C_{ij}==1] + 0 \cdot \text{prob}[C_{ij}==0]$.

$= \text{PROB}[z_i \text{ and } z_j \text{ are compared}]$

claim: $\text{prob}[z_i \text{ and } z_j] = \frac{2}{j-i+1}$

Look at endpoints: z_1 and z_j

Say $i=1$ $j=n$ $\text{prob}[z_i \text{ and } z_j] = \frac{2}{n}$ (smallest and largest compared)

Consider the set $z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$

$|z_{ij}| = j-i+1$

- initially z_{ij} is entirely contained in S.
- RQS(s) keeps selecting pivots and uses them to split S into smaller sets until it gets sets of size 0 or 1.
- As long as RQS(s) selects pivots that are not in z_{ij}, then z_{ij} remains entirely contain in one of the subset and z_i and z_j remain uncompared.

- At some point RQS(s) must select a pivot $p \in z_{ij}$ 2 cases.

① $z_i < p < z_j$ (between them) ⇒ z_i and z_j are not compared.

② $p = z_i$ or $p = z_j$ ⇒ z_i and z_j are compared.

$\text{Prob}[z_i \text{ and } z_j \text{ are compared}] = \text{Prob}[p=z_i \text{ or } p=z_i \mid p \in z_{ij}]$

$\Rightarrow \frac{2}{j-i+1}$

$$E(c) = \sum_{1 \le i < j \le n} = \frac{2}{j - i + 1}$$

$$\vdots$$

$$= 2n \left[1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}\right].$$

$(n \ge 2)$ $\quad H_n \le \ln(n) + 1$

H_n is $O(\log n)$

$E(c) = O(n \log n)$.

Disjoint Set 并查集

- n elements : $1, 2, 3, \cdots, n$
- initially, each element is in its own set.
  $\{1\}, \{2\}, \cdots, \{n\}$.
- each set has a representation element.

- "S_x" set represented by element x.
- OPs : makeset(x) : creates singleton set $\{x\}$
  Union(S_x, S_y): replace S_x and S_y with $S = S_x \cup S_y$.
  [Return pointer to representative of S]
  Find(z): find the set S that contains z.
  [Given a pointer to element z]

别名: Forest (Tree) disjoint-sets D.S.

$n=5$	S_1	S_2	S_3	S_4	S_5
	$\{1\}$	$\{2\}$	$\{3\}$	$\{4\}$	$\{5\}$
$U(S_3, S_4) \Rightarrow$	$\{1\}$	$\{2\}$	$\{3,4\}$	X	$\{5\}$

$F(4) = S_3 \Rightarrow$ Say, choose the smaller as representitive of set; 3 in this case.

$U(S_1, S_5) \Rightarrow$	$\{1,5\}$	$\{2\}$	$\{3,4\}$	X	X
$U(S_1, S_3) \Rightarrow$	$\{1,5,3,4\}$	$\{2\}$	X	X	X
$F(4) = S_1$					
$F(2) = S_2$					
$U(S_1, S_2) \Rightarrow$	$\{1,5,3,4,2\}$	X	X	X	X

$F(2) = S_1$

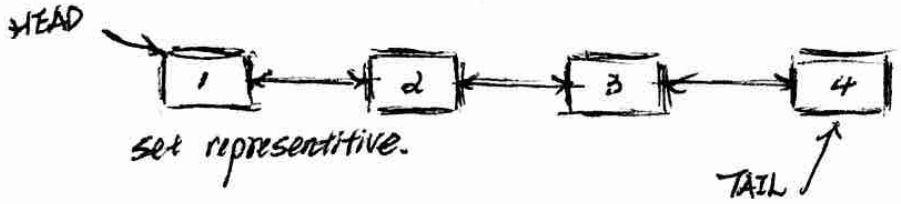
Union takes 2 set to 1.
Maximum union $n-1$

Want a data structure that minimizes the total cost of executing a sequence σ.

$n-1$ unions mixed with $m \geq n$ finds.

Implementation:

① Linked-List: (Doubly Linked-List)



UNION(i, j) $O(1)$ constant time.

MAKE SET(x) $O(1)$ constant time.

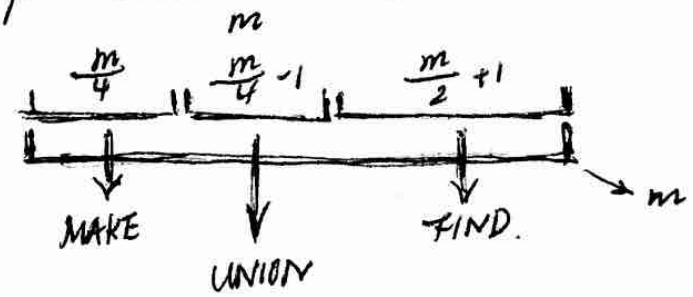
FIND(x) $O(n)$ the length of linked-list.

Total cost of σ : $O(n + m \cdot n)$

↑ union ↑ finds.

Amortized Complexity = $\frac{\text{WCSC for } m \text{ ops}}{m}$

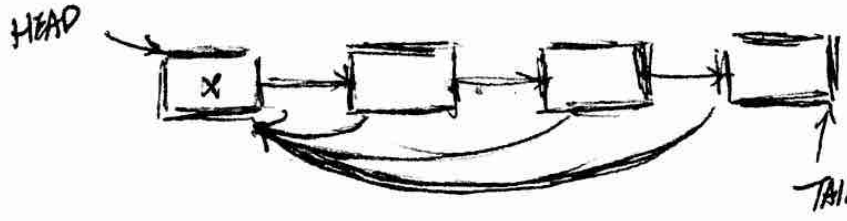
均摊复杂度



$\frac{m}{4}(1) + (\frac{m}{4} - 1)(1) + (\frac{m}{2} + 1)(m) \Rightarrow \Omega(m^2)$

$O(m^2)$ 含有 m 次 FIND.

② Augmented Linked-List (Union by Weight)



Single Linked-List，每个 node 又再指向其 representitive.

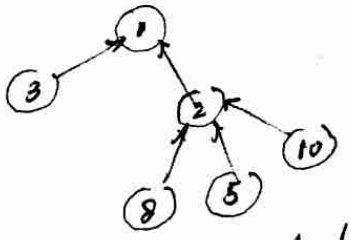
UNION(i, j) $O(n)$ 连完之后，再指向新 representitive.
MAKE SET(x) $O(1)$ constant time.
FIND(x) $O(1)$ constant time. 直接得到.

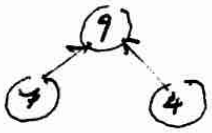
Total cost of b $O(m + n^2)$
↑ FINDS ↑ UNION)

使用 Weighted Union Rule: smallest list at end.
with W.U. rule, cost of b is: $O(m + n\log n)$
↑ FINDS.

Tree (forest) data structure.

$S_1 = \{1, 3, 2, 8, 5, 10\}$ $S_9 = \{9, 7, 4\}$.

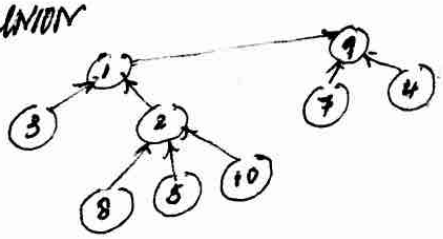




FIND: proportional to the height
cost of FIND: $O(1 + \text{length of find path})$.

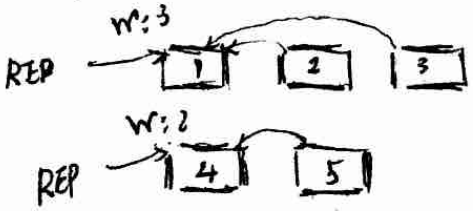
$S_6 = \{6\}$
(6)

UNION



cost of union: $O(1)$
constant time.
But we created a long tree.

By W.U.: 小的连到大的



选小的：

好的 $O(m + m\log n)$.

Minimize length of find paths.

@ BALANCING: W.U. [By. size]

fewer NODE.

root of smaller tree (By. size) becomes child of the root of the bigger tree.

+ the height of these trees never exceed $\log n$.

LEMMA: with W.U., any tree T of height h created during the execution of S contains at least 2^h nodes:

$$|T| \geq 2^h$$

好的: size 7 $\Rightarrow$ at least 2^7 nodes.

Proof by induction: on h.

a) $h=0$, any T of height $h=0$ has $|T|=1$ node.

b) Assume LEMMA holds for h, show it holds for $h+1$.

consider first time a tree T of size $h+1$ is created.

By taking something of h of height, link it to a new root.

Must show $|T| \geq 2^{h+1}$ nodes

By I.H.

$|A| \geq 2^h$ By W.U. Rule.

$|B| \geq |A| \geq 2^h$

$|T| = |A| + |B| \geq 2^h + 2^h = 2^{h+1}$

$2^h < n$ (max # of node)

$h < \log n$

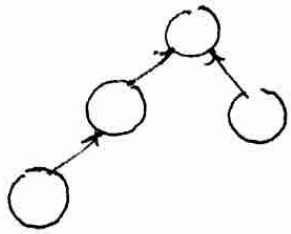
Therefore:

Total cost of executing S is

$O(n + m\log n)$ $m \geq n$

↑ union ↑ max height

③ Path compression



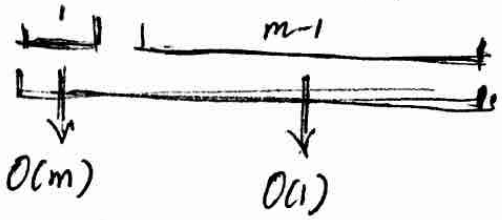
每个 child 是 linked list.

MAKE_SET(x) O(1) constant time.
UNION(i, j) O(1) constant time.

PATH COMPRESSION 的重点为了 FIND SET.

看④的 parent 是不是 rep.
若不是，改指向，一个一个trace上去。
为其使用 FIND_SET.

每进行一次 FIND_SET 就改指针. 2 O(n)
运行越来次数越好.



(1)(m) + (1)(m-1) = m+m-1 => Ω(m).

复杂度总结 + 重点数据结构小细节

MAX/MIN HEAP:

INSERT $O(\log n)$ 加到树的最右边，然后 Heapify.
EXTRACT $O(\log n)$ 最后的放上去，每个都 Heapify.
DELETE $O(\log n)$
SEARCH $O(n)$ 每个都看.
HEAP SORT $O(n \log n)$
BUILD (从 ARRAY) 不是每个都 HEAPIFY $\Rightarrow O(n)$
BUILD (一个个 INSERT) 每个 INSERT 为 $\log n \Rightarrow O(n \log n)$

BINOMIAL HEAP:

INSERT $O(\log n)$ 多少树就有多少个 UNION.
MERGE (UNION(Q_1, Q_2)) $O(\log n)$
EXTRACT_MIN $O(\log n)$ 把每棵树的 root 比较，然后 extract.
MINIMUM $O(\log n)$ $\log n$ 个比较。
MAKE $O(1)$
有 n 个 NODE
最多有 $\lfloor \log n \rfloor + 1$ 个树
$\alpha(n)$: 二进制表达式内 '1' 的数量.
$n - \alpha(n)$: edges 的数量总和.

有 2^k 个 node 就有 k 为 height.
有 k 个 node 至少有 2^k 个 node，最多有 $2^{k+1} - 1$ 个 node.

AVL TREE: 自平衡二叉树

INSERT $O(\log n)$
DELETE $O(\log n)$
SEARCH $O(\log n)$

HASH 哈希算法：

INSERT $O(1)$　　LOAD FACTOR: $\frac{n}{m}$

DELETE $O(1)$　　Average: $\frac{n+m}{2m}$

FIND $O(\frac{n}{m})$

BLOOM FILTER 布隆过滤器

假阳性误报概率：$(0.62)^{\frac{m}{n}}$

哈希算式需要的数量：$(\ln 2)\frac{m}{n}$

DISJOINT SET: 并查集

LINKED-LIST 实现法：

UNION(i,j) $O(1)$

MAKE_SET(x) $O(1)$

FIND-SET(x) $O(n)$

TOTAL COST OF $b = O(n + m \cdot n)$

↑ UNION　↑ FINDS

AUGMENTED LINKED-LIST 实现法：

UNION(i,j) $O(n)$

MAKE_SET(x) $O(1)$

FIND_SET(x) $O(1)$

TOTAL COST OF $b = O(m+n^2)$

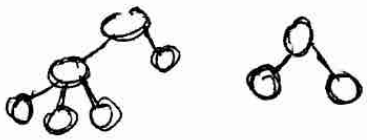
↑ FIND　↑ UNION

TREE (FOREST) 实现法：

UNION(i,j) $O(1)$

MAKE_SET(x) $O(1)$

FIND-SET(x) $O(1+$ length of path$)$



- PATH COMPRESSION 路径压缩法

MAKE_SET(x) $\Theta(1)$

UNION(i, j) $O(1)$

First-time FIND_SET $O(n)$

(as more find runs, runtime gets better).

Amortized Complexity Analysis.

- Given a data structure with OPS.
- $T(n)$ = worst-case (i.e. max) cost of doing a sequence of n OPS (over all possible sequence of n ops)
- $\frac{T(n)}{n}$ = Average cost of an OP in worst-case

  "Amortized cost" of an OP.

Two ways for amortized complexity analysis.

① Aggregate Analysis

② Accounting method

Examples:

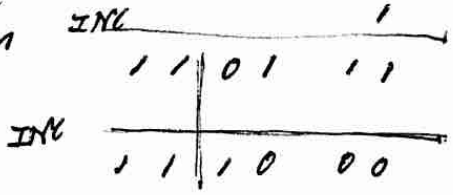
1. Incrementing a binary counter.

   K-Bit counter
   initially $A = 0$
   INC(A)

   cost INC(A) is: # of bits INC(A) flips

[2]

   $T(n)$ = cost of n successive INC(A) starting from $A = 0$.



[1]

[4]

$3 \times [1 \to 0]$: "3 resets"

$1 \times [0 \to 1]$: "1 set".

3	2	1	0	
0	0	0	0	
0	0	0	1	
0	0	1	0	} 1 flips
0	0	1	1	
0	1	0	0	
0	1	0	1	
0	1	1	0	
0	1	1	1	} 4 flips
1	0	0	0	

$T(n)$ = # of bit flips done by n : INC(s)

BIT 0 : Every single, increment flips

$\Rightarrow n$ times

BIT 1 : Every 2, increment flips

$\Rightarrow \lfloor \frac{n}{2} \rfloor$ times

BIT i : Every 2^i increment flips.

$\Rightarrow \lfloor \frac{n}{2^i} \rfloor$ times.

$$T(n) = \sum_{i=0}^{k-1} \lfloor \frac{n}{2^i} \rfloor \leq n \sum_{i=0}^{\infty} (\frac{1}{2})^i$$

$$= n \frac{1}{1-(\frac{1}{2})}$$

$$T(n) \leq 2 \cdot n.$$

$$\Rightarrow \frac{T(n)}{n} \leq 2.$$

Assume cost \$2 per INC

\$2 INC	0 0 0 0	Every one has extra \$1 attach to it
\$2 INC	0 0 0 1 \$1	
\$2 INC	0 0 1 \$1 0	Only need \$2 to complete INC.
	0 0 1 \$1 1 \$1	
	0 1 \$1 0 0	

Amortized Complexity: Example DYNAMIC TABLES.

Table T: | A | B | C | $\alpha(T) = \frac{3}{4}$ ← n elements / m slot

OPERATIONS: INSERT, DELETE.

to INSERT an item x when T is FULL ($\alpha(T) = 1$).

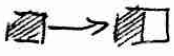
a) Allocate new T: size(new T) = 2 × size(T).

b) copy all items of T into new T.

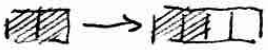
c) INSERT x into new T.

cost of n = 19 INSERTS

$T(19) = 19 + 1$



+2



+4

+8

+16:

$T(19) = 19 + \sum$ (all powers of 2 smaller than 19).

$T(n) = n + \sum$ (all powers of 2 smaller than n)

$$T(n) \leq n + \sum_{i=0}^{\lfloor \log n \rfloor} 2^i = n + \frac{2^{\lfloor \log n \rfloor + 1} - 1}{2 - 1}$$

$$\sum_{i=0}^{k} r^i = \frac{r^{k+1} - 1}{r - 1}$$

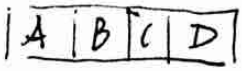
$T(n) \leq n + 2n = 3n$

Amortized: $\frac{T(n)}{n} \leq 3$ (Aggregate Analysis)

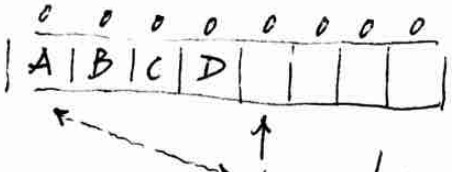
Actually compute by math.

验证以上的算法，最为昂贵的步骤是 COPY.

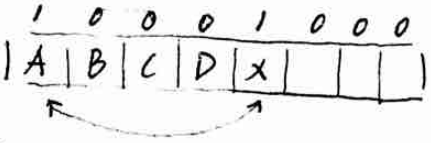
解决方式：overcharge each one a little to save up use later.

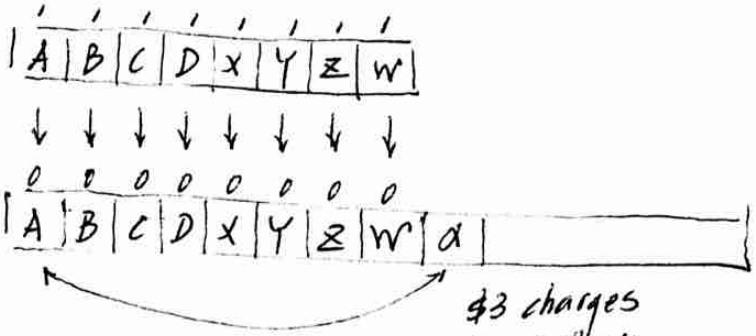


credit: 0

① 

(charge $3) → INSERT X in the slot
→ Next move
→ for your buddy, say A in this case

② credit 

=> 最终可得 

when the array becomes full,
every element in the array has a credit of 1
attaches to it → charges $3 each time.

① If $\alpha(T) = 1$ and INSERT OCCURS

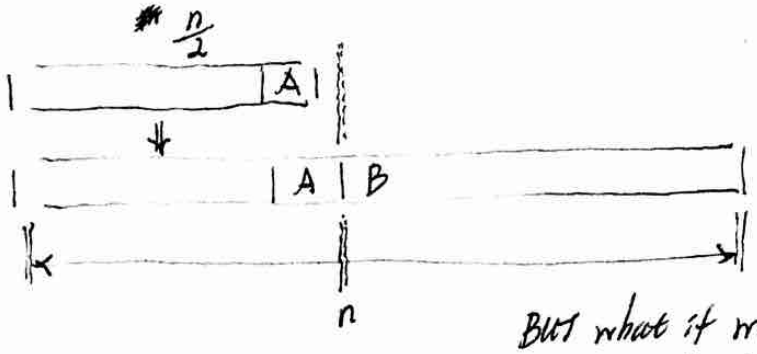
NEW T = 2 × size (T).

② If $\alpha(T) = \frac{1}{2}$ and DELETE OCCURS

NEW $T = \frac{1}{2}$ size (T)

如果直接将内存 free 掉，会形成许多的 fragments

我们想要保障整段完整内存。



2 INSERTS

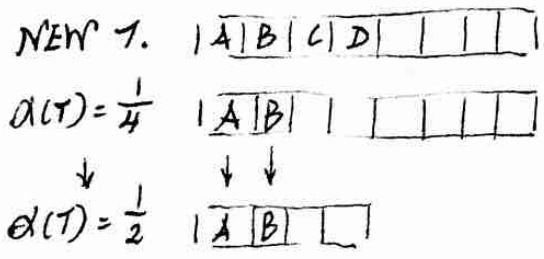
A & B.

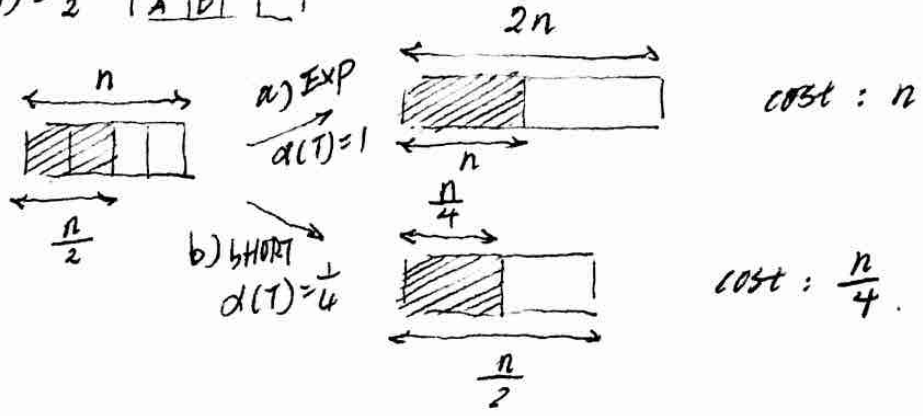
$\alpha(T)$ is more than half.

But what if we wanted to DELETE now?

这种方式太过跳跃 => 不稳定.

解决方法：$\frac{1}{4}$ 使用率才下降 ArraySize.



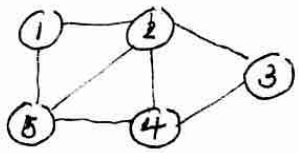


GRAPH 图算法

[CLRS 22, APPENDIX B4].

GRAPH: $G = (V, E)$
- V: set of Nodes $|V| = n$ 节点
- E: set of Edges $|E| = m$ 边.

UNDIRECTED GRAPH

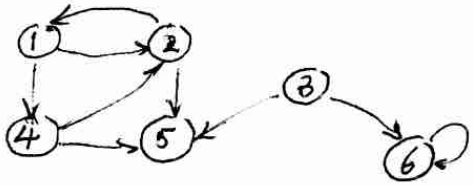


无向图：边代表的关系
没有方向的图.

$E = \{(1,2), (1,5), (2,5), (2,4), (4,5), (3,2), (3,5)\}$

$V = \{1, 2, 3, 4, 5\}$

DIRECTED GRAPH

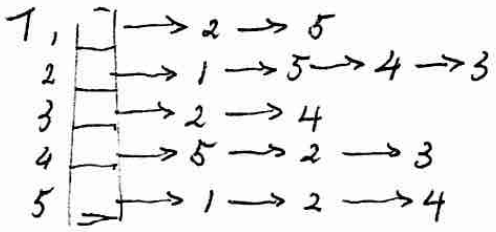


有向图：边代表的关系
有方向的图.

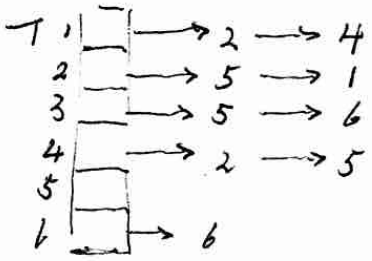
$E = \{(1,2), (1,4), (4,2), (2,5), (2,1), (4,5), (3,5), (3,6), (6,6)\}$

$V = \{1, 2, 3, 4, 5, 6\}$

ADJACENCY LIST 邻接表.



SIZE: $O(n+m)$



ADJACENCY MATRIX 邻接矩阵.

	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

SYMMETRIC

Size:
$O(n \times n)$
$= O(n^2)$

	1	2	3	4	5	6
1						
2					0	
3						
4		1				
5						
6						

NOT-SYMMETRIC.

ADJACENCY LIST 邻接表：A[1, ..., n]

A[u]: List of Nodes v such that $(u,v) \in E$.

ADJACENCY MATRIX 邻接矩阵：nxn matrix A[u,v].

$$A[u,v] \begin{cases} 1 & \text{iff } (u,v) \in E. \\ 0 & \text{iff } (u,v) \notin E. \end{cases}$$

Finding whether a certain (u,v) is in E

- O(1) with ADJACENCY MATRIX (just go check it)
- O(n) with ADJACENCY LIST (trace 链条).

GRAPH SEARCHES 图的搜索算法.

- BREADTH FIRST SEARCH BFS 广度搜索.
- DEPTH FIRST SEARCH DFS 深度搜索.

BFS: breadth-first-search 广度搜索

① Explore S (follow edges coco of S)

② Explore discovered node in order of discovery. (First discover, First explore)

=> PUT in a QUEUE.

分为三种颜色，不同颜色表示 node 的不同状态.

Color of (v): white undiscovered

grey discovered, but unexplored.

black discovered and explored.

记录谁发现了谁

⇒ KEEP TRAIK OF WHO DISCOVERED WHO.

$P(v) = u$ iff node u discovered v.

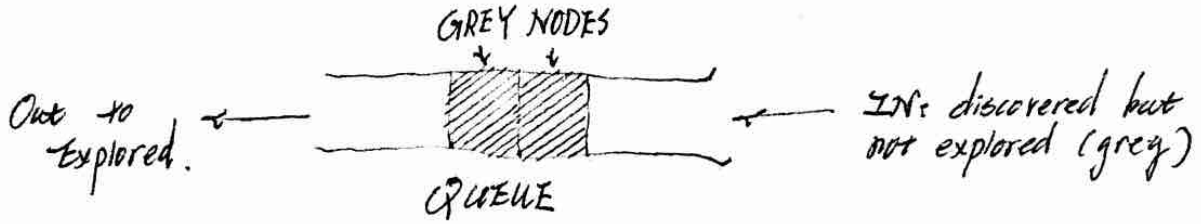
$s \to u_1 \to u_2 \to u_3 \to \cdots \to u \to v$. (DISCOVERY PATH)

$\Rightarrow d(s) = 0 \quad d(v) = d(u) + 1$.

广度查找树：

BFS(s) TREE : graph of all edges of all (u, v) such that u discovered v.

$G = (V, E) \quad s \in V \quad BFS(G, s)$



算法：

for all $v \in V - \{s\}$, color[v] ⇒ white, d[v] ⇒ ∞, P[v] ⇒ NIL

color[s] ⇒ Grey. d[s] ⇒ 0, P[s] ⇒ NIL.

Q ⇒ EMPTY.

ENQ(Q, s)

when $Q \neq \phi$

u ⇒ DEQ(Q)

FOR each edge $(u, v) \in E$

discovery:

if color(v) is white. then

color(v) ⇒ Grey;

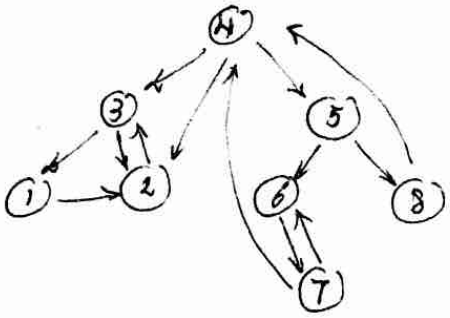
P[v] ⇒ u;

d[v] ⇒ d[u] + 1;

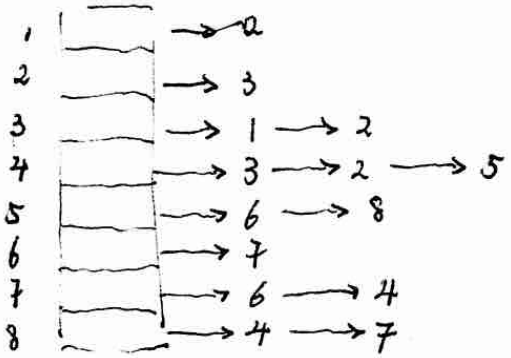
ENQ(Q, v)

colour[u] ⇒ BLACK.

END OF BFS.



ADJACENCY LIST:



从④开始.

|4| d=0

将其关系node放进Q.

|3|2|5|

Q: |2|5|

Explore 3

|3|

将其关系node放进Q.

Q: |2|5|1|

Explore 2

|2|

Q: |5|1|

[3 is Black ∴ DONE]

Explore 5 将其关系node放进Q

|5| → Q: |1|6|8|

Q: |1| ~~2 is~~

Explore 1

|1| Q: |6|8|

[2 is Black ∴ DONE]

Explore 6

|6| Q: |8|

将其关系node放进Q

Explore 8 Q: |8|7|

|8|

Q: |7|

4 is grey already.

7 is grey already.

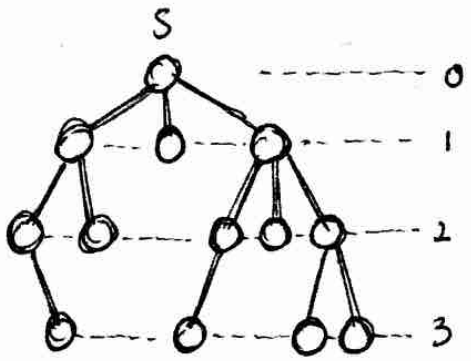
Explore 7

|7| → 6 is grey, 4 is grey, Q清空 → 6 is DONE, 4 is DONE

↓

DONE

BFS(s)

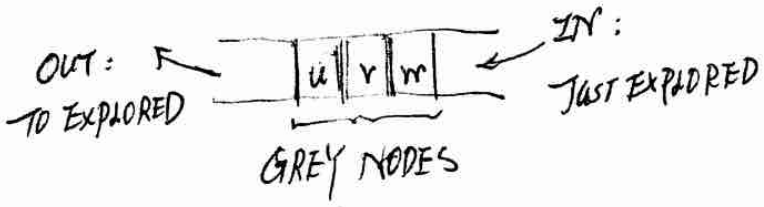


回顾：

COLOUR[v] = W ⟺ NOT DISCOVERED

= G ⟺ DISCOVERED BUT NOT EXPLORED

= B ⟺ DISCOVERED AND EXPLORED.



(A) v DISCOVERY PATH FROM s.

$s \to v_1 \to v_2 \to \cdots \to u \to v$.

$d[v]$ $\qquad$ $d[v] = d[u]+1$.

(B) v's shortest PATH from s.

$s \to v_1 \to v_2 \to \cdots \to v$.

$\to \delta(s,v) \Rightarrow$ By definition.

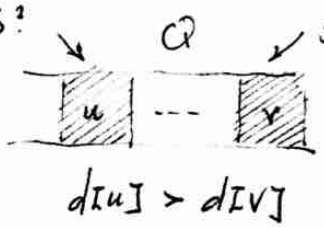
L0: After doing BFS(s): $\forall v \in V$: $\delta(s,v) \le d[v]$ (归纳)

L1: If u enters Q before v enters Q, then $d[u] \le d[v]$.

$d[u] \le d[v]$

定理 21 : If u ENTERS Q BEFORE v ENTERS Q then $d[u] \le d[v]$.

PROOF : Suppose, for contradiction, that 21 is false.
Let v be the first node that enters Q such that $d[u] > d[v]$ for some u that enters Q before v enters Q.



$v \neq s$ because s is first in Q.

$u \neq s \Rightarrow$ $d[u]$ is 0
$d[v]$ would be negative $\Rightarrow$ NO SUCH THING.
(because $d[u] = d[s] = 0$ and $d[v] \ge 0$)

$\Rightarrow$ Some u' and v' discovered u and v.

$$d[u] = d[u'] + 1$$
$$d[v] = d[v'] + 1.$$

$u' = v' \Rightarrow d[u] = d[v]$ contradicts $d[u] > d[v]$.

$\therefore u' \neq v' \Rightarrow$ u' is explored before v' is explored.
$\Rightarrow$ u' entered Q before v' entered Q.

$d[u'] \le d[v']$ because v' enters Q before v.

$\Rightarrow (d[u'] + 1) \le (d[v'] + 1)$

$\Rightarrow d[u] \le d[v]$

THEOREM: After doing BFS(s): $\forall v \in V$, $\delta(s,v) = d[v]$.

证明(反证法):

Suppose for contradiction, $\exists x \in V$ s.t. $d(s,x) \neq d[x]$.

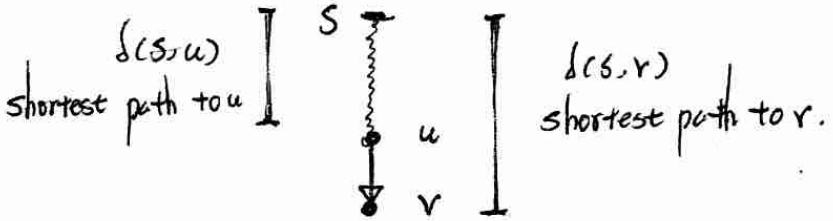
Let v be a closest node from s such that $\delta(s,v) \neq d[v]$.

By 20: $d[v] \geq \delta(s,v)$ (a)

consider a shortest path from s to v.

Let (u,v) be last edge on this path

$\delta(s,v) = \delta(s,u) + 1$ (b)



since u is closer to s than v.

by our definition of v:

$\delta(s,u) = d[u]$ (c)

$$d[v] \geq \delta(s,v) = \delta(s,u) + 1 = d[u] + 1$$

(a) (b) (c)

$d[v] \geq d[u] + 1$ (*).

Look at the moment we are going to explore u.

Consider the colour (v) just before u is explored (在 u 被 explore 之前)

① v is white $\Rightarrow$ u discovers v $d[v] = d[u] + 1$

$\Rightarrow$ contradicts (*)

② v is black (v is already explored)

$\Rightarrow$ v entered Q before u entered Q.

$d[v] \leq d[u]$ $\Rightarrow$ By Lemma① 2.1

$\Rightarrow$ contradicts (*)

③ v is grey

$\Rightarrow$ u hasn't been explored yet.

$\Rightarrow$ so some w discovered v before u's explore.

w enters Q before u enters Q.

By 2.1 $d[w] \leq d[u]$ $d[v] = d[w] + 1$

$\Rightarrow d[w] + 1 \leq d[u] + 1$

$\Rightarrow d[v] \leq d[u] + 1$

$\Rightarrow$ contradicts (*)

Depth First Search DFS

深度搜寻.

DFS: Last-Discovered First-Explored ⇒ Stack.

跟BFS相似，DFS仍保留以下特性：

- colour[v] = white, grey, black.
- P[v] = u IFF "u discovered v".

与BFS不同的是：

- d[u]: "TIME" when u was discovered 发现时间.
- f[u]: "TIME" when u finish exploration 终止时间.

To keep time: use global variable.

"TIME": a counter that is increased each time a node is discovered, or a node's exploration completes.

算法： $G = (V, E)$

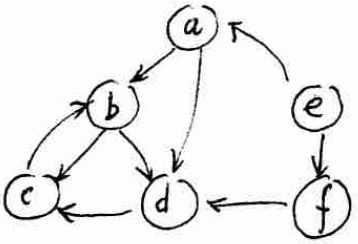
- For each $v \in V$:
  colour[v] ⇒ w, d[v] ⇒ ∞, f[v] ⇒ ∞, p[v] ⇒ NIL
- TIME ⇒ 0
- For each $v \in V$:
  if colour[v] = w Then DFS-EXPLORE(G, v).

END DFS.

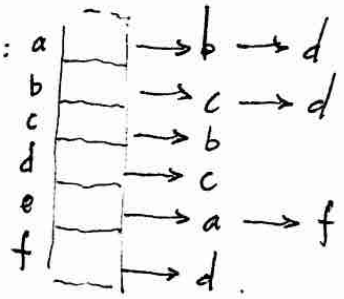
```
DFS_EXPLORE (G, u):
    COLOUR[u] => GREY   // u just discovered
    TIME => TIME+1
    d[u] => TIME   // u's discovery time.

    FOR EACH (u,v) ∈ E
        IF COLOUR[v] = WHITE THEN // u discovered v.
            P[v] => u
            DFS_EXPLORE (G, v)  // explore v.
    COLOUR[u] => BLACK  // done exploring.
    TIME => TIME+1
    f[u] => TIME
    RETURN
    END DES_EXPLORE
```

实际演示



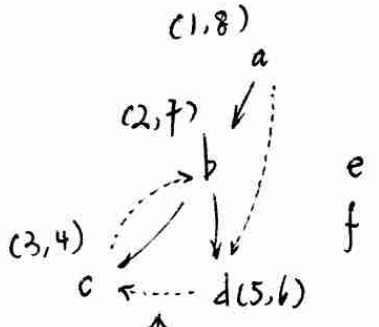
邻接表：



初始化：

a	b	c	d	e	f
W	W	W	W	W	W

全白.



已经 discover 过 c 了

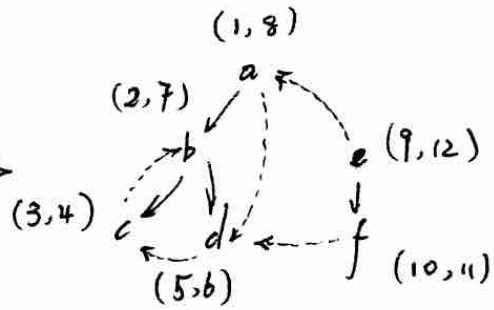
∴ attept discovery

=> done exploring

⇒

a	b	c	d	e	f
B	B	B	B	W	W

e 和 f 还为白色 → 继续



⇒

a	b	c	d	e	f
B	B	B	B	B	B

DFS 更多特性：

CLAIM ①：u is ancestor of v iff

$d[u] < d[v] < f[v] < f[u]$.

$O(m+n)$.

CLAIM ②：For any two nodes u and v,

you cannot have:

$d[u] < d[v] < f[u] < f[v]$.

- No over-lapping intervals.
- Only disjoint or nested.

CLAIM ③：If $u \longrightarrow v \in E$: $d[v] < f[u]$

FROM ancestor to descendant : FORWARD Edge.

FROM descendant to ancestor : BACK Edge.

ELSE : CROSS Edge.

准确定义：A DFS of a directed graph $G=(V, E)$ classifies its edges as follows

$(u, v) \in E$ is

① Tree Edge iff u is parent of v non-tree edges.

② Forward Edge u → ancestor of v.

③ Back Edge u → descendant of v.

④ Cross Edge not ①, ② or ③.

重温性质：

- u ancestor of v iff $d[u] < d[v] < f[v] < f[u]$.
- cannot have $d[u] < d[v] < f[u] < f[v]$.
- If $u \to v \in E$: $d[v] < f[u]$ 在u完成前，必须发现v.

Claim: $(u,v) \in E$ is of type $[u \to v]$.

① or ②
Tree Edge OR Forward Edge : $d[u] < d[v] < f[v] < f[u]$

③
Back Edge : $d[v] < d[u] < f[u] < f[v]$

如：发现v但v已经是灰色。
∴ v的发现时间比u早.
∴ v的完成时间比u晚.

④
Cross Edge (Not Nested) :

① $d[u] < f[u] < d[v] < f[v]$.

可是现在是 type $[u \to v]$
∴ $d[v] < f[u]$
∴ 这种情况不可行.

② $d[v] < f[v] < d[u] < f[u]$.

Theorem [12.9] White Path Theorem.

For all graphs G and All DFS of G, v becomes a descendant of u.

iff ⟹ At the time d[u] the dfs discovers u
There is a path from u to v (u ⇝ v in G)
That consists entirely of white nodes.

Suppose ⟹ v becomes a descendant of u

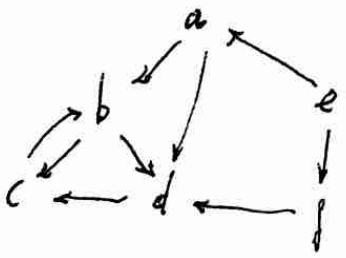
let $u \to u_1 \to u_2 \to u_3 \to \cdots \to u_k \to v$.

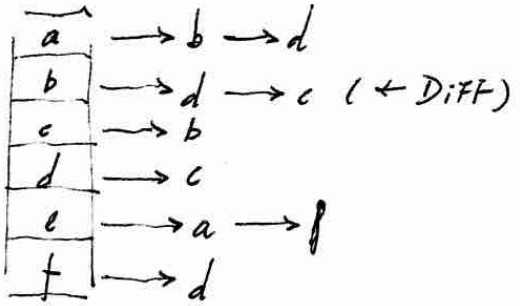
DFS discovery path.

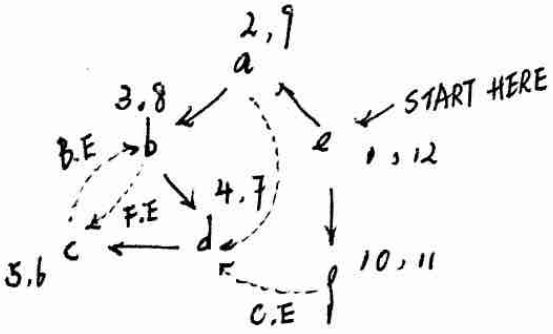
at the time d[u] that u is discovered,
u_1 was not yet discovered (white)

$\Rightarrow$ $u \to u_1 \to u_2 \to u_3 \to \cdots \to u_k \to v$
(white, white, white, white, white)

in fact, all of them in the path are white.

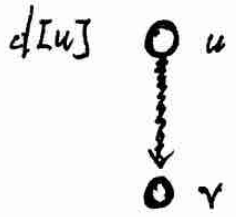






e	a	b	c	d	f
w	w	w	w	w	w

Other Direction :



CLAIM : every node in that path from u to v becomes descendant of u

PROVE BY CONTRADICTION :

⇒ Suppose , by contradiction , the claim is FALSE , one of the node in that path from u to v does not become descendant of u .

LET z be the closest node to u that does not become descendant of u . LET w be predecessor of z

⇒ (w = u) OR (w becomes descendant of u) .

① $d[u] < d[z]$

② $d[z] < f[w]$

③ $f[w] \leq f[u]$

$$d[u] < d[z] < f[u] < f[z]$$

⇒ Overlapping , we cannot have this .

⇒ $d[u] < d[z] < f[z] < f[u]$ ⇒ z is descendant of u

⇒ CONTRADICTION

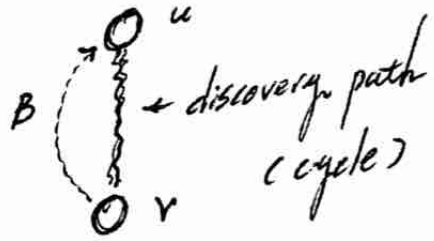
THEOREM [22.11]: ∀ directed G, ∀ DFS of G

G has a cycle iff DFS has a backedge.

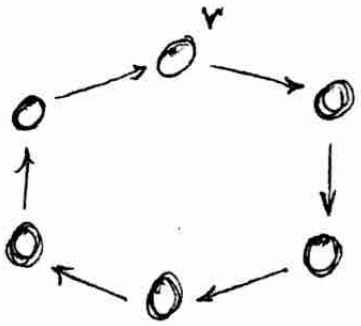
⇒ Suppose DFS of G has a back edge.

⇒ G has cycle.

⇒ Suppose G has a cycle C.



Graph with cycle C:



(the first node discovered)

FREEZE: nobody has discovered
⇒ all white.

v ⇒ become descendant of u.

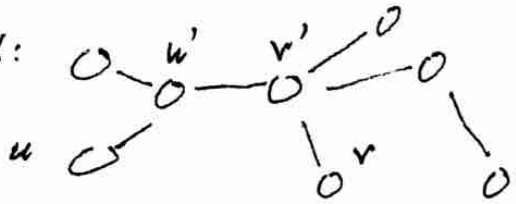
v → u ⇒ back edge ⇒ cycle.

MINIMUM SPANNING TREE (MST) 最小生成树.

定义：一个有 n 个结点的连通图的生成树是原图的极小连通子图，且包含原图中的所有 n 个结点，并且有保持图连通的最少的边.

- A tree is connected undirected graph without cycles.

Ex:



8 Nodes

7 Edges.

- FACT 1: A Tree with n nodes has n-1 edges.

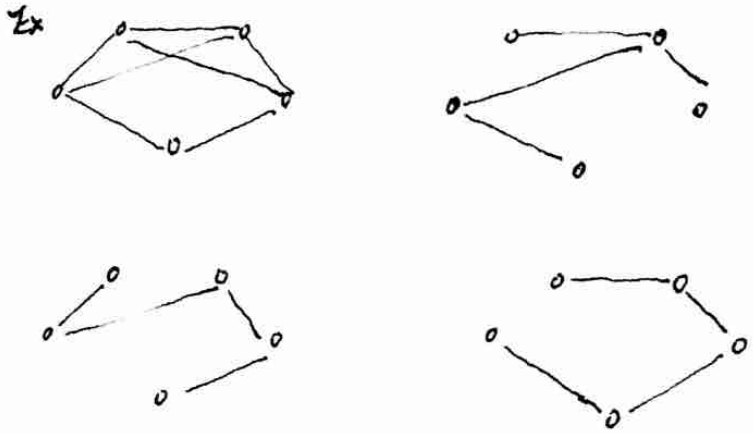
- FACT 2: a) Adding any edge to a tree creates a unique cycle containing this edge.

  b) Removing any edge from this cycle results in a tree again.

IN MST LECTURES: $G=(V, E)$ is an undirected connected graph

- A spanning tree of G is a tree $T=(V, E')$ s.t $E' \subseteq E$.

Ex

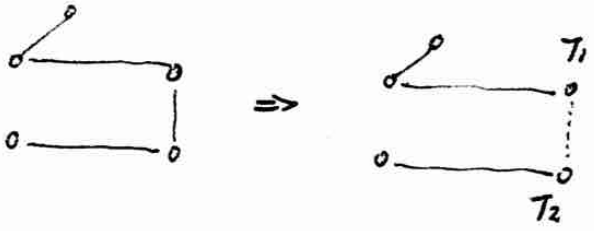


A SPANNING FOREST OF G

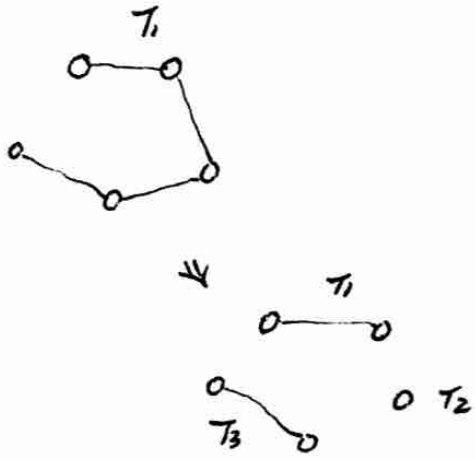
$T_1 = (V_1, E_1)$, $T_2 = (V_2, E_2)$, ..., $T_k = (V_k, E_k)$

$V_1, V_2, \ldots, V_k$ partition of V.

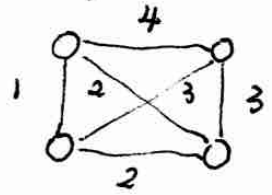
$\bigcup_{i=1}^{k} E_i \subseteq E$.



OR

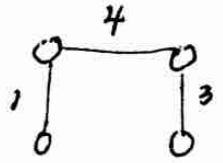


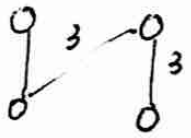
Suppose each edge $e \in E$ has a weight $w(e)$



SPANNING TREE T of G.

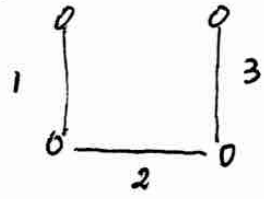
$w(T) = \sum_{e \in T} w(e)$

T_1: 

$w(T_1) = 8$ OR T_2: 

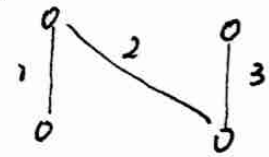
$w(T_2) = 7$.

OR

T_3 

$w(T_3) = 6$

OR.

T_4: 

$w(T_4) = 6$.

MINIMAL SPANNING TREES

$\Rightarrow$ NOT UNIQUE.

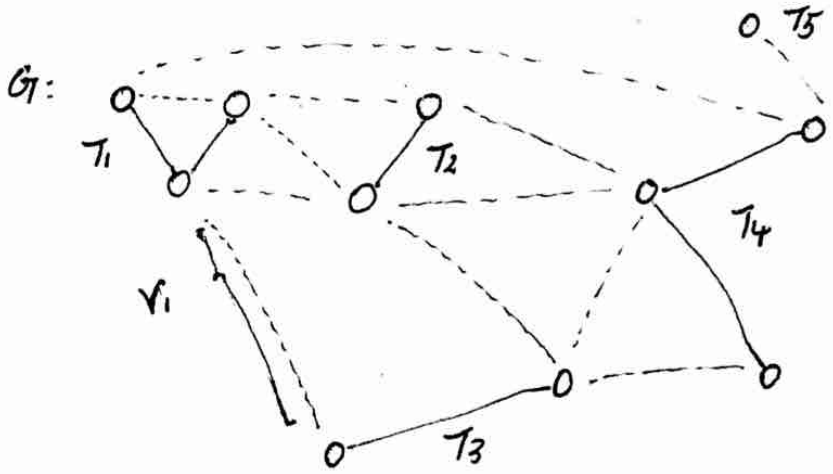
MST PROBLEM: Given a weighted $G = (V, E)$

Find an MST of G.

Completed connected G with n nodes has about n^{n-2}.

THEOREM

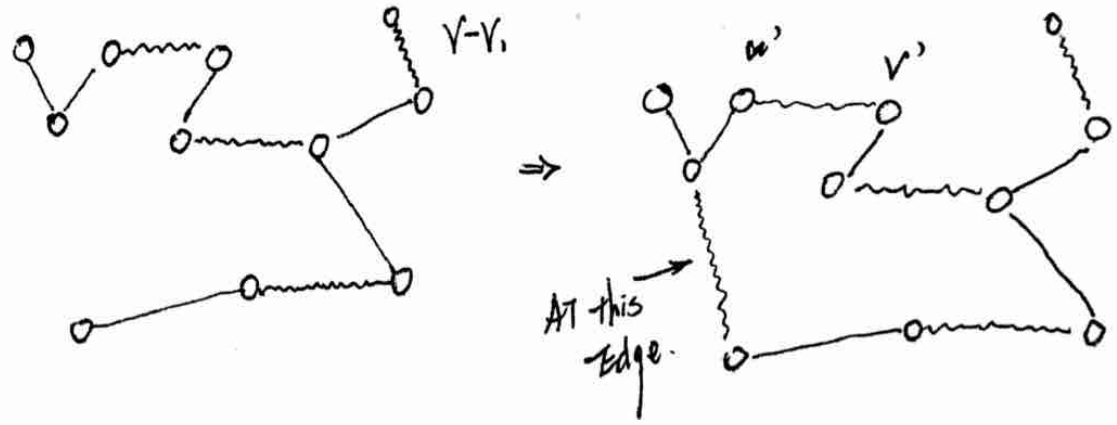
- Suppose some MST T contains the spanning forest $T_1, T_2, \ldots, T_k$ of G.
- LET (u, v) be an edge of min weight between T_i and the other trees $u \in V_i$ and $v \in V - V_i$ (the complement set).
- THEN some MST T^* of G contains $T_1, T_2, \ldots, T_k$ and (u, v)



T MST containing T_1, T_2, T_3, T_4, T_5.

CASE 1. $(u, v) \in T \Rightarrow$ DONE other parts are fragments

$T^* = T$



$\Rightarrow$ cycle created $\Leftrightarrow$ $T' = T + (u,v) \Rightarrow T'$ has a cycle C.

Since $u \in V_i$ and $v \in V - V_i$

$\exists\ u' \in V_i$ and $v' \in V - V_i \Rightarrow (u', v') \in E$.

$w(u,v) \leq w(u', v')$ from way uv was selected.

$T^* = T + (u,v) - (u', v')$ T^* is a spanning tree of T.

$w(T^*) = w(T) + \underbrace{w(u,v) - w(u', v')}_{\leq 0}$

$w(T^*) \leq w(T)$

CASE 2. $(u,v) \notin T$. add the edge.

$w(T^*) \leq w(T)$

$w(T^*) = w(T)$.

MST KRUSKAL'S ALGORITHM.

最小生成树 · 克鲁斯卡尔算法.

(A,B) → 1
(C,D) → 1
(B,D) → 2
(B,C) → 2
(E,D) → 4
(E,A) → 5
(E,C) → 6
(E,B) → 7

|V| = n
|E| = m

EDGE EXAMINED	IN MST?	SPANNING FOREST.
~~(A,B)1~~		{A} {B} {C} {D} {E}
(A,B)1	YES	{A,B} {C} {D} {E}
(C,D)1	YES	{A,B} {C,D} {E}
(B,D)2	YES	{A,B,C,D} {E}
(B,C)2	NO (SAME COMPONENT)	{A,B,C,D} {E}
(E,D)4	YES	{A,B,C,D,E}

⇒ n-1 edges.

方法：将图中边按其权值由小到大的次序顺序选取，若选边后不形成回路，则保留作为一条边，若形成回路则除去。依次选够 n-1 条边，即得最小生成树.

Given: connected undirected $G = (V, E)$

$V = \{1, 2, \ldots, n\}$

E: Array of m weighted edges.

Ex:

| (2,3) 7 | (4,1) 2 | ... |

↑ edge weight

```
BUILD_MIN_HEAP(E)
FOREST → {{1}, {2}, ..., {n}}
MST_EDGES ← Φ
WHILE |MST_EDGES| < n-1 DO
    (u,v) ← EXTRACT_MIN(E)
    Tu ← FIND(u); Tv ← FIND(v)
    IF Tu ≠ Tv. THEN UNION(Tu, Tv)
        UNION(Tu, Tv)
        MST_EDGES ← MST_EDGES + (u,v)
```

INVARIANT: MST_EDGES are contained in some MST of G.

PROOF : USE MST CONSTRUCTION THEOREM.

⇒ m edges
⇒ n nodes

Build min heap ⇒ O(m)

Forest {1}, {2}, {3}, ... ⇒ O(n)

(u, v) ← extract_MIN(E) ⇒ O(m × log m)

UNION(Tu, Tv) ⇒ O(1 × (n-1) times)

FIND ⇒ 2m Finds

TREE STRUCTURE FOR DISJOINT SETS (weight union + path compression)

2m Finds + n-1 unions

O(m log* n) times

⇒ THE DOMINATE COST IS : O(m × log m)

∴ KRUSKAL's ALGORITHM W.C. = O(m log m)

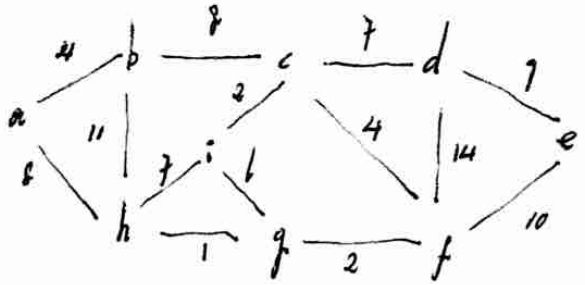
⇒ $m \leq n^2$

⇒ O(m log n) ⇒ O(|E| × log |V|)

⇒ $O(n^2 \log n)$

克鲁斯卡尔算法总结：

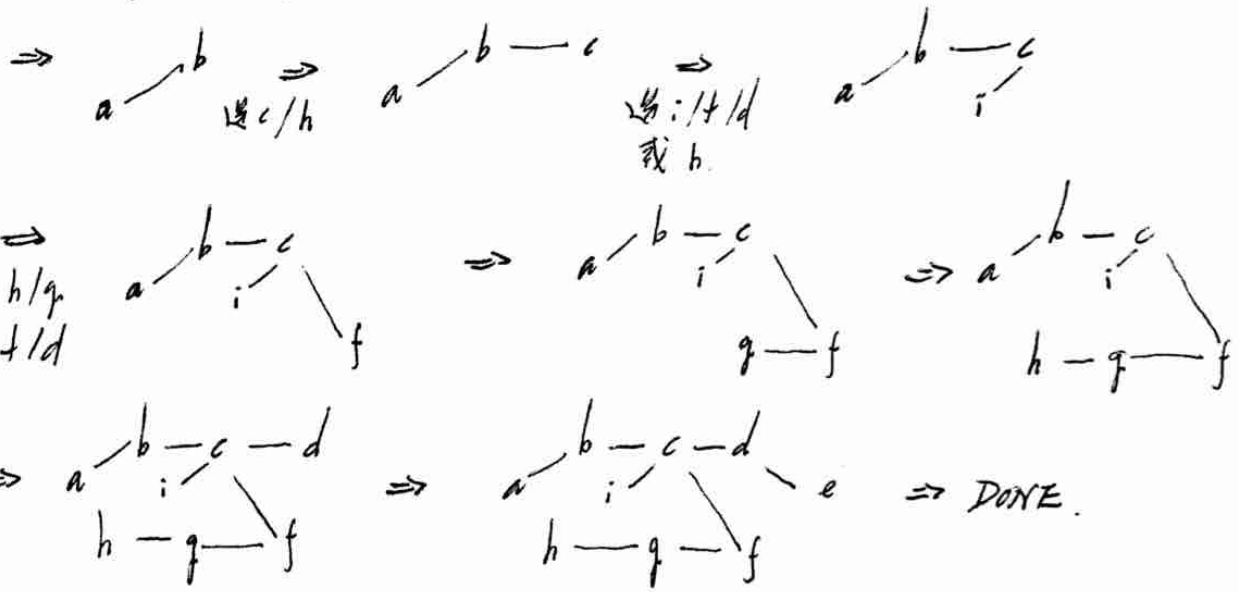
Prim's Algorithm 普利姆算法.



An undirected, connected, weighted graph G.

假设让 a 为我们的起点.

a 可以连 b 或 h. 但 b 的 cost 更小



Q	a	b	c	d	e	f	g	h	i
0	0	∞	∞						∞
1	-	4	∞	∞	∞	∞	∞	8	∞
2	-	-	8	∞	∞	∞	∞	8	∞
3	-	-	-	7	∞	14	∞	8	8
4	-	-	-	7	∞	4	6	8	-
5	-	-	-	7	10	-	2	8	-
6	-	-	-	7	10	-	-	1	-
7	-	-	-	7	10	-	-	-	-
8	-	-	-	-	9	-	-	-	-
9	-	-	-	-	-	-	-	-	-

Prim's algorithm has the property that the edges in the set A always form a single tree. The tree starts from an arbitrary root vertex r and grows until the tree spans all the vertices in V.

SAFE-EDGE: the safe edge added to A is always a least-weight edge in the graph that connects two distinct components. In Prim, the safe edge added to A is always a least-weight edge connecting the tree to a vertex not in the tree.

Each step adds to the tree A a light edge that connects A to an isolated vertex - one on which no edge of A is incident.

This strategy qualifies as GREEDY since at each step it adds to the tree an edge that contributes the minimum amount possible to the tree's weight.

We need a fast way to select a new edge to add to the tree formed by edges in A. The connected graph G and the root r of the MST to be grown are inputs

During the execution of the algorithm all vertices that are not in the tree reside in a min-priority Queue Q based on key attribute. For each vertex v, the attribute $v.key$ is the minimum weight of any edge connecting v to a vertex in the tree
$v.key = \infty$ if there is no such key (edge).
$v.\pi$ names the parent of v in the tree.

$A = \{(v, v.\pi) : v \in V - \{r\}\}$.

伪代码：

```
MST-PRIM(G, w, r)
  for each u ∈ G.V
      u.key = ∞
      u.π = NIL
  r.key = 0
  Q = G.V
  while Q ≠ 0
      u = EXTRACT-MIN(Q)
      for each v ∈ G.Adj[u]
          if v ∈ Q and w(u,v) < v.key
              v.π = u
              v.key = w(u,v)
```

```
MST-PRIM(G, w, r)
1  for each u ∈ G.V
2      u.key = ∞
3      u.π = NIL
4  r.key = 0
5  Q = G.V                          ← while Q is not empty
6  while Q ≠ ∅
7      u = EXTRACT-MIN(Q)
8      for each v ∈ G.Adj[u]        ← neighbours
9          if v ∈ Q and w(u,v) < v.key
10             v.π = u
11             v.key = w(u,v)       } DECREASE-KEY
```

LINE 1-5:

set the key of each vertex to ∞. (except for the root r, whose key is set to 0 so that it will be the first vertex processed), set the parent of each vertex to NIL, and initialize the min-priority queue Q to contain all the vertices.

LOOP INVARIANTS: prior to each iteration of the while loop of lines 6-11.

1. $A = \{(v, v.\pi): v \in V - \{r\} - Q\}$.
2. The vertices already placed into the minimum spanning tree are those in $V - Q$.
3. For all vertices $v \in Q$, if $v.\pi \neq NIL$, then $v.key < \infty$ and $v.key$ is the weight of a light edge $(v, v.\pi)$ connecting v to some vertex already placed into the minimum spanning tree.

LINE 7 identifies a vertex $u \in Q$ incident on a light edge that crosses the cut. Removing u from the set Q adds it to the set $V - Q$ of vertices in the tree, thus adding $(u, u.\pi)$ to A.

LINE 8-11 (FOR LOOP) updates the key and π attributes of every vertex v adjacent to u but not in the tree, thereby maintaining the third party loop invariant.

Runtime:
- implement Q as a binary-min heap $\Rightarrow$ BUILD_MIN_HEAP in $O(V)$ time.
- Body of while-loop executes $|V|$ times, each EXTRACT_MIN takes $O(\lg V)$ times $\Rightarrow$ $O(V \lg V)$ time in total.
- For loop executes $O(E)$ times altogether. sum of the lengths of all adjacency list is $2|E|$.
- DECREASE_KEY in for-loop, $|E|$ times $\Rightarrow$ $O(E \lg V)$

$\Rightarrow O(V \lg V + E \lg V) = O(E \lg V)$