Mead®

SCHEDULE

	1	2	3	4	5	6	7	8
TIME								
MONDAY	629 Analysis of Algorithms Tue / Thu CHEN 108 12:45 pm - 2:00 pm							
TUESDAY								
WEDNESDAY	681 Seminar HRBB 124 MW 04:10 pm - 05:25 pm							
THURSDAY								
FRIDAY								
SATURDAY								
SUNDAY								

629 Analysis of Algorithms

Sep 1 2015

algorithm solving problems :

* problem statement need to be specific

Ex: Given an array A[1..n] find the largest

* Algorithm design speed / running time

* Algorithm analysis

* Discussion

Ex = Given an array A[1..n], find the largest

Algo 1 time = O(n)

```
max = A[1]
for i = 2 to n do
    if max < A[i]
    then max = A[i]
return (max)
```

Algo 2

MAX(i, j) = return the largest in A[i, j]

```
if i ≥ j then return A[i]
else k = ⌊(i+j)/2⌋
    max1 = MAX(i, k)
    max2 = MAX(k+1, j)
    return largest(max1, max2)
main
    return MAX(1, n)
```

Let T(n) be the time on an array of n elements

T(1) = O(1) ≤ c recurrence relation

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

* replace O(f) by ≤ cf

$\leq 2T(n/2) + c$

* try a few steps to extend the recurrence

$T(1) \le c$

$T(n) \le 2\,T(n/2) + c$

$\Downarrow$

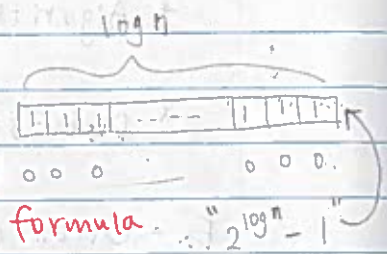
$T(n/2) \le 2\,T(n/2^2) + c$

$T(n) \le 2\left(2\,T(n/2^2) + c\right) + c$

$\le 2^2\,T(n/2^2) + 2c + c$

$\le 2^2\left(2\,T(n/2^3) + c\right) + 2c + c$

$= 2^3\,T(n/2^3) + 2^2c + 2c + c$



* from the extension, find a nonrecursive formula. ∴ "$2^{\log n} - 1$"

$\Rightarrow T(n) \le 2^k\,T(n/2^k) + 2^{k-1}c + 2^{k-2}c + \cdots + 2c + c$

let $n = 2^k$, $k = \log_2 n$

$\le \underbrace{2^{\log_2 n}}_{n}\,\underbrace{T(1)}_{\le c} + \left(2^{\log_2 n - 1}c + 2^{\log_2 n - 2}c + \cdots + 2c + c\right)$

$= c \cdot n + (n-1)c = (2n-1)c = O(n)$

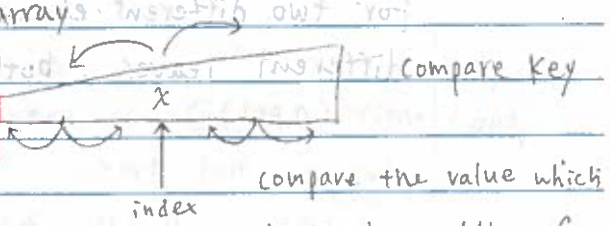
* calculate the value

Sep 3 2015

Given a set of items, how do we organize it so that we can search fast?

Ans: Sort them.

Search: Binary search $O(\log n)$



assume n is large

n	log n
32	5
1000	10
1,000,000	20
10^9	30

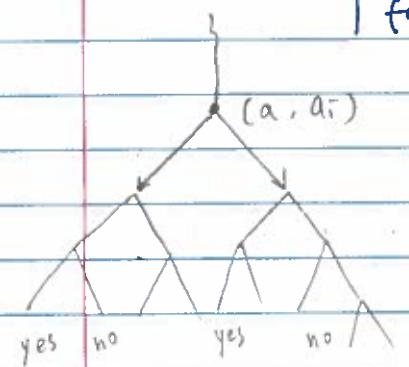
Do better? No!

If you use comparison, there is no other search algo that can be lower than $O(\log n)$.

- **Thm** Based on comparison, no search algo can do better than $O(\log n)$ on a set of n items.

pf $S = \{1, 2, 3, \ldots, n\}$

$$A(a, S) = \begin{cases} \text{true} & \text{if } a \in S \\ \text{false} & \text{if } a \notin S \end{cases}$$



Argument ① have to use conditional statements ex if/whil

② have to use comparision

For 2 different element in S, the tree leads to two different leaves both labeled "yes"

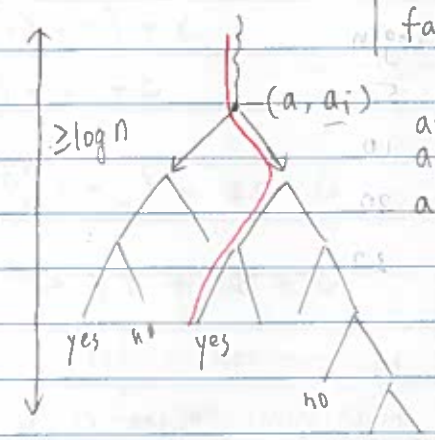
For two different elements in S, the tree leads to two different leaves, both labeled "yes"

Suppose not true

a_i, a_j $\quad a_i < a_j$

Let $a = a_i + 0.5$

$A(a, S) = \begin{cases} \text{true} & \text{if } a \in S \\ \text{false} & \text{if } a \notin S \end{cases}$



Suppose not true

$a_1 \; a_2 \quad a_2 < a_1$

$\geq n+1$ leaves

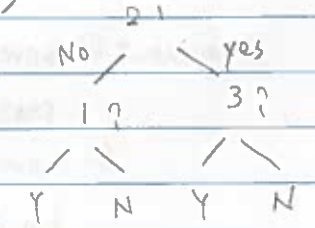
Why we use binary tree?

Because we want to keep only two children.

It's easier to merge, search .....

search

(1, 2, 3)



This means we have n+1 leaves

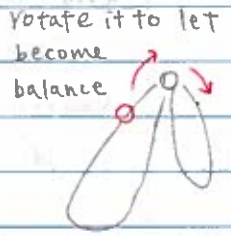
like database — add / delete / sort

Given a dynamic set of items, how do we organize it so that we can search fast?

& update

so that we can search & update in $O(\log n)$ time / opt ~~operation~~

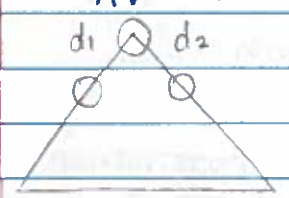
Ans: Yes! Use balanced trees



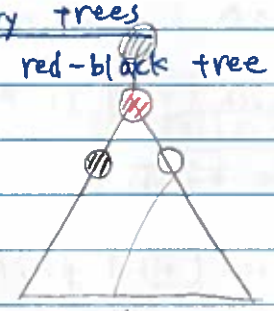
Q How do we keep it balanced? ⇒ By rotations

Binary trees

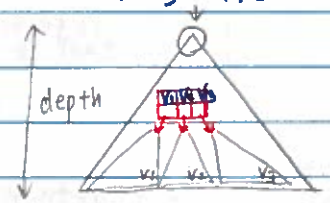
AVL tree



red-black tree



2-3 tree



depth

d_1 = records how deep the left side

d_2: right

⇒ to know if it is balanced

red: 0

black: 1

is a tree such that:

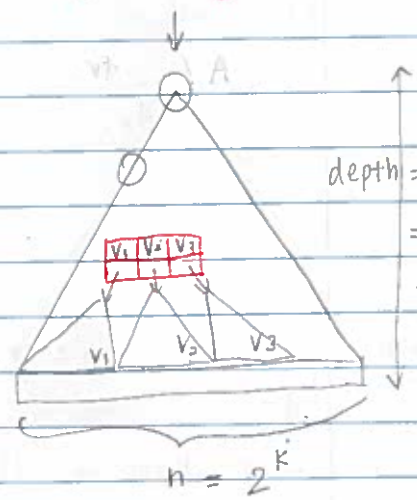
1. each non-leaf node has either 2 or 3 children
2. all leaves are at the same level
3. elements are stored in leaves only, i.e. sorted order
4. each non-leaf node has three values & three pointers to its three children

node :
record of
V_1, V_2, V_3 = key
P_1, P_2, P_3 = ↑ node
pointer
end

Search ($_2\overline{|3}$; a) $O(depth) \le O(\log n)$

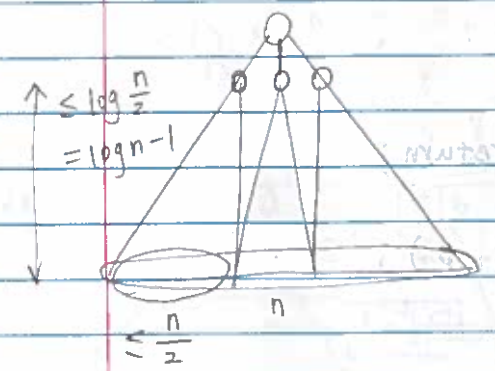
1. if $_2\overline{|3}$ is a leaf :
   return ($_2\overline{|3}\uparrow . V_1 == a$)
2. case :
   $_2\overline{|3}\uparrow , V_1 \ge a$
   return Search ($_2\overline{|3}\uparrow P_1$, a)
   $_2\overline{|3}\uparrow V_1 < a \le {}_2\overline{|3}\uparrow V_2$
   return Search ($_2\overline{|3}\uparrow P_2$, a)
   $_2\overline{|3}\uparrow P_3 \neq$ Null
   return Search ($_2\overline{|3}\uparrow P_3$, a)
   default = return false

depth ≤ log n



⇒ Worst case :
complete binary tree

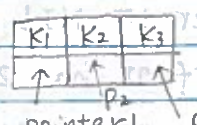
2-3 tree of n leaves has depth ≤ log n



Sep 8 2015 * 2-3 Tree

1. a node has 2 or 3 children
2. all leaves are the same level

type :
node = record of
K_1, K_2, K_3 = key
pointer P_1, P_2, P_3 = ↑ node



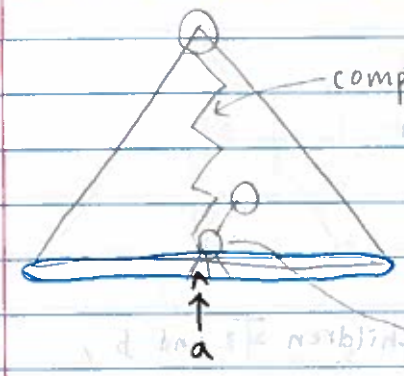
a leaf

$\log_{10} n \ge \frac{1}{4} \log_2 n$

* a 2-3 tree with n leaves has height ≤ $\log_2 n$

Search ($_2\overline{|3}$, a) can be done in time $O(\log n)$ O(depth) ≤

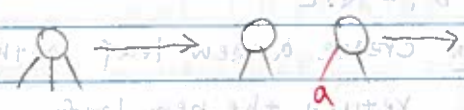
Insertion ($_2\overline{|3}$, a) = add a into $_2\overline{|3}$



compare a with K_1, K_2, K_3

if there are only two children, just add a in

what if there are already three children?



propogate upward check their parent until satisfied all invariants

Insert (v : ↑node ; a : key ; brother : ↑node) ;
(pointer) (pointer)

* v: is not a leaf

1. brother = Nil ; NIL
2. if v is the father of leaves
   then handle this properly ; return ;
3. Cases :
   v↑.k1 ≥ a : Insert (v↑.p1 , a , b) ;

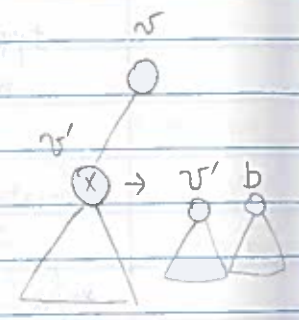
   check recursively

   v↑.k1 < a ≤ v↑.k2 or v↑.p3 == NIL : Insert (v↑.p2 , a , b) ;

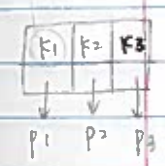
   check split or not

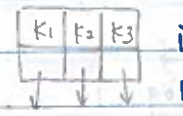
   default : Insert (v↑.p3 , a , b) .

4. if brother != NIL then
   Case 1 : if v has 2 children ,
   add b as another child of v ;



   else
   split the 3 children of v and b into 2 groups of 2 , create a new node brother and let it be the root of one of the groups and let v be the root of the other group





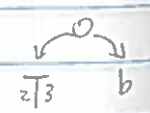
main (根 , a)

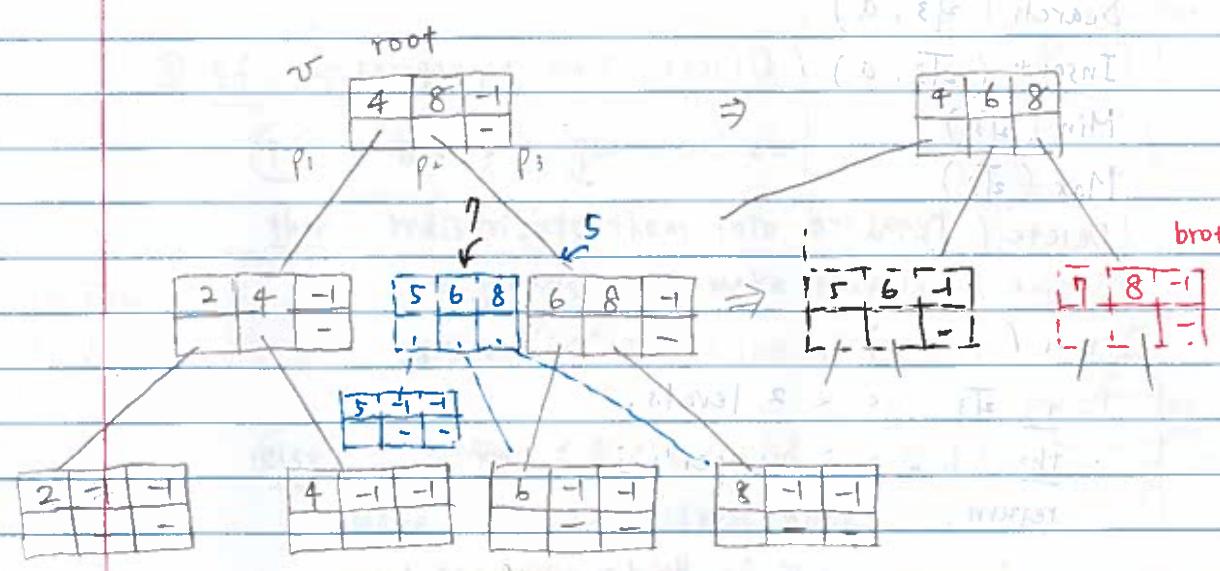
① if (根 is a leaf or 根 = φ)
then insert a properly , return

call by reference

② Insert (根 , a , b)

③ if b != NIL
then create a new leaf with children 根 and b ,
return the new root.





1st iteration

a = 5
1. brother = Null
2. —
3. Case 1 : —
   Case 2 : Insert (v.p2 , a , b)
4. —

a = 5
1. brother = Null
2. Return

a = 7
1. brother = Null
2. —
3. Case 1 : —
   Case 2 : Insert (v.p2 , a , b)
4. brother = new node

a = 7

It uses a lot of recurrence.

Sep 10 2015

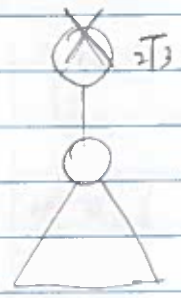
Search (2-3 , a)
Insert (2-3 , a)
Min (2-3)
Max (2-3)
Delete (2-3 , a)
} O(log n) time per operation

Splice / Split

main (2-3 , a)

1. if 2-3 has ≤ 2 levels, leaf
   then handle it properly ;
   return ;

2. Delete (2-3 , a , 1son) ← boolin value

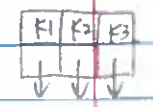


3. If 1son then
   2-3 = 2-3↑.p1 ;

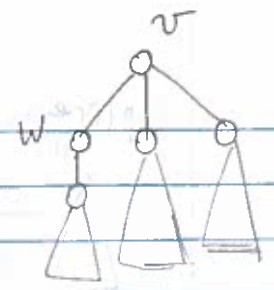
Delete (v, a, 1son)
0. 1son = false;
1. if v is the father of leaves, (= v is not a leaf)
   then handle it properly ; return ;

2. Case
   ① a ≤ v↑.k1 : w = v↑.p1 ;
   ② v↑.k1 < a ≤ v↑.k2
   or
   v↑.p3 = Nil : w = v↑.p2 ;
   default : w = v↑.p3 ;

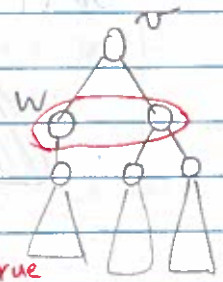
③ Delete (v, a, 1son');



④ If 1son' then (1son' = true)
   if v has ≥ 4 gran-children
   then redistribute them into at least two groups, and make them children of v, 1son = false



   else = v has ≤ 3 grandchild
   make w a 3-children node and the only child of v; 1son = true

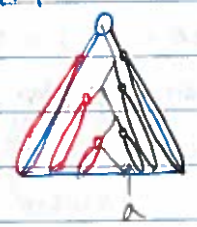


Splice

taller shoter

find a proper position, check, and add it in

Split

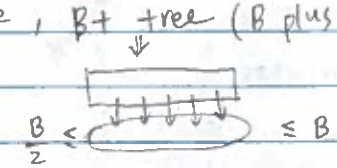


split the red / black tree in O(logn) time

Datastructures for efficient Insert / Delete / Search

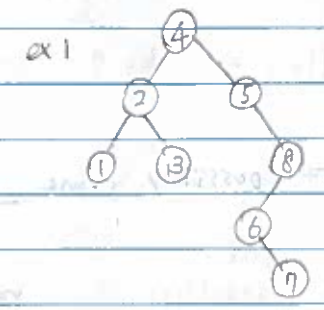
* by split / merge internal nodes

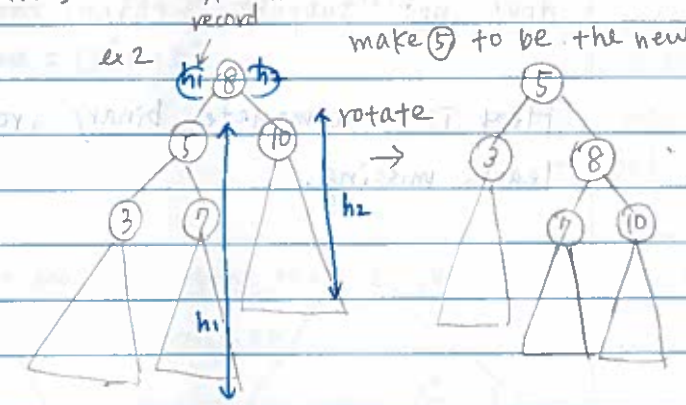
2-3 tree, B+ tree (B plus tree)



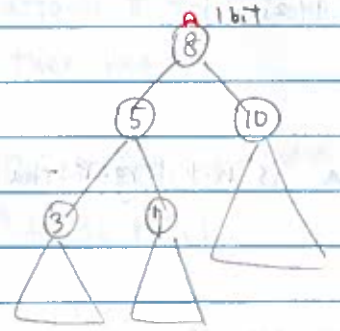
* by rotation on BST (binary search tree) to keep balance

in-order traverse = root → left → right



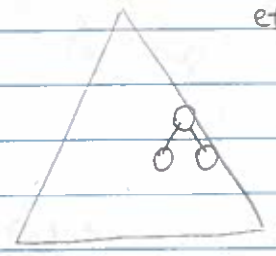


- AVL-tree : h1 & h2 can differ at least one, otherwise it needs to be rotated
- red-black tree : number of blace are the same for each path
  red can not be consecutive!



Sep 15 2015

Heap

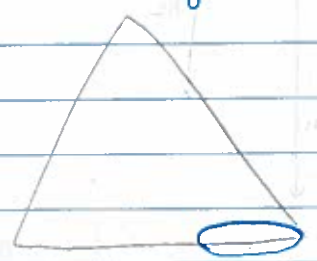


efficient for Insert / Delete but not for search

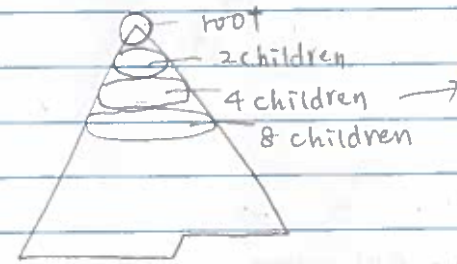
support Min, Insert, Delete

does not support Search efficiently ⇒ Why?

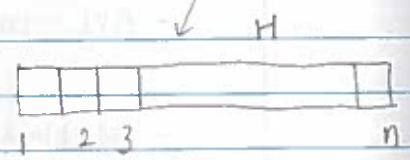
Heap is a compelete binary tree with possibly some rightmost leaves missing.



⇒



represented by an array



⇒ the child of H[i] are H[2i] and H[2i+1]

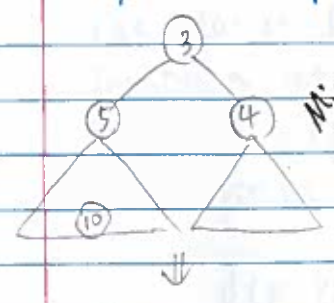
the father of H[i] is $H[\lfloor i/2 \rfloor]$

In a heap, every node has a value that is not larger than its children.

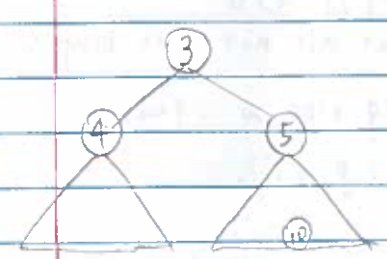
If heap supports your application, then use it.

If not, use 2-3 tree, AVL or red-black tree. (worse space)

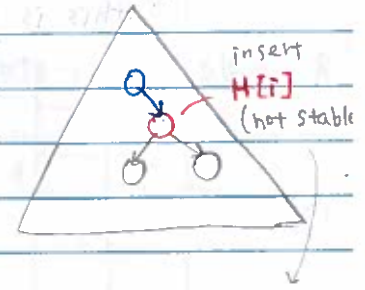
Why does heap not support search?



Min heap



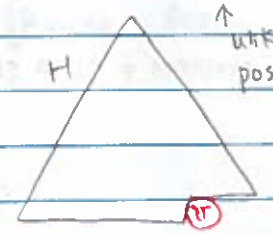
- Min(H)
  return (H[1])
- Heapfy (H, i)
  ⇕
  H[1, n] is a heap
  ⇕
  except the value of H[i]
  time = O(log n)



Check the value of H[i] & its fathers children

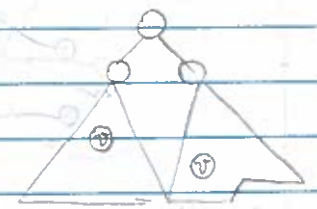
H[i] may be flipped

- Insert (H, v)



* we may have more than 2 identical value at different position.

time = O(log n)

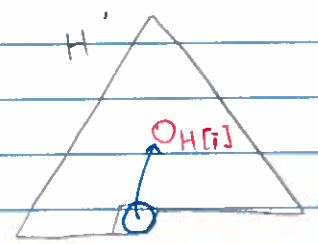


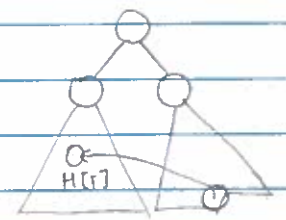
attach v to H[n+1]

then heapfy

- Delete (H, i) — index ⇒ exact position

⇓ delete H[i]





H[i] = H[n]

n = n-1

then heapfy

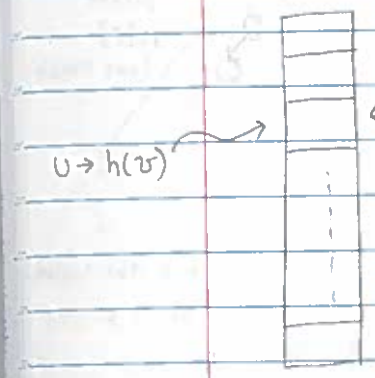
time = O(log n)

use the last node to replace H[i]

fill

Hashing

" this is not a model base on comparisons



$|h(v)| \approx \log_2 n$ (bit)

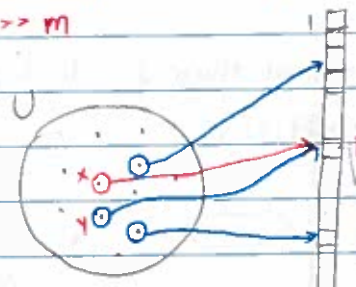
→ works for numbers that are not that large

integers

given a universal set U of n elements

suppose you are playing with subsets of randomly m integers in U

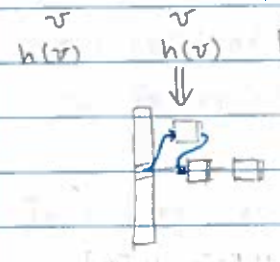
$n >> m$



$h = U \to [1, m]$

collision is a problem ⇒ use a chain to solve it

Search, Insert, Delete



O(nlogn) Algorithm for closest pair I

The Closest pair problem

Input: a set $P = \{p_1, p_2, \dots, p_n\}$ of n points in the plane ($\mathbb{R}^2$)

Notation: $d(p_i, p_j)$ = Euclidean distance

So if $p_i = (x_i, y_i)$ and $p_j = (x_j, y_j)$

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

Output: a pair $p^* q^* \in P$ of distinct points that minimize $d(p, q)$ over $p, q \in P$

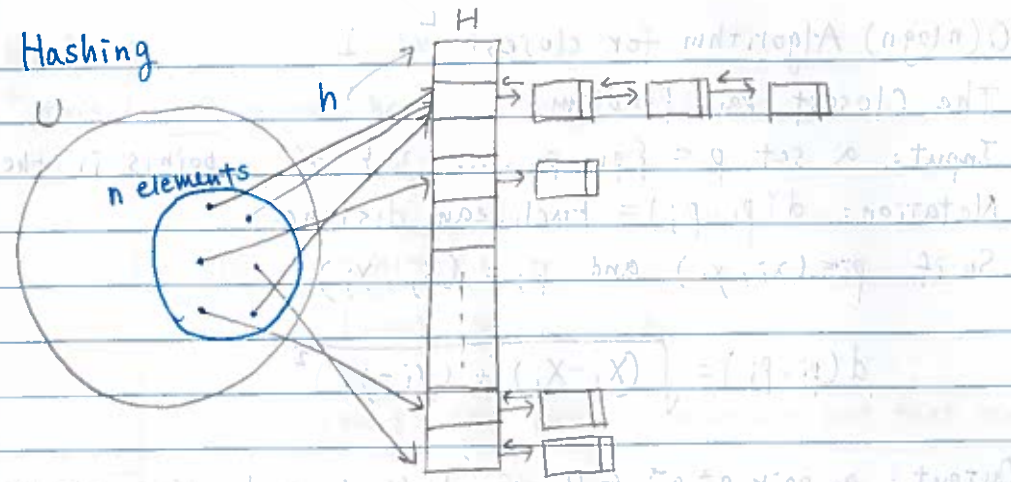
Initial Observations

Assumption: (for convenience) all points have distinct x-coordinates, distinct y-coordinates,

Brute-force Search: takes $\Theta(n^2)$ time

Sep 19 2015 Hashing

$h: U \to [0, 1, \cdots, m-1]$



Struct node {
    int key;
    struct node * next;
    struct node * prev;
    others;
}

Struct node * H[m]

(closed) Open hashing
(Open) Closed address hashing → put element to the chain

check collision → { Open hashing, Closed address hashing } { Closed hashing, Open address hashing ? }

time *

O(length of the list) Search (H, v)
// search the list H[h(v)]

O(1) Insert (H, v)
// insert into the head of list H[h(v)]

O(1) Delete (H, *p) // remove *p directly
Delete (H, v) = Search(H, v) + Delete(H, *p)

Assume h hashes each element into a slot with prob = $\frac{1}{m}$

under this assumption, how good the operations Search, Insert, Delete can be?

Probability Theory

random variable = is a variable that can be certain values, each with a certain probability.

X_i : the length of H[i]

expectation = prob. value

$E[X_i] = \sum_{v \text{ is the value taken by } X_i}$ prob. $[X_i = v] \cdot v$

if $X_i = \sum_{j=1}^{n} Y_{ij} \Rightarrow E[X_i] = \sum_{j=1}^{n} E[Y_{ij}] = n \cdot \frac{1}{m} = \frac{n}{m}$

$$Y_{ij} = \begin{cases} 1, & \text{if } h(j) = i \\ 0, & \text{if } h(j) \neq i \end{cases}$$

$E[Y_{ij}] = 1 \cdot \text{prob}[h(j) = i] + 0 \cdot \text{prob}[h(j) \neq i]$

$= \frac{1}{m}$

Search (H, v)

Case 1: v is not in H $\Rightarrow h(v) \Rightarrow H[h(v)] \Rightarrow$ steps = length of H[h(v)]

$E[\text{Time}] = \frac{n}{m}$

Case 2: v is in H $\Rightarrow$ intuition: # steps = $\frac{1}{2}$ length of H[h(v)]

Suppose H holds $x_1, x_2, \cdots, x_i, \cdots, x_n$, inserted in this order

T = the # of steps for searching v

$$T_k = \begin{cases} \#\text{ of nodes before } x_k \text{ in } H[h(v)] & \text{if } v = x_k \\ 0 & \text{otherwise} \end{cases}$$

$$T = \sum_{k=1}^{n} T_k$$

$$X_{kj} = \begin{cases} 1 & \text{if } v = X_k \text{ \& } X_j \text{ is before } X_k \text{ in } H[h(v)] \\ 0 \end{cases}$$

$$T_k = \sum_{j=1}^{n} X_{kj}$$

$X_1\ X_2\ X_3\ \cdots\ X_k\ \cdots\ X_j\ \cdots\ X_n$

$$T = \sum_{k=1}^{n} \sum_{j=1}^{n} X_{kj}$$

$$E[T] = \sum_{k=1}^{n} \sum_{j=1}^{n} E[X_{kj}]$$

$$E[X_{kj}] = \begin{cases} 1 \cdot \frac{1}{n} \cdot \frac{1}{m} & j > k \\ 0 & j \le k \end{cases}$$

$$E[T] = \sum_{k=1}^{n} \sum_{j=k+1}^{n} \frac{1}{nm} = \frac{1}{nm} \sum_{k=1}^{n} (n-k) = \frac{1}{nm} \frac{n(n-1)}{2} < \frac{n}{2m}$$

The expected # of steps for Search (H, v) $= \begin{cases} \frac{n}{m} & \text{if } v \text{ is not in } H \\ \frac{n}{2m} & \text{if } v \text{ is in } H \end{cases}$

Construction of hash function.

$h : U \longrightarrow [0, \dots, m-1]$

$[0 \dots k]$

k is very large $k \gg n$

$k \in U$, $k \bmod m$ $\longrightarrow$ not good in general

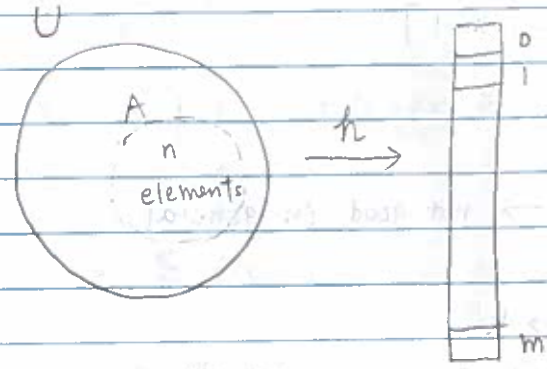
pick a prime p, $p > k$

pick two numbers a, b, $1 \le a < p$, $0 \le b < p$

$h(k) = ((k \cdot a + b) \bmod p) \bmod m$

$h = h_{p,a,b}$

Sep 22-2015 Hashing



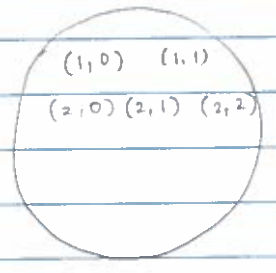
- if h is ideal, then Search, Insert, Delete take time $O(\frac{n}{m}+1)$
- if $n = O(m)$, then these operations take $O(1)$ time, expected.
- Construct a hash function, pick a prime $p \geq |U|$

  pick a, b, $\quad 1 \leq a \leq p-1$, $\quad 0 \leq b \leq p-1$

  then $h_{p,a,b}(x) = ((ax+b) \bmod p) \bmod m$

  makes a "good" hash from U to $[0, m-1]$

Collection of hash functions



Let C be a collection of hash functions from U to $[0, m-1]$, we say C is a universal hashing if for any two elements k and g in U, there are at most $|C|/m$ hash functions in C that make a collision of k and g.

$\{ f_{a,b,p} \mid 1 \leq a \leq p-1, 0 \leq b \leq p-1 \}$ is a universal hashing.

Thm given a set A of n elements in U, if you randomly pick a hash function h in a universal hashing, then the expected length of each list $H[i]$ is $\leq \frac{n}{m} + 1$

Proof: if $H[i]$ is empty, then done

if $H[i] \neq \phi$, pick an element $k \in A$

$h(k) = i$

define

$$Z_{k,j} = \begin{cases} 1 & \text{if } h(k) = h(j) \\ 0 & \text{otherwise} \end{cases}$$

then the length of $H[i]$

$$Z = \sum_{j \in A} Z_{k,j} = 1 + \frac{n-1}{m} \leq \frac{n}{m} + 1$$

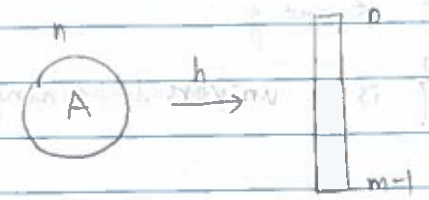
$$E[Z] = \sum_{j \in A} E[Z_{k,j}]$$

$$\| \quad 1 \cdot \frac{1}{m} + 0 \cdot \frac{m-1}{m}$$

Perfect hashing

h : with no collision

assume the given set A (if n elements) is fixed (no insert/delete)



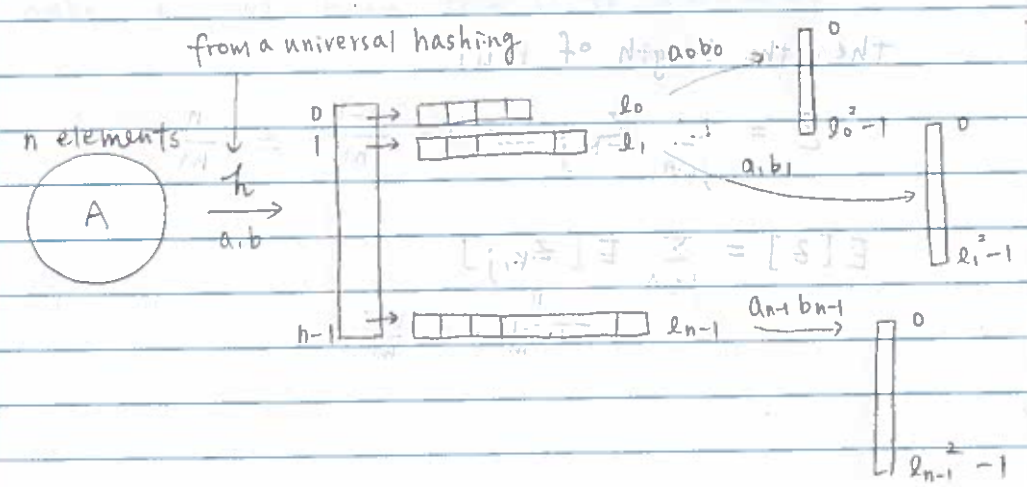
Thm Let $m = n^2$, then randomly pick a hash function from a universal hashing, then the hash function is perfect, with prob. $> \frac{1}{2}$

Proof

$$\text{prob}[\text{there is a collision}] \le \sum_{K,j \in A} \text{prob}[h(K) = h(j)]$$

$$= \binom{n}{2} \frac{1}{m}$$

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



- $\ell_j^2 = \ell_j + 2\binom{\ell_j}{2}$

The extra space used in this 2-step hashing

$$E\left[\sum_{j=0}^{m-1} \ell_j^2\right] = E\left[\sum_{j=0}^{m-1} \left(\ell_j + 2\binom{\ell_j}{2}\right)\right] = E\left[\underbrace{\sum_{j=0}^{m-1} \ell_j}_{n} + 2\sum_{j=0}^{m-1}\binom{\ell_j}{2}\right]$$

$$= n + 2E\left[\sum_{j=0}^{m-1}\binom{\ell_j}{2}\right] \le 2n-1$$

$\sum_{j=0}^{m-1}\binom{\ell_j}{2}$: # of collisions

(K, j) $\frac{1}{m}$

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

Sep 24 2015

$\{ f_{a,b,m} \mid 1 \le a \le p-1,\ 0 \le b \le p-1 \}$

$f_{a,b,m} = ((ax+b) \bmod p) \bmod m$

is a universal hashing from U to $[0, m-1]$ where p is prime $> |U|$

Constructing a perfect hashing for a set (A) of n elements in U

Algo Perfect Hash(A)

1. pick a prime $p > |U|$
2. randomly pick a and b, $1 \le a \le p-1$, $0 \le b \le p-1$

   Let $h = h_{a,b,n}$, and let X_i be the set of element in A that are hashed into i, $0 \le i \le n-1$
3. While $\sum_{i=0}^{n-1} |X_i|^2 > 4n$ do

   randomly pick a, b again to

   make $h = h_{a,b,n}$, with $X_0, \dots, X_{n-1}$
4. for $i=0$ to $n-1$ do

   randomly pick a_i and b_i, $1 \le a_i \le p-1$, $0 \le b_i \le p-1$

   let $h_i = h_{a,b,|X_i|^2}$

   while h_i is not perfect from X_i to $[0, \dots, |X_i|^2]$ do

   randomly pick a_i and b_i

   and make $h = h_{a,b,|X_i|^2}$

★ Output: $\{(a_1,b_1), (a_2,b_2), \dots, (a_{n-1}, b_{n-1})\}$

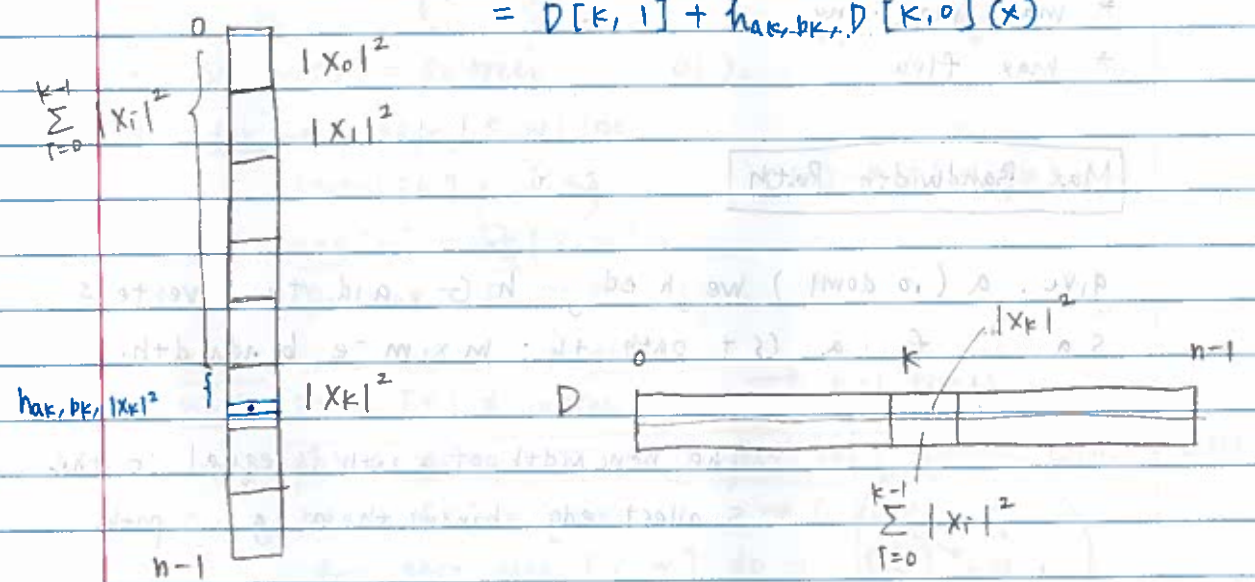
Our perfect hash function h_A is from A to $[0, N]$

where $N = \sum_{i=0}^{n-1} |X_i|^2 \le 4n$

$h_A(x)$: 1. $k = h_{a,b,n}(x)$

2. $h_A(x) = \sum_{i=0}^{n-1} |X_i|^2 + h_{a_k, b_k, |X_k|^2}(x)$

$= D[k,1] + h_{a_k, b_k, D[k,0]}(x)$



Optimization

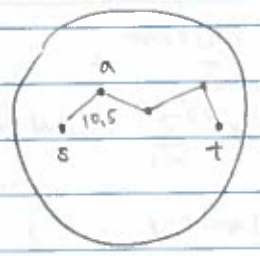
Graph optimization Problems

* Shortest path
* min. spanning tree
* max. matching
* max flow

Max Bandwidth Path

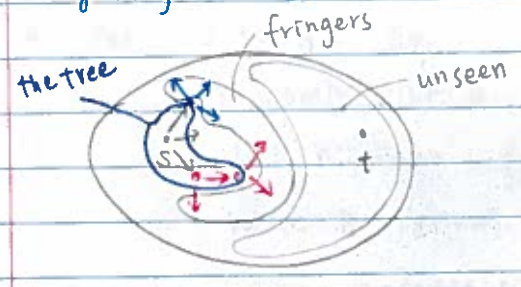
given a (randomly) weighted graph G, and two vertexs s and t, find a (s,t)-path that maximize bandwidth.

G



The bandwidth of a path is equal to the smallest edge bandwidth along the path.

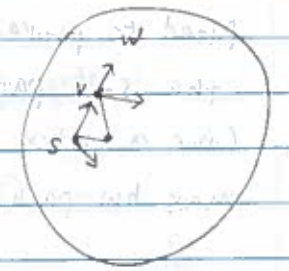
Alg (Dijkstra's)



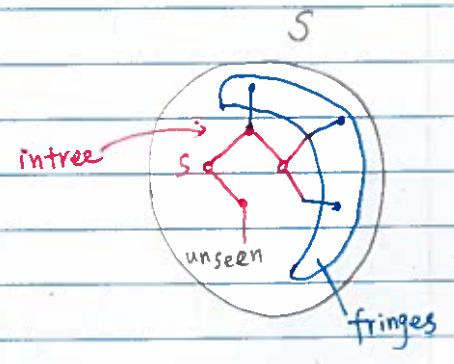
Max-bandwidth Path (G, s, t) time = $O(n^2)$

Alg (Dijkstra's)

1. for v = 1 to n do — O(n)
   for each vertex v = 1 ... n do
   status[v] = unseen;



2. status[s] = intree; — O(1)
3. for each edge [s, w] do
   status[w] = fringe; — O(n) → O(m log n)
   capacity → cap[w] = wt[s, w]; (wt: weight)
   Dad[w] = s;
4. while there are fringes do → n−1 times
   while status[t] ≠ intree do
   pick a fringer v of largest cap[ ]; — O(n) → O(1)
   status[v] = intree; → O(log n)
   for each edge [v, w] do — O(n) max- if using heap
   if status[w] = unseen — executes only ≤ m times for the entire algo execution
   then status[w] = fringe; — time = O(m log n)
   Dad[w] = v;
   cap[w] = min{cap[v], wt[v, w]}; — O(log n) ←
   elseif status[w] = fringe & cap[v] < min{cap[v], wt[v,w]}
   then Dad[w] = v;
   cap[w] = min{cap[v], wt[v, w]};
5. return (Dad[ ])



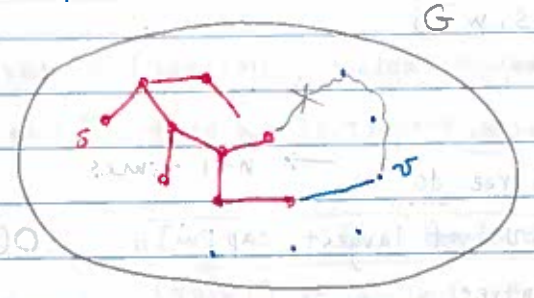
★ if we use a max-heap
⇒ time = O(m log n)

Sep 29-2015

Max-bandwidth path (G, s, t)

Need to prove:
the s-t path in the tree is a max-bandwidth path
Once a vertex v becomes "in-tree", the path in the tree is a max-bw path from s to v.

Proof by Contradiction & Induction



Approach #2
1. Construct a max spanning tree T;
2. Connect s and T in T
→ Kruskal (G, s, t)

Max Spanning Tree

Alg (Kruskal's) time = O(mlogn)

Kruskal (G, s, t)

O(mlogn) 1. sort the edges of G in nonincreasing order:
$e_1, e_2, \dots, e_m$

2. $T = \phi$; for v=1 to n do Makeset(v)

3. for i=1 to m do
$e_i = [v_i, w_i]$; if Find(v_i) ≠ Find(w_i)
if v_i and w_i are in two different components of T;
then $T = T + \{e_i\}$ → UNION(S_1, S_2)
// at this point, T is a max. spaning tree ;

(S_1 = Find(v_i), S_2 = Find(w_i))

4. connects s and t in T

def. Makeset(a)
make a single element set

UNION (S_1, S_2)
merge S_1 and S_2 into a single set

Find(a)
find the set that contains a

time = O(logn)

Oct 01- 2015

Alg 2 (kruskal's) time = O(mlogn)

0. $T = \phi$
1. Sort the edges in non-increasing order; $\{e_1, e_2, \dots, e_m\}$ — O(mlogn)
2. for vertex $v = 1$ to n do — O(n)
   MakeSet(v)
3. for $i = 1$ to m do — $n \le m-1$ — O(mlogn)
   let $e_i = [v_i, w_i]$;
   $r_1 = \text{Find}(v_i)$; $r_2 = \text{Find}(w_i)$;
   if $r_1 \neq r_2$ then
   $T = T \cup \{e_i\}$; (add e_i to T)
   UNION (r_1, r_2)
4. return the s-t path in T

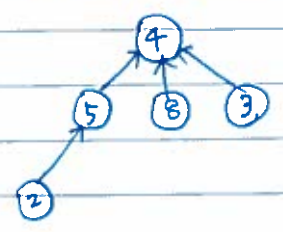
MakeSet(v): · Dad[v] = 0
make a set $\{v\}$

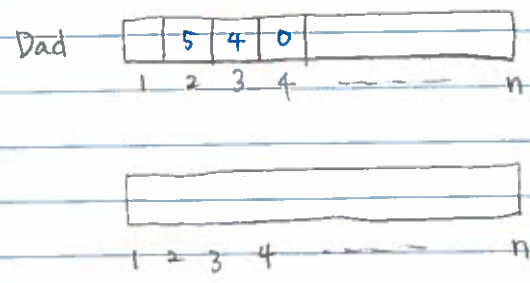
Find(v):
return a representative of the set containing v

UNION (r_1, r_2):
combine the two sets that contain r_1 & r_2, respectively

Use a tree to represent a set in which each node has pointer to it father

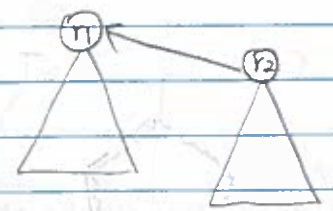
$\{2, 3, 5, 8, 4\}$ Dad[v] = 0



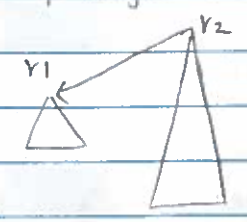


· Find(v);
1. w = v;
2. while Dad[w] ≠ 0 do
   w = Dad[w];
3. return (w)

· UNION (r_1, r_2)
Dad $[r_2] = r_1$;



What if height of r_2 is taller than r_1 ?

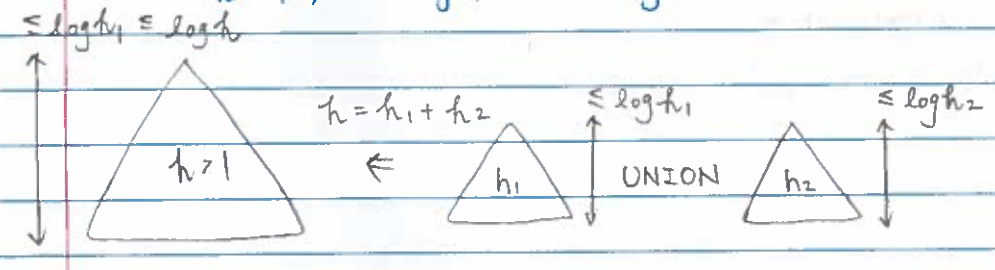


· UNION (r_1, r_2)
Case:
rank$[r_1]$ > rank$[r_2]$
Dad$[r_2] = r_1$;
rank$[r_1]$ < rank$[r_2]$
Dad$[r_1] = r_2$;
rank$[r_1]$ = rank$[r_2]$
Dad$[r_2] = r_1$; rank$[r_1]$++;

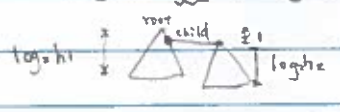
Thm with rank[·], the height of any tree is $\le \log n$

Pf by induction on the number h of nodes of a tree.

$h = 1$, height $= 0 = \log h$

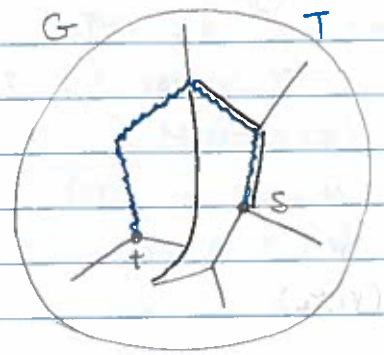


If $h_1 = h_2$, $h_2 \le \frac{h}{2}$ ⇒ height of $h \le \log h_2 + 1 \le \log \frac{h}{2} + 1 = \log h$

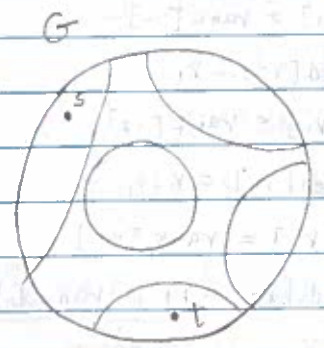


★ The alg 2 returns a max-bw path from s to t

Pf: T is max spaning tree.



Claim
all black edges have capacity not smaller than the blue ones



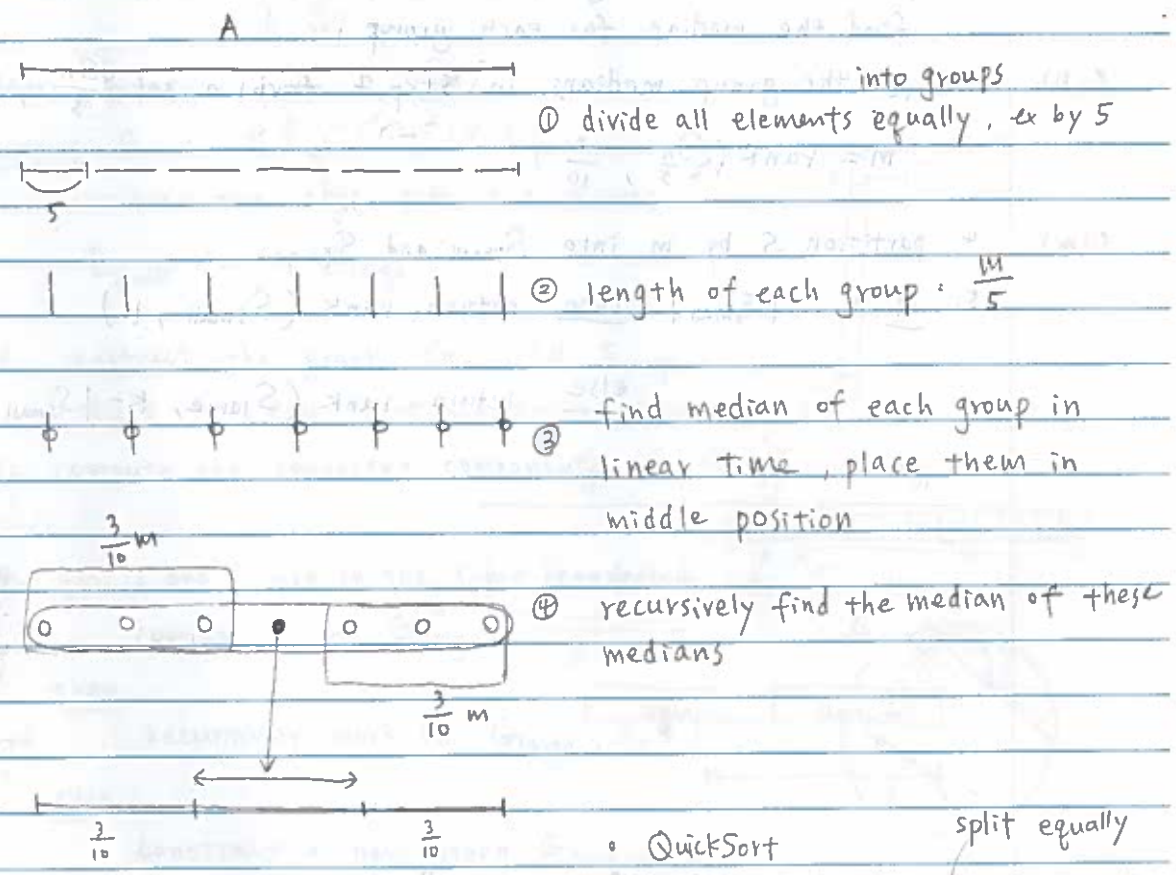
cut half of edges

Q: How to cut & get half of smaller edges in linear time?

Deterministic Selection

Given an array A[1, ..., m], how to split A[1, ..., m] into smaller half and larger half in linear time?

⇒ median finding — O(n)



① divide all elements equally, ex by 5 into groups

② length of each group: $\frac{m}{5}$

③ find median of each group in linear time, place them in middle position

④ recursively find the median of these medians

- QuickSort split equally
  choose pivot as median
  ⇒ O(nlogn)
  if not ⇒ $O(n^2)$

Oct 06 - 2015

rank (S, k) $T(n) = O(n)$

// return the k^{th} smallest in S

O(1) 1. if $|S| \le 10$, then work out directly, return;

O(n) 2. partition S into groups of 5 elements, find the median for each group;

O(n) 3. let the group medians in Step 2 form a set $S_{\frac{n}{5}}$;

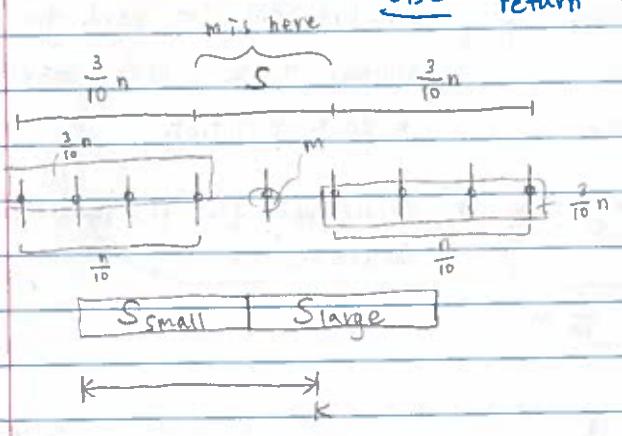
$m = \text{rank}(S_{\frac{n}{5}}, \frac{n}{10})$;

O(n) 4. partition S by m into S_{small} and S_{large}

5. if $k \le |S_{small}|$ then return rank (S_{small}, k)

else return rank $(S_{large}, k - |S_{small}|)$

$|S_{small}|, |S_{large}| \le \frac{7}{10}n$



$T(n) = O(n) + T(\frac{n}{5}) + T(\frac{7}{10}n)$

$\frac{n}{5} + \frac{7}{10}n = \frac{9}{10}n$ ↑ get rid of $\frac{1}{10}n$ elements in each iteration

$T(1) = O(1) \le c_1 \quad n \le 10$

$\Rightarrow T(n) \le c_1 n + T(\frac{n}{5}) + T(\frac{7}{10}n)$

guess $T(n) \le 10c_1 n$, $n \ge 1$

$1 \le n \le 10$, $T(n) \le 10c_1 n$

$T(n) \le cn + T(\frac{n}{5}) + T(\frac{7n}{10})$

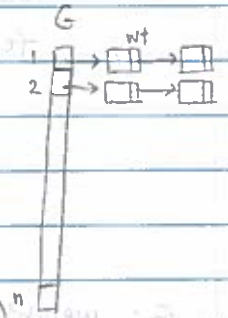
$\le c_1 n + 10c_1(\frac{n}{5}) + 10c_1(\frac{7n}{10})$

$= 10c_1 n$

Max bandwidth path $T(m)$

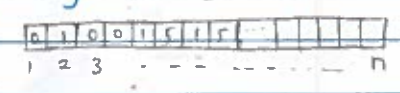
time input G, s, t (G = (V, E))

O(m) 1. partition the edges into two halves E_{small} and E_{large};



O(m) 2. construct the graph $G_{large}(V, E_{large})$ (delete the edges that are smaller than median)

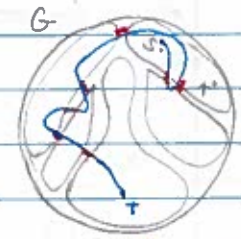
O(m) 3. compute the connected components of G_{large} (using DFS) O(max)



O(m) 4. if s and t are in the same connected component of G_{large},

then

recursively work on (G_{large}, s, t)



else

construct a new graph G_{new}, whose vertices are connected components of G_{large}, for two pieces in G_{large}, pick the largest capacity edge between them.

G_{new}

Let s' and t' in G_{new} contain s and t respectively

recursively work on (G_{new}, s', t')

Convert the s'-t' path in G_{new} into an s-t path in G

$\Rightarrow T(m) = O(m) + T(\frac{m}{2})$ assume $T(m) \le 2cm$

$\le cm + T(\frac{m}{2})$

$\le cm + 2c(\frac{m}{2})$

$= 2cm$

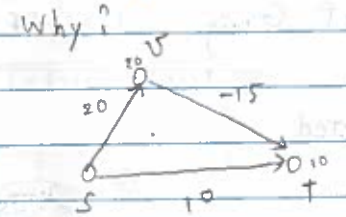
Oct. 8-2015 Dijkstra (G, s, t) (for shortest path)

1. make s in tree ;
   make all neighbors of s fringers ;
2. while there are fringes do
   pick the fringe v of min. wt ;
   make v in-tree ;
   for each edge [v, w] do
   if w is unseen then

   if w is fringe & then
   if w is in-tree then ← modification for negative weigh

G : weights are nonnegative !

Why ?



Why we can not find the shortest path by using Dijkstra if the weight of edges are negative ?

It only considers the path in tree.

Suppose a algo can find MST only for positive edges

Suppose A_1 : algorithm for MST with positive edges
G : with negative edges
1. add a number to each edge weight, giving G'
2. call A_1 on G' that gives T
3. T is a MST for G

Why algo. A_1 can not apply for finding shortest path of graph G with negative weight ?

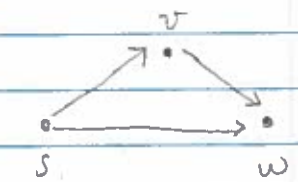
⇒ The # of edges for shortest path will vary.

MST always has same # of edges
if there are n vertices, there are n-1 edges for MST.
(n-1 = m)

Modified One :

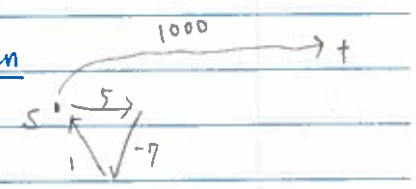
Dijkstra (G, s, t) (for shortest path)
1. make s in-tree
   make all neighbors of s fringers ;
   for each edge [v, w] do
   if you can improve wt(w)
   improve it.

Bellman-Ford
O(nm)
• Assume no negative cycles

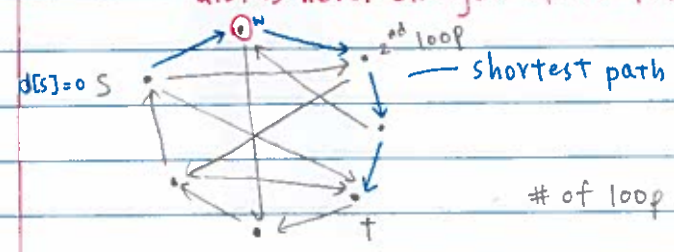


1. for v = 1 to n do
   dist[v] = +∞ ;
2. dist[s] = 0 ;
3. for each edge [v, w] do
   if dist[w] > dist[v] + wt[v, w] then
   dist[w] = dist[v] + wt[v, w] ;
   Dad[w] = v ;

n-1 loop ??? times — loop until no change



dist[w]
~~dist is never changed after first loop~~



# of loop ≤ length of shortest path
≤ n-1

4. for each edge [v, w] do
   if dist[w] > dist(v) + wt[v, w] then
   stop ('negative cycles')
5. return (Dad[·])

Formal Proof

if no negative cycles, (it) makes sense.
graph has

the shortest path

① a path in the cycle wouldn't be shorter than a path with cycle removed

② we only have to consider simple paths

③ each simple path has at most n-1

④ # of simple paths $\leq n! \cdot n$ ← finite #

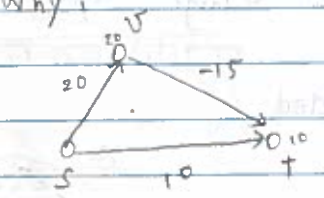
⑤ cut a vertex

Oct. 8-2015 Dijkstra (G, s, t) (for shortest path)

1. make s in tree;
   make all neighbors of s fringers;
2. while there are fringes do
   pick the fringe v of min. wt;
   make v in-tree;
   for each edge [v, w] do
   if w is unseen then
   if w is fringe & then
   if w is in-tree then ← modification for negative weight

G: weights are nonnegative!

Why?



Q: Why we can not find the shortest path by using Dijkstra if the weight of edges are negative?

A: It only considers the path in tree.

Suppose a algo can find MST only for positive edges

Suppose A_1: algorithm for MST with positive edges
G: with negative edges
1. add a number to each edge weight, giving G'
2. call A_1 on G' that gives T
3. T is a MST for G.

Why algo. A_1 cannot apply for finding shortest path of graph G with negative weight?

⇒ The # of edges for shortest path will vary.

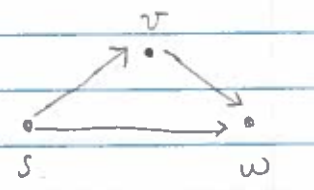
- MST always has fixed same # of edges
  if there are n vertices, there are n-1 (= m) edges for MST.

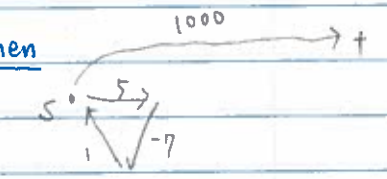
Modified One =

Dijkstra (G, s, t) (for shortest path)
1. make s in-tree
   make all neighbors of s fringers;
   for each edge [v, w] do
   if you can improve wt(w)
   improve it.

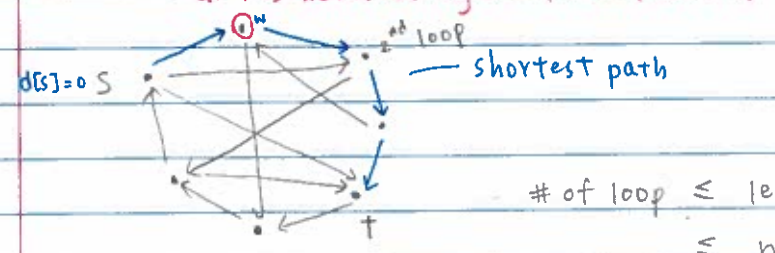
Bellman-Ford
O(nm)
- Assume no negative cycles

1. for v = 1 to n do
   dist[v] = +∞;
2. dist[s] = 0;
   loop ?? times → loop until no change
3. for each edge [v, w] do
   if dist[w] > dist[v] + wt[v, w] then
   dist[w] = dist[v] + wt[v, w];
   Dad[w] = v;





dist[w]
dist is never changed after first loop



shortest path

# of loop ≤ length of shortest path
≤ n-1

4. for each edge [v, w] do
   if dist[w] > dist(v) + wt[v, w] then
   stop ('negative cycles')

5. return (Dad[

Formal Proof — the shortest path

if no negative cycles, (it) makes sense.
graph has

① a path in the cycle wouldn't be shorter than a path with cycle removed

② We only have to consider simple paths

③ each simple path has at most n-1 edges

④ # of simple paths $\leq \underbrace{n!}_{\text{n node's ordering}} \cdot \underbrace{n}_{\text{a path has at most n nodes from S to T}}$ ← finite #

⑤ cut a vertex

~~permutation~~

Bellman-Ford Algo.

Given a weighted, directed graph $G = (V, E)$ with source s and weight function $w: E \to \mathbb{R}$, the Bellman-Ford algorithm returns a boolean value indicating whether or not there is a negative-weight cycle that is reachable from the source. If there is such a cycle, the algo. indicates that no solution exists. If there is no such cycle, the algo. produces the shortest paths and their weights.

Oct 13 2015 **Shortest Paths in graphs with negative edges.**

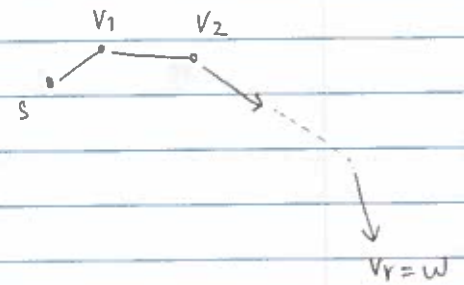
The shortest path problem is meaningless if there are negative cycles. (and only if)

If there is a shortest path from s to w.

$s = v_0, v_1, \dots, v_r = w$

then after i loops in step 3

$$\underset{\text{dist}}{wt}[v_i] = \sum_{j=1}^{i} [v_{j-1}, v_j]$$



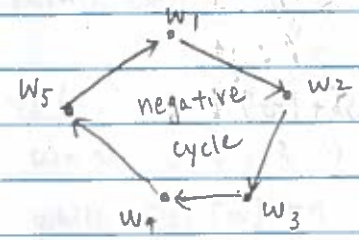
Bellman-Ford (G, s, t) $O(mn)$

1. for each vertex v do
   $\underset{\text{dist}}{wt}[v] = +\infty$;
2. $\underset{\text{dist}}{wt}[s] = 0$;
3. loop $n-1$ times (at most $n-1$ time, the algo can stop early at the time that no more dist[ ] change for all nodes.)
   for each edge $[v, w]$ do (m times)
   if $\underset{\text{dist}}{wt}[w] > \underset{\text{dist}}{wt}[v] + \text{weight}[v, w]$
   then $\underset{\text{dist}}{wt}[w] = \underset{\text{dist}}{wt}[v] + \text{weight}[v, w]$
   $\text{Dad}[w] = v$;
4. for each edge $[v, w]$ do
   if $\underset{\text{dist}}{wt}[w] > \underset{\text{dist}}{wt}[v] + \text{weight}[v, w]$
   then stop ('negative cycle')

   ← give another loop i.e., n^{th} time, if the dist[ ] of every node doesn't change, then the graph doesn't have negative cycle.
5. return (Dad[·])

If there is a negative cycle, then the algo. always reports this fact at step 4.

Suppose there is a negative cycle



$wt[w_1, w_2] + wt[w_2, w_3] + wt[w_3, w_4] + wt[w_4, w_5] + wt[w_5, w_1] < 0$

Claim: Suppose the algo does not report on step 4.

$dist[w_2] \le dist[w_1] + wt[w_1, w_2]$

$dist[w_3] \le dist[w_2] + wt[w_2, w_3]$

$dist[w_4] \le dist[w_3] + wt[w_3, w_4]$

$dist[w_5] \le dist[w_4] + wt[w_4, w_5]$

$+)\quad dist[w_1] \le dist[w_5] + wt[w_5, w_1]$

$0 \le wt[\ ] + \dots$

which contradicts to the claim

Kruskal (for Minimum Spanning Tree, MST)

1. sort the edges in nondecreasing (increasing) order;
   $e_1, e_2, \ldots, e_m$

2. $T = \phi$;

3. for $v = 1$ to n do MakeSet(v) ;

4. for $i = 1$ to m do
   $e_i = [v_i, w_i]$;
   $r_1 = Find(v_i)$;
   $r_2 = Find(w_i)$;
   if $r_1 \neq r_2$ then
   $T = T \cup \{e_i\}$;
   UNION(r_1, r_2) ;

Dad[1, ..., n] Dad[i] = father of i

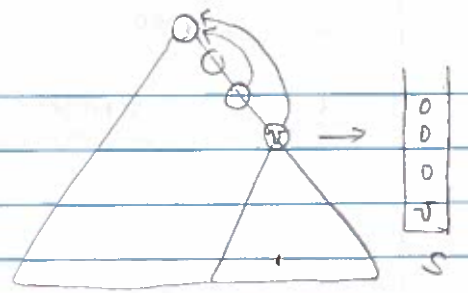
rank[1, ..., n] rank[i] = height of the tree rooted at i
rank[v] ≤ log n

MakeSet(v) O(1)
Dad[v] = 0 ; rank[v] = 0 ;

UNION(r_1, r_2) O(1)
case:
1. rank[r_1] > rank[r_2]
   Dad[r_2] = r_1
2. rank[r_1] < rank[r_2]
   Dad[r_1] = r_2
3. rank[r_1] = rank[r_2]
   Dad[r_2] = r_1 ; rank[r_1]++

union by ranks

FIND(v) O(log n)
1. w = v
2. while Dad[w] ≠ 0 do
   w = Dad[w];
3. return [w]

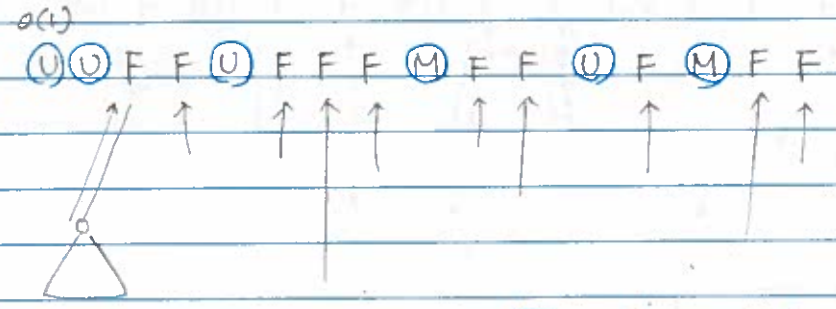


FIND(v)
1. w = v ; S = φ ;
2. while Dad[w] ≠ 0 do
   Stack → (S) ← w ;
   w = Dad[w];
3. while S ≠ φ do
   v ← (S) ; stack : LIFO
   Dad[v] = w
4. return [w]

find with path compression

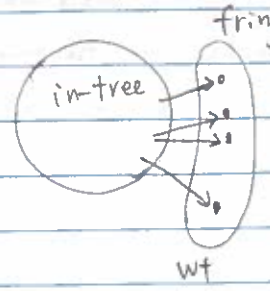
Given m U-F-M operations on a set $\{a_1, a_2, \ldots, a_n\}$.
How much time would these operations take?

≤ m log n, ≥ m

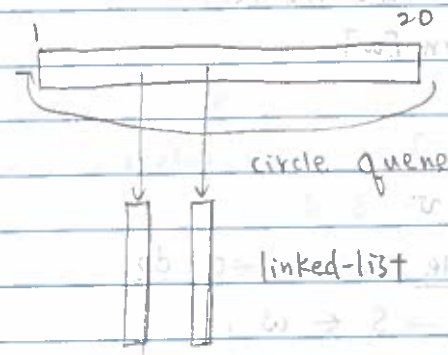


m + Time spent on all Finds = $\sum_{\text{find path } P_i}$ length(P_i)

edge wt ≤ 20



wt difference for fringes ≤ 20



Oct 20-2015

Make Set (a)

1. rank[a] = 0 ; Dad[a] = 0 ;

UNION (r_1, r_2)

Cases :

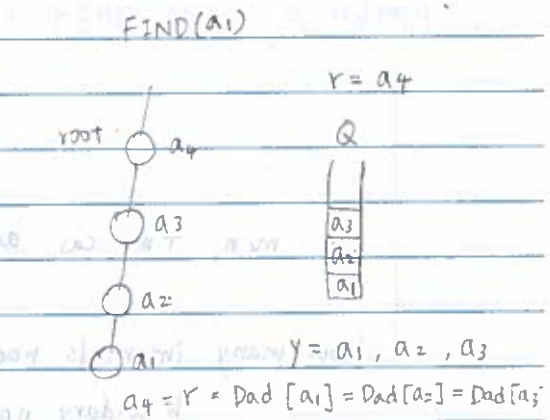
rank[r_1] > rank[r_2] : Dad[r_2] = r_1

rank[r_1] < rank[r_2] : Dad[r_1] = r_2

rank[r_1] = rank[r_2] : Dad[r_2] = r_1 ; rank[r_1] ++

FIND (a)

1. r = a ; Q = ϕ ;
2. while Dad[r] ≠ 0 do
   Q ← r ; r = Dad[r] ;
3. while Q ≠ ϕ do
   y ← Q ; Dad[y] = r
4. return (r)

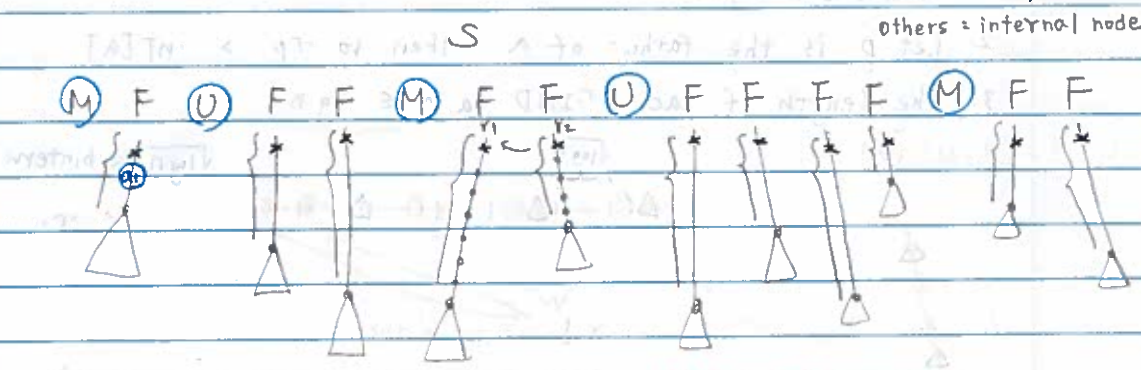


A sequence S of m M-U-F operations on $\{a_1, a_2, \dots, a_n\}$, find the complexity of S.

x : boundary node

others : internal nodes

S



Total Time taken by all FINDs ≈ $\sum$ length (P)

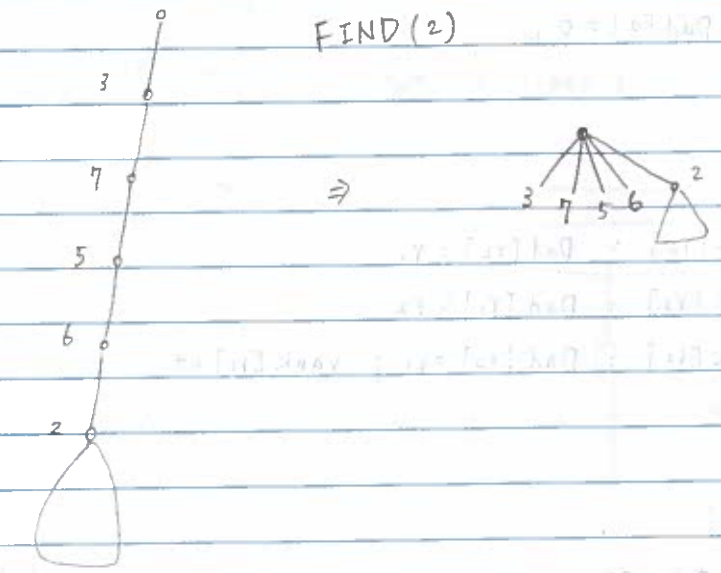
P : a FIND path

Assignment#3, Q#4

No FINDs before M & U, what's time complexity ?

$O(n+m)$

Ex



How many time can an element a_i be an internal node ? ≤ 1

How many internal nodes in the path ? n.

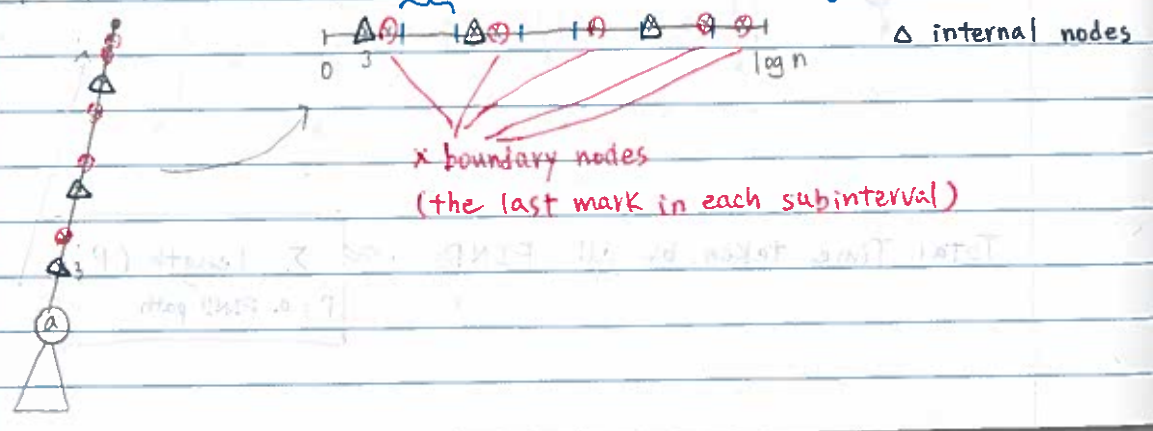
boundary nodes ? m

↑ No Union, no new roots

Facts

1. $rank[a] \leq \log n$
2. Let p is the father of a, then $rank[p] > rank[a]$
3. the length of each FIND path $\leq \log n$

FIND(a)



4. total # of boundary nodes $\leq m\sqrt{\log n}$

along all FIND paths

Fix an a_i, how many times can a_i be an internal node ?

After $\sqrt{\log n}$ times in which a_i is in a FIND path, a_i gets a father whose is rank $\geq \sqrt{\log n} + rank(a_i)$

↑ a_i becomes a boundary-node

5. Total # of internal nodes along all FIND paths $\leq n\sqrt{\log n}$

Oct 22 2015

MakeSet (x)

1. rank[x] = 0 ;
2. Dad[x] = 0 ;

UNION (r_1, r_2)

rank[r_1] > rank[r_2] : Dad[r_2] = r_1

rank[r_1] < rank[r_2] : Dad[r_1] = r_2

rank[r_1] = rank[r_2] : Dad[r_2] = r_1 , rank[r_1] ++

FIND (x)

1. w = x , S = ϕ ;
2. while Dad[w] $\neq$ 0 do
   S $\leftarrow$ w ; w = Dad[w] ;
3. while S $\neq \phi$ do
   v $\leftarrow$ S ; Dad[v] = w ;
4. return (w)

Given a sequence of m M-U-F operations on elements $\{a_1, a_2, \dots, a_n\}$.

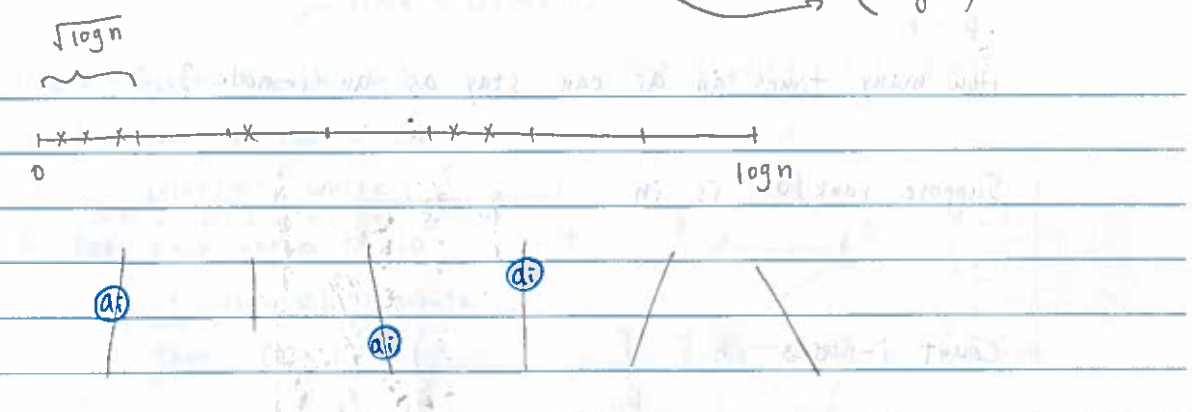
The time is proportional to

$$m + \sum_{p=\text{FIND path}} \text{length}(p) \qquad O((m+n)\sqrt{\log n}) \qquad O((m+n)\log^* n)$$

Facts :

1. rank[v] $\leq \log n$
2. rank of father > rank of son
3. for each FIND path p, the node ranks in p are distinct integers in [0 ... log n]
4. subinterval of length $\sqrt{\log n}$
5. b-nodes : the last in each subinterval
   i-nodes : not b-nodes
6. for each FIND path p, there are $\leq \sqrt{\log n}$ b-nodes $(\log^* n)$
7. each element a_i can be an i-node $\leq \sqrt{\log n}$ times $\leq m\sqrt{\log n}$ $(m \log^* n)$
8. total # of nodes on all FIND paths = total # of b-nodes + total # of i-nodes $\leq n\sqrt{\log n}$ $(2n \log^* n)$
   $\leq (m+n)\sqrt{\log n}$ $(m+2n)\log^* n$

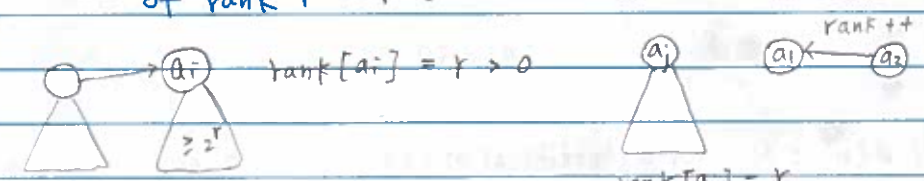
sub-int $\frac{(\log n)}{2} \rightarrow (\log n)^{\frac{2}{3}}$



$10\sqrt{\log n} \Rightarrow$ # b-nodes $\leq \frac{\sqrt{\log n}}{10}$

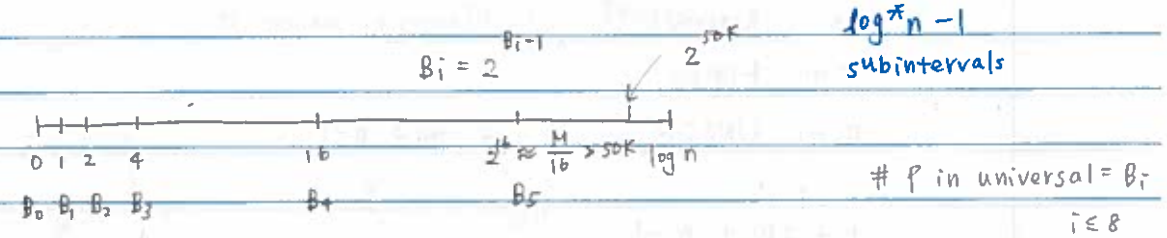
a_i as an i-nodes $\leq 10\sqrt{\log n}$

9. Lemma : for each rank value r, there are $\leq \frac{n}{2^r}$ nodes of rank r $r > 0$



Proof : (Induction)

$a_i \neq a_j$ do not share same decendants



$\log^* n = \min\{ i \mid 2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}} \} \geq n$ $\qquad 2^{2^{2^2}} = 2^{16}$

$\log^*(50k) = 5$

$$\frac{1}{2^{10}} + \frac{1}{2^{11}} + \frac{1}{2^{12}} + \cdots = \frac{1}{2^{10}} \times 2 = \frac{1}{2^9}$$

$B_i - r$

How many times an a_i can stay as an i-node ?

Suppose rank[a_i] is in $= r$

count i-nodes in

$$\sum_{r=B_{i-1}}^{B_i - 1} (B_i - r)\frac{n}{2^r} \le \sum_{r=B_{i-1}}^{B_i - 1} \frac{n B_i}{2^r} = n B_i \sum_{r=B_{i-1}}^{B_i} \frac{1}{2^r} \le n B_i \sum_{r=B_{i-1}}^{\infty} \frac{1}{2^r}$$

$$= n B_i \frac{2}{2^{B_{i-1}}} = 2n \qquad (2^{B_{i-1}} = B_i)$$

$\alpha(n) \le 7$ for all practical n

$\log^* n \le 8$

For Kruskal Algo. Sorting = $O(n \log n)$

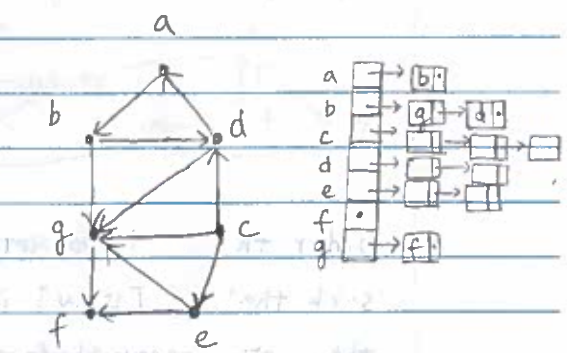
n	MakeSet	edges by weight
2m	FINDs	
m-1	UNION	$m \ge n-1$

$n + 2m + m - 1$

Oct-27-2015

Depth-First Search (G) — time = $O(m+n)$

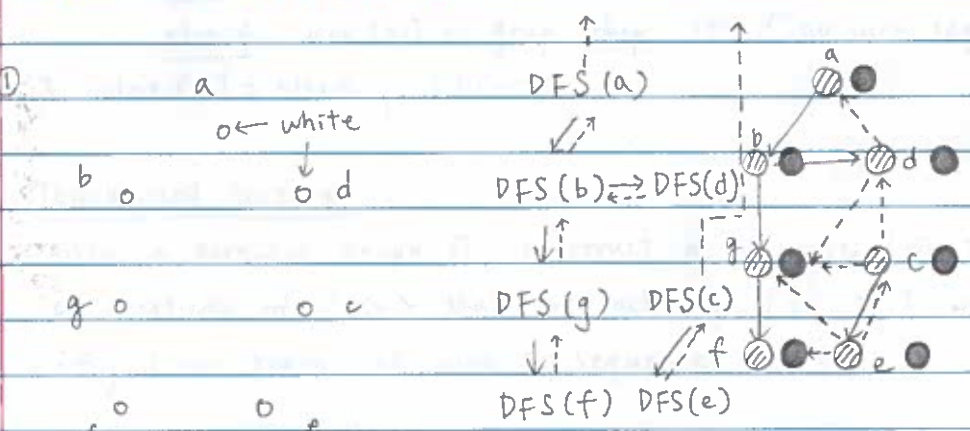
1. for each vertex v do
   color[v] = white ;
1' i = n ; A[1,...n] for output — for topological sorting
2. for each vertex v do
   if color[v] == white
   then DFS(v)

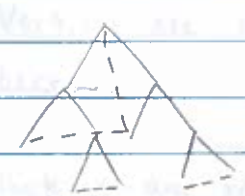


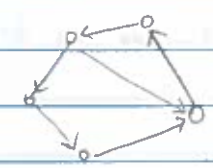
DFS(v)

1. color[v] = gray ;
2. for each edge [v, w] do — work for both undirected & directed graph
   if color[w] == white
   then DFS(w) ; else if color[v] == gray
3. color[v] = black A[i--]=v then return ('cycle')

step ①

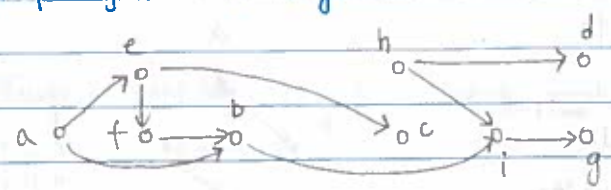






cycle ?

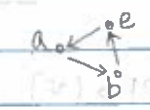
Topological Sorting



directed graph

a e f b c h d i g

order the vertices into a sequence S such that if [v, w] is an edge, then v appear before w in S.



v becomes black, color[v] = black

color [w] ?

Oct-29-2015

* For Topological Sorting

Depth-First-Search O(m+n) ⌒ graph can be sparse or dense

\\ output : an array S[1...n]

1. for each vertex v do color[v]==white;

1'. i = n

2 for each vertex v do

  if color[v] == white then DFS(v)

3. output S[1...n]

DFS(v)

1. color[v] == gray;

2. for each edge [v, w] do

  if color[w] == white

  then DFS(w)

  elseif color[w] == gray then stop('no such sequence')

3 color[v] = black ; S[i--] = v

Complexity:

DFS(v) is porportional to # of edges going out from v

Topological Sorting

given a directed graph D, construct a sequence $\{v_1, v_2, \ldots, v_n\}$ of vertices of D such that for each edge $[v_i, v_j]$ we must have $i<j$ (or repeat no such a sequence exists)

Vertices are ordered in the reversed order in which they become black.

look at any edge [v, w]

consider DFS(v) ⟶ color[w] = ?

↓

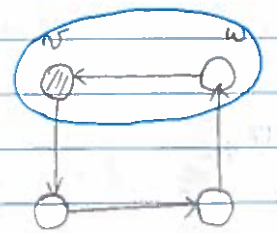
color[v] == gray

* black ✓

* white ✓

* gray → cycle

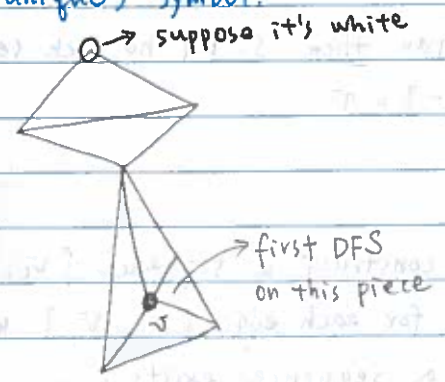
DFS(v) will examine this edge.



DFS(w)
↓
DFS(u_1)
↓
DFS(u_2)
↓
DFS(u)

Connected Component

given an undirected graph G, label the vertices so that all vertices in a connected component are labeled by the same (& unique) symbol.



DFS(v)
all vertice are white

when DFS(v) finishes
all become black

* For Connected Component

Depth- First- Search time = O(m+n)

\\ output cc[1 ... n]

1. for each vertex v do color[v] == white ;
1'. cc # = 0 ;
2 for each vertex v do
    if color[v] == white then cc# ++; DFS(v) ;
3 output cc[1 ... n]

DFS(v)

1. color[v] = gray ; cc[v] = cc # ;
2 for each edge [v, w] do
    if color[w] == white
    then DFS(w)

3 color[v] = black

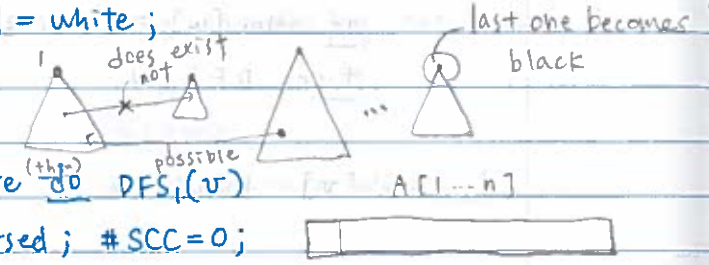
Strongly Connected Component (SCC)

an SCC of a directed graph is a maximal subgraph in which every vertex can reach every other vertex.

given a directed graph D, label the vertices so that all vertices in the SCC are labeled by the same & unique symbol

Algo-SCC(D) time = O(m+n)

1. for each v do color[v] = white;
2. i = n;
3. for each v do
   if color[v] == white (then) do $DFS_1(v)$
4. G_R = G with edge reversed; #SCC = 0;
5. for each vertex v do
   color[v] = white; // all vertices that become black after this call are exactly those that are in the same SCC with A[1]
6. $DFS_2(A[1])$



$DFS_1(v)$

1. color[v] = gray;
2. for each edge [v,w] do
   if color[w] == white
   then DFS(w)
3. color[w] = black; A[i--] = v

6. for i = 1 to n do
   if color[A[i]] == white
   then #SCC ++;
   $DFS_2(A[i])$

$DFS_2(v)$ // on G_R

1. color[v] = gray; SCC[v] = #SCC;
2. for each edge [v,w] do
   if color[w] == white
   then $DFS_2(w)$
3. color[v] = black

Nov-3-2015

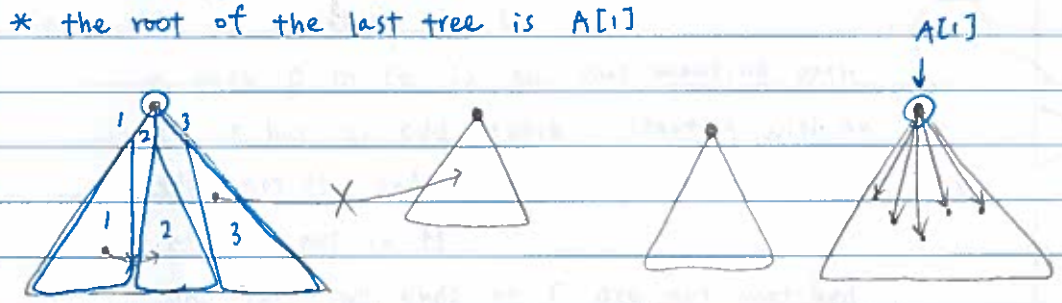
G = a directed graph

SCC = a max component in which vertices

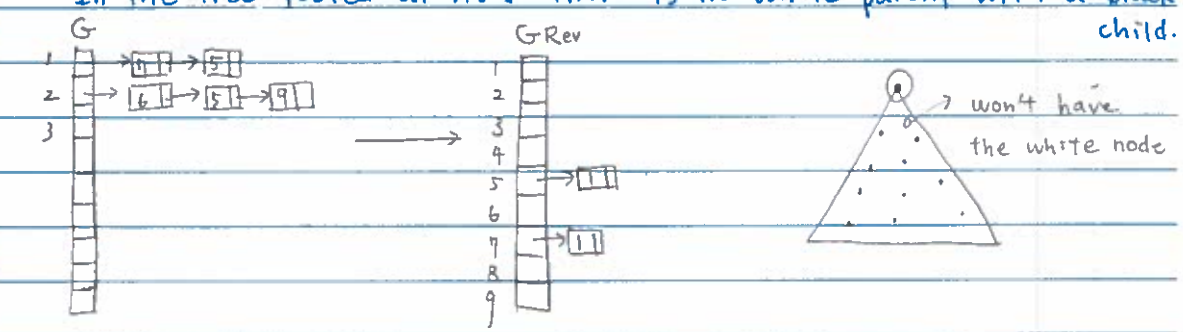
SCC's partition the vertices of G

Problem: given G, construct all SCC's.

* the DFS trees are constructed from left to right
* no edge goes from left to right
* the root of the last tree is A[1]



* no vertex that is not in the tree rooted at A[1] can be in the same SCC with A[1]
* every vertex in the same SCC with A[1] must be in the tree rooted at A[1]
* every vertex in that tree is reachable from A[1]
* exactly those vertices in that tree from which A[1] is reachable are in the same SCC with A[1]
* if we reverse the edges in G, then these vertices are exactly those reachable in DFS(A[1])
* In the tree rooted at A[1] there is no white parent with a black child.



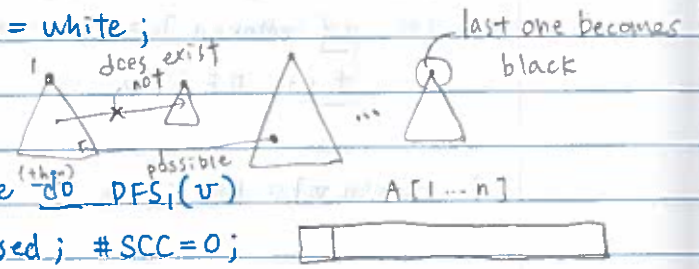
Strongly Connected Component (SCC)

an SCC of a directed graph is a maximal subgraph in which every vertex can reach every other vertex.

given a directed graph D, label the vertices so that all vertices in the SCC are labeled by the same & unique symbol

Algo-SCC(D) time = O(m+n)

1. for each v do color[v] = white;
2. i = n;
3. for each v do
   if color[v] == white (then) do DFS$_1$(v) possible
4. G_R = G with edge reversed; #SCC = 0;
5. for each vertex v do
   color[v] = white; // all vertices that become black after this call are exactly those that are in the same SCC with A[i]
6. DFS$_2$(A[i])



DFS$_1$(v)

1. color[v] = gray;
2. for each edge [v,w] do
   if color[w] == white
   then DFS(w)
3. color[w] = black; A[i--] = v

6. for i = 1 to n do
   if color[A[i]] == white
   then #SCC ++;
   DFS$_2$(A[i])

DFS$_2$(v) // on G_R

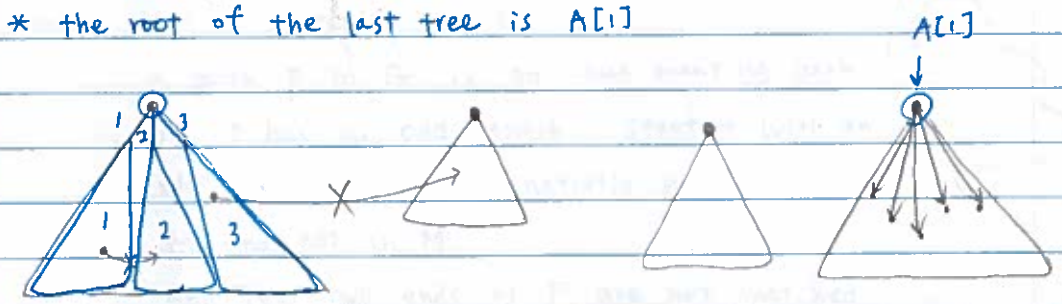
1. color[v] = gray; SCC[v] = #SCC;
2. for each edge [v,w] do
   if color[w] == white
   then DFS$_2$(w)
3. color[v] = black

Nov-3-2015

G = a directed graph

- SCC = a max component in which vertices are bi-connected
- SCC's partition the vertices of G
- Problem: given G, construct all SCC's.

* the DFS trees are constructed from left to right
* no edge goes from left to right
* the root of the last tree is A[1]



* no vertex that is not in the tree rooted at A[1] can be in the same SCC with A[1]
* every vertex in the same SCC with A[1] must be in the tree rooted at A[1]
* every vertex in that tree is reachable from A[1]
* exactly those vertices in that tree from which A[1] is reachable are in the same SCC with A[1]
* if we reverse the edges in G, then these vertices are exactly those reachable in DFS(A[1])
* In the tree rooted at A[1] there is no white parent with a black child.