CS 513 Data Structure and Algorithm

Martin Farach-Colton

Core 313 Friday 10:30-11:30

TA: Vishvajeet N [vishva.jeet@rutgers.edu](mailto:vishva.jeet@rutgers.edu)

# September 8, 2017

Maximum Subvector Sum

-10 -20 10 5 -3 -2 -1 6 4 -3 -16 32 12 -1

Output (12,13)

-10 -20 10 5 -3 -2 -1 6 4 -3 -16 -32 12 -1

Output (3,9)

Current Max

0 0 10 15 12 10 9 15 19 16 0 0 12 11

Def we say f(n) is O(g(n)) if there exists C, n0 that for any n>n0, f(n)<=cg(n)

Eg Def f(n) is O(g(n)) if lim f(n)/g(n) <=c, for some constant c

F(n)=3123n^3+17.5n^2-321n+1077.98

Find g(n) that f(n)=O(g(n))

g(n^4) can be the answer where c=0

Def f(n) is Ω(g(n)) if g(n) is O(f(n)) Ωgreater than, big-O smaller than

The big O notation has nothing to do with algorithms. Only with funcitons.

Def f(n) is Θ(g(n)) if f(n) is O(g(n)) and f(n) is Ω(g(n)) or lim f(n)/g(n)=c>0

L’Hosital’s Rule

Lim f(n)/g(n)=lim f’(n)/g’(n) (also true when n->0)

Product rule when taking derivatives

Multiplying Integers

X\*Y n bits

Long multiplication is O(n^2)

X-A\*2^(n/2)+B

B is low order bits (latter half of bits)

A is high order bits (first half of bits)

Y=C\*2^(n/2)+B

X\*Y=AC\*2^n+(AD+BC)\*2^(n/2)+BD plus a base case to form a recursive

T(n)=4T(n/2)+O(n)

Shifting, adding

Master’s theorem

The recurrence T(1)=c

T(n)=aT(n/b)+df(n)

For multiplicative f is

Θ(n^()) if a>f(b)

Θ(f(n)) If a<f(b)

Θ(n^() logn) If a=f(b)

Def f is multiplicative if f(ab)=f(a)f(b) Ex. F(n)=2n is not, f(ab)=2ab!=f(a)f(b) . f(n)=n is

In master’s theorem f(n) is multiplicative thanks to constant d

The recursive equation running time is O(n^2)

(A+B)(C+D)-AC-BD=AD+BC identity (no longer need to consider AD BC multiply)

The equation was simplified to (3 multiply instead of 4)

T(n)=3T(n/2)+O(n)

a=3 f(b)=2 O(n^(log 3))

Sorting

Bubble sort: running time n(n-1)/2 O(n^2)

Below n^2 and above n so Ω(n)

Worst case W(n)=Θ(n^2) a tight bound

(1). Worst-case running time

Let Dn={all inputs of size n}

For I in Dn, let T(I) be there running time on input I

Then worst case running time w(n)=max {T(I)} I in Dn

(2). Average-case

Let p(I)= probability of input I being given

Then A(n)= weighted average

Dn= set of permutations

|Dn|=n! (size of Dn)

p(I)=1/n!

An inversion in a permutation is a pair that is out of order

5,3,1,2,4

Inversion: 5,3

5,1

5,2

5,4

3,1

3,2

Every swap in bubble sort removes exactly 1 inversion

Average running time of bubble sort >= average # of inversion in a uniformly selected permutation

Comparison + swap swap only

4,2,1,3,5

# of inversion in (π) + # of inversion in () = (R means reverse)

Because every pair is an inversion in π or (not both)

Average # inversion in a permutation is /2

A(n)=Ω(n^2) at least lower bound, because is n^2

But w(n)=Θ(n^2) so A(n)=Θ(n^2)

Sorting algorithms

1. Comparison based vs not

Comparison based: Merge quick bubble insertion

Non-comparison:

Micheal Fragmon rumble memory fusion trees

1. In place =O(1) extra memory (besides the input array)

Bubble in place

Quick sort recursive cannot be in place because you are using the stack

How deep in quick sort can your recursive stack be ? O(log n)

1. Stable = you do not change the order of repeated elements ()

Merge sort (stable if when you are merging always choose the left array in a tie)

Quick sort (not stable partition)

A=[a1,a2,a3…..an]

(a1,1)(a2,2) if the first field is the same, check the second field, making it stable

1. Internal RAM vs External Disk

(Sorting data may be too big for RAM)

Next lecture:

Proof of lower bound of comparison-based sorting is nlogn

# September 15, 2017

Disk Access Model (DAM)

CPU 🡪 Memory M array

B

Reads Size B I/O each time to memory

I/Os are 5 to 6 orders of magnitude slower than instructions.

Since I/Os are expensive

Compression is the general thing with more CPU and less I/Os

Analyze Merge sort in DAM:

I/Os to merge two arrays of size N : O(N/B)

N divided by B (every I/O fetch B)

T(N)=2T(N/2)+N/B

T(M)=M/B

B is not a constant. So can’t apply master theorem. For some case, N, B, M can go infinity. None of them can be taken as constant.

Solve in Tree: N->N/2->N/4->……->N/B == M

Each level is N/B

Height is logN- logM = log(N/M)

Total is O(N/B\*log(N/M))

How to optimize this algorithm?

Do K-way Merge sort.

T(N)=KT(N/K)+N/B is the same for some K, need to be able to fit K+1 blocks in memory

Tree: N🡪 N/K🡪N/K^2🡪N/K^3🡪…..M

O()

K<=M/B fits K+1 blocks in memory

O()

It’s so ugly but we cannot do better.

This is the correct algorithm for sorting in external memory.

2-way merge sort is optimum in external sorting

Nothing has one level of memory , call cache

Higher way merge sort

Word-RAM\_ operations on words take O(1) time and words have w bits

What do we know about the size of w?

From theoretical point of view, you think your word size should not increase with your data size.

A[i]=j should be constant time O(1). Then how many bits in word?

How big can A be? Say the length of A has n items

How do I find the ith element in an array?

This is actually base A+i and multiply the size of the item.

How many bits to distinguish between N items? Log N

w>= log n

In constant time when w is at least log n bits.

Non-comparison based

Bucket sort Sort array A drawn from a universe u=[0,u-1}=[u-1]

Not in place, but stable

B[A[i]]

u aray tree ???

Runtime O(n+u) that does not mean sort in linear time. If you know u is small, you could use this algorithm instead of merge sort.

Radix sort

Stable sort by least significant digit

If we do in bucket sort. If there is K digit, each digit has [u-1]. Then it’s n by K array. O(K(n+u))

1 digit number in base u

2 digit number in base then you can represent any number in u

For example, to represent 16 bits we need 4 bits.

13 is D in base 16, 4 bits for each digit???

13 is 31 in base 4, 2 bits 2 digits???

2 digit radix sort of A takes O(n+)

3 digit radix sort O(n+) or O(n+)

…

K digit radix sort O(K(n+))

Convert binary to base 8 just chocking out certain bits and group them

Claim: sorting n elements in the range [], for constant c is O(n) time

Suppose I have a directed graph with n nodes, want to sort the nodes by the product of their in-degrees and out-degrees. How long does it take?

Van Emde Boas Tree (VEB tree)

Maintain a set typically in O(log n) time, say skip list

Insert(k,s)

Del(k,s)

Succ(k,s)

Pred(k,s)

We can do these in log log u time.

How many bits in a number of u ? log u

How many numbers can we have?

T(u)=T()+1 universe of plus a constant time of other things

T(u)=log log u

Suppose I can allocate space u in constant time. Now I have a table of size u

I can’t usually use the memory

I can go to the memory but anything can be there.

How can we allocate a huge amount of memory and safe to use it. We should know whether it’s something I wrote or it’s been there before? Or say whether it’s a safe position to read.

Write in the memory arbitrarily.

Next lecture:

Lower bound of external sorting.

Fusion trees???

# September 22, 2017 Van Emde Boas Tree

Van Emde Boas Tree

Maintain u=[u] set of integers upto u, under

Insert(x,s) O(loglog u)

Delect(x,s) O(logw)

Pred(x,s) min{y|y}

Succ(x,s)

Data payload and link data payload and link

Start out with nothing written in the array.

Log u=w bits

T(u)=T()+1 T’(w)=T’()+1

O(loglog u) O(log w)

First midterm for Van Emde Boas Tree (No Wikipedia allow for notation)

Say High(w)-> high order bits

W.high

Low(w)-> low order bits

W.low

W

W=1001

w.h=10 we shift it to the right and we got higher order

w.low=01 we may reserve order shift it to left and shift it back

Both in constant time

A=

We are not losing bits. It’s all widgets all the way down

Insert(x,w)

Insert(x.low,A[w.high])

Constant time

Succ(x,w) how do we find a successor or a number?

Successor of the low order bit in the widget, or the next non-empty widget

If

Combination Y=<x.high,Succ(x.low,A[x.high])>

The successor of the low order bit in appropriate widget. Keep high order bits, and replace the successor the low order bits. If the successor agrees in the next half of bits.

If it fails, Y not defined, then I need to find Z=next non-empty widget in A , return <Z,succ(Z,w)>

Specified by the minimum thing in there.

Use Van Emde Boas Tree This whole thing W(u) you are allowed to maintain integers up to u

Non empty

Each allows you to maintain integers up to

Insert(x.low,A[x.high])

Insert(x.high,nonEmpty)

But we are only allowed to recurse once. T(u)=T()+1 or it would be too much

How do we maintain it ?

If we insert the first one in Insert(x.high,nonEmpty), we have to store something in that widget with a constant amount of work.

We are going to store a Min and Max

How do you insert into a widget?

Insert(x, w)

If w.min not defined (aka W is empty) //not resursively

w.min=w.max=x return

If w.min=w.max, // not recursively

w.min=Min(x, w.min)

w.max=Max(x, w.max) return

If x<w.min

Swap(x, w.min) // same for w.max

// Guaranteed that x is between min and max

// If (A[x.high].min is defined) we will do it true or not //it’s already not empty, takes constant time

Insert(x.low, A[x.high]) //If we do for first, takes constant time

If(A[x.high].min=A[x.high].max) // this recursion only takes constant time

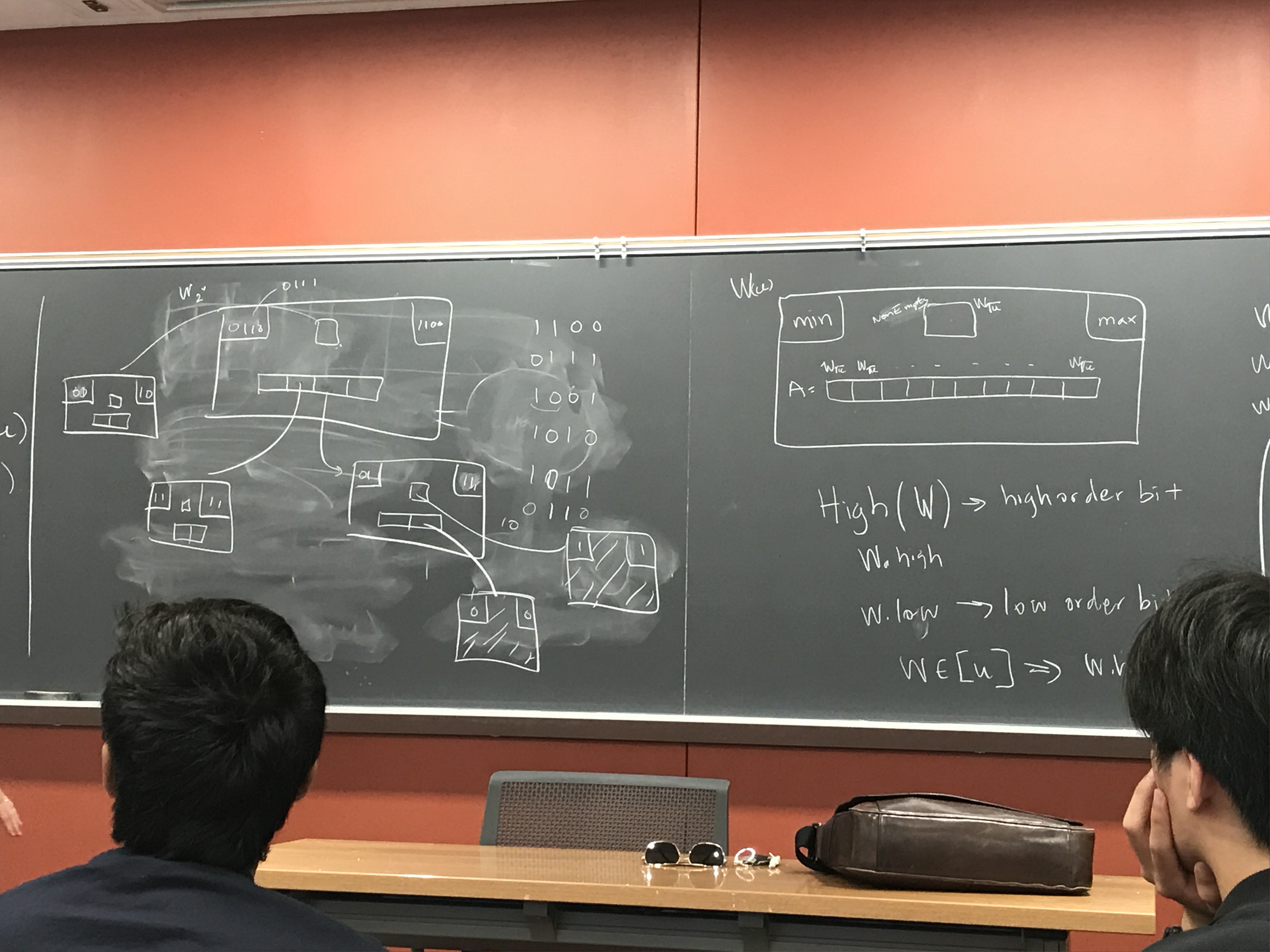
Insert(x.high,nonEmpty)

By keeping min, and max we are only doing one recursion.

We are doing recursively on high orders and and low order bit

Successors use max&min, precede use min

“Data structures are frozen algorithms”



Succ(x,w)

If (x<w.min)

Return w.min

If(x.low < A[x.high].max) //we know we will find a successor

Return <x.high, Succ(x.low,A[x.high])>

Y=Succ(x.high,nonEmpty) // if this barfed, return not defined. That’s what barf for

//Return<Y,A[Y].min>+ error checking

Or: If Y not defined

If x<x.max, return w.max

Else

Barf //a Barf for the recursive call, return not defind

Return<Y,A[Y].min>

Del(x,w)

If x=w.min=w.max, undefined w.min, w.max return

If nonEmpty is empty, fix x.min=x.max is needed

If x=w.min

Y= <nonempty.min, A[nonempty.min].min>

w.min=Y

Del(Y.low,A[nonEmpty.min])

If A[nonEmpty.min] is empty, Del(nonEmpty.min,nonEmpty)

//There would only be one recursive call of Del

If x=w.max, symmetric,

Del(x.low,A[x.high])

if A[x.high] is empty Del(x.high,nonEmpty)

# September 29, 2017

Fusion tree all VEB DPS in O()

VEB tree sorting

Insert each item in a vEMB tree nlogw

Repeated remove min element nlogw/O(nlogw)

A B tree takes I/Os

We pick a node, fetch that node, high branching factor so the tree is much shallower.

The lamest answer is B stands for binary.

X-fast trees (same time, less space, VEB-trees)

Trie out of bits (trie can be built out of a tree)

0111

1101

0110

For every level of the trie, have a pointer to the max and min below it. Build a doubly linked list at the each level. They all point to the last level.

To store these numbers, space:

O(nw) or O(nlogu) w=logu

Query’s next bit is 0

Min is successor

Successor, then left by 1 is predecessor

O(log w) to put and fetch

How long does it take to split a tree

Amortized insertion cost is O(logw+w/w)=O(log w)

Y-fast tree

f: u -> [n]

Pr{f(x)=f(y),x!=y} <= 1/n

If you want to hash n elements, how big does the table have to be so you expect no collisions? Choose k buckets

E[# of collisions] = (each of them )

2-level hash function

We do not expect a collision with quadratic

S has n elements position: K0 K1 ……Kn-1 there are

Ko^2/2 ……Kn-1^2/2 collisions here. Actual number of collisions is summation

f(x)=(ax+b) mod c

The expected # of collisions in the first level =

Summing up collisions in the bucket table.

m/B way merge sort is fastest

integer sorting no one has proved a lower bound.

## Lower bound for external memory sorting

General outline of the proof.

Min number of bits needed to specify answer divided by

Max number of bit is possible to learn by 1 I/O

I/Os of any sorting algorithm

# of orders of input? B words(elements)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

B! for the content of the block. How many bits do we need to distinguish between these guys? For K things, log K.

# of bits needed is log of

Max bits we learn from each I/O?

|  |  |
| --- | --- |
| M-B |  |
|  | B |

Only thing we learn is rank of B elements among those in memory

So we learn bits =O (B log )

Number of I/Os to sort is

# October 6, 2017

## Union-Find

Start sets : {1}{2}…{n}

Union(I,j) 🡪 union whichever set contains I with set that contains j

Find(i)🡪 retur I.d of set so that Find(i)=Find(j) iff i and j are in the same set

n=5 {1,2,3,4,5}

Union(1,2)

Union(2,3)

Union(1,3) Union(2,4)

Find(1)🡪1 Find(4)🡪1

Find(2)🡪1

Find(4)🡪4

|  |  |  |  |
| --- | --- | --- | --- |
| ID |  | Sets |  |
| 2 |  | 0 | ??? |
| 2 |  | 2 |  |
| 3 |  | 1 |  |
| 4 |  | 1 |  |
| 5 |  | 1 |  |

Find(i) return ID(i)

Union(I,j)

Move the shorter of Set(Find(i)), Set (Find(j)) onto the larger list

Each element that moves ends up in a list at least twice as long

Each element can move at most log n times,all unions returned take O(nlog n) time

f(n)=n-2

g(n)=f\*(n)=n/2

g\*(n)=log n

log\*(n)

how many time do you apply log before it is below 2

## Least Common Ancestors (LCA)

Linear time to pre-process , linear query time <O(n),O(n)>

Not balanced, arbitrary tree

a b a d c d e g e h i h j k j l j h e d a

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

RMQ (range minimum query)

Preprocess query A

RMQ(i,j) 🡪 position of min in A[i;j] 2-D array

Make a giant table, O(n^2) to fill in. <O(n^2),O(1)>

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 3 | 4 |
| 1 | 4 | 2 | 7 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 |
| 1 | 1 | 1 | 1 | 1 |
| 2 |  | 2 | 3 | 3 |
| 3 |  |  | 3 | 3 |
| 4 |  |  |  | 4 |

Starting position (n) width log n

1,2 2,3 3,4

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 1 | 3 | 3 | - |
| 4 | 1 | - | - | - |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

To fill 2^i entries, look at 2^i-1

slip query in size of 64 into 2 query in size of 32

<O(n log n), O(log)>

Fill in , query

<O(n log n), O(1)>

In a giant cell of 63, Look at 32 left aligned, and 32 right aligned.

a b a d c d e g e h i h j k j l j h e d a

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

Adjacent entries different by one

Solve RMQ (range minimum query) problems

We have an array A, break it into blocks of size log n /2

A

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |

Find min in each block and form A’

A’

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |

<O(n),O(1)> (substitute n/logn into < O(n log n), O(1)>)

NJ Turnpike (highway system)

Any query in A is at most 2 neighborhood queries in blocks/neighborhoods + at most 1 query in A’/ NJ Turnpike

A constant number of each

How long is NJTP? n/ logn bigger than linear

a b a d c d e g e h i h j k j l j h e d a

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

1 2 3 4 5 4 2 1

|  |  |  |
| --- | --- | --- |
| 1 | 1 | 1 |
|  | 2 | 3 |
|  |  | 3 |

121 🡪 010 type 2

232 🡪 010 type 2

343 🡪 010 type 2

454 🡪 010

565 🡪 010

654 🡪 0 -1 -2 type 0

323 🡪 0 -1 0 type 1

21

Normalize by tracking off the first

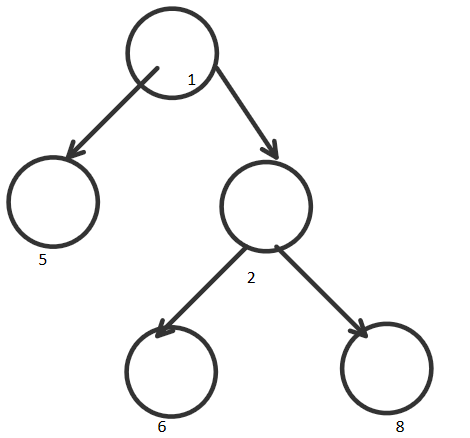
Each table is

There are possible tables, or

All these tables are much less than n

RMQ A=[5,1,7,6,2,8]

cartesian tree 6, 8, least common ancestor



subproblems into a table: Method of Four Russians

## Level ancestors

Pointer Doubling

<O(nlogn), O(logn)>

## Long path decomposition

Def the height of a node to be distance to the furthest leaf blow. Each node picks highest child

The tree then is decomposed into paths and each path can form an array

<O(N), O()>

## Double long path

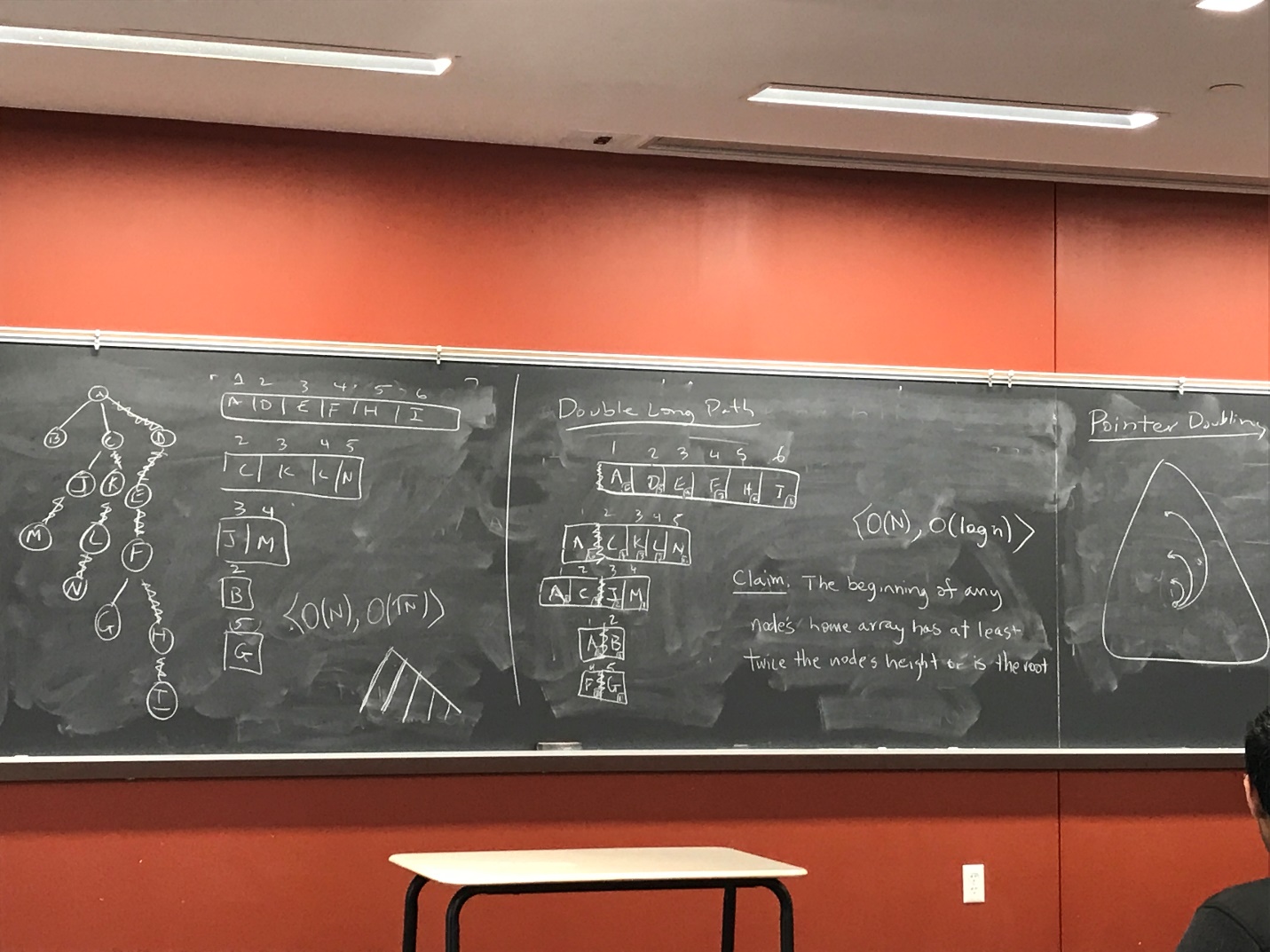
Increase these arrays up. But you cannot go higher than the root. Double the array but up to root.

<O(N), O(log n)>

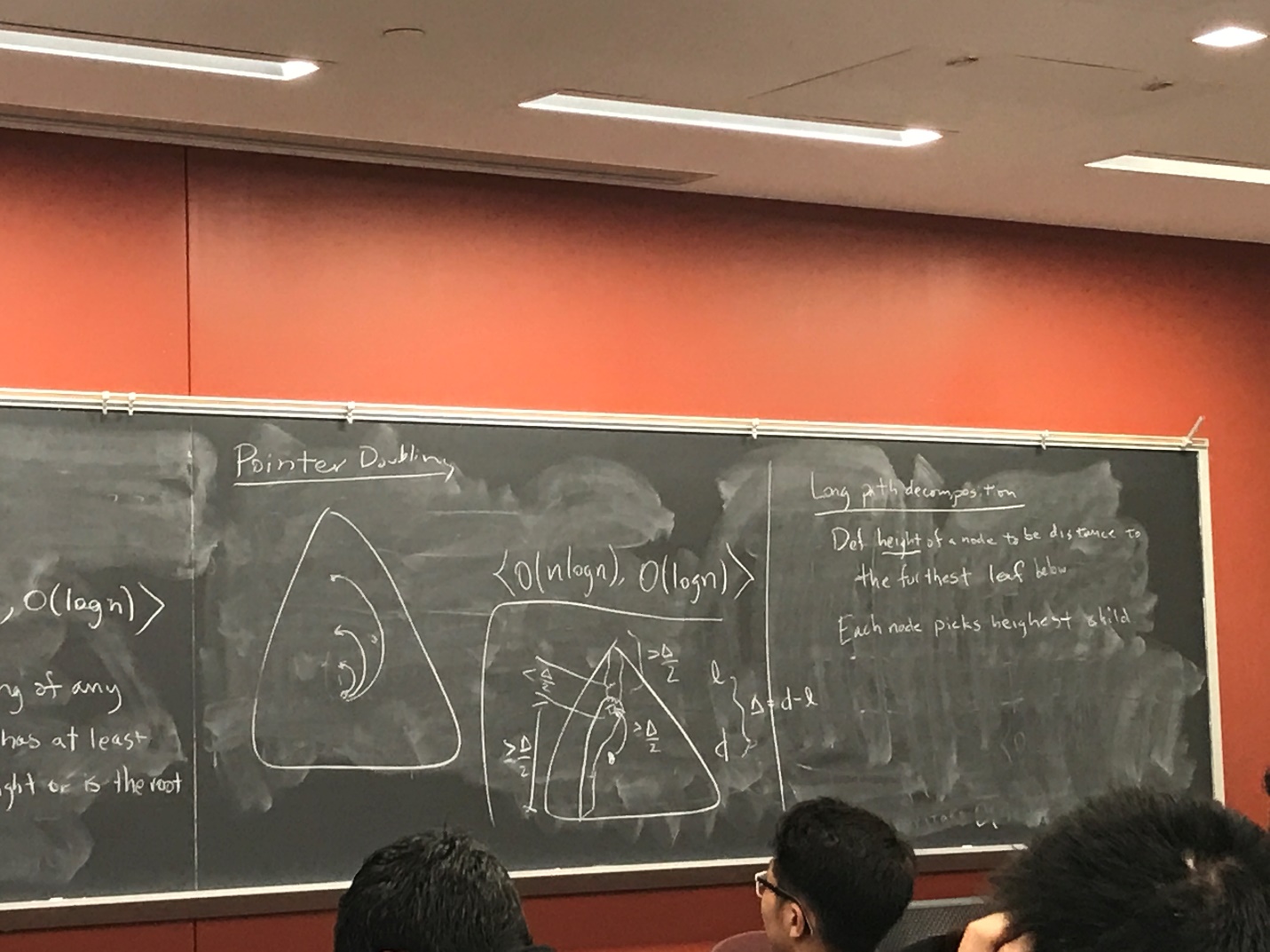
Each node appears only once in the right part of the array.

Claim: The beginning of any node’s home array has at least twice the node’s height or is the root.

Every time you double the height, and you can only double the height log n times.



## Pointer doubling



Log N/ 4

At most 4N / logn leaves

Linear time pre-processing in constant time query

# October 27, 2017

## Tree in minheap Order

Keys of children are 2 key of parent min at root

## Binomial Tree

B0 a node

B:: a root node of degree i

one child is a Bo, one a B1, …Bi-1

Claim: Two Bi can be merged into a Bi+1 in constant time.

Merge so min is still root

Claim: Bi has 2^i nodes

## Binomial heaps

Insert(v) O(logn) Insertion, create a Bo, meld

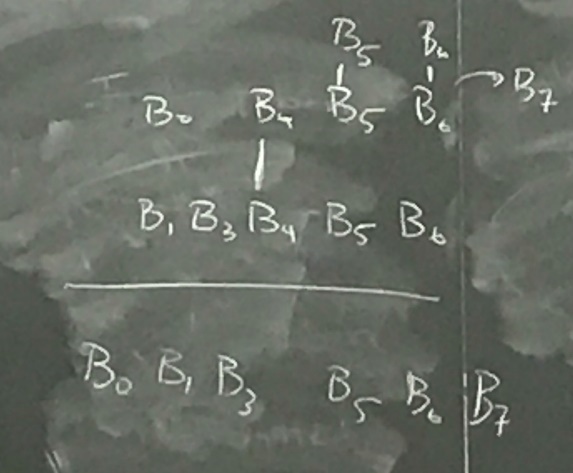
Delete-min O(logn) (extract-mim logn+ rearrange O(logn))

Meld (works just like binary addition) takes two heaps and combines them in 1, O(logn)

Collection of binomial trees in heap order, at most 1 per size

When we take two trees, the smaller one is up

Example:



Improvement: meld takes O(logn) at the cost of insertion and delete-mim. Binary heap takes linear time to merge.

## True Binomial Heaps

* Collection of binomial trees in heap order
* Any number of BTs per size
* Each tree has a merge credit

Insert--- make a new Bo no meld

Extract-min---lots of merges to clean up the mess

Insert----takes O(1) to gets charged a “Merge credit” for new tree/ total O(1) amortized.

Constant work to get rid of that tree. Each tree has a merge credit. Insertion pays for that.

n Bo trees.

Merge trees Bi + Bi 🡪 Bi+1 CPU cost O(1) constant time to merge two trees.

1 credit 1 credit 1 credit amortized cost of this merge is 0. We charge this operation

to a previous insertion.

Delete-min: we make up to logn new trees. They each need a merge credit.

logn time now(tree maybe be as big as log n)+ logn credits for

future merges

Amortized cost is: O(log n)

For any sequence of A insertions + B extract mins, the total cost is O(A+Blog n)

In average insertion is only constant.

Now add decrease key. The node may be smaller than its parent.

It would take log n time to do local swaps. But we want something constant.

So be min size of Bo

=1+++…+

=1+++…+

-=

Fibonacci grows exponentially. A Fibonacci tree is a binomial tree that is only allowed to lose 2 children

=+

This is the Fibonacci heap.

With Fibonacci heap, Prim and Dijstra’s all run in O(m+nlogn) ???

Next lecture: graph traversal. DFS/BFS. What kinds of edges in a graph. How to do thing like topological sort, strongly connected components.

## Depth-First Search

Put nodes on a stack as you traverse.

* Pop the stack and put unvisited neighbors onto stack.

## Breadth-First search

* Same with queue.

Tree edges:

* Edges of DFS/BFS
* Forward edge: non-tree edge to descendant
* Backward edge: edge to ancestor
* Cross edge: everything else

Undrected BFS: no forward,backward edges. Would have been traversed.

# November 3, 2017

## Connected Components –Maximal connect subgraph

For i=0 to n, if I not marked, CCDFS(i,i)

Node cc number

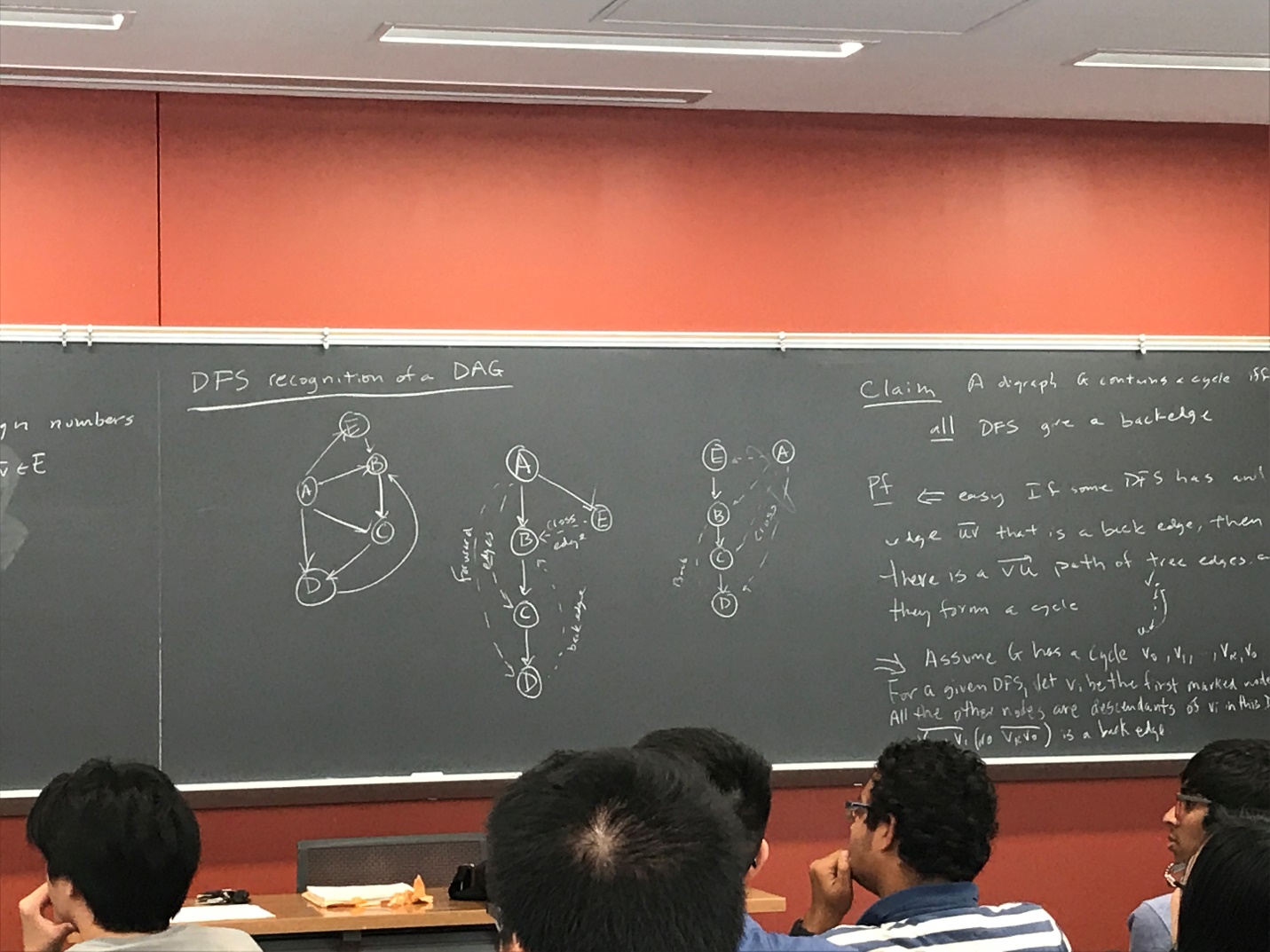
CCDFS(u,c) mark u with c

For v a neighbor of u

If v not mamrk,CCDFS(v,c)

O(n+m)

## DFS recognition of a DAG



Claim: a digraph a contains a cycle iff all DFS give a backedge

<= easy if some DFS has an edge <u,v> that is a backedge, then there is a <v,u> path of tree edge and they form a cycle.

* Assume G has a cycle v0,v1,…,vk,v0. For a given DFS, let vi be the first marked node in cycle. All the other nodes are descendants of vi in this DFS tree, <vi-1,vi> (<vk,v0>) is a backedge

## DFS for backedge

BEDFS(u)

Mark u

Set active bit of u.

The neighbors—active bit on = backedge

Unset active bit of u

## Topological Sort

Given a DAG G=(V,E), assign number 1,…,n to vertices such that , num[u]<num[v]

Cannot just use DFS

## Deactivation number

TS=n

DFS(u)

Mark u

Set active bit of u

The neighbors

Unset active bit of u

Num(u)=TS

TS- - decrement

Consider tree edge uv

V deactivated first

Forward edge uv

V deactivated first

Cross edge uv

V deactivated first

Back edge

Not applicable

## Transitive Closure of DAGs

-Transitive closure: if there is a path u🡪v in G, then therer is an edge in

DFS from each node O()

Fun fact: multiplying 2 mxn matrices takes time M(n)

O(nM(n))

I is the identity matrix. In graph, I is self-loops.

Compute by repeated squaring O(M(n)log n)

T=

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

=O(M(n))

M(n)=

|  |  |  |  |
| --- | --- | --- | --- |
|  | B1 | B2 | B3 |
| X1 | 1 | 0 | 1 |
| X2 | 1 | 0 | 0 |
| X3 | 1 | 1 | 1 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | X1 | X2 | X3 |  |  |  |  |
| A1 | 1 | 0 | 1 |  | 1 | 1 | 1 |
| A2 | 0 | 1 | 1 |  | 1 | 1 | 1 |
| A3 | 0 | 1 | 0 |  | 1 | 0 | 0 |

|  |  |  |  |
| --- | --- | --- | --- |
|  | A | X | B |
| A | - | 1 | - |
| X | - | - | 1 |
| B | - | - | - |

Product of these 2 matrices yields transitive closure

## Strongly Connected Components

Reduced graph where each strongly connected component is shrinked to one node, the remaining is a DAG.

DFS

A🡪B 🡪C 🡪 D🡪F 🡪G🡪E

B🡪E

G🡪D

Two nodes in same scc, then they are in same tree of DFS forest

T1 T2 T3 T4 T5 A

E F C B A B

G C, E

D D

F

G

Idea reversing edges () doesn’t change scc. Try DFS, then DFS on

* First DFS in any order
* Reverse edge s
* DFS in order of last finishing time first

A B C D G F E

## Online algorithm

Renting skis cost $50

Buying skis cost $500

You would ski until you break your leg and you will not ski after your leg is broken.

What is the optimal strategy ?

Compare with someone who knows the strategy

Online strategy

Subtribe until it reaches the price of permanent and buy it. It’s only a factor of 2 apart and it’s optimal.

## External memory algorithms

File systems and theory

## Randomized algorithms

# November 10, 2017

Different paying algorithms differ by eviction policy

## An online algorithm

A is a-competitive if constant b for every request sequence R :

Cost of servicing R in A is no more than a times the optimal to service R

What are some possible choices for eviction policy?

For example: FIFO, LRU (least recently used), LFU(least frequently used)

Suppose K=3, 3 slots Request sequence is 1,2,3,4,1,2,3,4

Cost : 1 1 1 1 1

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 4 | 4 | 4 |
| 2 | 2 | 1 | 1 |
| 3 | 3 | 3 | 2 |

OPT=furthest future (offline)

Cost: 1 1 1 1 - - 1 - - 1 - -

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 1 | 1 | 2 |
| 2 | 2 | 3 | 3 |
| 3 | 4 | 4 | 4 |

n request

So, LRU is no better than 3-competitive

## Claim

FF (Furthest Future) is the offline opt (offline means you know the future sequence)

Pf: the proof of this (take home final)

## Claim

LRU is no better than k-competitive

## Claim

No deterministic algorithm is better than K-competitive

Always request item just evicted , so A pays n

FF item is always at least K away (K items in memory) so OPT pay <= n/K

K is number of pages. Unit size number of items.

## Resource Augmentation

What if we compare LRU with 2K memory versus opt with K memory?

## Thm is 2-competitive with

There exists a b where for all R

Proof:

Divide LRU into phases, each with K faults

P1,r2,…,ri, ri+1,…,rj, rj+1,…,rR, rR+1,…

must have at least 1 fault in each phase

Boring case analysis

must have at least K faults in each phase

(Consider the at least 2K different pages requested during the phase, have start the phase with at least K of them, so it must fault on at least K of them. pays 2K per phase. pays >= K per phase)

## Cache oblivious

-unknowable M,B

-optimal cache evictions

M B reads in blocks of B

Sum=0 O(N/B) I/Os

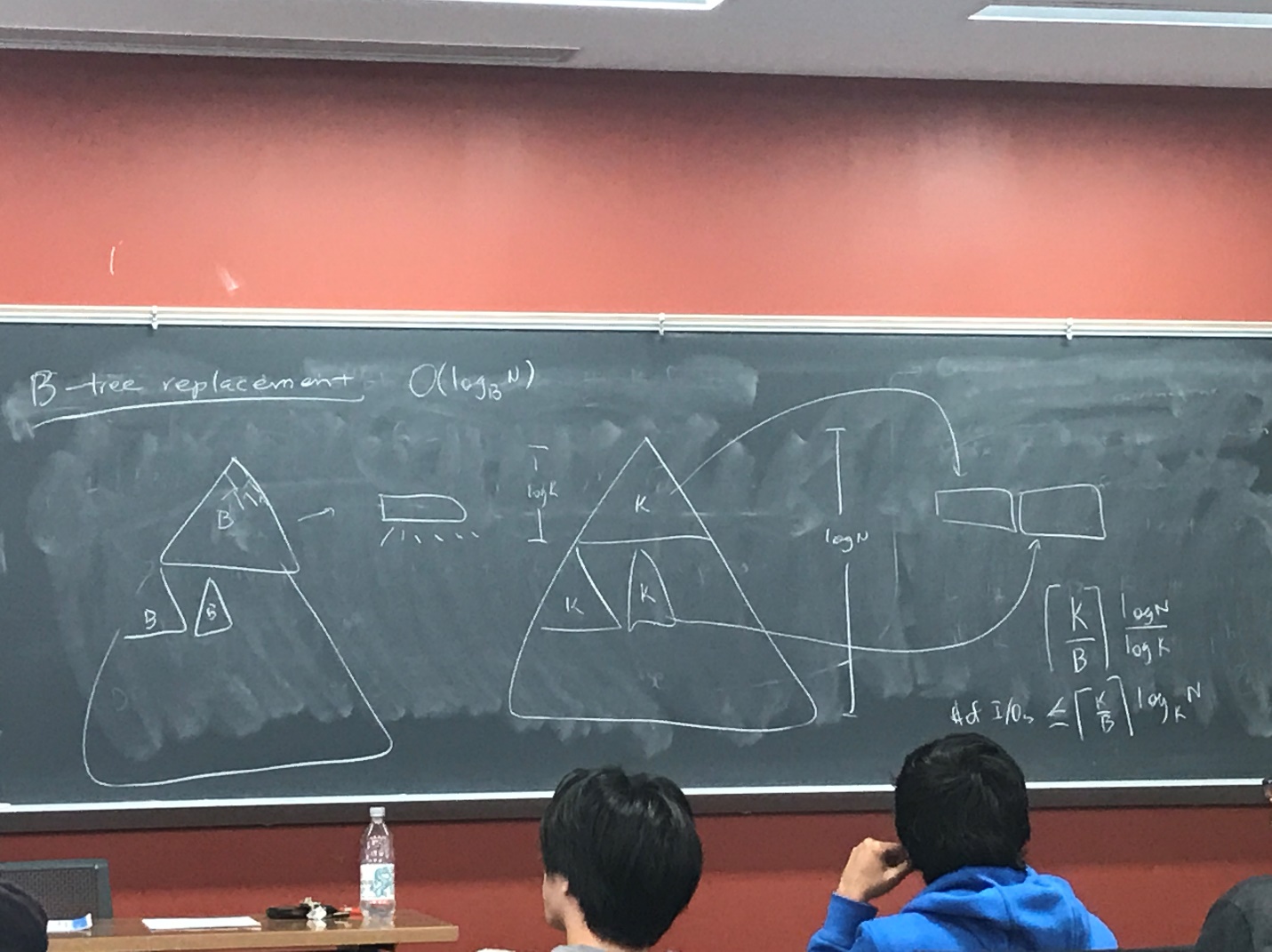
For i = 1 to N

Sum+=A[i]

### Binary search

O(log (N/B) ) I/Os

## B-tree replacement



Searches go in O(), which means branching factor of B

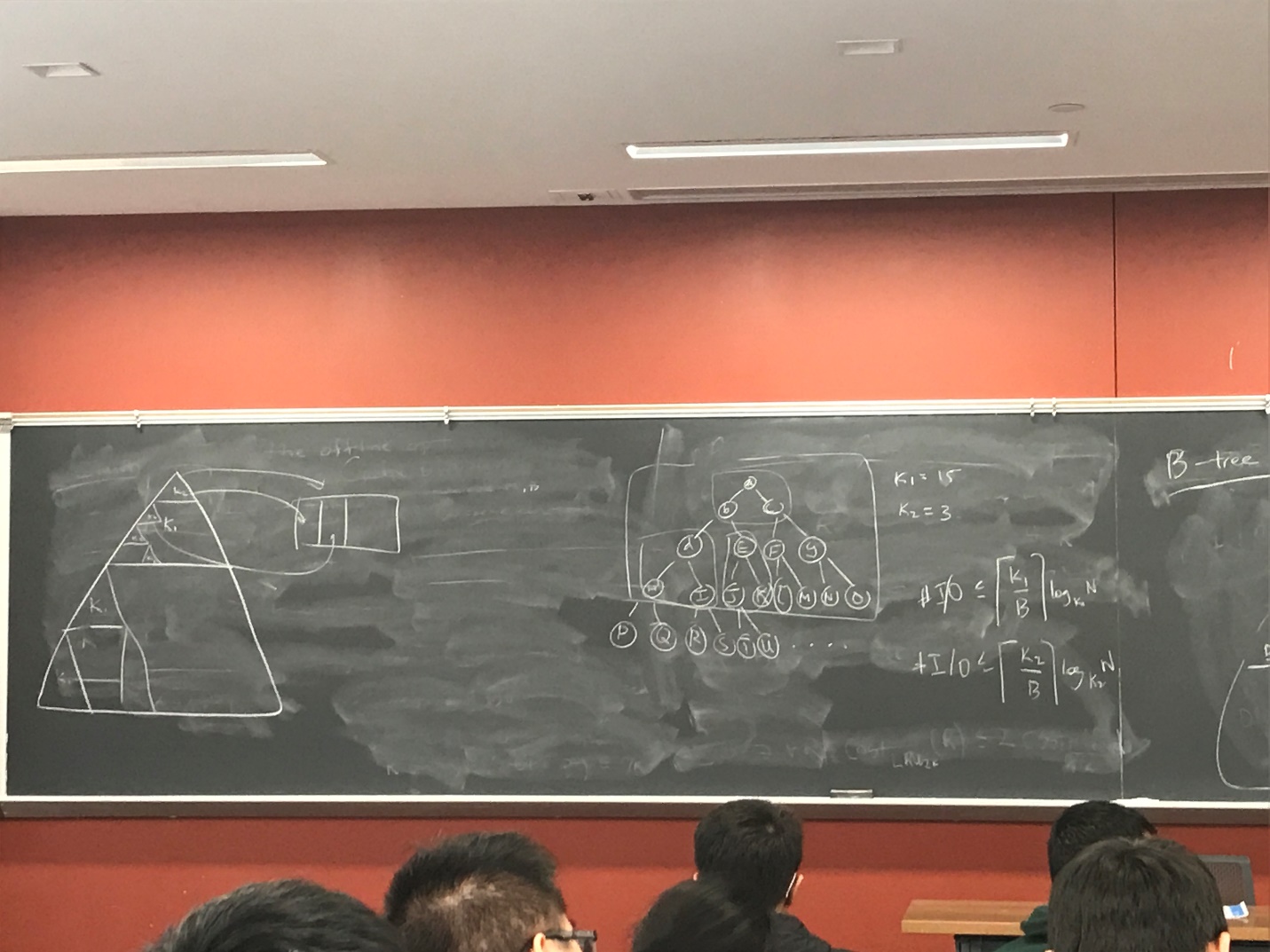
Number of I/Os <= upper bound

Suppose we have K1, inside K1 we have subdivision K2

Because K1 divides K2, we can do both at the same time

Binary searching on log B

Get to a constant log B because there is only log log .. choice of B



I/Os <=

I/Os <=

I/Os <=

I/Os <=

…

I/Os <=

<=

We do not know B. We have a guess B. Suppose

I/Os

Build all the B-trees and find the best B vs build K and nested recursive thing.

Why this recursion is faster than a B-tree?

Minimize page fault at one B and minimize cache line misses at a different B,simultaneously

Prefetching! There is no fixed B. For every fetch,

Cache line misses are cheap