Andy Yin
Therese Biedl
CS240
Lecture - Introduction
Jan 6, 2015

* Information
  - Administriva [web page]
  - Course Objective [-]
  - Designing Algorithms [S Sections 2.1, 2.2, 2.7
    GT Sections 1-1.1.1]
  - Personnel
    - Therese Biedl, biedl@uwaterloo.ca
    - ISA: Joseph (Danny) Sitter
      cs240@student.cs.uwaterloo.ca
    - Coordinator: Karen Anderson

  - two other sections: section 3 is different (enhanced)
    - lots of TA's; one IA

* Lectures
  - no slides, old-fashion styled with chalk
    - modules on the web have similar content

* Textbooks
  - on website

* Marks
  - Final 50%] must pass weighted average to pass course
  - Midterm 25%
  - Assignment 5 x 5% no lates allowed

* Cheating
  - copying or excessive collaboration
  - dealt with harshly

Hilroy

* Courtesy
- don't do anything that keeps your neighbour from learning

* Announcements
- assignment 0 is available, due Wed, Jan 14 : 8:30am
- assignment 1 should be available tomorrow, due Wed, Jan 21
- tutorials
  - start next week

* CS240 course objectives
- imagine you have a lot of data to store
  - english dictionary ← lookup words, rarely add words
  - music collection ← sort by artist, find favorite song by voting
  - large word document ← search for word
- you can store it in data structure we've seen before (arrays, lists)
  - but, can we store it in a <u>better</u> structure?
- our objectives for better
  - small runtime
  - use little memory

- in this course, we do theoretical analysis
  - develop ideas and pseudo-code that we usually don't implement, and analyze them using big-O notation

* Review of Previous Data Structures
- arrays, linked lists
- strings
- stack, queues
- abstract data types
- recursive algorithms
- binary tree
- sorting: insertion sort, selection sort, mergesort
- binary search
- binary search trees

* Review of Mathematic Concepts
  - arithmetic series, geometric series
  - harmonic series; $1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots \frac{1}{n} = \ln(n) + \text{constant}$
  - logarithms are in base 2, unless said otherwise
    - so $\log(n) = \log_2(n)$
  - logorithm rule: $\log_b(a) = \frac{\log(a)}{\log(b)}$

* Problem (eg, sorting)
  - description of the situation, and the desired change/outcome

* Problem Instance
  - one particular input to a problem
  - eg, [1, 2, 10, 3, 9, 4] for a sorting algorithm

* Instance Solution
  - the correct answer/change for this instance

* Solving a Problem
  - you must give a correct algorithm
  - by correct, for every instance, it finds a solution.
    algorithm, a finite-description process that gives an answer for all instances
  - usual process to solve a problem
    - algorithm design
    - argue correctness
    - analyze how good it is (efficiency)
      - lower bounds?
    - implement > not done as much in CS240
    - experiment

* Algorithm Design
  (1) write done the main idea in English.

- example: sort array A
  - an idea could be: for increasing i, make $A[0, \ldots, i]$ sorted by inserting $A[i]$ into sorted $A[0, \ldots, i-1]$

(2) pseudo-code (if details are needed)

Input: an array $A[0, \ldots, n-1]$

Output: A is sorted.

```
for i = 1 ... n-1 { // A[0..i-1] is sorted
    key = A[i]
    int j = i-1
    while i >= 0 and A[j] > key { // look at (i)
        A[j+1] = A[j]
        j = j-1
    }
    A[j+1] = key
}
```

(3) argue correctness
- no formal correctness
- but, give a lot of invariants
  - example: in comments above, unless stated otherwise
    (i) invariant for inner loop:
    - the new $A[j+1 \ldots i]$ contains items bigger than key in sorted order
    - $A[0 \ldots j]$ is in sorted order

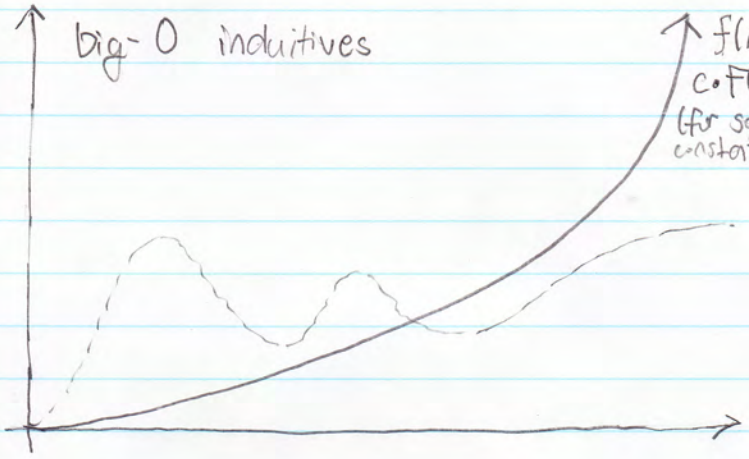
(4) argue termination
- finiteness
  - the algorithm returns an answers on all instances
  - it is especially important for recursive algorithm
    - write down what gets smaller

*Algorithm Analysis (efficiency)
- analyze runtime
- analyze the amount of memory used

- the main idea is to count every elementary operation as one

step
- sum them up and find the answer
  - these sums get messy, and are not fair anyways

- as a simplification, we will use big-O notation which hides the constant factors and make analysis simpler (but easier to get wrong)



- for big-O notation, we don't care what happens for small n

- definition
  - let $f(n)$, $g(n)$ be two positive function
  - we say that: $g(n) = O(f(n))$
    if $\exists\ c>0\ \exists\ n_0$ such that
    $g(n) \leq c \cdot f(n)$ for all $n \geq n_0$

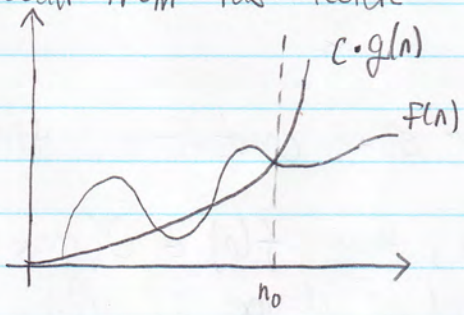
- we can also say that: $g(n)$ is in big O of $f(n)$
  $g(n)$ is asymptotically not bigger than $f(n)$

Hilroy

Andy Yin
Therese Biedl
CS240
Lecture -
Jan 8, 2015

* Information
  - Big-O examples, rules [S Section 2.4, 2.6]
  - Computer model [GT Section 1.1.2]
  - Friends of Big-O [GT Section 1.2]

* Big-O
  - recall from last lecture



$\exists c > 0, \exists n_0 > 0$ such that $f(n) \le c \cdot g(n) \quad \forall n \ge n_0$

(1) prove Big-O from first principles, find such a c and n_0

  - example

$$f(n) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}$$

  - to show $f(n) \in O(n^2)$
    - use $c = 1$, $n_0 = 1$, then

$$f(n) = \frac{n^2}{2} + \frac{n}{2} \le \frac{n^2}{2} + \frac{n^2}{2} = n^2 = c \cdot n^2 \quad \text{for all } n \ge 1 \ (= n_0)$$

$$\frac{n}{2} \le \frac{n^2}{2} \text{ if } n \ge 1$$

  - example

$$f(n) = \sum_{i=1}^{n} i^2 \le \sum_{i=1}^{n} n^2 = n^3$$

so $f(n) \in O(n^3)$ (use $c = 1$, $n_0 = 0$)

(2) prove Big-O using limit rule

- lemma
  - if $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ is finite (possibly 0), then $f(n) \in O(g(n))$

- example
  - let $f(n) = \log(n)$, $g(n) = \sqrt[100]{n} = n^{1/100}$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\log(n)}{n^{1/100}}$$

$$= \lim_{\text{l'Hôpital}\ n\to\infty} \frac{\text{constant}/n}{1/100 \cdot n^{1/100-1}} = 100 \cdot \text{constant} \cdot \lim_{n\to\infty} \frac{1}{n^{1/100}}$$

$$= 0$$

- so $\log(n) \in O(\sqrt[100]{n})$

(3) some rules for Big-O

- if $f(n) = a \cdot g(n)$, for some constant a, then $f(n) \in O(g(n))$
- if $f(n) = g(n) + h(n)$, then $f(n) \in O(\max\{g(n), h(n)\})$
- if $f(n)$ is a polynomial of degree d, then $f(n) \in O(n^d)$

- goal: analyze the runtime of algorithms using Big-O
  - need to clarify computer model
    - we are using the Random Access Machine Model
      - assume the computer has a set of memory cell
        - each cell stores one "word"
      - there is no limit on a size of a word
      - any access to a memory cell (for which, we know the address) take constant time
        - array-like, can access any cell in constant time
      - measure run-time by number of primitive operation
        - access memory cell
        - add, subtract, multiply, divide, round
        - compare numbers, evaluate boolean
        - following a reference

- start subroutine, end subroutine

* Algorithm Analysis (on pseudo code)
  - count the number of primitive operations
    - usually one line of code (LOC) → $O(1)$
  - loops
    - count the number of operations within the loop, then sum up over all executions
  - nested loop
    - nested sum
  - function calls
    - analyze separately, then add
    - if recursive, get a recursive function (later)

- example: find the runtime of Test

```
function Test(int n)
    sum := 0
    for i = 1 ... n                          ]
        for j = i + 1 ... n                  ]  n - i executions
            sum += (i - j)^2  // O(1)        ]
    return sum
```

- so, $O(1) + \sum_{c=1}^{n} \sum_{j=i+1}^{n} O(1)$
  - let c be a constant such that each "$O(1)$" term in this equation is at most c.
    - then runtime is at most $O(n^2)$

$$c + \sum_{i=1}^{n} \sum_{j=i+1}^{n} c = c + c \sum_{i=1}^{n} \sum_{j=i+1}^{n} 1 \quad \leftarrow \text{define as } n-i$$

$$= c + c \sum_{i=1}^{n} (n-i), \text{ let } k = n-i$$

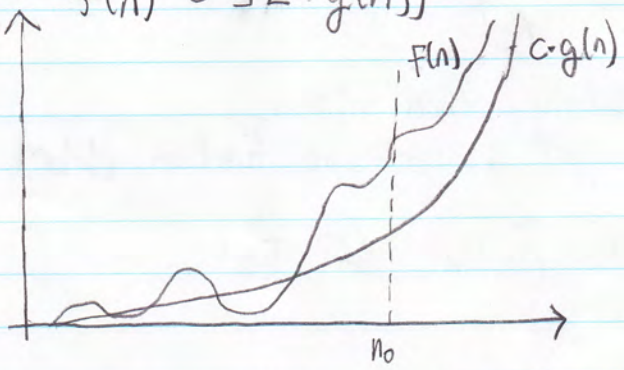
$$= c + c \sum_{k=0}^{n-1} k = c + c \cdot \frac{(n-1)n}{2} \in O(n^2)$$

* Announcements

- Piazza
- AI has been posted
- Office Hours: Friday 11-12, DC 2341

* Lower Bounds
  - the function Test had a runtime $O(n^2)$
    - we need to show that this runtime is tight
  - asymptotic notation for lower bounds
    - big-Omega

$f(n) \in \Omega(g(n))$



- formally, $f(n) \in \Omega(g(n)) \Leftrightarrow \exists\, c>0, \exists\, n_0>0$ such that $f(n) > c \cdot g(n)$ for all $n \geq n_0$

- example: prove $f(n) \in \Omega(n^3)$
  - let $f(n) = \sum_{i=1}^{n} i^2$ $\quad (\in O(n^3))$
  - proof

$$f(n) = \sum_{i=1}^{n} i^2 \geq \sum_{i=\frac{n}{2}+1}^{n} i^2 \geq \sum_{i=\frac{n}{2}+1}^{n} (n/2)^2$$

(this assumes n is even)

$$= \frac{n}{2} \cdot \left(\frac{n}{2}\right)^2 = \frac{n^3}{8} \in \Omega(n^3)$$

↑ use $c = \frac{1}{8}$, $n_0 = 1$

- Theta

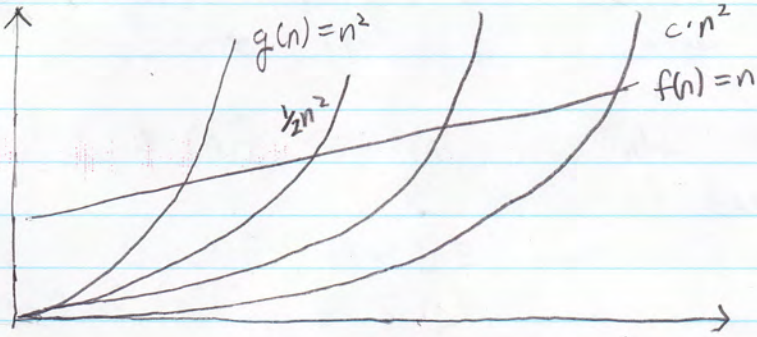
$f(n) \in \Theta(g(n))$

"$f(n)$ is in theta of $g(n)$" $\Leftrightarrow f(n) \in O(g(n)) \wedge f(n) \in \Omega(g(n))$

- example

$\sum_{i=1}^{n} i^2 \in \Theta(n^3)$

- two related concepts
  - little-o
    $f(n) \in o(g(n))$
    "$f(n)$ is in little-o of $g(n)$"
    "$f(n)$ is asymptopically smaller than $g(n)$"
    $\Leftrightarrow \forall c > 0, \exists n_0 > 0, f(n) \leq c \cdot g(n), \forall n \geq n_0$



- you can always pick a smaller c

$f(n) = n + 10 \leq c \cdot n^2$ for sufficiently large n no matter what c was

  - small-omega
    $f(n) \in \omega(g(n))$
    "$f(n)$ is in small-omega of $g(n)$"
    "$f(n)$ is asymptpically bigger than $g(n)$"
    $\Leftrightarrow \forall c > 0, \exists n_0 > 0, f(n) \geq c \cdot g(n), \forall n \geq n_0$

* Limit-rule
- let $f(n), g(n)$ be positive function
  - if $L = \lim_{n \to \infty} \frac{f(n)}{g(n)}$ exist, then

$$L = \begin{cases} 0 & \Rightarrow f(n) \in o(g(n)) \\ \text{constant} > 0 & \Rightarrow f(n) \in \Theta(g(n)) \\ \infty & \Rightarrow f(n) \in \omega(g(n)) \end{cases}$$

* Relationships
$f(n) \in \Theta(g(n)) \Leftrightarrow g(n) \in \Theta(f(n))$
$f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$
$f(n) \in o(g(n)) \Rightarrow f(n) \in O(g(n))$

$f(n) \in o(g(n)) \Rightarrow f(n) \notin \Omega(g(n))$

- a few statements that use these
  - insertion sort take $O(n^2)$ time
  - $1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} \in \theta(\log(n))$
  - binary searches uses $\log(n) + o(\log(n))$ comparison
  - sorting takes $\Omega(n \log(n))$ comparisons

- proof of $f(n) \in o(g(n)) \Rightarrow f(n) \notin \Omega(g(n))$
  - we need to show
    - it is false that $\exists c > 0, \exists n_0 > 0$
      $\forall n \geq n_0, f(n) \geq c \cdot g(n)$
    - show that $\forall c > 0, \forall n_0 > 0,$
      $\exists n \geq 0, f(n) < c \cdot g(n)$
      - we know that $\forall c_1 > 0, \exists n_1 > 0, \forall n \geq n_1$
        $f(n) < c_1 \cdot g(n)$

- to show this,
  - assume $0, c,$ and n_0 are given
  - pick $c_1 = c$
    - there exists a n_1 such that
      $f(n) < c_1 \cdot g(n) = c \cdot g(n), \quad \forall n \geq n_1$
  - pick $n_2 = \max\{n_0, n_1\}$, then
    $n_2 \geq n_0$
    $f(n_2) < c \cdot g(n_2)$, since $n_2 \geq n_1$

Andy Yin
Yosef
CS 240
Lecture
Jan 12, 2014

* General Big-O Variations

$f(n) \in O(g(n)) \iff \exists c, n_0 > 0$ such that
$\forall n \geq n_0,\ 0 \leq f(n) \leq c \cdot g(n)$

$f(n) \in \Omega(g(n)) \iff \exists c, n_0 > 0$ such that
$\forall n \geq n_0,\ 0 \leq f(n) \leq c \cdot g(n)$

$f(n) \in \Theta(g(n)) \iff \exists c_1, c_2, n_0 > 0$ such that
$\forall n \geq 0,\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$

$f(n) \in o(g(n)) \iff \forall c > 0, \exists n_0 > 0$ such that
$\forall n \geq n_0,\ 0 \leq f(n) \leq c g(n)$

$f(n) \in \omega(g(n)) \iff \forall c > 0, \exists n_0 > 0$ such that
$\forall n \geq n_0,\ 0 \leq c g(n) \leq f(n)$

* Aligning .tex

```
\begin{align*}
  25x + 5 &= 10 \\
        x &= \frac{13}{5} \\
\end{align*}
```

* Example 1: $12n^3 + 11n^2 + 10 \in O(n^3)$

$12n^3 + 11n^2 + 10 \leq 12n^3 + 11n^3 + 10n^3$
$= 33n^3$

$c = 33,\ n_0 = 1$

* Example 2: $n^2 - 3n \in \Omega(n^2)$

$cn^2 \leq n^2 - 3n$
$c \leq 1 - \frac{3}{n}$

$c = \frac{1}{2},\ n_0 = 6$

*Example 3: $1000m \in o(m \log(m))$

$1000m \leq cm \log(m)$

$1000 \leq c \cdot \log(m)$

$\frac{1000}{c} \leq \log(m)$

$2^{\frac{1000}{c}} \leq m$

$m_0 = 2^{\frac{1000}{c}}$

$c \cdot m \cdot \log(m) \geq cm \cdot \log(m_0)$

$c \cdot m \cdot \log(m) \geq c \cdot m \cdot \log\left(2^{\frac{1000}{c}}\right)$

$c \cdot m \cdot \log(m) \geq c \cdot m \cdot \frac{1000}{c}$

$cm \log(m) \geq 1000m$

*Example 4: $3^m \in \omega(2^m)$

$3^m \geq c \cdot 2^m$

$\frac{3^m}{2^m} \geq c$

$\left(\frac{3}{2}\right)^m \geq c$

$m \geq \log_{3/2}(c)$

$\forall m \geq m_0$

$3^m = \frac{3^m}{2^m} \cdot 2^m$

$= \left(\frac{3}{2}\right)^m \cdot 2^m$

$3^m \geq \left(\frac{3}{2}\right)^{m_0} 2^m$

$3^m \geq \left(\frac{3}{2}\right)^{\log_{\frac{3}{2}}(c)} \cdot 2^m$

$3^m \geq c \cdot 2^m$

*Example 5: Loop Analysis

```
function fun(m, i)
  cs = 0                                   // O(1)
  for i from 1 to 3m                       // first loop
    cs *= 4
    for j from 1388 to 2010                // second loop
      for k from 4i to 6i                  // third loop
        cs += k
```

first loop $\sum_{i=1}^{3m} \dots$ second loop $\sum_{j=1388}^{2010} \dots$ third loop $\sum_{k=4i}^{6i} O(1) \Rightarrow c = 1$

$$\Rightarrow \sum_{i=1}^{3m} \sum_{j=1388}^{2010} \sum_{k=4i}^{6i} 1$$

$$= \sum_{i=1}^{3m} \sum_{j=1388}^{2010} 2i$$

$$= 2 \sum_{i=1}^{3m} \sum_{j=1}^{623} i = 2 \sum_{i=1}^{3m} 623i$$

- recall $\sum_{i=1}^{m} i = \frac{m(m+1)}{2}$,

$$= 2 \cdot 623 \cdot \frac{(3m(3m+1))}{2}$$

$$= 5607m^2 + 1869m \in \Theta(m^2)$$

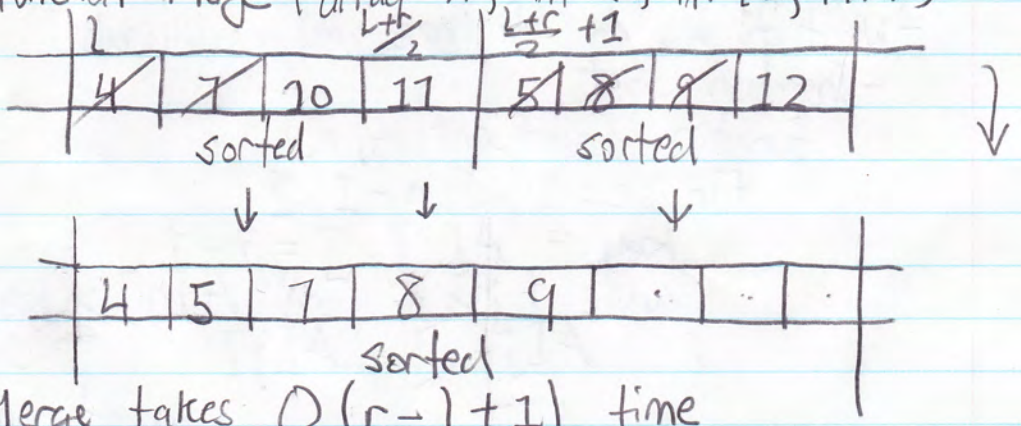
Andy Yin
Therese Biedl
CS 240
Lecture —
Jan 13, 2015

* Information
  - Algorithm Analysis [S 2.6]
  - Comparing Algorithms [S 2.7]
  - Priority Queues [S 9, 9.1]

* Analyzing Recursions
  - Mergesort

```
function Mergesort (array A, int l=0, int r=n-1) {
  // input: array A with n elements
  // output: A is sorted, precisely A[l ... r] is sorted
    if l<r {
      Mergesort (A, l, ⌊(l+r)/2⌋)
      Mergesort (A, (((l+r)/2)+1), r)
      Merge (A, l, (((l+r)/2)+1), r)
    }
}
```

function Merge (array A, int l1, int l2, int r)



- Merge takes $O(r-l+1)$ time
  - recursive calls → recursive run-time functions

* Recursive Runtime for Mergesort

- define $T(n)$ to be the runtime of Mergesort, if we have to sort n elements ($n = r - l + 1$)

$$T(n) = \begin{cases} O(1) & \text{if } n \leq 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n) & \text{otherwise} \end{cases}$$

- precise: $T(n) \leq \begin{cases} c & \text{if } n \leq 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + d \cdot n & \text{otherwise} \end{cases}$

for some constants c, d

- if n is even,

$$T(n) \leq \begin{cases} c & \text{if } n \leq 1 \\ 2 \cdot T(n/2) + dn & \text{otherwise} \end{cases}$$

*How to Analyze such a Program

- if $T(n) \leq 2T(n/2) + O(n)$, then $T(n) \in O(n \log n)$
- we could prove this by induction, in CS 341, there is a systematic method called the Master's Theorem

- another relation

$$L(n) = \begin{cases} L(n/2) + O(1) & \text{if } n \geq 2 \\ O(1) & \text{if } n \leq 1 \end{cases}$$

resolves to $L(n) \in O(n \log n)$

- what do we do if there are conditionals?
  - Insertion sort

```
function InsertSort(A, n)
  for c = 1 ... n-1 {
    key = A[i], j = i-1
    while [ j >= 0 and A[j] > key ) {        ]
      A[j+1] = A[j]                          ] O(1)   | inner loop
      j--                                    ]
    }
    A[j+1] = key
}
```

}

- the number of executions depend on the input (somewhere between 1 and i)
  - worst case analysis: What's the worst possible input?
    - that is, compute an upper bounds on a runtime

- for insertion sort,
  - the inner loop happens at most i time
  ⇒ runtime is less than/equal to $\sum_{i=1}^{n-1} i \cdot O(1)$

$$= O(1) \underbrace{\sum_{i=1}^{n-1} i}_{\in O(n^2)} \quad \in O(n^2)$$

  - if you use upper bounds, then also show that they are tight
    - given an instance (of size n) on which this runtime asymptotically occurs

- consider an array in reverse sorted order, then A[i] is smaller than all of A[0 ... i-1]
  ⇒ the inner loop must exchange at least i-1 times
  ⇒ runtime is less than/equal to $\sum_{i=1}^{n-1} (i-1)$

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

* Worst Case Analysis
- in CS240, the worst case analysis refers to the worst possible runtime

$$T^{worst}(n) = \max_{\text{instance } I \text{ such that } I \text{ has size } n} \{ T(I) \}$$

($T(I)$: runtime on instance I)

Hilroy

* Average Case Analysis
  - the average runtime per all instances of size n

$$T^{avg}(n) = \frac{\sum_{\text{instances } I \text{ of size } n} T(I)}{\text{number of instances of size } n}$$

* Best Case Analysis (not going seen much in CS240)
  - the best case runtime per all instances of size n

$$T^{best}(n) = \min_{\text{instance } I \text{ such that } I \text{ has size } n} \{ T(I) \}$$

* Growth Rates
  - constant $\Theta(1)$ (ie. accessing an item in an array)
  - logarithmic $\Theta(\log n) = \Theta(\log_{10} n) = \Theta(\ln n)$ (ie, binary search in a sorted array)
  - linear $\Theta(n)$ (ie, find max in unsorted array)
  - ??? $\Theta(n \log n)$ (ie, merge sort)
  - quadratic $\Theta(n^2)$ (ie, insertion sort)
  - polynomial $\Theta(n^d)$ (ie, matrix multiplication)
  - exponential $\Theta(2^n)$ (ie, optimization problems)

- all of these bounds are tight for such examples given, do not mix O and Θ
  - recall O means there is exists an upper bound

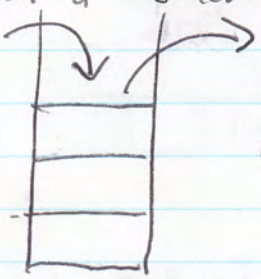
- the goal of this course is to find an algorithm or data structure that have small runtimes
  - ie, algorithm A_1 has runtime $O(n^2)$, and algorithm A_2 has runtime $O(n \log n)$, then is A_1 is better than A_2?
  - the argument does not hold well, since it's a upper bound

- if A_1 has runtime $\Theta(n^2)$
  A_2 has runtime $\Theta(n \log n)$, then
  - in general, A_2 is better, but be careful
    - analyze best case, and average case
    - analyze efficiency for small n, is it reasonable?

* Announcements
  - bring student ID
  - A0 is due tonight...

* Abstract Data Types (ADT): Priority Queues
  - recall we had a Stack data type

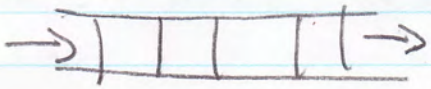
Stack



- last in, first out (LIFO)
- last added element has the highest priority of being removed

  - recall we also had a Queue data type

Queue



- First in, first out (FIFO)
- First added element has highest priority

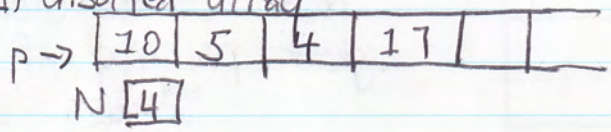
  - we can combine these to make a priority queue
    - we set for each item the priority of removing it
    - formal description of abstract data types (ADT)
    (1) can store items that have a priority
      - we call these priority "keys", keys can be sorted (in CS240, they are integers)
      - we really store them as key-value pairs, but code will often only show keys
    (2) Operation Insert(key)
      - inserts the item that has priority "key"
    (3) Operation DeleteMax()
      - remove and return item with the highest key (standard for

CS 240)

- application: Priority Queue sort (array A)
- initialize a priority queue p

```
for i = 0 ... n-1 {
    p.insert(A[i])
}
For i = n-1 ... 0 {
    A[i] = p.deleteMax()
}
// A is sorted
```

- implementation Priority Queues
  (1) unsorted array



$\Theta(1)$ [insert: $p[N] = \text{item to be inserted}$, $N++$

$\Theta(n)$ [deleteMax: search the entire array

Andy Yin
Therese Biedl
CS 240
Lecture
Jan 15, 2015

* Information
  - implementation of priority queues
  - heaps
  - insertion and deletion [S Chapter 9]

* Priority Queues
  - recall, a priority queue
    - stores items with keys (priorities)
    - operation: Insert(key)
    - operation: DeleteMax()

- Implementation: Unsorted Array

P	10	4	2	17	5	8	// array to store items
N	6						// current number of items

  - Insert: add new item at last spot in array
  - DeleteMax: search the whole array to find max, swap max with last item in array, then return max

  - note, overflow handling in arrays (for any implementation that uses arrays)
    - arrays must be initialized with a fixed size
    - if (during insert) the array becomes full
      - create a new array (twice the size)
      - copy old array to new array
      - free old array
      - replace reference to the array to be the new array

  - this means that every once in a while, insert will be slow ($\Theta(N)$ time, where N is the number of items in array)

- but since we double the size, the next n inserts don't have to do overflow-handling
  - so overflow handling will (when divided by all operations) will only cost $\Theta(1)$
  - in CS 466, we will deal with amortized runtime

- insert implementation

```
function Insert(key) {
    if (N == p.size())  -> overflow handling (ignore)
    p[N] = key          ] O(1)
    N++
}
```

- deleteMax implementation

```
function DeleteMax() {
    max_index = 0
    for i = 1 ... N-1 {
        if p[i] > p[max_index] {
            max_index = i
        }
    }
    max_priority = p[max_index]
    p[max_index] = p[N-1]
    N--
    return max_priority
}
```

- Implementation: Sorted Array

P:	2	4	5	8	10	17

N: [6]

- DeleteMax(): return p[N-1] and decrease N
- Insert(): find the place where key must be in array
  shift items to its right to make space
  insert key and increase N

- insert implementation

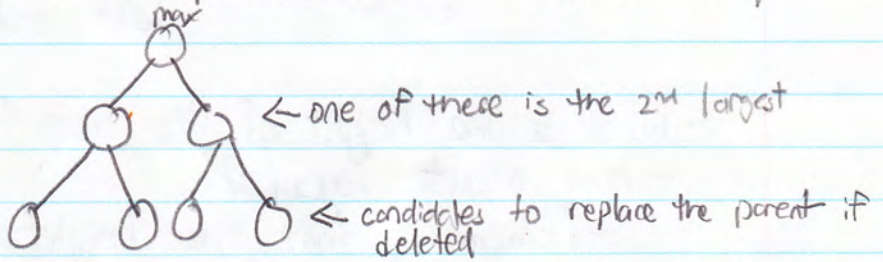
```
function Insert(key) {
    if (N == p.size())  -> overflow handling
    int j = N - 1
    while (j >= 0 && p[j] > key) {
        p[j+1] = p[j]
        j--
    }
    p[j+1] = key
    N++
}
```

(while loop)

- note this PQSort (priority queue sort) with this implementation is very similar to insertion sort
  - the while-loop in Insert executes at most N times. so the runtime of Insert is O(n)
    - this is tight if key is smaller than all items in p

- Implementation: Heaps
  - the idea is to easily know where the max is
    - somehow store the max as a reference, but if we delete the max, then how do we update it when deleting it?
    - we could keep the 1st reference, 2nd reference, but then we will have to find the 2nd reference after we delete it
    - the problem is that we will have to keep it in sorted order

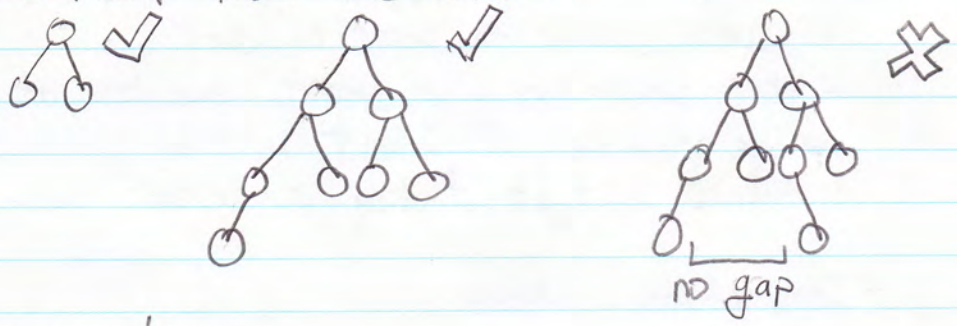
- another idea is to keeep condidates for the 2nd largest



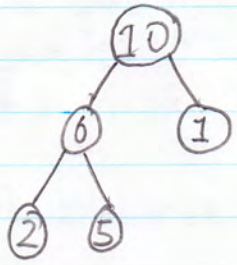
- note, binary trees has T consists of nodes and links between them
  - each node has less/equal to 2 children and less/equal to 1 parent

- you should know: node, root, leaves, interior nodes, parent, child, sibling, ancestor, descendant, height, depth, level, level-order, pre-order, and post-order

- definition
  - a (max-oriented) heap is a binary tree that satisfies the structural property and the (max-) heap property
  - structural property
    - all except the lowest level are completely full
    - the lowest level does not have to be full, but it is filled from the left



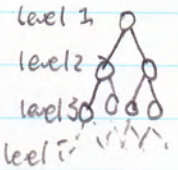
  - heap property
    - for any node i in the heap, key(p(i)) ≥ key(i), where p(i) denotes the parent of i



    - it is easy to find the 2^{nd} largest, but it is difficult to find the 3rd largest
    - there is no relation between the left and right subtree

- what is the height of the heap that has N nodes?
  - how about $\log_2(n)$?
    - remember that the height of a heap is $\Theta(\log n)$
  - proof: the height is $\log_2 n - \omega(\log n)$
    - assume that we have a heap with n nodes and height h
    - claim 1: any binary tree of height h has $N \leq 2^{h+1} - 1$

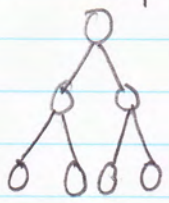
- consider a tree of height h, at any level i, there is a maximum nodes of 2^{i-1}
  - so level i contains less than / equal to 2^{i-1} nodes
  - the number of nodes is



$$\text{number of nodes in level } i \leq \sum_{i=1}^{h+1} 2^{i-1} = \sum_{j=0}^{h} 2^j = \frac{2^{h+1}-1}{2-1} = 2^{h+1}-1$$

- claim 1 $\Rightarrow N \leq 2^{h+1} - 1$
  $\Rightarrow N+1 \leq 2^{h+1}$
  $\Rightarrow \log(N+1) \leq h+1$
  $\Rightarrow h \geq \log(N+1) - 1$

- claim 2: any heap of height h satisfies $N \geq 2^h$



level 1 $= 2^0$
level 2 $= 2^1$
level 3 $= 2^2$
level h $= 2^{h-1}$
level $h+1 \geq 1$

- the number of nodes is

$$\sum_{i=1}^{h+1} \text{number of nodes on level } i$$
$$= \sum_{i=1}^{h} 2^{i-1} + \text{number of nodes on } h+1$$
$$\geq \sum_{j=0}^{h-1} 2^j + 1$$
$$= (2^h - 1) + 1 = 2^h$$

- claim 2 $\Rightarrow N \geq 2^h$
  $\Rightarrow \log N \geq h$

Andy Yin
Youcef Tebbal
CS 240
Tutorial
Jan 19, 2015

*Loop Analysis 1

```
for i from 1 to n                    ①
    for j from 1 to i                ②
        k=j
        while k > 1                  ③
            k /= 2
```

- we can over estimate and under estimate, where n is the upper bound
- over estimate
  - ③ runs in $\log j$ time
  - ② runs in i time
  - ① runs in n time

  $\Big\} O(n^2 \log(n))$
- under estimate
  - consider ① From $3n/4$ to n $\Rightarrow O(n)$
    ② From $n/2$ to $3n/4$ $\Rightarrow O(n)$
    ③ for $k = n/2$ $\Rightarrow O(\log(n))$

  $\Rightarrow O(n^2 \log(n))$

*Heaps
- the children B less than the parent, but adjacent children may not be less/equal/greater than the other parent
- there are two types of heaps: min heap, max heap (default)
- all levels are filled, except the last level (in which it must be filled from left to right)

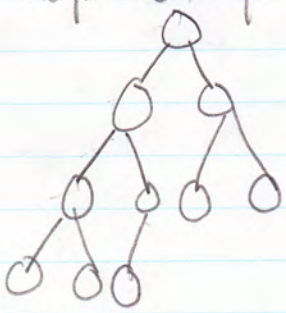
*Note
- from first principles means <u>no limits</u>, L'Hopital, etc

Andy Yin
Therese Biedl
CS 240
Lecture
Jan 20, 2015

* Information
  - storing heaps
  - insertion and deletion
  - heapsort
  - building heaps
  - applications of PQ

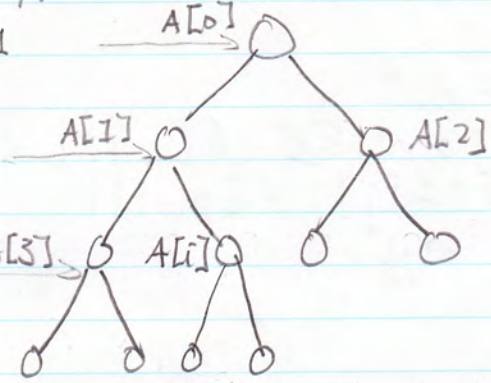
[5 9.2]
[5 9.3]
[5 9.4]
[-]

* Heaps
  - recall that heaps satisfy these properties



  - binary tree
  - structural property
  - MaxHeap property
    key[p(i)] ≥ key[i]
  - height ≈ log(n)

- we can store heaps using arrays
  - array A
    level 1



A[0] stores the root
A[1], A[2] stores children of the root

- we generally store item in the array in level-order

- since we have the structural property, we can find children and parent of A[i] easily

lc(i) = left_child(i) = index of left child of node stored at A[i]
= 2i + 1

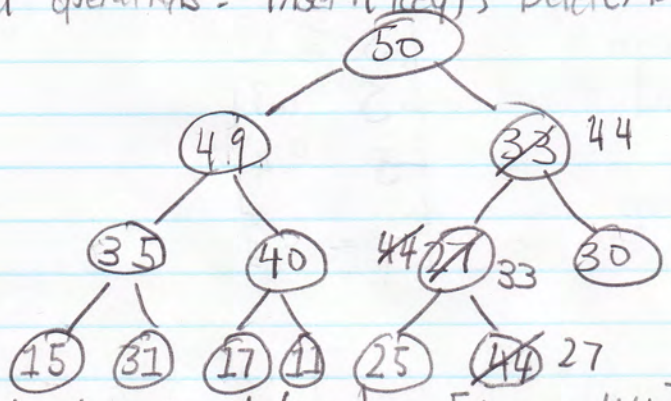
rc(i) = right_child(i) = 2i + 2

$p(i) = \text{parent}(i) = \left\lfloor \frac{i-1}{2} \right\rfloor$

- the index of leftmost in level i is $\sum_{l=1}^{i-1} 2^{l-1}$

* Heaps as Implementation of Priority Queues

- we need operations: Insert(key), DeleteMax()



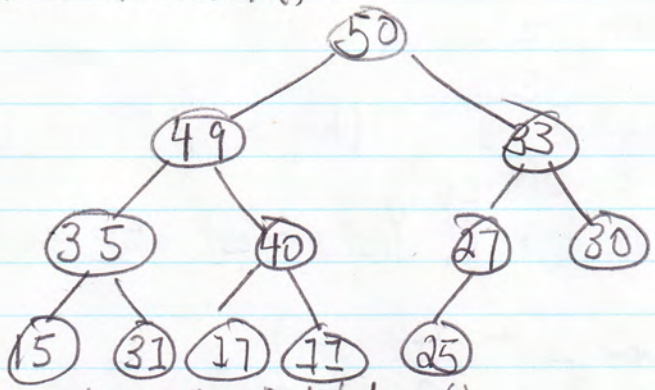
- we want to Insert(key) [key = 44]
  - only have one possible place for new node
  - put new key there
  - restore heap property by "bubbling up" until it holds

- pseudocode for Insert

```
function Insert(key) {
    if A isn't big enough -> overflow handling
    A[N] = key
    N++
    j = N - 1
    while (j > 0 && A[p(i)] < A[j]) {        // bubble up(N-1)
        swap(A[j], A[p(i)])
        j = p(j)
    }
}
```

- the bubble up takes time $O(\log(n))$, since every while-loop execution goes up one level
  - this is tight (has $\Omega(\log(n))$ runtime), since the tree has height of roughly $\log(n)$, and only occurs when the new key is a maximum

- For operation DeleteMax()



- we want to call DeleteMax()
  - max is at the root
  - swap max with last element
  - Fix heap-property by bubbling down while some child is bigger, swap with bigger of the children

- pseudocode for DeleteMax()

```
function DeleteMax() {
    rc = A[0]
    A[0] = A[N-1];
    N--
    j=0
    while (j ≤ N-1) {
      max_index = j
      if (lc(j) exists and  lc(j) ≤ N-1  and
      A[lc(j)] > A[j]){
        max-index = lc(j)
      }
      if (rc(j) exists and A[rc(j)] > A[max_index]) {
        max_index = rc(j)
      }
      if (max_index == j) {
        break
      }else {
        swap (A[j], A[max-index])
        j= max-index
      }
```

```
    }
    return rc
}
```

- the runtime is $O(\text{height of heap underneath the starting element}) \leq O(\log(N))$
  - this is tight if last element was smallest

* Priority Queue Sort (Heapsort)
  - we notice that PQSort takes
    $n *$ Insert + $n *$ DeleteMax time
    Heaps: Insert, DeleteMax take $O(\log N)$ time
    $\Rightarrow$ PQSort with heaps take $O(N \log N)$ time
    (PQSort with heaps = Heapsort)

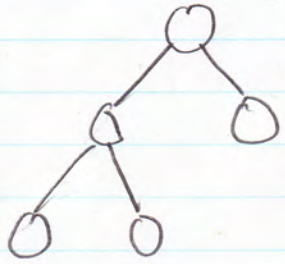
  - situations where Heapsort is bad:
    - if input-array is in sorted order, then insert takes maximum runtime
    - Heapsort is not stable
      - where stable means if two keys are equal, then in the output their order stays the same

* Announcements
  - A1 deadline: Friday, 8:30 am
  - A2 will be posted tomorrow

* Building Heaps
  - given an array A with N numbers
  - build a heap with these numbers

  - Option 1
    For $i = 0, \ldots, N-1$ {
      insert $A[i]$ into a heap
    }
  - has runtime $O(N \log N)$

- this is tight if input is in sorted order, because i^{th} insertion uses about $\log(i+1)$ comparisons



- there are

number of comparisons $\geq \sum_{i=0}^{N-1} \log(i+1)$

$\geq N/2$ entries that are less than $\log(N/2)$

$\geq N/2 \cdot \log(N/2) \in \Omega(N \log N)$

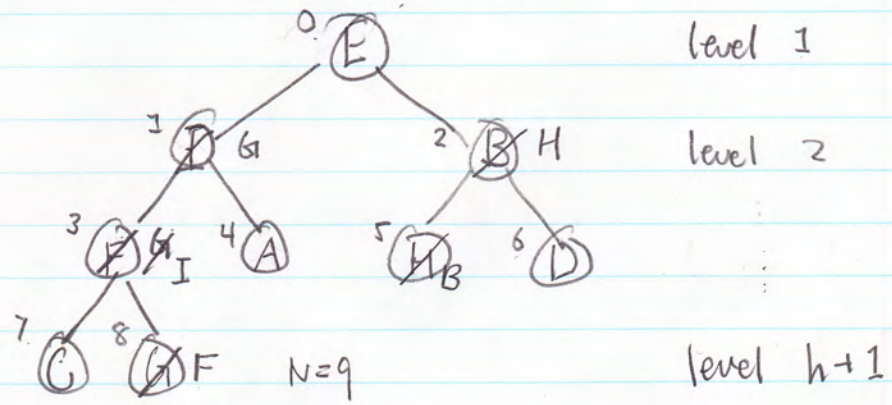
- Better Option

$A[0 \ldots N-1]$ contains items for a heap, but not necessarily with heap property

```
for i = ⌊N/2⌋ - 1 down to 0 {
    bubble_down(i)
}
```

- the invariant is for all $k > i$
  - the sub-heap rooted at k satisfies the heap-property

- but how long does it?
  - the goal is to show that it takes at most $2n$ key-comparisons ("$A[i] < A[j]$?")



- everything else is proportional to the number of comparisons → runtime is $O(N)$
- for ease of proof, assume $N = 2^p - 1$ (all levels are full, height is p)

- any node on level l uses $2 \cdot (p-l)$ comparisons for its bubble down
- the number of comparisons in total is

$$\sum_{l=1}^{p+1} \text{number of nodes on level } l \qquad 2\cdot(p-l)$$

$$= \sum_{l=1}^{p-1} 2^{l+1} \cdot 2(p-l) \qquad \leq 2(2^p - 1)$$

for $k = pl$

$$= \sum_{k=1}^{p-1} 2^{p-k} \cdot k \qquad \leq 2^p - 2p - 2$$

by induction

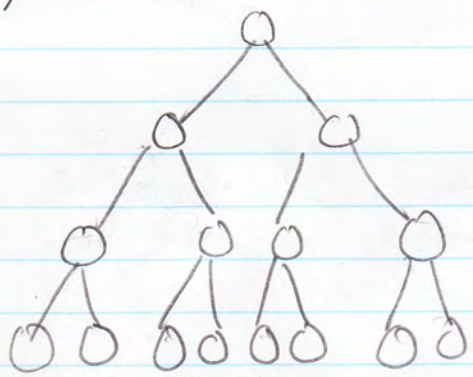
- you can ignore $k = p$

Andy Yin
Therese Biedl
CS 240
Lecture
Jan 22, 2015

* Information
  - Heapify [S 9.4]
  - Selection [S 7.8]
  - Average-case [GT 4.7]
  - Randomized version

* Heapify
  - pseudocode

```
function Heapify() {
  for i = N-1 down to 0 {
      bubbledown(i)
  }
}
```



- analysis, assume N is such that all levels are full

levels	number of nodes in level	number of comparisons for each node
lowest	$\frac{N+1}{2}$	0
one level above	$\frac{N+1}{4}$	2
two levels above	$\frac{N+1}{8}$	4
i levels above	$\frac{N+1}{2^{i+1}}$	$2i$

- so the total number of comparisons is equal

$\sum_{i=0}^{\cdots}$ number comparisons used for nodes that are i levels above the lowest

$$= \sum_{i=0}^{\cdots} \frac{N+1}{2^{i+1}} \cdot 2i \leq$$

$$(N+1) \sum_{i=1}^{\infty} \frac{i}{2^i} = 2(N+1)$$

- proof for $\sum_{i=1}^{\infty} \frac{i}{2^i} = 2$

$$\sum_{i=1}^{\infty} \frac{i}{2^i} = \frac{1}{2} + 2\cdot\frac{1}{4} + 3\frac{1}{8} + 4\cdot\frac{1}{16} + \dots$$

$$= \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \dots$$
$$+ \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \dots$$
$$+ \frac{1}{8} + \frac{1}{16} + \dots$$
$$+ \frac{1}{16} + \dots$$

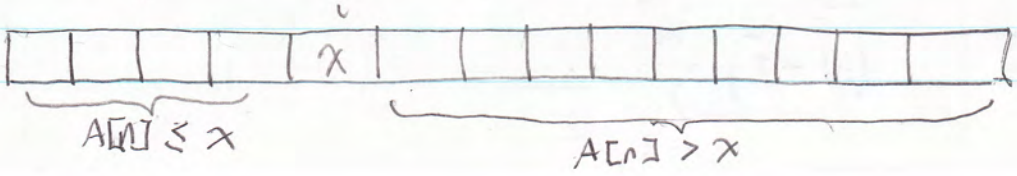
$$= 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots$$

$$= 2$$

* Selection Problem
- given an array A of N integer
  an integer $k \in \{0, \dots, N-1\}$
- we want k^{th} smallest item in A
  - we want $k=0$, the minimum
    $k = N/2$, the median

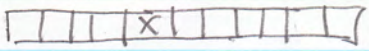
- solution I: sort
- solution II: i build max-heap, call deleteMax $N-k$ times
  II b: build min-heap, call deleteMin $k+1$ times
- solution III: divide and conquer

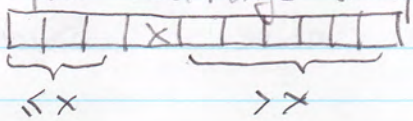
* QuickSelect(A, k)
- if A has 1 element, return it
  otherwise, choose pivot (one element of A (say at index p),
  we use $x = A[p]$ to split A)
  - for now, $p=0$
    partition A at pivot afterwards



let i be the index where x is now
if (i == k) return A[i]
if (i > k) return QuickSelect(A[0, ..., i-1], k)
if (i < k) return QuickSelect(A[i+1 ... N-1], k-(i+1))

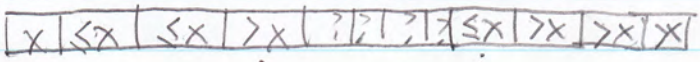
function partition (A, p) {
// input: array (unsorted)
// output: rearrange array such that we return index

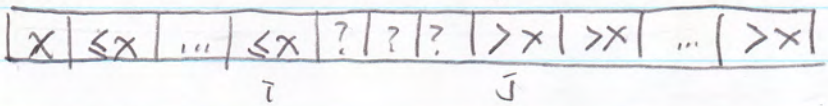




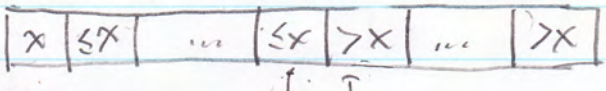
(temporarily put x out of the array)
swap A[0] and A[p]

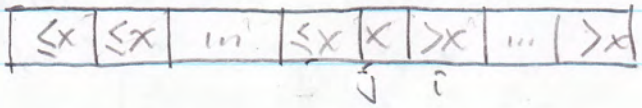
repeat until (i and j meet)
i = 1, j = N-1
while (A[i] ≤ A[0]) increase i
while (A[j] > A[0]) decrease j
swap (A[i], A[j])





-more detailed,
if (j < i)
do not swap
get out of the loop
swap (A[0], A[j])





-return j as index of where the pivot is

- observe that

i can only increase
j can only decrease
⇒ every element in $A[1, \ldots, N-1]$ is compared only once with $A[0]$
⇒ $N-1$ comparisons
⇒ $\Theta(N)$ runtime for partition

- runtime for QuickSelect
  $T(n)$ = worst case runtime for Quicksort on N item

$$= \begin{cases} d & \text{if } N=1 \\ T(\text{size of sub-array for recursion}) + c \cdot N & \text{if } N > 1 \end{cases}$$

  - for some constant d, and c.
- worst case
  - subarray has size $N-1$

$$T(N) \leq T(N-1) + c \cdot N$$
$$= cN + c(N-1) + c(N-2) + \ldots + c \cdot 2 \cdot d$$
$$\in \Theta(N^2) \qquad \left(\text{since } \sum_{i=1}^{N} i = \frac{N(N+1)}{2}\right)$$

- best case
  - no recursion, runtime $O(n)$ (one call of partition)

- average-case
  - recall $T(n) = \dfrac{\sum_{I \text{ instance of size } n} T(I)}{\text{number of instances of size } N}$
  - Quickselect has instance consist of array and integer k
    - can describe array via a permutation π
    - how would I have to permutate A to make it sorted?
    - we will ignore the k-part of the instance (assuming k as bad as possible)

$$T(N) = \frac{\sum_{\text{permutations of } \pi \text{ of } N \text{ elements}} T(\pi)}{N!}$$

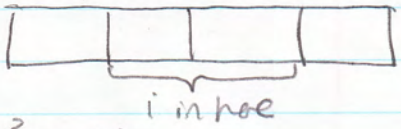
$T(\pi)$ describes a permutation π such that the array would have been (almost) sorted.

$$\frac{1}{N!}\sum_{\text{permutation } \pi \text{ such that } i \in [1/4N,\, 3/4N)} T(\pi) \quad +$$

$$\frac{1}{N!}\sum_{\text{permutation } \pi \text{ such that } i \notin [1/4N,\, 3/4N)} T(\pi)$$

- where i is the index of the pivot after partition

- note, if $i \in [1/4N,\, 3/4N)$



- then sub-array has size $\leq 3/4\,N$
- so $T(\pi) \leq T(3/4N) + C \cdot N$

$$\leq \frac{1/2\, N!\,(T(3/4N) + CN)}{N!}$$

Andy Yin
Youcef Tebbal
CS240
Tutorial
Jan 26, 2015

*Example - Quick Select Psuedo code

```
partition(A, p)
    A ← array of size m, 0 ≤ k < m
(1) swap(A[0], A[p])
(2) i ← 1, j ← m - 1
(3) loop
(4)   while i < m and A[i] ≤ A[0] do
(5)       i ← i + 1
(6)   while j ≥ 1 and A[j] > A[0] do
(7)       j ← j - 1
(8) if j < i break
(9) else swap(A[i], A[j])
(10) end loop
(11) swap(A[0], A[j])
(12) return j
```

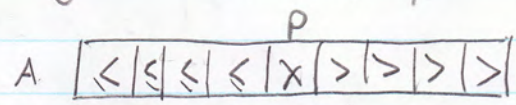
A[⌊n/2⌋] ← start from heapify (faster

*Example - Quick select

Andy Yin
Therese Biedl
CS 240
Lecture
Jan 27, 2015

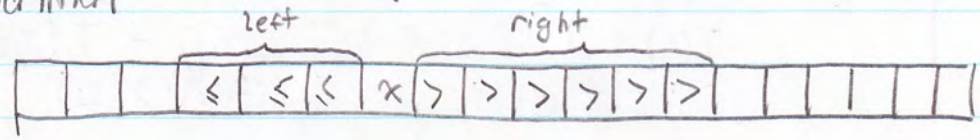
* Information
  Quicksort : worst case , best case [S 7.1, 7.2]
  Average case [-] (books and modules have other proof)
  Randomized [-]
  Speeding it up [S 7.3, 7.4, 7.5]
  Decision Trees [GT 4.4]

* Partition
  - recall that partition splits an array into parts less than, and parts greater than the pivot
  - ie,



  p ← index of pivot
  partition (A, p)

* Quick sort
  - an idea is to recursively sort the left and right subarray from partition




```
function quicksort (array A, int l, int r) {
    //post: A[l ... r] is sorted
    if (r > l) {
        p = pivot (A, l, r)  //subroutine to pick index of pivot
        i = partition (A, l, r, p)  // i : index where the pivot
        quicksort (A, l, i-1)       // is now
        quicksort (A, i+1, r)
    }
```

}

* Runtime of Quicksort

$T(n)$ = runtime for sorting n elements $(n = r - l + 1)$

$T(n) \in O(1)$ if $n \leq 1$

$T(n)$ = T(size of left subarray) + T(size of right subarray) + $T(n)$ if $n \geq 2$

- for easier calculation,
  - let c_1, n_0 be such that everything except the recursive calls takes take time $\leq c_1 \cdot n$ for all $n \geq n_0$
  - let $c_2 = \max\{T(0), T(1), T(2), \ldots, T(n_0)\}$
    $c = \max\{c_1, c_2\}$, then

$T(n) \leq c_2 \leq c$ (if $n \leq 1$)

$T(n) \leq c_2 \leq c \leq c \cdot n \leq T$(size of left array) + T(size of right array) + $c \cdot n$ (if $2 \leq n \leq n_0$)

$T(n) \leq T$(size of left array) + T(size of right array) + $c_2 \cdot n$
$\leq T$(size of left array) + T(size of right array) + $c \cdot n$ (if $n > n_0$)

- the recursion becomes

$T(n) \leq c$ (if $n \leq 1$)

$T(n) \leq T$(size of left array) + T(size of right array) + $c \cdot n$ (if $n \geq 2$)

- Worst case

$$T^{worst}(n) \leq \max_{0 \leq i \leq n-1} \{T(i) + T(n+i-1)\} + c \cdot n$$

- claim, $T^{worst}(n) \leq c \cdot \frac{n(n+1)}{2} \in O(n^2)$

  - proof by induction on n

$$T^{worst}(n) \leq \max_i \{T(i) + T(n-i+1)\} + c \cdot n$$
$$\leq c \cdot \max_i \left\{ \frac{i(i+1)}{2} + \frac{(n-i-1)(n-i)}{2} \right\} + c \cdot n$$

- with calculus, the maximum occurs at boundaries ($i=0$, $i=n-1$)

$$\leq c \cdot \frac{(n-1)n}{2} + c \cdot n = c \cdot \frac{n(n+1)}{2}$$

- on some examples, quicksort takes $\Omega(n^2)$ time
  - presume we have pivot a leftmost element
    - if A is sorted, then with this pivot rule, we always end with $i = 1$
      $\Rightarrow$ left subarray has size 0
      $\Rightarrow$ right subarray has size $n-1$
    $\Rightarrow T(n) = T(n-1) + c \cdot n$
    $= T(n-2) + c(n-1) + c \cdot n$
    $= \dots = T(1) + c \cdot 2 + c \cdot 3 + \dots + cn \in \Omega(n^2)$

- Best case

$$T^{best}(n) = \min_{0 \leq i \leq n-1} \{ T(i) + T(n-i-1) \} + c \cdot n$$

$$\leq T\left(\frac{n-1}{2}\right) + T\left(\frac{n-1}{2}\right) + c \cdot n$$
$$\approx 2 \cdot T\left(\frac{n}{2}\right) + c \cdot n \in O(n \log n)$$
(as in mergesort)

- we can show that this is tight

- Average case

$$T^{avg}(n) = \frac{\sum_{i=0}^{n-1} \left(\{ T^{avg}(i) + T^{avg}(n-i-1) \} + c \cdot n \right)}{n}$$

$$= \frac{1}{n} \cdot \sum_{i=0}^{n-1} T^{avg}(i) + \frac{1}{n} \sum_{i=0}^{n-1} T^{avg}(n-i+1) + c \cdot n$$

$$= \frac{2}{n} \sum_{i=0}^{n-1} T^{avg}(i) + c \cdot n$$

- claim, $T^{avg}(n) \leq \begin{cases} D & \text{if } n = 0, 1 \\ D \cdot n \cdot \ln(n) & \text{if } n \geq 2 \end{cases}$
  where $D = \max \{ T(0), T(1), T(2)/2 \cdot \ln(2), 10c \}$ Hilroy
- proof by induction on n

- base case
  $n = 0, 1, 2$
- the claim holds for the choice of D
- induction step

$$T^{avg}(n) \leq \frac{2}{n} \sum_{i=0}^{n-1} T^{avg}(i) + c \cdot n$$

$$= \frac{2}{n} \left(T(0) + T(1) + \sum_{i=2}^{n-1} T^{avg}(i) \right) + c \cdot n$$

$$\leq \frac{2}{n} \sum_{i=2}^{n-1} D \cdot i \cdot \ln(i) + \frac{2}{n} \left(\underbrace{T(0)}_{\leq D} + \underbrace{T(1)}_{\leq D} \right) + c \cdot n$$

$n \geq 3$

$$\leq \frac{2D}{n} \sum_{i=2}^{n-1} i \cdot \ln(i) + \frac{4D}{3} + c \cdot n$$

$$\leq \frac{1}{2} n^2 \ln(n) - \frac{1}{4} n^2$$

$$\leq \frac{2D}{n} \cdot \frac{1}{2} n^2 \ln(n) - \frac{2D}{n} \cdot \frac{n^2}{4} + \frac{4D}{3} + c \cdot n$$

$$\leq D \cdot n \cdot \ln(n) - \frac{D}{2} n + \frac{4D}{9} \cdot n + c \cdot n$$

$$\leq D \cdot n \cdot \ln(n) - \frac{D \cdot n}{18} \leq 0$$

- so $T^{avg}(n) \leq D \cdot n \cdot \ln(n) \in O(n \log n)$

* Announcements
- get going on A2

* Randomization
- we had some algorithms with bad worst-case runtime, but good on the average instance (presuming that all instances actually happen)
- to avoid bias among the instances, use randomization
  - if we choose a pivot p to be a random index in $\{2, \ldots, r\}$, then all indices are equally likely

$$P(\text{pivot uses the } i^{th} \text{ smallest element}) = \frac{1}{n}$$

- Runtime of randomized quicksort

$$T(n) = \overset{?}{} \quad T(i) + T(n-i-1) + c \cdot n$$

↑ compute expected runtime

$$T^{exp}(n) = \sum_{\text{random choices } R} P(R \text{ was taken}) \cdot \text{runtime with choice } R$$

- for quicksort

$$T^{exp}(n) = \sum_{\substack{\text{all possible} \\ \text{indices } i \text{ of pivot}}} P(i \text{ was the pivot index}) \cdot \text{runtime if pivot index was } i$$

$$= \sum_{i=0}^{n-1} \frac{1}{n} \left(T(i) + T(n-i-1) + c \cdot n \right)$$

$$= \left(\frac{2}{n} \sum_{i=0}^{n-1} T(i) \right) + c \cdot n \quad \in O(n \log n)$$

Andy Yin
Therese Biedl
CS 240
Lecture
Jan 29, 2015

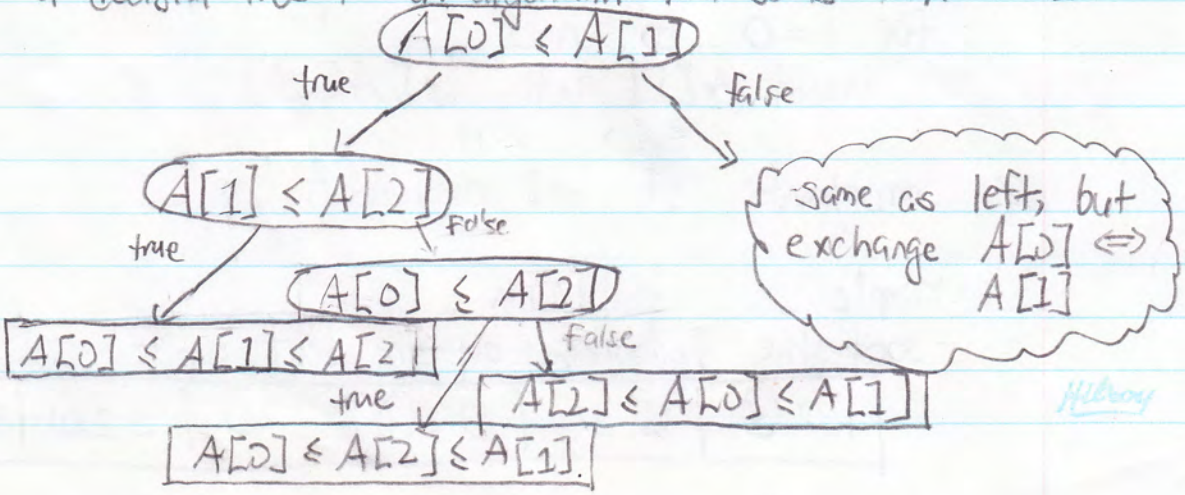
* Information

Lower Bounds for sorting	[GT 4.3]
Bucket sort	[-]
Count sort	[S 6.10]
Radix sort	[S 10.3, 10.5]

* Bounds on Sorting
  - we will show that every comparison-based sorting algorithm takes $\Omega(n \log n)$
  - we will see that some sorting algorithms takes $O(n)$ time, if input number are not too big

  - comparison-based algorithms
    - uses key comparisons, ie "$A[i] \leq A[j]$"
    - not allowed to use anything else about items in A.

* Decision Tree
  - a way to describe any comparison-based algorithm
  - example
    - a decision tree for an algorithm that sorts $A[0 \ldots 2]$:



- worst-case number of comparisons
  = length of longest path from root to leaf
  = height of decision tree

- theorem
  - any comparison-based algorithm A for sorting uses $\Omega(n \log n)$ comparisons
  - proof
    - since A is comparison-based, we can express it through a decision tree T
    - for n items to be sorted, the decision tree has greater/equal n! leaves (at least one leaf per permutation)
      - exercise, any binary tree of height h has greater/equal 2^h leaves
      - to reformulate this, any binary tree with l leaves has height greater/equal to $\log(l)$
        - therefore, height(T) $\geq \log(n!)$

$$= \log(n(n-1)(n-2)(n-3)\cdots 2\cdot 1)$$
$$\geq \log(n) + \log(n-1) + \log(n-3) + \ldots + \log(n/2)$$
$$\geq n/2 \cdot \log(n/2) = \frac{n}{2}\log(n) - \frac{n}{2} \in \Omega(n \log n)$$

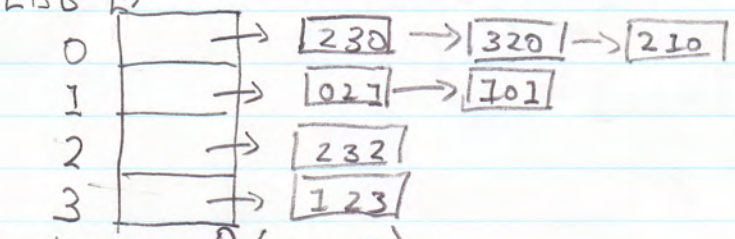
* Algorithms that sort integers (in a small range)
  - Bucket sort
  - assume A[i] is an integer in $\{0, \ldots, R-1\}$
  - create R empty lists ("buckets") L[0 .. R-1]

```
for i = 0 to n-1
    insert A[i] into L[A[i]]
                ∈ {0, ..., R-1}
concatenate lists and copy to A
```

- example
  - sort the following by the last digit

123③	230①	021①	320①	210①	232②	101①

Lists L:



- runtime: $O(n+R)$
- stable, equal-value items stay in same relative order
  - can we avoid lists?

- Key-index sorting
  - initialize array count[0,..., R-1] to all zeros

```
for i=0 ... n-1 {
    count [A[i]] ++
}
```

  - initialize array position [0,..., R-1] to all zeros

```
for j=1 ... R-1 {
    position[j] = position[j-1] + count[j-1]
}
```

- example
  - sort by last digit

123 ③	230 ⓪	021 ①	320 ⓪	210 ⓪	232 ②	101 ①

count

0	3
1	2
2	1
3	1

- length of L[i]

pos

0	~~0~~	~~1~~ 2
1	~~2~~	4
2	5	
3	~~6~~	7

- where will L[i] start

230	0	} L[0]
320	1	
	2	
021	3	} L[1]
	4	
	5	} L[2]
123	6	} L[3]

- initialize new array B[0,..., n-1]

```
for i=0,..., n-1 {
    value = A[i]    // the key of A[i]
```

```
    new_index = position[value]
    B[new_index] = A[i]
    position[value] ++
  }
  copy B to A
```

* Announcements
  - OPD paperwork due ASAP

* Sorting Huge Numbers
  - consider two numbers 230146819
    234067027
    - sort by comparing digits
    - assume all numbers have the same number of digits
    - sort numbers by leading digits
      - splits numbers into groups
      - sort each group recursively by next digit

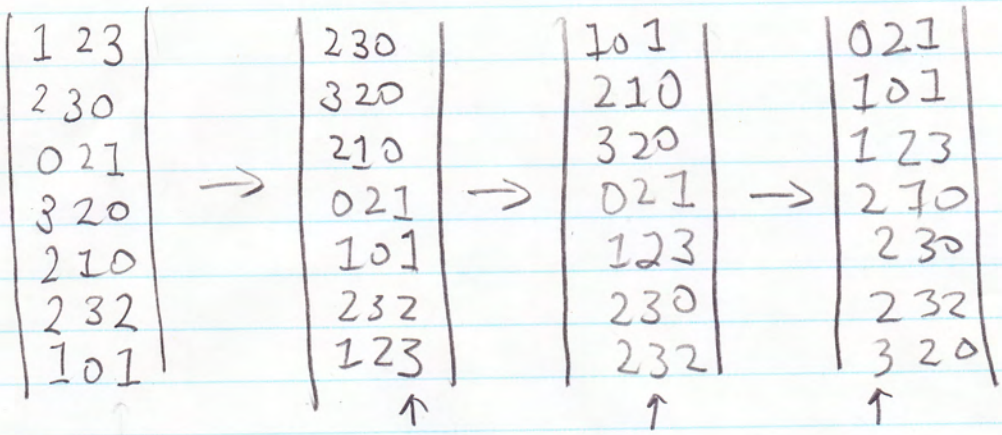
    - this is called MSD-radix-sort (most significant digit)

  - LSD-radix-sort(A)
    //pre: A contains m-digit numbers with digits in $\{0, \ldots, R-1\}$
    for d = m down to 1
      sort A by the d^{th} significant digit with stable sorting algorithm

  - runtime: $O(m \cdot (n+R))$ (if we are using key-index sorting)
    R = radix, typically R = 2, 10, 256, 26
    m = number of digits, typically m = 32 (typical word)

  - example
    - consider an array

123	230	021	320	210	232	707


123
230
021
320
210
232
101
230
320
210
021
101
232
123
101
210
320
021
123
230
232
021
101
123
210
230
232
320

Andy Yin
Youcef Tebbal
CS 240
Tutorial
Feb 2, 2015

*Example
- let A = [(0,a), (1,a), (0,b), (2,a), (0,c), (1,b), (4,a), (1,c), (0,d), (4,b)]
C = [0,0,0,0,0] → [4,3,1,0,2]
L = [0,0,0,0,0] → [0,4,7,8,8]
B = [(0,a), (1,a), (0,b), (2,a), (0,c), (1,b)]

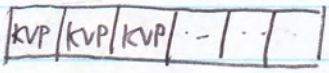
Andy Yin
Therese Biedl
CS 240
Lecture
Feb 3, 2015.

*Information
ADT Dictionary [S 12.1]
Binary Search Trees [S 12.5]
AVL-trees [GT 3.2]

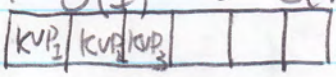
* ADT Dictionary
-store key-value pairs (KVP) as (key, value)
-insert (key, value)
-search (key) - returns KVP if in dictionary
-delete (key)

-assumptions
-all keys are distinct
-each KVP takes constant space
-keys can be compared in constant time

-trivial implementations
-unsorted array / list



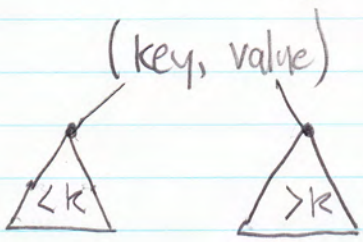
- insert: O(1)
- search: O(n)
- delete O(search) + O(1) → O(n)



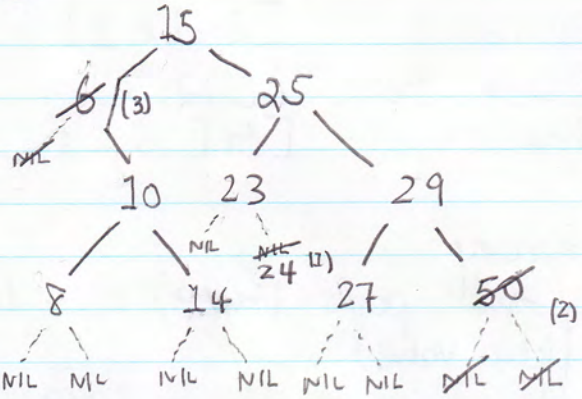
-sorted array
- insert: O(n)
- search: O(log n) (by binary search)
- delete: O(n)

*Binary Search Tree
-binary tree
-keys in the left subtree are smaller than key at root

- keys in right subtree are bigger



-example

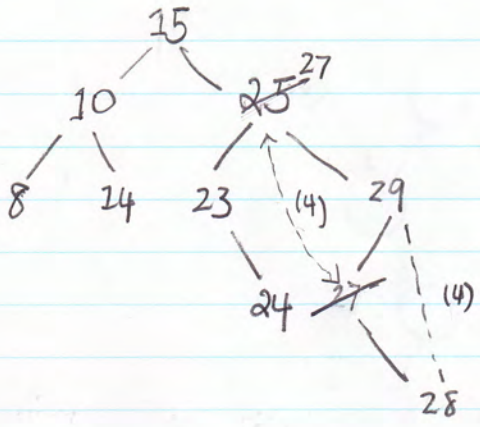


(1) insert(24)
(2) delete (50)
(3) delete (6)

-search
if tree is non-empty
from root:
if key is stored here, return KVP
else recurse in correct subtree
else return "not found"

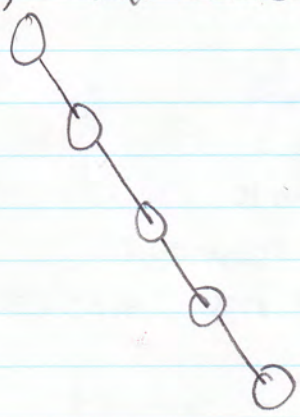
-insert
do a search, then insert (key, value) at empty subtree where search stopped

-delete
if key is at leaf, delete leaf
if node with key has exactly one child, bring subtree of child up
if node with key has two children, swap key with successor and delete successor
(the successor the left-most key after the current node)



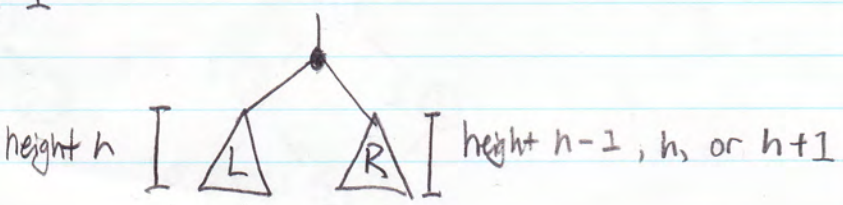
(4) delete (25)
successor 27

-the runtime has O(height) for all operations
height $\in \Omega(\log n)$, but some trees has height $n-1$

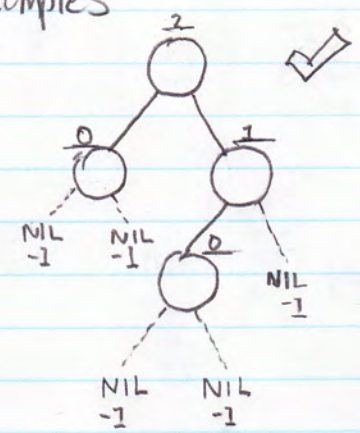


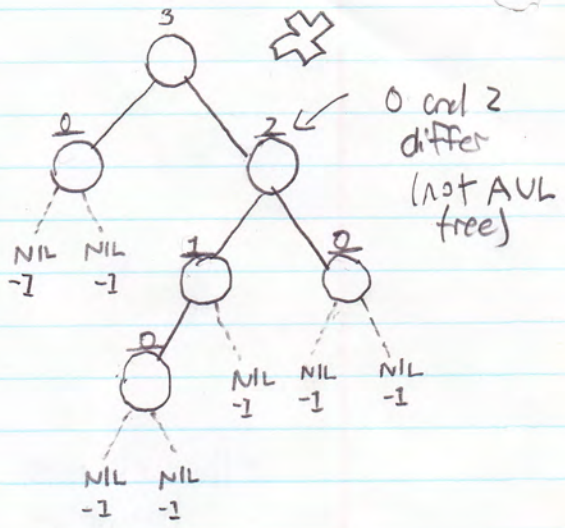
*Balanced Search Trees
- impose additional conditions on tree
-show we can do insert/delete while maintaining the conditions and not spending more than O(height) time
-show the height is $O(\log n)$ under those conditions

*AVL-trees (1962)
-structural conditions
-for any node, the heights of the left and right subtree differ by at most 1



-examples

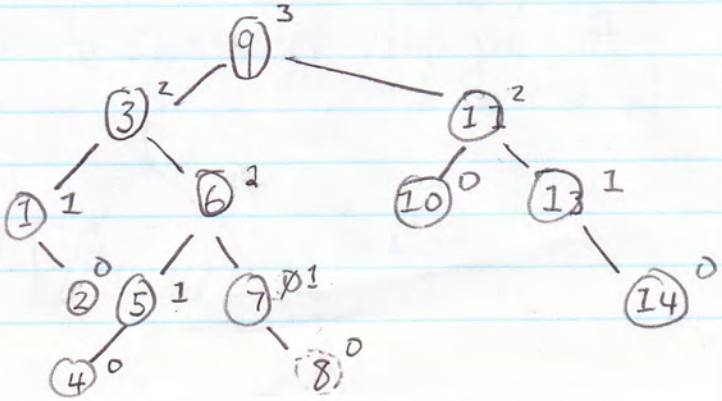




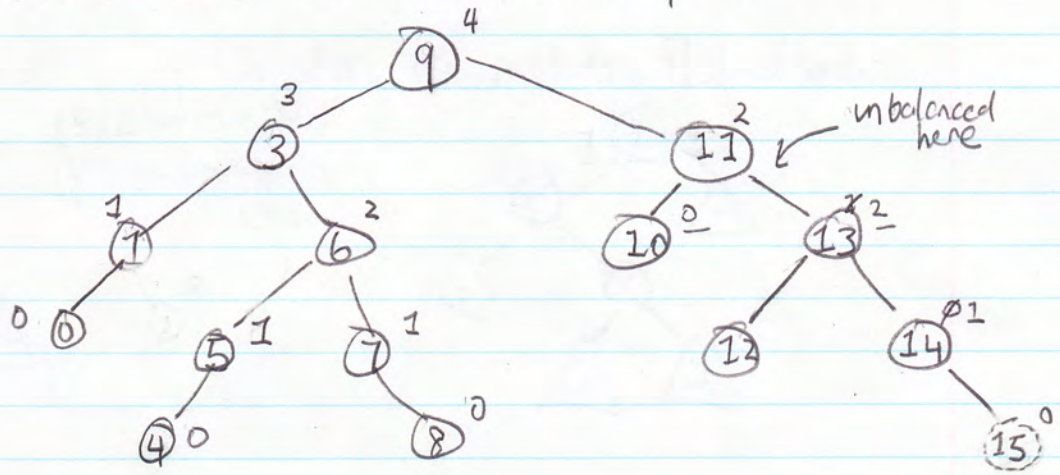
*Announcements
- this Friday's office hours maybe late or cancelled
- A1 remark requests (Danny)
- A2 due tomorrow
- Module 4 has some changes

*AVL trees - continued
- store like a binary tree, (KVP, left child, right child)
- stores the height of the subtree at the node at each node

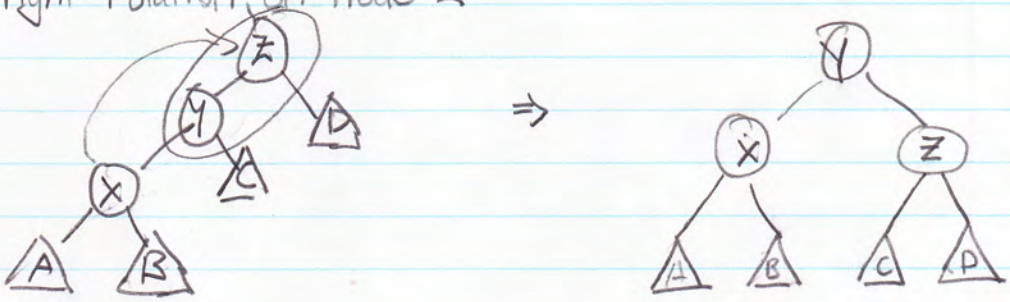
- dictionary operations on AVL-trees
  - search: exactly as for binary search tree
  - insert: find the empty subtree where we should insert
    insert there
    now check whether still AVL-tree and rebalance if not



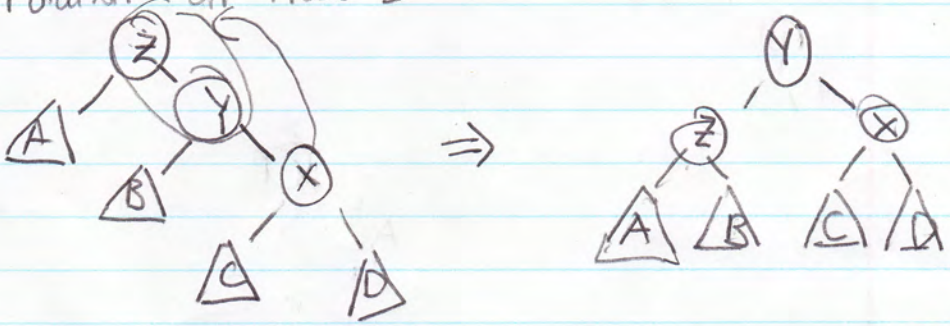
- go back up from new node and update heights
  - compute the new height of parents
    by height(parent) = max { new height of child
    height of sibling } + 1
  - if this changes height(parent), then check whether the heights of child and sibling differ by at most 1, and rebalance if not, then recurse at parent



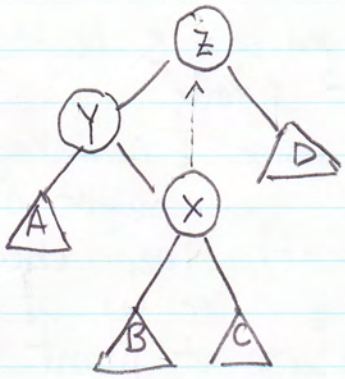
- 4 operations
  - right-rotation on node z



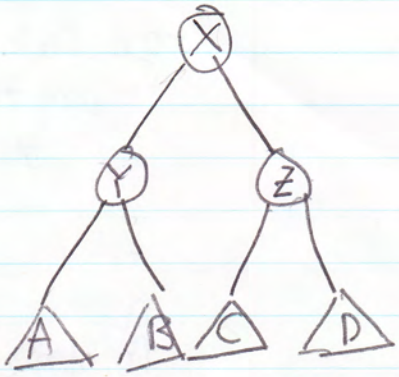
  - left-rotation on node z



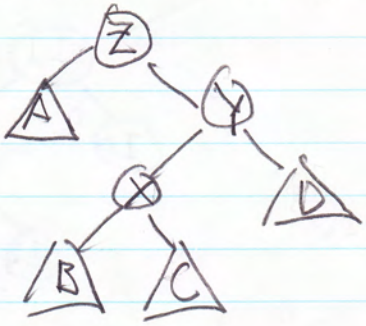
- double-right-rotation on node z



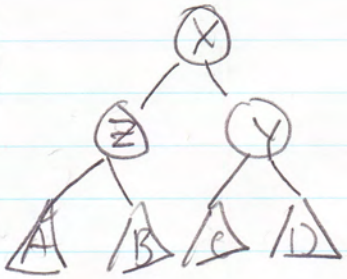
left-rotate(Y)
right-rotate(Z)
⇒



- double-left rotation on node Z



left-rotate(Z)
right-rotate(Y)
⇒

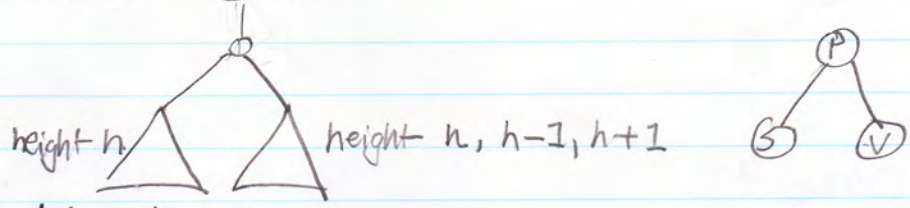


Andy Yin
Therese Biedl
CS 240
Lecture
Feb 5, 2015

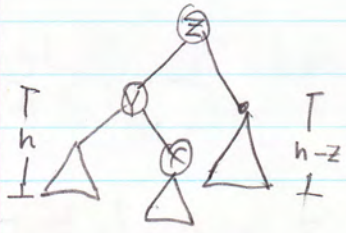
*Information
AVL Trees : Insert and Delete
AVL Trees : Height
2-3-trees
2-3-trees : Insert and Delete

[GT 3.2]
[S 13.3]

*AVL-trees - Insertion

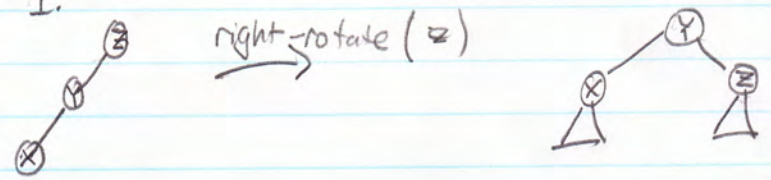



```
insert(k,w) {
  insert (k,w) at a new leaf, as in BST
  let v be the new node, v.height = 0
  while (v ≠ root) {
    p = parent(v), s = sibling(v)
    if |s.height - v.height| ≥ 2 {
      rebalance(p); break;
    } else {
      newHeight = max{v.height, s.height} + 1
      if newHeight == p.height {break}
      } else {
        p.height = newHeight, v = p}
```

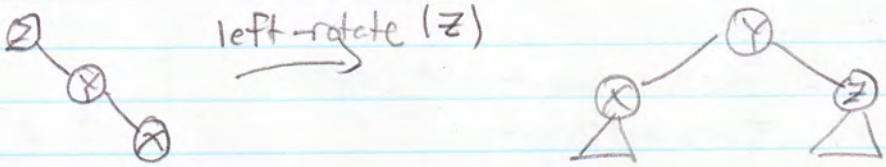



```
rebalance(z) {
  let y and x be child and
  grandchild of z in largest subtrees
  (in case of tie for x, pick x
  so that we are in case 1 or 2)
```

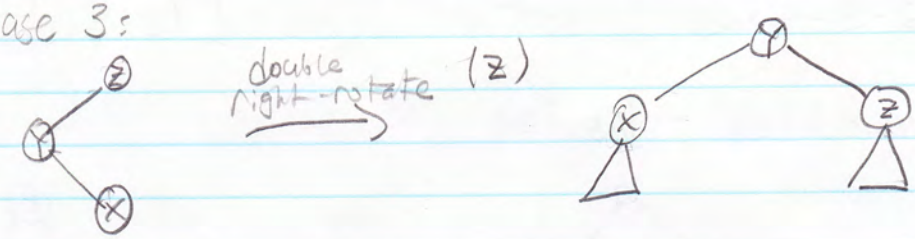
Case 1:



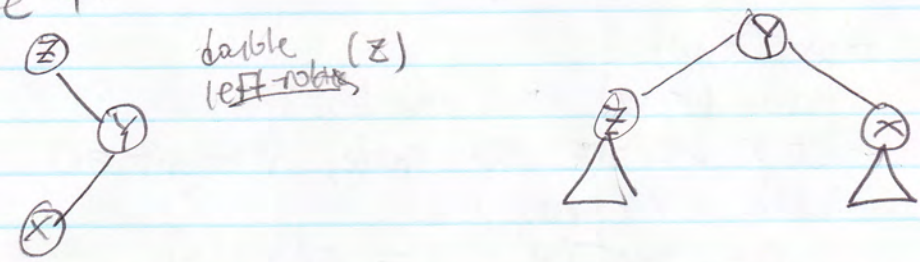
Case 2:



Case 3:



Case 4:

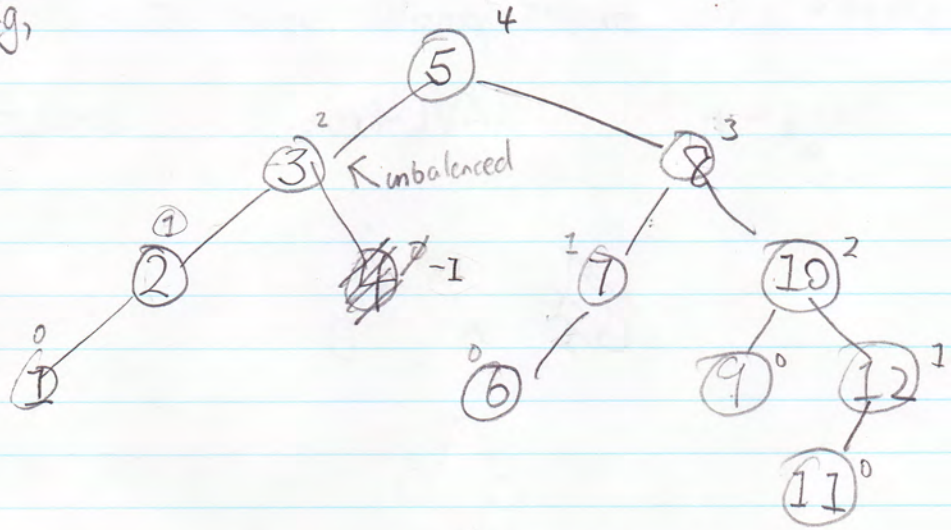


- we can show that with this algorithm, the subtree is balanced afterwards
  - if we had inserted, then the subtree is restored to its previous height
  - For insert, one rotation is enough
  - time for insert, O(height)

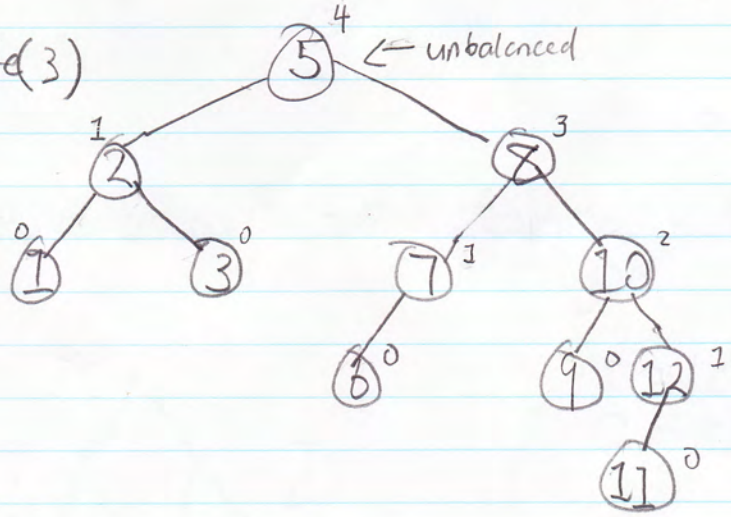
*AVL-trees - Deletion
- delete as in the BST
- let v be the node that was actually deleted
- parse upward from v, updating heights, and if a node is unbalanced, call rebalance
- after a rebalance, the subtree is balanced, but higher

up might be unbalanced

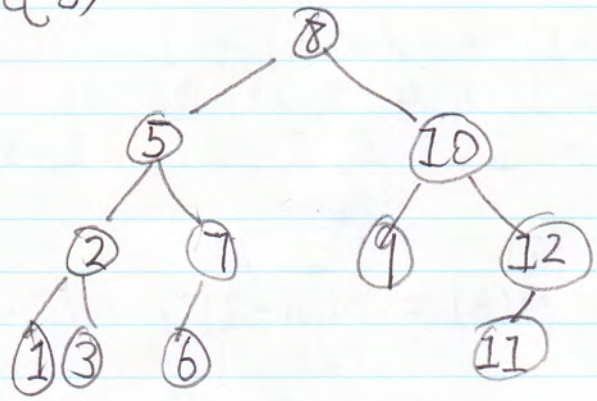
-eg,



right-rotate(3)



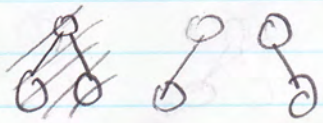
left-rotate(5)



runtime: O(height)

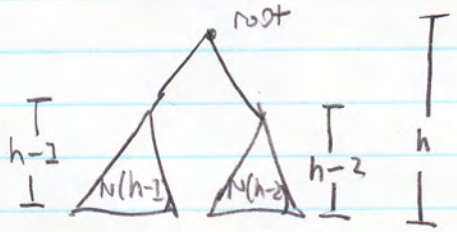
- what is the height of an AVL-tree?
  - find height $h \leq \log_c n \in O(\log n)$
    for some constant c

- So show that $n \geq c^k$ for some constant c
- change point of view, given a fixed height, what is the smallest possible number of nodes?

height h	AVL-tree	smallest number of nodes
0	•	1
1		2
2		4
3		7

$N(h)$ = smallest number of nodes in AVL-tree of height h

$N(0) = 1$
$N(1) = 2$



$L = N(h-1) + N(h-2) + 1$

$N(h) = 1, 2, 4, 7, 12, 20, 33, \ldots$

$N(h)+1 = 2, 3, 5, 8, 13, 21, 34, \ldots$

« Fibonacci numbers

- claim, $N(h) \geq (\sqrt{2})^h$
- proof, $N(h) = N(h-1) + N(h-2) + 2$
$\geq N(h-2) + N(h-2) = 2N(h-2)$
$\geq 2 \cdot 2 \cdot N(h-4)$
$\geq \ldots$
$\geq 2^i \cdot N(h-2i)$

$\geq 2^{h/2} \cdot N(0) = 2^{h/2} = (\sqrt{2})^h$ if h is even

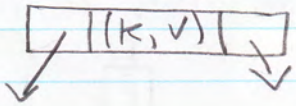
$2^{\frac{h-1}{2}} \cdot \frac{N(1)}{2} = 2^{\frac{h-1}{2}} > (\sqrt{2})^h$ if h is odd

- we now know,
  $N(h) \geq (\sqrt{2})^h$ nodes
  $\Rightarrow$ any AVL-tree of height h has $\geq (\sqrt{2})^h$ nodes
  $\Rightarrow n \geq (\sqrt{2})^h$
  $\Rightarrow h \leq \log_{\sqrt{2}} n \in O(\log n)$

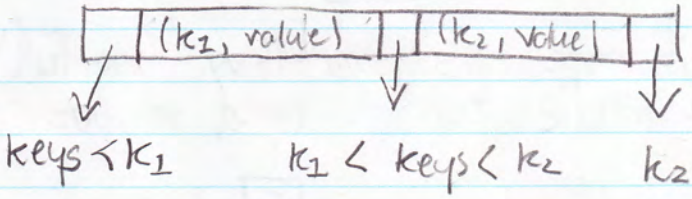
- any AVL-tree has height $\leq \log_{\sqrt{2}} n$
- insert, delete, search take $O(\log n)$ time!

*2-3-trees - Definition

- a new type of tree, a node may have 2 or 3 children
- node with one KVP, two children

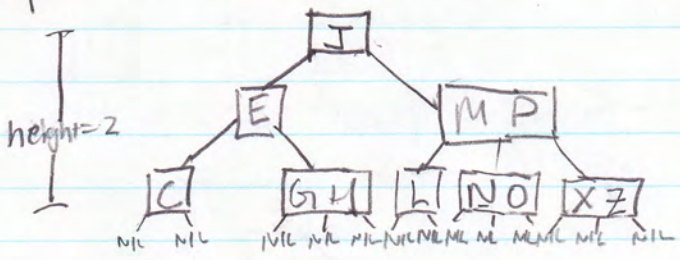


- node with two KVPs, three children



- the number of KVPs = number of children - 1 for all nodes

- restriction
  - all empty subtrees are on the same level

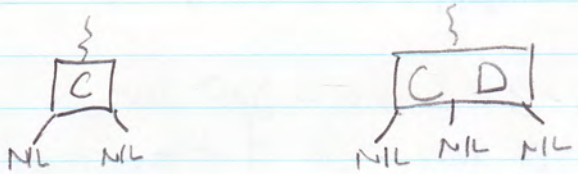


*2-3-trees - search

search(k)
- search the keys at root
- if they contain k, return that KVP
- otherwise, recurse in child where k should have been
  - or break if child is empty

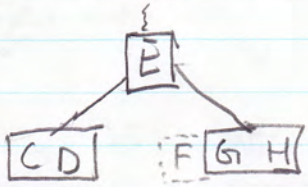
*2-3-trees - Insertion
insert(k, v)
- search for node at lowest level that would have had k
- if that node has one KVP, add second KVP to it

insert(D)

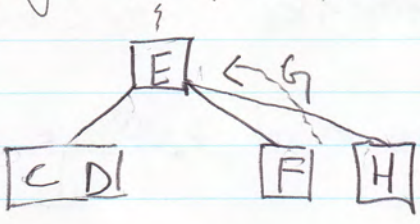


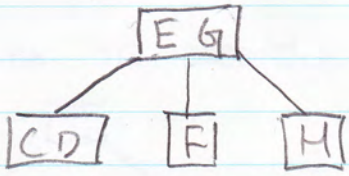
- what about if all nodes are filled?

insert(F)



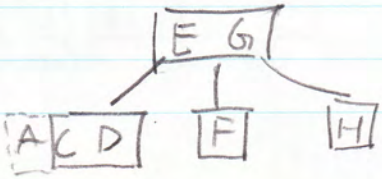
- if node has overflow (too full), split node, the middle key gets moved to parent



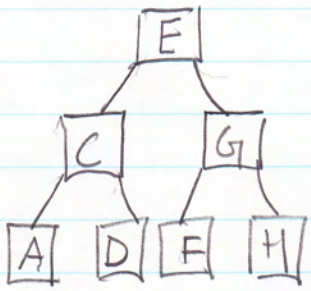


- if new parent is overfull, recurse at parent

insert(A)



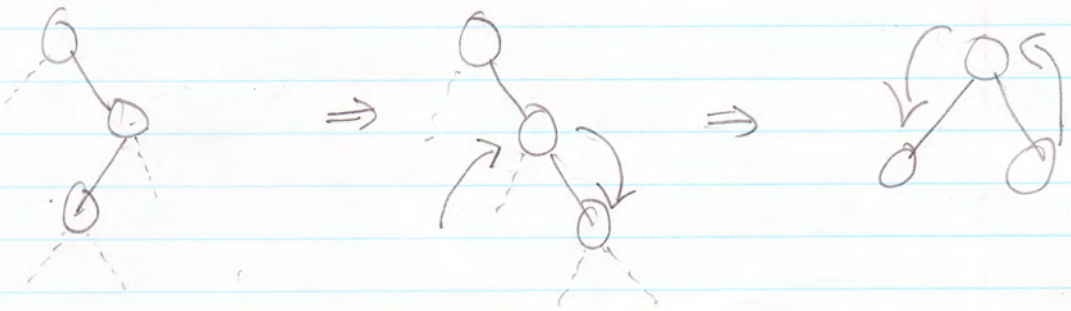
(not 2-3 tree)



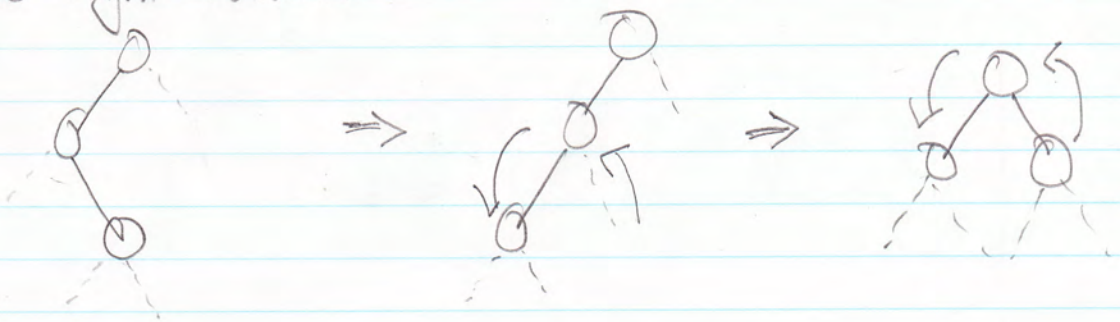
(now a 2-3 tree)

Amly Yin
Youcef
CS 240
Tutorial
Feb 9, 2015

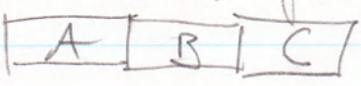
- double left rotation



- double right rotation



- intermediate siblings



- A has intermediate siblings B
- B has intermediate siblings A, C
- C has intermediate siblings B

- 2-3 Tree Deletion
  (1) pull node from intermediate siblings
  (2) otherwise recursively pull from parent down

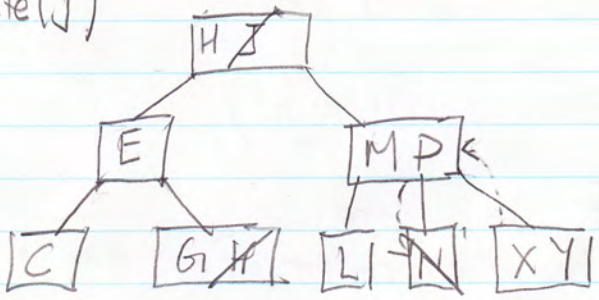
Andy Yin
Therese Biedl
CS 240
Lecture
Feb 10, 2015

*Information
2-3-trees [S 13.3]
a-b-trees
External Memory and B-trees [S 16.3, G T 14.12]

*2-3-trees
-recall last time we did 2-3 trees
-for insert, insert at the leaf
-if overfull, push KVP up (repeatedly if needed)

-deletion
-search for the KVP to delete
-swap KVP with its successor or predecessor, if it was not in the lowest level

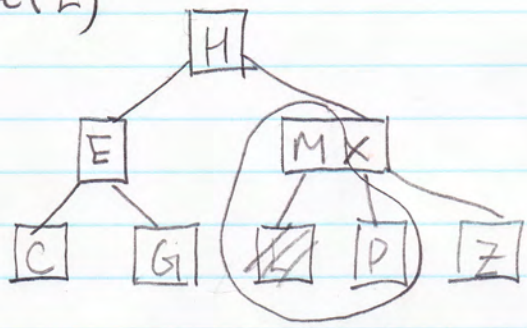
-ie, delete(j)



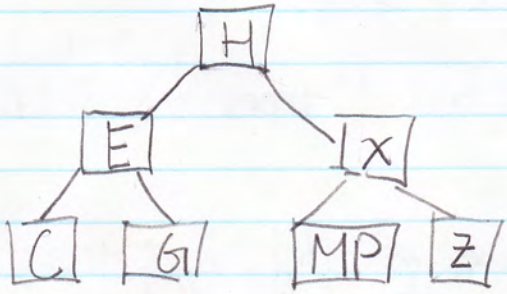
-if that leaf had sufficiently many keys, done, else, underflow this leaf
-if possible, transfer from sibling (move KVP from parent down, move KVP from sibling up), done

-ie, delete(N)

-ie, delete(L)

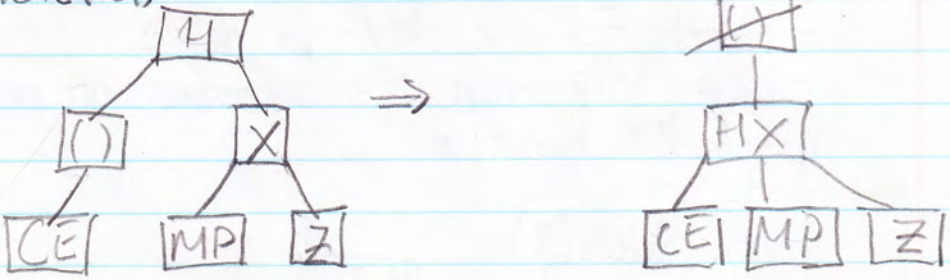


⇒



-else, fuse (combine sibling + one KVP from parent), recurse

-ie, delete(G)



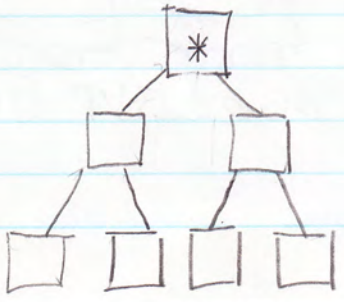
-the runtime is O(height)

*Height of 2-3-tree

-let n be the number of KVPs

-what is the smallest possible n for a given height h?

-this will give an upper bound on height later



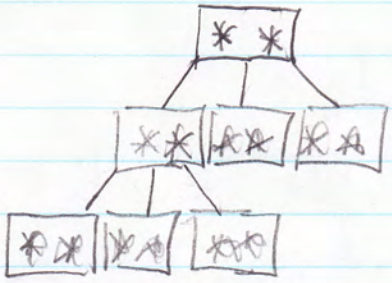
$\geq$ 1 KVP, $\geq$ 2 children

$\geq$ 2 nodes, $\geq$ 1 KVP, $\geq$ 2 children per node

$\geq$ 4 nodes, $\geq$ 1 KVP, $\geq$ 2 children per node

- so there will be at least 2^i nodes,
  total number of KVP $= 1+2+4+8+\ldots+2^i+\ldots+2^h$
  $n \geq 2^{h+1}-1$
  $h \leq \log(n+1)-1 \in O(\log n)$

- now, find a lower bound for h (by upper banding the number of KVPs)



$\leq$ 2 KVP, $\leq$ 3 children

$\leq$ 3 nodes, $\leq$ 2 KVPs, $\leq$ 3 children per node

$\leq$ 9 nodes

- in total, the number of KVPs in a 2-3-tree of height h

$$\leq \sum_{i=0}^{h} 3^i - 2 = 2 \cdot \sum_{i=0}^{h} 3^i = 2 \cdot \frac{3^{h+1}-1}{3-1} = 3^{h+1}-1$$

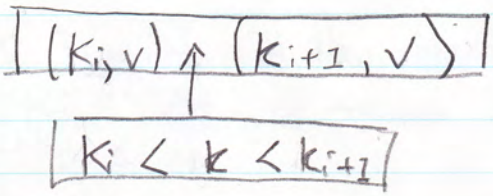
- so, $n \leq 3^{h+1}-1$
  $n+1 \leq 3^{h+1}$
  $\log_3(n+1) \leq h+1$

$$h \geq \log_3(n+1) - 1 \in \Omega(\log n)$$

- we can generalize 2-3-trees

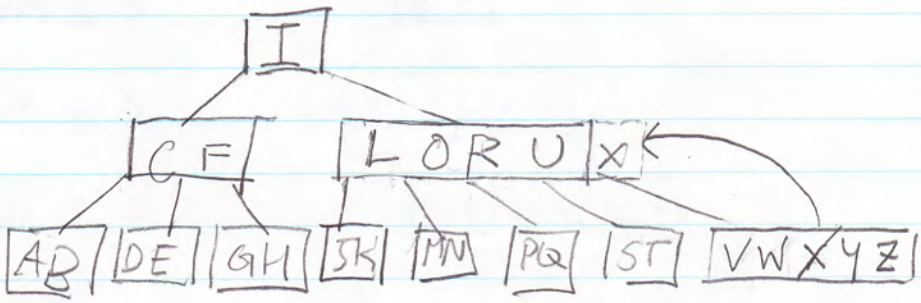
* (a, b)-trees
  - has at least a children, and at most b children
  - as with before, each node has
    number of children = number of KVP + 1
  - as with before, keys in subtree are "between" neighboring keys in the node

Hilroy

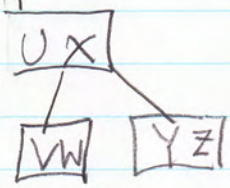


- as with before, all empty subtrees are on the same level
- every node has $\leq b$ children
- every non-root node has $\geq a$ children
  - the root has at least 2 children

- ie,



- insert exactly like before
  - insert at lowest level
  - if overfull, bring up middle KVP and recurse in parent



- works if $2 \leq a \leq \frac{b+1}{2}$

- delete works as before

- exercise
  - show height of an a-b-tree $\in O(\log_a n)$, $\Omega(\log_b n)$

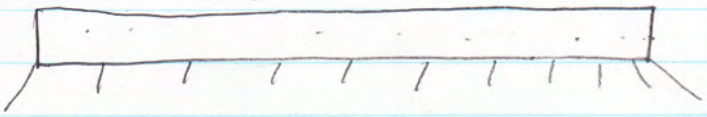
- convention
  - nodes store KVPs as sorted array
  - search: time $\underbrace{\log b}_{\text{binary search}} \cdot \underbrace{\log_a n}_{\text{height}}$

*External Memory
- the data we have doesn't fit on computer's main memory
- so use, memory tapes
  external hard drive
  cloud storage

- the difficulty lies in accessing external memory is much slower than work in internal memory
  - we will use a new computer model
    - only count how often we access external memory
  - call this the number of disk transfers
    - access could mean read/write

*Dictionaries with External Memories
- a B-tree of order M is an $[M/2]$-M-tree
  ($[M/2]$: a, M: b)

  - one node: $\leq M-1$ KVP, $\leq M$ children
    - choose M such that one node of the B-tree fits into main memory



height $\in O(\log_{M/2} n)$

- insert

$\leq 2 \cdot$ height of disk transfer

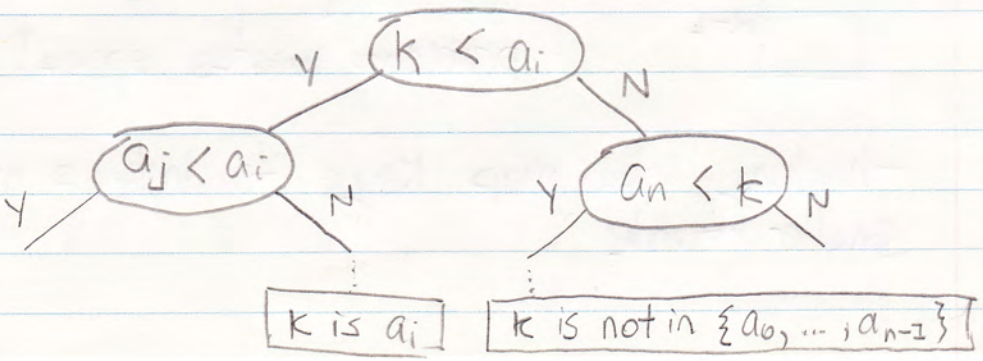
- search $\leq$ height of disk transfer
- delete $\leq 2 \cdot$ height of disk transfer

Andy Yin
Therese Biedl
CS 240
Lecture
Feb 12, 2015

* Information
  - Lower bounds for searching [-]
    Key-indexed search [S 12.2]
    Hashing with chaining [S 14.2]
    Open addressing [S 14.3, 14.4]
    Cuckoo Hashing [-]

* Lower Bounds for Search
  - recall we had balanced search trees that can implement dictionaries with $O(\log n)$ runtime for all operations

  - theorem
    - any comparison-based implementation of search(k) requires $\Omega(\log n)$ runtime
    - proof
      - use a decision tree argument
      - we can only use comparisions (between k and existing keys $\{a_0, a_1, \ldots, a_{n-1}\}$)

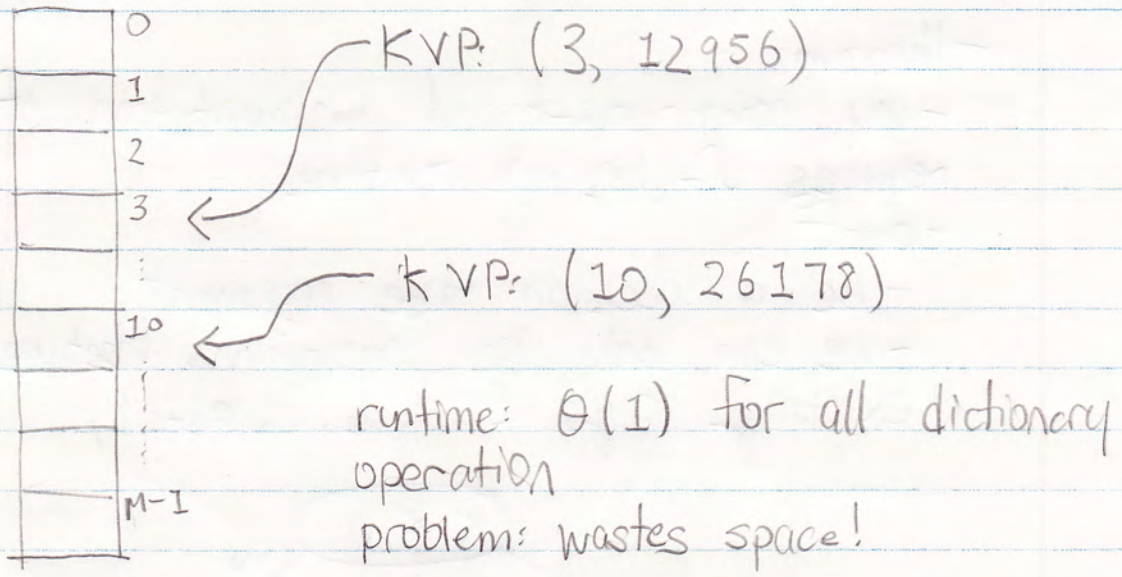


- the number of leaves is less/equal to
  n + 1
  "k is a_i" "k is not in dictionary"

- so height of decision tree $\geq$ log (number of leaves)
  $\geq \log(n+1)$
  - then in worst-case, the implementation of search used at least $\log(n+1)$ comparisons
    - so runtime is $\Omega(\log n)$

*Key-Indexed Search
- for the next three classes, we will discuss implementations of dictionaries that use values of the keys

- for key-indexed search, assume that the keys are integers in $\{0, 1, \ldots, M-1\}$, where M is the maximum key



runtime: $\Theta(1)$ for all dictionary operation
problem: wastes space!

- hashing idea: map keys to indices of a (much) smaller table

* Hashing Details
- keep an unordered array T (hash-table)
- one entry of T is known as the bucket or slot

- T has size M, but we choose the value for M
- we have a hash-function
  h: keys → {0, ..., M-1}
  such that k hashes to h(k)
  and h(k) is the hash-value of key k
- we assume that h(k) can be computed in O(1) time
- we choose the hash function (later)

* Main Idea for Hashing
- KVP (k, v) is stored at T[h(k)]

- example
  - for M = 11, let h(k) = k mod M
  - insert((44, value))
    ⇒ h(44) = 44 mod 11
    = 0
  - insert((36, value))
    ⇒ h(36) = 36 mod 11
    = 3

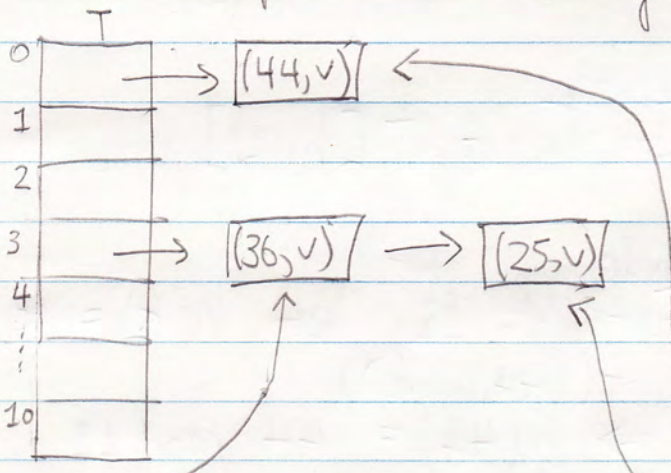
problem: multiple keys could want the same slot

- possible solutions

(1) use a linked-list at each T[i] (hashing with chaining)
(2) find an alternative slot for key k (open addressing)

* Announcements
- midterm: up to and including cuckoo hashing
- office hours: Friday 11am - 12pm, this week and next
- Final: April 17th, 9:00 am - 11:30 am

* Hashing with Chaining
- we assume T[i] stores a data structure that can hold multiple items (usually done with linked list)



insert(k, v):
add (k, v) to the list at T[h(k)]
- this takes O(1)

insert((44 value))
⇒ h(44) = 44 mod 11
= 0

insert((36, value))
⇒ h(36) = 36 mod 11
= 3

insert((25, value))
⇒ h(25) = 25 mod 11
= 3

- search(k): search through the entire list at T[h(k)]
runtime: O(1 + length of list)

- delete(k): search through the list at T[h(k)]
runtime: O(1 + length of list)

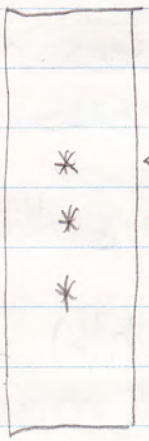
- worst case: all keys are in the same slot
  - we want a good hashing function
    uniform: P(key k hashes to slot i) = $\frac{1}{m}$
    - all slots are equally likely

- assume we have an uniform hash function, then the lists will have length
  $\frac{\text{number of keys}}{\text{number of slots}} = \frac{n}{m} = \alpha$ (load factor)
  - we control α (because we control M), but runtime is O(1) only if we keep the load factor small

* Open Addressing - General
  - for each key k, we have a sequence of slots to try: $\langle h(k,0), h(k,1), h(k,2), \ldots, h(k,M-1) \rangle$

  - a probe sequence is a sequence of indices in a hash table

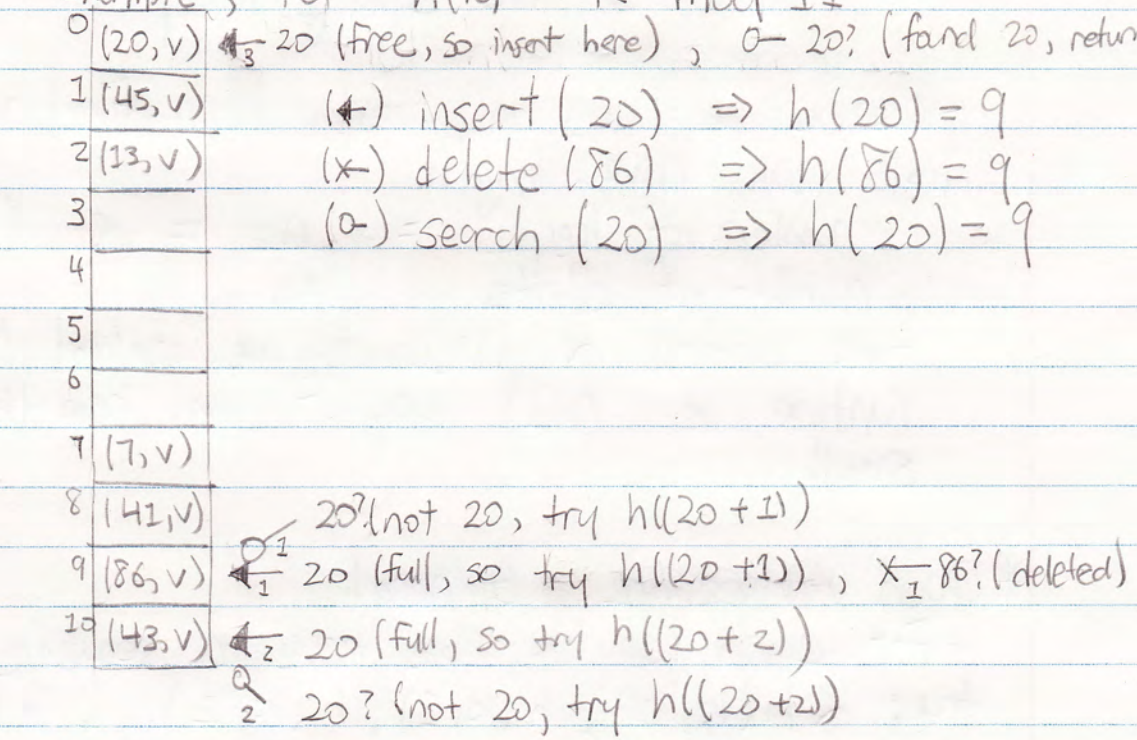
* Open Addressing - Linear Probing



$h(k,i) = i^{th}$ slot to try for insert
$= (h(k) + i) \bmod M$

```
insert (k) {
  for (i = 0, 1, 2, ...) {
    if T[h(k,i)] is empty {
      insert; break;
    }
  }
}
```

- we want a probe sequence that is a permutation of $\{0, \ldots, M-1\}$
  - try all of the slots, and we must have load factor $\alpha < 1$
  - example, let $h(k) = k \bmod 11$

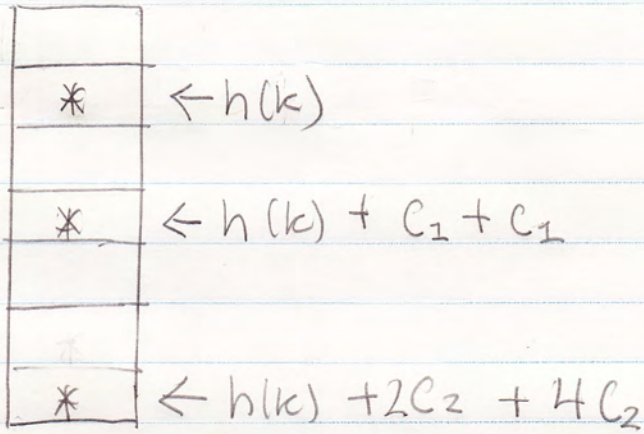


- delete(k) cannot just delete, or a later search might fail
  - lazy deletion is to replace deleted item with a special marker ν

- search(k) keeps searching if it sees ν
- insert(k) can use the slot it sees ν

- the disadvantage of linear probing is that it builds up clusters
  - ie, any key that hashes to h(k) will be hashed into the cluster, and will increase cluster size

* Other Probe Sequence Ideas
  - quadratic probing
    - ie, $h(k,i) = (h(k) + c_1 i + c_i i^2) \bmod M$
      - this function "hops around" more, but is harder to make a permutation for the probe sequence



  - double hashing
    - uses two hash functions: $h_1(k)$, $h_2(k)$ that are independent
    - ie, $h(h,i) = (h_1(k) + i \cdot h_2(k)) \bmod M$
      - we need $h_2(k) \neq 0$
    - the advantage is that there's less clustering, but we need two hash functions