Midterm #2 Review Day

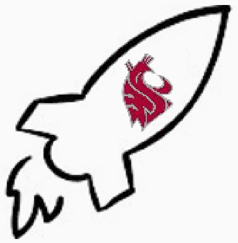
CptS 223 - Fall 2017 - Aaron Crandall

Today’s Agenda

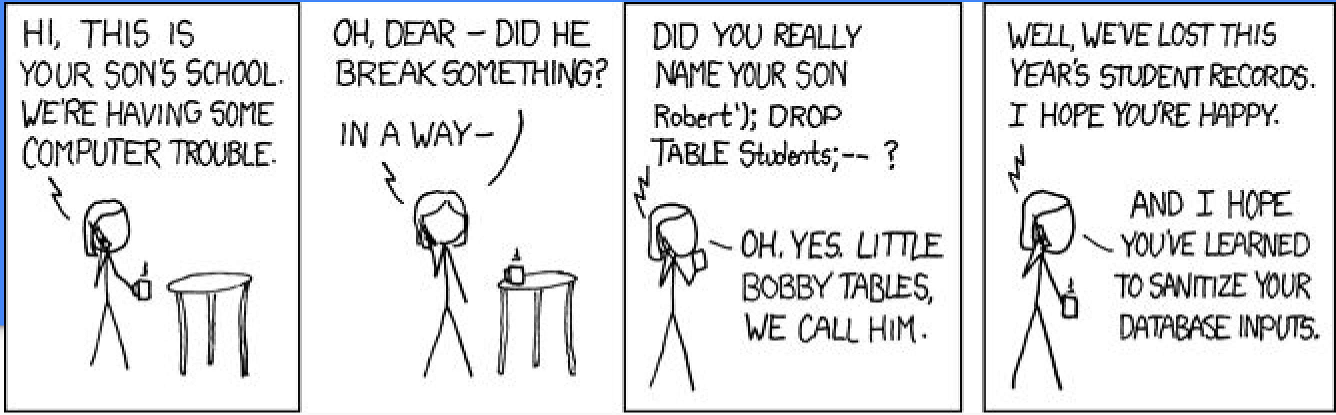
- Announcements
- Thing of the day
- Going over the sorting quiz
- Some Linux commands
- Hashing
- Heaps
- Sorting

Announcements

- Midterm on Friday
  - Hashing, Heaps, Sorting
- I’ll put HW3 solutions up at 12:01am tomorrow

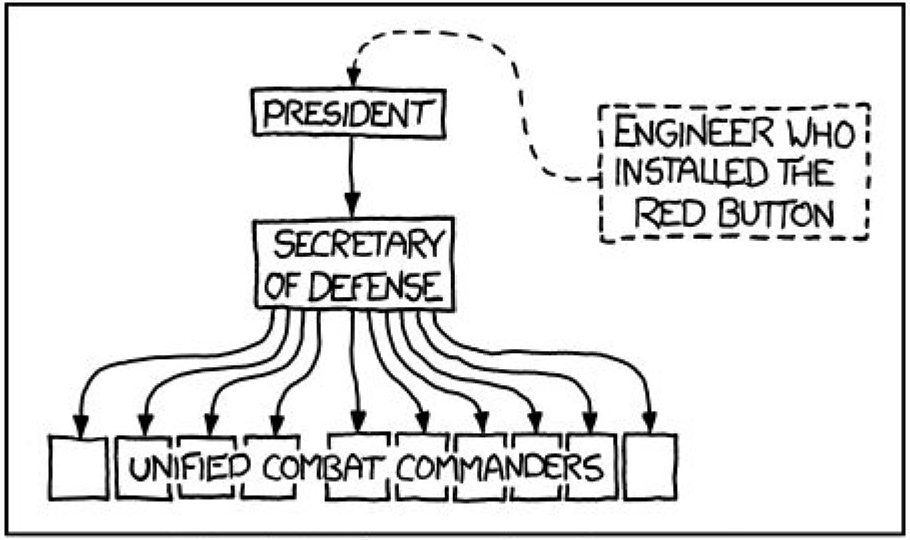


XKCD!


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY –
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Lessons:

- Users will give you all kinds of bad inputs over the years. They are not to be trusted!
- Engineers, CS, and nerds writ large ***could*** run the world. We just have better things to do.


PRESIDENT
ENGINEER WHO INSTALLED THE RED BUTTON
SECRETARY OF DEFENSE
UNIFIED COMBAT COMMANDERS

US NUCLEAR CHAIN OF COMMAND

Going over the Sorting Quiz

Some Linux Commands

wc
tail / head
grep
sed & awk
less
pstree

git

- This is a big topic and a wonderful tool once you learn the ins and outs of it!
- I use git often these days
- EECS has a git server for your use
  - Should be storing all programs there
  - *will* be using it later to collaborate on projects

I want to discuss GitHub.com for just a few moments

- A source control system - Primary location is github.com
- Provides you:
  - Version control
  - Branching
  - Merging
  - Issue tracking
  - Collaboration tools
  - Web-backends
  - Public and Private repositories (with your student account)
- Git is the #1 source control tool in industry today - your public profile as a coder should be on display, and github.com is the most common way

Midterm #2 review

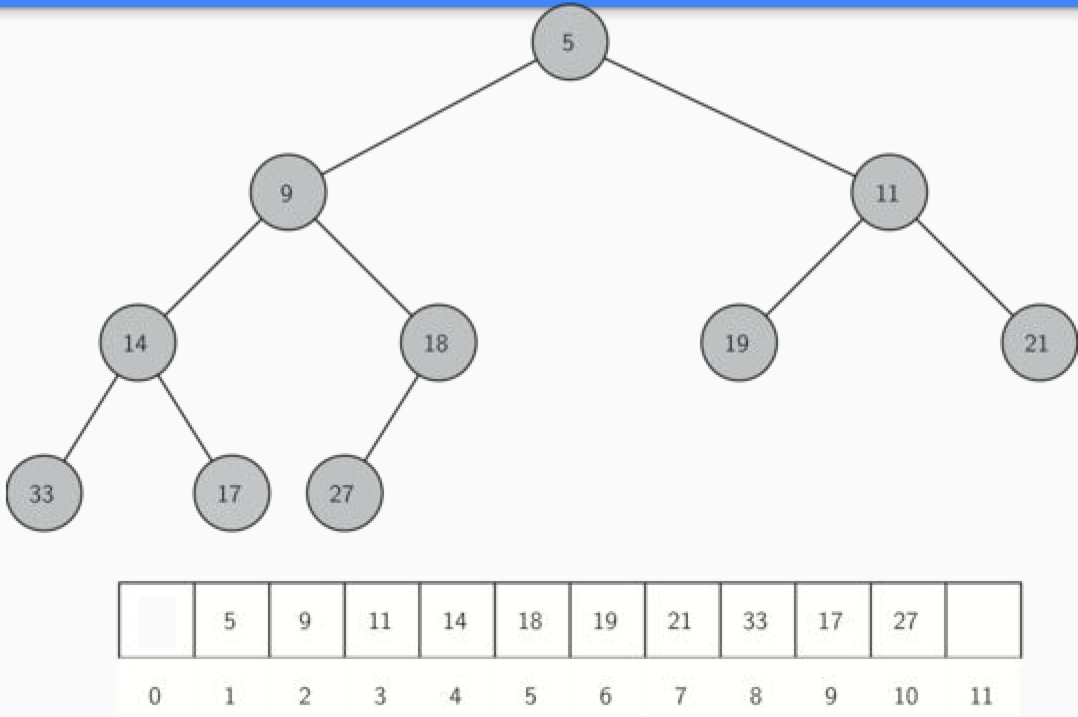
- Hashing
- Heaps
- Sorting

Part #1 - Heaps (Priority Queues)

- Efficient priority queues show up in many applications
- Only rule is: parents have to be less than all of their children
- MinHeaps and MaxHeaps - just pick the one you need
- Binary heaps (though, there’s n-ary heaps, with a 4-heap being more eff)
- Simple implementations: can be done in a vector
    - For 1-based indexes:
        - Children are: i * 2 && i * 2 + 1
        - Parent is: i/2
- Finding min/max is always O(1), merging lists into a heap are O(N) time, other operations are O(log N)

Why is this not a BST?

- No ordering to the children
- Only parent < children
- Tree is always balanced



	5	9	11	14	18	19	21	33	17	27	
0	1	2	3	4	5	6	7	8	9	10	11

Functions

- Insert: Add element to end of heap, then bubbleUp() / percolateUp()
  - is log(N) worst case, but normally only about 1.8 on average
- FindMin: Just read root of tree - is always O(1)
- DeleteMin:
  - Remove (return) root of tree
  - move last element in vector to root
  - bubbleDown() / percolateDown() from root - is log(N) worst case
- BuildHeap(List A, List B) - append two lists into a single list (call heap)
  - Then call percolateDown (adjust heap) on elements from (N/2..1)
  - Works in O(N) time since it's an average of 1.5-ish swaps each call on N/2 elements

Things to practice:

- inserting
- deleteMin
- buildHeap
- Remember the heap rules!

Heaps in the STL

- The STL has a class template for a heap
- It’s called “priority_queue” in header file queue
- Implements a max-heap rather than a min
    - Can be made min heap by changing comparison function when declaring priority_queue

Other heaps - d-heaps & leftist heaps

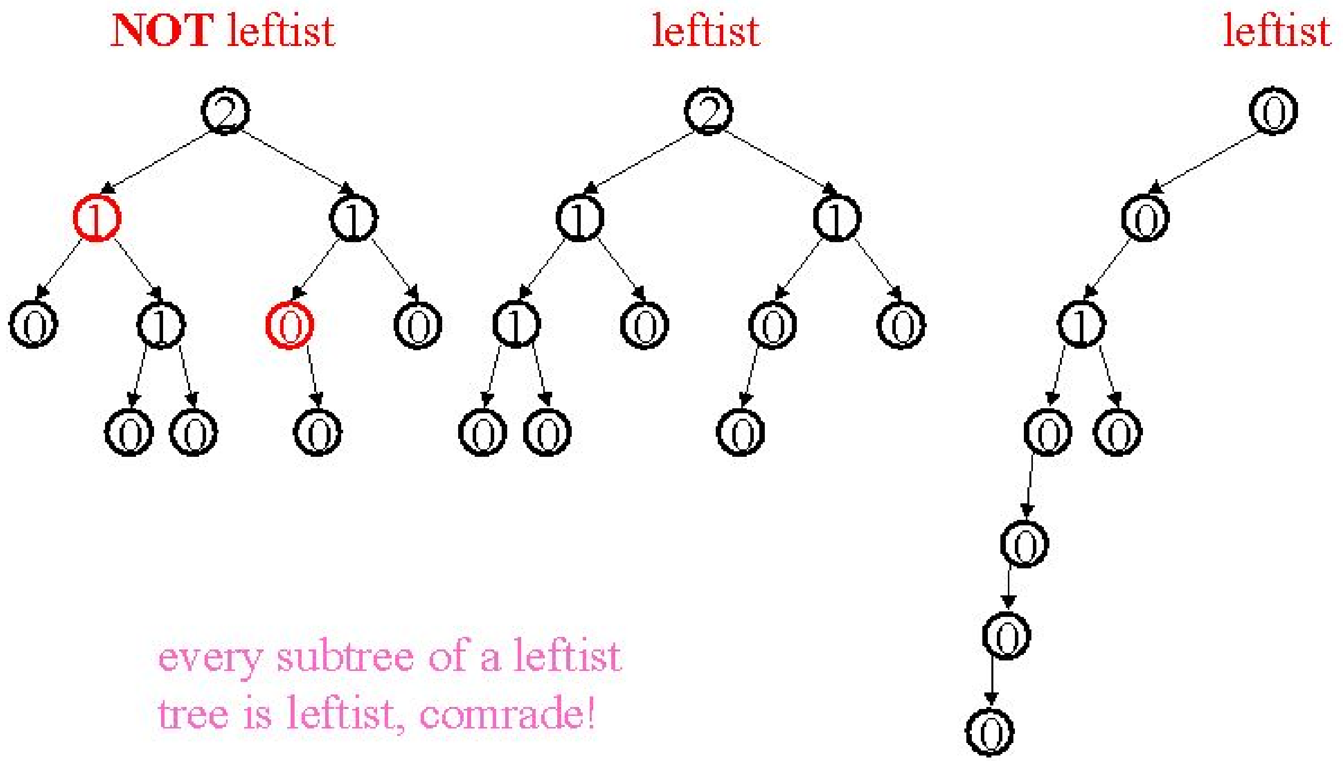
d-Heaps

- d-ary tree instead of a 2-ary (binary) tree
- Makes height log_d (N) instead of log_2 (N)
- Makes inserts: O(log_d N)
- Makes deleteMin: O(d log_d N)
- If d is not a power of 2, it's a penalty on array implementations
  - If power of 2, can use shift register multiplication for HUGE speedups!
  - C++ Syntax: << >>   -- Does binary shift of bits in an integer
- Ops are still O(log N)

Leftist Heaps

- Left child is height >= right child height
- Height of node is height of shortest child + 1
- Much like AVL tree, it keeps height values in the nodes
- Guarantees left tree is taller than right
- Merge/insert are then done on right trees
- Primarily used in situations where there’s lots of merging of heaps in your application - which can, and does, occur
    - See: distributed node computation where results come back in heaps to be merged

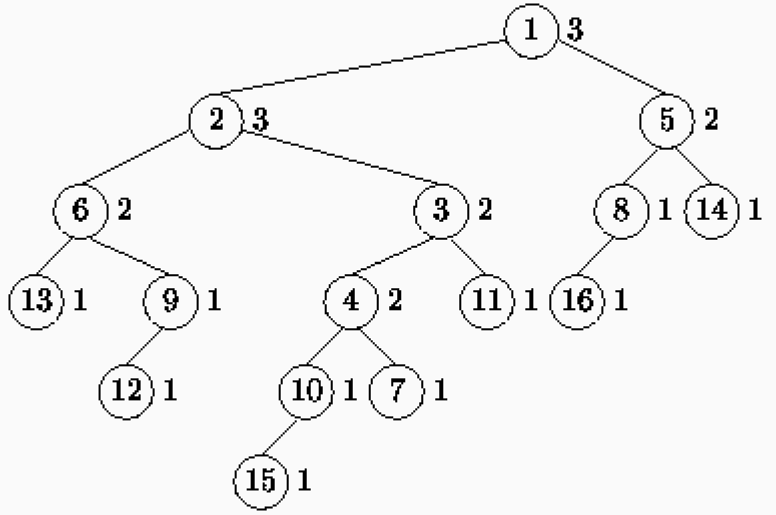
Leftist tree examples



every subtree of a leftist tree is leftist, comrade!

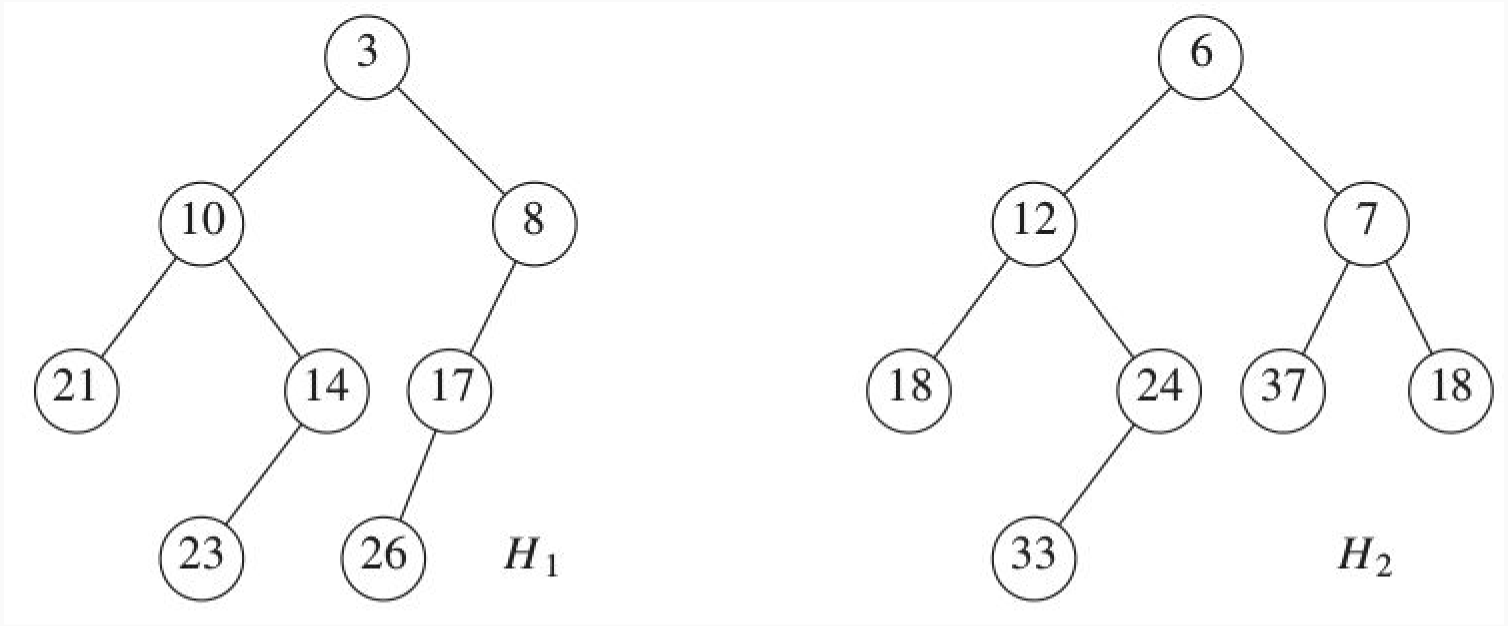
Calculating Heights of Nodes

- Height of a node is height of shorter child + 1
- Also, left tree height >= right tree height
- Biases left!

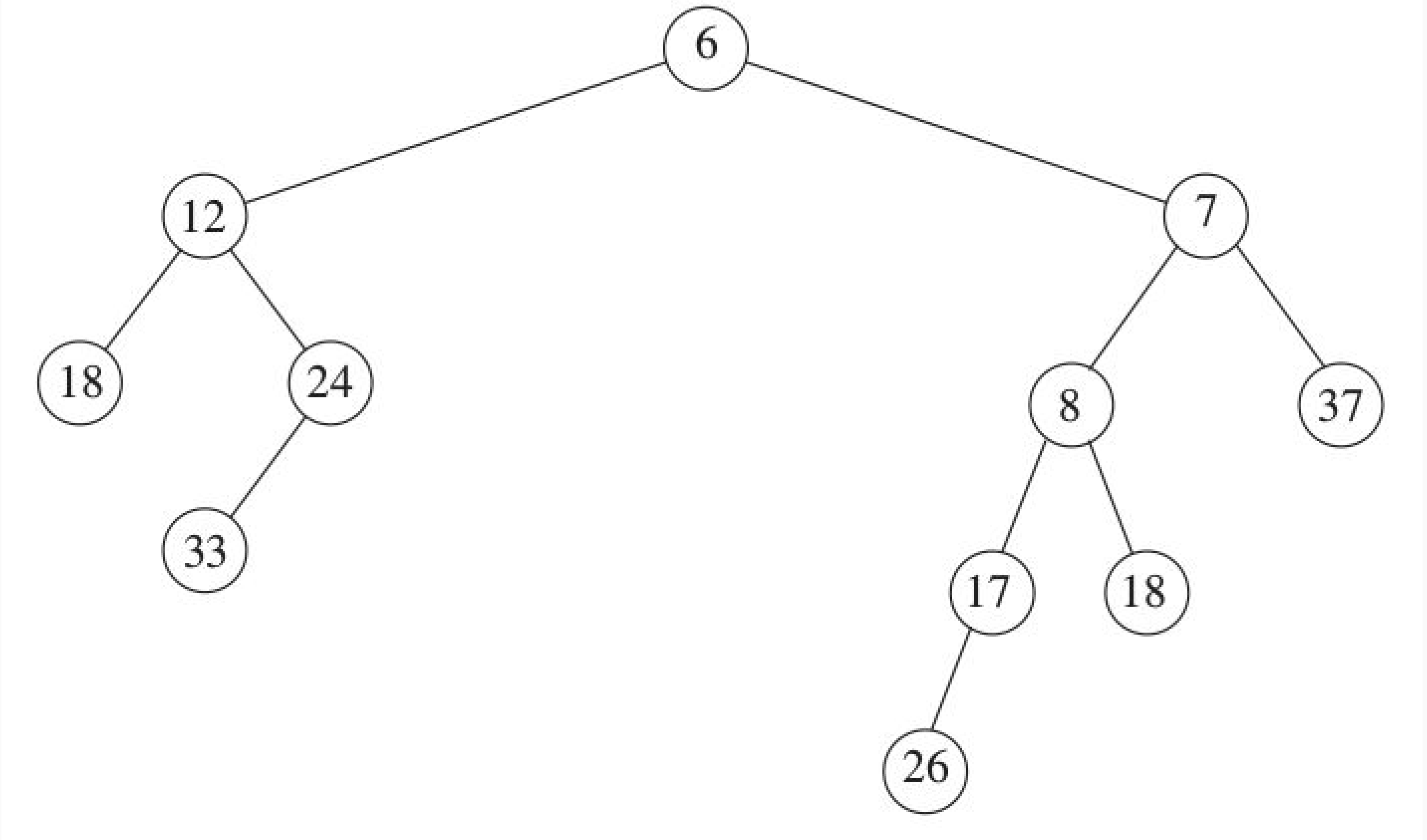


Means right tree is shorter

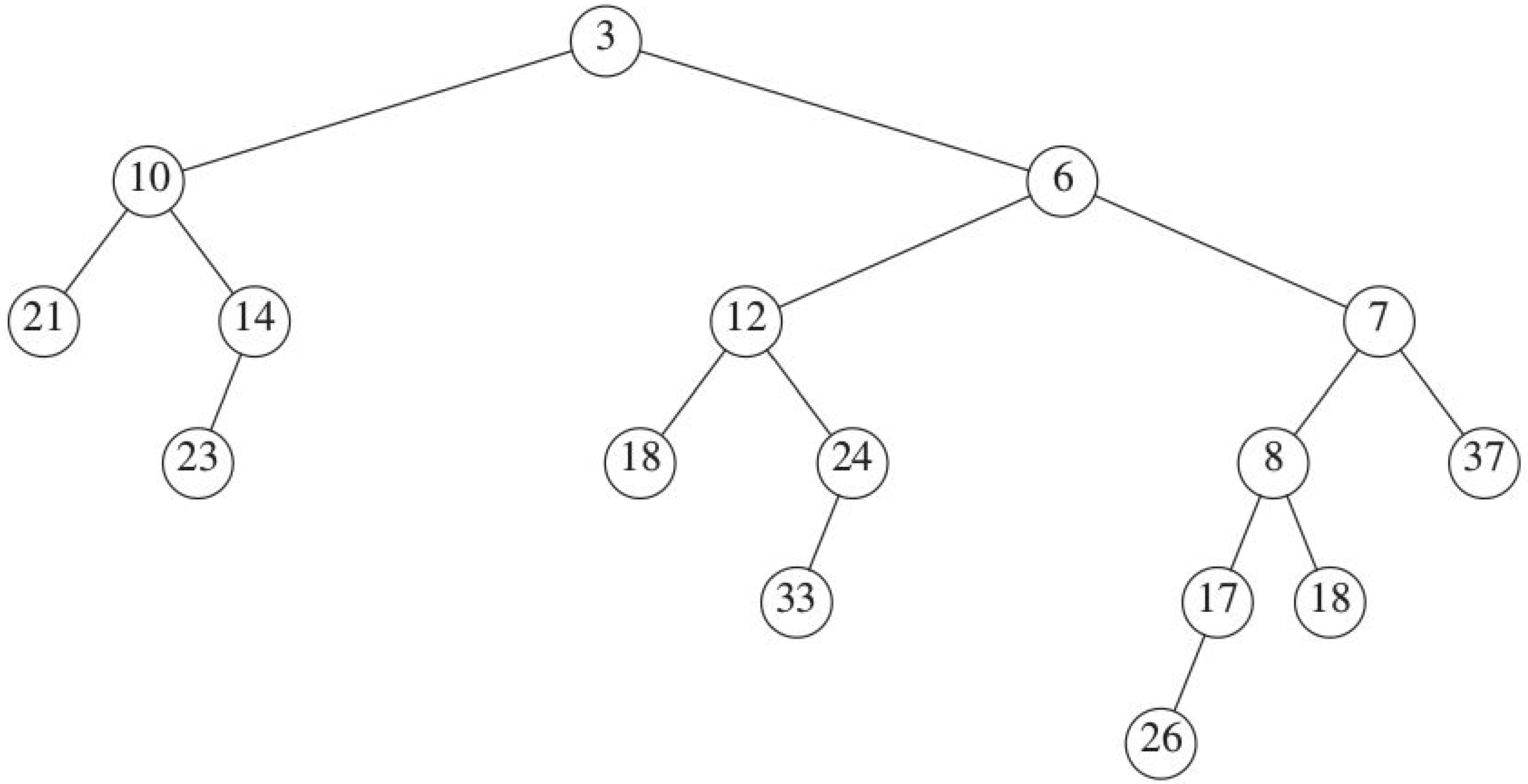
- Right path will have at most log(N+1) nodes
- This guarantees a smaller tree to work with
- Merges (Inserts are merges with a single node tree) are done on right sub tree of the tree with the smaller root with the whole other tree
- After the merge, the rest of the first tree is used as the root of a new tree
- Then, heights are re-calculated
    - If a node’s children violate the leftist rule, swap them



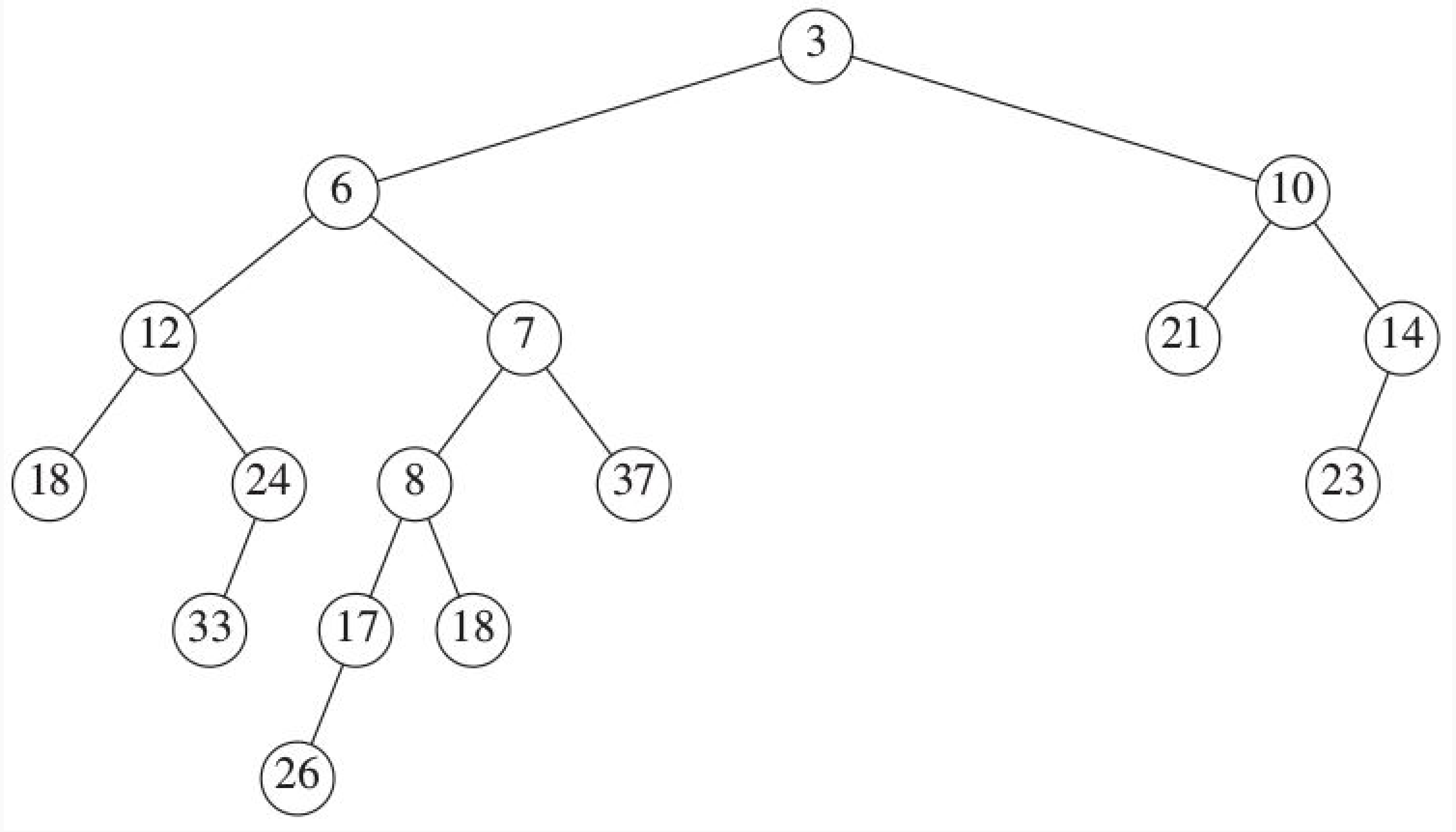
Right subtree of H1 is merged with H2
This is done recursively on each subtree as we go along



Result of top level recursive merger of subtrees



Then root and left tree from H1 are made to be the new overall root



To wrap up, the npl values of the nodes are examined and sub trees swapped as needed

Part #2 - Hashing

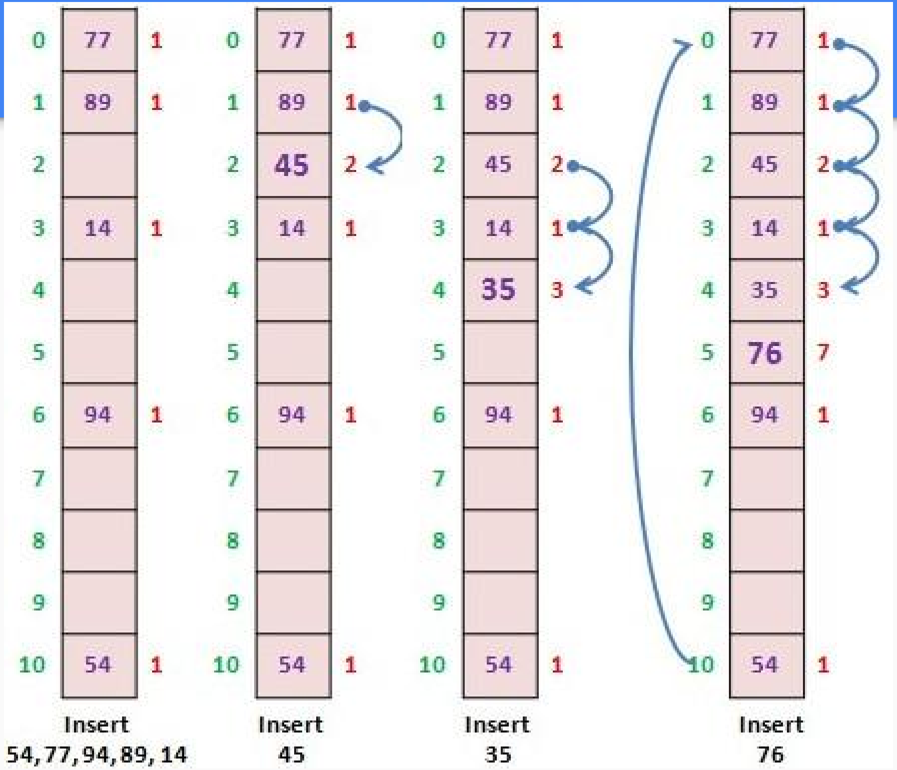
- What is the goal of using hash tables?
- Major components of a hash algorithm:
    - Hash function + Hash table + Collision resolution function
- What is a hash function?
- What is a hash table?
- What is a collision resolution function?

Collision Resolution Function

- None - Just error out if there’s no place for the new data
    - Could rehash to make more room and try again right away
- Linear probing
    - probe(i) = (hash(key) + i) % table size
- Quadratic probing
    - probe(i) = (hash(key) + i^2) % table size
- Separate chaining
    - Store buckets in linked lists - no probing
- Lazy deletion issue - probing cannot delete, only flag as “not there”
    - Things are really deleted if a rehash occurs because “not there” isn’t rehashed in again

Linear Probing

- Increment location if collision
- Works, but builds high and low density areas in the hash table
- probe(i) = (hash + i) % TableSize
    - For i in 1..TableSize
- Rehash if load factor (λ) >= 0.5

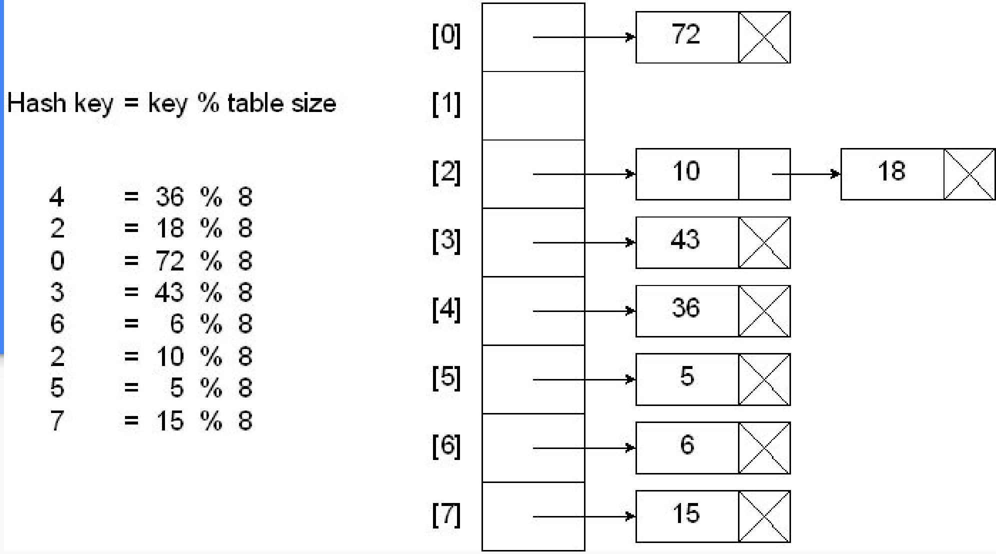


Quadratic Probing

- Use a quadratic function to probe
- probe(i) = (hashkey + i ^ 2) % TableSize
    - For i in 1..TableSize
- Will eventually work: provided TS is primary and load factor isn’t too high
- Rehash when load factor (λ) >= 0.5

Separate Chaining

- Each table bucket is a linked list
- Inserts always work
- Searches take O(1) for to calc hash key, plus O(list len) searches
- On average, this is O(1) + O(1.5)
- Rehash when load factor (λ) >= 1

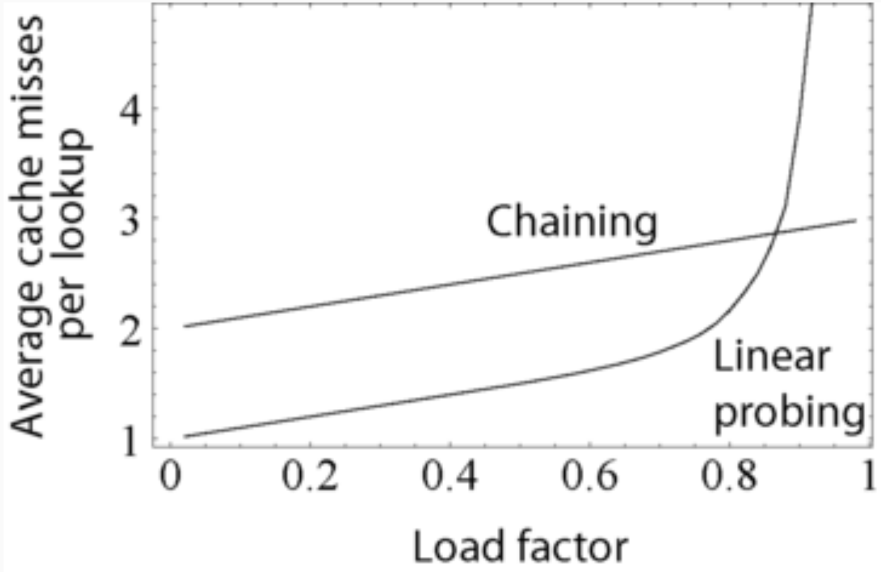


Hash key = key % table size

```
4    = 36 % 8
2    = 18 % 8
0    = 72 % 8
3    = 43 % 8
6    =  6 % 8
2    = 10 % 8
5    =  5 % 8
7    = 15 % 8
```

Load Factor

- The load on the hash table is based on how full it is
- Normally denoted by (λ)
- Calculated by N / TableSize



Why prime sized tables?

- Hashing and probing all computed with mod(TableSize)
- If it isn’t prime, then algorithms are not guaranteed to touch all buckets for any given key value

Rehashing

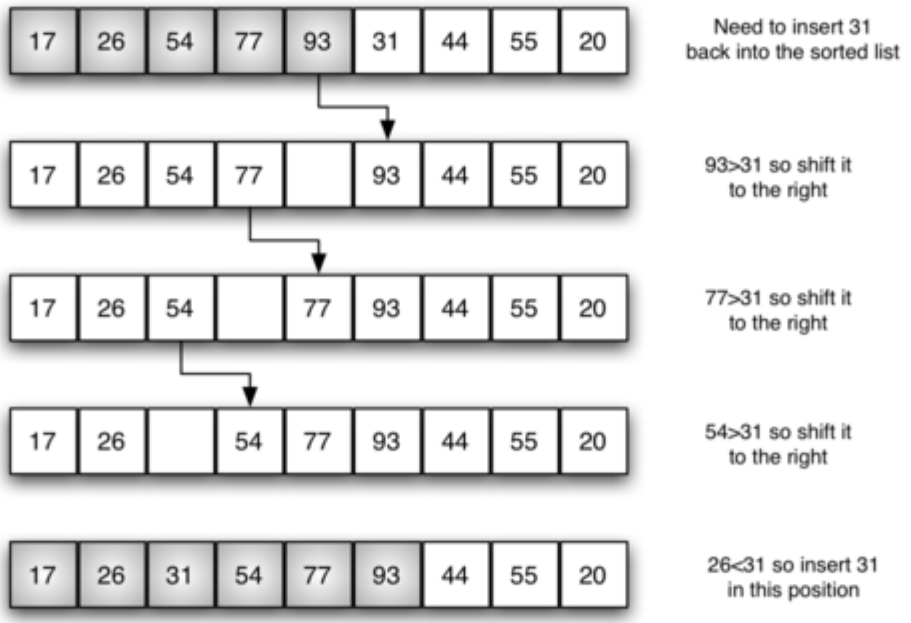
- If table is too full (or too empty?) the TableSize needs to be adjusted
- Normally, you double TableSize, then move up to next prime number
  - 11 -> 22 -> 23
  - 101 -> 202 -> 211
- Algorithm is simple, but doubles the table size memory footprint while rehashing is done

Part #3 - Sorting

- The ordering of elements through comparison or exploiting data properties
- Many algorithms available:
  - Bubblesort, Insertionsort, Mergesort, Heapsort, Quicksort
- Simple sorts that do local comparisons operate in O(N^2) time
  - Bubblesort, Insertionsort
- Sorts that do more work per step (long distance comparisons, group comparisons) can get down to O(N log N) time
  - Mergesort, Heapsort, Quicksort
- In special data cases you can do O(N) time sorting
  - Bucket (counting) sort, Radix sort

Insertionsort

- Move current element until it's in correct place (less than ll to right).
- Then move all elements right as needed
  - Creates lots of copying
  - Comparisons are local
- Works in N^2 time
- Stable sort



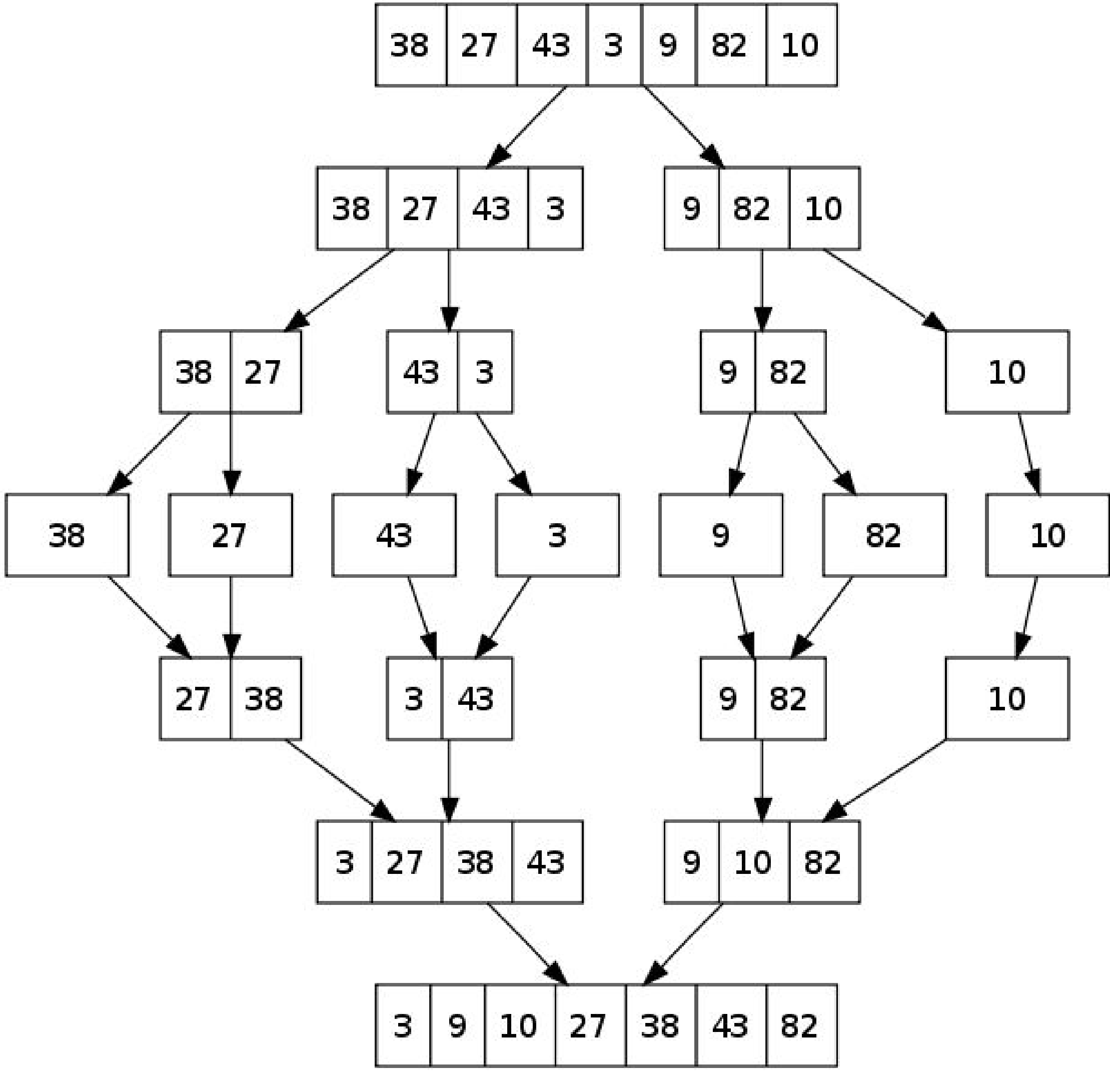
17 26 54 77 93 31 44 55 20
Need to insert 31 back into the sorted list
17 26 54 77 93 44 55 20
93>31 so shift it to the right
17 26 54 77 93 44 55 20
77>31 so shift it to the right
17 26 54 77 93 44 55 20
54>31 so shift it to the right
17 26 31 54 77 93 44 55 20
26<31 so insert 31 in this position

Heapsort

- Put everything in a heap, then pull them out with deleteMin in order
  - Buildheap is O(N), DeleteMin is O(log N) result is O(N log N) time
  - Takes two structures (one tree, one results vector)
- Not guaranteed to be stable

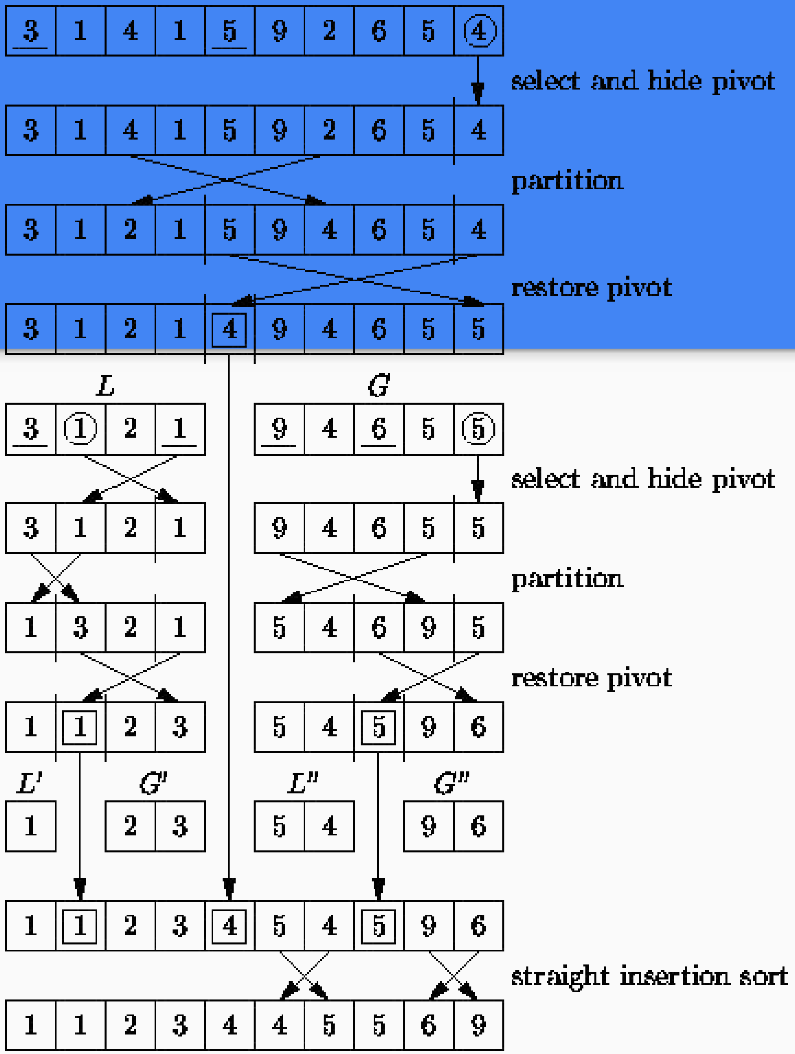
Mergesort: Divide

- Recursively divide list into smaller sub-lists until the sub-lists are sorted
- Merge back together. Merging: O(N+M) time Division: O(log N) time, Overall: O(N log N)
- low on comparisons, high on data moves
- Stable sort



Quicksort

- Pick a pivot, move all smaller to left, and all bigger to right
- Quicksort on left
- Quicksort on right
- Return {left, pivot, right} -> WIN
- Works in O(N log N) given good pivot values chosen
  - #1 approach: median of 3: [0, N/2, N]
  - Bad if always min/max value: N^2
- Stable sort



Bucket (Counting) sort

- Linear time sort for small integers
- Given: Arr = [A_1, A_2, … A_N] of positive integers smaller than M.
- Make int counts[M], initialized to all 0.
- Read input: for each value in Arr you increment counts with: counts[A_i]++
- Then print out all non-zero buckets counts[i] times
- Results are found in O(M+N) time
    - If M == O(N), then final result is O(N)
- Power comes through M-way comparisons in O(1) time
    - Similar to hashing algorithm

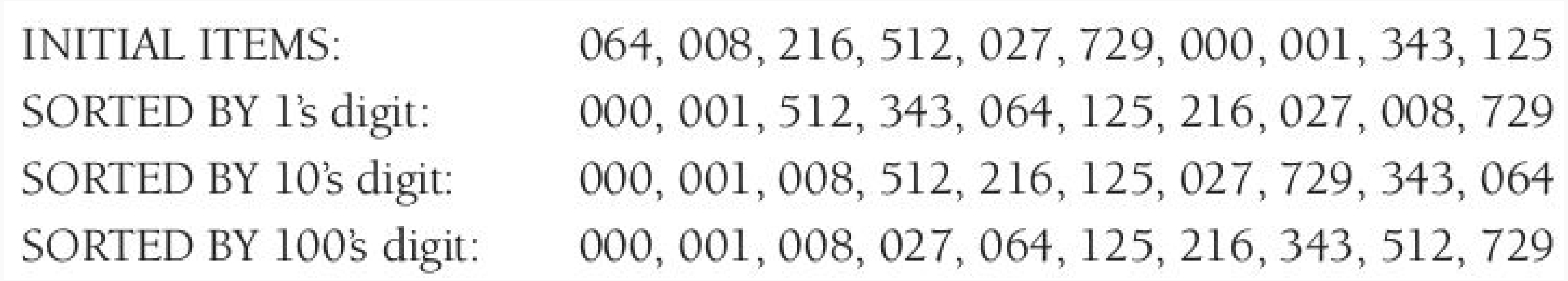
Different approach: Radix sort

- Used to sort punch cards, so also called card sort
- Can work on integers or strings
- Sorts by one indexed position at a time
  - Normally done right to left (won’t work otherwise!)
- Is a stable sort
- Effectively multi-pass bucket sort

Radix sort description

- Data must be N numbers in range(0..b^p-1)
  - N == set of numbers in array
  - b == number of buckets, which is determined by the size of the alphabet in the data
  - p == number of passes, which is the length of the strings/numbers
- Do passes over each digit or string position
  - Go from least significant to most significant
  - Put item into bucket as indexed by position (needs b buckets)
  - When done, read all items out in FIFO (to be a stable sort) order back to original array
  - Repeat p times, moving index from least to most significant position
- Result is done in O(p(N+b)) time!

Counting radix sort - sorting by numbers



INITIAL ITEMS:	064, 008, 216, 512, 027, 729, 000, 001, 343, 125
SORTED BY 1's digit:	000, 001, 512, 343, 064, 125, 216, 027, 008, 729
SORTED BY 10's digit:	000, 001, 008, 512, 216, 125, 027, 729, 343, 064
SORTED BY 100's digit:	000, 001, 008, 027, 064, 125, 216, 343, 512, 729

Each pass orders the items in a more significant way.

The results kind of pop into existence at the end.