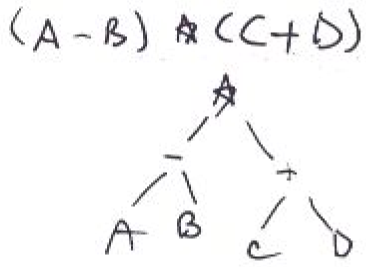
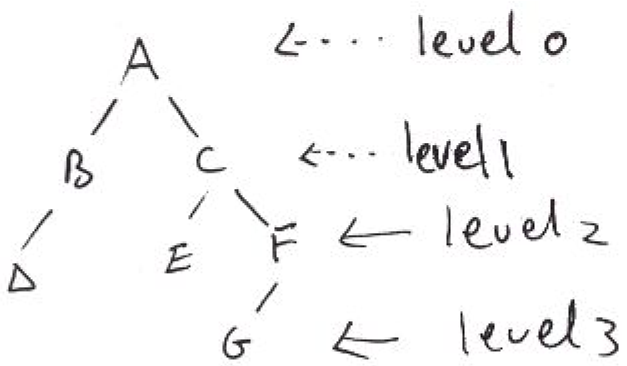
Trees

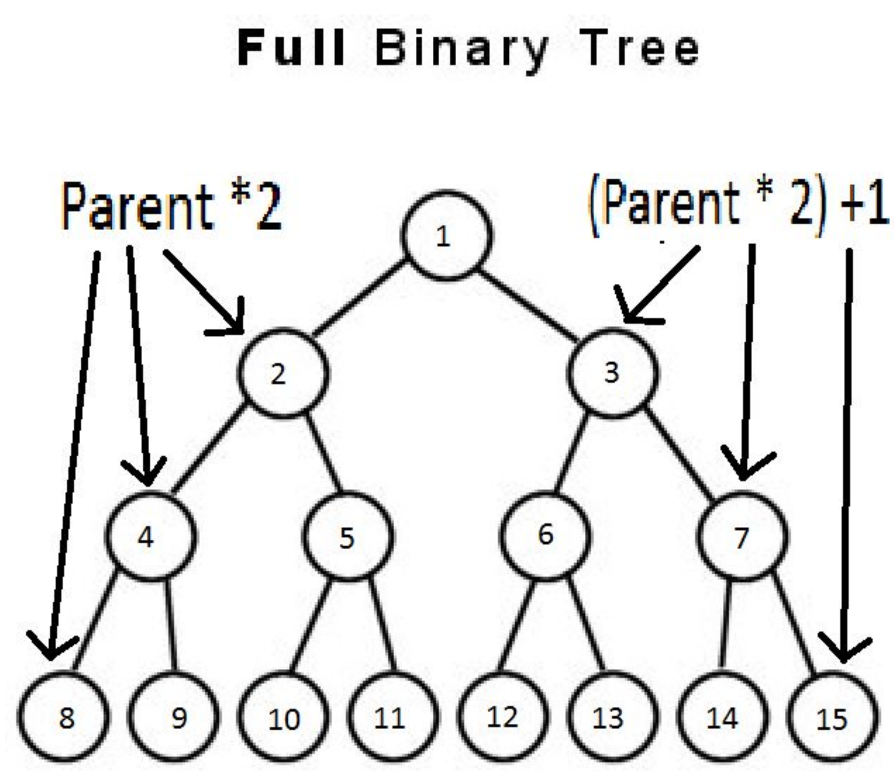
- File organization
- Expression trees
- Search trees
  - find data faster
  - index into large files or DBs
- Game trees
  - Keep possible next moves
    - (Postponed obligations)
- Encoding/decoding messages
  - huffman codes
- Priority Qs
  - items have priorities
  - tree data structure allows quickest access to highest priority items





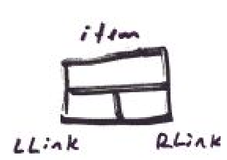
height of tree 3. (longest path)

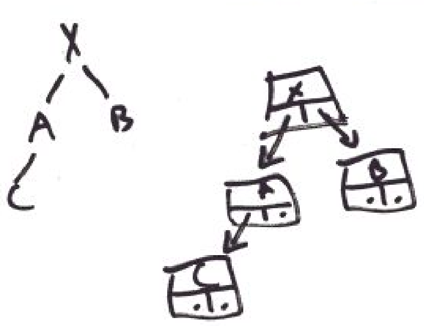
- Binary tree => Empty or has 1 node with 2 children, each child is a binary tree itself
- Complete tree have leaves to the left as possible



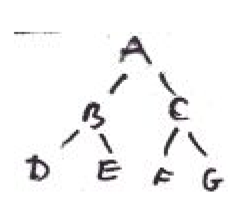
Representations

- Sequential => Uses array
  - root = A[1]
  - A[i]'s left child = A[2*i]
  - A[i]'s right child = A[2*i + 1]
  - A[i]'s parent = A[i/2]
    - Problems when tree long and thin and right-heavy
- Linked
  - node linked to left+right node





- Traversals
  - Breadth First
    - Level Order => ABCDEFG
      - Level by level
  - Depth First
    - Pre Order => ABDECFG
      - root, left, right
    - In order => DBEAFCG
      - left, root, right
    - Post order => DEBFGCA
      - left, right, root



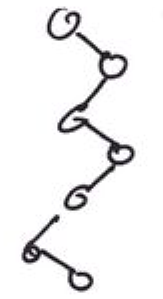
Binary Search Trees

- Left Node Value < Parent Node Value < Right Node Value
- Search: Just do it
  - If less than current node move left, if larger than current node move right
- Insert: Same as search, until you find an external node
- Delete:
  - If leaf/external node then just delete
  - If 1 child => Delete node, promote child
  - If 2 children
    - Copy largest node from the left subtree
    - Or copy smallest node from right subtree
      - Then delete the copied node from old spot

Optimally Balanced Trees?

- Complete binary search tree so that all leafs are on same level
    - Search time and insert time is O(cosn) at worst
        - Height proportional to logn
            - n = 2^(n+1) - 1
                - log2 (n+1) - 1 = h
                - log(n+1) < logn + 1
                - so h < logn
                - any complete BST
                    - h = floor(logn)
- Degenerate BST
    - Search time of insert time is O(n) at worst
    - Height proportional to n
    - keep trees optimally balanced for quickest search
    - PROBLEM => algorithm to re-balance them after insert in O(n)

degenerate bst.



B-Tree

- If tree is stored on disk, each pointer follow means you have to read from the disk
    - TIME CONSUMING
    - Solution
        - Allow more than one record (Key and data) in each node
        - Each read gets n records
        - Do 1 access
            - Put node into memory (RAM) and search for the desired key in **MEMORY (FAST BINARY SEARCH)**
- If tree is balanced, fastest insert/search/delete => **log(n)**

B-tree of order m

- a search (ordered) tree such that
    - root (can be by itself/ a leaf) has **j** keys
        - 1 <= j <= m-1
    - Other nodes have
        - At <u>least</u> **CEIL(M/2) - 1** keys
        - At <u>most</u> **M-1** keys
    - All internal nodes have **ONE MORE CHILD** than keys
    - Leafs
        - No kids
        - All on bottom-most level
            - Bottom-most level is full (none missing)

Order 3 B-tree

- root => 1 or 2 keys
    - M = 3 => ORDER 3 DUHH
        - ROOT HAS J KEYS
            - 1 <= j <= m-1
                - THIS EXPLAINS WHY ORDER 3 B-TREE MUST HAVE 1 OR 2 KEYS
- Other nodes
    - At <u>least</u> **CEIL(3/2) -1** = 2 - 1 = **1 key**
    - At <u>most</u> 3-1 = **2 keys**
        - **so 2-3 kids**
            - **THIS IS REASON WHY ORDER 3 B-TREE IS OFTEN CALLED 2-3 TREE**

Order 15 B-tree => another example

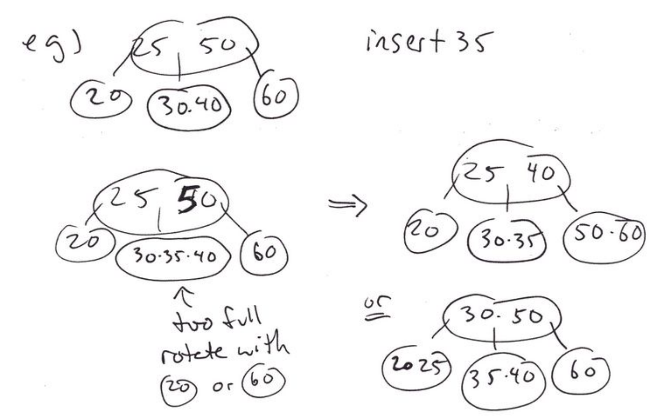
- m = 15
    - 1 <= j <= m
        - j can be 1-14 (2-15 kids)
- Others
    - At <u>least</u> CEIL(15/2) - 1 to 15 - 1 keys
        - 7-14 keys, 8-15 kids

Number of levels in B-tree order M with all nodes full?

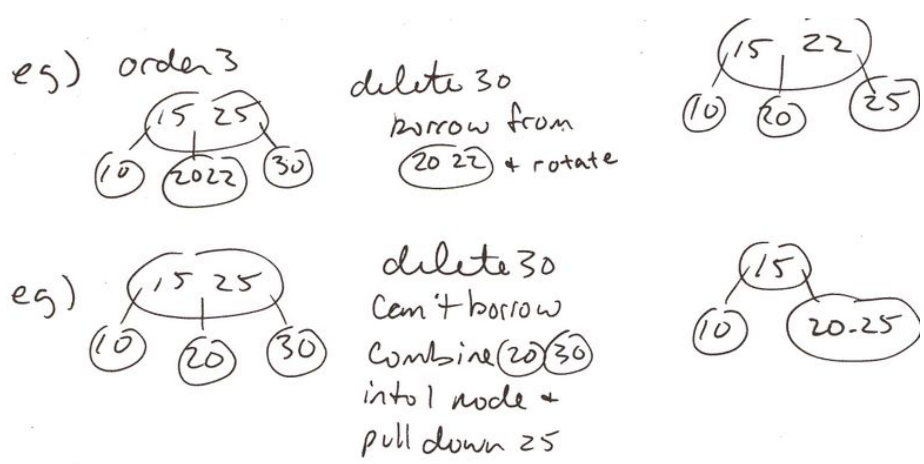
- T has n keys, p nodes, order m => **p = n/(m-1) ⇐ NUMBER OF NODES**

- Insertion
  - always insert **INTO AN EXISTING** leaf
  - if node too full
    1. move a key
    2. or change tree structure
  - To insert "k"
    1. search for k in tree => If k exists, there's error since don't need to insert again
    2. insert (in whatever node x)
    3. if x is too full
       - split in half
       - take out middle key and move it up to parent
       - Call parent node x now
       - repeat part iii until finished
  - EXAMPLE: http://www.scs.ryerson.ca/~dwoit/courses/cps305/coursedirPublic305/NOTES/trees/T18.gif

  - **ROTATION WITH SIBLING** ====================================>
    - only works if sibling not too full
    - only immediate left or right sibling



- Deletion
  - From leaf => EXTERNAL NODE
    - You can delete key **EASILY** if there are enough keys => Just remove
    - grab keys from for left/right sibling if you can't do above ^^
      - if can't, collapse nodes/push key down from parent node
    - may have have to collapse more nodes working up towards root



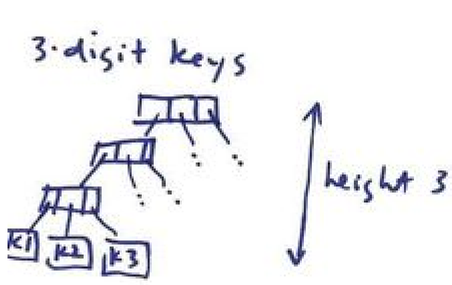
  - Non-leaf => NOT EXTERNAL NODE
    - replace key by inorder successor (predecessor)
      - must be in a leaf
    - delete the key from leaf as above

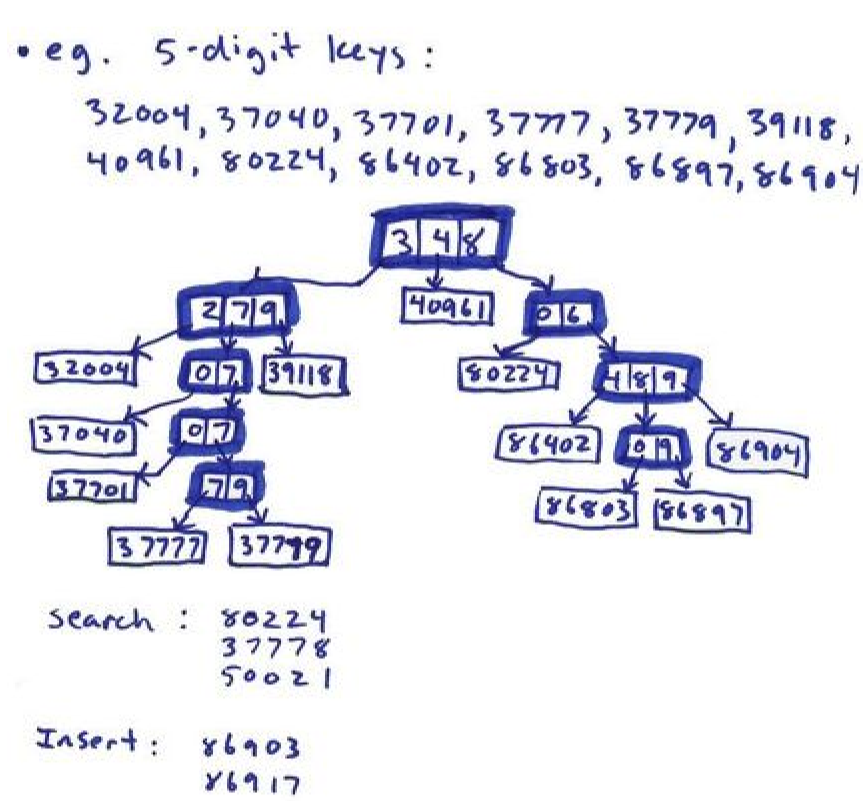
HUFFMAN CODING

- Data compression with trees to encode/decode messages
  - use 0/1s to encode data
  - minimize lengths of encodings
  - used in parts of: MP3, JPEG algorithms
- Steps
  - get "frequency" for each character
    - most used to least character
  - Sort the characters from most to least ======>
  - Create tree
    - http://www.scs.ryerson.ca/~dwoit/courses/cps305/coursedirPublic305/NOTES/trees/T22.gif
    - http://www.scs.ryerson.ca/~dwoit/courses/cps305/coursedirPublic305/NOTES/trees/T23.gif

Tries

- Data structure used for storage/retrieval of data
- Organization in `tree`based on individual characters in key
- Height
  - Worst case: O(m)
    - m = # chars in key
  - average case: O(m)





- Examples of Tries
  - Key serial ids (9 digits)
  - tries are 2x faster than 2-3 tree

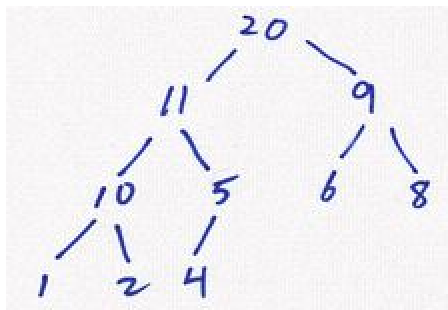
Implementation of tries

- each node has m fields (m = chrs in key)
- fast (ram)
- may waste space
- each internal node is a linked list

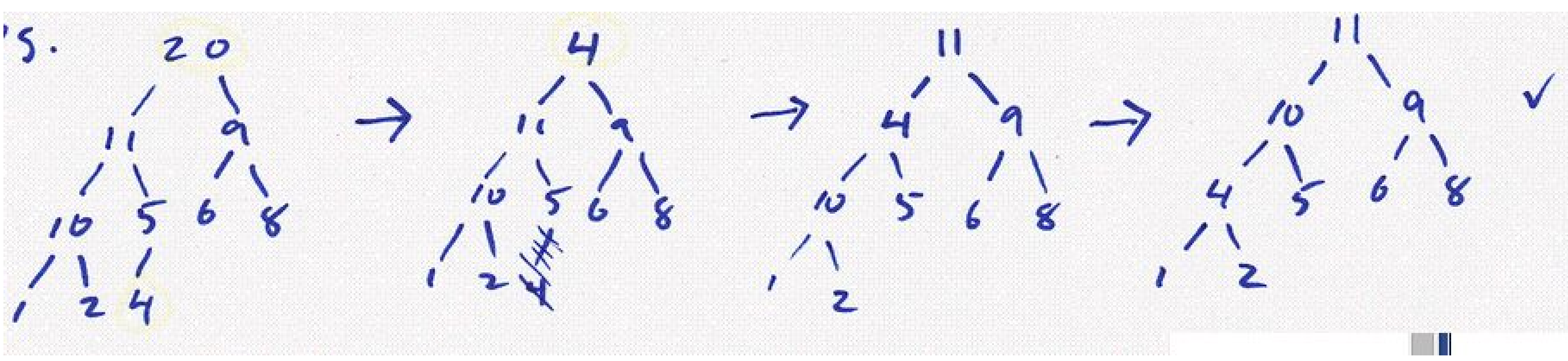
Applications of tries

- prefix completion (autocomplete)
- dictionaries
- replacing BSTs, hash tables in some cases
    - because worst case trie lookup O(m)
        - vs O(n) BST
        - vs O(n) hash
    - hash tables not enumerable

Heaps ==========================>



- Complete binary tree
- each node has <= it's parent key
    - sorting property
- Applications
    - Priority Queues
    - Sorting algorithms => Heap sort
    - Graphing algorithms
    - Selection algorithms
        - quickly find max/min/median/ kth largest item
- How to remove max (highest priority) item
    - take largest value from root of tree and delete it
    - take smallest value, right-most leaf on bottom level and put at root
    - restore sort property by bubbling (exchanging) new root ke into correct position



Removal Alg:

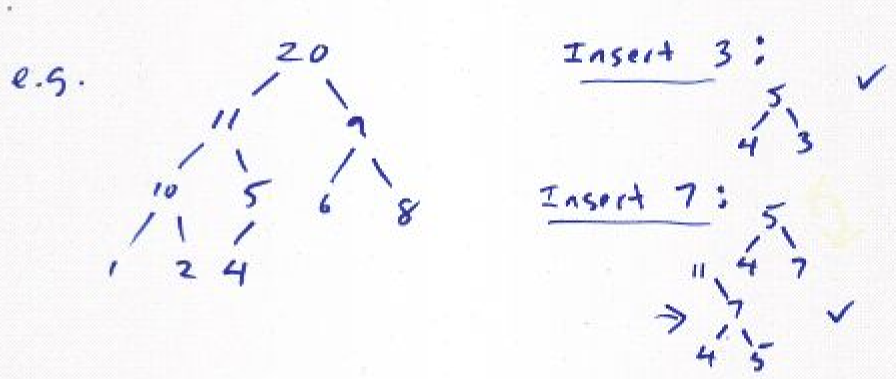
```
• Remove (Heap H)
      if empty(H) return NULL
      Removed = root(H)
      copy(root, Last_in_level_order(H))
      deleteNode(Last_in_level_order(H))
      if !empty(H) Reheapify(H, root)
      return (Removed)

• Reheapify (H, N)   <- node of H at which to start reheapify process.
      while !leaf(N)
          M = largest_child(N)
          if key_N ≥ key_M return
          exchange (M, N)
          //N is now child of M
```

O(log n) worst case
avg case

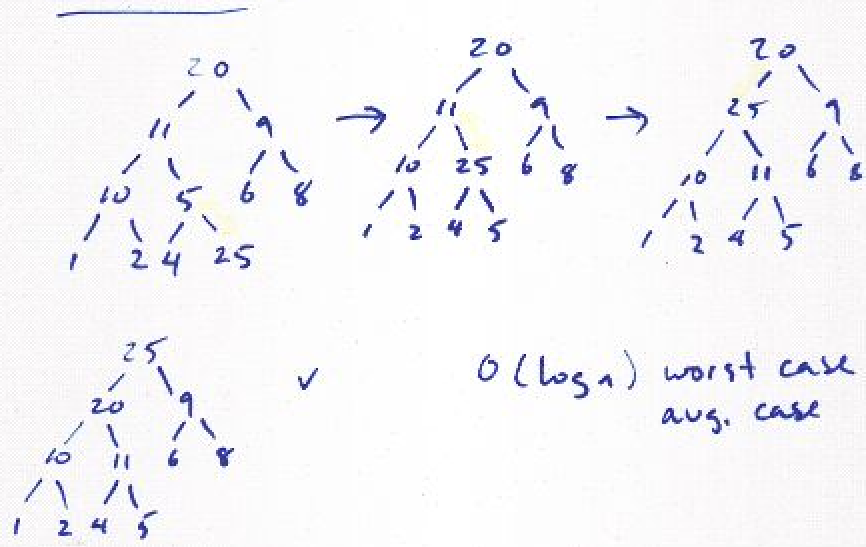
Inserting into Heap. T-3

- insert in next leaf (for complete BT)
- bubble-up new item - exchange with parent until sort property met. (parent ≥ child)

e.g.



Insert 25:



O(log n) worst case
avg. case

Build Heap from N unsorted Items

- Put items in complete BT structure
- Establish sort property
    - for each node, N, in reverse-level-order reheapify (H,N)
- O(n) to build heap

Heap Implementation

- linked nodes
- Array
  - efficient use of space (no "holes" in array)
  - sequential binary tree
- fast (RAM)
- space = size of N

HASH TABLES

- Examples
  - Associated arrays
  - Database indexing
  - Caches
  - Sets
  - Object representation
  - Unique data representation
- O(1) instead of O(logn) (BST/B-tree/binary search) or O(m) (trie)
- goes directly from key to record in table
  - assume to be randomly accessed
- If keys are integers they can be mapped to an array table
- What problems are there that can be solved via hash tables?
  - wasted space for SID (9 digits) need table (array) size 1,000,000,000
- Solution for above is to convert key into integer in desired range
- Hash function => h(k) converts the key, k, into an index (slot # for table)
- Collision: When h(k1) = h(k2)
- Collision resolution: produce what do after collision to find empty slow for key

Hash Functions

- Truncation => h(2647983) = 983
  - problem => may cut off unique part of the key => collision

2) Middle Square

e.g. K = 4263. $4263^2 = 18173169$

reduces patterns (collisions) somewhat

- Folding
  - partition k into sections and recombine
  - Example
  - k = 782146
    - 782+146 = 928 or 982+641 = 1423 => truncate if have too
  - Table size must be power of 10
- Division
  - h(k) = mod (K, M) => M = table size
  - best results when M is a prime (less collision and covers table well)

key	M=13 (prime)	M=12
558	12	6
723	8	3
692	3	8
876	5	0
574	2	10
945	9	9
716	1	8 ←
201	6	9 ←
946	10	10 ←

NO collisions

will tend to get multiples of factors of 12.
e.g. 4: 2x4 = 8
3: 3x3 = 9 etc

Collision Resolution

- ### Open addressing
  - Linear probe
    - Insert: If k gets hashed into full slot, s, put it in next empty slot upward =>s-1, s-2…
      - If hit bottom of table (slot 0), wrap around top or keep going
    - Retrieval: do h(k) = s
      - if k not in slow s, look in s-1,s-2… (with wrapping) until
        - k found
        - hit empty slow (not found)
    - Table full when m-1 slots occupied
    - Problem => Primary clustering
      - a few keys randomly near each other tend to collect into clusters
      - clusters combine into bigger clusters
  - Double hashing
    - no primary clustering (only random ones)
    - Instead going down by one, every collision go down by some other amount
      - The probe decrement p(k)

        - When h(k1) = h(k2), k1 follows different probe sequence than k2

key	h(key)	D.H. Probe Seq	L.P. Probe Sequence
914	5	97, 88, 79...	4, 3, 2, ...
712	5	99, 92, 85 ...	4, 3, 2, ...

        - Retrieving
            - Search for 7/2
            - try 5, 99, 92, 85…. until
                - empty slot => Not found
                - found
    - Deleting record in open addressing
        - delete key/record but set flag in spot to indicate “Keep searching”

Suppose h(207) = 5 + h(914) = 5 ^{x} → 97
Delete 207.
Retrieve 914. ⇒ Slot 5 empty ∴ NOT FOUND!

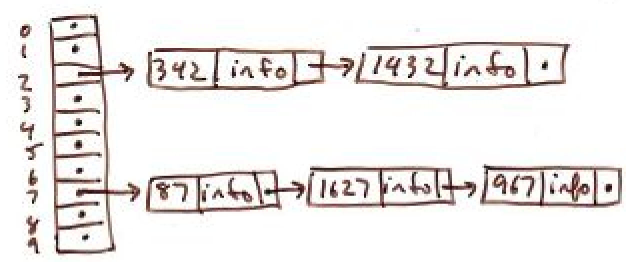
    - Problems with open addressing
        - fixed m
            - if more records than m
                - allocate larger table
                - rehash all records
        - keep searching can flag around most of the table, forcing us to search through most of the table in order to conclude not found

- **Separate chaining**
    - table entry contains pointer to linked list
    - h(k) = slot => add record to linked list for slot
    - PROS
        - delete = no effect on later retrievals
        - table size < open address and less need to reallocate to larger one
    - CONS
        - More space (links)
        - List too long ⇒ efficiency drops
    - General rule of thumb ⇒ If records > pointers = win
    - Improvements
        - insert at front
        - keep list ordered/tree (faster search)

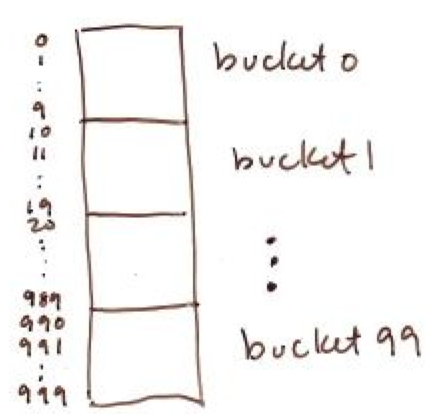
e.g.) M = 10, truncation to last digit
insert 87, 342, 1627, 1432, 967

initially:

finally:



- **Hashing with buckets**
    - Divide table into equal sized sub-tables (buckets)
    - initially hash into bucket using h(k)
    - within bucket, keep the items ordered
    - to retrieve: h(k) and then binary search in bucket
    - If bucket full, use different collision resolution policty
        - linear probe
        - double hash
        - chaining
            - keep pointer with each bucket to “overflow area” where rest of items are
    - Not as efficient as other collision res policies when hash table in memory
    - good performance on disk
        - Each probe required a disk read (EXPENSIVE)
        - with bucket, 1 read => loaded into memory then do a binary search



Analysis of Hashing

Analysis of Hashing

N = # entries in Table
M = tablesize

Load Factor $\alpha = N/M$

Linear Probe:

successful search (retrieval) $\frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$

unsuccessful $\frac{1}{2}\left(1 + \left(\frac{1}{1-\alpha}\right)^2\right)$

Double Hashing:

successful $\frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right)$

unsuccessful $\frac{1}{1-\alpha}$

Chaining:

successful $1 + \frac{1}{2}\alpha$

unsuccessful α

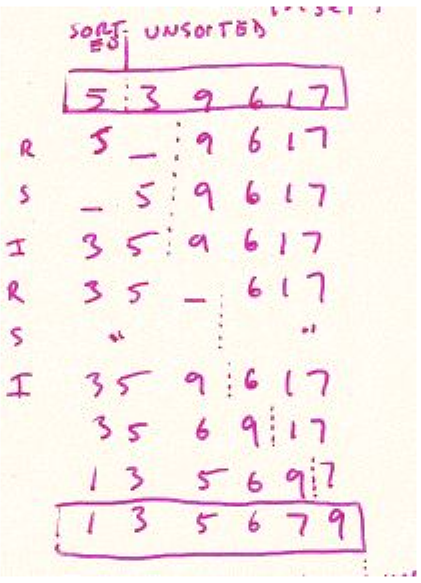
- Performance based on load factor(fullness of table), not # keys in table
- O(1)
  - load f <= 50%
- Keys not integers => Convert them
  - concatenation
  - converting from base X
  - then apply h(k), p(k)

- Collisions likely? YES GO TO H-14
- P(N) = collision probability
- Q(N) = no collision probability

Sorting Applications

- Examples
  - Commercial apps
  - operating system research
  - simulations
  - graph algorithms
  - huffman compression
  - order statistics
  - sorting animations
- Sorting Efficiency
  - # of comparisons ⇒ best average case = O(nlogn)
  - data moves ⇒ best average case = O(n)
- Insertion sort types
  - start with empty containers
  - insert 1 by 1
  - tree sort, insertion sort
- Address-type
  - items not compared to each other
  - categorized based on specific properties
  - radix sort, prox map sort
- Priority q
  - insert items into PQ
  - remove 1 by 1 ⇒ get sorted order
  - heapsort, selection sort
- Div and conquer
  - divide unsorted part into 2 parts
  - sort each part and recombine
  - quicksort mergesort
- Diminishing increment type: Shellsort
- Transposition-type: bubble sort

INSERTION SORT



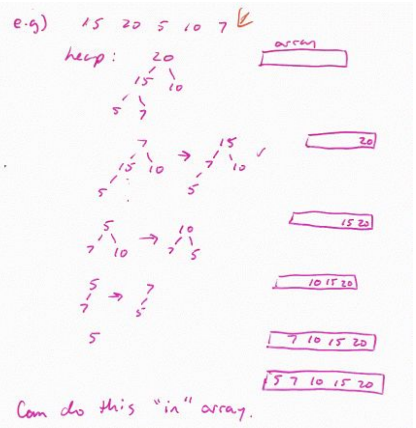
- division in 2 parts ⇒ LHS sorted, not RHS
- each step:
  - get next value X from right side
  - find the right spot in the left hand side it should go
  - remove it from right side
  - insert X
- Comparisons
  - Worst case ⇒ O(n^2)
  - Average => O(n^2)
  - Best ⇒ O(n)
- Data moves
  - Worst ⇒ O(n^2)
  - Average ⇒ O(n^2)
  - Best ⇒ O(n)

QUICKSORT

- Comparison
  - best case is when pivot is in middle, list is equally half ⇒O(nlogn)
  - Worst case is when pivot is at the end ⇒ O(n^2)
  - Average case ⇒ O(nlogn)

HEAP SORT ==>

- Push to left
- binary tree
- parent value >= kid
- all levels full except last
- comparisons and data moves similar
  - Worst case and average ⇒ O(nlogn)

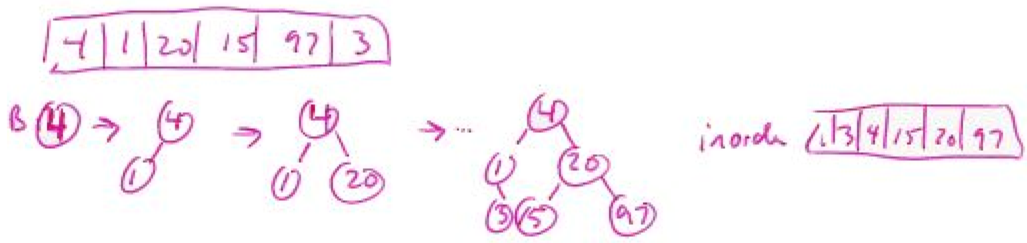


RADIX SORT ===================================>

- radix is <u>base</u>
- no comparisons only moves
- O(n)
- n passes ⇒ number of max digits

- r ⇒ # of keys

Tree sort
- unsorted array into BST
- placed using inorder algorithm
- Comparisons
  - best/avg ⇒ O(nlogn)
  - worst ⇒ O(n^2)
- data moves ⇒ O(n)



Merge sort
- Steps
  - list has only item return
  - divide list in half
  - mergesort each half
  - merge the two halves back into one
- All cases ⇒ O(nlogn)
- generally proved that for any comparison-based sort, fastest average comparison is O(nlogn

STABILITY
- if preserves relative order of equal keys
- STABLE: insertion, merge, radix, BSTS
- UNSTABLE: quicksort, heapsort

BEST SORT
- http://www.scs.ryerson.ca/dwoit/courses/cps305/coursedirPublic305/NOTES/sorting/S12.gif