B-Trees

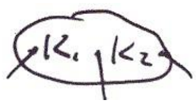
If tree stored on disk, each pointer flow = read from disk. This is expensive (time)

Solution?
- Allow more than 1 record (key+data)
- 1 read gets n records
- Do 1 access, read node into memory, then search for desired key in memory (fast binary search)
- If keep tree balanced, get fastest insert/search/delete times – O(logn)

Common Uses:
- Indexing into file on disk, filesystems: e.g. some linux, apple, win 7, 8
- Large tree stored on disk, databases: e.g. oracle

B-Tree of order m
- A search (ordered) tree such that
  - Root (maybe a leaf) has j keys 1 <= j <= m-1
  - All other nodes have:
    - At least $\lceil m/2 \rceil - 1$ keys
    - At most $m - 1$
  - All internal nodes have 1 more children than keys

    eg.

    2 keys, 3 kids
  - Leafs:
    - Have no kids.
    - All on bottom-most level
    - Bottom-most level is full (none missing)

How many levels in B-tree order m with each node as full as possible? (Tree is T)

T has n keys, p nodes => $p = \frac{n}{m-1}$ Why? Each node has m-1 keys

Level	# nodes
0	1
1	m
2	m^2
…	…
k	m^k

Total # node $1 + m + m^2 + m^3 + \cdots + m^k = \frac{m^{k+1}}{m-1}$

So $n = m^{k+1} - 1 \Rightarrow n + 1 = m^{k+1} \Rightarrow \log_m(n+1) = k + 1 \Rightarrow k = \log_m(n+1) - 1$

So # levels is $\log_m(n+1)$

e.g. m = 512, n = 262, 143 => $k = \log_{512}(262144) - 1 = 2 - 1 = 1$ # levels is 2
therefore B-tree order 512 can store ¼ million records in 2 levels (0, 1). At most 2 disk reads to find any node (record)

A balanced BST? (full)
at most $k = \log_2(262144) - 1 = 18 - 1 = 17$disk read

Order 3 B-Tree

- Root has 1 or 2 keys
- Other nodes:
    - At least $\lceil 3/2 \rceil - 1 = 2 - 1 = 1$ key
    - At most 3 – 1 = 2 keys, so 2 -3 kids
- Often called a 2-3 tree.

Order 15 B-Tree

- Root has 1 - 14 keys (2 – 15 kids)
- Other nodes:
    - $\lceil 15/2 \rceil - 1 = to\ 15 - 1$ keys => 7 – 14 keys, 8 – 15 kids

Order 256 B-Tree

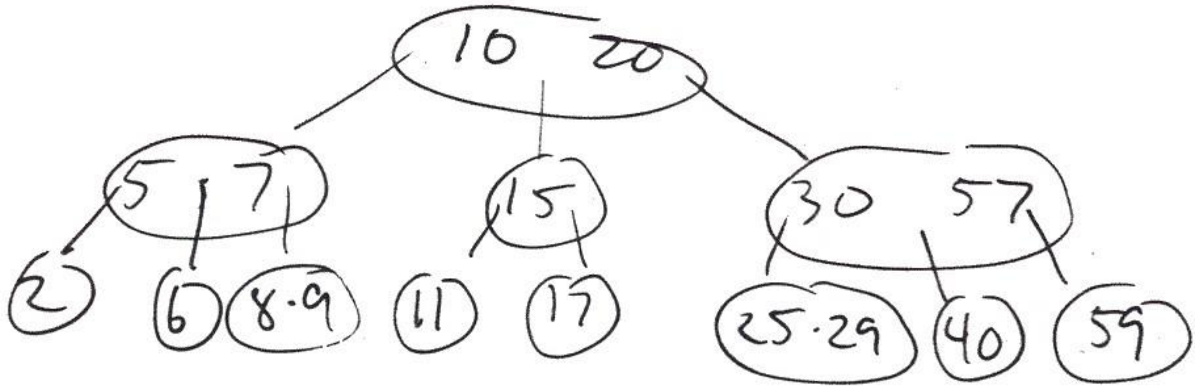
- Root has 1 - 255 keys
- Other nodes:
    - $\lceil 256/2 \rceil - 1 = to\ 256 - 1$ keys => 127 – 255 keys, 128 – 256 kids

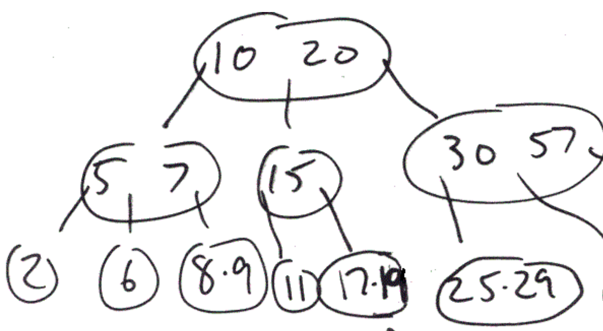
Insertion

- always insert into an existing leaf
- if node is too full
    - move a key or
    - change tree structure
- to insert "k"

a) search for k in tree. If found, error, else
b) insert (in node X)
c) if X too full, split it in half, take out "middle" key and move up to parent (call parent node X now)
d) repeat (c) until finished

e.g. insert 19 into this 2-3 tree



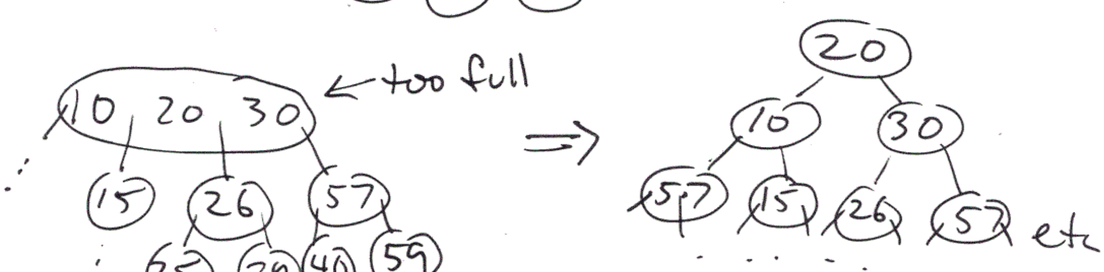


ok. node not too full

now insert 26

- goes in 25·29 ⇒ 25·26·29 too full

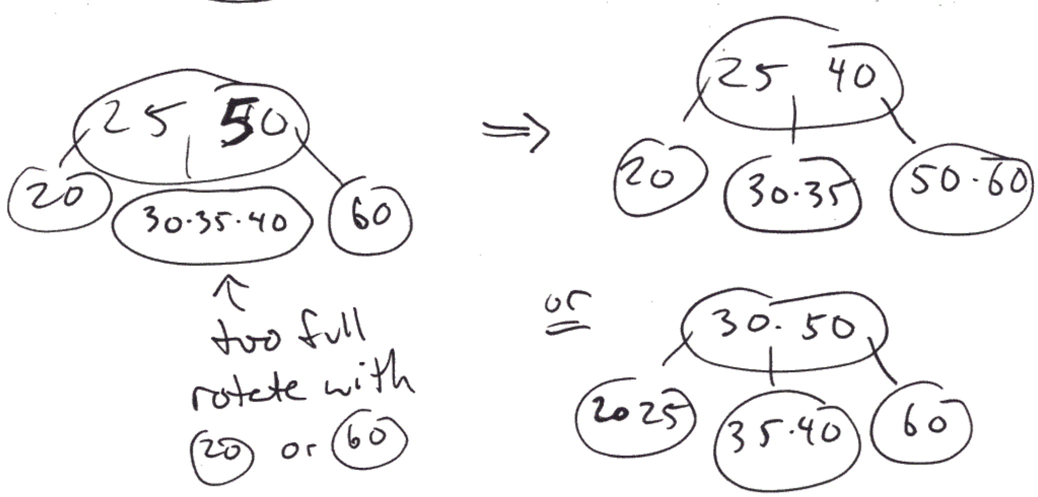
- split + move 26 to parent



- Remember, if move key up and node is not full, STOP

Another way of inserting

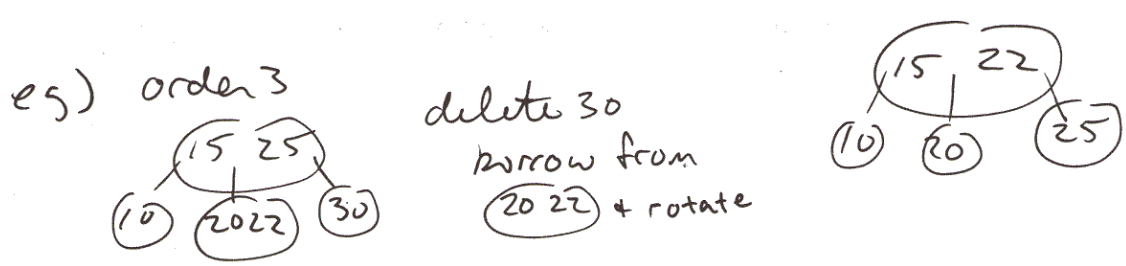
- "Rotation with sibling"
- only works if sibling not too full
- only consider immediate left or right sibling

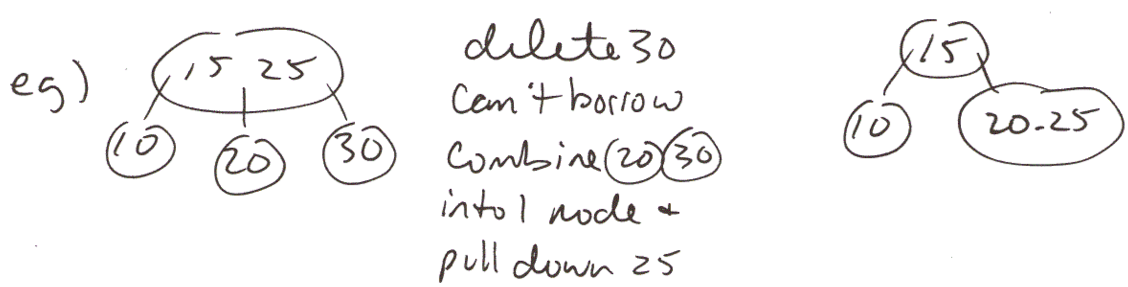


Deletion

From leaf:

- delete – if node full enough, done, otherwise
- "borrow" from L/R sibling. If can't – collapse node
- May have to collapse more nodes working up to root

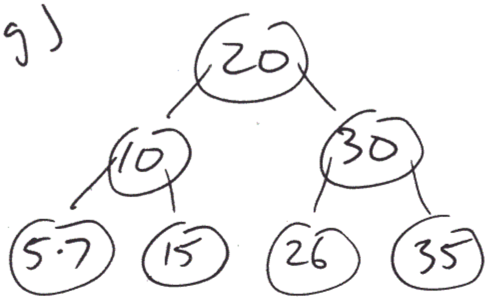




Non-leaf:

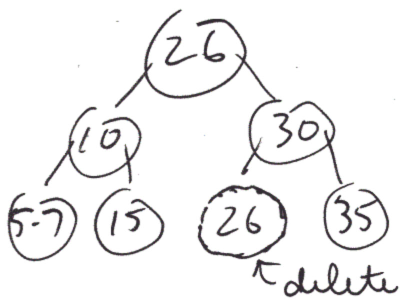
- replace key by in-order successor, predecessor (which must be in a leaf)
- delete the key from leaf as above

eg)

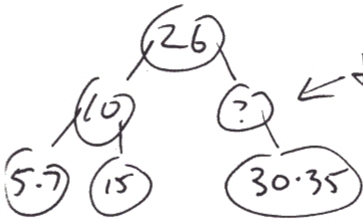


delete 20.

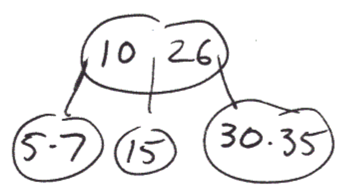
replace with 26 or 15



Can't borrow so collapse + pull 30 down



too small, so combine with 10 + pull down 26



done. note tree reduced by 1 level.

If used 15 instead of 26, tree NOT reduced by 1 level. why?

Huffman Coding

Data compression method using trees to encode/decode messages

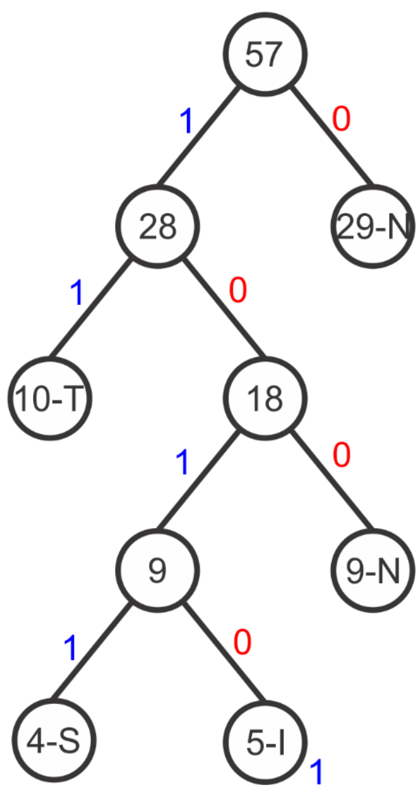
- use 0s and 1s to encode data
- minimize lengths of encodings
- used in parts of: MP3, JPG algorithms
  1) get “frequency” for each char
  2) sort
  3) create tree
     a. 2 smallest
     b. in chart, replace the 2 from a by sum and resort
     c. repeat from a until done - merging when possible
  4) replace frequencies with chars
  5) get char encoding by path through tree

Construct a Huffman tree for the alphabet. When pairing 2 items, the one with the smaller frequency goes on the left, and its arc has label 1. If two items with the SAME frequency are being paired, then this is how you must determine which one goes on the left:

- if both items are characters in the alphabet above, then the one that is smaller, in alphabetical order, as given above, goes on the left.
- if one is a character of the alphabet, and the other is a created node, then the created node goes on the left
- if both are created nodes, then the one was the one created first goes on the left.

Chars	Frequency	Huff Code
E	29	0
T	10	11
N	9	100
I	5	1010
S	4	1011

Result



Efficiency?

Chars	Frequency	Huff Code	# bits in Huff Code	# bits in ASCII Code	3-bit Code
E	29	0	29x1 = 29	29x8 = 232	29x3 = 87
T	10	11	10x2 = 20	10x8 = 80	10x3 = 30
N	9	100	9x3 = 27	9x8 = 72	9x3 = 27
I	5	1010	5x4 = 20	5x8 = 40	5x3 = 15
S	4	1011	4x4 = 16	4x8 = 32	4x3 = 12
			112	456	171

Huffman Code vs. ASCII

112:456 => Huffman Code is 24.6% of ASCII

Huffman Code vs. 3-Bit Code

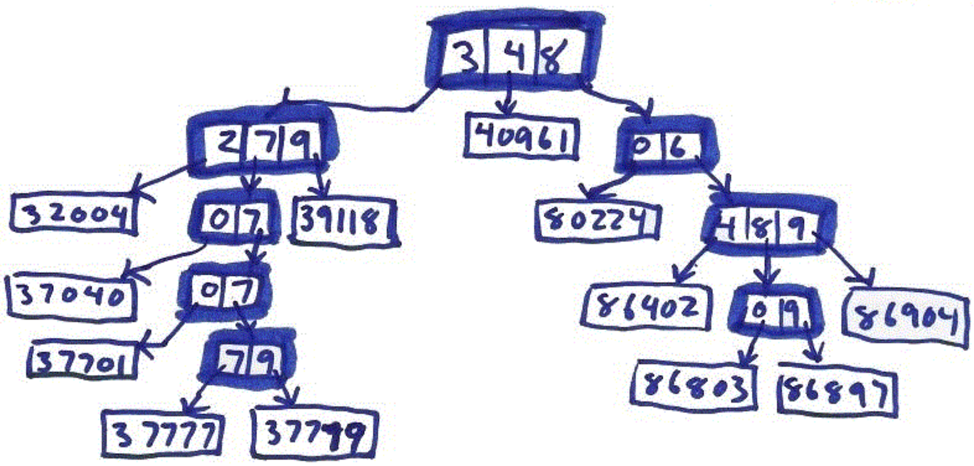
112: 171=> Huffman Code is 65.5% of ASCII

Tries

- a tree data structure for storage/retrieved of data
- organization in “tree” based on individual characters in key
- vs. say BST, AVL tree which use whole-key, comps (<, >, =) to organize records

• eg. 5-digit keys:

32004, 37040, 37701, 37777, 37779, 39118,
40961, 80224, 86402, 86803, 86897, 86904

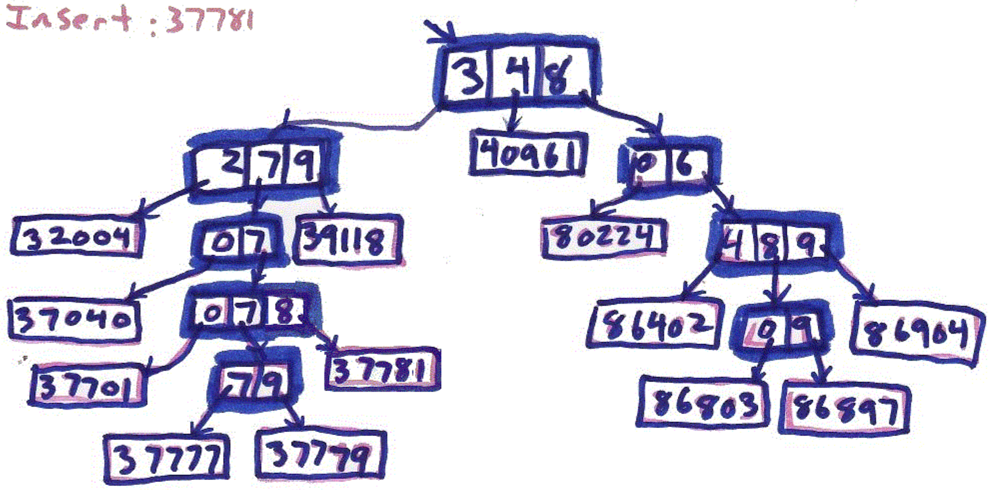


Search: 80224
37778
50021

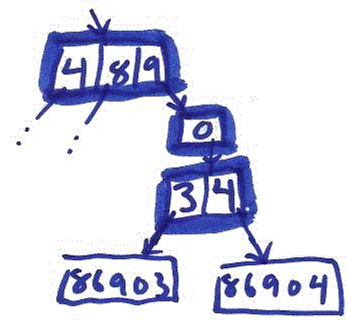
Insert: 37781
86903
86917

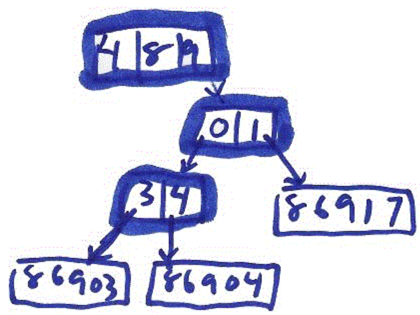
T-25

Insert : 37781



Insert : 86903
86917



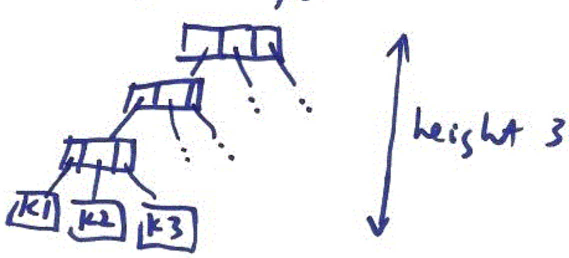


Trie height?

Worse case is O(m)

- m = maximum number of characters in key
- e.g. 3-digit keys

3-digit keys



Average case is O(m)

e.g. keys are Student numbers (9 digits). Trie contains approximately <= 10^9 keys, therefore, max height is 9 or ~ 9, worst case

Trie vs. AVL

- An AVL tree with N approximately 10^9 keys, the height is $1.44 * \log_2(10^9 + 2) \approx 44$
- Trie is ~ 4x faster

Trie vs. Balanced BST

- A balanced BST height is approximately $\log_2(10^9) \approx 30$
- Tri is ~ 3x faster

Trie vs. 2-3 tree

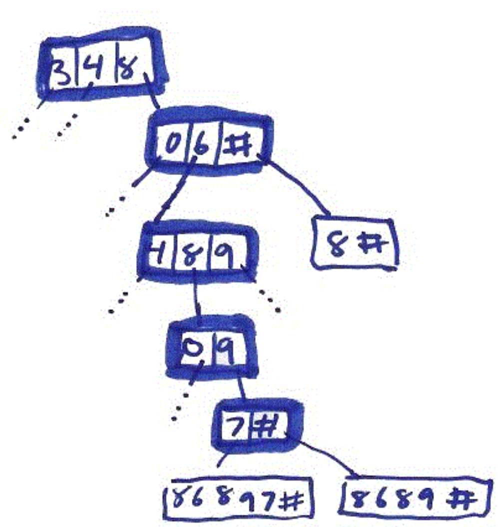
- A 2-3 tree height is approximately $\log_3(10^9 + 1) - 1 \approx 18$
- Tri is ~ 2x faster

What to do if you have keys with varying length?

Typically:

- append special char (say #) to each key
- no key is prefix of another

e.g. previous tree – insert 8#, 8689#



Trie Implementation

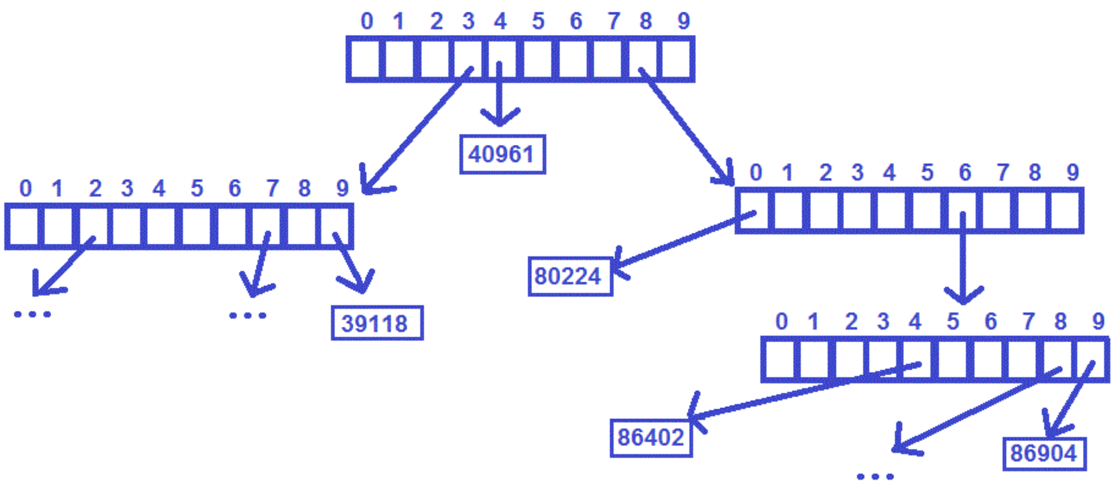
- each internal node has m fields, where m is how many characters in key

e.g. case sensitive alpha keys:

a	b	c	…	z

digit keys:

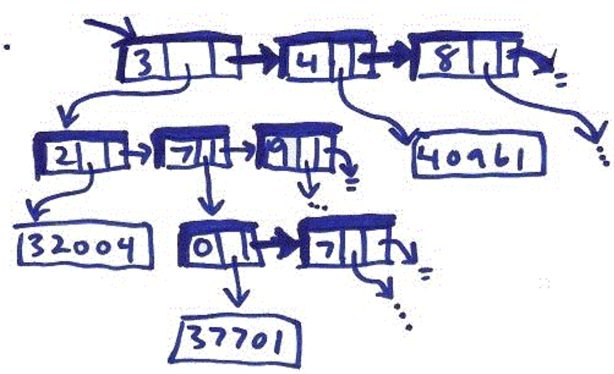
0	1	2	…	9



- Fast (random access)
- may waste space

- each internal node a L.List

e.g.



Trie uses:

Note: many searches only follow first fe links to find record, therefore, if you have only prefix of key … search and get all possible completions

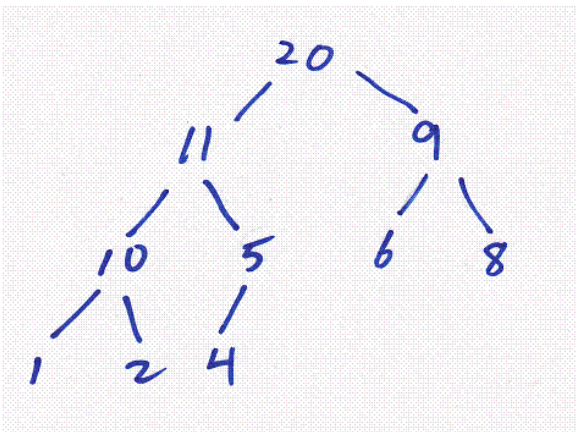
- Prefix completion (autocomplete)
  - unix/win command prompt completion
  - browser url address bar completion
  - cell phone (contact list item completion, text prediction when typing)
  - DB queries – plice get first few chars in license plate, list of matches and narrow down based on other chriteria
  - all customers with area code xxx
- Dictionaries
  - insert/delete/search entries (on e.g., cell phone fast
  - store and search dictionary, find closet word for spell-check
- Replacing BSTs, AVLs, hash tables in some cases
  - since worst case trie look up is O(m)
  - vs. O(n) BST; $(1.44 * \log_2(10^9 + 2)$ AVL;

Heaps

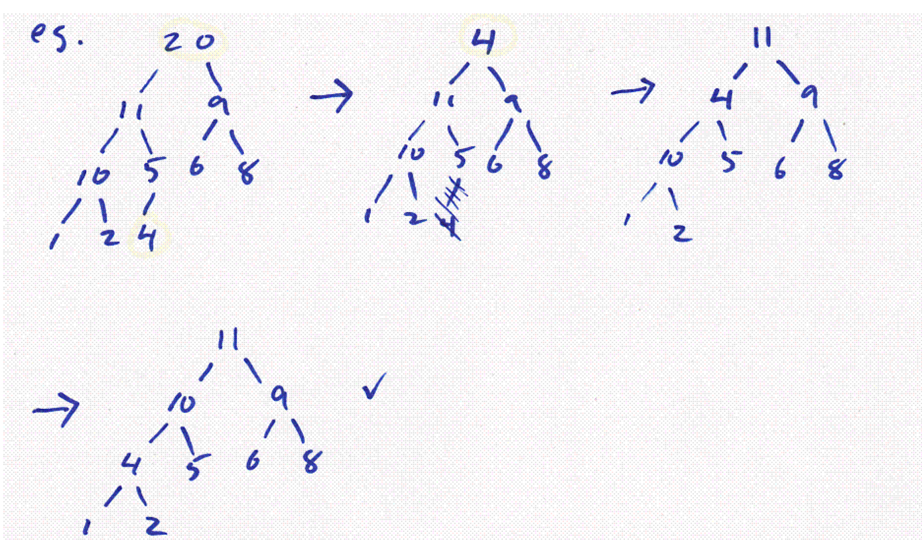
- Complete Binary Tree (structure property)
- Each node has key <= its parent key (sort property)

Heaps uses:
- Priority Qs.
- Sorting Algorithm (heapsort)
- Graph algorithms: shortest path
- Selection algorithms: quickly find max, min median, k^{th} largest item

Example of Max heap



How to remove item from heap?
- For max heap, can only remove largest (highest priority) item
- take it from root and fix "hole"
- fill hole with key in right most leave on bottom level (delete leaf)
- restore short property by bubbling/exchanging down, new root key into correction position (largest child exchange



Removal Algorithm:

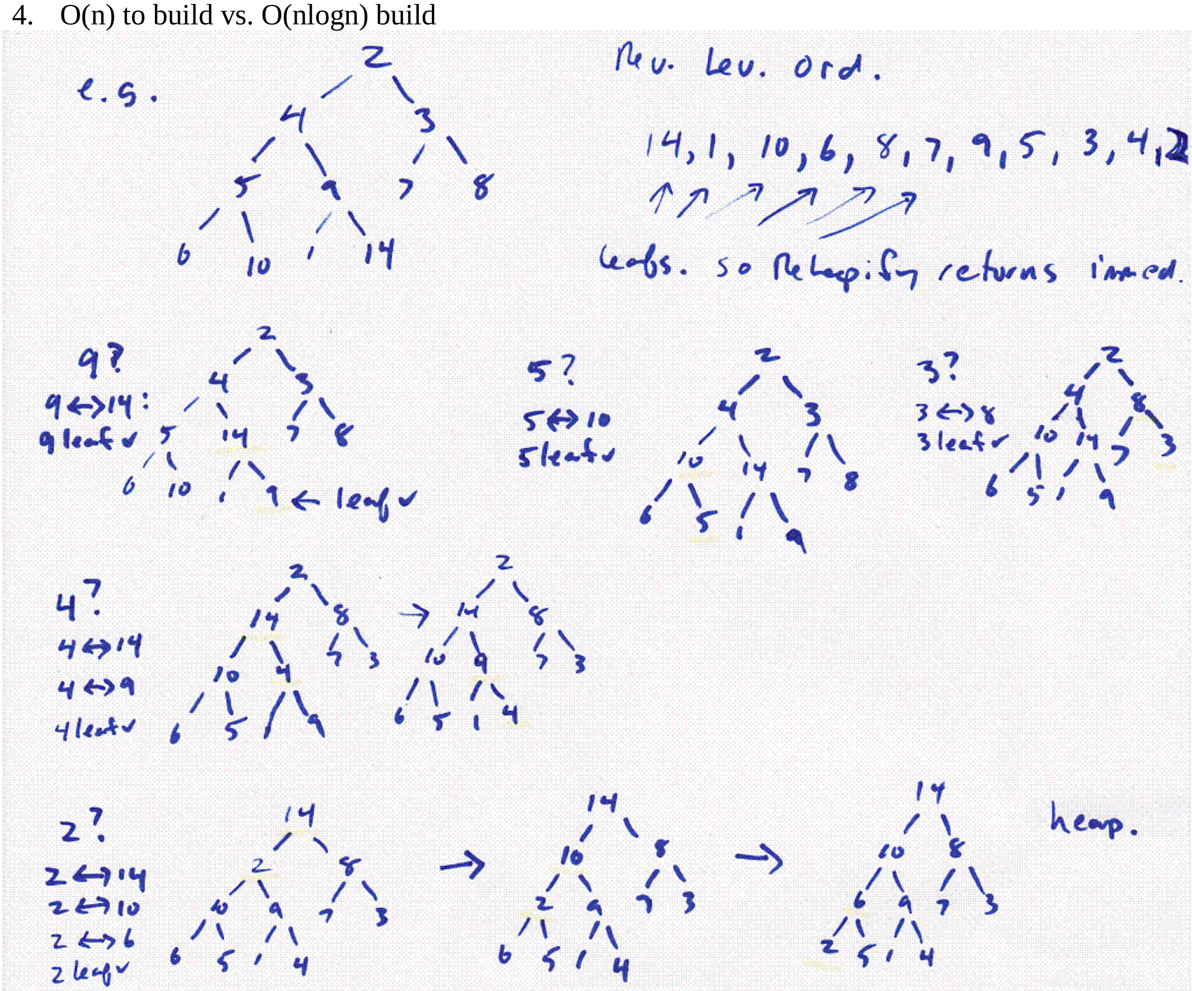
```
Remove (Heap H)
  if empty(h) return NULL
  Removed =  root(H)
  copy (root, lastInLevelOrder(H))
  deleteNode(lastInLevelOrder(H))
  if !empty(h)
    Reheapify (H, root)
  return(Removed)

Reheapify (H, N)
  while !leaf(N)
    M=largest.child(N)
    if keyn >= keym return
    exchange (M, N)
```

Worst case is O(logn)

Build Heap from N unsorted items

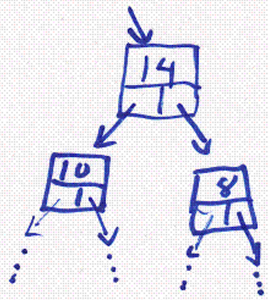
1. Put items in Complete BT structure
2. Establish sort property

   for each node N, in reverse-level order
3. Reheapify (H, N)
4. O(n) to build vs. O(nlogn) build



Heap Implementation

- Linked nodes



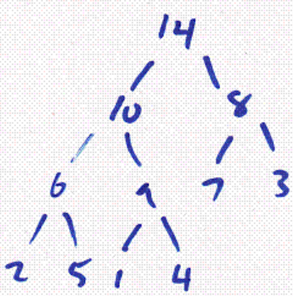
  - space?

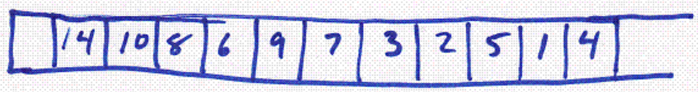
N(int) +
2N(ptr)
+ ptr

- Array.

  - efficient use of space (no "holes" in array)
    because ...

COMPLETE BT





A[i]'s children at
A[2i]
A[2i+1]

  - fast (random access)
  - space?

N(int)

https://visualgo.net/en/heap

https://www.cs.usfca.edu/~galles/visualization/Heap.html