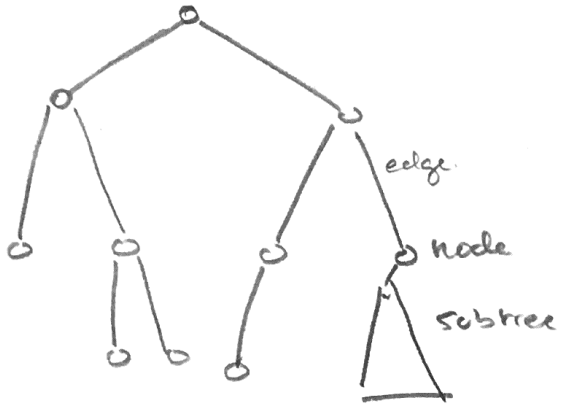
Tree Data Structures

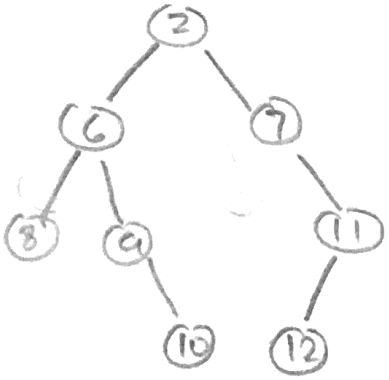
Trees

Trees are a node connected to a number of subtrees.



Traversing trees.

- preorder : visit node, recurse on children
- inorder : recurse on first child, visit node, recurse on second child. (binary trees only).
- postorder : traverse children, visit node.



preorder :	2	6	8	9	10	7	11	12
inorder :	8	6	9	10	2	7	12	11
postorder :	8	10	9	6	12	11	7	2

- refer to code in printout.

Runtime

- So all traversals run in time linear in # of nodes. $\Theta(n)$.

- you can bound # of nodes in k-ary tree

$$h+1 \leq N \leq \frac{k^{h+1}-1}{k-1}$$

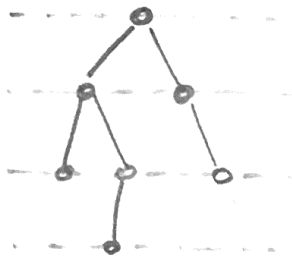
h = height, k = max # of children.

- You can bound height ...

$$h \in \Omega(\log_k n) = \Omega(\log n)$$

$$h \in O(n).$$

- Iterative deepening.




```
void doLevel(Tree T, int level)
    if level = 0:
        visit T
    else:
        ∀ child c of T:
            doLevel(c, level-1)
```

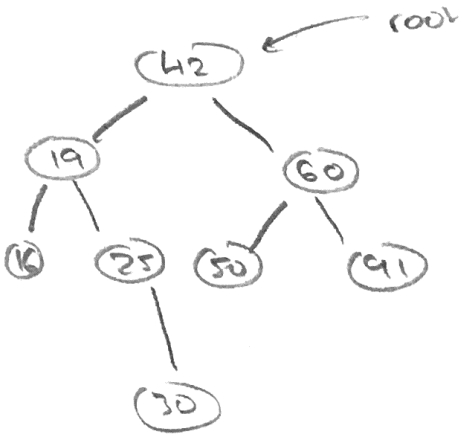
runtime: For bushy trees $\Theta(n)$ and for right leaning tree $\Theta(n^2)$

Binary Search Trees

To store data in an ordered manner you can use a BST. It is a tree w/ following invariant.

- all nodes to the left of parent have smaller keys
- all nodes to right have higher.

Consider...

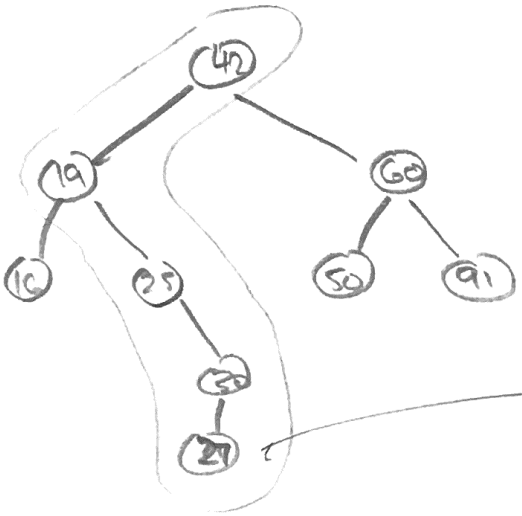


← valid BST

Operations on BSTs.

Finding: idk just do it if target key less than root go left, if greater go right if equal then smile

Add: insert 27.

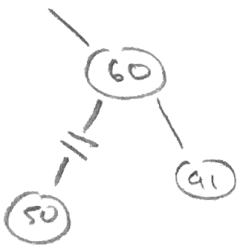


algorithm: Follow the path down and insert appropriately.

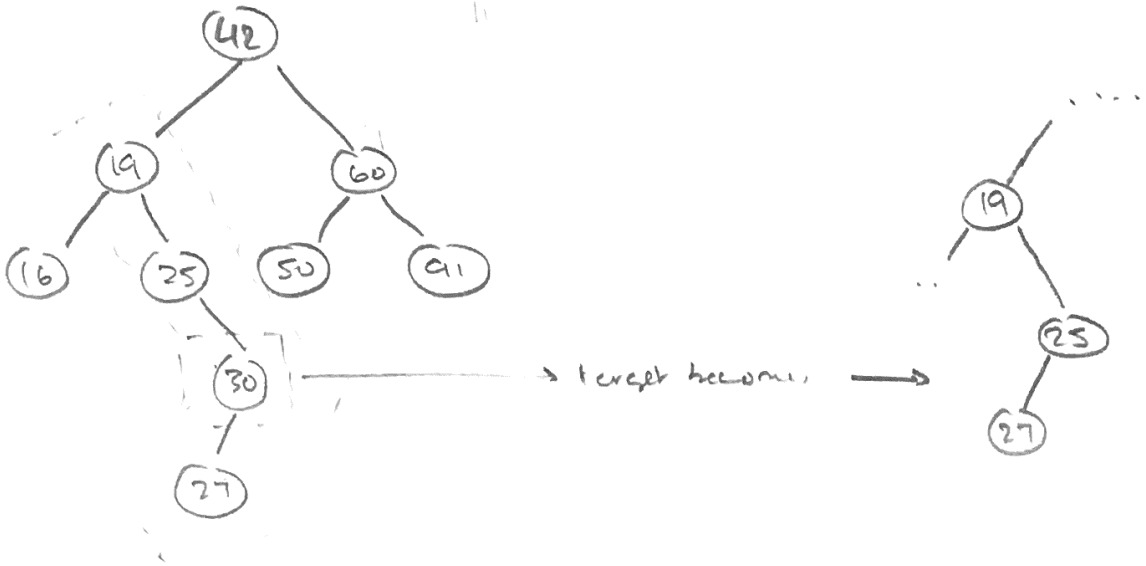
adds 27 here

Deleting: Split into three cases

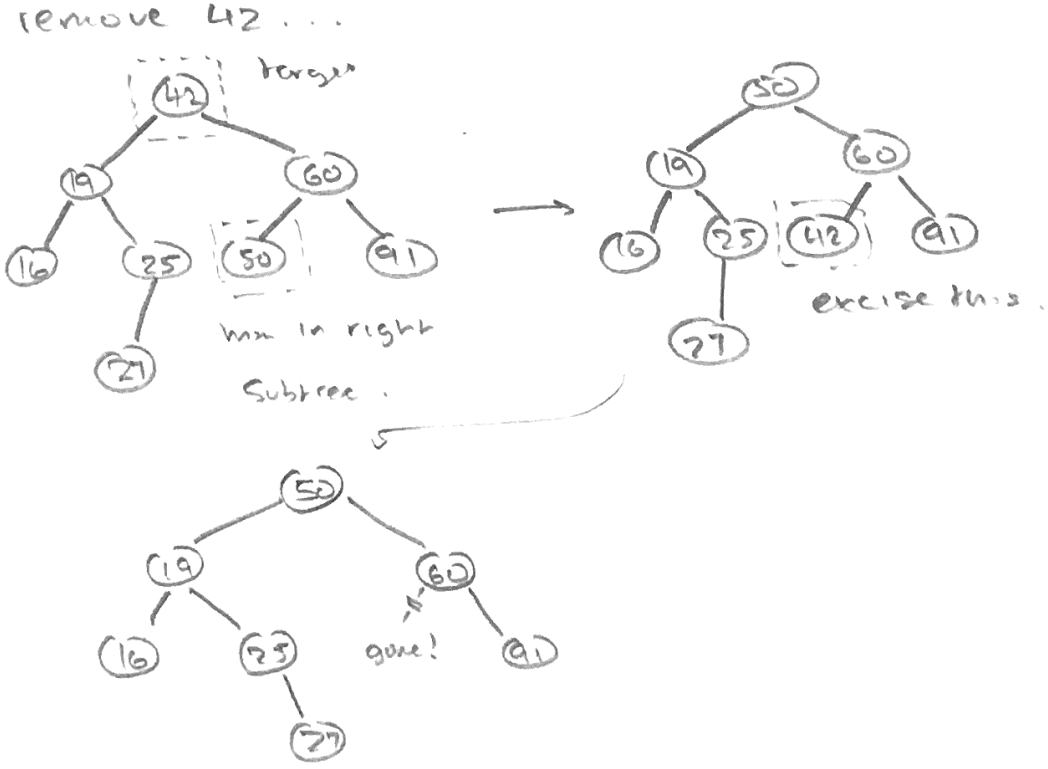
1. no children: just delete it. ex: delete 50



- Node has one child: just connect the link. Ex: delete 30.



- Node has two children: Find minimum of right subtree, swap then delete.



Quadtrees

Goal: store information on a discretized Cartesian plane.

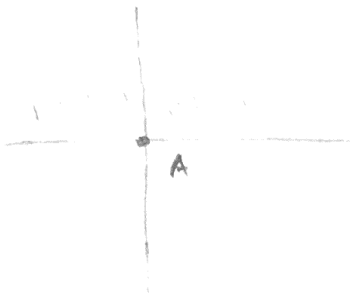
- idea: recursively divide region into quadrant and store points in terms of NW, NE, SW, SE.

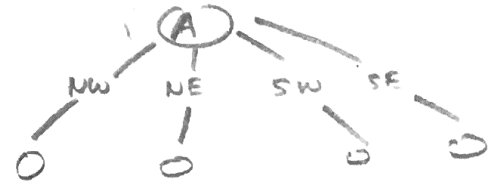
Definition: A quadtree is either...

1. empty
2. an item at point (x,y) called the root plus 4 children quadtrees containing points: NW, NE, SW, SE

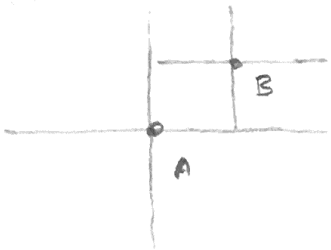
Example:

point A added.

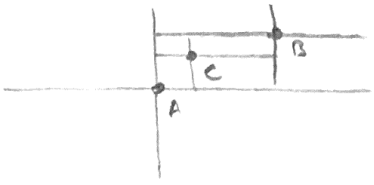


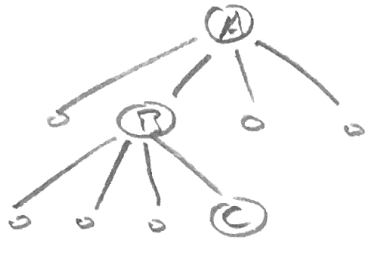


point B added.



point C added.





etc. on...

PR - Quadtrees

You can define point region quadtrees. Instead of centering axis at point just draw in the axis arbitrarily

- check point out ...

Heaps

A special kind of tree and is constrained by property

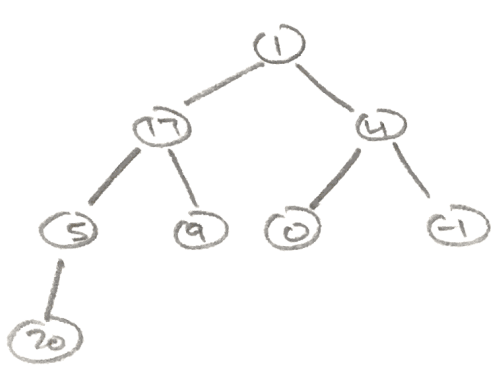
- Both children are less than the parents key (max heap)

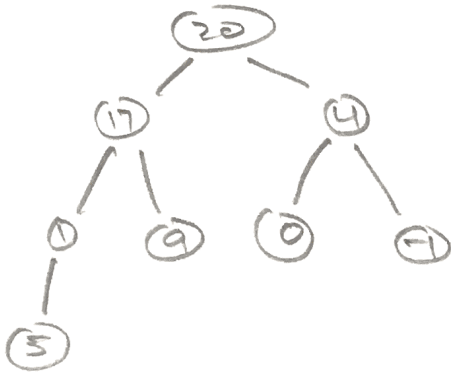
heaps are bushy (balanced) because you are allowed to insert anyways.

Operations:

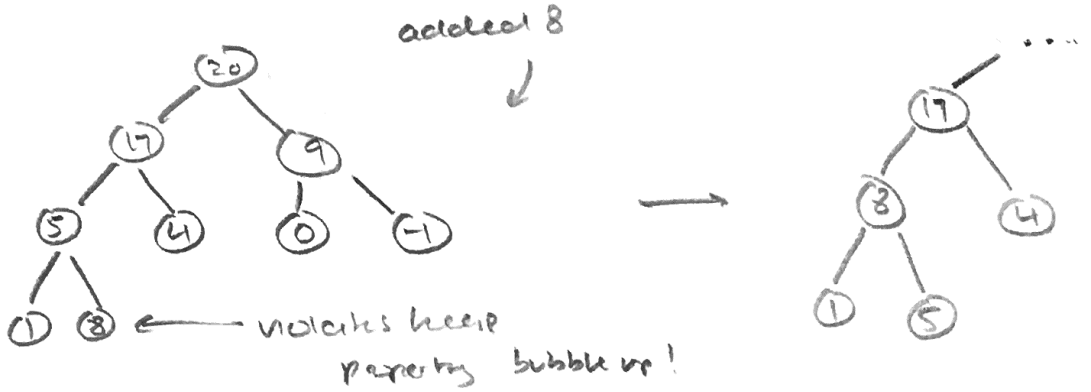
- Heapify - used to construct a heap. Begin with near complete tree and ∀ elements in reverse order sink it down.

1 17 4 5 9 0 -1 20

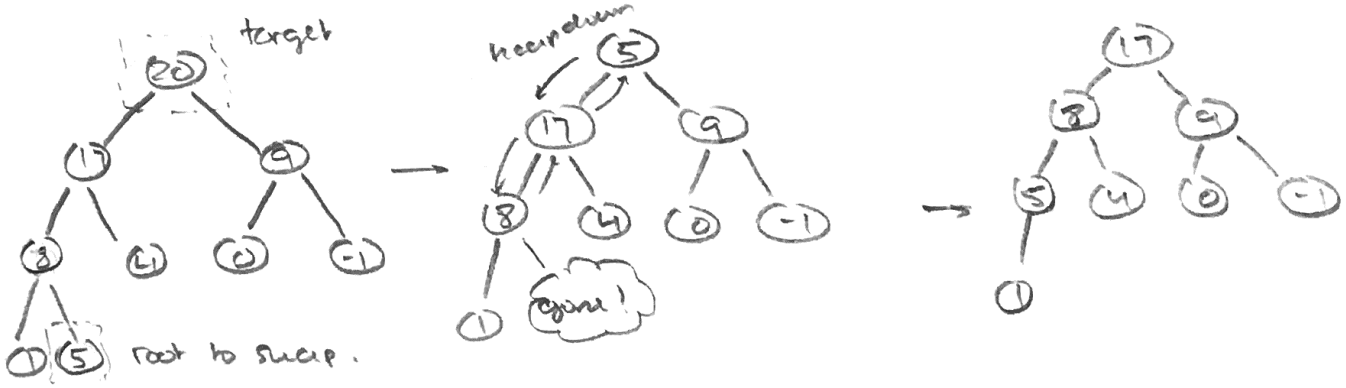




- bubble up - move a node up until nothing is violated.
- bubble down - move a node down until nothing is violated. (Swap parent with largest child)
- insertion - add a next null leaf and bubble up.

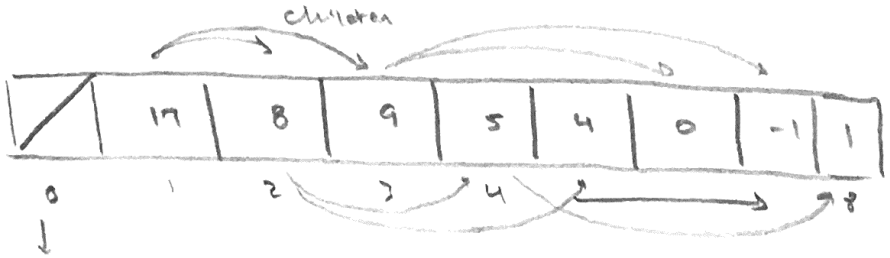


- remove-max. Swap root with last added leaf node. Then, delete leaf, bubble down root.



Heaps in arrays:

- you can compactly store a heap because of near completeness.



child of a node is stored at $2k$, $2k+1$

parent of a node is $\lfloor \frac{k}{2} \rfloor$

Range Queries on BSTs

Suppose we want to find all nodes within a range (lower, upper). Algorithm on a BST is simple.

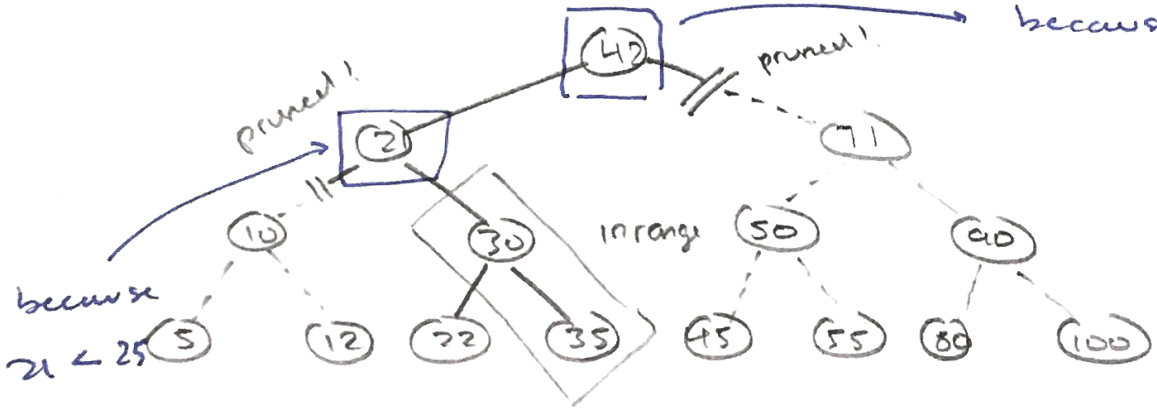
```
range ( node , upper , lower ) :
    if ( lower < node ) :
        range ( node.left , upper , lower )
    if ( lower ≤ node ∧ node < upper ) :
        in range!
    if ( upper > node )
        range ( node.right , upper , lower )
```

lower < current. if ~~in range~~ gotta recurse left

yes in range!

if upper > current then recurse right.

example: search for $x \in [25, 40)$



runtime: time $\in O(h + M)$ h: height and M amount of data in range.

Balanced Trees

"A lot of good time bounds on BSTs are only true when BST is balanced.

we can thus control self balancing trees

$\Theta(\log n)$ performance for searches insert delete

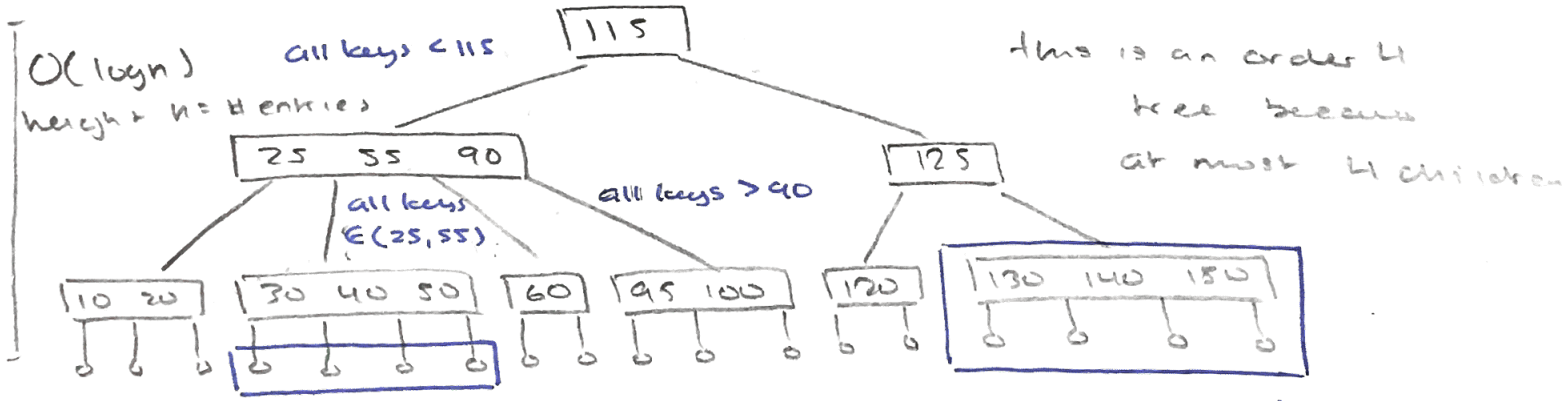
B-Trees

The idea is to enforce a tree to only grow and shrink at the root. B-trees achieve this by maintaining constant # of children

Def: M-order B-tree is an M-ary search tree $M > 2$ where each node except root has between h children.

$$\lceil M/2 \rceil \leq h \leq M$$

The best way to explain is by example.



best case height is actually...

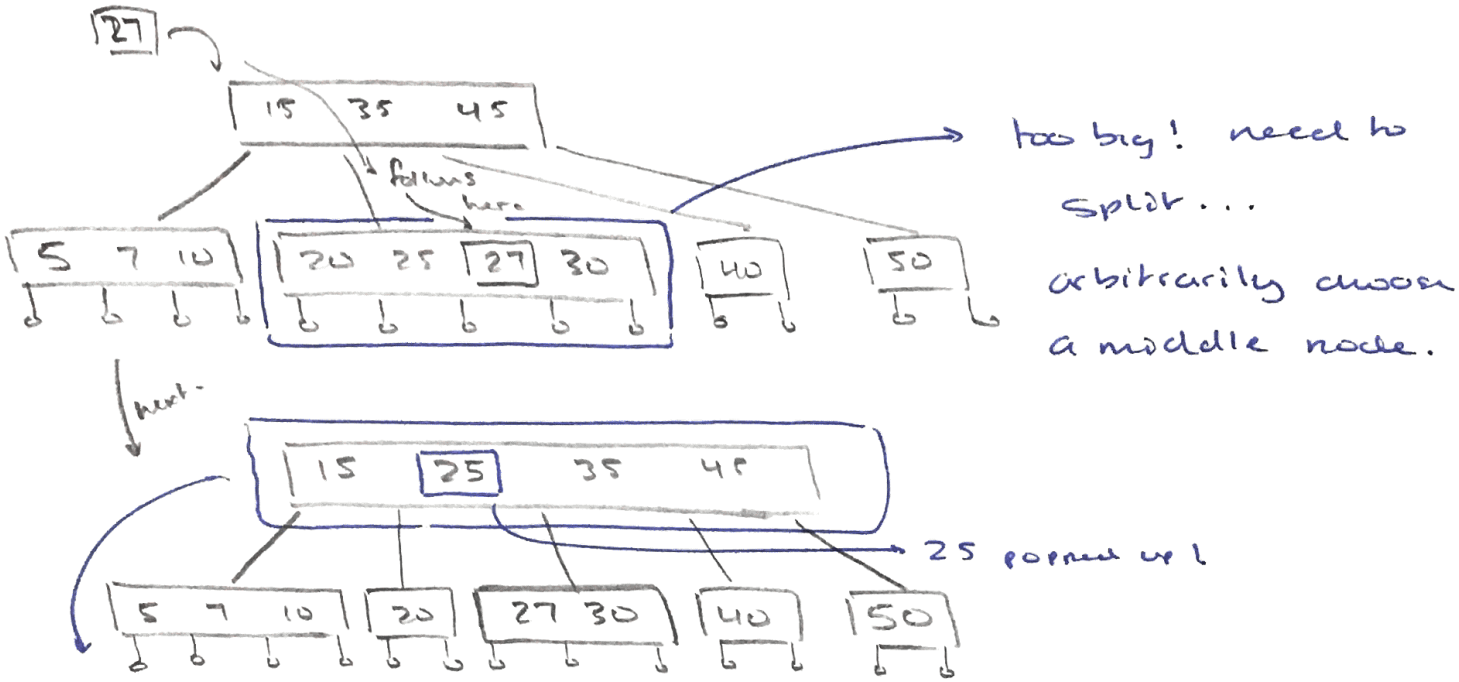
$$\lceil \log_m (n+1) \rceil$$

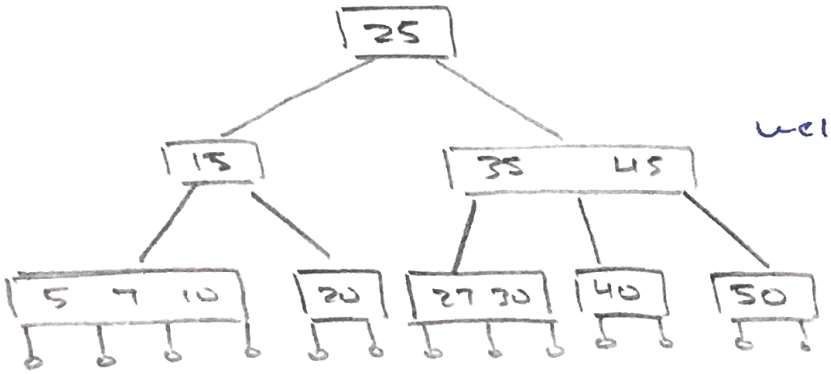
worst case...

$$h \leq \lfloor \log_d \left(\frac{n+1}{2} \right) \rfloor \quad d = \lceil m/2 \rceil$$

adding to a B-tree: First place node by traversing tree, then split and push up appropriately.

Consider adding 27 to...

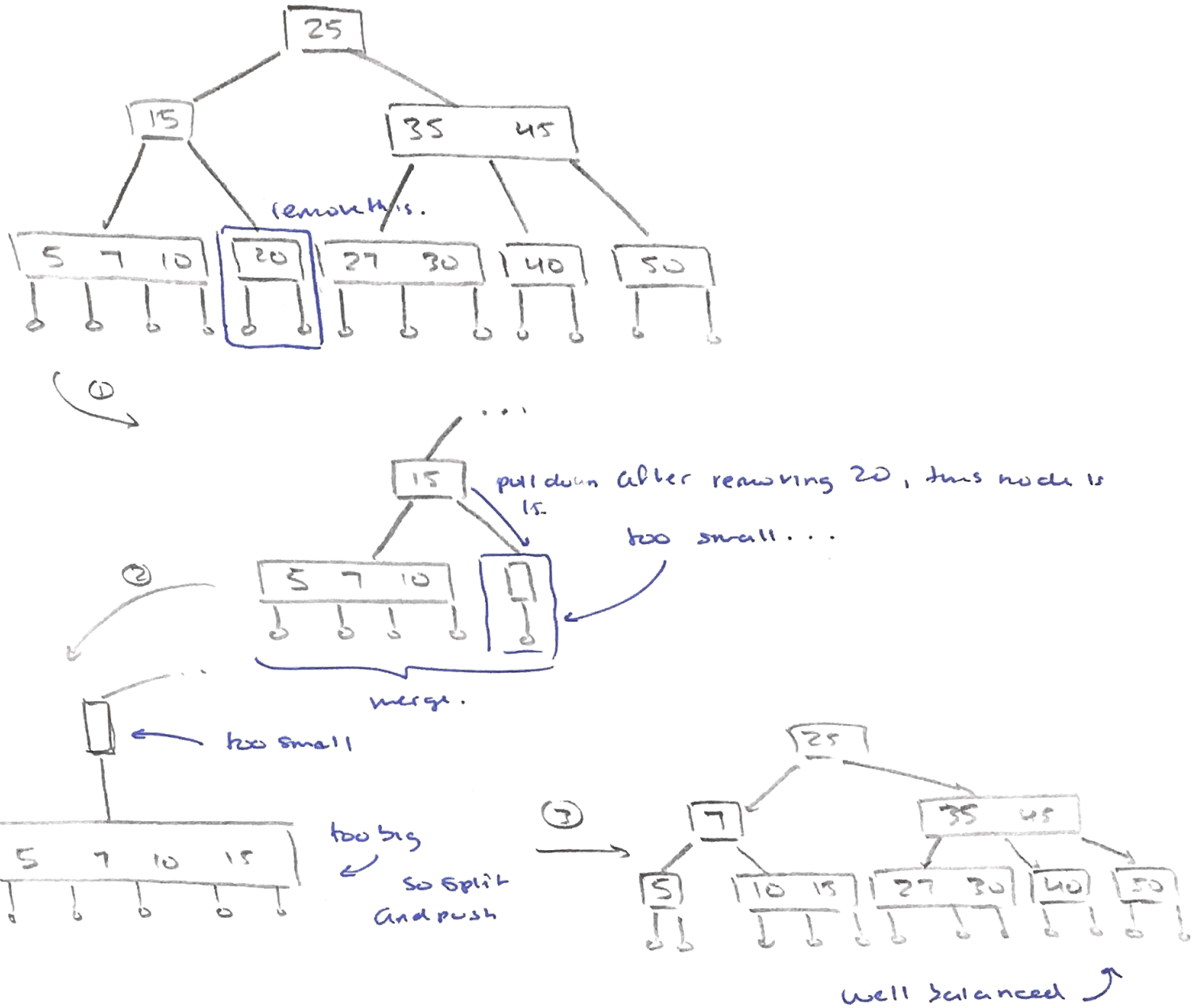




Removing: To delete a node, remove a key. If too small, move down from parent, if parent too small breaks up another child.

- move key to be deleted all the way down
- if deletion leaves original node too small:
  - pull down parent key and merge w/ sibling
  - split and send key back up

Example: remove 20.



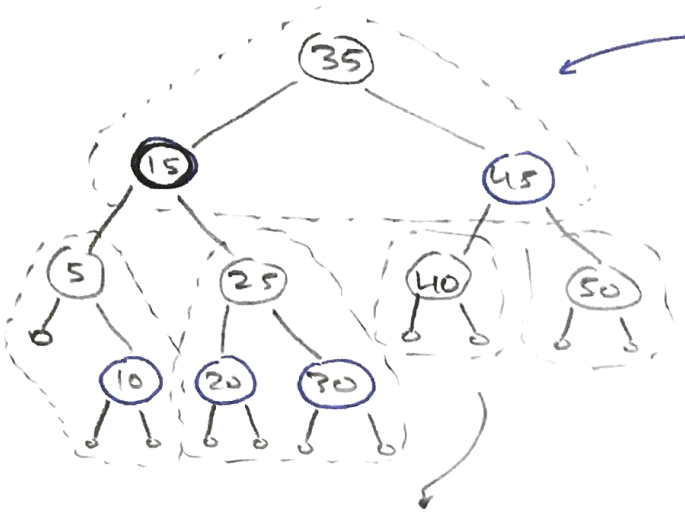
Red-Black Trees

Turns out 2-3-4 trees are an alternate representation of red black trees.

Here we color nodes and provide different restrictions:

1. The root node and all null leaves are black.
2. Every child of a red node is black.
3. $\forall$ paths root → leaf, # black nodes constant.
4. Every internal node has 2 children.

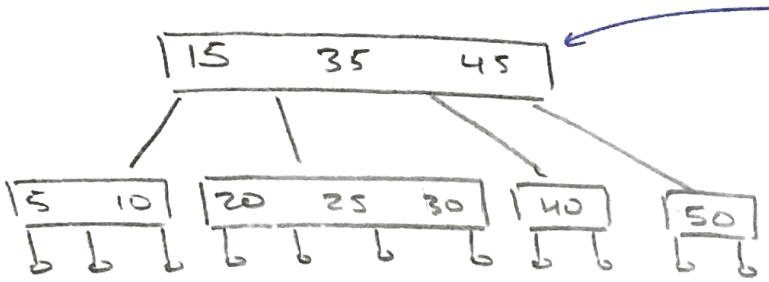
example.



note that all paths have 2 black nodes.

also ▣ represents red...

but also this corresponds to a 2-3-4 tree!



each black node + 2 red children corresponds to a big node.

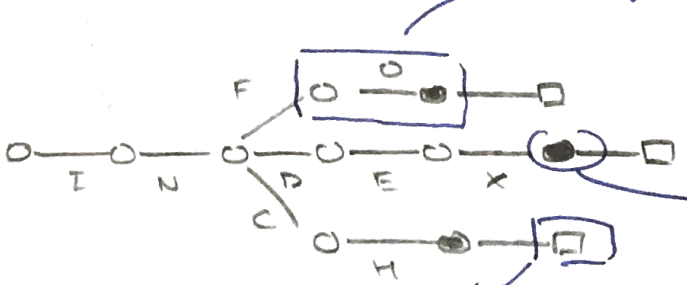
- to do operations convert RB tree to B-tree and vice versa. (Note all record is $\Theta(\log N)$) N = # of elem.

Tries

If comparison sort is to Red black + B-tree then tries are to radix sort

We want to guarantee $O(L)$ search time in spite of $\Theta(M)$ comparison → $\Theta(ML)$ operations. where L = key length

Tries example



edges correspond to letter

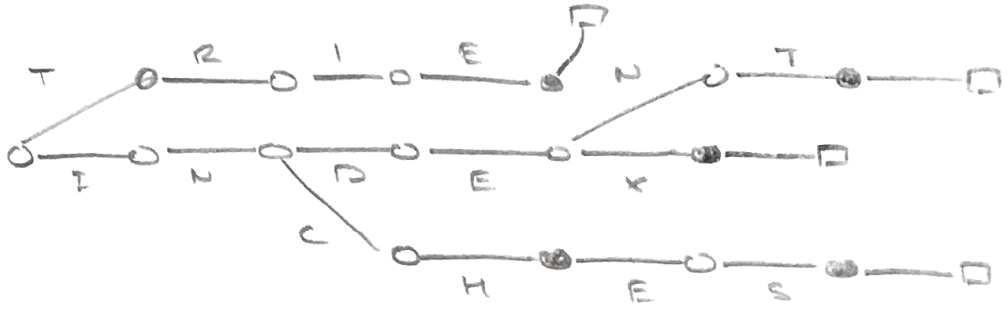
dark circle represent terminal node. traversing to that node records "index"

null nodes.

adding is easy trash ...

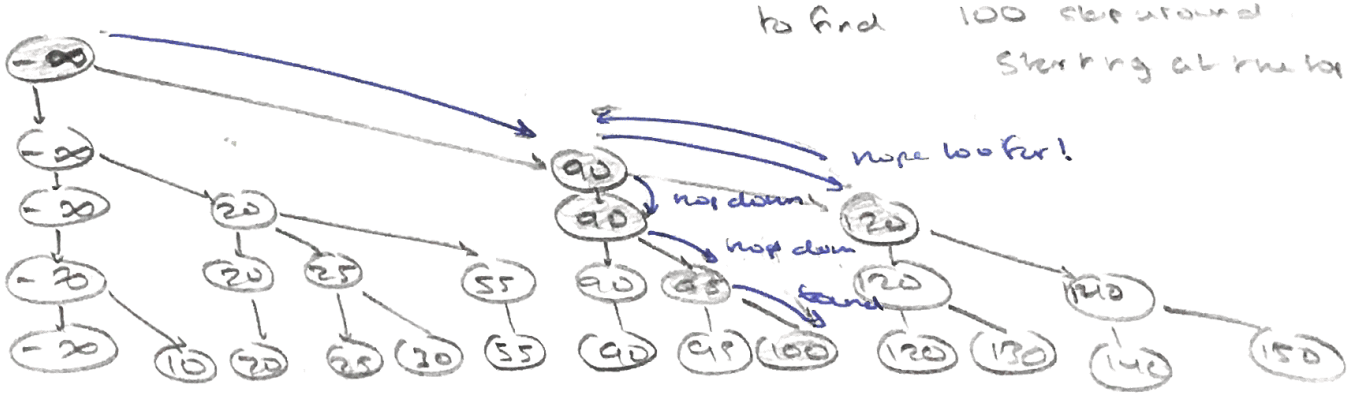
add · indent , tree , inches.

note $\Theta(L)$ performance for searches insertions and deletion.

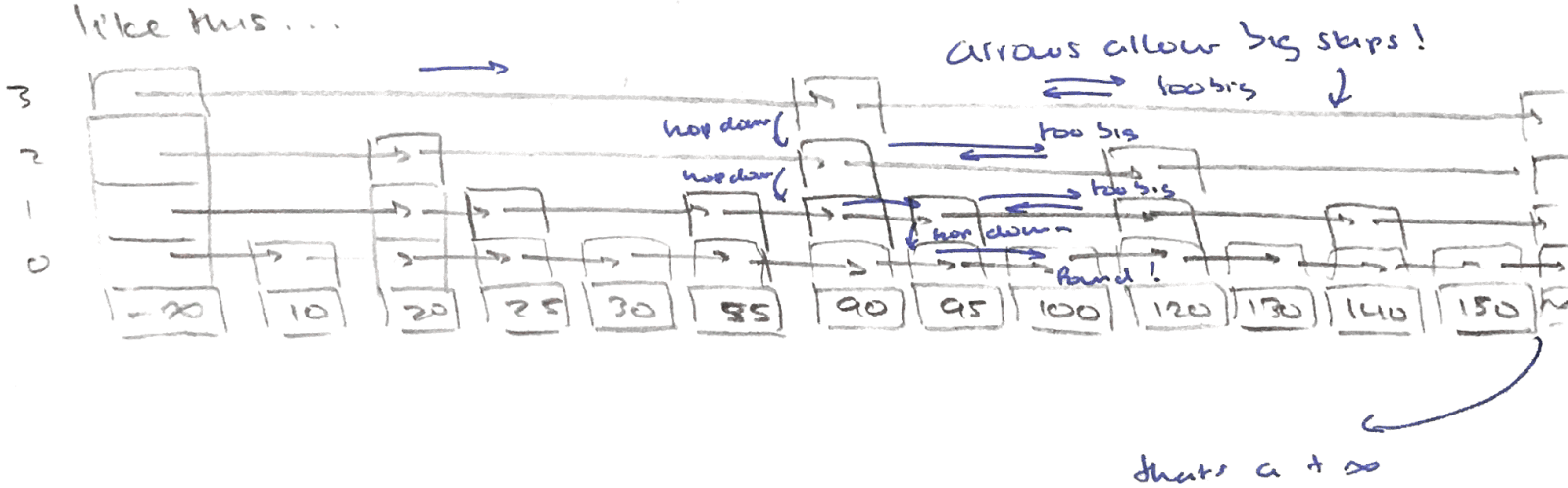


SkipList

More of a sequential data structure but can be treated as an n-ary search tree.



We require skip lists be sequential. Often times it looks like this...



to search: start at highest level, jump forward until overshoot. then move down a level. continue until element found or at level = 0.

to add and remove: randomly generate height of a node based on 1/2 probability. cracked.

- gives probable $\Theta(\log n)$ performance: searches insertion and deletion.

Converting 2-3-4 → Red-Black