Advanced Data Structures

Skip Lists
+ Scapegoat
Trees

Recap:

- HW1: partially done, posted by Wed.
- Sub on Friday & next Wednesday

(no class next Monday)

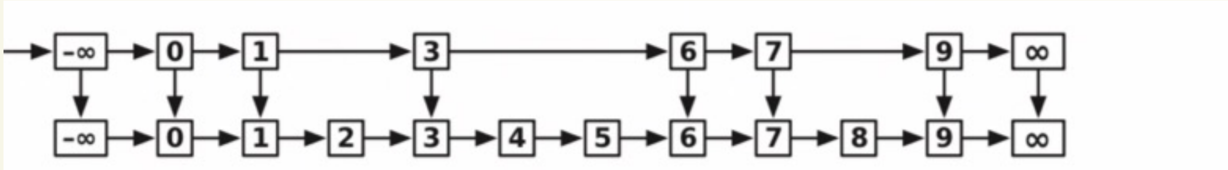
Next: Skip Lists

(Bill Pugh, 1990)

An alternative to balanced binary search trees.

Essentially, just a sorted list where we add shortcuts - but to speed up, we'll duplicate some elements.

For each item, duplicate with probability $\frac{1}{2}$:

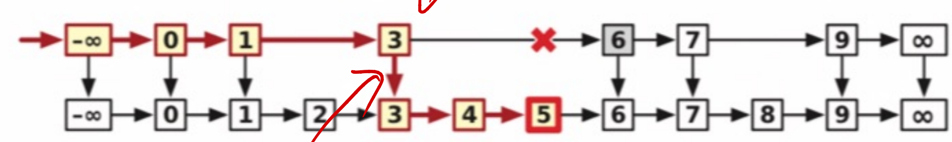


A linked list with some randomly-chosen shortcuts.

plus some sentinal nodes

Searching:

next ptr



Searching for 5 in a list with shortcuts.

down ptr

Scan in top list.

If found, great!

Otherwise:

Look until next element is too large

follow down ptr

+ Scan lower list

Some probability!

Expectation: $\sum_{\text{values possible}} (\text{value})(\text{prob of value})$

<u>Ex:</u> 6 sided dice

$E[\text{value}) = \frac{1}{6}\cdot 1 + \frac{1}{6}\cdot 2 + \frac{1}{6}\cdot 3 + \dots + \frac{1}{6}\cdot 6$

$= 3.5$

Each node is copied with prob $= \frac{1}{2}$,

$E[\#\text{ nodes in top}] = n/2$

worst case exp. # comp. in top is $n/2$

Goal: Bound expected # of comparisons

Prob[a node is follow by k without duplicates]

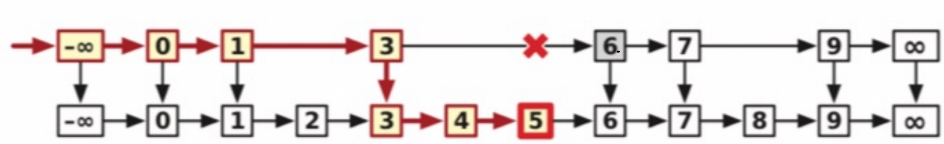
$$= \underbrace{\frac{1}{2}\cdot\frac{1}{2}\cdots\frac{1}{2}}_{k} = \frac{1}{2^k}$$

So: Expected[# comparisons in lower list]

$$= 1 + 1\cdot\frac{1}{2} + 1\cdot\frac{1}{2^2} + \cdots + \frac{1}{2^k} = \sum_{k=0}^{n} 2^{-k}$$

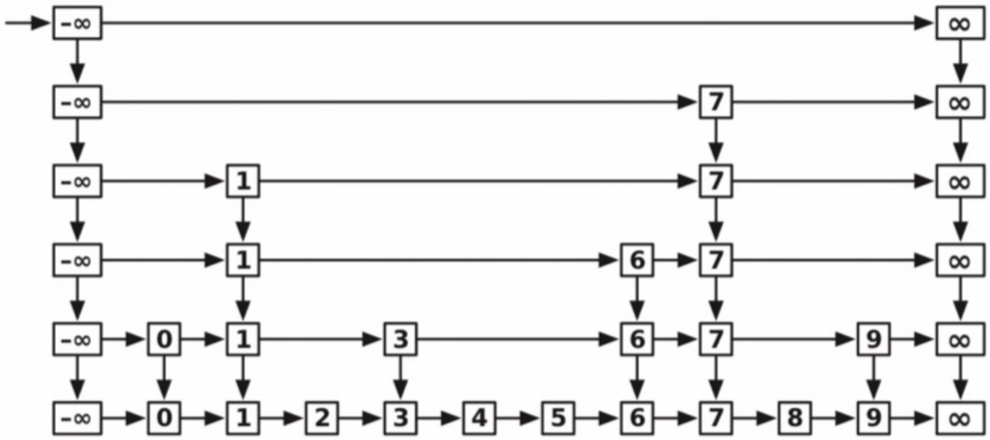
$$\leq 2$$

$\Rightarrow$ expect $\frac{n}{2} + 2$ comparisons



Searching for 5 in a list with shortcuts.

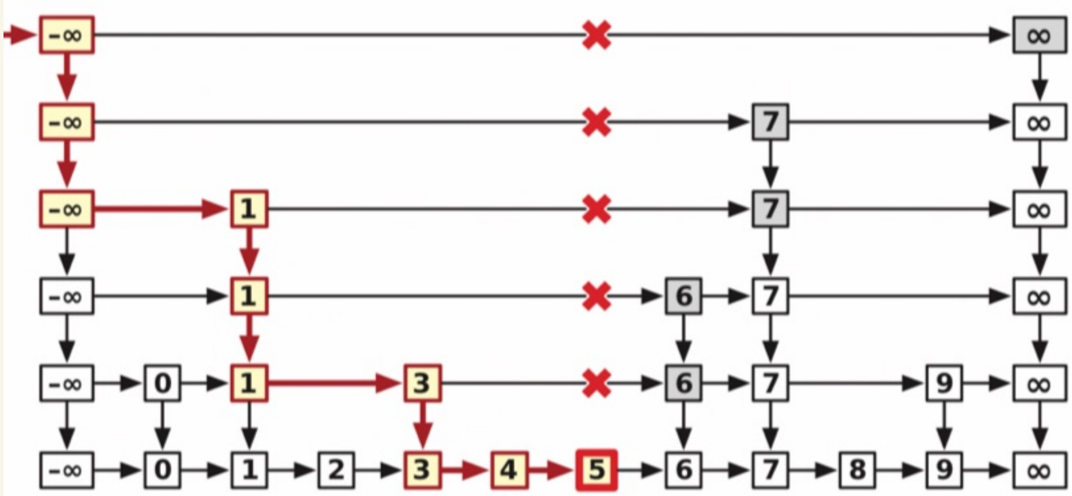
What next? Recurse!



A skip list is a linked list with recursive random shortcuts.

To search:

<u>SkipListFind(x, L):</u>
  $v \leftarrow L$
  while ($v \neq$ Null and $key(v) \neq x$)
    if $key(right(v)) > x$
      $v \leftarrow down(v)$
    else
      $v \leftarrow right(v)$
  return v



Searching for 5 in a skip list.

How many levels?

Well, $E[\text{size at level } i] = \frac{1}{2} E[\text{size at level } i-1]$

So (intuitively):

$O(\log n)$ runtime

Each time we add a level,
$E[\#\text{ searches}]$
goes down by $\frac{1}{2}$.

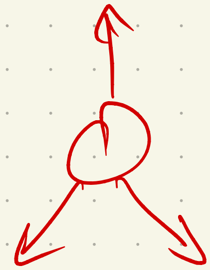
More formally?

See posted notes!

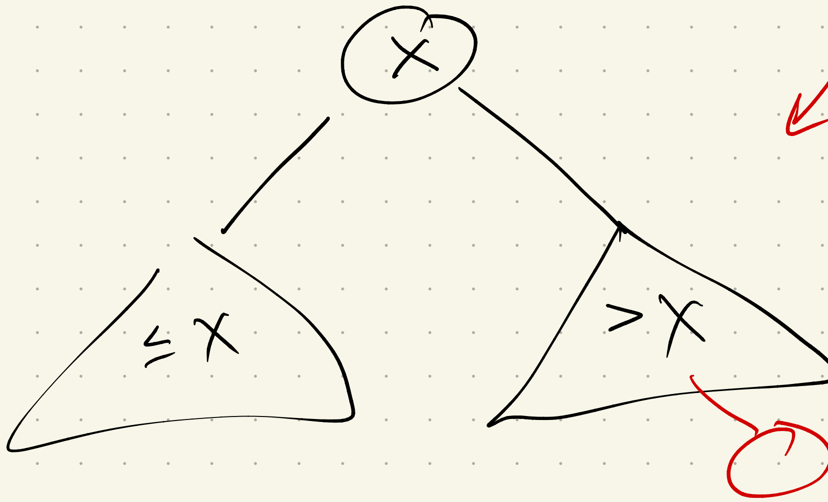
(Assumes some probability...)

Binary Search Trees:

What is the "best" one?



Recap:



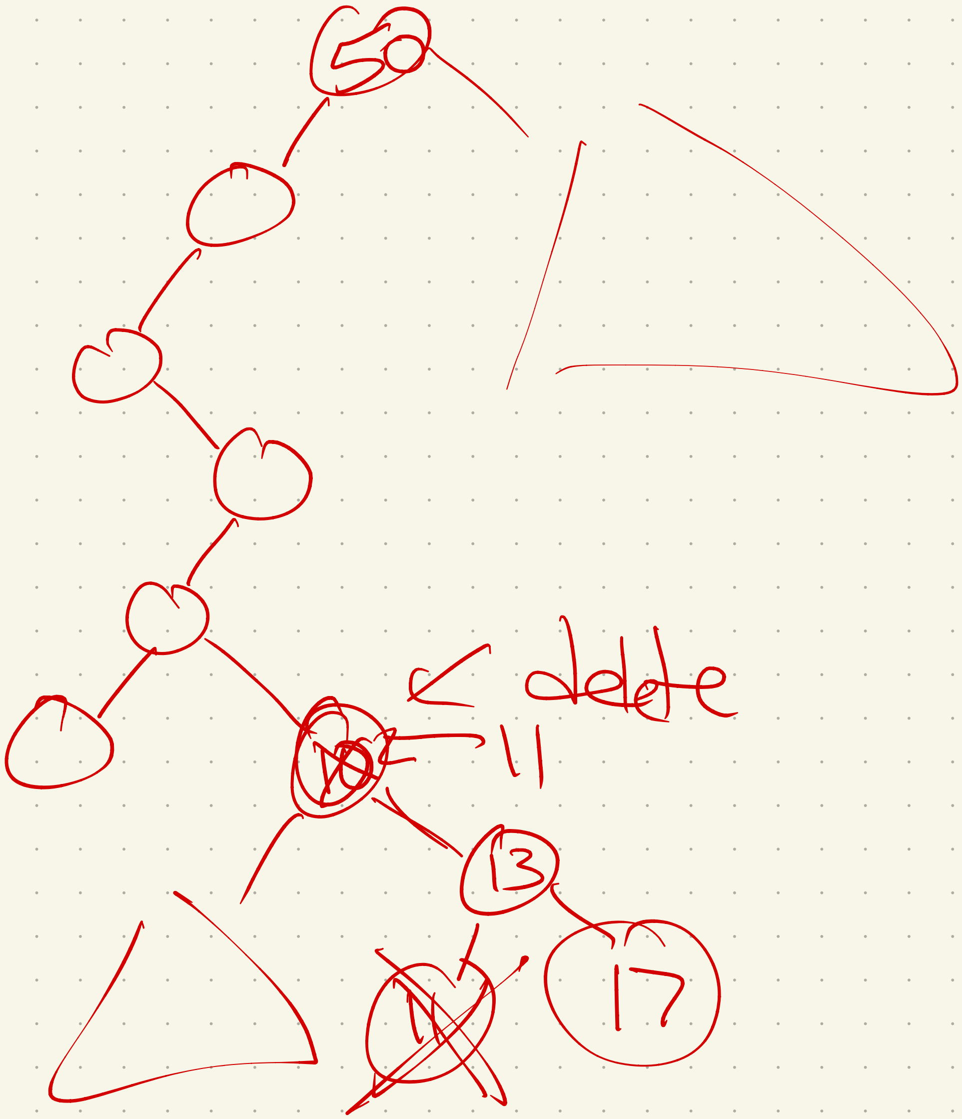
Search:

start at root
if v == target
  return yes
else if ≤ target
  recurse left
else
  recurse right

Insert:

while (v has children)
  if x ≤ v
    go left
  else
    go right

Delete:



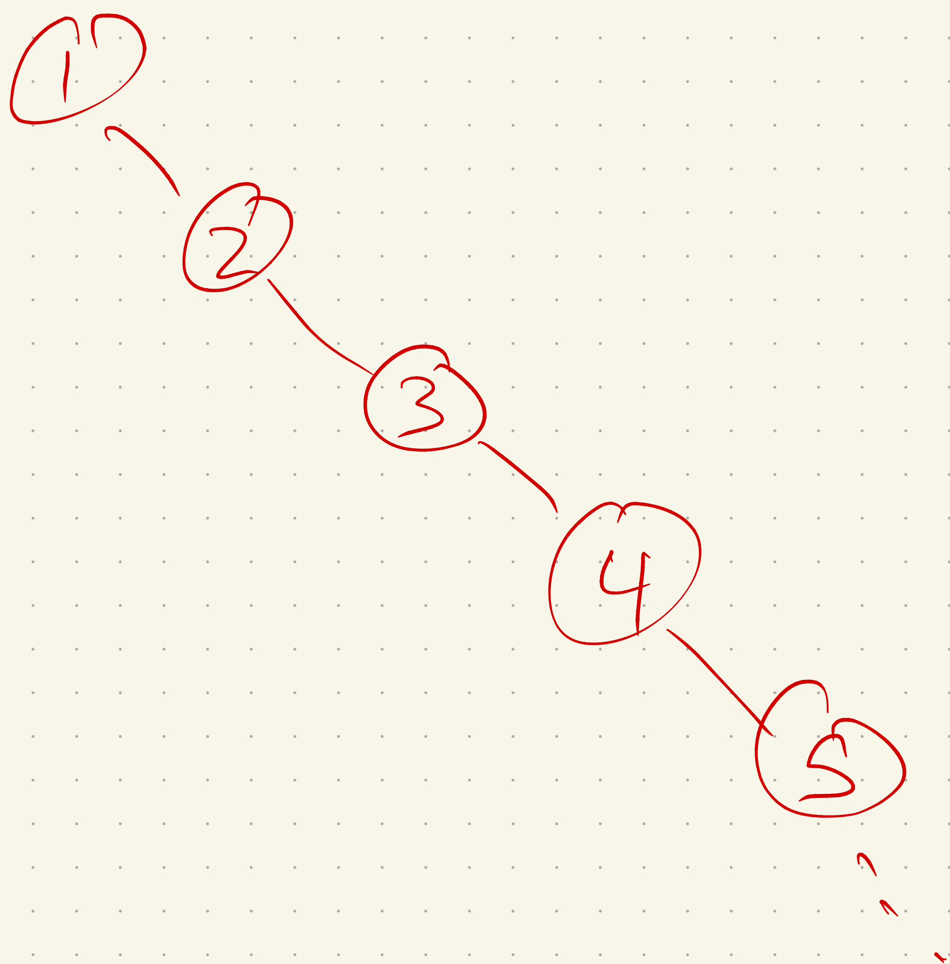
- Find next node on inorder traversal

Data Structures Class

- "Vanilla" BSTs (no rotations or balancing)

Runtime: $O(n)$

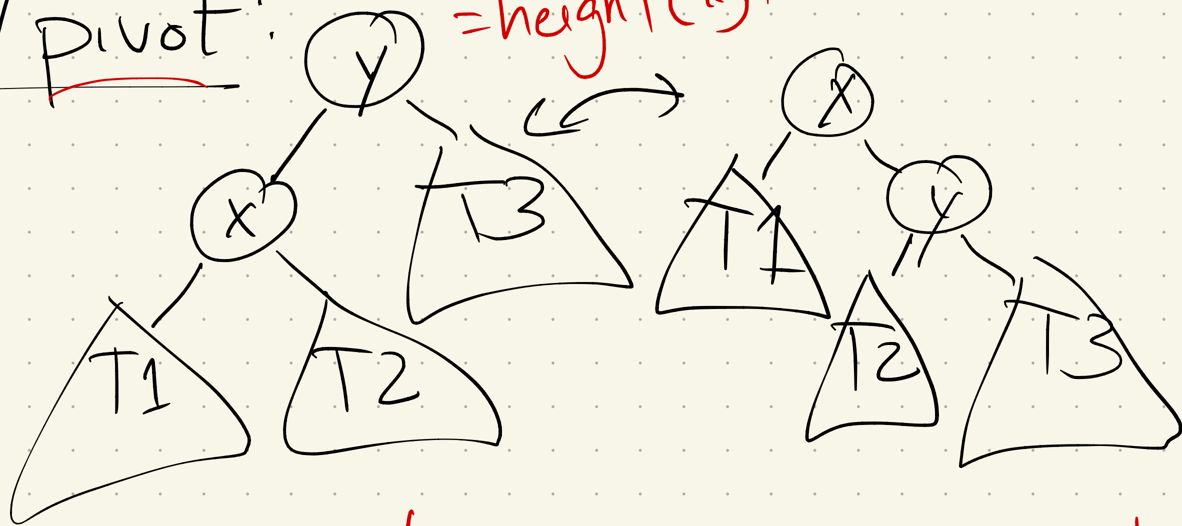
How can it get this bad?



BSTrees: balancing

Rotation/pivot:

height(y) = height(x)+1



unbalanced: left (or right) is too big

- AVL trees ← want $|h(\ell(v)) - h(r(v))| \leq 1$
- Red-Black trees
  - ↳ $O(\log_2 n)$

~~Today~~ This week:
- Scapegoat Trees
- Splay Trees

Terminology I'll assume:

- search key
- node
- left/right child, parent
- internal/leaf node
- root
- ancestor/descendant
- preorder, inorder, postorder

Recap:

- Height(v): distance to furthest leaf in v's subtree
- Depth(v): distance from v to the root
- Size(v): # of nodes in v's subtree

Scapegoat Trees:

[Anderson '89, Galperin-Rivest '93]

Supports amortized $O(\log n)$.

Basic idea:

- Standard BST search
- Delete: mark "deleted" node. When tree is half dirty, rebuild into perfect tree.

Runtime:

Claim: rebuild a perfect tree in linear time

$\Rightarrow O(\log n)$ amortized time

And insert:

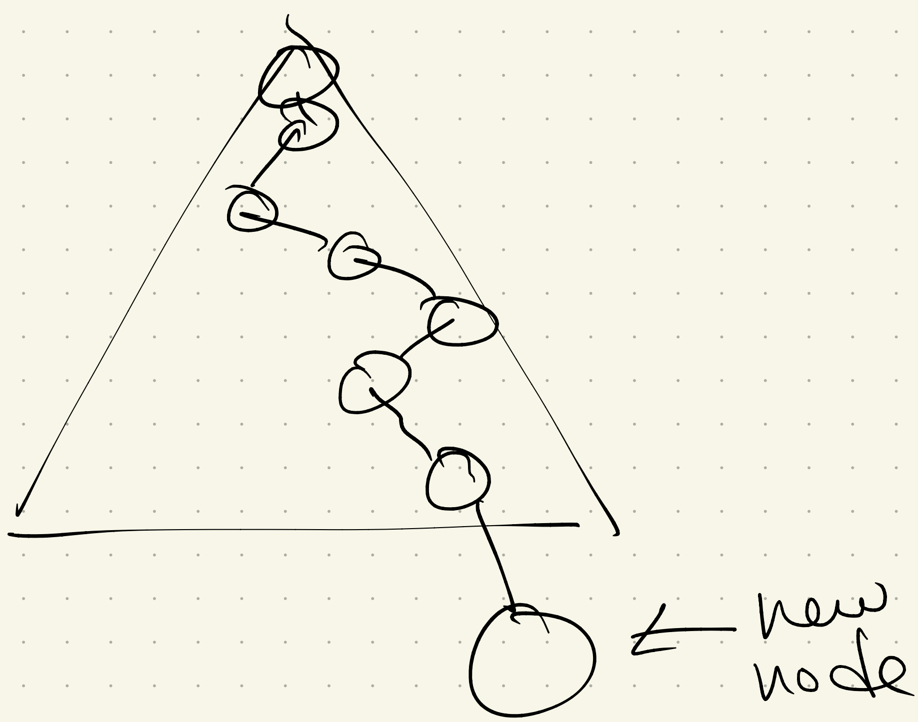
Standard insert

But: if imbalanced, rebuild a subtree containing new leaf

Dfn: Fix any $\alpha > 2$.

A node is imbalanced if $\text{height}(v) > \alpha \lg(\text{size}(v))$

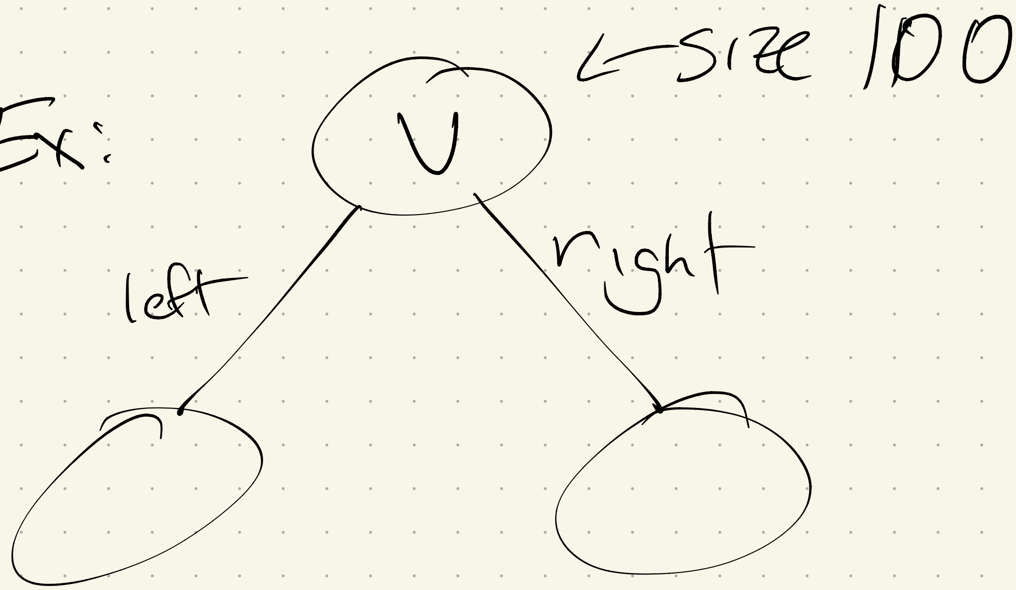
So here:



Let: $I(v) =$

$$\max\{0, |\text{size}(\text{left}(v)) - \text{size}(\text{right}(v))| - 1\}$$

Ex:



Lemma:

Just before rebuilding at v,

$$I(v) = \Omega(n)$$

proof:

If imbalanced, $h(v) > \alpha(\lg \text{size}(v))$

(by dfn of imbalanced)

but left(v) + right(v) were not imbalanced:

$h(\text{left}(v)) \leq$

$h(\text{right}(v)) \leq$

Wlog: Assume insert on left; so:

Some intense math:

So: takeaway

$$I(v) = \Omega(\mathrm{size}(v))$$

This means ~size(v) insertions since the last rebuilding.

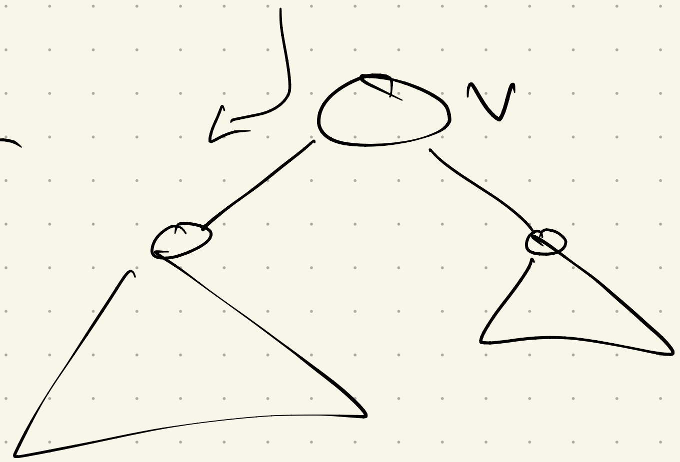
So: rebuild! How?

Several ways to do this in $O(\mathrm{size}(v))$ time.

(HW question!)

Claim: ≤ 1 tree rebuild for each insertion

Pf



Final runtime:

Find:

Delete:

Insert: