B-trees and 2-3-4 trees

Last Time

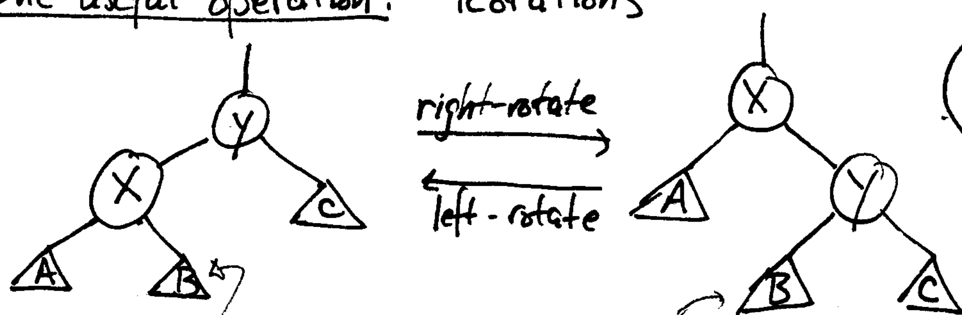
- Binary Search Trees
  Insert, Delete, Search, Min, Max, Successor, Predecessor $\longrightarrow \Theta(h)$
- n random inserts build BST with height $\Theta(\lg n)$ on avg

Adversary can still prepare non-random data the leads to unbalanced trees and thus long execution times approaching $\Theta(n)$

Idea this time:

Rebalancing to guarantee height $\Theta(\lg n)$ deterministically using local rebalancing transformation

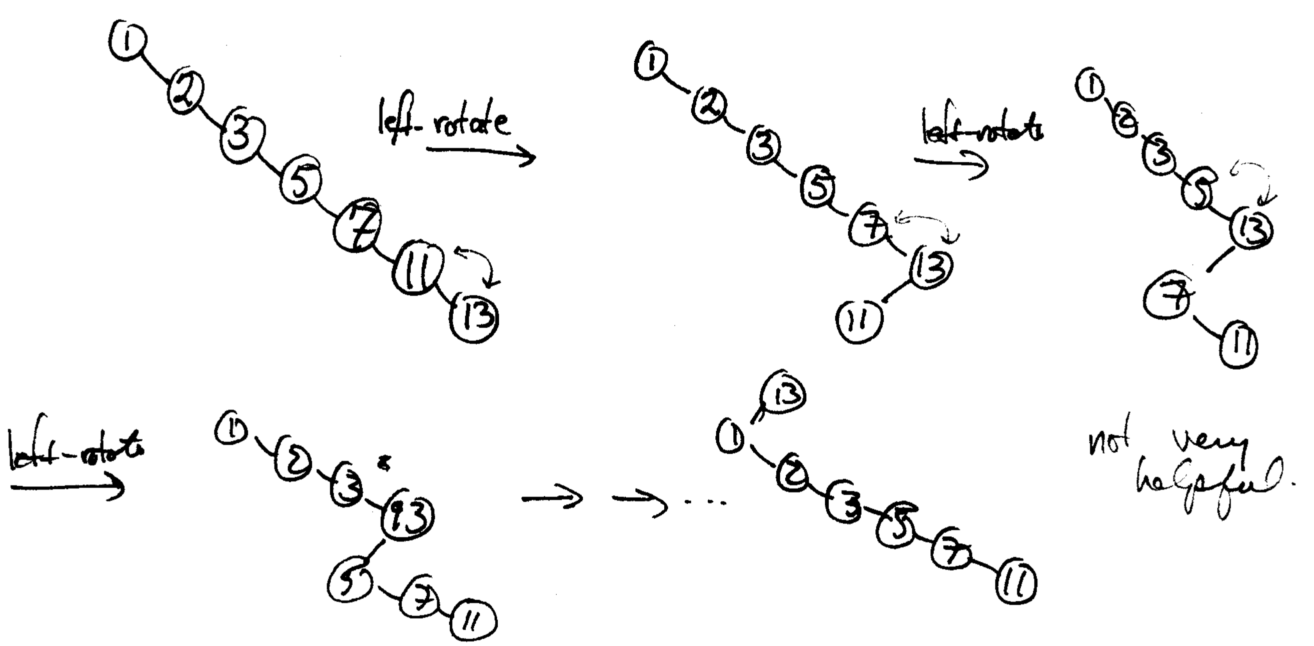
One useful operation: Rotations



all elements of B are $\geq X$ and $\leq Y$ by BST property

all elements of B are $\leq Y$ and $\geq X$

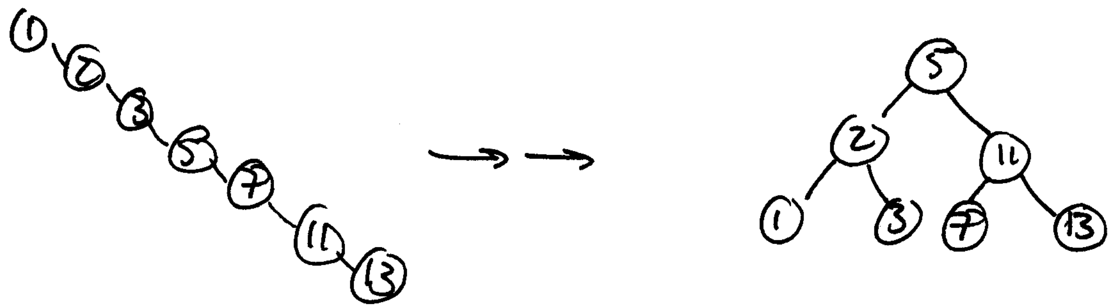
Time: $\Theta(1)$ because only involves moving pointers locally (note that all parent-child relationships are stored)



not very helpful.

Questions:

[1] What sequence of rotations causes?



[2] What is the running time of such an algorithm on a linear tree of length n?

[3] What is the worst-case running time of an algorithm that, after every insertion or deletion, converts BST into ~~linear~~ sorted array/linear tree and applies rotations as above to create balanced tree?

Attempt at Global Soln

Another solution (better) — Red-Black Trees (CLRS chpt 13)

others: AVL trees, AA-trees, treaps, k-neighbor trees, scapegoat trees, splay trees, 2-3 trees, B-trees, 2-3-4 trees

→ Enforce reasonably well balanced trees (difference in longest & shortest paths from root to leaf is ≤ 2-fold for red-black tree). ~~After~~ ~~each~~ After/during each insert/delete rebalance using rotations or other operations

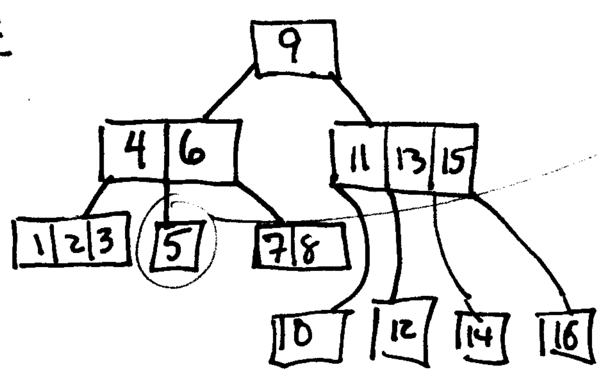
$\Rightarrow \Theta(\lg n)$ operations because $h = \Theta(\lg n)$

2-3-4 trees use a different idea

: relax binary constraint

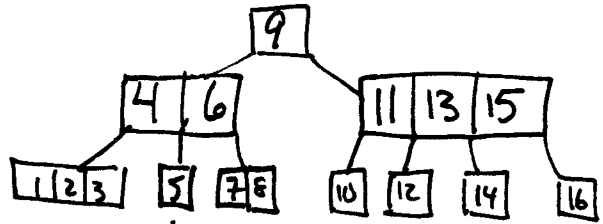
- allow internal nodes to have 2, 3, or 4 ~~[illegible]~~ children
- force all leaves to be at same depth
- nodes with # children $c (\le 4)$ store $c-1$ (≤ 3) keys to facilitate search
- leaves can store up to 3 keys also

example:

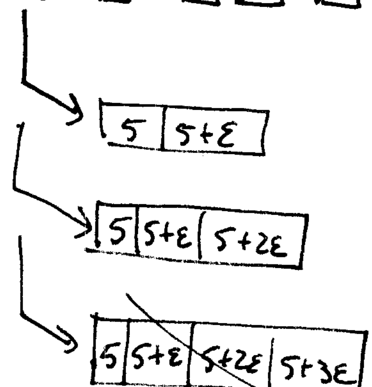


Tree property:

- Each node is a sorted array
- Each child node has keys intermediate in value between pair of elements in parent node.



~~Search for 12~~
• like binary search, but multi-way

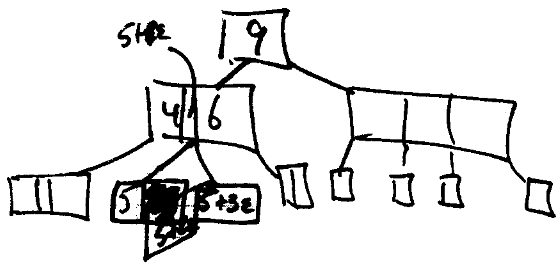


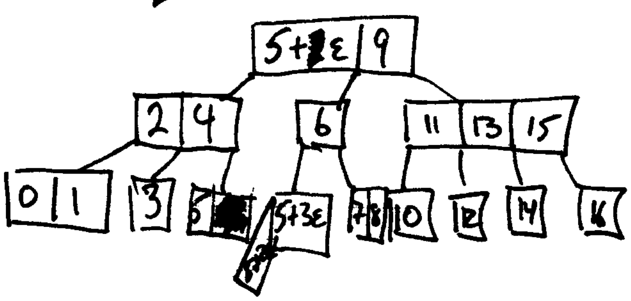
Insert 5+ε

Insert 5+2ε

Insert 5+3ε

Can not have 4 elements in a node
Must "split" this node
which includes insertion
into parent





Insert 0
- causes double split

Ideas

- Every node branches until reach leaves, all at same level
  - prevents very uneven ~~trees~~ (unbalanced) trees
- height is $\Theta(\lg n)$
- variability in branching factor leads to flexibility for fast upgrades
  - worst case involves split at every level of tree, and insertion of new root above current root $\Theta(h) = \Theta(\lg n)$

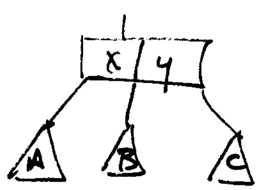
this is the only mechanism for gaining height

More formal and general class of trees to which 2-3-4 trees belong:

B-trees, with parameter $t \geq 2$ (case of $t=2 \Rightarrow$ 2-3-4 trees)

- Every non-leaf node has $\geq t$ and $\leq 2t$ children (except root, which has ≥ 2 children) (leaf has 0 children)
- Each non-leaf node stores one key in between every adjacent pair of children
- # keys = # children − 1 is $\geq t-1$ and $\leq 2t-1$
  - this key bound is enforced on leaves, as well

e.g.



all keys of $A \leq x \leq$ all keys of $B \leq y \leq$ all keys of C

Lemma: Height of B-tree $= \Theta(\log_t n) = \Theta(\lg n)$

Proof: ~~# leaves $\leq n$~~
~~branching factor $\geq t$, except at root~~
~~height $\leq \log_t n + 1$~~
~~subtree of root~~

- root has at least one key
- all other nodes have at least $t-1$ keys

- $n \geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1}$

$$= 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right) = 2t^h - 1$$

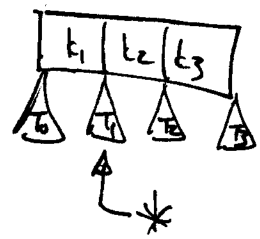
$$h \leq \log_t\left(\frac{n+1}{2}\right)$$

Search

- Visit nodes in root-to-leaf path
- At each node
  - examine all keys
  - if desired key found, done
  - else, find where desired key would fit among the ordered keys and follow that pointer] Could do binary search within node

(e.g.) Search(k)

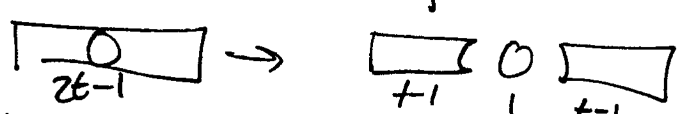
$k_1 < \underline{k} < k_2 < k_3$



Time: • $\Theta(t)$ to visit a node — binary search could make it $\lg t$
• height $\Theta(\log_t n)$

$$= \Theta(t \log_t n) \longrightarrow \Theta(\lg n) \text{ for } t = O(1)$$

Insert (This is where things start to get interesting)
- find leaf where new key belongs (using search)
- if leaf has fewer than $2t-1$ keys [has room] then add new key to leaf, keeping keys in sorted order $\rightarrow$ $\Theta(t)$ time ← may need to shift data
- else ~~Exxx~~ [leaf is full]
  - insert new key into left or right half ($\Rightarrow$ overflowing)
  - split node into left.median.right

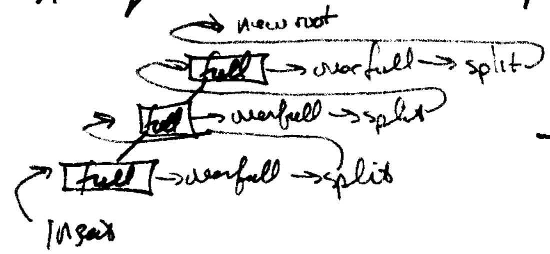


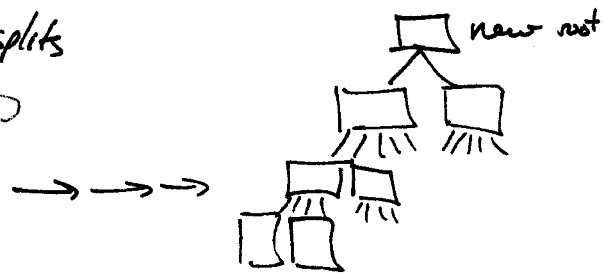
- promote ~~parent~~ median up to parent node
  - → if parent now overflowing, recursively split etc.
  - if root splits, create new root with 2 children & increment height.

Notes: ① this is the only way height grows: root becomes key from ... ② Book gives more efficient approach where splits occur on way down tree when traversing (when full) ← keeps constant

Time for Insert: $\Theta(t \log_t n)$, same as search

Analysis: at most h splits





Delete: Worst Case $\Theta(t \log_t n)$

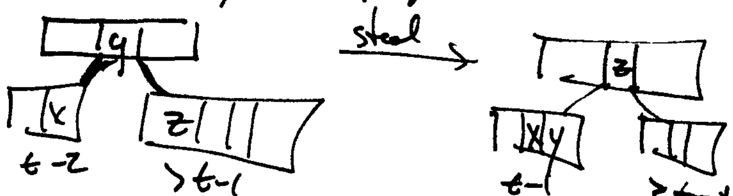
- if key is not in a leaf, replace it with successor (which is in leaf)

  min ~~of node~~ greater than node ~~that~~ if all keys distinct

- now just delete from leaf
  - remove key from leaf
  - if leaf still has $\geq t-1$ keys, then done
  - else [underflowing]   {2 tricks here}

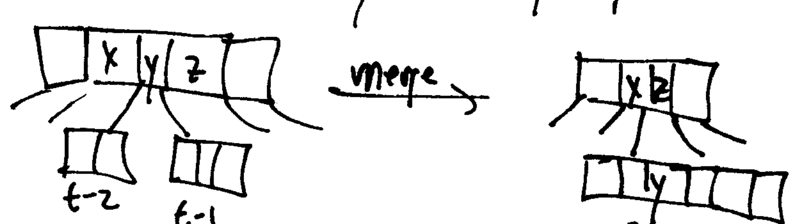
  ① try to steal from siblings
  - if an adjacent ~~key~~ sibling has $> t-1$ keys, ~~then~~ shift through parent

must maintain B-tree property



② If adjacent siblings have ~~only~~ $t-1$ keys (close to underflow), then merge with one of them ~~plus~~ and parent key.

  − essentially reversing a split



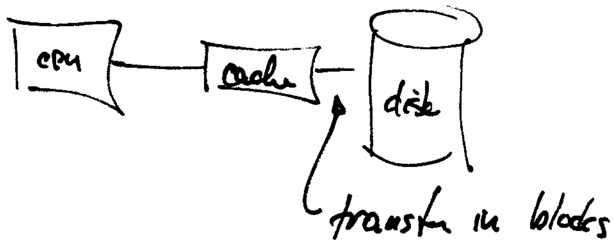
$2t-2 \geq t-1$ & $\leq 2t-1$

− this can lead to underflow in parent and require propagation up tree to removal of root node.

External Memory

B-trees used to exploit cache & disk, in practice

Model: Can read/write B elements stored together in one block transfer one node of B-tree



goal: minimize # block transfers

let $t = B$ in B-tree

Search/Insert/Delete are $\Theta(\log_B n)$ block transfers

(& this is optimal for this problem)

CLRS has more to say about this.