Finger search trees

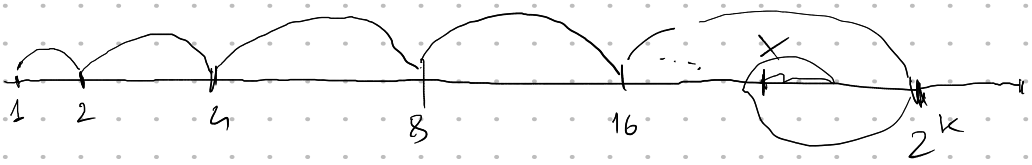
Motivating example:
Binary search on a line.

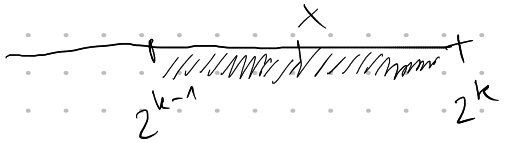
- 1000 sorted keys
- search (x)
- $\lceil \log_2 1000 \rceil$ comparisons
- best possible

(Guess a number between 1, 1000)

- What if x likely closer to left side?
- What if list is unbounded / of unknown size?

(Guess a number ≥ 1)





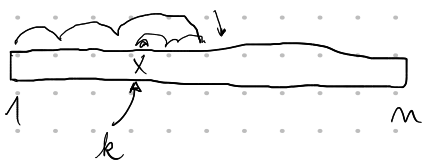
- doubling search
  1. query 1, 2, ---, 2^k, until we overshoot x
  2. binary search between 2^{k-1}, 2^k

comparisons

1. $k = \lceil \log_2 d \rceil$ $\quad d = \text{rank}(x)$ (how many smaller items in the list)
2. $2^{k-1} < d$ / $\log_2 d$ comparisons (binary search on 2^{k-1})

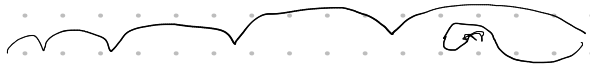
total: $2 \cdot \log_2 d$ comparisons

rank of searched item



$2 \cdot \log k \overset{?}{<} \log n$

Exercise: improve $2 \log d$

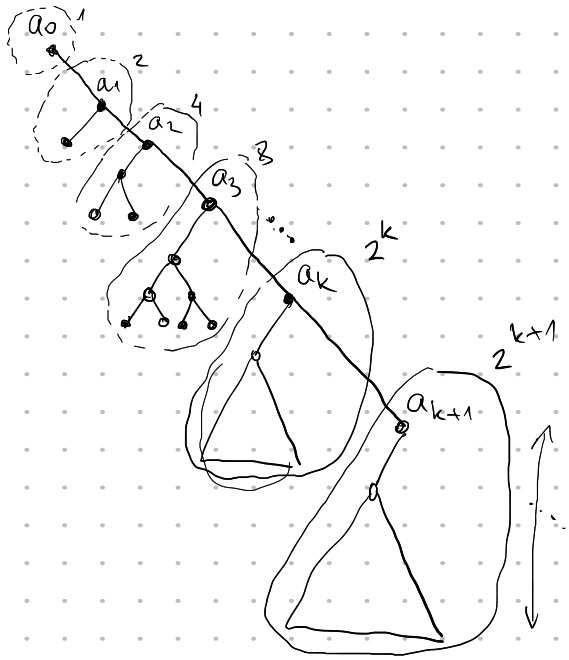


Summary (doubling search)

binary search in time $O(\log d)$ instead of $O(\log n)$

- d = # items in set smaller than target x (rank(x))
- n = size of sorted set

Doubling search viewed as Binary Search Tree



- right spine with complete trees hanging to the left (of increasing sizes)

- depth $\approx 2 \log_2 n$
- captures "doubling search"
- how to do insert/delete?
  - should maintain subtree-sizes as powers of 2
  - maintain approximately (doubling/halving)
  - when subtree too large/small, split/merge

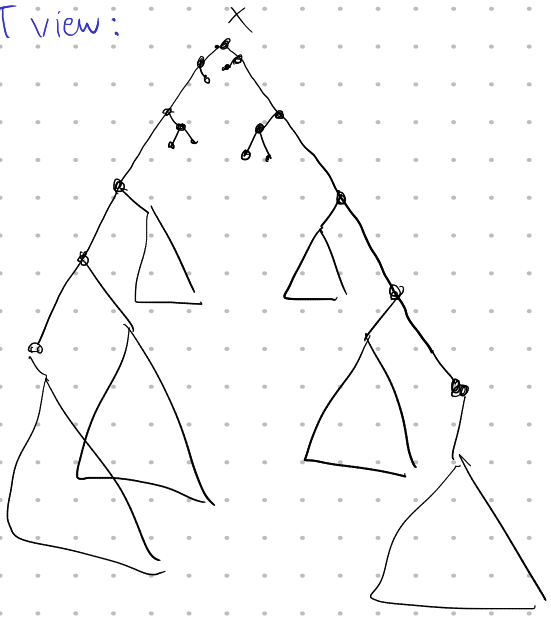
(doable, but tricky)

So far we assumed that search starts at minimum key.

Suppose we search around x.

→ Doubling search from x.

BST view:



what if x not fixed?

Goal: Given a pointer x (finger)

search(y)

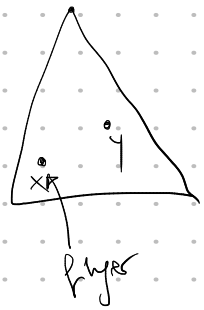
Should be efficient if "x is close to y"

Search should take $O(\log d)$

d: # elements in the tree that are in $[x, y]$

Dynamic?

$O(\log d)$ not worse than $O(\log n)$



"finger search"

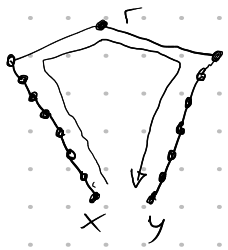
First idea

- go up from x to lca(x,y)
- go down from lca(x,y) to y

Two issues

- how to detect lca(x,y)
- even if we detect lca(x,y), the path may be too long

e.g.

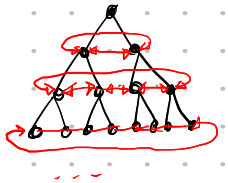


finger search from x to y

lca(x,y) = r

d = O(1) only element between x,y is r

General approach



- use level lists (construct circular doubly linked lists over nodes at each level)

- two extra pointers per node
- helps to navigate tree
- makes detection of $lca(x,y)$ possible

Issue: how to maintain level links under insert/delete/... ?
(doable, but tricky – see it later)

- search, insert, delete take $O(\log d)$ time

items between x, y

finger

searched / inserted / deleted item

- additionally: split, join

Applications

(Technically, should write $O(1+\log(1+d))$)

(1) An adaptive heap

- insert(x) $O(\log d)$ time } amortized
- extract-min $O(1)$ time

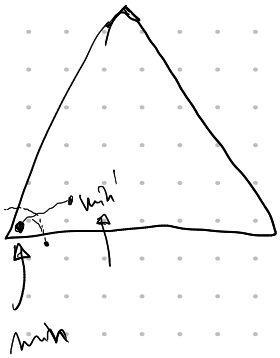
↳ d: rank of inserted element x
(# elements in heap with key smaller than x)

$d \leq n$

insert very efficient for elements close to min.

Implementation

- finger search tree, with pointer (finger) at the minimum.

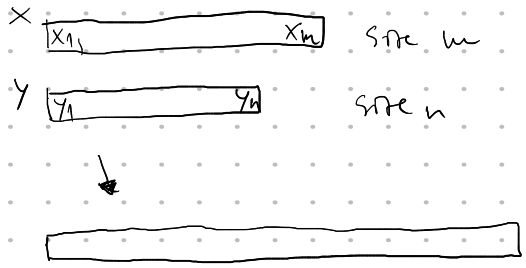


- extract-min
  - search for successor min' of min
    (finger search takes $O(1)$ time)
  - delete / return min $O(1)$
  - set min pointer to new minimum min'
- insert(x)
  - insert from finger min

$$O(\lg d)$$

rank(x) = # elements in tree between min and x

(2) Merging two sorted lists



eg. in Mergesort

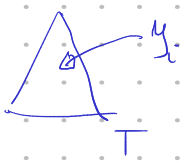
Standard "tape merging" takes $O(m+n)$ comparisons

What if $m \gg n$?

- assume X stored as BST T

(Obs. that we can build a BST from a sorted list X without comparisons in $O(m)$ time)

merge (X, Y):
  for $i = \overline{1, n}$
    insert y_i into T

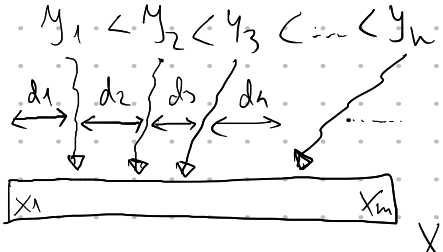


- cost of insert

$$O(\log(m+n)) = O(\log m)$$

- total cost of merge

$$O(n \cdot \log m)$$



X

(Again, assume X is given as finger search tree. If not, we can build it from a sorted list with no comparisons)

Improvement

Idea: insert y_i using finger search from y_{i-1}

cost of insert y_i is

$$\log(d(y_{i-1}, y_i))$$

# of elements of X between y_{i-1}, y_i

Let $d_i = |X \cap [y_{i-1}, y_i]|$

elements of X in $[y_{i-1}, y_i]$

$\sum_{i=1}^{n} d_i \leq m$ $\qquad$ d_1 := # elements of X in $(-\infty, y_1)$

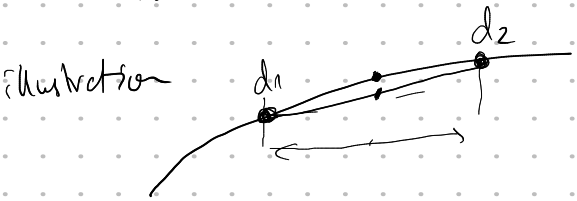
Total cost of merging:

$$\sum_{i=1}^{n} \log(d_i) \overset{*}{\leq} n \cdot \log\left(\frac{\sum d_i}{n}\right) \overset{*}{\leq} n \cdot \log \frac{m}{n}$$

* log function is concave

$$\frac{1}{n} \sum f(d_i) \leq f\left(\frac{\sum d_i}{n}\right) \quad \text{(Jensen-Ineq.)}$$

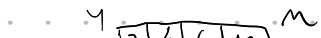
Illustration



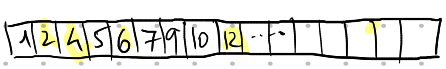
Using binary search we improve $O(n \cdot \log m)$ to $O(n \log \frac{m}{n})$.

$(m > n)$

Theorem Merging two sorted lists of sizes m, n requires $\Omega(n \cdot \log \frac{m}{n})$ comparisons.

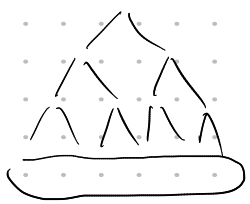


merge(X,Y)



($m+n$ positions in result, we just need to fix positions that come from Y)

How many possible "merging orders" are there: $\binom{m+n}{n}$



possible merging orders

$\binom{m+n}{n}$ leaves, depth of decision tree $\geq \log_2 \binom{m+n}{n}$

Any merging algorithm can be seen as decision tree, nodes are comparisons, leafs are possible merging orders

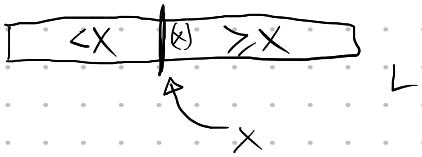
$\Rightarrow$ For some input, merging will take $\log_2 \binom{m+n}{n}$ comparisons

$$\log_2 \binom{m+n}{n} \overset{(*)}{\geq} \log_2 \left(\frac{m+n}{n}\right)^n = \Omega\left(n \cdot \log \frac{m}{n}\right)$$

(*) $\binom{a}{b} = \frac{a!}{b!(a-b)!} = \frac{(a-b+1)\cdot(a-b+2)\cdot \ldots \cdot a}{1 \cdot 2 \cdot \ldots \cdot b} = \frac{a-b+1}{1} \cdot \ldots \cdot \frac{a-1}{b-1}\frac{a}{b} \geq \left(\frac{a}{b}\right)^b$

(3) Collection of splittable lists.

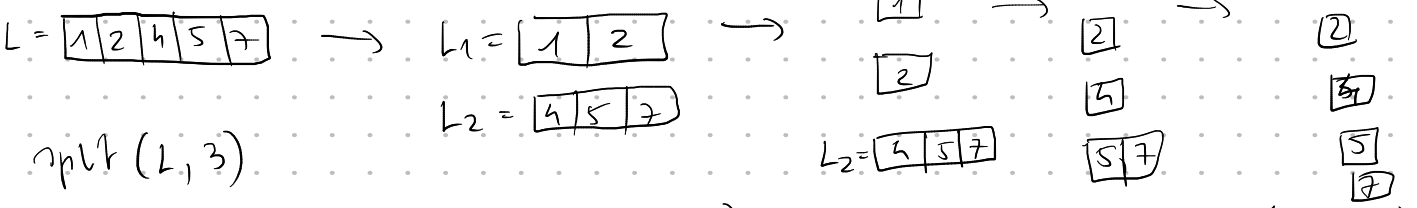
- maintain a set of sorted lists
- initially a single list of size n
- operation split (L, x)



find x in L, split L into $L_{<x}$, $L_{\geq x}$
- remove L from collection
- add two new lists into collection.

- continue until all lists are singletons (of size 1)

- assume all splits are proper



Question: total time

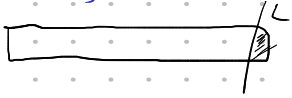
single split take $O(\log n)$

$O(\lg(n-1))$

naively: $n-1$ splits in total: $O(n \log n)$ time

Would like $O(n)$ time overall.

unlucky case:



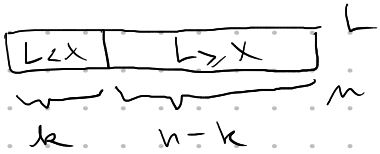
$T(n) = \log n + T(n-1)$

resolves to

$T(n) = \Theta(n \log n)$

look more carefully at a single split

split(L, x)

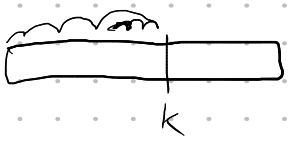


cost:

$O(\log(\min\{k, n-k\}))$

(instead of $O(\log n)$)

Doubling search from both ends:



$O(\log k)$

$O(\log(n-k))$

Best of both by searching simultaneously from both ends

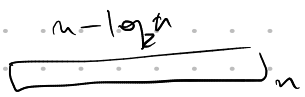
cost: $O(\log(\min\{k, n-k\}))$

Analysis

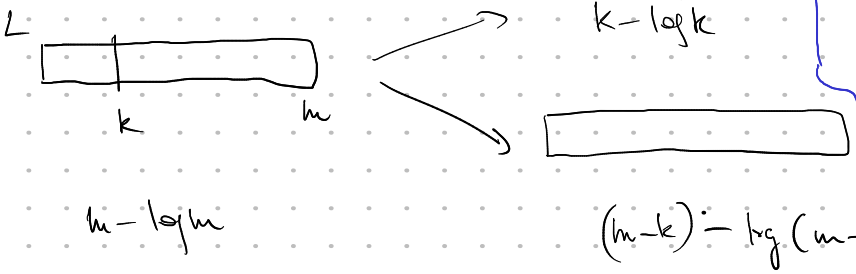
Amortized analysis.

Define potential of a list of size n is $n - \log_2 n$

Total potential Φ is sum of list potentials.



- $\Phi_0 = n - \log_2 n$
- $\Phi \geq 0$
- $\Delta\Phi$



Note: amortized analysis unusual because $\Phi_0 = n - \log_2 n > 0$.

total cost: $C = \sum_{i=1}^{n+1} \text{cost}_i$ → actual cost of i-th op.

amortized cost $A = \sum_{i=1}^{n+1} (\text{cost}_i + \Phi_{i+1} - \Phi_i)$

telescope:

$A = C + \Phi_{final} - \Phi_{initial}$

$C = A + \Phi_{initial} - \Phi_{final}$

A: $O(n)$; $\Phi_{initial}$: $n - \log_2 n$; Φ_{final}: ≤ 0

$\Rightarrow C \in O(n)$

(initial potential just gets added to total cost.)

- amortized cost = actual cost + $\Delta\Phi$

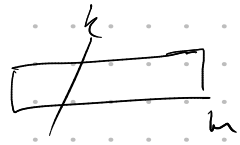
$$= \log(\min\{k, n-k\}) + k - \log k + (n-k) - \log(n-k) - n + \log n$$

$$= \log(\min\{k, n-k\}) + \log n - \log k - \log(n-k)$$

$$= *$$

$$\left[-\log k - \log(n-k) = -\log\left(\min\{k, n-k\}\right) - \log\left(\max\{k, n-k\}\right) \right]$$

$$\overset{*}{=} \log n - \underline{\log\left(\max\{k, n-k\}\right)} \overset{**}{=}$$



$$\max\{k, n-k\} \geq \frac{n}{2}$$

$$\overset{*}{\leq} \log_2 n - \log_2 \frac{n}{2} \leq 1$$

Can be made more interesting by adding operations insert, merge, etc.

amortized cost of split ≤ 1

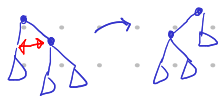
total cost of $n-1$ splits: $O(n)$.

Implementing finger search trees

idea:

balanced search tree
augmented with level-links.

(AVL/red-black not most suitable)



insert, delete, search } $O(\log d)$ time

"distance" between starting node x and target y

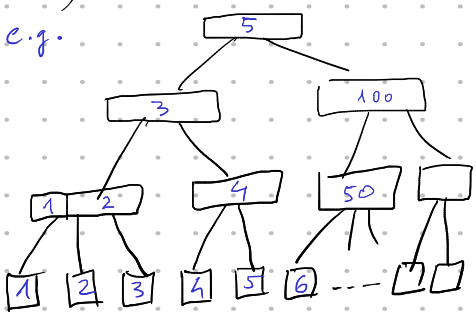
items between x, y

split / join

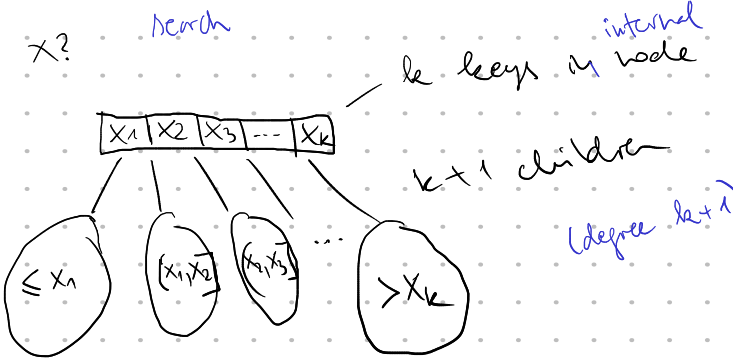
Alternative search tree: (a,b)-tree → used in Databases, file systems, etc. (B-tree)

Recap: What is an (a,b)-tree:

e.g.



Features:
- keys stored only in leaves
- every leaf has same depth
- keys sorted left-to-right
- internal nodes are used only for navigation
- internal nodes have degree $\geq a$, $\leq b$

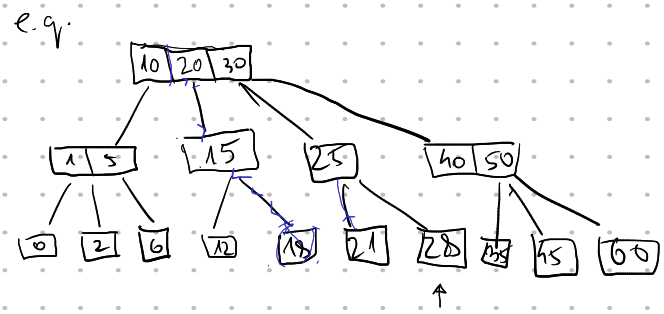


We only use: $a = 2$, $b = 4$ (In practical applications $a \approx \log n$, $b \approx 2 \log n$)

$(2,4)$ – trees

degrees of internal nodes are: 2, 3, 4

keys in each internal node: 1, 2, 3

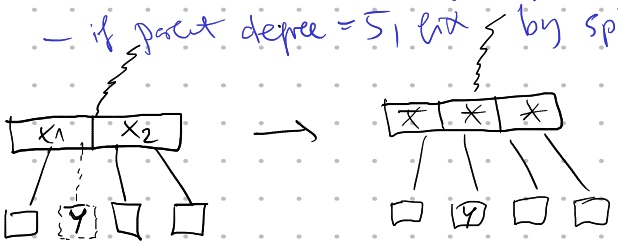


- search in $(2,4)$-tree takes $O(\log n)$ time

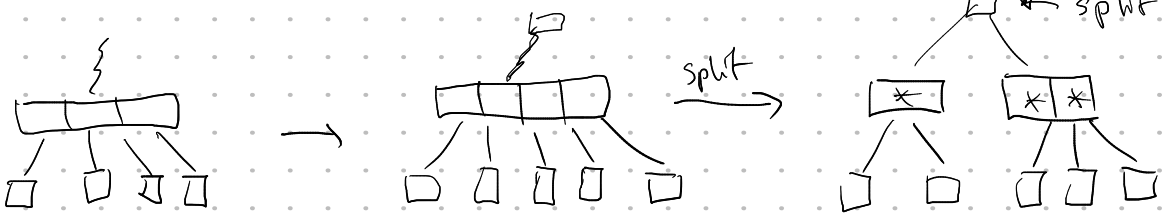
(like normal search in BST, keys in internal nodes are sorted; they determine which path we take)

degree $\geq 2 \Rightarrow$ depth $\leq \log_2 n + 1$

- insert: – find where new leaf should go
  – attach as extra leaf, update parent keys
  – if parent degree = 5, fix by splitting



- if parent node had degree 2 or 3 before, O.k.
- if parent already had degree 4, need to split.



Split nodes of degree 5, could propagate up towards root

- if root itself splits, tree height incr. by 1.

Cost of insert:

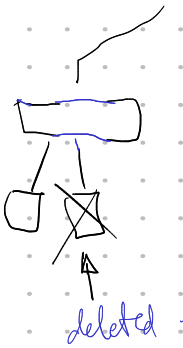
- cost of search $O(\log n)$
- cost of insert $O(1) + \#\text{splits} \in O(\log n)$

Amortized # of splits is $O(1)$

Exercise: deg 4 → deg 2 + deg 3

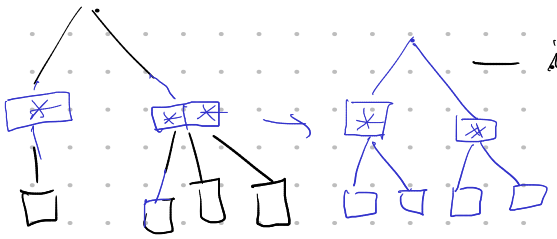
Delete

- find leaf to be deleted
- remove leaf
- fix parent if needed

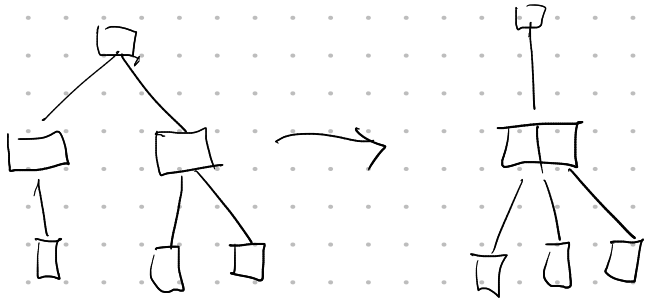


- if degree of parent was 3, 4 then O.k.
- if degree of parent was 2:
  - look at neighbor of parent, if any of them has degree ≥ 3, borrow a child
  - if neighbor has degree = 2, merge (update keys accordingly)

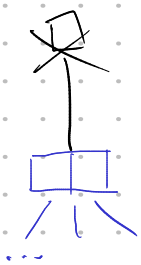
borrow:



merge:



- merging can propagate up to root
- if root itself ends up with single child, then remove root, tree height decr. by 1.



Cost of delete

- $O(\log n)$ for search
- for actual delete $O(1)$ + #merges ($\in O(\log n)$)

merges is amortized $O(1)$

Merge:

deg2 + deg2 → deg3

Exercise

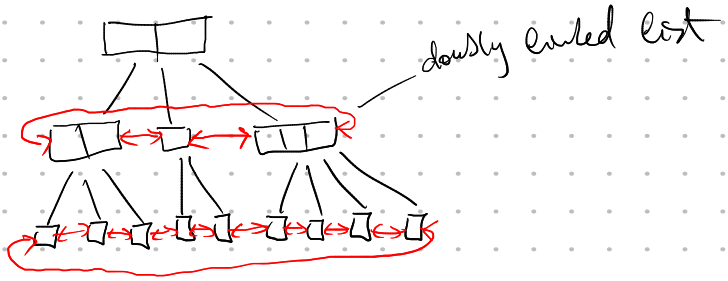
Summary:

search, insert, delete } $O(\log n)$ time

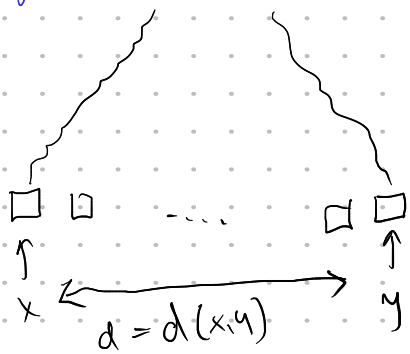
insert, delete } $O(1)$ amortized time if we have pointer to location and need not search

Finger search trees

Extend (2,4)-trees with level-links (between nodes at same level)



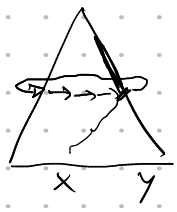
Finger search for y, starting from x



Search should cost $O(\log d)$

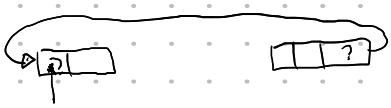
Suppose $x < y$

(other case symmetric)



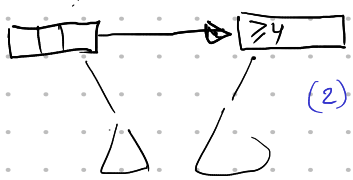
(1)

Go up from x until one of the following happens:



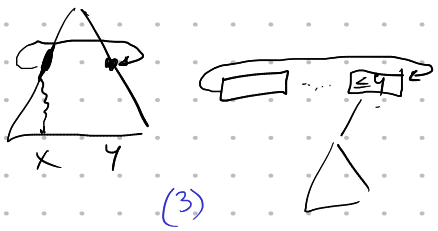
(1) we reach right path of tree

(2) right level-neighbor has a key $\geq y$



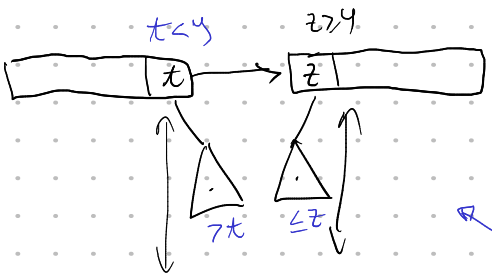
(2)

(3) we reach left path of tree, left neighbor (rightmost) has a key $\leq y$



(3)

In all 3 cases, search down for y in one or two subtrees

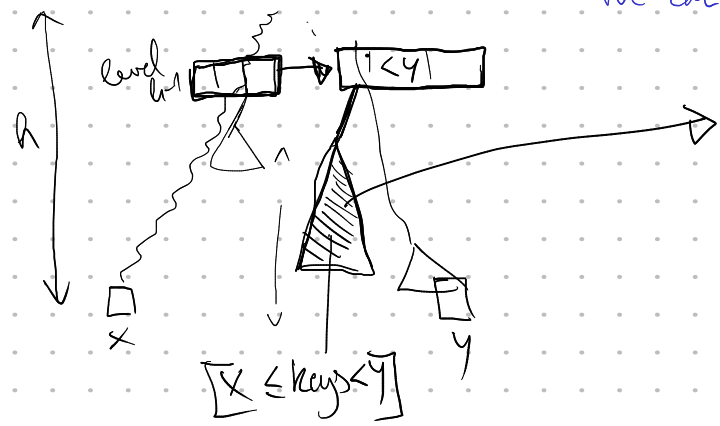


case when y could be in one of two subtrees

Analysis of search

look at step before last, when (1)–(3) did not hold.

We can find a subtree with keys in $[x,y]$



subtree has $\leq d$ leaves

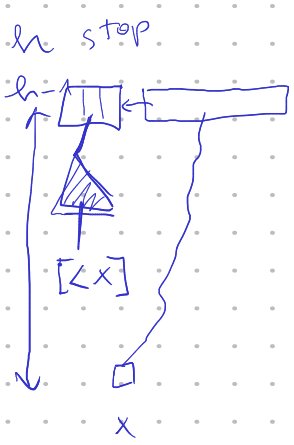
subtree has $\geq 2^{h-2}$ leaves

$2^{h-2} \leq d \quad / \log()$

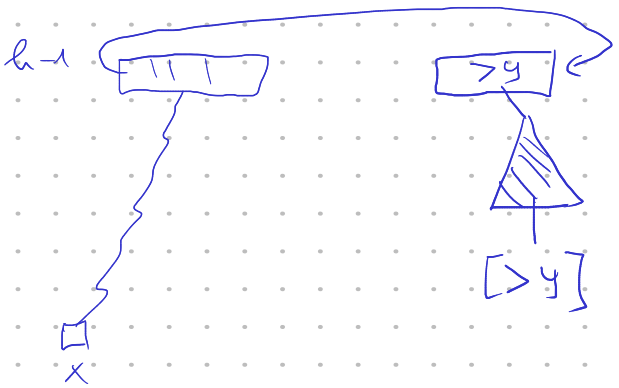
$h-2 \leq \log d \Rightarrow$ cost $= O(\log d)$

Finer analysis:
We can also find a subtree with keys outside of $[x,y]$

case 1.



case 2.



In both cases we have a subtree with root at level $h-1$, with keys outside $[x,y]$

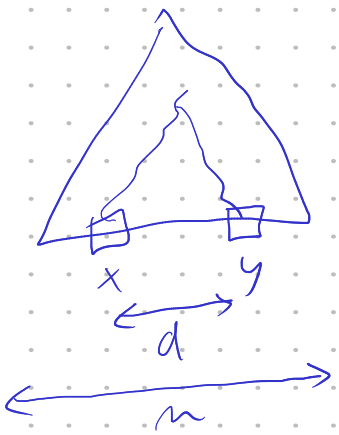
subtree has $\geq 2^{h-2}$ leaves

subtree has $\leq n-d$ leaves

$2^{h-2} \leq n-d \Rightarrow h = O(\log(n-d))$

$\Rightarrow$

Cost of search in finger search tree $= O(\log \min\{d, n-d\})$



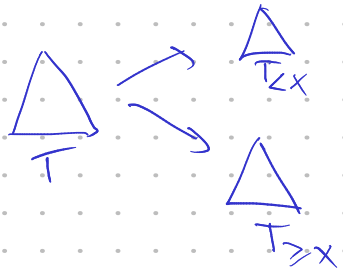
Insert / Delete

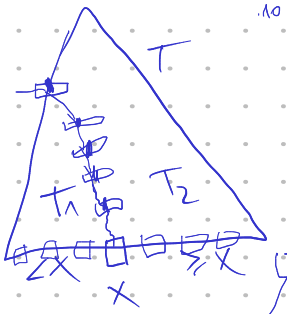
must maintain level-links under merge/split (easy)

Split / Join in a finger search tree

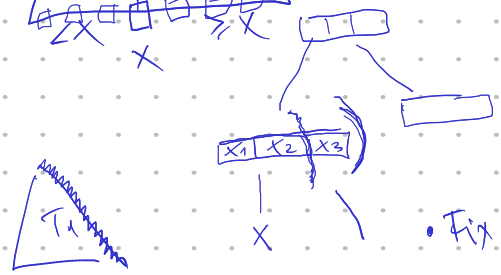
Split (T, x)

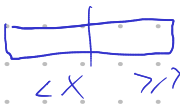
FST ↙ ↘ key

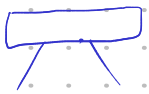


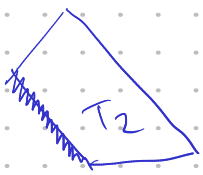


- Go up from x, until reaching leftmost/rightmost path
  Split all internal nodes along path









- Fix T_1, T_2
- restore level-links
- fix all degree-1 nodes by merging
  (just like for delete)

Cost: $O(\log n)$

Join

join (T_1, T_2)

- Assume keys in T_1 are $<$ keys in T_2
- Combine T_1, T_2 into a single tree

suppose $h_1 > h_2$

- Walk up the right path of T_1 up to level h_2
- Link root of T_2 as new rightmost child
- Fix level-links

- if degree of internal node became 5, fix by splitting
  (just like for insert)

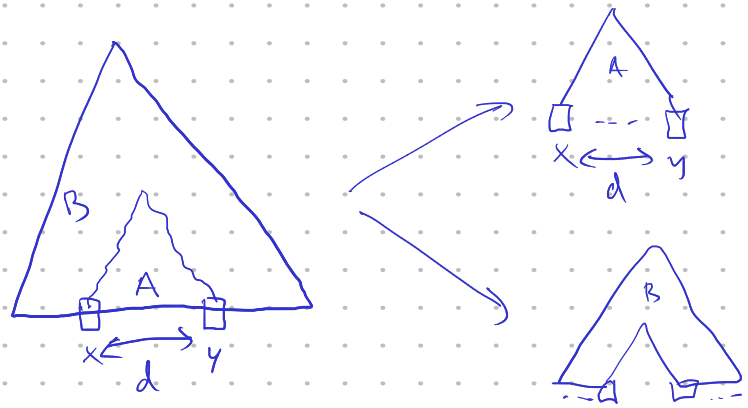
Cost = $O(\log n)$

$h_1 < h_2$

cost $O(\log n)$

Fiker split (2-pivot split)

Split (T, x, y)



Split T into parts

$[x, y]$

outside $[x, y]$

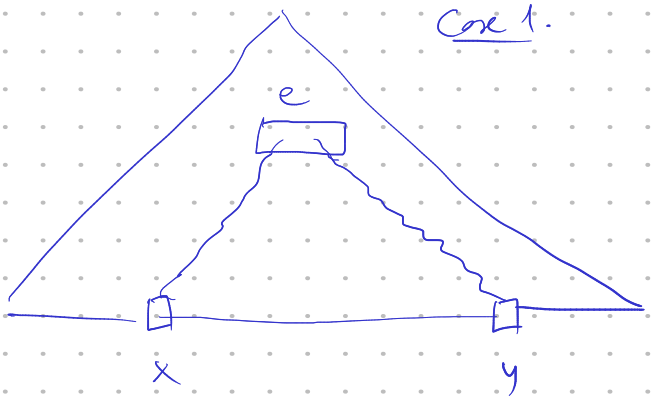
return two layer trees

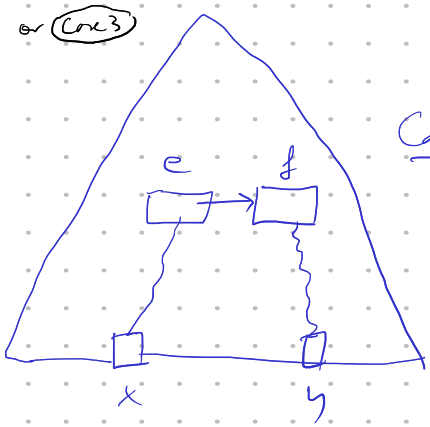
cost: $O(\log \min\{d, n-d\})$

Idea: walk up at the same time from both x and y, until we reach the same internal node or two neighboring nodes

Case 1 or Case 2 or Case 3

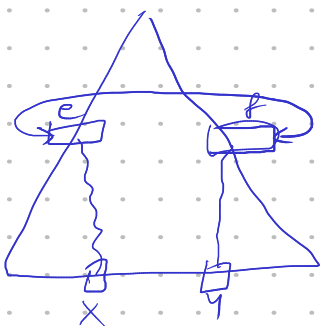
Case 1.



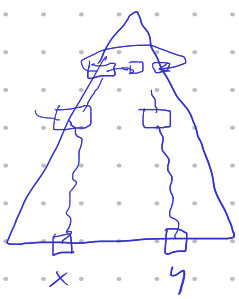


Case 2.

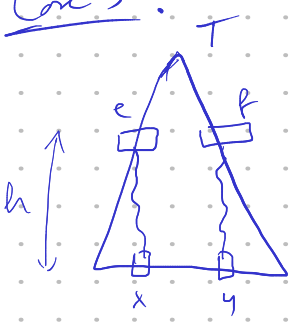
e or f may be or maybe



Case 3.

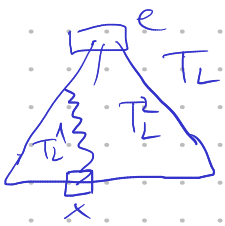


Case 3.



① Detach subtree rooted at e (call it T_L)

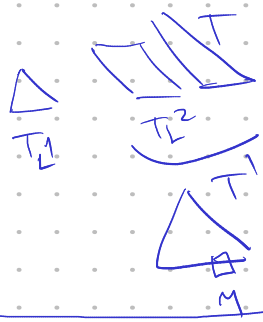
- fix remainig T (...)



② Split T_L at x

→ T_L^1 ($< x$)

→ T_L^2 ($\geq x$)

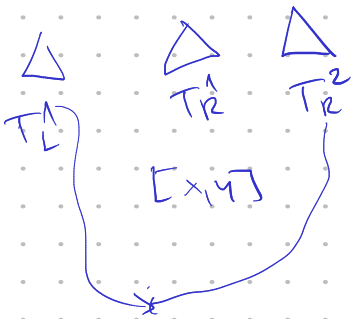


③ Join T_L^2 and remainig T → T'

④ Split T' at y

→ T_R^1 ($\leq y$)

→ T_R^2 ($> y$)



⑤ return { T_R^1 (within [x,y])

join(T_L^1, T_R^2) (outside [x,y])

cost $O(h) = O(\log \min\{d, n-d\})$

analysis similar to search.

(other cases omitted)