2 Fundamental Data Structures

04 February 2020

Sebastian Wild

version 2020-02-04 14:50

ADT = abstract data type

- list of supported operations
  - what should happen
  - <u>not</u> how to do this

    <u>not</u> how data is stored

VS

data structures

- how data is stored in memory
- algorithms to work on data

ex: stack

pop() → removes topmost element

push(v) → adds v to top of stack

pop pop push(1)

~~1~~

~~2~~ 1

3

4

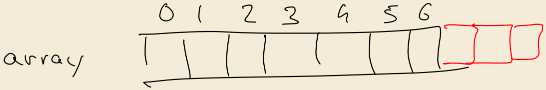
5

Outline

2 Fundamental Data Structures

2.1 Stacks & Queues

2.2 Resizable Arrays



- arrays have fixed size

  ↳ if we need more space, allocate new array & copy old data

- doubling arrays: when array is full double its size

- if array becomes too empty (deletions)

  ↳ if $\leq \frac{1}{4}$ full ⇝ halve size

→ space $\Theta(n)$ $n = \#$ elements stored

Java Generics

Stack<String>

Stack<Integers>

implement ONCE stack with type parameter

use stack with <u>many</u> different types

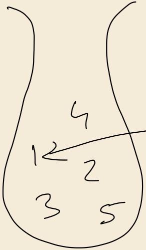
Iterators

ADT abstracts linear scan over collection of items

- hasNext()
- next move ahead & return element

2.3 Priority Queues

ADT

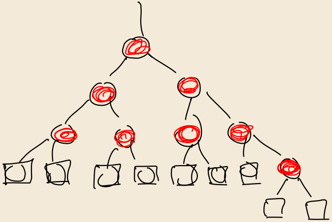


- isEmpty
- delMin
- insert
- decreaseKey
  changeKey

Heaps

- binary tree

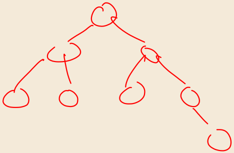
extended binary trees



every node has 0 or 2 children

⟷

binary trees

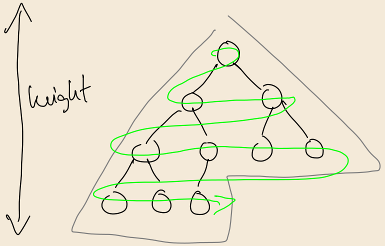


nodes have left / right children (both can be missing)

- complete binary tree + flush-left last level – lowest level as far to the left as possible

"heap-shaped trees"

Why heap-shaped trees?



n nodes $\rightarrow$ 1 heap-shape

$\Rightarrow$ easy

can use array : store nodes in level-order starting at 1

$\rightsquigarrow$ find parent of node k at $\lfloor \frac{k}{2} \rfloor$

left child at $2k$

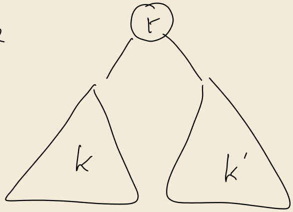
right " $2k+1$

- minimal height among binary trees with n nodes

heap-order

entire subtree version

every node



$k \geq r$

$k' \geq r$

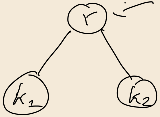
all keys in subtrees are $\leq$ key in root

$\Rightarrow$ root stores minimum

$\Updownarrow$

children-version

every node

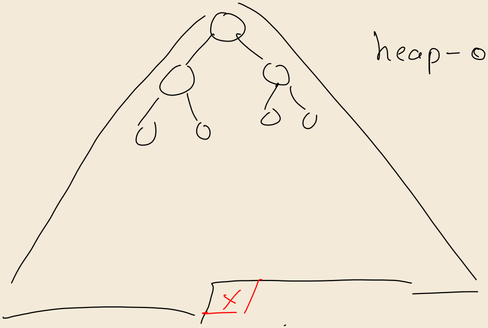


$k_1 \geq r$

$k_2 \geq r$

Insertion

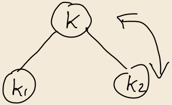
insert new key x



heap-ordered

1. store x in lowest level, next free position

2. repair heap-order

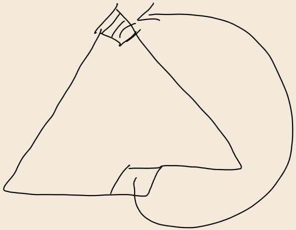


$\boxed{k_2 < k}$ violation

→ swap them!

repeat up the tree

delete Min

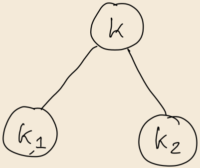


1. delete root (and return its key)
2. move "last" element to root
   - rightmost on lowest level
3. repair heap-order

$k > k_1$ and/or $k > k_2$

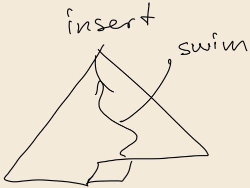
(a) find $\min\{k_1, k_2\}$

(b) swap k with smaller child



continue along the changed child link

Analysis



worst case in both cases: follow one path

$\Rightarrow$ cost $=$ # levels $=$ height of tree

$\sim \lg n$

PQ = 2 ADT

MinPQ | MaxPQ

insert

delMin | delMax

2 separate ADTs

min-oriented binary heaps

max-oriented binary heaps

2.4 Binary Search Trees

ADT : Symbol table

aka dictionaries

associative arrays   A['hello'] = 17

maps

partial

≈ mathematical functions

$\boxed{\text{dynamic}}$ — change function over time

---

primitive implementations

① unordered list   sequential search   easy add

= check every [|]   hard find

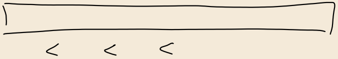
$O(n)$ $n = \#$ keys

$[k_1 \mid v_1] \rightarrow [k_2 \mid v_2] \rightarrow [\ \mid\ ]$

(2) sorted array binary search

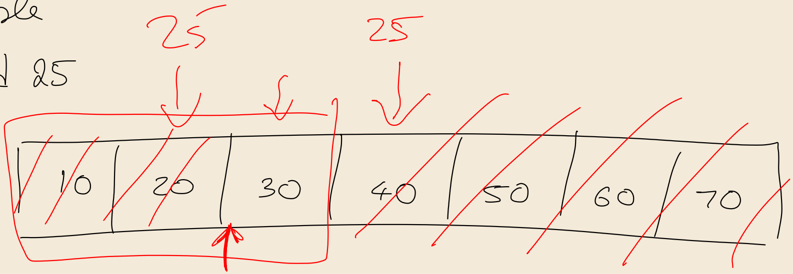
good find

hard to change



Example

find 25



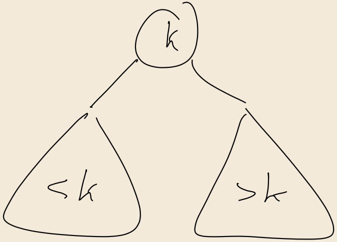
$\sim \lg n$

BSTs $\hat{=}$ dynamic sorted "array"

- binary tree = nodes have left / right child
  both can be empty

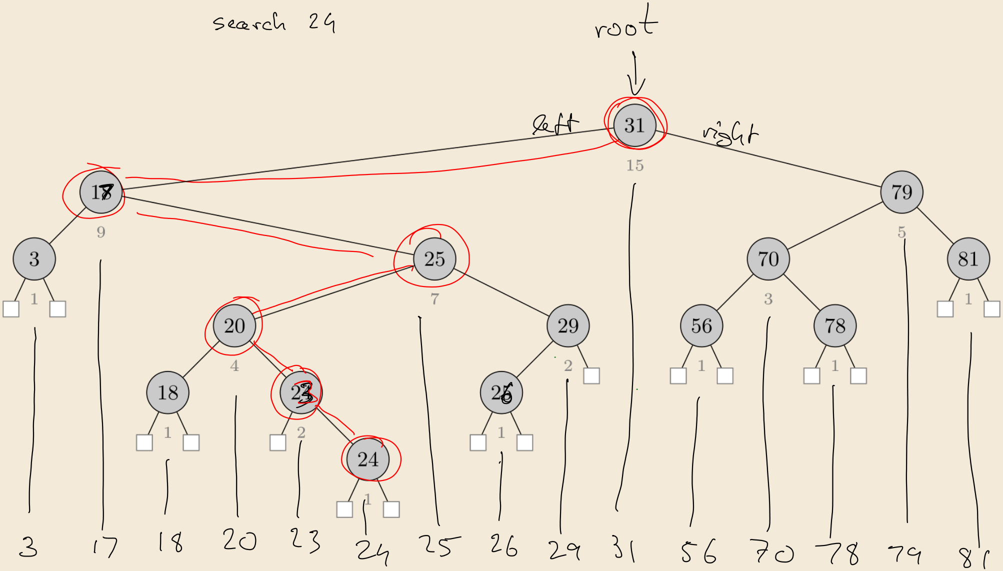
+

- search-tree property
  (symmetric order)



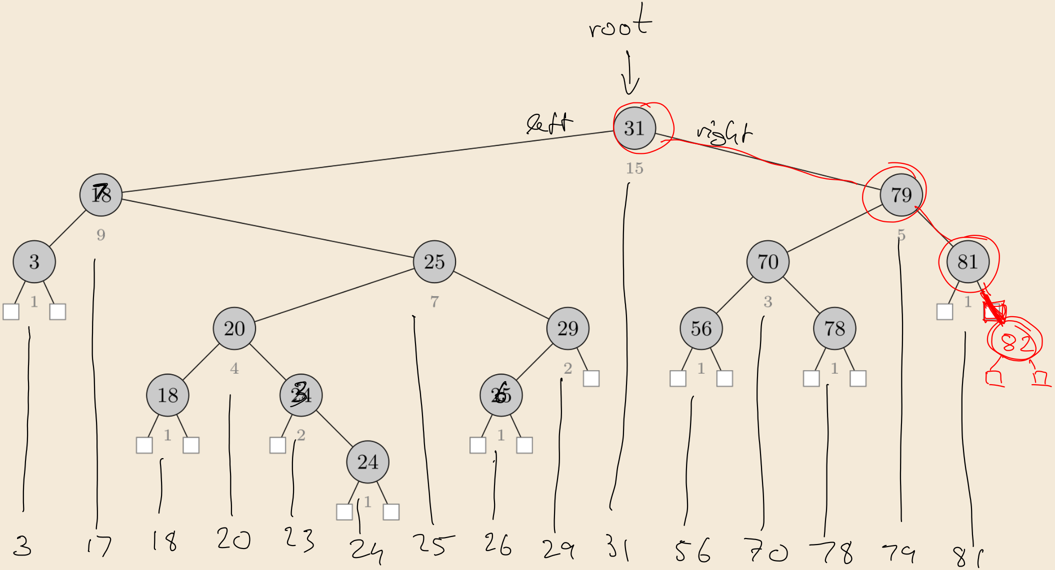
in Java: Node { left, right }
BST { root }

search 24



root
left
right
31
15
17
9
79
5
3
1
25
7
70
3
81
1
20
4
29
2
56
1
78
1
18
1
23
2
26
1
24
1
3 17 18 20 23 24 25 26 29 31 56 70 78 79 81

insert 82



root
left
right
31
15
18
9
79
5
3
1
25
7
70
3
81
1
82
20
4
29
2
56
1
78
1
18
1
24
2
24
1
25
1
3 17 18 20 23 24 25 26 29 31 56 70 78 79 81

delete 3 easy leaf

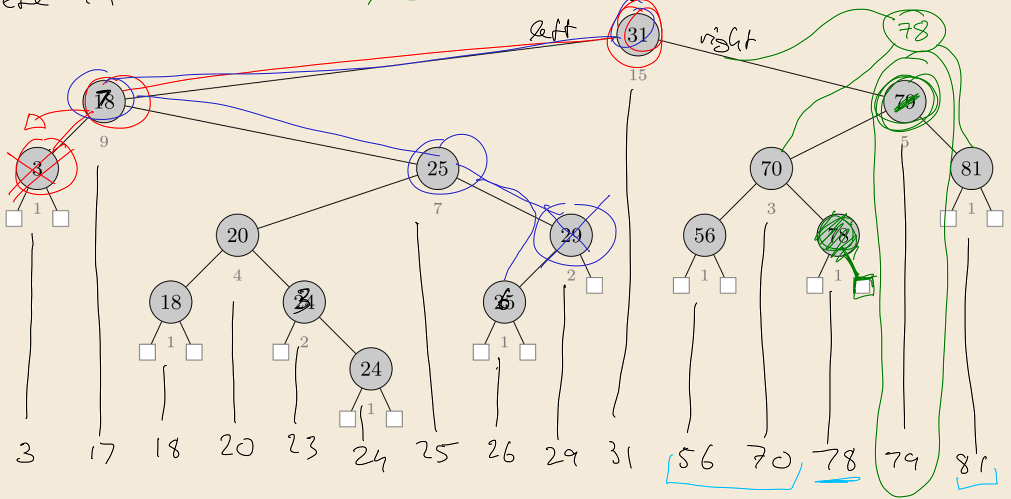
delete 29 easy unary node

delete 79 hard binary

find inorder predecessor (left, right, right, ...)

swap with 79

delete 78

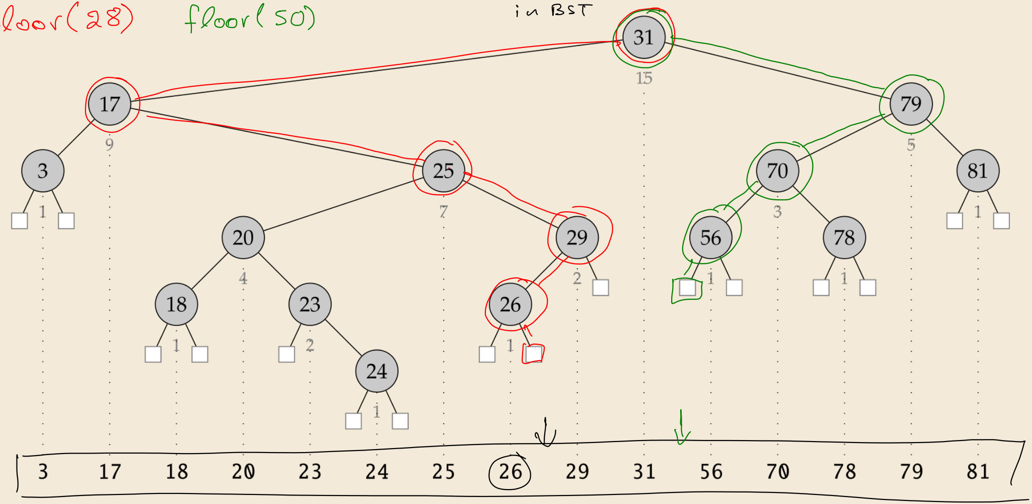


min / max ✓

floor(x) / ceiling(x)

$$\text{floor}(x) = \begin{cases} \text{value of } x & x \text{ in BST} \\ \text{value of largest } y < x \text{ in BST} & x \text{ not in BST} \end{cases}$$

floor(28) floor(50)



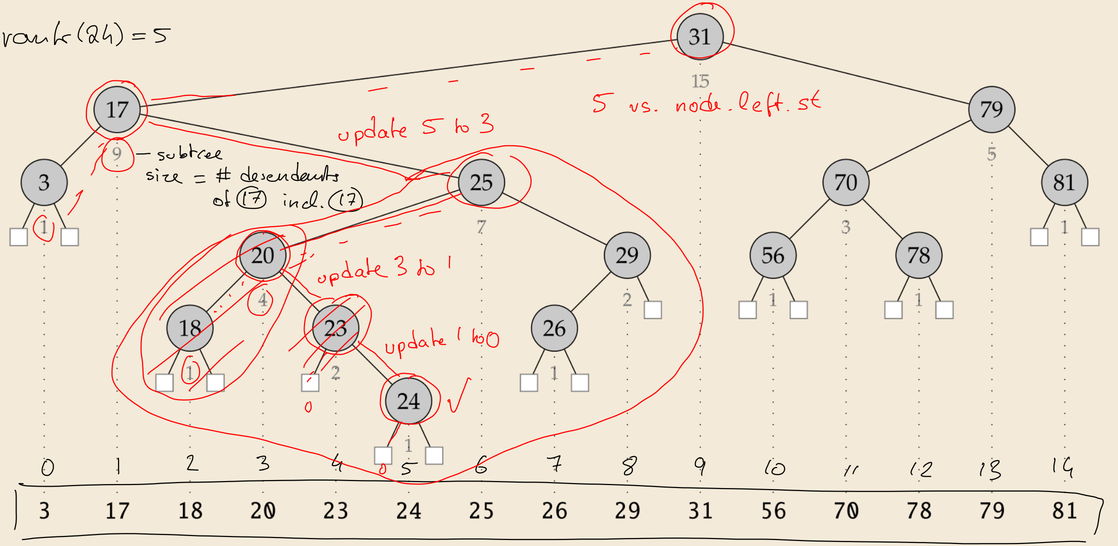
rank select(5) = 24

$\hat{=}$ A[5]

rank(x) = # elements < x

Idea: Maintain subtree size in each node.

rank(24) = 5

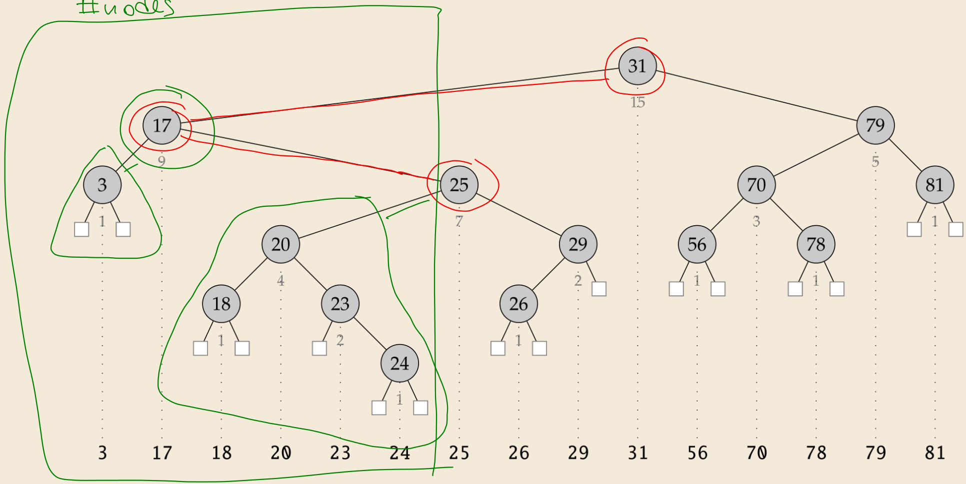


rank(25)

rank: walks up search path: add up sizes of left subtrees

=

#nodes

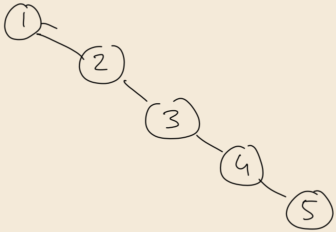


<u>Cost:</u>

worst-case key

search	O(height(tree))	height = length of longest root-to-leaf path
insert	"	
delete	"	
min/max	"	(length of the left/right spine)
floor/ceiling	"	
rank	"	
select	"	

↳ better had low height trees



↯ inserting keys in sorted order $\Rightarrow$ height $= n$

BUT (1) inserting keys in <u>random</u> order

height $= O(\log n)$ in expectation

& "with high probability"

(2) we can enforce $O(\log n)$ height by balancing rules

BBSTs

- AVL trees
- red-black trees
- 2-3 trees

height $= O(\log n)$

insert, delete

take $O(\log n)$ time

magic black box for COMP 526

implemented in most programming libraries

TreeMap