2 Fundamental Data Structures

10 February 2022

Sebastian Wild

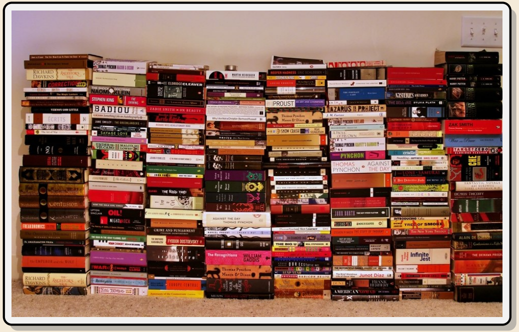
COMP526 (Spring 2022)
University of Liverpool

version `2022-01-31 22:19` *H*

Learning Outcomes

1. Understand and demonstrate the difference between *abstract data type (ADT)* and its *implementation*
2. Be able to define the ADTs *stack*, *queue*, *priority queue* and *dictionary / symbol table*
3. Understand *array*-based implementations of stack and queue
4. Understand *linked lists* and the corresponding implementations of stack and queue
5. Know *binary heaps* and their performance characteristics
6. Understand *binary search trees* and their performance characteristics

Unit 2: *Fundamental Data Structures*



Outline

2 Fundamental Data Structures

2.1 Stacks & Queues

Abstract Data Types

abstract data type (ADT)

- list of supported operations
- **what** should happen
- **not:** how to do it
- **not:** how to store data

$\approx$ Java `interface`
(with Javadoc comments)

VS.

data structures

- specify exactly
  how data is represented
- algorithms for operations
- has concrete costs
  (space and running time)

$\approx$ Java `class`
(non abstract)

Why separate?

- Can swap out implementations $\rightsquigarrow$ "drop-in replacements")

$\rightsquigarrow$ reusable code!

- (Often) better abstractions
- Prove generic lower bounds ($\rightsquigarrow$ Unit 3)

Stacks

Stack ADT

- `top()`
  Return the topmost item on the stack
  Does not modify the stack.
- `push(`x`)`
  Add x onto the top of the stack.
- `pop()`
  Remove the topmost item from the stack
  (and return it).
- `isEmpty()`
  Returns `true` iff stack is empty.
- `create()`
  Create and return an new empty stack.

Linked-list implementation for Stack

Invariants:

- maintain `top` pointer to topmost element
- each element points to the element below it (or `null` if bottommost)

Linked stacks:

- require $\Theta(n)$ space when n elements on stack
- All operations take $O(1)$ time

Array-based implementation for Stack

Can we avoid extra space for pointers?
$\rightsquigarrow$ array-based implementation

Invariants:

- maintain array `S` of elements, from bottommost to topmost
- maintain index `top` of position of topmost element in `S`.

What to do if stack is full upon `pop`?

Array stacks:

- require *fixed capacity* C (known at creation time)!
- require $\Theta(C)$ space for a capacity of C elements
- all operations take $O(1)$ time

2.2 Resizable Arrays

Digression – Arrays as ADT

Arrays can also be seen as an ADT! ... but are commonly seen as specific data structure

Array operations:

- `create(n)` *Java*: `A = new int[n];`
  Create a new array with n cells, with positions $0, 1, \ldots, n-1$
- `get(i)` *Java*: `A[i]`
  Return the content of cell i
- `set(i,x)` *Java*: `A[i] = x;`
  Set the content of cell i to x.

$\rightsquigarrow$ Arrays have fixed size (supplied at creation).

Usually directly implemented by compiler + operating system / virtual machine.

⚠ **Difference to others ADTs:** ***Implementation usually fixed***
to “a contiguous chunk of memory”.

Doubling trick

Can we have unbounded stacks based on arrays? Yes!

Invariants:

- maintain array `S` of elements, from bottommost to topmost
- maintain index `top` of position of topmost element in `S`
- maintain capacity C = `S.length` so that $\frac{1}{4}C \le n \le C$

$\rightsquigarrow$ can always push more elements!

How to maintain the last invariant?

- before `push`
  If $n = C$, allocate new array of size $2n$, copy all elements.
- after `pop`
  If $n < \frac{1}{4}C$, allocate new array of size $2n$, copy all elements.

$\rightsquigarrow$ ***"Resizing Arrays"***

← an implementation technique, not an ADT!

Amortized Analysis

- Any individual operation `push` / `pop` can be expensive!
  $\Theta(n)$ time to copy all elements to new array.
- **But:** An one expensive operation of cost T means $\Omega(T)$ next operations are cheap!

distance to boundary ↙

Formally: consider "credits/potential" $\Phi = \min\{n - \frac{1}{4}C, C - n\} \in [0, 0.6n]$

- amortized cost of an operation = actual cost (array accesses) $- 4 \cdot$ change in Φ
  - cheap `push`/`pop`: actual cost 1 array access, consumes ≤ 1 credits $\rightsquigarrow$ amortized cost ≤ 5
  - copying `push`: actual cost $2n + 1$ array accesses, creates $\frac{1}{2}n + 1$ credits $\rightsquigarrow$ amortized cost ≤ 5
  - copying `pop`: actual cost $2n + 1$ array accesses, creates $\frac{1}{2}n - 1$ credits $\rightsquigarrow$ amortized cost 5

$\rightsquigarrow$ **sequence** of m operations: total actual cost $\le$ total amortized cost + final credits
here: $\le 5m + 4 \cdot 0.6n = \Theta(m + n)$

Queues

Operations:

- `enqueue(`x`)`
  Add x at the end of the queue.

- `dequeue()`
  Remove item at the front of the queue and return it.

Implementations similar to stacks.

Bags

What do Stack and Queue have in common?

They are special cases of a ***Bag***!

Operations:

- `insert(`x`)`
  Add x to the items in the bag.
- `delAny()`
  Remove any one item from the bag and return it.
  (Not specified which; any choice is fine.)
- roughly similar to Java's `Collection`

Sometimes it is useful to state that order is irrelevant $\rightsquigarrow$ Bag
Implementation of Bag usually just a Stack or a Queue

2.3 Priority Queues

Priority Queue ADT – min-oriented version

Now: elements in the bag have different *priorities*.

(Max-oriented) Priority Queue (MaxPQ):

- `construct(`A`)`
  Construct from from elements in array A.
- `insert(`x`,`p`)`
  Insert item x with priority p into PQ.
- `max()`
  Return item with largest priority. (Does not modify the PQ.)
- `delMax()`
  Remove the item with largest priority and return it.
- `changeKey(`x`,`p'`)`
  Update x's priority to p'.
  Sometimes restricted to *increasing* priority.
- `isEmpty()`

Fundamental building block in many applications.

PQ implementations

Elementary implementations

- unordered list $\rightsquigarrow$ $\Theta(1)$ `insert`, but $\Theta(n)$ `delMax`
- sorted list $\rightsquigarrow$ $\Theta(1)$ `delMax`, but $\Theta(n)$ `insert`

Can we get something between these extremes? Like a "slightly sorted" list?

Yes! *Binary heaps.*

Array view

Heap = array A with
$\forall i \in [n] : A[\lfloor i/2 \rfloor] \geq A[i]$

$\equiv$

store nodes in level order in $A[1..n]$

Tree view

Heap = tree that is
(i) a complete binary tree
(ii) heap ordered

all but last level full
last level flush left

parent $\geq$ children

Binary heap example

Why heap-shaped trees?

Why complete binary tree shape?

- only one possible tree shape $\rightsquigarrow$ keep it simple!
- complete binary trees have minimal height among all binary trees
- simple formulas for moving from a node to parent or children:
  For a node at index k in A
  - parent at $\lfloor k/2 \rfloor$
  - left child at $2k$
  - right child at $2k+1$

Why heap ordered?

- Maximum must be at root! $\rightsquigarrow$ `max()` is trivial!
- But: Sorted only along paths of the tree; leaves lots of leeway for fast inserts

how? ... stay tuned

Insert

Delete Max

Heap construction

Analysis

Height of binary heaps:

- *height* of a tree: # edges on longest root-to-leaf path
- *depth/level* of a node: # edges from root $\rightsquigarrow$ root has depth 0
- How many nodes on first k *full* levels? $\displaystyle\sum_{\ell=0}^{k} 2^{\ell} = 2^{k+1} - 1$

$\rightsquigarrow$ Height of binary heap: $h = \min k \text{ s.t. } 2^{k+1} - 1 \ge n = \lfloor \lg(n) \rfloor$

Analysis:

- `insert`: new element "swims" up $\rightsquigarrow$ $\le h$ steps (h cmps)
- `delMax`: last element "sinks" down $\rightsquigarrow$ $\le h$ steps ($2h$ cmps)
- `construct` from n elements:

  cost = cost of letting *each node* in heap sink!

$$\begin{aligned}
\text{cost} &= \text{cost of letting each node in heap sink!} \\
&\le 1 \cdot h + 2 \cdot (h-1) + 4 \cdot (h-2) + \cdots + 2^{\ell} \cdot (h-\ell) + \cdots + 2^{h-1} \cdot 1 + 2^{h} \cdot 0 \\
&= \sum_{\ell=0}^{h} 2^{\ell}(h-\ell) = \sum_{i=0}^{h} \frac{2^h}{2^i} i = 2^h \sum_{i=0}^{h} \frac{i}{2^i} \le 2 \cdot 2^h \le 4n
\end{aligned}$$

Binary heap summary

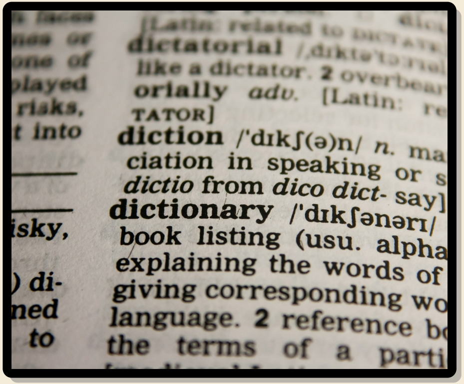
Operation	Running Time
`construct(`$A[1..n]$`)`	$O(n)$
`max()`	$O(1)$
`insert(`x`,`p`)`	$O(\log n)$
`delMax()`	$O(\log n)$
`changeKey(`x`,`p'`)`	$O(\log n)$
`isEmpty()`	$O(1)$
`size()`	$O(1)$

2.4 Binary Search Trees

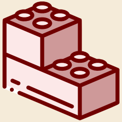
Symbol table ADT

Java: java.util.Map<K,V>

Symbol table / Dictionary / Map / Associative array / key-value store:



- put(k,v) *Python* dict: d[k] = v
  Put key-value pair (k, v) into table
- get(k) *Python* dict: d[k]
  Return value associated with key k
- delete(k)
  Remove key k (any associated value) form table
- contains(k)
  Returns whether the table has a value for key k
- isEmpty(), size()
- create()



Most fundamental building block in computer science.

(Every programming library has a symbol table implementation.)

Symbol tables vs mathematical functions

- similar interface
- but: mathematical functions are *static* (never change their mapping)
  (Different mapping is a *different* function)
- symbol table = *dynamic* mapping
  Function may change over time

Elementary implementations

Unordered (linked) list:

- 👍 Fast `put`
- 👎 $\Theta(n)$ time for `get`

$\rightsquigarrow$ Too slow to be useful

Sorted *linked* list:

- 👎 $\Theta(n)$ time for `put`
- 👎 $\Theta(n)$ time for `get`

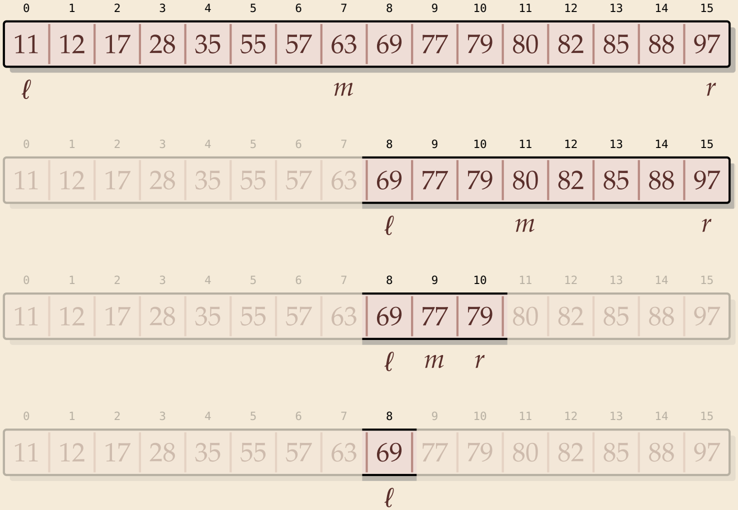
$\rightsquigarrow$ Too slow to be useful

$\rightsquigarrow$ *Sorted order does not help us at all?!*

Binary search

It does help . . . if we have a sorted ***array****!*

Example: search for 69



Binary search:

- halve remaining list in each step (± 1)

$\leadsto$ $\le \lfloor \lg n \rfloor + 1$ cmps in the worst case

⚠ needs random access

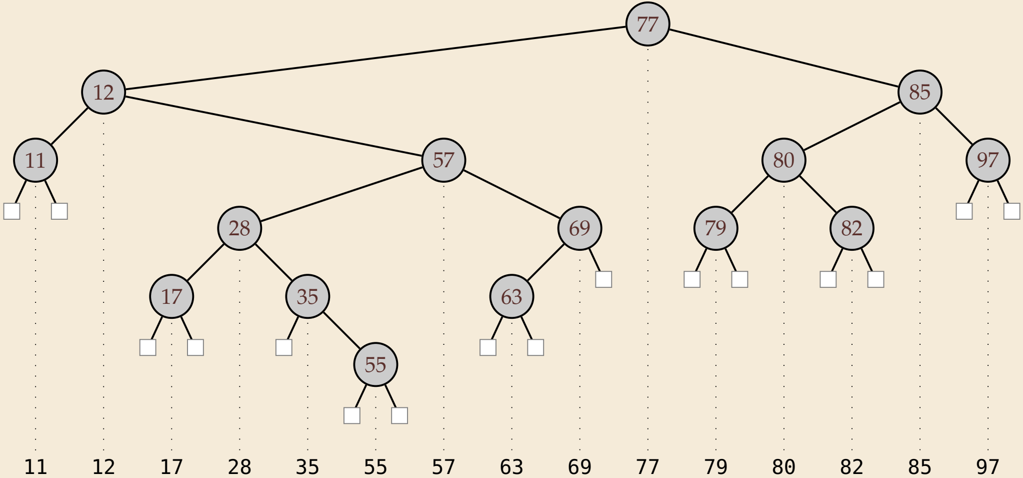
Binary search trees

Binary search trees (BSTs) $\approx$ dynamic sorted array

- binary tree
  - Each node has left and right child
  - Either can be empty (`null`)
- Keys satisfy ***search-tree property***

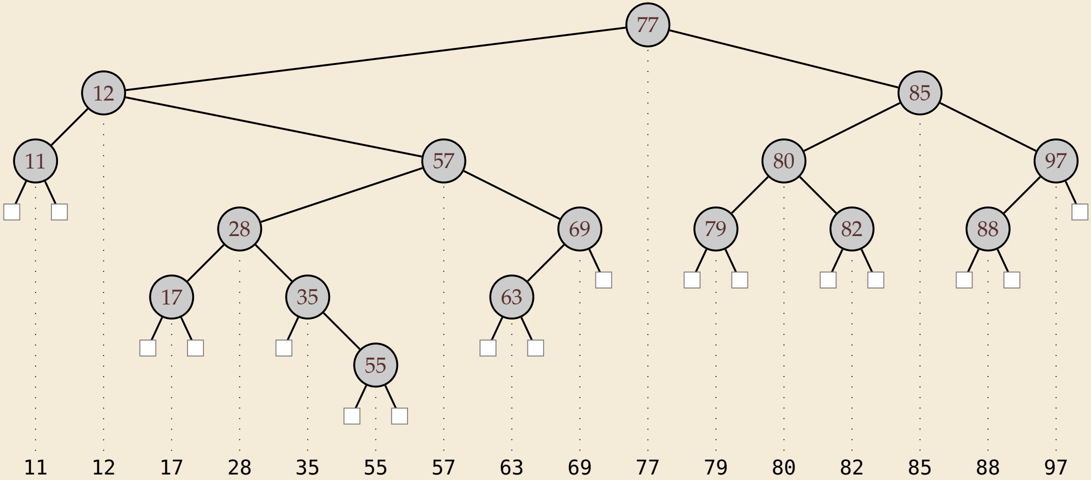
all keys in left subtree $\le$ root key $\le$ all keys in right subtree

BST example & find



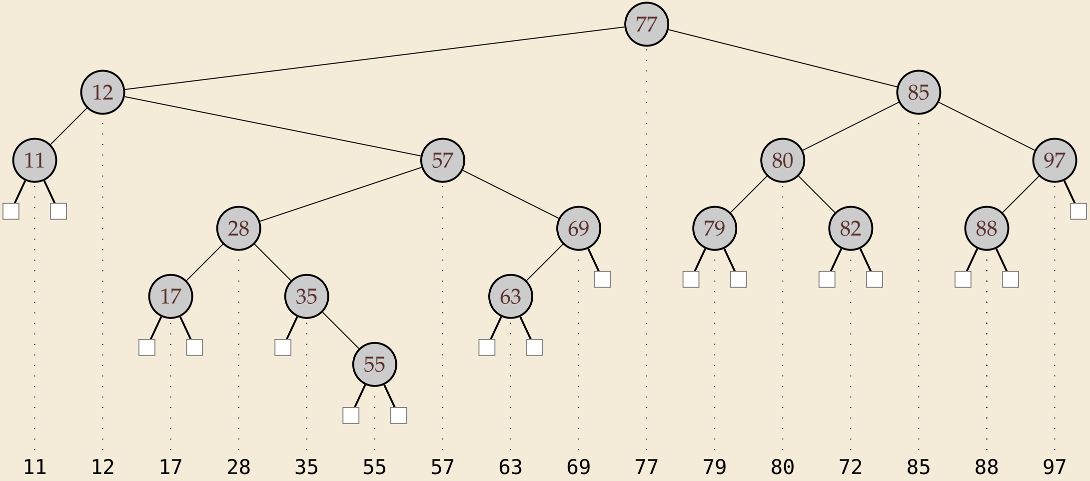
BST insert

Example: Insert 88



BST delete

- Easy case: remove leaf, e. g., 11 ⇝ replace by `null`
- Medium case: remove unary, e. g., 69 ⇝ replace by unique child
- Hard case: remove binary, e. g., 85 ⇝ swap with predecessor, recurse



Analysis

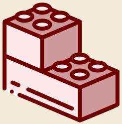
BST summary

Operation	Running Time
`construct(`$A[1..n]$`)`	$O(nh)$
`put(`k`,`v`)`	$O(h)$
`get(`k`)`	$O(h)$
`delete(`k`)`	$O(h)$
`contains(`k`)`	$O(h)$
`isEmpty()`	$O(1)$
`size()`	$O(1)$

2.5 Ordered Symbol Tables

Ordered symbol tables

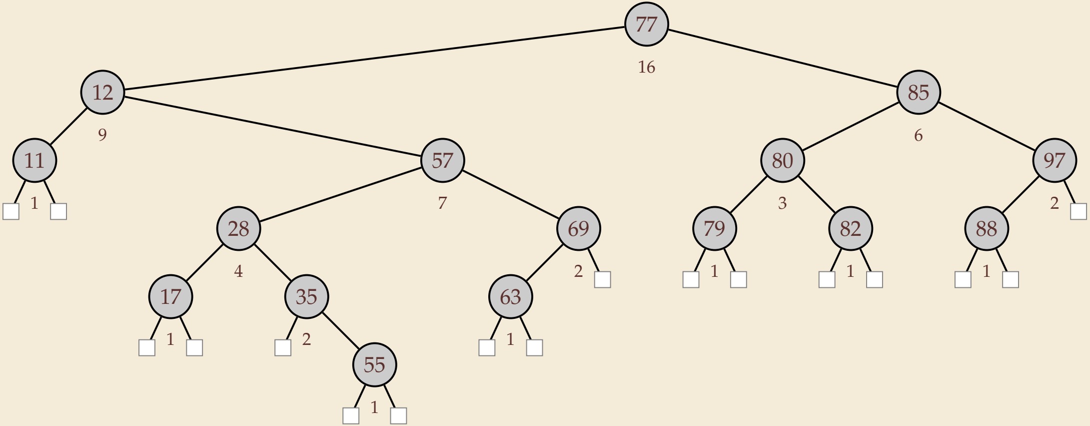
- `min()`, `max()`
  Return the smallest resp. largest key in the ST
- `floor(`x`)`, $\lfloor x \rfloor = \mathbb{Z}$`.floor(`$x$`)`
  Return largest key k in ST with $k \leq x$.
- `ceiling(`x`)`
  Return smallest key k in ST with $k \geq x$.
- `rank(`x`)`
  Return the number of keys k in ST $k < x$.
- `select(`i`)`
  Return the ith smallest key in ST (zero-based, i. e., $i \in [0..n)$)



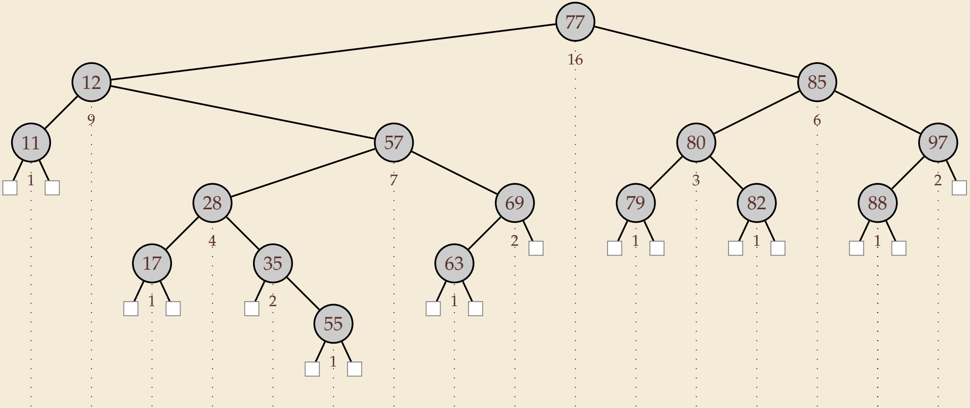
With select, we can simulate access as in a truly dynamic array!.

(Might not need any keys at all then!)

Augmented BSTs

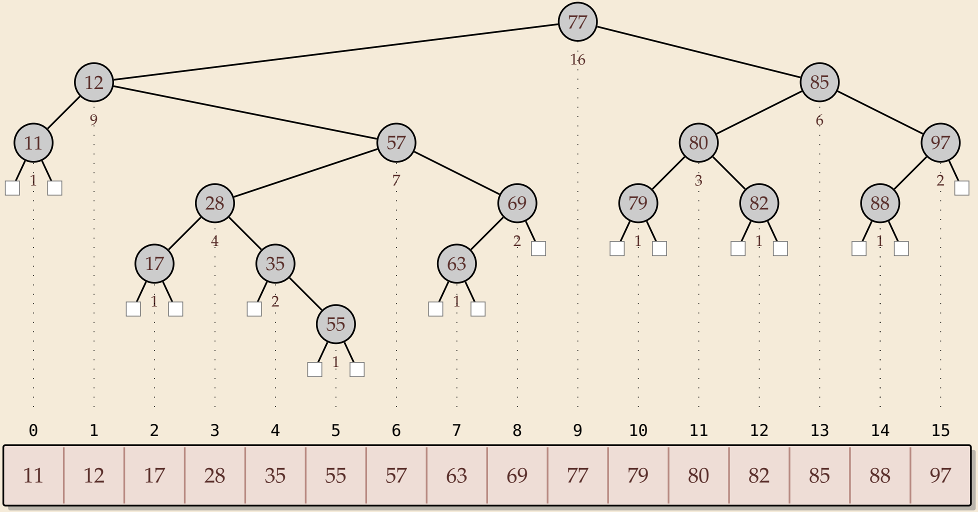


Rank



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
11	12	17	28	35	55	57	63	69	77	79	80	82	85	88	97

Select



77
16
12
9
85
6
11
1
57
7
80
3
97
2
28
4
69
2
79
1
82
1
88
1
17
1
35
2
63
1
55
1
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
11 12 17 28 35 55 57 63 69 77 79 80 82 85 88 97

2.6 Balanced BSTs

Balanced BSTs

Balanced binary search trees:

- imposes shape invariant that guarantees $O(\log n)$ height
- adds rules to restore invariant after updates
- many examples known
  - *AVL trees* (height-balanced trees)
  - *red-black trees*
  - *weight-balanced trees* ($\mathrm{BB}[\alpha]$ trees)
  - ...

Other options:

I'd love to talk more about all of these ...
(Maybe another time)

- **amortization:** *splay trees*, *scapegoat trees*
- **randomization:** *randomized BSTs*, *treaps*, *skip lists*

BSTs vs. Heaps

Balanced binary search tree

Operation	**Running Time**
`construct(`$A[1..n]$`)`	$O(n \log n)$
`put(`k`,`v`)`	$O(\log n)$
`get(`k`)`	$O(\log n)$
`delete(`k`)`	$O(\log n)$
`contains(`k`)`	$O(\log n)$
`isEmpty()`	$O(1)$
`size()`	$O(1)$
`min()` / `max()`	$O(\log n)$ $\rightsquigarrow$ $O(1)$
`floor(`x`)`	$O(\log n)$
`ceiling(`x`)`	$O(\log n)$
`rank(`x`)`	$O(\log n)$
`select(`i`)`	$O(\log n)$

~~**Binary heaps**~~ *Strict Fibonacci heaps*

Operation	**Running Time**
`construct(`$A[1..n]$`)`	$O(n)$
`insert(`x`,`p`)`	~~$O(\log n)$~~ $O(1)$
`delMax()`	$O(\log n)$
`changeKey(`x`,`p'`)`	~~$O(\log n)$~~ $O(1)$
`max()`	$O(1)$
`isEmpty()`	$O(1)$
`size()`	$O(1)$

- apart from faster `construct`, BSTs always as good as binary heaps
- MaxPQ abstraction still helpful
- and faster heaps exist!