Big-Oh basic properties

$$\left.\begin{array}{l} f(n) = O(g_1(n)) \\ h(n) = O(g_2(n)) \end{array}\right. \Rightarrow f(n)\cdot g(n) = O(g_1(n)\cdot g_2(n))$$

$$f(n) = O(g(n)) \Rightarrow k\cdot f(n) = O(g(n))$$

$$\left.\begin{array}{l} f(n) = O(g_1(n)) \\ h(n) = O(g_2(n)) \end{array}\right. \Rightarrow (f+h) = O(g_1+g_2)$$

$$f(n) = h(n) + O(g(n))$$

$$\Rightarrow f(n) - h(n) = O(g(n))$$

little-Oh

$$f(x) = o(g(x)) \ : \ \forall M \ (\exists x_0 \mid f \leq M\cdot g, \ x_0 < x)$$

↖ no matter how small

$$\Rightarrow \frac{f}{g} \leq M, \ x_0 < x$$

$$\Rightarrow f/g \to 0, \ x \to \infty$$

Big-Ω

$$f(n) \geq k\cdot g(n) \qquad n_0 < n$$

$$f(n) = O(1) \Rightarrow f(n) \leq k\cdot 1 \qquad n_0 < n$$

Lower-order terms can be ignored.

$$\begin{aligned} f(n) &= O(g(n)) \\ h(n) &= O(1) \end{aligned} \quad \Rightarrow \quad \begin{aligned} f(n) + h(n) &\le k_1 g(n) + k_2(1) \\ &\le k_1 g(n) + k_2 g(n) \\ &\le (k_1 + k_2)\, g(n) = O(g(n)) \end{aligned}$$

$$\max(n_0^f, n_0^h) < n$$

$$\begin{aligned} f(n) &= O(g(n)) \\ h(n) &= O(1) \end{aligned} \quad \Rightarrow \quad \begin{aligned} f(n) \cdot h(n) &\le k_1 g(n) \cdot k_2(1) \\ &\le k_1 \cdot k \cdot g(n) \\ &= O(g(n)) \end{aligned}$$

$$\max(n_0^f, n_0^h) < n$$

```
Prog(n)
    init()
    for i = 1 to n
        setup()
        search(n)
```

Suppose

init() runs in $O(1)$ time

setup() runs in $O(1)$ time

search runs in $O(n)$ time

Total runtime

$$r(n) = O(1) + \underset{\text{n loop iterations}}{n} \left(\underbrace{O(1) + O(n)}_{\text{time for 1 iteration of loop}} \right)$$

$$\begin{aligned} r(n) &\le k_1(1) + n(k_2(1) + k_3 n) \\ &= k_1 + k_2 \cdot n + k_3 \cdot n^2 \\ &\le (k_1 + k_2) n + k_3 n^2 \\ &\le (k_1 + k_2 + k_3) n^2 = O(n^2) \end{aligned}$$

$$\max\{n_0^1, n_0^2, n_0^3\} < n$$

$$r(n) = O(1) + g(n)$$

$$\begin{aligned} g(n) &= n(O(1) + O(n)) \\ &= n \cdot O(n) \\ &= O(n^2) \end{aligned}$$

$$r(n) = O(1) + O(n^2) = O(n^2)$$

Suppose runtime is exactly,

$$r(n) = 100 + 5\cdot n + 10\cdot n\log(n)$$

Big-Oh is meant to capture the "most important" aspects of runtime complexity. When n gets moderately large, lower-order terms become insignificant.

Suppose an improved version has runtime,

$$r^{new}(n) = r(n) - 10n$$

or

$$\Delta r = r(n) - r^{new}(n) = 10n$$

Consider the % improvement

$$\frac{\Delta r}{r(n)} = \frac{10\cdot n}{100 + 5n + 10n\log n}$$

$$\left(\text{Let } n = 2^{30} = 1G\right) \overset{\sim}{\Rightarrow} \frac{10^{10}}{10^2 + 5\cdot 10^9 + 10^{10}\cdot 30}$$

$\approx 1/30$ or about 3% improvement

find(k, list)

find k-th largest in unordered list.

```
heap = buildHeap(list)
for (i = 1 to k)
        x = deleteMax(heap)
return(x)
```

Suppose runtime of buildHeap() is $b(n)$ for list of size n.

$$b(n) = O(1)\left[\log(n) + \cdots + \log(2)\right]$$
$$= O(n\log n)$$

Suppose runtime of deleteMax() is $g(n)$ for heap of size n.

runtime is $r(n) = \underbrace{O(1)}_{\uparrow\ \text{return + assignment overhead}} + b(n) + k\cdot \underbrace{O(1)}_{\uparrow\ \text{loop overhead}} + \underbrace{\left(g(n) + g(n-1) + \cdots + g(n-k+1)\right)}_{\sum_{0}^{k-1} g(n-i)}$

$$\leq k\cdot g(n)$$
$$\leq k\cdot\log n$$

$$r(n) = O(1) + O(k) + O(k\cdot g(n)) + b(n)$$
$$= O(k\cdot\log n) + b(n)$$
$$= O(n\log n)$$

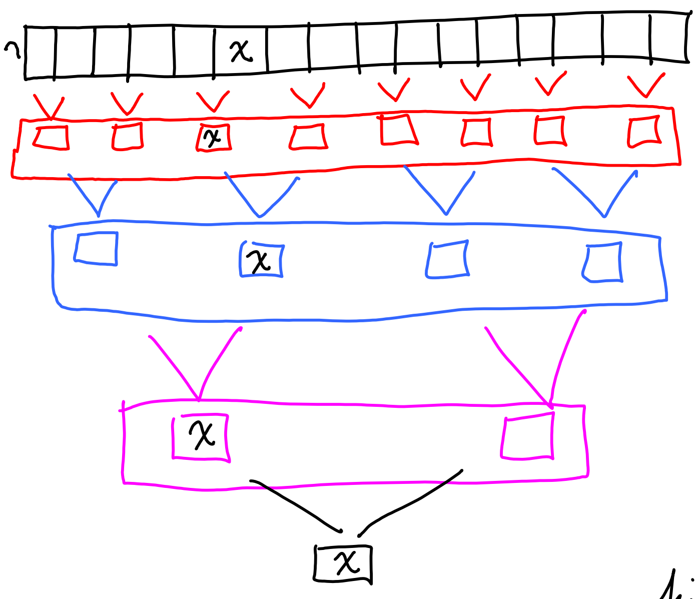
(n log n) is certainly an upper bound, but is there another, tighter bound which shows that find() runs significantly faster than (n log n)?

Can we find x_k from $\{x_i\}$ where $\underbrace{x_1 = \min, \ldots, x_n = \max}_{\text{sorted order}}$ w/o sorting? Our find() essentially sorts the list until it finds the k-th largest.

buildHeap(n) has an upper bound on its runtime of $O(n \log n)$. But, could we re-implement it to run faster, or is the task of building a heap $\Omega(n \log n)$? What if we skip building a heap, can we do find() faster, or is the task of finding k-th max $\Omega(n \log n)$?

Tournament tree?

find(k, list) has runtime $r(n) = \Theta(n)$?



x is max

list.size = n

get maxs by pairs, $\Theta(n)$

list.size = $n/2$

get maxs by pairs, $\Theta(n/2)$

list.size = $n/4$

get maxs by pairs, $\Theta(n/4)$

list.size = $n/8$

$$\text{find}(1, n) \Rightarrow \sum_{i=0}^{\log n} \Theta(n/2^i) = \Theta(1) \sum_{0}^{\log n} n/2^i$$

$$= \Theta\left(n \cdot \sum_{0}^{\log n} 1/2^i\right)$$

$$1 + 1/2 + 1/4 + \cdots \leq 2 \Rightarrow \quad = \Theta(n)$$

Finding the 2nd max, y: Since y was 2nd max, it won every match it was in, until it lost to x. Collect all items x was compared to into a new list, L, as we go through the tournament. Run y = find(1, L). What's the total runtime to find y given the original list?

If extending this scheme is going to work, what are the requirements on runtimes?

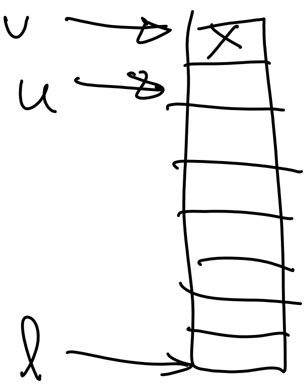
Suppose we have a scheme that breaks a job of size n into smaller pieces of size $n/f(n)$, and $r(n)$ time. Suppose each piece is solved in $r(x)$ time, $x = n/f(n)$. Suppose the work to combine them into a solution is $h(n)$. Finally, there are $p(n)$ sub-problems.

$$r(n) = h(n) + p(n) \cdot r\left(n/f(n)\right)$$

Sorting

Info theoretic bound, Ω

Selection Sort



$x = \text{findMax}(v, l)$

$\Theta(1)\cdot n + \Theta(1)\cdot(n-1) + \dots + \Theta(1)\cdot(1)$

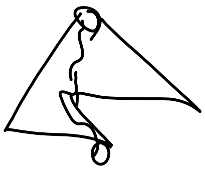
$\Theta(1)\cdot\sum_{i=1}^{n} i = \Theta(1)\,\frac{n(n-1)}{2}$

$= \Theta(n^2)$

$\text{sort}(n) = \Omega(n^2)$?

heap sort

```
HS(list)
    heap = buildHeap(list)
    for 1 to n
        newlist.insert( deleteMax(heap))
```



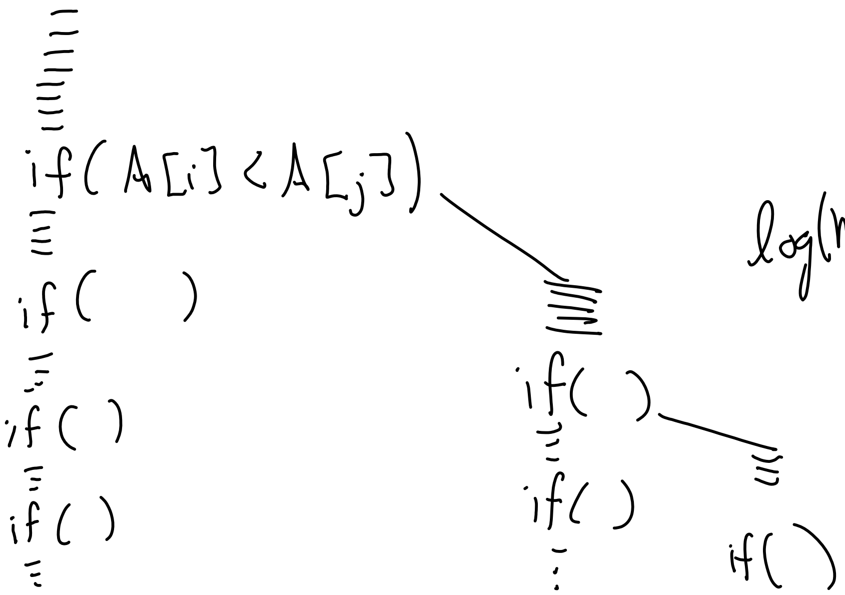
$\log(i)$

$\sum_{i=1}^{n}\log(i)$

$\Theta(n\log n) + \Theta(n\log n)$

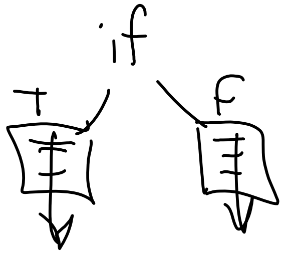
$\Theta(n\log n)$

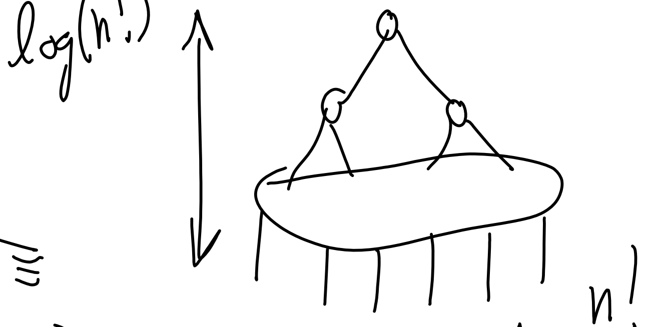
Sort (list$_1$)



list$_1$ is sorted

size n

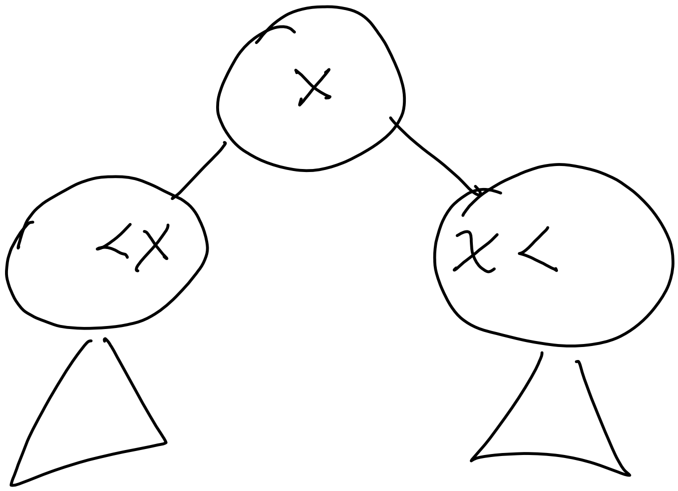




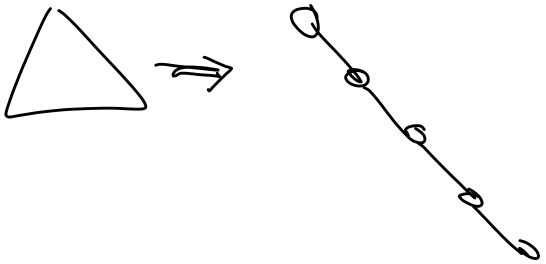
$\log(n^n) = \overset{\text{depth}}{n \log n}$

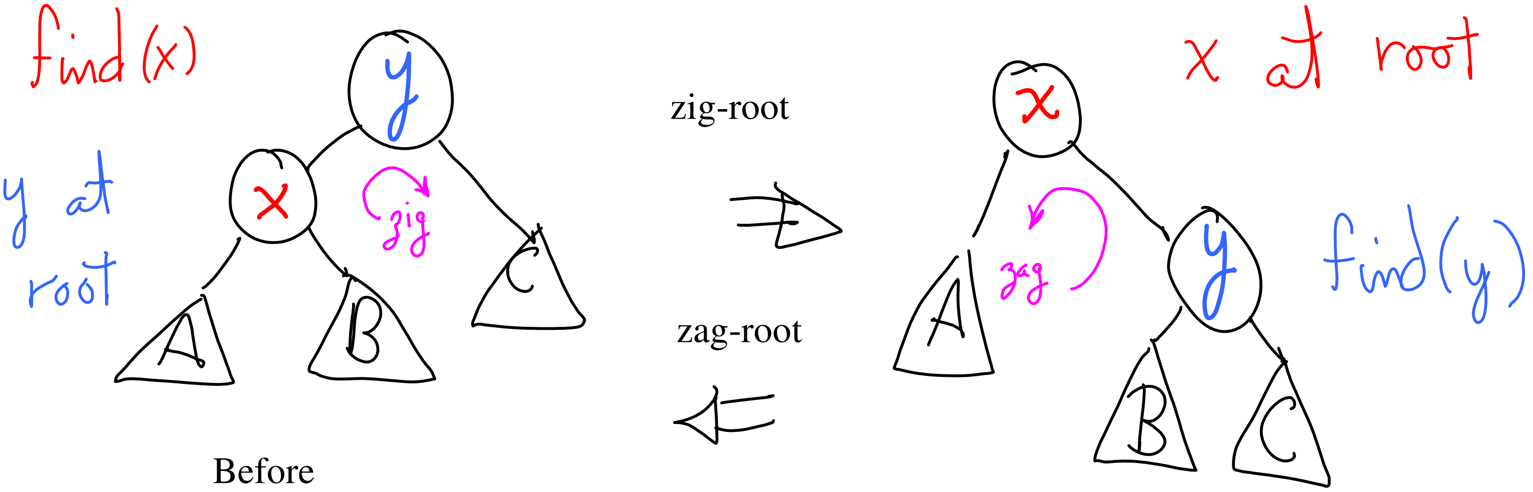
complexity of sorting = $\Omega(n \log n)$

Splay trees: self-adjusting binary search trees. Last accessed item is at root (fast to access again), tree is binary search tree, tree tends to be balanced, items most accessed are close to root.



Invariant: For splay tree T,
if X is at the root of any subtree, every item in left-subtree is less thant X, and every item in right-subtree is more than X.





zig-root

zag-root

Before

x < y,   x <=== y

y < C,   y ===> C

A < x,   A <=== x.

x < B,   B ===> x

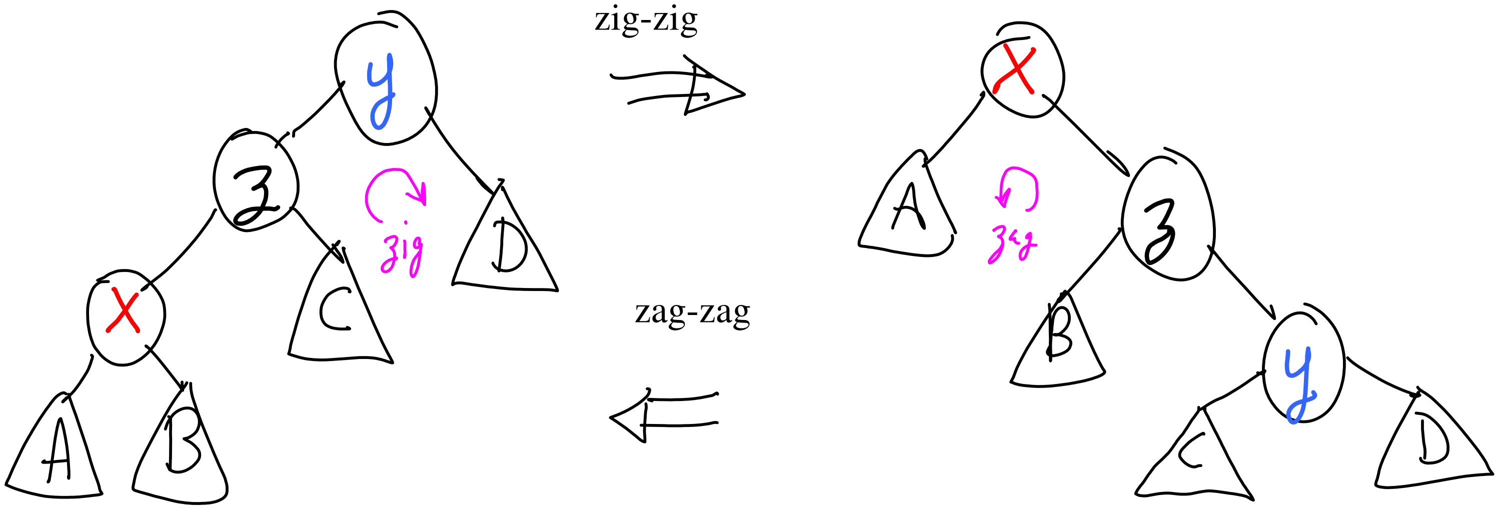
After

A < x,   A <=== x

x < y,   x ===> y

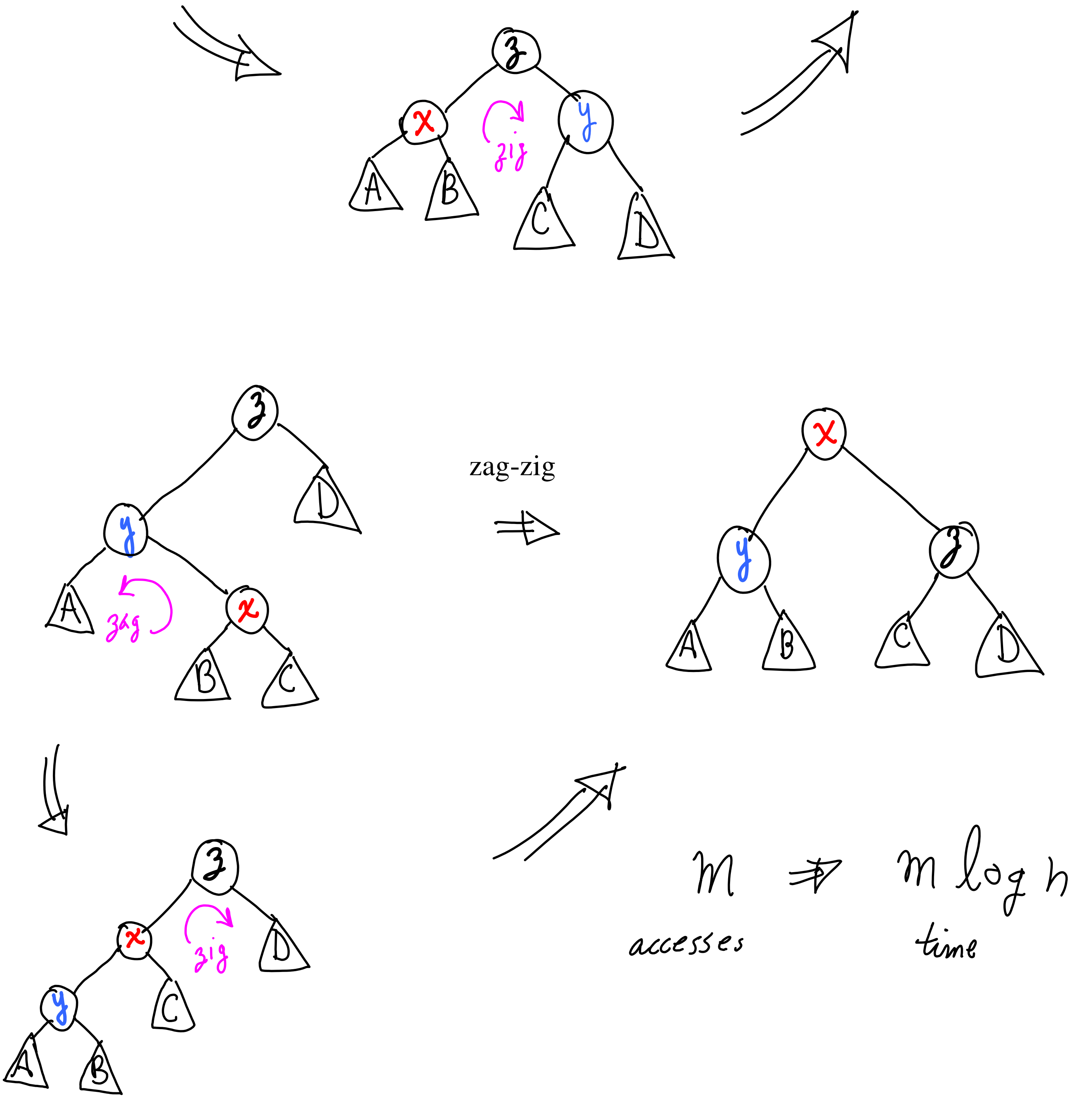
B < x,   B <=== x

x < C,   x ===> C



zig-zig

zag-zag



Q. Exam 3

Write pseudo-code to implement find(x) for a splay tree. The function find() does a binary search for item x. NOTE: The tree is not guaranteed to be full; that is, some non-leaf nodes might have only one child. Nevertheless, the splay property is invariant. For example, sub-tree D above might be empty, in which case z has a NULL right-child pointer. After finding x in the tree, find () splays (zig-zig, zag-zig, zig-zag, zag-zag) recursively to the root. At the root the last step is either a zig or a zag, unless the previous operation put x at the root.

Hints: Use recursion; nodes should have parent, left, and right pointers; Data in the nodes can be integer.