CS 331, Fall 2026 Today: Data structures

Mini-lecture 3 (9/2)

Data structures

Many algos require same subroutines.

- Maintain a set of items
- Find smallest item
- Predecessor / successor / search / ...

Data structure = be lazy & solve just once

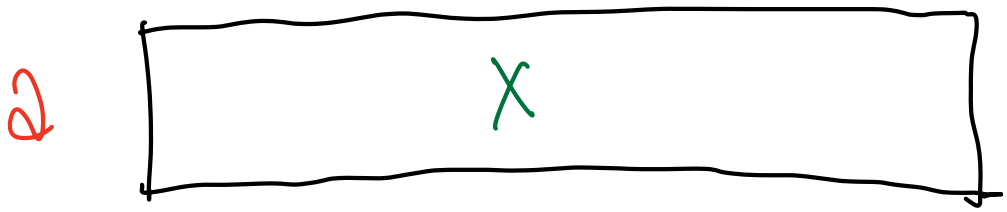
Reusable abstraction of common subroutines

Presented as an API: inner workings irrelevant today

Notation

x, y, z ← objects, fit in $O(\log(n))$-bit words

a, b, c ← addresses where objects stored



address

$a = \&x$

$x = *a$

dereference

Lists (Pt I, Section 7.1)

Goal: maintain set of objects w/ indices

Supported operations:

- Insert
- Delete
- Query

} ideally, $O(1)$ time?

Array

Size

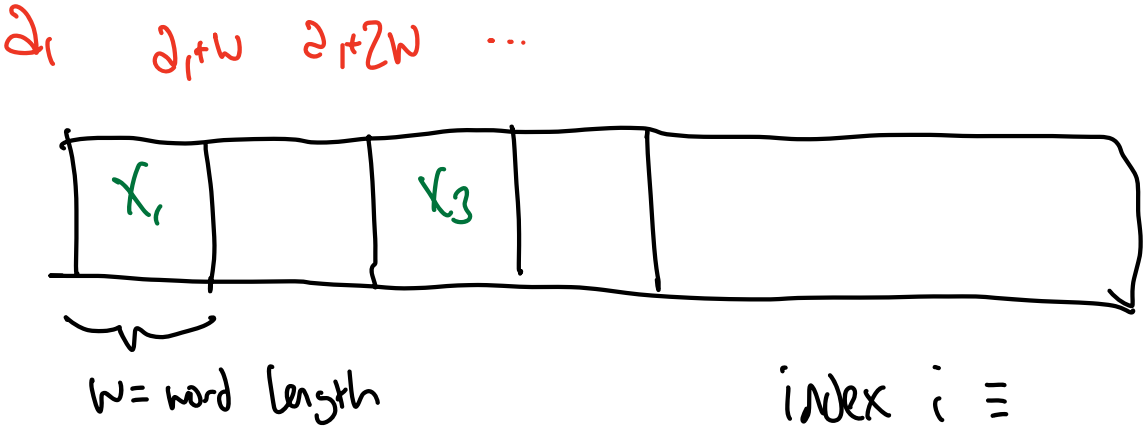
Cost

API: Init(n) O(1)

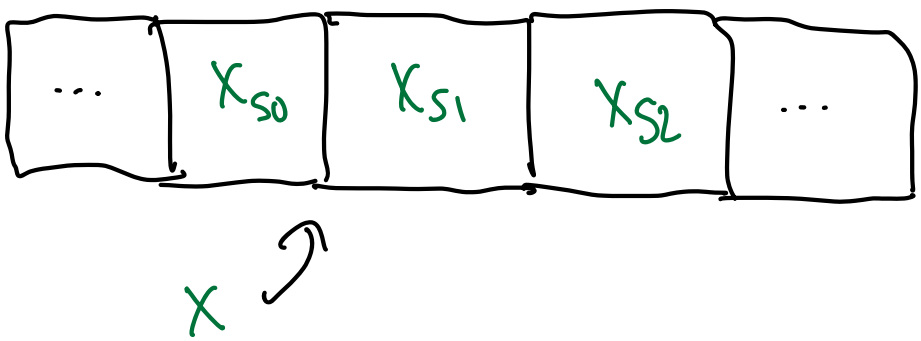
Insert(x, i) O(1)

Delete(i) O(1)

Query(i) O(1)



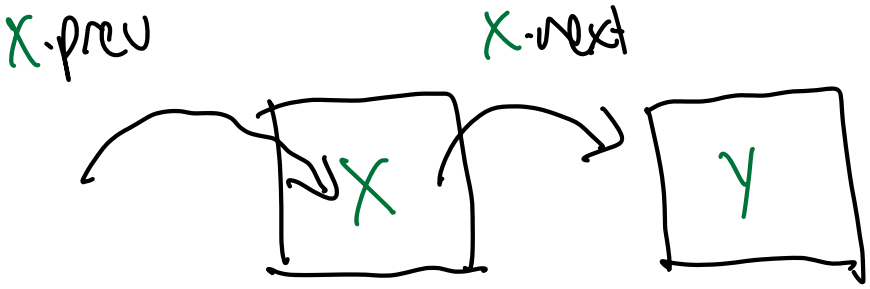
What's it bad at?



Linked List

Key idea: if you know $a = \&x$

Can easily manipulate x's neighbors



Tradeoff: dynamic indices

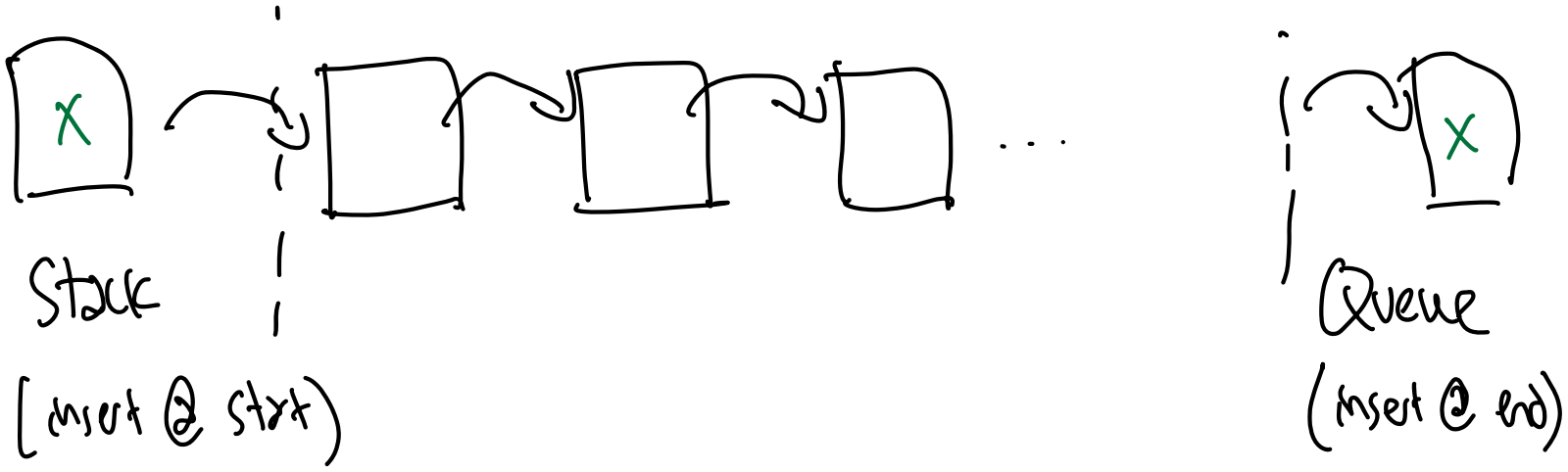
API:

Init()	O(1)
InsertAfter(x, a)	O(1)
Delete(a)	O(1)
Query(a)	O(1)

Q: How long would these take if a = index?

Stack/Queue

Restricted LinkedList (can simplify algo code!)

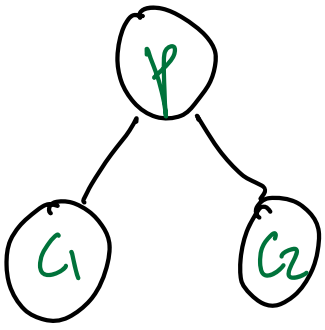


Stack: Push(x), Pop(), Peek() = all O(1)

Queue: Queue(x), Dequeue(),

Heaps (Part I, Section 7.2)

Heap invariant:
"smaller rises"

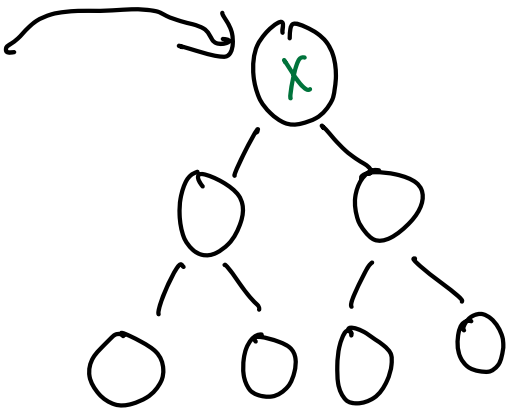


$p \leq c_1$

$p \leq c_2$

Heaps always maintain heap invariant after steps!

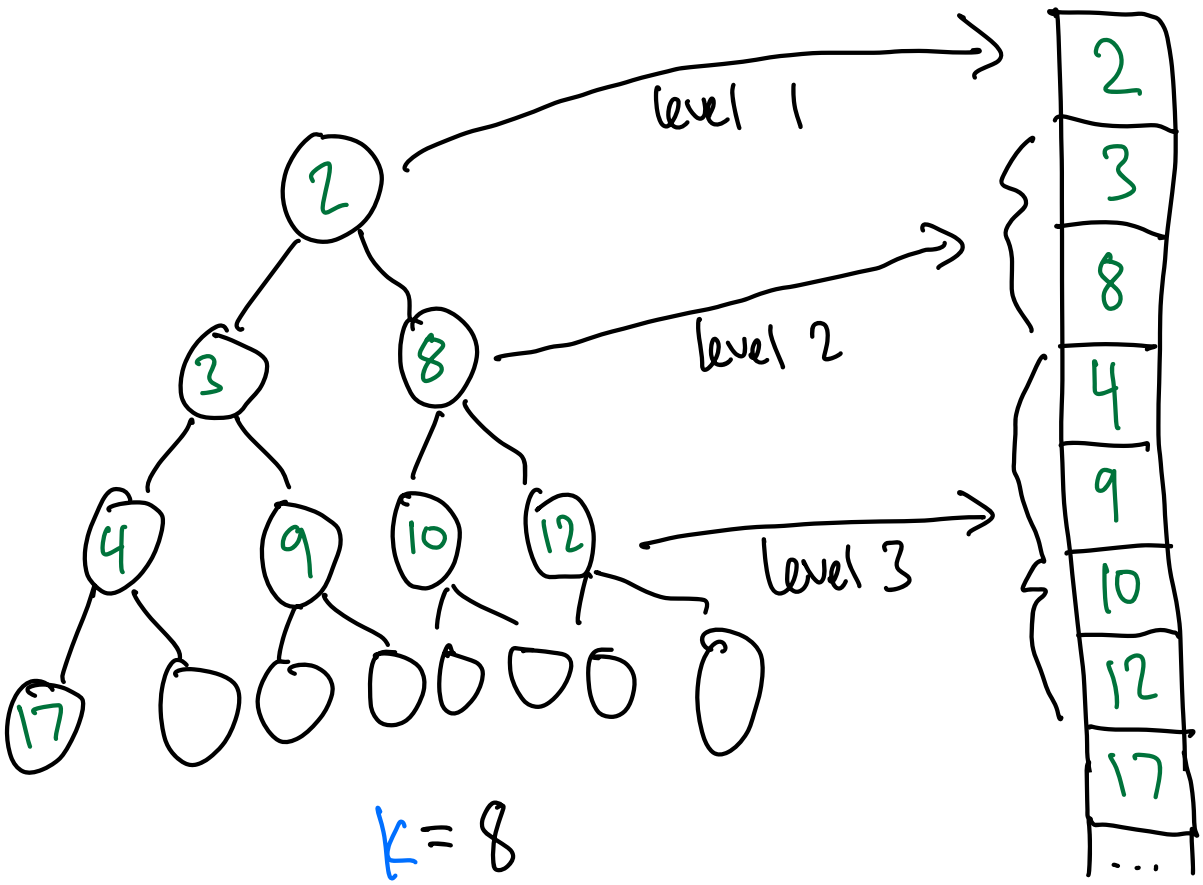
PeekMin(): O(1) time



ExtractMin(): O(log(n)) time

n = max # objects

Idea: always maintain as 1st K entries in binary tree ≡ Array in background



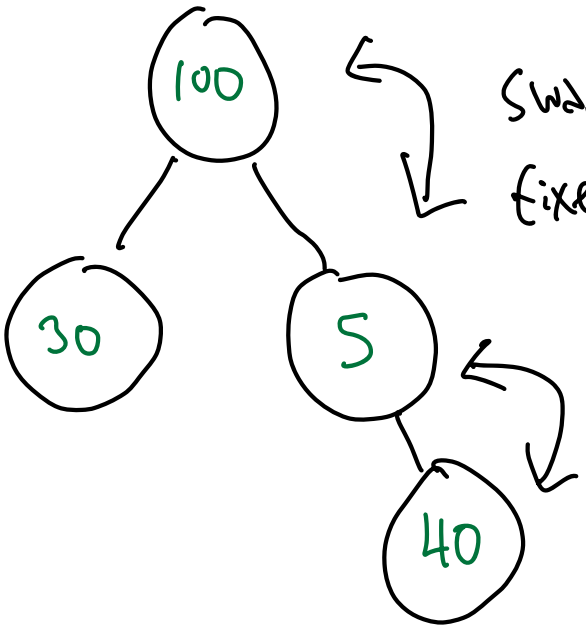
K = 8

To ExtractMin(),

- Swap 1st & k^{th} objects
- delete k^{th} object, k--
- let 1st object "sink"

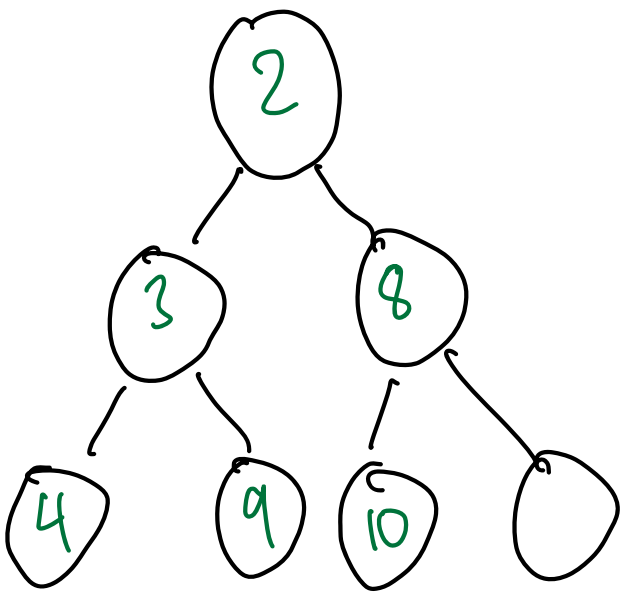
O(log(n)) time →

Sinking:



swap w/ smaller child
fixes locally, +1 level
recurse...

Delete(a): O(log(n)) time if you know where it is

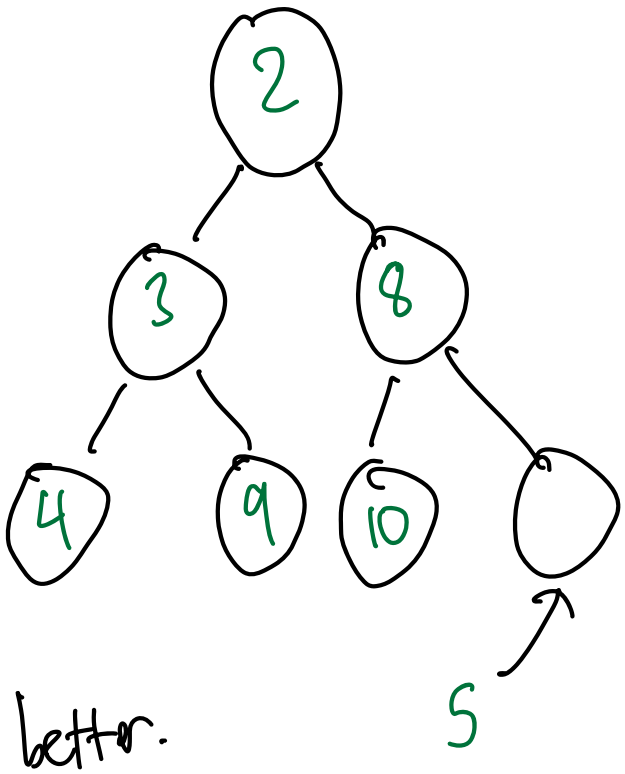


- Swap with k^{th}
- delete k^{th}, k--
- let old k^{th} sink

insert(x): O(log(n)) time

- add at end, k++
- let new object "float"

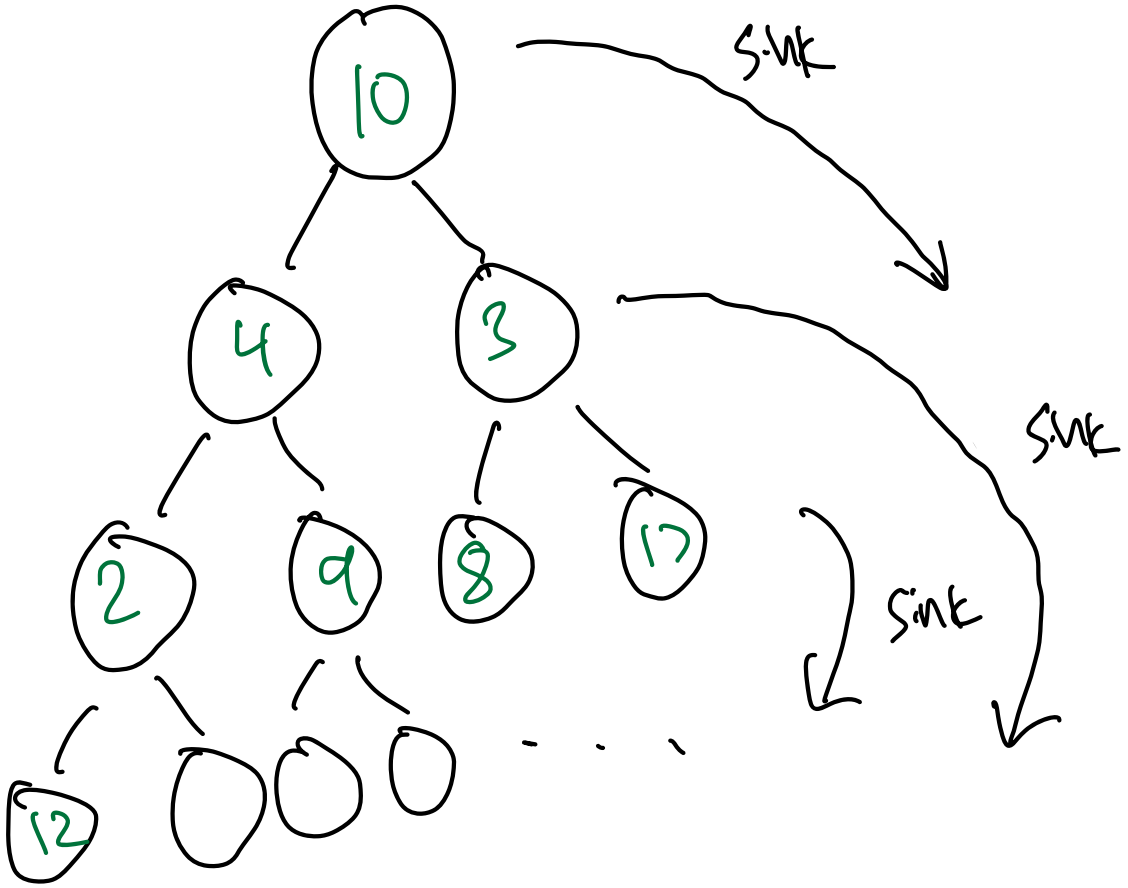
  (swap w/ parent if smaller)



Old objects can't create

issues: heap invariant only gets better.

init(): O(n) time



1 x h

2 x (h-1)

4 x (h-2)

⋮

n/2 x 1

bottom heavy

Binary Search Trees (Part I, Section 7.3)

Motivation: Linked List insert = $O(1)$

search = $O(n)$

Can we balance costs?

"Sorted List" insert = $O(\log(n))$

search = $O(\log(n))$

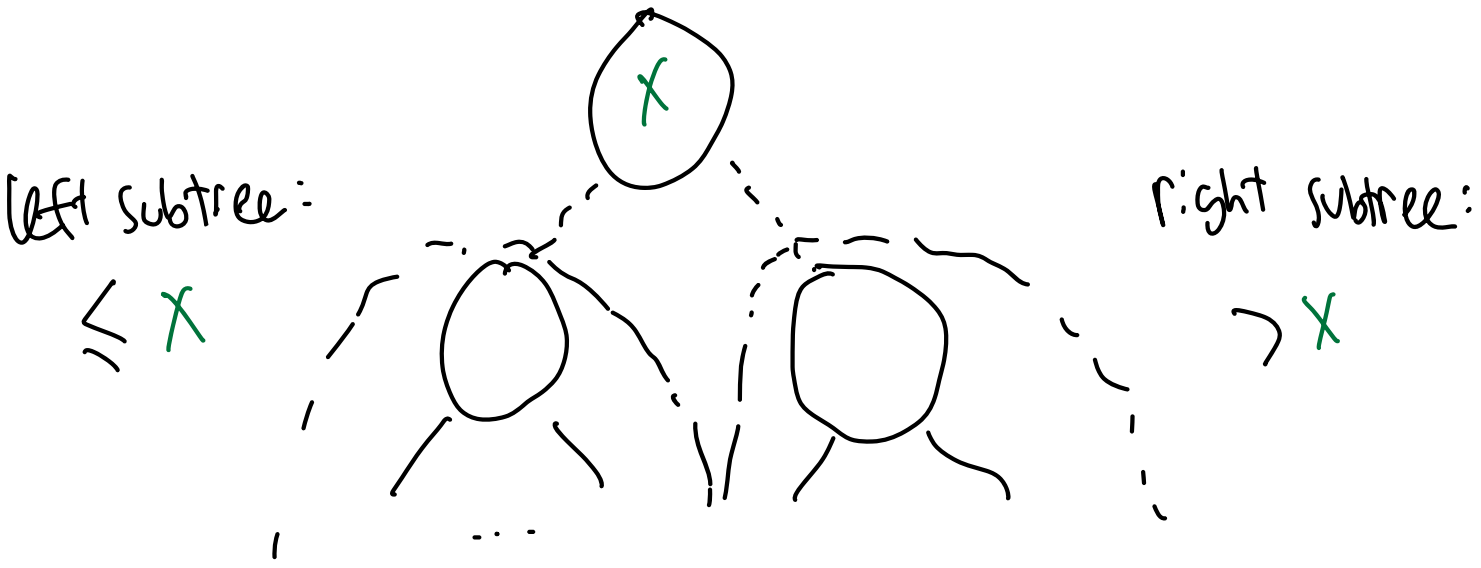
Yes! This is what BST does. (+ more)

API: Search(x)
Insert(x)
Delete(x)
Query(:)
Index(x)

} all $O(\log(n))$

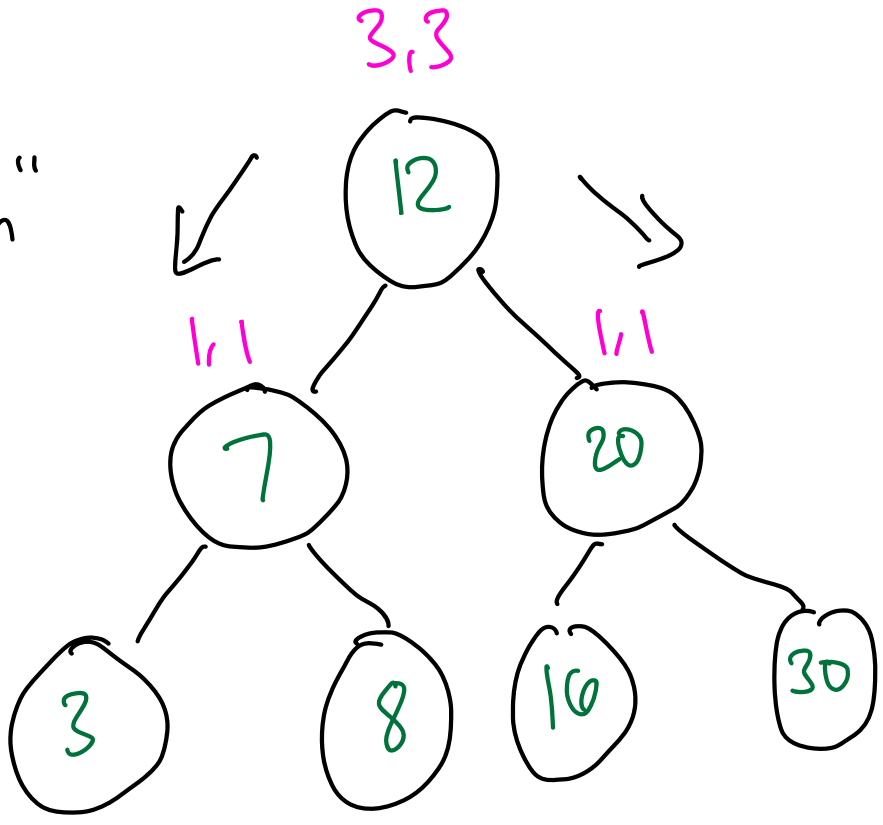
Ex: implement predecessor

BST invariant



Search(x) : height of tree h

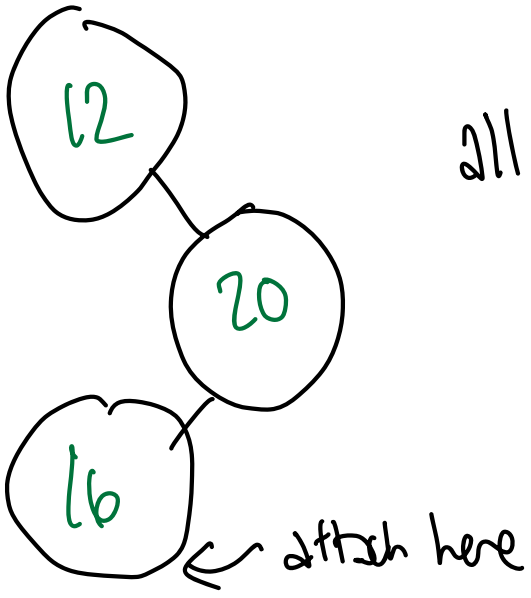
"binary search"



Index / Query: augment w/ subtree sizes

Simple Insert: Search + attach

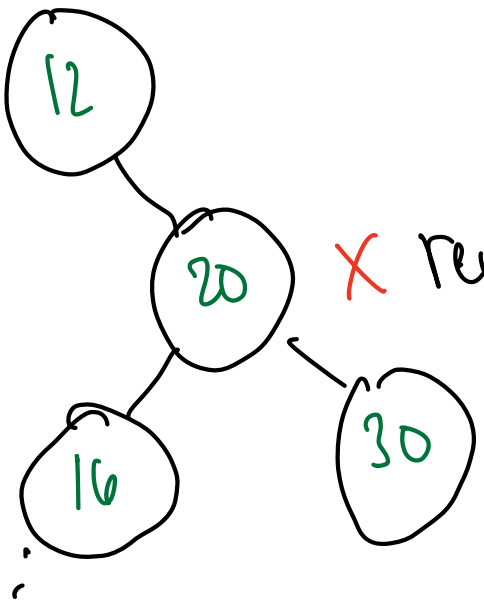
$x = 19$



all subtrees correct

Simple Delete: Search + Splice

$x = 20$



X replace w/ predecessor = left node

only hard case: two children

Hard part: maintaining $h = O(\log(n))$

AVL tree, red black tree