CSE 373 SP 18 - KASEY CHAMPION

B-Tree Insertions, Intro to Heaps

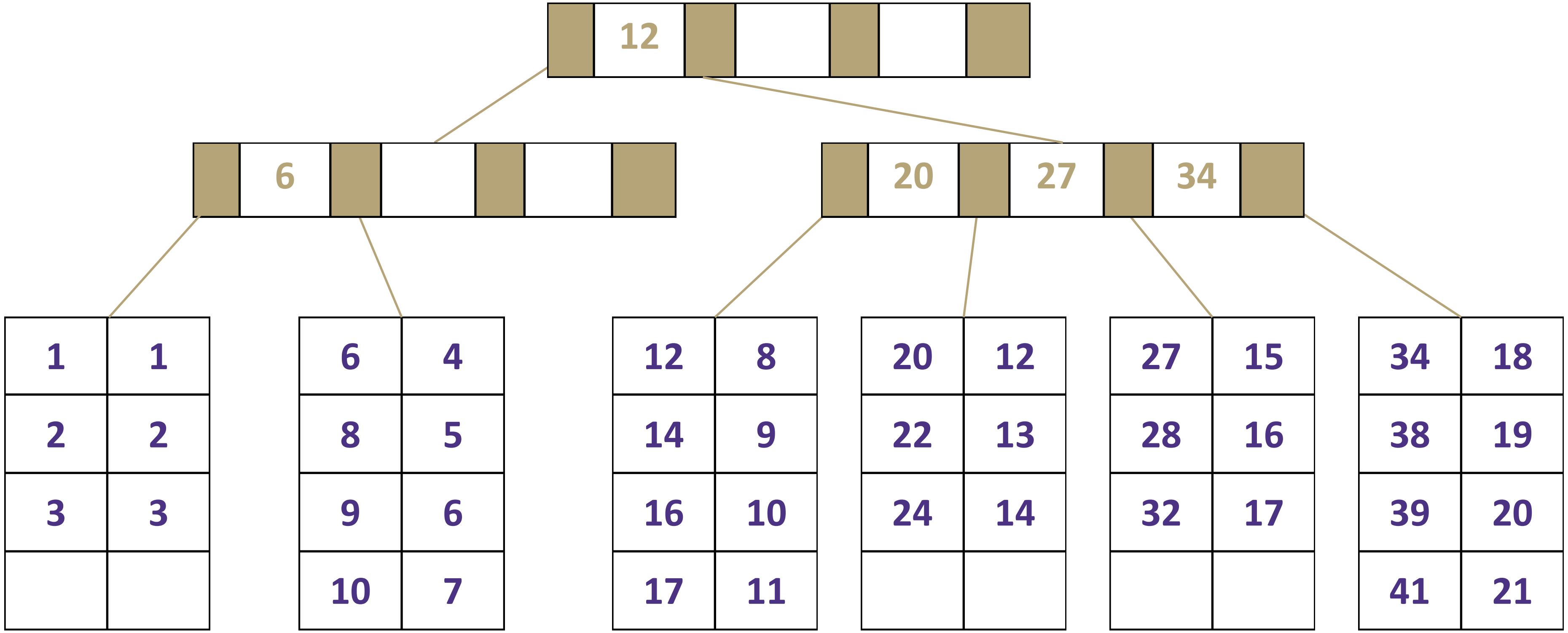
Data Structures and Algorithms

Warm Up

What operations would occur in what order if a call of get(24) was called on this b-tree?

What is the M for this tree? What is the L?

If Binary Search is used to find which child to follow from an internal node, what is the runtime for this get operation?



Administrivia

1. Midterm grades will be published by Friday
2. HW #4 is due Friday
3. 1 partner must fill out partner form by Friday
4. HW Grade Review Requests coming next week

Review: B-Trees

Has 3 invariants that define it

1. B-trees must have two different types of nodes: internal nodes and leaf nodes
   - An **internal node** contains M pointers to children and M – 1 **sorted** keys.
   - M must be greater than 2
   - **Leaf Node** contains L key-value pairs, sorted by key.

2. B-trees order invariant
   - For any given key k, all subtrees to the left may only contain keys that satisfy x < k
   - All subtrees to the right may only contain keys x that satisfy k >= x

3. B-trees structure invariant
   - If n<= L, the root is a leaf
   - If n >= L, root node must be an internal node containing 2 to M children
   - All nodes must be at least half-full

Put() for B-Trees

Build a new b-tree where M = 3 and L = 3.

Insert (3,1), (18,2), (14,3), (30,4) where (k,v)

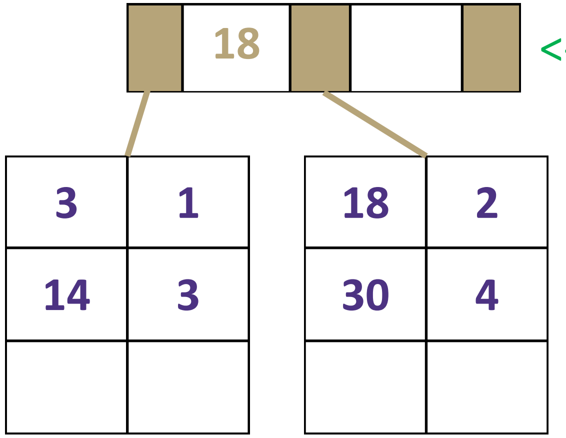
When n <= L b-tree root is a leaf node

wrong ->

3	1
18	2
14	3

No space for (30,4) ->**split** the node

Create two new leafs that each hold ½ the values and create a new internal node



<- use smallest value in larger subset as sign post

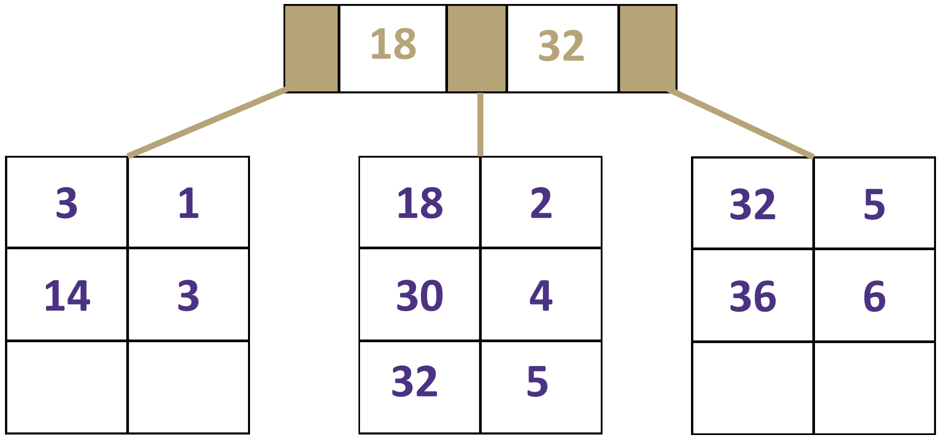
2. B-trees order invariant

For any given key k, all subtrees to the left may only contain keys that satisfy x < k

All subtrees to the right may only contain keys x that satisfy k >= x

You try!

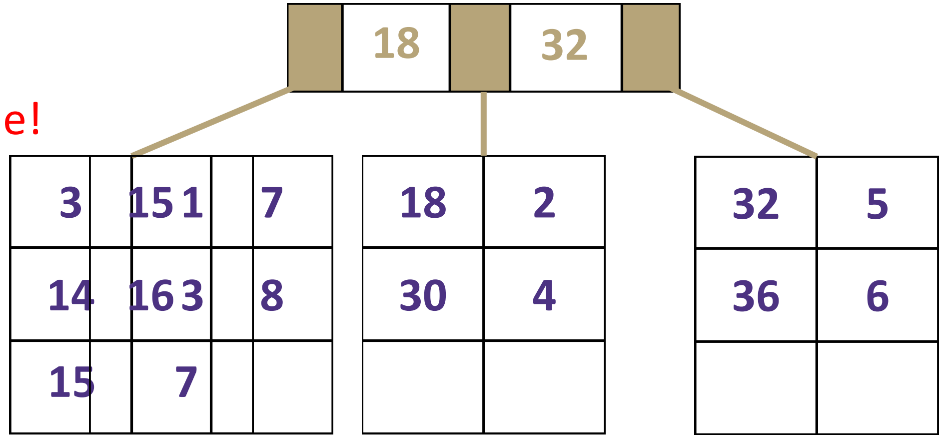
Try inserting (32, 5) and (36, 6) into the following tree



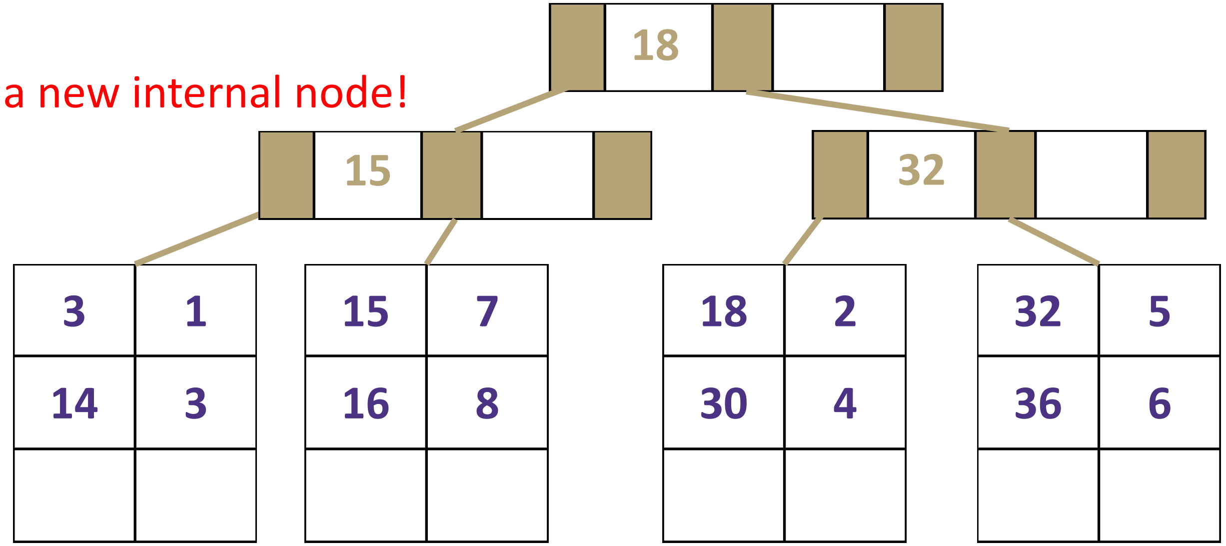
Splitting internal nodes

Try inserting (15, 7) and (16, 8) into our existing tree

Make a new internal node!



Make a new internal node!



B-tree Run Time

Time to find correct leaf **Height = $\log_m(n)\log_2(m)$ = tree traversal time**

Time to insert into leaf **$\Theta(L)$**

Time to split leaf **$\Theta(L)$**

Time to split leaf's parent internal node **$\Theta(M)$**

Number of internal nodes we might have to split **$\Theta(\log_m(n))$**

All up worst case runtime: **$\Theta(L + M\log_m(n))$**

New Topic: Heaps

Priority Queue ADT

Imagine you have a collection of data from which you will always ask for the extreme value

Min Priority Queue ADT	Max Priority Queue ADT
state Set of comparable values - Ordered based on "priority" **behavior** **removeMin()** – returns the element with the smallest priority, removes it from the collection **peekMin()** – find, but do not remove the element with the smallest priority **insert(value)** – add a new element to the collection	**state** Set of comparable values - Ordered based on "priority" **behavior** **removeMax()** – returns the element with the largest priority, removes it from the collection **peekMax()** – find, but do not remove the element with the largest priority **insert(value)** – add a new element to the collection

Implementing Priority Queue

Idea	Description	removeMin() runtime	peekMin() runtime	insert() runtime
Unsorted ArrayList	Linear collection of values, stored in an Array, in order of insertion	O(n)	O(n)	O(1)
Unsorted LinkedList	Linear collection of values, stored in Nodes, in order of insertion	O(n)	O(n)	O(1)
Sorted ArrayList	Linear collection of values, stored in an Array, priority order maintained as items are added	O(1)	O(1)	O(n)
Sorted Linked List	Linear collection of values, stored in Nodes, priority order maintained as items are added	O(1)	O(1)	O(n)
Binary Search Tree	Hierarchical collection of values, stored in Nodes, priority order maintained as items are added	O(n)	O(n)	O(n)
AVL tree	Balanced hierarchical collection of values, stored in Nodes, priority order maintained as items are added	O(logn)	O(logn)	O(logn)

Let’s start with an AVL tree

AVLPriorityQueue<E>

state

`overallRoot`

behavior

`removeMin()` `- traverse through tree all the way to the left, remove node, rebalance if necessary`

`peekMin()` `- traverse through tree all the way to the left`

`insert()` `- traverse through tree, insert node in open space, rebalance as necessary`

What is the worst case for peekMin()? **O(logn)**

What is the best case for peekMin()? **O(1)**

Can we do something to guarantee best case for these two operations?

Binary Heap

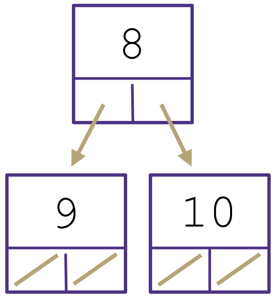
A type of tree with new set of invariants

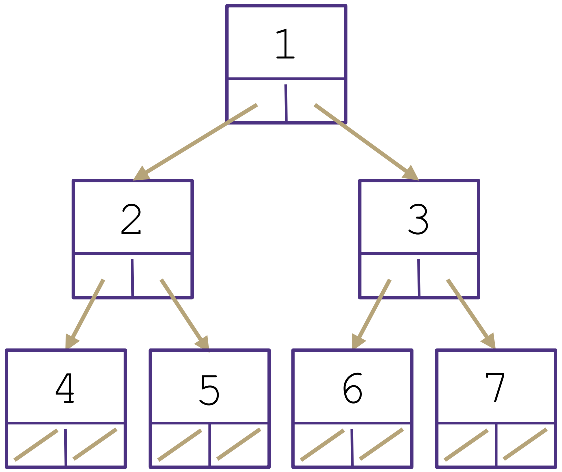
1. Binary Tree: every node has at most 2 children

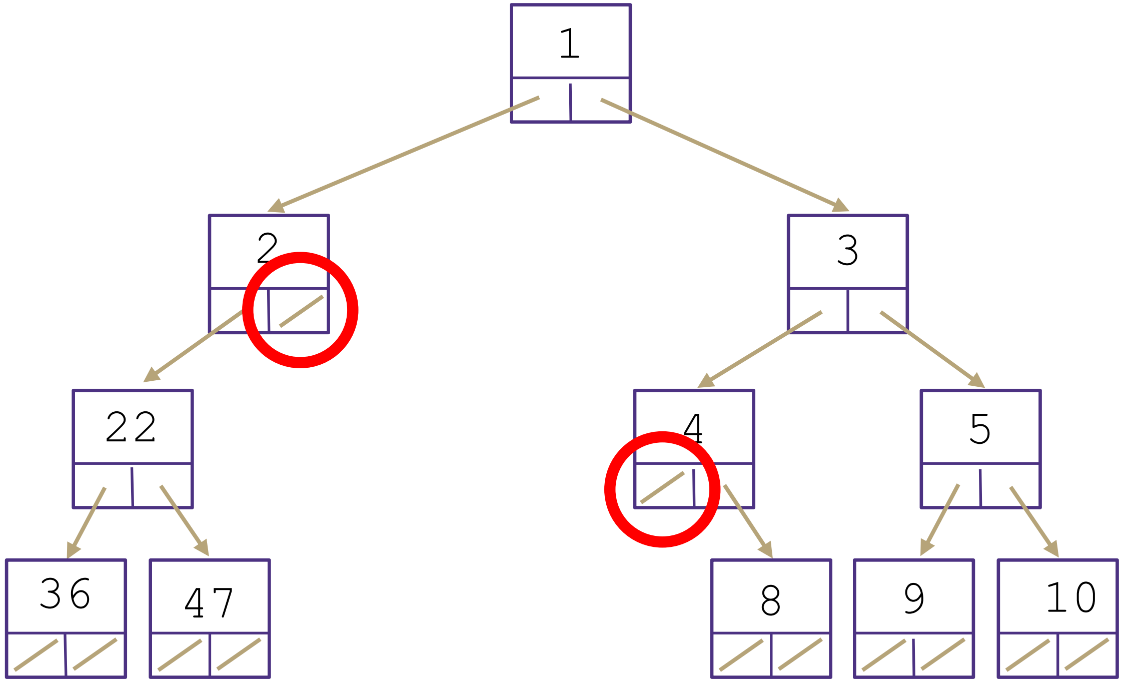
2. Heap: every node is smaller than its child

3. Structure: Each level is “complete” meaning it has no “gaps”

- Heaps are filled up left to right

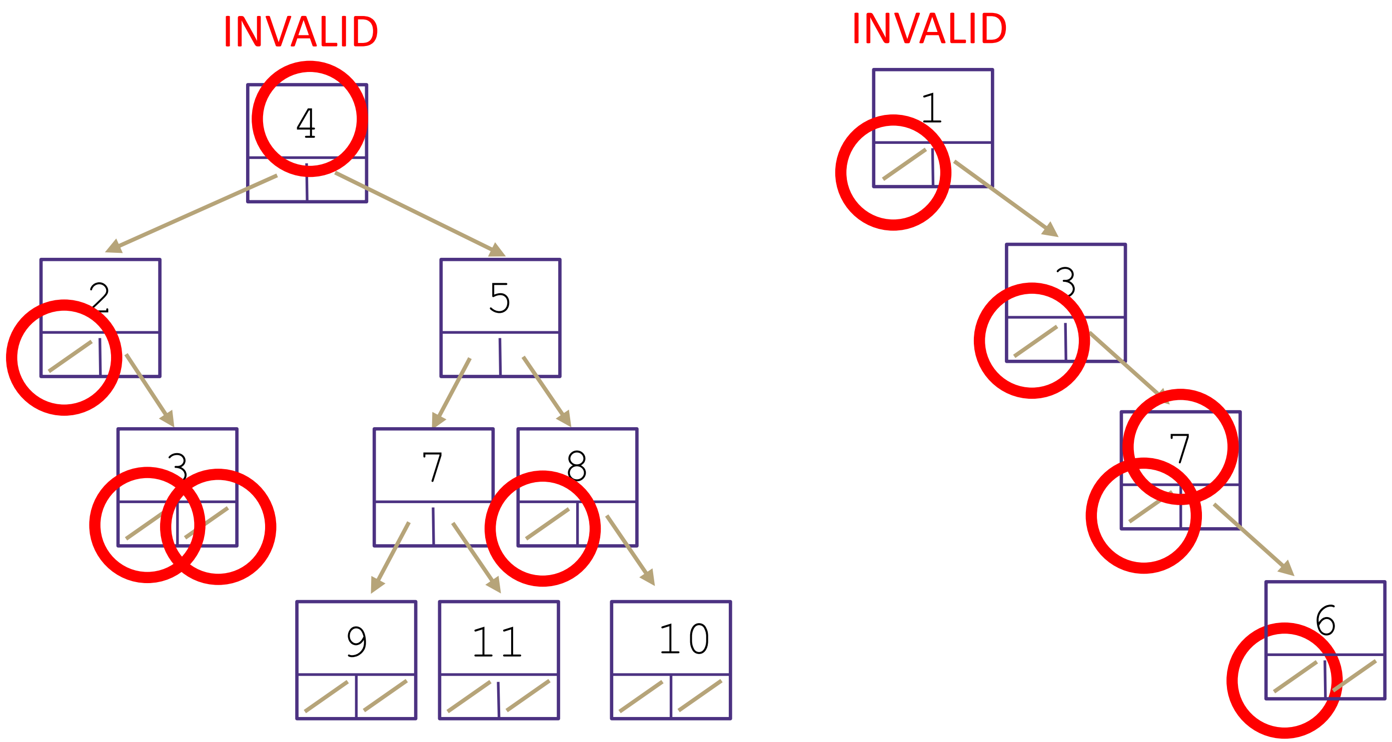




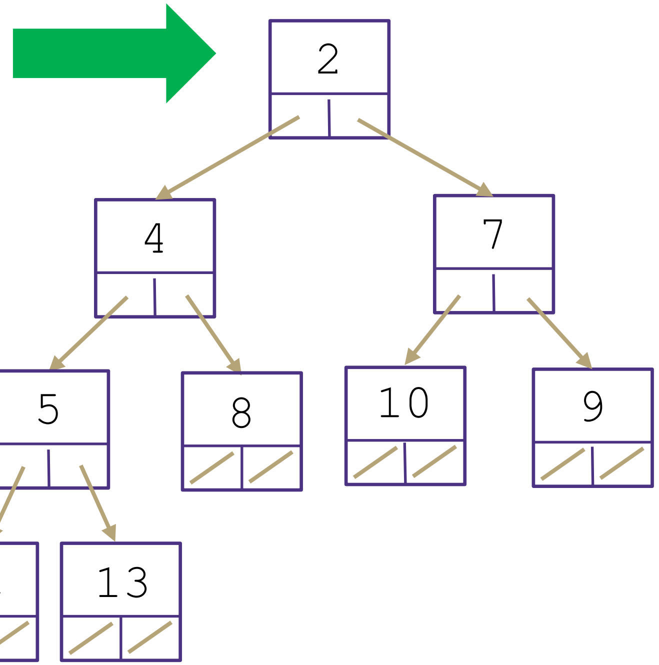


Self Check - Are these valid heaps?

Binary Heap Invariants:
1. Binary Tree
2. Heap
3. Complete

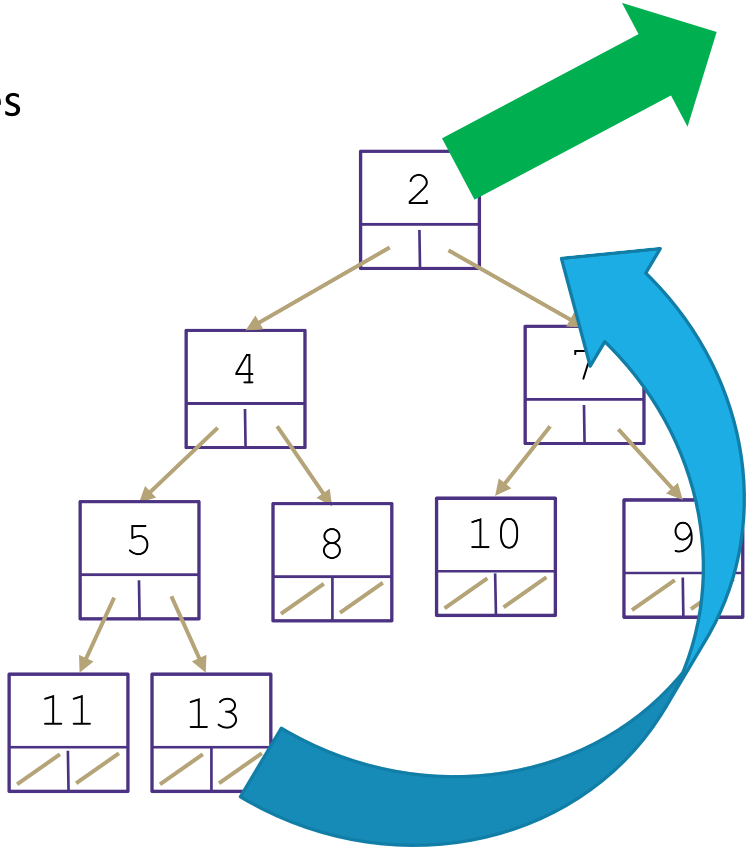


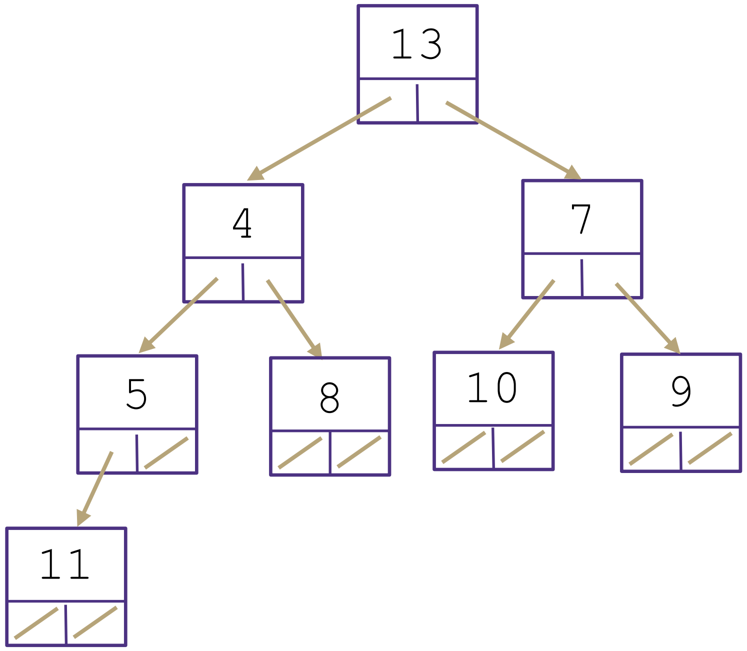
Implementing peekMin()



Implementing removeMin()

Removing overallRoot creates a gap
Replacing with one of its children causes lots of gaps
What node can we replace with overallRoot that wont cause any gaps?

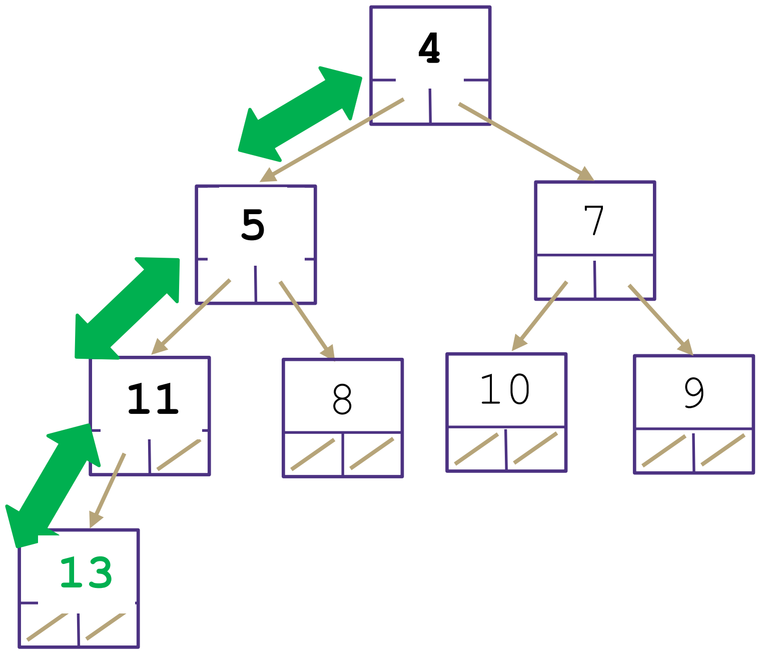




Structure maintained, heap broken

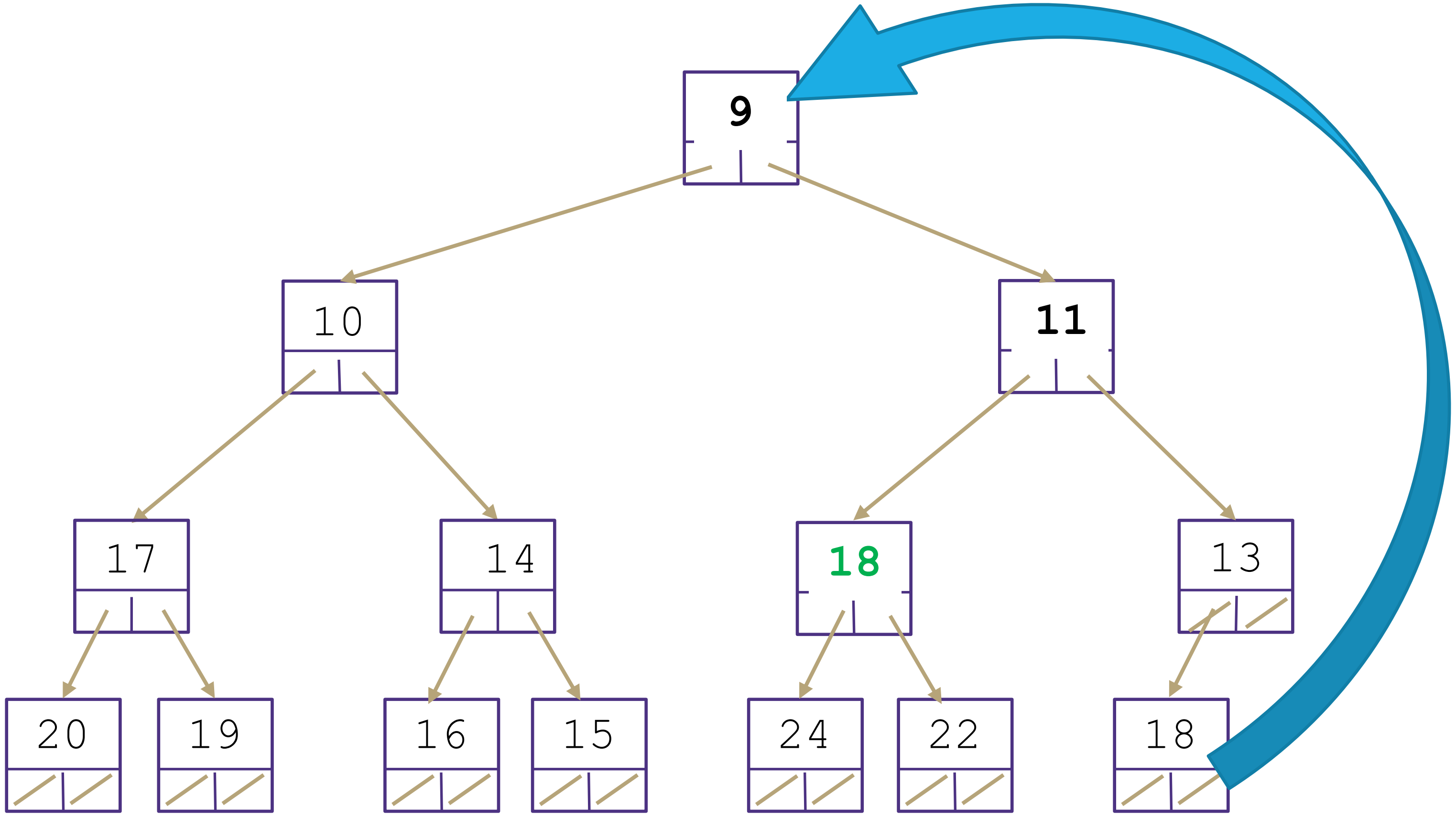
Fixing Heap – percolate down

Recursively swap parent with smallest child




```
percolateDoen(node) {
    while (node.data is bigger than its children) {
        swap data with smaller child
    }
}
```

Self Check – removeMin() on this tree



Implementing insert()

Insert a node to ensure no gaps

Fix heap invariant

percolate **UP**