

COMP2400

Relational Databases

Lecture 28: Binary and B-trees for indexing

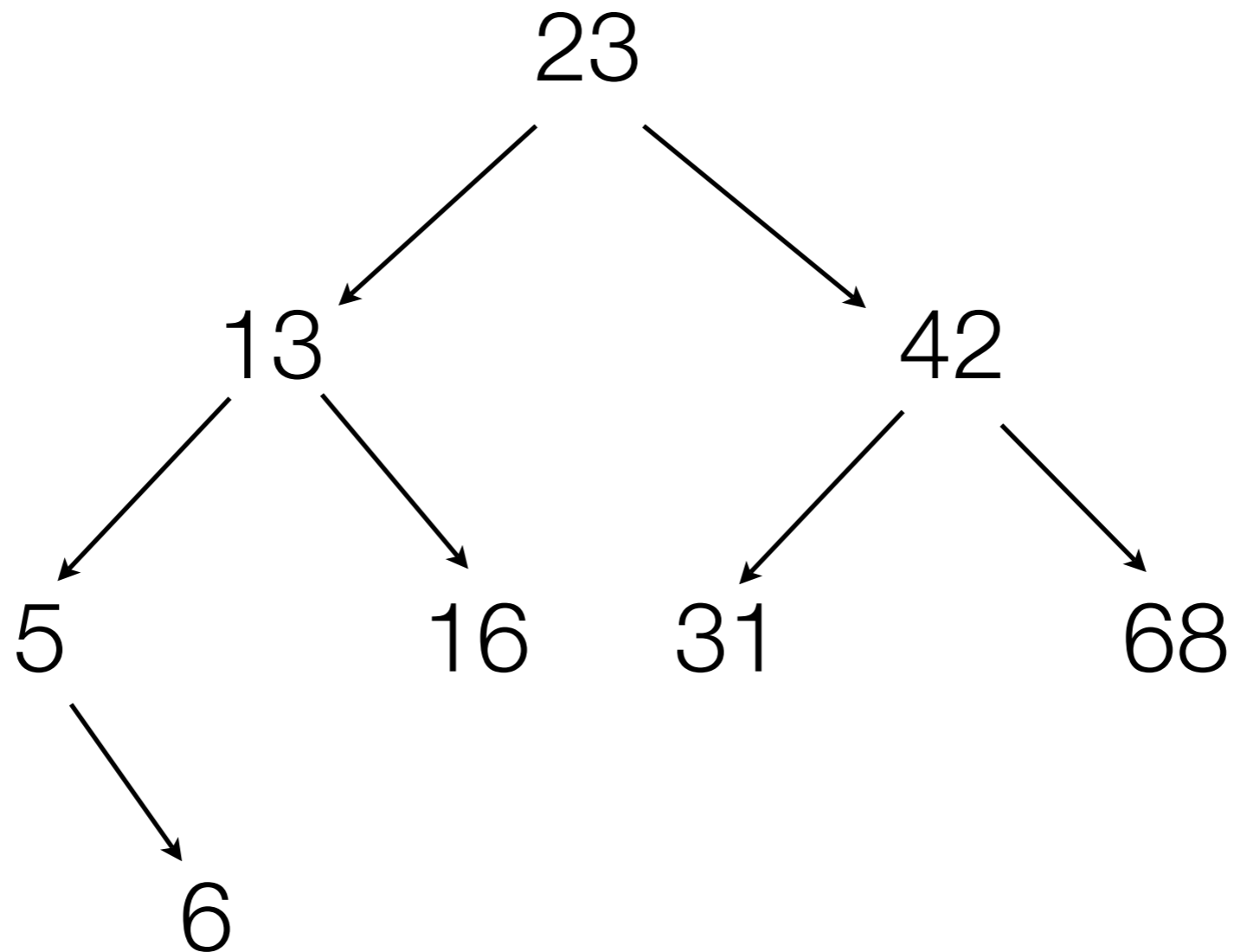
Ben Lippmeier

Australian National University

Semester 2

2008

Binary search trees organise *ordered* values.



Terminology

- We call the ordered value in a node the *key*.
- The node at the top of the tree is called the *root*.
- The sub-trees of a node are its *branches*.
- A node whose sub-trees are both empty is called a *leaf*.
- Nodes which are not leaves are called *internal nodes*.
- The *depth* of a tree is the maximum number of nodes that can be encountered during a downward traversal.

Invariant

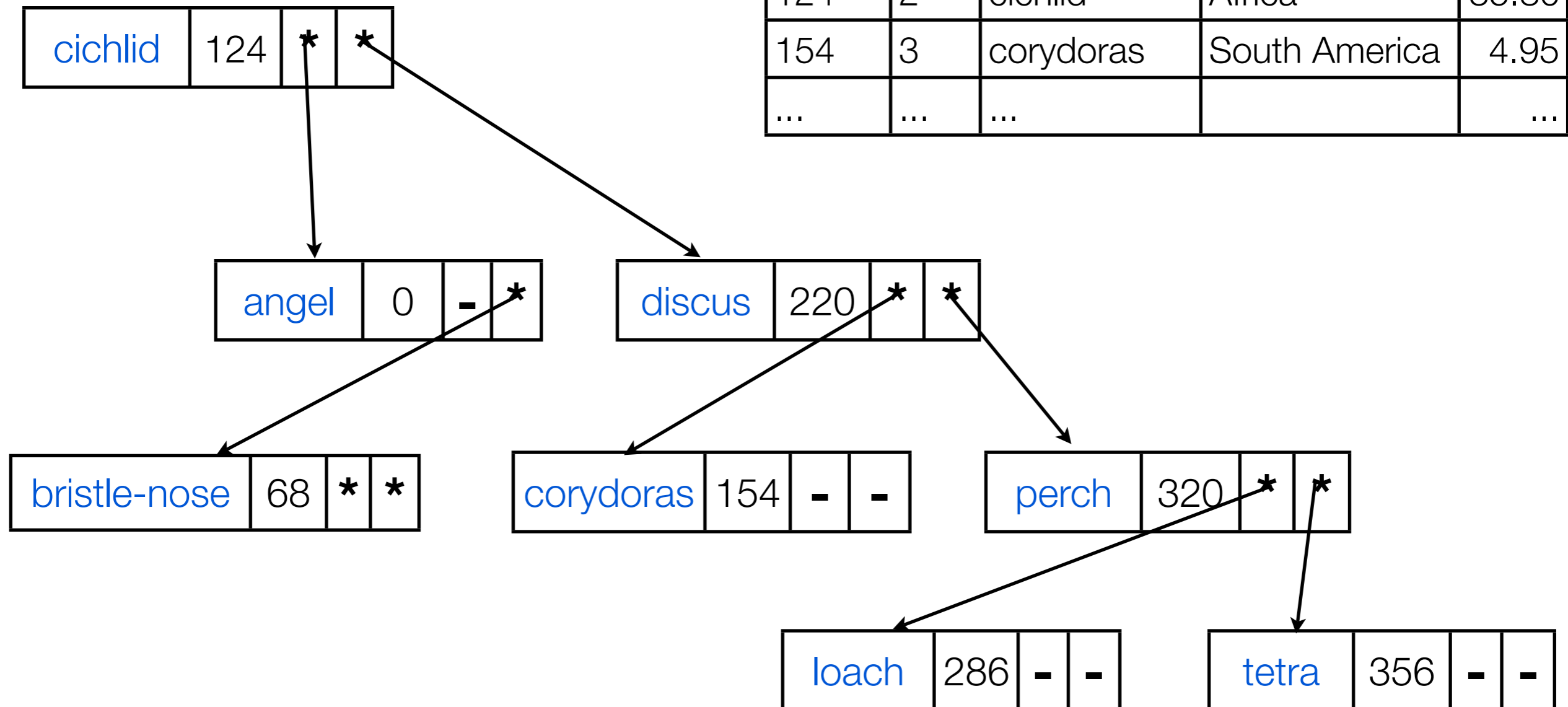
For any give node, call its key **k**.

- The keys in the left branch of that node are always less than **k**
- The trees in the right branch are always more than **k**.

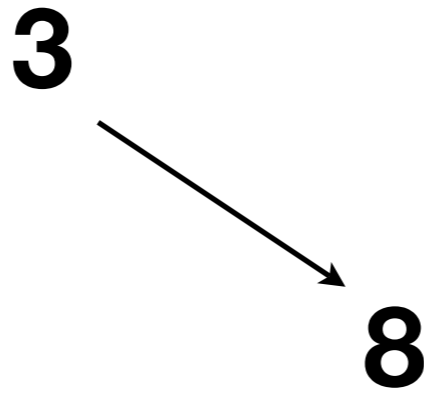
For this to hold the keys must have a total ordering defined for them.

Binary trees as indexes

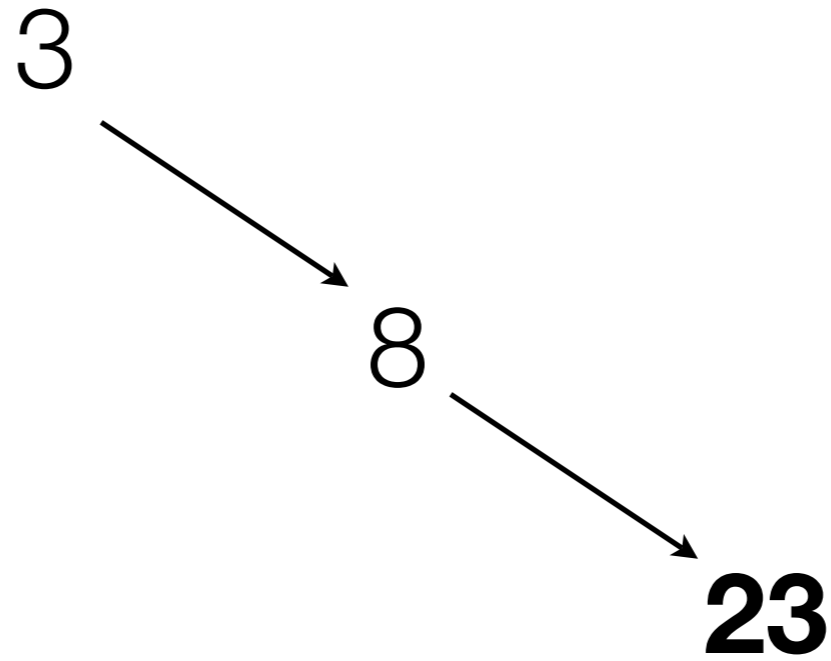
offset	id	species	origin	price
0	0	angel	South America	5.20
68	1	bristle-nose	Asia	15.50
124	2	cichlid	Africa	35.80
154	3	corydoras	South America	4.95
...



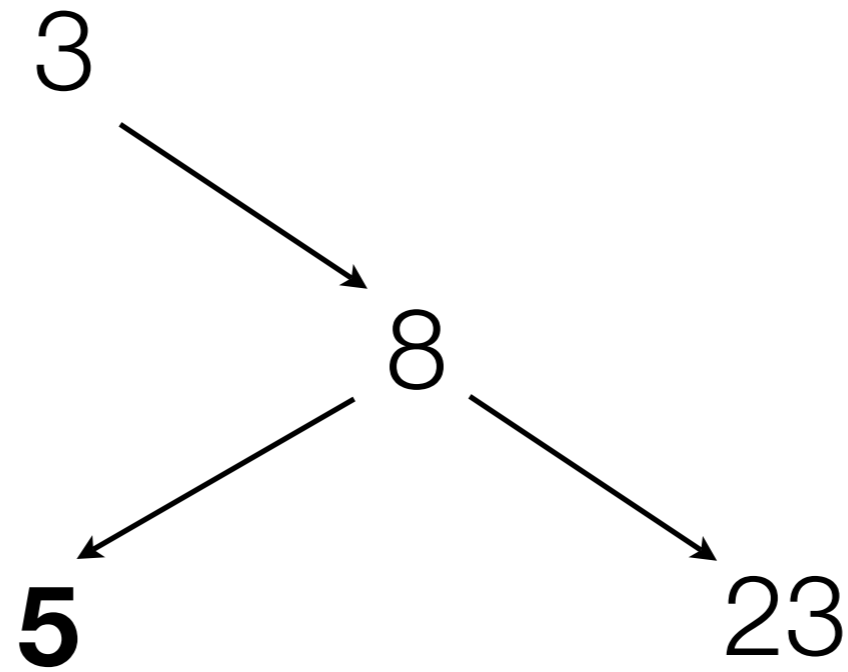
Insert: **3**, **8**, 23, 5, 4, 9, 6, 2, 7, 12



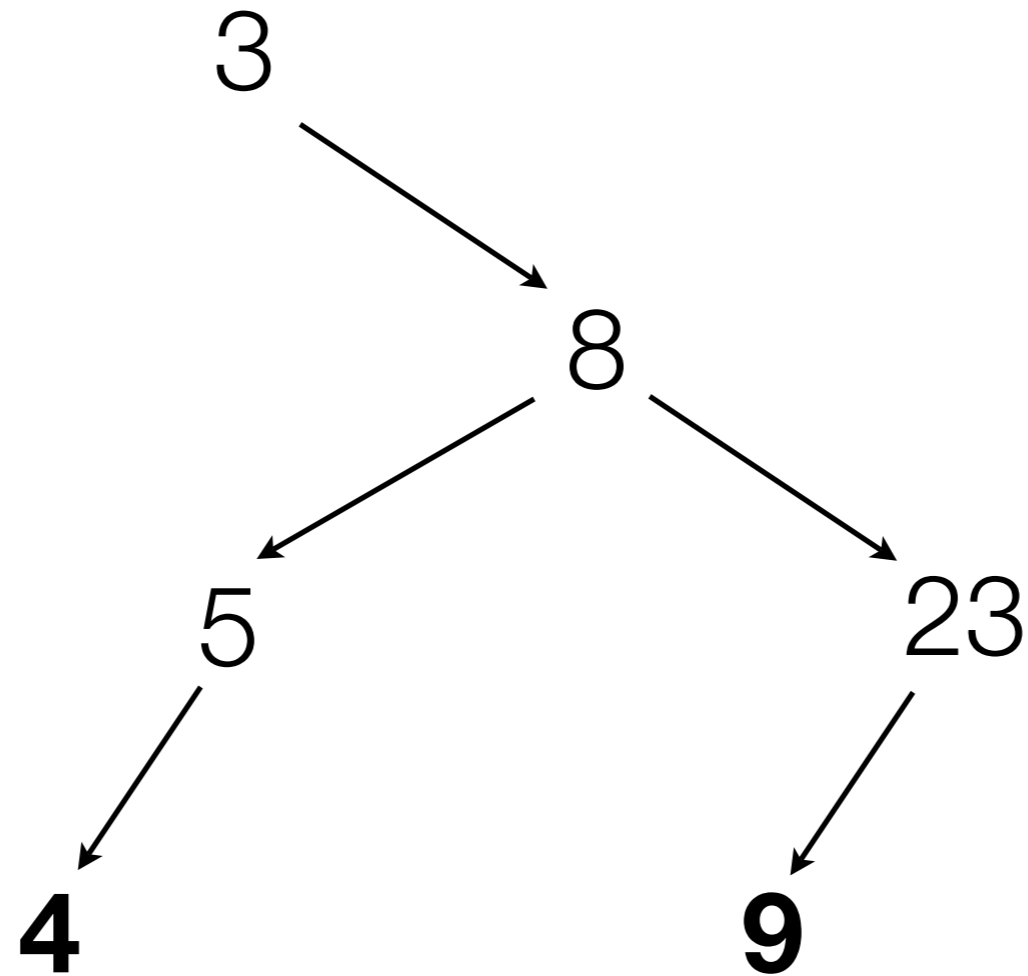
Insert: 3, 8, **23**, 5, 4, 9, 6, 2, 7, 12



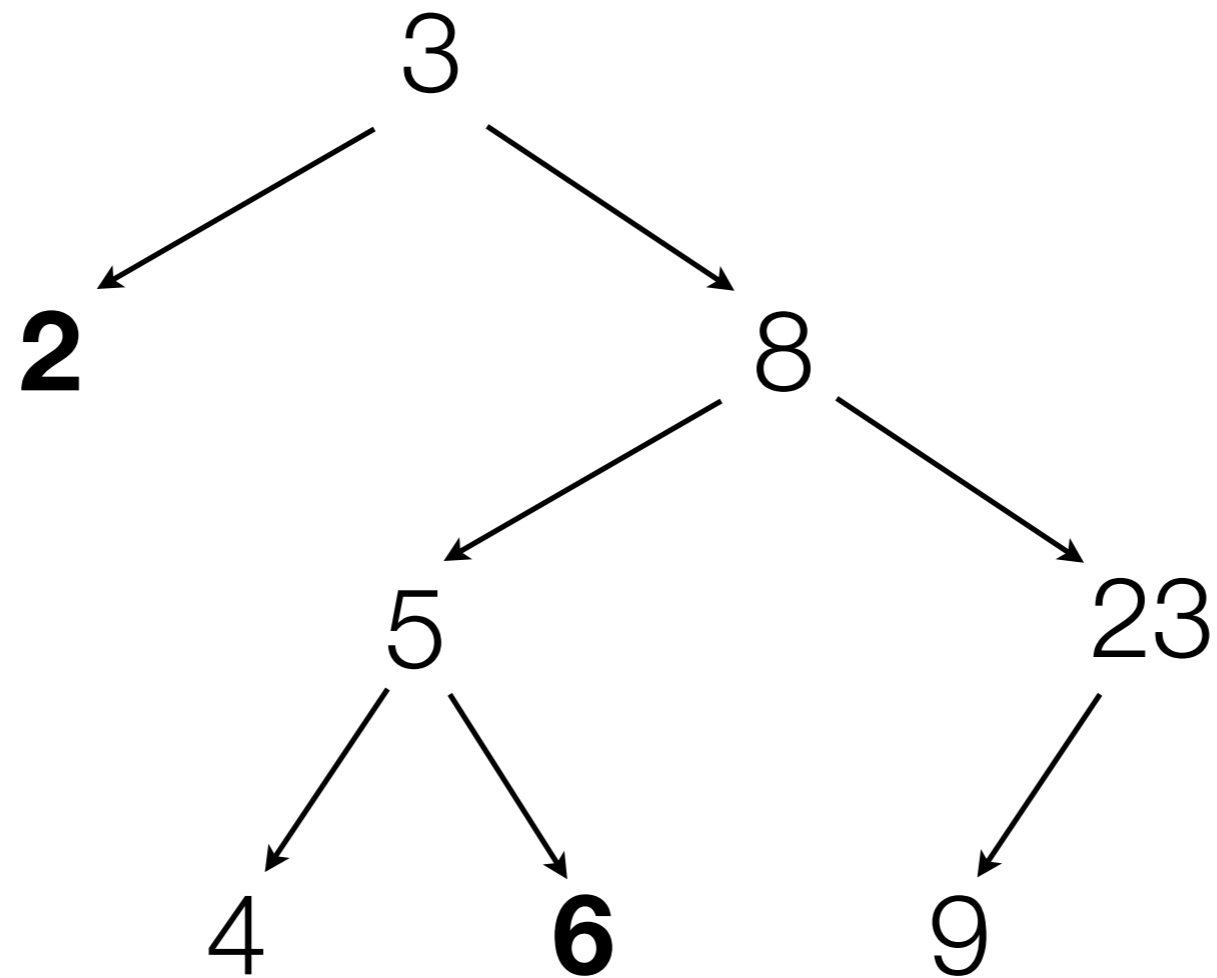
Insert: 3, 8, 23, **5**, 4, 9, 6, 2, 7, 12



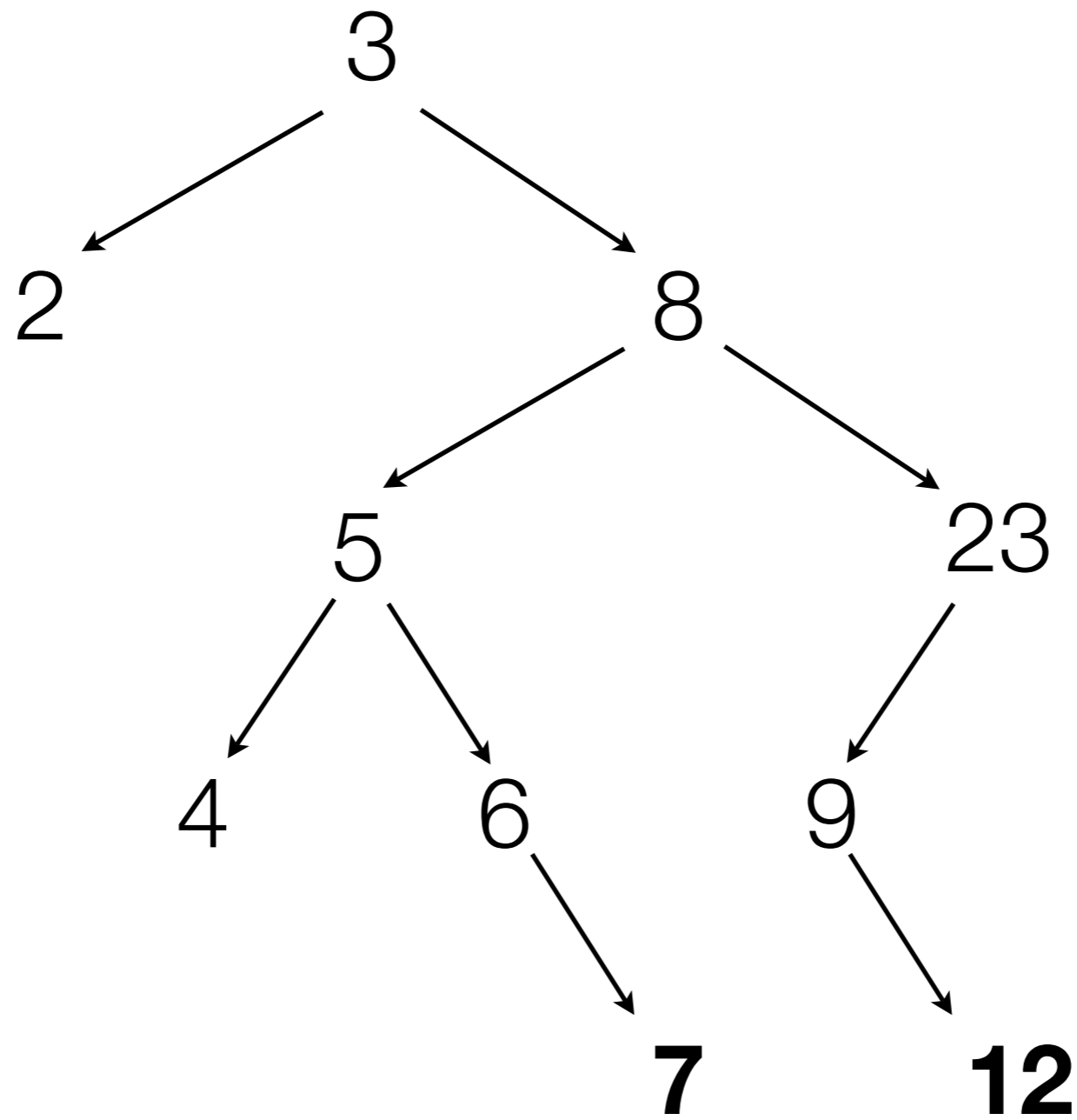
Insert: 3, 8, 23, 5, **4**, **9**, 6, 2, 7, 12



Insert: 3, 8, 23, 5, 4, 9, **6**, **2**, 7, 12

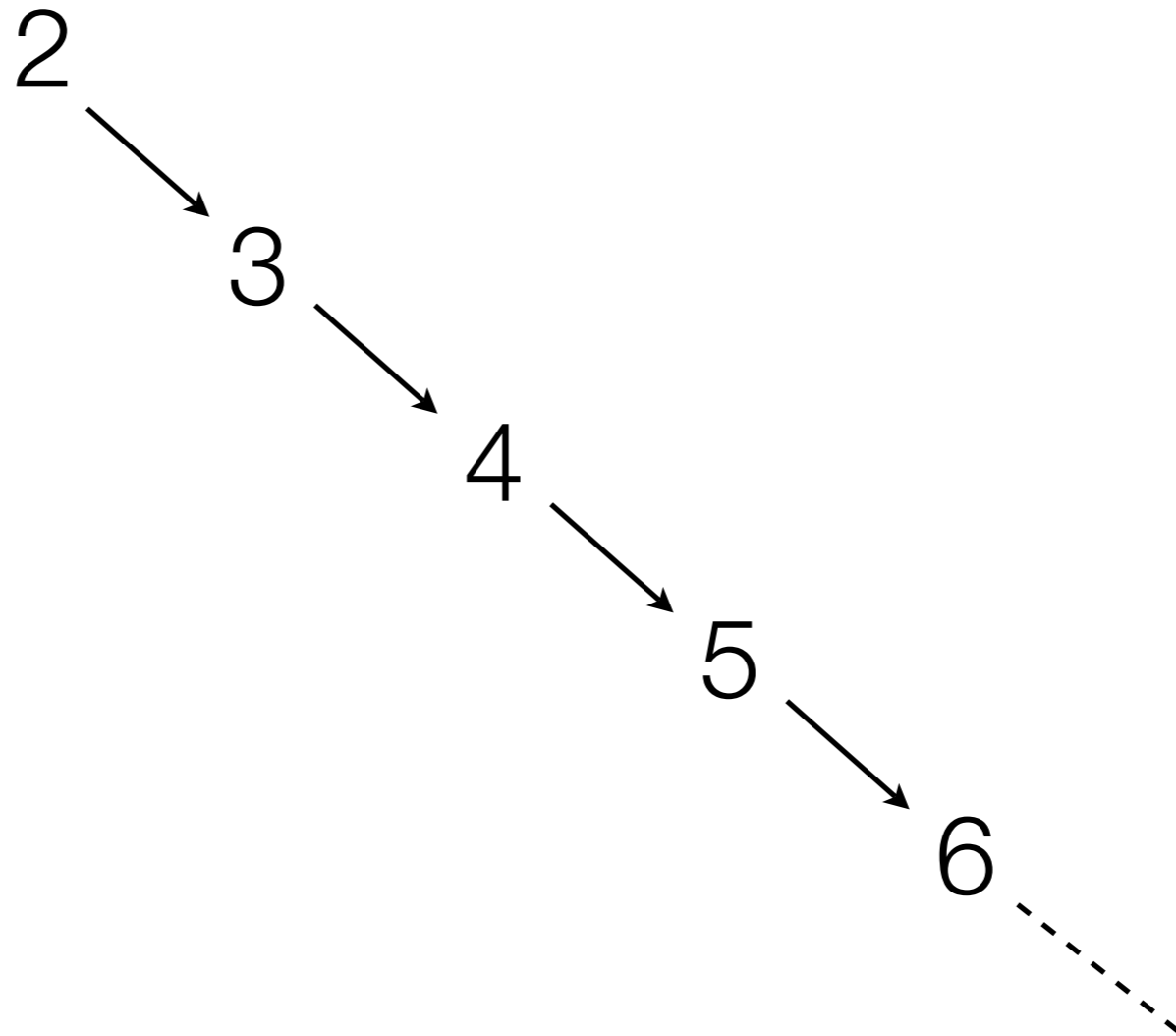


Insert: 3, 8, 23, 5, 4, 9, 6, 2, **7, 12**



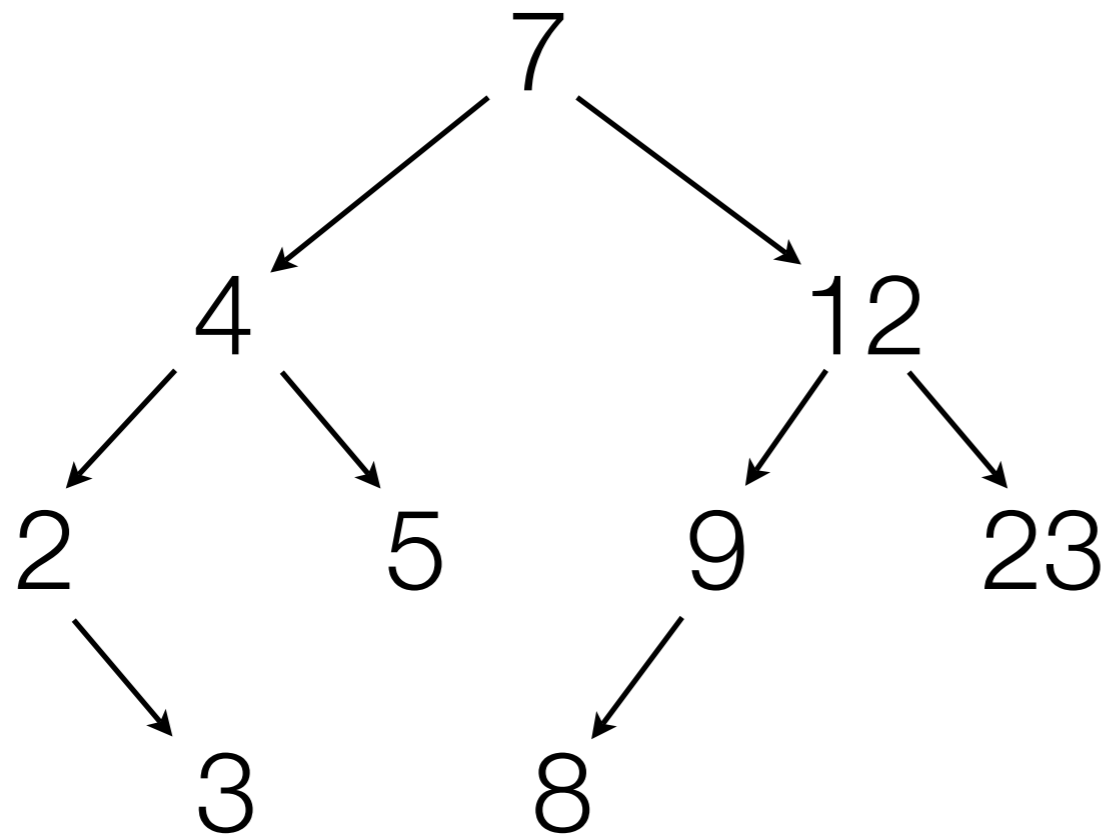
Binary search trees can become *unbalanced*.

- The structure of the tree depends on the order the keys are inserted.
- When keys are inserted in-order, simple binary search trees become **lists**.



For efficient lookup, the tree must be balanced.

- In a balanced binary tree: for each node, the depth of the left branch must be no greater than one plus the depth of the right branch, and *visa versa*.
- For a balanced tree with n nodes, we need a maximum of $\text{ceil}(\log_2(n))$ comparisons to find a key. Compare with $n/2$ for linear search.



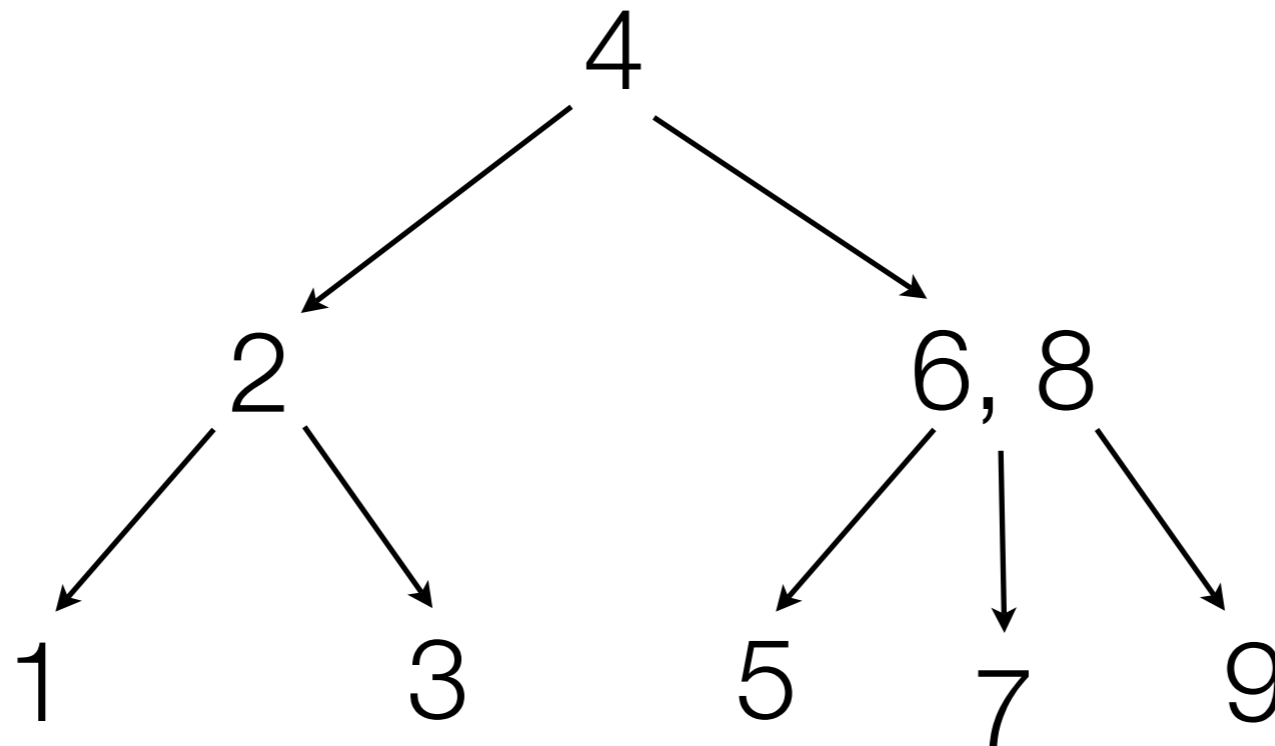
n	= 9
$\log_2(n)$	≈ 3.17
$\text{ceil}(\log_2(n))$	= 4
	= <i>depth of tree</i>

Flavors of search trees.

- Red/black trees.
 - add extra invariants to keep the tree semi-balanced.
- AVL trees
 - restructure an unbalanced tree to restore balance.
 - tree rotations change the structure of the tree, but maintain the order invariant on the keys.
- B-trees
 - allow more than one key, and two branches per node.
 - can make efficient use of block based stores (ie, hard disk drives).
 - variants are commonly used in the implementation of DBMSs.
 - also used in file systems such as ext3, the standard Linux file system.

(2,3) trees allow 2 or 3 branches per node.

- A (2,3) tree is once instance of the general B-tree structure.
- All keys in the central branch of a 3-node must be greater than the keys in the left branch and less than the keys in the right branch.
- If there are multiple keys in a node they must also be in-order.



Insert: **1**, 2, 3, 4, 5, 6, 7, 8, 9

1

The first one is easy!

Insert: 1, **2**, 3, 4, 5, 6, 7, 8, 9

1, 2

We can have two keys per node
so just add 2 to the root.

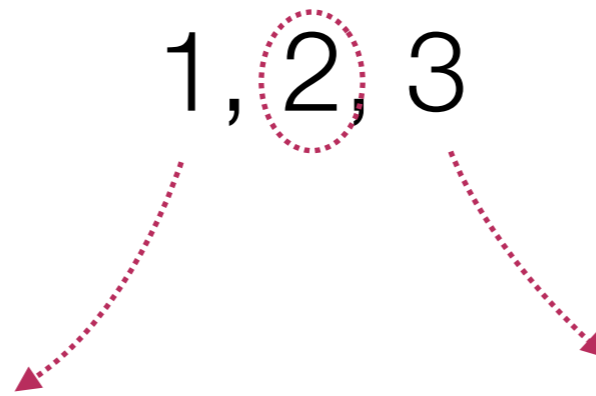
Insert: 1, 2, **3**, 4, 5, 6, 7, 8, 9

1, 2, **3**

After adding the third we have too many keys in the root node.

Insert: 1, 2, 3, 4, 5, 6, 7, 8, 9

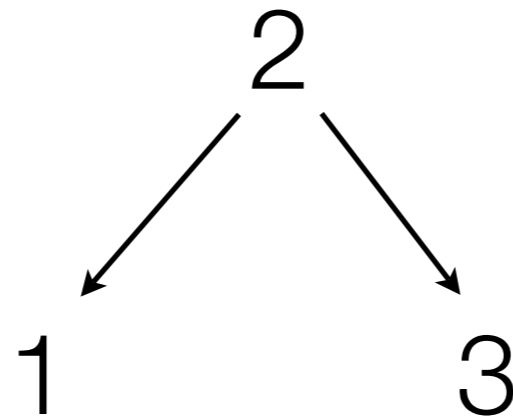
(split)



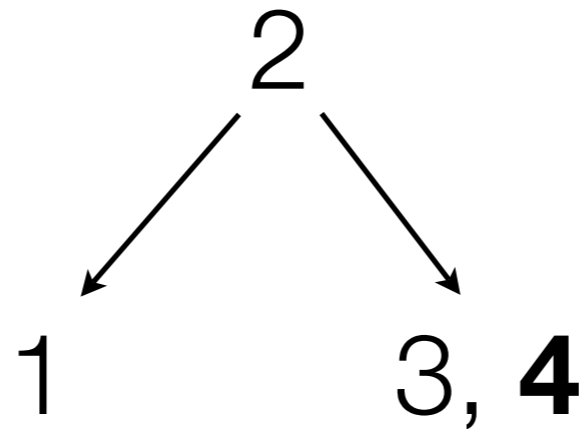
Split this node into parts to restore the invariant.
The median/middle key becomes the parent.

Insert: 1, 2, 3, 4, 5, 6, 7, 8, 9

(split)

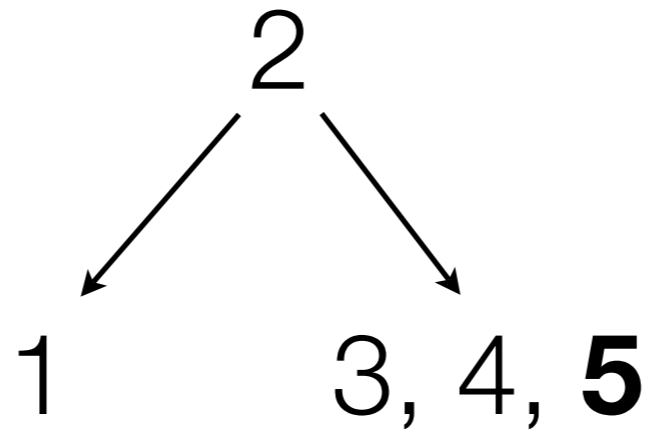


Insert: 1, 2, 3, **4**, 5, 6, 7, 8, 9

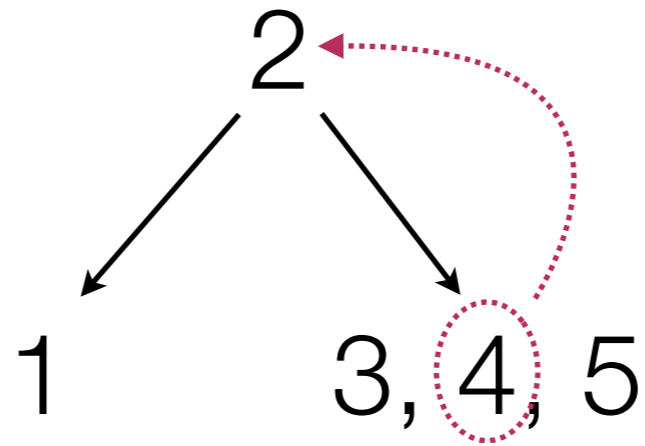


Except for the root
we always insert keys into leaf nodes.

Insert: 1, 2, 3, 4, **5**, 6, 7, 8, 9



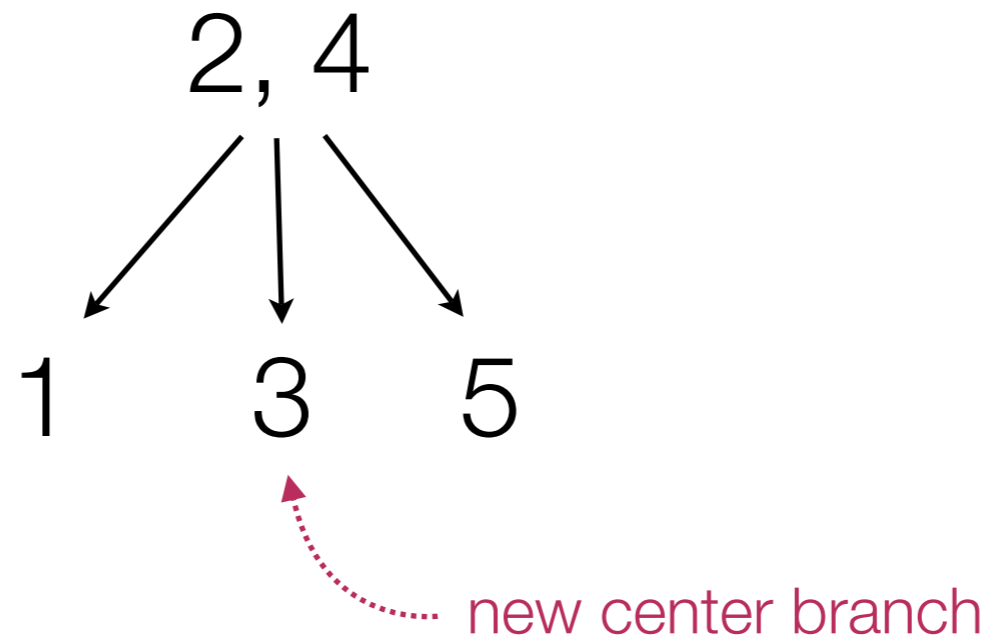
Insert: 1, 2, 3, 4, 5, 6, 7, 8, 9



When splitting a non-root node,
add the median key to its parent.

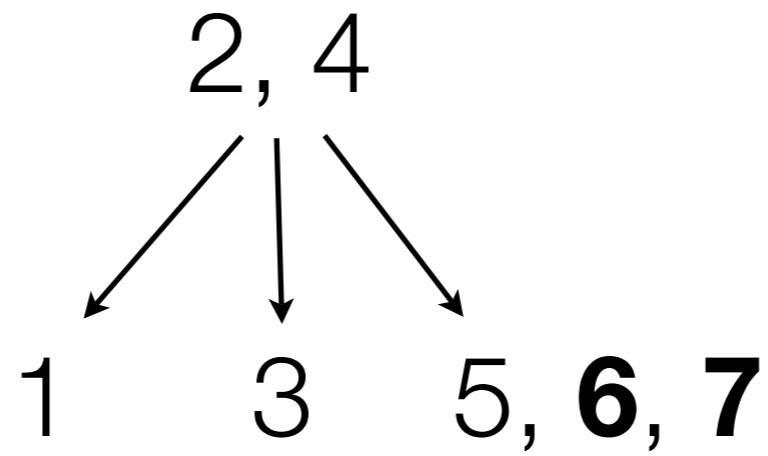
Insert: 1, 2, 3, 4, 5, 6, 7, 8, 9

(split)

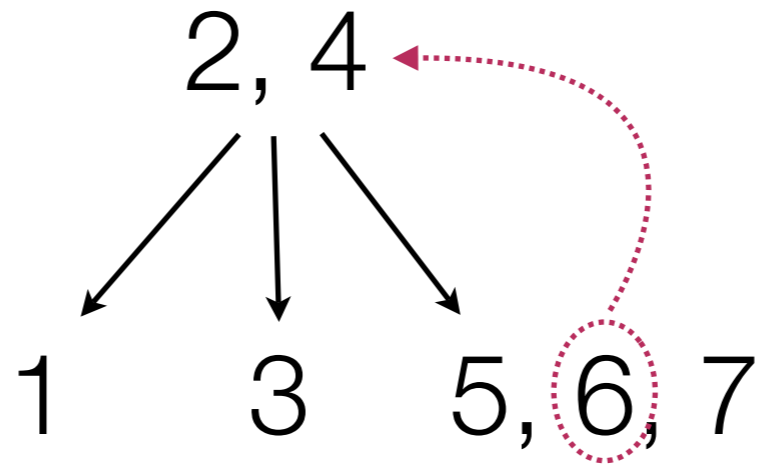


As the node being split was on the *left* of its parent,
the keys *less than* the median become
the new center branch

Insert: 1, 2, 3, 4, 5, **6**, **7**, 8, 9



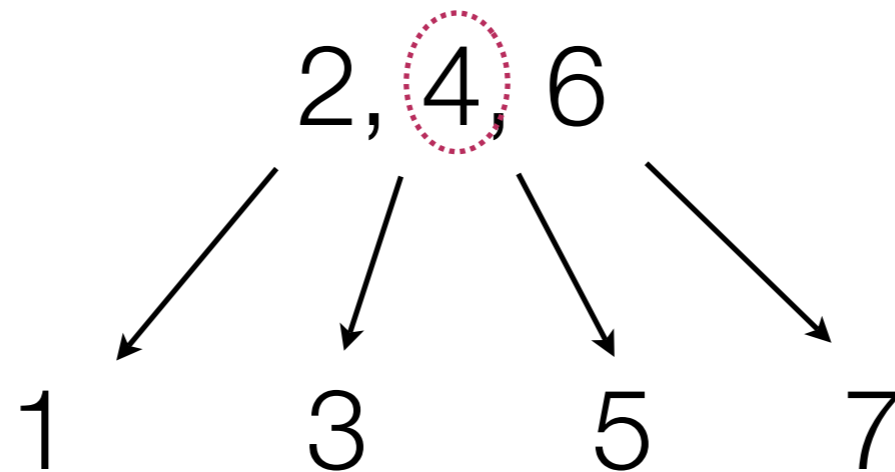
Insert: 1, 2, 3, 4, 5, 6, 7, 8, 9



Insert median into parent and split the overfull node.

Insert: 1, 2, 3, 4, 5, 6, 7, 8, 9

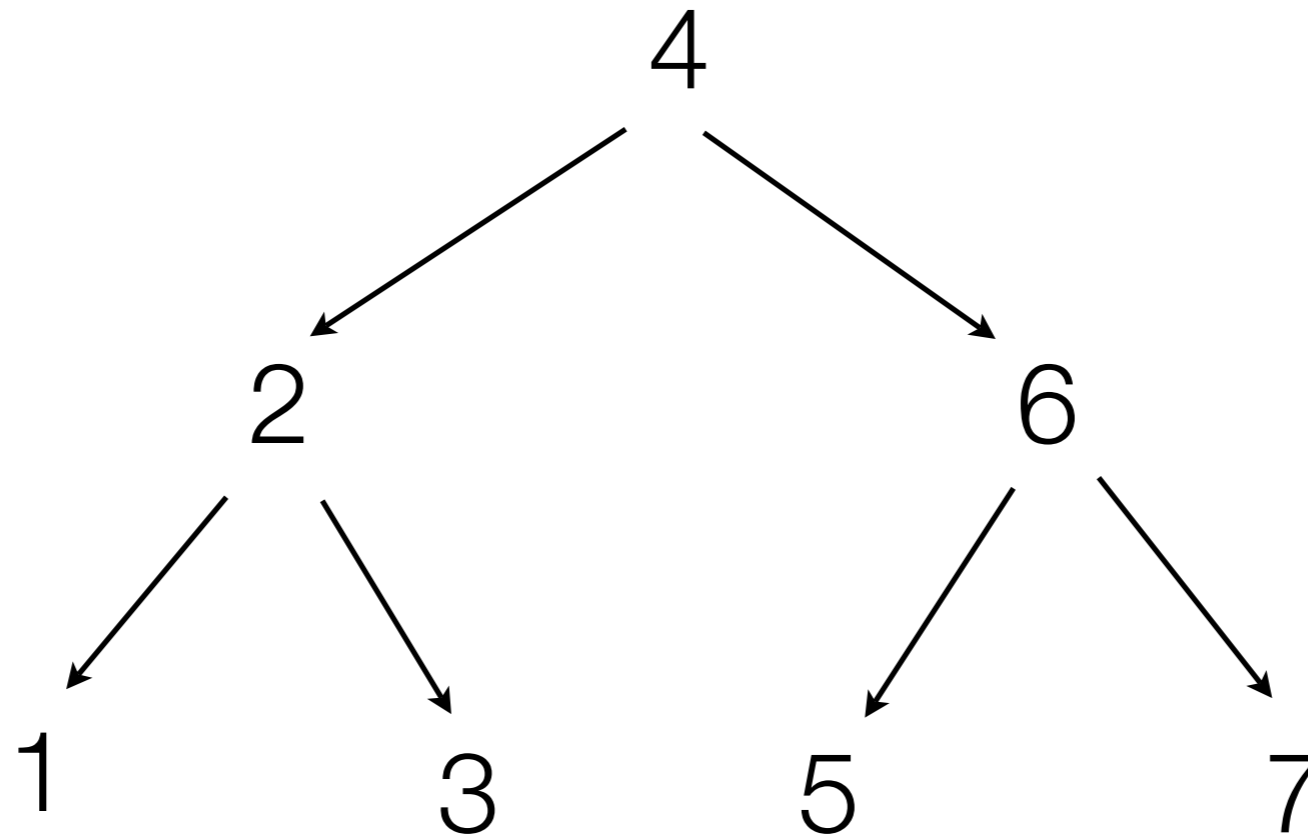
(split)



When splitting a 3-node, the left two and right two branches follow their respective keys...

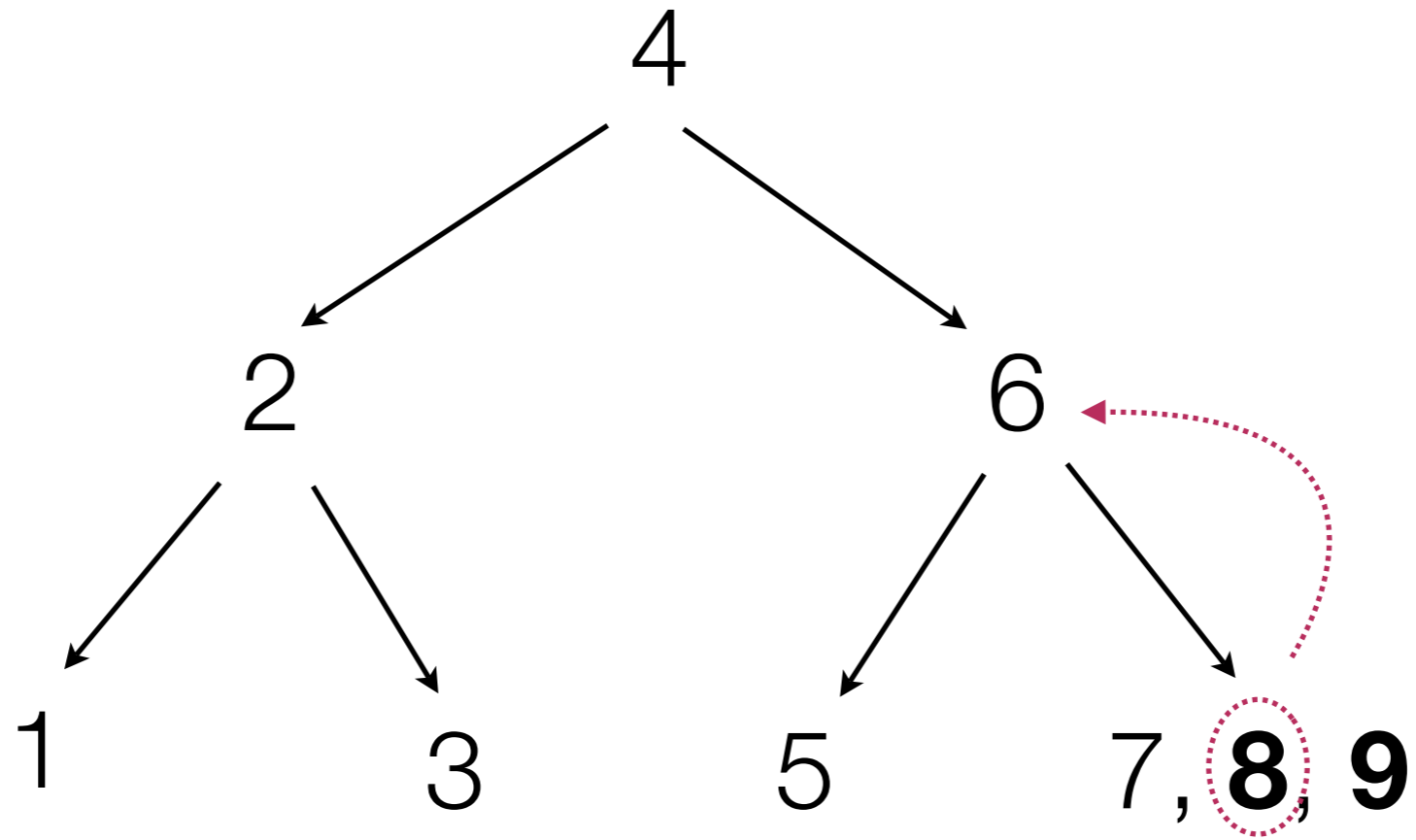
Insert: 1, 2, 3, 4, 5, 6, 7, 8, 9

(split)



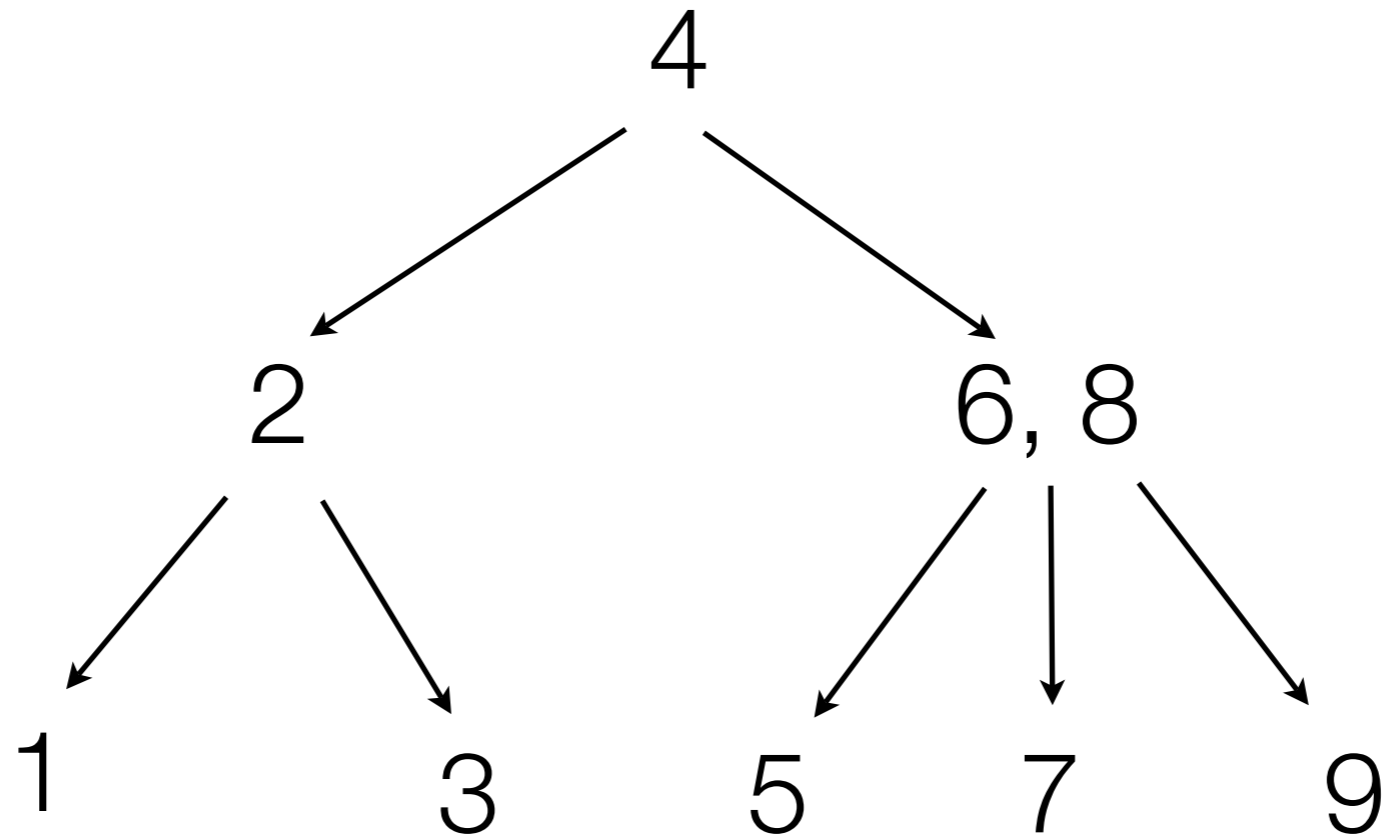
Notice how median keys rise to the top of the structure.
This is what keeps the tree balanced.

Insert: 1, 2, 3, 4, 5, 6, 7, **8**, **9**



Insert: 1, 2, 3, 4, 5, 6, 7, 8, 9

(split)



High order B-Trees for efficient disk access.

- The order of a B-tree is the maximum number of keys that can appear in a node before it must be split.
- If a node has m keys it can have up to $(m-1)$ branches.
- If a B-tree is to be stored on disk we can place each node in its own block and store as many keys as will fit in the block.
- Having many keys per node reduces the depth of the tree and the number of block accesses required when searching it.

High order B-Trees for efficient disk access.

- When inserting a new key into the tree, the keys *within* the target node must be reordered to maintain the intra-node invariant.
- If the tree is stored on disk, the time to reorder the keys in a block/node is much less than the time to access it.

In this case having many keys per node pays off.

- However, if the tree is stored in RAM it may be more efficient to stick to binary trees.

In this case we can insert a new node in the tree without touching as much of the surrounding data.

Very high performance applications should consider the configuration of the processors data cache when choosing a tree layout.

Insert: **3, 8, 23**, 5, 4, 9, 6, 2, 7, 12

3, 8, 23

This is an example of a tree of order 4.
We can insert the first three keys directly.

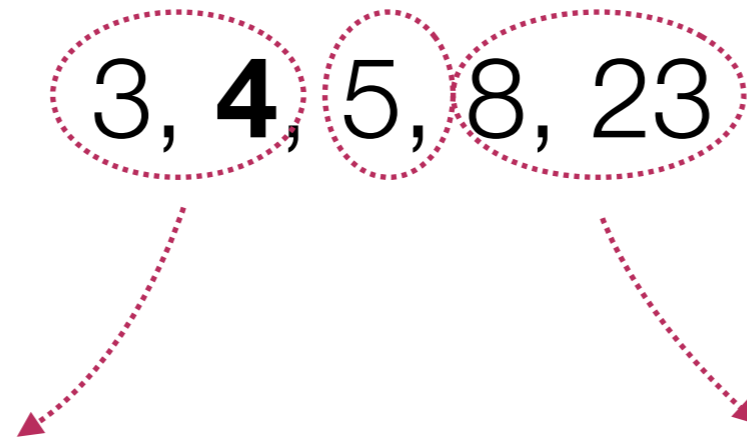
Insert: 3, 8, 23, **5**, 4, 9, 6, 2, 7, 12

3, **5**, 8, 23



To insert 5 into this node, we must shift 8 and 23 to the right to make room and maintain the ordering.

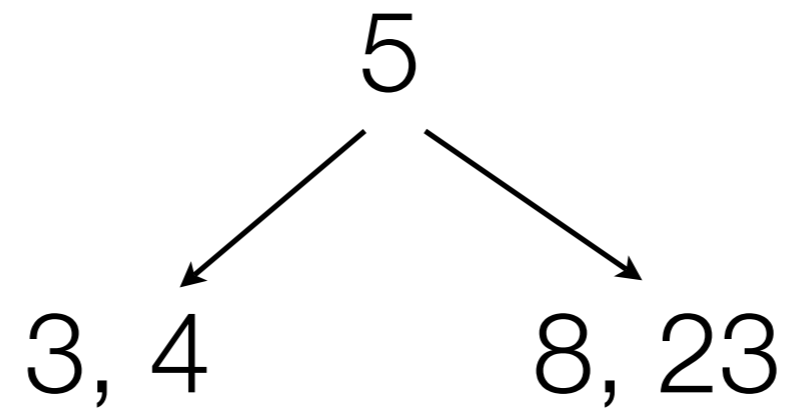
Insert: 3, 8, 23, 5, **4**, 9, 6, 2, 7, 12



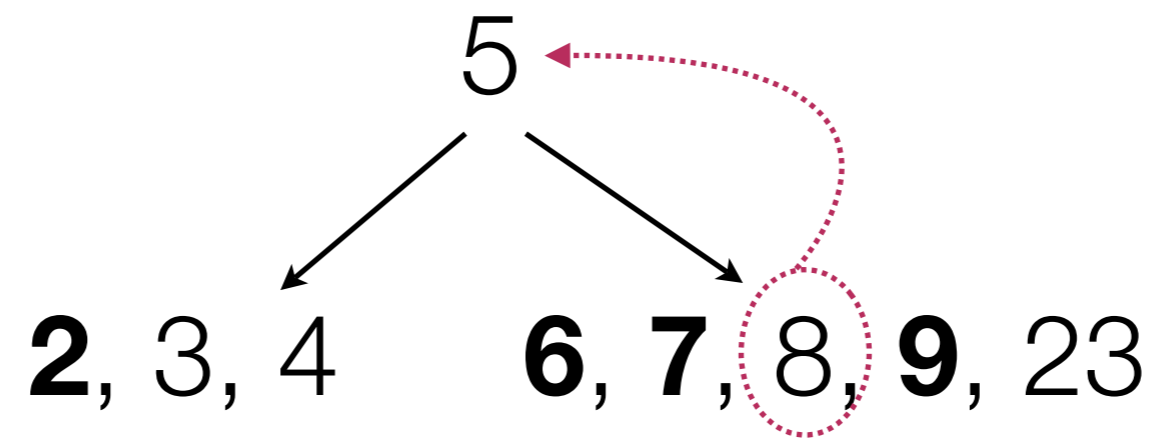
Now the number of keys in the root node is more than the order, so we must split it.

Insert: 3, 8, 23, 5, 4, 9, 6, 2, 7, 12

(split)

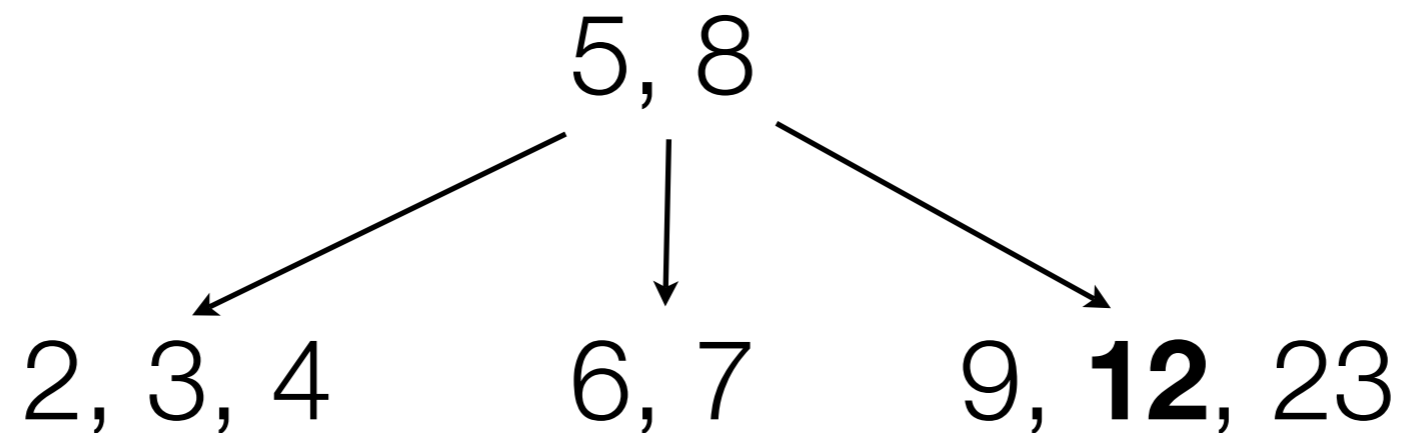


Insert: 3, 8, 23, 5, 4, **9**, **6**, **2**, **7**, 12



Insert: 3, 8, 23, 5, 4, 9, 6, 2, 7, **12**

(split)



The order invariant holds at all times!

For a given key k

- all the keys to its left are less than k
- all the keys to its right are more than k

If this is not true then you have made a mistake.

Deleting nodes from B-trees

- When we delete keys from a tree it can become unbalanced.
- A common invariant is to require each non-root node to have a minimum of $(order / 2)$ keys.

After deletion the tree can be restructured to maintain this invariant.

- This form of unbalancing is less severe than the unbalancing of binary trees due to in-order insert.
- For large databases, perhaps rebalance data structures nightly in times of low activity. This helps reduce peak load on production databases.
- In PostgreSQL, “deleted” tuples are simply flagged and left in the database. The DB maintainer must run “vacuum” to actually reclaim the space.

B+ Trees

- Only store record pointers in leaf nodes.
Keys in internal nodes are duplicated in the leaves.
- We can also add a *chaining* pointer for efficient linear search.

When we evaluate `SELECT * FROM table`, we can now load all the data by tracing along the bottom level of the tree, ignoring the higher structure.

