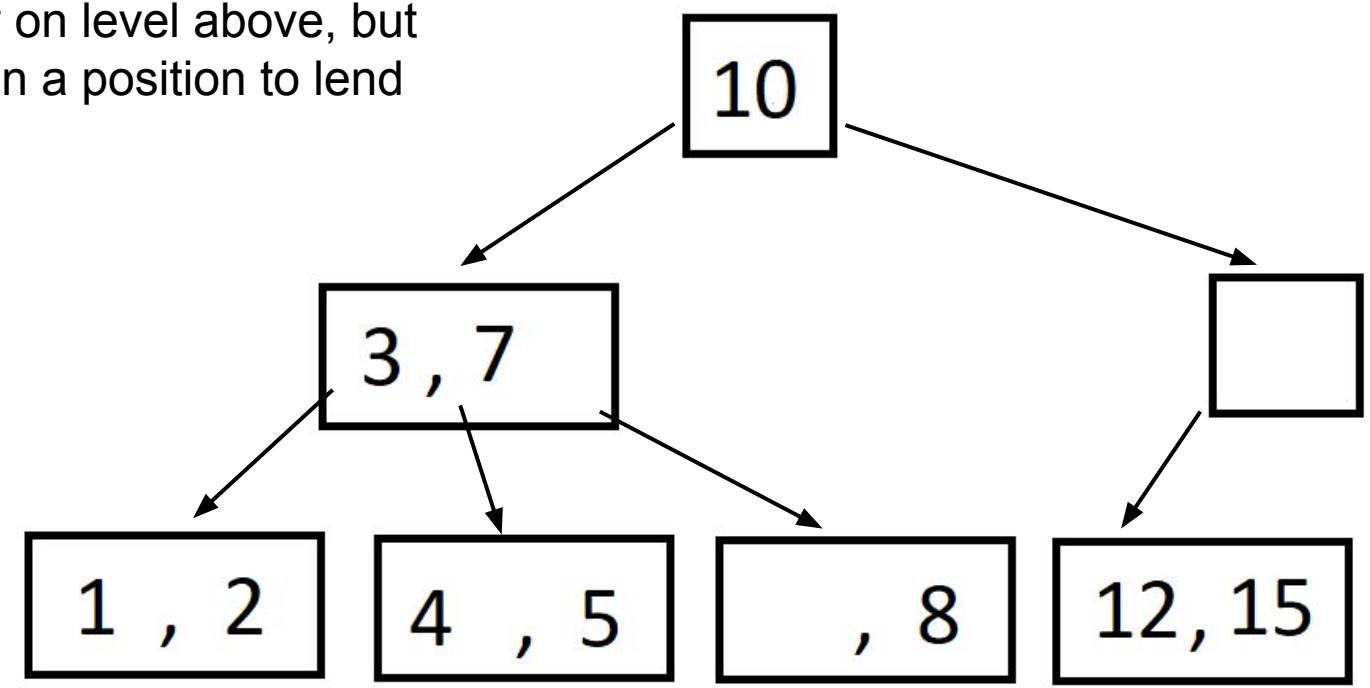
Exercise 4 Sol. 4/6

15 coming down creates a underflow on level above, but sibling is in a position to lend key



DBMS Tutorial 05.12.2018

Topics

B+ Trees

B Trees

B+ Trees

While one or two levels of index are often very helpful in speeding up queries, there is a more general structure that is commonly used in commercial systems.

This family of data structures is called B-trees, and the particular variant that is most often used is known as a B+ tree.

In essence:

- B-trees automatically maintain as many levels of index as is appropriate for the size of the file being indexed.
- B-trees manage the space on the blocks they use so that every block is between half used and completely full.

A B-tree organizes its blocks into a tree that is balanced, meaning that all paths from the root to a leaf have the same length.

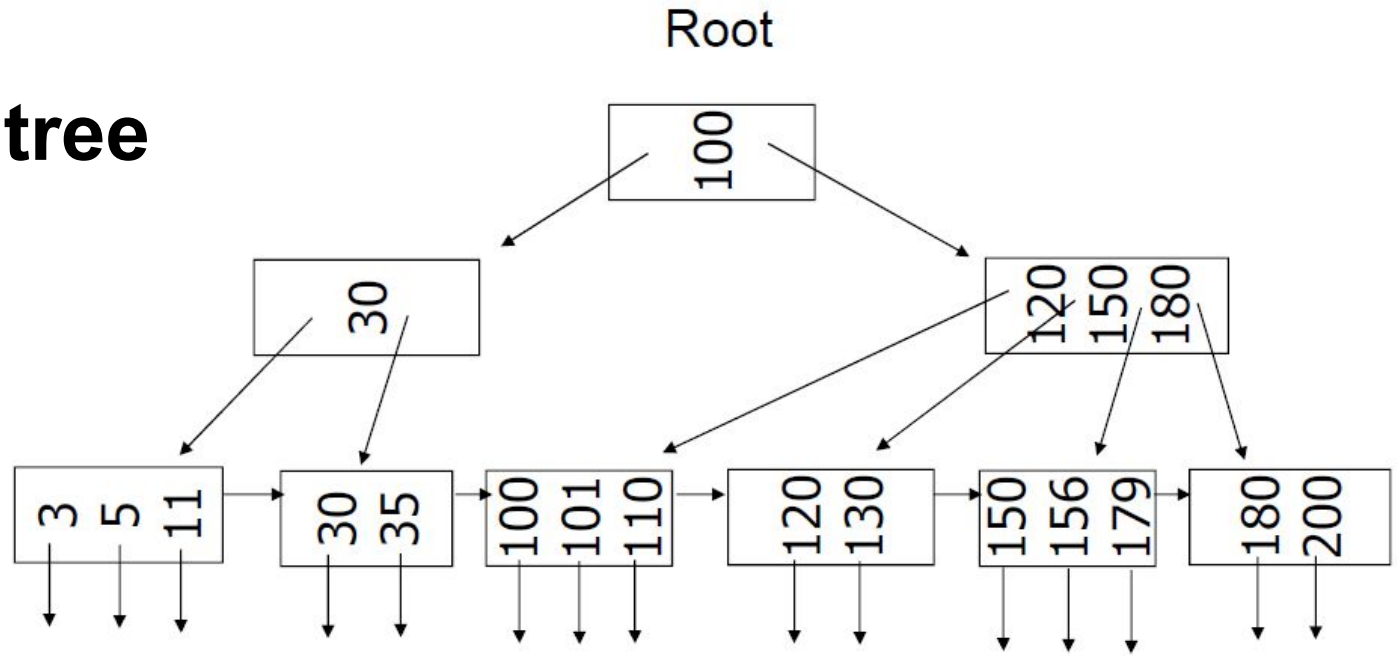
Typically, there are three layers in a B-tree: the root, an intermediate layer, and leaves, but any number of layers is possible.

n : Order of a B Tree

- determines the layout of all blocks of the B-tree
- each block will have space for n search-key values and n + 1 pointers
- n chosen to be as large as will allow n+1 pointers and n keys to fit in one block

Nodes : Root, Intermediate, Leaf

B+ tree

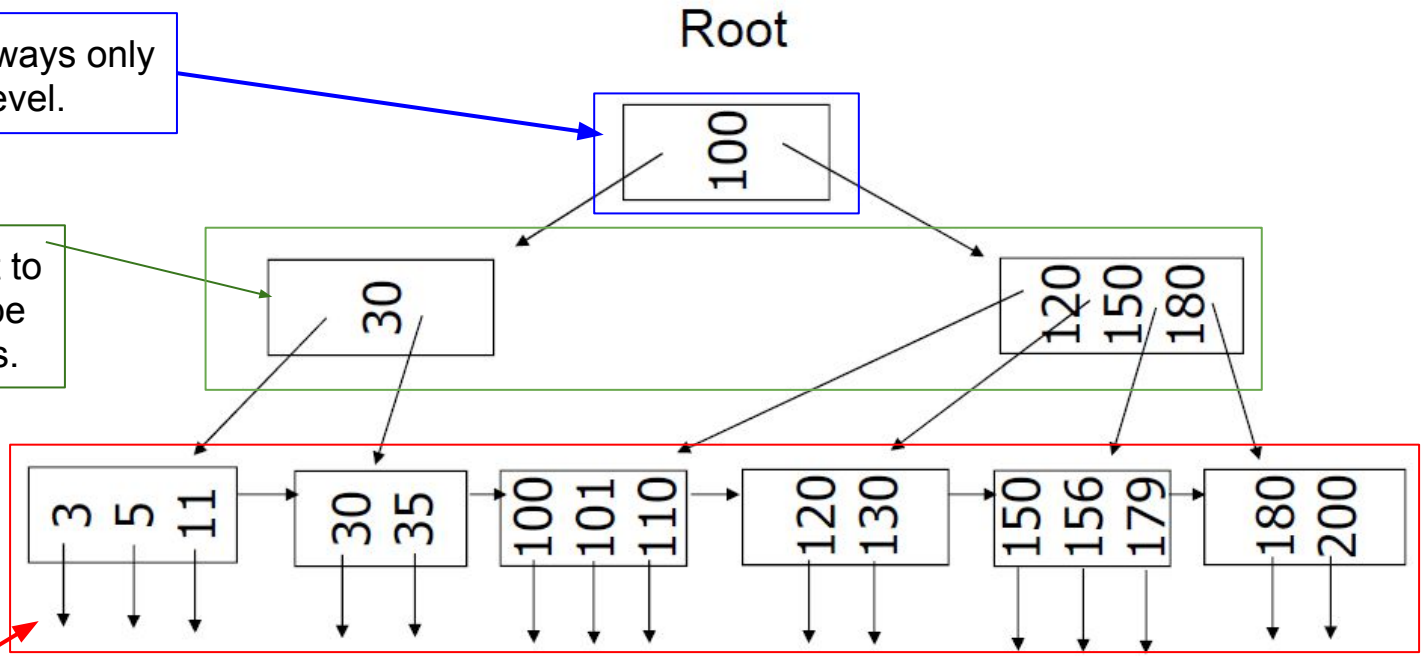


Henceforth whatever rules we discuss will be for B+ trees. We shall cover the variant B-tree towards the end.

Nodes : Root, Intermediate, Leaf

The top level node, always only one node at the root level.

Intermediate levels point to child nodes. There can be many intermediate levels.



Leaf nodes: at the last level, contain pointers to records (sequence pointers.)

Keys sorted from left to right.

Some Rules : Leaf Nodes

- The keys in leaf nodes are copies of keys from the data file. These keys are distributed among the leaves in sorted order, from left to right.

- At a leaf, the last pointer points to the next leaf block to the right, i.e., to the block with the next higher keys. Among the other n pointers in a leaf block, at least floor((n + 1)/2) of these pointers are used and point to data records; unused pointers are null and do not point anywhere. The *i* th pointer, if it is used, points to a record with the *i* th key.

Rules : Intermediate/Interior nodes

- All n + 1 pointers can be used to point to blocks at the next lower level.
- At least ceil((n + 1)/2) of them are actually used (but if the node is the root, then we require only that at least 2 be used, regardless of how large n is).
- If j pointers are used, then there will be j-1 keys, say K_1, K_2, .. K_{j-1}.
- The first pointer points to a part of the tree where some of the records with keys less than K_1 will be found. The second pointer goes to that part of the tree where all records with keys that >= K_1, but less than K_2 will be found, and so on.
- The jth pointer gets us to the part of the B-tree where some of the records with keys >= K_{j-1} are found.

Note that some records with keys far below K_1 or far above K_{j-1} may not be reachable from this block at all, but will be reached via another block at the same level.

Some Rules : Root

- At the root, there are at least two used pointers. All pointers point to the tree blocks at the level below.

Special case : If an index has only one record, a root node will have only one pointer. In this case root = leaf.

Some Rules : max and min

	Max. Keys	Max. Ptrs	Min. Keys	Min. Ptrs
Root	n	n+1	1	1 (if root=leaf)
Leaf (non-root)	n	n (to records)	f [(n+1)/2]	f [(n+1)/2]
Non-leaf (non-root)	n	n+1	c[(n+1)/2]-1	c[(n+1)/2]

For a tree of order n.

Back to the example

n = 3

Maximum:

⇒ leaf nodes can have 3 keys and 3 pointers to records at most (and one pointer to the next leaf node in sequence)

⇒ non leaf nodes can have 3 keys and 4 pointers at most (same for root)

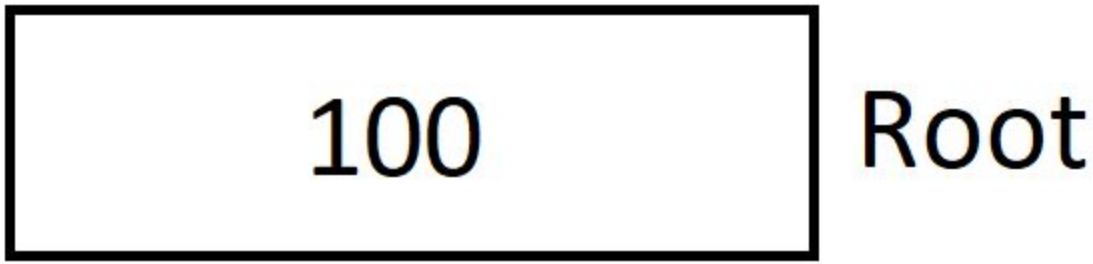
Minimum:

⇒ non leaf nodes need to have 1 key and 2 pointers at least

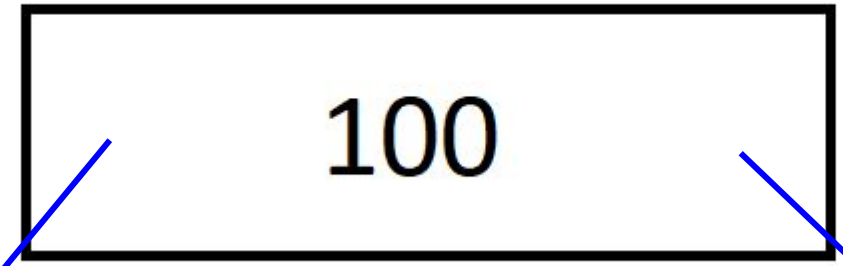
⇒ leaf nodes need to have 1 key and 1 pointer to records at least (and one pointer to the next leaf node in sequence)

⇒ root can have 1 key + 1 ptr (if only one record) otherwise at least 1 key and two pointers.

Back to the example



Back to the example



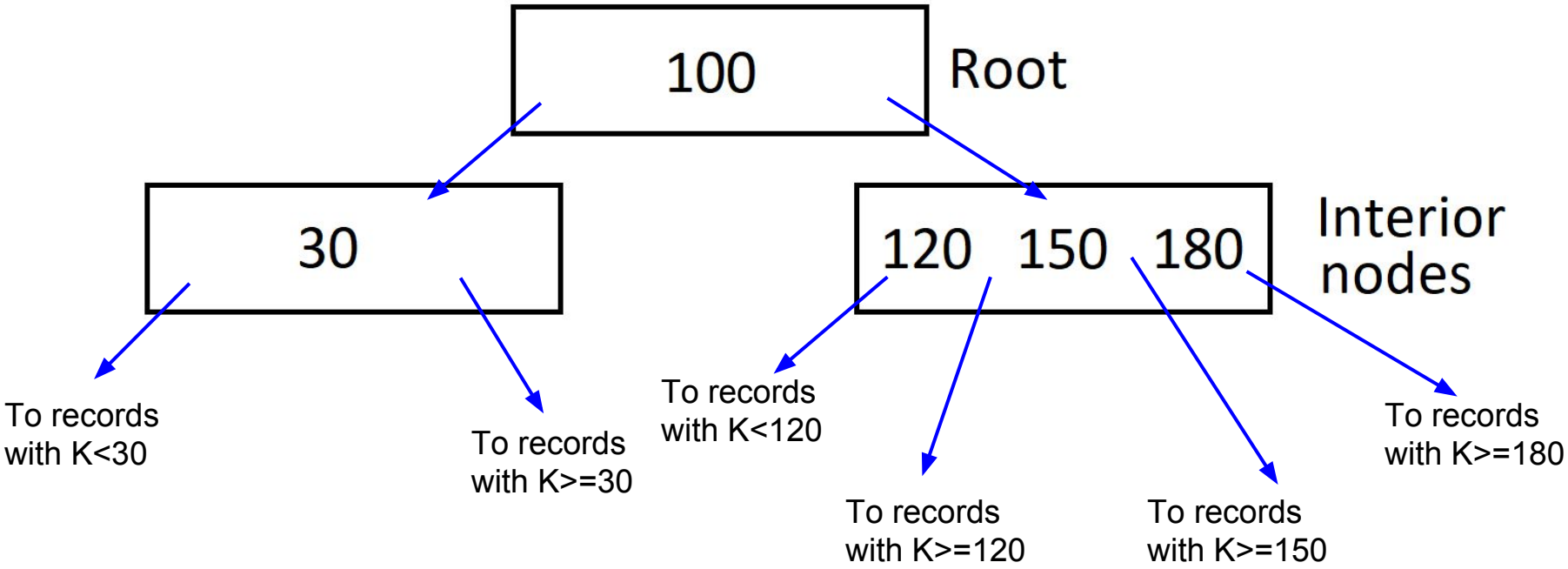
Root

All the pointers to records with **key < 100** can be found by following this pointer

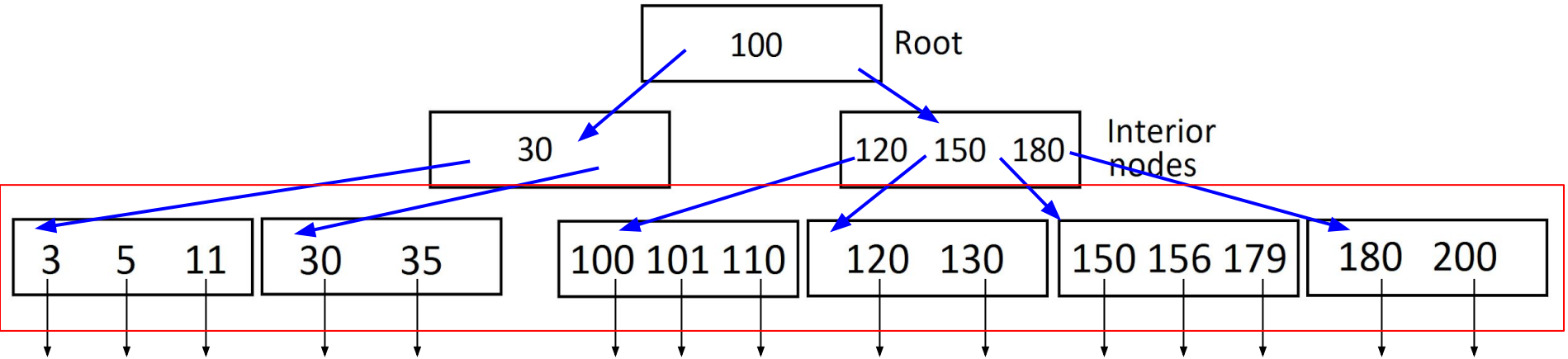
All the pointers to records with **key >= 100** can be found by following this pointer

Note: Equality to the right! If you are looking for K =100, follow the right pointer!

Back to the example

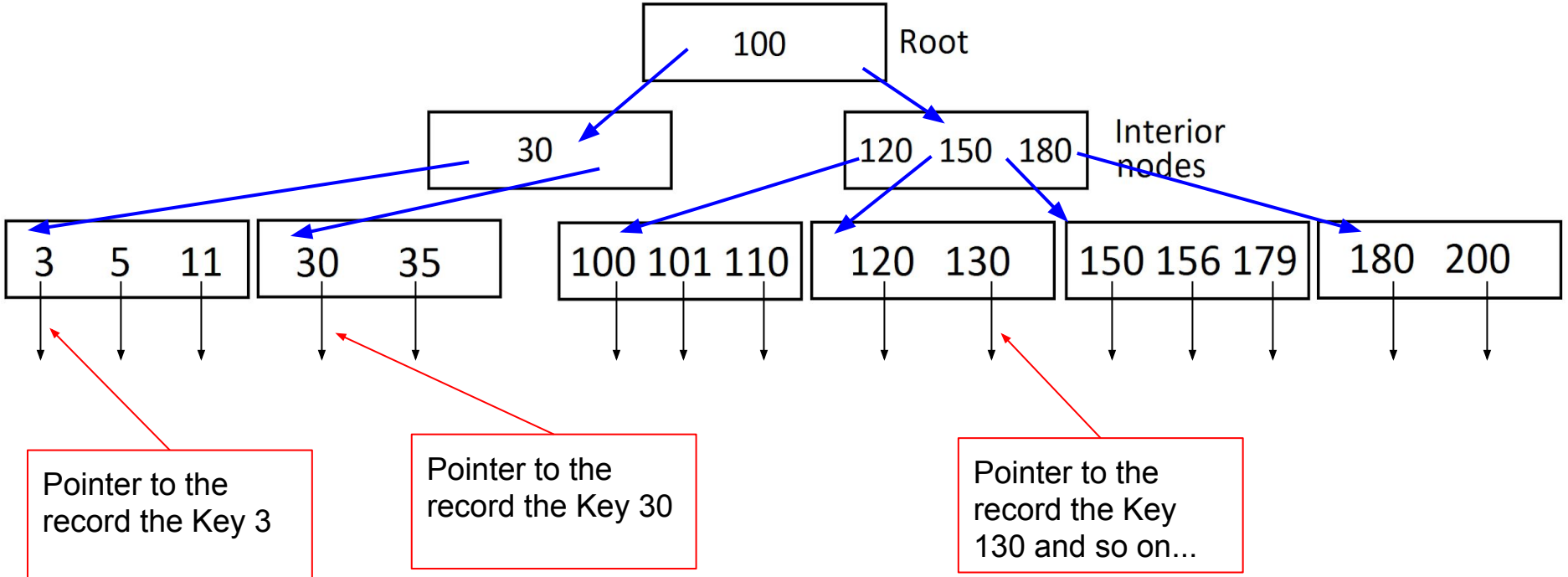


Back to the example



Leaf nodes. Note: we haven't shown the sequence pointer here, but each leaf node points to the next one in the sequential order..

Back to the example



Some Notes

The sequence of pointers at the leaves of a B-tree can play the role of any of the pointer sequences coming out of an index file that we learned about in the last chapter.

Some possibilities:

1. B+tree can be a dense index. That is, there is one key-pointer pair in a leaf for every record of the data file. The data file may or may not be sorted by search key.

2. B+tree can be a sparse index with one key-pointer pair at a leaf for each block of the data file (If the data file is sorted by the search key).

B+ Tree Insertion 1/2

We try to find a place for the new key in the appropriate leaf, and we put it there if there is room.

• If there is no room in the proper leaf, we split the leaf into two and divide the keys between the two new nodes, so each is half full or just over half full.

- Split of leaf results in insert of key-child pair at level above. Key is **copied** to level above
- Thus, recursive splitting all the way up the tree is possible (if there is room, insert it; if not, split the parent node and continue up the tree)

Convention: If the number of keys in the two nodes resulting from the split is uneven, put one more key in the left node

B+ Tree Insertion 2/2

- split of non-leaf results in **moving** one key to level above

- if we try to insert into the root, and there is no room, then we split the root into two nodes and create a new root at the next higher level; the new root has the two nodes resulting from the split as its children.

Split Leaf : **Copy** one K-P pair to the above node.
Split non-leaf : **Move** one K-P to the node above.

Splitting Leaf

LEAF (L1) with capacity n, we need to split when n+1 key arrives. We create a sibling to the right (L2).

1. L1 keeps the ceil[(n+1)/2], remaining go to L2.

For example:

-If n=3, we will split when 4th key arrives. We have four keys now, L1 gets first 2, and L2 the remaining 2.

-If n=4, we split when 5th key arrives, L1 gets first 3 keys and L2 gets remaining 2.

Minimum requirement of floor[(n+1)/2] is satisfied for both.

2. The first key from L2 gets copied to parent node.

Splitting non-Leaf

Interior node (N1) with capacity n, we need to split when n+1 *th* key arrives. We create a sibling to the right (N2).

1. N1 keeps the first ceil[n/2], N2 gets the last floor[n/2].

For example:

-If n=3, we will split when 4th key arrives. We have four keys now, N1 gets first 2, and N2 the last 1. **(Middle Key moves up)**

-If n=4, we split when 5th key arrives, N1 gets first 2 keys and N2 gets the last 2. **(Middle Key moves up)**

2. The middle key is moved to parent node.

Insertion Rules Summary

1. Find appropriate leaf - if space available - INSERT
2. Else SPLIT, divide keys and copy middle key to parent
3. If copying results in parent splitting, Move key to parent's parent.

Leaf Split : L1 keeps the ceil[(n+1)/2], remaining go to L2. Copy first key of L2 to parent

non-Leaf Split: N1 keeps the first ceil[n/2], N2 gets the last floor[n/2]. Middle moves to parent.

Root Split: same as non-Leaf, middle key is the new root.

Exercise 13.3.3 DSCB

Exercise 13.3.3 : Suppose pointers are 4 bytes long, and keys are 12 bytes long. How many keys and pointers will a block of 16,384 bytes have?

Exercise 13.3.3 DSCB Ans

Exercise 13.3.3 : Suppose pointers are 4 bytes long, and keys are 12 bytes long. How many keys and pointers will a block of 16,384 bytes have?

Ptrs =1024 (in a leaf node there will be 1023 ptrs to records, and one sequence ptr.)

Keys = 1023

12n+4(n+1) = 16384
12n+ 4n +4 = 16384
12n+ 4n = 16384 - 4
16n = 16380
n = 16380 /16
n = 1023 keys (when rounding off take the lower pointer)
n+1 = 1024 pointers

Since in any block, Number of Ptrs = Number of Keys + 1, if we can have n keys, we will have n+1 ptrs, therefore: 12n + 4(n+1) = 16384

Exercise 1

Q. A B+ tree of order n = 5 and 3 levels.

a) What is the maximum number of keys you can store.
b) What is the minimum number of keys for 3 levels.

Repeat for n = 120 and 3 levels

c) max Keys

d) min Keys

Exercise 1 Ans

Q.a) and b) B+ tree of order n = 5 and 3 levels.

c) and d) B+ tree of order n = 120 and 3 levels.

a) Max : 180
b) Min : 18
c) Max : 1756920 (121*121*120)
d) Min : 7320 (2*61*60)

Exercise 2

Insert the following in a B+ tree

A. 1, 2, 3,.... 18 and n=4
B. 15, 13, 2, 3, 1, 10, 7, 5, 4, 11, 9, 6, 12, 8, 14 and n=4
C. Repeat 1 and 2 for n=2 and n=5

Exercise 2 Sol

Insert the following in a B+ tree

A. 1, 2, 3,.... 18 and n=4
B. 15, 13, 2, 3, 1, 10, 7, 5, 4, 11, 9, 6, 12, 8, 14 and n=4
C. Repeat 1 and 2 for n=2 and n=5

Match your answers at http://goneill.co.nz/btree-demo.php

(Note: though this particular website follows our rules of insertion (there are many different rules out there), for the correct answer you need to give the order as n+1, ie for the first part use 5 on the website instead of 4, and so on...)

B+ Tree Deletion 1/2

Search for key being deleted. If found in the leaf, delete.

If the lower limit on occupancy is violated:

1. First look for an adjacent sibling that is above lower limit; transfer a key-pointer pair from that sibling.

Convention: If you have the choice, use left sibling

A transfer between non-leaves involves a key in the parent and also results in the transfer of a child

2. If none, then there must be two adjacent leaves, one at minimum, one below minimum. Just enough to merge nodes.

Convention: If you have the choice, use left sibling

B+ Tree Deletion 2/2

- A merge is the opposite of a split: delete key in parent when merging leaves; move key from parent into merged node for non-leaves
- Merging two nodes is effectively deleting one of them. We need to adjust the keys at the parent, and then delete a key and pointer at the parent. If the parent is still full enough, then we are done. If not, then we recursively apply the deletion algorithm at the parent.

B+ Tree Deletion Summary

1. Delete Key
2. If lower limit violated - BORROW
   a. If choice - use left sibling
   b. Transfer might require updating parents, specially for right sibling
3. If borrow not possible - MERGE
   a. If choice - use left sibling
   b. Delete key in parent when merging leaf nodes
   c. Move key from parent to merged node for non-leaf nodes.

Exercise

Do the following Deletions from the tree constructed in Exercise 2:

6, 11, 5, 2, 12, 15, 7, 8

Match your answers at http://goneill.co.nz/btree-demo.php

(Note: though this particular website follows our rules of insertion (there are many different rules out there), for the correct answer you need to give the order as n+1, ie for the first part use 5 on the website instead of 4, and so on...)

Exercise 13.3.9

If we use the 3-key, 4-pointer nodes how many different B+ trees are there when the data file has the following numbers of records: (a) 6 (b) 10 !! (c) 15.

Not very important, just for discussion and thinking!

Exercise 13.3.9

If we use the 3-key, 4-pointer nodes how many different B+ trees are there when the data file has the following numbers of records: (a) 6 (b) 10

a) 2
b) 12+2

These are the possible options I have been able to think of, let me know if can find more possibilities!

B Tree (a variant of B+ trees)

Non-leaf nodes no longer contain only the pointer to child nodes, but also **record** pointers corresponding to the keys they contain.

Number of pointers in the non-leaf nodes there is no longer just K+1 (where K is the number of keys) but rather

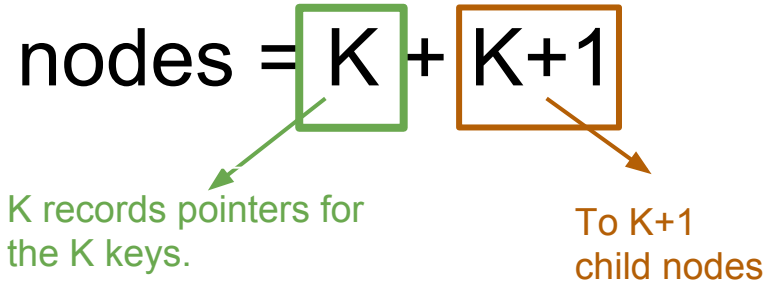
No. of pointers in non leaf nodes = K + (K+1)

B Tree (a variant of B+ trees)

Non-leaf nodes no longer contain only the pointer to child nodes, but also **record** pointers corresponding to the keys they contain.

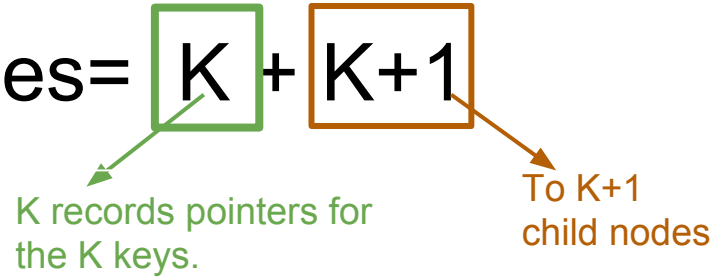
Number of pointers in the non-leaf nodes there is no longer just K-1 (where K is the number of keys in the node) but rather:

No. of pointers in non leaf nodes = K + K+1



B Tree (a variant of B+ trees)

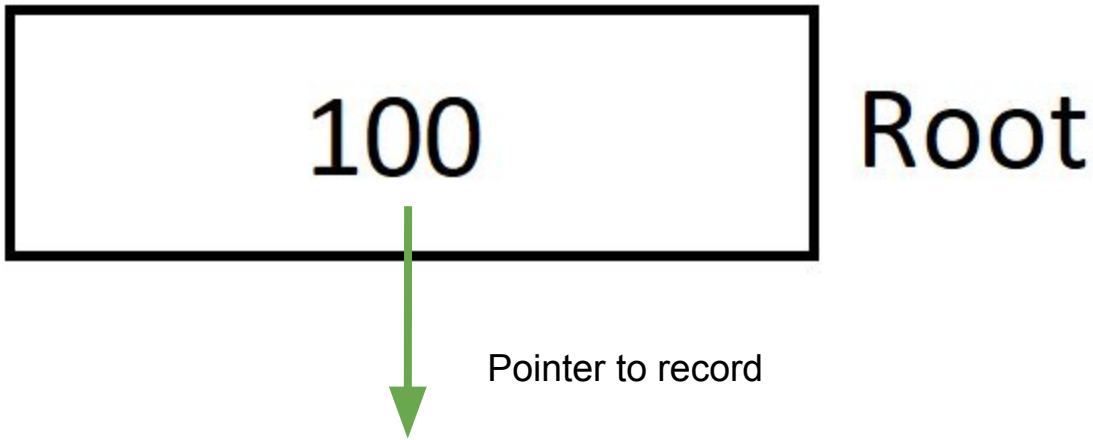
No. of pointers in non leaf nodes= K + K+1



No. of Keys in such a non leaf node = K

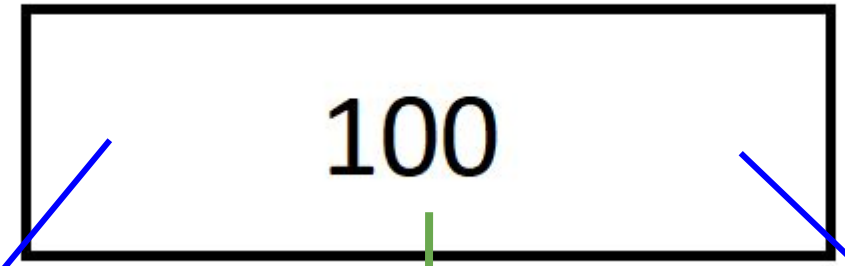
In a leaf Node, number of Keys and Pointers will be equal.

Back to the example



Note: if there is only one node, and the root is the leaf as well, then B and B+ tree will be the same!

B Tree Example



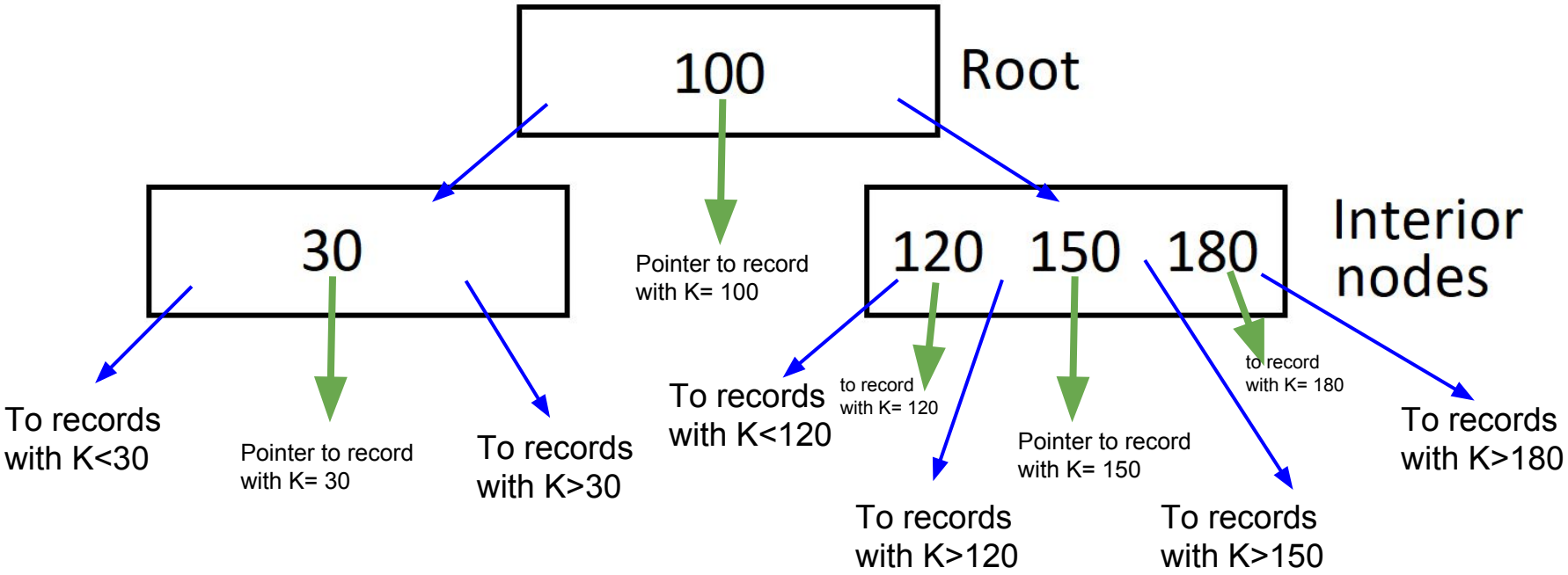
All the pointers to records with **key < 100** can be found by following this pointer

Pointer to record with K= 100

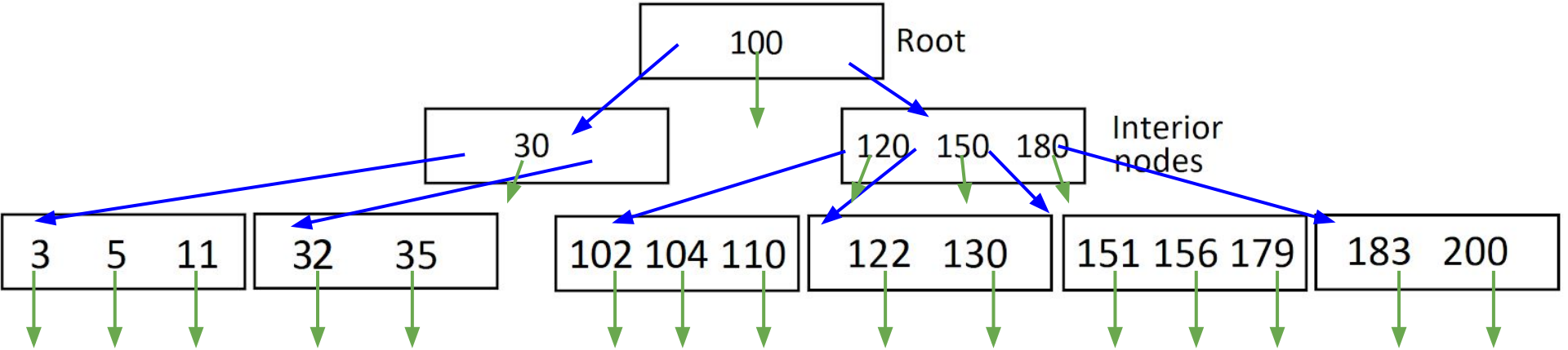
All the pointers to records with **key > 100** can be found by following this pointer

Note: **Equality no longer to the right**! If you are looking for K =100, you can just follow the pointer in the node you find 100.

B Tree Example

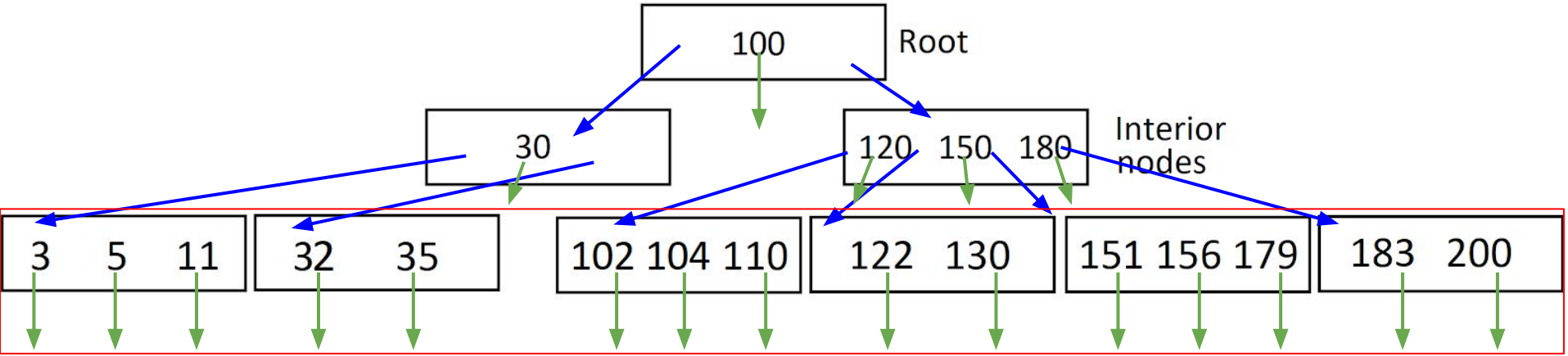


B Tree Example



Green pointers denote pointers to records, while blue ones are pointers to child nodes.

B Tree Example



Leaf nodes are no longer in a sequence and therefore it doesn't make sense to have the sequence pointer now.

Exercise 1

For - Pointers = 4 bytes

- Keys = 4 bytes
- Blocks 1024 bytes fixed size.

How many records can we store in a full 3 level:

a) B+ Tree
b) B Tree

Exercise 1 : Ans

For - Pointers = 4 bytes

- Keys = 4 bytes

- Blocks 1024 bytes fixed size.

How many records can we store in a full 3 level:

$4n + 4(n+1) = 1024$

a) B+ Tree

R I L

128*128*127

b) B Tree

R I L

85+ 86*85 + 86*86*128

Exercise 2

Consider a database that has the following characteristics:

- 2KB fixed-size blocks
- 12-byte pointers
- 56-byte block headers

We want to build an index on a search key that is 8 bytes long.

Calculate the maximum number of records we can index with

a) 3-level B+ tree (2 levels plus the root)
b) 3-level B tree

Exercise 2: Sol 1/2

Key Size = 8bytes

Pointer Size = 12bytes

Block size = 2048 bytes,

Space available in block = 2048 - 56 = 1992 bytes

a) 3-level B+ tree (2 levels plus the root)

If order of the B+ tree is n, then:

$$8*n + 12(n+1) \leq 1992 \Rightarrow n = 99$$

Max. keys = No. of leaf nodes * keys in one leaf node

= (100*100)*99

R I L

Exercise 2: Sol 1/2

Key Size = 8bytes

Pointer Size = 12bytes

Block size = 2048 bytes,

Space available in block = 2048 - 56 = 1992 bytes

a) 3-level B+ tree (2 levels plus the root)

If order of the B+ tree is n, then:

$8*n + 12(n+1) \leq 1992 \Rightarrow n = 99$

Each node can hold 99 keys and 100 pointers.
At non-leaf level 100 pointers means 100 child nodes.
Remember interior nodes are themselves child nodes of the level above (here the root).

Max. keys = No. of leaf nodes * keys in one leaf node

= (100*100)*99

Exercise 2: Sol 2/2

b) 3-level B tree

At Non-leaf: $8^{K}*n + 12^{DP}*n + 12^{BP}(n+1) \leq 1992 \Rightarrow n = 61$

At leaf: $8*n + 12*n \leq 1992 \Rightarrow n = 99$

i.e. A non-leaf node can hold 61 Key-recordpointer pairs(+ 62 pointers to child nodes), and a leaf node can hold 99 key-recordpointer pairs.

Therefore Max Keys = 61 + 62*61 + 62*62*99 = 384399

Exercise 2: Sol 2/2

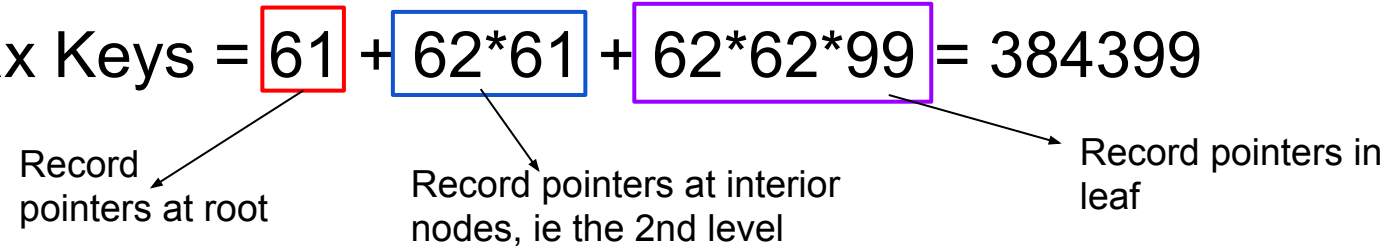
b) 3-level B tree

At Non-leaf: $8*n + 12*n+12(n+1) \leq 1992 \Rightarrow n = 61$

At leaf: $8*n + 12*n \leq 1992 \Rightarrow n = 99$

i.e. A non-leaf node can hold 61 Key-recordpointer pairs(+ 62 pointers to child nodes), and a leaf node can hold 99 key-recordpointer pairs.

Therefore Max Keys = 61 + 62*61 + 62*62*99 = 384399



Important Note!

In real life block size is always fixed, therefore in a B Tree implementation a leaf node will hold more Keys than a non-leaf node.

But while discussing the insertion/deletion algorithms, to make our task easier, we shall consider n (the order of the tree) as the maximum number of keys we can insert in both leaf and non-leaf nodes, so that we can focus on the algorithm and our task does not become very complex!

In the following slides you will see this as we discuss insertion/deletions.

Rules for B Tree

During insertions we will consider the maximum number of keys in leaf and non-leaf to be equal and equal to the order of the tree.

Minimum keys and pointers are the same for leaf nodes and non-leaf nodes.

Some Rules : max and min

	Max. Keys	Max. Ptrs	Min. Keys	Min. Ptrs
Root	n	n+1	1	1 (if root=leaf)
Non-leaf And Leaf	n	n+1	$c[(n+1)/2]-1$	$c[(n+1)/2]$

For a tree of order n.

Insertions and Deletions

Insertion: Same as B+Tree non-leaf nodes.

Split-Merge Rules for all Nodes:

Split: MOVE one key up

Merge: MOVE a parent to handle underflow

Please refer to the **Splitting non-leaf** slide for B+ Tree as the rules are the same for all the nodes of a B Tree (i.e. both the leaf and non leaf nodes of a B Tree follow the Insertion/split/merge rules of a non-leaf node of a B+Tree).

Deletion:

1. When deleting a leaf, if underflow occurs, first try to borrow, if not possible merge. (On merge, MOVE a parent to handle the resulting underflow.)
2. When deleting a key that is in a non-leaf: replace the key with the next larger key from the leaf.
3. When deleting root, replace root with smallest key on right side of the tree.

- Handling of underflow starts from leaf.

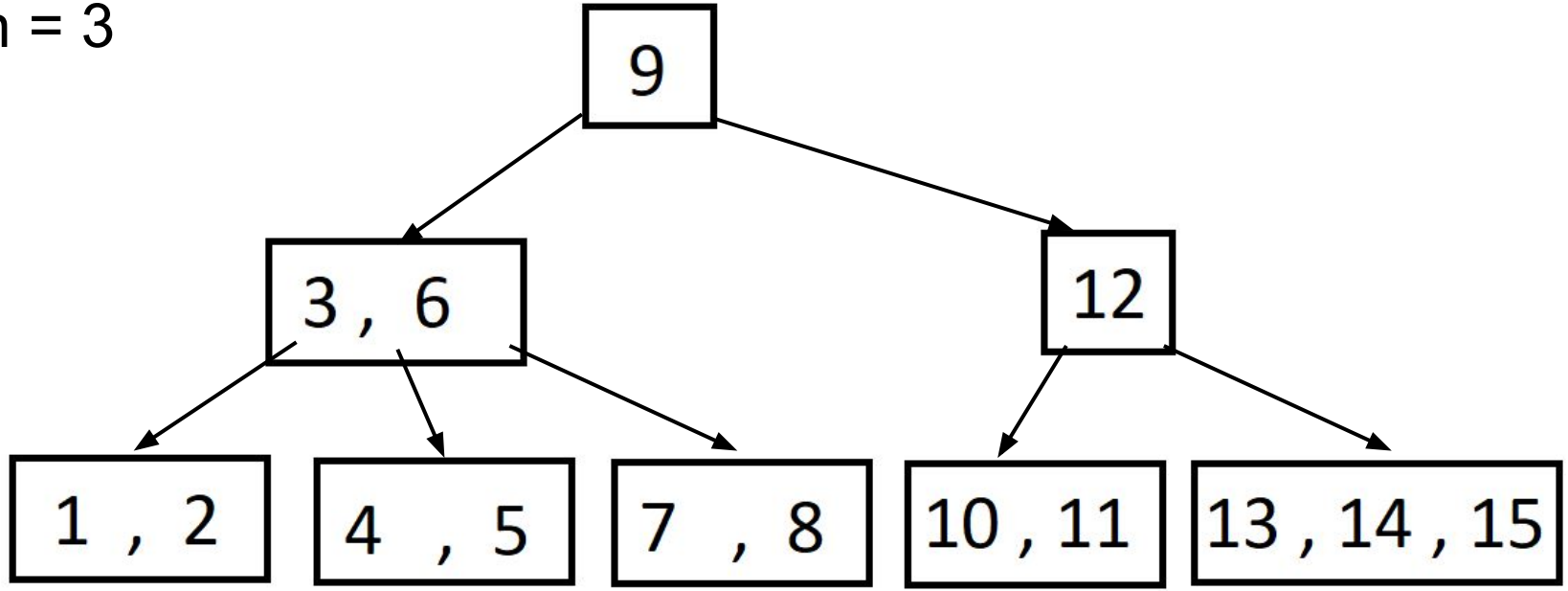
Exercise 3

Insert into a B Tree the numbers. 1,2,3….15.

a) for n=3
b) for n=4
c) Delete the following keys from the B tree created in (a).
   9, 11, 6, 14, 15

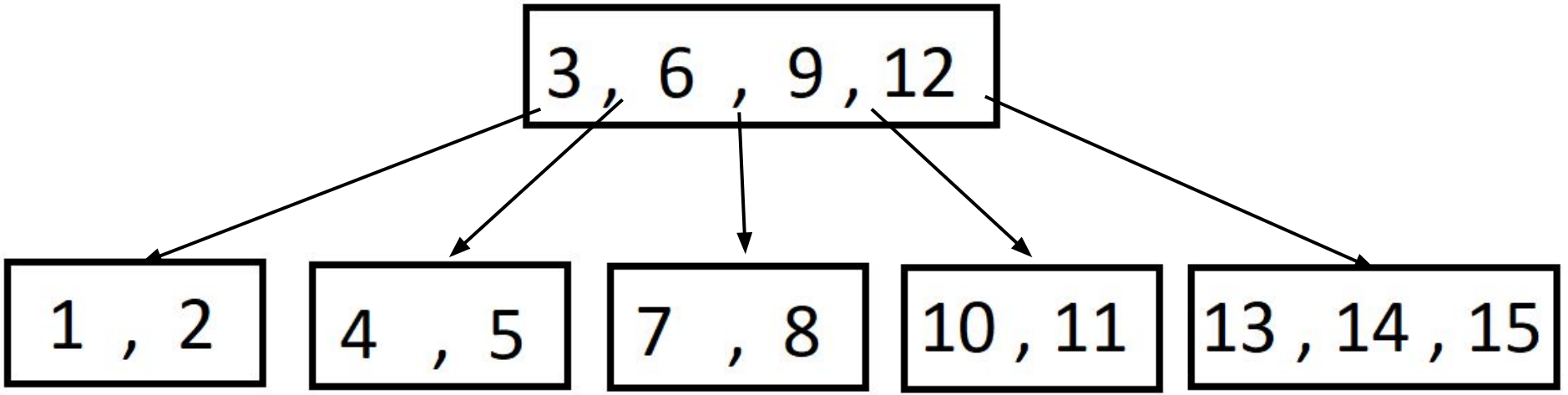
Exercise 3a Sol

n = 3



Exercise 3b Sol

n = 4



Please insert 16, and 17 in this tree and see what happens.

Exercise 3(ab) Notes

For n = 3 the first split happens when we insert key 4, This results in two children (with 1,2 on the left and 4 on the right) and 3 is the new root.

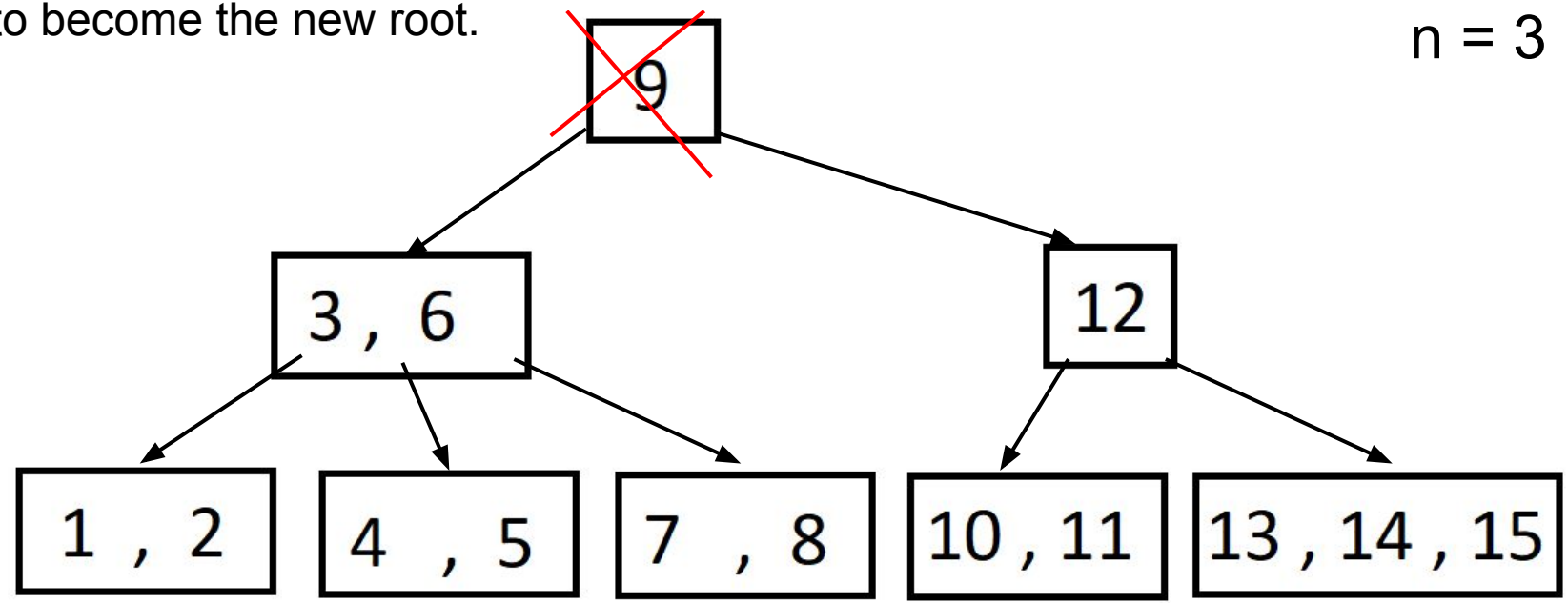
For n=4, the first split happens when we insert key 5, the split creates two child nodes (which get 1,2 and 4,5 keys)and 3 becomes the new root.

Hint (for inserting 16 and 17): You should get 9 as the new root.

Exercise 3c Deletions -9 a

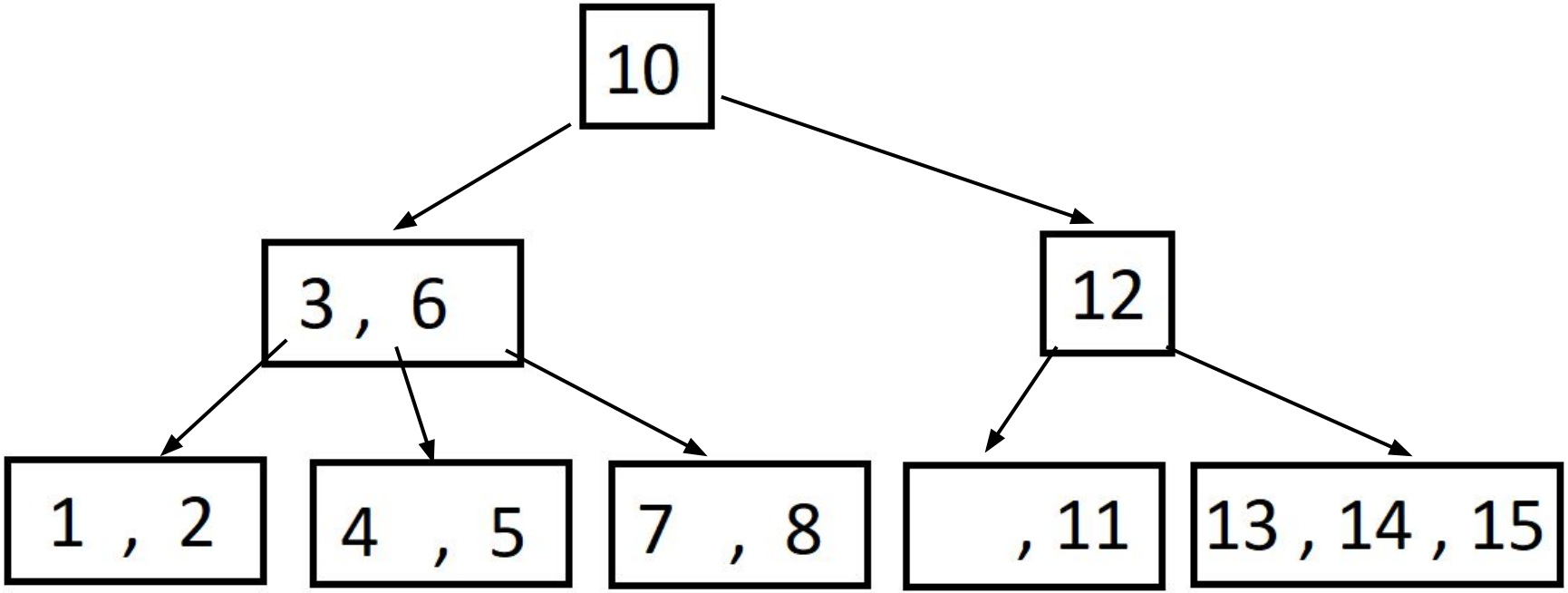
After Deleting 9, 10 will move up to become the new root.

n = 3



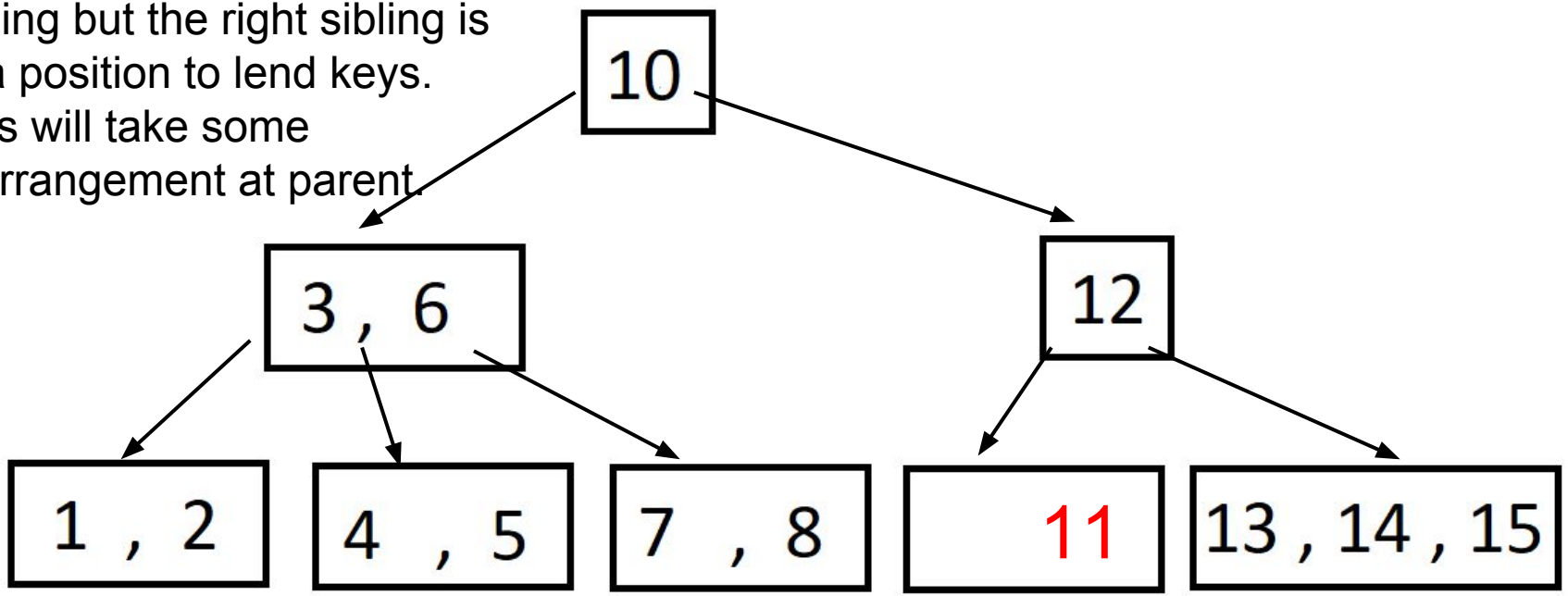
Exercise 3c Deletions -9

No underflows so we don't have to do anything else.

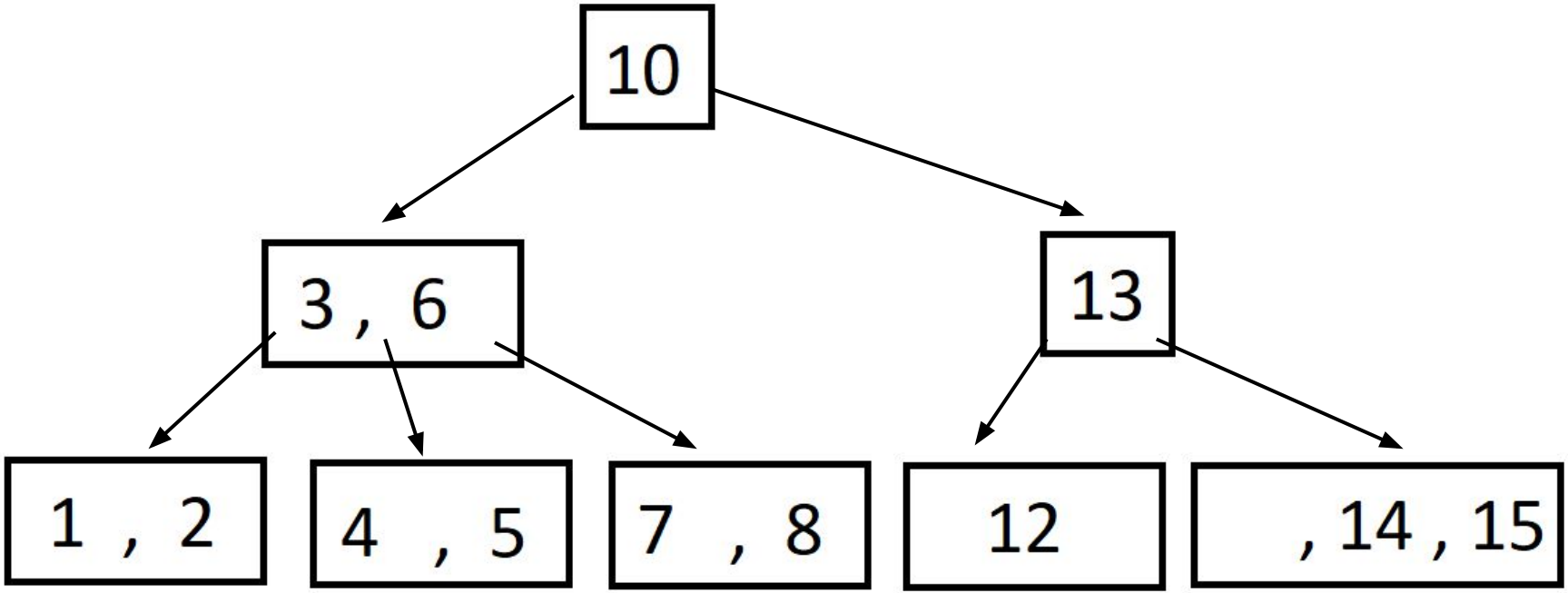


Exercise 3c Deletions -11 (a)

We will have an underflow after deleting 11. No left sibling but the right sibling is in a position to lend keys. This will take some rearrangement at parent.

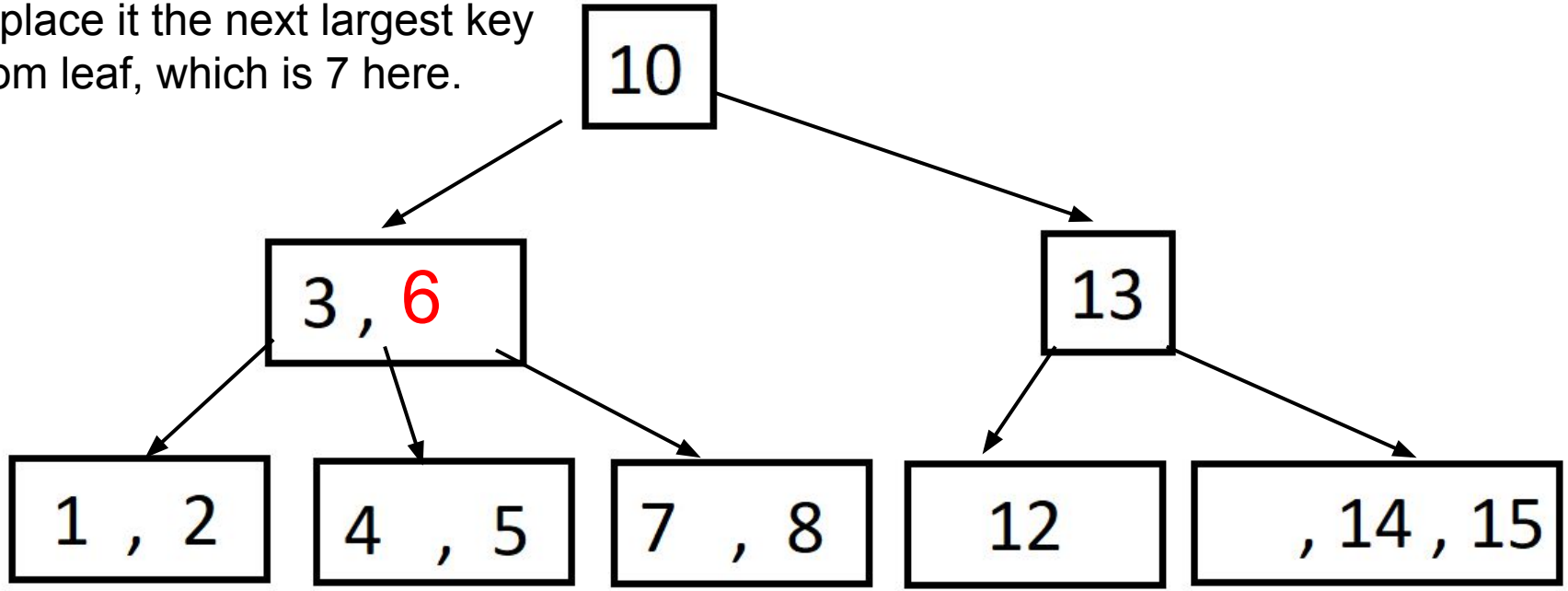


Exercise 3c Deletions -11 (b)

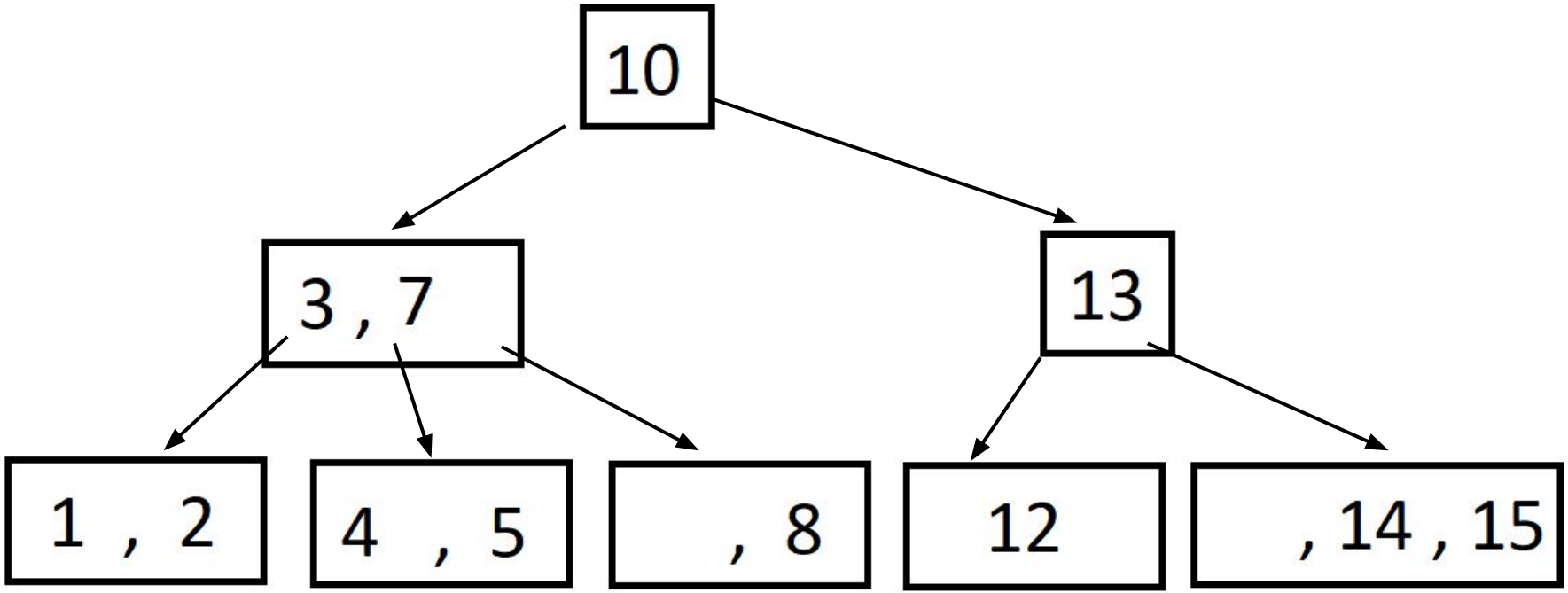


Exercise 3c Deletions -6 (a)

6 is a non-leaf key. When deleting a non-leaf key we replace it the next largest key from leaf, which is 7 here.

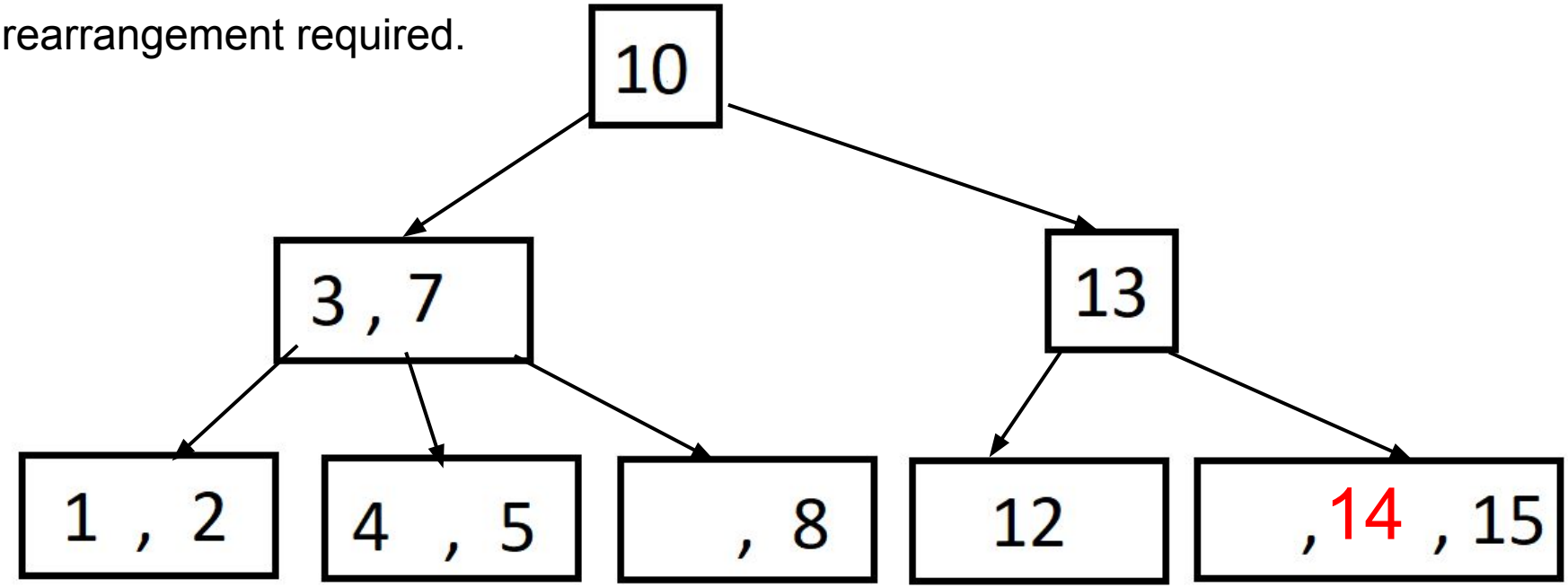


Exercise 3c Deletions -6 (b)



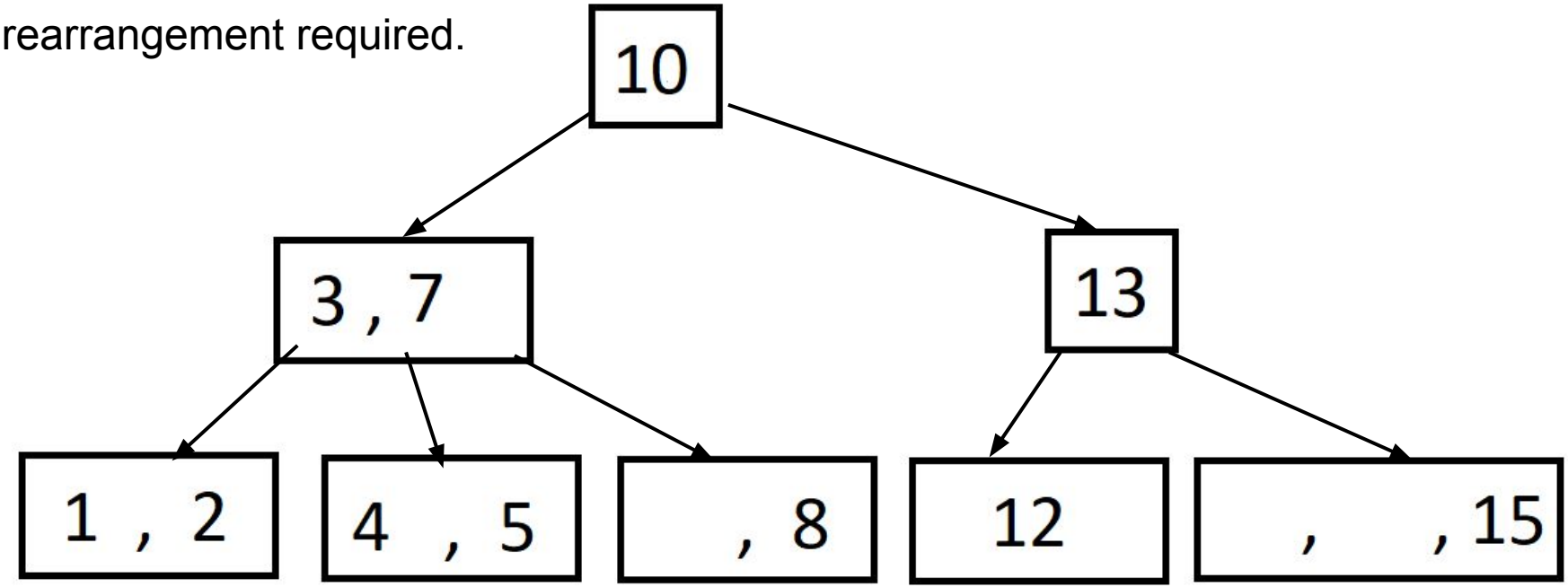
Exercise 3c Deletions -14 a

Deleting 14 results in no underflows so no rearrangement required.



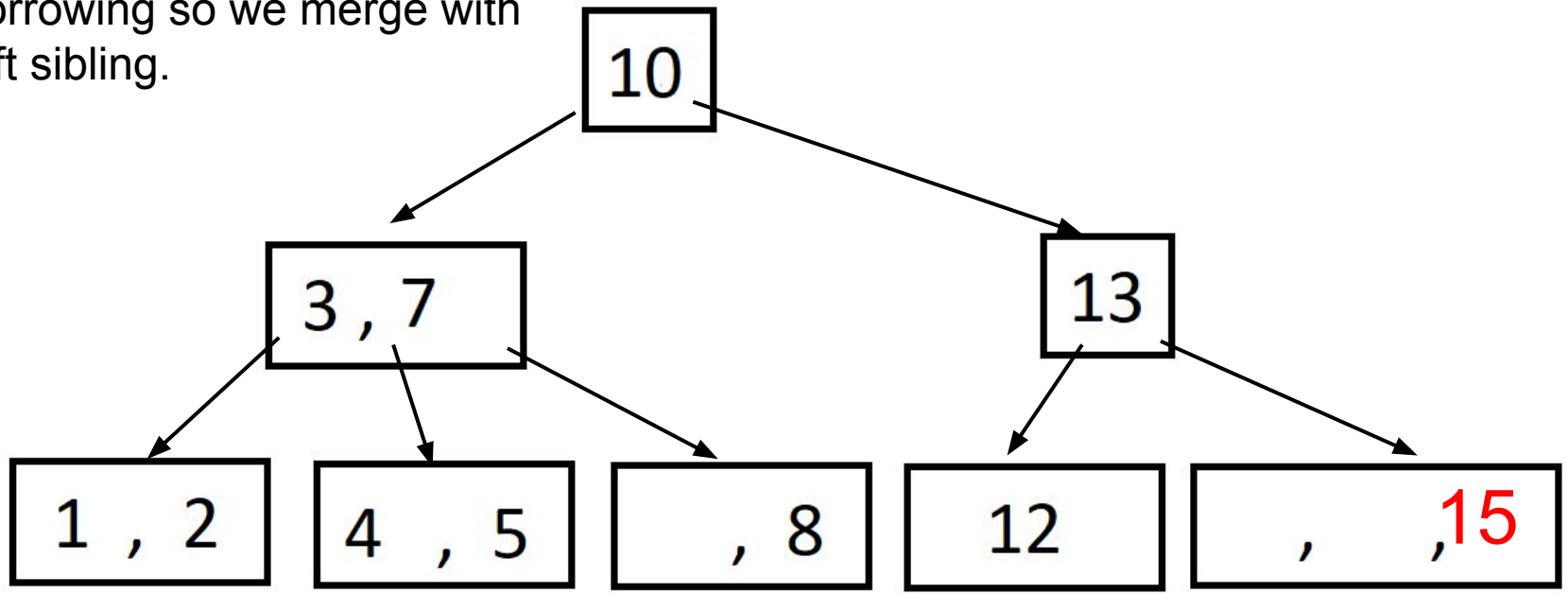
Exercise 3c Deletions -14 b

Deleting 14 results in no underflows so no rearrangement required.



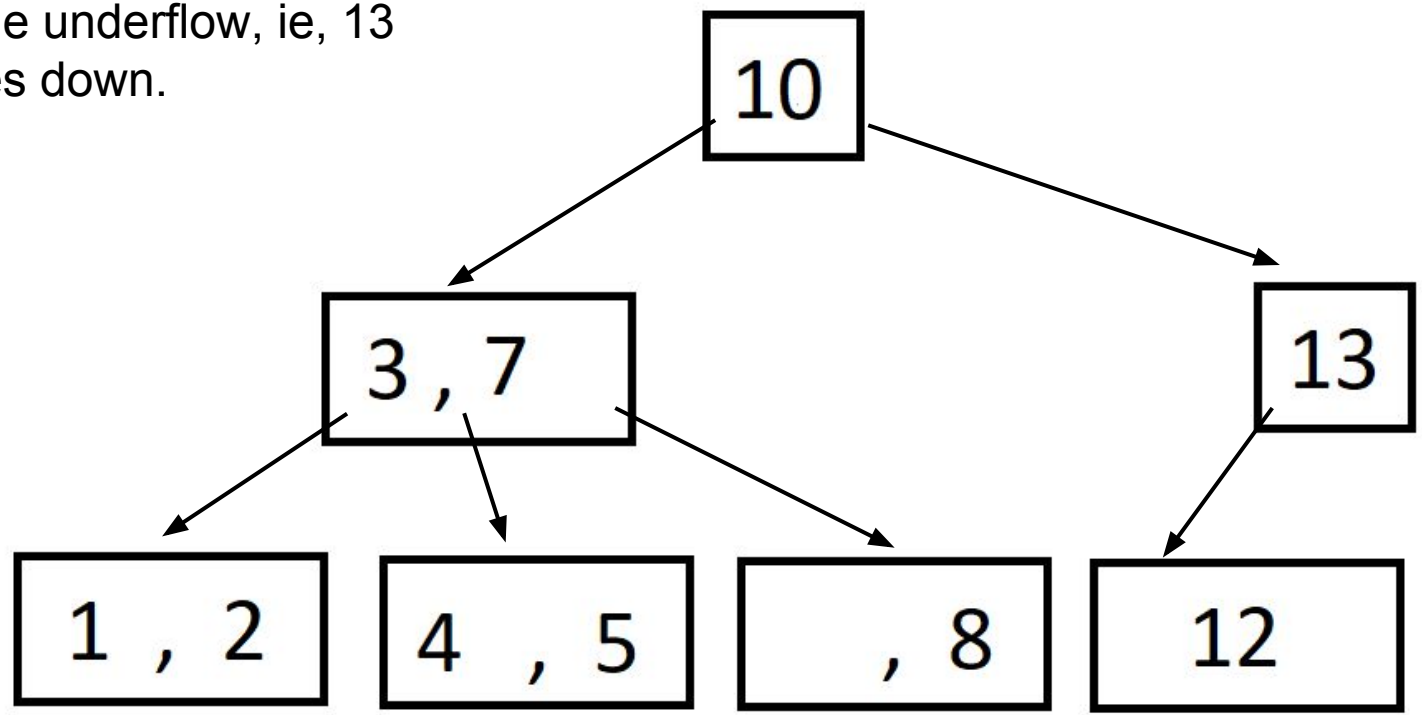
Exercise 3c Deletions -15 (a)

Deleting 15 will result in an underflow, no possibility of borrowing so we merge with left sibling.



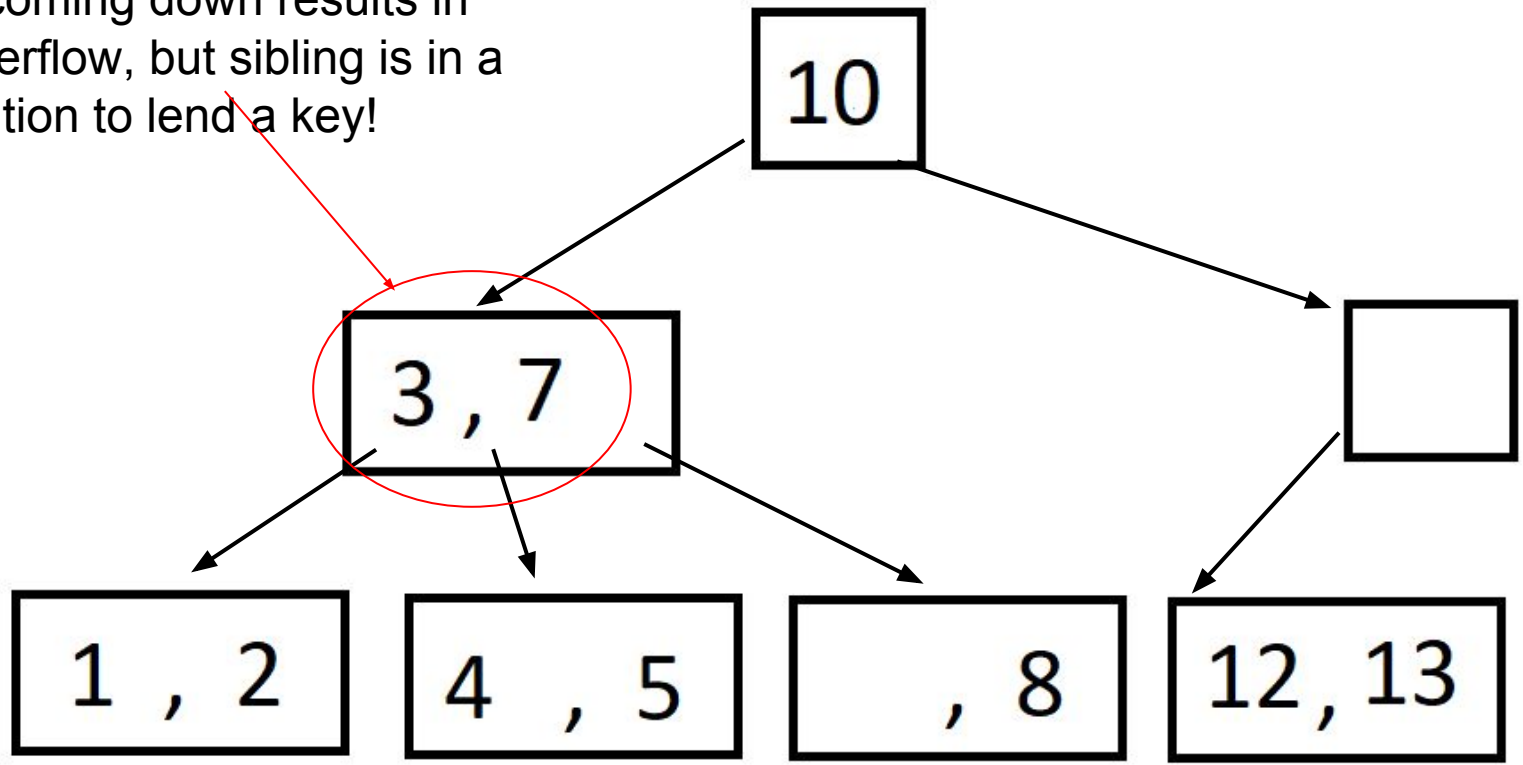
Exercise 3c Deletions -15 (b)

On merge, MOVE parent to handle underflow, ie, 13 comes down.



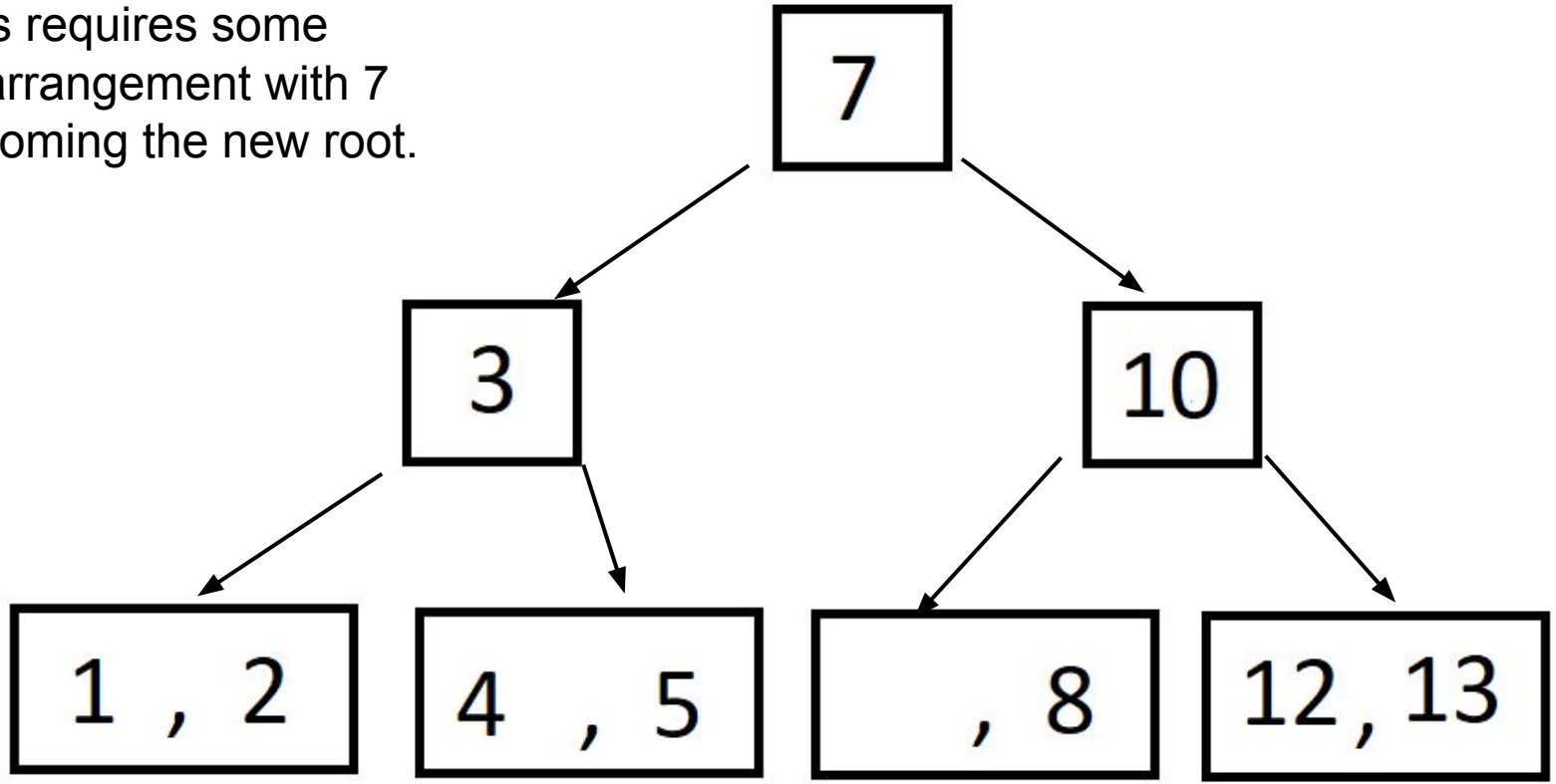
Exercise 3c Deletions -15 (c)

13 coming down results in underflow, but sibling is in a position to lend a key!



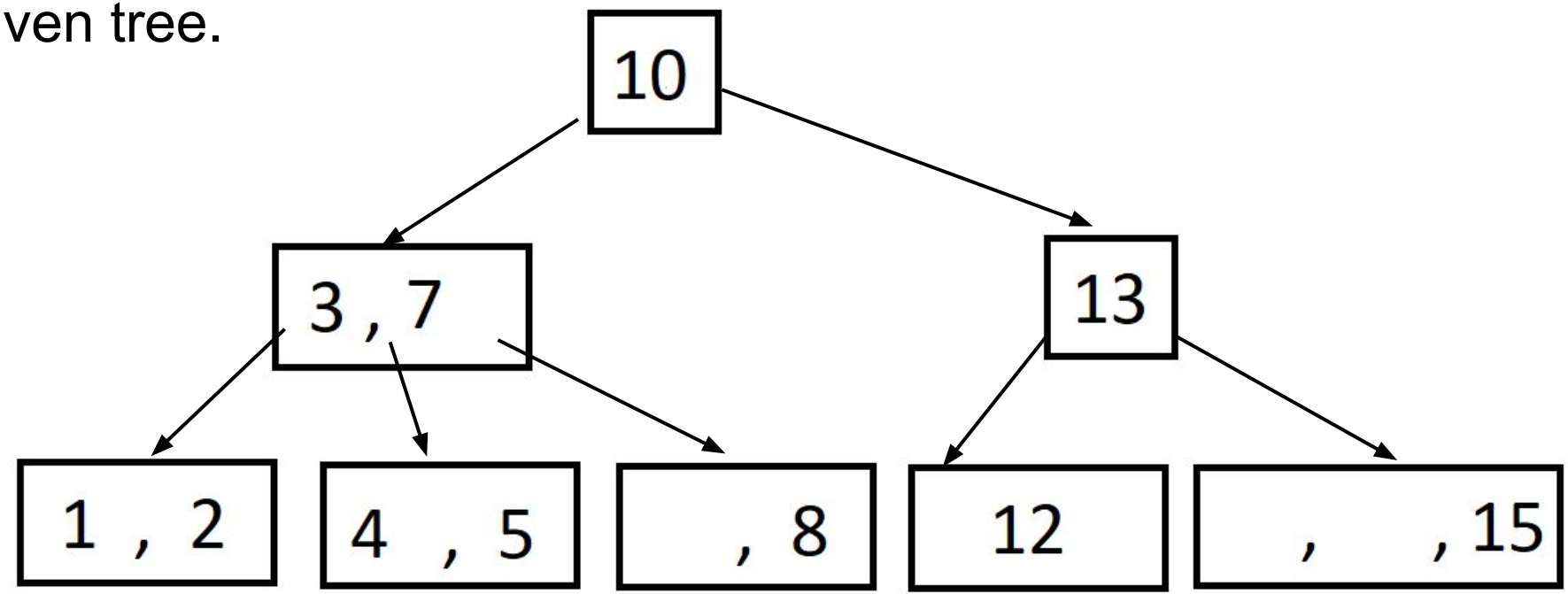
Exercise 3c Deletions -15 (d)

This requires some re-arrangement with 7 becoming the new root.



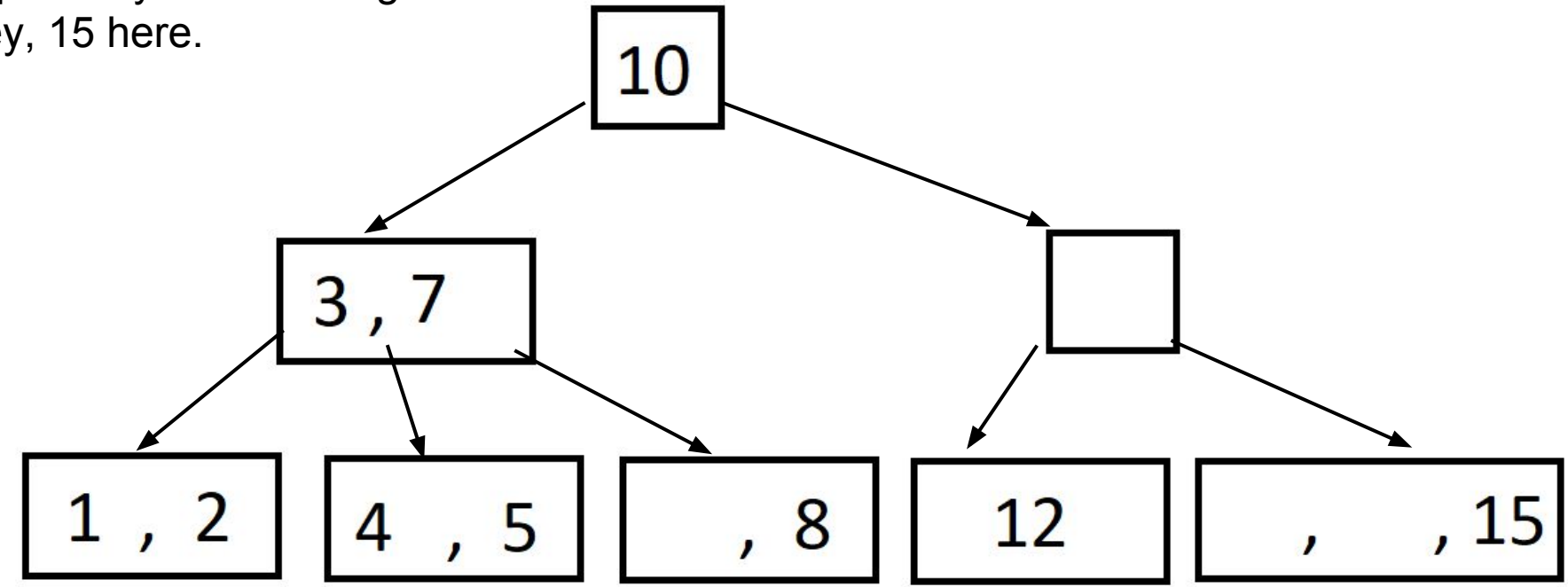
Exercise 4

Delete 13 from the given tree.



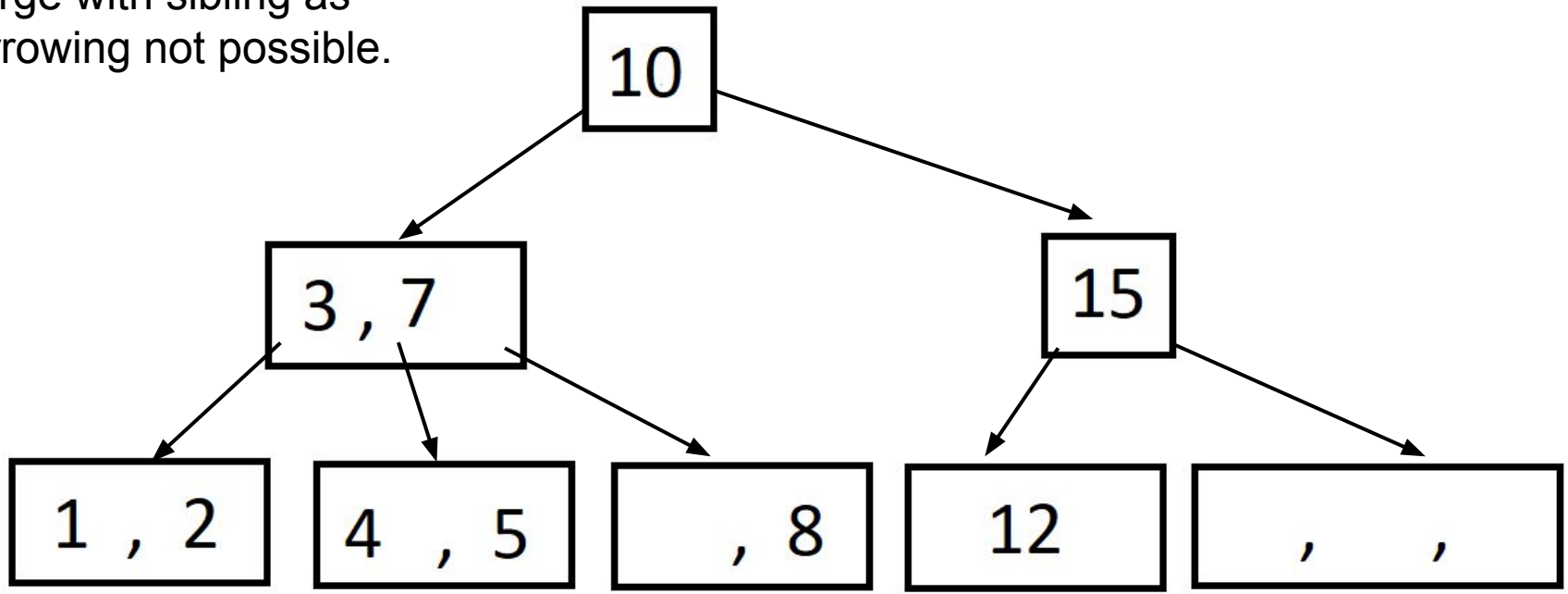
Exercise 4 Sol. 1/5

When deleting non-leaf we replace by the next largest key, 15 here.



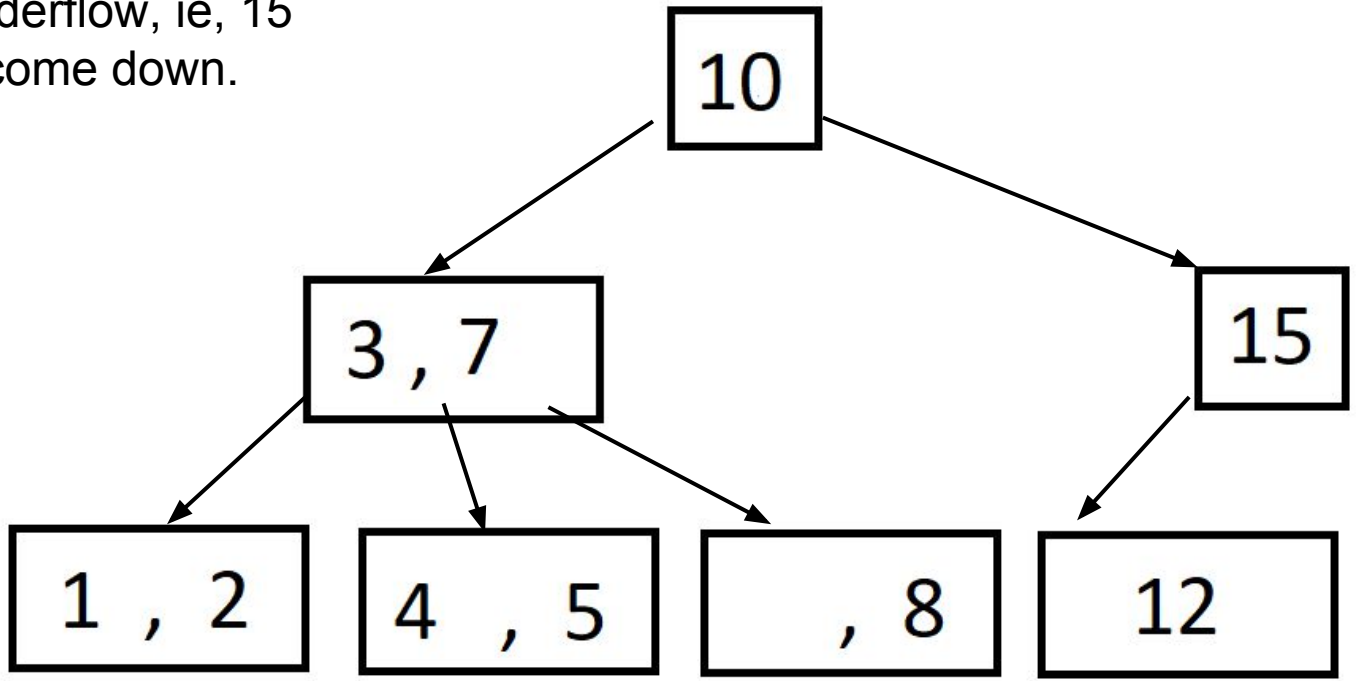
Exercise 4 Sol. 2/5

Underflow at leaf, need to merge with sibling as borrowing not possible.



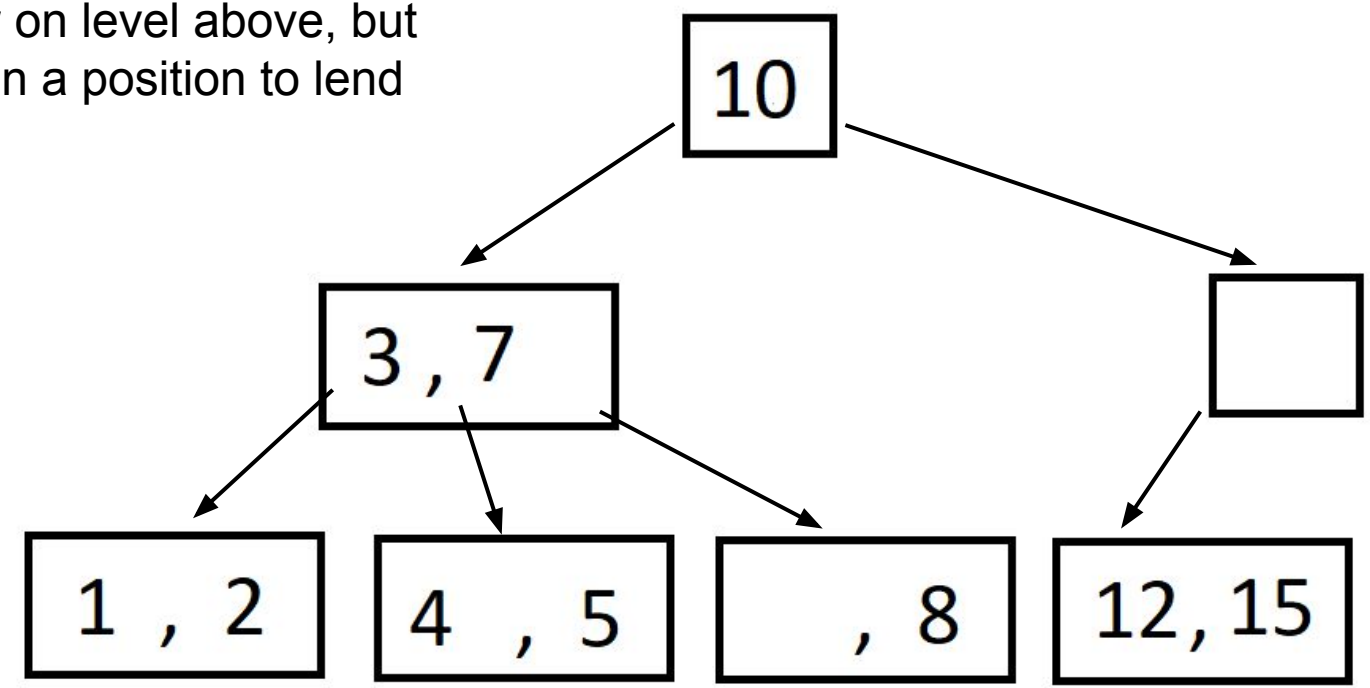
Exercise 4 Sol. 3/5

On merge, MOVE parent to handle underflow, ie, 15 needs to come down.



Exercise 4 Sol. 4/5

15 coming down creates a underflow on level above, but sibling is in a position to lend key



Exercise 4 Sol. 5/5

This requires some re-arrangement with 7 becoming the new root.