B-Trees

B-tree(a special type of balanced multiway search tree)

- B-tree of order n is a ***balanced*** multiway search tree of order n in which each ***non root node*** contains atleast (n-1)/2 keys.
- It is a balanced tree: it means all leaves are at the same level.
- **Root node** can contain less than (n-1)/2 keys.
- If we are creating a B-tree of order 7 then:
- *Maximum no. of keys a node can have is 6*
- *Minimum no. of keys a node can have is 3* ***except root***
- All the leaves of the tree will be at same level

Operations on B-Tree

- Insertion operation
- Search operation
- Deletion operation

Building B-tree (Search and Insert)

How to build a B-Tree: Algorithm to insert elements….(**key**)

1. Start searching the B-tree in similar fashion as MST and locate the leaf into which the **key** should be inserted
2. If located leaf is not full, insert the **key** in proper position in that node

else

1. If root node, create two new nodes: split the contents of old node as left and right node
2. n/2 lower keys go into the left node, n/2 larger keys go into the right node and middle key will remain with the root node

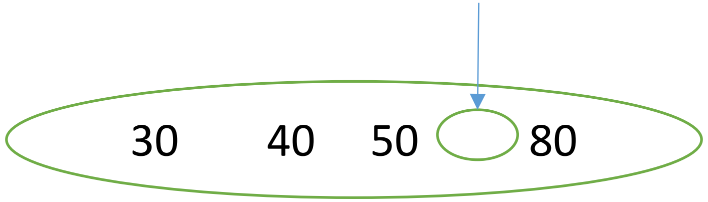
    else

Cont..

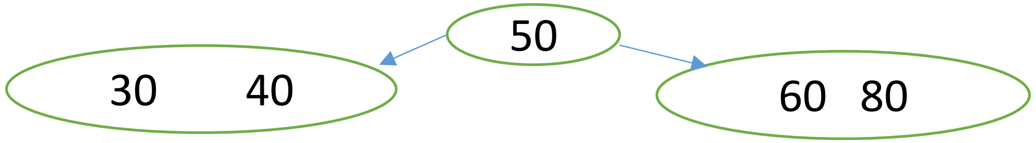
1. Create a new node and split the contents of old node as left and right node
2. n/2 lower keys go into the left node, n/2 larger keys go into the right node
3. The **separator key** or **the middle key** goes up to the **father node** (if father node not full)
4. If father node full, then father node is broken in the same way

Build Btree of order 5

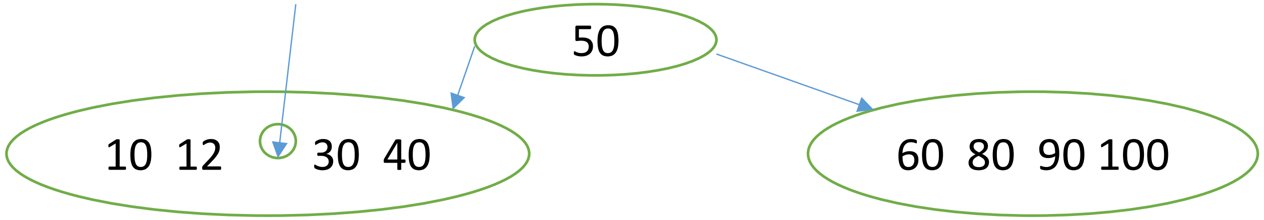
- 30, 40, 50, 80



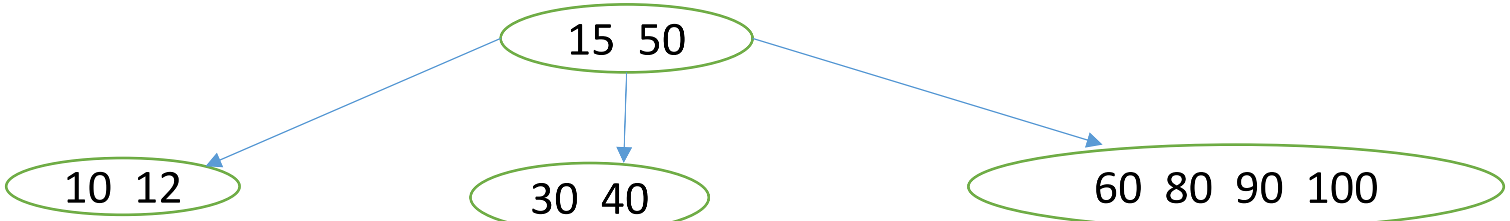
- 60



- 10, 90, 100, 12

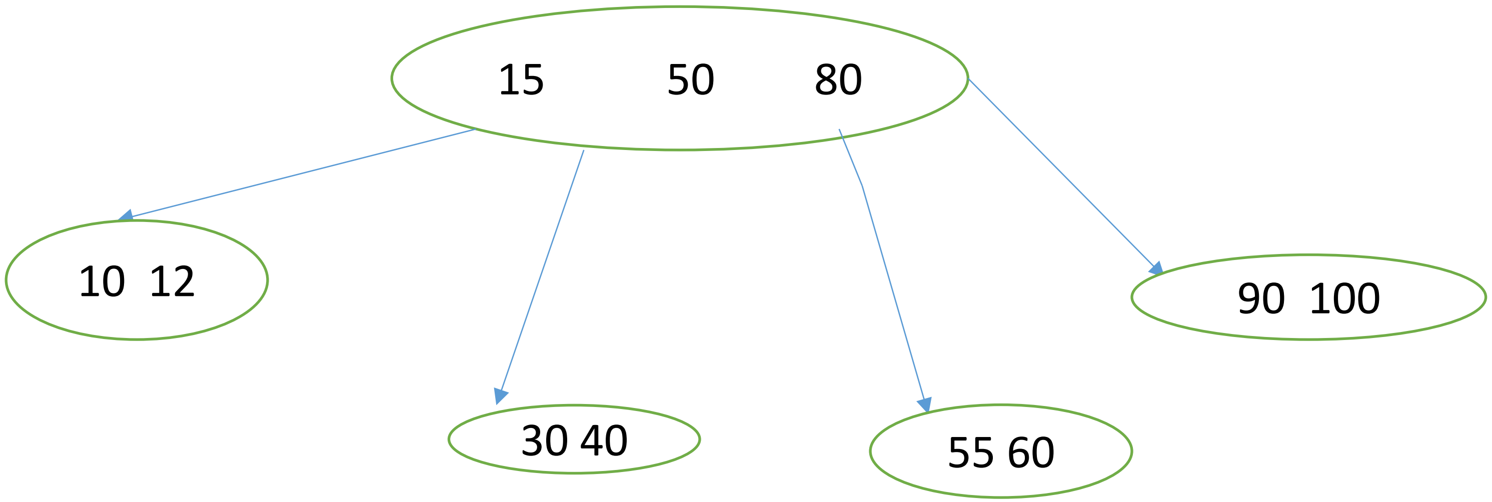


- 15



Building B-Tree

- Add 55



Example

- 15, 30, 45, 1, 3, 7, 90, 105, 34, 37, 42, 48, 54, 69, 60, 74, 100, 84, 89, 9, 22, 24

- // defining the structure of the node

```
struct btree_node {
  int data_item[n-1];
 int counter;
  struct btree_node *link[n];
 struct b-tree_node *father;
};
```

struct btree_node *root_node= NULL;

- // creating a Node
- **struct** btree_node *create_node(**int** data_item){
-   **struct** btree_node *new_node;
-   new_node = (**struct** btree_node *)malloc(**sizeof**(**struct** btree_node));
-   new_node -> data_item[0] = data_item;
-   new_node -> counter = 1;
-   new_node -> link[0] = NULL;
-   new_node -> link[1] = NULL;
-   **return** new_node;
- }

Insertion procedure

- **Find the appropriate node**: call to a recursive search function where we pass the address of the root node, it searches the node and returns an integer value "**position**"
- **Position** gives the index of either the **subtree** or the data_item found
- If the node is full then:
- **Create two more nodes**
- **First n-1/2 items will go in the first node**, middle will remain in the original node and **rest n-1/2 will go to the second node**
- Lots of pointers needs to be changed and set.

Detailed algorithm to insert data items in a B-tree

1. **Declare the structure of B-tree node** (taking some order m)

2. Define constant m

3. **Function to create a node**: initialize all address fields as NULL and return address of the node

4. **Create root-node** (can be kept as global)

5. **Function to add value in an existing node (assuming** node address is known and no of elements are less than m-1).

6. **Function to search in B-tree**: simple node search, if not found, make recursive call to search in the child node

Cont.

- **Insert the data in the node once we found the place**:

Three scenarios may occur:

1. **The node contains less than m-1 elements** : call node_insert
2. **Node contains m-1 items**:

a. **Node is a root node**:

- create two more nodes
- Transfer left (m-1)/2 data items to one node called left node
- Transfer rest (m-1)/2 data items to another node called right node
- Adjust the pointers of root node accordingly

- **If it is not the root node**:

1. **Check the father node, if father node not full**:

- create two more nodes
- Transfer left (m-1)/2 data items to one node called left node
- Transfer rest (m-1)/2 data items to another node called right node
- Move middle data key to father node and adjust the pointers

2. **If father node full**:

- Check father of father and repeat the same process

Deletion in B-Tree

Two methods:

1. **Just mark the key/record deleted**
2. **Actually delete the key/record**

We have already examined the first method. It works in the same way as it does in top down multiway search tree.

We will examine actual deletion in B-Tree

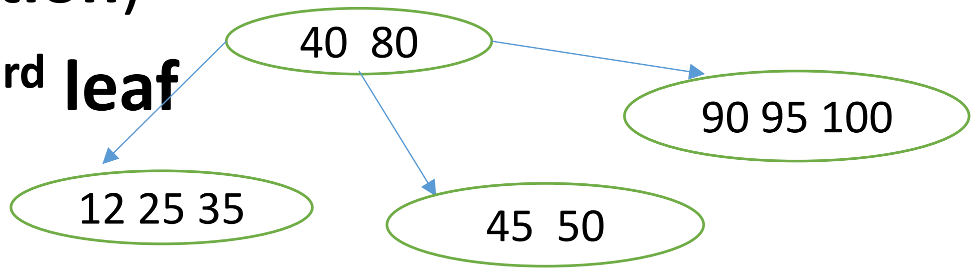
Deleting a key in B-tree (Example 1)

- **While deleting a key from B-tree, we must maintain the properties of B-tree i.e. every node has atleast (n-1)/2 keys except root and tree is balanced**
- Different cases: (example order 5)

1. Deletion from a leaf:

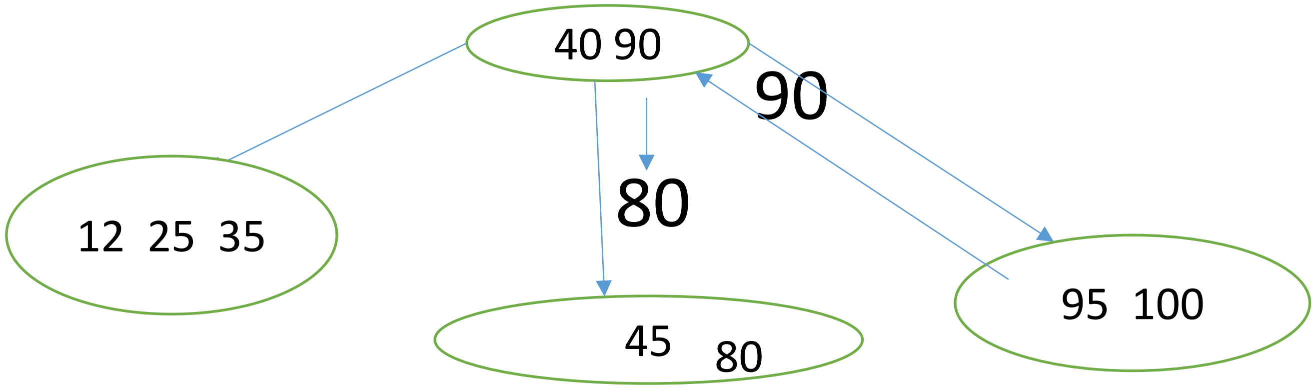
Case 1: If leaf contains ***more than*** (n-1)/2 keys, simply delete the key and compact the node (**Simple deletion**)

Case 1: **means deleting from 1st or 3rd leaf**



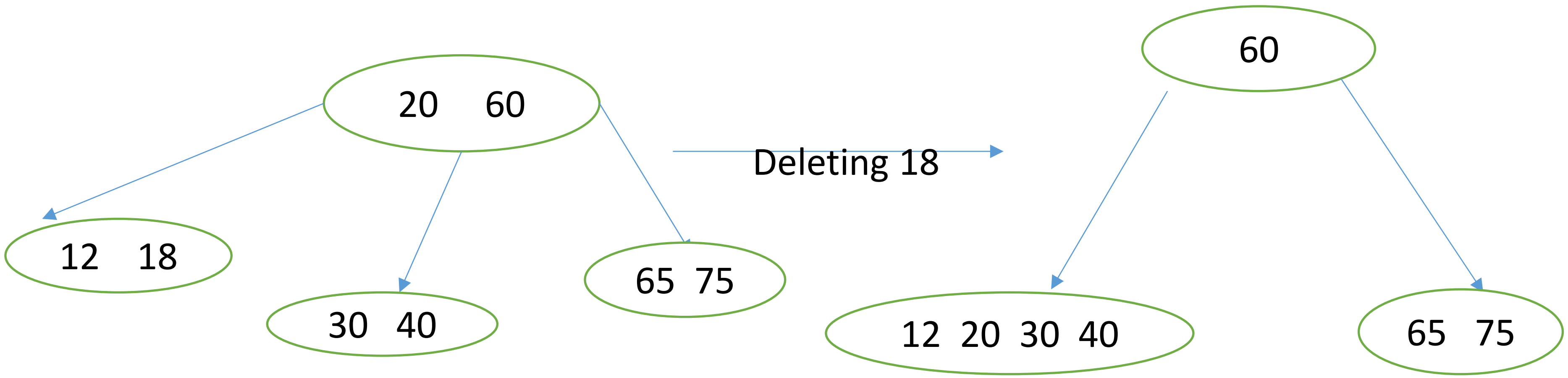
Deleting from B-Tree leaf

- **Case 2**: If leaf has (n-1)/2 keys, then examine the node's younger brother or elder brother and if anyone contains ***more than*** (n-1)/2 keys, move the extra key from brother to father and from father to this node (**taking one key from father and father key is replaced by brother)**
- *Deleting 50*



Deleting from B-Tree leaf

- **Case 3**: If both the brothers have minimum number of keys, then concatenate the node with one of its brother i.e. **merge the two nodes taking one key from father node**

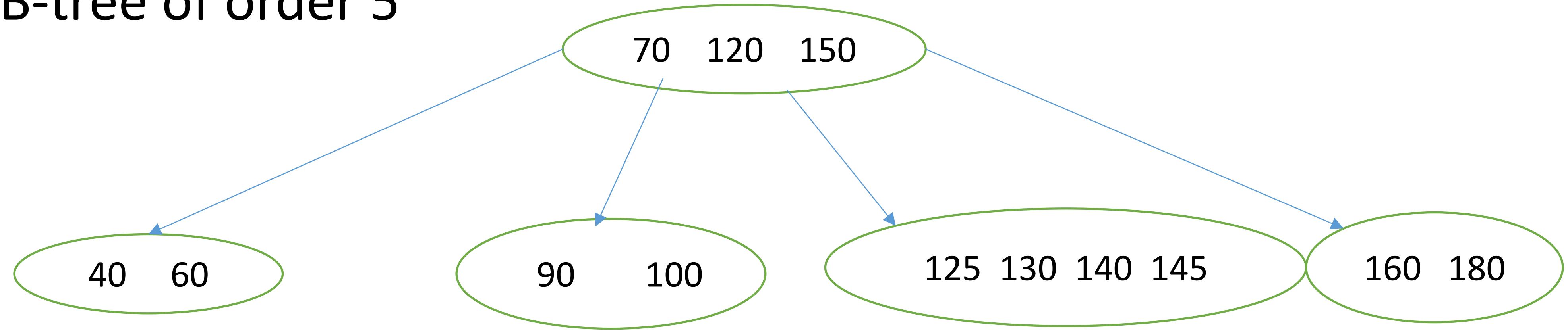


Deleting a key in B-tree, continued..

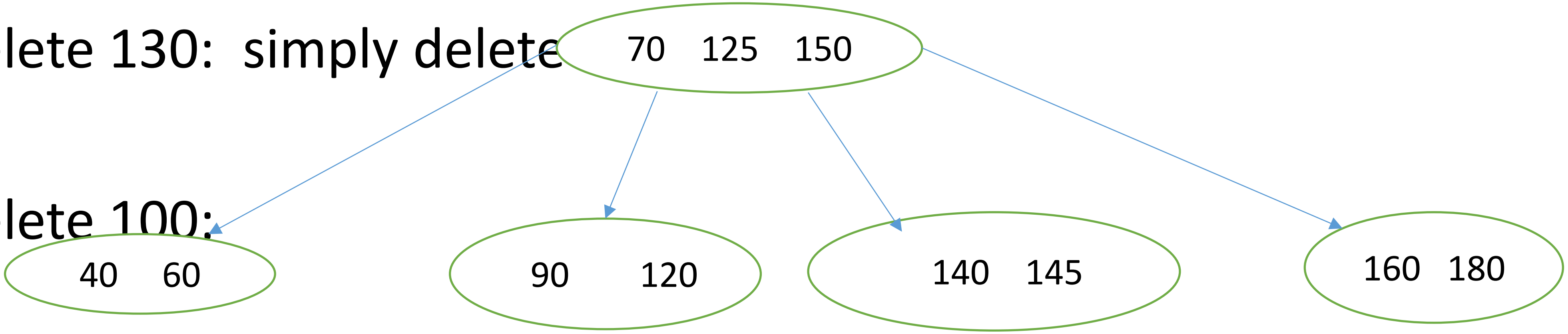
Case 4: But, if father node contains only minimum number of keys and it does not have any extra key to spare then in that case it can borrow from its father and brother

Example 2

- B-tree of order 5



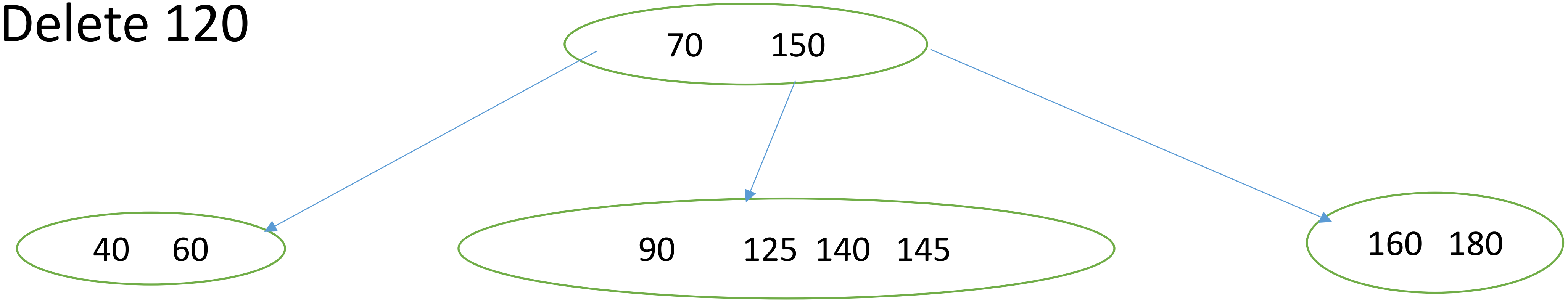
- Delete 130: simply delete



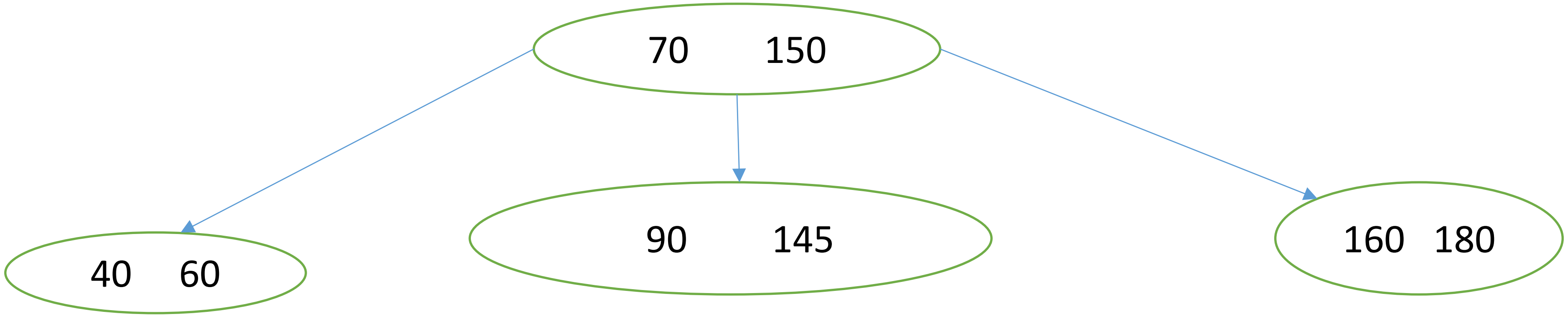
- Delete 100:

Cont…

- Delete 120

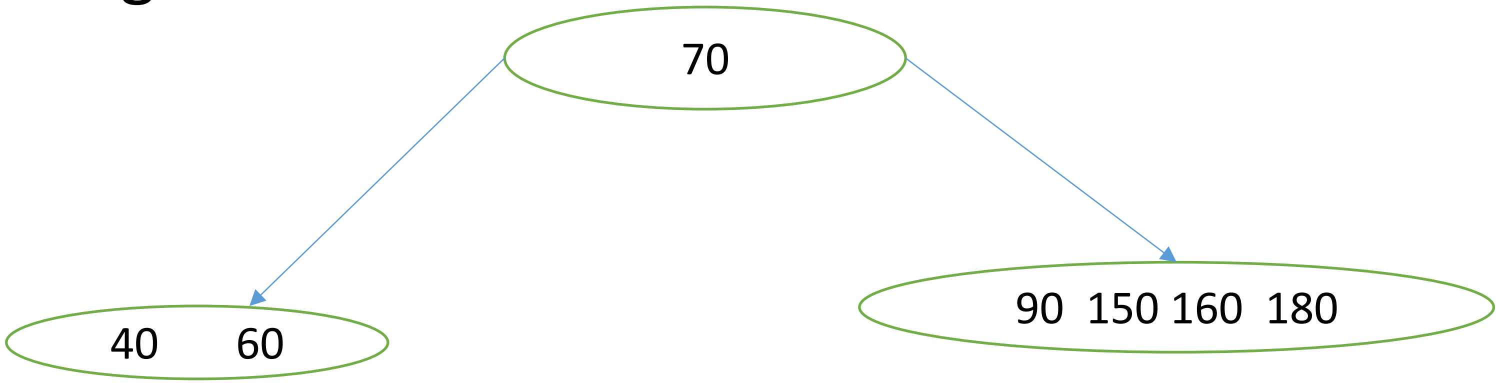


- After deleting 125 and 140, we will have



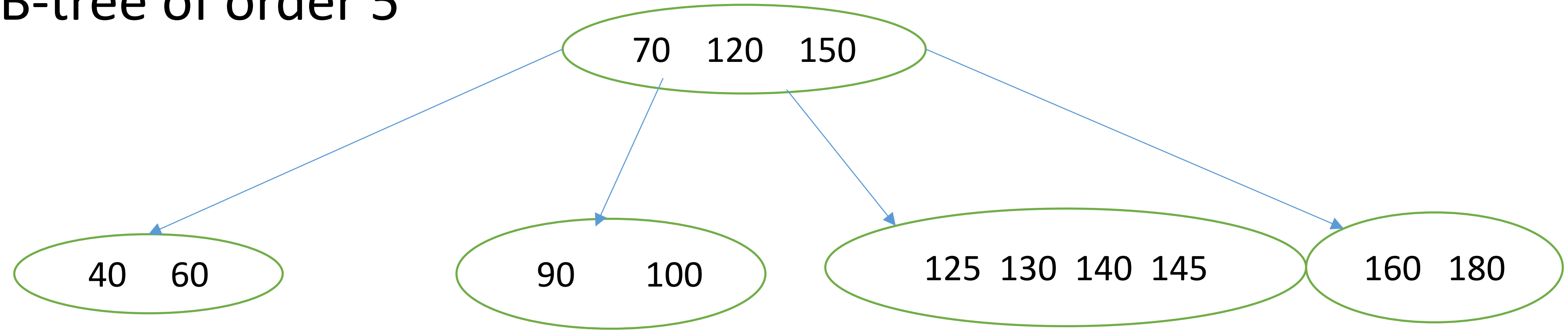
Cont..

- After deleting 145:

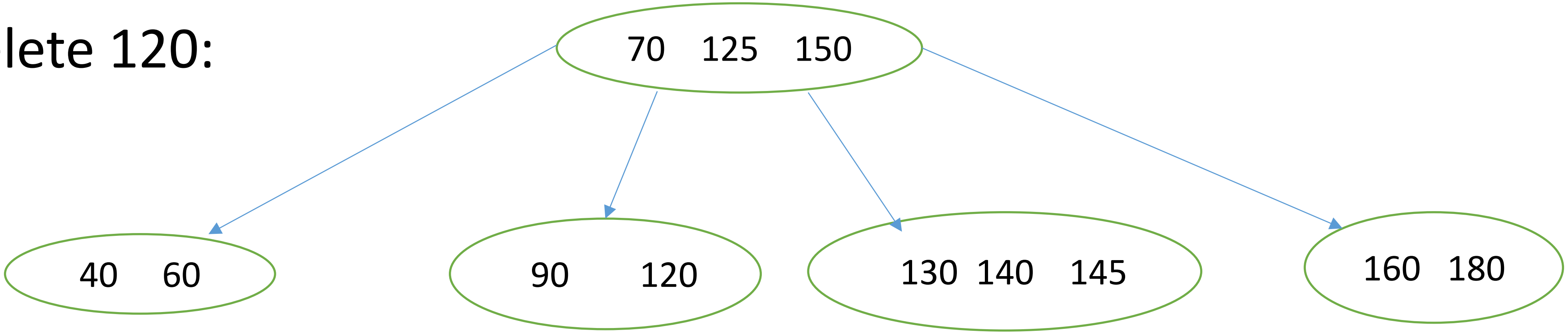


Example 3

- B-tree of order 5

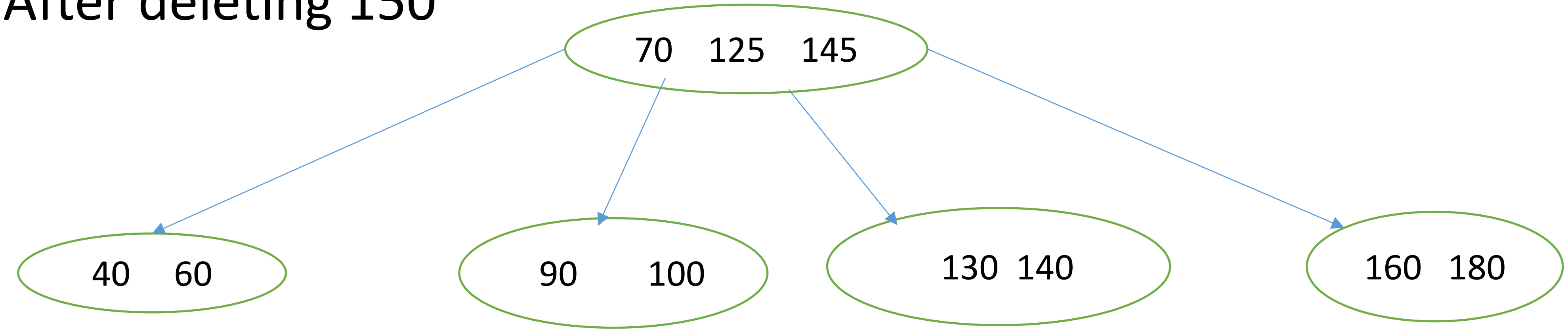


- Delete 120:

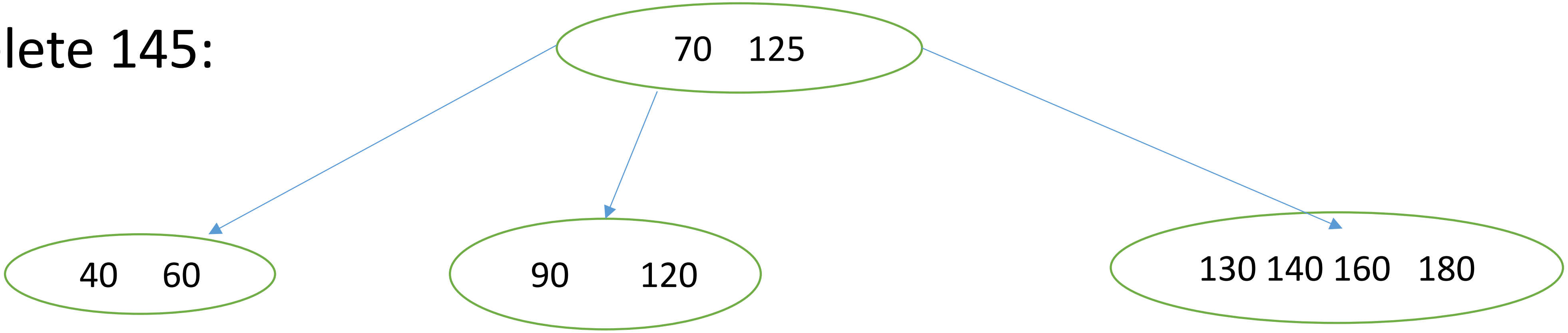


Cont…

- After deleting 150



- Delete 145:



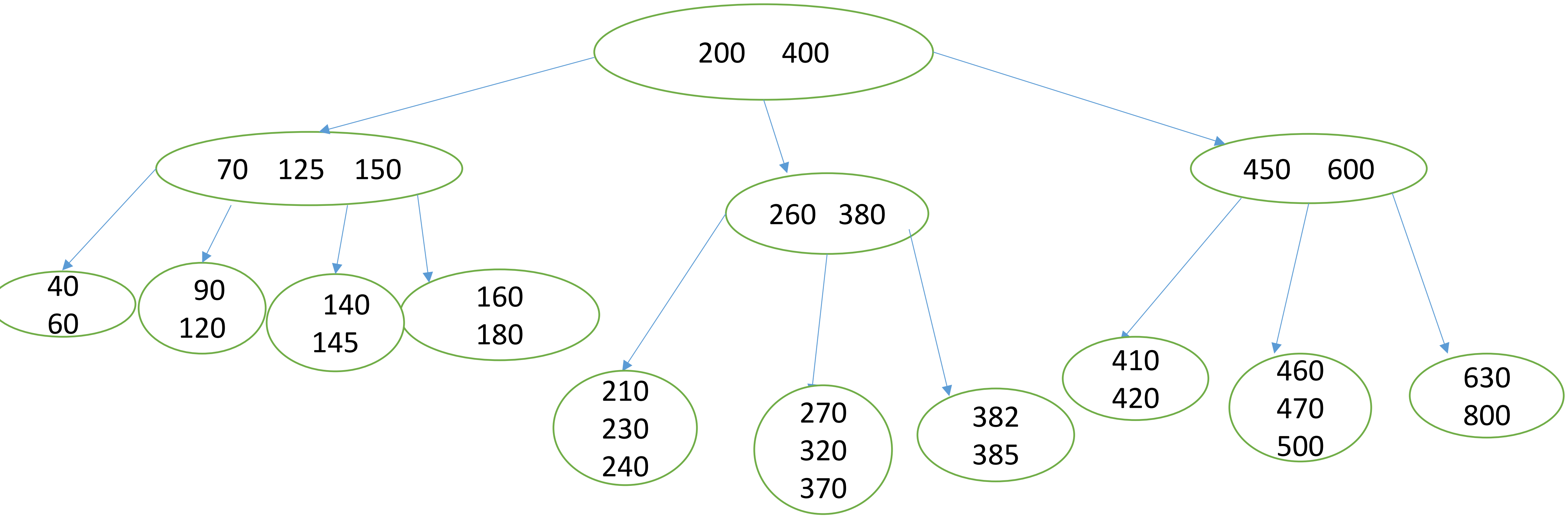
Example 3 cont…

- After deleting 70

- Delete 125: Delete 130: delete 140: delete 90

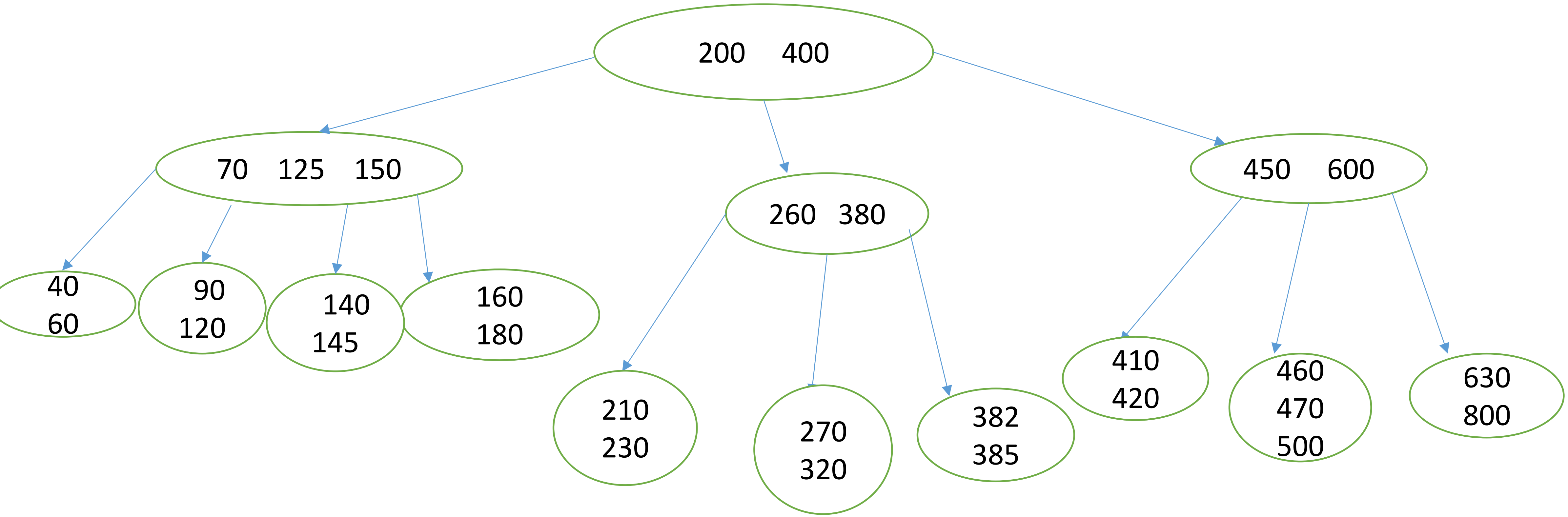
Example 4

- B-tree of order 5 and height 2



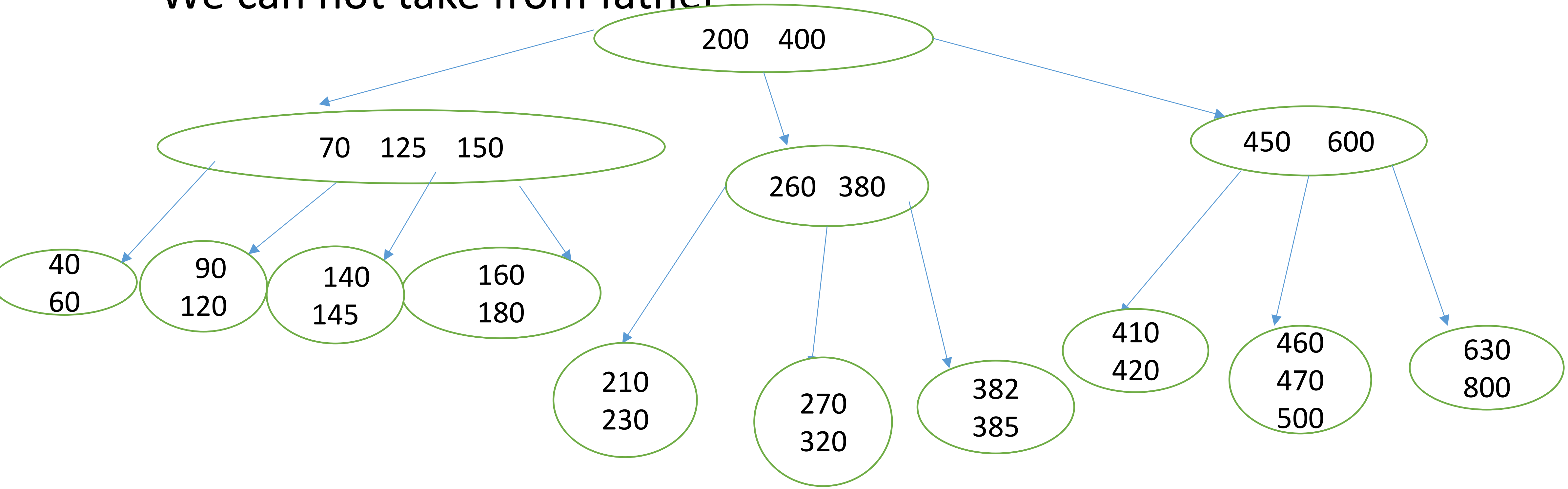
Cont..

- After deleting 240 and 370



Example

- Deleting 260 (the operations will be same for any key in this branch)
- We can not take from father



Cont…

Deleting a key from non leaf node:

- Replace it by its immediate successor and it will be in the leaf node
- So, deletion from the non-leaf node is equivalent to deleting from the leaf node

Applications of B-tree

- B-trees are used in databases to store indexes that allow for efficient searching and retrieval of data.
- B-trees are used in file systems to organize and store files efficiently
- Hard drives, flash memory, and CD-ROMs are examples of storage devices that use B-Trees to avoid sluggish, clumsy data access.
- Multilevel indexing is possible with the indexing feature.