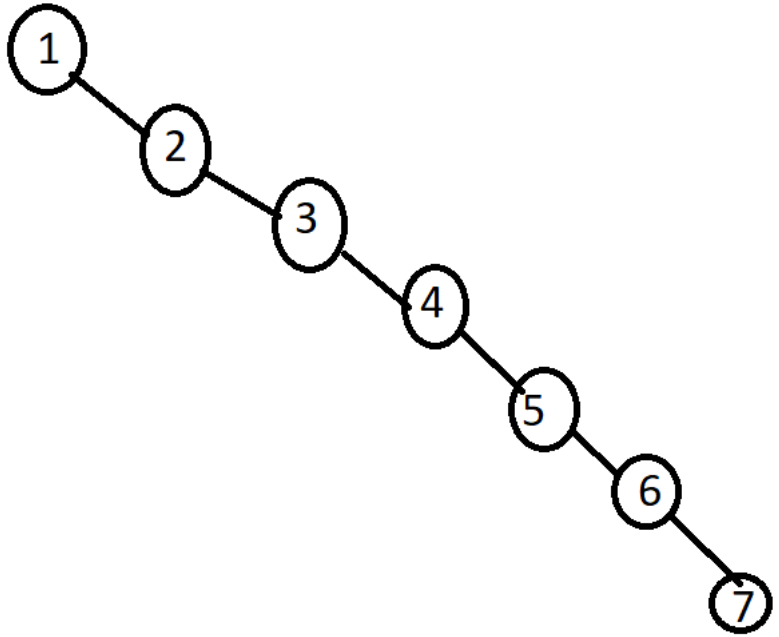
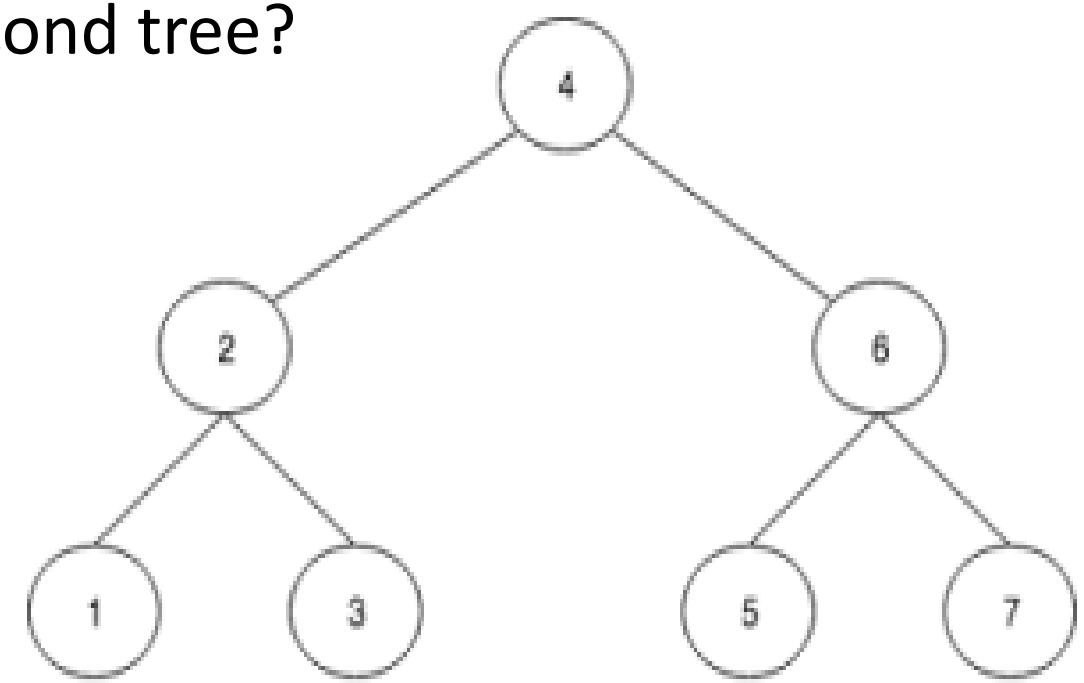
B Tree

Dr. Tanvir Ahmed

Background

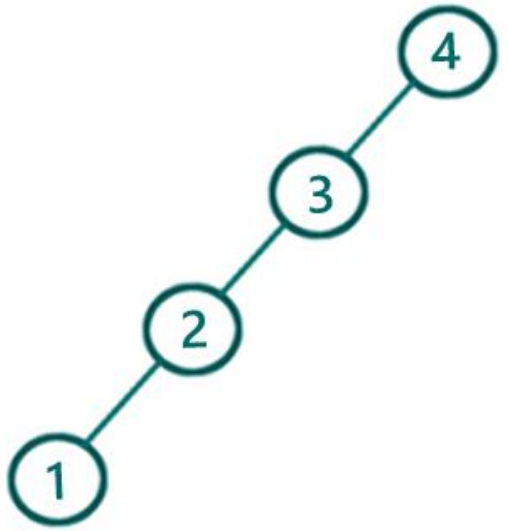
- Let us build a binary search tree with the following numbers: 1, 2, 3, 4, 5, 6, 7.
- You will get the left tree from the following list, which is completely unbalanced.
- How many nodes would you need to access if you need to search for 7? How about searching for 7 in the second tree?





Background

- **B tree is another tree-based data structure**
- **Let us review some issues with previous data structures:**
  - **unbalanced Binary Search Trees (BST)**
  - we like binary search trees because we can find things "quickly" (O(log n) on average)
  - unfortunately, the worst case is still O(n):
  - If we insert a sorted list of items into a tree
    - the first (least) item goes at the root
    - the next item is larger, so goes to the right
    - the next item is larger still, so goes to the right of the right
    - ... and so on
  - the resulting tree very much resembles a linear list
  - this happen very rarely if the items are inserted in truly random order



Background

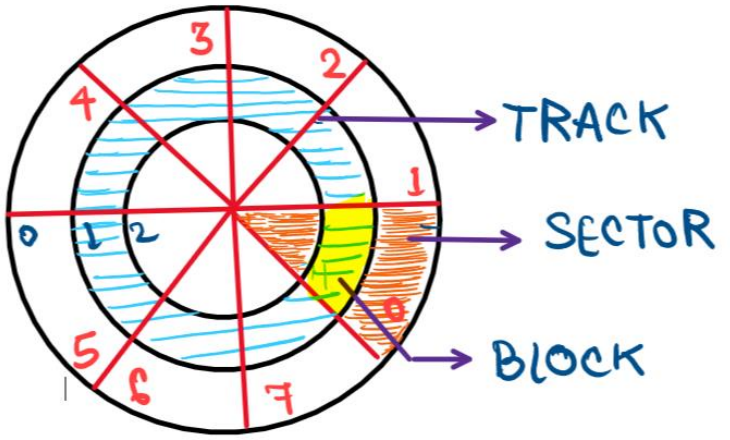
- **balanced trees**
- A balanced tree can help us to achieve O(log n) search
- search, addition, and removal time are proportional to height, so balanced trees are nice
- we can take an unbalanced tree and **rebalance** it:
  - take elements in a random order and insert them into a new tree
- we can also add new restrictions to our trees so they stay balanced:
  - incremental re-balancing (AVL, red-black, splay trees)
  - all leaves have the same depth **(B-trees)**

How B tree can help?

- **B-tree is heavily used for indexing data in a database.**
- **As data are stored in disk, let us get some idea about it**

Concept of disks

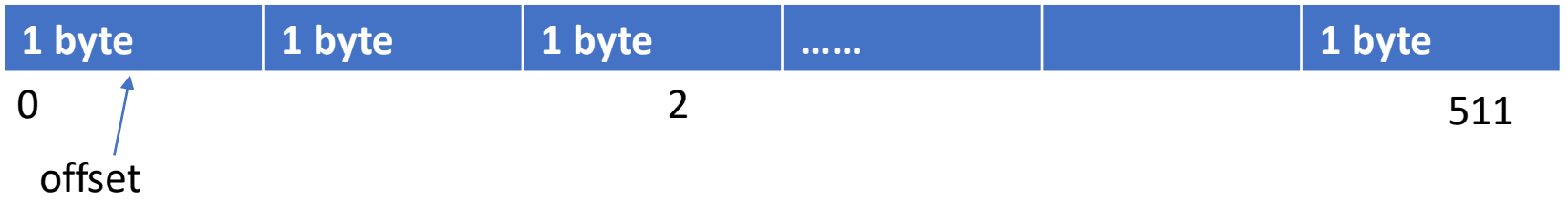
- **As data in the database is stored in disk, let us get an overview on disks**
- **A disk has tracks (the circular lines), sectors (the red lines), and blocks (the intersection boxes in the following picture)**
- **Each block has a block address: (Track number, sector number)**
  - **For example, the yellow colored block address is: (1, 0) as it is located at tract 1 and section 0.**
- A block can store a specific amount of bytes



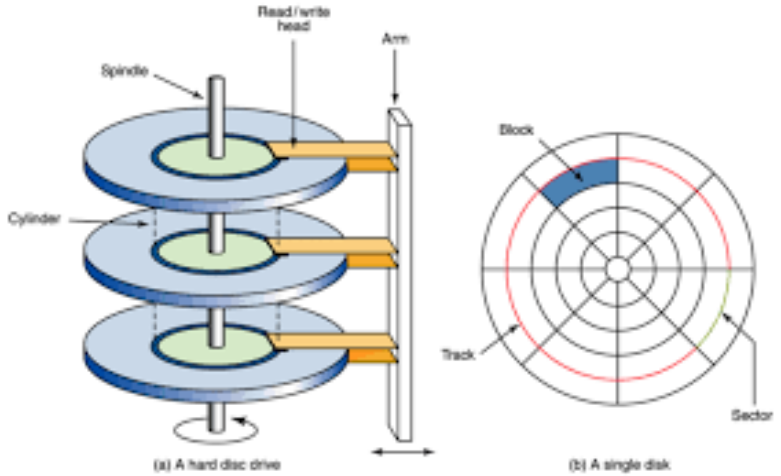
- For example, let's say 512 bytes will be stored in each of our example disk's block
- Generally, your operating system will read data from a block at a time as it is efficient

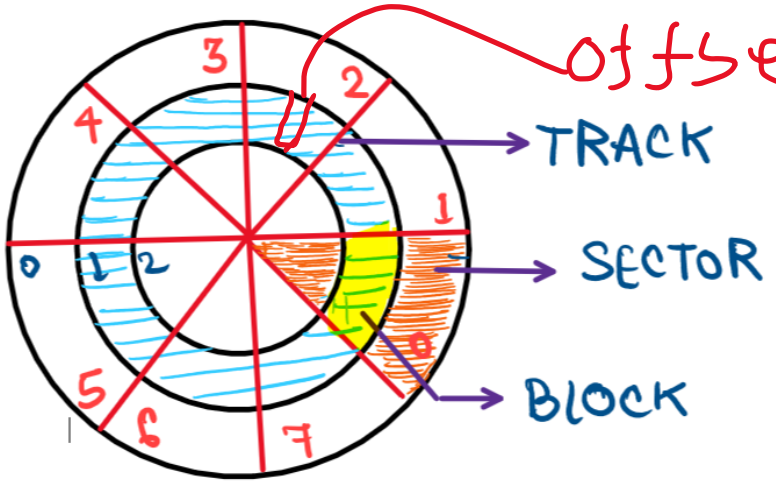
Let us see how a B tree can help

- **Let's take a block which is 512 bytes, it will look like the following:**



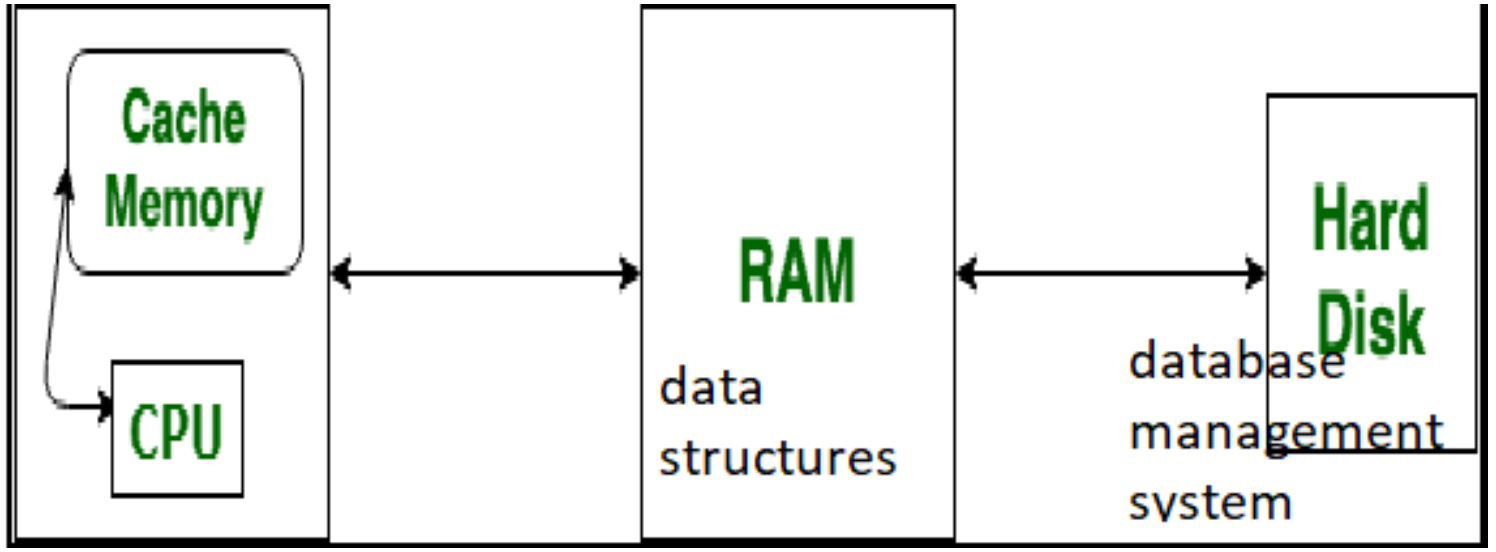
- In the above block, each byte has its address. This is known as offset address (address of that byte)
- The address starts from zero and ends at 511 (if block size is 512 in our example)
- For example, in our above picture 2 is the offset of the 3rd byte in the block
- So, to access any particular byte, we need to know the track number, sector number and offset
- In order to read a particular bock, the disk needs to rotate and the head needs to go to particular sector





Concept of Main Memory and data structure

- I hope you already know that any data need to be processed are brought to the main memory (RAM)
- We use different data structures to store our data in ram so that we can process them efficiently
- The data stored in the disk is managed by the database management system (DBMS). So, we can efficiently access the information in disk



Concept of database management system(DBMS)

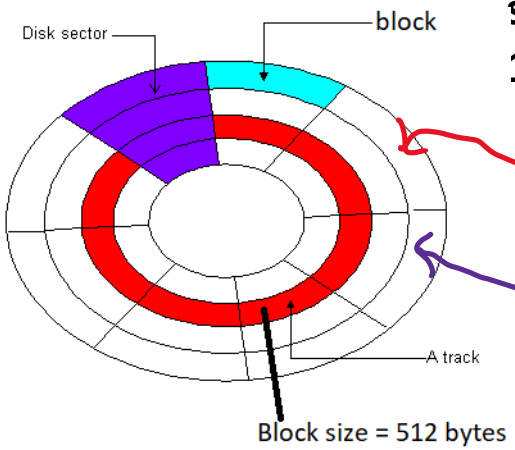
- Consider the following student table and size of each field in the table.

Column name	Size (in bytes)
sid	10
Sname	50
Dept	10
Gpa	8
HomeAddress	50

Total size for a student record = 128 Bytes

- If a record takes 128 bytes, how many records you can store in a block in disk?
  - **512/128 = 4 records/block**
- So, each 4 records will take a block.
- See the red marked rows are taking a block and purple colored records are taking a block.
- So, for 100 records of our database table, we will need **100/4 = 25 blocks**

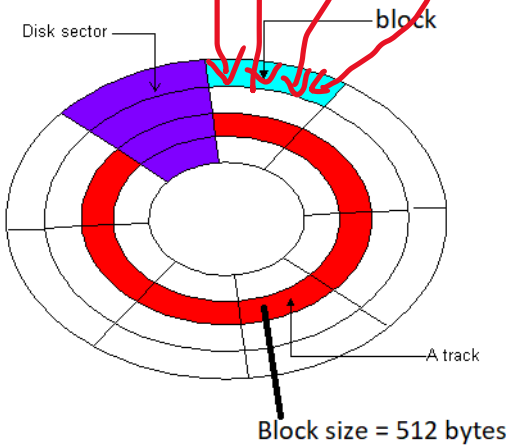
Sid	Sname	dept
1	Adam	…
2	Jones	…
3	Andrew	…
4	Huzaifa	…
5	Nusair	…
6	Manha	…
7	Nadia	…
8	Fatima	…
9	Ayat	…
…..	…	…
…..	….	….
100	Sarah	…



Concept of Index

- How many blocks we might need to access if we need to search for a particular student by their id or name?
  - Maybe all 25 blocks in worst case.
  - We might need to check block by block for the record
- Can we reduce the time?
- Yes, we can index. The other table here shows a concept of index, that stores the key for a record and address of the record in the disk.
- Now where to store the index?
  - We also store it in the disk.
- How much space needed for our index?
- Let's say, each index record takes 10+6 = 16 bytes
- So, how many index entries per block?
  - each block size/ entry size = 512/16 = 32 index
- For 100 entries, we need 100/32 = 3.2 ~ 4 blocks needed to store our index

Sid	pointer
1 (10 bytes)	(6 bytes)
2	
3	
4	
…. 100	….



Sid	Sname	dept
1	Adam	…
2	Jones	…
3	Andrew	…
4	Huzaifa	…
5	Nusair	…
6	Manha	…
7	Nadia	…
8	Fatima	…
9	Ayat	…
…..	…	…
…..	….	….
100	Sarah	…

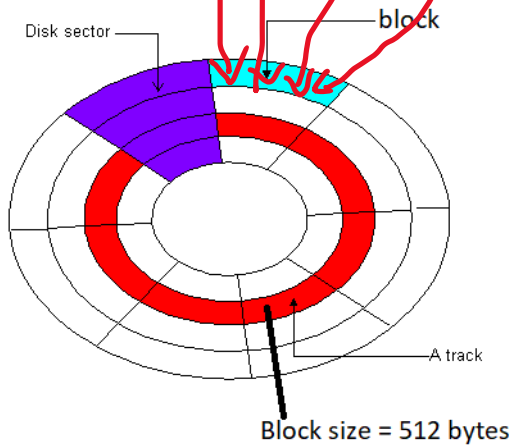
So, did the indexing help?

- Yes! If you want so search any item by ID, you need to go to index.
  - So, at most we need access 4 blocks to get the id and pointer
  - And then we simply access the block of actual data.
  - So, total 4+1 blocks at most (not 25 without index)

That is the benefit of index!

Sid	pointer
1 (10 bytes)	(6 bytes)
2	
3	
4	
…. 100	….

Sid	Sname	dept
1	Adam	…
2	Jones	…
3	Andrew	…
4	Huzaifa	…
5	Nusair	…
6	Manha	…
7	Nadia	…
8	Fatima	…
9	Ayat	…
…..	…	…
…..	….	….
100	Sarah	…



Multi level index

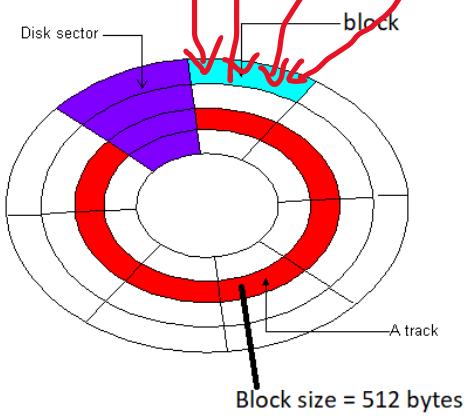
- How about if we have 1000 records? We will need 250 blocks for records
- For index we will need 40 blocks (as we needed 4 blocks for 100 entries)
- So, index itself got larger and even we will have to spend pretty good amount of time just searching index itself
- Now, we can have index of index
  - That is called multi-level index

Sid	pointer
1	
33	
…	

Sid	pointer
1 (10 bytes)	(6 bytes)
2	
3	
4	
…. 1000	….

Sid	Sname	dept
1	Adam	…
2	Jones	…
3	Andrew	…
4	Huzaifa	…
5	Nusair	…
6	Manha	…
7	Nadia	…
8	Fatima	…
9	Ayat	…
…..	…	…
…..	….	….
1000	Tausif	…

- In this level of index, for each block of the low level index, we have an entry in this new level index
- As a block is 512 and an entry is 16 bytes
  - A block can store 512/16 = 32 entries of the index
  - And we simply need one entry in our new level index to point this 32 entries



Multi level index

- So, how many entries and blocks needed for our top level index?
  - If you remember, we needed 40 blocks for our low level index
  - So, upper-level index will need to point 40 blocks (40 entries)
  - So, we simply need 2 blocks for the top level index! (40/32 enries ~ 1.2)

Sid	pointer
1 (10 bytes)	(6 bytes)
33	
….	

2 blocks

Sid	pointer
1 (10 bytes)	(6 bytes)
2	
3	
4	
…. 1000	….

40 blocks

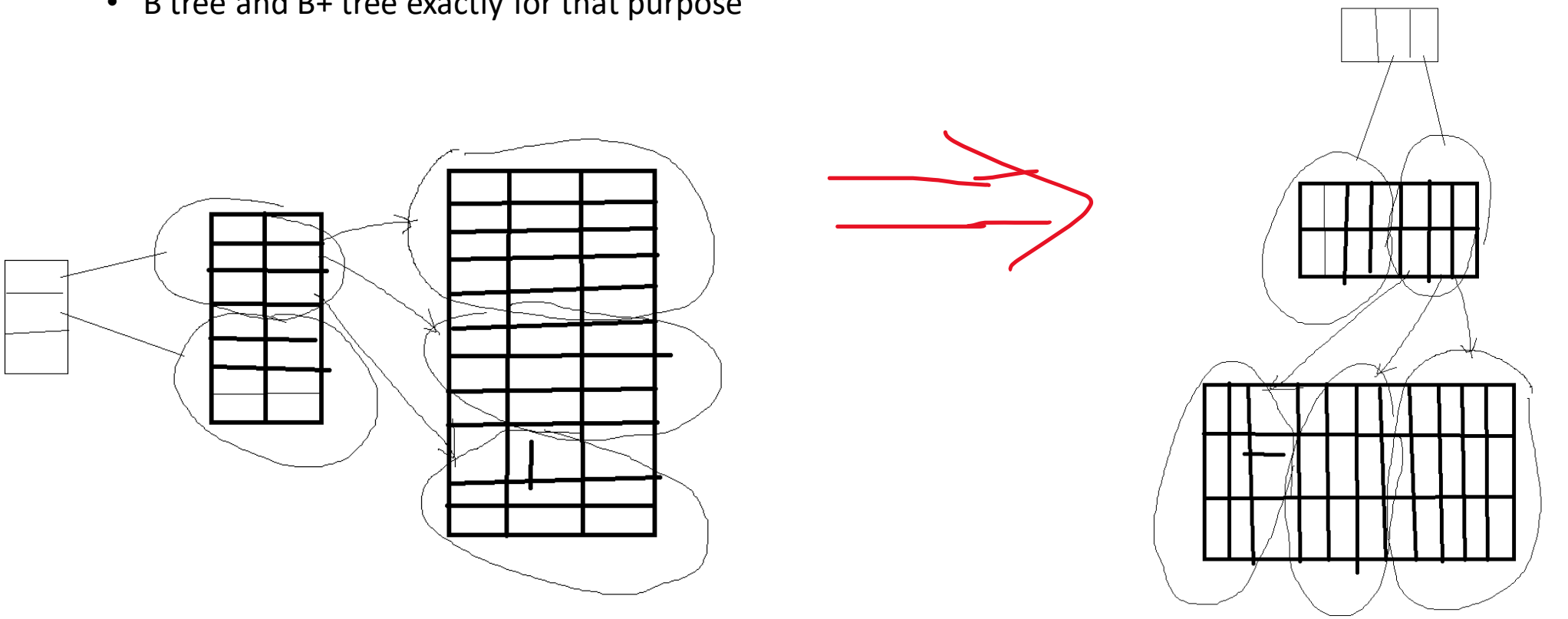
Sid	Sname	dept
1	Adam	…
2	Jones	…
3	Andrew	…
4	Huzaifa	…
5	Nusair	…
6	Manha	…
7	Nadia	…
8	Fatima	…
9	Ayat	…
…..	…	…
…..	….	….
1000	Tausif	…

250 blocks

- Now, if we want to search for an item, we start from
  - top level index(max 2 blocks)
  - Then directly go to that block of lower level index pointed by our upper level
  - And from there we directly go to the record (just 2+1+1 = 4 block access!) (Wow!!!)

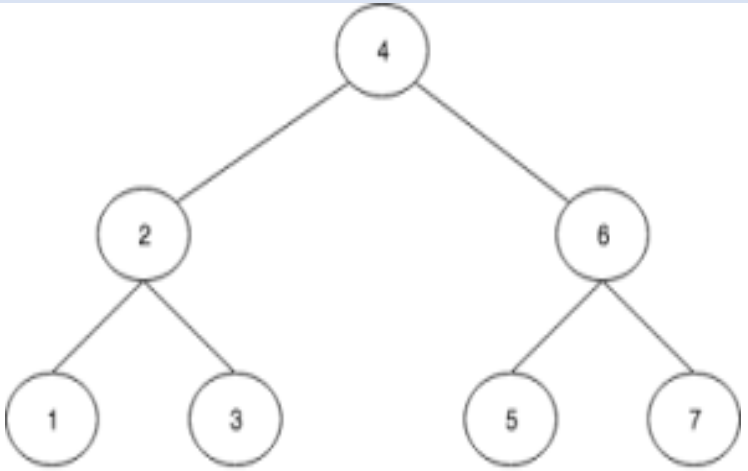
The main concept of B tree and B+ tree

- The main concept of B tree and B+ tree actually comes from the multi-level index.
- So, if we conceptually draw the indices, it will look like the figure on the left side. If you just draw it in to bottom, we see the figure on the right side (looks like a tree!)
- Does it mean we have to update the index manually if more data is added or any data is removed?
  - No, it will expand and adjust accordingly and we need to know how does that work!
  - B tree and B+ tree exactly for that purpose

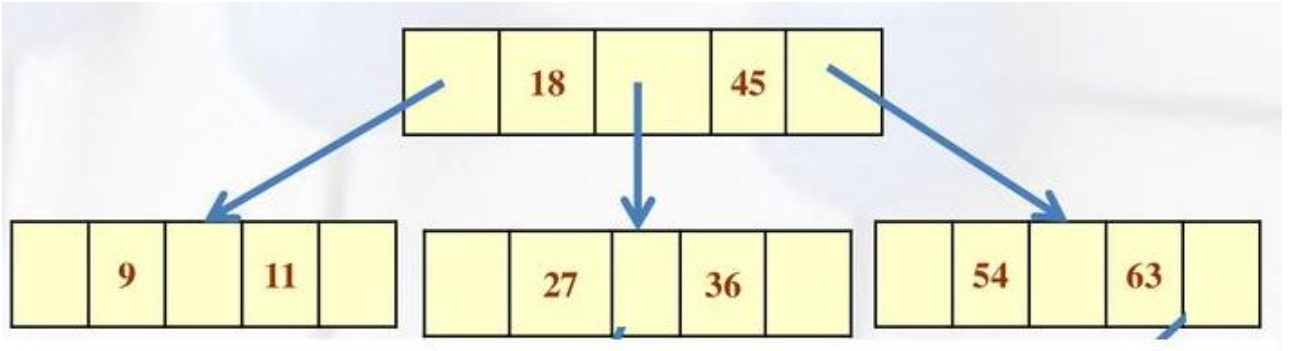


m-way (multiway) search tree

- Let us try to go back to our binary search tree.
  - In a BST, we can have **only one key** at each node
  - Also, each node, can have **only 2 children.**
  - That's why it is called BST.



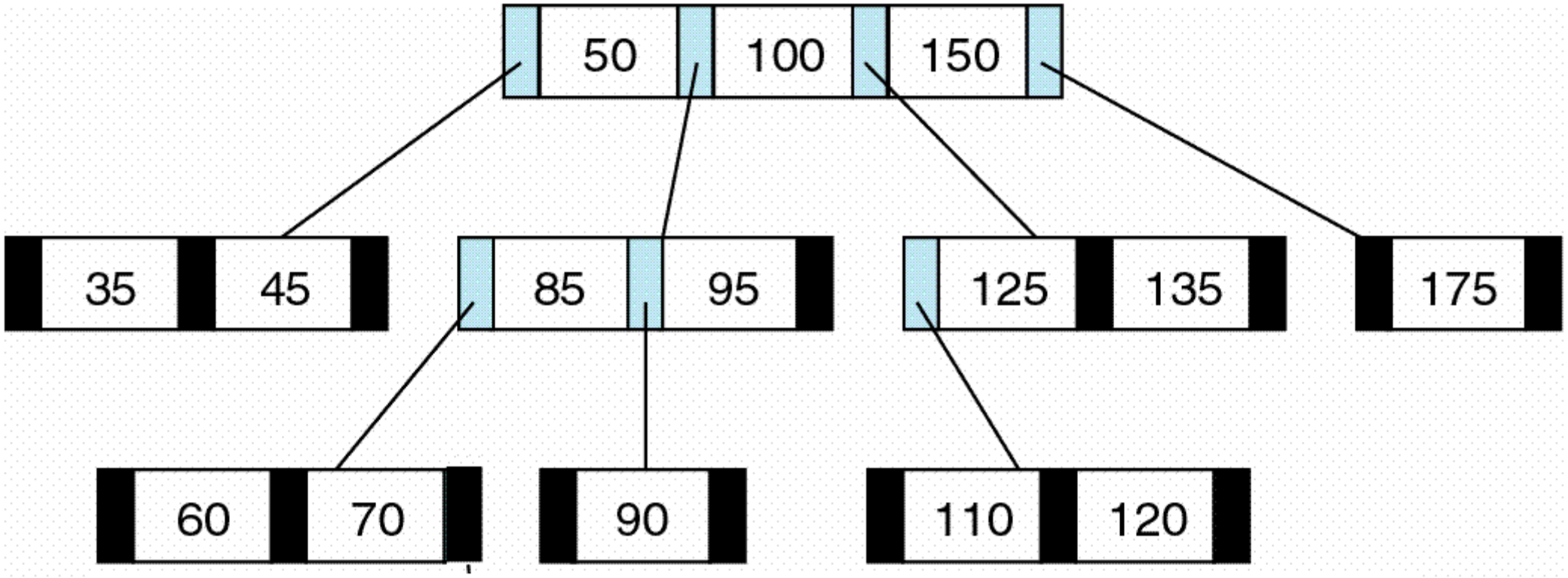
- So, can we have a tree that has more than 1 key and more children?
- See the example.
- Each node contains 2-keys and maximum 3 children!
- So, it is called 3-way search tree
- See, it follows almost same way of BST searching
- M-way means, (m-1) keys for each node and at most m children



Multiway

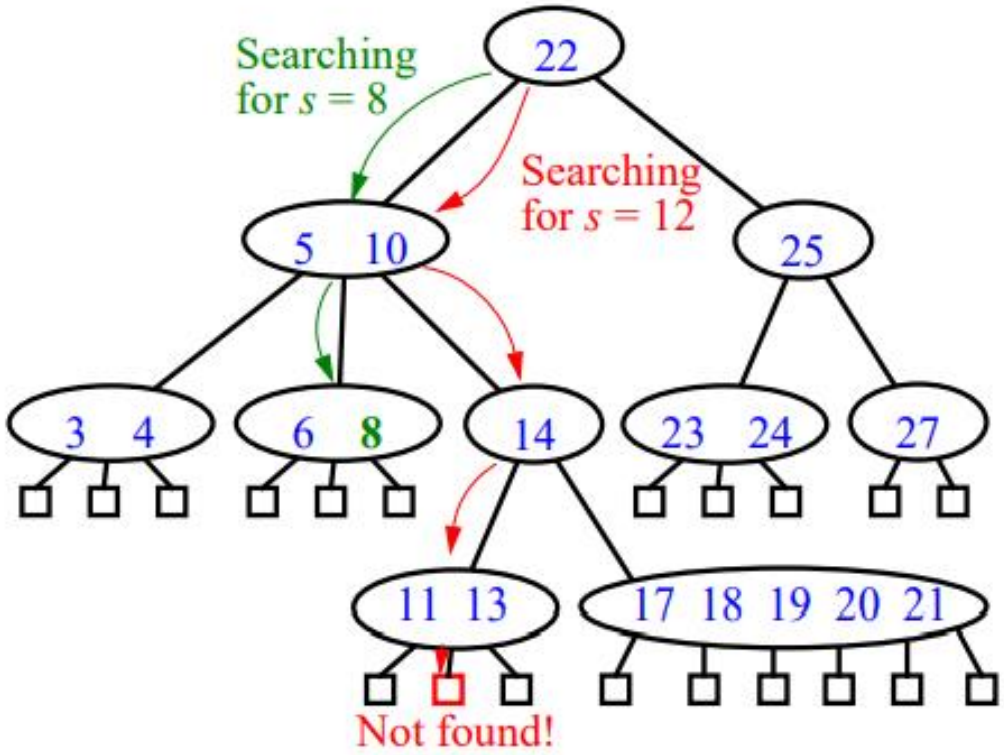
- If a node contains k items, (let these be I1, I2, ... and Ik, in numeric order), then that node will contain k+1 subtrees.
  - (Let these subtrees be S1, S2, ... and S_{k+1}.)
- In order to create a search tree with a reasonable order, we have to place constraints on the values stored in each subtree.
- In particular we have:
  - All values in S1 are less than I1.
  - All values in S2 are less than I2, but greater than I1.
  - All values in S3 are less than I3, but greater than I2.
  - ...
  - All values in S_k are less than I_k, but greater than I_{k-1}.
  - All values in S_{k+1} are greater than I_k.
- From this point on, if I refer to a subtree to the left of a value, I_m, I am referring to the subtree directly to the left of the value, Sm.
- Also, I will refer to S_{m+1} as the subtree to the right of I_m.

An example of Multiway Search Tree



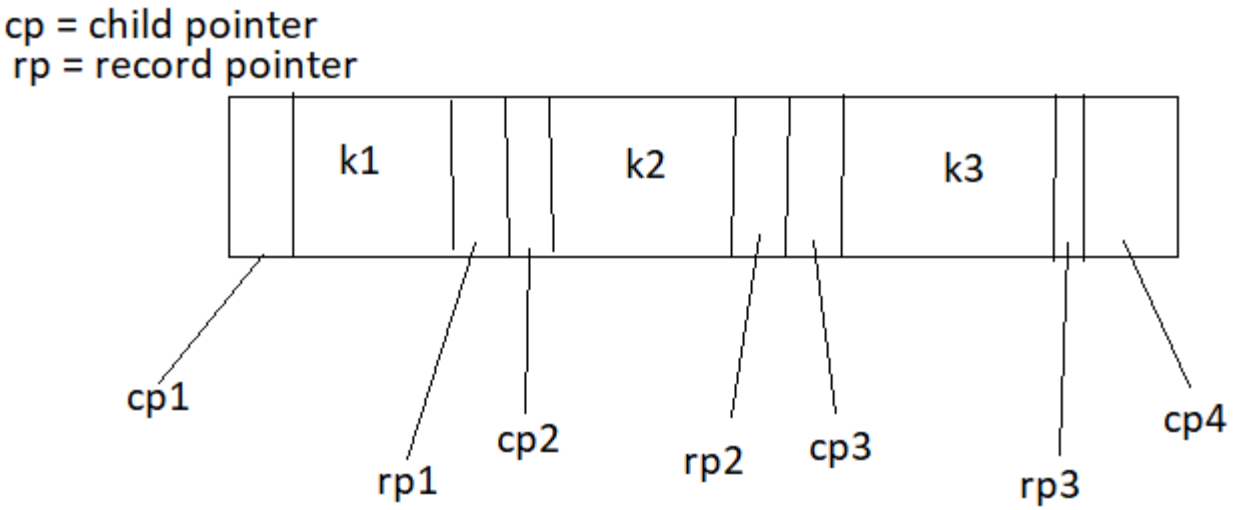
Searching in a multiway search tree

- Similar to binary searching
- If search key $s < k_1$, search the leftmost child
- If $s > k_{d-1}$, search the rightmost child
- That's it in a binary tree; what about if $d > 2$?
- Find two keys k_{i-1} and k_i between which s falls, and search the child v_i.



Can we use m-way search tree for multi level indexing?

- Yes. You just need to maintain record pointer for each key in the node structure
- See the following example of 4-way search tree, that maintains child pointer and record pointer
- But the problem with m-way search tree is that there is not well defined rule on insertion and utilizing the spaces in upper level before going the next level. So, the tree can become tall like BST



Sid	Record pointer
1	
2	
3	
4	
…. 1000	….

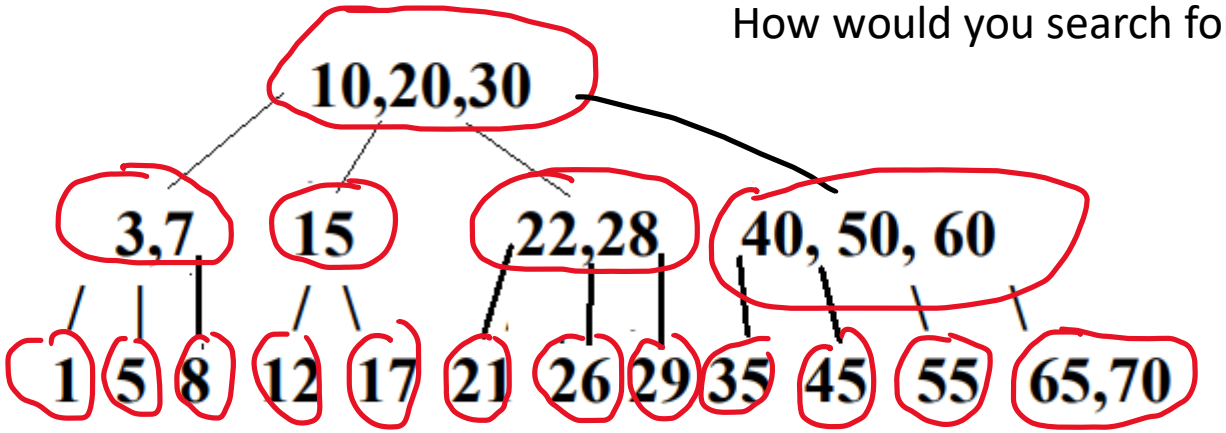
Sid	Sname	dept
1	Adam	…
2	Jones	…
3	Andrew	…
4	Huzaifa	…
5	Nusair	…
6	Manha	…
7	Nadia	…
8	Fatima	…
9	Ayat	…
….. …..	… ….	… ….
1000	Tausif	…

B-Trees (2-4 Tree)

- B-tree is a type of m-way search tree with specific rules.
- Let us do it for 2-4 trees. It is a B-tree with m = 4
- A 2-4 Tree is a specific type of multitree. Here are the specifications for a valid 2-4 Tree:
- 1) Each node has between 1 to 3 values
- 2) Each node has in between 2 and 4 children.
- 3) All leaves have the same depth

Here is an example of a 2-4 Tree:

How would you search for 21 in this tree?

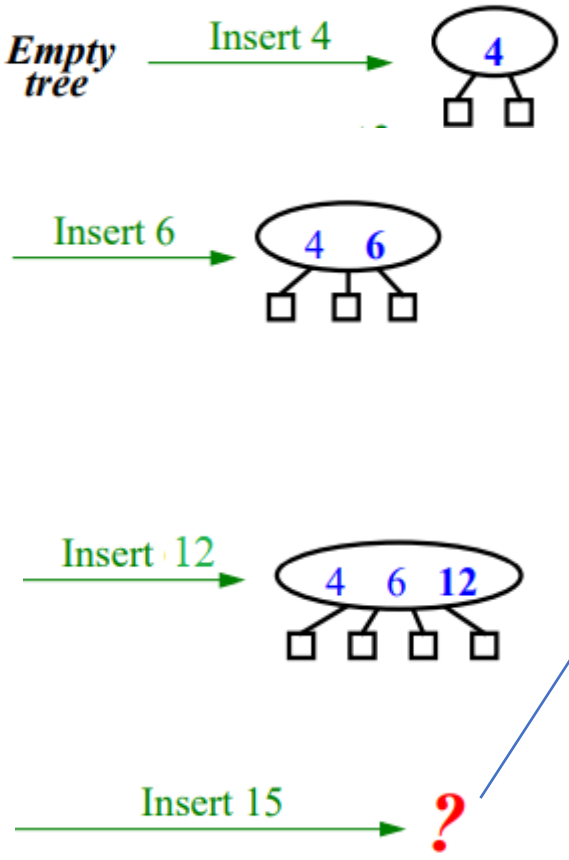


Insert in 2-4 tree

- We need to maintain the rules discussed in the previous slide
- Additionally, the creation process is bottom up.
- Add elements only to existing nodes (if possible)
- What problems may arise during insertion?
  - A node can become full (contains 3 values already)
  - You might say, we just create a node and insert it into next level. But, the problem with this is that not ALL of the external nodes will have the same depth after this.
  - Instead, we will do is attempt to push a value up to the parent of the inserted node.
  - We will see example:

Inserting in a 2-4 tree

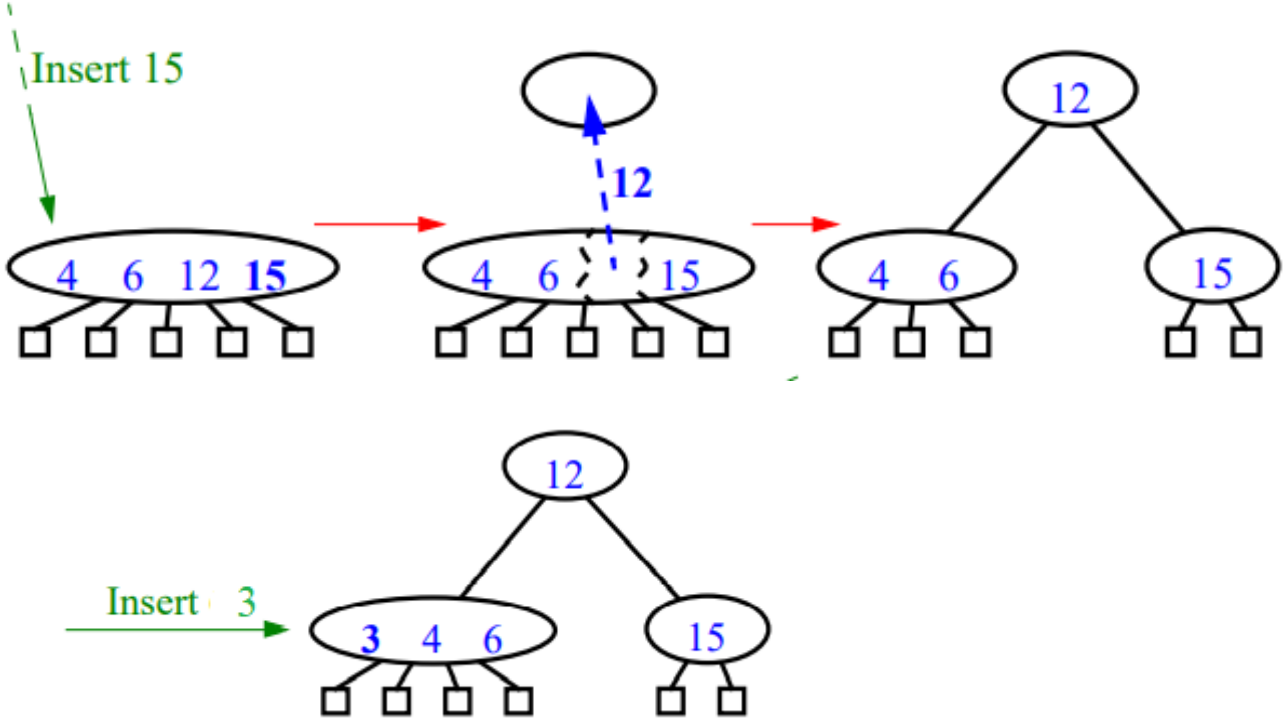
- Let us insert the following items: 4, 6, 12, 15, 3, 5, 17



- What if that makes a node too big?
  - *overflow*
- Must perform a split operation
  - replace node v with two nodes v_x and v_y
  - v_x gets the first two keys
  - v_y gets the last key
  - send the other key up the tree (parent)
  - if v is root, create new root with third key
  - otherwise just add third key to parent
- Example will clarify

Inserting in a 2-4 tree

- **Let us insert the following items: 4, 6, 12, 15, 3, 5, 17**

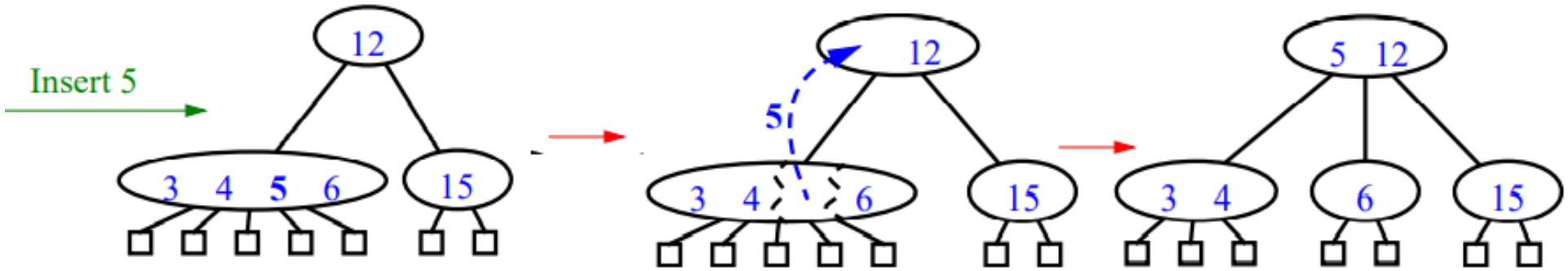


Can you insert 5

- What if that makes a node too big?
  - *overflow*
- Must perform a split operation
  - replace node v with two nodes v_x and v_y
  - v_x gets the first two keys
  - v_y gets the last key
  - send the other key up the tree (parent)
  - if v is root, create new root with third key
  - otherwise just add third key to parent
- Example will clarify

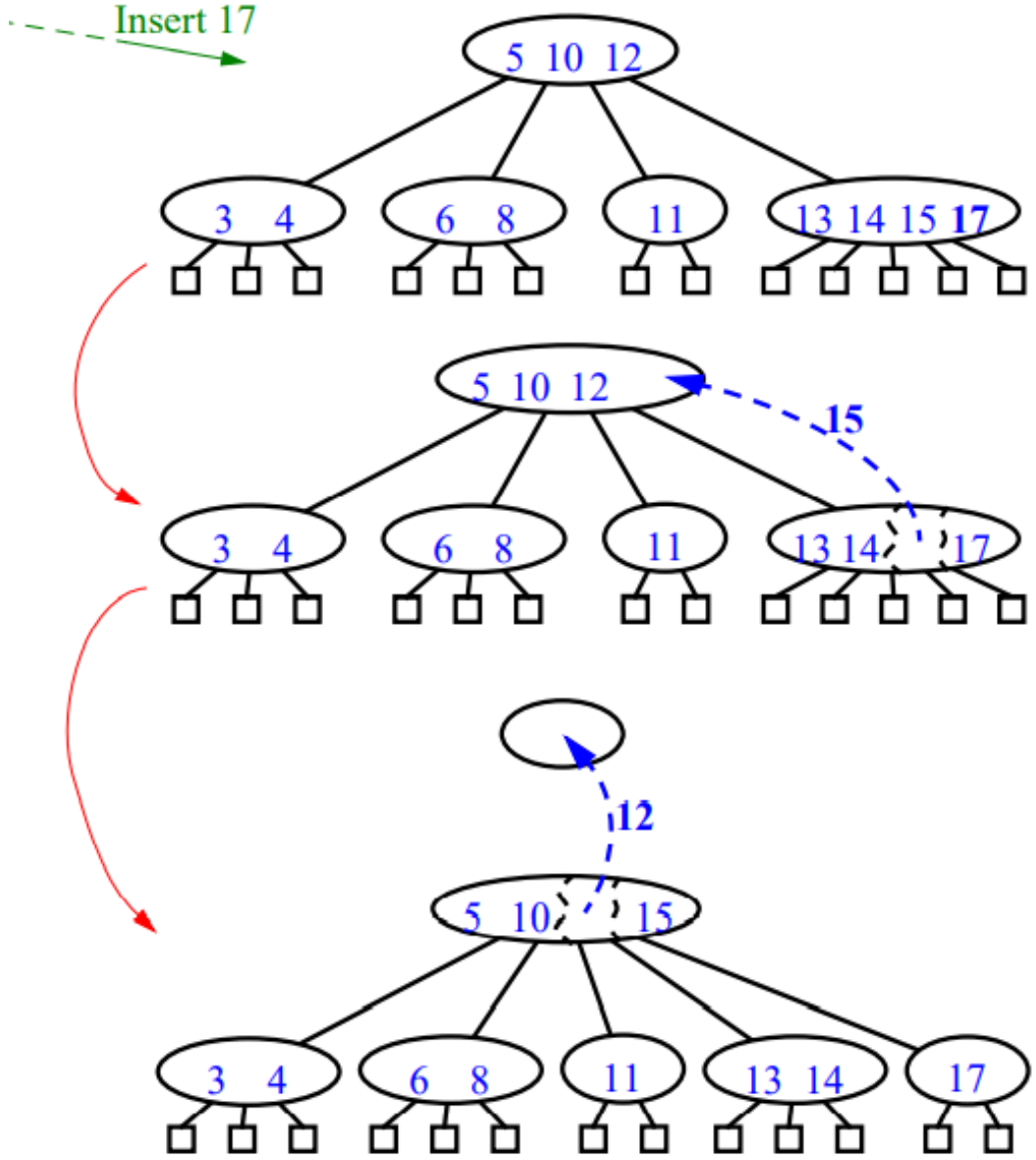
Inserting in a 2-4 tree

- **Let us insert the following items: 4, 6, 12, 15, 3, 5, 17**

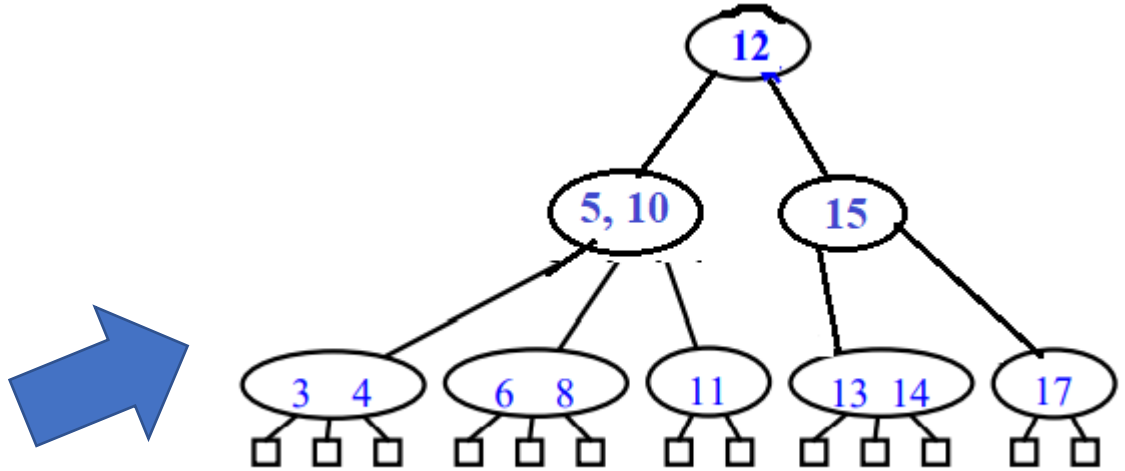


- **Tree always grows from the top, maintaining balance**
- **Now the next challenge is inserting 17.**
  - **The parent will become full!**
- **Let us keep doing the same thing**

Inserting in a 2-4 tree



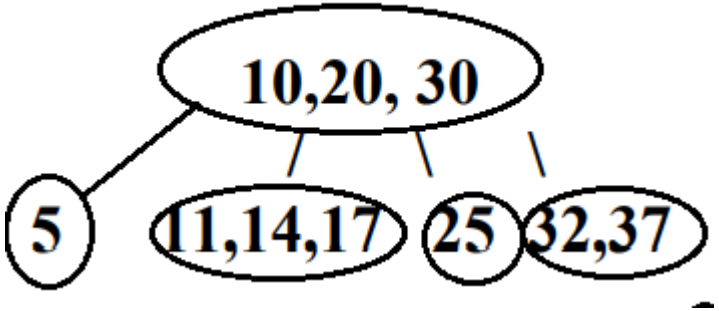
- **Let us insert the following items: 4, 6, 12, 15, 3, 5, 17**



Goes up!. That's why it is bottom up

Can you try the following?

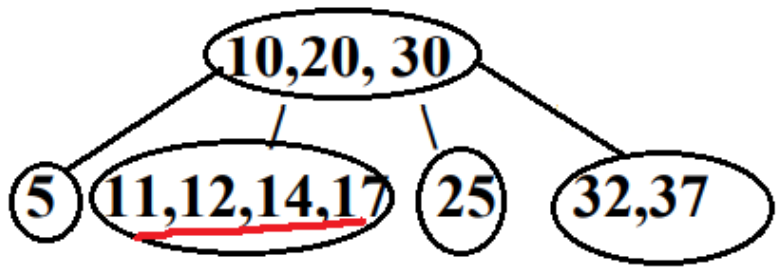
- Try to insert(12) in the following 2-4 tree



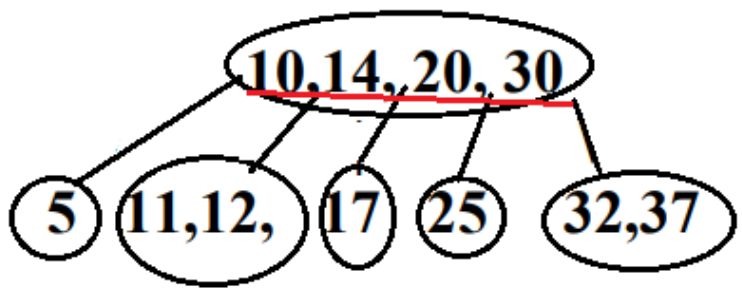
- Answer in the next slide

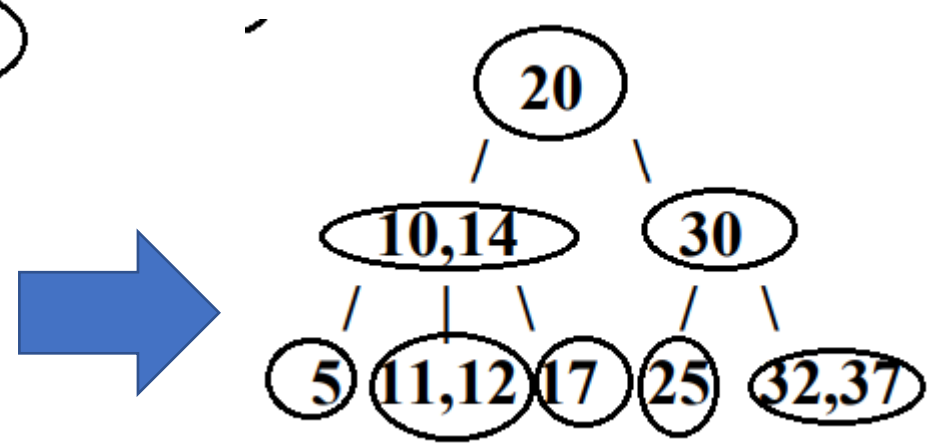
Solution to the insertion in the previous slide

Initially, we have:



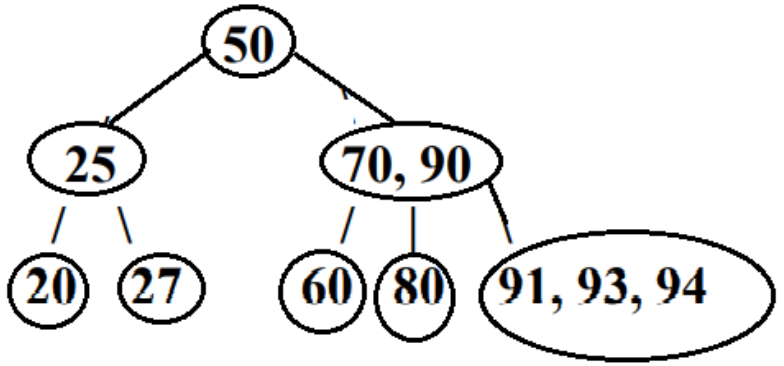
Then, using the rule stated above, we have:



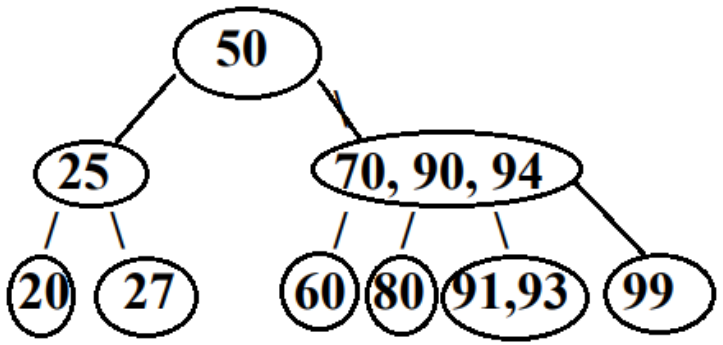


Just another example (for your practice)

Here is one more example:



Insert 99:

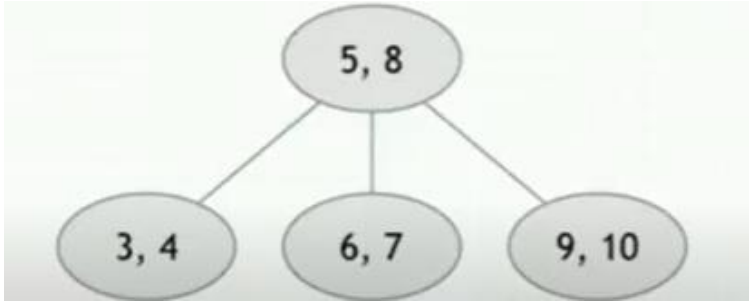


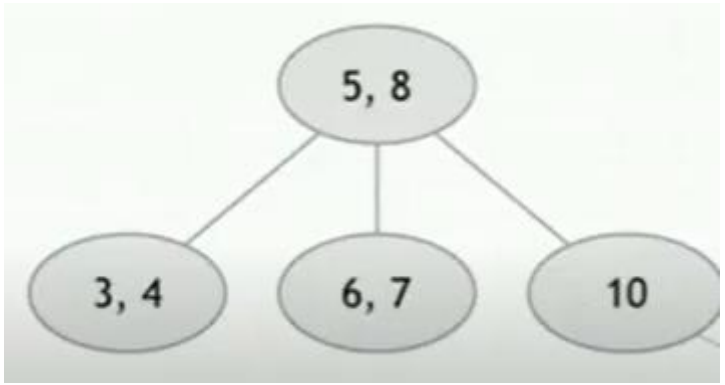
Implementation Ideas

- Implementation Ideas
- 1) Each node is represented by 2 arrays:
  - a) An array of values
  - b) An array of references to Nodes.
- 2) Each node stores
  - a) A linked list of values
  - b) A linked list of references to Nodes.
- Advantage of the first design:
  - 1) Easier to access each item in a node and modify values in a node.
- Advantage of the second design:
  - 1) Never have extra values or references stored in a node

Delete operation

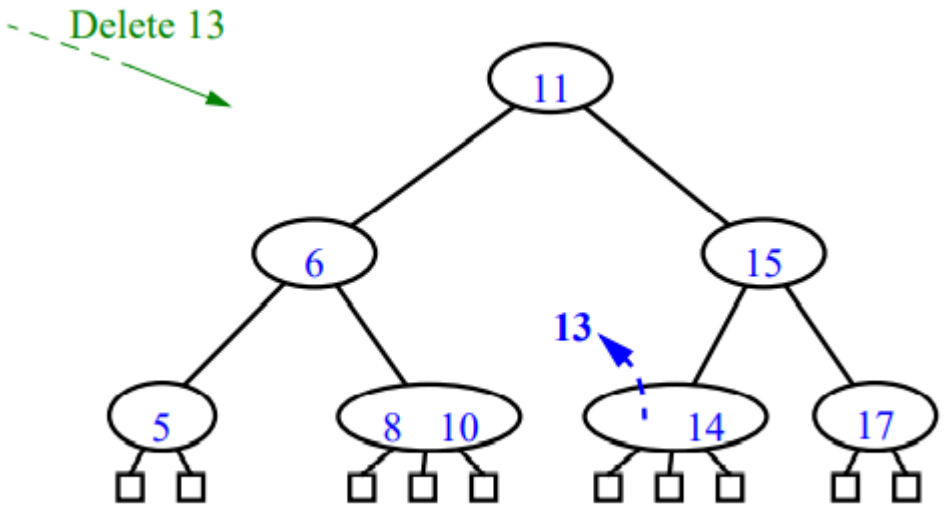
- A bit trickier
- 1. Find the element to be deleted (simply multi-way search)
- **Case 0: Element is 3 node or 4 node leaf (simple case)**
  - 3 node means- a node with 2 elements and 3 children
  - 4 node means- a node with 3 elements and 4 children
  - Solution to case 0 is just simply remove the value (No structural change in the tree)
- Example of case 0:
  - Remove 9 from the following tree:





Delete operation

- Another example of case 0:



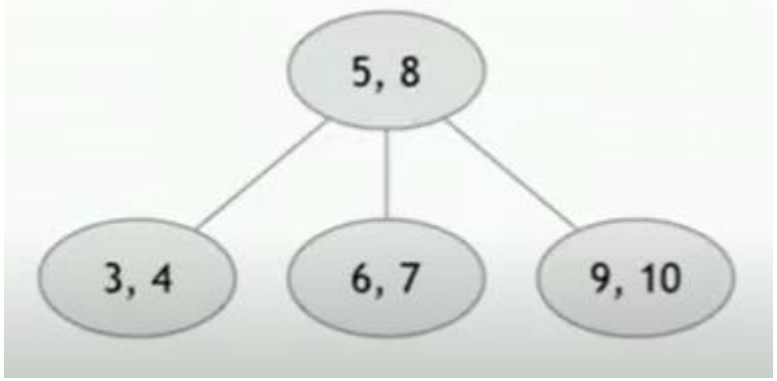
Delete operation

- **Case 1: Element is not a leaf node**
  - **It is like BST deletion where you will bring the largest number from the left subtree or smallest number from the right subtree.**
  - **Let us take from the right side**
  - **Steps:**
    - **Remember the position of the element and continue until we reach a leaf containing the element's successor (in our case smallest in the right sub tree)**
    - **Swap the element with the successor**
    - **If the leaf is not a 2-node (it means have more than 2 values), delete the element**

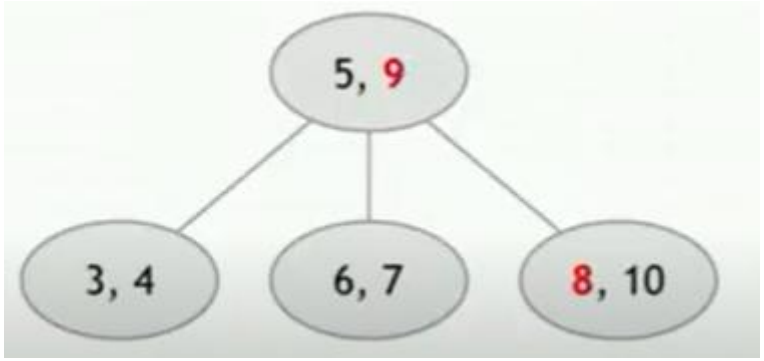
    - **Note that 2-node means a node with only one element**

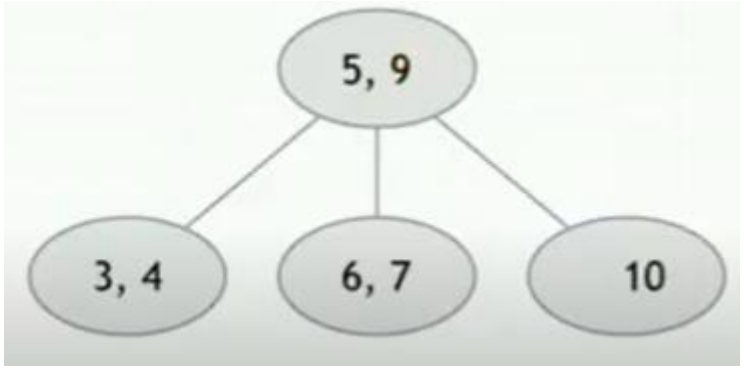
Delete operation

- Case 1: example. Let us **delete 8** from the following tree. As 8 is not a leaf node, it is falling our case 1.



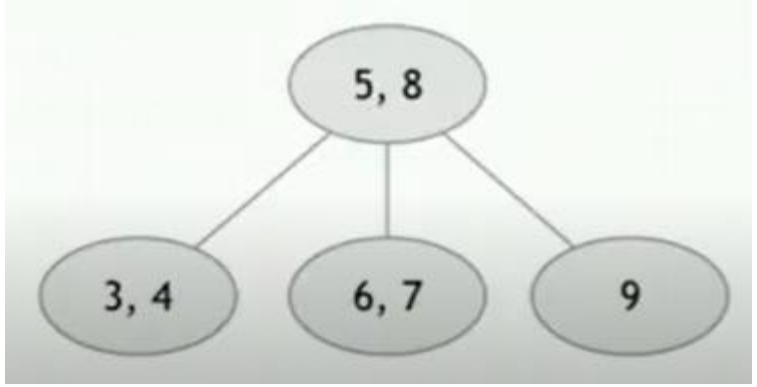
- Who is the successor?
  - 9 (the smallest number in the right subtree of 8)
- So, we swap 8 and 9
- Then, as the leaf node we are dealing with is not a 2-node (has more than 2 items), we simply remove the element 8.





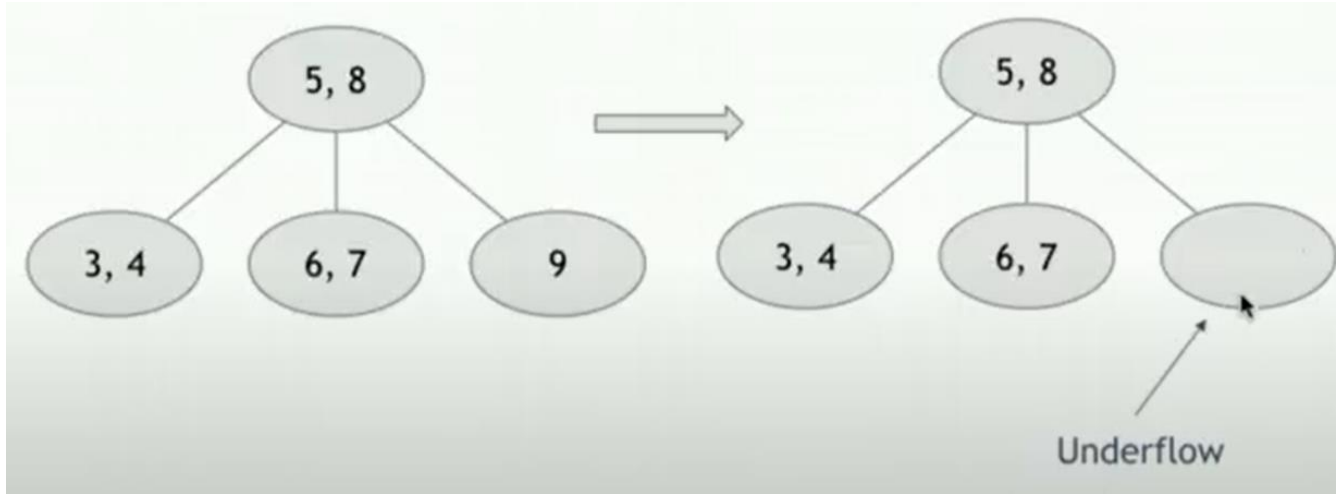
Delete operation

- **Case 2:** Element is in a 2-node leaf (it is the only element in the node)
- So, in this case if you remove the item, there will not be anything remaining there (underflow!)
- This case has **3 sub cases:**
  - **Subcase 1 (2.1):** The parent node and sibling nodes are not 2-node (they have more than one items in them)
  - Solution:
  - Pull an item from the parent and replace it with the element that we are deleting
  - Now, balance it by moving a item from the sibling
  - Example: **Remove 9** from the following tree (see the sibling and the parent are not 2-node)



Deletion case 2.1 example (remove 9)

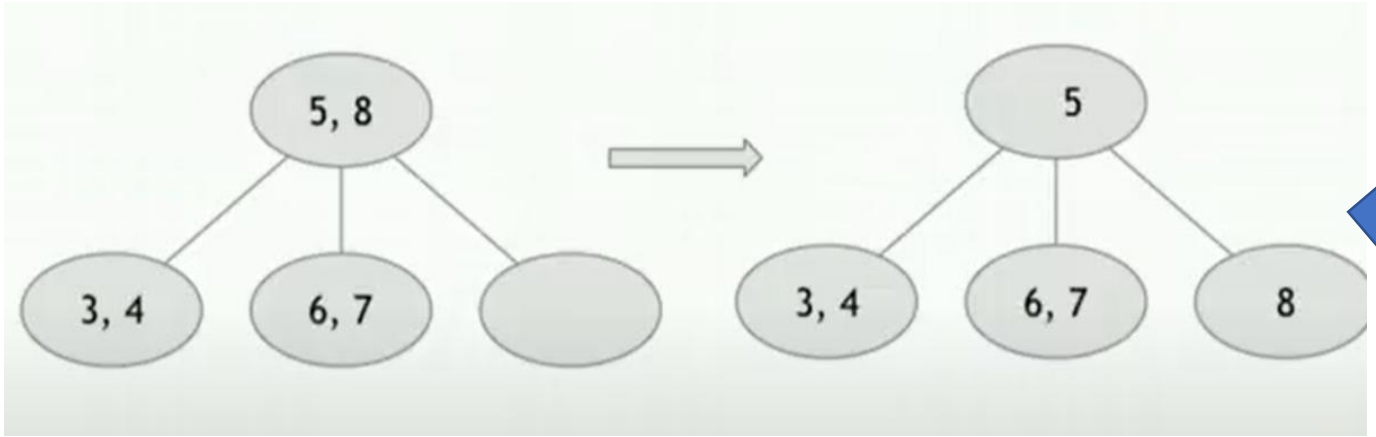
Removing 9 creates underflow in the node

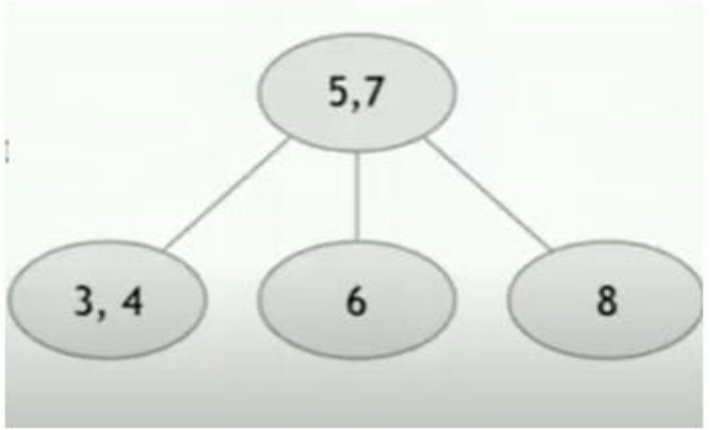


Now the parent node does not make sense! It can't have 3 children with only one value in it

Bring up one of the element from sibling

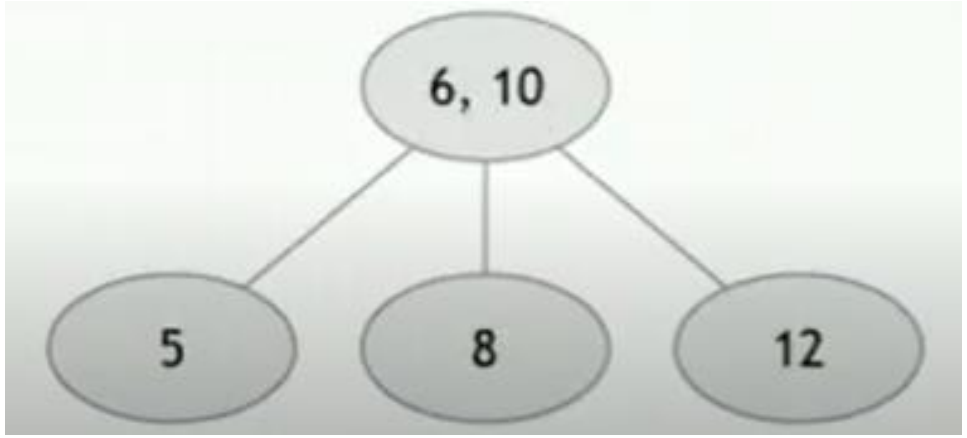
Pull a parent element and put that in the (empty node) position of 9





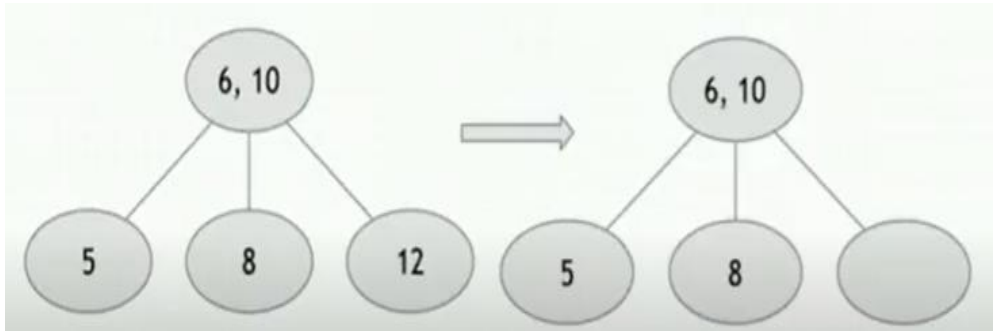
Delete operation

- **Case 2:** Element is in a 2-node leaf (it is the only element in the node)
  - **Subcase 2 (2.2): The** sibling nodes are 2-node (they have only one items in them)
  - Solution:
  - Pull an item from the parent and replace it with the element that we are deleting
  - Fuse the siblings into one node (merge)
  - Example: **Remove 12** from the following tree (see the sibling has only one item)

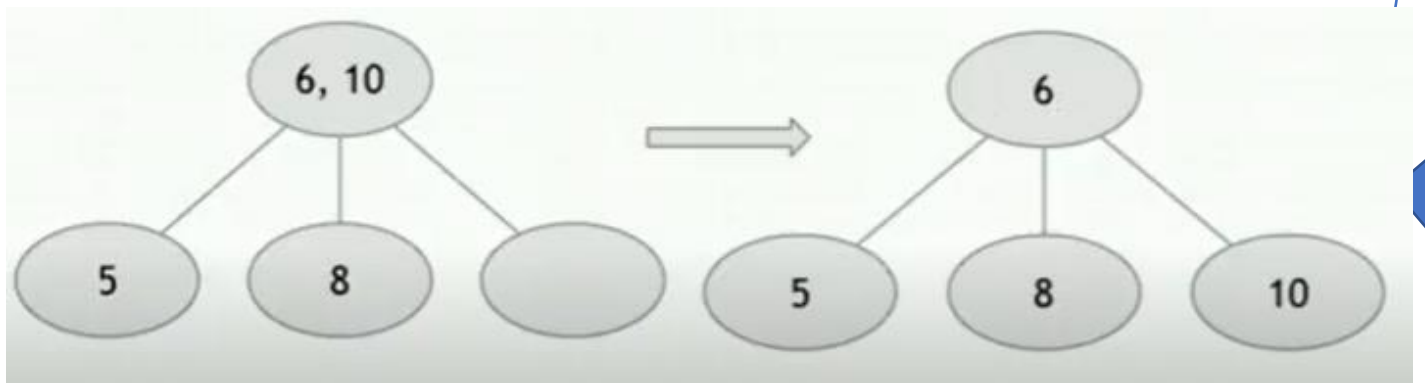


Deletion case 2.2 example (remove 12)

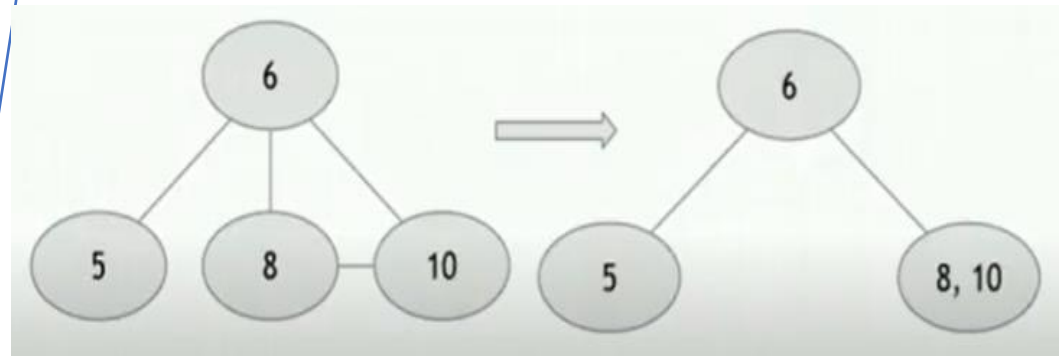
Removing 12 creates underflow in the node



Pull a parent element and put that in the (empty node) position of 9

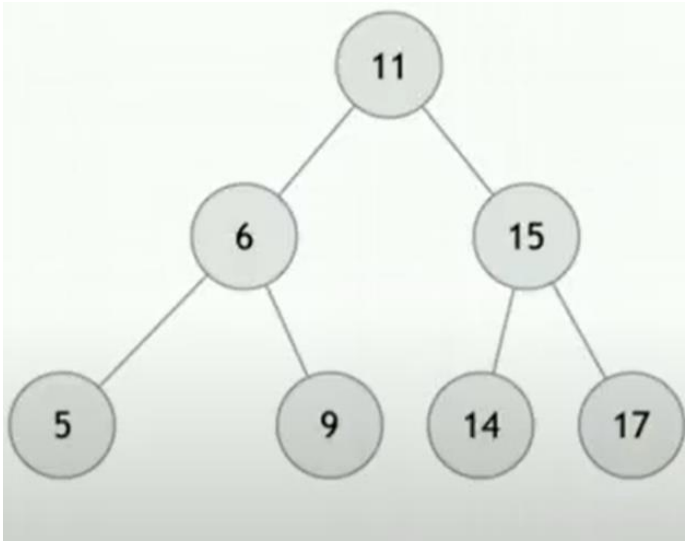


Now fuse the two siblings



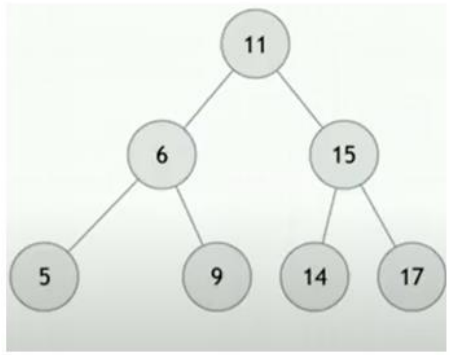
Delete operation

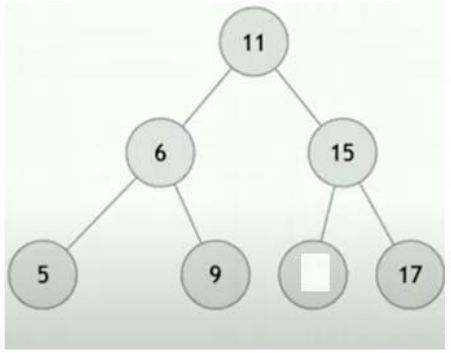
- **Case 2:** Element is in a 2-node leaf (it is the only element in the node)
  - **Subcase 3 (2.3): The parent node is a** 2-node (it has only one items in it)
  - Solution:
  - Pull an item from the parent and replace it with the element that we are deleting
  - Fuse the siblings into one node (merge)
  - Do it multiple times
  - Example: **Remove 14** from the following tree (see the parent has only one item)



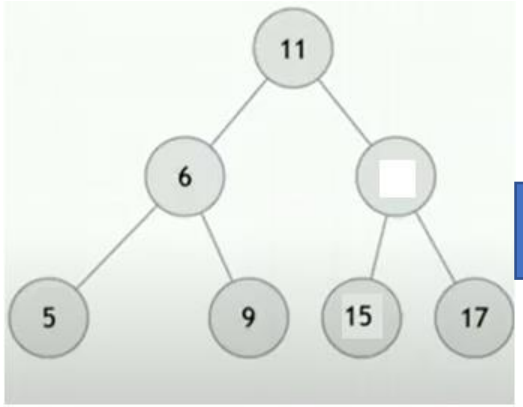
Deletion case 2.3 example (remove 14)

Removing 14 creates underflow in the node

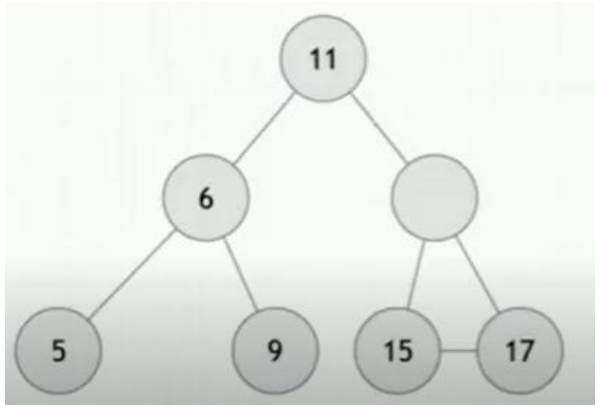




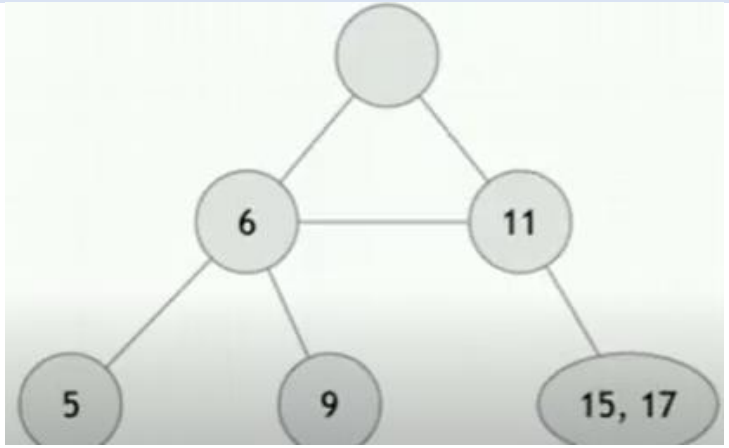
Pull the parent element and put that in the (empty node) position of 9



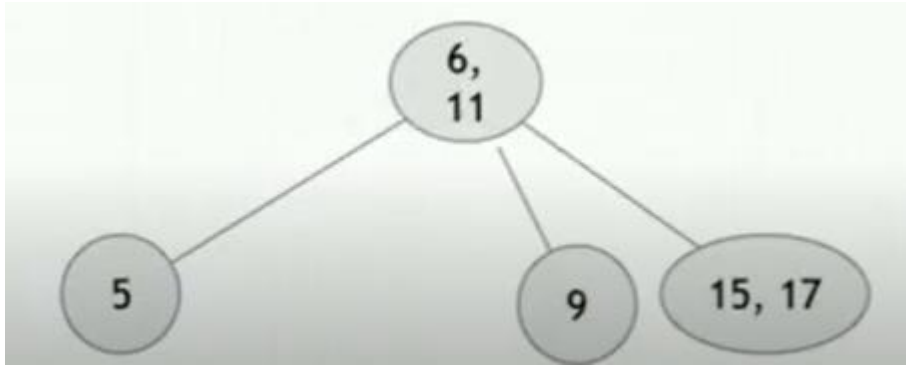
Now fuse the two siblings



After fusing, still the parent is underflow, and we do the same there as well (pull empty node's parent to there and fuse with sibling

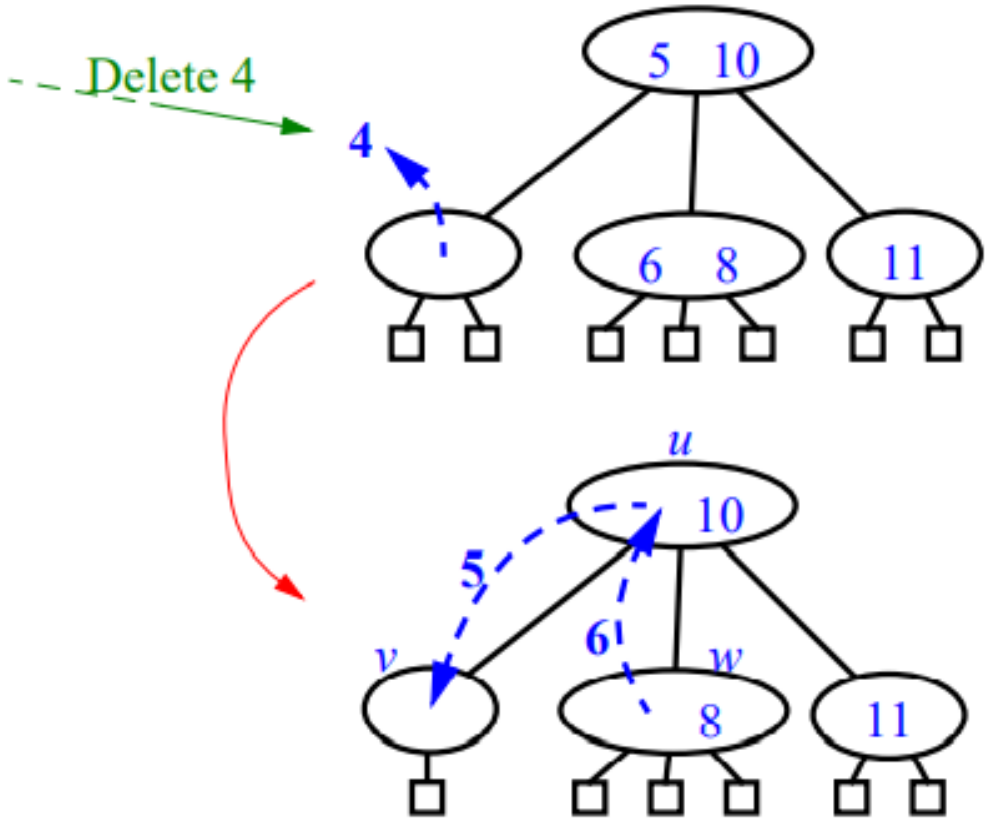


Final result after removing 14



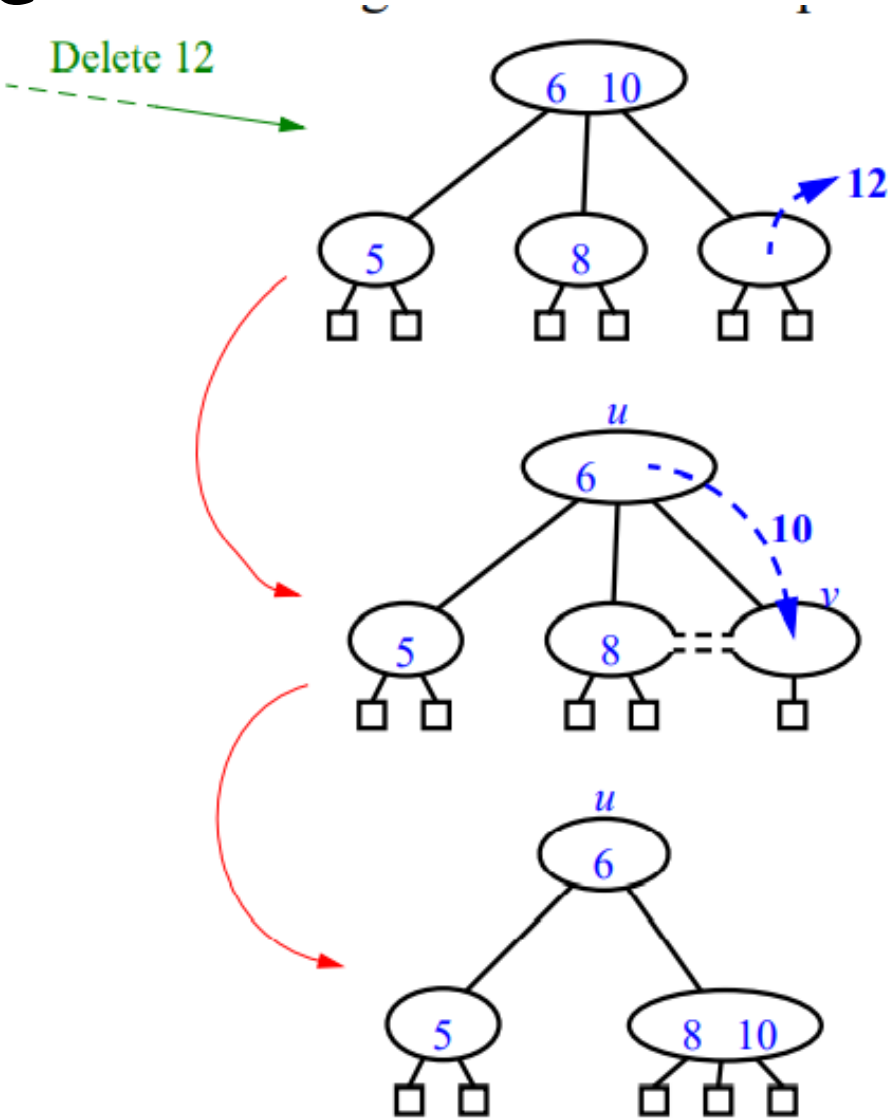
More example for practice

- If you delete 4, (case 2.1)
- Pull an item from the parent,
- replace it with an item from a sibling - called transfer



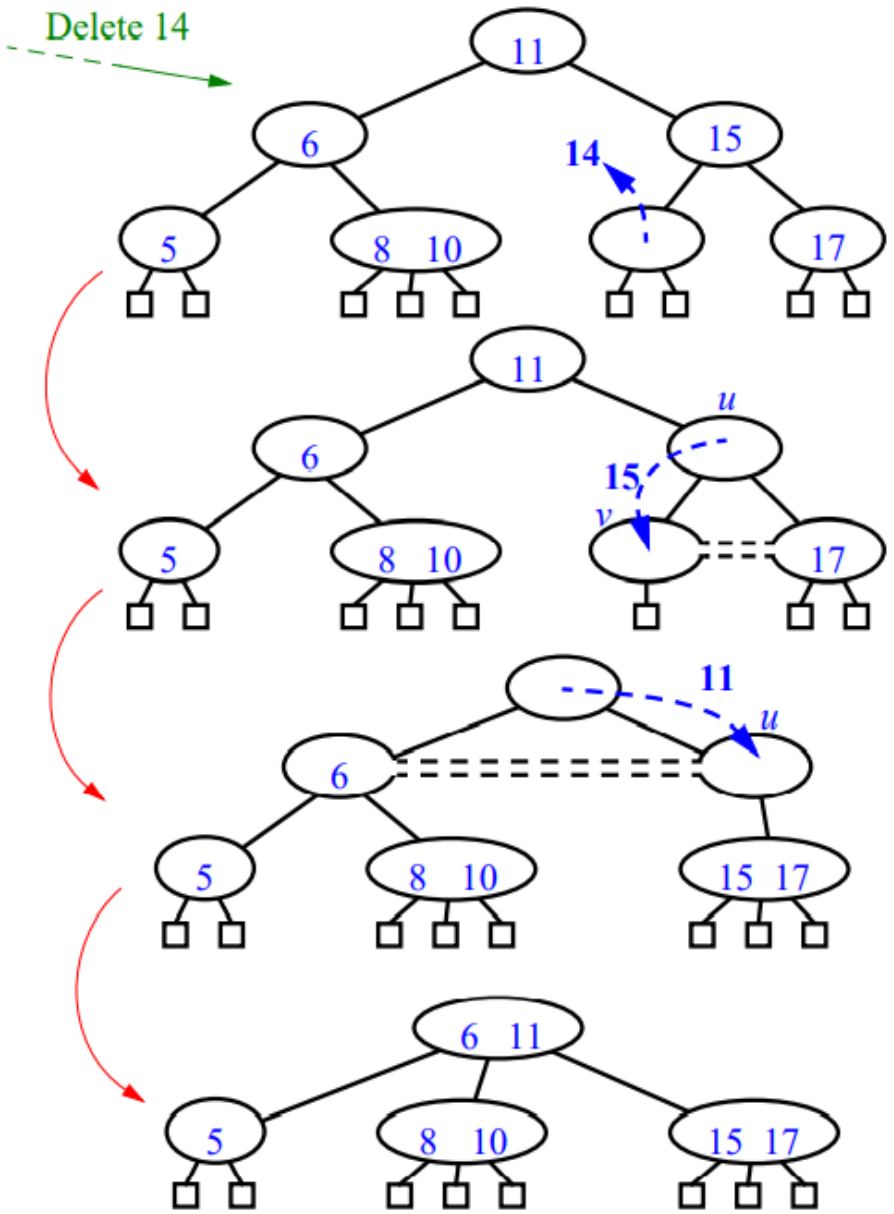
More example for practice

- If you delete 12, **(case 2.2)**
- Pull an item from the parent,
- Fuse the siblings



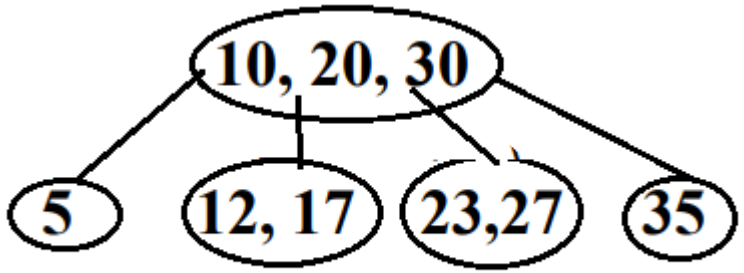
More example for practice

- If you delete 14, (case 2.3)

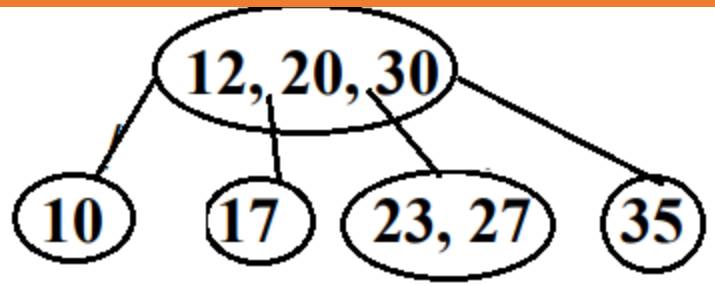


More deletion exercises

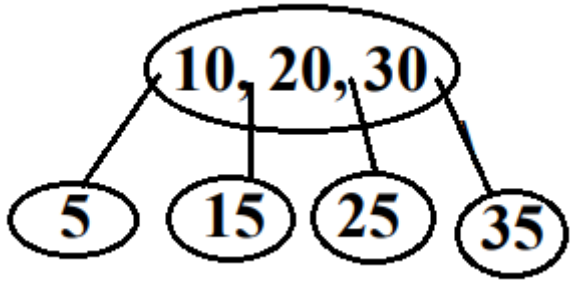
- Delete 5 from the following B-tree



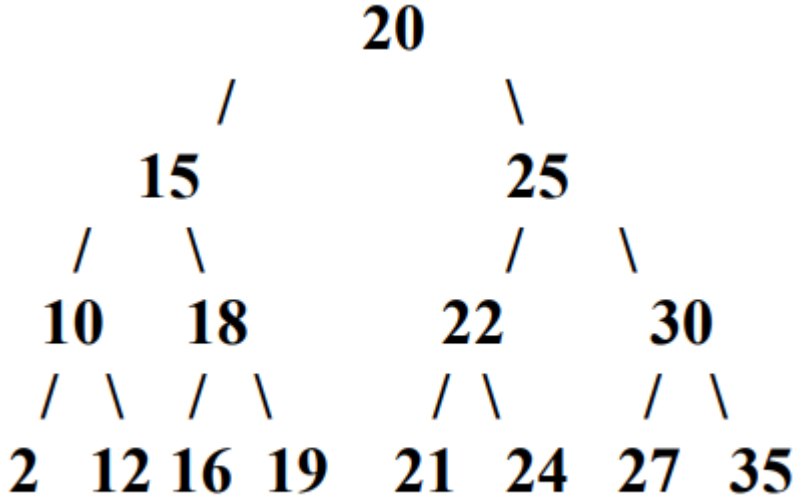
- Delete 35 from the following B-tree



- Delete 5 from the following B-tree



- Delete 24 from the following B-tree

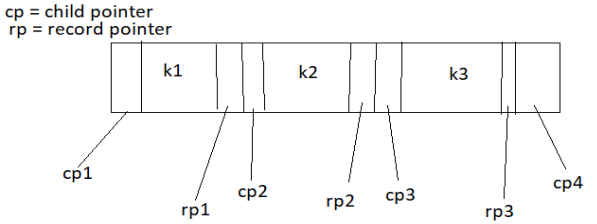


Time-complexity

- • The height of a (2,4) tree is O(log n)
- Split, transfer, and fusion each take O(1)
- • Search, insertion and deletion each take O(log n)

Concept of B+ tree

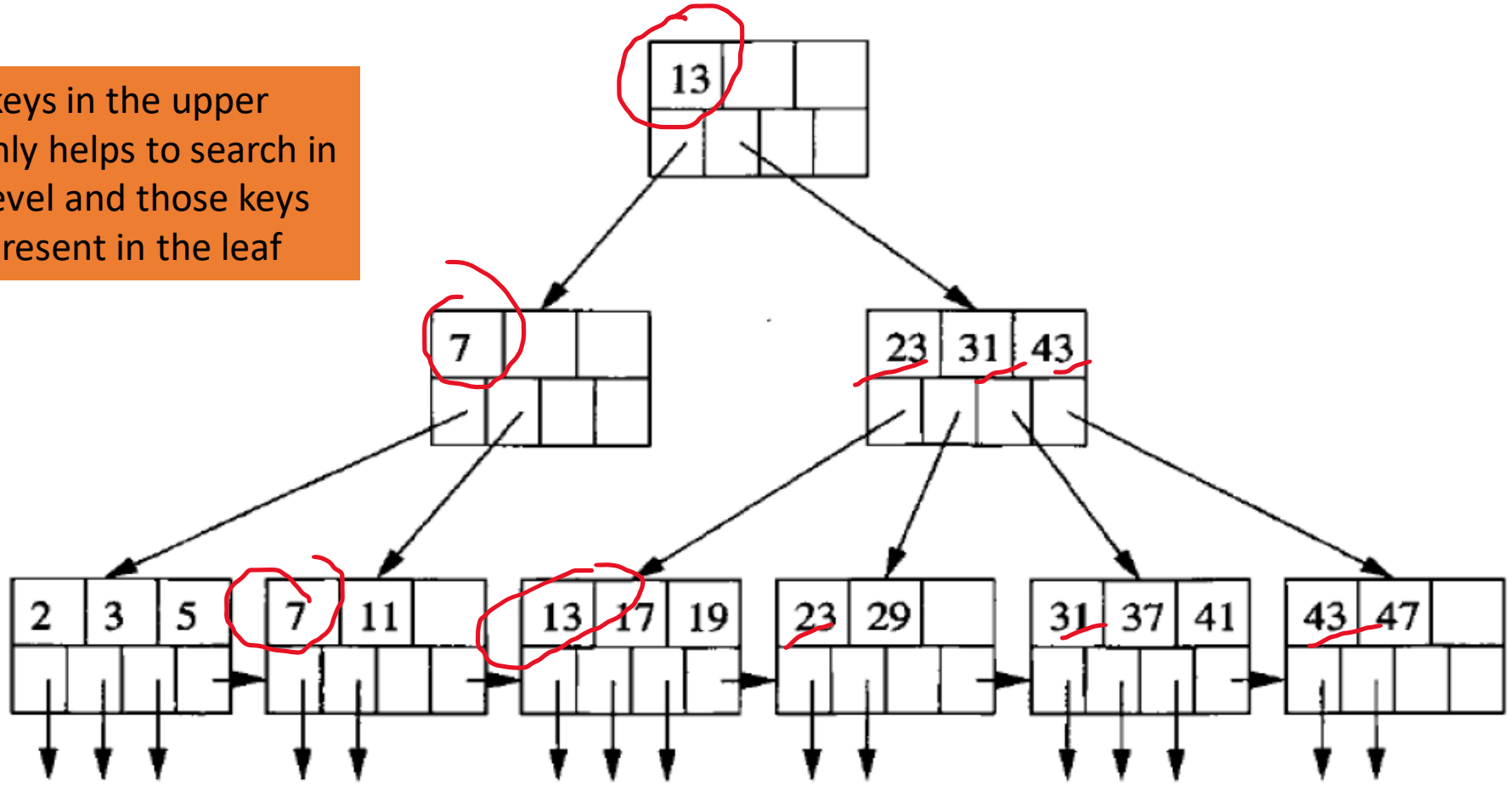
- B+ tree is an extension of B tree
- In case of indexing, a B tree node contains record pointer, in addition to child pointer. It is applicable for any node, regardless whether the node is a leaf node or not



- In case of B+ tree, all the keys are also available in the leaf level and the record pointer is only available in the leaf level . So, in this case, there maybe some duplicate entries in a sense that an item in the upper level is also present in the leaf level.
  - Additionally, the leaf level nodes also connected like a linked list
  - See an example in the next slide

EXAMPLE: B+-TREE

See, the keys in the upper level mainly helps to search in the leaf level and those keys are also present in the leaf



Differences between B tree and B+ tree

S.NO	**B tree**	**B+ tree**
1.	All internal and leaf nodes have data pointers	Only leaf nodes have data pointers
2.	No duplicate of keys is maintained in the tree.	Duplicate of keys are maintained and all nodes are present at leaf.
3.	Insertion takes more time and it is not predictable sometimes.	Insertion is easier and the results are always the same.
4.	Deletion of internal node is very complex and tree has to undergo lot of transformations.	Deletion of any node is easy because all node are found at leaf.
5.	Leaf nodes are not stored as structural linked list.	Leaf nodes are stored as structural linked list.
6.	No redundant search keys are present..	Redundant search keys may be present..

Thank you!