COMP1002 DATA STRUCTURES AND ALGORITHMS

LECTURE 10: ADVANCED TREES

Curtin University

Discipline of Computing

Last updated: May 19, 2020

This Week

- Review of binary tree complexity analysis
- Self-balancing trees
  - Red-Black Trees
  - 2-3-4 Trees
  - B ('Block') Trees

Types of Binary Trees

- We have talked in previous lectures about the types of binary trees that exist in terms of their structure
- Binary tree types:
  - Complete binary tree (balanced)
  - Almost-complete binary tree (almost balanced)
  - Degenerate binary tree (not desirable!)

Maintaining Balanced Trees

- It is desirable to have balanced trees
  - This is difficult to achieve and maintain
- We usually follow a set of rules to get us reasonably close to a completely balanced tree
  - Though these rules will not necessarily give us a perfectly balanced tree
- Red-Black trees, 2-3-4 trees and B trees are examples of such self-balancing trees

Red-Black Trees – Properties

- Colour Rule:
  - Each node is either red or black
- Root Rule:
  - The root is always black
- Parent Rule:
  - A red node's children are *always* black
  - A black node's children can be either red or black
- Black Height Rule:
  - Every path from root to leaf (or to a *null child*) must contain the same number of black nodes

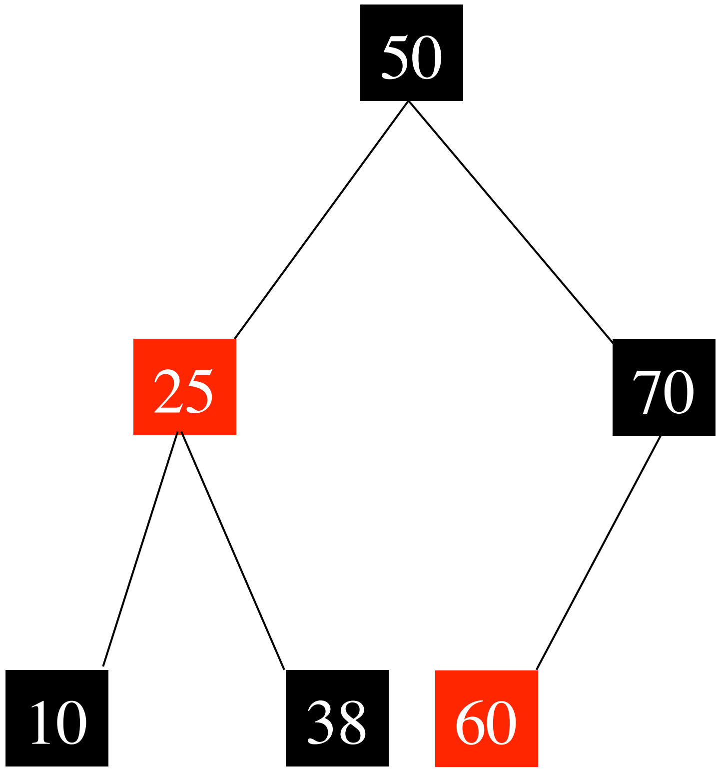
Red-Black Trees – Properties

- Black height rule + Parent rule enforces balance
  - A path has at most half its nodes being red
  - But every path must have the same number of black nodes
  - So the longest possible path alternates red-black, and the shortest possible path is all black
    - *e.g.*, black height = 7
    - Then longest possible path = **7** + **7** = **14**
    - And shortest possible path = **7** + **0** = **7**
  - Thus the worst case is only twice as bad as the best case, hence the worst case is O(2 log N) = O(log N)

Red-Black Trees – Properties

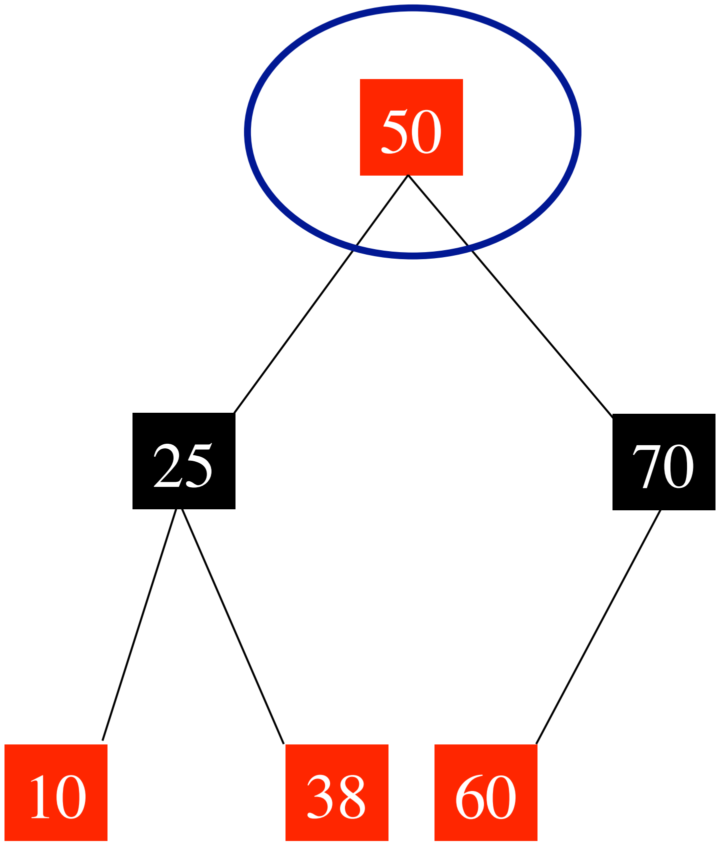
- New nodes that are inserted are always red
  - Since new nodes don’t have children we minimise any potential rule violations ie: we won’t violate rules 1, 2 and 4 but may violate rule 3 (Parent Rule)

Red-Black Trees – Examples



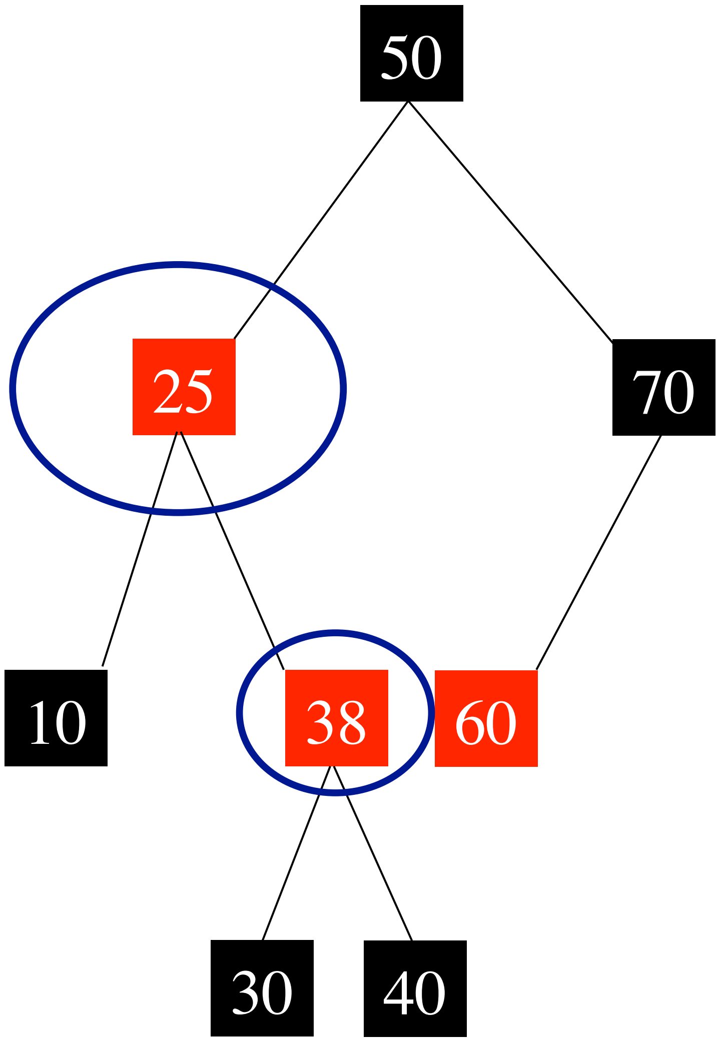
- ☑Every node is either red or black
- ☑Root = black
- ☑Red nodes have black children
- ☑Every path from the root to a leaf node or to a null child contains the same number of black nodes

Red-Black Trees – Violations



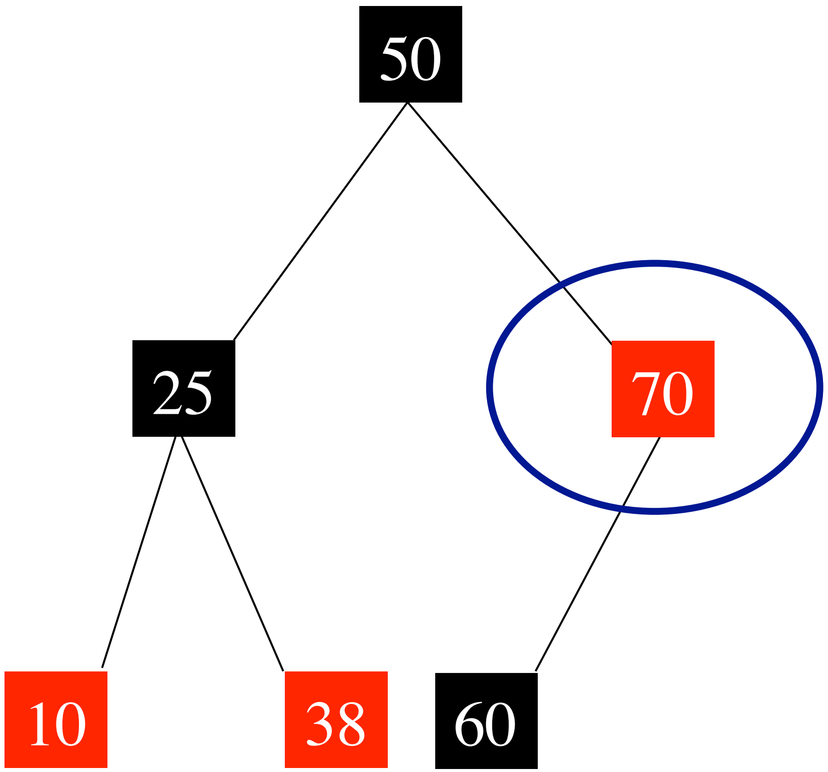
- Black root node violation
  – 50 is a red node

Red-Black Trees – Violations



- Violated red parent rule – red node 25 has red child node 38

Red-Black Trees – Violations



- ☑ Path from 50-25-10-[38] has two black nodes
- ☑ Path from 50-70-60 has two black nodes
- ☒ Path from 50-70 has only one black node – violation

Red-Black Trees – Violation Fixes

- So what do we do if any of the rules are violated after inserting a new item, which results in an incorrect (ie: unbalanced) Red-Black tree?
  - Switching the colours of the parent and children
  - Switch the colour of a single node
  - Rotate and possibly graft sub-trees into new positions

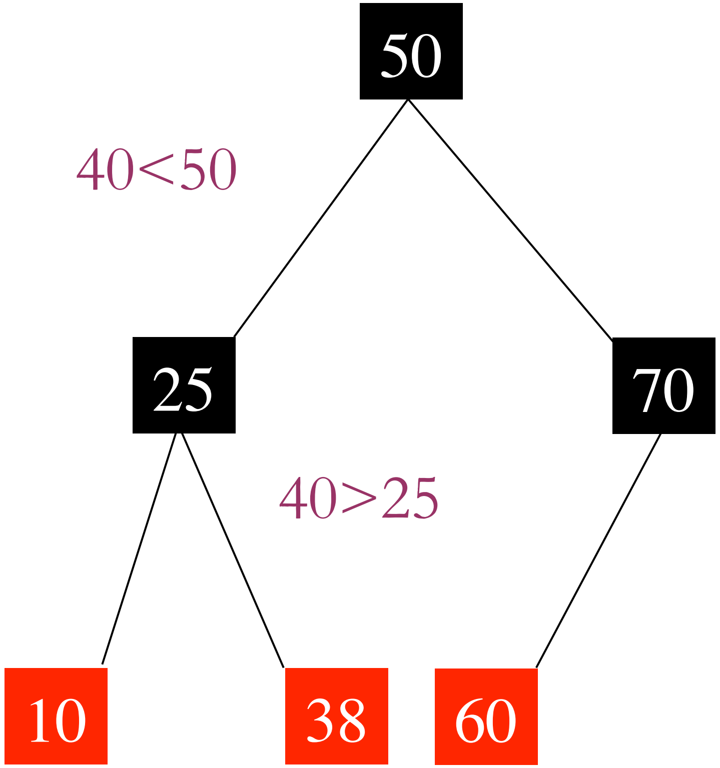
Switching Colours

- We can use the switching of colours to turn red nodes into black ones
- Useful since we always insert new nodes as red nodes
- Note that a switch of colours will not violate the black height property

Inserting a Node

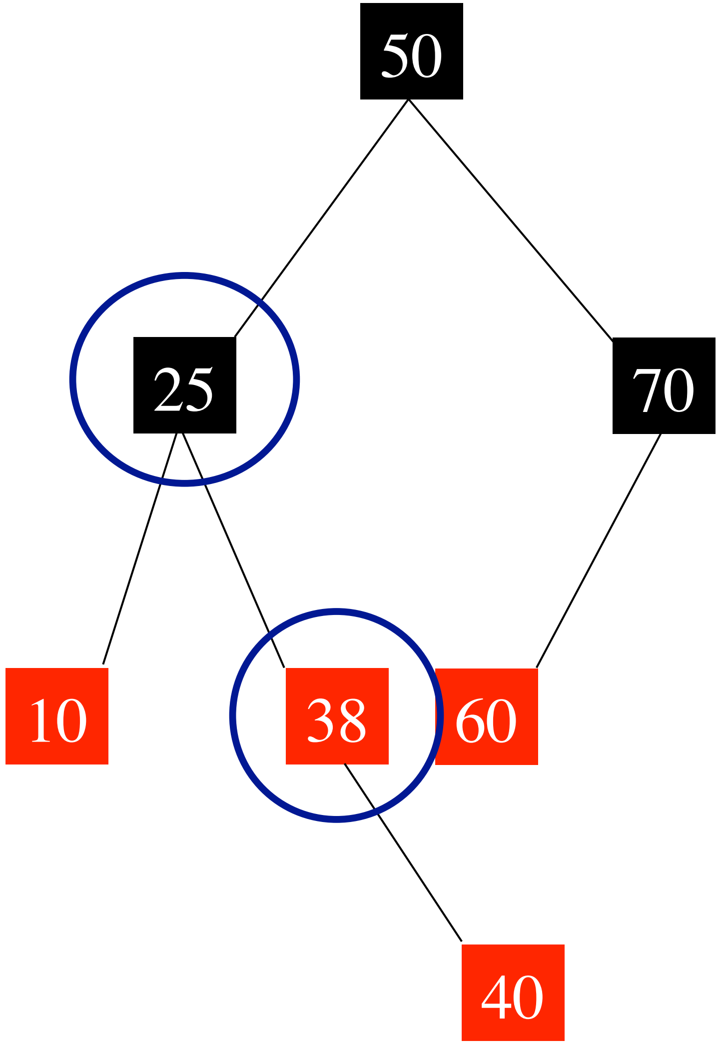
- Works the same as for a binary search tree
  - New node always ends up towards the bottom of the tree as a leaf node
  - Remember left child < parent < right child for BST
  - So traverse from root comparing left and right child keys with the key of the node to be inserted until you reach leaf node or suitable null child

Inserting a Node



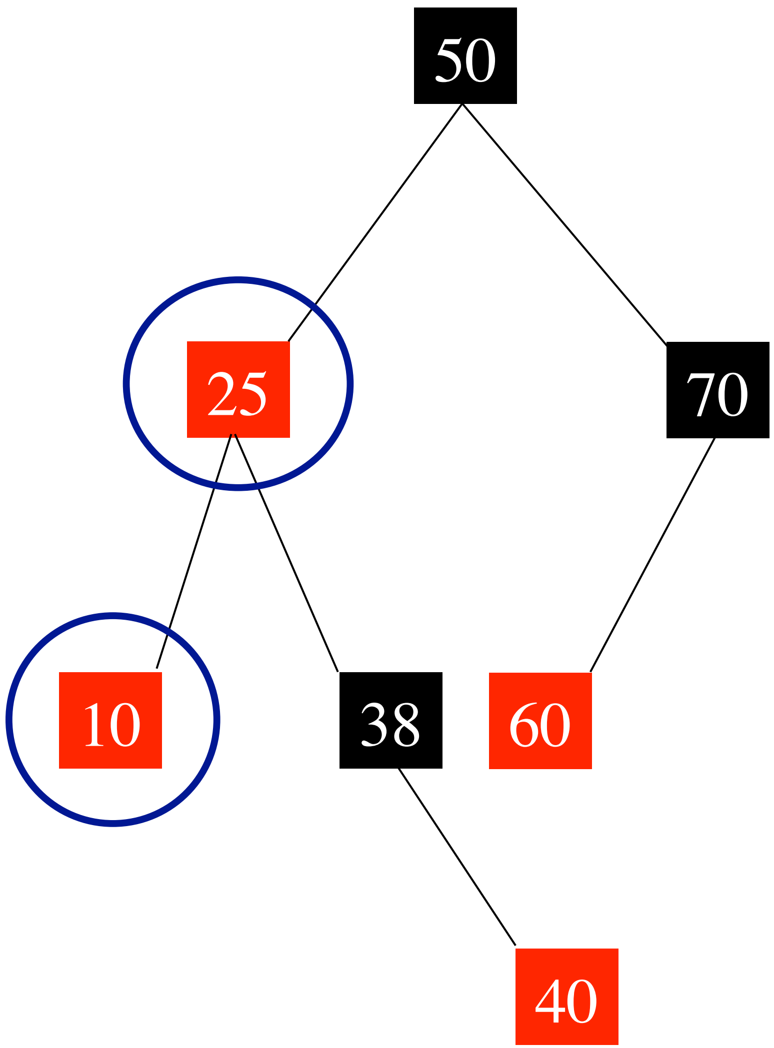
- Want to insert 40
- Will be inserted as a red node

Inserting a Node



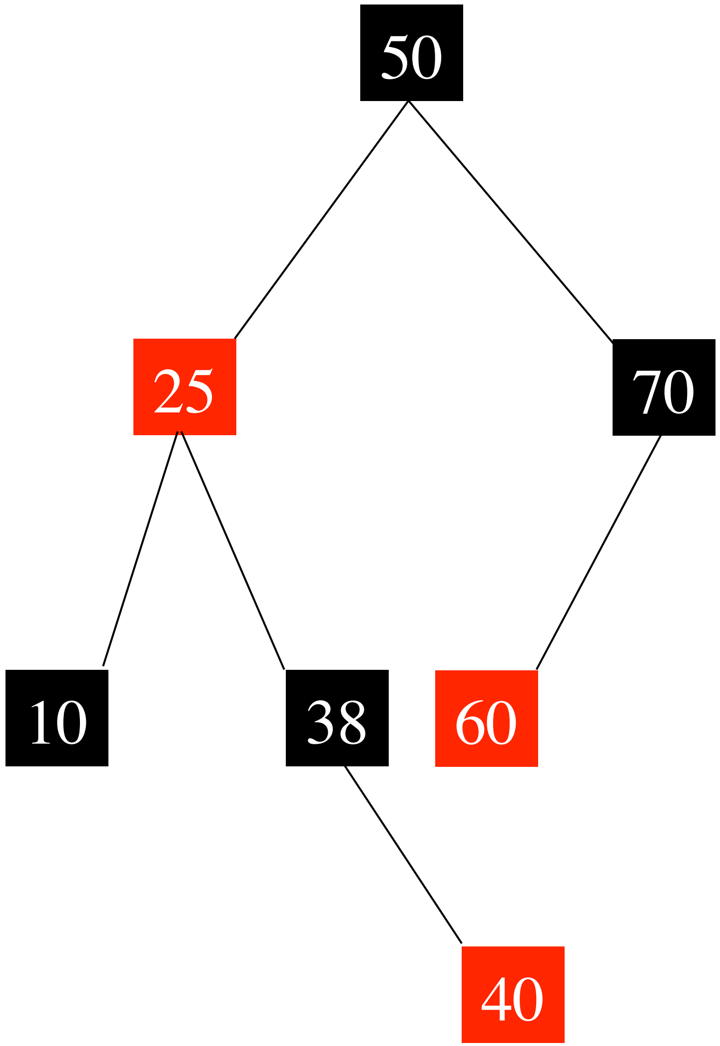
- Oops, we are violating the red parent rule
- Can flip colour of node 38 with node 25

Inserting a Node



- Unfortunately now node 10 has a red parent!
- We can flip the colour of 10 from red to black

Inserting a Node



- Now all paths have the same number of black nodes and the parent rule has not been violated

Rotation of Nodes

- Can also do rotation of nodes
- The rotation takes place with respect to the root of a particular sub-tree
- A bit complicated – if you are interested see extra lecture notes DSA-10a_redBlack.ppt

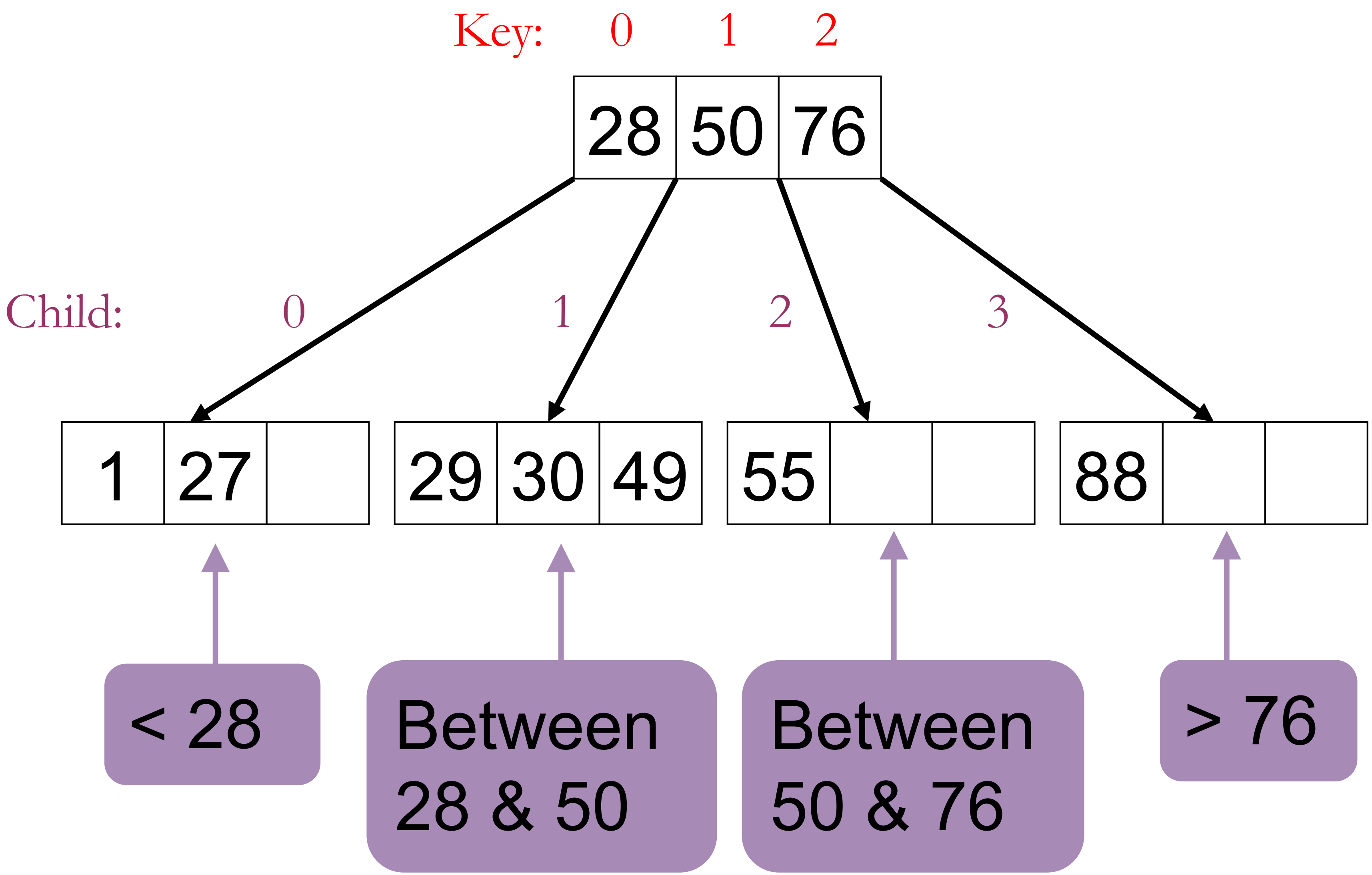
Multi-way Trees

- Multi-way trees can have more than one data item per node
- Examples are 2-3-4 Trees and B-Trees
  - A 2-3-4 tree can have 1, 2, or 3 keys in node
  - A 2-3-4 tree can have 2, 3 or 4 children
- B-trees can have many data items and children
  - one more child than items

2-3-4 Tree Properties

- All leaves are on the same (bottom) level
- Convention:
  - Keys in order from left-to-right within a node
  - Items in left-most node are less than key 0
  - Items in right-most node are greater key 2
  - Items in the middle-left tree are between keys 0 and 1
  - Items in the middle-right tree are between keys 1 and 2

2-3-4 Tree Properties

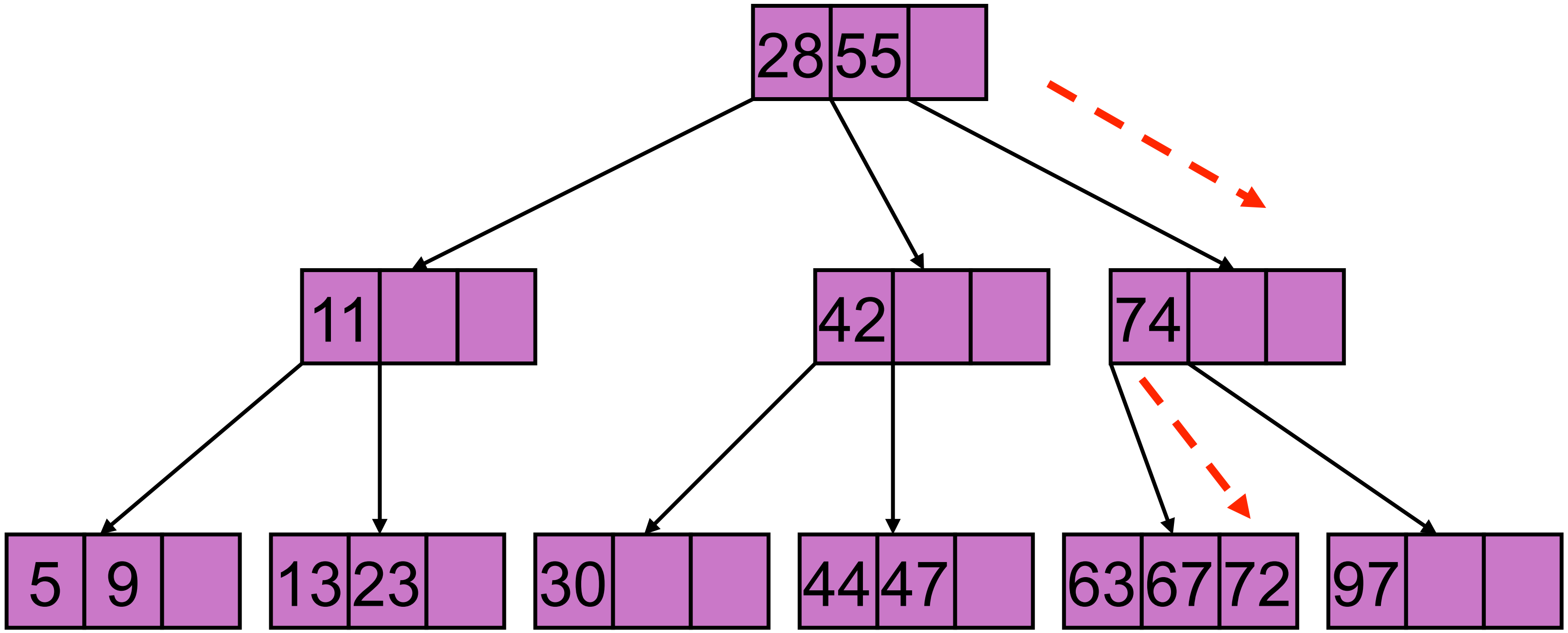


Searching for Items in 2-3-4 Trees

- Exactly the same as for binary search trees
  - Though now will have to consider more than one key per node

Searching for Items in 2-3-4 Trees

- Search for 72
  - Check 72 <= 28 and 72 <= 55 – no, so follow key 2
  - Check 72 <= 74 – no, so follow key 0
  - Check 72 <= 63, 67 and 72 – found 72



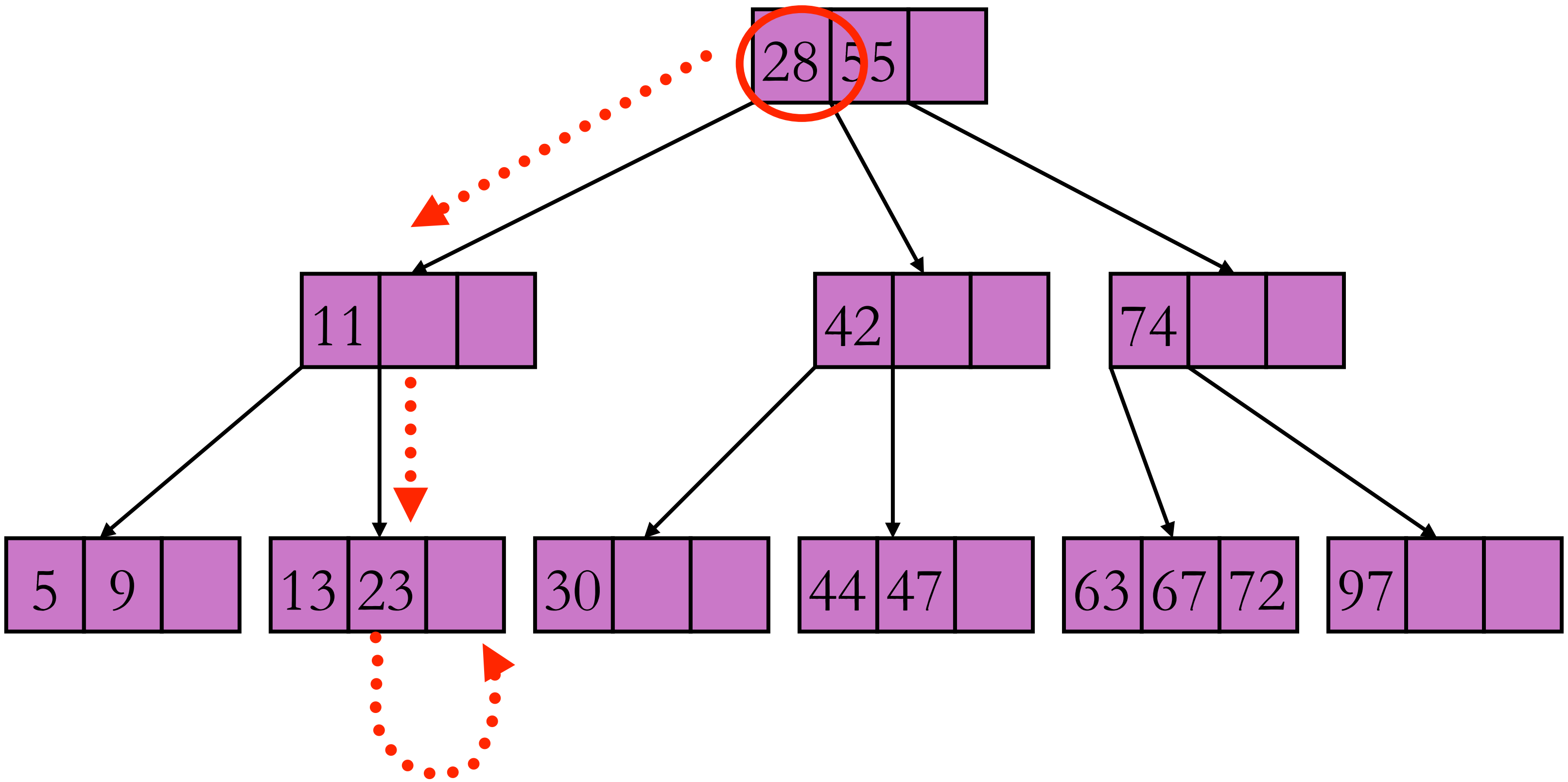
Insertion in 2-3-4 Trees

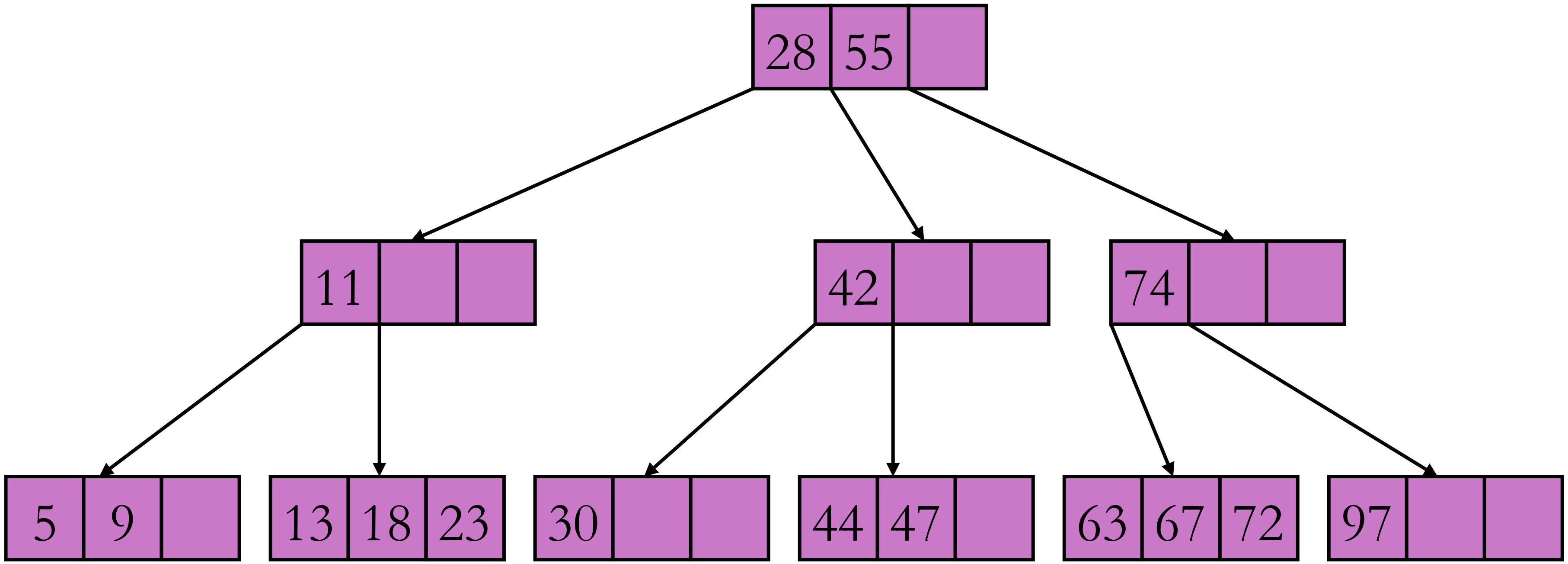
- New keys are always added to the bottom of the tree in a leaf
- Top-down insertion:
  - Search for leaf in which to insert key
  - If encounter a full node on the way, split it
  - May have to move keys in the leaf

Insert 18

18 < 28

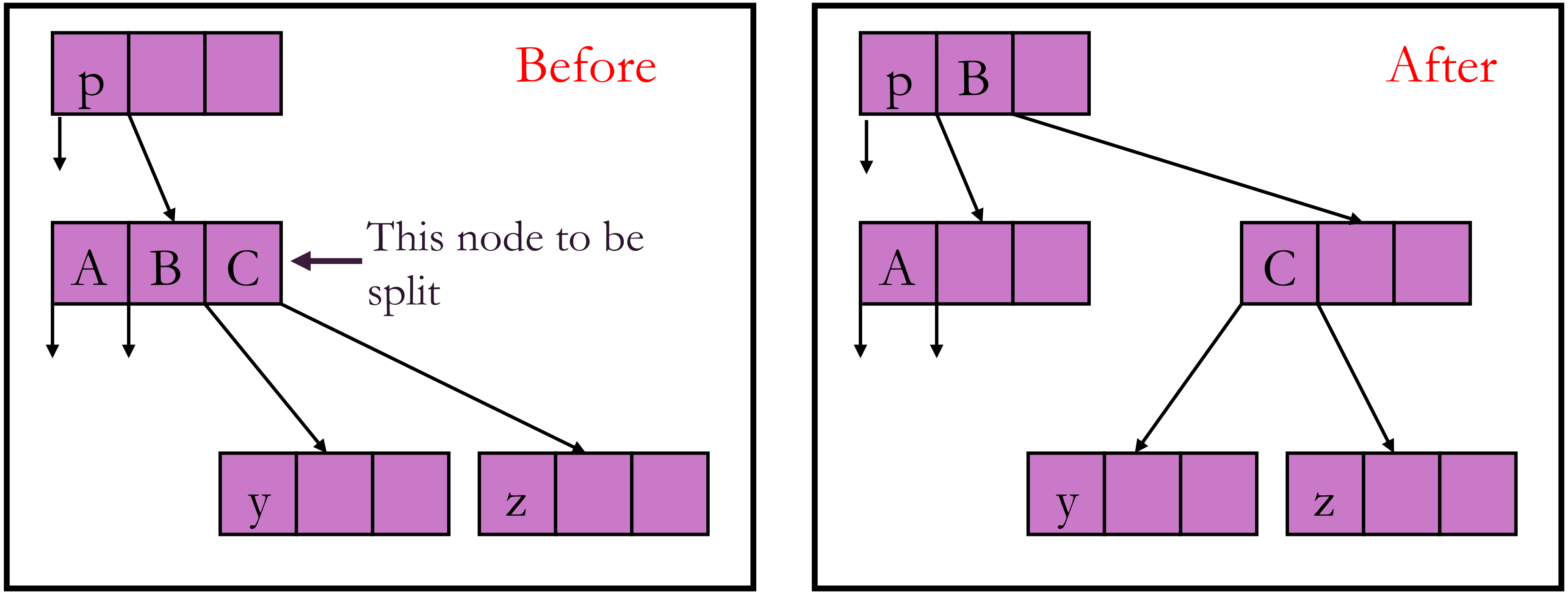
18 < 11





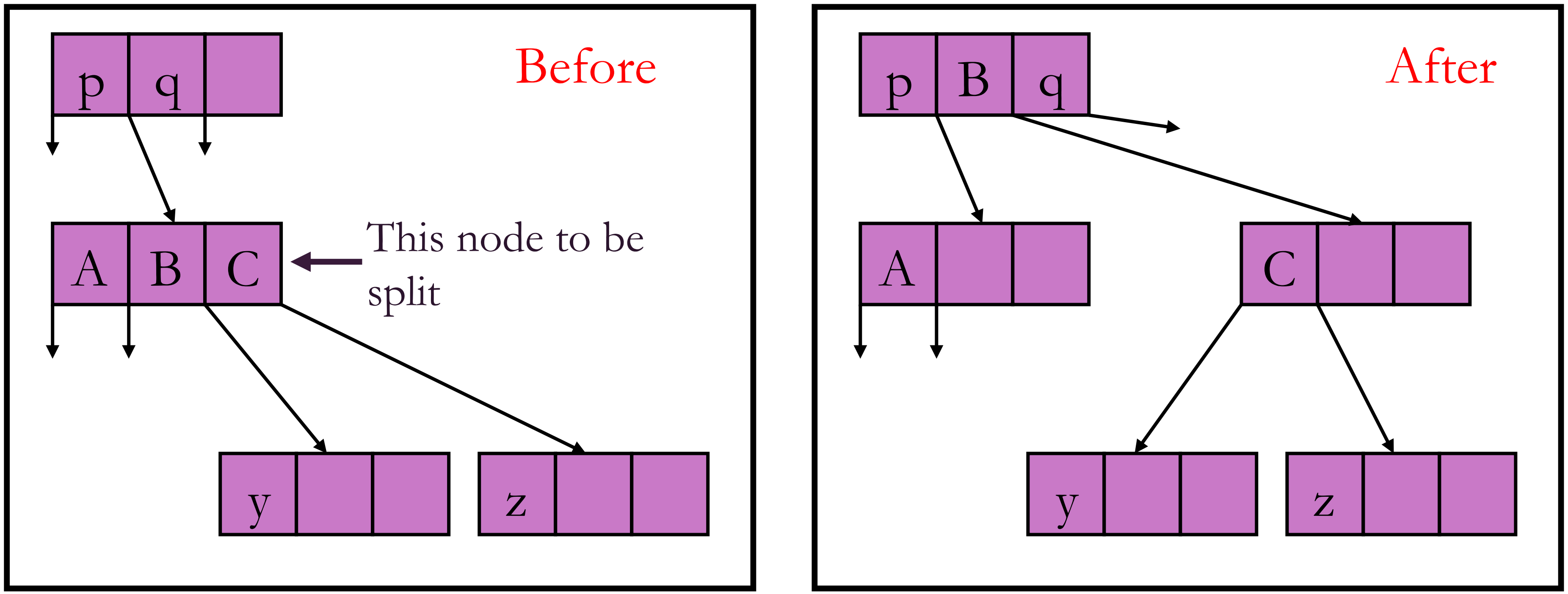
Tree after inserting 18

Splitting Node I (not root)

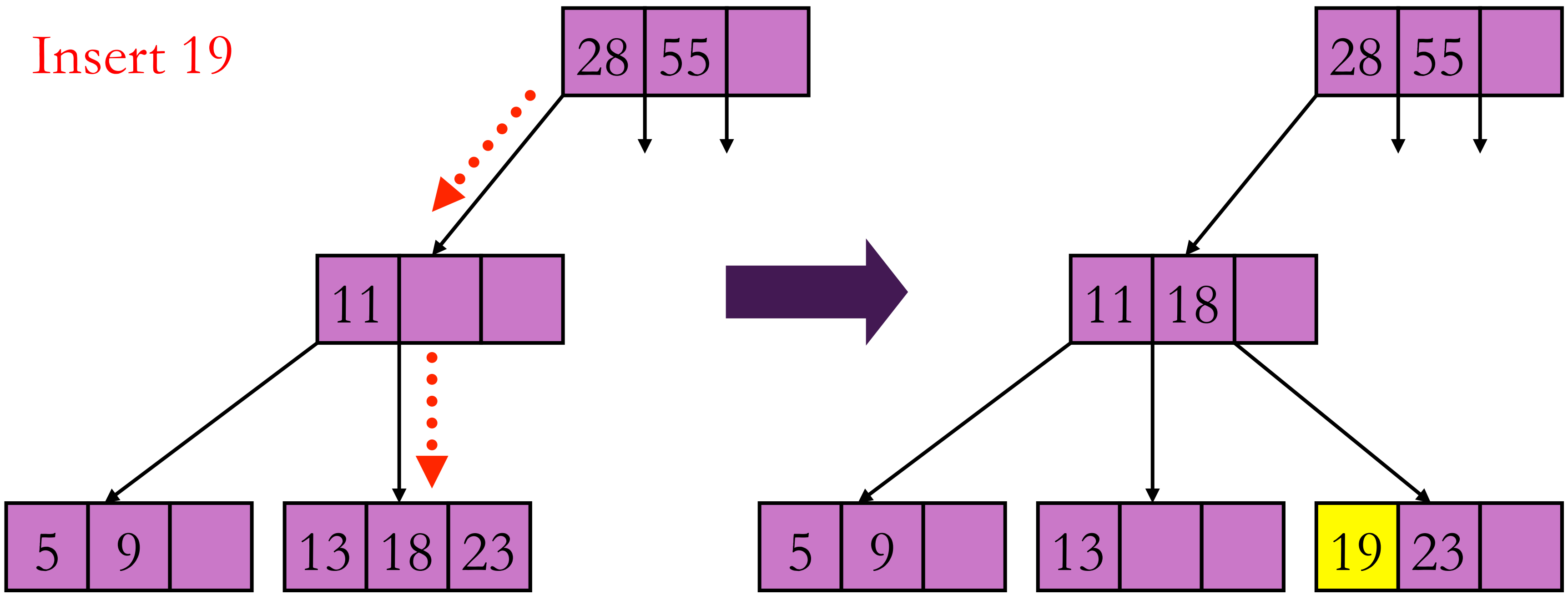


1. Create a new node which is right sibling of A/B/C
2. Move C to new node
3. Move B up
4. Connect y and z to the new node

Splitting Node II (not root)



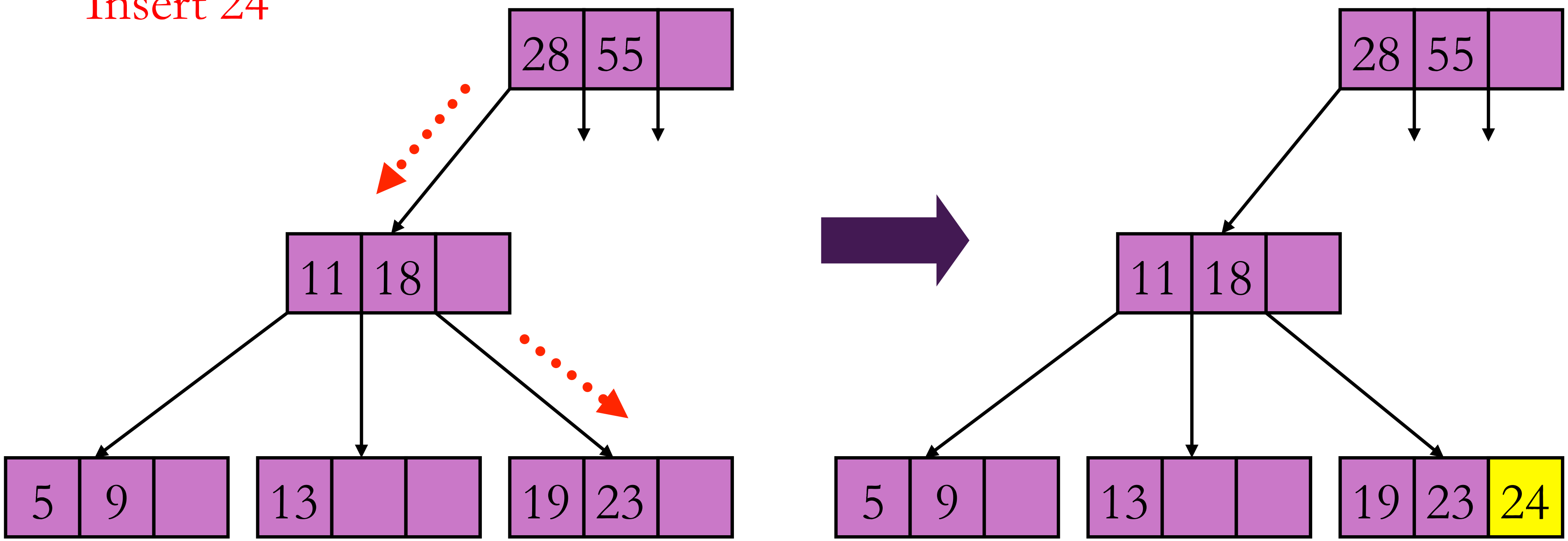
1. Create a new node which is right sibling of A/B/C
2. Move C to new node
3. Move B up, hence move q over (b'cos B < q)
4. Connect y and z to the new node



Splitting Node Scenario I

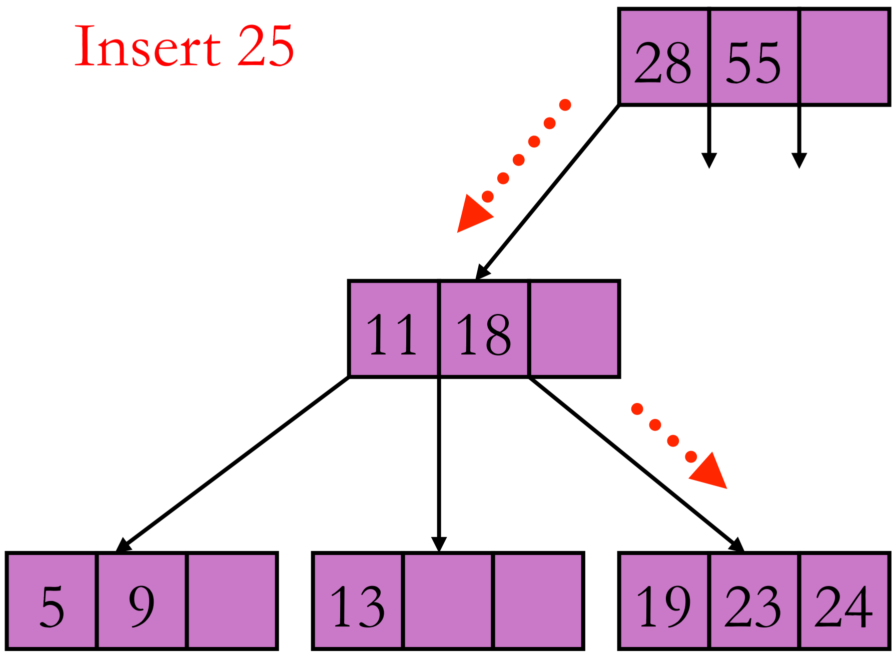
- Create a new node which is right sibling of 13/18/23
- Move 23 to new node
- Move 18 up
- Insert 19

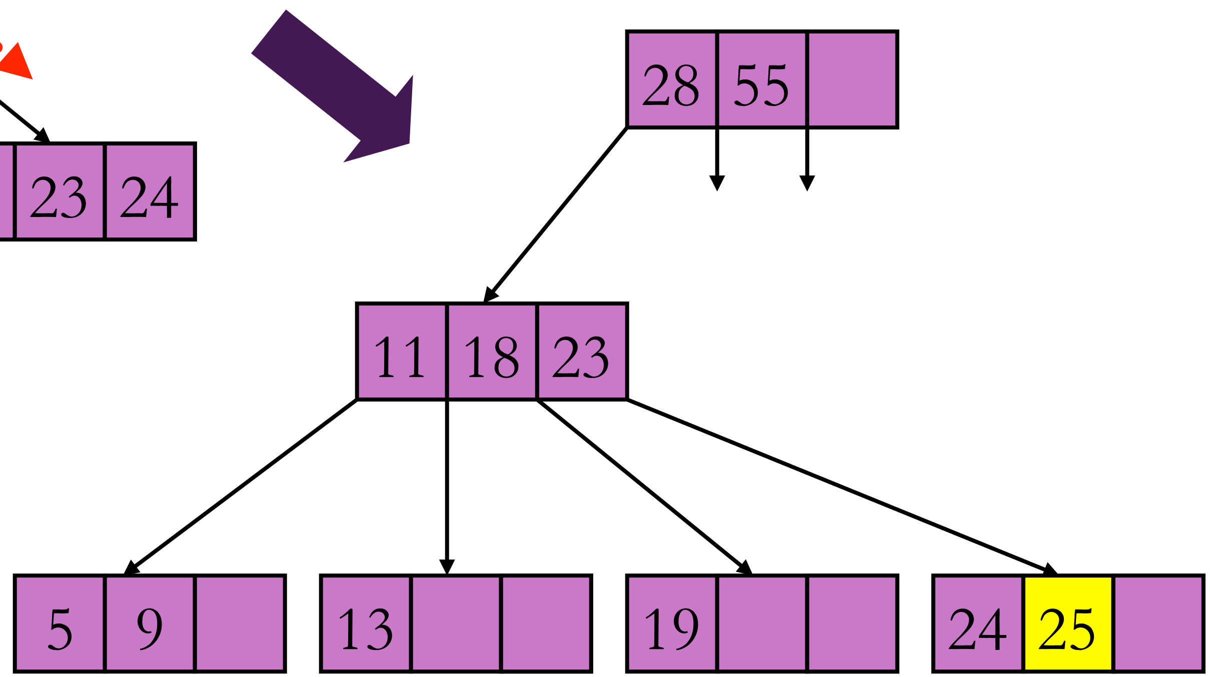
Insert 24

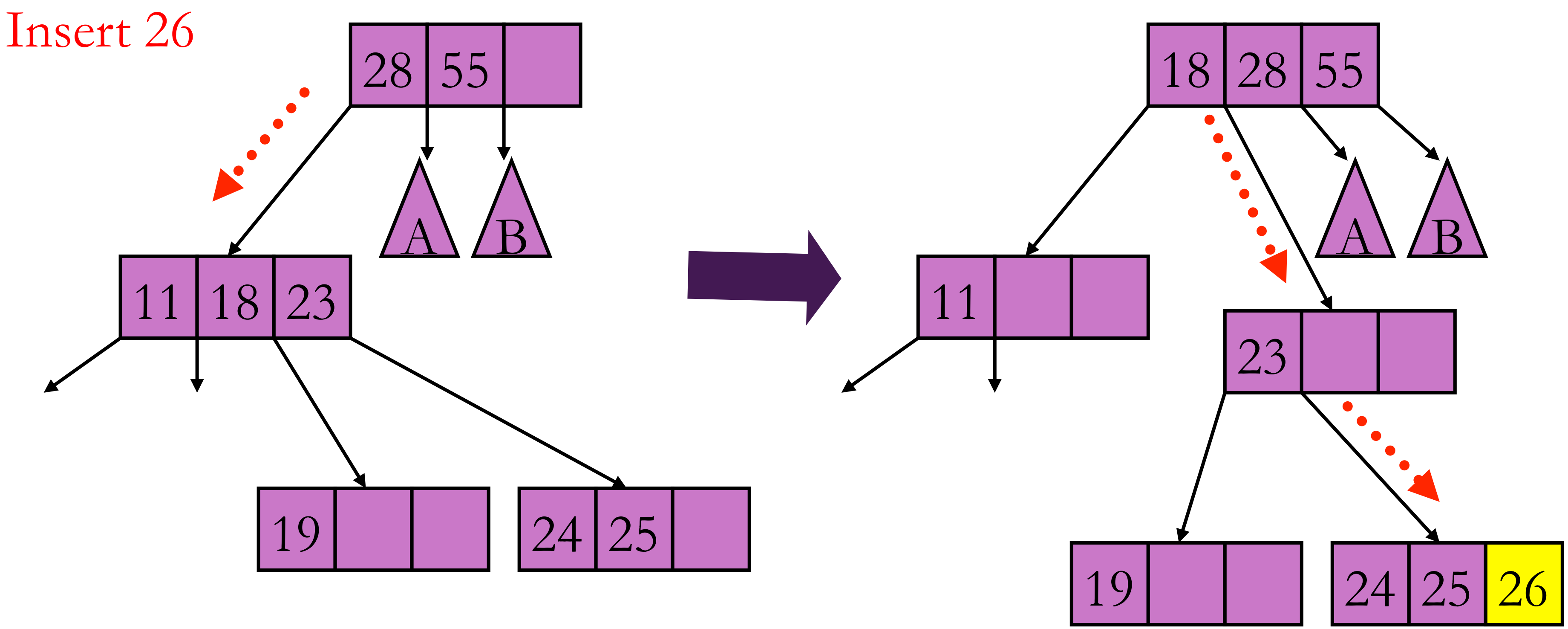


Splitting Node Scenario II

1. Create a new node which is right sibling of 19/23/24
2. Move 24 to new node
3. Move 23 up
4. Insert 25

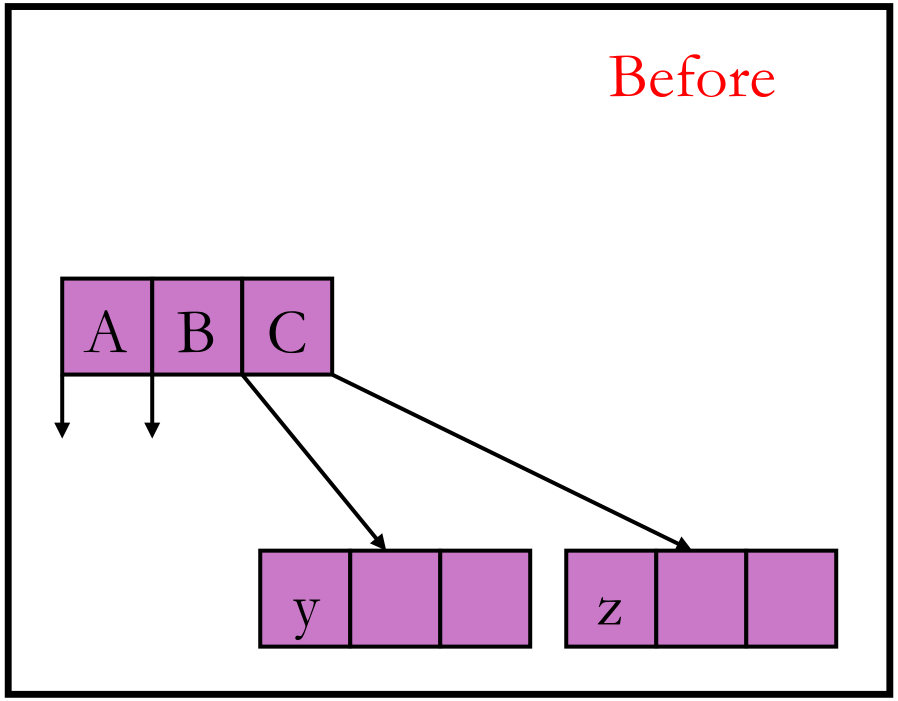


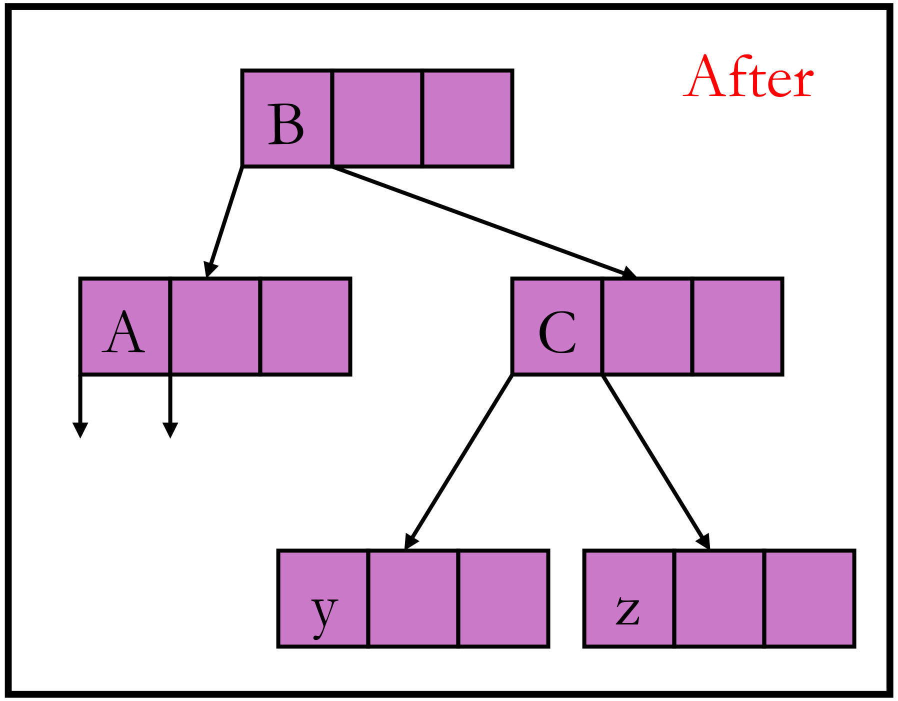




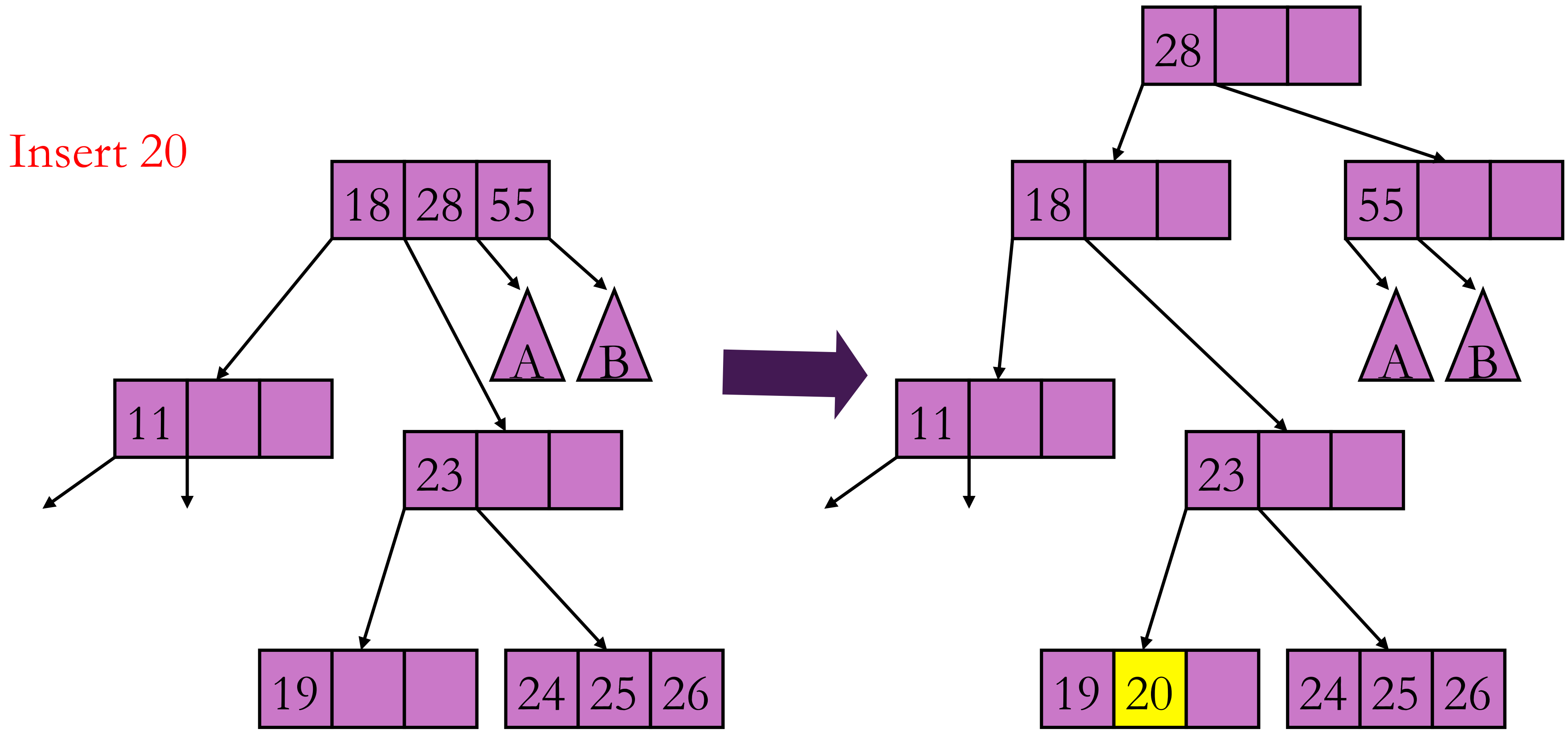
Split 11/18/23, then back up one node and continue

Splitting Node III – the root





1. Create a new root above affected node.
2. make A/B/C left child of this node
3. Create a new node which is right sibling of A/B/C
4. Move C to new node
5. Move B up to new root
6. Move y and z over to new node



Split root, then continue from new root

Summary

- Leaves are all on same level
- Searching is an extension of binary search tree idea
- When inserting, split on the way down
- Splitting on the way down means that parent always has room for “B” (the middle key) to come up
- Moving “B” up may make a full node, but that can be handled on next insert

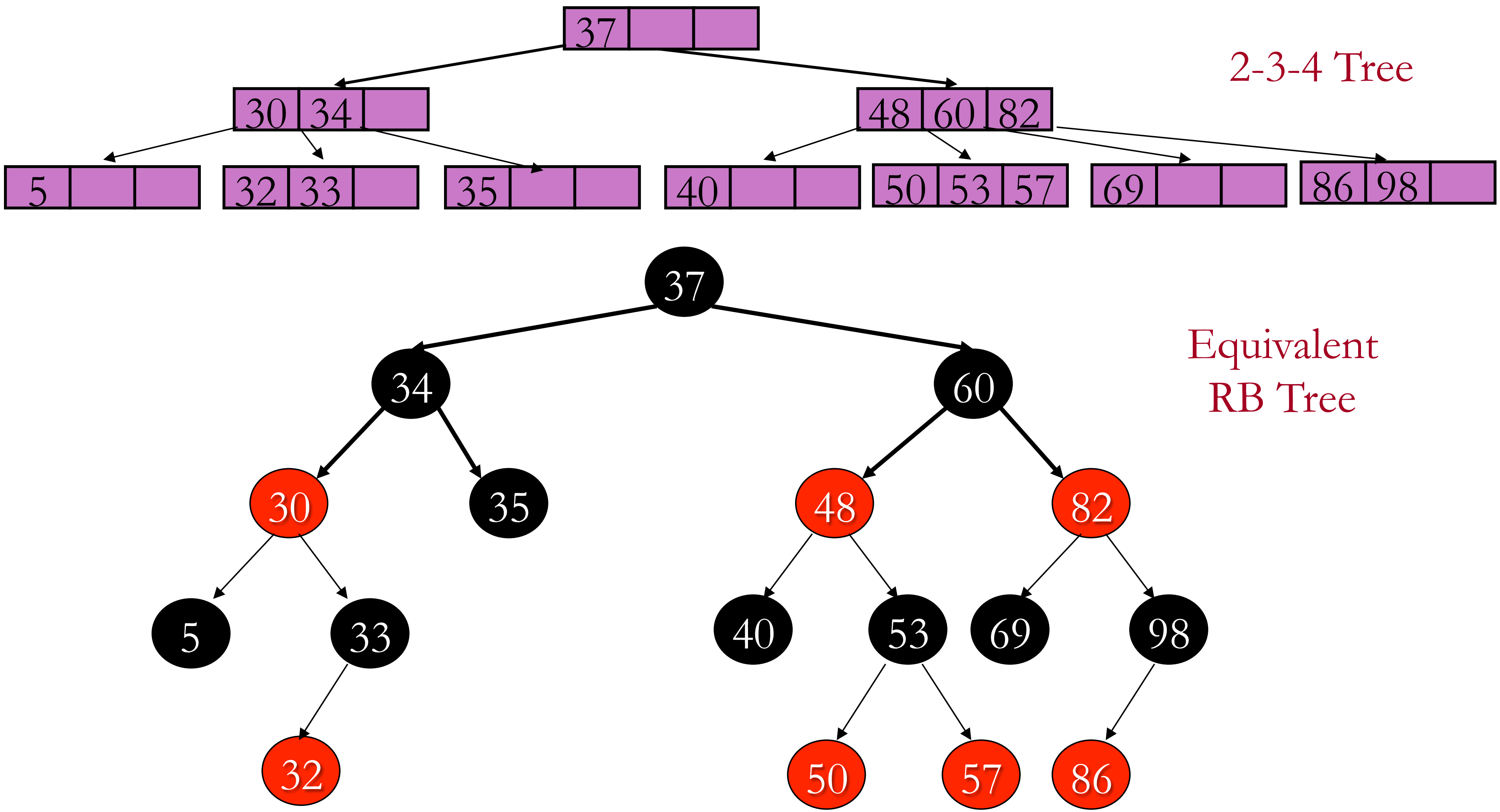
2-3-4 Trees : Time Complexity

- Since a 2-3-4 tree can have four children, the number of levels is actually *fewer* than a binary tree
  - $O(\log_4 N)$ levels – better than $O(\log_2 N)$ of an almost-complete binary tree
  - This implies less steps to find node you are looking for
  - NOTE: Still the same *order* of complexity: $O(\log N)$
- However, each node has 2, 3 or 4 keys
  - 2, 3 or 4 times more processing per node
- Hence total speed is actually slightly slower than Balanced Binary tree

2-3-4 Trees and Red-Black Trees

- Can transform a 2-3-4 tree into a Red-Black tree
  - A 1-key node is a black node
  - A 2-key node is a black node with a red child
  - A 3-key node is a black node with 2 red children
- Note that Red-Black trees evolved from 2-3-4 trees

2-3-4 -> RB example



External Storage of Data

- So far we have talked about trees where data has been stored in RAM memory
- Can also store data on external disk drives
  - Cheaper
  - (More) permanent
  - But slower access compared to RAM
    - Seek time (large)
    - Rotational latency
    - Transfer time (smaller than seek time but still longer than RAM access)

Sequential files

- Write data to disk in sorted order (e.g. phone book using last name as key)
- Memory - Binary search in memory takes $\log_2 N$ probes
- Disk - Same on disk, but now on blocks of data, rather than a single piece of data.

Example

- Phone book of 500,000 entries
- Each record is 512 bytes
- Size = 256 Mb (uncompressed!)
- Assume block size of 8192 bytes
- 8192/512 = 16 records per block
- 31,250 blocks
- BS in RAM = $\log_2 500000$ = 19 probes
  - about 0.2 msec
- BS on disk = $\log_2 31250$ = 15 probes
  - about 150 msec for searching only

But…

- How do we insert/delete in a sorted file onto disk?
- Same as an array: have to move all other records
- Using the previous phone book example:
  - On average 15,625 blocks to move (half of 31,250)
  - Have to read and write
  - About 10 milliseconds per operation
  - More than 5 minutes!

Array vs Tree

- In RAM we can use a binary tree to improve on sorted arrays
- Disks can transfer **blocks** quickly
- So make a tree node the size of a block
- ie a node will contain many records

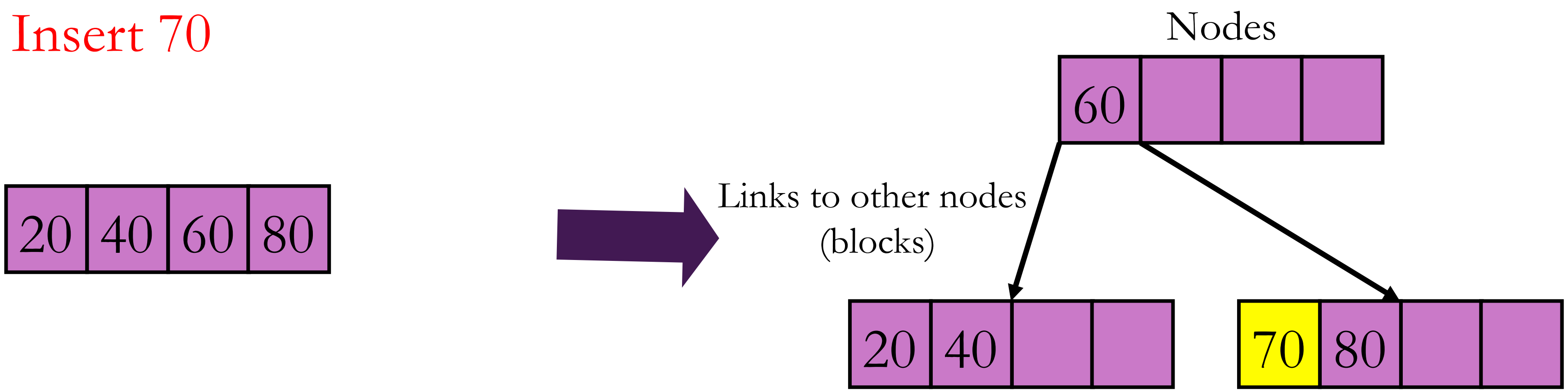
B-Trees – Disk Storage and Access

- B-Trees are used to get fast access to data that is stored on disk rather than in memory
  - ‘B’ for ‘Block’-Tree
- The idea is to store the tree’s nodes on disk, then load the nodes into memory only as needed
  - Each node says where child nodes are *on the disk*
  - Good for databases: look up row based on primary key
    - This is called ‘indexing’ on the key
  - Too much data to store in memory, so have the data reside on disk and use the primary key to navigate the tree

B-Trees

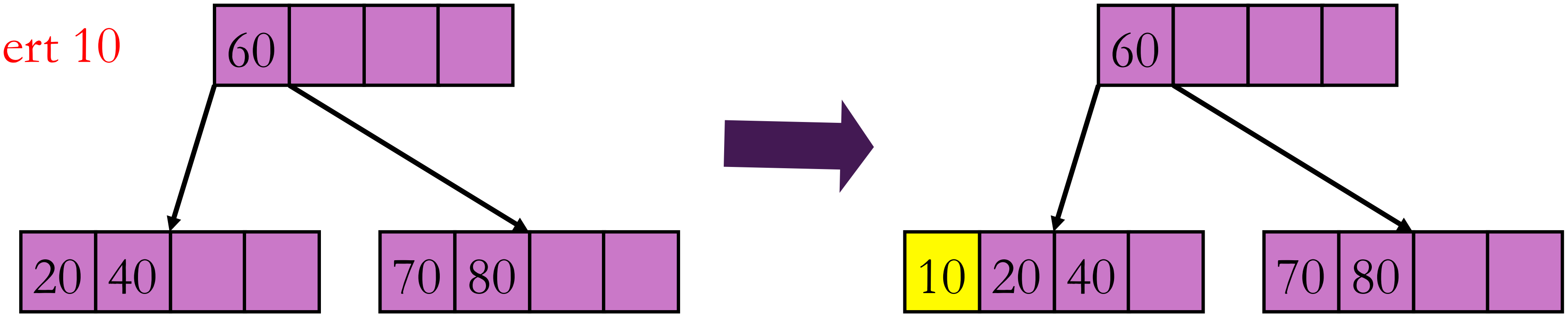
- Uses 2-3-4 style splitting to keep tree shallow
- But we want nodes **as full as possible** so block transfer is not wasted
  - When split, put half in old node and half in new node
  - Middle of all data goes up to parent (ie node keys plus new item)
- Node splits are performed from the **bottom up** rather than top down (as in 2-3-4 trees)

Example (root split)

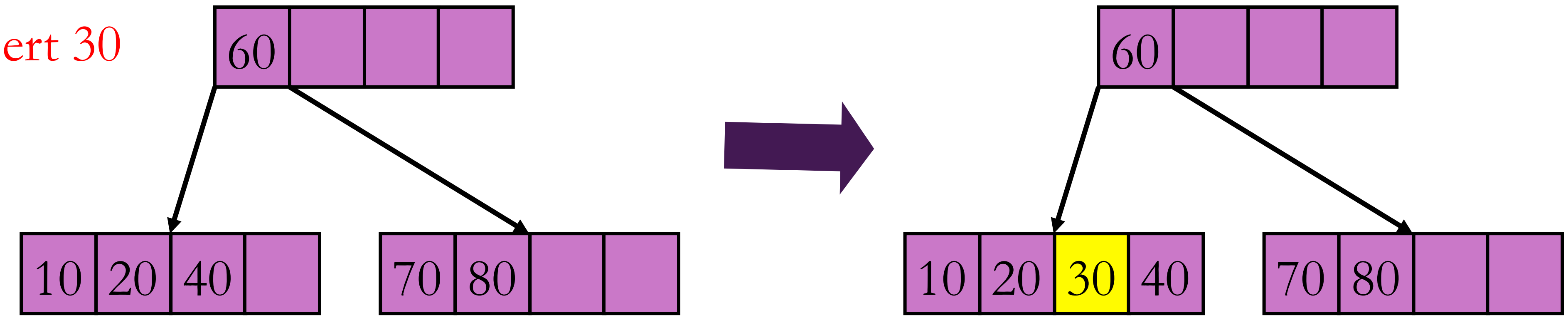


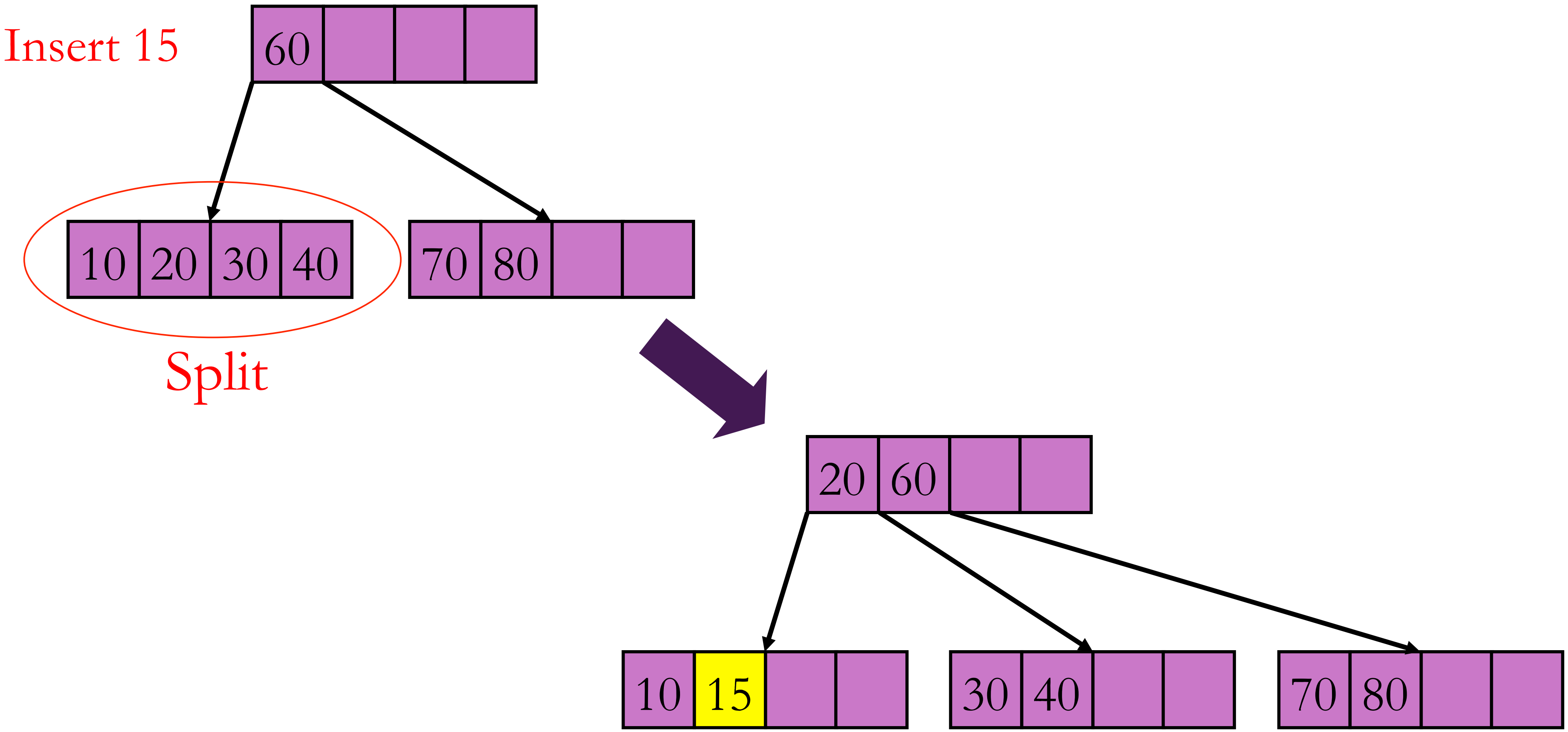
- Data items are sorted: 20, 40, 60, 70, 80
- Middle goes up
- Left half stays put
- Right half goes to new node

Insert 10

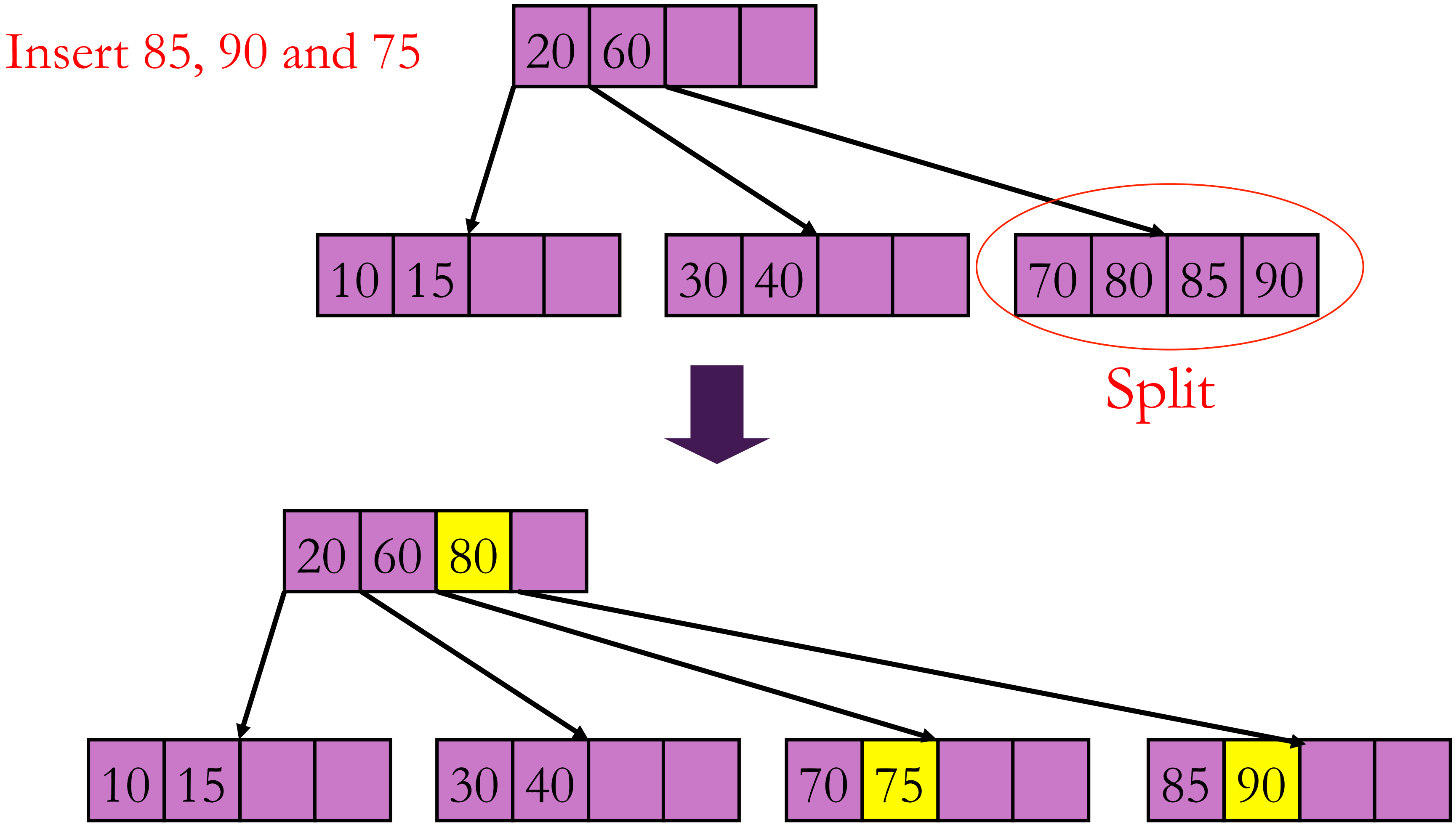


Insert 30



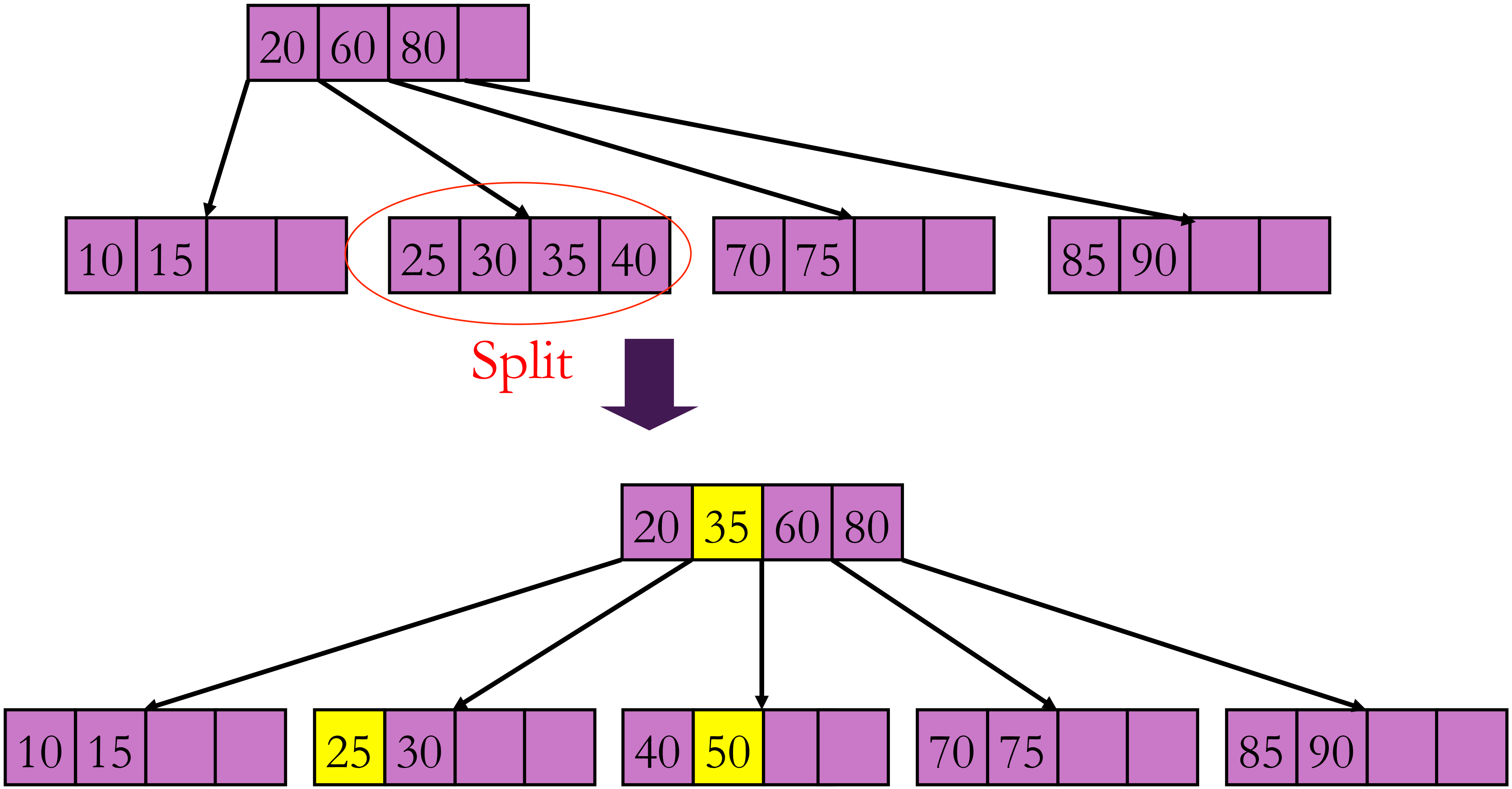


10 15 20 30 40 - Split, 20 goes up



70 75 80 85 90 - Split, 80 goes up

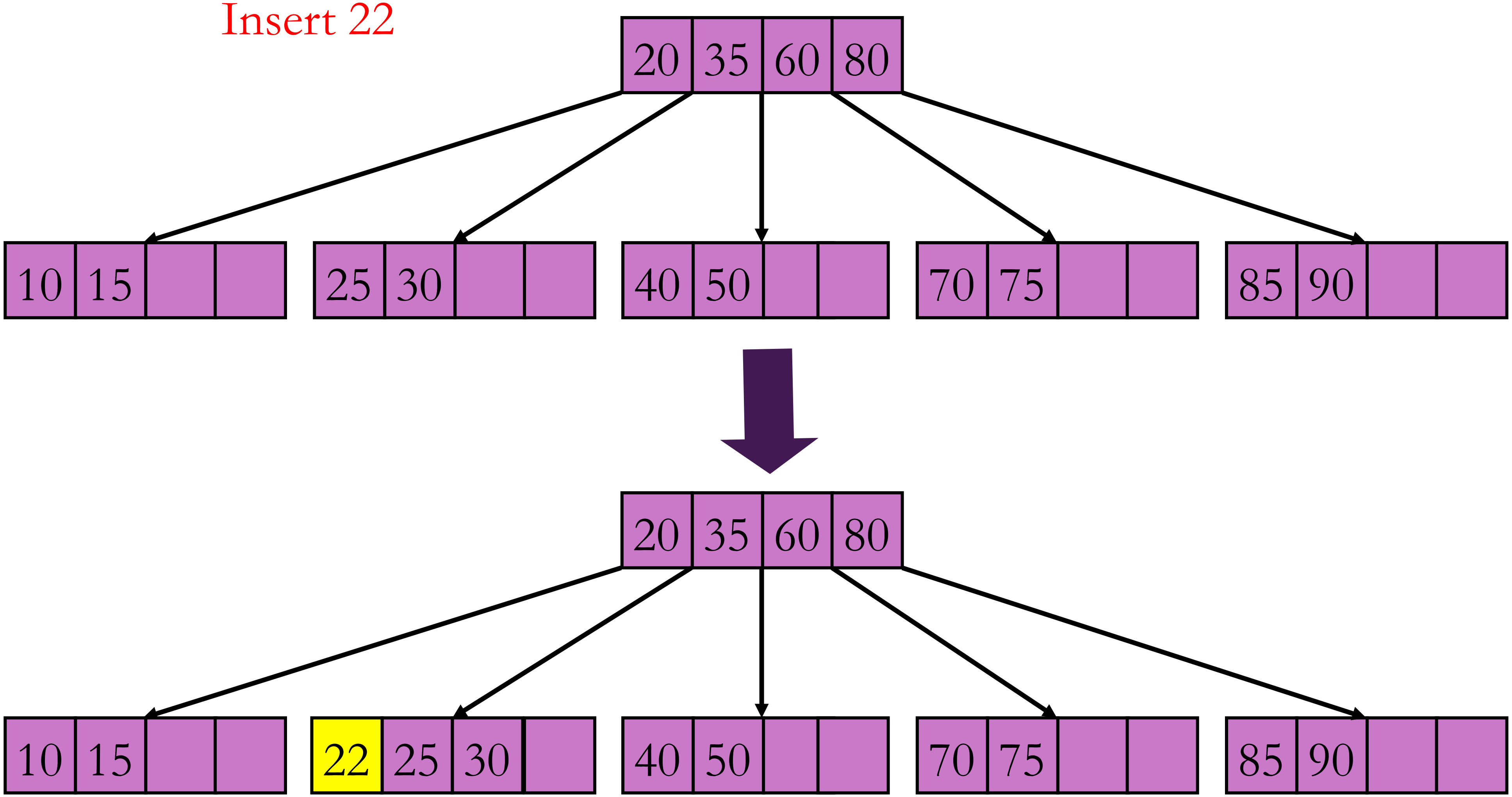
Insert 25, 35, 50



25 30 35 40 50 - Split, 35 goes up

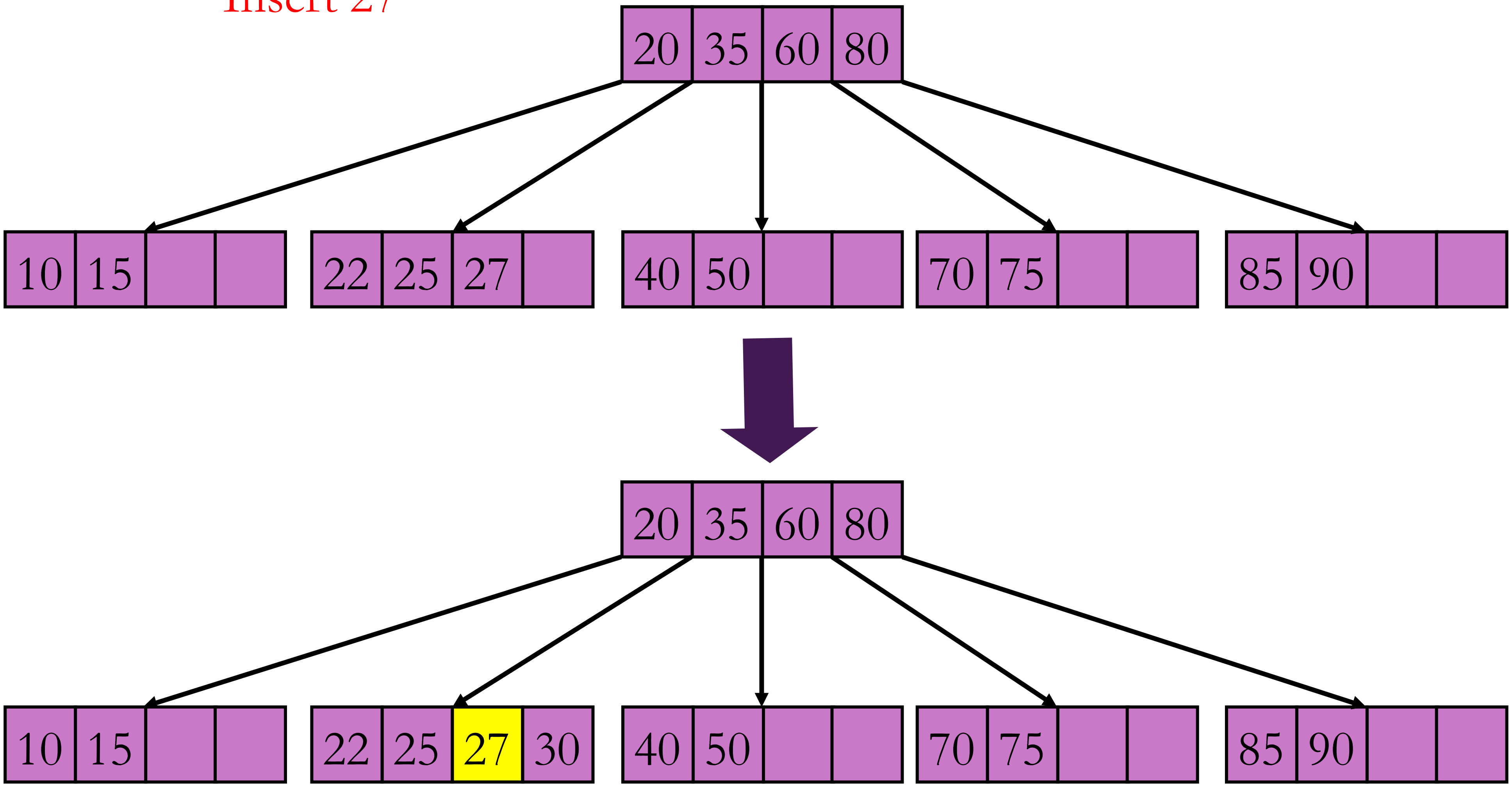
No splits required

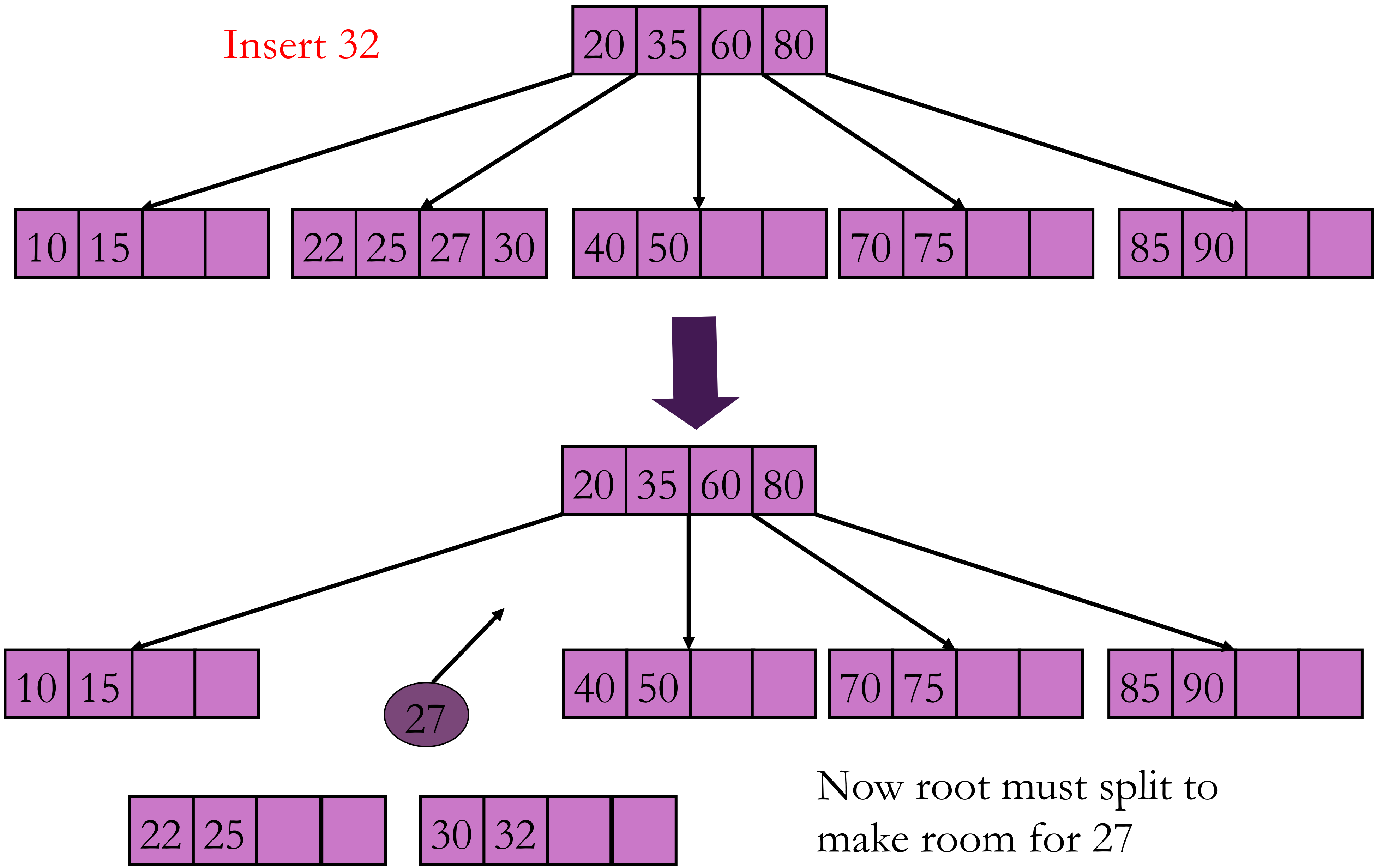
Insert 22



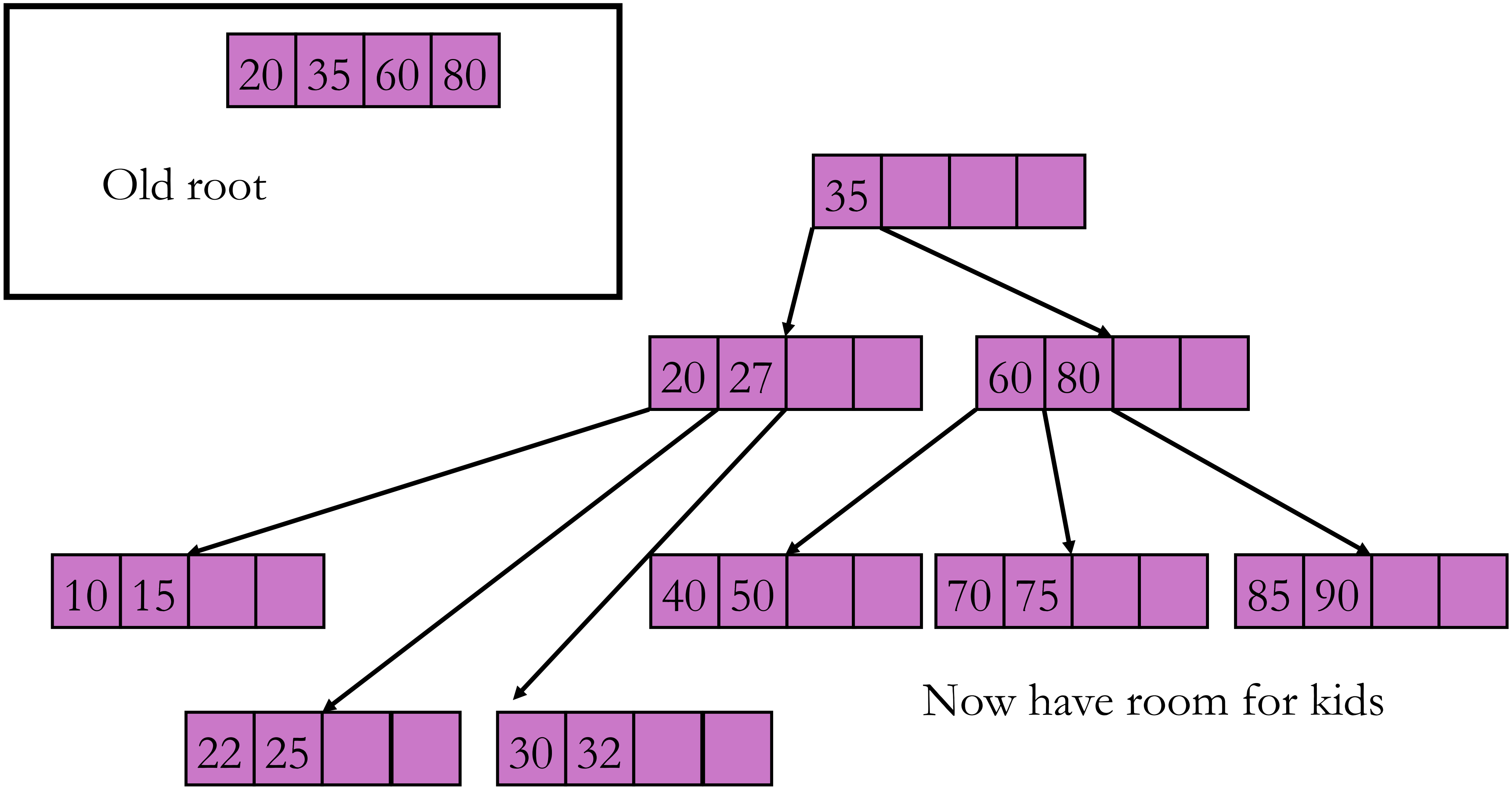
No splits required

Insert 27





Insert 32
20 35 60 80
10 15
22 25 27 30
40 50
70 75
85 90
20 35 60 80
10 15
27
40 50
70 75
85 90
22 25
30 32
Now root must split to make room for 27


20 35 60 80
Old root
35
20 27
60 80
10 15
40 50
70 75
85 90
22 25
30 32
Now have room for kids

B-tree efficiency

- Back to phone book example
- 16 records per block
- Every node is at least half full (except root)
- Say 8 records per block with 9 child links
- So tree is $\log_9 500000 < 6$ levels = approx 60 milliseconds

Next Week

- Data structures and algorithms – beyond DSA...