ITT DUBLIN
Institute of Technology Tallaght
Institiúid Teicneolaíochta Tamhlacht

S e a r c h T r e e s

Koffman et al. chapter 11

Goodrich et al. chapter 9

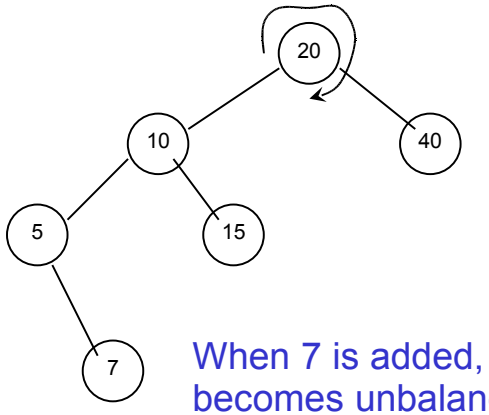
Review of The Binary Search Tree

- we have already looked at the binary search tree
  - The BST stores and retrieves items based on a key value
    - **it is a dictionary data structure**
  - We saw that any item could be found in O(h) time, where h was the height of the tree
  - If the tree is well balanced, then the height will be log(n), so this would be an O(logn) algorithm
  - If items are added completely randomly, we would expect that over time it would remain well-balanced
  - But in the worst case scenario, for example when items added become progressively smaller, the tree would not be well balanced
  - In fact the height might be n (each item at a new level)
  - Worst case analysis for binary search is O(n)
  - No better than that for a sequenced-based unordered dictionary search
- This has led computer scientists to develop more specialised binary search trees as well as alternative non-binary tree structures to use for efficient searching

Tree balance and rotation

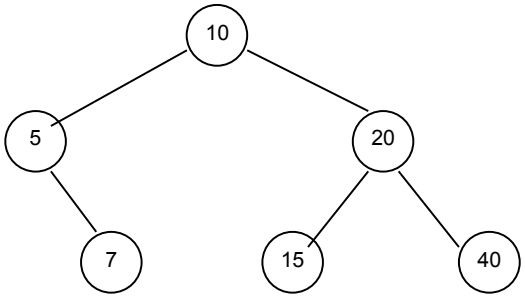
- The key to having a better binary search tree is to make sure that it is always ‘well-balanced’
- To balance the subtrees at a given node without losing the binary search tree property will involve ‘rotation’ left or right of one or more nodes
  - This will change the relative heights of left and right sub trees
- Example:

Start with a balanced tree



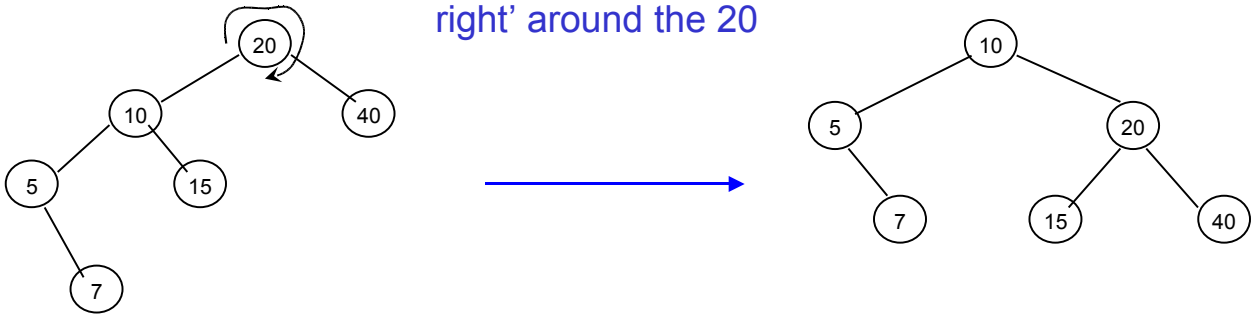
When 7 is added, tree becomes unbalanced

Rebalance by a ‘rotation right’ around the 20



- Intuitively, we can see that we drop the 20 down to become the right child of the 10, and the present right child of 10 will become the left child of the 20
  - The BST property will be maintained because 20 must be greater than 10, its previous left child, so can move to be its right child
  - And both children of 10 must be less than its parent (20) because 10 was a left child, so 15 can be the left child of 20

Right rotation about the root: the algorithm



Exercise for you:

Show a tree that would need a left rotation.

write the algorithm and code for rotateLeft.

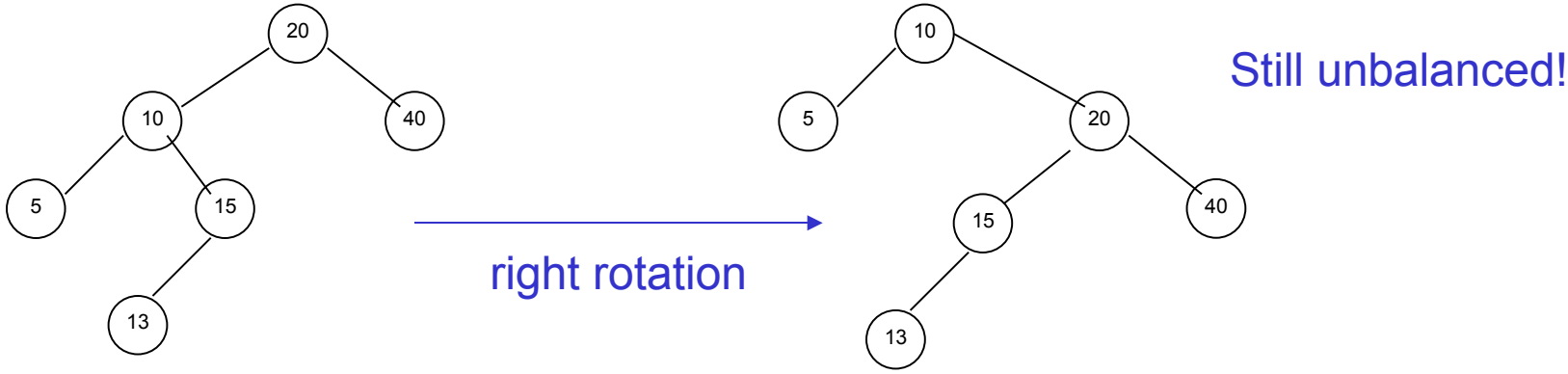
1. Remember the value of root->left (temp = root->left)
2. Set root->left to the value of temp->right
3. Set temp->right to root
4. Set root to temp

```
void rotateRight(BTNode * &localRoot)
{
   BTNode *temp = localRoot->left;
    localRoot->left = temp->right;
   temp->right = localRoot;
   localRoot = temp;
}
```

We have changed the root node … If the 'root' is actually the root of a sub-tree, we would also have to update the child details of its parent

A tree can be unbalanced in many ways (...well, 4 ways)

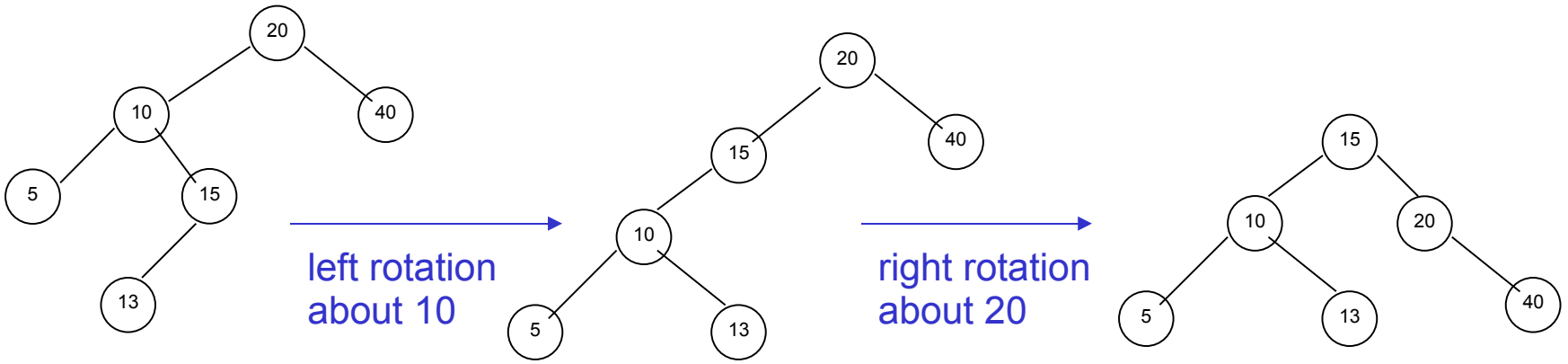
- A single right rotation balanced the tree when we added a 7 … … but suppose we add a 13?



- The reason this is a more complicated problem is that the tree is a left-right unbalanced tree
  - it is the tree rooted at its left child that is higher, but the right subtree of that left child is the childs higher subtree!!
- When we added the 7 before we had a left-left unbalanced tree
  - the left child had a higher left subtree

Dealing with a left-right unbalance

- First perform a left rotation on the left child of the unbalanced node
- Then the right rotation on the root



Exercise: What about a right-left unbalance?

Summary of rotations to fix unbalance at a sub-root

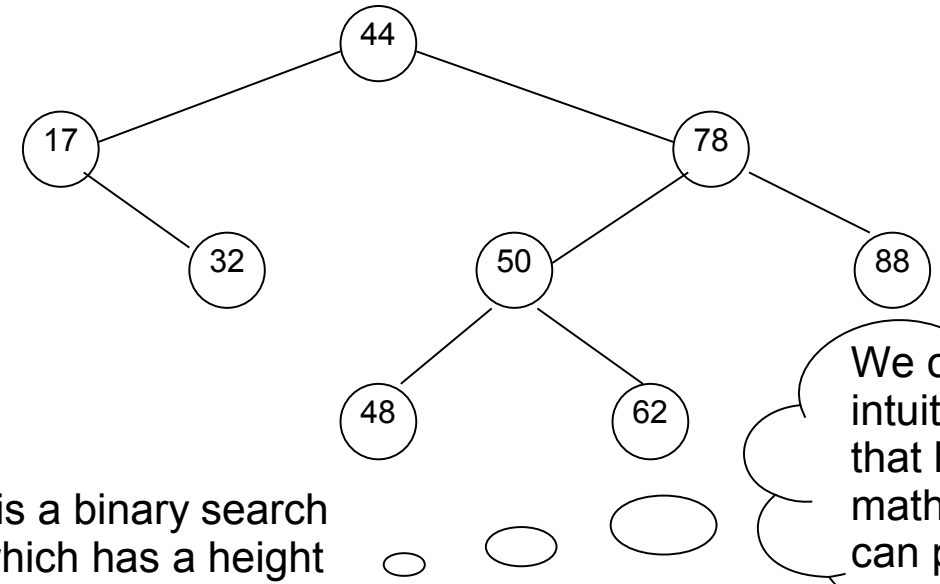
- L-L unbalance:
  - Sub-root right rotation
- L-R unbalance
  - L.child left rotation
  - Sub-root right rotation

- R-R unbalance
  - Sub-root left rotation
- R-L unbalance
  - R.child right rotation
  - Sub-root left rotation

The AVL Tree

named after the initials of its inventors: Adel'son-Vel'skii and Landis

- Must maintain the properties of any Binary Search Tree
- Adds a rule to the binary search tree definition which maintains logarithmic height for the tree
- The height-balance property
  - For every node which has 2 children, the height of the sub trees rooted at each child differ by at most 1
  - If a node has only one child, that child must be a leaf node



As you see, this does not necessarily mean it is a complete tree

We can see intuitively that it has that height, mathematicians can prove it for us.

But it is a binary search tree which has a height of O(log n)

Insertion into an AVL

- Insertion is done in the same way as for any Binary Search Tree
  - But then the tree has to be rebalanced using rotations, so that it remains an AVL tree

- The algorithm for rebalancing after insertion
  1. Start with the inserted node, and move UP the tree, checking that each ancestor is the root of a balanced tree
     - This step will need to be expanded a bit
     - We must have a way to navigate to a parent
     - And to calculate the 'balance' of each node
  2. if an 'unbalanced' ancestor is found, rebalance it.
     - depending on the shape of the tree rebalance may require one or two rotations
     - It can certainly be rebalanced in constant time (not dependent on the number of nodes in the tree)

- the maximum no of moves up the tree in step 1 depends on the height of the tree ($\log n$)
- .. and it could be proved that insertion will require at most ONE re-balance
  - Insertion is $O(\log n)$

Deletion from an AVL

- We first delete the node as for a regular Binary Search Tree, but then have to examine the tree and re-balance it if necessary.
- In the case of deletion, it may be necessary to repeat the re-balancing step at many levels
  - We start with the parent of the leaf node which has been deleted (check back on how to do this to the BST overheads)
  - Walk up the tree, checking for a node which needs re-balancing
  - After re-balancing, continue walking up the tree looking for the next unbalanced node.
- The maximum no of steps up the tree is still determined by the height of the tree
  - deletion can be done in O(log n) time
  - Although it may include MANY re-balances at different levels of the tree

Efficiency of the AVL

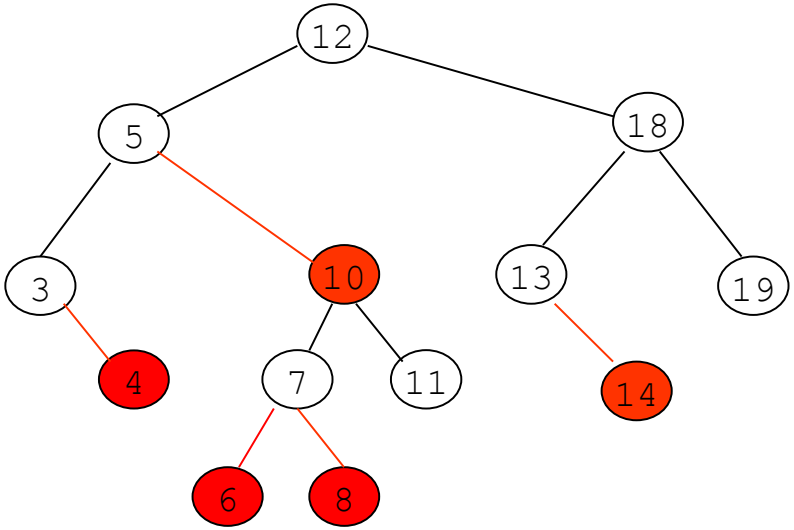
- The height of the tree is always maintained as a logarithmic height, so all algorithms which have a running time proportional to the height of the tree are O(log n)

- We have shown that
  - Insertion can be done in O(log n) time.
  - Deletion can be done in O(log n) time

- And the search itself is dependent on the height of the tree, as it is for any Binary Search Tree
  - Search is done in O(log n) time
    There is **no worst case** as there was for the regular BST

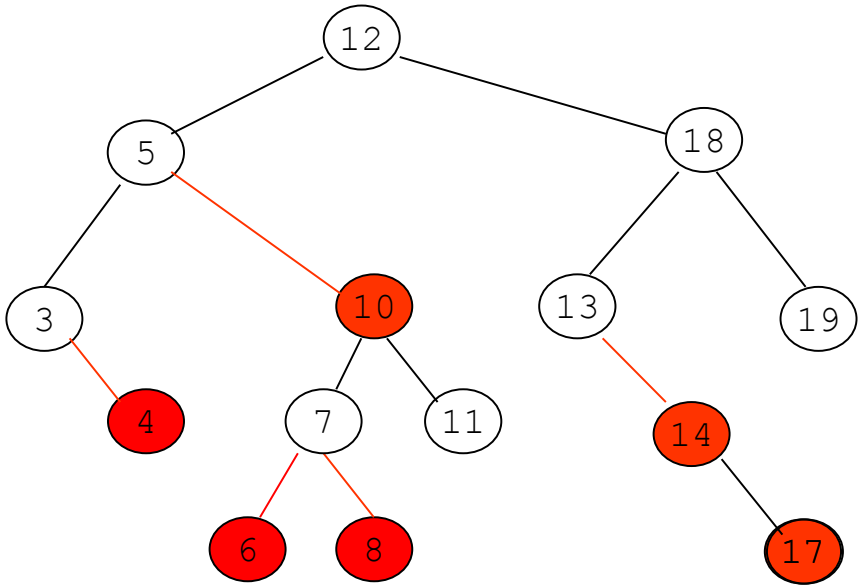
The red-black tree

- The red-black tree is another specialised binary search tree which may have advantages over the AVL in some cases.
- In a red-black tree, we attribute an extra property to each node - the colour, which may be - guess what!
- The rules for a red-black tree are:
  1. It obeys the binary search rule
  2. The root is black (root property)
  3. The children of a red node are always black
  4. Every path from the root to a leaf node has the same no of black nodes



Note that by some standards, this would be considered unbalanced with a L-R unbalance, but it obeys all the requirements for a red-black tree – and requirement 4 can be taken as the definition of balance in this tree

Inserting and removing into a red-black tree - 1



Reminder; The rules for a red-black tree are:

1. It obeys the binary search rule
2. The root is black
3. The children of a red node are black
4. All paths to a leaf node from root have the same no of black nodes

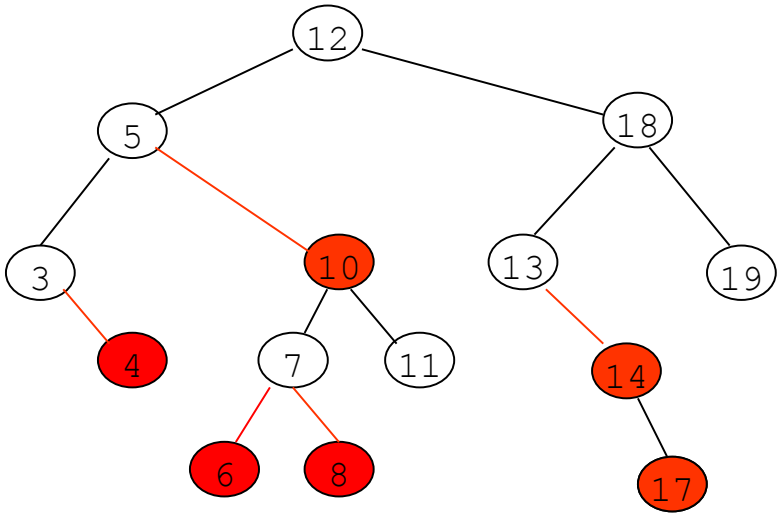
Insertion algorithm

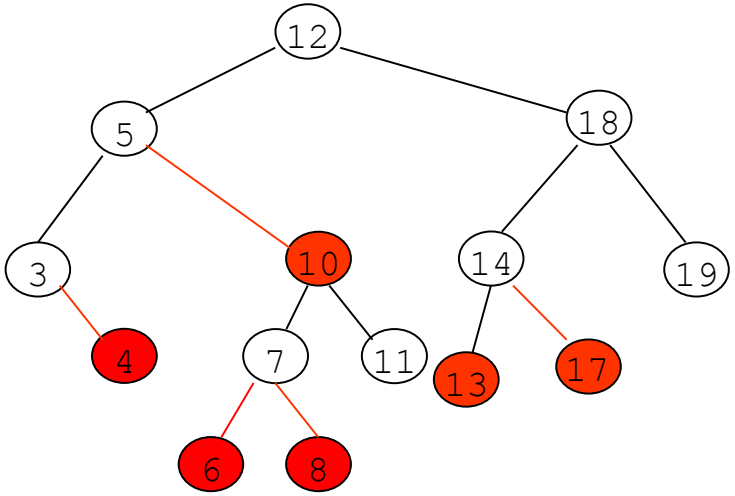
2. First insert new item according to BST rules
3. Then decide on its colour to preserve rule 4
4. Then fix problem if it breaks rule 3

Example: insert 17 into this tree:

- It goes to right of the 14
- If it is black, the path to it from the root has one too many black nodes, so it must be red
- Now we must fix the red-child problem!

Inserting and removing into a red-black tree - 2





- There is an algorithm we will NOT study, which shows how to work up the tree re-colouring wherever necessary and *re-structuring (balancing) at most once* after an insertion
- And it is possible to show that this can always be done in O(log n) time

- Similarly, a removal can be done by working up the tree re-colouring where necessary and *re-balancing at most twice*
- So both are done in O(log n) time

- This is an improvement on the *AVL tree, because that could need many re-structurings* in the case of removal

Multi-way search trees

- Other data structures used for searching abandon the binary tree for a tree where each node can have many children
- Each node of the tree can also store many key values
  - Up to one less than the no of children held by the node
- You can see that a binary tree is a special case of the multi-way tree
  - Each node stores one key value, and has 2 children
  - one or other of its children may be NULL, that is allowed
- A multiway tree may have some advantages (although it's operations are still O(log n), the re-structuring steps may be simpler in some cases)
- One multiway tree which gives such benefits is the 2-4 tree.
  - In a 2-4 tree each node can have up to 4 children, and it is also a requirement that every leaf node is at the same depth
  - Actually, a red-black tree and a 2-4 tree have a correspondence, each could be re-drawn as the other

Appendix - Expanding the AVL algorithm

- In order to manage the 'walking up the tree and checking the balance' bit we will need to provide some extra info with each Node.
- A proposed first cut:

```
class AVLNode
{
   friend class AVLTree;
public:
   AVLNode(int val);
private:
   AVLNode *parent;
   AVLNode *leftChild;
   AVLNode *rightChild;
   int value;
   int balance;     // -1 => left tree is 1 higher.
                    // +1 => right tree is 1 higher,
                    //  0 => right and left same height
   void attachLeft(AVLNode *node);
   void attachRight(AVLNode *node);
}
```

AVL 'Attach left' pseudo-code

```
set left child = node to attach
set parent of node to attach
decrement balance
If balance is now zero, finished (no heights increased)
If balance is -1, (the only other option)
//walk up the tree updating balances
      tempNode = parent
      while parent is not NULL and no rebalance done yet
          if we are left child of the parent, decrement parent balance
          else increment parent balance

          if parent balance is not -1, 0 or +1,  rebalance parent
          else continue walking up the tree: tempNode = tempNode->parent
      end while
End if
// can also stop early if parent balance updated to 0, no heights changed above
    this
```

I will provide code for this AVL algorithm on moodle