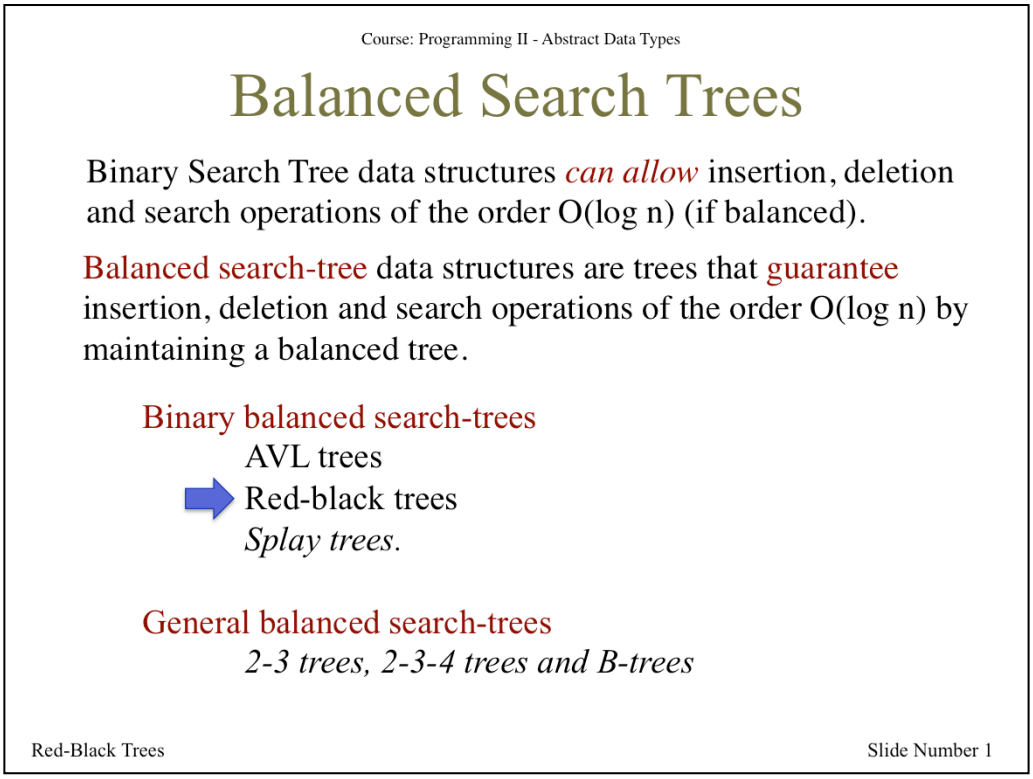
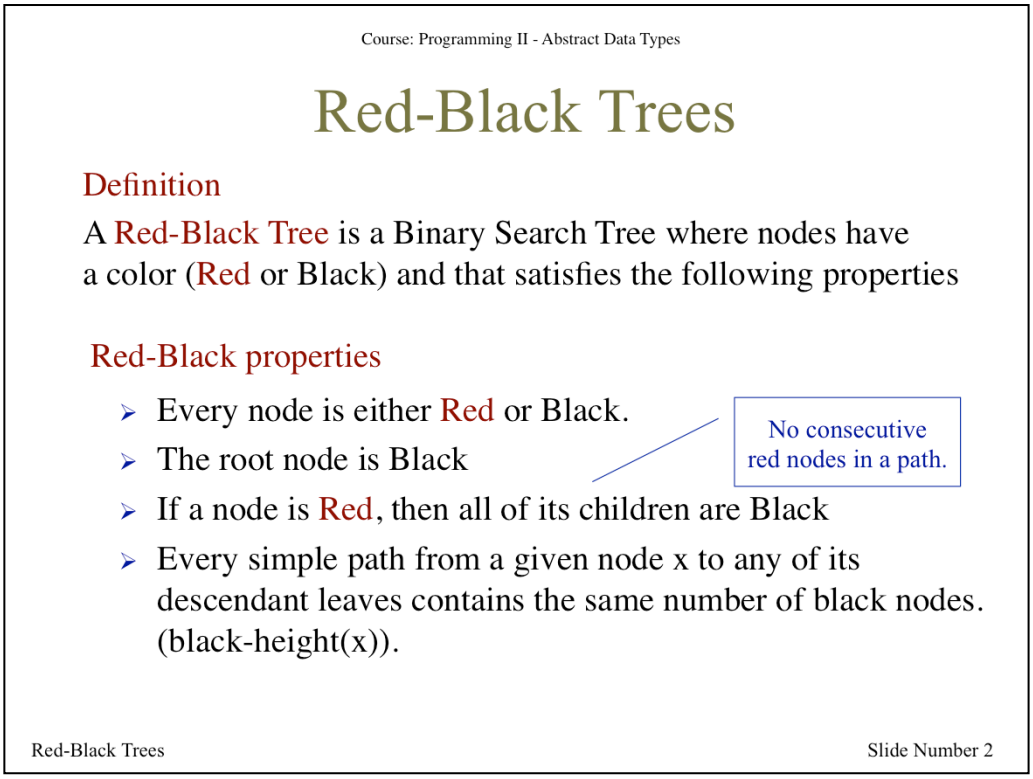


So far we have seen that Binary Search Trees are data structures that, on average, represent an improvement over linked lists or array lists, in the operation of insertion, deletion and search. In lists these three operations take linear time (i.e. are of the order O(n)) with respect to the number of elements in the structure. In a binary search tree the three operations (in particular insertion and search) can be of the order O(log n) provided that the tree is balanced (or maintained balanced). The example operation of deleting an element from a binary search tree that we have seen in Unit 5 tries to maintain the tree as much balanced as possible but it does not guaranteed the height (h) of the tree to be always equal to h = log n.

Balanced search-tree data structures are instead data structured that guarantee search, insertion and deletion to be always of the order O(log n) and they do so by maintain the search-tree structure always balanced, **i.e. a search-tree whose height is of the order O(log n).** Balanced search-trees can be general trees and examples are 2-3 trees, 2-3-4 trees or B trees. These are trees whose nodes may have up to a given constant number of children. For instance 2-3-4 trees are still search trees (with elements sorted according to their key value), whose nodes may have zero, 1, 2, 3 or 4 children. You may encounter these types of general balanced tree in algorithm courses. In this course we concentrate on binary balanced search-trees. These are essentially binary search trees with extra properties that need to be satisfied by the insertion and deletion operations in order to maintain the height of the tree to be of the order O(log n).

There are three types of binary balanced search trees: AVL tree, red-black trees and splay trees. We have seen in the previous lecture (Unit 6) the AVL trees and how the type of rebalancing mechanism that these trees use. The notion of **balanced tree** in these data structures is that for each node, the absolute value of the different between the left and right sub-trees has to be no more than 1. It is easy to see that this property guaranteeing the height of the tree is be always of the order O(log n). The longest path in the tree will always be 1+length(shortest path) of the tree.

We will consider in this lecture another type of binary balanced search tree: the **red-black tree**, and we'll see how the differ from the AVL trees. We will not be able to cover splay trees (but they are simple)



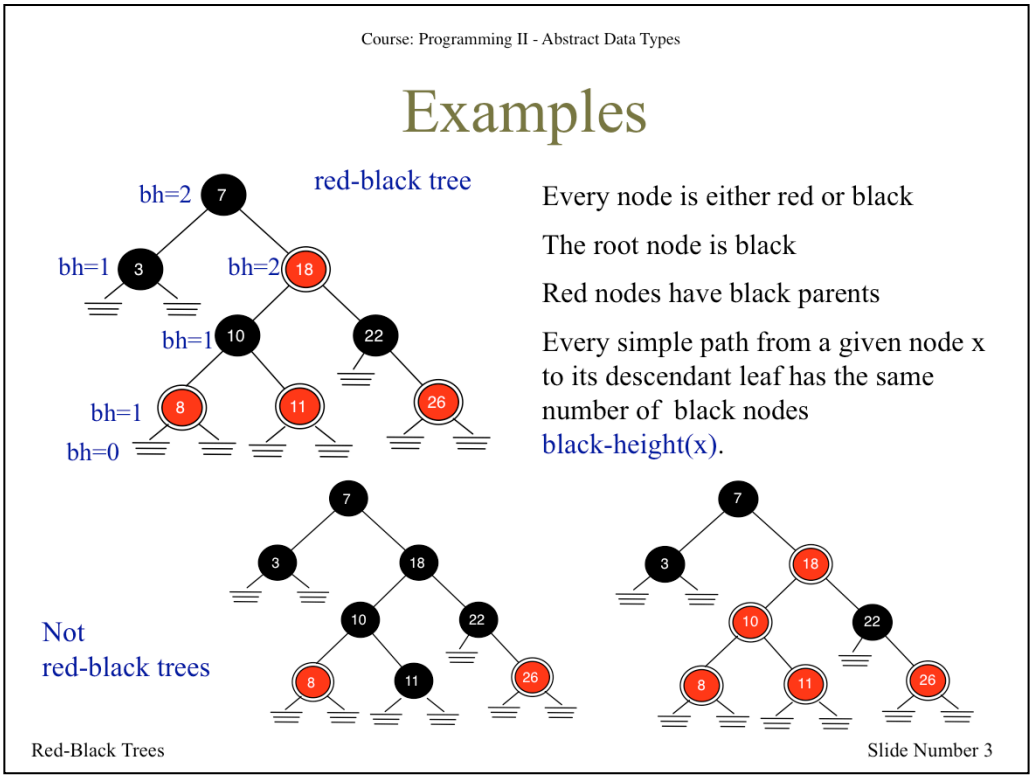
Red-Black Trees are a special type of binary search tree structured that guarantees the search tree to be always of height O(log n) during dynamic insertion and/or deletion of nodes.

They are binary search trees that have a little bit of extra information in each node: the color field (red or black) and that satisfy the above four red-black tree properties. The first property above (every node has to be either black or red is essentially captured by the underlying data structure (i.e. node class used) and by the constructor of that class (i.e. guarantee that whenever a node is created the color field is not undefined; which color to set it to will be shown later). The next three properties are the key properties that have to be maintained by the insert and deletion operation. We will see in this lecture only the insert operation as this is what you will be covering mainly in your PPT lab.

A red-black tree, in addition to the requirements imposed by the fact that it is a binary search tree (i.e. specific order over the keys in the left sub tree, root and right subtree), has to satisfy the fact that the children of a red node must be black and that every path from a given node x to all its descendent leaves must have the same number of black nodes. This also gives us that number of black nodes in every path from the root of the tree to a leaf or to a node with only one child, is the same. Note that the property that children of red nodes must be black can also be expressed by saying that every red node must have a black parent node. In other words in every path from the root to a leaf or a node with one child, there can never be two consecutive red nodes. In every paths nodes must alternate red, black, red, black, etc.. or you can have consecutive black nodes, but you must never have two consecutive red nodes.

Note that the extra field does not increase the space complex of these trees with respect to binary search trees as it is just a single bit information (true of false/ 0 or 1). In the rest of these slides I will use color to visualize the red nodes and the black nodes better but I will also use double circle for red nodes and single circle for black nodes as the printout is in black and white.

The key question is how can the above properties guarantee that the tree has height of the order O(log n)? Why do we care about these properties? Let's see first and example.



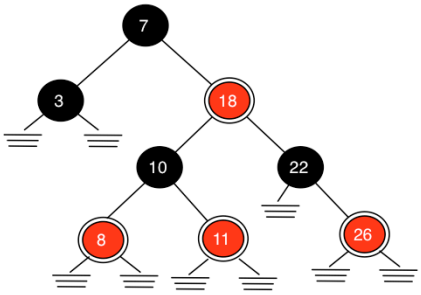
So, as you can seen in the top tree drawn here, the tree is a binary search tree. If you consider the number value as the key according to which the nodes are ordered, for every node the elements that are in the left subtree of that node are smaller than the element in that node and the elements that are in the right subtree of that node are bigger that the element in that node. The root node is black. There are no consecutive red nodes in any path and the more complex property about black-height of a node is also satisfied. To show this we would like for every node to measure the number of black nodes (excluding itself) that are present in every path starting from that node and ending to a null node and show that this number is the same for every path starting at that node. In other words we want to compute for every node its black-height. This assumes that null nodes are black. The black-height of a null node is 0 (since you cannot count the node itself). The black-height of node 8 is 1 and this is the case for every pat starting from 8. The black-height of node 10 is also 1 since you cannot count the node itself and every path from 10 has only one black node (the null node), etc.

What about if we had all the nodes in the above tree black? Would it still be a red-black tree.

Two examples of differently colored node f the same tree structure that are not red-black trees are given at the bottom of this slide.

Height of a Red-Black Tree

Theorem 1: A red-black tree with n elements has height
$h \leq 2 \log(n+1)$



Intuition:
Merge red nodes into their black parents

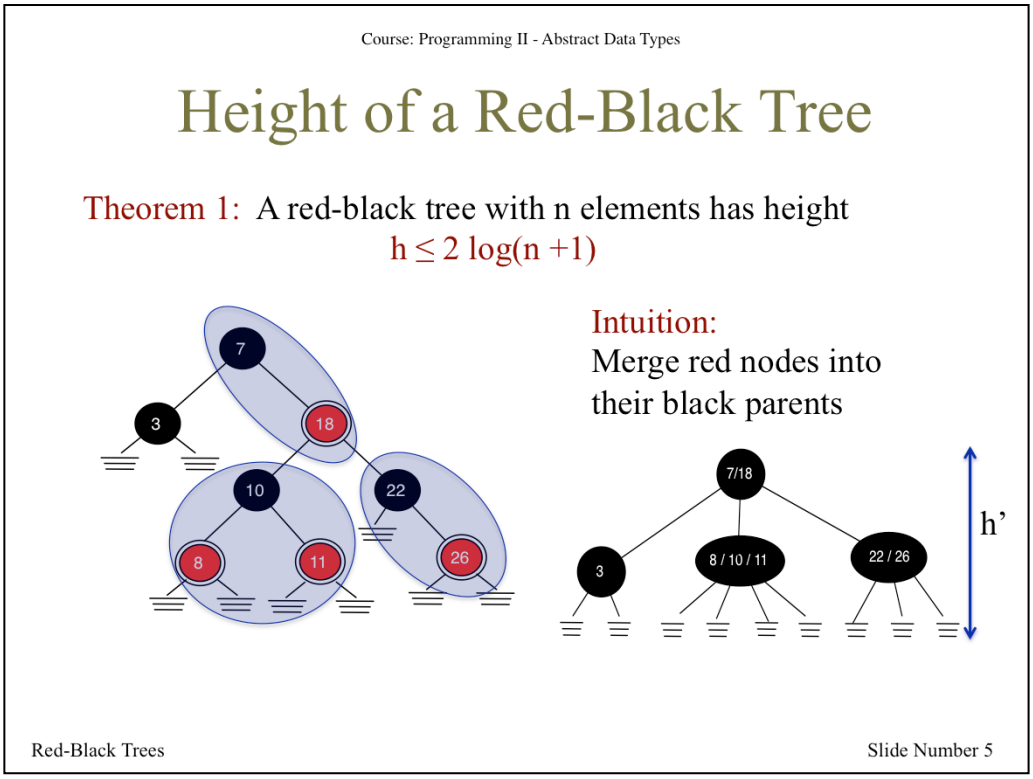
These properties may seem a bit arbitrarily defined, but actually have interesting consequences. One of the key goals that these properties try to achieve is that they should force a tree to have a height of the order O(log n). They indeed do so although it may not be that obvious. Properties 3 and 4 in slide 2 are the properties mainly responsible for guaranteeing this feature of our binary search trees.

The second goal or desire is that these properties should be easy to maintain, i.e. to re-establish during an insertion or a deletion of an element from the tree. We do not want the process of maintaining these properties to be too computationally expensive. They should still be allow the insertion and deletion operation to be of the order O(log n).

We will first see the intuition behind the proof of this theorem that shows that a red-black tree has a height of the order O(log n). Because of Properties 3, we know that every red node must have a black parent. We have also seen that the black-height of a node (and therefore of a black node) is the number of black nodes that are included in every path coming out from that node to the null leaves. In a sense the notion of black-height of a node ignores the number of red nodes that are in a path. Since we cannot have two consecutive red nodes, a key characteristic of red-black trees is that the longest path from the root to a null leaf can only be twice as long as the shortest path from the root to the leaf. Intuitively, this is because if we consider such a shortest path as P1, the root node would have a black-height given by the number of black nodes that appear in this shortest path. Since this number has to be the same in all other paths from the root node any other longer path would have to have additional red nodes (cannot have additional black nodes because this would violate property 4 of red-black tree). Also because by property 3 we cannot have two consecutive red nodes, the only way that the longest path can be made of that length by alternating any black node (in the shortest path) with a red node. So by adding a number of red nodes equal to the number of black nodes that are in the shortest path. Hence the length of this longest path would be twice bigger than the length of the shortest path.

So we can argue that if h' is the length of the shortest path in a red-black tree, the length of the longest path is given less than or equal to 2 h' (namely $h \leq 2\ h'$).

The question is how are we going to show that the length of the shortest path has as upper bound log (n +1). This is briefly shown in the next slide.



The resulting tree is what is known also a 2-3-4 tree, where nodes can have 2, 3 or 4 children. The property of this tree is that all null nodes have the same depth. This is because of Property 4 of the red-black trees. The height of each node is its own black-height. This is indeed defined in terms of black nodes that appear in the path. So since we have removed only the red nodes, for each black node left the length from that node to the null node is now the same as the black-height value of that node. Since by Property 4 all paths from a black node to its descending null leaves have the same number of black nodes, now all paths from a black node to its descending null leaves have the name of nodes (i..e the same length). This holds also for the root node. So all the null nodes have the same height. And the height of this 2-3-4 tree is the black-height of the root node.

This shows us that we can transformed a red-black tree into a 2-3-4 tree which is pretty balanced (as the null leaves are all at the same level). Let's call the height of the 2-3-4 tree h' and the height of the initial red-black tree h. We can prove the theorem by showing that h' has as upper bound log(n+1) and, since $h \leq 2\ h'$ because of what we argued in the previous slide, we can conclude that $h \leq 2 \log (n+1)$.

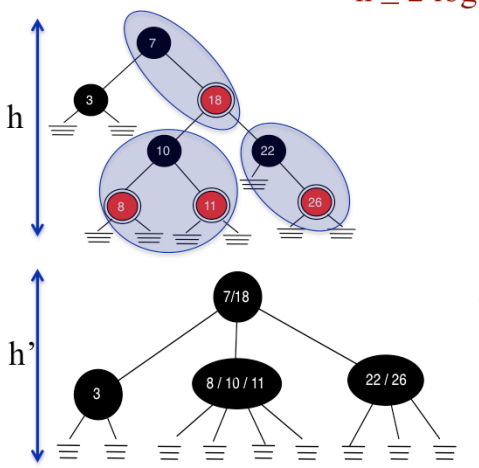
It is possible to show that in a binary tree where every node has two children, the number of null leaves with n (not null) nodes is given by n+1. So, the 2-3-4 tree has n+1 leaves since we haven't removed any null leaf from the red-black tree when we have merged the red nodes with their parents. Moreover, in the 2-3-4 tree the minimum number of leaves that the tree with height h' could have is 2^{h'}. So we can say that $2^{h'} \leq n+1$, which gives us that **$h' \leq \log (n+1)$.**

Because of Property 3 of red-black trees we know that $h \leq 2\ h'$. So we can conclude that

$h \leq 2 \log (n+1)$, which is what we wanted to prove.

Height of a Red-Black Tree

Theorem 1: A red-black tree with n elements has height $h \leq 2 \log(n+1)$



- Number of null leaves in each tree is n+1
- Number of null leaves in a 2-3-4 tree is at least $2^{h'}$

$2^{h'} \leq n+1 \Rightarrow h' \leq \log(n+1)$

Also $h \leq 2\, h'$

$\hookrightarrow$ $h \leq 2 \log(n+1)$

We can then conclude that Properties 3 and 4 of red-black trees are sufficient to guarantee that such a tree will have height less than of equal to 2 log (n+1), i.e. of the order of O(log n), where n is the number of nodes (keys).

In general red-black trees are not fully balanced, or perfect or complete trees but their height has an upper bound of 2 log (n+1), which makes the insertion and deletion operations of the order O(log n).

Interface for Red-Black trees

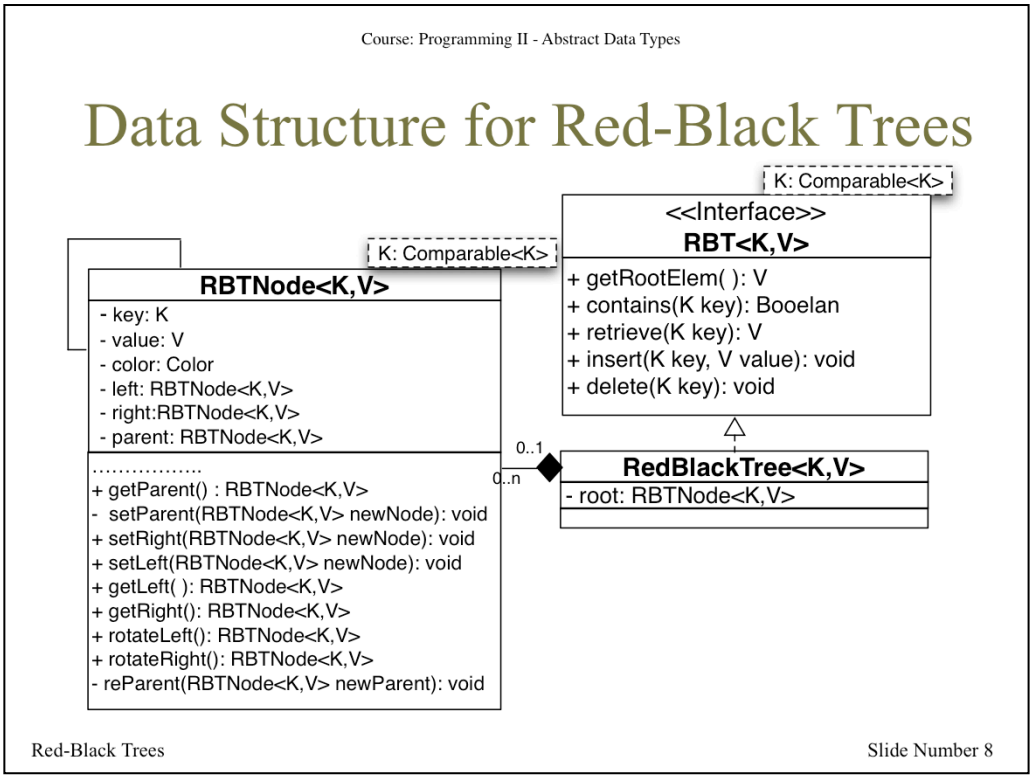
```
public interface RBT<K extends Comparable<? super K>,V>{
    public boolean contains(K searchKey);
    //Post: Searches for an element with key equal to searchKey in the tree.
    //Post: Returns false if no element is found.
    public V retrieve(K searchKey);
    //Post: Retrieves an element with key equal to searchKey from the tree. Returns either the value of
    //Post: the element if found, or null if no element with given searchKey exists.
    public void insert(K key, V newValue);
    //Post: Adds a new element to the tree with given key and newValue, preserving the properties of
    //Post: the red-black tree. If the tree includes already an element with the given key, it replaces its
    //Post: value with newValue.
    public void delete(K searchKey) throws TreeException;
    //Post: Removes an element with key equal to searchKey from the tree, preserving the properties of
    //Post: the red-black tree. Throws an exception if such element is not found.
}
```


As you can see the interface of a red-black tree is the same as that of a binary search tree. The three key operations are retrieve, insert and delete.

The retrieve operation, as any other query access procedure, that can be defined does not modify the structure of the tree and therefore it does not affect the four properties. A given red-black tree will remain red-black tree after the retrieve of an element has been performed. Since a red-black tree is a binary search tree, the implementation of this access procedure is identical to that of a binary search tree.

This is not the case for insert and delete. If we were going to apply the same method as the one used in a binary search tree, we would be able to preserve the fact that the modified tree is a binary search tree (the position of the node will be in the right place with respect to the ordering over the key values), but it would not necessarily guarantee the other four specific properties of a red-black tree.

We will see that these two operation require some recoloring of existing nodes in order to preserve Property 3 and some form of restructuring in order to preserve Property 4. In some cases these steps need to be recursively applied from the point of insertion of the new element up to the root of the tree. So re-establish the red-black trees properties would still require operations of the order O(log n).



The data structure for a Red-black tree can in principle be equal to that of Binary Search tree with the exception that the node has to include also a color field (as previously mentioned). However, the implementation of the operations of insert and delete become simpler to encode if the node object stores also a parent field.

Of course in the case of the root node the parent field will be null.

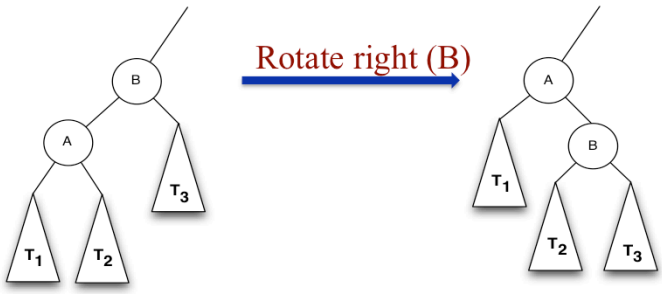
Since having a parent field is like having a double link between two node objects, when we set left and right child of a link we must not forget to set the parent of that child to be equal to the current node. An example implementation of the setLeft() access procedure is given below:

```
public void setLeft(RBTNode<K,V> newLeft){
   if (left != null && left.getParent() == this) {
       left.setParent(null);  // it is important to make the current left no loner reference its parent
   }
  this.left = newLeft;
  if (newLeft != null) {
      newLeft.setParent(this);
  }
}
```

It is also better in this case to make the node object responsible for handling its own rotation. In your PPT lab you are asked to implement these two rotate access procedures. A brief description of these two rotation algorithms is given in the next few slides.

We will see the algorithm for implementing the insert access procedure and leave out the delete access procedure.

Rotation revisited




```
rotateRight( ){
// Rotates the current object to the right and returns its new parent
        get leftChild of B;
        set T2 to be the left subtree of B;
        reparent the old leftChild of B;
        set B to be the right child of the old leftChild of B
}
```


Rotation operations for binary search trees are of two kinds: rotate left and rotate right. The double rotation operations that we have seen in the AVL trees are essentially two consecutive steps of these two basic rotations.

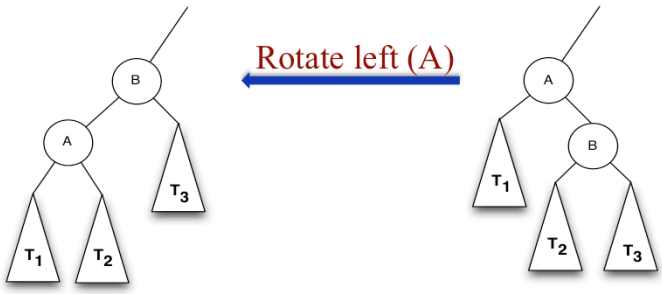
The slide shows on the left hand-side part of a binary search tree. We do know that the property of a binary search tree holds for this given tree. So we know that
for every key $k1 \in T_1$, $k2 \in T_2$ and $k3 \in T_3$ the following holds:

$$k1 < A < k2 < B < k3$$

(Note that in more general cases of duplicate keys in the tree, the < should be replaced with ≤.)

The rotation right of node B, “tilts downwards” the link between B and A (its left child), so that the parent of B becomes now the parent of A and A is the new parent of B. Of course the respective left and right sub-trees will need to be rearranged so that the above property of binary search tree ordering holds.

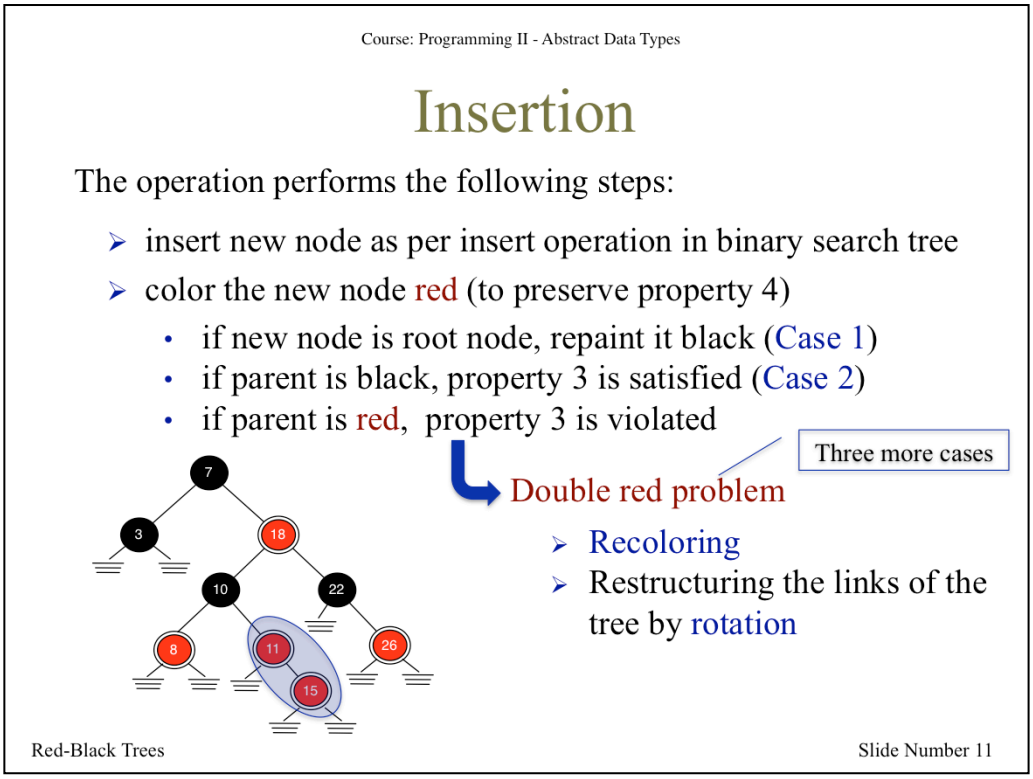
Rotation revisited




```
rotateLeft( ){
// Rotates the current object to the right and returns its new parent
        get rightChild of A;
        set T2 to be the right subtree of A;
        reparent the old rightChild of A;
        set A to be the left child of the old rightChild of B
}
```


This slide shows the case of left rotation starting from the part of the tree given in the right hand-side of the slide. The binary search tree property needs still to be preserved after the operation.

The rotation left of node A, “tilts downwards” the link between A and B (its right child), so that the parent of A becomes now the parent of B and B is the new parent of A. Of course the respective left and right sub-trees will need to be rearranged so that the property of binary search tree ordering holds.

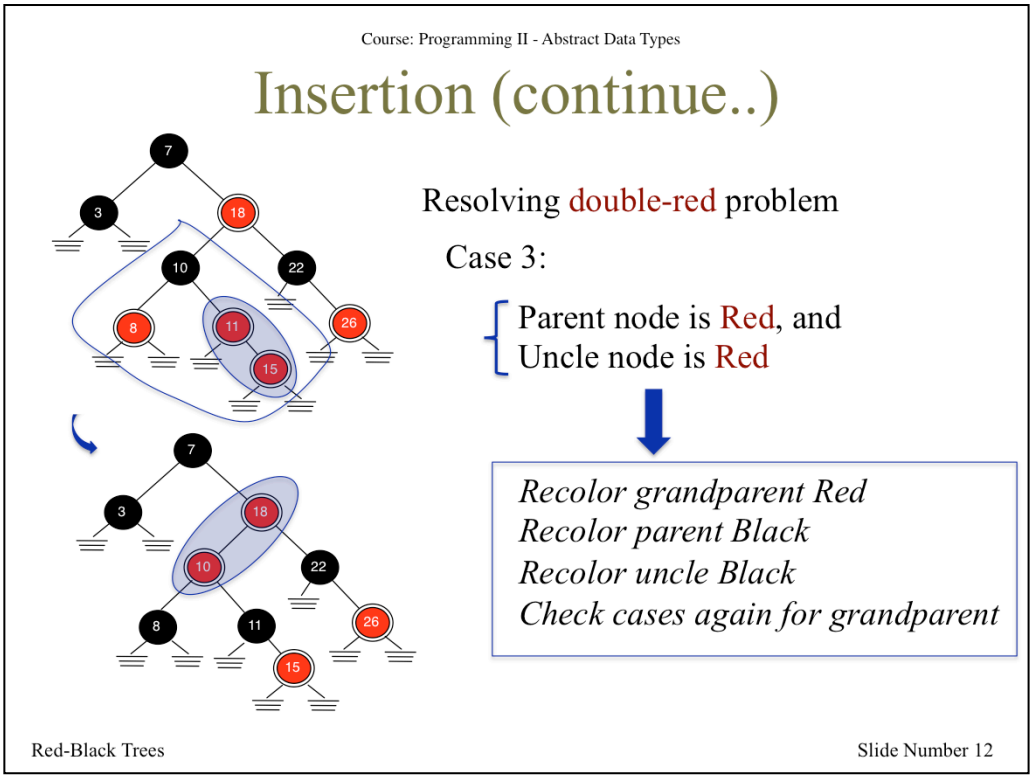


The insertion operation starts with the basic method we have seen for inserting a node in a binary search tree. Essentially the given tree is scanned from root to the leaves until either the given key is found (and so the element value is replaced, or the key is not found and in this case we have reached a null leaf. The operation will have then to create a new node and attach it to its parent. In the example shown in this slide, the new node to add is 15.

This standard operation of insertion of binary search tree does not color the newly created tree. So in particular we have to color the new node. Property 4 requires that the number of black nodes in any path from any given node to its descendent (null) leaves is the same. Given that our current tree is already red-black tree (i.e. satisfies this property for any internal node), if we had to color the newly created node black we would unbalance the black-height of its parent node (or possible any of its ancestor node). So the best thing to do is to color the new node red in order to preserve Property 4. However, we have to be careful because the tree has also other properties and in particular Property 3 which requires that parent of every red node must be black. We need therefore to check the parent of the newly created node.

If there is no parent (i.e. the new node is actually the root node of the tree) then we have to recolor the node itself black because we have to satisfies Property 2, which states that the root of a red-black tree must be black.

If there is a parent then we need to check the color of the parent node to see whether Property 3 is satisfied. If the parent is black we are find, and nothing more needs to be done, since all properties would in this case be satisfied. If the parent is red then the insertion of the new node as red node has caused a violation of Property 3, called “double-red” problem. In this case the insertion operation has to provide some fix in order to re-establish Property 3. The fix-up of the insertion includes 5 fix-up cases, (including the already mentioned two cases), which make use of one or more recoloring of the nodes in the tree and zero or more restructuring operations of the tree. The latter are done using the basic idea of rotation (single left, single right, double left-right, or double right-left rotation) that we have seen in the case of AVL trees. We define now each of these five fix-up cases in terms of when they occur and what

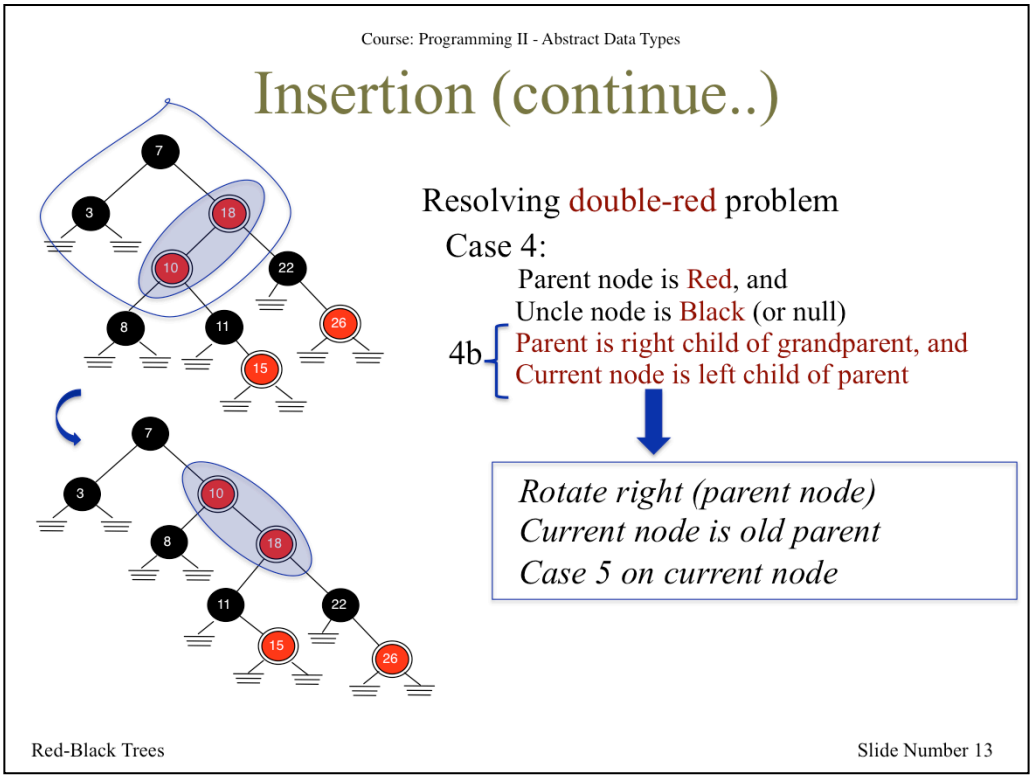


In slide 9 we have mentioned that the insertion of a new node and the setting of its color to red, although satisfies Property 4 of a red-black tree, may case violation of Property 3. This happens when the parent of the inserted node is itself red. In the example above we have the new node to be the number 15 and its parent (node 11) is also red. This configuration of the tree violates Property 3. The first thing that we could do is trying to do some recoloring so to re-establish Property 3. The question is what node to recolor. . If we recolored the parent 11 to be black this would reestablish Property 3 (as the parent of 11 is now black) but will definitely break Property 4. This is because it would include an additional black node that wasn't there. The node 10 would have black height equal to 2 on the path from 10 to a null leaf of 15, and black height equal to 1 on the path from 11 to its own (left) null leaf.

Recoloring has always to be a swapping process between parents and its own two children in order to maintain property 4 from that node, i.e. maintain the same number of black nodes on the paths. In this case the problem of node 11 is that its left child is null so it's already black and no swap would take place here. However, if you look at the grandparent of the new node we would notice that its left child (i.e. the uncle of the new node inserted) is red. Clearly the grandparent has to be black otherwise the tree would not have been a red-black tree. It is therefore possible in this case to swap the color between node 10 and its own children: make node 11 red and its children black. In this way we resolve the double-red problem between 15 and 11 whilst maintaining Property 4. We call this CASE 3.

Note that in this case we can safely assume that there is a grandparent because the parent is red and of the parent was the root by property 1 of red-black tree it would have been black.

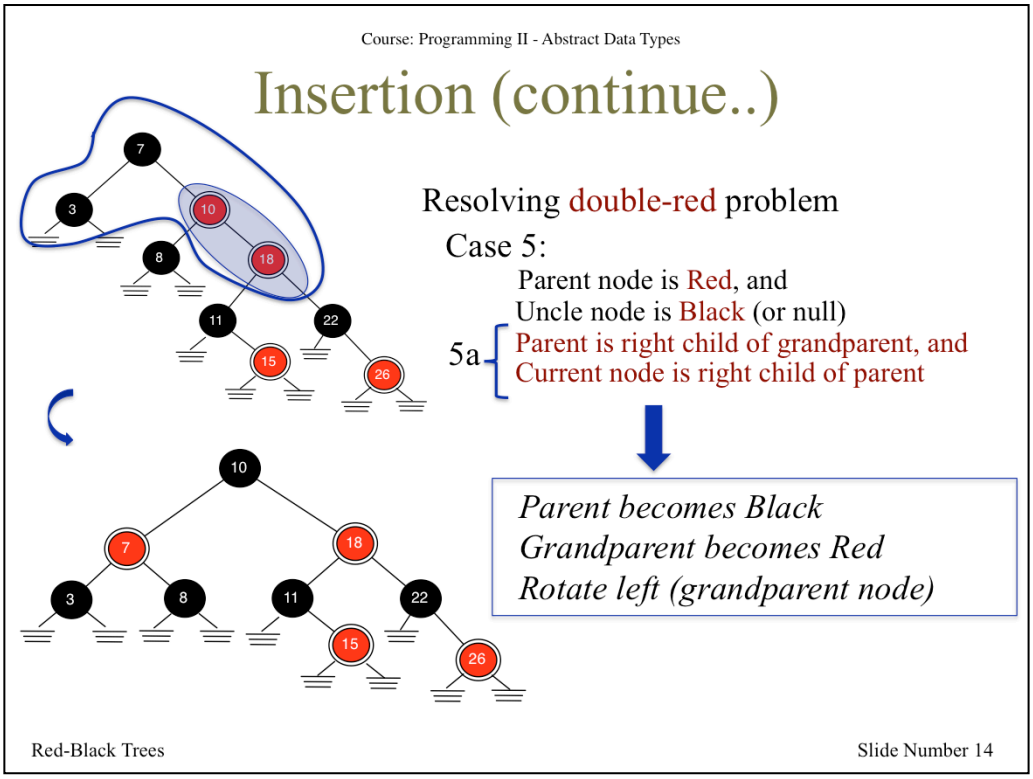
CASE 3: This applies when PARENT and UNCLE of the inserted node are both RED. In this case we make the Grandparent Red, and Parent and Uncle black, but we need then to recursively check from the grandparent now, since changing its color might introduce a double-red problem between itself (node 10 newly colored Red) and its own parent. So the newly colored node might now break Property 1 since it might itself be the root of the tree, in which case it has to be set to black or it might break property 3 (in case its respective parent is red). The newly colored red node 10 could be seen as a newly inserted node since it has just been painted red. So we need to check again from Case 1 onwards. (Case 3 resolves one double-red problem locally, but pushes it up the tree towards the root).



The resolution of a double-red problem includes more than one just case. In the previous slide Case 3 was for when the parent and the uncle of the node are both red. But look now at the current tree that we have generated. The double red problem has moved between node 10 and node 18. They are both red. In this case the grandparent is black but the children are not both red. So a color swap would violate Property 4 as it would make node 3 STAY black, node 7 become red and node 18 become black. But in this case the black-height of 7 would be 2 on its left branch and it would be equal to 3 on the path from 7 to the left child of node 8. So recoloring is not possible (i.e. Case 3 in not applicable). So, how can we resolve the double red problem? We can try to restructure first the tree (by rotations) and then apply some recoloring. In principle we can apply one of four possible restructuring. As we have seen in the AVL trees the type of rotation depends on where the new introduced node has been inserted (left subtree of left child, right subtree of left child, right subtree of right child, left subtree of right child). In our case, we can take the newly modified node (i.ee node 10) as an "inserted" node. This is in the left subtree of the right child of the the grandparent node 7. We call this specific situation of Case 4 as **CASE 4b**. A double rotation (i.e. a right and then left rotation) is therefore needed. The right rotation is applied to the parent (node 18) giving the tree drawn at the bottom of this slide. The double red problem is still not solved now the parent node (i.e. node 18) becomes the current node. The second rotation (i.e. rotation left on the grandparent of the current node) is dealt by what we call CASE 5 described in the next slide.

Note that at the end of the first rotation right the current node is 18, its parent is red and its uncle is black. But the node 18 is now in the right subtree of the right child of the node to rotate. So CASE 5 will perform a rotation left.

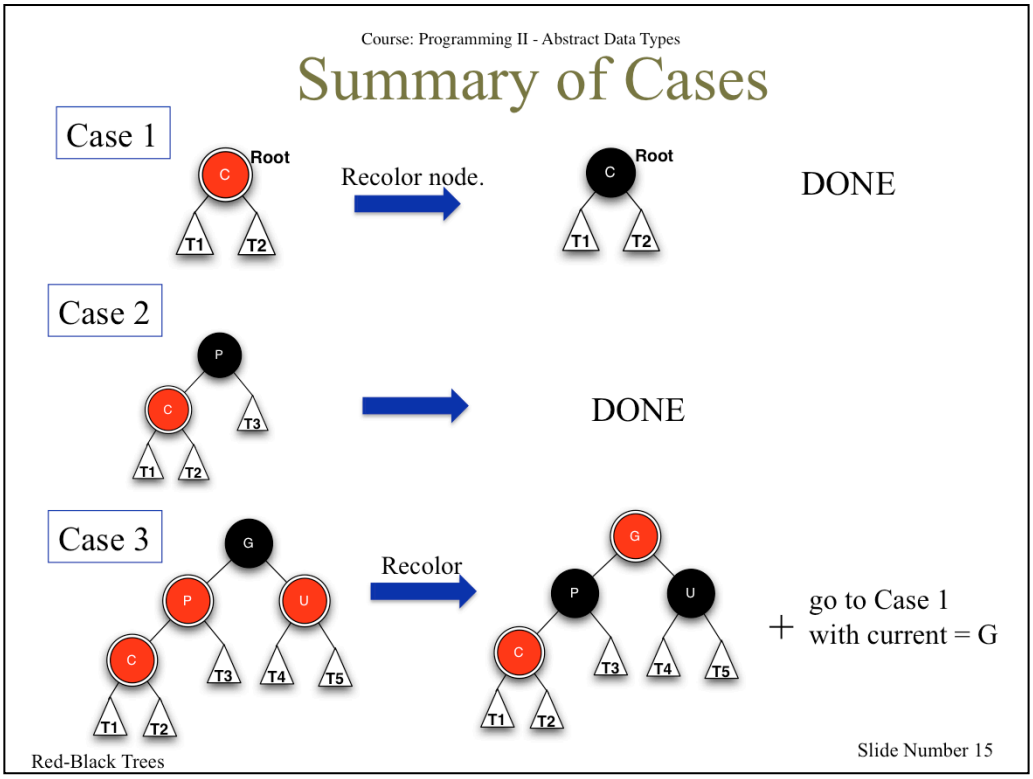
CASE 4 has a symmetric situation, which is when the parent is the left child of the grandparent and the current node is the right child of the parent. We refer to this situation as **CASE 4a** and what we need to do is apply first a rotation left on the parent node, then take the parent node as the current node and perform CASE 5 (which will do a rotation right).



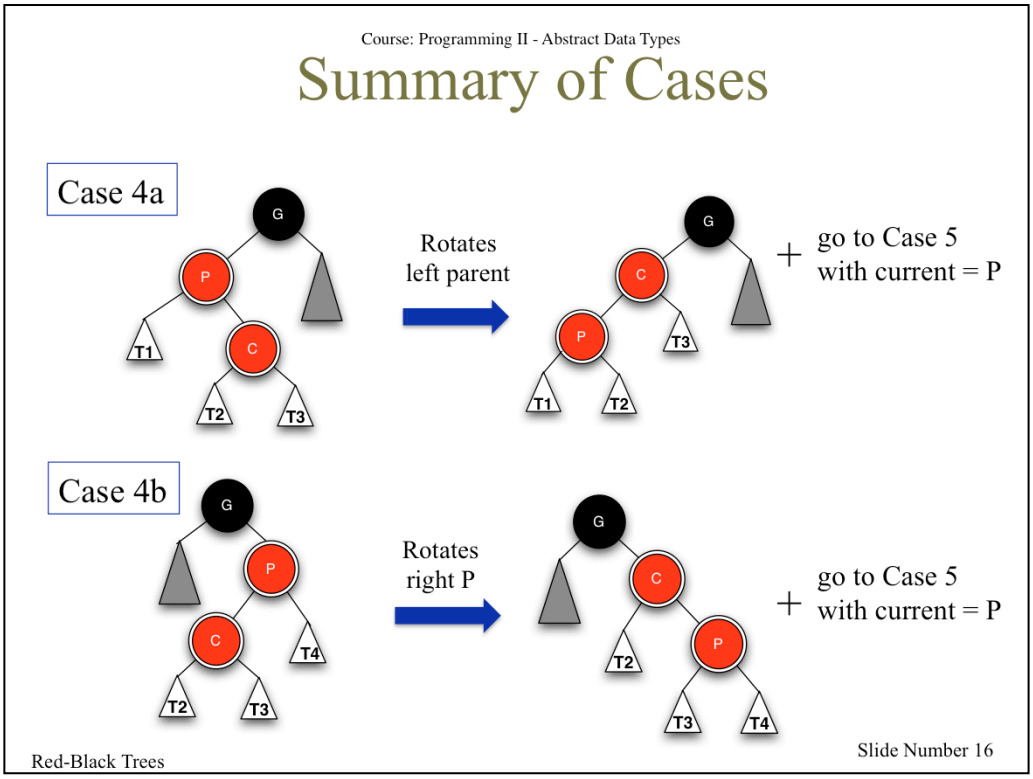
To complete the last step of our fix-up we have one more case to consider. This is what we referred to as CASE 5 in the previous slide. It basically applies when the current node is Red (i.e. node 18 in this top tree), its parent is red too (i.e. node 10) and its uncle is black (i.e. node 3), and the two consecutive red nodes are on a straight line. This case requires one single rotation step. This could be either a left rotation or a right rotation depending of where the current node is. In the situation described above the parent (i.e. node 10) and the current node (i.e. node 18) are both right children. The rotation needed here is a left rotation on the grandparent node. By if we were doing so we would have the grandparent node to be now node 10 (which is red) and its left child be node 7 and right child node 18 which is also red.

So we need first to swap the colors of the parent node and of the grandparent nodes and then apply the left rotation. These operations would give us the tree drawn at the bottom left of this slide. We refer to these steps as CASE 5a.

Case 5 has also the symmetric case when the parent is a left child and the current node is the left child of its parent. We refer to this as CASE 5b, which would perform the same color change but then perform a right rotation of the grandparent node.

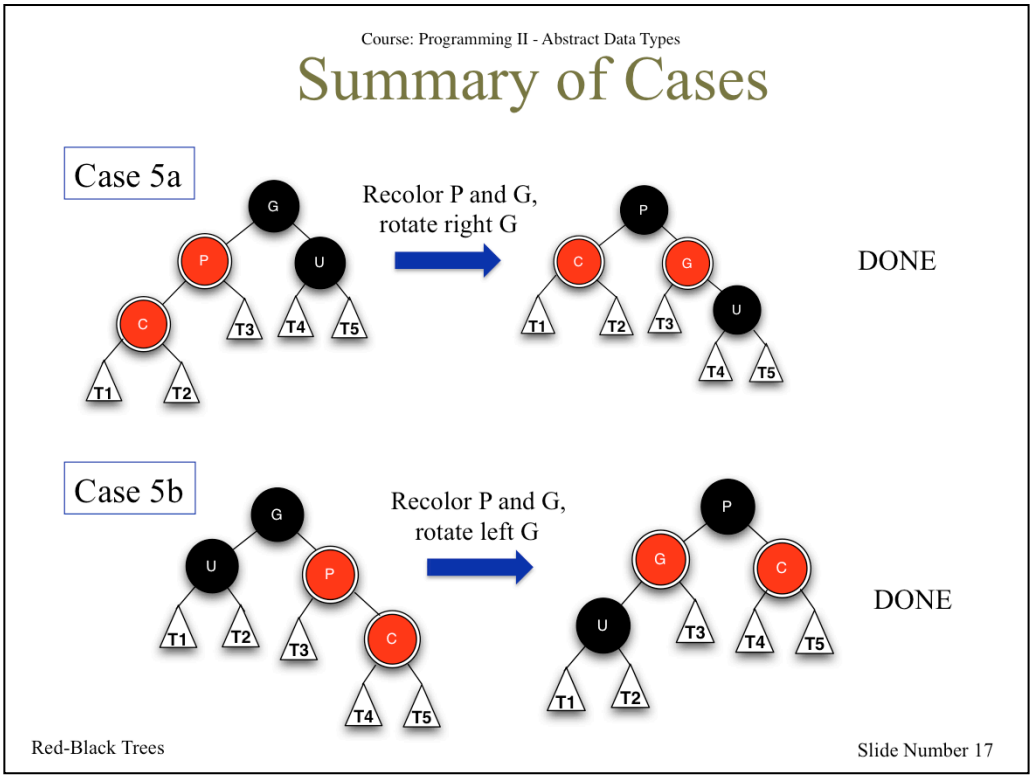


In summary, Cases 1, 2 and 5 are terminal cases. Case 4 calls Case 5 so could be considered as a terminal case, whereas Case 3 is the only recursive case as it requires recursive check traversing the tree upwards until any of the terminal cases are executed.



Note that in both situations of CASE 4, the new path from P to subtree T2 no longer pass through node C as it was before applying the rotation. Similarly, the paths from C to subtrees T1 and T2 pass now through node P whereas before they were not. However, since both C and P are red nodes, this rotation does not violate Property 4. The black height value of C and P stays unchanged. In both case, after one rotation Property 3 is still violated. This is why Case 5 is called.

In this slide and in the next the grey triangle means a subtree that if not null has as a root which is black.



In this case, Property 4 also is satisfied by the two rotations. This is because any path that now goes from P to its descendent leaves will either pass through G as they did before or through C. Both these two nodes are red so the black height of the nodes is not affected, by the recoloring and rotation.

Implementing RedBlackTree

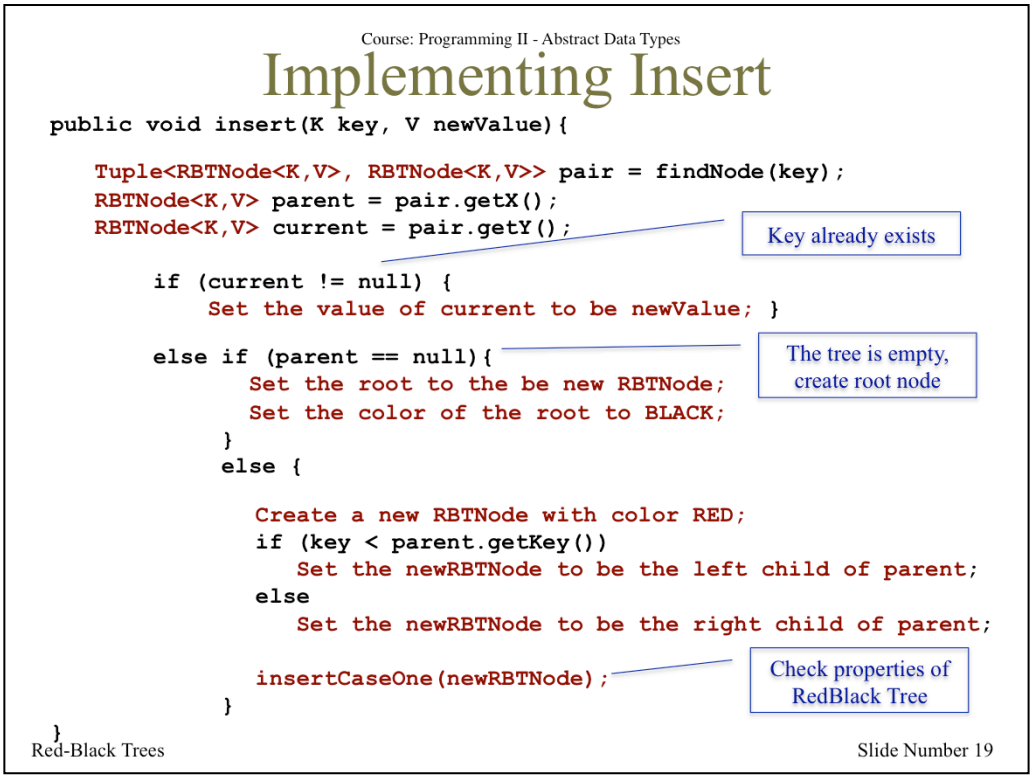
```
public class RedBlackTree<K extends Comparable<K>, V>
                                                    implements RBT<K,V>{
   private RBTNode<K,V> root;

   public RedBlackTree() { this.root = null;}

   public V retrieve(K key){
        RBTNode<K,V> node = retrieveElem(root, key).getValue();
        if (node == null){throw new NoSuchElementException();}
        return node.getValue();
   }

   public void insert(K key, V newValue){
        << next slide>>

   }
}
```



This slide shows a pseudo code for the insert operation. The approach is slightly different from the recursive call we have seen with the insert for a Binary Search Tree. the underlying idea is similar in that the findNode private auxiliary method traverses the tree in the same fashion as the search in a binary search tree would do in order to locate the given key if it exists. In this case we have assumed that the findNode returns a pair of RBTNodes = <parent, current>.

Clearly if the key is found then the value of the returned current Node is set to the newValue.

If the tree is empty, then the parent node in the returned pair would be null. In this case we have just to create a new node and set its color to Black to satisfy Property 1.

On the second else condition, the returned pair is a node Null and a not null parent. This corresponds to the situation where the findNode has traversed the tre and located the place where the new node has to be inserted. The creation of the new Node happens here (setting it to be RED) and it gets linked to the parent node according to the key values. Now all the checks for cases we have seen in the previous slides need to be done to guarantee that the insertion satisfies the RBT Properties.

The insertCaseOne procedure is responsible for setting the root node back to black (this is needed because during the recoloring and restructuring of the tree, we might arrive back to a red root node. Clearly the first time that we call this procedure, the node is not the root because we have just inserted it at the bottom of the tree.

InsertCaseOne has to call InsertCaseTwo, since it is needed to check whether the new node has a red parent. If the parent is black than InsetCaseTwo is finished and the insert procedures finishes without necessarily traversing the tree all the way back to the root. If the parent is red then we need to see which of the remaining cases we have. So InsertCaeTwo calls InsertCaseThree. This auxiliary method looks at the uncle of the newly created Node. If the uncle is red than InsertCaseThree does the recoloring and calls again InsertCaseOne but this type from the grandparent of the currentNode.

If the uncle is black than InsertCaseFour is called on the current node. This does a first rotation and then InsertCaseFive is called.

Analysis

AVL trees and Red-Black trees are data structure that guarantee insertion, deletion and search operations to be of the order O(log n).

How do AVL trees and Red-Black trees compare?

- The height of an AVL tree is also logarithmic in n.
- An AVL tree can be colored to become a Red-Black tree,
- A Red-Black tree cannot be seen as an AVL tree:
  - Red-black trees can have larger heights than AVL trees
  - AVL trees are more rigidly balanced than Red-Black trees
- Red-Black tree have slightly faster insertions and deletion than AVL trees (TreeMap in Java uses Red-Black trees).