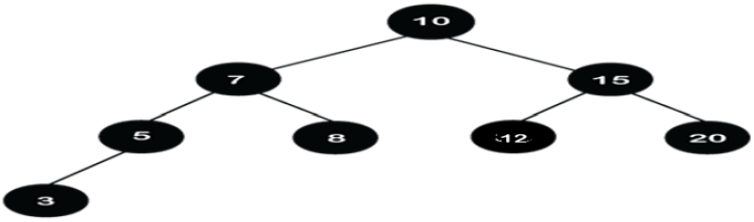
4.9 Red Black Trees Construction

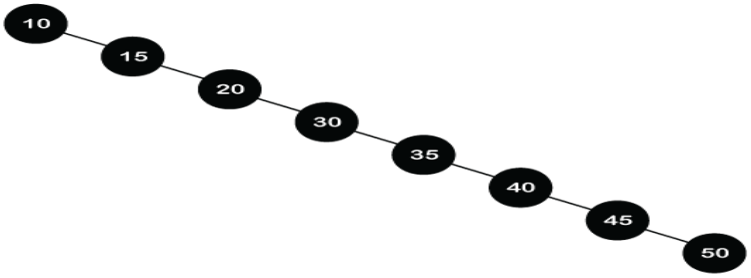
The Red-Black tree is a binary search tree. In a binary search tree, the values of the nodes in the left sub tree should be less than the value of the root node, and the values of the nodes in the right subtree should be greater than the value of the root node.

Each node in the Red-black tree contains an extra bit that represents a color to ensure that the tree is balanced during any operations performed on the tree like insertion, deletion, etc.

Let's understand the different scenarios of a binary search tree.



In the above tree, if we want to search the 80. We will first compare 80 with the root node. 80 is greater than the root node, i.e., 10, so searching will be performed on the right subtree. Again, 80 is compared with 15; 80 is greater than 15, so we move to the right of the 15, i.e., 20. Now, we reach the leaf node 20, and 20 is not equal to 80. Therefore, it will show that the element is not found in the tree. After each operation, the search is divided into half.



The above tree shows the right-skewed BST. If we want to search the 80 in the tree, we will compare 80 with all the nodes until we find the element or reach the leaf node.

In the above BST, the first one is the balanced BST, whereas the second one is the unbalanced BST. We conclude from the above two binary search trees that a balanced tree takes less time than an unbalanced tree for performing any operation on the tree.

Therefore, we need a balanced tree, and the Red-Black tree is a self-balanced binary search tree. Now, the question arises that ***why do we require a Red-Black tree*** if AVL is also a height-balanced tree. The Red-Black tree is used because the AVL tree requires many rotations when the tree is large, whereas the Red-Black tree requires a maximum of two rotations to balance the tree. The main difference between

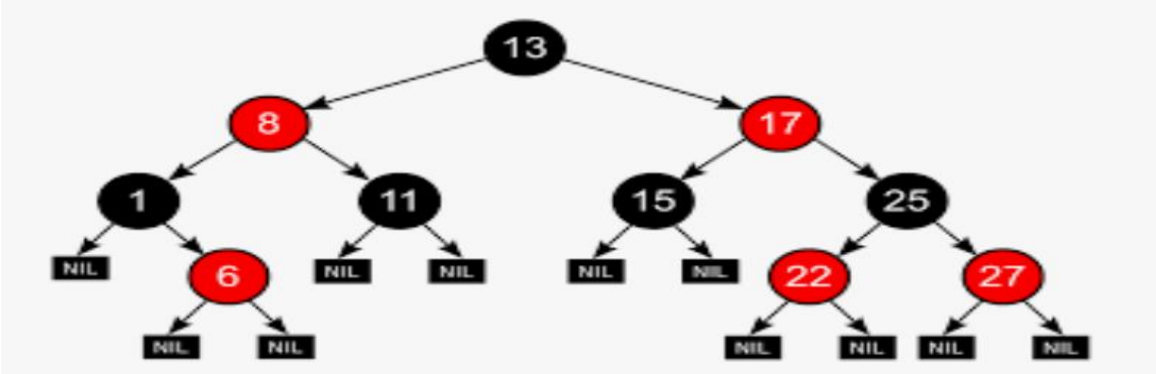
the AVL tree and the Red-Black tree is that the AVL tree is strictly balanced, while the Red-Black tree is not completely height-balanced. So, the AVL tree is more balanced than the Red-Black tree.

Insertion is easier in the AVL tree as the AVL tree is strictly balanced, whereas deletion and searching are easier in the Red-Black tree as the Red-Black tree requires fewer rotations.

As the name suggests that the node is either colored in **Red** or **Black** color. Sometimes no rotation is required, and only recoloring is needed to balance the tree.

Properties of Red-Black tree

- It is a self-balancing Binary Search tree. Here, self-balancing means that it balances the tree itself by either doing the rotations or recoloring the nodes.
- This tree data structure is named as a Red-Black tree as each node is either Red or Black in color. Every node stores one extra information known as a bit that represents the color of the node. For example, 0 bit denotes the black color while 1 bit denotes the red color of the node. Other information stored by the node is similar to the binary tree, i.e., data part, left pointer and right pointer.
- ***In the Red-Black tree, the root node is always black in color.***
- In a binary tree, we consider those nodes as the leaf which have no child. In contrast, in the Red-Black tree, the nodes that have no child are considered the internal nodes and these nodes are connected to the NIL nodes that are always black in color. The NIL nodes are the leaf nodes in the Red-Black tree.
- ***If the node is Red, then its children should be in Black color. In other words, we can say that there should be no red-red parent-child relationship.***
- ***Every path from a node to any of its descendant's NIL node should have same number of black nodes.***



Is every AVL tree can be a Red-Black tree?
Yes, every AVL tree can be a Red-Black tree if we color each node either by Red or Black color. But every Red-Black tree is not an AVL because the AVL tree is strictly height-balanced while the Red-Black tree is not completely height-balanced.

Insertion in Red Black tree
The following are some rules used to create the Red-Black tree:

1. If the tree is empty, then we create a new node as a root node with the color black.
2. If the tree is not empty, then we create a new node as a leaf node with a color red.
3. If the parent of a new node is black, then exit.
4. If the parent of a new node is Red, then we have to check the color of the parent's sibling of a new node.

4a) If the color is Black, then we perform rotations and recoloring.

4b) If the color is Red then we recolor the node. We will also check whether the parents' parent of a new node is the root node or not; if it is not a root node, we will recolor and recheck the node.

Let's understand the insertion in the Red-Black tree.

10, 18, 7, 15, 16, 30, 25

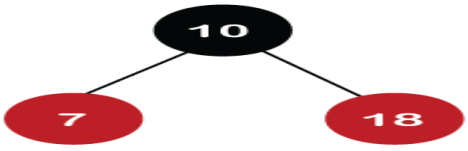
Step 1: Initially, the tree is empty, so we create a new node having value 10. This is the first node of the tree, so it would be the root node of the tree. As we already discussed, that root node must be black in color, which is shown below:

Step 2: The next node is 18. As 18 is greater than 10 so it will come at the right of 10 as shown below.

We know the second rule of the Red Black tree that if the tree is not empty then the newly created node will have the **Red** color. Therefore, node 18 has a Red color, as shown in the below figure:

Now we verify the third rule of the Red-Black tree, i.e., the parent of the new node is black or not. In the above figure, the parent of the node is black in color; therefore, it is a Red-Black tree.

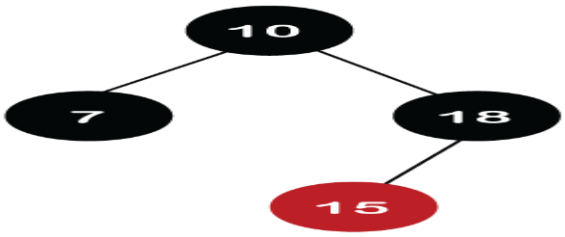
Step 3: Now, we create the new node having value 7 with Red color. As 7 is less than 10, so it will come at the left of 10 as shown below.



Now we verify the third rule of the Red-Black tree, i.e., the parent of the new node is black or not. As we can observe, the parent of the node 7 is black in color, and it obeys the Red-Black tree's properties.

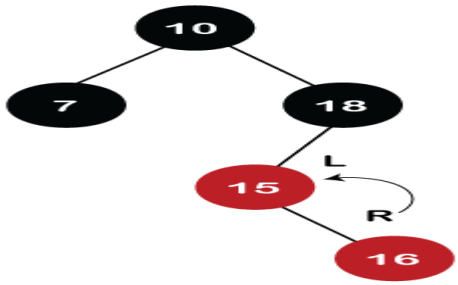
Step 4: The next element is 15, and 15 is greater than 10, but less than 18, so the new node will be created at the left of node 18. The node 15 would be Red in color as the tree is not empty.

The above tree violates the property of the Red-Black tree as it has Red-red parent-child relationship. Now we have to apply some rule to make a Red-Black tree. The rule 4 says that ***if the new node's parent is Red, then we have to check the color of the parent's sibling of a new node.*** The new node is node 15; the parent of the new node is node 18 and the sibling of the parent node is node 7. As the color of the parent's sibling is Red in color, so we apply the rule 4b. The rule 4b says that we have to recolor both the parent and parent's sibling node. So, both the nodes, i.e., 7 and 18, would be recolored as shown in the below figure.



We also have to check whether the parent's parent of the new node is the root node or not. As we can observe in the above figure, the parent's parent of a new node is the root node, so we do not need to recolor it.

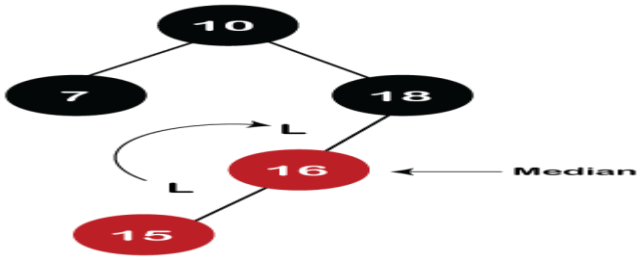
Step 5: The next element is 16. As 16 is greater than 10 but less than 18 and greater than 15, so node 16 will come at the right of node 15. The tree is not empty; node 16 would be Red in color, as shown in the below figure:



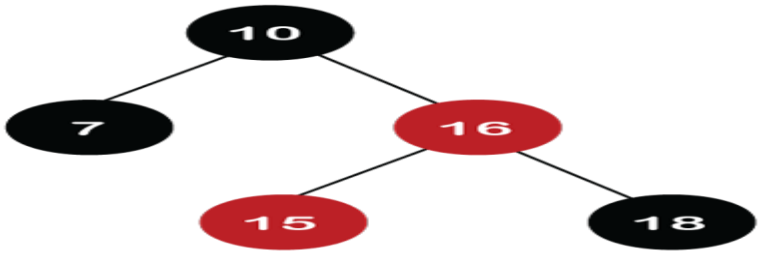
In the above figure, we can observe that it violates the property of the parent-child relationship as it has a red-red parent-child relationship. We have to apply some rules to make a Red-Black tree. Since the new node's parent is Red color, and the parent of the new node has no sibling, so rule **4a** will be applied. The rule **4a** says that some rotations and recoloring would be performed on the tree.

Since node 16 is right of node 15 and the parent of node 15 is node 18. Node 15 is the left of node 18. Here we have **an LR** relationship, so we require to perform two rotations. First, we will perform right, and then we will perform the left rotation. The right rotation would be performed on nodes 15 and 16,

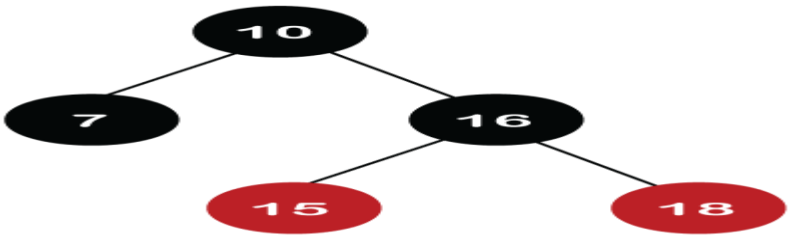
where node 16 will move upward, and node 15 will move downward. Once the right rotation is performed, the tree looks like as shown in the below figure:



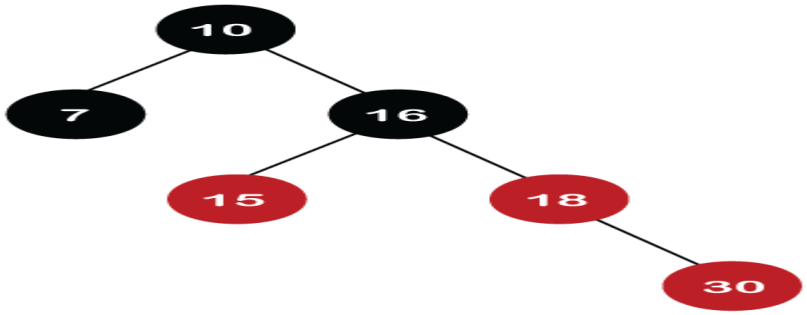
In the above figure, we can observe that there is **an LL** relationship. The above tree has a Red-red conflict, so we perform the left rotation. When we perform the left rotation, the median element would be the root node. Once the left rotation is performed, node 16 would become the root node, and nodes 15 and 18 would be the left child and right child, respectively, as shown in the below figure.



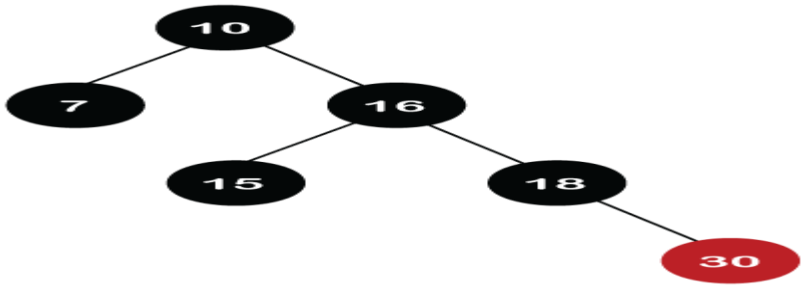
After rotation, node 16 and node 18 would be recolored; the color of node 16 is red, so it will change to black, and the color of node 18 is black, so it will change to a red color as shown in the below figure:



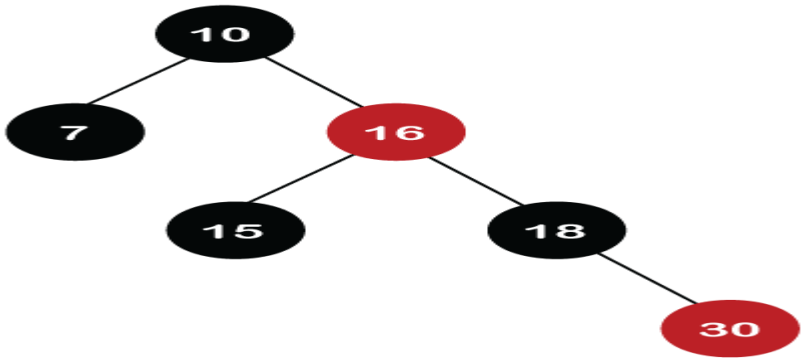
Step 6: The next element is 30. Node 30 is inserted at the right of node 18. As the tree is not empty, so the color of node 30 would be red.



The color of the parent and parent's sibling of a new node is Red, so rule 4b is applied. In rule 4b, we have to do only recoloring, i.e., no rotations are required. The color of both the parent (node 18) and parent's sibling (node 15) would become black, as shown in the below image.



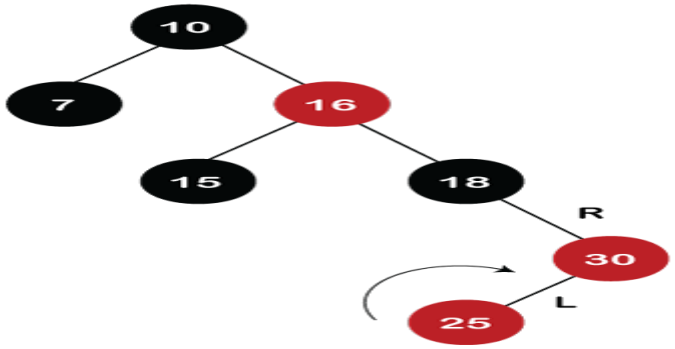
We also have to check the parent's parent of the new node, whether it is a root node or not. The parent's parent of the new node, i.e., node 30 is node 16 and node 16 is not a root node, so we will recolor the node 16 and changes to the Red color. The parent of node 16 is node 10, and it is not in Red color, so there is no Red-red conflict.



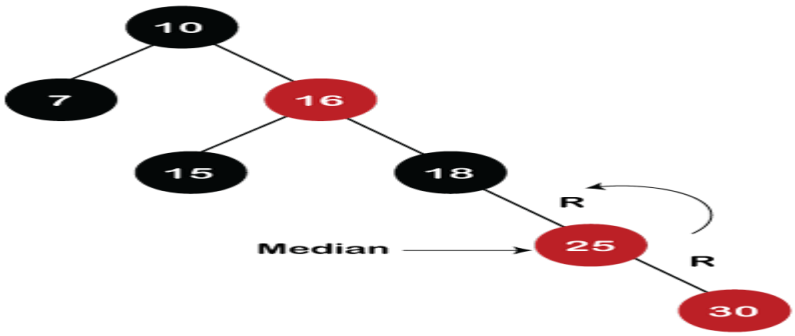
Step 7: The next element is 25, which we have to insert in a tree. Since 25 is greater than 10, 16, 18 but less than 30; so, it will come at the left of node 30. As the tree is not empty, node 25 would be in Red color. Here Red-red conflict occurs as the parent of the newly created is Red color.

Since there is no parent's sibling, so rule 4a is applied in which rotation, as well as recoloring, are performed. First, we will perform rotations. As the newly created node is at the left of its parent and the

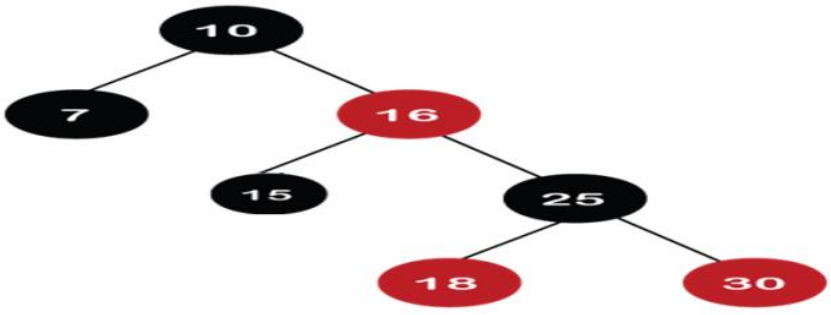
parent node is at the right of its parent, so the RL relationship is formed. Firstly, the left rotation is performed in which node 25 goes upwards, whereas node 30 goes downwards, as shown in the below figure.



After the first rotation, there is an RR relationship, so right rotation is performed. After right rotation, the median element, i.e., 25 would be the root node; node 30 would be at the right of 25 and node 18 would be at the left of node 25.



Now recoloring would be performed on nodes 25 and 18; node 25 becomes black in color, and node 18 becomes red in color.

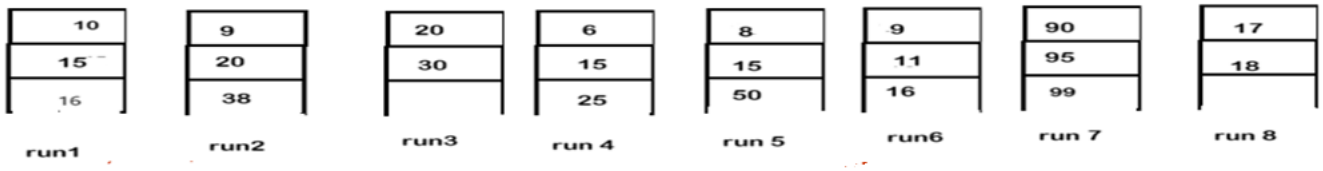


The above tree is a Red-Black tree as it follows all the Red-Black tree properties.

4.10 Selection Trees (Tournament Trees)

Suppose we have **k** ordered sequences, called runs that are to be merged into a single ordered sequence. Each run consists of some records and is in non-decreasing order of a designated field called the key. Let **n** be the number of records in all **k** runs together.

Ex – **k=8**



The most direct way to merge k runs is to make k-1 comparisons to determine the next record to output. For k>2, we can achieve a reduction in the number of comparisons needed to find the next smallest element by using the **selection tree data structure**. There are two kinds of selection trees: winner trees and loser trees.

(1) winner tree

(2) loser tree

1. Winner Trees

A winner tree is a complete binary tree in which each node represents the smaller of its two children. Thus, the root node represents the smallest node in the tree.

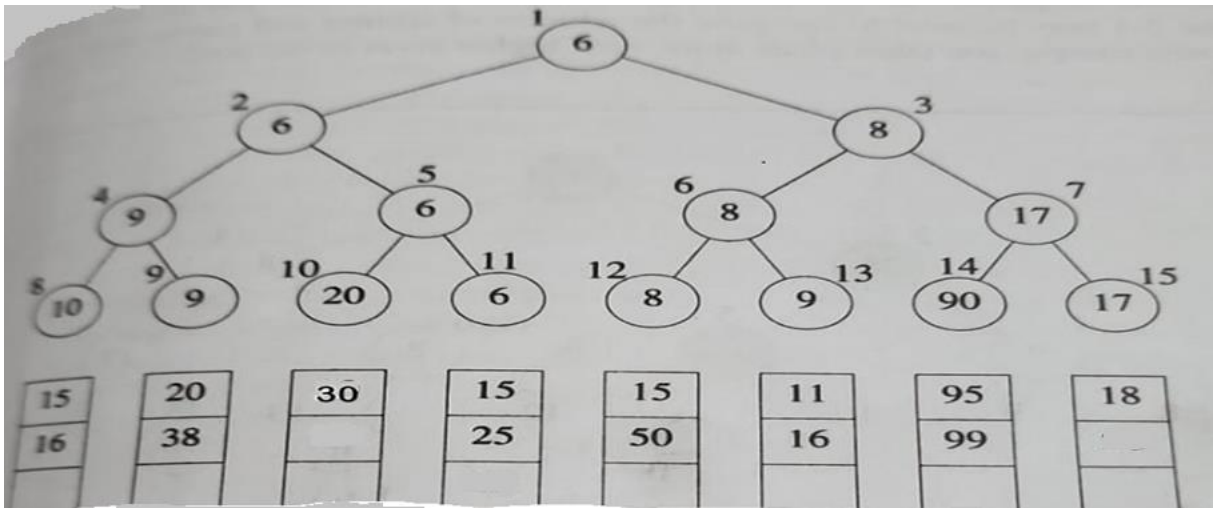
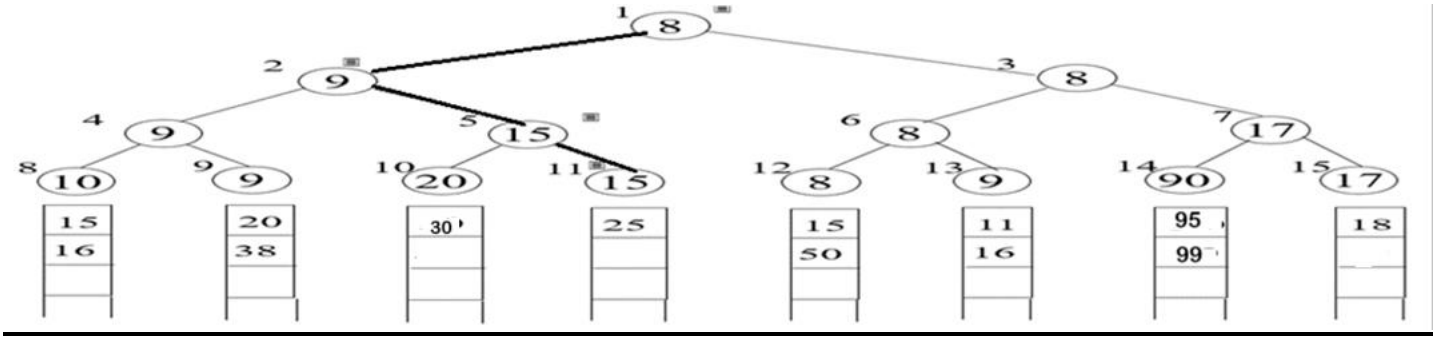


Figure 1

The construction of this winner tree may be compared to the playing of a tournament in which the winner is the record with smaller key. Then each nonleaf node in the tree represents the winner of the tournament, and the root node represents the overall winner, or the smallest key. Each leaf node represents the first record in the corresponding run. The root has the smallest key and so will be outputted. Now, the next record from run 4 enters the winner tree. It has a key value of 15. To restructure the tree, the tournament has to be replayed only along the path from node 11 to the root. The tournament is played between the sibling nodes and the

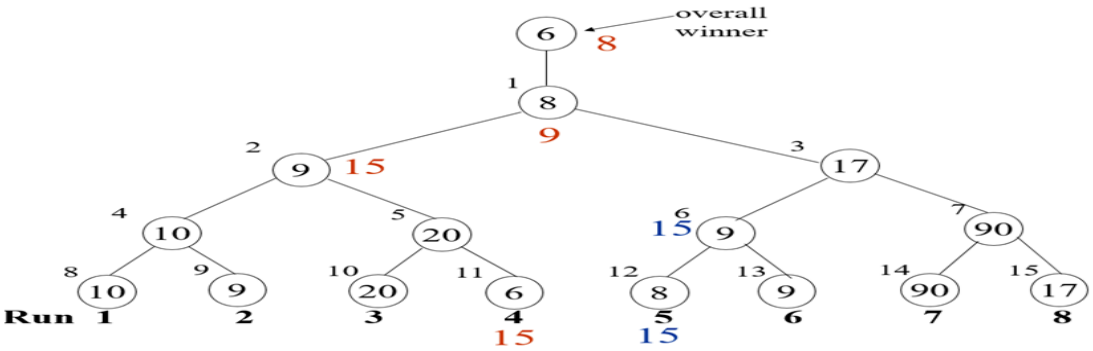
result put in the parent node.



Selection tree after one record has been output and the tree restructured (nodes that were changed are marked and the path is darkened)

2. Loser Trees

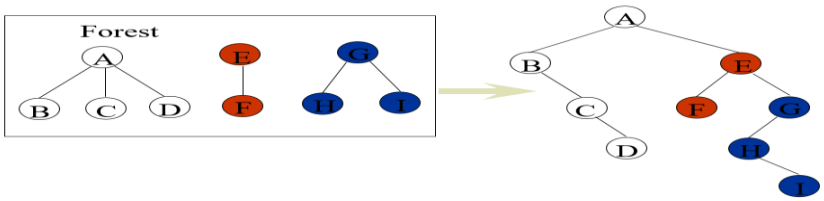
Tournaments are played between sibling nodes along the path from node 11 to root. *Since these sibling nodes represent the losers of tournaments played earlier, we can simplify the restructuring process by placing the record that loses the tournament rather than to the winner of the tournament. A selection tree in which nonleaf node retains the loser is called a loser tree.* The leaf nodes represent the first record in each run. An additional node, node 0, has been added to represent the overall winner of the tournament. Following the output of the overall winner, the tree is restructured by playing tournaments along the path from node 11 to node 1. The records with which these tournaments are to be played are readily available from the parent nodes. As a result, sibling nodes along the path from node 11 to 1 are not accessed.



Tree of losers corresponding to Figure 1

4.11 Forest

A forest is a set of n >= 0 disjoint trees



Transform a forest into a binary tree

- If $T_1, T_2, \ldots, T_n$ is a forest of trees, then the binary tree corresponding to this forest denoted by $B(T_1, T_2, \ldots, T_n)$
- algorithm
  (1) is empty, if n = 0
  (2) has root equal to root(T_1);
  has left subtree equal to $B(T_{11},T_{12},\ldots,T_{1m})$
  has right subtree equal to $B(T_2,T_3,\ldots,T_n)$

Forest Traversals

- Preorder
  - If F is empty, then return
  - Visit the root of the first tree of F
  - Taverse the subtrees of the first tree in forest preorder
  - Traverse the remaining trees of F in forest preorder

  (A, B, C, D, E, F, G, H, I) – Preorder traversal of forest is same as preorder traversal of its corresponding binary tree.
- Inorder
  - If F is empty, then return
  - Traverse the subtrees of the first tree in forest inorder
  - Visit the root of the first tree
  - Traverse the remaining trees of F is forest inorder

  (B, C, D, A, F, E, H, I, G) – Inorder traversal of forest is same as inorder traversal of its corresponding binary tree.
- Postorder
  - If F is empty, then return
  - Traverse the subtrees of the first tree in forest postorder
  - Traverse the remaining trees of F is forest postorder
  - Visit the root of the first tree

  (B, C, D, F, E, H, I, G, A) – Postorder traversal of forest is not same as postorder traversal of its corresponding binary tree.
- Levelorder

  In a level-order traversal of a forest, nodes are visited by level, beginning with the roots of each tree in the forest. Within each level, nodes are visited from left to right. Levelorder traversal of forest is not same as levelorder traversal of its corresponding binary tree.

References:

http://btechsmartclass.com/DS/U3_T3.html

http://www.iare.ac.in/sites/default/files/DS.pdf