Red-Black Trees

Red-Black Trees

- **BST** work well if the data is inserted into the tree in **random order**. They work much slower if the data is inserted in already **sorted order**. When the values to be inserted are already ordered, a binary tree becomes unbalanced.
- **Red-Black tree** is a **BST** with some added **features**. In the red-black tree **balance** is achieved during **insertion** and **deletion**. As an item is being inserted, the **insertion routine** checks that certain characteristics of the tree are not **violated**. If they are, it takes corrective action. By **maintaining these characteristics**, the tree is kept **balanced**.

Red-Black Trees

- **Red-Black tree characteristics**
- There are **two characteristics**:

1. The **nodes** are **colored**.
2. During **insertion and deletion**, **rules** are followed that preserve **various arrangements** of these **colors**.

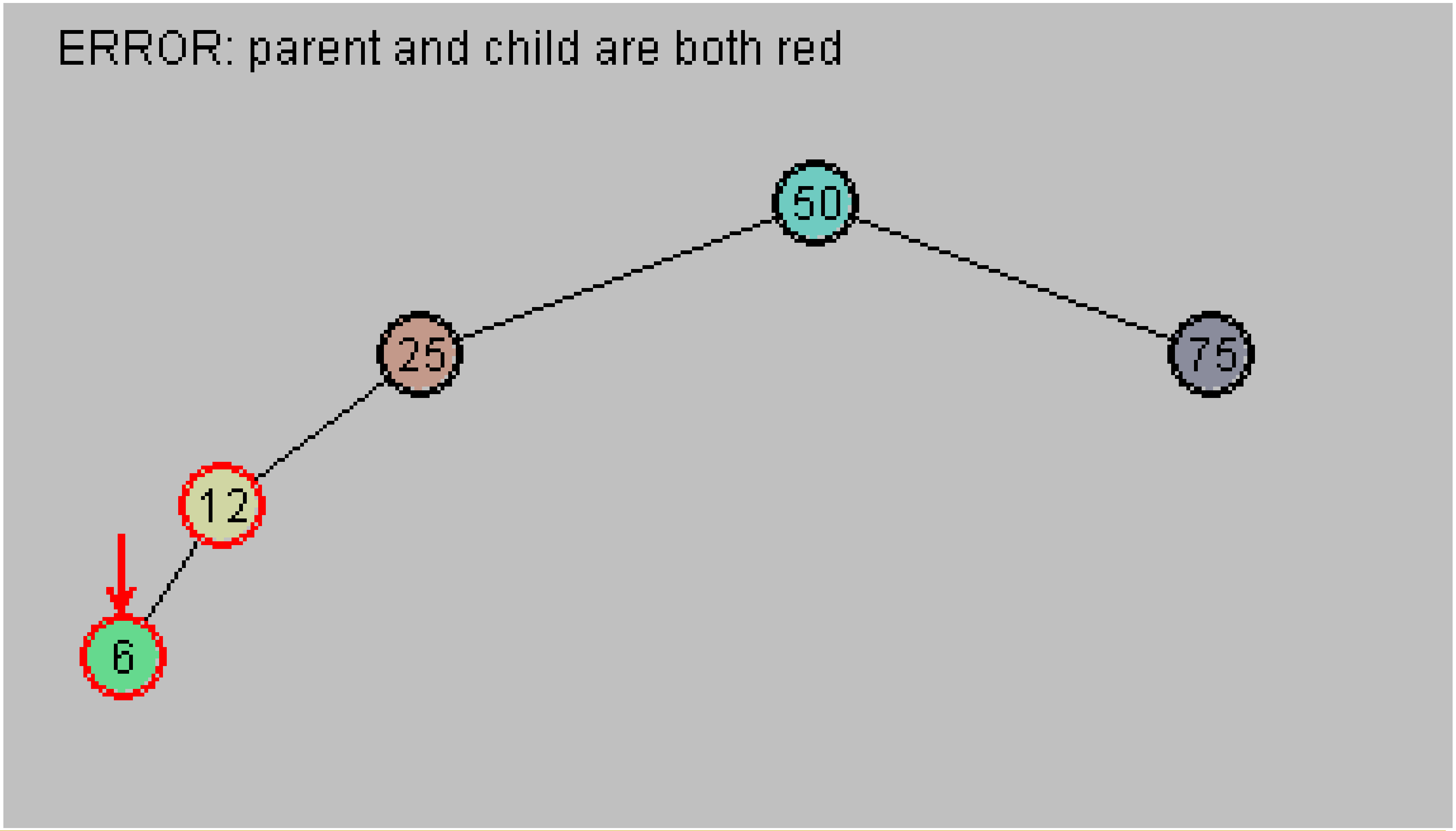
Red-Black Trees

- **Red-Black Rules**
- During **insertion and deletion** from Red-Black tree rules which are called **Red-Black rules** should be followed. If they are followed the tree will be balanced. These rules are:
  1. **Every node is either red or black**.
  2. The **root** is always **black**.
  3. If a node is **red**, its **children** must be **black**.
  4. **Every path** from the root to a leaf, or to a null child, must contain the same number of **black nodes.**
- **In fact these rules keep the tree balanced.** If **one path** is **longer** than another, it must have **more black** nodes, **violating rule 4**, or it must have **two adjacent red nodes**, **violating rule 3**.

Red-Black Trees

- The **difficulty** is that, **operations can change** the tree and possibly destroy the **coloring** properties. This make **insertion and deletion** difficult, especially removal.

Red-Black Trees



Red-Black Trees

```
class rbtree
{
    String name;
    String address;
    int idno
    boolean isRed;
    rbtree left;
    rbtree right;
    }
```

Red-Black Trees

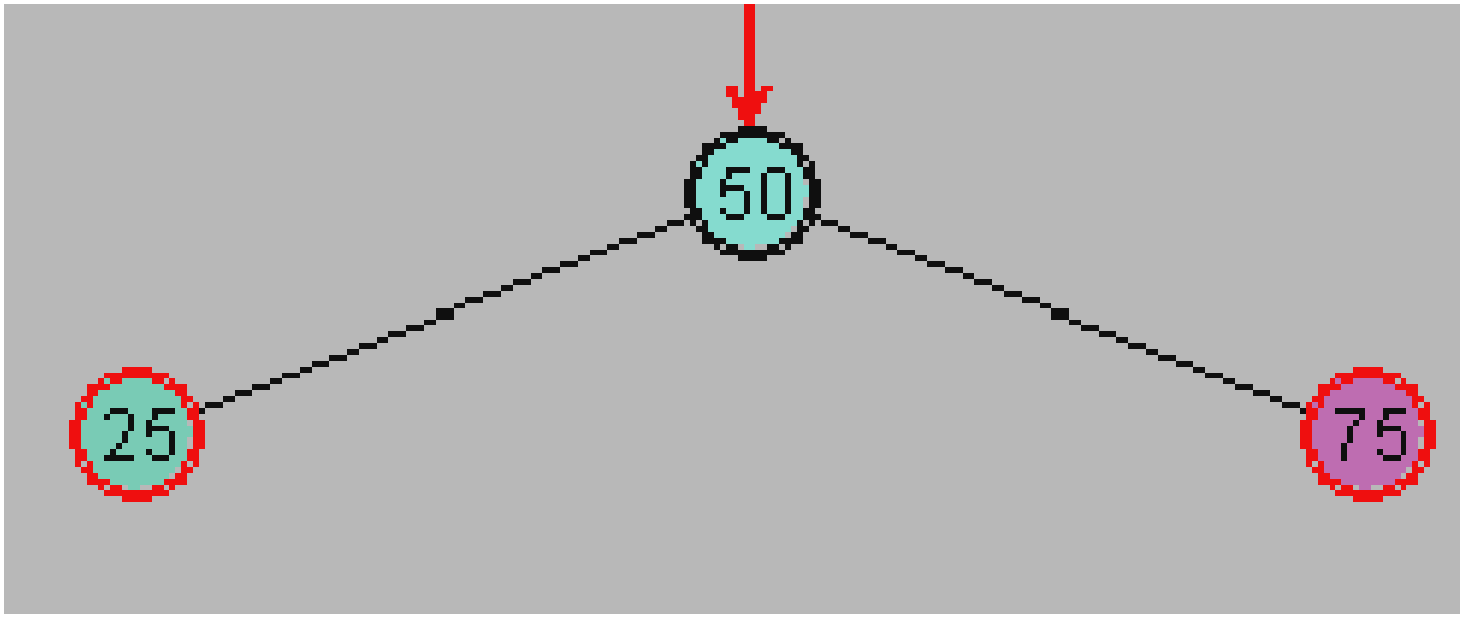
- **Duplicate Keys**
- What happens if there is more than one data item with the **same key**? This presents slight **problem** in **red-black** trees. It is important that nodes with the same key are **distributed** on both sides of other nodes with the same key. That is, if key arrive in the order **50,50,50**, you want the second 50 to go to the right of the first one, and the third 50 to go to the left of the first one. **Otherwise** the tree become **unbalanced**.
- **Distributing** nodes with equal keys could be handled by some kind of **randomizing process** in the insertion algorithm. However, the **search process** then become more **complicated** if all items with the same key must be found.

Red-Black Trees

- **Fixing Rule Violations**
- Suppose you see that the color rules are **violated**. How can you fix things so your tree is in compliance? There is **two actions** you can take:
  - You can **change the colors of nodes**.
  - You can perform **rotations**.

Red-Black Trees

- Experiments
- Insert 50, 25, 75.

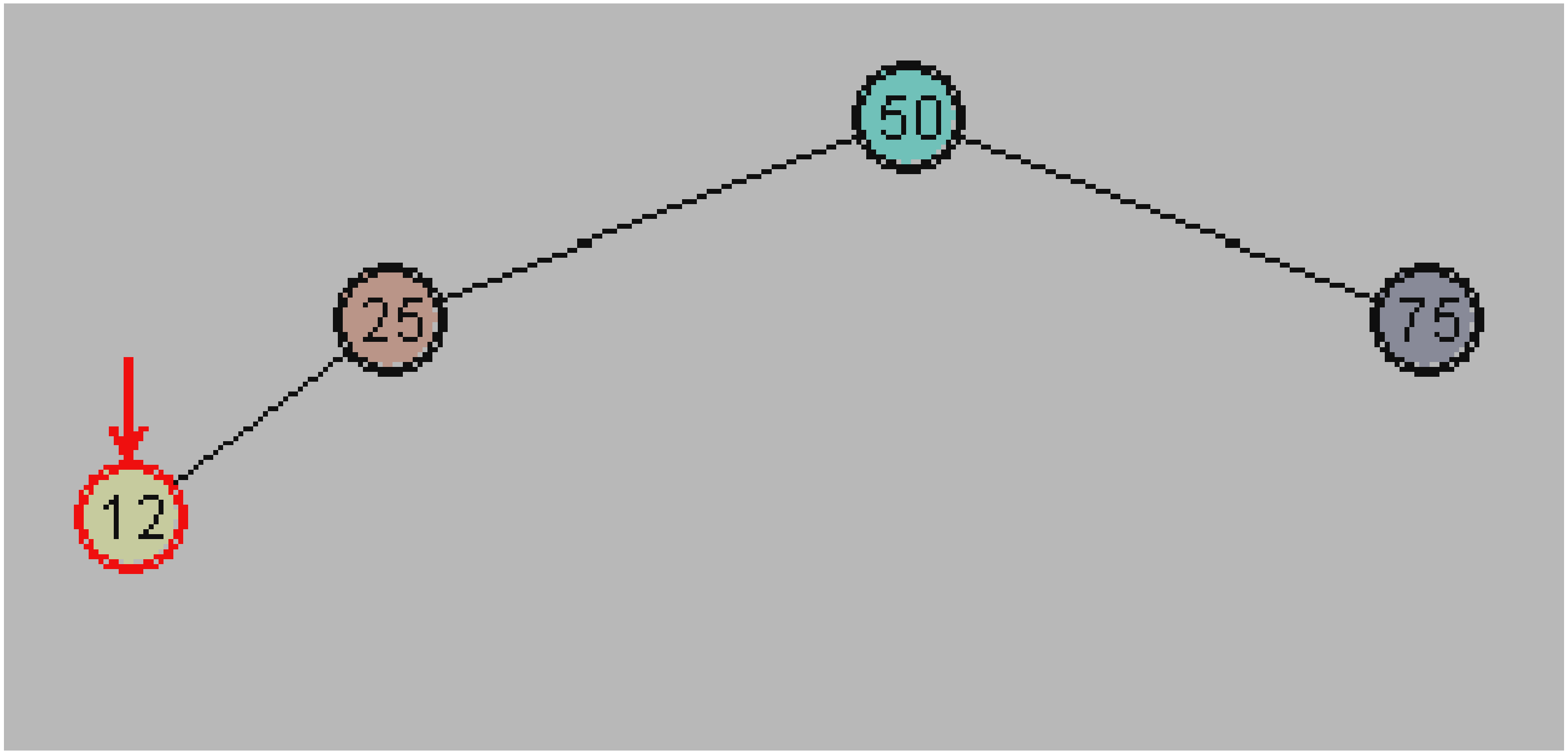


Data Structures & Algorithms

Red-Black Trees

- You can see that the tree is **Red-black tree**. The newly inserted nodes are colored red (except the root). **Inserting a red node** is less likely to **violate** the **red-black** rules than inserting a black one. This is because if a new red node is attached to a black one, **no rule is broken**. If you attach a red node to a red node, **rule 3** will be violated. However, it will happen **half of the time**. But if we add a **new black node** it will **always** change the black height for its path, **violating rule 4**.
- In a red-black tree, **if a rule is violated** we must used **rotation or flip color.**
- Try to **insert 12** in the tree. When we insert this item the **rule is violated**. If we color the new node (node 12) **black rule 4 is violated** and if we color the new node **red rule 3 is violated**. Therefore, we need **either rotation or color flip**. Here we change the **color of 25** and **75**. These two nodes become black and the new node **(12) is red**.

Red-Black Trees

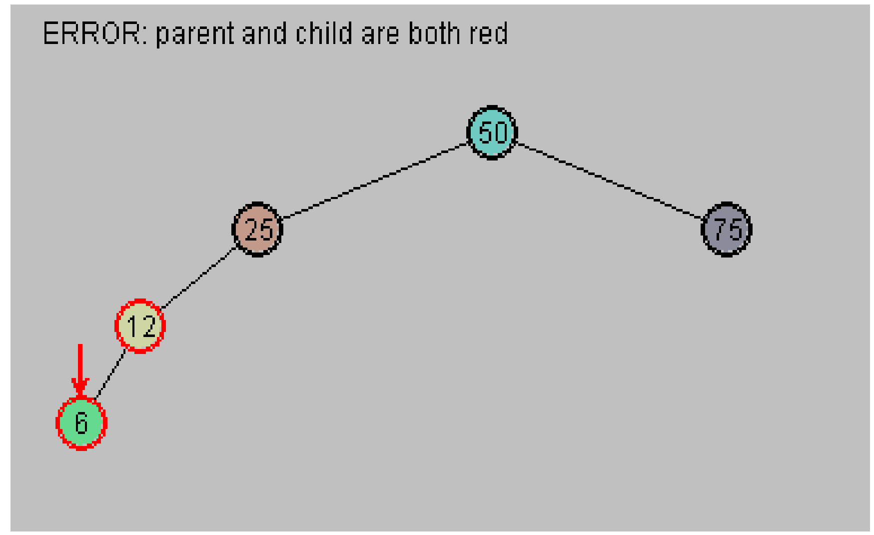


Red-Black Trees

- **After color flip** and inserting new node the tree is still **red-black** tree. Note that in the above tree one path (**left path**) is one node more than the **right path**. But the tree is **still red-black** tree and no rules are violated. The tree is **not very unbalanced**. But if **one path** is **two or more nodes** longer than the other than the **rules will be violated.**

Red-Black Trees

- Now try to **insert 6** into the tree. **Rule 3** has been **violated**.
- How can we **fix** this rule?
- The **easy way** is to change one of the node to **black** (i.e. node 6).



Red-Black Trees

- But now **rule 4 is violated**. Left sub-tree has more black nodes than the right sub-tree. Therefore, this problem can be fixed with a **rotation** and **some color changes**.

Red-Black Trees

- **Rotation**
- To balance a tree you have to **physically rearrange** the nodes. For example, if all the nodes are to the **right of the tree** you have to move some of them to the **left side**. This is called **rotation**. Rotation must do two things in once:
  - **Raise** some nodes and **lower** others to help **balance** the tree.
  - **Ensure** that the **characteristics** of a BST are not violated.

Remember that the **rotation** must keep the tree as a **BST**.

Red-Black Trees

- **Inserting a new node**
- For this example consider the following:
  - **X** is a particular **node**
  - **P** is the **parent** of X
  - **G** is the **grandparent** of X
- On the **way down the tree** to find the **insertion point**, you perform a **color flip** whenever you **find a black node with two red children** (a **violation of rule 2**). **Sometimes** the **flip** causes a **red-red conflict** (a **violation of rule 3**). Call the **red child X** and the **red parent P**. The conflict can be fixed with a **single rotation or a double rotation**, depending on whether **X** is an **outside or inside grandchild of G**. Following color flips and rotations, you continue down to the **insertion point** and insert the new node.

Red-Black Trees

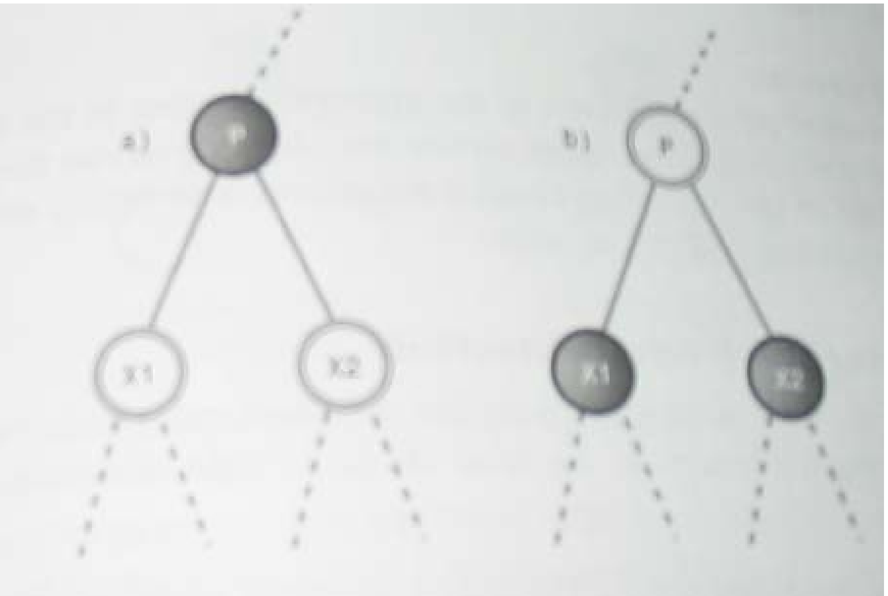
- **Color flips on the way down**
  - The **insertion routine** in a red-black tree starts off doing essentially the **same thing** it does in an **ordinary BST**: It **follows a path** from the root to the **place** where the **node should be inserted**, going **left or right** at each node depending on the relative size of the node's key and the search key. However, in a **red-black** tree, getting to the insertion point is **complicated** by **color flips and rotations**.

Red-Black Trees

- To make **sure** that the **color rules** are not **broken**, it needs to perform **color flips** when necessary. Here is the rule:
  - Every time the **insertion routine** encounters a **black node** that **has two red children**, it must change the **children to black** and the **parent to red** (unless the parent is the root)

Red-Black Trees

- **How does a color flip affect the red-black rules?**
  - Call the **top node P**, for parent, and its **left child X1** and **right child X2**. See figure 9.a.

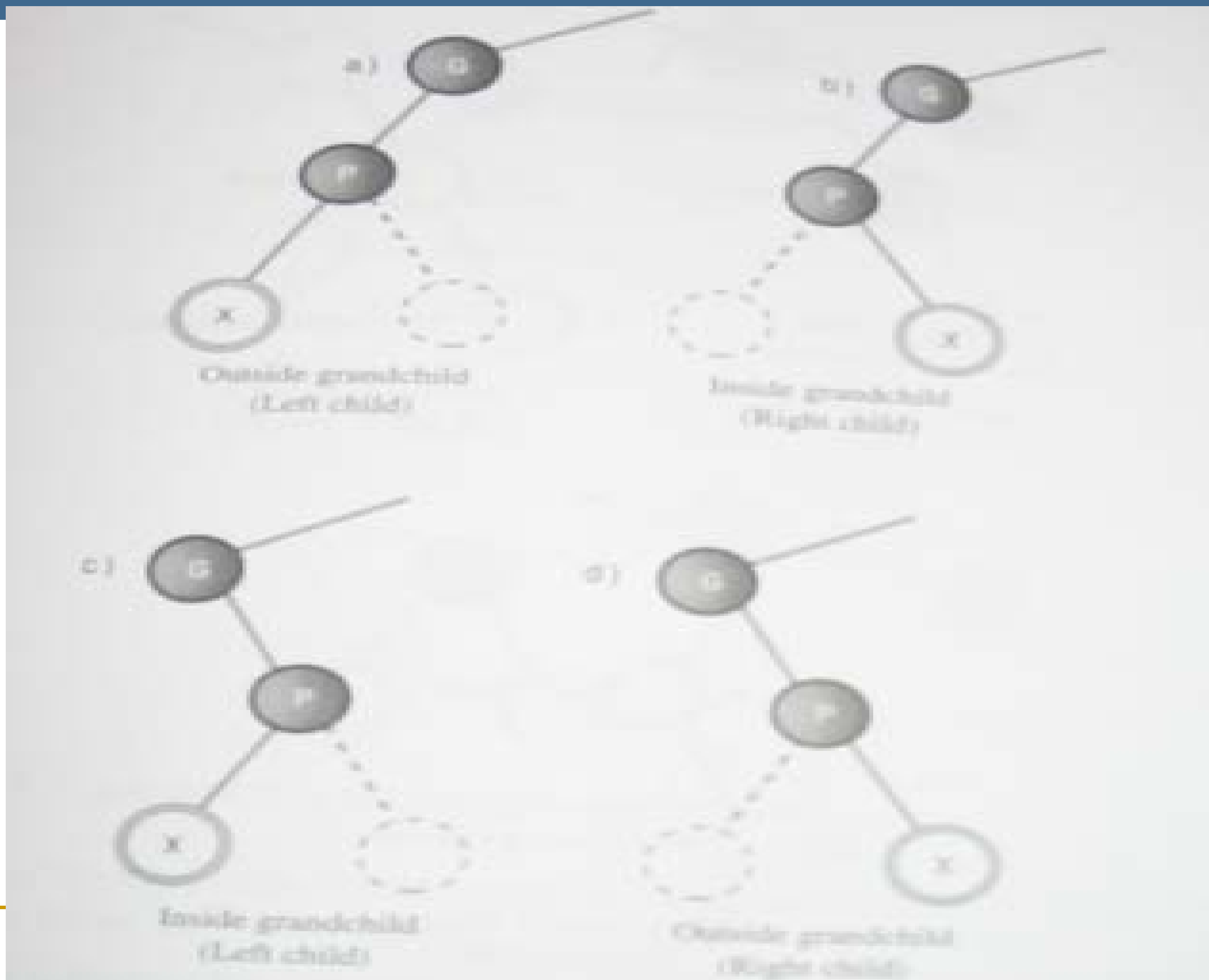


Color flip.

Red-Black Trees

- Remember that the newly inserted node which we call it X is always red. X may be located in various positions relative to P and G, as shown in the following figure:

Red-Black Trees

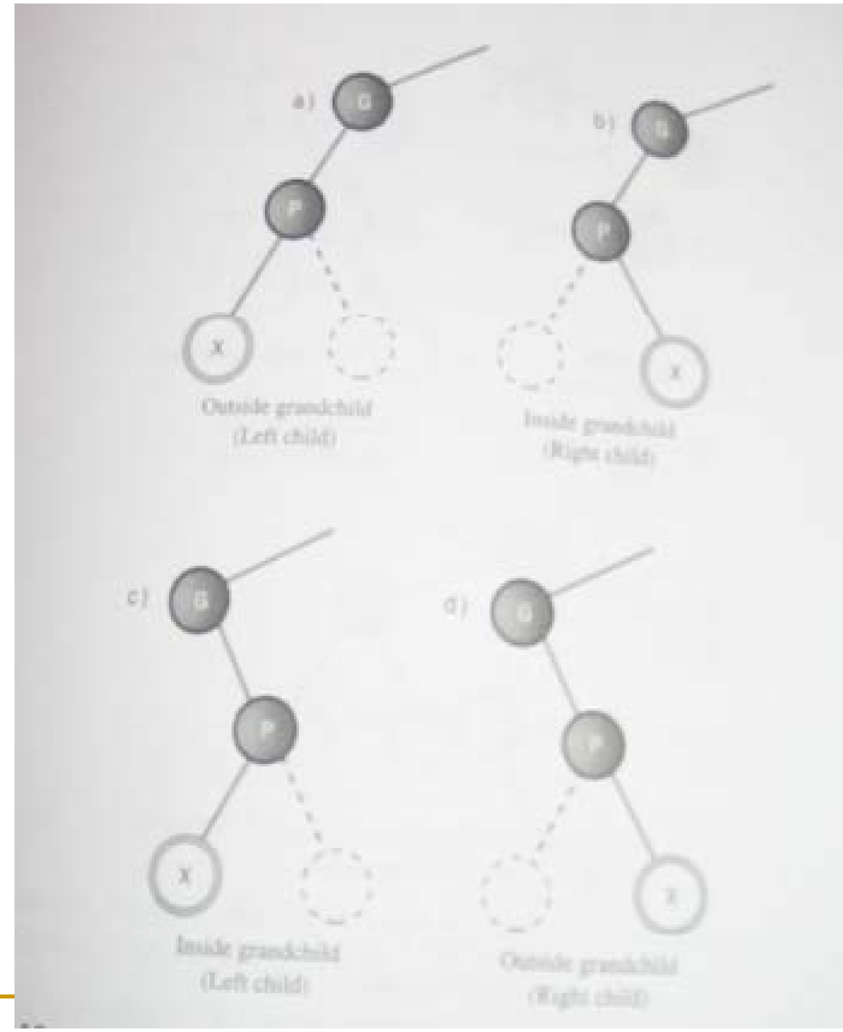


Red-Black Trees

- The action we take to **restore** the **red-black rules** is determined by the colors and configuration of X and its relatives. Perhaps surprisingly, nodes can be arranged in only **three major ways**. Each possibility must be dealt with in a different way to preserve red-black correctness and thereby lead to a balanced tree.

Red-Black Trees

- We will list the **three possibilities**. The following figure shows that they look like:
- Remember that **X is always red.**
  - **P is black**.
  - **P is red** and **X** is an **outside** grandchild of **G**.
  - **P is red** and **X** is an **inside** grandchild of **G**.



Red-Black Trees

- **P is black**:
  - **If P is black**, the **newly inserted** node is always **red**. So we don't need to do any thing. The **insertion** is **complete**.
- **P is red an X is outside**:
  - **If P is red** and **X** is an **outside** grandchild, we need a **single rotation** and some color changes. For example insert the following nodes: **50, 25, 75, and 12**. You will need to do **color flip** before you insert the **12**. Now **insert 6**, which is X, the new node. Figure 9.14 a shows the resulting tree.

Red-Black Trees

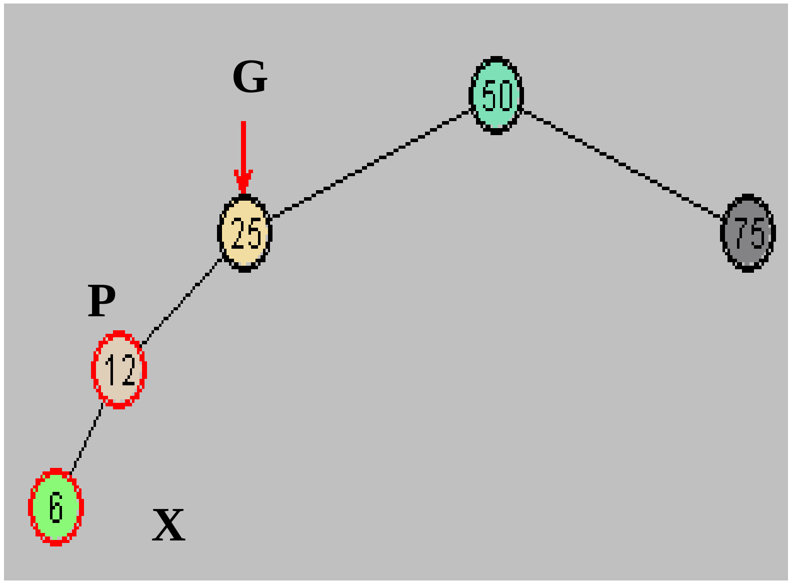
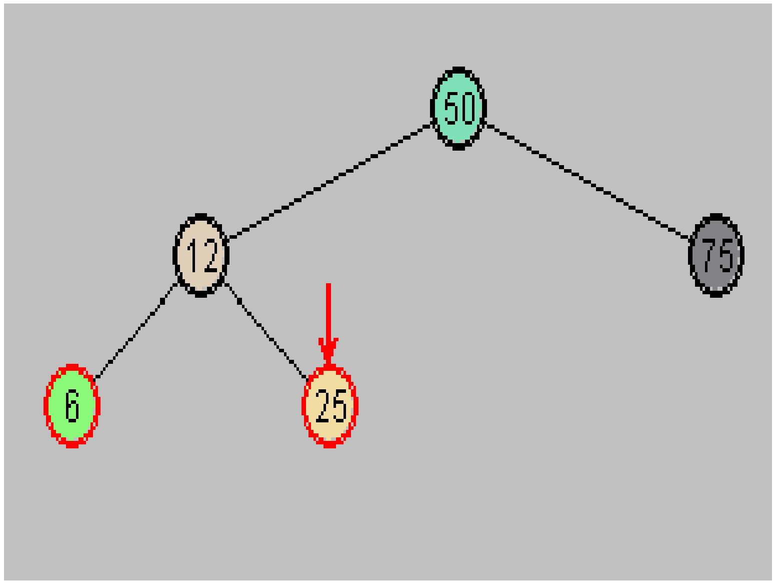


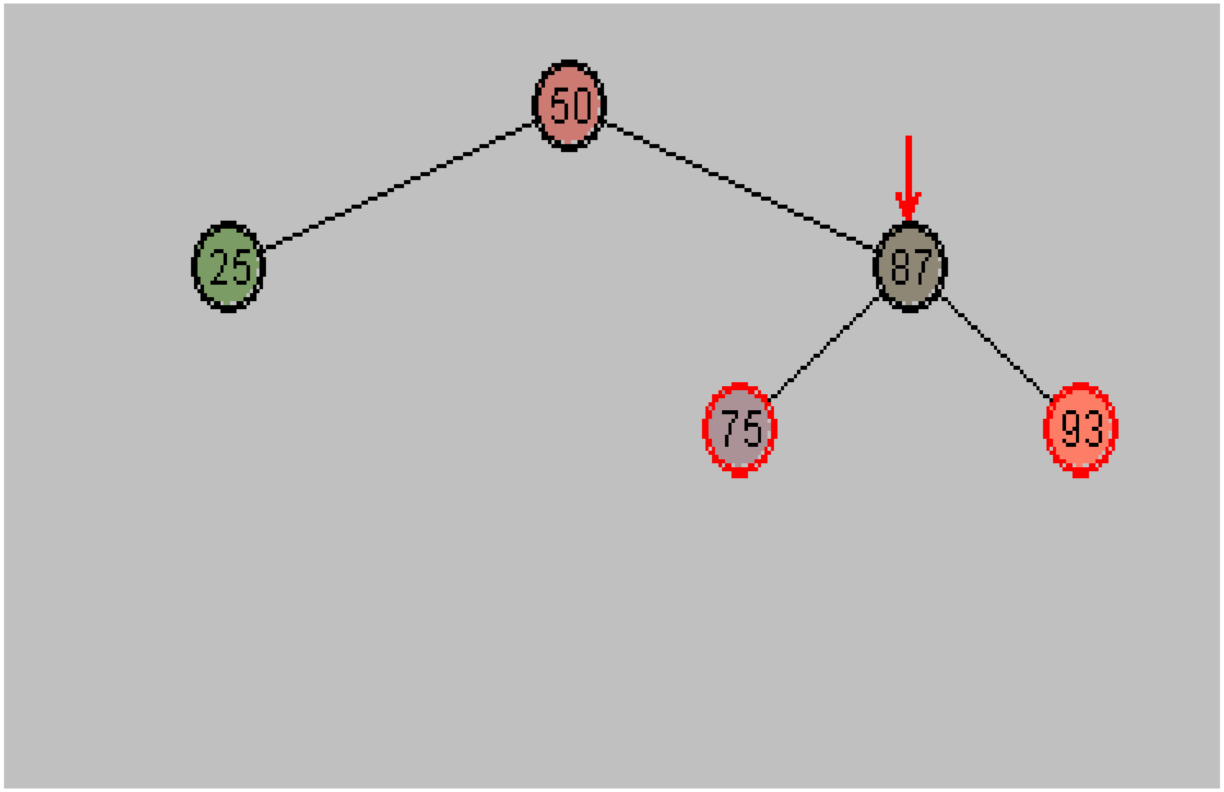
Figure 9.14 a



Red-Black Trees

- In this situation, we can take **three steps** to **restore** red-black correctness and thereby balance the tree. Here are the tree:
  - Switch the **color of X's grandparent** G (25 in this example).
  - Switch the **color of X's parent** P (12).
  - **Rotate** with X's grandparent G (25) at the top, in the direction that raises X (6). This is a right rotation in the example.
- In this example, **X was an outside** grandchild and a **left child**. There is a **symmetrical** situation when the **X** is an **outside** grandchild but a **right child**. Try this by creating the tree **50, 25, 75, 87, 93** (with color flip when necessary). Fix it by changing the colors of 75, and 87, and **rotating left** with 75 at the top. Again the tree is balanced.

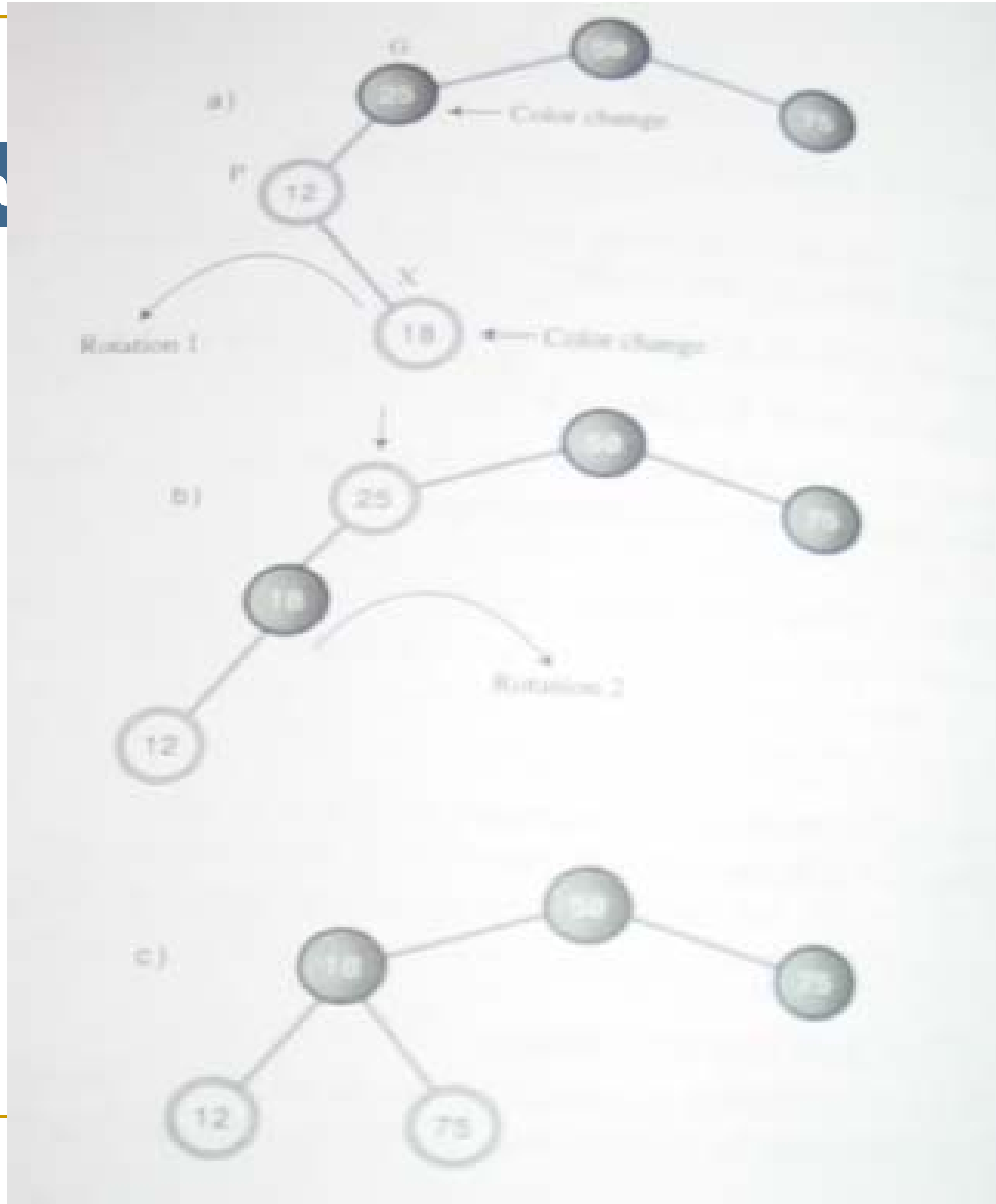
Red-Black Trees



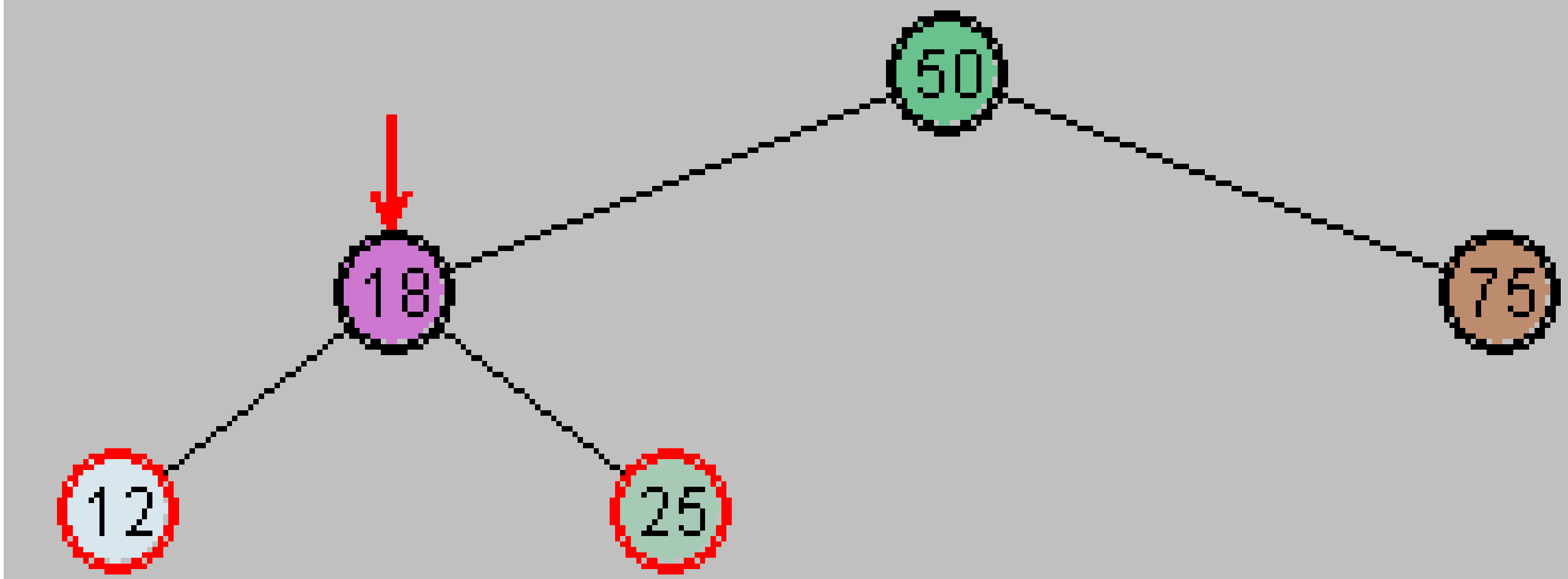
Red-Black Trees

- **P is red and X is inside**:
  - **If P is red** and **X** is an **inside** grandchild, we need **two rotations** and some **color changes**. To see this create a tree with **50, 25, 75, 12, 18** (again you need color flip before inserting 12). The result is shown in figure 9.15.

Red-Bla


a)
G
P
12
X
18
Color change
Color change
Rotation 1
b)
25
12
Rotation 2
c)
12
75

Red-Black Trees



Red-Black Trees

- Note that the **18** node is an **inside** grandchild. It and its parent are both red, so again you see the error message: **parent and child both red**. Fixing this arrangement is slightly more **complicated**. If we try to rotate right with the arrangement node G (25) at the top, as we did in Possibility 2, the inside grandchild X (18) moves across rather than up, so the tree is no more balanced than before. A different solution is needed.
- The trick when **X** is an **inside** grandchild is to perform **two rotations** rather than one. The **first** changes **X** from an **inside** grandchild to an **outside** grandchild, as shown in figure 9.15 a and 9.15 b. Now the situation is similar to Possibility 1, and we can apply the same rotation, with the grandparent at the top, as we did before. The result is shown in figure 9.15 c. We must recolor the nodes. We do this before doing any rotations.

Red-Black Trees

- Here are the steps:
  - **Switch the color** of X's **grandparent** (25 in this example).
  - **Switch the color of X** (not its parent; X is 18 here).
  - **Rotate** with X's parent P at the top (not the grandparent; the parent is 12), in the direction that raises X (a left rotation in this example).
  - **Rotate again** with X's grandparent (25) at the top, in the direction that raises X (a right rotation).
- **The rotation and re-coloring** restore the tree to red-black correctness and also balance it.

Red-Black Trees

The efficiency of Red-black trees

- A red-black tree allows for **searching, insertion, and deletion** in **O (log N) time**. The only **penalty** is that the storage required for each node is increased slightly to accommodate the red- black color (a **boolean variable**).
- The **time** for **insertion** and **deletion** are **increased** by a constant factor because of having to perform **color flips** and **rotation** on the way down and at the insertion point. On the **average**, an **insertion** requires about **one rotation**. Therefore, **insertion** still takes **O (log N) time** but is **slower** than insertion in the ordinary binary tree.
- Because in **most applications** there will be **more searches** than **insertion and deletion**, there is probably not much overall time penalty for using a red-black tree instead of an ordinary tree. The **advantage** is that in a red-black tree sorted data does not lead to **slow O(N)** performance.

Red-Black Trees