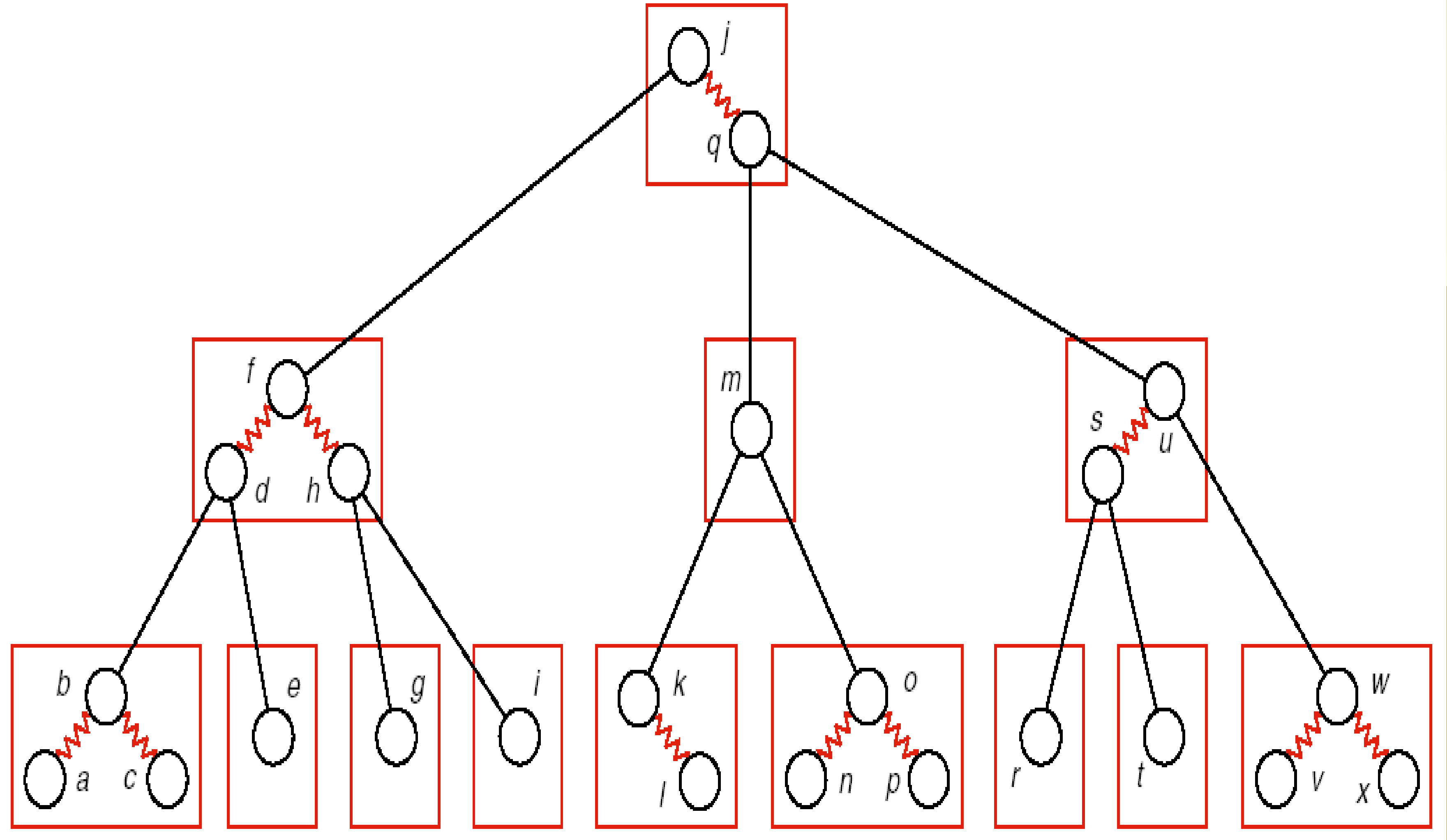
Red-Black Trees as B-Trees of Order 4

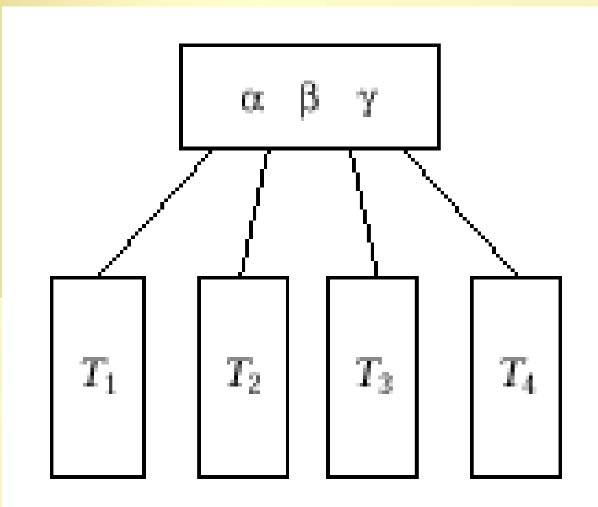
- ***A red-black tree*** **is a binary search tree, with links colored red or black, obtained from a B-tree of order 4 by the above conversions.**

- **Start with a B-tree of order 4, so each node contains 1, 2, or 3 entries.**

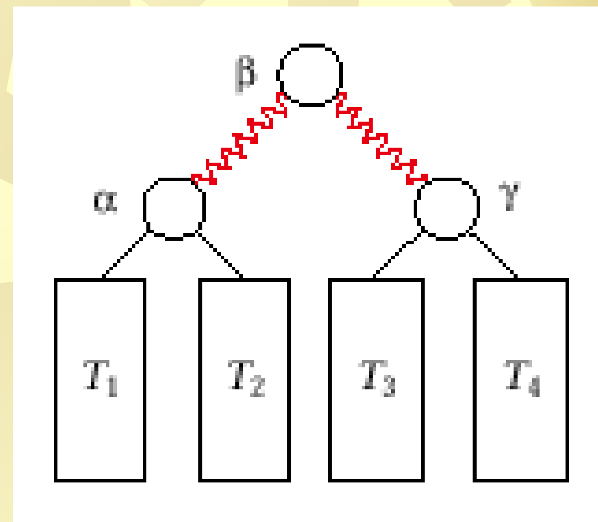

j
q
f
d
h
m
s
u
b
a
c
e
g
i
k
l
o
n
p
r
t
w
v
x

Red-Black Trees as B-Trees of Order 4

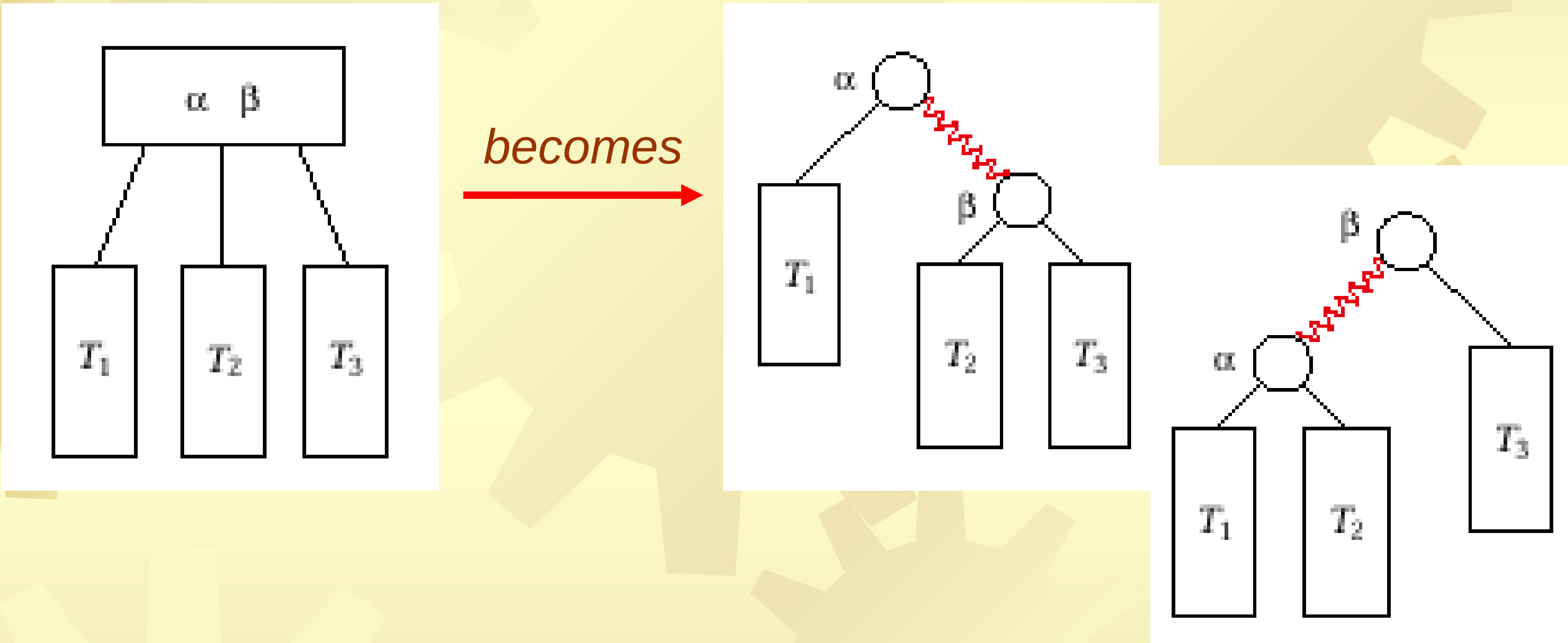
- **Convert a node with 3 entries into a binary search tree by:**



becomes



- **A node with two entries has two possible conversions:**



- A node with one entry remains unchanged.

Operations

- **Searching** and **traversal** of a red-black tree are exactly the same as for an ordinary binary search tree.
- **Insertion** and **deletion**, require care to maintain the underlying B-tree structure.
- Each node of a red-black tree is colored with the **same color as the link** immediately above it. We thus need keep only one extra bit of information for each node to indicate its color.

Red-Black Trees as Binary Search Trees

- **We adopt the convention that the root is colored black and all empty subtrees (corresponding to NULL links) are colored black.**
- **The B-tree requirement that all its empty subtrees are on the same level becomes:**

The Black Condition

Every simple path
from the root to an empty subtree
goes through the same number of black nodes.

Red-Black Trees as Binary Search Trees

- **To guarantee that no more than three nodes are connected by red links as one B-tree node, and that nodes with three entries are a balanced binary tree, we require:**

The Red Condition

If a node is red, then its parent exists and is black.

Define

- ***A red-black tree*** **is a binary search tree in which each node has either the color red or black and that satisfies the black and red conditions.**
- ***THEOREM*** **: The height of a red-black tree containing n nodes is no more than 2lgn.**

Analysis of Red-Black Trees

- Searching a red-black tree with n nodes is O(log n) in every case.
- The time for insertion is also O(log n).
- An AVL tree, in its worst case, has height about 1.44 lg n and,on average, has an even smaller height. Hence red-black trees do not achieve as good a balance as AVL trees.
- Red-black trees are not necessarily slower than AVL trees,since AVL trees may require many more rotations to maintain balance than red-black trees require.

Red-Black Tree Specification

- The red-black tree class is derived from the binary search tree class.
- We begin by incorporating colors into the nodes that will make up red-black trees:
- Note the inline definitions for the constructors and other methods of a red-black node.

```
enum Color {red, black};
template <class Record>
struct RB_node: public Binary_node<Record>
  {
  Color color;
  RB_node(const Record &new_entry)
      {       color = red; data = new_entry;
              left = right = NULL; }
  RB_node( )
      {       color = red; left = right = NULL; }
  void set_color(Color c) { color = c; }
  Color get_color( ) const { return color; }
};
```

Modified Node Specification

- To invoke get_color and set_color via pointers, we must add virtual functions to the base struct Binary_node.
- After this modification, we can reuse all the methods and functions for manipulating binary search trees and their nodes.
- The modified node specification is:

```
template <class Entry>
struct Binary_node {
   Entry data;
   Binary_node<Entry> *left;
   Binary_node<Entry> *right;
   virtual Color get_color( ) const        { return
   red; }
   virtual void set_color(Color c) { }
   Binary_node( )                     { left = right =
   NULL; }
   Binary_node(const Entry &x)
        {       data = x; left = right = NULL; }
};
```

Red-Black Insertion

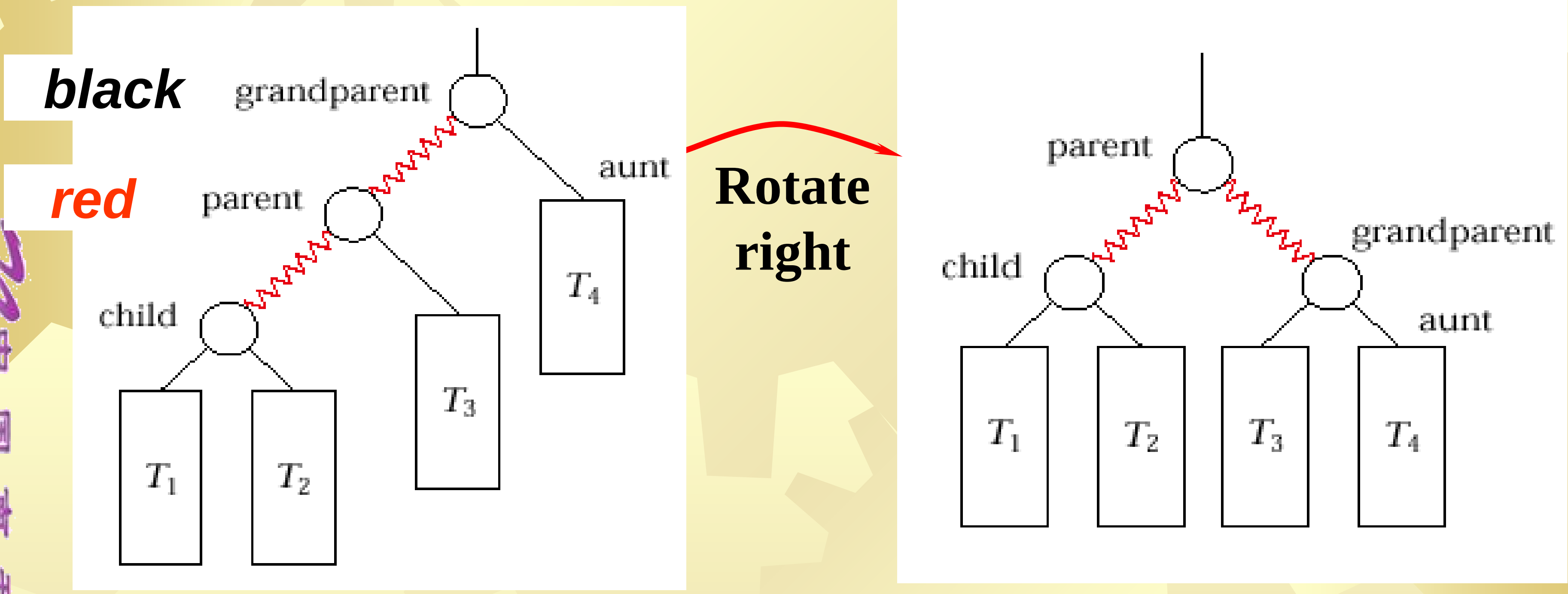
- We begin with the standard recursive algorithm for insertion into a binary search tree. The new_entry will be in a new leaf node.
- The black condition requires that the new node must be red.
- If the parent of the new red node is black, then the insertion is finished, but if the parent is red, then we have introduced a violation of the red condition into the tree.

- The major work of the insertion algorithm is to remove a violation of the red condition, and we shall find several different cases that we shall need to process separately.
- We postpone this work as long as we can. When we make a node red, we do not check the conditions or repair the tree. Instead, we return from the recursive call with a status indicator showing that the node just processed is red.

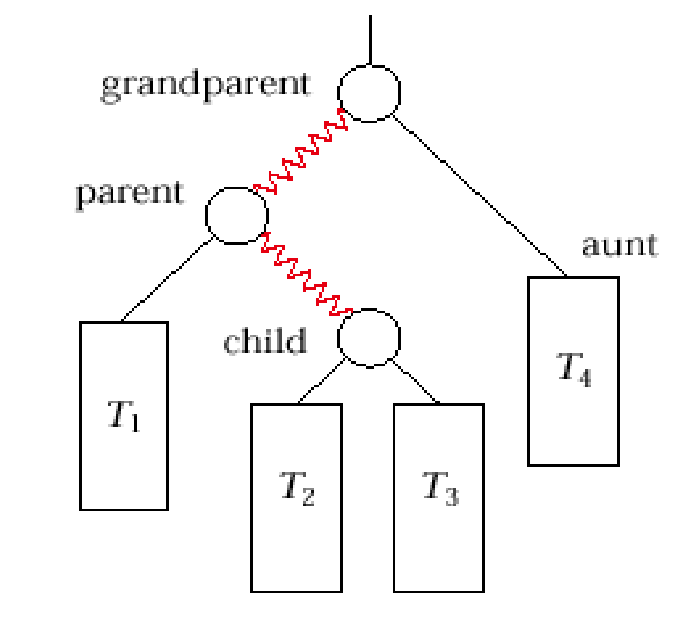
- **After this return, we are processing the parent node.**
- **If the parent is black, then the conditions for a red-black tree are satisfied and the process terminates.**
- **If the parent is red, then we set the status variable to show two red nodes together, linked as left child or as right child.Return from the recursive call.**

- We are now processing the grandparent node. Since the root is black and the parent is red, this grandparent must exist. By the red condition, this grandparent is black since the parent was red.
- At the recursive level of the grandparent node, we transform the tree to restore the red-black conditions, using cases depending on the relative positions of the grandparent, parent,and child nodes.
- See following diagram.

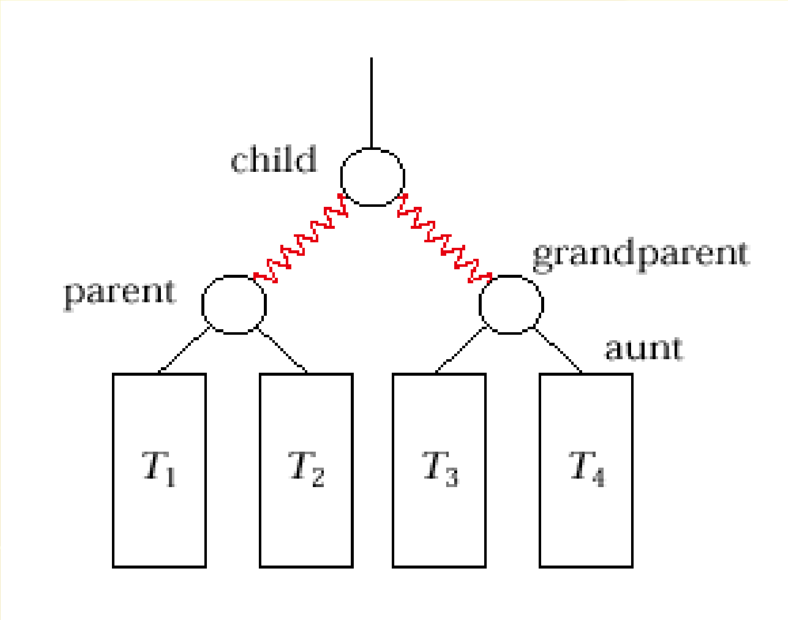
Red-Black Insertion



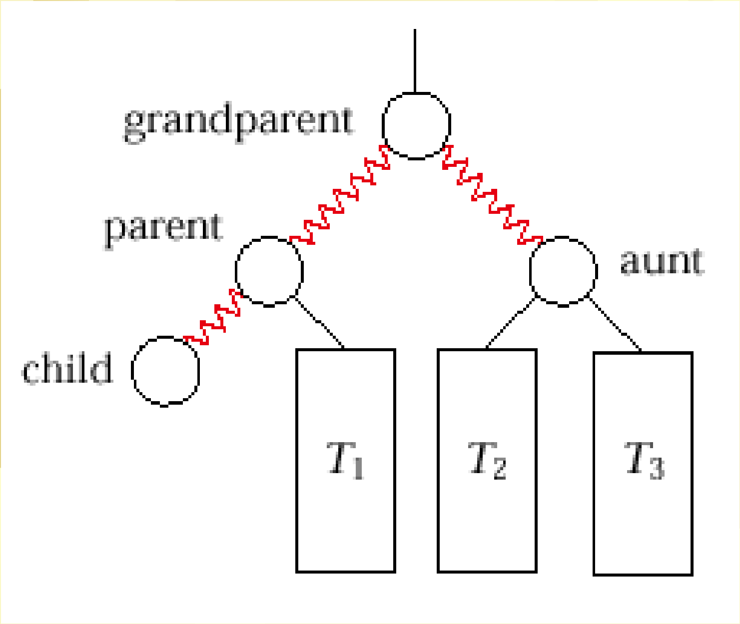
Red-Black Insertion



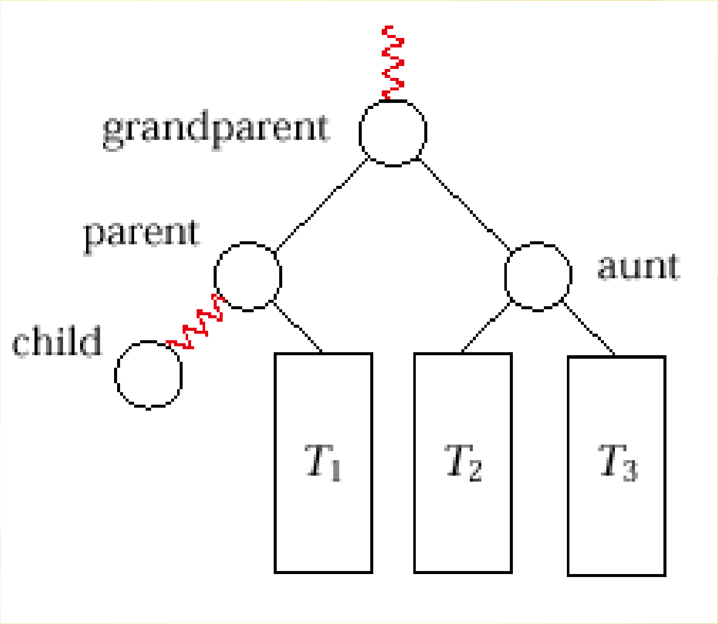
Double Rotate right



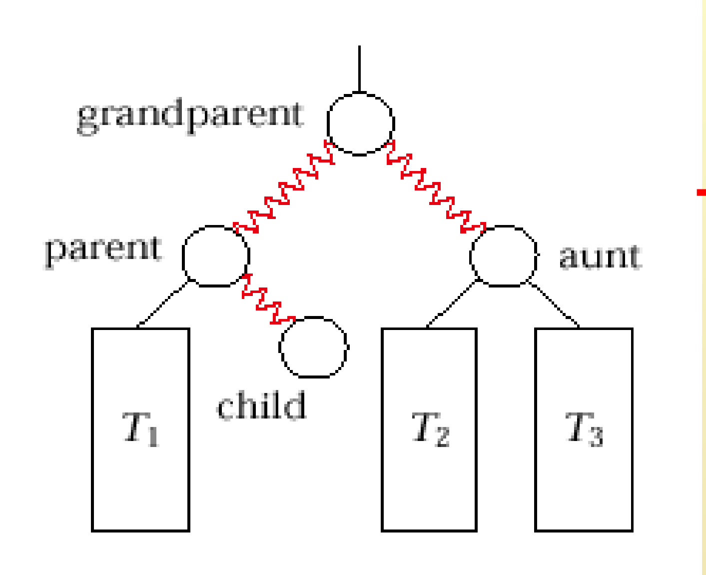
Red-Black Insertion



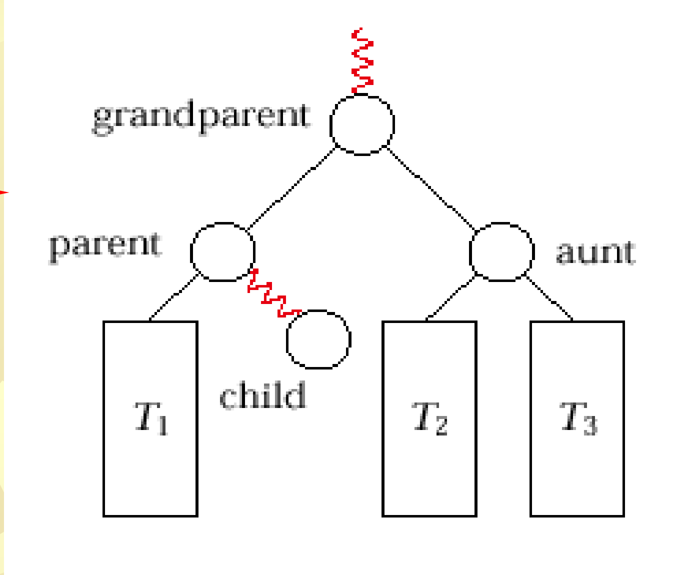
Color flip



Red-Black Insertion



Color flip



Class specification

```
template <class Record>
class RB_tree: public Search_tree<Record> {
public:
        Error_code insert(
                const Record & new_entry);
private:
                // Add prototypes for auxiliary functions
    here.
};
```

Status indicator values

enum RB_code {okay, red_node, left_red, right_red, duplicate};

/ These outcomes from a call to the recursive insertion function describe the following results:*

okay: The color of the current root (of the subtree) has not changed; the tree now satisfies the conditions for a red-black tree.

red_node: The current root has changed from black to red; modification may or may not be needed to restore the red-black properties.

right_red: ***The current root and its right child are now both red; a color flip or rotation is needed.***

left_red: ***The current root and its left child are now both red; a color flip or rotation is needed.***

duplicate: ***The entry being inserted duplicates another entry; this is an error.*/***

Public Insertion Method

template <class Record>

Error_code RB_tree<Record> :: insert(const Record &new_entry)

/* Post: If the key of new_entry is already in the RB_tree , a code of duplicate_error is returned. Otherwise, a code of success is returned and the Record new_entry is inserted into the tree in such a way that the properties of an RB-tree have been preserved.

Uses: Methods of struct RB_node and recursive function rb_insert . */

```
{
    RB_code status = rb_insert(root, new_entry);
    switch (status) { // Convert private RB_code to public
    error_code .
        case red node: //Always split the root node to keep it
    back
            root->set color(black); /* Doing so prevents
    two red nodes at the top of the tree and a resulting attempt to
    rotate using a parent node that does not exist.*/
        case okay:
            return success;
        case duplicate:
            return duplicate error;
        case right_red:
        case left_red:
            cout << "WARNING: Program error in
                RB_tree::insert" << endl;
        return internal_error;}}
```

Recursive Insertion Function

```
template <class Record>
RB_code RB_tree<Record> ::
   rb_insert(Binary node<Record> * &current,
             const Record &new_ntry)
```

/ Pre: current is either NULL or points to the first node of a subtree of an RB_tree*

Post: If the key of new_entry is already in the subtree, a code of duplicate is returned. Otherwise, the Record new_entry is inserted into the subtree pointed to by current . The properties of a red-black tree have been restored, except possibly at the root current and one of its children, whose status is given by the output RB_code .

*Uses: Methods of class RB_node , rb_insert recursively, modify_left , and modify_right . */*

```
{RB_ code status,
        child status;
if (current == NULL) {
   current = new RB_node<Record>(new_entry);
   status = red _node;}
else if (new_entry == current->data)        return duplicate;
else if (new_ entry < current->data) {
   child status = rb_insert(current->left, new_entry);
   status = modify_left(current, child_status);}
else {
   Child_status = rb_insert(current->right, new_entry);
   status = modify_right(current, child_status);
}
return status;}
```

Checking the Status: Left Child

```
template <class Record> RB_code
   RB_tree<Record> ::
modify_left(Binary_node<Record> * &current,
RB_code &child_status)
/* Pre: An insertion has been made in the left subtree of
   *current that has returned the value of child_status
   for this subtree.
 Post: Any color change or rotation needed for the tree
   rooted at current has been made, and a status code
   is returned.
 Uses: Methods of struct RB_node , with
   rotate_right ,double_rotate_right , and Flip_color . */
```

Checking the Status: Left Child

```
{
    RB_code status = okay;
    Binary_node<Record>
                *aunt = current->right;
    Color aunt_color = black;
    if (aunt != NULL)
        aunt_color = aunt->get_color( );
```

```
switch (child_status) {
case okay: break;  // No action needed, as tree is already OK.
case red_node:
    if (current->get_color( ) == red)
        status = left_red;
   else status = okay; break; // curr is black, left is red, so OK.
case left_red:
   if (aunt_color == black)
       status = rotate_right(current);
   else status = flip_color(current); break;
case right_red:
  if (aunt_color == black)
     status = double_ rotate_ right(current);
  else status = flip_color(current); break;
}return status;}
```

Pointers and Pitfalls

- Trees are flexible and powerful structures both for modeling problems and for organizing data. In using trees in problem solving and in algorithm design, first decide on the kind of tree needed (ordered, rooted, free, or binary) before considering implementation details.
- Most trees can be described easily by using recursion; their associated algorithms are often best formulated recursively.

Pointers and Pitfalls

- For problems of information retrieval, consider the size, number,and location of the records along with the type and structure of the entries while choosing the data structures to be used. For small records or small numbers of entries, high speed internal memory will be used, and binary search trees will likely prove adequate. For information retrieval from disk files, methods employing multiway branching, such as tries, B-trees, and hash tables, will usually be superior.

Pointers and Pitfalls

- Tries are particularly well suited to applications where the keys are structured as a sequence of symbols and where the set of keys is relatively dense in the set of all possible keys. For other applications, methods that treat the key as a single unit will often prove superior. B-trees, together with various generalizations and extensions, can be usefully applied to many problems concerned with external information retrieval.