Chapter 11

MULTIWAY TREES

1. Orchards, Trees, and Binary Trees
2. Lexicographic Search Trees: Tries
3. External Searching: B-Trees
4. Red-Black Trees

On the Classification of Species

Definitions:

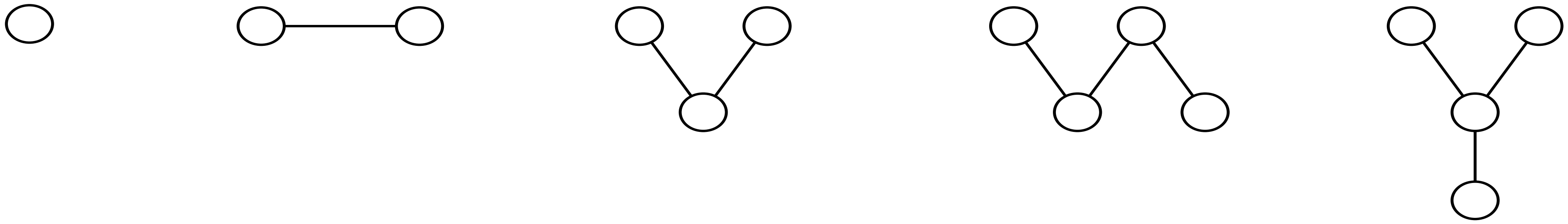
- A ***(free) tree*** is any set of points (called ***vertices***) and any set of pairs of distinct vertices (called ***edges*** or ***branches***) such that (1) there is a sequence of edges (a ***path***) from any vertex to any other, and (2) there are no ***circuits***, that is, no paths starting from a vertex and returning to the same vertex.

- A ***rooted tree*** is a tree in which one vertex, called the ***root***, is distinguished.

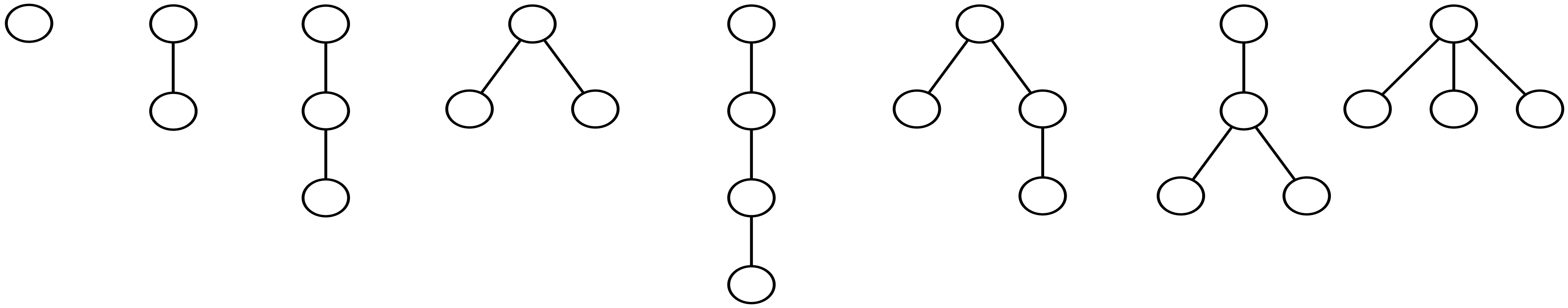
- An ***ordered tree*** is a rooted tree in which the children of each vertex are assigned an order.

- A ***forest*** is a set of trees. We usually assume that all trees in a forest are rooted.

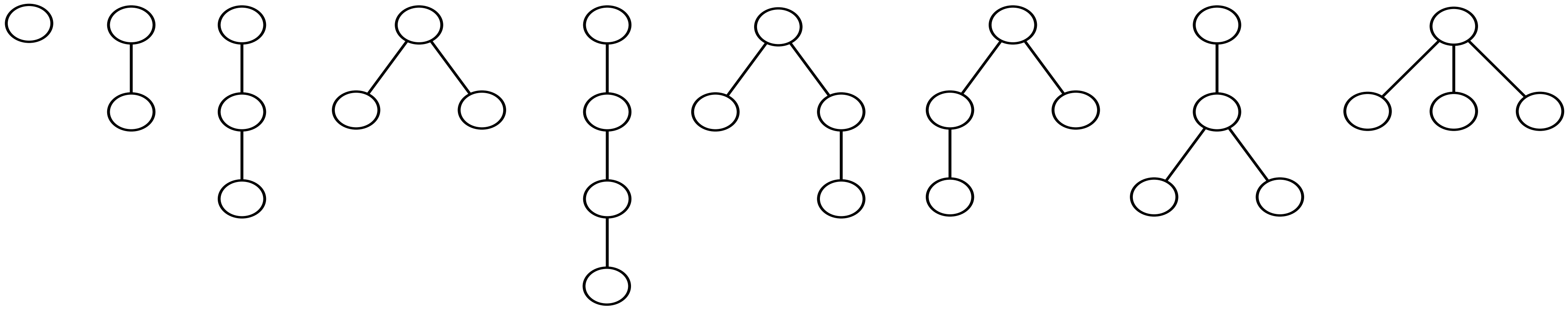
- An ***orchard*** (also called an ***ordered forest***) is an ordered set of ordered trees.



Free trees with four or fewer vertices
(Arrangement of vertices is irrelevant.)



Rooted trees with four or fewer vertices
(Root is at the top of tree.)

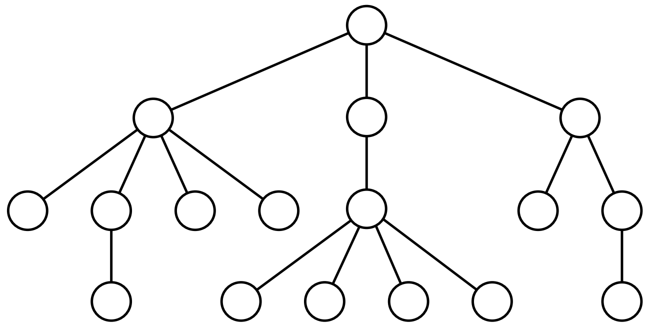


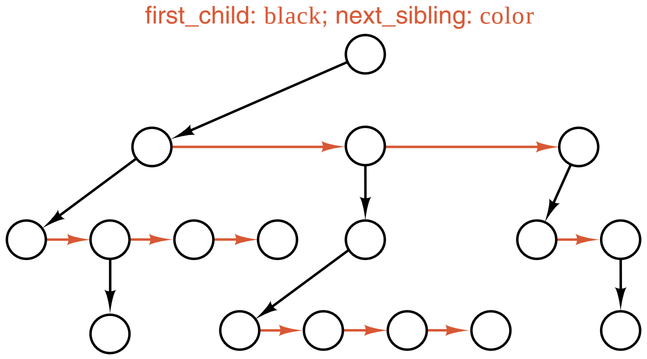
Ordered trees with four or fewer vertices

Implementations of Ordered Trees

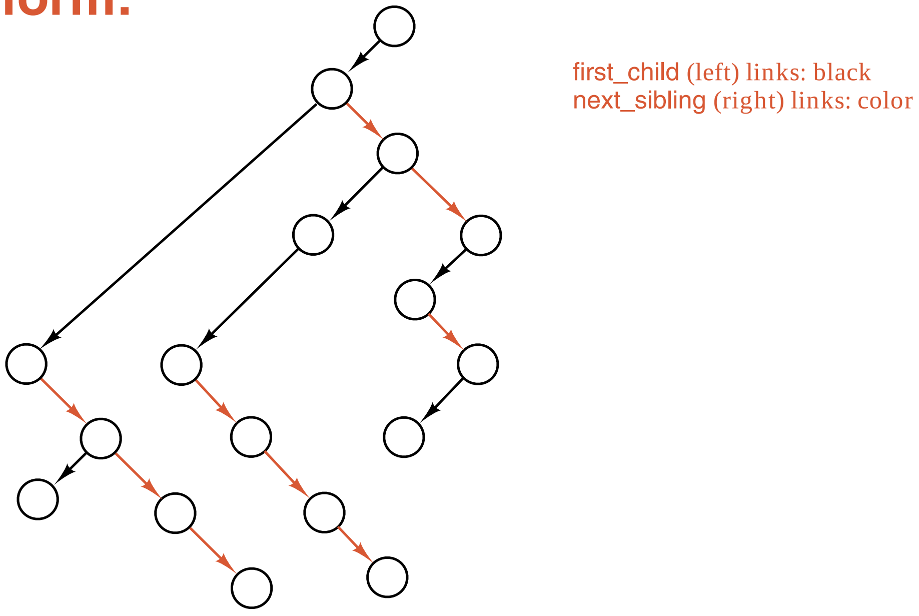
- Multiple links
- first_child and next_sibling links
- Correspondence with binary trees

Linked implementation of ordered tree:





Rotated form:


first_child (left) links: black
next_sibling (right) links: color

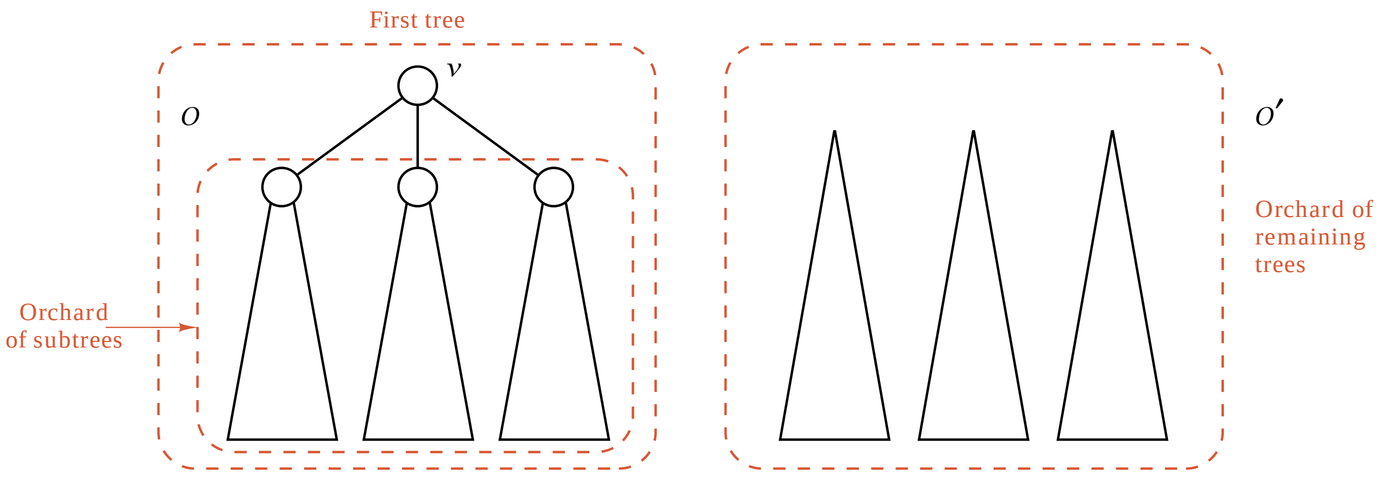
Recursive Definitions

DEFINITION A ***rooted tree*** consists of a single vertex v, called the ***root*** of the tree, together with a forest F, whose trees are called the ***subtrees*** of the root.

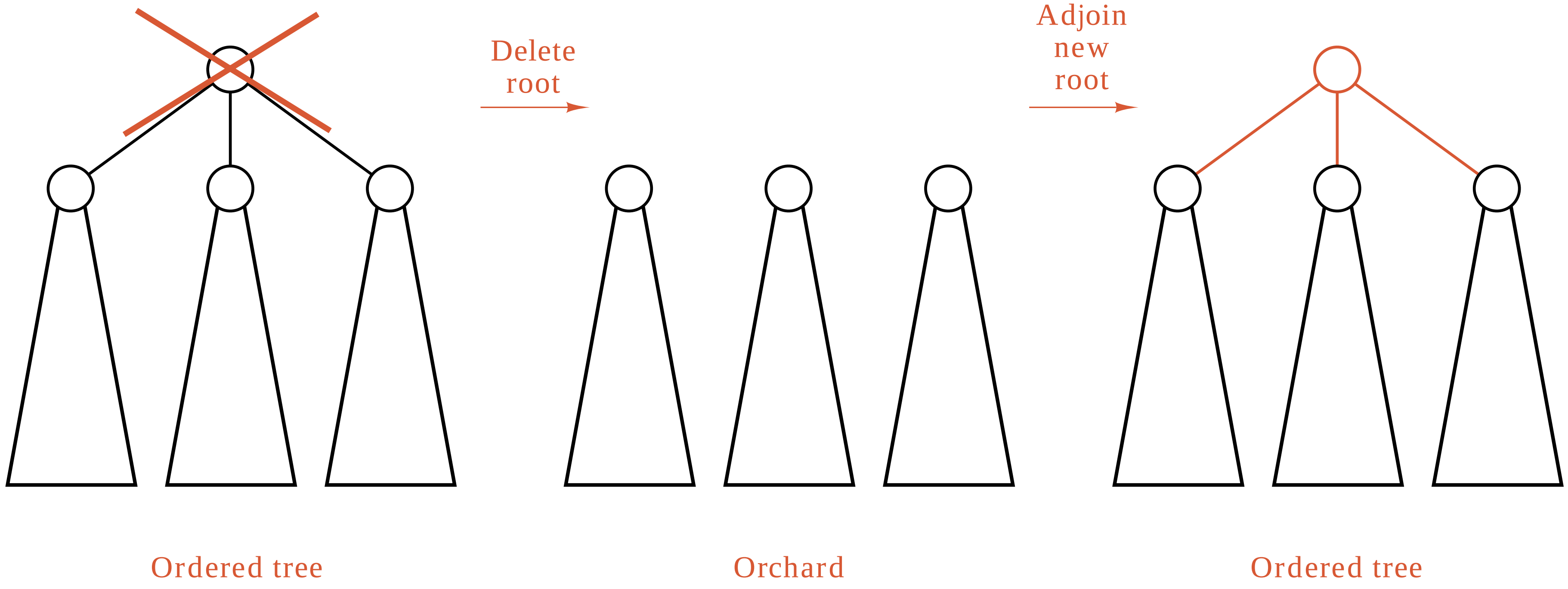
A ***forest*** F is a (possibly empty) set of rooted trees.

DEFINITION An ***ordered tree*** T consists of a single vertex v, called the ***root*** of the tree, together with an orchard O, whose trees are called the ***subtrees*** of the root v. We may denote the ordered tree with the ordered pair $T = \{v, O\}$.

An ***orchard*** O is either the empty set $\emptyset$, or consists of an ordered tree T, called the ***first tree*** of the orchard, together with another orchard O' (which contains the remaining trees of the orchard). We may denote the orchard with the ordered pair $O = (T, O')$.



Deleting and adjoining a root



The Formal Correspondence

DEFINITION A binary tree B is either the empty set $\emptyset$ or consists of a root vertex v with two binary trees B_1 and B_2. We may denote the binary tree with the ordered triple $B = [v, B_1, B_2]$.

THEOREM 11.1 Let S be any finite set of vertices. There is a one-to-one correspondence f from the set of orchards whose set of vertices is S to the set of binary trees whose set of vertices is S.

Proof. Define

$$f(\emptyset) = \emptyset.$$

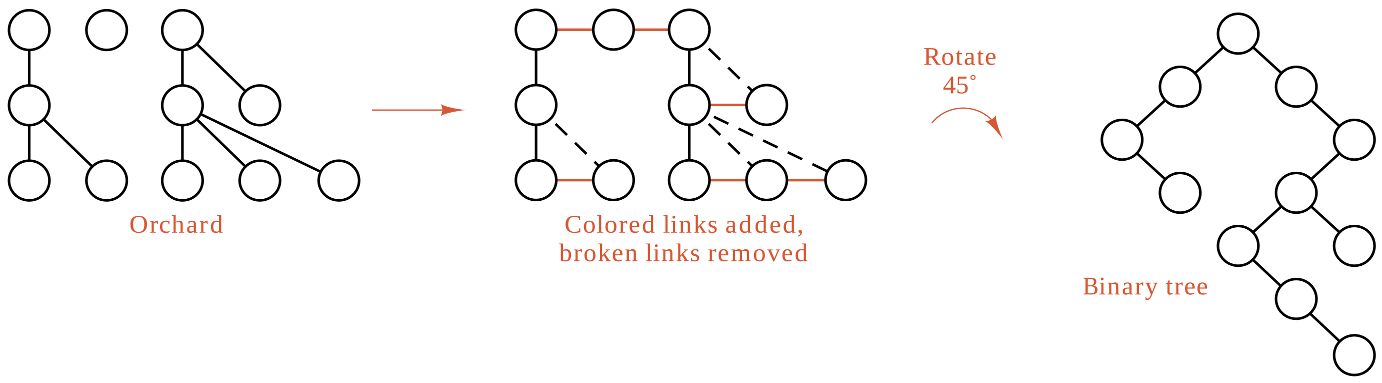
Define

$$f(\{v, O_1\}, O_2) = [v, f(O_1), f(O_2)].$$

Show by mathematical induction on the number of vertices that f is a one-to-one correspondence.

Rotations:

1. Draw the orchard so that the first child of each vertex is immediately below the vertex.
2. Draw a vertical link from each vertex to its first child, and draw a horizontal link from each vertex to its next sibling.
3. Remove the remaining original links.
4. Rotate the diagram 45 degrees clockwise, so that the vertical links appear as left links and the horizontal links as right links.



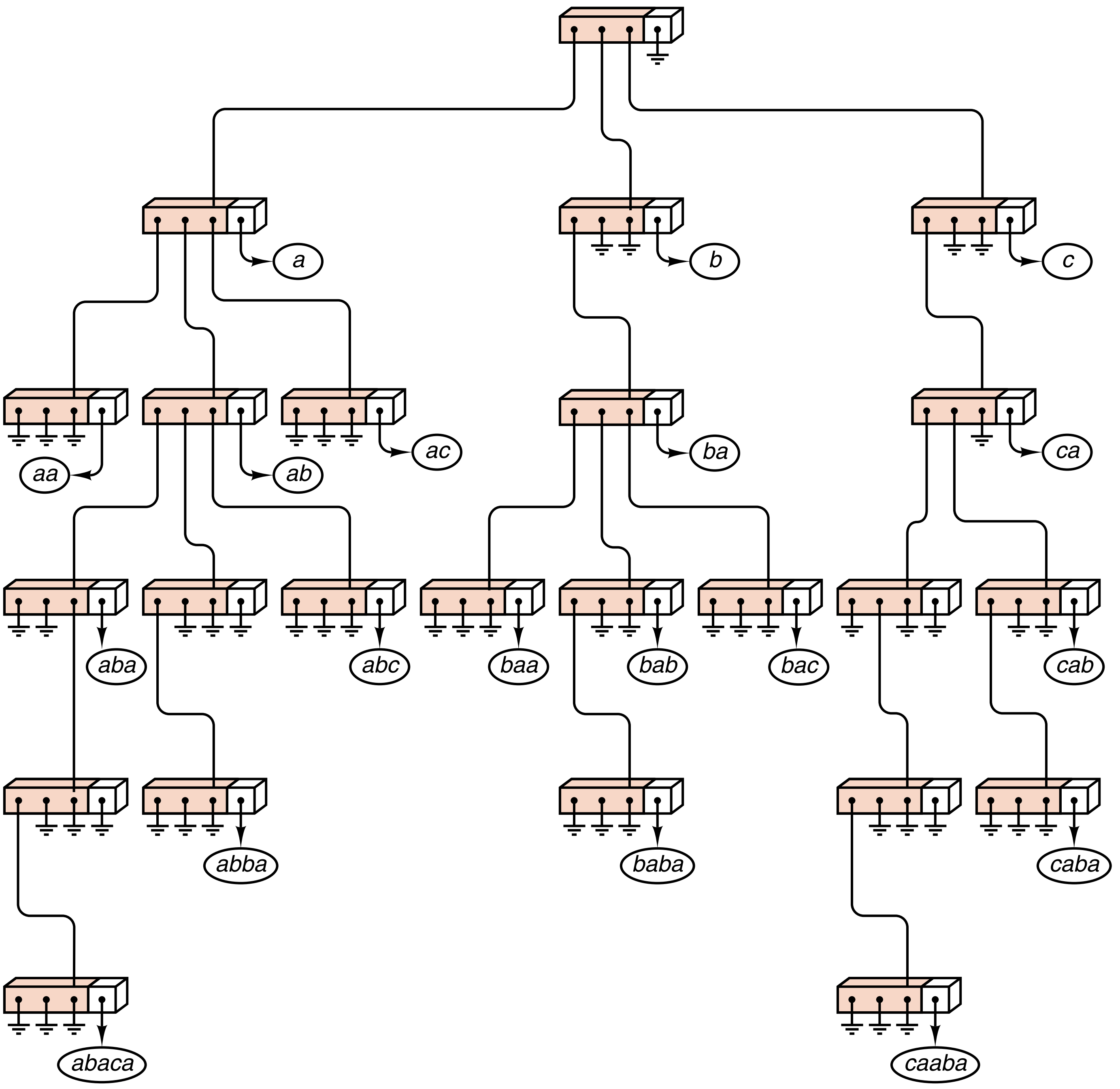
Summary:

Orchards and binary trees correspond by any of:

- first_child and next_sibling links,
- rotations of diagrams,
- formal notational equivalence.

Lexicographic Search Trees: Tries

DEFINITION A ***trie*** of order m is either empty or consists of an ordered sequence of exactly m tries of order m.



C++ Trie Declarations

- Every Record has a Key that is an alphanumeric string.
- Method **char** key_letter(**int** position) returns the character in the given position of the key or returns a blank, if the key has length less than position.
- Auxiliary function **int** alphabetic_order(**char** symbol) returns the alphabetic position of the character symbol, or 27 for nonblank, nonalphabetic characters, or 0 for blank characters.

```
class Trie {
public:                            //  Add method prototypes here.

private:                           //  data members
   Trie_node *root;
};


const int num_chars = 28;


struct Trie_node {
//  data members
   Record *data;
   Trie_node *branch[num_chars];
//  constructors
   Trie_node( );
};
```

Searching a Trie

```
Error_code Trie :: trie_search(const Key &target, Record &x) const
/* Post: If the search is successful, a code of success is returned, and the output
         parameter x is set as a copy of the Trie's record that holds target. Otherwise,
         a code of not_present is returned.
   Uses: Methods of class Key. */

{
   int position = 0;
   char next_char;
   Trie_node *location = root;
   while (location != NULL &&
           (next_char = target.key_letter(position)) != ' ') {
         //  Terminate search for a NULL location or a blank in the target.
      location = location->branch[alphabetic_order(next_char)];
         //  Move down the appropriate branch of the trie.
      position++;
         //  Move to the next character of the target.
   }

   if (location != NULL && location->data != NULL) {
      x = *(location->data);
      return success;
   }
   else
      return not_present;
}
```

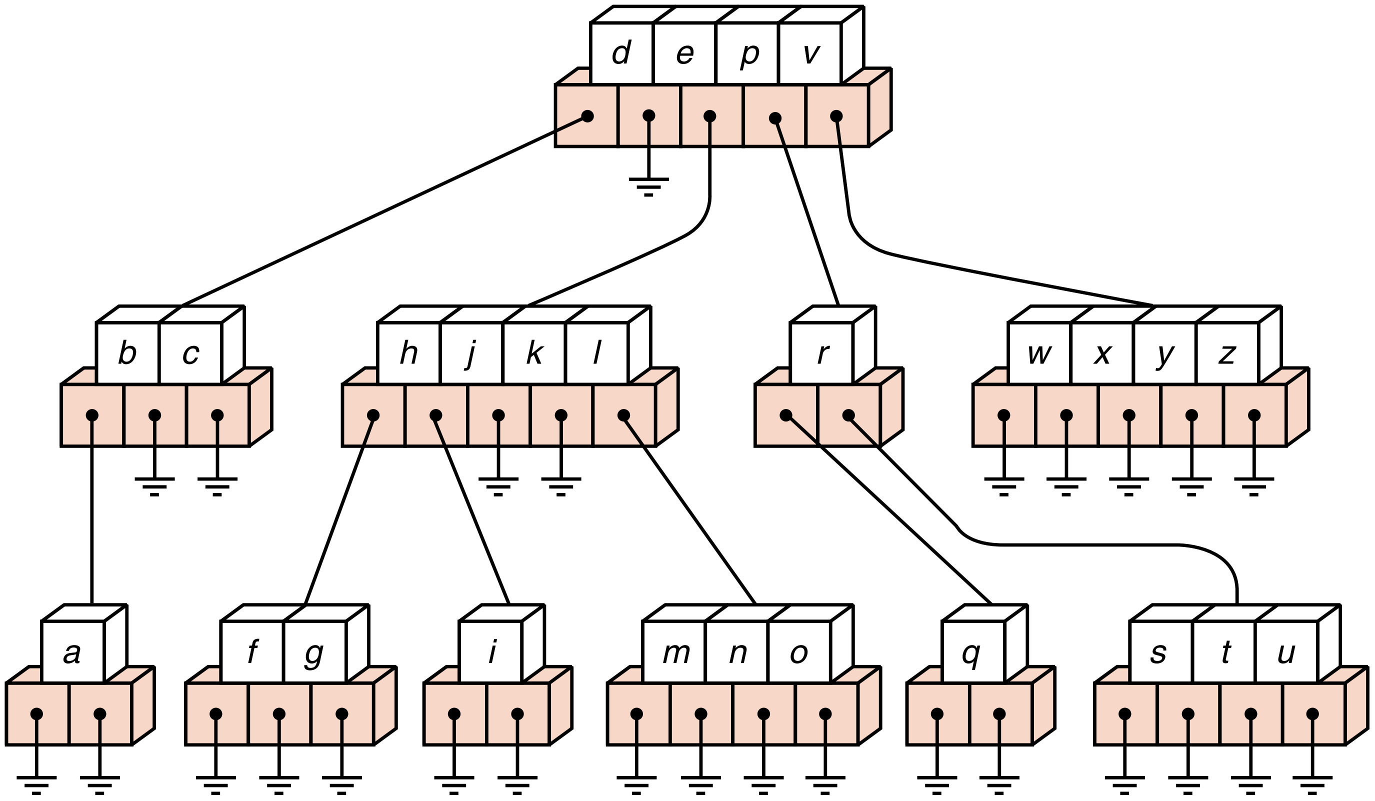
Insertion into a Trie

```
Error_code Trie :: insert(const Record &new_entry)
/* Post: If the Key of new_entry is already in the Trie, a code of duplicate_error
         is returned. Otherwise, a code of success is returned and the Record
         new_entry is inserted into the Trie.
   Uses: Methods of classes Record and Trie_node. */
{
   Error_code result = success;
   if (root == NULL) root = new Trie_node;   //  Create a new empty Trie.
   int position = 0;                         //  indexes letters of new_entry
   char next_char;
   Trie_node *location = root;               //  moves through the Trie

   while (location != NULL &&
          (next_char = new_entry.key_letter(position)) != ' ') {
      int next_position = alphabetic_order(next_char);
      if (location->branch[next_position] == NULL)
         location->branch[next_position] = new Trie_node;
      location = location->branch[next_position];
      position++;
   }
   //  At this point, we have tested for all nonblank characters of new_entry.
   if (location->data != NULL) result = duplicate_error;
   else location->data = new Record(new_entry);
   return result;
}
```

The number of steps required to search a trie or insert into it is proportional to the number of characters making up a key, not to a logarithm of the number of keys as in other tree-based searches.

Multiway Search Trees

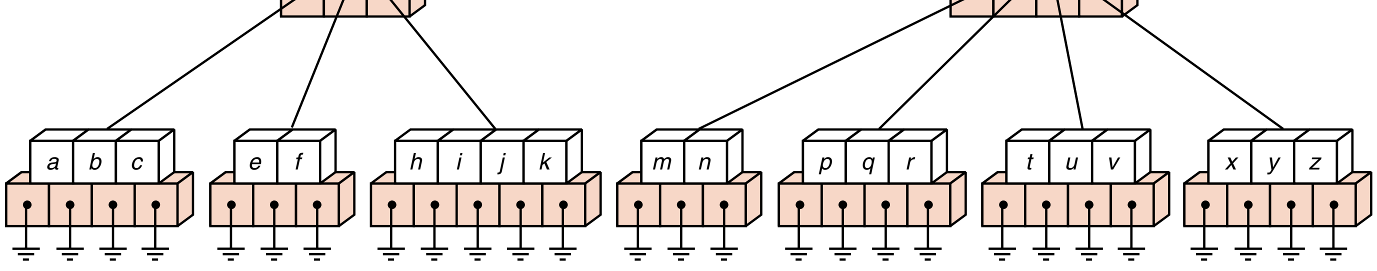
- An m***-way search tree*** is a tree in which, for some integer m called the ***order*** of the tree, each node has at most m children.
- If $k \le m$ is the number of children, then the node contains exactly $k - 1$ keys, which partition all the keys into k subsets consisting of all the keys less than the first key in the node, all the keys between a pair of keys in the node, and all keys greater than the largest key in the node.



Balanced Multiway Trees (B-Trees)

DEFINITION A ***B-tree of order*** m is an m-way search tree in which

1. All leaves are on the same level.
2. All internal nodes except the root have at most m nonempty children, and at least $\lceil m/2 \rceil$ nonempty children.
3. The number of keys in each internal node is one less than the number of its nonempty children, and these keys partition the keys in the children in the fashion of a search tree.
4. The root has at most m children, but may have as few as 2 if it is not a leaf, or none if the tree consists of the root alone.



Insertion into a B-Tree

In contrast to binary search trees, B-trees are not allowed to grow at their leaves; instead, they are forced to grow at the root. General insertion method:

1. Search the tree for the new key. This search (if the key is truly new) will terminate in failure at a leaf.

2. Insert the new key into to the leaf node. If the node was not previously full, then the insertion is finished.

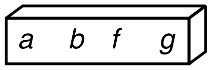
3. When a key is added to a full node, then the node splits into two nodes, side by side on the same level, except that the median key is not put into either of the two new nodes.

4. When a node splits, move up one level, insert the median key into this parent node, and repeat the splitting process if necessary.

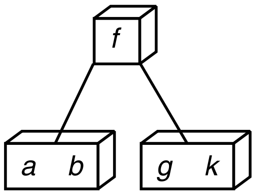
5. When a key is added to a full root, then the root splits in two and the median key sent upward becomes a new root. This is the only time when the B-tree grows in height.

Growth of a B-Tree

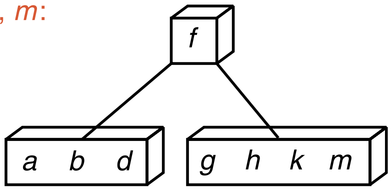
1. *a, g, f, b*:



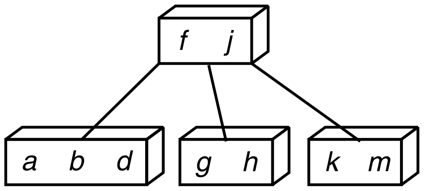
2. *k*:



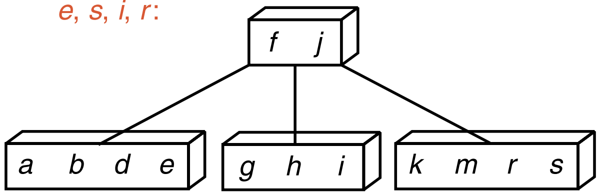
3. *d, h, m*:



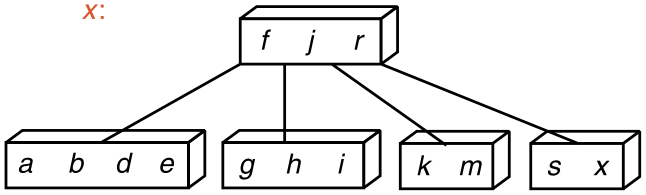
4. *j*:



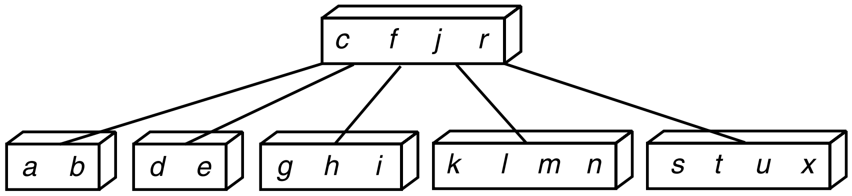
5. *e, s, i, r*:



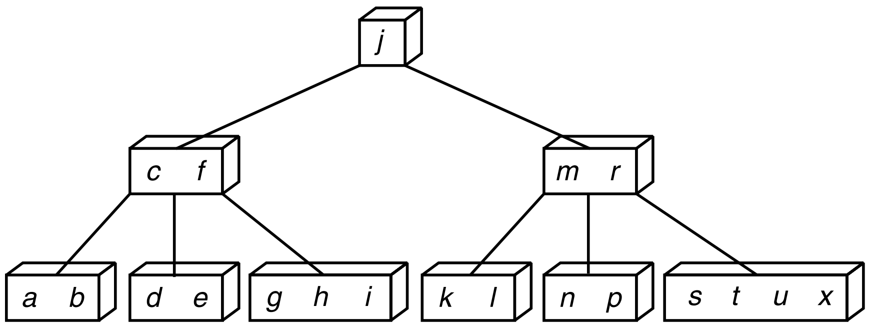
6. *x*:



7. *c, l, n, t, u*:



8. *p*:



B-Tree Declarations in C++

We add the order as a second template parameter. For example, `B_tree<int, 5> sample_tree;` declares sample_tree as a B_tree of order 5 that holds integer records.

B-tree class declaration:

```
template <class Record, int order>
class B_tree {
public:                          //  Add public methods.
private:                         //  data members
   B_node<Record, order> *root;
                                 //  Add private auxiliary functions here.
};
```

Node declaration:

```
template <class Record, int order>
struct B_node {
//  data members:
   int count;
   Record data[order − 1];
   B_node<Record, order> *branch[order];
//  constructor:
   B_node( );
};
```

Conventions:

- count gives the number of records in the B_node.
- If count is nonzero then the node has count + 1 children.
- branch[0] points to the subtree containing all records with keys less than that in data[0].
- For $1 \le$ position $\le$ count $-$ 1, branch[position] points to the subtree with keys strictly between those in the subtrees pointed to by data[position − 1] and data[position].
- branch[count] points to the subtree with keys greater than that of data[count − 1].

Searching in a B-Tree

Public method:

```
template <class Record, int order>
Error_code B_tree<Record, order> :: search_tree(Record &target)
/* Post: If there is an entry in the B-tree whose key matches that in target, the pa-
         rameter target is replaced by the corresponding Record from the B-tree
         and a code of success is returned. Otherwise a code of not_present
         is returned.
   Uses: recursive_search_tree */
{
  return recursive_search_tree(root, target);
}
```

Recursive function:

```
template <class Record, int order>
Error_code B_tree<Record, order> :: recursive_search_tree(
            B_node<Record, order> *current, Record &target)
/* Pre:  current is either NULL or points to a subtree of the B_tree.
   Post: If the Key of target is not in the subtree, a code of not_present is
         returned. Otherwise, a code of success is returned and target is set to
         the corresponding Record of the subtree.
   Uses: recursive_search_tree recursively and search_node */

{
  Error_code result = not_present;
  int position;
  if (current != NULL) {
    result = search_node(current, target, position);
    if (result == not_present)
      result =
        recursive_search_tree(current->branch[position], target);
    else
      target = current->data[position];
  }
  return result;
}
```

- This function has been written recursively to exhibit the similarity of its structure to that of the insertion function.
- The recursion is tail recursion and can easily be replaced by iteration.

Searching a Node

This function determines if the target is present in the current node, and, if not, finds which of the count + 1 branches will contain the target key.

```
template <class Record, int order>
Error_code B_tree<Record, order>::search_node(
   B_node<Record, order> *current, const Record &target, int &position)
/* Pre:  current points to a node of a B_tree.
   Post: If the Key of target is found in *current, then a code of success is returned,
         the parameter position is set to the index of target, and the corresponding
         Record is copied to target. Otherwise, a code of not_present is returned,
         and position is set to the branch index on which to continue the search.
   Uses: Methods of class Record. */

{
   position = 0;
   while (position < current->count && target > current->data[position])
      position++;                    //  Perform a sequential search through the keys.
   if (position < current->count && target == current->data[position])
      return success;
   else
      return not_present;
}
```

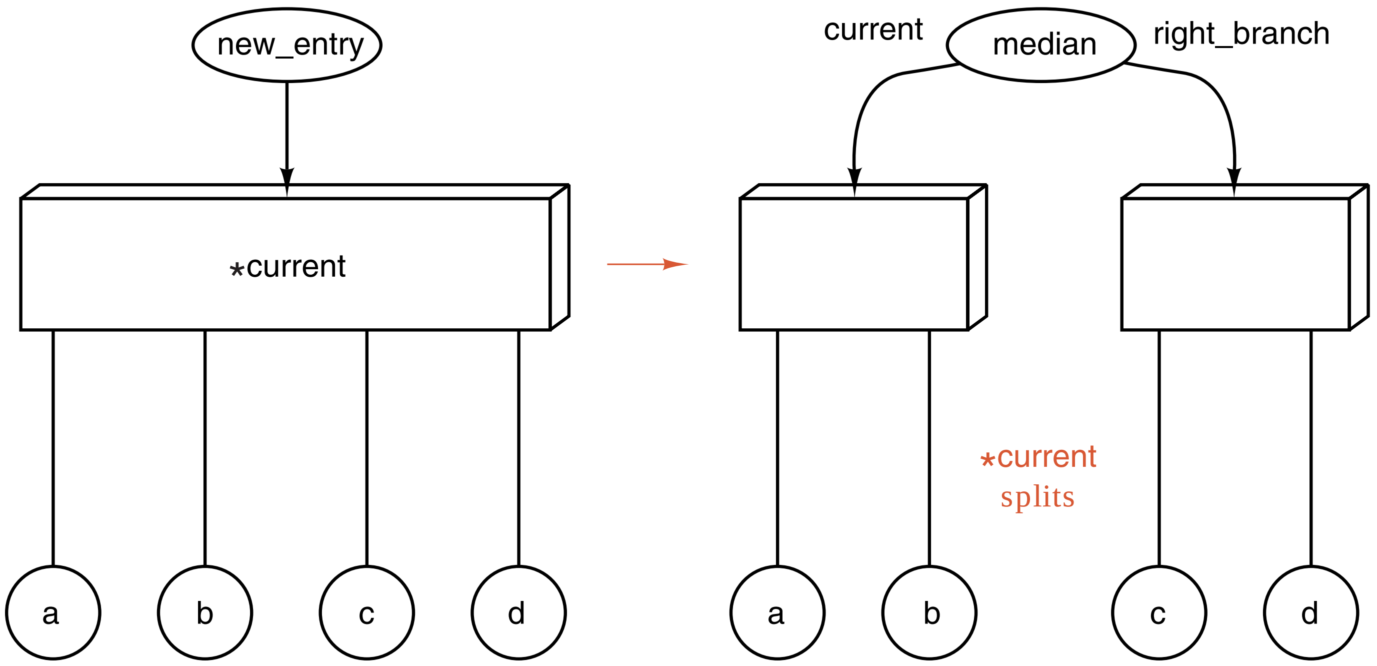
- For B-trees of large order, this function should be modified to use binary search instead of sequential search.
- Another possibility is to use a linked binary search tree instead of a sequential array of entries for each node.

Insertion: Parameters and push_down

- Insertion is done with recursion in a function called push_down.
- We require that the record new_entry being inserted is not already present in the tree.

The recursive function push_down uses three more output parameters.

- current is the root of the current subtree under consideration.
- If *current splits to accommodate new_entry, push_down returns a code of overflow, and the following come into use:
- The old node *current contains the left half of the entries.
- median gives the median record.
- right_branch points to a new node containing the right half of the former *current.



Public Insertion Method

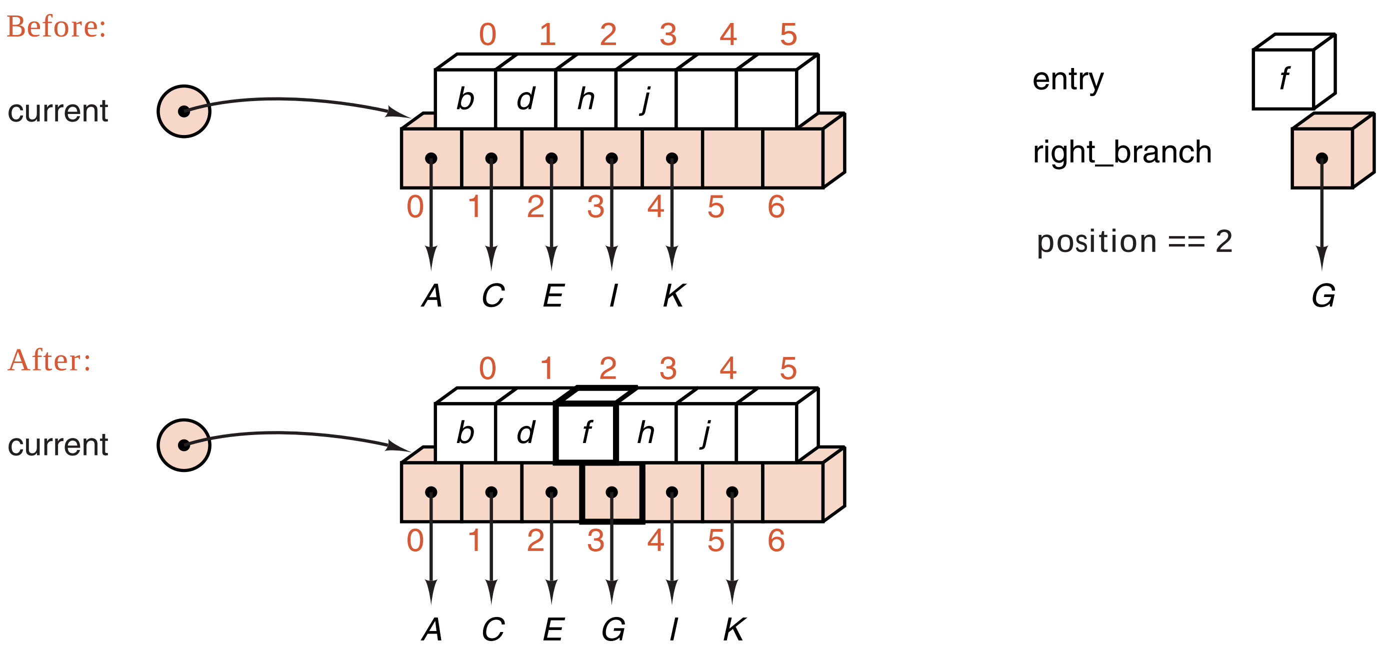
```
template <class Record, int order>
Error_code B_tree<Record, order>::insert(const Record &new_entry)
/* Post: If the Key of new_entry is already in the B_tree, a code of duplicate_error
         is returned. Otherwise, a code of success is returned and the Record
         new_entry is inserted into the B-tree in such a way that the properties of a
         B-tree are preserved.
   Uses: Methods of struct B_node and the auxiliary function push_down. */

{
   Record median;
   B_node<Record, order> *right_branch, *new_root;
   Error_code result =
      push_down(root, new_entry, median, right_branch);

   if (result == overflow) {        //   The whole tree grows in height.
                                    //   Make a brand new root for the whole B-tree.
      new_root = new B_node<Record, order>;
      new_root->count = 1;
      new_root->data[0] = median;
      new_root->branch[0] = root;
      new_root->branch[1] = right_branch;
      root = new_root;
      result = success;
   }
   return result;
}
```

Recursive Insertion into a Subtree

```
template <class Record, int order>
Error_code B_tree<Record, order> :: push_down(
                        B_node<Record, order> *current,
                        const Record &new_entry,
                        Record &median,
                        B_node<Record, order> * &right_branch)
/* Pre:   current is either NULL or points to a node of a B_tree.
   Post:  If an entry with a Key matching that of new_entry is in the subtree to which cur-
          rent points, a code of duplicate_error is returned. Otherwise, new_entry
          is inserted into the subtree: If this causes the height of the subtree to grow,
          a code of overflow is returned, and the Record median is extracted to be
          reinserted higher in the B-tree, together with the subtree right_branch on its
          right. If the height does not grow, a code of success is returned.
   Uses:  Functions push_down (called recursively), search_node, split_node, and
          push_in.  */

{
   Error_code result;
   int position;
   if (current == NULL) {
                    //   Since we cannot insert in an empty tree, the recursion terminates.
      median = new_entry;
      right_branch = NULL;
      result = overflow;
   }
```

Recursive Insertion, Continued

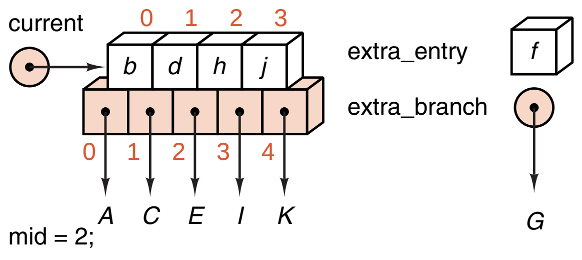
```
else {                                   //   Search the current node.
   if (search_node(current, new_entry, position) ==  success)
      result = duplicate_error;
   else {
      Record extra_entry;
      B_node<Record, order> *extra_branch;
      result = push_down(current->branch[position], new_entry,
                                  extra_entry, extra_branch);

      if (result ==  overflow) {
                              //   Record extra_entry now must be added to current
         if (current->count < order − 1) {
            result = success;
            push_in(current, extra_entry, extra_branch, position);
         }

         else split_node(     current, extra_entry, extra_branch, position,
                              right_branch, median);
         //   Record median and its right_branch will go up to a higher node.
      }
   }
}
return result;
}
```



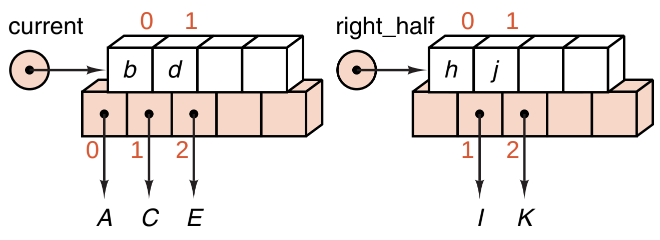
```
template <class Record, int order>
void B_tree<Record, order> ::
    push_in(B_node<Record, order> *current,
            const Record &entry,
            B_node<Record, order> *right_branch,
            int position)
/* Pre:  current points to a node of a B_tree. The node *current is not full and entry
         belongs in *current at index position.
   Post: entry has been inserted along with its right-hand branch right_branch into
         *current at index position. */
{
  for (int i = current->count; i > position; i--) {
                                      //  Shift all later data to the right.
    current->data[i] = current->data[i − 1];
    current->branch[i + 1] = current->branch[i];
  }
  current->data[position] = entry;
  current->branch[position + 1] = right_branch;
  current->count++;
}
```

Example of Splitting a Full Node

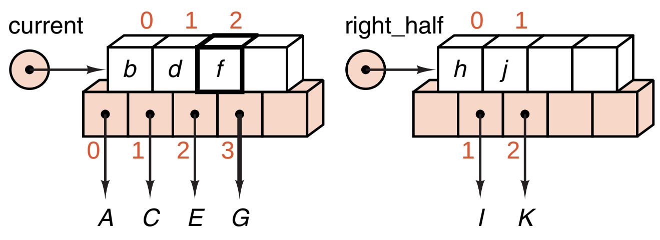
Case 1: position == 2; order == 5;
(extra_entry belongs in left half.)



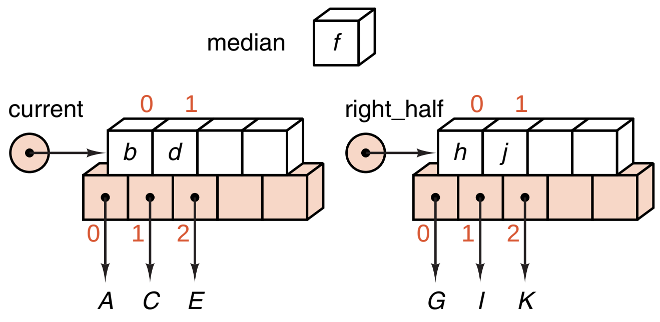
Shift entries right:



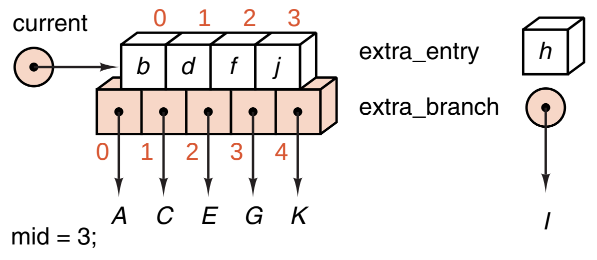
Insert extra_entry and extra_branch:



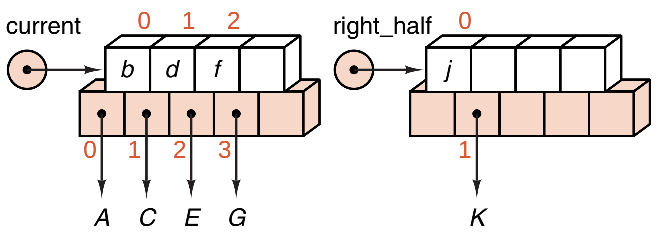
Remove median; move branch:



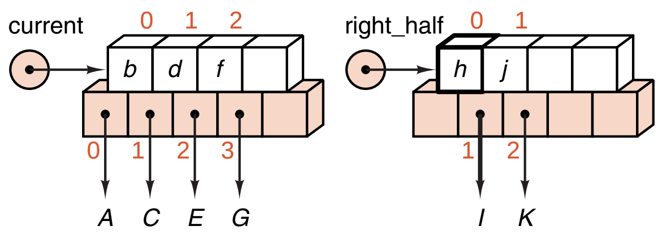
Case 2: position == 3; order == 5;
(extra_entry belongs in right half.)



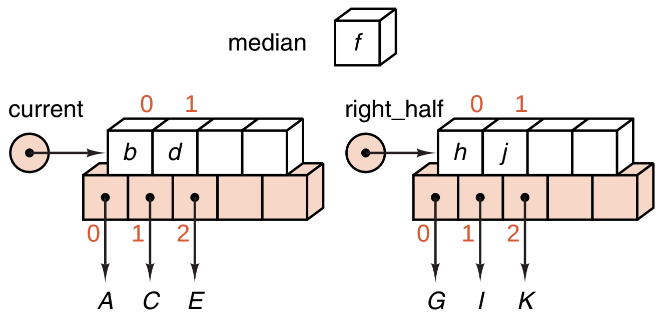
Shift entry right:



Insert extra_entry and extra_branch:



Remove median; move branch:



Function split_node, Specifications

```
template <class Record, int order>
void B_tree<Record, order> :: split_node(
   B_node<Record, order> *current,   //   node to be split
   const Record &extra_entry,   //   new entry to insert
   B_node<Record, order> *extra_branch,
                        //   subtree on right of extra_entry
   int position,        //   index in node where extra_entry goes
   B_node<Record, order> * &right_half,   //   new node for right half of entries
   Record &median)   //   median entry (in neither half)
/* Pre:  current points to a node of a B_tree. The node *current is full, but if there
         were room, the record extra_entry with its right-hand pointer extra_branch
         would belong in *current at position position, 0 ≤ position < order.
   Post: The node *current with extra_entry and pointer extra_branch at position
         position are divided into nodes *current and *right_half separated by a
         Record median.
   Uses: Methods of struct B_node, function push_in. */
```

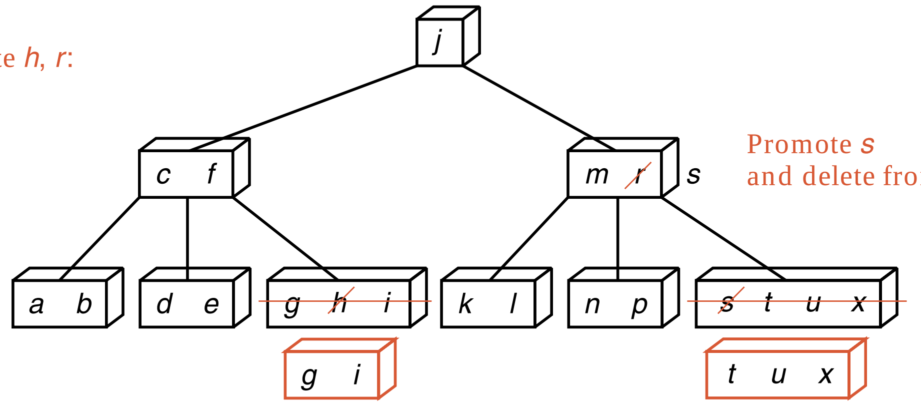
Function split_node, Action

```
{
   right_half = new B_node<Record, order>;
   int mid = order/2;   //   The entries from mid on will go to right_half.

   if (position <= mid) {   //   First case: extra_entry belongs in left half.
      for (int i = mid; i < order − 1; i++) {   //   Move entries to right_half.
         right_half->data[i − mid] = current->data[i];
         right_half->branch[i + 1 − mid] = current->branch[i + 1];
      }
      current->count = mid;
      right_half->count = order − 1 − mid;
      push_in(current, extra_entry, extra_branch, position);
   }

   else {             //   Second case: extra_entry belongs in right half.
      mid++;          //   Temporarily leave the median in left half.
      for (int i = mid; i < order − 1; i++) {   //   Move entries to right_half.
         right_half->data[i − mid] = current->data[i];
         right_half->branch[i + 1 − mid] = current->branch[i + 1];
      }
      current->count = mid;
      right_half->count = order − 1 − mid;
      push_in(right_half, extra_entry, extra_branch, position − mid);
   }

      median = current->data[current->count − 1];
                      //   Remove median from left half.
      right_half->branch[0] = current->branch[current->count];
      current->count−−;
}
```

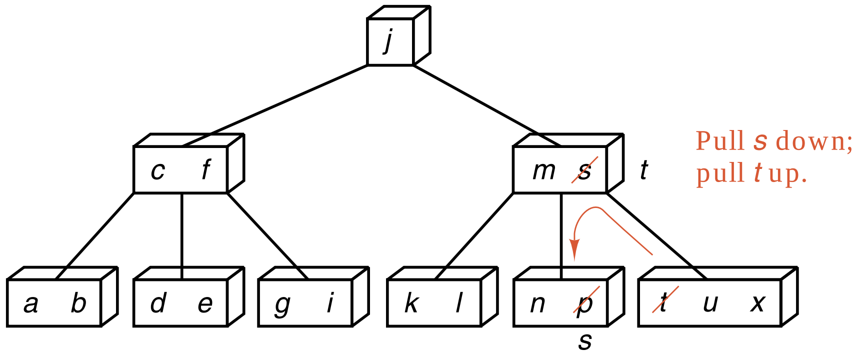
Deletion from a B-Tree

- If the entry that is to be deleted is not in a leaf, then its immediate predecessor (or successor) under the natural order of keys is guaranteed to be in a leaf.
- We promote the immediate predecessor (or successor) into the position occupied by the deleted entry, and delete the entry from the leaf.
- If the leaf contains more than the minimum number of entries, then one of them can be deleted with no further action.
- If the leaf contains the minimum number, then we first look at the two leaves (or, in the case of a node on the outside, one leaf) that are immediately adjacent to each other and are children of the same node. If one of these has more than the minimum number of entries, then one of them can be moved into the parent node, and the entry from the parent moved into the leaf where the deletion is occurring.
- If the adjacent leaf has only the minimum number of entries, then the two leaves and the median entry from the parent can all be combined as one new leaf, which will contain no more than the maximum number of entries allowed.
- If this step leaves the parent node with too few entries, then the process propagates upward. In the limiting case, the last entry is removed from the root, and then the height of the tree decreases.

1. Delete *h*, *r*:

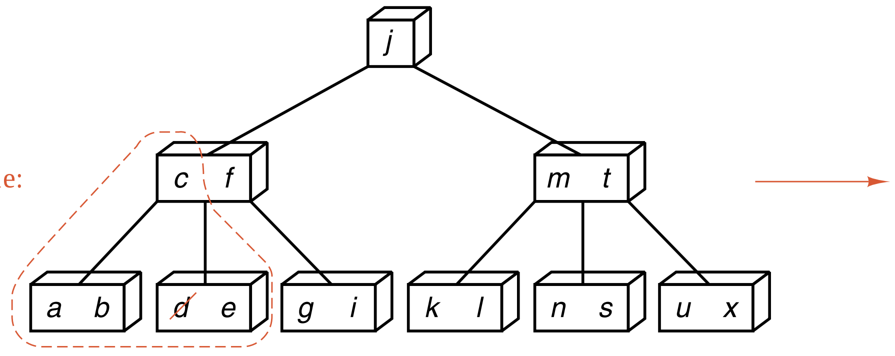


2. Delete *p*:

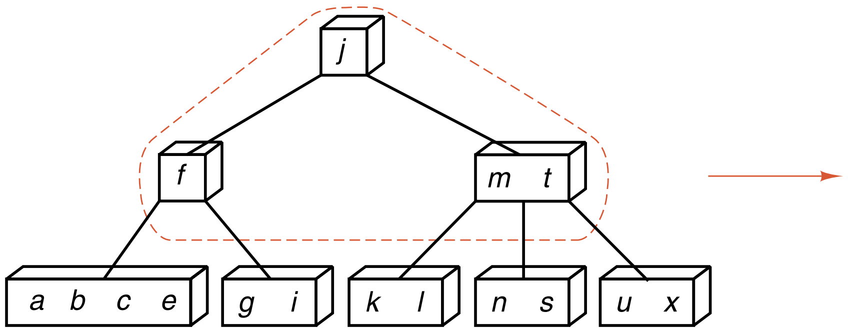


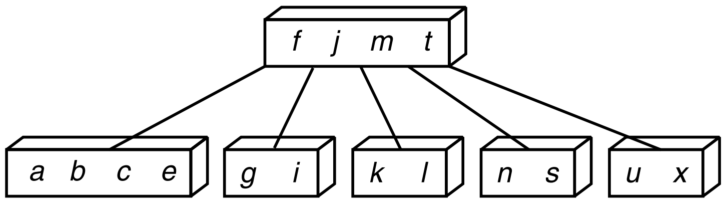
3. Delete *d*:

Combine:



Combine:





Public Deletion Method

```
template <class Record, int order>
Error_code B_tree<Record, order> :: remove(const Record &target)
/* Post: If a Record with Key matching that of target belongs to the B_tree, a code of
         success is returned and the corresponding node is removed from the B-tree.
         Otherwise, a code of not_present is returned.
   Uses: Function recursive_remove */
{
   Error_code result;
   result = recursive_remove(root, target);
   if (root != NULL && root->count == 0) {  //   root is now empty.
      B_node<Record, order> *old_root = root;
      root = root->branch[0];
      delete old_root;
   }
   return result;
}
```

Recursive Deletion

```
template <class Record, int order>
Error_code B_tree<Record, order> :: recursive_remove(
   B_node<Record, order> *current, const Record &target)
/* Pre:   current is either NULL or points to the root node of a subtree of a B_tree.
   Post:  If a Record with Key matching that of target belongs to the subtree, a
          code of success is returned and the corresponding node is removed from the
          subtree so that the properties of a B-tree are maintained. Otherwise, a code of
          not_present is returned.
   Uses:  Functions search_node, copy_in_predecessor, recursive_remove (re-
          cursively), remove_data, and restore. */
{ Error_code result;
   int position;
   if (current == NULL) result = not_present;
   else {
      if (search_node(current, target, position) == success) {
                                              //   The target is in the current node.
         result = success;
         if (current->branch[position] != NULL) {  //   not at a leaf node
            copy_in_predecessor(current, position);
            recursive_remove(current->branch[position],
                             current->data[position]);
         }
         else remove_data(current, position);   //   Remove from a leaf node.
      }
      else result = recursive_remove(current->branch[position], target);
      if (current->branch[position] != NULL)
         if (current->branch[position]->count < (order − 1)/2)
            restore(current, position);
   }
   return result;
}
```

Auxiliary Functions

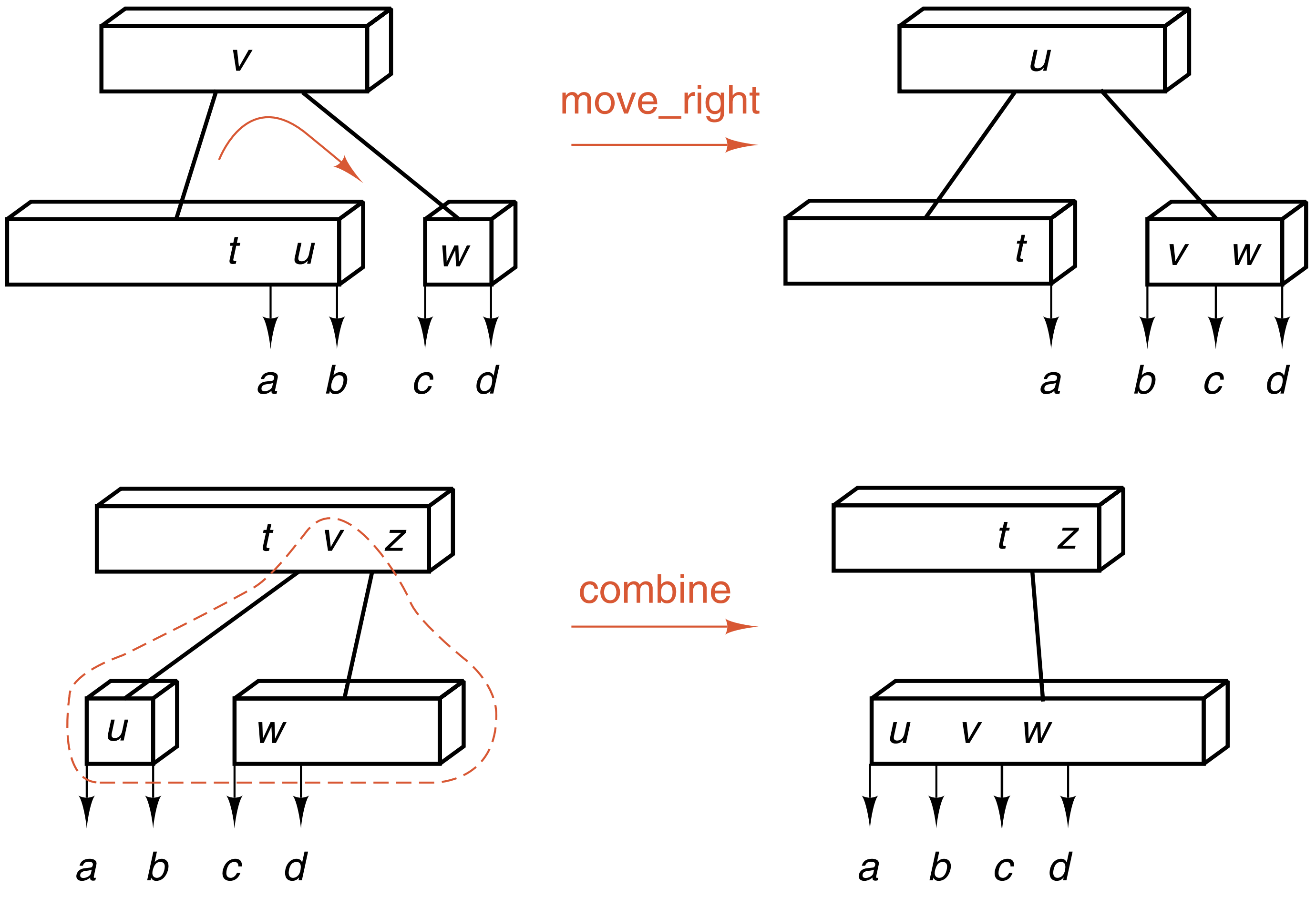
Remove data from a leaf:

```
template <class Record, int order>
void B_tree<Record, order>::
          remove_data(B_node<Record, order> *current,
                          int position)
/* Pre:  current points to a leaf node in a B-tree with an entry at position.
   Post: This entry is removed from *current.  */
{
   for (int i = position; i < current->count − 1; i++)
      current->data[i] = current->data[i + 1];
   current->count−−;
}
```

Replace data by its immediate predecessor:

```
template <class Record, int order>
void B_tree < Record, order > ::copy_in_predecessor(
                        B_node<Record, order> *current, int position)
/* Pre:  current points to a non-leaf node in a B-tree with an entry at position.
   Post: This entry is replaced by its immediate predecessor under order of keys.
         */
{
   B_node<Record, order> *leaf = current->branch[position];
                                   //  First go left from the current entry.
   while (leaf->branch[leaf->count] != NULL)
      leaf = leaf->branch[leaf->count];
                                   //  Move as far rightward as possible.
   current->data[position] = leaf->data[leaf->count − 1];
}
```

Restore Minimum Number of Entries



Function to Restore Minimum Node Entries

```
template <class Record, int order>
void B_tree<Record, order> ::
   restore(B_node<Record, order> *current, int position)
/* Pre:   current points to a non-leaf node in a B-tree; the node to which
          current->branch[position] points has one too few entries.
   Post:  An entry is taken from elsewhere to restore the minimum number of entries in
          the node to which current->branch[position] points.
   Uses: move_left, move_right, combine.  */

{
   if (position == current->count)  //    case: rightmost branch
      if (current->branch[position − 1]->count > (order − 1)/2)
         move_right(current, position − 1);
      else
         combine(current, position);

   else if (position == 0)             //    case: leftmost branch
      if (current->branch[1]->count > (order − 1)/2)
         move_left(current, 1);
      else
         combine(current, 1);

   else                                //    remaining cases: intermediate branches
      if (current->branch[position − 1]->count > (order − 1)/2)
         move_right(current, position − 1);
      else if (current->branch[position + 1]->count > (order − 1)/2)
         move_left(current, position + 1);
      else combine(current, position);
}
```

Function move_left

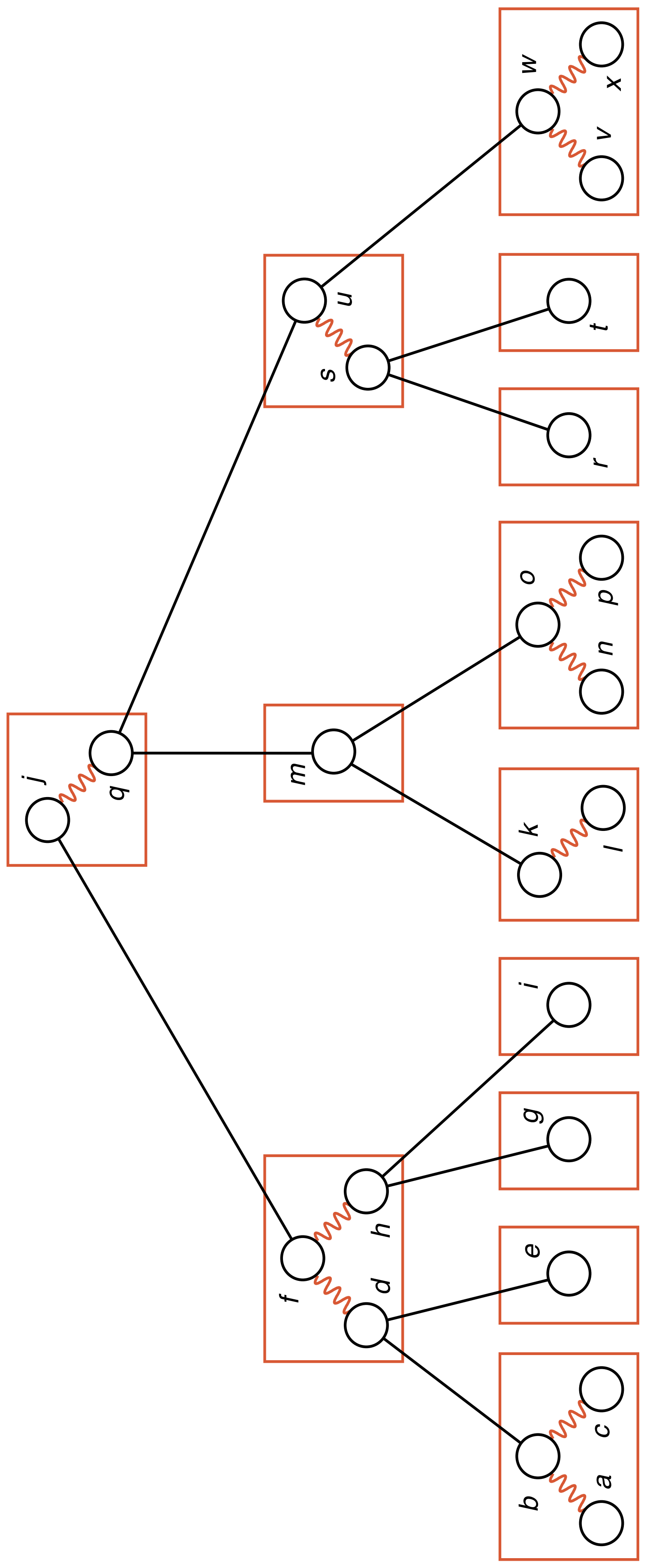
```
template <class Record, int order>
void B_tree<Record, order>::
  move_left(B_node<Record, order> *current, int position)
/* Pre:  current points to a node in a B-tree with more than the minimum number of
         entries in branch position and one too few entries in branch position − 1.
   Post: The leftmost entry from branch position has moved into current, which has
         sent an entry into the branch position − 1. */

{
  B_node<Record, order>
    *left_branch = current->branch[position − 1],
    *right_branch = current->branch[position];

  left_branch->data[left_branch->count] =
    current->data[position − 1];  //  Take entry from the parent.

  left_branch->branch[++left_branch->count] =
    right_branch->branch[0];

  current->data[position − 1] = right_branch->data[0];
                                  //  Add the right-hand entry to the parent.
  right_branch->count−−;

  for (int i = 0; i < right_branch->count; i++) {
                                  //  Move right-hand entries to fill the hole.
    right_branch->data[i] = right_branch->data[i + 1];
    right_branch->branch[i] = right_branch->branch[i + 1];
  }

  right_branch->branch[right_branch->count] =
    right_branch->branch[right_branch->count + 1];
}
```

Function move_right

```
template <class Record, int order>
void B_tree<Record, order>::
  move_right(B_node<Record, order> *current, int position)
/* Pre:   current points to a node in a B-tree with more than the minimum number of
          entries in branch position and one too few entries in branch position + 1.
   Post:  The rightmost entry from branch position has moved into current, which has
          sent an entry into the branch position + 1.  */

{
  B_node<Record, order>
    *right_branch = current->branch[position + 1],
    *left_branch = current->branch[position];

  right_branch->branch[right_branch->count + 1] =
    right_branch->branch[right_branch->count];

  for (int i = right_branch->count ; i > 0; i−−) {
                                        //   Make room for new entry.
    right_branch->data[i] = right_branch->data[i − 1];
    right_branch->branch[i] = right_branch->branch[i − 1];
  }

  right_branch->count++;
  right_branch->data[0] = current->data[position];
                                        //   Take entry from parent.
  right_branch->branch[0] =
    left_branch->branch[left_branch->count−−];
  current->data[position] =
    left_branch->data[left_branch->count];
}
```

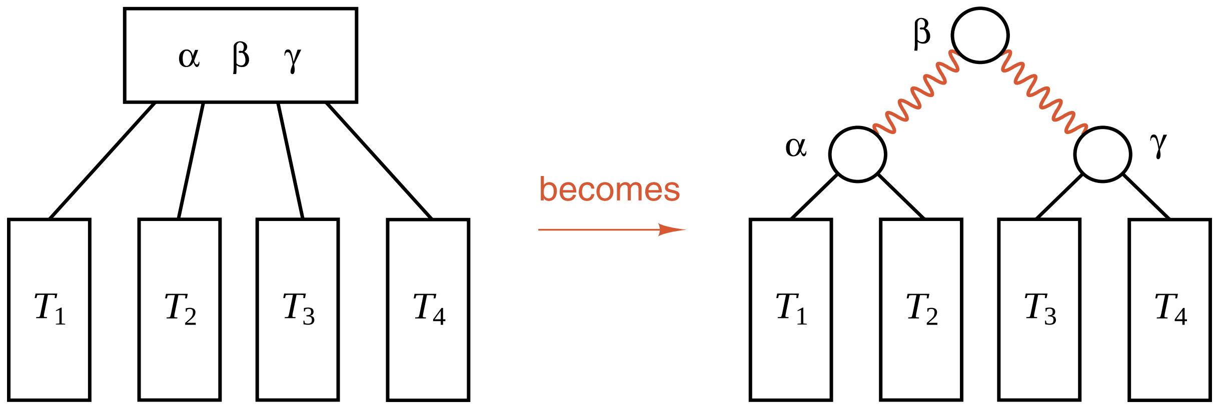
Function combine

```
template <class Record, int order>
void B_tree<Record, order>::
  combine(B_node<Record, order> *current, int position)
/* Pre:  current points to a node in a B-tree with entries in the branches position and
         position − 1, with too few to move entries.
   Post: The nodes at branches position − 1 and position have been combined
         into one node, which also includes the entry formerly in current at index
         position − 1. */
{
  int i;
  B_node<Record, order>
    *left_branch = current->branch[position − 1],
    *right_branch = current->branch[position];
  left_branch->data[left_branch->count] =
    current->data[position − 1];
  left_branch->branch[++left_branch->count] =
    right_branch->branch[0];
  for (i = 0; i < right_branch->count; i++) {
    left_branch->data[left_branch->count] =
      right_branch->data[i];
    left_branch->branch[++left_branch->count] =
      right_branch->branch[i + 1];
  }
  current->count−−;
  for (i = position − 1; i < current->count; i++) {
    current->data[i] = current->data[i + 1];
    current->branch[i + 1] = current->branch[i + 2];
  }
  delete right_branch;
}
```

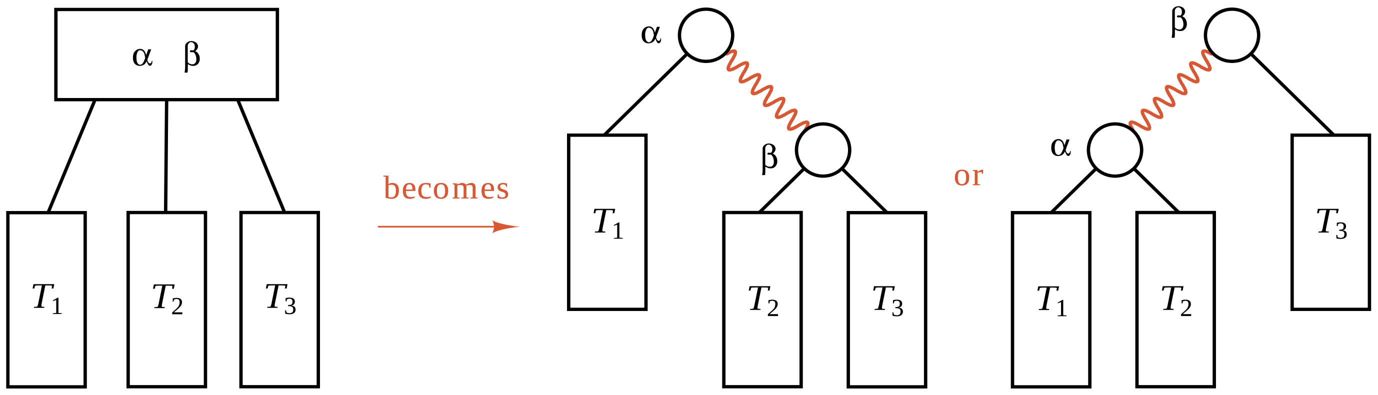

j
q
m
f
d
h
s
u
b
a
c
e
g
i
k
l
n
o
p
r
t
v
w
x

Red-Black Trees as B-Trees of Order 4

- Start with a B-tree of order 4, so each node contains 1, 2, or 3 entries.
- Convert a node with 3 entries into a binary search tree by:



- A node with two entries has two possible conversions:



- A node with one entry remains unchanged.
- 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.
- 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.

Red-Black Trees as Binary Search Trees

- 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.
- 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.

- 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.

- We can now define:

DEFINITION 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.

Analysis of Red-Black Trees

THEOREM 11.2 The height of a red-black tree containing n nodes is no more than $2\lg n$.

- Searching a red-black tree with n nodes is $O(\log n)$ in every case.
- The time for insertion is also $O(\log n)$. [To show this, we first need to devise the insertion algorithm.]
- 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:

```
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; }
};
```

- Note the inline definitions for the constructors and other methods of a red-black node.

Modified Node Specification

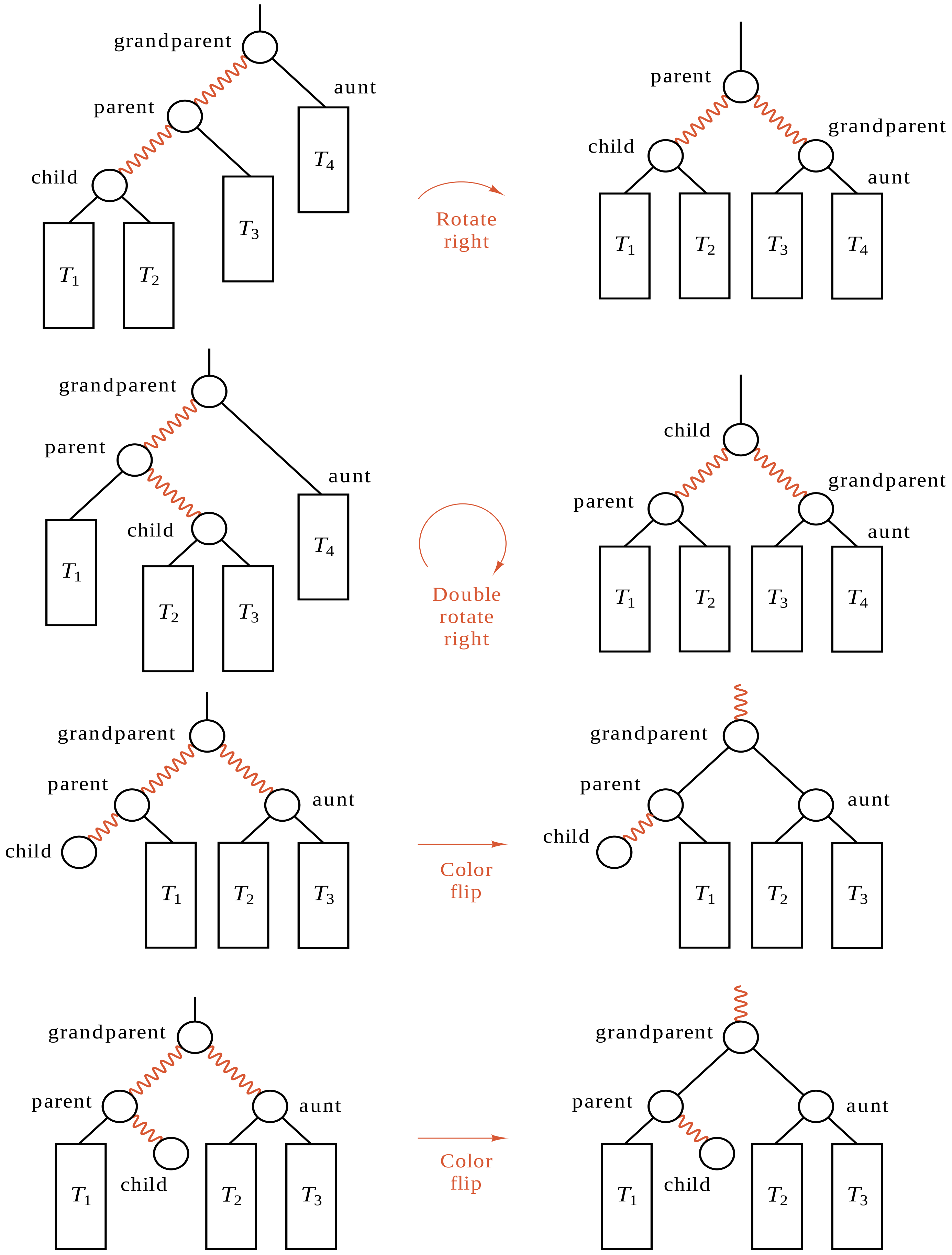
- To invoke get_color and set_color via pointers, we must add virtual functions to the base **struct** Binary_node.
- The modified node specification is:

```
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; }
};
```

- With this modification, we can reuse all the methods and functions for manipulating binary search trees and their nodes.

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.


grandparent
parent
aunt
child
T_1
T_2
T_3
T_4
Rotate right
parent
child
grandparent
aunt
T_1
T_2
T_3
T_4
grandparent
parent
aunt
child
T_1
T_2
T_3
T_4
Double rotate right
child
parent
grandparent
aunt
T_1
T_2
T_3
T_4
grandparent
parent
aunt
child
T_1
T_2
T_3
Color flip
grandparent
parent
aunt
child
T_1
T_2
T_3
grandparent
parent
aunt
child
T_1
T_2
T_3
Color flip
grandparent
parent
aunt
child
T_1
T_2
T_3

Insertion Method Implementation

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 fol-
   lowing 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 black.
         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_entry)
/* 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 mod-
          ify_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.  */
{  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;      //   current 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

1. 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.

2. Most trees can be described easily by using recursion; their associated algorithms are often best formulated recursively.

3. 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. 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.