24

24 Two more "trees"

Computer science students often have great difficulty in explaining to their parents why they are spending so much time studying "trees" and "strings". But I must impose upon you again. There are a couple more trees that you need to study. They are both just more elaborate versions of the binary tree lookup structure illustrated in Section 21.5.

AVL tree

The first, the "AVL" tree, is an "improved" binary tree. The code for AVL deals with some problems that can occur with binary trees which reduce the performance of the lookup structure. AVL trees are used for the same purpose as binary trees; they hold collections of keyed data in main memory and provide facilities for adding data, searching for data associated with a given key, and removing data.

BTree

The second, the "BTree" tree, is intended for data collections that are too large to fit in main memory. You still have data records with a "key" field and other information; it is just that you may have hundreds of thousands of them. A BTree provides a means whereby most of the data are kept in disk files, but a fast search is still practical. The BTree illustrated is only slightly simplified; it is pretty close to the structures that are used to implement lookup systems for many large databases.

These two examples make minor use of "inheritance" as presented in Section 23.6. The AVL tree is to store data items that are instances of some concrete class derived from class `KeyedItem`:

```
class KeyedItem {
public:
    virtual       ~KeyedItem() { }
    virtual long  Key(void) const = 0;
    virtual void  PrintOn(ostream& out) const  { }
};
```

A `KeyedItem` is just some kind of data item that has a unique long integer key associated with it (it can also print itself if asked).

The BTree stores instances of a concrete class derived from class `KeyedStorable-Item`:

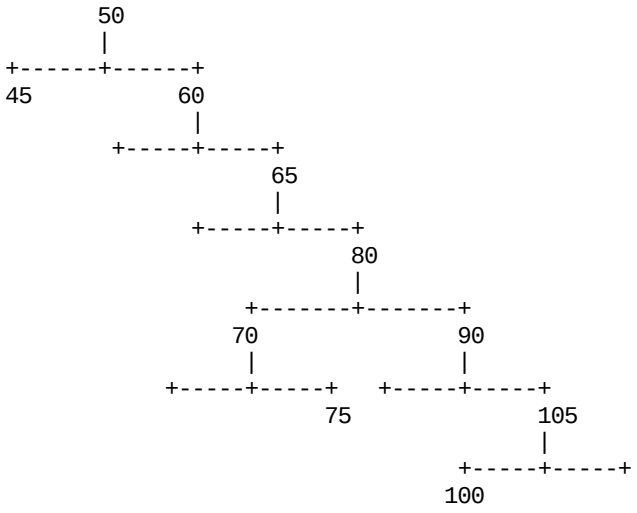
```
class KeyedStorableItem {
public:
    virtual       ~KeyedStorableItem() { }
    virtual long  Key(void) const = 0;
    virtual void  PrintOn(ostream& out) const  { }
    virtual long  DiskSize(void) const = 0;
    virtual void  ReadFrom(fstream& in) = 0;
    virtual void  WriteTo(fstream& out) const = 0;
};
```

These data items must be capable of transferring themselves to/from disk files using binary transfers (`read()` and `write()` calls). On disk, the data items must all use the same amount of space. In addition to any other data that they possess, these items must have a unique long integer key value (disallowing duplicate keys simplifies the code).

24.1 AVL TREES

24.1.1 What's wrong with binary trees?

Take a look at a binary tree after a few "random" insertions. The following tree resulted when keyed data items were inserted with the keys in the following order: 50, 60, 65, 80, 70, 45, 75, 90, 105, 100:



The tree is a little out of balance. Most binary trees that grow inside programs tend to be imbalanced.

This imbalance does matter. A binary search tree is supposed to provide faster lookup of a keyed data item than does an alternative structure like a list. It is supposed

to give O(lgN) performance for searches, insertions, and deletions. But when a tree gets out of balance, performance decreases.

Fast lookup of keyed data items is a very common requirement. So the problems of simple binary search trees become important.

24.1.2 Keeping your balance

You can change the code so that the tree gets rearranged after every insertion and deletion. Figure 24.1 illustrates a couple of rearrangements that could be used to keep the tree balanced as those data items were inserted.

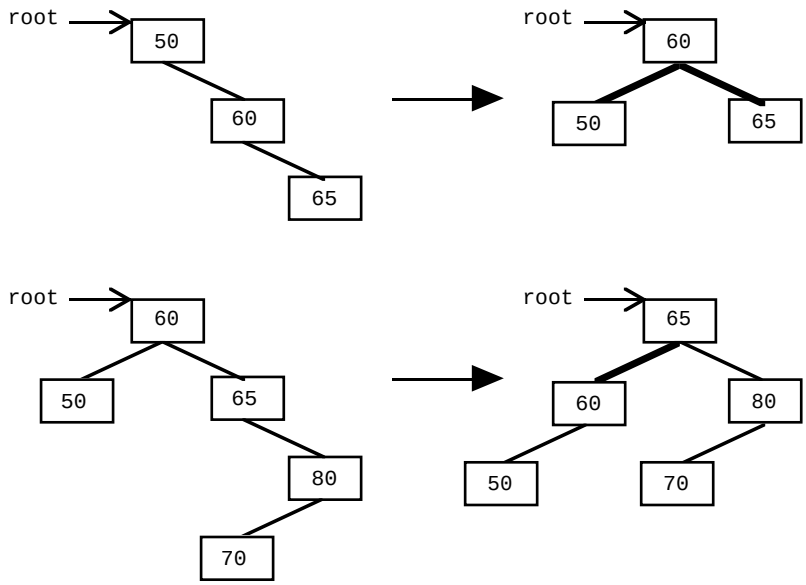


Figure 24.1 Rearranging a tree to maintain perfect balance.

Such rearrangements are possible. But a tree may take quite a lot of rearranging to get it balanced after some data item gets inserted. In some cases, you may have to alter almost all of the left- and right- subtree pointers. This makes the cost of rearrangement directly proportional to the number of items in the tree, i.e. O(N) performance.

There is no point in trying to get a perfectly balanced tree if balancing cost are O(N). Although rebalancing does keep search costs at O(lgN) you are interested in the overall costs, and thus O(N) costs for rebalancing after insertions and deletions count against the O(lg(N)) searches.

However, it has been shown that a tree can be kept "more or less balanced". These more or less balanced trees have search costs that are O(lgN) and the cost of

rebalancing the tree until it is "more or less balanced" is also O(lg(N)). (The analyses of the algorithms to demonstrate these costs is far too difficult for this introductory treatment).

There are many different schemes for keeping a binary search tree "more or less balanced". The best known was invented by a couple of Russians (Adelson-Velskii and Landis) back around 1960. They defined rules to characterize a "more or less balanced tree" and worked out the rearrangements that would be necessary if an insertion operation or a deletion operation destroyed the existing balance. The rearrangements are localized to the "vine" that leads from the root to the point where the change (insertion/ deletion) has just occurred and it is this that keeps the cost of rearrangements down to O(lg(N)). A tree that satisfies their rules is called an AVL tree.

AVL tree

The following definitions together characterize an AVL tree:

- Height of tree (or subtree):
  The height of a binary tree is the length of the longest path from its root to a leaf.

- AVL property:
  A node in a binary tree has the "AVL property" if the heights of the left and right subtrees are either equal or differ by 1.

- AVL tree:
  An AVL tree is a binary tree in which every node has the AVL property.

Figure 24.2 illustrates some trees with examples of both AVL and non-AVL trees.

You can check a tree by starting at the leaf nodes. The "left and right subtrees" of a leaf node don't exist, or from a different perspective they are both size zero. Since they are both size zero they are equal so a leaf node has evenly balanced subtrees.

You climb up from a leaf node to its parent node and check its "left and right subtrees". If the node has two subtrees both with just leaves, then it is even. If it has only one leaf below it, it is either "left long" or "right long".

As you climb further up toward the root, you would need to keep a count of the longest path down through a link to its furthest leaf. As you reached each node, you would have to compare these longest paths down both the left and right subtrees from that node. If the longest paths are equal or differ by at most one, you can label the node as "even", or "left long", or "right long". If the lengths of the paths differ by more than one, as is the case with the some of the nodes in the second pair of trees shown in Figure 24.2, then you have found a situation that violates the AVL requirements.

Checking the "AVL-ness" of an arbitrary tree might involve quite a lot of work. But if you are building up the tree, you can keep track of its state by just having an indicator on each node that says whether it is currently "even", "left long", or "right long". This information is sufficient to let you work out what rearrangements might be needed to maintain balance after a change like the addition or deletion of a node.

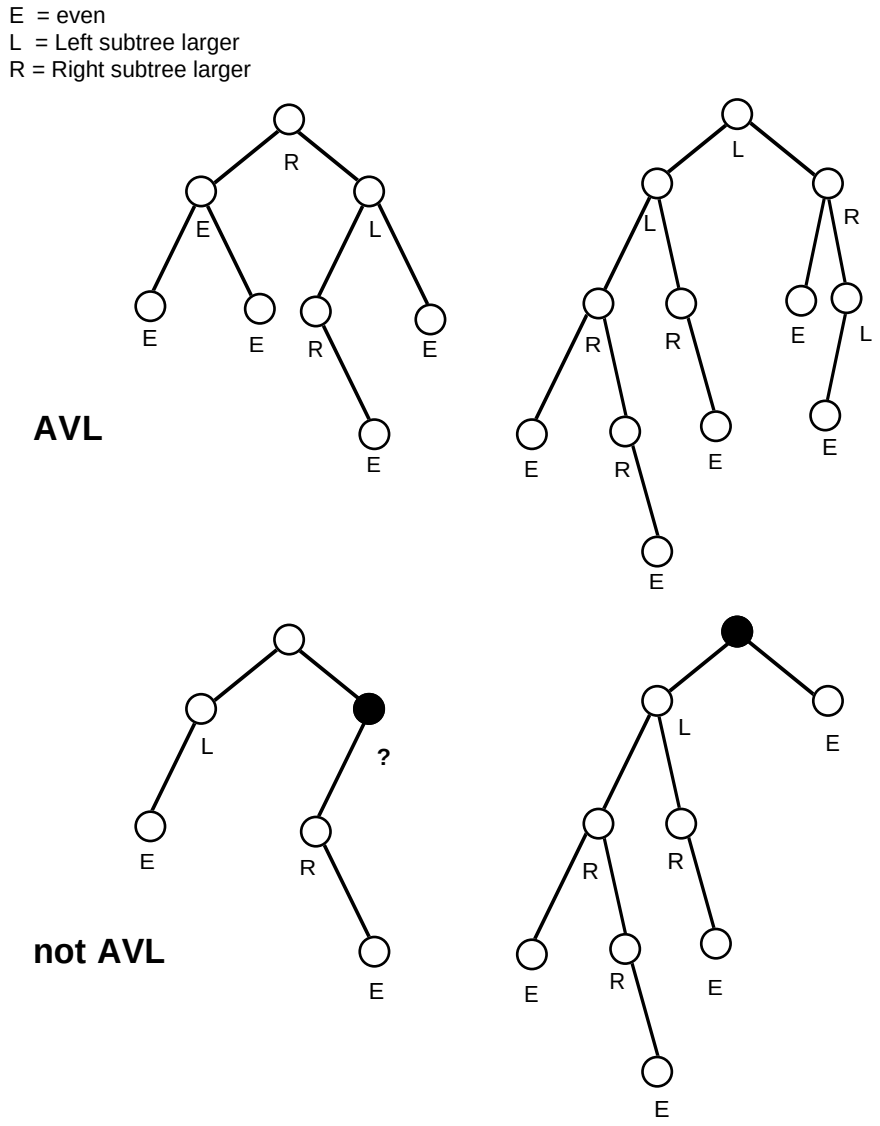


Figure 24.2 Example trees: AVL and not quite AVL.

Adding Nodes to an AVL tree

Figure 24.3 illustrates some of the possible situations that you might encounter when adding an extra node below an existing node. The numbers shown on the nodes

represent the keys for the data items associated with the tree node. Of course it is still essential to have the binary search tree property: data items whose keys are less than the key on a given node will be located in its left subtree, those whose keys are greater will be in its right subtree. (The key values will also be used as "names" for the nodes in subsequent discussions.)

The addition of a node can change the balance at every point on the path that leads from the root to the parent node where the new node gets attached. The first example shown in Figure 24.3 starts with all the existing nodes "even". The new data value must go to the left of the node 27; it was "even" but is going to become "left long". The value 6 has to go to the left of 19; so 19 which was "even" also becomes "left long".

The second example shown, the addition of 21, shows that additions sometimes restore local balance. Node 19 that was "left long" now gets back to "even". Node 27 is still "left long".

The next two examples shown in Figure 24.3 illustrate additions below node 6 that make nodes 27 and 19 both "left too long". They have lost their AVL properties. Some rearrangements are going to be performed to keep the tree "more or less balanced".

The final example shows another case where the tree will require rebalancing. Although this case does not need any changes in the immediate vicinity of the place where the new node gets added, changes are necessary higher up along the path to the root.

Rearrangements to the tree

Adelson-Velskii and Landis explored all the possible situations that could arise when additions were made to a tree. Then, they worked out what local rearrangements could be made to restore "more or less balance" (i.e. the AVL property) in the immediate vicinity of an out of balance node.

The tree has to be reorganized whenever a node becomes "left too long" or "right too long". Obviously, there is a symmetry between the two cases and it is only necessary to consider one; we will examine the situation where a node is "left too long".

As illustrated in Figure 24.4, there are two variations. In one, the left subtree of the "left too long" node is itself "left long"; in the second variation, the node at the start of the left subtree is actually "right long". Adelson-Velskii and Landis sorted out the slightly different rearrangements of the tree structure that would be needed in these two cases. Their proposed local rearrangements are also shown in Figure 24.4.

Local rearrangements to fix up an imbalanced node

The problem that has got to be resolved by these rearrangements is that the left branch of the tree below the current root (node 27) now has a height that is two greater than the right branch. The left branch must shrink, the right branch must grow. Since the tree must be kept ordered, the only way the right branch can grow and the left shrink is to push the current root node down into the right branch, replacing it at the root by an appropriate node from the left branch.

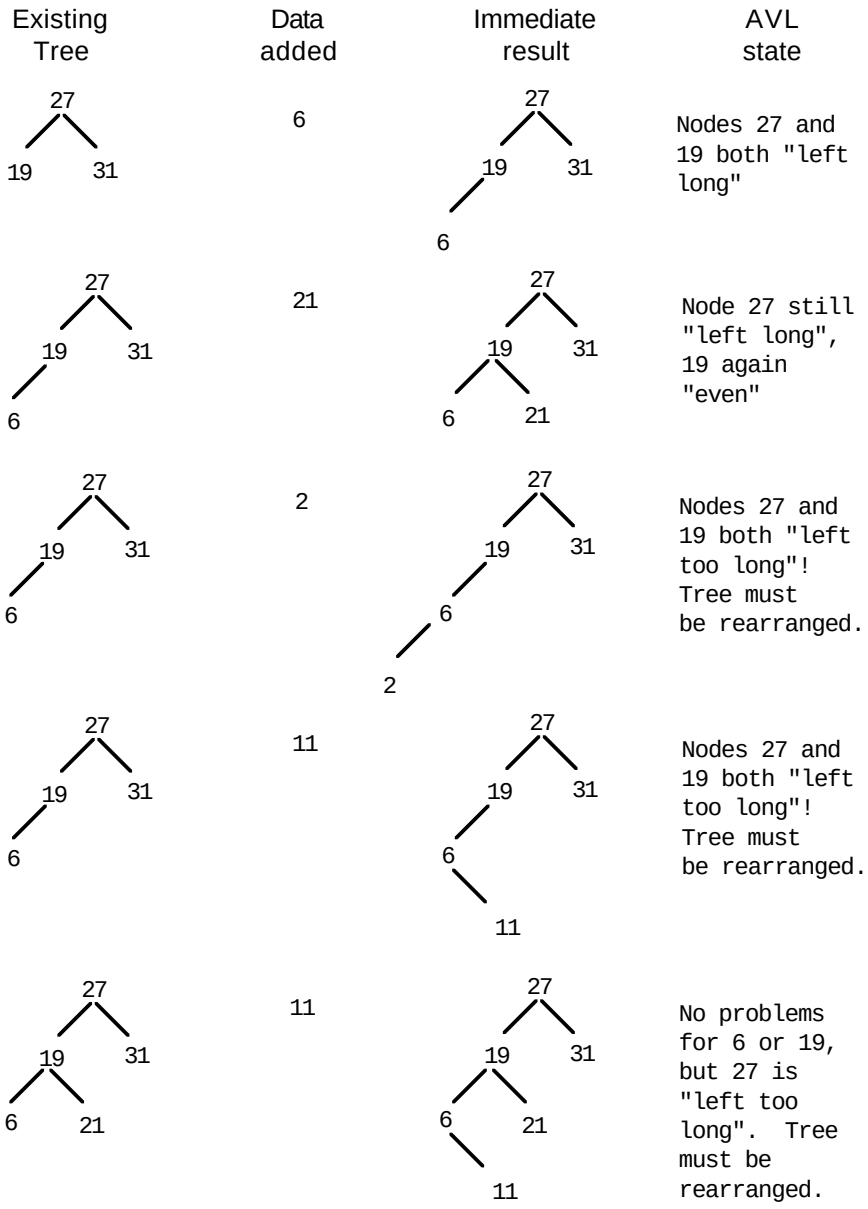


Figure 24.3 Effects of some additions to an AVL tree.

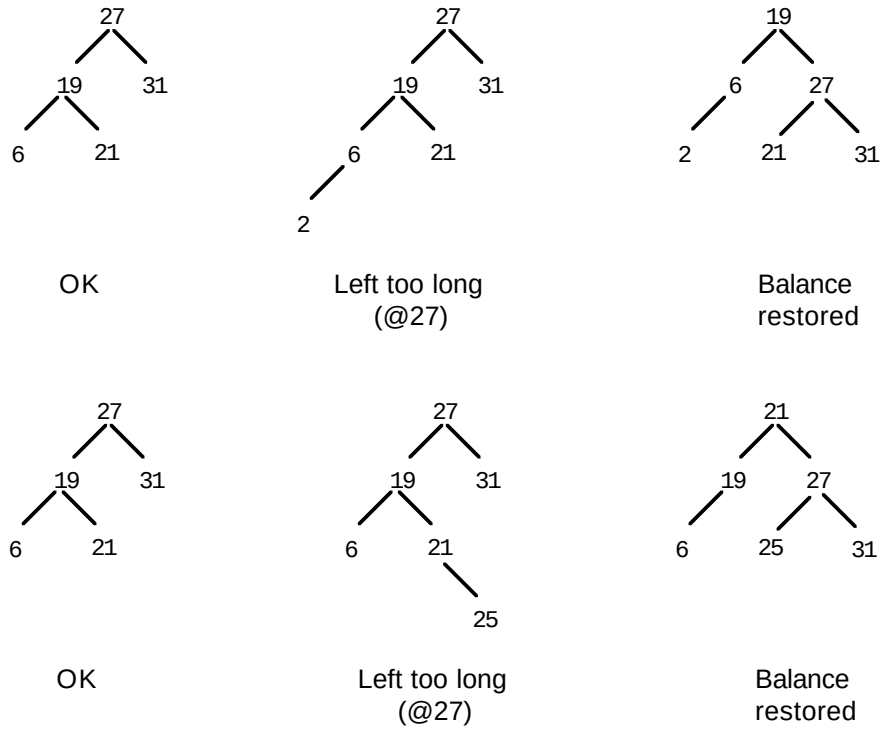


Figure 24.4 Manoeuvres to rebalance a tree after a node is added.

In the first case, the left subtree starting at node 19 has a small right subtree (just node 21). The tree can be rearranged by moving the current root node 27 down into the right tree, rehooking the small tree with node 21 to the left of node 27 and moving node 19 up to the root. The right subtree of the overall tree has grown by one (node 27 pushed into the subtree), and the left subtree has shrunk as node 19 moves upwards pulling its left subtree up with it. The tree is now balanced, all nodes are "even" except node 6 which is "left long". The order of nodes is maintained. Keys less than the new root value, 19, are down the left subtree, keys greater than 19 are in the right tree. Nodes with keys greater than 19 and less than 27 can be found by going first down the right tree from 19 to 27, then down the left tree below node 27.

In the second case, it is the right subtree below node 19 that is too large. This time the rearrangements must shorten this subtree while growing the right branch of the overall tree. Once again, node 27 gets pushed into the right subtree; this time being replaced by its left child's (19) right child (21). Any nodes attached below node 21 must be reattached to the tree. Things in its right subtree (e.g. 25) will have values greater than 21 and less than 27. This right subtree can be reattached as the left subtree of node 27 once this has been moved into position. The left subtree below node 21

(there is none in the example shown) would have nodes whose keys were less than 21 and greater than 19. This left subtree (if any) should be reattached as the right subtree below node 19 after node 21 is detached.

The tree is of course defined by pointer data members in the tree nodes. Rearrangements of the tree involve switching these pointers around. The principles are defined in the following algorithm outline. At the start, the pointer `t` is supposed to hold the address of the node that has got out of balance (node 27 in the example). This pointer `t` could be the "root pointer" for the entire tree; more commonly it will be either the "left subtree" pointer or the "right subtree" pointer of some other tree node. The value in this pointer gets changed because a subtree (if not the entire tree) is getting "re-rooted".

```
tLeft = t->left_subtree          // pointer to node 19
if(tLeft ->balance is "left_long")
    // attach subtree starting at node 21 as left subtree of
    // node 27
    t->"left subtree link"  =
           tLeft->"right subtree link"

    // attach subtree starting at node 27 as right subtree of
    // node 19
    tLeft->"right subtree link" = t;

    nark nodes referenced by t and tLeft as both now "even"

    // make node with 19 the new root
    change pointer t to refer to old left node

else
    // get pointer to node 21
    tLeftRight = tLeft->right_subtree /

    // left from 21, here NULL, gets attached to right of
    // 19
    tLeft->right_subtree = tLeftRight->left_subtree

    // make subtree starting at 19 as the left subtree of 21
    tLeftRight->left_subtree = tLeft;

    // make 21's right subtree (25), the new left subtree of 27
    t->left_subtree = tLeftRight->right_subtree

    // make subtree starting at 27 the new right subtree of 21
    tLeftRight->right_subtree= t;

    // Fix up balance records on nodes
    t->balance = (tLeftRight->balance == LEFT_LONG) ?
                          RIGHT_LONG : EVEN;
    tLeft->balance = (tLeftRight->balance == RIGHT_LONG) ?
                          LEFT_LONG : EVEN;
```

First case shown in Figure 24.4

Second case shown if Figure 24.4

```
        tLeftRight->balance = EVEN

        // Re-root the subtree, now starts with node 21
        t = tLeftRight
```

Organizing the overall process

The addition of a new node below an existing node may make that node "left long" or "right long" by changing the tree height. (When a node has one leaf below it, the addition of the other possible leaf does not change the tree's height, it simply puts the node back into even balance.) If the tree's height changes, this may necessitate rearrangement at the next level above where a node may have become "left too long" (or "right too long"). But it is possible that the change of height in one branch of a tree only produces an imbalance several levels higher up. For example, in the last example shown in Figure 24.3, the addition of node 11 did not cause problems at node 6, or at node 19, but did cause node 27 to become unbalanced.

The mechanism used to handle an insertion must keep track of the path from root to the point where a new node gets attached. Then after a new node is created, its data are filled in, and it gets attached, the process must work back up the path checking each node for imbalance, and performing the appropriate rebalancing rituals where necessary.

Recursive driver function

The process of recording the path and then unwinding and checking the nodes is most easily handled using a recursive routine. It starts like the recursive insertion function shown for the simpler binary tree; the key for the new item, is compared with that in the current node and either the left or right branch is followed in the next recursive call. When there is no subtree, you have found the point where the new node is to be attached, so you build the node and hook it in. The difference from the simple recursive insertion function is that there is a lot of checking code that reexamines "balance" when the recursive call returns.

The algorithm is:

```
insert(dataitem, link)
    if (the link is null)
            create a new node, fill in data
            set the link to point to the new node
            set a flag saying that the tree has grown larger
            return

    currentnode is tree node referenced by link

    if (the key value in the currentnode
            equals the key for the dataitem)
                    report an error, duplicate keys are not allowed
                    set flag to say that the tree is unchanged
                    return

    if (the key value in the currentnode
            is smaller than the key for the dataitem)
```

Terminate recursion when have position to attach new node
Notify caller that tree has grown

```
                recursively call insert, passing the dataitem          Recursive call down
                      and the left-link of currentnode                 into appropriate
                                                                       subtree
                if (tree size is now reported as changed)              Fixup if tree has
                      if current node was even,                        grown taller
                            mark it left long
                            leave "tree changed" flag set
                      if current node was right long,
                            mark it as even
                            clear "tree changed" flag
                      if current node was already left long            Do a local
                            do a local rearrangement                   rearrangement if
                            clear "tree changed" flag                  necessary

                return

        similar code for insertion in right subtree
```

Deletion of nodes

Deletions of nodes present two problems.

Deletion with promotion

The first problem is identical to that encountered with the simple binary trees; it is easy to unhook a leaf node, or excise a node that only has one subtree, but you can't simply cut out a node that has two subtrees attached. The solution is identical to that used for the simpler binary tree. If a tree node has two subtrees, it is kept in the tree; its associated data are removed and replaced by data promoted from a node lower in the tree. The promoted data will be that with the largest key value smaller than the key of the data being removed (the "predecessor" of the deleted data). The node from which data was promoted is then removed from the tree.

Fixing up balance after a deletion

The extra problem is of course that deletions may leave a node unbalanced. After a deletion is done, it is necessary to check back along the path to the root verifying that nodes are still balanced and performing any local rearrangement that might be needed for a node that has become unbalanced.

Recursive driver function

Naturally, the process is handled recursively. During the "inward" phase of recursion, the recursive function gets down to the node that must be removed (either the node with the deleted data, or the node from which data have been promoted). This inward recursion has function calls for each level, the local variables in the stack frame for each call define the various nodes traversed from the "root" to the node that is to be removed. Once found, the node can be excised and the tree marked as having its size changed.

As the recursion is unwound, each node on the path gets its chance to consider the effects of the change in tree size on its balance. Sometimes, nodes will find themselves out of balance, and then there must be local rearrangements to the tree.

Code structure

Because deletion is a fairly complex process, it has to be broken down into many separate functions. The basic driver function will be based on the following algorithm.

The driver function will be given the key for the data item that is to be removed, and the root pointer. It involves a recursive search down through the tree. On each recursive call the argument `t` will be either the left or right subtree link from one of the tree nodes traversed.

```
delete(bad_key, t)
    if(t is NULL)
            set flags to say tree not changed
            return

    if(bad_key equals key in node referenced by t)
            DeleteRec(t);
            return

    if(bad_key < t->Key()) {
            Delete(bad_key,t->LeftLink())

            if(fResizing == CHANGED_SIZE)
                    Check_balance_after_Left_Delete(t);
            return

    Similar code dealing with right subtree
```

Hit null pointer, key wasn't present

If find key, use auxiliary function to do the delete

Otherwise recursively search down in left subtree

On unwind of recursion, do rebalancing

Or search down right subtree

The auxiliary `DeleteRec()` function can sort out simple cases like a leaf node or a node with only one child, but will have to use other auxiliary functions to deal with more complex situations where data have to be replaced with information "promoted" from lower in the tree. Dealing with a node that has one child is simple, the link that lead to the node that is to be deleted is reset to point to the child. (A leaf, no children, doesn't have to be treated as a special case; the code handling nodes with one child also covers the case of no children.)

```
DeleteRec(t)
    if(t->RightLink() is NULL)
            x = t
            t = t->LeftLink()
            fResizing = CHANGED_SIZE
            delete x;
    else
    if(t->LeftLink() is NULL)
            similar

    else
            Del(t,t->LeftLink())

            if(fResizing == CHANGED_SIZE)
                    Check_balance_after_Left_Delete(t);
```

Replace node with only one child by its sole child

Use auxiliary function to promote data from left subtree

Left subtree may have shrunk, fix up

The auxiliary function `Del()` is given a pointer to the node that is being changed and, in the initial call, a pointer to the node's left subtree. It has to find the replacement

data that are to be promoted. The data will be that associated with the largest entry in this left subtree, i.e. the rightmost entry in the subtree. Naturally, `Del()` starts by recursively searching down to find the necessary data.

```
Del(t, r)
    if(r->RightLink() is not NULL)
           Del(t,r->RightLink())
           if(fResizing equals CHANGED_SIZE)
                  Check_balance_after_Right_Delete(r);
    else {
           t->Replace(r->Data())

           // unlink the node from which data have been
           // promoted, replacing it by its left subtree
           // (if any exists)
           x = r
           r = r->LeftLink()

           // note tree size as changed
           // and get rid of discarded record
           fResizing = CHANGED_SIZE;
           delete x;
```

Code for the recursive search and the fixup as unwind recursion

Code that handles the promotion when data are found

The main issues still to be resolved are how to check the balance at a node after deletions in its left or right subtrees and how to fix things up if the balance is wrong.

AVL rearrangements for deletions

Once again, Adelson-Velskii and Landis had to sort out all the possible situations and work out the correct rearrangements that would both keep the entries in the tree ordered, and the overall tree "more or less balanced".

The checking part is relatively simple, the code is something like the following (which deals with the case where something has been removed from a node's right subtree):

```
Check_balance_after_Right_Delete(t)
    switch (t->Balance()) {
case LEFT_LONG:
           // Right branch from current node was already
           // shorter than left branch, and it has shrunk.
           // Have to rebalance
           Rebalance_Right_Short(t);
           break;
case EVEN:
           t->ResetBalance(LEFT_LONG);
           fResizing = UNCHANGED;
           break;
case RIGHT_LONG:
           t->ResetBalance(EVEN);
           break;
    }
```

If the node had been "even", all that has happened is that it becomes "left long". This can simply be noted, and there is no need to consider changes at higher levels. If it had been "right long" it has now become "even". Its "right longedness" may have been balancing something else's "left longedness"; so it is possible that there will still be a need to make changes at higher levels. The real work occurs when you have a node that was already "left long" and whose right subtree has grown shorter. In that case, rebalancing operations are needed. There are symmetrically equivalent rebalancing operations for a node that was "right long" and whose left subtree has grown shorter.

The actual rearrangements are illustrated in Figure 24.5 for the case where a node was right long and whose left branch has shrunk.

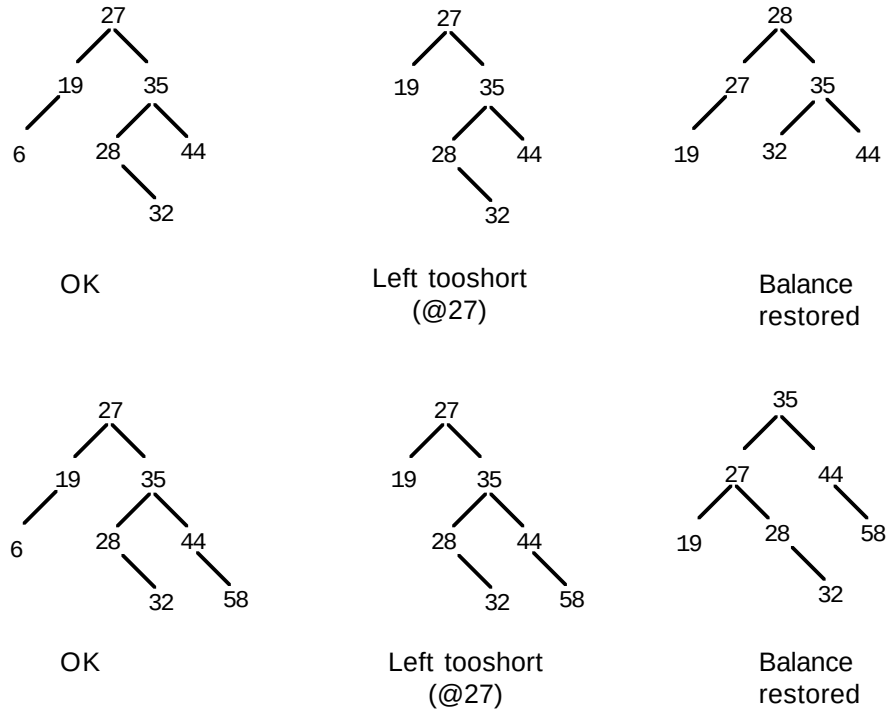


Figure 24.5 Rebalancing a tree after a deletion.

The rearrangements needed depend on the shape of the right subtree of the node that has become "left too short". If this right subtree is itself "left long" (the first example shown in Figure 24.5), then the tree is restored by pushing the current root down into the left branch (making that longer) and pulling a node up from the "left subtree" of the "right subtree" to make the new root for this tree (or subtree). If the right subtree is

evenly balanced (second example in Figure 24.5) or right long (not shown) then slightly different rearrangements apply.

Once again the rearrangements involve shifting pointers around and resetting the balance records associated with the nodes.

24.1.3 An implementation

The following code provides an example implementation of the AVL tree algorithms. The code implementing some functions has been omitted; as already noted, there are symmetrically equivalent "left" and "right" operations so only the code of one version need be shown.

The AVL tree is meant to be used to store pointers to any kind of object that is an instance of a class derived from abstract class `KeyedItem`. The header file should contain declarations for both `KeyedItem` and `AVLTree`. The implementation for class `AVLTree` uses an auxiliary class, `AVLTreeNode`, whose instances represent the tree nodes. This is essentially a private structure and is defined in the implementation file. Since class `AVLTree` has data members that are `AVLTreeNode*` pointers, there has to be a declaration of the form "`class AVLTreeNode;`" in the header file.

Definition of abstract class (base class for classes representing data items)

```
class KeyedItem {
public:
    virtual       ~KeyedItem() { }
    virtual long  Key(void) const = 0;
    virtual void  PrintOn(ostream& out) const  { }
};

inline ostream& operator<<(ostream& out, const KeyedItem& d)
{ d.PrintOn(out); return out; }
inline ostream& operator<<(ostream& out, const KeyedItem* dp)
{ dp->PrintOn(out); return out; }

class AVLTreeNode;
```

The public interface for class `AVLTree` is similar to that for the simple binary tree class. There are several private member functions that deal with issues like those rebalancing manoeuvres.

class AVLTree

```
class AVLTree
{
public:
    AVLTree();
    ~AVLTree();

    int           NumItems(void) const;

    int           Add(KeyedItem* d);
```

```
    KeyedItem     *Find(long key);
    KeyedItem     *Remove(long key);

private:
    void Insert1(KeyedItem* d, AVLTreeNode*& t);
    void Rebalance_Left_Long(AVLTreeNode*& t);
    void Rebalance_Right_Long(AVLTreeNode*& t);

    void Delete1(long bad_key, AVLTreeNode*& t);
    void Check_balance_after_Left_Delete(AVLTreeNode*& t);
    void Check_balance_after_Right_Delete(AVLTreeNode*& t);

    void Rebalance_Left_Short(AVLTreeNode*& t);
    void Rebalance_Right_Short(AVLTreeNode*& t);
    void DeleteRec(AVLTreeNode*& t);
    void Del(AVLTreeNode*& t, AVLTreeNode*& r);

    AVLTreeNode   *fRoot;
    int           fNum;

    KeyedItem     *fReturnItem;
    int           fResizing;
    int           fAddOK;
};
```

Auxiliary functions for insertion and consequent rebalancing

Auxiliary functions for deletion and consequent rebalancing

Data members

```
inline int AVLTree::NumItems(void) const { return fNum; }
```

inline access function

The principal data members are a pointer to the root of the tree and a count for the number of entries in the tree. The other three data members are essentially "work" variables for all those recursive routines that scramble around the tree; e.g. `fResizing` is the flag used to record whether there has been a change in the size of a subtree.

The tree does not "own" the data items that are inserted. The `Remove()` function returns a pointer to the data item associated with the "bad key". "Client code" that uses this AVL implementation can delete data items when appropriate. The destructor for the tree gets rid of all its `AVLTreeNodes` but leaves the data items untouched.

The implementation file starts with declarations of some integer flags and an enumerated type used to represent node balance. Then class `AVLTreeNode` is defined:

```
const      short  CHANGED_SIZE = 1;
const      short  UNCHANGED = 0;

enum eBALANCE { LEFT_LONG, EVEN, RIGHT_LONG };

class AVLTreeNode {
public:
    AVLTreeNode(KeyedItem *d);

    AVLTreeNode*&        LeftLink(void);
```

Class AVLTreeNode

```
    AVLTreeNode*&       RightLink(void);

    long                Key(void) const;
    eBALANCE            Balance(void) const;
    KeyedItem           *Data(void) const;

    void                Replace(KeyedItem *d);
    void                ResetBalance(eBALANCE newsetting);
private:
    eBALANCE      fbalance;
    KeyedItem     *fData;
    AVLTreeNode   *fLeft;
    AVLTreeNode   *fRight;
};
```

An `AVLTreeNode` is something that has a balance factor, a pointer to some keyed data, and pointers to the `AVLTreeNodes` at the head of left and right subtrees. It provides three member functions that provide read access to data such as the balance factor, and two functions for explicitly changing the data associated with the node, or changing the balance.

Note functions that return reference values

In addition, there are the functions `LeftLink()` and `RightLink()`. These return references to the nodes left and right tree links. Because these functions return reference values, calls to these functions can appear on the left hand side of assignments. Although a little unusual, such functions help simplify the code of class `AVLTree`. Such coding techniques are somewhat sophisticated. You probably shouldn't yet attempt to write anything using such techniques, but you should be able to read and understand code that does.

All the member functions of class `AVLTreeNode` are simple; most can be defined as "inline".

Member functions of AVLTreeNode

```
AVLTreeNode::AVLTreeNode(KeyedItem *d)
{
    fbalance = EVEN;
    fLeft = fRight = NULL;
    fData = d;
}

inline eBALANCE AVLTreeNode::Balance(void) const
{ return fbalance; }
…
inline void AVLTreeNode::Replace(KeyedItem *d) { fData = d; }
…
inline AVLTreeNode*& AVLTreeNode::LeftLink(void)
{ return fLeft; }
```

AVLTree

The constructor for class `AVLTree` needs merely to set the root pointer to `NULL` and the count of items to zero. The destructor is not shown. It is like the binary tree

destructor illustrated at the end of Section 23.5.2. It does a post order traversal of the tree deleting each `AVLTreeNode` as it goes.

Constructor

```
AVLTree::AVLTree()
{
    fRoot = 0;
    fNum = 0;
}
```

The `Find()` function is just a standard search that chases down the left or right links as needed. It could be implemented recursively but because of its simplicity, an iterative version is easy:

AVLTree::Find()

```
KeyedItem* AVLTree::Find(long sought_key)
{
    AVLTreeNode *t = fRoot;
    for(; t != NULL; ) {
           if(t->Key() == sought_key) return t->Data();
           else
           if(t->Key() > sought_key) t = t->LeftLink();
           else
           t = t->RightLink();
           }
    return NULL;
}
```

The `Add()` and `Remove()` functions provide the client interface to the real working functions. They set up initial calls to the recursive routines, passing in the root pointer for the tree. Private data members are used rather than have the functions return their results; again this is just so as to slightly simplify the code in a few places.

AVLTree::Add() and AVLTree::Remove()

```
int AVLTree::Add(KeyedItem* d)
{
    fAddOK = 0;
    Insert1(d,fRoot);
    if(fAddOK)
           fNum++;
    return fAddOK;
}

KeyedItem *AVLTree::Remove(long bad_key)
{
    fReturnItem = NULL;
    Delete1(bad_key, fRoot);
    if(fReturnItem != NULL)
           fNum--;
    return fReturnItem;
}
```

The main driver routine for insertion is a straightforward implementation of the algorithm outlined earlier:

```
void AVLTree::Insert1(KeyedItem* d, AVLTreeNode*& t)                    Main driver routine
{                                                                        for insertion of extra
                                                                         node
                                                                         Item is not present,
    if(t == NULL) {                                                      make a node for it
           t = new AVLTreeNode(d);
           fResizing = CHANGED_SIZE;
           fAddOK = 1;
           return;
           }

    if(d->Key() == t->Key()) {                                           Duplicates not
           // cout << "Duplicate entry ignored\n";                       allowed
           fResizing = UNCHANGED;
           return;
           }


    if(d->Key() < t->Key()) {                                            Insert small items in
           Insert1(d,t->LeftLink());                                     left subtree
           if(fResizing == CHANGED_SIZE) {
                  switch (t->Balance()) {
case LEFT_LONG:
                         Rebalance_Left_Long(t);                         Rebalance if
                         t->ResetBalance(EVEN);                          necessary
                         fResizing = UNCHANGED;
                         break;
case EVEN:
                         t->ResetBalance(LEFT_LONG);
                         break;
case RIGHT_LONG:
                         t->ResetBalance(EVEN);
                         fResizing = UNCHANGED;
                         break;
                         }
                  }
           return;
           }

    Similar code for right subtree                                       Insert large items in
}                                                                        right subtree
```

Functions like `Rebalance_Left_Long()` have a "reference to a `AVLTreeNode` pointer" as arguments. These functions may need to reset the pointer; this is why it gets passed by reference. The pointer used as an argument might be the tree's root pointer, `fRoot`, or the `fLeft` or `fRight` data member of some `AVLTreeNode` object.

Rebalancing the tree after an addition
Use balance of left child to select appropriate rebalance manoeuvre

```
void AVLTree::Rebalance_Left_Long(AVLTreeNode*& t)
{

    if((t->LeftLink())->Balance() == LEFT_LONG) {
           AVLTreeNode *tptr = t->LeftLink();
           t->LeftLink() = tptr->RightLink();
           tptr->RightLink() = t;
           t->ResetBalance(EVEN);
           t = tptr;
           }
    else {
           AVLTreeNode *tptr = t->LeftLink();
           AVLTreeNode *tptr2 = tptr->RightLink();

           tptr->RightLink() = tptr2->LeftLink();
           tptr2->LeftLink() = tptr;
           t->LeftLink() = tptr2->RightLink();
           tptr2->RightLink() = t;

           t->ResetBalance(
                  (tptr2->Balance() == LEFT_LONG) ?
                                RIGHT_LONG : EVEN);

           tptr->ResetBalance(
                  (tptr2->Balance()  == RIGHT_LONG) ?
                                LEFT_LONG : EVEN);
           t = tptr2;
           }
}
```

The code highlighted in bold shows calls to the "reference returning" function `LeftLink()`. The first call is on the right hand side of an assignment so the compiler arranges to get the value from the `fLeft` field of the object pointed to by `t`. In the second case, the call is on the left of an assignment. The compiler gets the address of `t`'s `fLeft` data field, and then stores, in this location, the value obtained by evaluating `tptr->RightLink()`. The code highlighted in italics illustrates where the function is changing the value of the pointer passed by reference (in effect, "re-rooting" the current subtree).

The corresponding function `Rebalance_Right_Long()` is similar and so is not shown.

Deletion functions

The main driver routine for deletion and the functions for checking balance after left or right deletions are simple to implement from the outline algorithms given earlier. The `DeleteRec()` function (which removes nodes with one or no children and arranges for promotion of data in other cases) is:

```
void AVLTree::DeleteRec(AVLTreeNode*& t)
{
    fReturnItem = t->Data();
    if(t->RightLink() == NULL) {
```

```
            AVLTreeNode *x = t;
            t = t->LeftLink();
            fResizing = CHANGED_SIZE;
            delete x;
            }
    else
    if(t->LeftLink() == NULL) {
            AVLTreeNode *x = t;
            t = t->RightLink();
            fResizing = CHANGED_SIZE;
            delete x;
            }
    else {
            Del(t,t->LeftLink());
            if(fResizing == CHANGED_SIZE)
                    Check_balance_after_Left_Delete(t);
            }
}
```

The `Del()` function deals with the processes of finding the data to promote and the replacement action:

```
void AVLTree::Del(AVLTreeNode*& t, AVLTreeNode*& r)
{
    if(r->RightLink() != NULL) {
            Del(t,r->RightLink());                          // Recursive search down to replacement data
            if(fResizing == CHANGED_SIZE)
                    Check_balance_after_Right_Delete(r);
            }
    else {
            AVLTreeNode   *x;                               // Doing the replacement
            t->Replace(r->Data());
            x = r;
            r = r->LeftLink();
            fResizing = CHANGED_SIZE;
            delete x;
            }
}
```

There are symmetrically equivalent routines for rebalancing a node after deletions in its left or right subtrees. This is the code for the case where the right subtree has shrunk:

```
void AVLTree::Rebalance_Right_Short(AVLTreeNode*& t)        // Code to rebalance after a deletion
{
    AVLTreeNode* tptr = t->LeftLink();

    if(tptr->Balance() != RIGHT_LONG) {
            t->LeftLink() = tptr->RightLink();
            tptr->RightLink() = t;
```

```
                if(tptr->Balance() == EVEN) {
                        t->ResetBalance(LEFT_LONG);
                        tptr->ResetBalance(RIGHT_LONG);
                        fResizing = UNCHANGED;
                        }
                else {
                        t->ResetBalance(EVEN);
                        tptr->ResetBalance(EVEN);
                        }
                t = tptr;
                }
        else {
                AVLTreeNode *tptr2 = tptr->RightLink();
                tptr->RightLink() = tptr2->LeftLink();
                tptr2->LeftLink() = tptr;
                t->LeftLink() = tptr2->RightLink();
                tptr2->RightLink() = t;
                t->ResetBalance((tptr2->Balance() == LEFT_LONG) ?
                                        RIGHT_LONG : EVEN);
                tptr->ResetBalance((tptr2->Balance() == RIGHT_LONG) ?
                                LEFT_LONG : EVEN);
                t = tptr2;
                tptr2->ResetBalance(EVEN);
                }
    }
```

The functions not shown are all either extremely simple or are the left/right images of functions that have been given.

24.1.4 Testing!

Just look at the AVL algorithm! It has special cases for left subtrees becoming too long on their own left sides, and left subtrees becoming too long on their right subtrees, code for right branches that are getting shorter, and …. It has special cases where data elements must be promoted from other tree cells. These operations may involve searches down branches of trees. The tree has to be quite large before there is even a remote possibility of some these special operations being invoked.

The simpler abstract data types like the lists and the queues shown in Chapter 21 could be tested using small interactive programs that allowed the tester to exercise the various options like getting the length or adding an element. Such an approach to testing something like the AVL tree is certain to prove inadequate. When arbitrarily selecting successive addition and deletion operations, the tester simply won't pick a sequence that exercises some of the more exotic operations.

The approach to testing has to be more systematic. You should provide a little interactive program, like those in Chapter 21, that can be used for some preliminary tests. A second non-interactive test program would then be needed to thoroughly test

all aspects of the code. This second program would be used in conjunction with a "code coverage tool" like that described in Chapter 14.

Both the test programs would need some data objects that could be inserted into the tree. You would have to define a class derived from class `KeyedItem`, e.g. class `TextItem`:

TextItem – a class of object that can be put into an AVLTree

```
class TextItem : public KeyedItem
{
public:
    TextItem(const char* info, long k);
    ~TextItem();
    long Key(void) const;
    void PrintOn(ostream& out) const;
private:
    char   *fText;
    long   fk;
};

TextItem::TextItem(const char *info, long k)
{
    fk = k;
    fText = new char[strlen(info) + 1];
    strcpy(fText, info);
}

TextItem::~TextItem()
{
    delete [] fText;
}

void TextItem::PrintOn(ostream& out) const
{
    out << "[ " << fText << ", " << fk << "] ";
};

long TextItem::Key(void) const { return fk; }
```

A `TextItem` object is just something that holds a long integer key and a string. The interactive test program can get the user to enter these data; the way the data are generated and used in the automated program is explained later.

The main function for an interactive test program, `HandTest()`, is shown below. It has the usual structure with a loop offering user commands.

```
AVLTree gTree;

void HandTest(void)
{
    KeyedItem*    d;
    for(;;) {
```

Get command

```
char ch;
cout << "Action (a = Add, d = Delete, f = Find,"
        " p = Print Tree, q = Quit) : ";
cin >> ch;

switch (ch) {
```

An "add" command results in the creation of an extra `TextItem` that gets put in the tree. (The function `AVLTree::Add()` returns a success/failure indicator. A failure should only occur if an attempt is made to insert a record with a duplicate key. If the add operation fails, the "duplicate" record should be deleted.)

Adding TextItems

```
case 'a':
case 'A':
                {
                long key;
                char buff[100];
                cout << "Key : " ; cin >> key;
                cout << "Name : "; cin >> buff;
                d = new TextItem(buff, key);
                if(gTree.Add(d) != 0)
                       cout << "Inserted OK" << endl;
                else {
                       cout << "Duplicate " << endl;
                       delete d;
                       }
                }
                break;
```

A "delete" command gets the key for the `TextItem` to be removed then invokes the trees remove function. Function `AVLTree::Remove()` returns `NULL` if an item with the given key was not present. If the item was found, a pointer is returned. The item can then be deleted.

There would also be a "find" command (not shown, is trivial to implement), a "quit" command, and possibly a "print" command. During testing it would be useful to get the tree displayed so it might be worth adding an extra public member function `AVLTree::PrintTree()`. The algorithm required will be identical to that used for the simpler binary tree.

Deleting TextItems

```
case 'd':
case 'D':
                {
                long   bad;
                cout << "Enter key of record to be removed";
                cin >> bad;
                d = gTree.Remove(bad);
                if(d == NULL)
                       cout << "No such record" << endl;
```

```
                    else {
                           cout << "Removing " << *d << endl;
                           delete d;
                           }
                    }
                    break;
  case 'f':
  case 'F':
                    …
                    …
                    break;
  case 'p':
  case 'P':
                    gTree.PrintTree();
                    break;
  case 'q':
  case 'Q':
                    return;
  default:
                    cout << "?" << endl;
                    }
            }
  }
```

Hand testing will never build up the large complex trees where less common operations, like promotion of data, get fully tested. You need code that performs thousands of insertion, find, and deletion operations on the tree and which checks that each operation returns the correct result.

Mechanism for an automated test

This is not as hard as it might seem. Basically, you need a testing function that starts by loading some records into the tree and then "randomly" chooses to add more records, delete records, or search for records. The function will need a couple of control parameters. One determines the number of cycles (should be 10000 to 20000). The other parameter, `testsisze`, determines the range used for keys; there is a limit, `kTESTMAX`, for this parameter. The use of the `testsisze` parameter is explained below.

```
void AutoTest()
{
    int testsize;
    int runsize;
    cout << "Enter control parameters for auto-test ";
    cin >> testsize >> runsize;
    assert(testsize > 1);
    assert(testsize < kTESTMAX);

    Initialize(testsize);
    for(int i=0; i < testsize / 2; i++)
           Add(testsize);

    for(int j=0; j < runsize; j++) {
```

```
            int r = rand() % 4;
            switch(r) {
    case 0:         Add(testsize); break;
    case 1:         Find(testsize); break;
    case 2:
    case 3:
                    Remove(testsize); break;
                    }
            }
        cout << "Test complete, counters of actions: " << endl;
        for(i = 0; i < 6; i++)
            cout << i << ":  " << gCounters[i] << endl;
    }
```

As you can see, the loop favours removal operations. This makes it likely that at some stage all records will be removed from the tree. There are often obscure special cases related to collections becoming empty and then being refilled so it is an aspect that you want to get checked.

Function `AutoTest()` uses the auxiliary functions, `Add()`, `Find()`, and `Remove()` to do the actual operations. These must be able to check that everything works correctly.

Keeping track of valid keys

Correct operation can be checked by keeping track of the keys that have been allocated to `TextItem` records inserted in the tree. When creating a new `TextItem`, the test program gives it a "random" key within the permitted range:

```
TextItem *MakeATextItem(int testsize)
{
    int r = rand() % testsize;
    return new TextItem("XXXX", r);
}
```

The `Add()` function keeps track of the keys that it has allocated and for which it has inserted a record into the tree. (It only needs an array of "booleans" that record whether a key has been used):

```
const int kTESTMAX       = 5000;
short      gUsed[kTESTMAX];
```

If the same randomly chosen key has already been used, an addition operation should fail; otherwise it should succeed. The `Add()` function can check these possibilities. If something doesn't work correctly, the program can stop after generating some statistics on the tree (function `ReportProblem()`, not shown). If things seem OK, the function can increment a count of operations tested:

```
const int ADD_OK         = 0;
const int ADD_DUP        = 1;
const int FIND_OK        = 2;
const int FIND_EMPTY     = 3;
```

```
const int REMOVE_OK      = 4;
const int REMOVE_FAIL    = 5;

long       gCounters[6];        // record test operations

void Add(int testsize)
{
    TextItem *t  = MakeATextItem(testsize);
    long k = t->Key();
    if(gUsed[k] ==0) {
           /* Should get a successful insert */                    "Fresh" key, Add()
           if(gTree.Add(t) != 0) gCounters[ADD_OK]++;              should work
           else {
                  cout << "Got a 'duplicate' response when"
                         "should have been able to add"
                                << endl;
                  ReportProblem(k);
                  exit(1);
                  }
           gUsed[k] = 1;                                          Mark key in use
           }
    else {
           /* Should get a duplicate message */                   Already used key,
           if(gTree.Add(t) == 0) {                                Add() should fail
                  gCounters[ADD_DUP]++;
                  delete t;
                  }
           else {
                  cout << "Failed to notice a duplicate" << endl;
                  ReportProblem(k);
                  exit(1);
                  }
           }
}
```

Of course, the `gUsed[]` and `gCounters[]` arrays have to be initialized. The `Initialize()` function is called at the start of the `AutoTest()` function:

```
void Initialize(int testsize)
{
    for(int i = 0; i < testsize; i++)
           gUsed[i] = 0;
    for(int j = 0; j < 6; j++)
           gCounters[j] = 0;
}
```

The functions `Find()` and `Remove()` can also use the information in the `gUsed[]` array. Function `Find()` (not shown) randomly picks a key, inspects the corresponding `gUsed[]` array to determine whether or not a record should be found, then attempts the `gTree.Find()` operation and verifies whether the result is as expected.

`Remove()` is somewhat similar. However, if it successfully removes a `TextItem`, it must also delete it and clear the corresponding entry in the `gUsed[]` array:

```
void Remove(int testsize)
{
    long k = rand() % testsize;
    KeyedItem *d = gTree.Remove(k);
    if(gUsed[k] == 0) {
           /* Remove operation should have failed */

           if(d == NULL) gCounters[REMOVE_FAIL]++;
           else {
                  cout << "Removed a thing that wasn't there"
                         << endl;
                  ReportProblem(k);
                  exit(1);
                  }
           }
    else {
           /* Remove operation should succeed */
           if(d != NULL) {
                  gCounters[REMOVE_OK]++;
                  delete d;
                  gUsed[k] = 0;
                  }
           else {
                  cout << "Failed to find and remove a data"
                         "item" << endl;
                  ReportProblem(k);
                  exit(1);
                  }
           }
}
```

Trying to remove item with non-existent key

Trying to remove an item that should be present

Runs can be made with different values for the `testsize` parameter. Large values (3000 - 5000) result in complicated deep trees (after all, the first step involves filling the tree with `testsize/2` items). These trees have cases where data have to be promoted from remote nodes, leading to a long sequence of balance checks following the deletion.

Small values of the `testsize` parameter keep the tree small, force lots of "duplicate" checks, and make it more likely that all elements will be deleted from the tree at some stage in the processing.

The test program can use the `gCounters[]` counts to provide some indication as to whether the tests are comprehensive. But this is still not adequate. You may know that your tree survived 10,000 operations but you still can't be certain that all its functions have been executed.

Complex algorithms like the AVL code require testing with the code coverage tools. The code was run on a Unix system where the tcov tool (Chapter 14) was available.

Several different runs were performed and the final accumulated statistics were analyzed using tcov. A fragment of tcov's output is as follows:

Output from code coverage tool

```
          void AVLTree::Delete1(long bad_key, AVLTreeNode*& t)
122513 -> {
                  if(t == NULL)  {
 10217 ->                 fResizing = UNCHANGED;
                          return;
                          }

112296 ->         if(bad_key == t->Key()) {
  4902 ->                 DeleteRec(t);
                          return;
                          }

107394 ->         if(bad_key < t->Key()) {
 56338 ->                 Delete1(bad_key,t->LeftLink());
                          if(fResizing == CHANGED_SIZE)
  3070 ->                     Check_balance_after_Left_Delete(t);
  3070 ->                 return;
                          }

 51056 ->         Delete1(bad_key,t->RightLink());
                  if(fResizing == CHANGED_SIZE)
  2996 ->                 Check_balance_after_Right_Delete(t);
  2996 ->         return;

          }
          void AVLTree::Check_balance_after_Left_Delete(
               AVLTreeNode*& t)
  4338 -> {

                    switch (t->Balance()) {
          case LEFT_LONG:
  1480 ->                 t->ResetBalance(EVEN);
                          break;
          case EVEN:
  2256 ->                 t->ResetBalance(RIGHT_LONG);
                          fResizing = UNCHANGED;
                          break;
          case RIGHT_LONG:
   602 ->                 Rebalance_Left_Short(t);
                          break;
                  }
  4338 -> }
```

Such results give greater confidence in the code. (The tcov record, along with the test programs, form part of the "documentation" that you should provide if your task was to build a complex component like an AVL tree.)

Code coverage by hand

If you can't get access to something like tcov, you have to achieve something similar. This means adding conditionally compiled code. You will have to define a global array to hold the counters:

```
#ifdef TCOVING
int __my__counters[1000];
#endif
…
```

and you will have to edit every function, and every branch statement within a function, to increment a counter:

```
void AVL::Insert(
{
#ifdef TCOVING
    __my__counters[17]++;
#endif
```

Finally, you have to provide a function that prints contents of the table when the program terminates.

Unfortunately, this process is very clumsy. It is easy to make mistakes such as forgetting to associate a counter with some branch, or to have two bits of code that use the same counter (this is quite common if the main code is still being finalized at the same time as being tested). The printouts of counts aren't directly related to the source listings, the programmer has to read the two outputs together.

Further, the entire process of hand editing is itself error prone. Careless editing can easily cut a controlled statement from the `if()` condition that controls it!

The best solution is to use a code coverage tool on some other platform while pressing your IDE supplier for such a tool in the next version of the software. (The supplier should oblige, code coverage tools are as easy, or easier, to add to a compiler than the time profilers which are commonly available.)

Don't really trust it even if tcov says its tested

A final warning, even if tcov (or equivalent) says that you've tested all the branches in your code, you still can't be certain that it is correct. You really should try that exercise at the end of Chapter 21 where "live" listcells were "deleted" from a list and the program still ran "correctly" (or at least it ran long enough to convince any typical tester that it was working correctly).

Memory problems (leaks, incorrect deletions, other use of "deleted" data) are the reason for having an automated program that performs tens of thousands operations. If the tests are lengthy enough you have some chance of forcing memory bugs to manifest themselves (by crashing the program several minutes into a test run). Unfortunately, some memory related bugs are "history dependent" and will only show up with specific sequences of operations; such bugs are exceptionally difficult to find.

Testing can establish that your code has bugs; testing can not prove that a program is bug free. Despite that, it is better if code is extensively tested rather than left untested.

24.2 BTREE

24.2.1 Multiway trees

You are not restricted to "binary trees", there are alternative tree structures. In fact, there are numerous forms of "multiway tree". They all serve much the same role as an AVL tree. They are "lookup" structures for keyed data. These trees provide `Add()`, `Find()`, and `Remove()` functions. They provide a guarantee that their performance on all operations in close to O(lg(N)) (with N the number of items stored in the tree).

These trees have more than one key in each of their tree nodes and, consequently, more than two links down to subtrees. The data inserted into a tree are kept ordered; data items with small keys are in "left subtrees", data with "middling" keys can be found down in other subtrees, and data with large keys tend to be in "right subtrees". The trees keep themselves "more or less balanced" by varying the number of data items stored in each node. The path from root to leaf is kept the same for all leaves in the tree (a major factor in keeping costs O(lg(N))).

Although the structures of the nodes, and the forms for the trees, are radically different from those in the AVL tree, there are some similarities in the overall organization of the algorithms.

An operation like an insertion is done by recursively chasing down through the tree to the point where the extra data should go. The new item is added. Then, as the recursion unwinds, local "fix ups" are performed on each tree node so as to make certain that they all store "appropriate" numbers of data items and links.

What is an "appropriate" number of data items for a node? How are "fix ups" done when nodes don't have appropriate number of items? Each different form of multiway tree has slightly different rules with regard to these issues.

Deletions are also handled in much the same way as in AVL and binary trees. You search, using a recursive routine, for the item that is to be removed. Items can easily be cut out of "leaf nodes". Items that are in "internal nodes" (those with links down to subtrees) have to be replaced by data "promoted" from a leaf node lower in the tree (the successor, or predecessor, item with the key immediately after, or before, that of the item being removed). If data are promoted from another node, the original copy of the promoted data must then be cut out from its leaf node.

Once the deletion step has been done, the recursion unwinds. As in the case of insertion, the "unwinding" process must "fix up" each of the nodes on the path back to the root. Once again, different forms of multiway tree have slightly different "fix up" rules.

Of course, searches are fairly simple. You can chase down through the tree following the links. As nodes can have more than one key, there is an iterative search through the various keys in each node reached.

2-3 Trees

You will probably get to study different multiway trees sometime later in your computer science career. Here, only one simple version need be considered. It is usually called a "two-three" tree because each node can hold two keys and three links down to subtrees. It has similarities to, and can act as an introduction to the BTree which the real focus of this section.

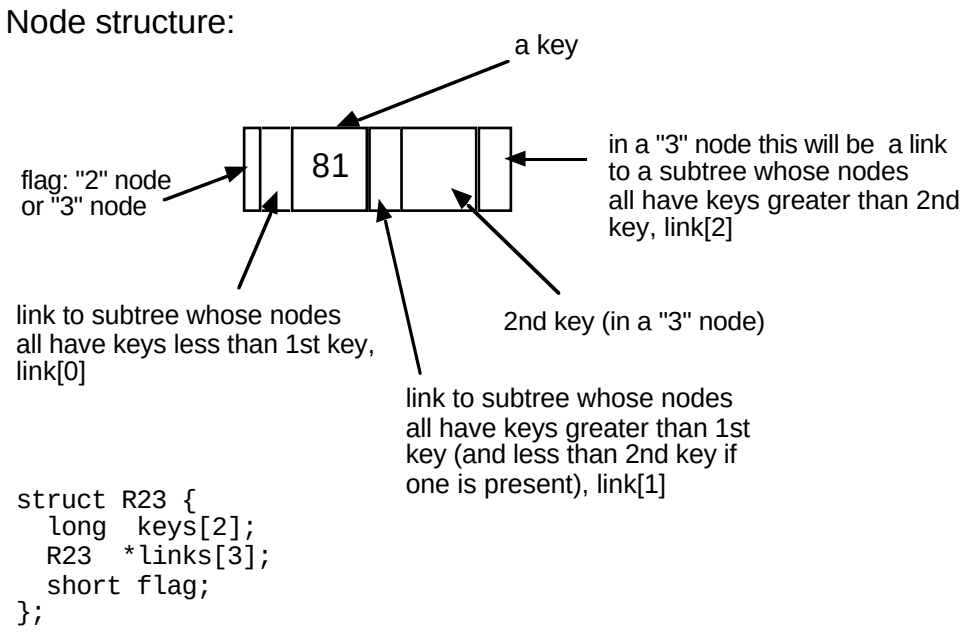
For simplicity, these 2-3 trees will be shown storing just integer keys. If you were really implementing this kind of tree, the "integer key" fields used in the following discussion would be replaced by structures that comprised an integer key and a pointer to the real data item (as was done in the binary tree example in Chapter 21).

Figure 24.6 illustrates the form of a tree node for this simplified 2-3 tree, while Figure 24.7 illustrates an actual tree built using these nodes. The tree node has an array of two longs for the keys, an array of three pointers for the links to subtrees and a "flag" indicating whether it is a "2-node" (one key, two links used), or a "3-node" (both keys and all three links used). In leaf nodes, the link fields will be `NULL`.

Search

The search algorithm is simple:

```
compare search key with 1st (only?) key in node
if equal
    report record as found
```




```
struct R23 {
  long  keys[2];
  R23  *links[3];
  short flag;
};
```

Figure 24.6 Structure of a node for a "2-3" multiway tree.

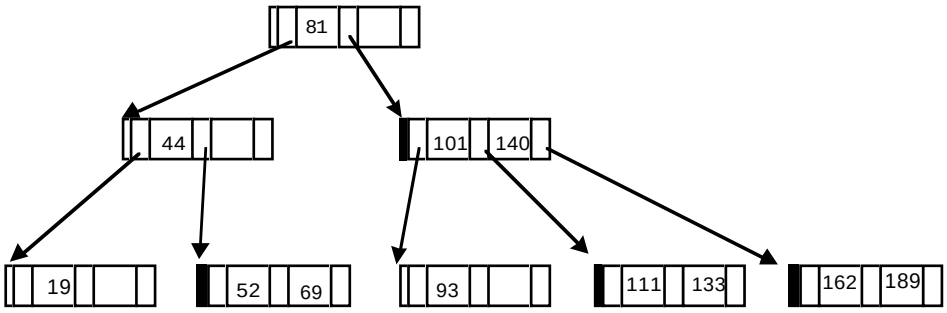


Figure 24.7 An example "2-3" tree.

```
else
if less
    search down subtree 0
else
if greater
    if this is a 2-node then search down subtree 1
    else
           compare search key with 2nd key in node
                  if equal
                         report record as found
                  else
                  if less
                         search down subtree 1
                  else
                         search down subtree 2
```

Insertion

New keys are inserted into leaf nodes. In some cases this is easy. In the example tree shown in Figure 24.7, insertion of the key value 33 is easy. The search for the correct place goes left down `link[0]` from the node with key 81, and again left down `link[0]` from the node with key 44. The next node reached is a leaf node. This one has only one entry, 19, so there is room for the key 33. The key can be inserted and the node's flag changed to mark it as a "3-node" (two keys, potentially three links though currently all these links are `NULL`).

Splitting nodes to make room for another inserted key

Insertion of the key value 71 would be more problematic. Its place is in the leaf node with the keys 52 and 69; but this node is already fully occupied. Insertion into a full leaf node is handled by "splitting the node". There will then be three keys, and two leaf nodes. The least valued of the three keys goes in the "left" node resulting from the split; the key with the largest value goes in the "right" node; while the middle valued key gets moved up one level to the parent node. This is illustrated in Figure 24.8.

The two leaf nodes are both "2-nodes", while their parent (the node that used to hold just key 44) now has two keys and three links and so it is now a "3-node".

The results of two additional insertions are illustrated in Figures 24.9 and 24.10.

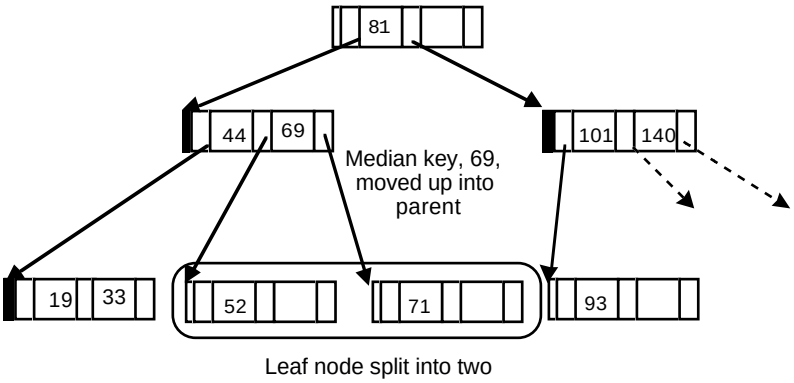


Figure 24.8 Splitting a full leaf node to accommodate another inserted key.

First, the key 20 is inserted. This should go into the leaf currently occupied by keys 19 and 33. Since this leaf is full, it has to be split. Key 19 goes in the left part, key 33 in the right part and key 20 (the median) has to be inserted into the parent. But the parent, the node with 44 and 69 is itself full. So, it too must be split. The left part will hold the smallest key (the 20) and have links down to the nodes with 19 and 33. The right part will hold the key 69 and links down to the nodes with keys 52 and 71. The median key, 44, must be pushed into the parent, the root node with the 81. This node has room; it gets changed from a 2-node to a 3-node. The resulting situation is shown in Figure 24.9.

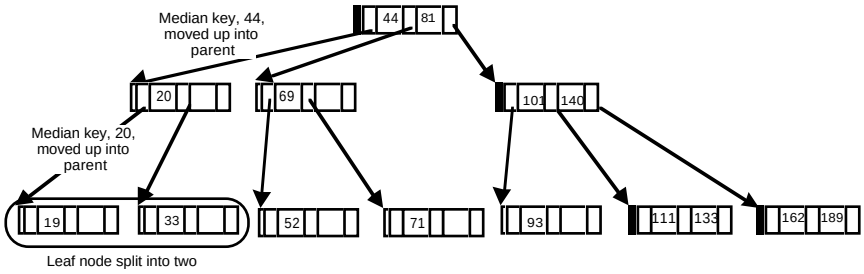


Figure 24.9 Another insertion, another split, and its ramifications.

Insertion of the next key value, 161, causes more problems. If should go in the leaf node where the values 162 and 189 are currently location. As this leaf is full, it must be split. The new key 161 can go in the left part; the large key 189 can go in the new right node; and the median key, 162, (and the link to the new right node) get passed back to be inserted into the parent node. But this node, the one with the 101 and 140 keys is also full. So it gets split. One part gets to hold the key 101 and links down to the node with 93 and the node with 111 and 133. The new part gets to hold the key 162 and its

links down to the node with 161 and the node with 189. The median value, 140, has to go in the parent node. But this is full. So, once again a split occurs. One node takes the 44, another takes the 140 and the median value, 81, has to be pushed up to the parent level.

Growing a new root

There isn't a parent. The node with keys 44 and 81 used to be the root node of the tree. So, it is time to "grow a new root". The new root holds the 81 key. The result is as shown in Figure 24.10.

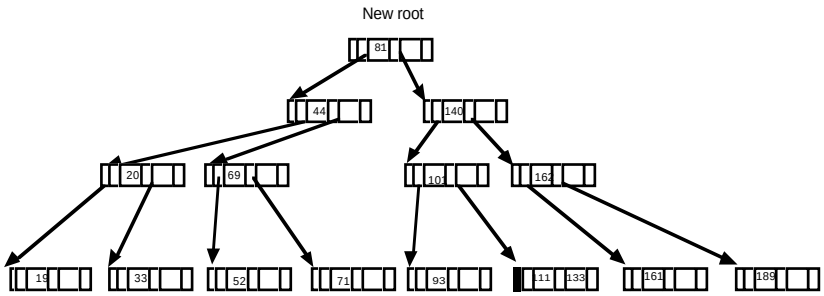


Figure 24.10 The tree grows a new root.

Trees in computer programs are always strange. Their branch points have "children" and their leaves have "parents". They grow downwards, so "up" means closer to the root not nearer to the leaves. These multiway trees add another aberrant behaviour; they grow at the root rather than at the ends of existing branches. It is done this way to keep those paths from root to leaf the same for all leaves; this is required as its part of the mechanism that guarantees that searches, insertions, (and deletions) have a cost that is proportional to O(lg(N)).

BTrees

A BTree is simply a 2-3 tree on steroids. Its nodes don't have two keys and three links; instead its going to be something like 256 keys and 257 links. A fully populated BTree (one where all the nodes held the maximum possible number of keys) could hold 256 keys in a one level tree, around 60000 keys in a two level tree, sixteen million keys in a three level tree. Figure 24.11 gives an idea as to the form of a node and shape of a BTree. The node now has a count field rather than a flag; the count defines the number of keys in the node.

A BTree can be searched, and data can be inserted into a BTree, using algorithms very similar to those that have just been illustrated for the 2-3 tree. You can implement BTrees that work this way, where all the links are memory pointers, and the real data records are accessed using pointers that are stored in the nodes along with their key values.

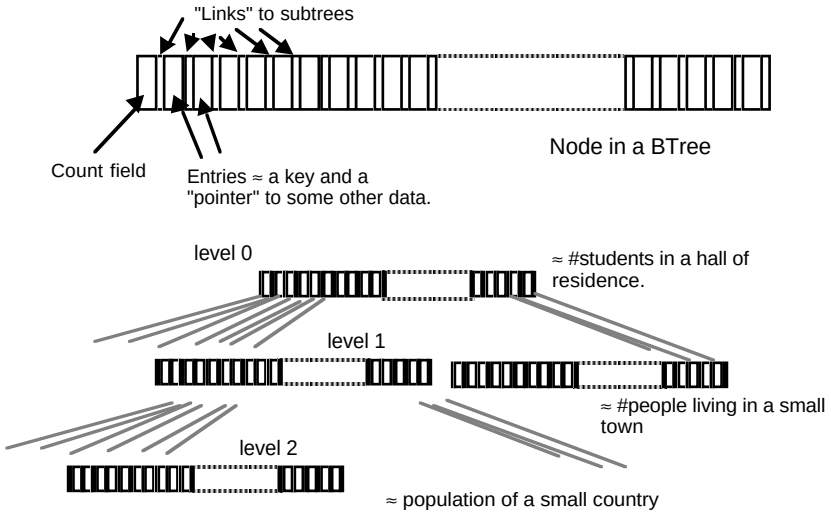


Figure 24.11 Features of a BTree.

But you don't have any really good reasons for using a memory resident BTree like that. If all your data fit in main memory, you might as well use something more standard like an AVL tree.

Exceeding memory limits

However, if you have a really large collection of keyed data records, it is likely that they won't all fit in memory. A large company (e.g. a utility like an electricity company) may have records on two million customers. Each of these customer records is likely to be a thousand bytes or more (name, address, payment records, …). Now two thousand million bytes of data requires rather more "SIM" chips than fit in the average computer. You can't keep such data in memory instead they must be kept on disk.

This is where the BTree becomes useful As will be explained more in the next two sections, it allows you to keep both your primary data records, and your search tree structure, out on disk. Only a few nodes from the tree and a single data record ever need be in primary memory. So you can have very large data collections, provided that you have sufficient disk space (and most PCs support disks with up to 4 gigabytes capacity).

24.2.2 A tree on a disk?

A binary file can always be treated as an array of bytes. If you know where a data record is located (i.e. the "array index" of its first byte) you can use a "seek" operation on a file to set the position for the next read or write operation. Then, provided you know the size of the data record, you can use a low level read or write operation to

transfer the necessary number of bytes. These operations have been illustrated previously with the examples in Chapters 18 (the customer records example), 19 and 23 (the different InfoStore examples).

This ability to treat a file as a byte array makes it practical to map something like a tree structure onto a disk file. We can start by considering simple binary trees that hold solely an integer key. A memory version of such a tree would use structures like the following:

```
struct binr {
    long          key;
    binr          *left_p;
    binr          *right_p;
};
```

with address pointers `left_p` and `right_p` holding the locations in memory of the first node in the corresponding subtree. If we want something like that on a disk file, we will need a record like the following:

```
struct dbinr {
    long          key;
    daddr_t       left;
    daddr_t       right;
};
```

The values in the `left` and `right` data members of a `dbinr` structure will be byte locations where a node is located in the disk file. These will be referred to below as "disk addresses", though they are more accurately termed "file offsets". The type `daddr_t` ("disk address type) is an alias for long integer. It is usually defined in one of the standard header files (stdlib, unistd or unix, or sys_types, or …). If you can't locate the right header you can always provide the typedef yourself:

```
typedef long daddr_t;
```

Storing the tree structure in a disk file

Figure 24.12 illustrates how a binary tree might be represented in a disk file (the numbers used for file offsets assume that the record size is twelve bytes which is what most systems would allocate for a record with three long integer fields).

The first record inserted would go at the start of the file (disk address 0); in the example shown this was the record with key 45.. Initially, the first node would have -1s in its `left` and `right` link fields (-1 is not a valid disk address, this value serves the same role as `NULL` in a memory pointer repesentation of a tree).

The second record added had key 11. Its record gets written at the end of the existing file, so it starts at byte 12 of the file. The `left` link for the first node would be changed to hold the value 12. Similarly, addition of a record with key 92 results in a new node being created on disk (at location 24) and this disk address would then be written into the appropriate link field of the disk record with key 45.

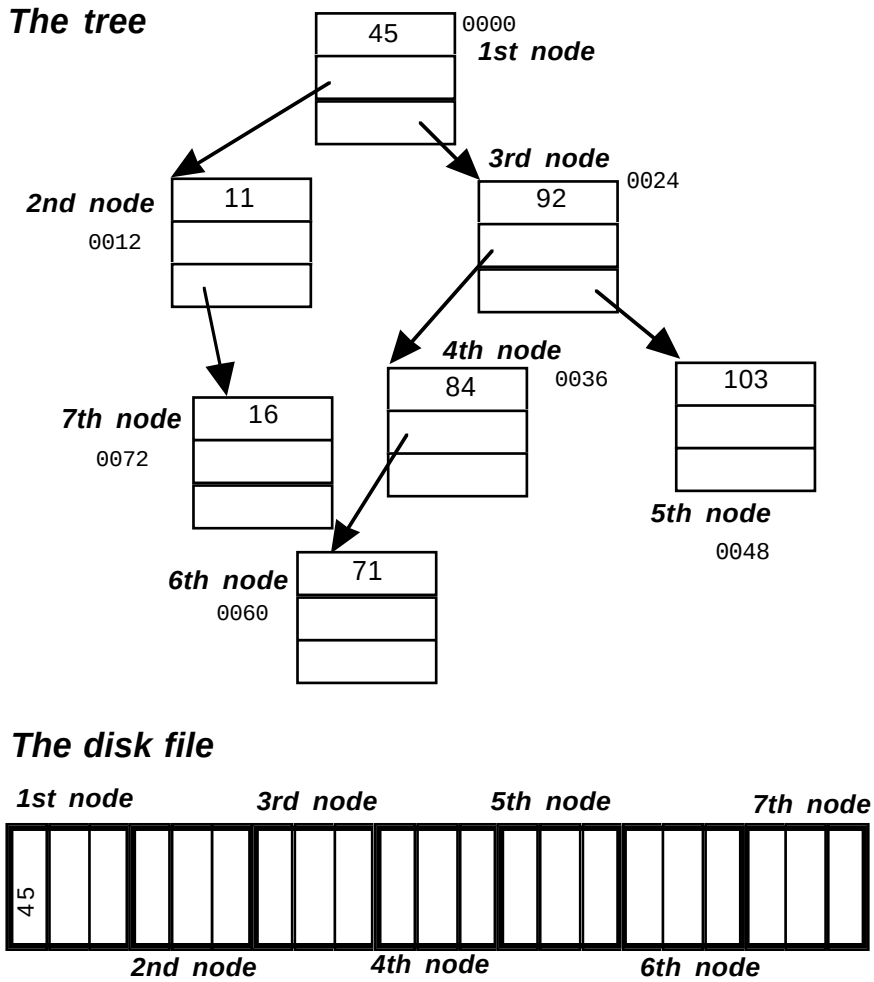


Figure 24.12 Mapping a binary tree onto records in a disk file.

You should have no difficulty in working out how the rest of the file gets built up and the links get set.

Such a tree on disk can be searched to determine whether it contains a record with a given key. The code would be something like the following:

```
fstream    treefile;
int search(long sought)
{
    // Assume that treefile has already been opened successfully
    dbinr  arec;
```

```
    // "root" node will be at location 0 of file
    long diskpos = 0;
    for(;;) {
            treefile.seekg(diskpos);
            treefile.read((char*)&arec, sizeof(dbinr));
            if(sought == arec.key) return 1;
            if(sought < arec.key)
                    diskpos = arec.left;
            else diskpos = arec.right;
            if(diskpos == -1)
                    return 0;
            }
}
```

This is just another version of an iterative search on a binary tree (similar to the search function illustrated in Section 24.1 for searching an AVL tree). It is a relatively expensive version; each cycle of the loop involving disk transfer operations.

Store data in a separate file

Normally, you would have data records as well as keys. You would use two files; one file stores the tree structure, the other file would store the data records (a bit like the index file and the articles file in the InfoStore example). If all the data records are the same size, there are no difficulties. The record structure for a tree node would be changed to something like:

```
struct dbinr {
    long          key;
    daddr_t       dataloc;      // extra link to datafile
    daddr_t       left;
    daddr_t       right;
};
```

With the extra field being the location of the data associated with the given key; this would be an offset into the second data file.

The search routine to get the data record associated with a given key would be:

```
fstream    treefile;
fstream    datafile
int search(long sought, datarec& d)
{
    // Assume both files have already been opened successfully
    dbinr  arec;
    long diskpos = 0;
    for(;;) {
            treefile.seekg(diskpos);
            treefile.read((char*)&arec, sizeof(dbinr));
            if(sought == arec.key) {
                    datafile.seekg(arec.dataloc);
                    datafile.read((char*)&d, sizeof(datarec));
                    return 1;
                    }
```

```
                if(sought < arec.key)
                        diskpos = arec.left;
                else diskpos = arec.right;
                if(diskpos == -1)
                        return 0;
                }
        }
```

The records are stored separately from the tree structure because you don't want to read each record as you move from tree node to tree node. You only want to read a data record, which after all might be quite large, when you have found the correct one.

It should be obvious that there are no great technical problems in mapping binary trees (or more elaborate things like AVL trees) onto disk files. But it isn't something that you would really want to do.

The iterative loops in the search, and the recursive call sequences involved in the insertion and deletion operations require many tree nodes to be read from the "tree file". As illustrated in Figure 24.13, the tree nodes are going to be stored in disk blocks that may be scattered across the disk. Each seek and read operation may involve relatively lengthy disk operations (e.g. as much as 0.02 seconds for the operating system to read in the disk block containing the next tree node).

If the trees are deep, then many of the operations will involve reading multiple blocks. After all, a binary tree that has a million keys in it will be twenty levels deep. Consequently each search operation on a disk based binary tree may require as many as twenty disk seeks to find the required tree node (and then one more seek to find the corresponding data).

A BTree with a million keys is going to be only three levels deep if its nodes have ≈250 keys. Searching such a tree will involve three seeks to get just three tree nodes, and then the extra seek to find the data. Three disk operations is much better than twenty. BTrees are just as easy to map onto disks as are other trees. But because of their shallow depths, their use doesn't incur so much of a penalty.

The structure for a BTree tree node, and the form of the BTree index file, are illustrated in Figure 24.14. The example node has space for only a few keys where a real BTree has hundreds. Small sized nodes are necessary in order to illustrate algorithms and when testing the implementation. Many of the more complex tree rearrangements occur only when a node becomes full or empty. You would have to insert several million items if you wanted to fill most of the nodes in a three level tree with 250 keys per node; that would make testing difficult. Testing is a lot easier if the nodes have only a few keys.

The node has a count specifying the number of key/location pairs that are filled, an array of these key/location pairs, and an array of links. The links array is one larger than the keys array and a BTree node either has no links (a leaf node) or has one more link than it has keys. The entries in the links array are again disk addresses; they are the addresses of the BTree nodes that represent subtrees hung below the current BTree node. A BTree node has the following data members:

- Tree node of a few continguous bytes

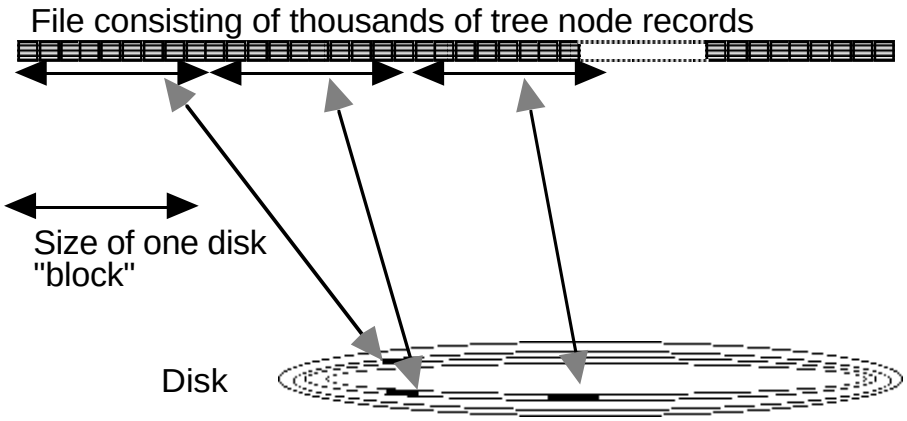


Figure 24.13 Mapping a file onto disk blocks.

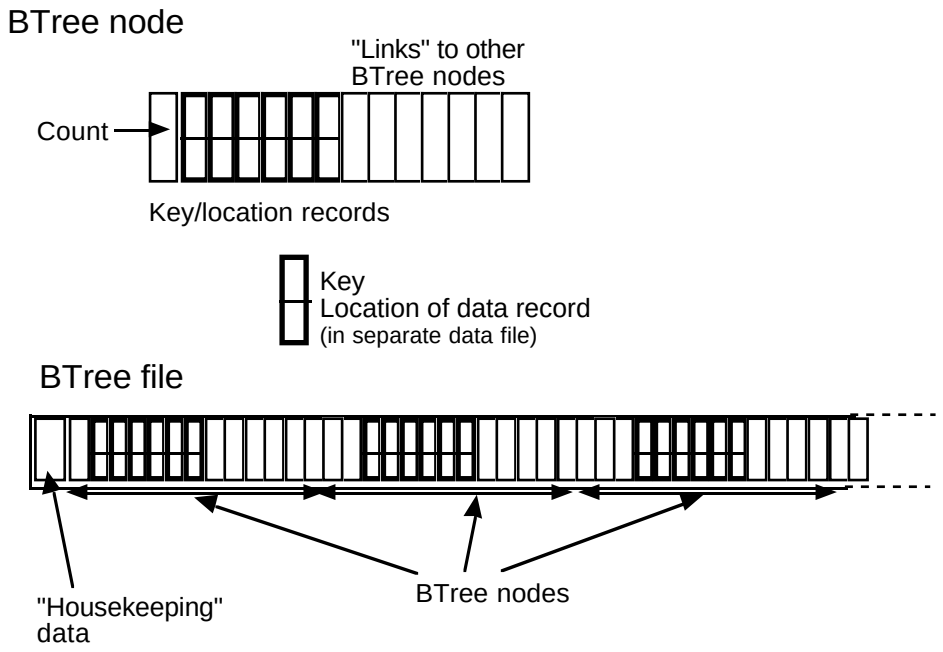


Figure 24.14 BTree nodes and the BTree file.

```
int         n_data;
KLRec       data[MAX];    /* 0..n_data-1 are filled */
daddr_t     links[MAX+1]; /* 0..n_data are filled */
```

where `KLRec` is defined as "`struct KLRec { long fKey; daddr_t  fLocation; };`". (For simplicity, most of the later diagrams will omit the link to the data record and show solely the values of the keys in those key/location fields. Similarly, the disk addresses of subtrees, that would be in the link fields, are usually not shown.)

The BTree file consists mainly of BTree node records, but there is small amount of "housekeeping information" that has to be kept at the start of the file. In this slightly simplified implementation, the only housekeeping data used is a link (disk address) to the BTree node that represents the current root for the tree, and a count for the number of items.

class BTree

"Client programmers" using a BTree will see it as basically something that owns an index file (the one with the BTree nodes) and a data file, and which provides `Add()`, `Find()`, and `Remove()` operations that efficiently transfer data records (instances of classes derived from `KeyedStorableItem`) to/from the data file. (For convenience, an "add" operation specifying an existing key should be treated as an "update"; the existing data record with the given key is overwritten by the new data.)

```
class BTree
{
public:
    BTree(const char* filename);
    ~BTree();

    int    NumItems(void) const;

    void   Add(KeyedStorableItem& d);
    int    Find(long key, KeyedStorableItem& rec);
    void   Remove(long key);
private:
    …

    fstream       fTreeFile;
    fstream       fDataFile;
};
```

This implementation is simplified. Data items once written to disk will always occupy space. Deletion of a data item simply removes its key/location entry from the index file. Similarly, BTree nodes that become empty and get discarded also continue to occupy space on disk; once again, they are simply unlinked from the index structure.

A real implementation would employ some extra "housekeeping data" to keep track of deleted records and discarded BTree nodes. If there are "deleted records", the next request for a new record can be satisfied by reusing existing allocated space rather than by extending the data file. The implementation of this recycling scheme involves

keeping two lists, one of deleted data records and the other of discarded BTree nodes. The links of these lists are stored in the "deleted" data records of the corresponding disk files (i.e. a "list on a disk"). The starting points of the two "free lists" are included with the other "housekeeping information" at the start of the index file.

24.2.3 BTree: search, insertion, and deletion operations

Rules defining a BTree

All the more elaborate trees have rules that define how their nodes should be organized. You may find minor variations in different text books for the rules relating to BTrees. The rules basically specify:

- A BTree is a tree with nodes that have the data members previously illustrated:

```
int         n_data;       // number of keys in current node
KLRec       data[MAX];    // key-location pairs for data
daddr_t     links[MAX+1]; // links to subtrees
```

- If a node `x` is an internal node, it will hold `x.n_data` keys and `(x.n_data + 1)` links to subtrees. Every internal node contains one more link than it has keys.

- If a node is a leaf, all its link fields are "`NULL`". (i.e. the -1 value for "no disk address", `NO_DADDR`).

- The keys within a node are kept ordered: `x.data.fKey[0] < x.data.fKey[1] < ….`

- The keys in a node separate the ranges of keys stored in subtrees. So `x.link[0]` links to the start of a subtree containing records whose keys will all be less than `x.data.fKey[0]`; `x.link[1]` points to a subtree containing records whose keys `(k)` are in the range `x.data.fKey[0] < k < x.data.fKey[1]`. The final link, `x.link[x.n_data]`, connects to a subtree with records having keys greater than `x.data.fKey[x.n_data]`.

- Every leaf is at the same depth.

- There are lower and upper bounds on the number of keys in a node. Apart from the root node, every node must contain at least `MAX/2` keys and at most `MAX` keys.

  The root node may contain fewer than `MAX/2` keys.

- If an insertion or a remove operation results in a node that violates these conditions, the tree must be reorganized to make all nodes again satisfy these conditions.

Find

Searching the tree for a record with a given key is relatively simple. It involves just a slight generalization of the algorithm suggested earlier for searching a 2-3 tree.

You start by loading the root node of the tree (reading it from the index file into memory). Next you must search in the current node for the key. You stop when you find the key, or when you find a key greater than the value sought. If the key was found, the associated location information identifies where the data record can be found in the data file. The data record can then be loaded and the Find routine can return a success indicator.

If the key is not matched, the search should continue in a subtree (provided that there is a subtree). The search for the key will have stopped with an index set so that it either identifies the position of the matching key or the link that should be used to get to the BTree node at the start of the required subtree.

The driver routine is iterative. It keeps searching until either the key is found, or a "null" link (i.e. a -1 disk address) is encountered in a link field.

Some of the work is done the `BTree::Find()` function itself. But it is worth making a class `BTreeNode` to look after details of links and counts etc. A `BTreeNode`, once loaded from disk, can be asked to check itself for the key.

```
BTree::Find(long key, KeyedStorableItem& rec)
    initialize disk-address to hold address of root node
           (from "housekeeping information" in index file)

    while( disk-address is valid) {
           load a BTreeNode, current, from the specified
                  disk-address
           ask current node to search itself for "key"
           if (key was found)
                  get associated data location, loc
                  GetDataRecord(rec,loc);
                  return success

           otherwise use identified link
            disk-address = current.links[index];
           }
    report key not present
```

Get and check a BTreeNode from disk

If find key, get data from data file

Otherwise, use link with disk address of subtree

Searching inside a node

The `BTreeNode` object would have to find the required key, returning a success or failure indicator. It would also have to set an index value identifying the position of the key (or of the link down to the subtree where the required key might be located). Since the keys in a node are ordered, you should use binary search. For simplicity, a linear search is shown in the following implementation. The loop checks successive keys, incrementing `index` each time, until either the key is found or all keys have been checked. (You should work through the code and convince yourself that, if the key is not present, the final value of `index` will identify the link to the correct subtree).

```
int BTreeNode::SearchInNode (long  keysought, int& index)
{
    for (index = 0; index < n_data; index++)
           if (data[index].fKey == keysought)
                  return 1;
           else if (data[index].fKey > keysought)
                  return 0;
    return 0;
}
```

Add

The algorithm is essentially the same as that illustrated for the 2-3 tree:

```
recursively...

    chase down through links until find position where
           record should go

    if find a record with same key, replace old data record
    else "insert" new record into node
           if record fits, return success
           else
                  split the node
                         have MAX+1 records (MAX already
                                in full node, and the extra
                                record being inserted)
                         leave MAX/2 with lowest keys in
                                existing node
                         put MAX/2 with highest keys in
                                new node
                         return  the median (middle value)
as unwind recursion:
    check if given a median record to insert, if get
           one then insert (with again possible split...)
```

This is another case where a substantial number of auxiliary functions are needed to handle the various different aspects of the work. The implementation given in the next section uses the following functions for class BTree:

```
BTree::Add(KeyedStorableItem& d);      // Interface

BTree::DoAdd(…);                       // Main recursion

BTree::Split(…);                       // Splitting nodes
BTree::SplitInsertLeft(…);             // auxiliary splitting
BTree::SplitInsertMiddle(…);           //     functions
BTree::SplitInsertRight(…)
```

as well as functions in the auxiliary BTreeNode class:

```
BTreeNode::SearchInNode(…)          // Check for key
BTreeNode::InsertInNode(…)          // Simple insertion
BTreeNode::NotFull()                // Check if full
```

Add()

The `Add()` function is the client interface. It sets up the initial call to the main `DoAdd()` recursive function. It also has to deal with the special case of growing a new root for the tree (as illustrated for the 2-3 tree in Figure 24.10):

```
Add
    invoke DoAdd()
           passing it as arguments the new data record, and
           the disk address of the current root of the tree

    test "work flag" returned by DoAdd(),
    if work flag is set create new root as follows
           BTreeNode new_root;
           fill in number of keys as 1,
           insert (median) KLRec returned by DoAdd()
           insert two links, link[0] to link to current root
                  link[1] to link to disk address that
                         DoAdd() reports for newly created node
           Save the new root node to the index file
           Update the "root" info. in the housekeeping part of
                  the index file
```

Recursive DoAdd() function

The recursive function `DoAdd()` is the most complex. It has three aspects. There is an inward recursion aspect; this chases down subtree links through the tree. When the recursion process is complete and the correct point for the record has been found in the tree, the data get saved to disk. The final aspect is organizing the "fix up" operations as recursion is unwind.

Several BTreeNodes on the stack

Each recursive call to `DoAdd()` will load another `BTreeNode` into the stack `BTreeNodes` are going to be a few hundred to a few thousand bytes in size. Since the maximum limit of the recursion is defined by the depth of the tree (which won't be large), this stacking up of the `BTreeNodes` will not use excessive memory. When the insertion point is found, the `BTreeNodes` in the stack are those that define the path back to the root. These are the nodes that may need to be "fixed up" if a node was full and had to be split.

Inward recursion to find place for record

Inwards recursion aims to get to the point in the tree where the new data should go. Since there are now many subtrees below each tree node, one aspect of the inward recursion process is a search through the current node to find the appropriate link to follow for a given key value.

Terminating recursion, by replacing a data record

The inwards recursive phase terminates on either of two conditions. There is the special case of finding an existing record with the key. In this case, the data are

replaced in the data file and a flag is set to indicate that no work is necessary as the recursion unwinds.

Terminating recursion by inserting a new record

The other terminating condition is that the recursive call has been made with a "null" disk address passed as an argument. This means that recursion has reached the bottom of the tree. The new data record should be added to the data file. Its disk address and its key get placed in a `KLRec`. This is returned to the preceding level of recursion for processing.

Unwinding recursion and fixing up records

The final aspect of `DoAdd()` is the mechanism for unwinding recursion. This starts by checking a "work flag" returned by the recursive call. If the flag is not set, function `DoAdd()` can simply return; but if the flag is set then a `KLRec` and link value have to be inserted into the `BTreeNode` in the current stack frame and the updated node must be written back to disk.

Most of the remaining complexities relate to insertions into a `BTreeNode`. These operations are outlined after the complete `DoAdd()` algorithm.

Naturally, like all recursive routines, the termination conditions for `DoAdd()` come first. So the actual structure of the function involves the termination tests, then the setting up of a further recursive call, and finally, after the recursive call, the fix-up code. The algorithm for `DoAdd()` is as follows:

```
DoAdd( arguments include record to be inserted and disk address
          of next node from index file, …)
    Check "disk address" argument passed in call                  Termination
                                                                  conditions
    if (disk address is  NO_DADDR) {                              Check whether "off
          Save new record to data file                            end of tree"
          Update housekeeping info (number of records etc)
          Return details of key for new data record and
                its location in data file, and set flag
                to indicate fix up required
          return;
          }

    Load a BTreeNode from the index file                          Load another tree
          into current stack frame                                node into memory

    Ask current node to search for given key                      Check for key
    if(found) {
          Find location in data file for record with              If find key, terminate
                this key                                          by replacing data
          Replace with new data
          Set flag to say fix up not required
          return;
          }

    // If key not found, 'index' variable will have been set     Recursive call
    //     so as to identify link to subtree
    DoAdd(newData, current.links[index],
                …);
```

Check "work flag" on return from recursion

```
if (fix up flag not set)
        return;
```

Do fix up operations:

```
if (current Btree node is not full)
```

Simple insertion

```
        Insert details of key/disk location into
                current node
else {
```

Or splitting of node

```
        Split the current node,
        Get some key/location records transferred
                to a new tree node and add this to
                index file
        Get a "median record" to be passed by for
                insertion by caller
        Set flag to indicate caller must do fix up.
        }

Have changed current BTreeNode by adding to it, or
by splitting it, so write it to disk
```

Insertion into a partially filled BTreeNode

Insertion of an extra key into a partially filled `BTreeNode` is the simplest case, see Figure 24.15. The `BTreeNode` can be given details of the new key (more strictly, a `KLRec`, key/location pair, defining the data item), and the position where this is to go in the node's array of `KLRecs`. Existing entries with keys that are greater in value should be moved to the right in the array to make room; the new `KLRec` entry can be inserted and the count of entries incremented. Every time a `BTreeNode` is changed, it has to be written back to the index file.

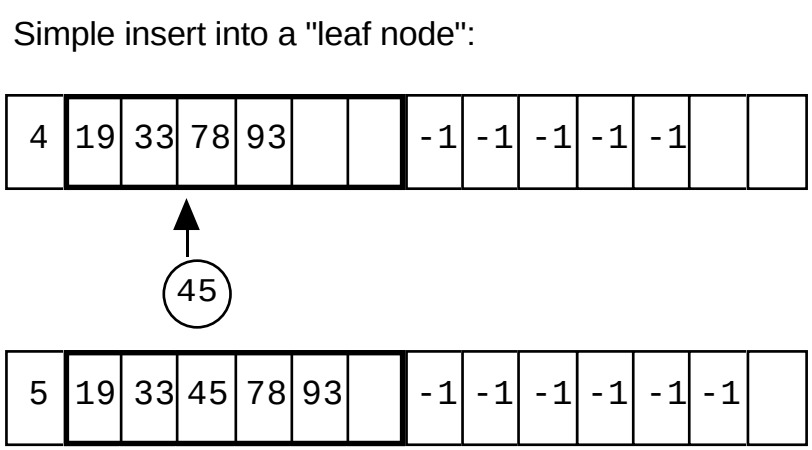


Figure 24.15 Simple insertion into a partially filled node.

If the insertion is into an "internal" BTreeNode, then there will be an extra link that has to be inserted in addition to the KLRec. This extra link is the disk address of an extra BTreeNode that results from a "split" operation at a lower level..

The BTreeNode can have a single member function that deals with insertions of a KLRec and associated extra link (the extra link argument will be -1 in the case of insertion into a leaf node):

```
void BTreeNode::InsertInNode(KLRec& info, daddr_t  diskpos,
            int index)
{
    for (int i = n_data - 1; i >= index; i--) {
            data[i+1] = data [i];
            links[i+2] = links[i+1];
            }
    data[index] = info;
    links[index+1] = diskpos;
    n_data++;
}
```

Move existing entries right to make room

Insert key/location pair and link to subtree

Insertion into a full BTreeNode – splitting the node

If a BTreeNode already has MAX keys, then it has to be split, just like a full 2-3 node would get split. There will be a total of MAX+1 keys (the MAX keys already in the node and the extra one). Half (those with the lowest values) are left in the existing node; half (those with the greatest values) are copied into a newly created BTreeNode, and one, the median value, gets moved up into the parent BTreeNode.

The basic principles are as illustrated in Figure 24.16. This shows an insertion into a full leaf node (all its subtree links are -1). An extra BTreeNode is created (shown as "node02"). Half the data are copied across. Both nodes are marked as half empty. Then, both would be written to disk (the new BTreeNode, "node02", going at the end of the index file, the original BTreeNode, "node01", being overwritten on disk with the updated information). The median key, and the disk address for the new "node02" would then have to be inserted into the BTreeNode that is the parent of node01 (and now of node02 as well).

There are really three slightly different versions of this split process. In the first, the extra key has a low value and it gets inserted into the left (original) node. In the second, the new key is the median value; it gets moved to the parent. Finally, there is the situation where the new key is large and it belongs in the right (new) node. The reshuffling processes that move data around are slightly different for the three cases and so are best handled by separate auxiliary functions.

The overall BTree::Split() function has to be organized along the following lines:

```
Split
    given: a BTreeNode to be split "node_to_split"
           an extra KLRec key/data-location pair
           an extra link (disk address of a BTreeNode that
                  is to become a subtree of "node_to_split"
```

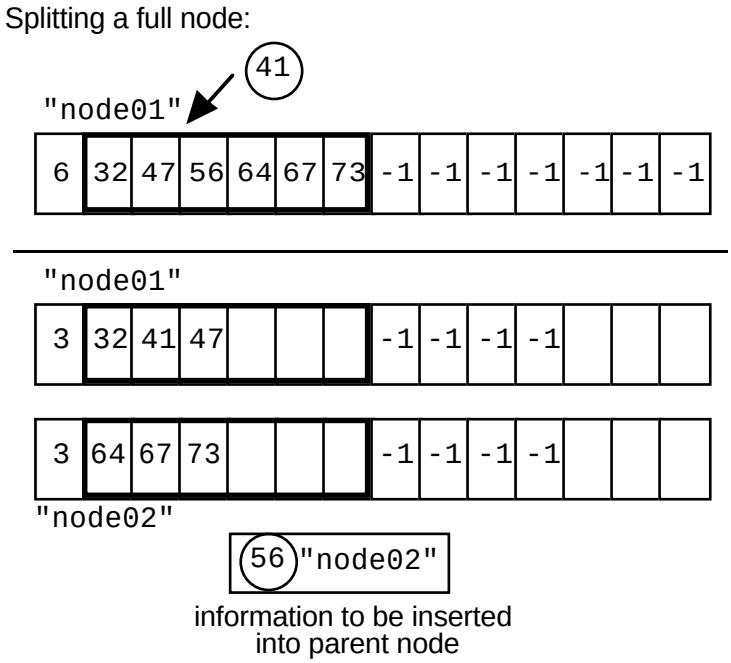


Figure 24.16 Splitting a BTreeNode.

```
        and an index specifying where new entry to go.
    if (index > MIN)
        use an auxiliary "Insert Right" function to
            get new key into right node
    else
    if (index == MIN)
        use an auxiliary "Insert Middle" function to
            split the node, sharing the existing entries
            between old and new parts
            keeping the new key as "median"
    else
        use an auxiliary "Insert Left" function to
            get new key into left node
    return
        Median value to get put into parent node
        disk address of the newly created node.
```

The auxiliary functions have loops that shuffle `KLRec` records and links between the old and new `BTreeNode` records. Though the code is fairly simple in structure, there are lots of niggling little details relating to which link values end up in the link arrays of the two nodes.

The basic operations are as shown in Figure 24.17 for the case of inserting the new data in the right hand node; the process is:

```
Insert in Right
    given: a BTreeNode to be split "node_to_split"
           an extra KLRec key/data-location pair
           an extra link (disk address of a BTreeNode that
                  is to become a subtree of "node_to_split"
           an index specifying where new entry to go

    BTreeNode newNode; // BTreeNode created on stack
    for (i = MAX-1, j = MIN-1; i >= index; i--, j--) {          Copy1
           newNode.links[j+1] = nodetosplit.links [i+1];
           newNode.data [j] = nodetosplit.data [i];
           }
    newNode.links [j+1] = extralink;                            Insertion
    newNode.data[j] = extradata;
    for (j--; i > MIN; i--, j--) {                              Copy2
                  newNode.links[j+1] = nodetosplit.links [i+1];
                  newNode.data[j] = nodetosplit.data [i];
                  }
    newNode.links[0] = nodetosplit.links[MIN+1];
    newNode.n_data = nodetosplit.n_data = MIN;                  Clean up
    return_diskpos = MakeNewDiskBNode(newNode);

    return
           Median value to get put into parent node
                  (nodetosplit.data [MIN])
           disk address of the newly created node.
                  (return_diskpos)
```

The operations all take place on a temporary `BTreeNode` created in the stack (as a local variable of the "insert in right" function). When this has been filled in successfully, an auxiliary function gets it into the data file (at the end of the file) and returns its disk address for future reference.

The `KLRec` with the median valued key, and the address of the extra `BTreeNode`, are passed back to the calling level of the recursion. There they have to be inserted into the parent, which may again split. The process is identical in concept to that shown in Figures 24.8, 24.9 and 24.10 for the 2-3 trees.

As also illustrated previously for the 2-3 trees, if there is no parent node, a new root node has to be created for the tree. This process is illustrated in Figure 24.18.

Initially a single full BTreeNode

The figure shows a BTree index file that initially has a single full node containing the keys 20, 33, 45, 56, 67, and 79 (links to data records in the datafile are not shown). There are no subtrees; so all the BTree structure link fields are -1; and the count field is 6. The `BTreeNode` is assumed to be 80 bytes in size; starting at byte 4 (after a minimal housekeeping record that contains solely the byte address of the first record).

Insertion of key 37 will force the record to split as there would now be seven keys and these nodes have a maximum of six. The three highest keys (56, 67, and 79) are

shifted into a new `BTreeNode` (see Figure 24.18). This would start at byte 84 of the disk file. The node would be assembled in a structure on the stack and then be written to the disk file.

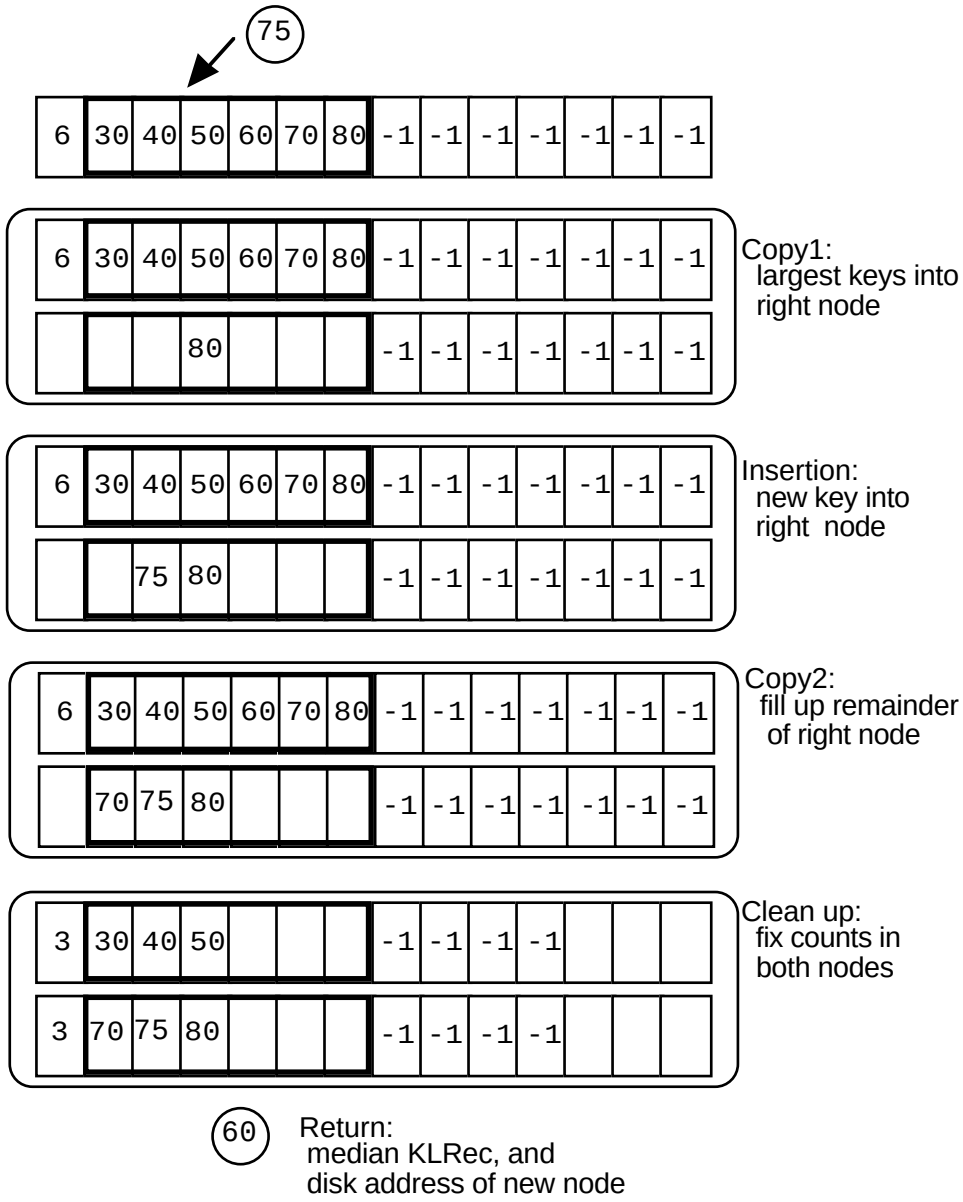


Figure 24.17 Reorganizing a pair of BTreeNodes after a "split".

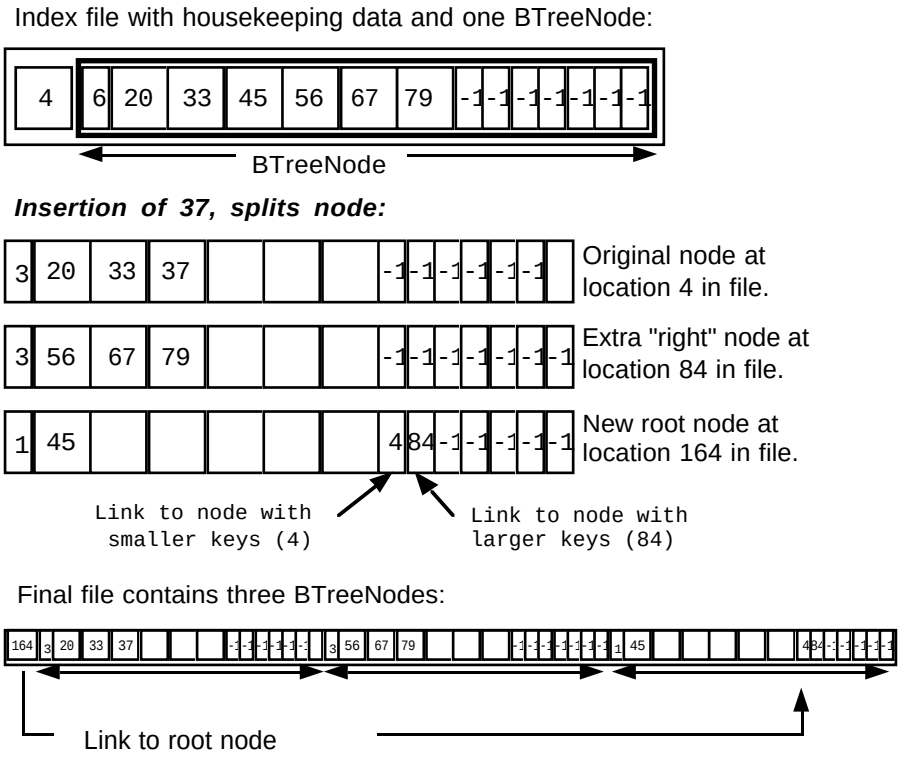


Figure 24.18 Splitting leading to a new root node.

The three lowest keys (20, 33, 37) would go in the original `BTreeNode` starting at byte 4 of the file. This `BTreeNode` would first be composed in memory and then would overwrite the existing record on disk.

The median, with key 45, would have to go into a new root `BTreeNode`. It would need links down to the original `BTreeNode` at 4 (`link[0]`, all the keys less than 45) and the `BTreeNode` just created at location 84 in the file (`link[1]`, all the keys greater than 45). The new root `BTreeNode` would get written to the file starting at byte 164.

Finally, the housekeeping data at the front of the file would be updated to hold the disk address of the new root node.

Remove

As already noted, records are removed in much the same way as in AVL trees. A recursive routine hunts down through the tree to find the key for the record that is to be removed. (If the key is not found, the routine returns some failure indication.) As it

recurses down through the tree, the routine loads `BTreeNodes` from the file into the stack, as in previous examples these define the path back to the root and are the nodes that may need to be fixed up.

If the key is found in an internal node, data must be promoted from a leaf node (the implementation in the next section promotes the successor key – the smallest key greater than the key to be deleted). Once the promotion has been done, the original promoted data must be deleted. So the routine further recurses down through the tree until it has the leaf node from where data were taken.

All actual deletions take place on leaf nodes (either because the key to be deleted was itself in a leaf node, or because a key was taken from a leaf node to replace a key in an internal node). A deletion reduces the number of keys in the node. The remaining keys are rearranged to close up the space left by the key that was removed. If there are still at least `MAX/2` keys in the leaf, then essentially everything is finished. The node can be written back to the file. Recursion can simply unwind (if an internal node was modified by having data replaced with promoted data, then it gets written to the file during the unwinding process).

Deficient nodes need "fixing up"

The difficulties arise when a node gets left with less than `MAX/2` keys. Such a node violates the BTree conditions (unless it happens to be the root node); it is termed a "deficient node". A deficient node can't do anything to "fix itself up". All it can do is report to its parent node. This is achieved, in the recursive procedure, by a node that detects deficiency setting a return flag; the flag is checked at the next level above as the recursion unwinds.

If a parent node sees that a child node has become deficient, it can "fix up" that child node by shifting data from "sibling nodes". There are a couple of different situations that must be handled. These are illustrated in Figures 24.19 and 24.20 .

Moving data from a sibling node

Figure 24.19 illustrates a "move" operation. The initial tree (shown in Pane 1 of Figure 24.19) would have five nodes (only four are shown). The root has three keys (300, 400, and 500) and four links down to subtrees. The first subtree (not shown) would contain the keys less than 300. The second subtree, node n1, contains keys greater than 300 and less than 400. The third subtree (in `link[2]`) has the keys between 400 and 500. The final subtree has the keys greater than 500.

Node n2 has exactly `MAX/2` keys. If one of its keys is removed, e.g. 440, it is left "deficient" (Pane 2 of Figure 24.19). It cannot do anything to fix itself up. But, it can report to its problem to its parent node (which in this case is the root node).

The root node can examine the sibling nodes (n1 and n3) on either side of the node that has just become deficient. Node n1 has four keys (> `MAX/2`). The deficiency in node n2 could be made up by "transferring a key" from n1. That would in this case leave both nodes n1 and n2 with exactly `MAX/2` keys.

But of course, you can't simply transfer a key across between nodes because the keys also have to be in order. There has to be a key in the root node such that it is greater than all keys in its left subtree and smaller than all keys in its right subtree.

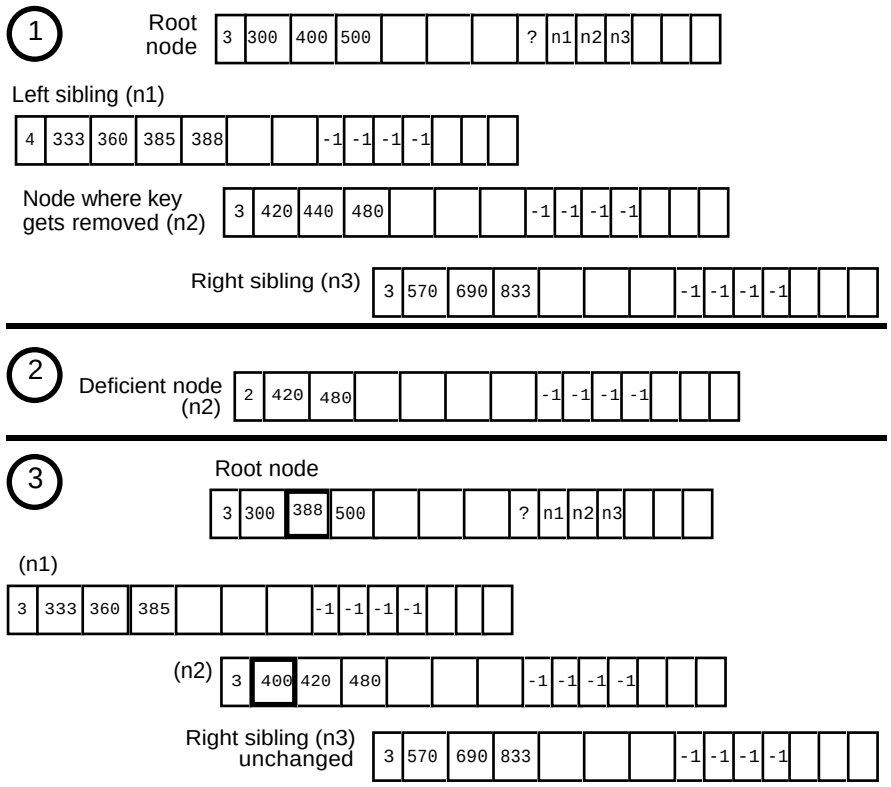


Figure 24.19 Moving data from a sibling to restore BTree property of a "deficient" BTreeNode.

The "transfer of a key" actually involves taking the largest key in a left sibling (or smallest key in a right sibling) and using this to replace a key in the parent. The key from the parent is then used to restore the deficient node to having at least the minimum number of keys.

In the example shown (Pane 3 of Figure 24.19), the key 388 is taken from node n1 (leaving it with three keys) and moved up into the parent (root) node where it replaces the key 400. Key 400 is moved down into node n2.

Everything has been restored. The link down "between" keys 300 and 388 (`link[1]` of the root node) leads to all keys in this range (i.e. to node n1 with keys 333, 360 and 385). The link down between keys 388 and 500 leads to the "subtree" (i.e. node n2) with keys between 388 and 500 (keys 400, 420, and 480). All nodes continue to satisfy the BTree requirements on their minimum number of keys.

For a "move" or transfer to take place, at least one of the siblings of a deficient node must have more than `MAX/2` keys. Move operations can take a key from the left sibling or the right sibling of a deficient node. Of course, if the deficient node is on `link[0]`

of its parent then it has no left sibling and a move can only occur from a right sibling. If the deficient node is in the last subtree link of its parent, only a move from a left sibling is possible. If a node has both left and right sibling, and both siblings have more than `MAX/2` keys, then either can be used. In such cases, it is best to move a key from the sibling that has the most keys.

Combine operations

Sometimes, both siblings have just the minimum `MAX/2` keys. In such situations, "move" operations cannot be used. Instead, there is another way of "fixing up" the node. This alternative way combines all the existing keys in the deficient node and one of its siblings into a single node and ceases to use one of the `BTreeNodes` in the file. Figure 24.20 illustrates a combine operation.

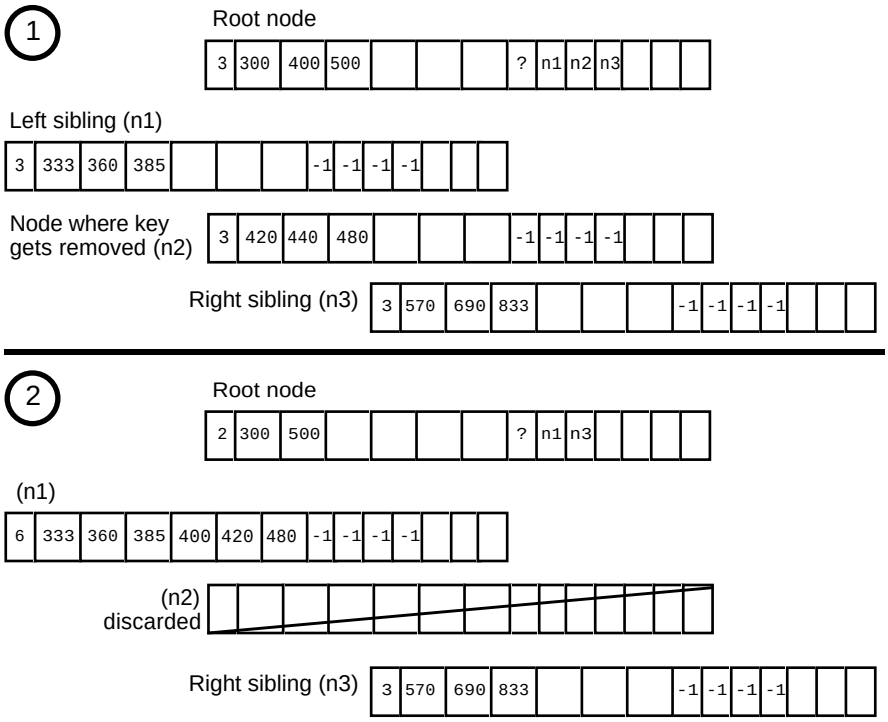


Figure 24.20 Combining BTreeNodes after removing a key.

Since a `BTreeNode` has to be removed, there will be one fewer link down from the parent. Since all the links in a `BTreeNode` must either be `NULL` or links down to subtrees, this means that the keys in the parent node have to be squeezed up a bit. There will be exactly `MAX/2` keys from a sibling, `MAX/2 - 1` keys from the node that became deficient. These are combined along with one key from the parent to produce a full `BTreeNode`.

In the example shown in Figure 24.20, the initial state has each of the three nodes n1, n2, and n3 with three keys. Removal of key 440 from n2 leaves it deficient. The parent can not shift a key from either n1 or n3 for that would leave the donor deficient. So, instead, key 400 from the parent, and the remaining keys 420 and 480 are shifted into n1, filling it up so that it has six keys. (Alternative rearrangements are possible; for example, key 500 from the parent and the three keys from node n3 could be shifted into n2 to fill it up and leave n3 empty. It doesn't matter which rearrangement is used.)

The `BTreeNode` that becomes empty, n2 in Figure 24.20, is "discarded". It is no longer linked into the tree structure. (In a simple implementation, it becomes "dead space" in the file; it continues to occupy part of the disk even though it isn't again used.)

As shown in Figure 24.20, the parent node now only has three links down. One goes to a node (not shown) with keys less than 300. The second is to the full node n2 with all the keys between 300 and 500. The third link is to the node, n3, with the keys greater than 500. This leaves the parent node with just two keys.

Since the parent provides one key, it may become "deficient". If the parent becomes deficient, it has to report this fact to its parent during the unwinding of the recursion. A move or combine operation would then be necessary at that level. Removal of a key from a leaf can in some circumstances cause changes at every node on the path back to the root.

The root node is allowed to have fewer than `MAX/2` keys. If the root node is involved in a combine operation, it ends up with fewer keys. Of course, it is possible to end up with no keys in the current root! Figure 24.21 illustrates such an occurrence.

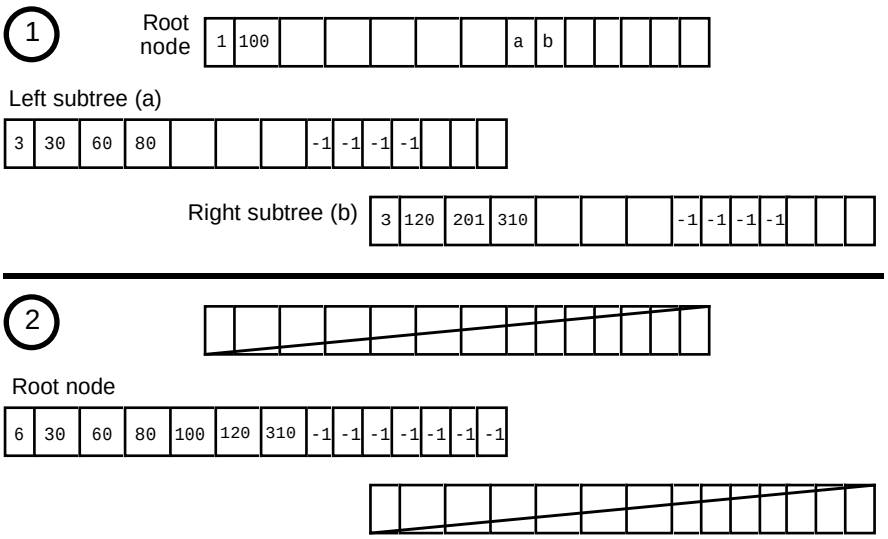


Figure 24.21 Removal of a key may lead to a change of root node.

In this case, two `BTreeNodes` get discarded and the tree is "re-rooted" on the node that originally formed the top of the left subtree. The "housekeeping" information at the start of the BTree index file would have to be updated to reflect such a changed.

Overall removal process

Removal of a record thus involves:

- finding the leaf node with the key (if the key is in an internal node, a promotion operation must be done and then the original copy of the promoted data must be found in a leaf node);
- removal of the key from the leaf;
- unwinding the recursive search, performing "fix ups" on any nodes that become deficient;
- nodes get "fixed up" by parents performing move or combine operations affecting the deficient node, a sibling, and the parent node.

Unlike `Remove()` for the AVL tree which returns the address of a data record in memory, `BTree::Remove(…)` simply performs the action. (The data record is in the data file. It doesn't actually get destroyed; the reference to it in the index is removed making it inaccessible. Again, the space it occupies becomes "dead space" on disk).

Obviously, many auxiliary functions have to be used in the implementation of `BTree::Remove()`. The implementation given in the following section has the following functions:

```
BTree::Remove(long key);          // interface
BTree::DoRemove(…);               // main recursive routine
BTree::DeleteKeyInNode(…);        // removal of key from node
BTree::DeleteInLeaf(…;            //   special case, leaf node
BTree::Successor(…);              // get data to replace key in
                                  internal node
BTree::Restore(…);                // Organize fix up of deficient
                                  child node
BTree::MergeOrCombineRight(…);    // Merge child and right sibling
BTree::MergeOrCombineLeft(…);     // Merge child and left sibling
BTree::MoveRight(…);              // Move key out of left node
BTree::MoveLeft(…);               // Move key out of right node
BTree:Combine(…);                 // Combined deficient node and
                                  sibling
```

In addition, the implementation relies on several functions of class `BTreeNode`:

```
BTreeNode::SearchInNode(…);       // Finding key
BTreeNode::InsertAtLeft(…);       // Shifting keys into node
BTreeNode::InsertAtRight(…);
BTreeNode::ShiftLeft(…);          // Moving keys around
BTreeNode::Compress(…);
BTreeNode::Deficient(…);          // Checking status of node
BTreeNode::MoreThanMinFilled(…);
```

Remove() driver function

Function `Remove()` provides the interface; its only argument is the key corresponding to the data record that must be deleted. The function has to check that a tree exists, someone might try to delete data before any have been entered. If there is a BTree to search, the `Remove()` function should load in the root node and set up the call to the main recursive `DoRemove()` function. When `DoRemove()` returns, a check should be made for the special case of needing a new root node (the situation illustrated in Figure 24.21).

```
Remove(key)
    if(there is no tree!)
           return;

    Load current root_node

    DoRemove(key, root_node);
    if (root_node has no keys left)
           set housekeeping data to record new root
    else
           save root_node back on disk
```

Main recursive DoRemove()

Function `DoRemove()` is a recursive function with some parallels to the `DoAdd()` function already considered. It has to recursively chase down through the tree to find the key. There has to be a check to stop recursion. As recursion unwinds, any necessary fix up operations are performed.

Termination of recursion and handling of delete

At each recursive level, the `BTreeNode` to be worked on has already been loaded at the previous level (e.g. `Remove()` loads the root node). Since the `BTreeNode` is already on the stack, it can be checked for the key; if the key is present the auxiliary `DeleteKeyInNode()` function is called to remove it (this will involve a further recursive call to `DoRemove()` if the node is an internal node). When the deletion has been done, function `DoRemove()` can return. Function `DoRemove()` returns a flag indicating whether it has left a node deficient.

Setting up a recursive call

If the key was not found, the function has to set up a further recursive call using the appropriate link down to a subtree. If it encounters a "null" link, this means that the specified key was not present, in that case the function can simply return. Usually, there will be a valid disk address in the link. The `BTreeNode` at this location in the index file should be loaded onto the stack and the recursive call gets made.

Unwinding recursion

The unwinding process uses the auxiliary function `Restore()` to fix up any deficient nodes. Other nodes that may have been modified just get written back to the index file.

```
DoRemove(long bad_key and reference to a BTreeNode on the
stack)
    Get node to search for the key
    if(found)
           Delete bad_key from this node
           update count of keys in housekeeping records
```

Termination of recursion

```
        return indication of whether node was left deficient
```

Setting up recursive call

```
    Get link to subtree
    if(subtree == NO_DADDR) return 0;

    Load next BTreeNode onto stack
```

Recursive call

```
    repairsneeded = DoRemove(bad_key, nextNode);
```

Unwinding recursion and fixing up

```
    if (repairsneeded)
            Restore(…);
    else
            SaveBTreeNode(nextNode, subtree);

    return indication of whether node was left deficient
```

Deletion of a key

The function `DeleteKeyInNode()` sorts out how to delete a key. The auxiliary function `DeleteInLeaf()` deals with the easy case of deletion in a leaf (leaves are easy to recognize, all subtree links are "null"). (Deletion in a leaf is trivial; all higher valued keys are moved one place left so overwriting the key that has to be removed. The count field for the node is then decremented.) If the node is an internal node, the auxiliary function `Successor()` is employed to get the key/location pair of next higher key (the lowest valued key in the right subtree). Then, `DoRemove()` must be called recursively to get rid of the original copy of the promoted key/location pair.

```
DeleteKeyInNode(node to work on and index of key to be deleted
    if (subtree links are null)
                DeleteInLeaf(aNode, index);
                return;
```

Promotion of successor

```
    Replace entry with data promoted from
            Successor( right subtree);

    Load node at top of right subtree

    Use DoRemove to remove promoted data from right subtree

    Fixup
```

Restore() function

The `Restore()` function has to determine whether the deficient node is the leftmost child (in which case can only merge with right sibling), or the rightmost child (can only merge with a left sibling), or an intermediate case with both left and right siblings. If both siblings exist, the merge operation should use the sibling with more keys. The checks involve loading the sibling nodes into memory.

```
Restore(parent node, deficient node, index of parent's link
            down to deficient node)
    if(index == 0)
```

```
            MergeOrCombineRight(…);
    else
    if(index == parent.n_data)
            MergeOrCombineLeft(…);
    else
            Load left sibling into a temporary BTreeNode

            int left_num = temp.n_data;

            Load right sibling into a temporary BTreeNode

            int right_num = temp.n_data;

            if(left_num >= right_num)
                    MergeOrCombineLeft(…);
            else
                    MergeOrCombineRight(…);
```

MergeOrCombine

The "MergeOrCombine Left / Right" functions check the occupancy of the selected sibling node. If it has sufficient keys, a "move" is done; otherwise the more complex combine operation is performed. The `MoveLeft()` function is similar in organization.

MoveRight()

The `MoveRight()` operation takes a `KLRec` (key/data location pair) from the parent and the rightmost link down from the left node and inserts these into the deficient node (the `BTreeNode` does the actual insertion, it moves all existing entries one place right then inserts the extra data). Then the rightmost `KLRec` is moved from the left node up into the parent.

Figures like 24.19 were simplified in that the nodes operated on were leaf nodes. Often they will be internal nodes with links down to lower levels. Thus, to the right of key 388 there would be a link down to a subtree with all keys greater than 388 and less than 400. This link down would have to be moved into `link[0]` of the deficient node.

```
MoveRight(parent node, two siblings, and index position)
    Get KLRec from parent at indexed position
    Get last down link from left node
    Get other BTreeNode to insert KLRec and link
    Replace parent KLRec with rightmost data from left node
    decrement count field in left node
```

The `Combine()` shifts a KLRec from the parent and remaining data from the other node.

24.2.4 An implementation

The header file, BTree.h, would contain the declaration for the pure abstract class `KeyedStorableItem` along with the main class `BTree`:

```
class KeyedStorableItem {
public:
    virtual       ~KeyedStorableItem() { }
    virtual long  Key(void) const = 0;
    virtual void  PrintOn(ostream& out) const  { }
    virtual long  DiskSize(void) const = 0;
    virtual void  ReadFrom(fstream& in) = 0;
    virtual void  WriteTo(fstream& out) const = 0;
};

inline ostream& operator<<(ostream& out,
                  const KeyedStorableItem& d)
{ d.PrintOn(out); return out; }
inline ostream& operator<<(ostream& out,
           const KeyedStorableItem* p_d)
{ p_d->PrintOn(out); return out; }
```

A `KeyedStorableItem` is essentially something that can report its key and transfer itself to/from a disk file.

The BTree code is parameterized according to the number of keys in each node. This number needs to be small during testing but should be enlarged for a production version of the code.

Class `BTree` makes use of `BTreeNode` objects and `KLRec` objects and its functions have pointers and references of these types. They are basically a detail of the implementation so their definitions go in the ".cp" implementation file. The types however must be declared in the header:

```
#define    MIN    3
#define    MAX    (2 * MIN)

class      BTreeNode;
struct     KLRec;
```

Class `BTree` has a simple public interface. The constructor takes a string that will be the "base name" for the index and data files (e.g. if the given name is "test", the files used will be "test.ndx" and "test.dat").

Public interface of class BTree

```
class BTree
{
public:
    BTree(const char* filename);
    ~BTree();

    int    NumItems(void) const;

    void   Add(KeyedStorableItem& d);
    int    Find(long key, KeyedStorableItem& rec);
    void   Remove(long key);
```

All the complexity exists in the private implementation part.

The implementation needs some structure to represent the "housekeeping data" In this simple implementation, this consists of the root address and a count of items in the collection. The declarations of the various auxiliary functions come after the declaration of this housekeeping structure.

Auxiliary private member functions for disk transfers

The first group of member functions deal with disk transfers. The `BTree` object can look after reading and writing its housekeeping information and its `BTreeNodes`. (The nodes could have been made responsible for reading and writing themselves, it doesn't make much difference). `KeyedStorableItem` objects are responsible for their own data transfers, but class `BTree` is responsible for the files. It is the `BTree` object that must perform the seek operations used to position the read/write file pointers before another object is asked to transfer itself. The implementations for most of these functions are simple, just a seek followed by a request to some other object to read or write itself.

The Get and Save functions use existing entries in the files. The MakeNew functions add extra entries at the end of files (an extra `BTreeNode` or an extra data record as appropriate). In a more sophisticated implementation, the MakeNew functions could be enhanced to reuse "dead space" left where data records or `BTreeNodes` have been deleted.

```
private:

    struct HK {
           daddr_t       fRoot;
           long   fNumItems;
    };

    /* Disk i/o group */
    void   GetBTreeNode(BTreeNode& bnrec, daddr_t diskpos);
    void   SaveBTreeNode(BTreeNode& bnrec, daddr_t diskpos);
    void   GetDataRecord(KeyedStorableItem& datarec,
                               daddr_t diskpos);
    void   SaveDataRecord(KeyedStorableItem& datarec,
                               daddr_t diskpos);
    daddr_t       MakeNewDiskBNode(BTreeNode& bnode);
    daddr_t       MakeNewDataRecord(KeyedStorableItem& data);

    void          SaveHK(void);
    void          LoadHK(void);
```

The next declarations will be of all the auxiliary functions needed for the Add operation followed by all the auxiliary functions needed for the Remove operation. The main recursive `DoAdd()` function has a complex argument list because it most pass back data defining any new information that needs to be inserted into a node (the reference arguments like `return_diskpos`).

Auxiliary functions for Add

```
    void    DoAdd(KeyedStorableItem& newData, daddr_t filepos,
                        int& return_flag, KLRec& return_KLRec,
                        daddr_t& return_diskpos);
    void    Split(BTreeNode& nodetosplit,
                        KLRec& extradata, daddr_t extralink,
                        int index, KLRec& return_KLRec,
                        daddr_t& return_diskpos);
    void    SplitInsertLeft(BTreeNode& nodetosplit,
                        KLRec& extradata, daddr_t extralink,
                        int index, KLRec& return_KLRec,
                        daddr_t& return_diskpos);
    void    SplitInsertMiddle(BTreeNode& nodetosplit,
                        KLRec& extradata, daddr_t extralink,
                        int index, KLRec& return_KLRec,
                        daddr_t& return_diskpos);
    void    SplitInsertRight(BTreeNode& nodetosplit,
                        KLRec& extradata, daddr_t extralink,
                        int index, KLRec& return_KLRec,
                        daddr_t& return_diskpos);
```

Auxiliary functions for Remove()

```
    int     DoRemove(long badkey, BTreeNode& cNode);
    void    DeleteKeyInNode(BTreeNode& aNode, int index);
    KLRec   Successor(daddr_t subtree);
    void    DeleteInLeaf(BTreeNode& leaf, int index);

    void    Restore(BTreeNode& parent, BTreeNode& deficient,
                int index);
    void    MergeOrCombineRight(BTreeNode& parent,
                BTreeNode& deficient, int index);
    void    MergeOrCombineLeft(BTreeNode& parent,
                BTreeNode& deficient, int index);

    void    MoveRight(BTreeNode& parent, BTreeNode& left,
                BTreeNode& right, int index);
    void    MoveLeft(BTreeNode& parent, BTreeNode& left,
                BTreeNode& right, int index);

    void    Combine(BTreeNode& parent, BTreeNode& left,
                        BTreeNode& right, int index);
```

Data Members

Once all the auxiliary functions have been declared, the data members can be specified. A `BTree` object needs somewhere to store its housekeeping information in memory, two input/output file streams, and records of the size of the files.

```
    HK          fHouseKeeping;
    fstream     fTreeFile;
    fstream     fDataFile;
    long        fTreefile_size;
    long        fDatafile_size;
};
```

The implementation file would start with the full declarations for struct KLRec (given earlier) and class BTreeNode:

```
class BTreeNode {
    friend class BTree;
private:

    int           SearchInNode(long keysought, int& index);
    void          InsertInNode(KLRec& data, daddr_t,  int);
    void          InsertAtLeft(KLRec& data, daddr_t downlink);
    void          InsertAtRight(KLRec& data, daddr_t downlink);
    void          ShiftLeft(void);
    void          Compress(int index);

    int           NotFull() { return (n_data==MAX) ? 0 : 1; }
    int           Deficient() { return (n_data<MIN) ? 1 : 0; }
    int           MoreThanMinFilled()
                         { return (n_data>MIN) ? 1 : 0; }

    int           n_data;
    KLRec         data[MAX];    /* 0..n_data-1 are filled */
    daddr_t       links[MAX+1]; /* 0..n_data are filled */
};
```

Class BTree is made a friend of BTreeNode so that code of member functions of class BTree can work directly with things like the n_data count or the links[] array.

Implementation of class BTreeNode

The member functions of class BTreeNode are all simple (some are just inline functions in the class declaration). Most of the rest involve iterative loops running through the entries. Functions InsertInNode() and Search() were both shown earlier.

The InsertAtLeft() and InsertAtRight() functions are used during move operations when fixing up deficient nodes. The InsertAtLeft() moves existing data over to make room, adds the new data and increments the counter. The InsertAtRight() function (not shown) is simpler; it merely adds the extra data in the first unused positions and then increments the counter.

```
void BTreeNode::InsertAtLeft(KLRec& info, daddr_t downlink)
{
    for(int i = n_data - 1; i >= 0; i--) {
           data[i+1] = data[i];
           links[i+2] = links[i+1];
           }
    links[1] = links [0];
    data[0] = info;
    links[0] = downlink;
```

```
    n_data++;
}
```

Function `ShiftLeft()` moves `KLRec` and link entries leftwards after the leftmost entry has been removed. It gets used to tidy up a node after its least key has been removed during a "move" operation required to fix up a deficient sibling. Function `Compress()` is used to tidy up a parent node after one of its keys (that identified by argument index) has been removed as part of a Combine operation.

```
void BTreeNode::ShiftLeft(void)
{
    n_data--;
    links[0] = links [1];
    for(int i = 0; i < n_data; i++) {
           data[i] = data[i+1];
           links[i+1] = links[i+2];
           }
}

void BTreeNode::Compress(int index)
{
    n_data--;
    for(int i = index; i < n_data; i++) {
           data[i] = data[i+1];
           links[i+1] = links[i+2];
           }
}
```

Implementation of class BTree's constructor and destructor

The constructor has to make up the names for the index and data files and then open them for both input and output. If the files can not be opened, the program should terminate (you could make the code "throw an exception" instead, see Chapter 29).

```
BTree::BTree(const char* filename)
{
    char   buff[100];
    strcpy(buff,filename);
    strcat(buff,".ndx");
    fTreeFile.open(buff, ios::in | ios::out);
    if(!fTreeFile.good()) {
           cerr << "Sorry, can't open BTree index file." << endl;
           exit(1);
           }
    strcpy(buff,filename);
    strcat(buff,".dat");
    fDataFile.open(buff, ios::in | ios::out);
    if(!fDataFile.good()) {
```

Opening the files

```
            cerr << "Sorry, can't open BTree data file." << endl;
            exit(1);
            }
```

If the files have zero length, then the program must be creating a new tree; otherwise, details of current size and location of current root node must be obtained from the index file. The housekeeping data record needs to be set appropriately:

```
    fTreeFile.seekg(0, ios::end);                                   Get file size
    long len = fTreeFile.tellg();
    if(len == 0) {
            fHouseKeeping.fNumItems = 0;                            Initialize for new
            fHouseKeeping.fRoot = NO_DADDR;                         BTree file
            SaveHK();
            fTreefile_size = sizeof(HK);
            fDatafile_size = 0;
            }
    else {
            LoadHK();                                               Load data
            fTreefile_size = len;                                   characterising
            fDataFile.seekg(0, ios::end);                           existing BTree file
            fDatafile_size = fDataFile.tellg();
            }
}
```

The destructor, not shown, should save the housekeeping details and then close the two data files.

Implementation of class BTree's disk transfer functions

These functions are all very similar, so only a few representative examples are shown:

```
void BTree::SaveHK(void)
{
    fTreeFile.seekp(0);
    fTreeFile.write((char*)&fHouseKeeping, sizeof(HK));
}

void BTree::GetBTreeNode(BTreeNode& bnrec, daddr_t diskpos)
{
    fTreeFile.seekg(diskpos);
    fTreeFile.read((char*)&bnrec, sizeof(BTreeNode));
}

void BTree::GetDataRecord(KeyedStorableItem& datarec,
    daddr_t diskpos)
{
```

```
    fDataFile.seekg(diskpos);
    datarec.ReadFrom(fDataFile);
}

daddr_t BTree::MakeNewDiskBNode(BTreeNode& bnode)
{
    SaveBTreeNode(bnode, fTreefile_size);
    daddr_t diskpos = fTreefile_size;
    fTreefile_size += sizeof(BTreeNode);
    return diskpos;
}

daddr_t BTree::MakeNewDataRecord(KeyedStorableItem& data)
{
    SaveDataRecord(data,fDatafile_size);
    daddr_t       diskpos = fDatafile_size;
    fDatafile_size += data.DiskSize();
    return diskpos;
}
```

Implementation of class BTree::Find()

The `Find()` function is given a key and a reference to a `KeyedStorableItem` (of course this will really be a reference to an instance of some class derived from class `KeyedStorableItem`). If `Find()` can find the key in the index file, it arranges for the `KeyedStorableItem` to load itself. Function `Find()` returns a 0/1 indicator (1 for success, 0 for failure i.e. key not present).

The function is a straightforward implementation of the iterative tree walk algorithm shown earlier:

```
int BTree::Find(long key, KeyedStorableItem& rec)
{
    daddr_t diskpos = fHouseKeeping.fRoot;
    while( diskpos != NO_DADDR) {
           int    index;
           BTreeNode     current;
           GetBTreeNode(current, diskpos);
           if (current.SearchInNode(key, index))  {
                  daddr_t loc = current.data[index].fLocation;
                  GetDataRecord(rec,loc);
                  return 1;
                  }
            diskpos = current.links[index];
           }
    return 0;
}
```

Implementation of class BTree::Add() and related functions

The `Add()` function itself is a straightforward implementation of the algorithm given earlier:

```
void BTree::Add(KeyedStorableItem& d)
{
    KLRec         rec_returned;
    daddr_t       filepos_returned;
    int           workflag;
    DoAdd(d, fHouseKeeping.fRoot, workflag,
                  rec_returned, filepos_returned);
    if(workflag != 0) {
           BTreeNode new_root;
           new_root.n_data = 1;
           new_root.links[0] = fHouseKeeping.fRoot;
           new_root.data[0] = rec_returned;
           new_root.links [1] = filepos_returned;
           fHouseKeeping.fRoot = MakeNewDiskBNode(new_root);
           }
}
```

Make initial call to DoAdd() passing root node

Create new root if necessary

Recursive DoAdd()

The recursive `DoAdd()` function has two input arguments and three output arguments. The input arguments are the new `KeyedStorableItem` (which is passed by reference) and the disk address of the next `BTreeNode` that is to be considered. The output arguments (all naturally passed by reference) are the flag variable (whose setting will indicate if any fix up operation is needed) together with any `KLRec` and disk address that needed to be returned to the caller.

```
void BTree::DoAdd(KeyedStorableItem& newData, daddr_t filepos,
           int& return_flag, KLRec& return_KLRec,
           daddr_t& return_diskpos)
{
    int           index;
    BTreeNode     current;
    long          key = newData.Key();

    return_flag = 0;

    if (filepos == NO_DADDR) {
           return_KLRec.fKey = key;
           return_KLRec.fLocation = MakeNewDataRecord(newData);
           return_diskpos = NO_DADDR;
           return_flag = 1;
           fHouseKeeping.fNumItems +=1;
           return;
           }
```

Create new data record

Loading node and checking for key

```
    GetBTreeNode(current, filepos);

    int found = current.SearchInNode(key, index);

    if(found) {
```

Replace old data with new data

```
        daddr_t location = current.data[index].fLocation;
        SaveDataRecord(newData, location);
        return;
        }
```

Recursive call

```
    KLRec         rec_coming_up;
    daddr_t       diskpos_coming_up;
    int           need_insert_or_split;

    DoAdd(newData, current.links[index],
                  need_insert_or_split,
                  rec_coming_up, diskpos_coming_up);

    if (need_insert_or_split == 0)
        return;
```

Insert into current node

```
    if (current.NotFull())
        current.InsertInNode(rec_coming_up,
                             diskpos_coming_up, index);
```

Or split current node

```
    else {
        Split(current,
              rec_coming_up,  diskpos_coming_up,
              index,
              return_KLRec, return_diskpos);
        return_flag = 1;
        }
    SaveBTreeNode(current, filepos);
}
```

Node splitting

The algorithm for `Split()` was given earlier. Its implementation is simple as it merely needs to sort out whether the extra data are to be inserted belong in the existing (left) node, a new right node, or should be returned to the parent node. The different SplitInsert functions get called as required. The algorithm for `SplitInsertRight()` was given earlier. The example implementation code shown here is for the other two cases.

```
void BTree::SplitInsertMiddle(BTreeNode& nodetosplit,
    KLRec& extradata, daddr_t extralink, int index,
    KLRec& return_KLRec, daddr_t& return_diskpos)
{
    BTreeNode newNode;
    int           i,j;
```

Move contents of top half of node into new node

```
    for (i = MAX-1, j = MIN-1; i >= MIN; i--, j--) {
        newNode.links[j+1] = nodetosplit.links[i+1];
        newNode.data[j] = nodetosplit.data[i];
```

```
            }

    newNode.links[0] = extralink;
    newNode.n_data = nodetosplit.n_data = MIN;
    return_KLRec = extradata;
    return_diskpos = MakeNewDiskBNode(newNode);                    Save new node
}


void BTree::SplitInsertLeft(BTreeNode& nodetosplit,
    KLRec& extradata, daddr_t extralink, int index,
    KLRec& return_KLRec, daddr_t& return_diskpos)
{
    BTreeNode newNode;
    int           i,j;
    for (i = MAX-1, j = MIN-1; i >= MIN; i--, j--) {               Copy data across to
                  newNode.links[j+1] = nodetosplit.links[i+1];     new node
                  newNode.data[j] = nodetosplit.data[i];
                  }
    newNode.links [0] = nodetosplit.links [MIN];
    return_KLRec = nodetosplit.data[MIN-1];                        Pick value to be
                                                                   passed back
    for (i--; i >= index; i--) {                                   Shift values across to
           nodetosplit.links[i+2] = nodetosplit.links[i+1];        make room
           nodetosplit.data[i+1] = nodetosplit.data[i];
           }

    Slot in the new information.
    nodetosplit.links[index+1] = extralink;
    nodetosplit.data[index] = extradata;
    newNode.n_data = nodetosplit.n_data = MIN;
    return_diskpos = MakeNewDiskBNode(newNode);                    Save new node
}
```

Implementation of class BTree::Remove() and related functions

Function `Remove()` itself is simple. As explained earlier, it merely needs to set up the initial recursive call and check for the (uncommon!) case of a need to change the root when the existing root becomes empty:

```
void BTree::Remove(long key)
{
    if(fHouseKeeping.fRoot == NO_DADDR)
           return;

    BTreeNode     root_node;
    GetBTreeNode(root_node, fHouseKeeping.fRoot);

    (void) DoRemove(key, root_node);
```

```
    if (root_node.n_data == 0)
            fHouseKeeping.fRoot = root_node.links [0];
    else
            SaveBTreeNode(root_node, fHouseKeeping.fRoot);
}
```

(The housekeeping data don't have to be saved immediately. They are saved by the destructor that closes the BTree files.)

DoRemove()

The `DoRemove()` function implements the algorithm given earlier:

```
int BTree::DoRemove(long bad_key, BTreeNode& cNode)
{
    int           index;
    int           found = cNode.SearchInNode(bad_key, index);
    if(found) {
            DeleteKeyInNode(cNode,index);
            fHouseKeeping.fNumItems--;
            return cNode.Deficient();
            }

    daddr_t subtree = cNode.links[index];
    if(subtree == NO_DADDR)
            return 0;

    BTreeNode     nextNode;
    int           repairsneeded;
    GetBTreeNode(nextNode, subtree);
    repairsneeded = DoRemove(bad_key, nextNode);
    if (repairsneeded)
            Restore(cNode, nextNode, index);
    else
            SaveBTreeNode(nextNode, subtree);
    return cNode.Deficient();
}
```

Recursive call

The result returned by the function indicates whether the given node has become deficient. If it is deficient, then the caller will discover that "repairs (are) needed".

DeleteKeyInNode()

The `DeleteKeyInNode()` function shows the details of setting up the mechanism to find a key to promote followed by the call back to `DoRemove()` to get rid of the original copy of this key.

```
void BTree::DeleteKeyInNode(BTreeNode& aNode, int index)
{
    if (aNode.links [index+1] == NO_DADDR) {
                    DeleteInLeaf(aNode, index);
                    return;
                    }
```

Check for simple case, deletion in leaf

```
    daddr_t subtree = aNode.links[index + 1];
    aNode.data[index] = Successor(subtree);

    long promotedskey = aNode.data[index].fKey;
    BTreeNode     nxtNode;
    GetBTreeNode(nxtNode, subtree);

    int repairsneeded = DoRemove(promotedskey, nxtNode);
    if (repairsneeded)
           Restore (aNode, nxtNode, index+1);
    else
           SaveBTreeNode(nxtNode, subtree);
}
```

Promote data from right subtree

Removal of original of promoted data

The `DeleteKeyInLeaf()` function is trivial (shift higher keys left inside node, decrement count) and so is not shown.

Successor

The `Successor()` function involves an iterative search that runs down the left links as far as possible. The function returns the `KLRec` (key/data location pair) for the next key larger than that in the call to `Remove()`.

```
KLRec BTree::Successor(daddr_t subtree)
{
    BTreeNode     aNode;
    while(subtree != NO_DADDR) {
           GetBTreeNode(aNode, subtree);
           subtree = aNode.links[0];
           }
    return aNode.data[0];
}
```

Restore()

The `Restore()` function (which chooses which sibling gets used to move data or combine with the deficient node) is simple to implement from the algorithm given earlier.

MergeOrCombine Left()

The function `MergeOrCombineLeft()` illustrates the implementation for one of the two MergeOrCombine functions. The "right" function is similar.

```
void BTree::MergeOrCombineLeft(BTreeNode& parent,
                  BTreeNode& deficient, int index)
{
    BTreeNode     left_nbr;
    daddr_t       left_daddr = parent.links[index-1];

    GetBTreeNode(left_nbr, left_daddr);
    if(left_nbr.MoreThanMinFilled()) {
           MoveRight (parent, left_nbr, deficient, index-1);
           SaveBTreeNode(left_nbr, left_daddr);
           SaveBTreeNode(deficient, parent.links[index]);
           }
    else {
```

```
            Combine(parent, left_nbr, deficient, index-1);
            SaveBTreeNode(left_nbr, left_daddr);
            }
    }
```

MoveLeft() The explanation given in the previous section included an algorithm for `MoveRight()`; this is the implementation for `MoveLeft()`:

```
void BTree::MoveLeft(BTreeNode& parent, BTreeNode& left,
            BTreeNode& right, int index)
{
    KLRec rec_from_parent = parent.data[index];
    daddr_t downlink = right.links[0];
    left.InsertAtRight(rec_from_parent, downlink);
    parent.data[index] = right.data[0];
    right.ShiftLeft();
}
```

Combine() Function `Combine()` removes all data from the node given as argument `right`, shifting these values along with information from the parent down into the `left` node:

```
void BTree::Combine(BTreeNode& parent, BTreeNode& left,
    BTreeNode& right, int index)
{
    KLRec rec_from_parent = parent.data[index];
    daddr_t downlink = right.links[0];
    left.InsertAtRight(rec_from_parent, downlink);

    for(int j = 0; j < right.n_data; j++)
            left.InsertAtRight(right.data[j], right.links[j+1]);

    parent.Compress(index);
}
```

24.2.5 Testing

The problems involved in testing the BTree code, and their solution, are exactly the same as for the AVL tree. The BTree algorithms are complex. There are many special cases. Things like promoting a key from a leaf several levels down in the tree are only going to occur once the tree has grown quite large. Operations like deleting the current root node are going to be exceedingly rare. You can't rely on simple interactive testing.

Instead, you use the technique of a driver program that invokes all the basic operations tens of thousands of times. The driver program has to be able to test the success of each operation, and terminate the program if it detects something like a supposedly deleted item being "successfully" found by a later search. A code coverage

tool has to be used in conjunction with the driver to make certain that every function has been executed.

The driver needed to test the BTree can be adapted from that used for the AVL tree. There are a few changes. For example, insertion of a "duplicate" key is not an error, instead the old data are overwritten. Data records are not dynamically created in main memory. Instead, the program can use a single data record in memory filling it in with data read from the tree during a search operation, or setting its data before an insert.

EXERCISES

1 Complete the implementation and testing of the AVL class.

2 Complete the implementation and testing of the BTree class.

3 Add a mechanism for "recycling" the space occupied by "dead" BTreeNodes.