8.3 Application: 2-3 Trees

We have seen in Section 7.3 that an ordered binary search tree can be used to implement an ordered list. The primary drawback of such an implementation is that, though the search efficiency may be as good as \(O(\log n)\), this fast search efficiency cannot be guaranteed. It depends on the order in which data arrive for insertion into the tree. In Section 7.6, we examined height-balancing as an implementation strategy for guaranteeing an \(O(\log n)\) search efficiency. We also noted that the constant of proportionality involved in this big-\('O\) efficiency figure is approximately 1.5.

In this section we examine another tree-based technique for implementing an ordered list. Called a 2-3 tree, this technique can improve on the efficiency of a height-balanced tree by guaranteeing a search path that never exceeds \(\log n + 1\). That is, it matches the efficiency of a full binary tree. The price paid for this efficiency is space - a price we will analyze more fully after examining the technique. Formally, we can define a 2-3 tree as follows.

Definition: A 2-3 tree consists of a general tree and a precedence relationship precedes with the following properties.

1. Every node in the 2-3 tree has room to store two informational fields. Call these informational fields firstinfo and secondinfo. Typically, these informational fields can represent two full-fledged data records.
2. Every node in the 2-3 tree has room for three pointers to other nodes. Call these pointers firstChild, secondChild, and thirdChild.
3. Every node in the 2-3 tree has either
   a. firstinfo with active data and secondinfo with an empty flag, or
   b. the data in firstinfo preceding that in secondinfo according to the precedes relationship for the tree.
4. In any given nonleaf node:
   a. All data in the subtree referenced by firstChild must precede firstinfo, and all data in the subtree referenced by secondChild must follow firstinfo in the precedes relationship for the tree.
   b. If secondinfo has active data, then all data in the subtree referenced by secondChild must precede secondinfo, and all data in the subtree referenced by thirdChild must follow secondinfo.
5. All leaf nodes are on the same level.

Figure 8.5
2-3 tree node.

Table 8.3

<table>
<thead>
<tr>
<th>firstChild</th>
<th>firstinfo</th>
<th>secondChild</th>
<th>secondinfo</th>
<th>thirdChild</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>firstinfo</td>
<td>NULL</td>
<td>secondinfo</td>
<td>thirdChild</td>
</tr>
</tbody>
</table>

According to this definition, we can think of a 2-3 tree node that is not at leaf level as the structure in Figure 8.5. An example of a three-level 2-3 tree with integer informational keys appears in Figure 8.6. Those nodes in which the secondinfo field is not active appear simply as data nodes with only one integer in them.

A minimal public interface for the 2-3 tree as an ADT is provided in the following class definition. In addition to a constructor that specifies the precedes relation for the tree, the class provides operations to add and search for nodes in the tree.

Figure 8.6
A 2-3 tree with integer data. Nodes with only one integer have a secondinfo field that is not active.

```cpp
// BaseData is either a C++ built-in type, or a C++ class that has
// an assignment operator that overloaded the "=" operator and an equality
// test that overloads the "==" operator.
template <class BaseData>
class TwoThreeTree {
  public:
    // Interface for TwoThreeTree constructor
    // GIVEN: An uninitialised TwoThreeTree object;
    // precedes -- a function to compare BaseData values;
    // GIVEN: x and y -- values to compare
    // RETURN as value of function;
    // TRUE if x precedes y;
    // FALSE if x and y are equal, or
    // if y precedes x
    // RETURN: The TwoThreeTree object is initialised to the empty tree
    // with precedes establishing the hierarchical ordering of the tree.
    TwoThreeTree(BOOL precedes); (const BaseData &x, const BaseData &y));
```

From Patterning and Naps Data Structures book.
8.3 Application: 2-3 Trees

The search algorithm for 2-3 trees is similar to that for a binary search tree. That is, we start at the root of the tree. A comparison of the target item to the informational fields indicates whether the target is in the current node or, based on the relationship of the target to firstInfo and secondInfo, which child pointer to follow. For instance, to find 650 in the tree of Figure 8.6, you should verify that

1. We follow the thirdChild pointer from the root, because 650 follows 500.
2. From the level 1 node containing 600 and 700 we follow the secondChild pointer, because 650 is between 600 and 700.
3. At level 2 we find the target.

This algorithm is formalized in the following implementation of the search function from the TwoThreeTree class definition. Following what has become a common trick in implementing recursive tree-based algorithms, the search function itself merely serves as a front-end function that passes the root pointer for the tree to the real recursive function. Here that recursive workhorse is called privSearch and is declared as a private function member of the TwoThreeTree class. As the following documentation indicates, the privSearch function does a bit more than needed in merely searching the tree. In particular, when the target is not found, it will return a pointer to the leaf node where it expected the target to be. This additional information will later allow us to use privSearch in the implementation of the add operation.

In addition to the privSearch function, we need to add another private subordinate function called leafNode to our class definition. This function simply tests whether a given node is a leaf node. Finally, we must establish a convention for determining if the informational members of a TwoThreeNode contain active data or as empty flag. The convention we adopt is to store NULL as the value for an empty flag. By type casting, we can then compare BaseData variables to NULL in checking for the empty flag.

```cpp
template <class BaseData>
class TwoThreeNode
{
  public:
    BaseData firstInfo, secondInfo; // The data in the nodes
    TwoThreeNode *firstChild, *secondChild, *thirdChild;
    TwoThreeNode *parent; // Parent pointer facilitates moving up tree
};
```

We then include two protected members in the TwoThreeTree class definition—a root pointer for the tree and a precedes function. The constructor merely sets the root pointer to NULL and its precedes function to the precedes parameter that it receives.

```cpp
template <class BaseData>
class TwoThreeTree
{
  protected:
    TwoThreeNode<BaseData> *root;
    BOOLEAN (*precedes)(const BaseData &x, const BaseData &y);
  // Other members specified as before
};
```

// Augmenting the TwoThreeTree class definition with appropriate private // functions for implementing the add operation

```cpp
template <class BaseData>
class TwoThreeTree
{
  private:
    // Internal auxiliary functions

    // Interface for privSearch function
    // GIVEN: target -- a pointer to the root of a TwoThreeTree;
    //        item -- a value of type BaseData that contains, perhaps in a special key field, a value to be searched for using the equality test for BaseData.
    // RETURN: item -- a value of type BaseData. If target can be found in the tree, item contains the entire contents of the tree node (key value and all associated data) that matches target,
```

Search Algorithm

The search algorithm for 2-3 trees is similar to that for a binary search tree. That is, we start at the root of the tree. A comparison of the target item to the informational fields indicates whether the target is in the current node or, based on the relationship of the target to firstInfo and secondInfo, which child pointer to follow. For instance, to find 650 in the tree of Figure 8.6, you should verify that

1. We follow the thirdChild pointer from the root, because 650 follows 500.
2. From the level 1 node containing 600 and 700 we follow the secondChild pointer, because 650 is between 600 and 700.
3. At level 2 we find the target.

This algorithm is formalized in the following implementation of the search function from the TwoThreeTree class definition. Following what has become a common trick in implementing recursive tree-based algorithms, the search function itself merely serves as a front-end function that passes the root pointer for the tree to the real recursive function. Here that recursive workhorse is called privSearch and is declared as a private function member of the TwoThreeTree class. As the following documentation indicates, the privSearch function does a bit more than needed in merely searching the tree. In particular, when the target is not found, it will return a pointer to the leaf node where it expected the target to be. This additional information will later allow us to use privSearch in the implementation of the add operation.

In addition to the privSearch function, we need to add another private subordinate function called leafNode to our class definition. This function simply tests whether a given node is a leaf node. Finally, we must establish a convention for determining if the informational members of a TwoThreeNode contain active data or as empty flag. The convention we adopt is to store NULL as the value for an empty flag. By type casting, we can then compare BaseData variables to NULL in checking for the empty flag.

```cpp
// Augmenting the TwoThreeTree class definition with appropriate private // functions for implementing the add operation
```
and the pointer where references the node where target is located. If target is not found, item is unreliable, and the where pointer references the last node where target would have been located if it were in the tree.

RETURN as value of function:
TRUE if target could be found in the TwoThreeTree; FALSE if target is not in the tree.

BOOLEAN privSearch(TwoThreeNode<baseData> *rt, const baseData &target, baseData &item, TwoThreeNode<baseData> *where) {

// Interface for leafNode function
// GIVEN: t -- a pointer to a TwoThreeNode within a TwoThreeTree.
// RETURN as value of function:
// TRUE if t references a leaf node within the TwoThreeTree and FALSE otherwise.

BOOLEAN leafNode(TwoThreeNode<baseData> *t);

// Other class members declared as before

// Implementation of search, privSearch, and leafNode operations
template <class baseData>
BOOLEAN TwoThreeTree<baseData>::search(const baseData &target, baseData &item) {
TwoThreeNode<baseData> *where;

// Just pass the root pointer to the recursive workhorse
return privSearch(rt, target, item, where);
}

template <class baseData>
BOOLEAN TwoThreeTree<baseData>::privSearch(TwoThreeNode<baseData> *rt, const baseData &target, baseData &item, TwoThreeNode<baseData> *where) {

BOOLEAN found, atLeastOneItem, twoItems;

if (rt == NULL) // target will not be found in the empty tree
{
where = NULL;
return (FALSE);
}

found = FALSE;
// NULL is used as an empty flag in informational field.
// Type casting allows comparison of baseData variable to NULL.
if (((int)(rt->firstInfo)) != NULL)
atLeastOneItem = TRUE;
else
atLeastOneItem = FALSE;
if (((int)(rt->secondInfo)) != NULL)
twoItems = TRUE;
else
twoItems = FALSE;
if ((rt->firstInfo == target)) // Check first key
{
found = TRUE;
item = rt->firstInfo;
else
if (twoItems)
{
if (rt->secondInfo == target) // Check second key
{
found = TRUE;
item = rt->secondInfo;
}

if (found || leafNode(rt))
{
where = rt;
return (found);
}

if (target was found at rt or rt references a leafnode, return where as rt
if (found || leafNode(rt))
{

found = FALSE;
}

// If reach this point, there must be at least one item in the node, so
// make recursive call(s) to search appropriate subtrees:
if (precedes(target, rt->firstChild, target, item, where))
return (privSearch(rt->firstChild, target, item, where));
if (twoItems)
return (privSearch(rt->secondChild, target, item, where));
if (precedes(target, rt->secondChild, target, item, where))
return (privSearch(rt->secondChild, target, item, where));
else
return (privSearch(rt->thirdChild, target, item, where));
}

template <class baseData>
BOOLEAN TwoThreeTree<baseData>::leafNode(TwoThreeNode<baseData> *t) {
if (rt->firstChild == NULL) && (rt->secondChild == NULL) &&
(t->thirdChild == NULL)
return (TRUE);
else
return (FALSE);
}

8.3 Application: 2-3 Trees

else
if (twoItems)
{
if (rt->secondInfo == target) // Check second key
{
found = TRUE;
item = rt->secondInfo;
}

Search Efficiency

Stipulation 5 in our definition of a 2-3 tree is critical in an analysis of search efficiency for this data structure. By its guarantee that all leaf nodes are at the same level, it assures us that a 2-3 tree with N information items will have a maximal search path no longer than that in a full binary search tree with the same N items. From our earlier analysis of binary search trees, we know that this maximal path length is \log_2 N + 1.

Call recursively
with this root Call recursively
with this node with this tree if Call recursively
precedes target secondInfo is
empty or target
precedes secondInfo
Hence, provided we can develop add and remove operations that maintain a tree in a fashion dictated by the five stipulations in our definition of a 2-3 tree, we have a scheme for implementing ordered lists that is evidently on a par with full binary search trees and slightly better than height-balanced trees. What price have we paid? As usual, the trade-off is space. Here, we run the risk of having numerous secondInfo fields filled with the empty flag. (You will explore how many such fields can be wasted in the exercises and problems for this chapter.) If these fields are actually large data records, you may decide the wasted space is not worth the relatively minor gain in speed over height-balanced trees.

Add Algorithm

To ensure that stipulation 5 of the definition is met, 2-3 trees exhibit the rather curious behavior of adding a new level to the tree by sprouting a new root instead of a leaf at a deeper level. We can illustrate this phenomenon and the algorithm that controls it by making a few insertions on the tree of Figure 8.6.

Example 8.3

Insert 250 as an informational item in the tree of Figure 8.6 on page 357. Here a search algorithm would dictate that, if 250 were in the tree, it should have been found in the same node as 200. Since there is room for another informational field in the node containing 200, the place to insert 250 is obvious—as the secondInfo field in that same node. The resulting tree is shown in Figure 8.7.

Example 8.4

Add 850 to the 2-3 tree of Figure 8.7. The situation in this example is a bit more complex. As in Example 8.3, the search algorithm tells us that 850 should have been in the level-2 node containing 750 and 800 if it were in the tree. But it cannot fit there since we already have two data items. So, we will look at the tree items 750, 800, and 850; choose the middle one (800) to pass back up to the parent node; and then create two nodes containing one item each (for 750 and 850). Item 800 and the one-node tree containing 850 will then be passed back to the node containing 600 and 700. This is illustrated in Figure 8.8. If there were only one informational item here, 800 could be added as the secondInfo field and the node containing 850 added as the thirdChild. Unfortunately, there is not enough room, so the splitting process must be repeated with 600, 700, and 800. This time 700 is chosen as the middle value with two subtrees rooted at 600 and 800 also created. This is highlighted in Figure 8.9.

Figure 8.7

2-3 tree of Figure 8.6 with 250 inserted.

Figure 8.8

As leaf-level nodes become crowded, they are split with data being passed back up the tree to parent node.

Figure 8.9

Splitting with transfer back up the tree must occur until we find a node with room to expand.

Figure 8.10

Final configuration of tree from Figure 8.7 after adding 850.
Item 700 is then passed back up the tree along with the subtree rooted at 800 for an encore of the splitting phenomenon. At the root, 400, 500, and 700 are compared. We split at 500, forming subtrees rooted at 400 and 700 and creating a new root with 500 as its first INFO data item. The resulting tree, now with a level added at the top, is shown in Figure 8.10.

The following implementation of the add operation for the TwoThreeTree class considers the two possibilities illustrated in Examples 8.3 and 8.4, that is, adding an item in a leaf node with room to expand or forcing a split with data being passed recursively up the tree until we find a node with room to store additional data. The details of splitting a node (perhaps recursively) are deferred to a subordinate function called split. Finally, as a third possibility, the add function must consider the insertion of the first item in an empty tree as a special case.

```cpp
// Augmenting the TwoThreeTree class definition with subordinate functions,
// in addition to private, that are needed in implementing the add operation

// Interface for split function
// GIVEN: t -- a pointer to the root of an entire TwoThreeTree containing
// a node to be split;
// tree -- a pointer to the root of current subtree that must be
// split to add the data parameter;
// // branch -- a pointer to a subtree to be added to the node
// referenced by tree if the addbranch parameter is TRUE.
// // if addBranch is FALSE, branch has no well-defined value.
// // data -- the data item forcing the split of the current root
// referenced by tree.
// // addBranch -- a BOOLEAN value that is TRUE if a branch must be added
to the splitting node in addition to the data item.
// // if addBranch is FALSE when a leaf node is being split and
// TRUE when an interior node is being split.
// // RETURN: t will become a pointer to a new root for the entire TwoThreeTree
// if the tree parameter is without a parent node. Otherwise t is
// unchanged. The node referenced by tree will be split based on the
// value in the data parameter. The node will contain just one
// informational item and have a new sibling to its right.

void split (TwoThreeNode<BaseData> *at, TwoThreeNode<BaseData> *tree,
TwoThreeNode<BaseData> *branch, const BaseData &data,
BOOLEAN addBranch);

// Interface for makeTreeNode function
// GIVEN: No preconditions.
// RETURN as value of function: A pointer to a TwoThreeNode initialized with all
// informational fields set to the empty flag and all parent and child
// pointers set to NULL.

TwoThreeNode<BaseData> *makeTreeNode();

// Other class members declared as before
```

8.3 Application: 2-3 Trees

```cpp
// Implementation of the add, split, and makeTreeNode functions
template <class BaseData>
BOOLEAN TwoThreeTree<BaseData>::add(const BaseData &item)
{
    BaseData info;
    BOOLEAN found;
    TwoThreeNode<BaseData> *leaf;
    found = privSearch(root, item, info, leaf); // Find leaf node for key
    if (leaf == NULL) // Make initial root
        root = makeTreeNode();
    root->firstINFO = item;
    return(TRUE);
}
if (found)
    return(FALSE); // Item already in tree
    // Add item if it will fit in leaf, otherwise call split to split leaf
    if (((int)(leaf->secondINFO) != NULL))
        // Then there are two items
        split(root, leaf, leaf, item, FALSE);
    else if (precedes(leaf->firstINFO, item))
        leaf->secondINFO = item;
    else
        leaf->secondINFO = leaf->firstINFO;
        leaf->firstINFO = item;
    return(TRUE);
```
```c
parent = makeTreeNode();
tree->parent = parent;
parent->firstChild = tree;
t = parent;
else
    parent = tree->parent;

// Make sibling node with empty flags and NULL children
sibling = makeTreeNode();
sibling->parent = parent;

// Determine which of the three key values is the middle one in terms
// of value and assign it to middle. Put/leave the smallest one in the
// current node (tree), put the largest one into the new node (sibling).
if (precedes(data, tree->firstInfo))
    middle = tree->firstInfo;
tree->firstInfo = data;
sibling->firstInfo = tree->secondInfo;
tree->secondInfo = (BaseData) NULL;
else
    if (precedes(data, tree->secondInfo))
    {
```
siblings->secondChild = tree->thirdChild;
siblings->firstChild = branch;
tree->thirdChild = NULL;
siblings->secondChild->parent = siblings;
siblings->firstChild->parent = siblings;
}
else

siblings->secondChild = branch;
siblings->firstChild = tree->thirdChild;
tree->thirdChild = NULL;
siblings->secondChild->parent = siblings;
siblings->firstChild->parent = siblings;
}
} // if (addBranch)

// Now "promote" middle up to the tree's parent node and determine whether
// that node must be split. If middle will fit, arrange the parent node
// accordingly and return. If not, then make a recursive call to split with
// the parent node as tree, the sibling node as branch, and middle as data.

if (parent->firstInfo == NULL) // Then firstInfo is empty flag
{
        parent->firstInfo = middle;
        parent->firstChild = tree;
        parent->secondChild = siblings;
        return;
}

if (parent->secondInfo != NULL)
{
        split(t, parent, siblings, middle, TRUE);
        return;
}

if (precedes(parent->firstInfo, middle))
{
        parent->secondInfo = middle;
        parent->thirdChild = siblings;
        return;
}

parent->secondInfo = parent->firstInfo;
parent->firstInfo = middle;
parent->thirdChild = parent->secondChild;
parent->secondChild = siblings;
// end of split function

template <class BaseData>
TwoThreeNode<BaseData> *TwoThreeTree<BaseData>::makeTreeNode()
{
        TwoThreeNode<BaseData> *p;

        p = new TwoThreeNode<BaseData>;
        p->parent = NULL;
        p->firstChild = NULL;
        p->secondChild = NULL;
        p->thirdChild = NULL;
        p->firstInfo = (BaseData) NULL;
        p->secondInfo = (BaseData) NULL;
        return(p);
    }