## 9.3 Insertion and deletion in binary dictionary

When maintaining a dynamic set of data we may want to insert new items into the set and also delete some old items from the set. So one common repertoire of operations on a set of data, S, is:

| | |
|---|---|
| **in( X, S)** | X is a member of S |
| **add( S, X, S1)** | Add X to S giving S1 |
| **del( S, X, S1)** | Delete X from S giving S1 |

Let us now define the *add* relation. It is easiest to insert new data at the bottom level of the tree, so that a new item becomes a leaf of the tree at such a position that the ordering of the tree is preserved. Figure 9.9 shows changes in a tree during a sequence of insertions. Let us call this kind of insertion

**addleaf( D, X, D1)**

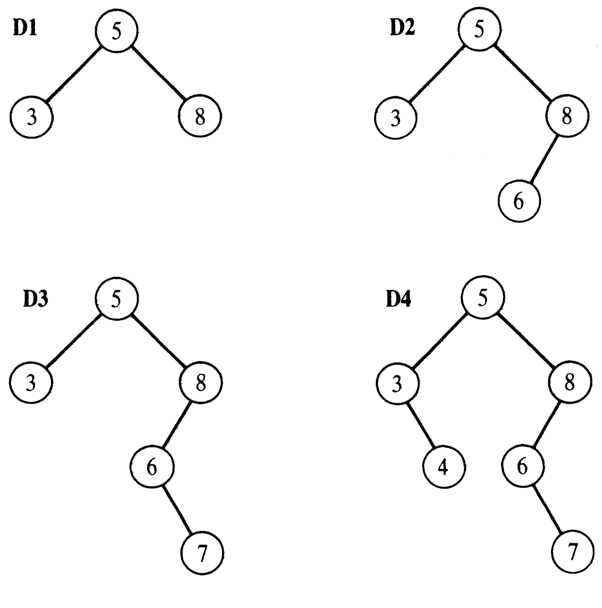

**Figure 9.9** Insertion into a binary dictionary at the leaf level. The trees correspond to the following sequence of insertions: **add( D1, 6, D2)**, **add( D2, 7, D3)**, **add( D3, 4, D4)**.

```
addleaf( nil, X, t( nil, X, nil) ).

addleaf( t( Left, X, Right), X, t( Left, X, Right) ).

addleaf( t( Left, Root, Right), X, t( Left1, Root, Right) ) :-
  gt( Root, X),
  addleaf( Left, X, Left1).

addleaf( t( Left, Root, Right), X, t( Left, Root, Right1) ) :-
  gt( X, Root),
  addleaf( Right, X, Right1).
```

**Figure 9.10** Inserting an item as a leaf into the binary dictionary.

Rules for adding at the leaf level are:

- The result of adding X to the empty tree is the tree **t( nil, X, nil)**.
- If X is the root of D then D1 = D (no duplicate item gets inserted).
- If the root of D is greater than X then insert X into the left subtree of D; if the root of D is less than X then insert X into the right subtree.

Figure 9.10 shows a corresponding program.

Let us now consider the *delete* operation. It is easy to delete a leaf, but deleting an internal node is more complicated. The deletion of a leaf can be in fact defined as the inverse operation of inserting at the leaf level:

```
delleaf( D1, X, D2) :-
  addleaf( D2, X, D1).
```

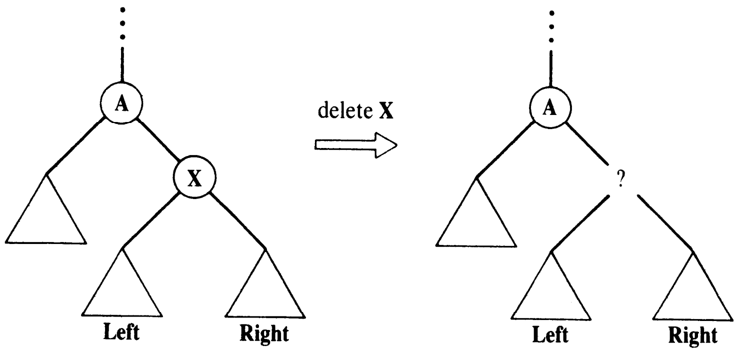

**Figure 9.11** Deleting X from a binary dictionary. The problem is how to patch up the tree after X is removed.

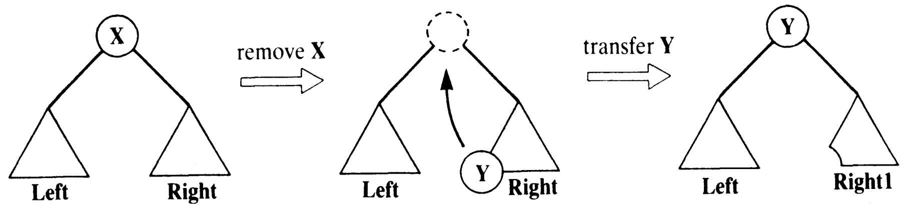

**Figure 9.12** Filling the gap after removal of X.

Unfortunately, if X is an internal node then this does not work because of the problem illustrated in Figure 9.11. X has two subtrees, **Left** and **Right**. After X is removed, we have a hole in the tree and **Left** and **Right** are no longer connected to the rest of the tree. They cannot both be directly connected to the father of X, A, because A can accommodate only one of them.

If one of the subtrees **Left** and **Right** is empty then the solution is simple: the non-empty subtree is connected to A. If they are both non-empty then one idea is as shown in Figure 9.12. The left-most node of **Right**, Y, is transferred from its current position upwards to fill the gap after X. After this transfer, the tree remains ordered. Of course, the same idea works symmetrically, with the transfer of the right-most node of **Left**.

According to these considerations, the operation to delete an item from the binary dictionary is programmed in Figure 9.13. The transfer of the left-most node of the right subtree is accomplished by the relation

**delmin( Tree, Y, Tree1)**

```
del( t( nil, X, Right), X, Right).

del( t( Left, X, nil), X, Left).

del( t( Left, X, Right), X, t( Left, Y, Right1) ) :-
  delmin( Right, Y, Right1).

del( t( Left, Root, Right), X, t( Left1, Root, Right) ) :-
  gt( Root, X),
  del( Left, X, Left1).

del( t( Left, Root, Right), X, t( Left, Root, Right1) ) :-
  gt( X, Root),
  del( Right, X, Right1).

delmin( t( nil, Y, R), Y, R).

delmin( t( Left, Root, Right), Y, t( Left1, Root, Right) ) :-
  delmin( Left, Y, Left1).
```

**Figure 9.13** Deleting from the binary dictionary.

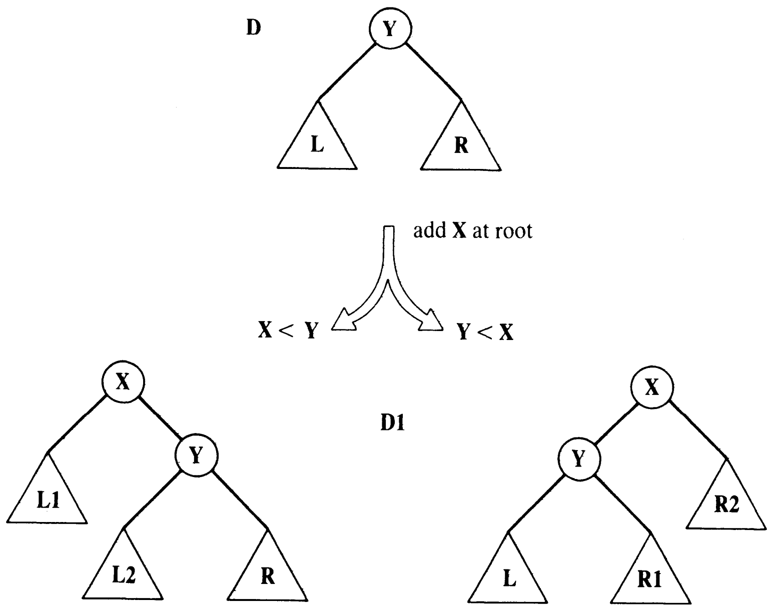

**Figure 9.14** Inserting **X** at the root of a binary dictionary.

where Y is the minimal (that is, the left-most) node of **Tree**, and **Tree1** is **Tree** with Y deleted.

There is another elegant solution to *add* and *delete*. The *add* relation can be defined non-deterministically so that a new item is inserted at any level of the tree, not just at the leaf level. The rules are:

To add X to a binary dictionary D either:

- add X at the root of D (so that X becomes the new root), or
- if the root of D is greater than X then insert X into the left subtree of D, otherwise insert X into the right subtree of D.

The difficult part of this is the insertion at the root of D. Let us formulate this operation as a relation

**addroot( D, X, D1)**

where X is the item to be inserted at the root of D and D1 is the resulting dictionary with X as its root. Figure 9.14 illustrates the relations between X, D and D1. The remaining question is now: What are the subtrees L1 and L2 in Figure 9.14 (or R1 and R2 alternatively)? The answer can be derived from the following constraints:

- L1 and L2 must be binary dictionaries;

- the set of nodes in L1 and L2 is equal to the set of nodes in L;
- all the nodes in L1 are less than X, and all the nodes in L2 are greater than X.

The relation that imposes all these constraints is just our **addroot** relation. Namely, if X were added as the root into L, then the subtrees of the resulting tree would be just L1 and L2. In Prolog: L1 and L2 must satisfy the goal

```
addroot( L, X, t( L1, X, L2) )
```

The same constraints apply to R1 and R2:

```
addroot( R, X, t( R1, X, R2) )
```

Figure 9.15 shows a complete program for the 'non-deterministic' insertion into the binary dictionary.

The nice thing about this insertion procedure is that there is no restriction on the level of insertion. Therefore *add* can be used in the inverse direction in order to delete an item from the dictionary. For example, the following goal list

```
add( D, X, D1) :-
  addroot( D, X, D1).                          % Add X as new root

add( t( L, Y, R), X, t( L1, Y, R) ) :-         % Insert X into left subtree
  gt( Y, X),
  add( L, X, L1).

add( t( L, Y, R), X, t( L, Y, R1) ) :-         % Insert X into right subtree
  gt( X, Y),
  add( R, X, R1).

addroot( nil, X, t( nil, X, nil) ).            % Insert into empty tree

addroot( t( L, X, R), X, t( L, X, R) ).        % X already in tree

addroot( t( L, Y, R), X, t( L1, X, t( L2, Y, R) ) ) :-
  gt( Y, X),
  addroot( L, X, t( L1, X, L2) ).

addroot( t( L, Y, R), X, t( t( L, Y, R1), X, R2) ) :-
  gt( X, Y),
  addroot( R, X, t( R1, X, R2) ).
```

**Figure 9.15** Insertion into the binary dictionary at any level of the tree.

constructs a dictionary D containing the items 3, 5, 1, 6, and then deletes 5 yielding a dictionary DD:

```
add( nil, 3, D1), add( D1, 5, D2), add( D2, 1, D3),
add( D3, 6, D), add( DD, 5, D)
```