Binary Trees, Binary Search Trees and AVL Trees

[N.B.: This presentation is based on parts of a chapter in [Wirth]. The Pascal programs have been redesigned and text fragments have been scanned from the book and edited to fit the new algorithms. Some introductory sections with basic notions and algorithms have been added. *Kees Hemerik*]

1Introduction

In this note we consider some variations on binary search trees, with the ultimate goal to design data structures that enable us to maintain large collections of data elements in such a way that for a collection of N elements the following operations can all be performed in O(log N) time:

- Find (find the location of a given data item in the collection)
- Insert (insert a new data item into the collection)
- Delete (delete a given data item from the collection)

The note begins with some general tree notions and algorithms, and subsequently introduces binary search trees, rotation operators, and AVL-trees [Adelson-Velskii & Landis].

2Basic Definitions

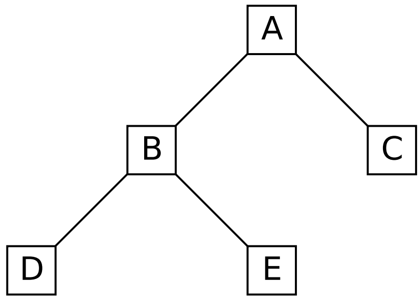
Assume some set S. The set BT(S) of *binary trees* over S can be defined inductively as the smallest set X such that:

- $\varepsilon \in X$
- for all $L \in X$, $a \in S$, $R \in X$: $<L, a, R> \in X$

Some examples:

- ε
- $< \varepsilon , D , \varepsilon >$
- $< < \varepsilon , D, \varepsilon > , B , < \varepsilon , E , \varepsilon > >$
- $< < < \varepsilon , D , \varepsilon > , B , < \varepsilon , E , \varepsilon > > , A , < \varepsilon , C , \varepsilon > >$

Trees are usually represented graphically in an obvious way. E.g. the last example above would be drawn as:



Functions on binary trees can be defined recursively. E.g. the function h giving the height of a binary tree is defined as follows:

- $h\ \varepsilon = 0$
- $h < L , a , R > = 1 + (h\ L\ \mathbf{max}\ h\ R)$

In Pascal, binary trees over some assumed data type TData can be defined as follows:

```
type
  TData = ...;

  PNode = ^TNode;
  TNode =
  record
    FData: TData;
    FLeft: PNode;
    FRight: PNode
  end;
```

The empty tree ε will be represented by the pointer value **nil**. For the construction of a tree node the following function will be convenient:

```
function MakeNode(AData: TData; ALeft: PNode; ARight: PNode): PNode;
var
  H: PNode;
begin
  New(H);
  with H^ do
  begin
    FData := AData;
    FLeft := ALeft;
    FRight := ARight;
  end;
  Result := H;
end;
```

Function definitions like the one for the height function h above can be mapped to Pascal function definitions in a straightforward way:

```
function H(P: PNode): Integer;
begin
```

```
  if P = nil
  then Result := 0
  else Result := 1 + Max( H(P^.FLeft), H(P^.FRight) )
end;
```

3 Traversals

3.1 Standard Tree Traversals

Many operations on trees consist of systematically traversing the tree in some pre-determined order and applying some operation to the data in each node visited. There exist some standard traversal strategies, generally known as *pre-order*, *in-order*, and *post-order* respectively. They are defined as follows:

Pre-order traversal of tree with root top:

If the tree is not empty :

1. Visit node top
2. Traverse, in pre-order, node top's left subtree
3. Traverse, in pre-order, node top's right subtree

In-order traversal of tree with root top:

If the tree is not empty :

1. Traverse, in in-order, node top's left subtree
2. Visit node top
3. Traverse, in in-order, node top's right subtree

Post-order traversal of tree with root top:

If the tree is not empty :

1. Traverse, in post-order, node top's left subtree
2. Traverse, in post-order, node top's right subtree
3. Visit node top

3.2 Recursive Tree Traversals

In Pascal, these traversals can be coded simply using recursive procedures that take as parameter, in addition to the pointer to the root of the tree, a procedure that has to be applied to the data in each node:

```
type
  TAction = procedure(AData:TData);
  // the type of the procedure that has to be applied to the node data

procedure PreOrder(ANode: PNode; AAction: TAction);
begin
  if ANode <> nil then
    with ANode^ do
    begin
      AAction(FData);
      PreOrder(FLeft, AAction);
```

```
      PreOrder(FRight, AAction)
    end;
end;

procedure InOrder(ANode: PNode; AAction: TAction);
begin
  if ANode <> nil then
    with ANode^ do
    begin
      InOrder(FLeft, AAction);
      AAction(FData);
      InOrder(FRight, AAction)
    end;
end;

procedure PostOrder(ANode: PNode; AAction: TAction);
begin
  if ANode <> nil then
    with ANode^ do
    begin
      PostOrder(FLeft, AAction);
      PostOrder(FRight, AAction)
      AAction(FData);
    end;
end;
```

3.3Tree Traversals by Means of a Stack

The standard tree traversals can also be coded by means of a stack. The stack holds two kinds of obligations or tasks:

- Visit a given node (perform action on its data);
- Traverse the tree having a given node as its root.

Initially, the stack contains a single task, viz. traverse the entire tree. While the stack is not empty, its top element can be popped. Depending on its form, one of the following tasks is performed:

- If the task is of the form "Visit node Anode", the action is performed on its data;
- If the task is of the form "Traverse tree with root Anode", three new tasks are generated and pushed on the stack, viz.:
    - "Visit node Anode";
    - "Traverse the left subtree of ANode" ;
    - "Traverse the right subtree of ANode" ;

The desired traversal order (i.e. *pre-order*, *in-order* or *post-order*) determines the order in which the newly generated tasks should be pushed onto the stack. Remember that the stack is a LIFO (last-in, first-out) device and that therefore the tasks should be pushed in an order opposite to the order in which they are to be executed. For instance, to obtain a pre-order traversal, the tasks should be pushed in the following order:

1. "Traverse the right subtree of ANode";
2. "Traverse the left subtree of ANode";
3. "Visit node ANode".

Just as in the recursive case, the handling of empty (**nil**) trees deserves some attention. There are two possibilities:

- Allow nil pointers in the tasks on the stack. In this case, after popping a task, its node reference should be checked to be non-**nil** before performing any action on it;
- Disallow nil pointers in the tasks on the stack. In this case, a task should only be pushed onto the stack if its node reference is non-**nil**.

In the code samples below, the first alternative has been chosen. Furthermore, the tasks have been coded by means of the types TVisitKind and TTask and the stack has been coded by means of a class TStack, of which only the public interface has been shown. As was to be expected, the code of the three traversal procedures is very similar. The only difference is in the order in which the newly generated tasks are pushed on the stack.

```
type
  TVisitKind = (vkNode, vkTree);

  TTask =
  record
    FKind: TVisitKind;
    FNode: PNode;
  end;

  TVisitStack =
  class(TObject)
  private
  ...
  public
    // construction/destruction
    constructor Create;
    destructor Destroy; override;

    // queries
    function Top: TTask;
    function Count: Integer;
    function IsEmpty: Boolean;

    // commands
    procedure Pop;
    procedure Push(ATask: TTask);
    procedure PushTask(AKind: TVisitKind; ANode: PNode);
  end;
```

```
procedure Preorder(ANode: PNode; AAction: TAction);
var
  VStack: TVisitStack;
  VTask: TTask;
```

```
begin
  VStack := TVisitStack.Create;
  VStack.PushTask(vkTree, ANode);

  while not VStack.IsEmpty do
  begin
    VTask := VStack.Top;
    VStack.Pop;
    with VTask do
    begin
      if FNode <> nil then
      begin
        case VTask.FKind of
          vkNode:
            begin
              AAction(FNode^.FData);
            end;
          vkTree:
            begin
              VStack.PushTask(vkTree, FNode^.FRight);
              VStack.PushTask(vkTree, FNode^.FLeft);
              VStack.PushTask(vkNode, FNode);
            end;
        end{case};
      end{if};
    end{with};
  end{while};

  VStack.Free;
end;

procedure Inorder(ANode: PNode; AAction: TAction);
var
  VStack: TVisitStack;
  VTask: TTask;
begin
  VStack := TVisitStack.Create;
  VStack.PushTask(vkTree, ANode);

  while not VStack.IsEmpty do
  begin
    VTask := VStack.Top;
    VStack.Pop;
    with VTask do
    begin
      if FNode <> nil then
      begin
        case VTask.FKind of
          vkNode:
            begin
              AAction(FNode^.FData);
            end;
          vkTree:
```

```
          begin
            VStack.PushTask(vkTree, FNode^.FRight);
            VStack.PushTask(vkNode, FNode);
            VStack.PushTask(vkTree, FNode^.FLeft);
          end;
      end{case};
    end{if};
  end{with};
 end{while};

 VStack.Free;
end;

procedure Postorder(ANode: PNode; AAction: TAction);
var
  VStack: TVisitStack;
  VTask: TTask;
begin
  VStack := TVisitStack.Create;
  VStack.PushTask(vkTree, ANode);

  while not VStack.IsEmpty do
  begin
    VTask := VStack.Top;
    VStack.Pop;
    with VTask do
    begin
      if FNode <> nil then
      begin
        case VTask.FKind of
          vkNode:
            begin
              AAction(FNode^.FData);
            end;
          vkTree:
            begin
              VStack.PushTask(vkNode, FNode);
              VStack.PushTask(vkTree, FNode^.FRight);
              VStack.PushTask(vkTree, FNode^.FLeft);
            end;
        end{case};
      end{if};
    end{with};
  end{while};

  VStack.Free;
end;
```

4Binary Search Trees

4.1Order

If (S,<) is a totally ordered set, the set BT(S) of binary search trees over S can be given additional structure, thus enabling more efficient searching and modification. The idea is to organize the tree in such a way that for each subtree < L , A, R > the data elements in the left subtree L are all smaller than A, whereas the data elements in the right subtree R are all greater than A. When trying to locate a particular data element X in such a tree it suffices to repeatedly compare X to the data element A, say, in a node:

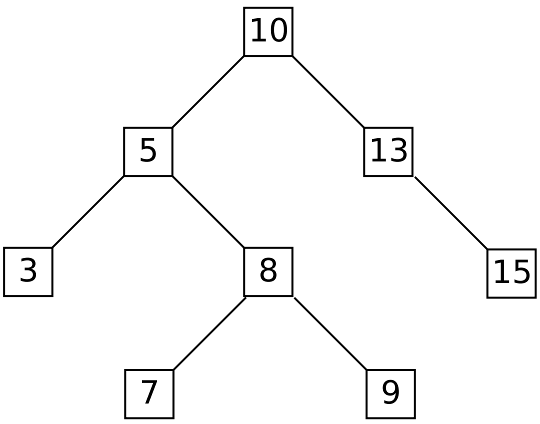
- if X < A , the search continues in the left subtree
- if X = A , the element has been found
- if X > A , the search continues in the right subtree

If the tree is well-balanced (this will be the subject of section 5), this search process can locate any element in a tree with N nodes in O(log N) steps. If the tree is not balanced, the search described here may still require O(N) steps, however. In this section we consider algorithms for the operations Find, Insert, and Delete for arbitrary binary search trees. In section 5, these will be refined to operations on balanced binary search trees.

Formally, the set BST(S,<) of binary search trees over the totally ordered set (S,<) is the set { t $\in$ BT(S) | bst(t) }, where the predicate bst is defined as follows:

- bst ε = true
- bst < L , a , R > =
  (bst L) **and** (bst R) **and** (for all x in L. x < a) **and** (for all x in R. a < x)

The following is an example of a binary search tree over the natural numbers:



From now on we assume that the type TData is equipped with a total ordering < . Since many of the algorithms to follow will require some 3-way comparison (i.e. less, equal, greater), we introduce a new type and comparison function:

```
type
  TCompare = (ls, eq, gt);

function Compare(X,Y: TData): TCompare;
begin
  if X < Y then Result := ls
  else if X = Y then Result := eq
  else Result := gt;
end;
```

4.2Find

The Find operation has to locate the node containing a given data element X, if any. The Pascal codings, both recursive and iterative, are straightforward:

Find using recursion:

```
function Find(X: TData; ANode: PNode): PNode;
begin
  if ANode = nil
  then Result := nil
  else
    case Compare(X, ANode^.FData) of
      ls: Result := Find(X, ANode^.FLeft);
      eq: Result := ANode;
      gt: Result := Find(X, ANode^.FRight)
    end{case}
end;
```

Find using repetition:

```
function Find(X: TData; ANode: PNode): PNode;
var
  H: PNode;
begin
  H := ANode;
   while (H <> nil) and (H^.FData <> X) do // use conditional
and
    if X < H^.FData
    then H := H^.FLeft
    else H := H^.Fright;
  Result := H;
end;
```

4.3 Insert

The Insert procedure takes two parameters: a value parameter X (the value to be inserted) and a **var** parameter P (the pointer to the root of the tree in which X has to be inserted). Upon return from the procedure P is the pointer to the modified tree.

N.B.: In this procedure, and in many others to follow, the use of **var** parameters is essential. Without var parameters the code of many algorithms may become significantly more complicated.

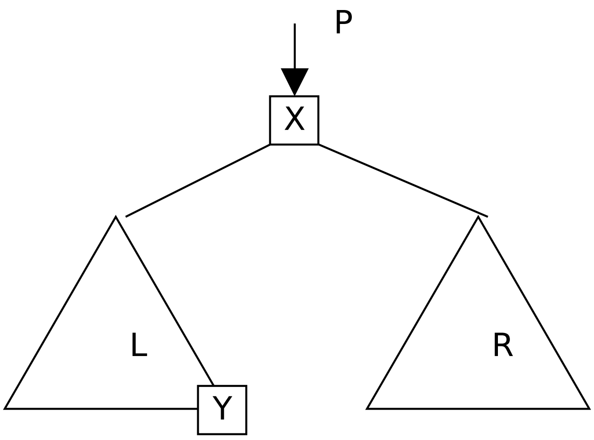
```
procedure Insert(X: TData; var P: PNode);
begin
  if P = nil
  then P := MakeNode(X, nil, nil)
  else
    case Compare(X, P^.Fdata) of
      ls: Insert(X, P^.FLeft);
      eq: {X already present; do not insert again}
      gt: Insert(X, P^.Fright)
    end;
end;
```

4.4 Delete

The Delete operation is somewhat more complicated than the Find and Insert operations. Finding the location of the element X to be deleted proceeds similarly to Find and Insert. For the actual deletion process of a node - referenced by P - containing X, three cases have to be distinguished, however:

- X occurs in a leaf: in this case P may simply be set to **nil**;
- X occurs in a node with only one subtree: in this case P is made to refer to that subtree;
- X occurs in a node with two subtrees (see figure below): in this case removal of node P would result in two subtrees without a parent. Node P is therefore left in place, but an element Y of one of its subtrees is removed and used to replace X . If the binary search property of the whole tree is to be preserved, there are only two candidates for Y : the maximal element of L and the minimal element of R . Let us choose for Y the maximal element of L . Due to the bst property, Y must occur as the rightmost element of L . It can be reached from the root of L by following the right branch as

long as possible, i.e. until a node Q is encountered with no right subtree. The value Y in this node can be copied to node P. Thereafter node Q can be deleted. Since Q has at most one subtree, its removal is easy.



The deletion process outlined above is implemented by means of the following procedures Delete and DelRM. Delete locates the node containing X , if any, and handles the cases of 0 and 1 subtrees directly. The case of two subtrees is handled using the procedure DelRM (delete rightmost), which removes the rightmost node from its argument tree R and returns that node in S .

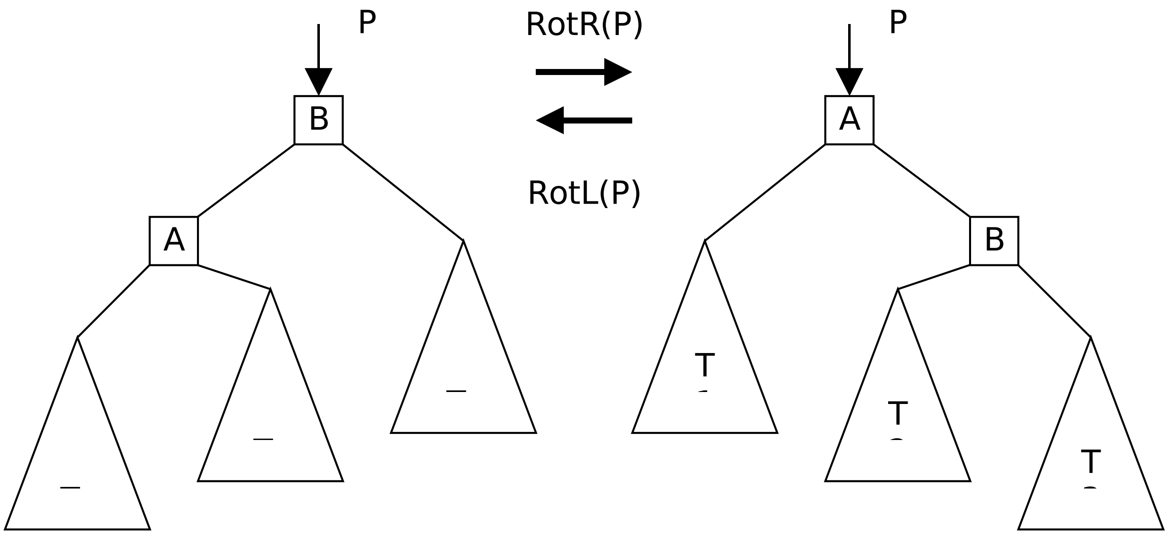
```
procedure DelRM(var R: PNode; var S: PNode);
// Make S refer to rightmost element of tree with root R;
// Remove that element from the tree
begin
  if R^.FRight = nil
  then begin S := R; R := S^.FLeft end
  else DelRM(R^.FRight, S);
end;

procedure Delete(X: TData; var P: PNode);
var
  Q: PNode; // Node to be deleted
begin
  if P = nil
  then {skip}
  else
    case Compare(X, P^.FData) of
      ls: Delete(X, P^.FLeft);
      gt: Delete(X, P^.FRight);
      eq:
        begin
          if P^.FRight = nil then begin Q := P; P := P^.FLeft
end
             else if P^.FLeft = nil then begin Q := P; P :=
P^.FRight end
```

```
          else
            begin
              DelRM(P^.FLeft, Q);
              P^.FData := Q^.FData
            end;
          Dispose(Q)
        end;
    end{case}
end;
```

4.5Rotations

Rotations are operations that preserve the contents and the bst property of a binary tree, but rearrange the relative positions of some neighbouring nodes and subtrees. Rotations may be used to improve the balance of a tree (as will be done in section 5) or to move certain data elements closer to the root. The two rotations RotL (Rotate Left) and RotR(Rotate Right) are depicted graphically below. Note how RotL moves T1 one node closer to the root and T3 one node further away from the root. Note also that RotL and RotR are each other’s inverse.



The rotations can be performed by the following procedures:

```
procedure RotL(var P: PNode);
var
  P1: PNode;
begin
  P1 := P^.FRight;
  P^.FRight := P1^.FLeft;
  P1^.FLeft := P;
```

```
  P := P1;
end;

procedure RotR(var P: PNode);
var
  P1: PNode;
begin
  P1 := P^.FLeft;
  P^.FLeft := P1^.FRight;
  P1^.FRight := P;
  P := P1;
end;
```

5 AVL Trees

5.1Balancing

The operations Find, Insert, and Delete, as defined for binary search trees in section 4 , still have worst case complexity O(N). This is due to the fact that binary search trees without further restrictions may still be very skew (consider e.g. the extreme case in which elements are inserted in increasing order; in that case the tree will degenerate into an ordered linear list). The situation may be improved by requiring that trees are balanced, i.e. that for each node its subtrees are of about the same height. One such definition has been postulated in [Adelson-Velskii & Landis]. Their balance criterion is the following:

> A tree is *balanced* if and only if for every node the heights of its two subtrees differ by at most 1.

The criterion can be formally defined by means of the predicate bal , defined as:

- bal ε = true
- bal < L , a , R > = (bal L) **and** (bal R) **and** | h L - h r | <= 1

Binary search trees satisfying this criterion are often called AVL-trees (after their inventors).

Algorithms for insertion and deletion that do rebalancing critically depend on the way information about the tree’s balance is stored. An extreme solution lies in keeping balance information entirely implicit in the tree structure itself. In this case, however, a node’s balance factor must be rediscovered each time it is affected by an insertion or deletion, resulting in an excessively high overhead. The other extreme is to attribute an explicitly stored balance factor to every node. We shall subsequently interpret a node’s balance factor as the height of its right subtree minus that of its left subtree. To this end we introduce a new type TBal, and we modify the definitions of type TNode and function MakeNode as follows:

```
type
  TBal = -1..1;

  PNode = ^TNode;
  TNode =
  record
    FData: TData;
    FLeft: PNode;
    FRight: PNode;
    FBal: TBal;    // balance factor: h(FRight) - h(FLeft)
  end;

function MakeNode(AData: TData; ALeft: PNode; ARight: PNode; ABal: TBal):
PNode;
var
  H: PNode;
begin
  New(H);
  with H^ do
  begin
    FData := AData;
    FLeft := ALeft;
    FRight := ARight;
    FBal := ABal;
  end;
  Result := H;
end;
```

5.2Find

The function Find is the same as the one for ordinary binary search trees. See 4.2.

5.3Insert

Let us now consider what may happen when a new node is inserted in a balanced tree. Given a root *r* with the left and right subtrees *L* and *R,* three cases must be distinguished. Assume that the new node is inserted in *L* causing its height to increase by 1 :

1. *hL* = *hR: L* and *R* become of unequal height, but the balance criterion is not violated.
2. *hL* < *hR: L* and *R* obtain equal height, i.e., the balance has even been improved.
3. *hL* > *hR:* the balance criterion is violated, and the tree must be restructured.

Consider the tree in Fig. 4.31. Nodes with keys 9 and 11 may be inserted without rebalancing; the tree with root 10 will become one-sided (case 1); the one with root 8 will improve its balance (case 2). Insertion of nodes 1, 3, 5, or 7, however, requires subsequent rebalancing.

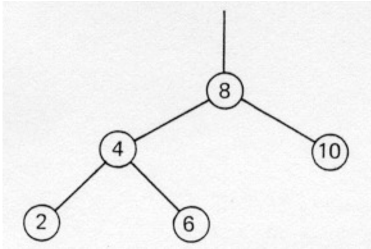


Fig 4.31 Balanced tree

Some careful scrutiny of the situation reveals that there are only two essentially different constellations needing individual treatment. The remaining ones can be derived by symmetry considerations from those two. Case 1 is characterized by inserting keys 1 or 3 in the tree of Fig. 4.31, Case 2 by inserting nodes 5 or 7.

The two cases are generalized in Fig. 4.32 in which rectangular boxes denote subtrees, and the height added by the insertion is indicated by crosses. Simple applications of the rotation operators defined in section 4.5 restore the desired balance:

- For case 1: apply a right rotation to node B;
- For case 2: first apply a left rotation to node A (thus reducing this case to case 1); subsequently apply a right rotation to node C.

Their result is shown in Fig. 4.33; note that the only movements allowed are those occurring in the vertical direction, whereas the relative horizontal positions of the shown nodes and subtrees must remain unchanged.

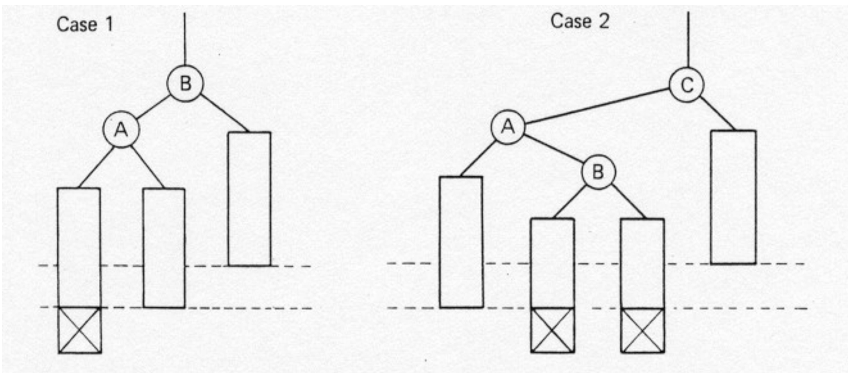


Fig. 4.32 Imbalance resulting from insertion

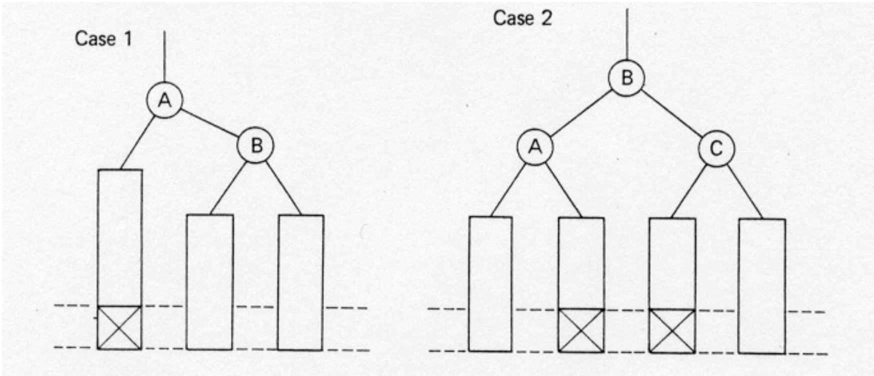


Fig 4.33 Restoring the balance

The process of node insertion consists essentially of the following three consecutive parts :

- Follow the search path until it is verified that the key is not already in the tree.
- Insert the new node and determine the resulting balance factor .
- Retreat along the search path and check the balance factor at each node.

Although this method involves some redundant checking (once balance is established, it need not be checked on that node's ancestors), we shall first adhere to this evidently correct schema because it can be implemented through a pure extension of the already established Insert procedure of section 4.3 . This procedure describes the search operation needed at each single node, and because of its recursive formulation it can easily accommodate an additional operation "on the way back along the search path." At each step, information must be passed as to whether or not the height of the subtree (in which the insertion had been performed) had increased. We therefore extend the procedure's parameter list by the Boolean Higher with the meaning "*the subtree height has increased.*" Clearly, Higher must denote a variable parameter since it is used to transmit a result.

Assume now that the process is returning to a node P^ from the left branch (see Fig. 4.32), with the indication that it has increased its height. We now must distinguish between the three situations involving the subtree heights prior to insertion :

- *hL* < *hR, P^.FBal* = +1, the previous imbalance at p has been equilibrated.
- *hL* = *hR, P^ .FBal* = 0, the weight is now slanted to the left.
- *hL* > *hR, P^.FBal* = -1, rebalancing is necessary.

This leads to the following scheme for procedure Insert (compare with section 4.3):

```
procedure Insert(X: TData; var P: PNode; var Higher: Boolean);
begin
  if P = nil
```

```
  then begin P := MakeNode(X, nil, nil, 0); Higher := true end
  else
    case Compare(X, P^.Fdata) of
      ls:
        begin
          Insert(X, P^.FLeft, Higher);
          if Higher then {Left branch has grown higher}
            case P^.FBal of
              1: begin P^.FBal := 0; Higher := false end;
              0: begin P^.FBal := -1 end;
             -1: begin {Rebalance}
                   // ... REORDER TREE ...
                   P^.FBal := 0;
                   Higher := false;
                 end{-1}
            end{case}
        end;{ls}
      eq: {X already present; do not insert again}
      gt:
        begin
          Insert(X, P^.Fright);
          if Higher then {Right branch has grown higher}
            case P^.FBal of
             -1: begin P^.FBal := 0; Higher := false end;
              0: begin P^.FBal := 1 end;
              1: begin {Rebalance}
                   // ... REORDER TREE ...
                   P^.FBal := 0;
                   Higher := false;
                 end{1}
             end{case}
        end;{gt}
    end;{case}
end;{Insert}
```

In case of rebalancing after insertion in the left subtree, inspection of the balance factor of the root of the left subtree (i.e. P^.FLeft^.FBal) determines whether case 1 or case 2 of Fig. 4.32 is present. If that node has also a higher left than right subtree, then we have to deal with case 1, otherwise with case 2. (Convince yourself that a left subtree with a balance factor equal to 0 at its root cannot occur in this case.) The necessary rebalancing operations are performed by means of the rotation procedures RotL and RotR. Hence the rebalancing code can be refined to:

```
if P^.FLeft^.FBal = -1
then {single R rotation}
  begin
    RotR(P);
    //adjust balance factor: ...
  end
else {double LR rotation}
  begin
    RotL(P^.FLeft);
    RotR(P);
    //adjust balance factor: ...
```

```
                end;
              P^.FBal := 0;
              Higher := false;
```

The rebalancing code following insertion in the right subtree of P is symmetrical to this.

In addition to pointer rotation, the respective node balance factors also have to be adjusted. From fig. 4.33 it is clear that in case 1 the new balance factor of node B (i.e. P^.FRight^.FBal) is 0. In case 2, the old balance factor of node B (now P^.FBal) should be inspected in order to find the new balance factors of node A (now P^.FLeft^.FBal) and of node B (now P^.FRight^.FBal). Adding these adjustments (and their symmetrical counterparts) results in the folowing complete Insert procedure:

```
procedure Insert(X: TData; var P: PNode; var Higher: Boolean);
begin
  if P = nil
  then begin P := MakeNode(X, nil, nil, 0); Higher := true end
  else
    case Compare(X, P^.Fdata) of
      ls:
        begin
          Insert(X, P^.FLeft, Higher);
          if Higher then {Left branch has grown higher}
            case P^.FBal of
              1: begin P^.FBal := 0; Higher := false end;
              0: begin P^.FBal := -1 end;
             -1: begin {Rebalance}
                   if P^.FLeft^.FBal = -1
                   then {single R rotation}
                     begin
                       RotR(P);
                       //adjust balance factor:
                       P^.FRight^.FBal := 0;
                   else {double LR rotation}
                     begin
                       RotL(P^.FLeft);
                       RotR(P);
                       //adjust balance factor:
                       if P^.FBal = -1
                       then
                         begin P^.FLeft^.FBal := 0; P^.FRight^.FBal :=1 end
                       else
                         begin P^.FLeft^.FBal :=-1; P^.FRight^.FBal:=0 end;
                     end;
                   P^.FBal := 0;
                   Higher := false;
                 end{-1}
            end{case}
        end;{ls}
      eq: {X already present; do not insert again}
      gt:
```

```
        begin
          Insert(X, P^.Fright, Higher);
          if Higher then {Right branch has grown higher}
            case P^.FBal of
             -1: begin P^.FBal := 0; Higher := false end;
              0: begin P^.FBal := 1 end;
              1: begin {Rebalance}
                   if P^.FRight^.FBal = 1
                   then {single L rotation}
                     begin
                       RotL(P);
                       //adjust balance factor:
                       P^.FLeft.FBal := 0;
                     end
                   else {double RL rotation}
                     begin
                       RotR(P^.FRight);
                       RotL(P);
                       //adjust balance factor
                       if P^.FBal = +1
                       then
                        begin P^.FRight^.FBal := 0; P^.FLeft^.FBal :=-1 end
                       else
                        begin P^.FRight^.FBal := 1; P^.FLeft^.FBal := 0 end
                     end;
                   P^.FBal := 0;
                   Higher := false;
                 end{1}
             end{case}
        end;{gt}
    end;{case}
end;{Insert}
```

The working principle is shown by Fig. 4.34. Consider the binary tree (a) which consists of two nodes only. Insertion of key 7 first results in an unbalanced tree (i.e., a linear list). Its balancing involves a single left rotation, resulting in the perfectly balanced tree (b). Further insertion of nodes 2 and 1 result in an inbalance of the subtree with root 4. This subtree is balanced by an single right rotation (d). The subsequent insertion of key 3 immediately offsets the balance criterion at the root node 5. Balance is thereafter re-established by the more complicated RL double rotation; the outcome is tree (e). The only candidate for loosing balance after a next insertion is node 5. Indeed, insertion of node 6 must invoke the fourth case of rebalancing outlined in (4.63), the LR double rotation. The final tree is shown in Fig. 4.34(f).

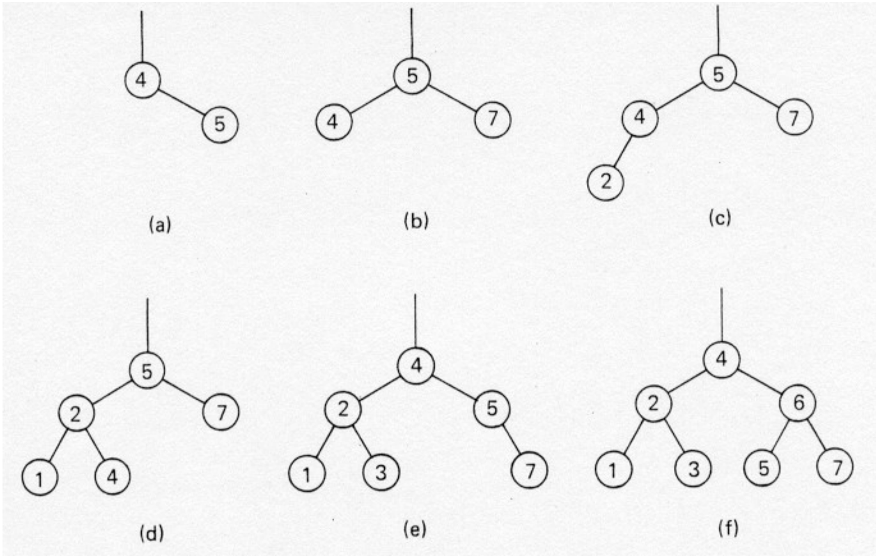


Fig. 4.34 Insertions in balanced tree

5.4Delete

Our experience with tree deletion suggests that in the case of balanced trees deletion will also be more complicated than insertion. This is indeed true, although the rebalancing operation remains essentially the same as for insertion. In particular, rebalancing consists of either a single or a double rotation of nodes.

The basis for balanced tree deletion is procedure Delete of section 4.4. The easy cases are terminal nodes and nodes with only a single descendant. If the node to be deleted has two subtrees, we will again replace it by the rightmost node of its left subtree. As in the case of the balanced Insert procedure (section 5.3), a Boolean variable parameter Shorter is added with the meaning "the height of the subtree has been reduced." Rebalancing has to be considered only when Shorter is true. Shorter is assigned the value true upon finding and deleting a node or if rebalancing itself reduces the height of a subtree. We introduce the two (symmetric) balancing operations in the form of procedures since they have to be invoked from more than one place in the deletion algorithm. Note that Balance1 is applied when the left, Balance2 after the right branch had been reduced in height.

The operation of the procedure is illustrated in Fig. 4.35. Given the balanced tree (a), successive deletion of the nodes with keys 4, 8, 6, 5, 2, 1, and 7 results in the trees (b) ...(h).

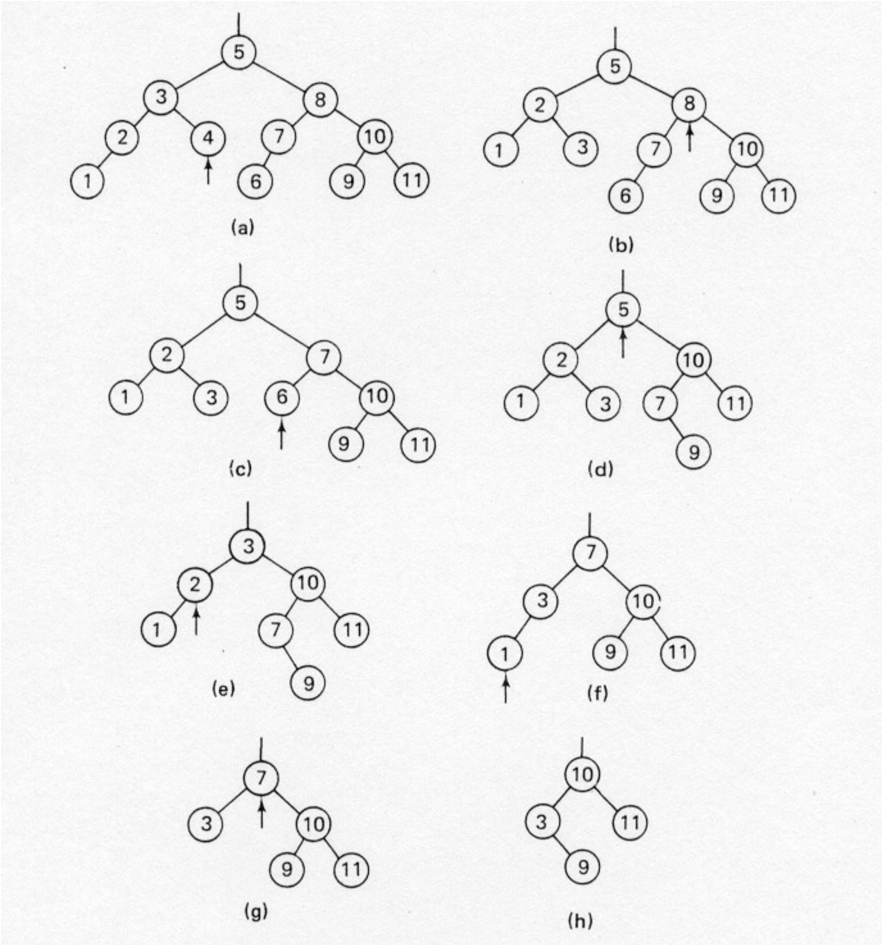


Fig 4.35 Deletions in balanced tree

The deletion of key 4 is simple in itself since it represents a terminal node. However, it results in an unbalanced node 3. Its rebalancing operation involves a single right rotation. Rebalancing becomes again necessary after the deletion of node 6. This time the right subtree of the root (7) is rebalanced by an single left rotation. Deletion of node 2, although in itself straightforward since it has only a single descendant, calls for a complicated *RL* double rotation. The fourth case, an *LR* double rotation, is finally invoked after the removal of node 7, which at first was replaced by the rightmost element of its left subtree, i.e., by the node with key 3 [NOTE: THIS INCORRECT, THERE IS ONLY A SINGLE R ROTATION AROUND 3].

Evidently, deletion of an element in a balanced tree can also be performed with - in the worst case - O(log N*)* operations. An essential difference between the behavior of the insertion and deletion procedures must not be overlooked, however. Whereas insertion of a single key may result in at most one rotation (of two or three nodes), deletion may require a rotation at *every* node along the search path. Consider, for instance, deletion of the rightmost node of a Fibonacci-tree. In this case the deletion of any single node leads to a reduction of the height of the tree; in addition, deletion of its rightmost node requires the maximum number of rotations. This therefore represents the worst choice of node in the worst case of a balanced tree, a rather unlucky combination of chances! How probable are rotations, then, in general ? The surprising result of empirical tests is that whereas one rotation is invoked for approximately every two insections, one is required for every five deletions only. Deletion in balanced trees is therefore about as easy - or as complicated - as insertion.

The code of procedures DelRM and Delete follows (compare with section 4.4):

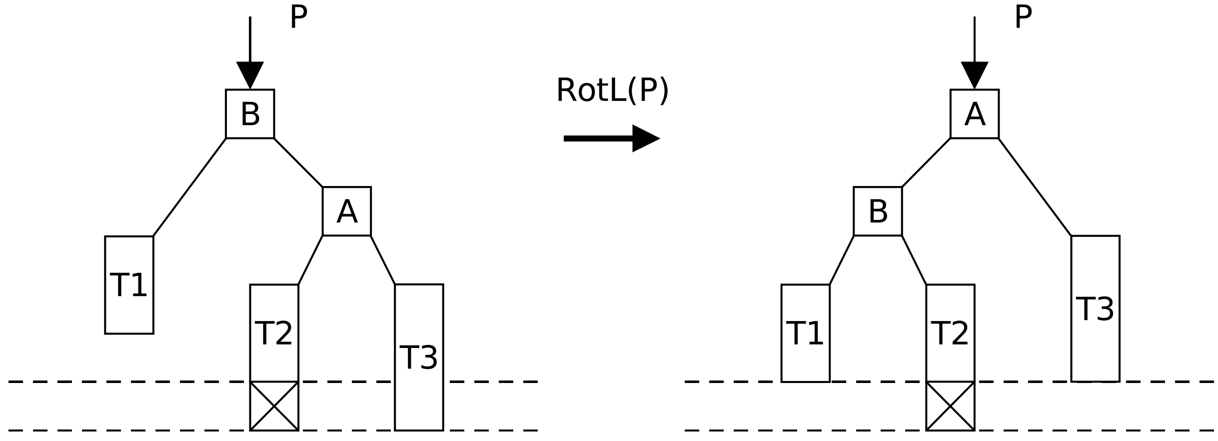
```
procedure DelRM(var R: PNode; var S: PNode; var Shorter: Boolean);
// Make S refer to rightmost element of tree with root R;
// Remove that element from the tree
begin
  if R^.FRight = nil
  then begin S := R; R := S^.FLeft; Shorter := true end
  else
    begin DelRM(R^.FRight, S, Shorter);
      if Shorter then Balance2(R, Shorter)
    end
end;


procedure Delete(X: TData; var P: PNode; var Shorter: Boolean);
var
  Q: PNode;
begin
  if P = nil
  then Shorter := false
  else
    case Compare(X, P^.FData) of
      ls:
        begin
          Delete(X, P^.FLeft, Shorter);
          if Shorter then Balance1(P, Shorter)
        end;
      gt:
        begin
          Delete(X, P^.FRight, Shorter);
          if Shorter then Balance2(P, Shorter)
        end;
      eq:
        begin
          if P^.FRight = nil
          then begin Q := P; P := P^.FLeft; Shorter := true end
          else if P^.FLeft = nil
          then begin Q := P; P := P^.FRight; Shorter := true end
          else
            begin
              DelRM(P^.FLeft, Q, Shorter);
              P^.FData := Q^.FData;
              if Shorter then Balance1(P, Shorter)
            end;
          Dispose(Q)
```

```
        end;{eq}
    end{case}
end;{Delete}
```

The rebalancing operations look similar to those in the Insert procedure, yet there are some subtle differences. Consider e.g. the cases to be dealt with by procedure Balance1. In these cases the left subtree of P has become so short that rebalancing is necessary. Similar to the Insert case, the balance factor of the other subtree (here P^.FRight^.FBal) is inspected to distinguish between two cases:

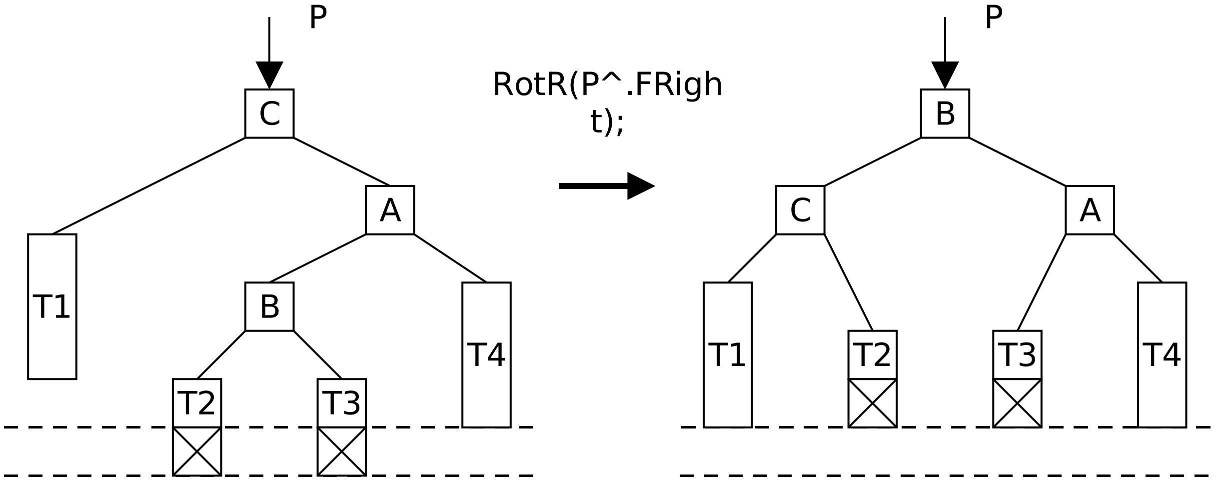
Case 1:
This case corresponds to the left part of the figure below:



Note that the crossed part may or may not be present. This can be detected by inspecting B1 := P^.FRight^.FBal. If B1 = 1, the crossed part is absent; if B1 = 0, it is present. After performing the rotation RotLP), the balance factors can be adjusted, depending on B1. This leads to the following code:

```
if B1 = 0
then
  begin P^.FBal :=-1; P^.FLeft^.FBal :=1; Shorter := false end
else
  begin P^.FBal := 0; P^.FLeft^.FBal := 0 end;
```

Case 2:
This case corresponds to the left part of the figure below.



Note that, unlike fig. 4.32, possibly both crossed parts may be present. This can be detected by inspecting B2 := P^.Fright^.FLeft^.FBal . After performing the rotations, the balance factors can be adjusted, depending on B2. This leads to the following code:

```
if B2=+1 then P^.FLeft^.FBal := -1 else P^.FLeft^.FBal := 0;
if B2=-1 then P^.FRight^.FBal := 1 else P^.FRight^.FBal := 0;
```

```
procedure Balance1(var P: PNode; var S: PNode; var Shorter: Boolean);
var B1, B2: -1..1;
{Shorter = true, left branch has become less high}
begin
  case P^.FBal of
   -1: begin P^.FBal := 0 end;
    0: begin P^.FBal := 1; Shorter := false end;
    1: begin {Rebalance}
         B1 := P^.FRight^.FBal;
         if B1 >= 0
         then {single L rotation}
           begin
             RotL(P);
             //adjust balance factors:
             if B1 = 0
             then
               begin P^.FBal :=-1; P^.FLeft^.FBal :=1; Shorter := false end
             else
               begin P^.FBal := 0; P^.FLeft^.FBal := 0 end;
           end
         else {double RL rotation}
           begin
             B2 := P^.FRight^.FLeft^.FBal;
             RotR(P^.FRight);
             RotL(P);
             //adjust balance factors:
             if B2=+1 then P^.FLeft^.FBal := -1 else P^.FLeft^.FBal := 0;
             if B2=-1 then P^.FRight^.FBal := 1 else P^.FRight^.FBal := 0;
             P^.FBal := 0;
           end;
```

```
      end;{1}
  end{case}
end;{Balance1}
```

The code of procedure Balance2 follows by symmetry:

```
procedure Balance2(var P: PNode; var S: PNode; var Shorter: Boolean);
var B1, B2: -1..1;
{Shorter = true, right branch has become less high}
begin
  case P^.FBal of
    1: begin P^.FBal := 0 end;
    0: begin P^.FBal := -1; Shorter := false end;
   -1: begin {Rebalance}
         B1 := P^.FLeft^.FBal;
         if B1 <= 0
         then {single R rotation}
           begin
             RotR(P);
             //adjust balance factors}
             if B1 = 0
             then
              begin P^.FBal :=1; P^.FRight^.FBal :=-1; Shorter:= false end
             else
               begin P^.FBal := 0; P^.FRight^.FBal := 0 end;
           end
         else {double LR rotation}
           begin
             B2 := P^.FLeft^.FRight^.FBal;
             RotL(P^.FLeft);
             RotR(P);
             //adjust balance factors
             if B2=-1 then P^.FRight^.FBal := 1 else P^.FRight^.FBal := 0;
             if B2= 1 then P^.FLeft^.FBal := -1 else P^.FLeft^.FBal := 0;
             P^.FBal := 0;
           end;
      end;{-1}
  end{case}
end;{Balance2}
```

6References

- Niklaus Wirth; *Algorithms and Data Structures,* Prentice-Hall, Englewood Cliffs, NJ, 1986, ISBN: 0-13-022005-1, pp. 215 – 226.
- G. M. Adelson-Velskii and Y. M. Landis. *An algorithm for the organization of information*. Soviet Math. Dokl., 3:1259--1262, 1962.