CHAPTER 10

PROGRAM DESIGN BY DATA REFINEMENT

The case is made above for studying languages and their implementations. There is, however, a large body of material on program development relating to other application areas. This material frequently uses specification by pre-/post-conditions and abstract data types; the related design techniques are justified by rules about control constructs and data refinement. This chapter exemplifies the relevant parts of VDM. Special emphasis is put on the refinement of abstract data types to machine representations since this is the activity which is more relevant to the early stages of program development.

In this chapter, modules are isolated and developed separately. This focusses on a different area of VDM from that considered in earlier parts. So far attention has been on the order of actions as fixed by combinators; here the concern is with the individual actions. Clearly, both aspects must be considered in a development method.

The example considered is the access to data via keys. This is a problem which is of great importance to many computer applications and which relates to the topic of databases discussed later in this part of the book. The material here is derived from [Fielding 80a]. Full details of the "Rigorous Approach" to software developments are given in [Jones 80a] and overviews are available in [Jones 81b, Stoy 81a]. Another interesting application of this material is given in a recent UK Department of Industry survey of Program Design Methods in which VDM is applied to the definition of KAPSE (Kernel Ada Support Environment) (see also [Clemmensen 82a]).

CONTENTS

10.1 INTRODUCTION

Map data objects are used repeatedly in the specifications of systems. The retrieval of information via a key is also a widespread data processing problem. There are a large number of established methods to implement such maps (e.g. hashing). In this chapter both binary and, so-called, B-trees are developed as representations of abstract maps. These designs are given as examples of the development method rather than in an attempt to contribute new algorithms.

A binary tree can contain one key at each node (data may, optionally, be stored also in the node). The simplest update algorithms do not result in a balanced tree. A B-tree can contain different numbers of keys in different nodes. The number must, however, lie between bounds m (the "order" of the tree) and $2*m$. Furthermore, all "leaves" of a B-tree occur at the same depth. Algorithms exist for insertion and deletion which ensure that the tree remains balanced. The particular type of tree considered here is known as a Bplus-tree. It has the additional property that it can provide sequential access as well as random access via key.

The rigorous method of specification and development is described with the aid of the simpler problem of representing abstract maps as binary trees. This example enables the entire refinement process, from abstract specification down to corresponding program code, to be illustrated.

The B-tree development starts with the same abstract specification of a map from keys to data, with operations defined for finding, inserting and deleting a key. Then, two levels of refinement follow, which represent a map as a tree structure and have corresponding operations which model the operations of the initial specification. The first of these levels represents a tree as a set of nested sets, with the leaves of the tree consisting of mappings from keys to data. The second level represents the tree by using lists - each intermediate node consisting of an ordered list of keys and a list of nodes.

Each stage of the refinement is related to the preceding stage by a retrieve function and is shown to be correct with respect to the preceding stage in accordance with the conditions for the refinement of data types and operations given in [Jones 80a] and summarized in Section 2. No code is provided here for B-trees. A Pascal program is given in full detail in [Fielding 80a].

10.2 THE RIGOROUS METHOD OF SPECIFICATION AND DESIGN

In the "rigorous method", a specification is written as a constructive specification of a data type. Development can then proceed either by operation decomposition or by data refinement. What is described below is the terminology and notation used in "constructive specification" and development by "data refinement".

A program is considered to be an operation (or operations) on a "state" of a particular class. An "operation" can change the values of the components, that comprise a state, but cannot alter its structure. (The notation for operations etc. has been changed from the original in order to fit the current context.)

In order to specify a program, a class of states must be defined. It is best to design the structure of the states by choosing a data type which matches the problem as closely as possible. Such a data type is one which probably cannot be implemented directly, and is known as an "abstract data type" - it is considered to be characterized by its operations. The notation used in defining states is "abstract syntax" (for a description see Chapter 2). An example of a state description is:

$Mkd = Key \xrightarrow{m} Data$

(In contrast, a data type could be defined "implicitly", by using axioms to relate its operations to each other. The "constructive" approach, which is used here, specifies the effects of the operations in terms of the underlying abstract state.)

Operations are specified by using predicates (pre- and post-conditions) as this produces shorter specifications which embody the properties required without specifying how they are to be achieved.

An operation is specified by three clauses, in the following format:

1. $OP: A1 \times A2 \times \ldots \times An \Rightarrow R1 \times R2 \times \ldots \times Rm$

A class of states is associated with a group of operations. The types of any arguments accepted and results produced are shown explicitly.

2. $pre\text{-}OP: State \times A1 \times A2 \times \ldots \times An \rightarrow Bool$

This is a predicate which specifies over what subset of the class of states the operation should work.

3. $post\text{-}OP: State \times A1 \times A2 \times \ldots \times An \times State \times R1 \times R2 \times \ldots \times Rn \rightarrow Bool$

This is a predicate which defines the required relationship between the initial and final states. Examples of operation specifications are:

$FIND: Key \Rightarrow Data$

$pre\text{-}FIND(m,k) \quad \underline{\underline{\Delta}} \quad k \in \underline{dom} m$

$post\text{-}FIND(m,k,m',d) \quad \underline{\underline{\Delta}} \quad m'=m \wedge d=m(k)$

(This operation works on the states shown above (*Mkd*) and returns the data item associated with the given key.)

$INSERT: Key \times Data \Rightarrow$

$pre\text{-}INSERT(m,k,d) \quad \underline{\underline{\Delta}} \quad \neg(k \in \underline{dom} m)$

$post\text{-}INSERT(m,k,d,m') \quad \underline{\underline{\Delta}} \quad m'=m \cup [k \mapsto d]$

$DELETE: Key \Rightarrow$

$pre\text{-}DELETE(m,k) \quad \underline{\underline{\Delta}} \quad k \in \underline{dom} m$

$post\text{-}DELETE(m,k,m') \quad \underline{\underline{\Delta}} \quad m'=m \setminus \{k\}$

Notice that the pre-conditions define these operations to be partial. Subsequent uses of post-conditions show how they can be used to define a range of valid results.

The state of a specification is chosen to be as abstract as possible providing it can be used to describe the required operations. The choice of a map (*Mkd*) satisfies this criterion. Assuming that the implementation language does not support such maps, development must now proceed by "data refinement". That is, a less abstract "representation" is chosen and new operations are defined in terms of the more concrete objects. Precise criteria are laid down for the correctness of such steps. Refinement may take place in several steps.

(The refinement process, like other "top-down" methods should be seen as providing a structure for documentation rather than an order of thought. One particular area where a designer may need to backtrack is in the choice of "data type invariants".)

One possible representation for *Mkd* is binary trees. If a tree is not empty, it consists of a node which contains a key, the associated data and two trees (either or both of which may be <u>*nil*</u>). Thus:

$Bintree = [Binnode]$

$Binnode :: s\text{-}lt{:}Bintree \quad s\text{-}k{:}Key \quad s\text{-}d{:}Data \quad s\text{-}rt{:}Bintree$

< on keys

One criterion for a good specification is the minimization of extra well-formedness conditions on the states. In [Jones 80a] these are referred to as "data type invariants". The definition of *Mkd* is ideal in that there is no need for a data type invariant; it is typical of representations that the invariants become increasingly complex. Here *Bintree* requires an invariant which states that only those trees for which all nodes have ordered keys are considered "valid":

$invnd\colon Binnode \to Bool$
$invnd(mk\text{-}Binnode(lt,k,d,rt)) \ \underline{\underline{\Delta}}$
$\quad (\forall lk \in collkeys(lt))(lk<k) \wedge (\forall rk \in collkeys(rt))(k<rk)$

$collkeys\colon Bintree \to Key\text{-}set$
$collkeys(t) \ \underline{\underline{\Delta}}$
$\quad \underline{if}\ t=\underline{nil}\ \underline{then}\ \{\}$
$\quad \underline{else}\ collkeys(s\text{-}lt(t)) \cup \{s\text{-}k(t)\} \cup collkeys(s\text{-}rt(t))$

The set *Bintree* is now considered to contain only objects all of whose nodes are valid with respect to *invnd*.

The need to record a data type invariant arises because, although it may be evident from the operations, it is required explicitly in later development correctness proofs and will also prevent errors in future alterations to the specification. Each operation must be shown to preserve any data type invariant which might exist. The correctness condition for "preservation of validity" is:

$$(\forall s \in Valids)(pre\text{-}OP(s,args) \wedge post\text{-}OP(s,args,s',res) \supset s' \in Valids)$$

Since, in the specification *Mkd* has no data type invariant, this is obvious for the operations *FIND*, *INSERT* and *DELETE*.

A "retrieve function" relates a representation to its abstraction and is the basis for data refinement proofs. Objects of a representation may contain more information than those of the abstraction and so a retrieve function operates on a state of the representation and retrieves the necessary information for the corresponding state in the abstraction.

The retrieve function for binary trees is:

$retr: Bintree \to Mkd$
$retr(n) \triangleq \underline{if}\ n=\underline{nil}\ \underline{then}\ [\,]$
$\quad \underline{else}\ (\underline{let}\ mk\text{-}Binnode(lt,k,d,rt) = n\ \underline{in}$
$\qquad \underline{merge}([k \mapsto d], retr(lt), retr(rt)))$

Having established the connection between the abstraction and the representation, the correctness of the latter can be considered. It is required that the retrieve function be defined on all valid (representation) states. In the case of *Bintree*, *invnd* ensures that the maps to be merged have disjoint domains and thus *retr* is total over valid binary trees. A more formal proof is given in [Fielding 80a] by showing by (structural) induction that:

$\underline{dom}\ retr(t) = collkeys(t)$

The second correctness condition on the data type itself shows that there exists at least one (valid) representation for each (valid) abstract object; there may exist more than one. For the problem in hand this becomes:

$(\forall m \in Mkd)(\exists t \in Bintree)(retr(t)=m)$

A formal proof by induction on $\underline{domm}$ is given in [Fielding 80a].

Having checked the representation itself, the new operations which work on the representation must be considered. The operation equivalent to *FIND* is:

$FINDBIN: Key \Rightarrow Data$
$pre\text{-}FINDBIN(t,k) \quad \triangleq\ k \in collkeys(t)$
$post\text{-}FINDBIN(t,k,t',d) \triangleq\ t'=t \wedge d=findb(t,k)$

$findb: Bintree \times Key \xrightarrow{\sim} Data$
$pre\text{-}findb(t,k) \triangleq k \in collkeys(t)$

$$\mathit{findb}(\mathit{mk\text{-}Binnode}(lt,mk,md,rt),k) \triangleq$$
$$\underline{\mathit{if}}\ k=mk\ \underline{\mathit{then}}\ md\ \underline{\mathit{else}}\ \underline{\mathit{if}}\ k<mk\ \underline{\mathit{then}}\ \mathit{findb}(lt,k)\ \underline{\mathit{else}}\ \mathit{findb}(rt,k)$$

Notice that *findb* is defined for valid trees because the data type invariant ensures that recursion is always into the appropriate sub-tree. The "preservation of validity" can now be considered: in this case *FINDBIN* is an identity on states so must preserve the invariant.

To establish that *FINDBIN* "models" (i.e. is correct with respect to) *FIND*, two conditions must be established.

The first condition, the "domain condition", shows that the pre-condition is sufficiently wide and has the form, for the binary tree example:

$$(\forall t\in \mathit{Bintree})(\mathit{pre\text{-}FIND}(\mathit{retr}(t),k) \supset \mathit{pre\text{-}FINDBIN}(t,k))$$

Rewriting this using the definitions gives:

$$(\forall t\in \mathit{Bintree})(k\in \underline{\mathit{dom}}\,\mathit{retr}(t) \supset k\in \mathit{collkeys}(t))$$

This can be proved by structural induction on *Bintree* - the details of the proof are not given here, but can be found in [Fielding 80a].

The second condition is known as the "results condition" and requires that given any state satisfying the pre-condition, and the result state after being operated on by the operation on the representation (i.e. a state satisfying the post-condition), this pair of states must satisfy the post-condition of the operation on the abstraction when viewed through the retrieve function. The form of this condition for this example is:

$$(\forall t\in \mathit{Bintree})(\mathit{pre\text{-}FIND}(\mathit{retr}(t),k) \wedge \mathit{post\text{-}FINDBIN}(t,k,t',d) \supset$$
$$\mathit{post\text{-}FIND}(\mathit{retr}(t),k,\mathit{retr}(t'),d))$$

Expanding this gives:

$$(\forall t\in \mathit{Bintree})(k\in \underline{\mathit{dom}}\,\mathit{retr}(t) \wedge t'=t \wedge d=\mathit{findb}(t,k) \supset$$
$$\mathit{retr}(t')=\mathit{retr}(t) \wedge d=(\mathit{retr}(t))(k))$$

Both operations are required to be identities on states so that it is only necessary to show:

$(\forall t \in Bintree)(k \in \underline{dom}\, retr(t) \wedge d = findb(t,k) \supset d = (retr(t))(k))$

This proof can again be done by structural induction and the details are given in [Fielding 80a].

The version of the *INSERT* operation which is to work on the binary tree representation is:

```
INSERTBIN: Key × Data =>
pre-INSERTBIN(t,k,d)        ≜ ¬(k∈collkeys(t))
post-INSERTBIN(t,k,d,t')    ≜ t'=insb(t,k,d)

insb: Bintree × Key × Data ⥲ Bintree
pre-insb(t,k,d) ≜ ¬(k∈collkeys(t))
insb(t,k,d) ≜
    if t=nil then mk-Binnode(nil,k,d,nil)
    else (let mk-Binnode(lt,mk,md,rt) = t  in
          if k<mk then mk-Binnode(insb(lt,k,d),mk,md,rt)
          else mk-Binnode(lt,mk,md,insb(rt,k,d))
```

Since *INSERTBIN* actually changes the tree, it is necessary to show "preservation of validity". This proof is given in detail to illustrate a proof by structural induction. It is required to show that if t is valid (i.e. is in *Bintree*) then:

$$\neg(k \in collkeys(t)) \wedge t' = insb(t,k,d) \supset$$
$$t' \in Bintree \wedge collkeys(t') = collkeys(t) \cup \{k\}$$

As a basis, assume that:

$t = \underline{nil}$

then:

$insb(t,k,d) \triangleq mk\text{-}Binnode(\underline{nil},k,d,\underline{nil})$

The definition of *invnd* gives:

$$invnd(mk\text{-}Binnode(\underline{nil},k,d,\underline{nil}))$$
$$= (\forall lk \in collkeys(\underline{nil}))(lk<k) \wedge (\forall rk \in collkeys(\underline{nil}))(k<rk)$$
$$= \underline{true}$$

and:

$$collkeys(mk\text{-}Binnode(\underline{nil},k,d,\underline{nil})) = \{k\}$$

This base case is like the argument for zero in a proof by mathematical induction on the natural numbers. In a mathematical induction proof, the induction step is from n-1 to n; with structural induction the step is from sub-trees to trees built from such sub-trees. As induction hypothesis it is assumed that *insb* applied to either *t1* or *t2* preserves validity and adds the given key. Then:

```
insb(mk-Binnode(t1,mk,md,t2),k,d) =
     if k<mk then mk-Binnode(insb(t1,k,d),mk,md,t2)
     else mk-Binnode(t1,mk,md,insb(t2,k,d))
```

Consider the case:

$$k<mk$$

by induction hypothesis:

$$insb(t1,k,d)\in Bintree \wedge collkeys(insb(t1,k,d))=collkeys(t1)\cup\{k\}$$

Thus:

$$(\forall lk\in collkeys(insb(t1,k,d)))(lk<mk)$$

because of the assumption of validity on the starting tree and the case assumption. Furthermore:

$$\begin{aligned}&collkeys(mk\text{-}Binnode(insb(t1,k,d),mk,md,t2))\\ =\;& collkeys(insb(t1,k,d)) \cup \{mk\} \cup collkeys(t2)\\ =\;& collkeys(t1) \cup \{mk\} \cup collkeys(t2) \cup \{k\}\\ =\;& collkeys(mk\text{-}Binnode(t1,mk,md,t2)) \cup \{k\}\end{aligned}$$

as required. The other case $(mk<k)$ is proved in the same way. Thus the preservation of validity follows for all trees.

Establishing that *INSERTBIN* models *INSERT* requires that the domain and result conditions be established. The domain condition requires:

$$(\forall t\in Bintree)(pre\text{-}INSERT(retr(t),k,d) \supset pre\text{-}INSERTBIN(t,k,d))$$

which becomes:

$$(\forall t \in Bintree)(\neg(k \in \underline{dom}\, retr(t)) \supset \neg(k \in collkeys(t)))$$

which is proved exactly as for *FINDBIN*. The result rule becomes:

$$(\forall t \in Bintree)(pre\text{-}INSERT(retr(t),k,d) \wedge post\text{-}INSERTBIN(t,k,d,t') \supset post\text{-}INSERT(retr(t),k,d,retr(t')))$$

which expands to:

$$(\forall t \in Bintree)(\neg(k \in \underline{dom}\, retr(t)) \wedge t'=insb(t,k,d) \supset retr(t')=retr(t) \cup [k \mapsto d])$$

The reader can use this example to practice proof by structural induction. The final operation to be provided on binary trees is:

```
DELETEBIN: Key =>
pre-DELETEBIN(t,k)      ≜ k∈collkeys(t)
post-DELETEBIN(t,k,t')  ≜ t'=delb(t,k)

delb: Bintree × Key ⥲ Bintree
pre-delb(t,k) ≜ k ∈ collkeys(t)
delb(mk-Binnode(lt,mk,md,rt),k) ≜
    if k<mk then mk-Binnode(delb(lt,k),mk,md,rt)
    else if mk<k then mk-Binnode(lt,mk,md,delb(rt,k))
         else if lt=nil ∧ rt=nil then nil
              else if lt=nil
                   then (let (rk,rd,rt') = bringlo(rt) in
                         mk-Binnode(lt,rk,rd,rt'))
                   else (let (lk,ld,lt') = bringhi(lt) in
                         mk-Binnode(lt',lk,ld,rt))

bringlo: Bintree ⥲ Key × Data × Bintree
pre-bringlo(t) ≜ t≠nil
bringlo(mk-Binnode(lt,k,d,rt)) ≜
    if lt≠nil then (let (lk,ld,lt') = bringlo(lt) in
                    (lk,ld,mk-Binnode(lt',k,d,rt)))
    else if rt=nil then (k,d,nil)
         else (let (rk,rd,rt') = bringlo(rt) in
              (k,d,mk-Binnode(nil,rk,rd,rt')))
```

(The *bringhi* function is similar to *bringlo*.) It is interesting to observe that, even in the simple case of binary trees, deletion is more complex than insertion; this observation will apply with more force in the case of B-trees. The fact that *DELETEBIN* preserves validity can again be proved by structural induction. The base case must be a tree containing exactly the key to be deleted (i.e. $mk\text{-}Binnode(\underline{nil},k,d,\underline{nil})$) and a subsidiary proof is required to show that:

$$\begin{aligned}&bringlo(t)=(k,d,t') \supset\\ &\quad t'\in Bintree \wedge collkeys(t)=collkeys(t')\cup\{k\} \wedge\\ &\quad d=retr(t)(k) \wedge (\forall k'\in collks(t'))(k<k')\end{aligned}$$

The domain condition for *DELETEBIN* is similar to those above; the reader should be able to produce the appropriate results condition. (Another exercise for the interested reader is to design and justify a version of *DELETEBIN* which identifies the special case where exactly one subnode is $\underline{nil}$ and avoid the "bring" operation.)

The operations *FINDBIN*, *INSERTBIN* and *DELETEBIN* show the main techniques for manipulating binary trees. The *Binarytree* object, however, is not directly realizable in Pascal. A further step of refinement is needed to represent the tree-like structures using pointers and variables on the heap. The Pascal "heap" can be thought of as a mapping from pointers (*Ptr*) to node representations (*Binnoderep*); the root of the tree is an (optionally *nil*) pointer. Thus the state of the actual program is:

$$\begin{aligned}Bintreerep &:: ROOT: [Ptr]\\ &\ HEAP: (Ptr \xrightarrow{m} Binnoderep)\end{aligned}$$

$$\begin{aligned}Binnoderep &:: lptr: [Ptr]\\ &\ key\ : Key\\ &\ data: Data\\ &\ rptr: [Ptr]\end{aligned}$$

This representation requires an invariant stating that all non-$\underline{nil}$ pointers are defined and that the pointers define a tree structure (there are no joins or loops). The retrieve function (to *Bintree*) is obvious and is not given here. It is the essence of the "rigorous approach" to use formal definitions and proofs as appropriate. The knowledge of the formal basis makes it possible to complete details if necessary. The code corresponding to the above development is now given. The fact that the tree is updated in-place, makes some of the code simpler than the "functional programming" style used above. Code to trap situations where operations are used outside their pre-conditions has also been included.

```
program bintree;
type key  = integer;
     ptr  = ↑ node;
     node = record lptr: ptr;
                   key : key;
                   data: char;
                   rptr: ptr
            end;
var root: ptr;

function find(p:ptr; k:key): char;
begin
    if p=nil then writeln('error - not found')
    else with p↑ do
             begin
                if key=k then find := data
                else if k<key then find := find(lptr,k)
                     else find := find(rptr,k)
             end
end;

procedure add(var p:ptr; k:key; d:char);
begin
    if p=nil
       then begin
                new(p);
                with p↑ do
                   begin
                      key := k; data := d; lptr := nil; rptr := nil
                   end
            end
       else with p↑ do
                begin
                   if k<key then add(lptr,k,d)
                   else if key<k then add(rptr,k,d)
                        else writeln('error - not inserted',k)
                end
end;
```

```
  procedure delete(var p:ptr; k:key);
      procedure bringlo(var p: ptr; var k:key; var d:char);
      begin
          with p↑ do
              if lptr<>nil then bringlo(lptr,k,d)
              else begin
                      k := key; d := data;
                      if rptr = nil then dispose(p)
                      else bringlo(rptr,key,data)
                   end
      end;
      procedure bringhi(var p:ptr; var k:key; var d:char);
      begin
          with p↑ do
              if rptr<>nil then bringhi(rptr,k,d)
              else begin
                      k := key; d := data;
                      if lptr = nil then dispose(p)
                      else bringhi(lptr,key,data)
                   end
      end;
  begin
      if p=nil then writeln('error - key not found',k)
      else with p↑ do
          begin
              if k<key then delete(lptr,k)
              else if key < k then delete(rptr,k)
                  else if (lptr=nil) ∧ (rptr=nil) then dispose(p)
                      else if  rptr<>nil  then  bringlo(rptr,key,data)
                          else bringhi(lptr,key,data)
          end
  end { delete };
  :
  :
  end.
```

The code given above is so close to the design on abstract trees that detailed proofs are not given. If, however, it were decided to avoid the use of recursion and program the operations via loops, the rules of operation decomposition of [Jones 80a] (using loop invariants etc.) could be used.

10.3 B-TREE OVERVIEW

B-trees provide a useful structure for storing maps from keys to data. There is minimal space overhead for small indexes while very large ones support fast information retrieval. As described in the preceding section, binary trees can become unbalanced. Although there is much work published on balancing binary trees, one of the attractions of B-trees is the relative simplicity of update algorithms which preserve balance. Perhaps the decisive reason for using B-trees is the ability to choose the "order" of the tree so that node size approximates to the physical block size of the storage medium on which the index is to be stored. For descriptions of B-trees see [Comer 79a, Knuth 75a, Wirth 76a].

If the degree of a node is defined to be the number of sons it has and a leaf is a terminal node which has no sons, then a B-tree of order m satisfies:

(a) The degree of a non-leaf node, other than the root, lies between bounds $m+1$ and $2m+1$.

(b) The degree of the root is between bounds 2 and $2m+1$, unless it is a leaf.

(c) If the root is a leaf, it may contain from 0 to $2m$ keys. Otherwise the number of keys contained in any node lies between bounds m and $2m$ and a non-leaf node with k sons will contain $k-1$ keys.

(d) All the leaves occur at the same depth (i.e. the tree is balanced).

In a B-tree the data associated with a key occurs in the node containing the key. However a Bplus-tree is a special form of B-tree which has all key-data pairs in the leaves and non-leaf nodes do not contain data. The keys in non-leaf nodes serve as separators and are usually copies of some of the keys which are the maximum keys of the leaf nodes. Deletion may sometimes cause non-key values to be left as separators.

The leaves of a Bplus-tree may be linked which facilitates sequential processing of the data by the "next" operation.

A description of the algorithms for performing the find, insert and delete operations on the Bplus-tree is given in section 4 below.

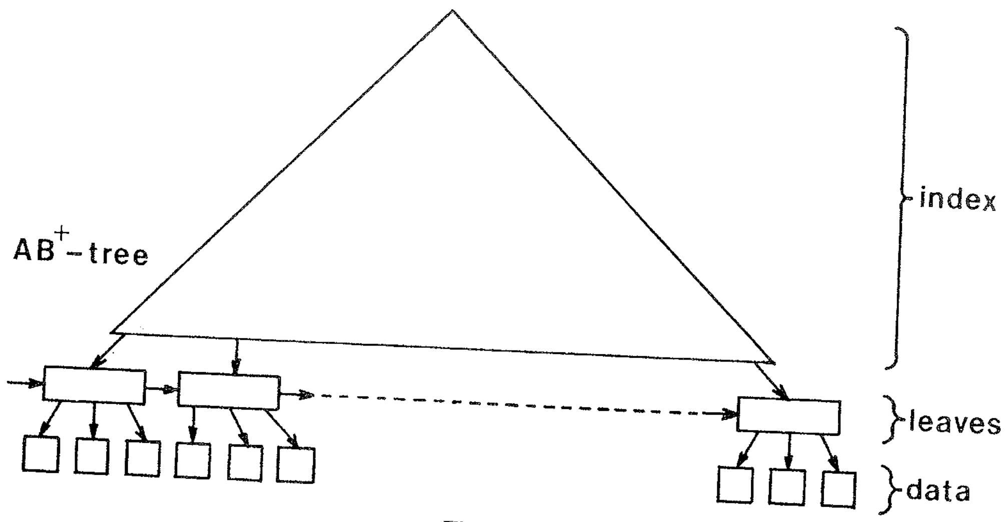

Fig. 1

The Bplus-tree is, of couse, a representation of the mapping (*Mkd*); the operations whose development is to be considered are already specified as *FIND*, *INSERT* and *DELETE*.

10.4 FIRST REPRESENTATION

An essential part of documenting a design is to decide on the order in which design decisions are to be recorded. This representation embodies the decision to use tree-like structures but does not consider issues of order of keys within sub-nodes (the choice between nodes can be thought of for now as being made by searching all sub-nodes!) The B-tree used in this section poses the main problems of node splitting, merging etc. but is sufficiently abstract to make these operations easier to comprehend and justify. It is interesting to note that proving preservation of validity by *INSERTB* and *DELETEB* is more difficult than showing that the individual operations model the specifications.

The "order" of the tree (m) is used in defining the size bounds on nodes. Rather than having an extra argument to most functions, m is referred to freely below. The overall structure of the representation considered in this section is:

$Btree = Node$

$inv b(t) \triangleq t \in Inode \supset 2 \leq size(t) \wedge (\forall sn \in t)(invlosize(sn))$

$Node = Inode \mid Tnode$

Thus the *Node* at the root has rather looser lower bounds on size than other nodes. All nodes, however, share the same upper bounds.

$$Tnode = Key \underset{m}{\to} Data$$
$$invt(tn) \quad \triangleq \; invhisize(tn)$$

$$Inode = Node\text{-}set$$
$$invi(in) \quad \triangleq \; invhisize(in) \wedge balanced(in) \wedge disjks(in)$$

$$size(n) \quad \triangleq \; \underline{if}\ n \in Tnode\ \underline{then}\ \underline{card}\ \underline{dom}n\ \underline{else}\ \underline{card}n$$

$$invlosize(n) \quad \triangleq \; size(n) \geq minisize(n) \wedge (n \in Inode \supset (\forall sn \in n)(invlosize(sn)))$$

$$invhisize(n) \quad \triangleq \; size(n) \leq maxisize(n)$$

$$minisize(n) \quad \triangleq \; \underline{if}\ n \in Tnode\ \underline{then}\ m\ \underline{else}\ m+1$$

$$maxisize(n) \quad \triangleq \; \underline{if}\ n \in Tnode\ \underline{then}\ 2*m\ \underline{else}\ 2*m+1$$

A major aspect of the invariant for an *Inode* is that all *(Tnode)* leaves are at the same depth:

$$balanced(in) \quad \triangleq \; (\exists d \in Nat)(\forall sn \in in)(deptheq(sn,d))$$

$$deptheq(n,d) \quad \triangleq \; \underline{if}\ n \in Tnode\ \underline{then}\ d=1\ \underline{else}\ (\forall sn \in n)(deptheq(sn,d-1))$$

Although this representation is not fixing the way in which keys are split between nodes, it is necessary to ensure that keys are contained in at most one sub-node:

$$disjks(in) \quad \triangleq \; (\forall sn1, sn2 \in in)(sn1 = sn2 \vee is\text{-}disj(collks(sn1), collks(sn2)))$$

$$collks(n) \quad \triangleq \; \underline{if}\ n \in Tnode\ \underline{then}\ \underline{dom}n\ \underline{else}\ \underline{union}\{collks(sn) \mid sn \in n\}$$

$$is\text{-}disj(s1,s2) \; \triangleq \; \neg(\exists e)(e \in s1 \wedge e \in s2)$$

Whenever the sets (Btree, Node etc.) are referred to below, it is assumed that only "valid" objects which satisfy these invariants are to be considered.

The relation between *Btree* and *Mkd* is expressed by the following retrieve function:

$retrn(n) \triangleq \underline{if}\ n \in Tnode\ \underline{then}\ n\ \underline{else}\ \underline{merge}\{retrn(sn)\ |\ sn \in n\}$

The disjoint keys condition on *Inodes* ensures that *retrn* is defined for all (valid) *Btrees*.

The adequacy condition requires that:

$(\forall m \in Mkd)(\exists t \in Btree)(m = retrn(t))$

This can be proved by induction on $\underline{dom}m$ but notice that particular care is required with the basis: it is to permit the representation of small maps that *invlosize* is not applied to the root.

A useful function to aid readability of what follows is:

$kof\colon Key \times Node \to Bool$
$kof(k,n) \triangleq k \in collks(n)$

An immediate lemma, which can be proved by structural induction, is:

$\underline{dom}retrn(n) = \{k\ |\ kof(k,n)\}$

The *FINDB* operation can be explained in terms of Figure 2.

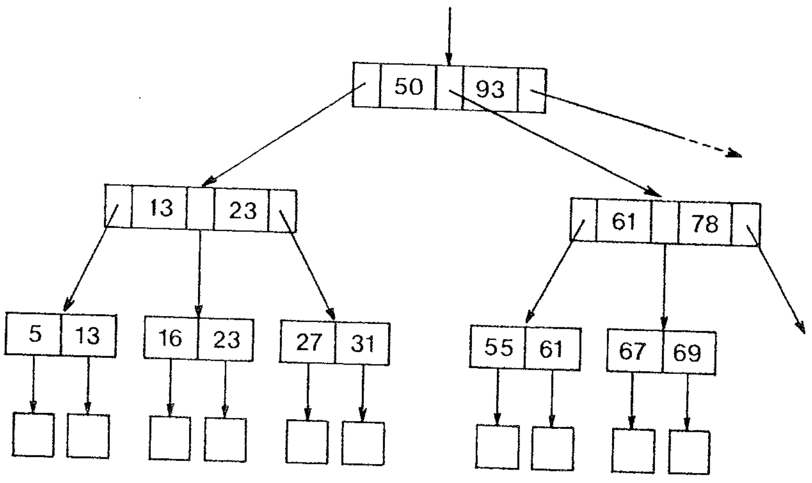

Fig. 2

Suppose that key *67* is to be found. The search starts at the root and 3 possible paths may be taken. For keys ≤ 50 the leftmost path would be taken; for keys > *50* and $\leq$ *93* the centre path is chosen and for keys > *93* the rightmost path is selected. This selection process is repeated at each node until a leaf is reached. If a match is not found in the leaf, then the key is not in the tree. The find must search all the way to a leaf as all the keys reside in the leaves, and the key values in non-leaf nodes simply serve as separators: these nodes do not contain data. (The keys used in examples indicate how selection is achieved in the final algorithm - in the current representation there are no keys in *Inodes*.)

Formally:

$FINDB\colon Key \Rightarrow Data$
$pre\text{-}FINDB(t,k) \quad \triangleq kof(k,\cancel{n})$ t
$post\text{-}FINDB(t,k,t',d) \quad \triangleq t'=t \wedge d=find(k,t)$

$find\colon Key \times Node \xrightarrow{\sim} Data$
$pre\text{-}find(k,n) \quad \triangleq kof(k,n)$
$find(k,n) \quad \triangleq \underline{if}\ n\in Tnode\ \underline{then}\ n(k)\ \underline{else}\ find(k,sel(n,k))$

$sel\colon Inode \times Key \xrightarrow{\sim} Node$
$pre\text{-}sel(in,k) \quad \triangleq kof(k,in)$
$post\text{-}sel(in,k,n) \quad \triangleq n\in in \wedge kof(k,n)$

For valid nodes, a lemma relating *find* and *retrn* is:

$$kof(k,n) \supset find(k,n)=retrn(n)(k)$$

this can be proved by structural induction on the form of *Node*: for elements of *Tnode* the result is immediate; for elements of *Inode* the post-condition of *sel* is required.

The fact that *FINDB* preserves validity is immediate since it does not change the state. The domain condition follows immediately from the lemma on *retrn* and *kof*. The results condition follows immediately from the lemma relating *find* and *retrn*.

The *INSERTB* operation can be thought of as working in two stages. Firstly a find operation is carried out, which must progress all the way down to the correct leaf for insertion. The insertion takes place in the leaf and the balance of the tree is restored, if necessary, by a procedure which works up from the leaf to the root. If the find stops at a leaf that is not full, the new key and data are simply inserted. If however, the leaf is full (i.e. it contains *2m* keys) it must be "split" into two nodes. The smallest *m* keys and the associated data form one node; the largest *m* keys and data form the second node; a copy of the middle key is inserted into the keylist of the parent node to become a separator. If the parent node is not full, the key can be added and the insertion process completed. If the parent node is full, it must be split in a similar manner, but instead of only a copy of the middle key being promoted to the parent node, the actual key is promoted. If the splitting process propagates all the way to the root, and it also has to be split, then the tree increases one level in height: it grows from the root.

For example insertion in a tree of order 1. Consider insertion of the key *56* into:

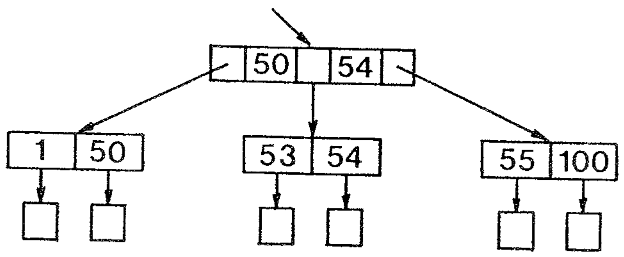

Fig. 3

yields:

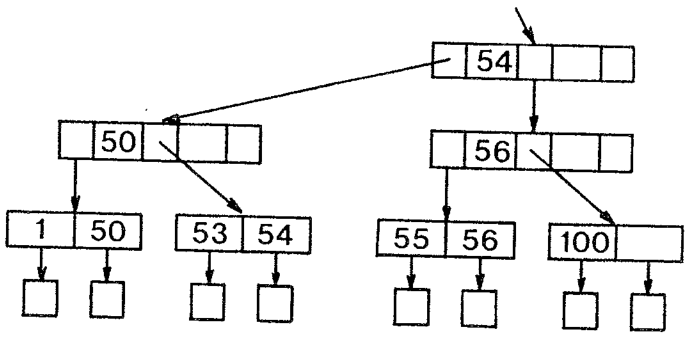

Fig. 4

Formally:

INSERTB: Key × Data ⇒

```
pre-INSERTB(t,k,d)        ≜ ¬kof(k,n)
post-INSERTB(t,k,d,t')    ≜ let ns' = insn(t,k,d) in
                              t' = if cardns'=1 then el(ns') else ns'

insn: Node × Key × Data ⥲ Node-set
pre-insn(n,k,d) ≜ ¬kof(k,n)
insn(n,k,d)        ≜ if n∈Tnode then (let n' = n ∪ [k↦d]                    in
                                        if invhisize(n') then {n'}
                                        else {(n'|ks) | ks∈splits(domn')})
                                  else (let sn∈n                                  in
                                        let sns' = insn(sn,k,d)               in
                                        let n'    = (n-{sn}) ∪ sns'            in
                                        if invhisize(n') then {n'}
                                        else splits(n')

el: X-set ⥲ X
pre-el(s)      ≜ cards = 1
post-el(s,e) ≜ {e} = s

splits: X-set → (X-set)-set
post-splits(s,p) ≜ (∃s1,s2)(p={s1,s2} ∧ s1∪s2=s ∧ is-disj(s1,s2) ∧
                         (∃i∈Nat0)(∀ss∈p)(i≤cardss≤i+1)
```

Notice the non-deterministic aspects of this description: the node in which new data is inserted is not determined nor is the rule for splitting keys between nodes. It is now necessary to show that *INSERTB* preserves validity of *Btree*. Once again, structural induction is used - but here it is desirable to separate the base case as three lemmas. For each of these lemmas, assume:

$tn \in Tnode, \quad \neg kof(k,tn), \quad ns' = insn(tn,k,d)$

The first lemma ensures basic validity and defines the resulting keyset:

$ns' \in Tnode\text{-}set \wedge 1 \leq size(ns') \leq 2 \wedge$
$disjks(ns') \wedge collks(ns') = collks(tn) \cup \{k\}$

the proof is as follows: $\underline{let}\ n' = tn \cup [k \mapsto d]$

consider the case: $invhisize(n')$

$ns' = \{n'\}$

and all results are immediate. Alternatively:

$\neg invhisize(n')$
$ns' = \{(n'|ks) \mid ks \in splits(\underline{dom}\, n')\}$

from *splits* and *invhisize*:

$(\forall snm' \in ns')(invhisize(snm'))$
$disjks(ns')$
$\underline{card}\, ns' = 2$
$collks(ns') = collks(n')$
$\qquad = collks(tn) \cup \{k\}$

which concludes the proof.

The second lemma establishes preservation of *invlosize*:

$invlosize(tn) \supset (\forall nsm' \in ns')(invlosize(nsm'))$

The third lemma is:

$retrn(ns') = retrn(tn) \cup [k \mapsto d]$

The interested reader should find these proofs straightforward. The corresponding three lemmas for (general) elements of *Node* each assume:

$n \in Node, \quad \neg kof(k,n), \quad ns' = insn(n,k,d)$

The lemma on basic validity is:

$ns' \in Node\text{-}set \wedge 1 \le size(ns') \le 2 \wedge$
$(\forall nsn' \in ns')(deptheq(nsn', d) = deptheq(n, d)) \wedge$
$disjks(ns') \wedge collks(ns') = collks(n) \cup \{k\}$

The proof is by structural induction on *Node*. The basis is an immediate consequence of the corresponding lemma on *Tnodes*. For the induction step ($n \in Inode$):

$\underline{let}\ sn \in n$
$\underline{let}\ sns' = insn(sn, k, d)$

By induction hypothesis:

$sns' \in Node\text{-}set \wedge 1 \leq size(sns') \leq 2 \wedge$
$(\forall snsn' \in sns')(deptheq(snsn',d) = deptheq(sn,d)) \wedge$
$disjks(sns') \wedge collks(sns') = collks(sn) \cup \{k\}$

$\underline{let}\ n' = (n - \{sn\}) \cup sns'$

so:

$n' \in Node\text{-}set \wedge size(n) \leq size(n') \leq size(n)+1 \wedge$
$(\forall nsn' \in n')(deptheq(nsn',d) = deptheq(n,d+1)) \wedge$
$disjks(n') \wedge collks(n') = collks(n) \cup \{k\}$

It is now only necesssary to consider cases defined by *invhisize* to conclude the proof.

The lemma on the lower bounds of size is:

$invlosize(n) \supset (\forall snm' \in ns')(invlosize(nsm'))$

and the final lemma on the result is:

$retrn(ns') = retrn(n) \cup [k \mapsto d]$

Both of these lemmas are proved by structural induction using the corresponding *Tnode* results for the basis.

The preservation of validity is an immediate consequence of the first two of the general lemmas. The domain condition follows from the original lemma relating *retrn* and *kof*. The results condition follows from the last lemma discussed.

The *DELETEB* operation is like insertion in that it occurs in two stages, starting with a find to locate the leaf containing the key to be deleted. After this key has been removed, if the leaf then has less than *m* keys, balance the leaf in question; if greater than *2m*, the keys of the two nodes are evenly divided between the nodes and the original separator key in the parent node is overwritten with a copy of the middle key of the two nodes concerned ("redistribution"). If the sum of the keys is less than *2m*, the leaves are "merged" (the opposite of splitting) and the separator key in the parent node is discarded.

Redistribution and merging are slightly different in non-leaf nodes. If the redistribution of two non-leaf nodes occurs, the separator key in the parent node is replaced by the middle key of the two nodes concerned. If a merge occurs in two non-leaf nodes, the separator key in the parent node is pulled down and added to the combined node. If merging propagates all the way up to the root the height of the tree can decrease by one level.

For example deletion in a B-tree of order 2:

(a) Redistribution in leaf nodes

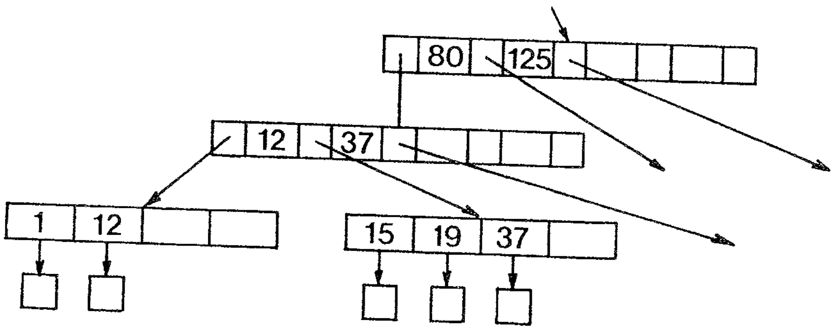

Fig. 5

Deleting key *12* produces:

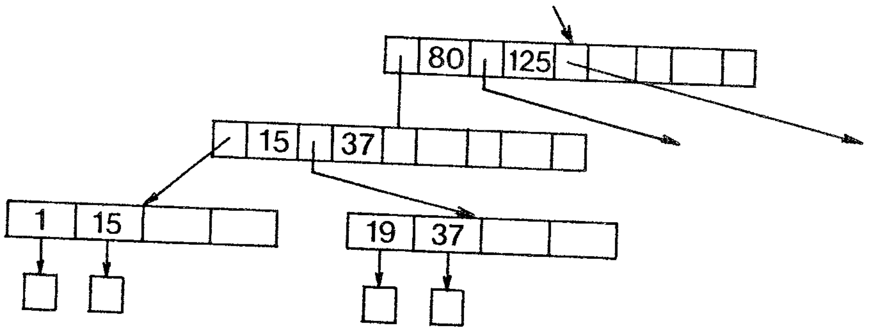

Fig. 6

(b) Merging in leaf nodes:

Now, deleting key *15* produces:

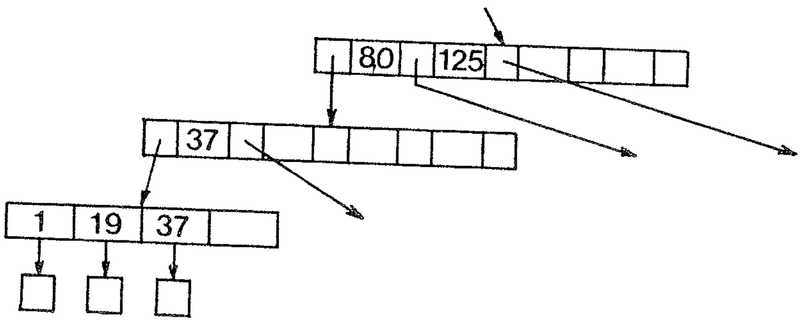

Fig. 7

(c) Redistribution in non-leaf nodes - starting with:

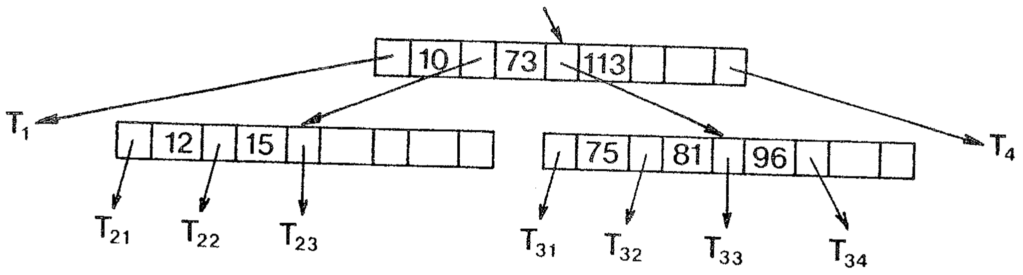

Fig. 8

Merging of nodes T_{22} and T_{23} causes redistribution:

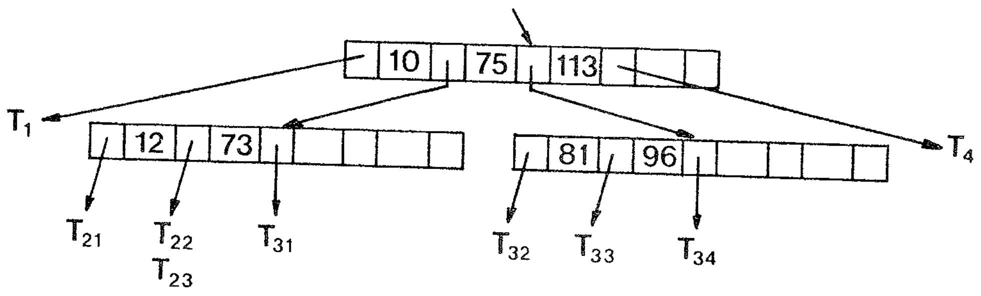

Fig. 9

(d) Merging in non-leaf nodes - a further merge of T_{33}/T_{34} results in:

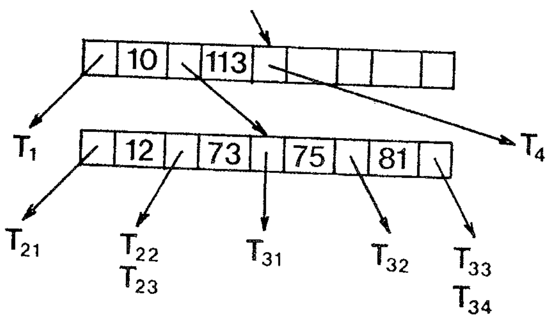

Fig. 10

Formally:

```
DELETEB: Key =>
pre-DELETEB(t,k)       $\triangleq$ kof(k,t)
post-DELETEB(t,k,t') $\triangleq$
    let rn = deln(t,k) in
    t' = if rn∈Tnode ∧ size(rn)=1 then el(rn) else rn

deln: Node × Key $\xrightarrow{\sim}$ Node
pre-deln(n,k) $\triangleq$ kof(k,n)
deln(n,k) $\triangleq$
    if n∈Tnode then n\{k}
    else (let cn = sel(n,k) in
          let rn = del(cn,k) in
          if invlosize(rn) then (n-{cn})∪{rn}
          else (let nn∈n-{cn} in
                let rest = n-{cn,nn} in
                if size(rn)+size(nn)<2*minsize(rn) then rest∪{rn∪nn}
                else if cn ∈ Inode
                    then rest ∪ {nsn           | nsn∈splits(rn∪nn)}
                    else rest ∪ {((rn∪nn)|ks)  | ks∈splits(domrn∪domnn)}))
```

As with the correctness of the B-tree insert operation, the key to justifying *DELETEB* is a series of lemmas. Two lemmas concern *Tnodes* and assume:

tn∈Tnode, kof(k,tn), tn'=deln(tn,k)

The basic validity result is:

tn'∈Tnode ∧ size(tn')=size(tn)-1 ∧ collks(tn')=collks(tn)-{k}

There is no guarantee about preserving the minimum size of a *Tnode*. The result is such that:

retrn(tn') = retrn(tn)\{k}

Both of these lemmas are straightforward. Proofs by induction, using the above results, can be given for three lemmas on general nodes - for each assume:

$n \in Node,\ kof(k,n),\ n' = deln(n,k)$

The three lemmas are:

$$n' \in Node \wedge size(n') < size(n) \wedge$$
$$(deptheq(n',d) = deptheq(n,d)) \wedge collks(n') = collks(n) - \{k\}$$

$$invlosize(n) \wedge n' \in Inode \supset (\forall sn' \in n')(invlosize(sn'))$$

$$retrn(n') = retrn(n) \setminus \{k\}$$

These lemmas make the proof both of preservation of validity and that *DELETEB* models *DELETE* straightforward.

10.5 SECOND REPRESENTATION

Having solved the crucial problem of balancing the trees, this stage of refinement can introduce the keys into *Inodes*. This makes it possible to remove the arbitrary selection of sub-nodes for insertion and the completely unrealistic specification of *sel*. The invariant of this representation can conveniently be split into those constraints already considered and those (e.g. order of the key lists) which could not have been stated earlier. (The authors of this paper are grateful to Lockwood Morris for proposing this split.) This division significantly simplifies the proof.

The definition of the new representation is:

$$\begin{array}{ll}
Btreep & = Nodep \\
invp(t) & \triangleq invb(retrb(t)) \\
Nodep & = Tnodep \mid Inodep \\
Tnodep & = Key \underset{m}{\rightarrow} Data \\
invtp(tnp) & \triangleq invt(retrn(tnp)) \\
Inodep & :: s\text{-}keyl{:}Key^{+}\ \ s\text{-}treel{:}Nodep^{+}
\end{array}$$

$$\begin{array}{ll}
invip(inp) & \triangleq invi(retrn(inp)) \wedge \\
& \quad (\underline{let}\ mk\text{-}Inodep(kl,tl) = inp\ \underline{in} \\
& \quad \underline{len}kl = \underline{len}tl - 1 \wedge \\
& \quad (\forall i \in \underline{inds}\ kl)(setle(collksp(tl[i]),\{kl[i]\}) \wedge \\
& \qquad\qquad setl(\{kl[i]\}, collksp(tl[i+1]))))
\end{array}$$

Notice, that it is a consequence of *invip* that the keylist should be ordered.

$collksp(n) \quad \triangleq \underline{if}\ n \in Tnode\ \underline{then}\ \underline{dom}\ n$
$\quad\quad \underline{else}\ \underline{union}\{collksp(sn) \mid sn \in \underline{elems}\, s\text{-}treel(n)\}$

The relations between sets of keys are:

$setle(ks1,ks2) \triangleq (\forall k1 \in ks1, k2 \in ks2)(k1 \le k2)$

$setl(ks1,ks2) \triangleq (\forall k1 \in ks1, k2 \in ks2)(k1 < k2)$

The necessary retrieve function is:

$retrb: Nodep \to Node$
$retrb(np) \quad \triangleq \underline{if}\ np \in Tnodep\ \underline{then}\ np$
$\quad\quad \underline{else}\ \{retrb(s\text{-}treel(np)[i]) \mid i \in \underline{inds}\, s\text{-}treel(np)\}$

The development and justification of this stage is not pursued in detail. It is, however, worth noticing an unusual feature of this step of data refinement. Because of the freedom to place keys in any order in a *Btree*, not all elements can be represented by a *Btreep*. Strictly speaking, this violates the adequacy condition. There is an explanation in [Jones 80a] as to how to deal with this general problem. Here, however, it is easier to think of the algorithms at this level resolving non-determinacy of the higher level: since the *Btree* algorithms were proven without constraint, any algorithms which simply define the choice to be made are correct. The remaining part of the refinement (i.e. the addition of keylists) fits the usual scheme for data refinement.

Only the find operation is considered on this representation (see [Fielding 80a] for details of the others).

$FINDP: Key \Rightarrow Data$
$pre\text{-}FINDP(tp,k) \quad \triangleq kofp(k,tp)$
$post\text{-}FINDP(tp,k,tp',d) \triangleq tp'=tp \wedge d=findp(k,tp)$

$kofp(k,np) \quad \triangleq k \in collksp(np)$

$findp: Key \times Nodep \xrightarrow{\sim} Data$
$pre\text{-}findp(k,np) \quad \triangleq kofp(k,np)$

```
findp(k,np)           ≜ if npϵTnodep then np(k)
                          else (let i = indexp(k,s-keyl(np)) in
                                findp(k,s-treel(np)[i]))

indexp: Key × Key* → Nat
pre-indexp(k,kl)      ≜ is-ordered(kl)
post-indexp(k,kl,i)   ≜ k≤hdkl ∧ i=1 ∨ kl(len kl)<k ∧ i=lenkl+1
                          ∨ kl[i-1]<k≤kl[i]

is-ordered: Key* → Bool
```

Although the indirect definition of part of the invariant (via *retrb* and *invb*) simplifies the proof at this level, it would be necessary to "bring down" the invariant before proceeding to the next stage. This would involve defining functions such as *invhisizep*.

10.6 FURTHER DEVELOPMENT

The step of representing *Btreep* in, say, Pascal is very similar to the task of representing binary trees considered in section 2 above. Code for UCSD Pascal for all three operations is given in [Fielding 80a]. Possible further extensions include:

The implementation of the *NEXT* operation, for which provision has already been made in the record layout.

The implementation of Bplus-trees using disk files and disk file addresses rather than storage and *pointer* variables and the extension of this to provide a separate file access package for general use (the difficulties of the strong typing of Pascal have to be overcome to achieve this extension).

After the above two extensions the implementation of recovery procedures becomes possible.

Finally, allowing features such as key compression and multi-user access are also possible future developments.