THE SPECIFICATION OF ABSTRACT MAPPINGS AND THEIR IMPLEMENTATION AS B^+-TREES

Elizabeth Fielding

Technical Monograph PRG-18
September 1980

Oxford University Computing Laboratory,
Programming Research Group,
45, Banbury Road,
OXFORD, OX2 6PE

ACKNOWLEDGEMENTS

I am most grateful to Cliff Jones for suggesting and supervising this project. His unfailing interest and guidance helped to make this work both a pleasant and worthwhile experience.

I also appreciate the help given to me by Jim Kaubisch and Malcolm Harper in using the UCSD system and the Sanders printer.

CONTENTS PAGE

PAGE

1. INTRODUCTION

The purpose of this project was to use the "rigorous approach" [5] to software development to develop an implementation of a type of B-tree as a structure for storing mappings from keys to data. A B-tree is a generalised binary tree, for which there exist algorithms for inserting and deleting keys which ensure that the tree remains balanced. The particular type of B-tree specified, known as a B^+-tree, was chosen because it can allow random access to any key as well as sequential access to keys, as all keys reside in the leaves.

Initially, the simpler problem of representing abstract mappings as binary trees was tackled (and is shown) in order to illustrate and understand the entire refinement process, from abstract specification down to corresponding program code. Only the find operation was considered.

The B^+-tree development starts with an abstract specification of a mapping from keys to data, with operations defined for finding, inserting and deleting a key. Then, two levels of specification follow, each less abstract, which represent a mapping as a tree structure and have corresponding operations for finding, insertion and deletion, which model the operations of the initial specification. The first of these levels, Representation 1, 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, Representation 2, represents the tree by using lists - each non-terminal node consisting of an ordered list of keys and a list of nodes, and each terminal node again consisting of a mapping from keys to data.

Each stage of the refinement is related to the preceeding stage by a retrieve function and is shown to be correct with respect to the preceeding stage in accordance with Data Type and Operation Proof Rules for Refinement [5], as described in Chapter 2. As the structure of the data types and operations of Representation 3 are similar to those of Representation 2, the correctness proofs for Representation 2 are appealed to in the arguments of the correctness of Representation 3.

The final stage of the development is the corresponding realization - running PASCAL code which implements the data types and operations. Included in the code as an indication of its correctness are weakest pre-condition type assertions [2] [3].

2. THE RIGOROUS METHOD OF SPECIFICATION AND DEVELOPMENT.

What follows has been summarized from reference [5], which should be consulted for a complete exposition.

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 (or variables) that comprise a state, but cannot alter its structure.

In order to specify a program, a class of states must be defined, and 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 this is known as an *abstract data type* and is considered to be characterized by its operations. The notation used in defining state descriptions is *abstract syntax* (for a description see [5], Ch. 14), and set, list and mapping notations are also used for describing abstract data types. An example of a *state description* is:

Studc :: N:Student-name-set Y:Student-name-set

which defines a class of states

Studc = {<n,y>|n,y ∈ Student-name-set}

This means that any object in the class of states Studc has two variables, with names N and Y, and in any particular state these will each be a set of student names.

A data type could be defined *implicitly*, by using axioms to relate its operations to each other, but the *constructive* approach, which is what has been used, specifies what the effects of the operations will be in terms of the underlying 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 4 clauses, in the following format:

1) States: s

This specifies the name of the class of states for the operation.

2) Type: a1 ... an → r1 ... rm

This specifies the types of any arguments accepted and results produced.

3) pre-OP: State a1 ... an → Bool

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

4) post-OP: State a1 ... an State r1 ... rm → Bool

This is a predicate of two states and defines the required relationship between the initial and final state.

An example of implicit operation definition:

(Using the state Studc, the operation RES tests whether a student

RES

States: Studc

Type: Student-name → Bool

pre-RES(<n,y>,nm) ≙ nm ∈ (n ∪ y)

post-RES(<n,y>,nm,<n',y'>,b) ≙ n' = n *and* y' = y *and* b ⟺ (nm ∈ y)

Now that the specification style and definitions have been defined, the method of stepwise development called *data refinement* can be described.

The initial stage is a specification which is chosen to be as abstract as possible, but must capture the required properties of the problem to be specified. A good abstraction will shorten and clarify the specification. A *data type invariant* is a predicate which muet be true of all states which can be created by an operation in a specification. (Such states are said to be *valid*). The choice of data type is made so as to minimize any data type invariant required. An example of a data type invariant which might be required for states of class Studc, could state that a student name could not be a member of both the N and Y sets simultaneously:

$$inv\text{-}Studc(\langle n,y\rangle) \triangleq n \cap y = \{\}$$

The need to record a data type invariant arises because, although it may be evident from the specification and reality, it will be required explicitly in later development correctness proofs and it will also prevent errors in future alterations to the

specification. After the operations have been specified, each operation must be shown to preserve any data type invariant which might exist. The rule for *preservation of validity* is:

$$(\forall s)(inv(s) \text{ and } pre\text{-}OP(s,args) \text{ and } post\text{-}OP(s,args,s',res) \Rightarrow inv(s')) \qquad (A)$$

Refinement is the term given to creating a less abstract *realization* for an abstract data type, which uses a more concrete data type, known as a *representation*, and has new operations which *model* those of the specification. Refinement is also concerned with relating a realization to its specification and *proof of its correctness*. The operations of the representation should be proved to preserve the validity according to rule (A) for the invariant and states of the representation.

Refinement is an iterative process, consisting of a series of developments, each of which is successively more concrete and is proved to be correct with respect to the stage preceeding. Should earlier invariants prove inadequate, backtracking may be required.

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.

An example of a representation for Studc might be based on a

class of states containing lists of student names:

Studc1 : : NL: Student-name-list BL: Bool-list

and assuming a data type invariant:

inv-Studc1(<nl,bl>) ≙ *len* nl = *len* bl

the corresponding retrieve function might be:

retr-Studc: Studc1 → Studc
retr-Studc(<nl,yl>) ≙ <[nl(i)|1 ≤ i ≤ *len* nl *and not* bl(i)],
[nl(i)|1 ≤ i ≤ *len* nl *and* bl(i)]>

The first part of a *refinement proof* is proof of *data type* correctness. Two rules exist for this.

The first of these rules is used to prove the totality of the retrieve function over valid states of the representation. It has the form:
(Note: a suffix of 1 indicates that the function or object is an element of the realization)

$$(\forall s1)(inv1(s1) \Rightarrow (\exists s)(s = retr(s1) \textit{ and } inv(retr(s1)))) \quad (B)$$

The second of the data type proof rules is concerned with the concept of *adequacy*. That is,

"for each (valid) element of the abstract data type there must exist at least one value of the representation which is mapped by

the retrieve function onto the abstract value". ([5], p183)

There may exist more than one value. The formal proof rule to show adequacy is:

$$(\forall\ s)(inv(s) \Rightarrow (\exists\ s1)(invl(s1)\ \textit{and}\ s = retr(s1))) \qquad (C)$$

The second part of a refinement proof is concerned with *operation modelling*. Two rules exist for proving that each operation of the realization models the corresponding one of the specification.

The first rule, the *domains rule*, shows that the pre-condition is sufficiently wide and has the form:

$$(\forall\ s1)(invl(s1)\ \textit{and}\ pre\text{-}OP(retr(s1),args) \Rightarrow pre\text{-}OP1(s1,args)) \qquad (D)$$

The second rule is known as the *results rule*, and states that given any state satisfying the pre-condition of OP1, and the result state after being operated on by OP1 (i.e. a state satisfying the post-condition of OP1), this pair of states must satisfy the post-condition of OP when viewed through the retrieve function. The form of this rule is:

$$(\forall\ s1)(invl(s1)\ \textit{and}\ pre\text{-}OP1(s1,args)\ \textit{and}\ post\text{-}OP1(s1,args,s1',res) \Rightarrow post\text{-}OP(retr(s1),args,retr(s1'),res)) \qquad (E)$$

The refinement step of providing a realization and proving it correct may be repeated until a sufficiently concrete stage is

reached. As the method is *rigorous*, rather than formal, the refinement proofs need not be done formally if an informal argument can show their truth satisfactorily. In the development of B^+-trees which follows, the proofs of the first stage of refinement are fairly formally presented whereas those of the second stage indicate how the proof could be written (often by appeal to the structure of the earlier proofs).

The final development stage is that of operation decomposition. Examples of formal proofs by weakest pre-conditions [2] [3], were followed in this step which uses "decorating assertions" in the program code as a rigorous argument of its correctness.

3. REPRESENTATION OF ABSTRACT MAPPINGS AS BINARY TREES

3.1 Specification

Since it is intended to use a binary tree as a structure in which to store a mapping from keys to data, using a mapping as the data type in the initial specification is a good abstraction. It allows the find operation to be specified in terms of its effect without prescribing how it is to work.

SPECIFICATION

S0 = Key $\rightarrow$ Data

APPLY0
States: S0
Type: Key $\rightarrow$ Data
pre-APPLY0(s_0,k) $\cong$ k $\in$ *dom* s_0
post-APPLY0(s_0,k,s_0',r) $\cong$ $s_0' = s_0$ *and* r = s_0 (k)

3.2 Representation 1

This stage of the refinement represents a mapping as a binary tree. If the tree is not empty, each node of the tree contains a key and its associated data as well as a pair of pointers (either or both of which may be null), to a left and right subtree respectively. All the keys occurring in the left subtree of a particular node will have values less than the value of the key in the node, and the values of the keys of the right subtree will be greater than the node key. This is stated in the data type invariant.

```
S1 = [Bt1]
Bt1 = S1 Key Data S1



invS1: S1 → Bool
invS1(s1) ≜ s1 = nil or (let <lt,k,d,rt> = s1 in
            (∀ lk ∈ xks(lt))(lk < k) and invS1(lt) and
            (∀ rk ∈ xks(rt))(rk > k) and invS1(rt))

where xks: S1 → Key-set
      xks(s1) ≜ if s1 = nil
                then { }
                else let <lt,k,d,rt> = s1 in
                     {k} ∪ union {xks(lt), xks(rt)}

where union: (X-set)-set → X-set
      union(ss) ≜ {e | (∃ s ∈ ss)(e ∈ s)}



retrS0: S1 → S0
retrS0(s1) ≜ if s1 = nil
             then [ ]
             else let <lt,k,d,rt> = s1 in
                  [k → d] ∪ munion {retrS0(lt), retrS0(rt)}

where munion: (Key →Data)-set → (Key → Data)
      pre-munion(ms) ≜ (∀ m1,m2 ∈ ms)(dom m1 ∩ dom m2 = { } or m1 = m2)
      munion(ms)     ≜ [k → d | (∃ m ∈ ms)(k ∈ dom m and d = m(k))]

NOTE 3.1: pre-munion is fulfilled from retrS0 because of the invariant invS1.



APPLY1
States: S1
Type:   Key → Data
pre-APPLY1(s1,k)      ≜ k ∈ xks(s1)
post-APPLY1(s1,k,s1',d) ≜ s1' = s1 and d = appiyf1(s1,k)

where appiyf1: Bt1  Key → Data
      pre-applyf1(s1,k) ≜ k ∈ xks(s1)
      applyf1(s1,k)     ≜ let <lt,k',d,rt> = s1 in
                          if k = k'
                          then d
                          else if k < k'
                               then appiyf1(lt,k)
                               else appiyf1(rt,k)

NOTE 3.2 applyf1 cannot be undefined because of the pre-condition, pre-appiyf1, and
the invariant, invS1.
```

Two functions which are of use in the correctness proofs which follow are:

is-pdisj: (X-set)*-list $\longrightarrow$ Bool
is-pdisj(sl) $\triangleq$ ($\forall$ i,j $\in$ {1 .. *len* sl} | i $\neq$ j)(is-disj(sl(i), sl(j))

where is-disj: X-set X-set $\longrightarrow$ Bool
is-disj(s1,s2) $\triangleq$ s1 $\cap$ s2 = { }

3.2.1 Proofs of the Correctness of Representation 1

The refinement proofs which follow show that Representation 1 is correct with respect to the specification given in Section 3.1.

(A) Preservation of the invariant:

The rule to be proved is:

pre-APPLY1(s1,k) *and* invS1(s1) *and* post-APPLY1(s1,k,s1',d) $\Longrightarrow$ invS1(s1')

Since APPLY1 is an identity operation on s1, the preservation of the invariant follows immediately.

(B) Totality of the retrieve function

The rule to be proved is:

(1) $(\forall\ s1 \in S1)(invS1(s1) \Longrightarrow (\exists\ s0 \in SO)(s0 = retrSO(s1)))$

Proof: By structural induction on s1

BASIS

(2) $s1 = nil$

(3) $invS1(s1)$

(4) $s0 = [\ ] = retrSO(s1)$ which concludes the basis.

INDUCTIVE HYPOTHESIS

If $s1 = \langle lt,k',d,rt \rangle$ assume

(5) $invS1(lt) \Longrightarrow (\exists\ s0 \in SO)(s0 = retrSO(lt))$

(6) $invS1(rt) \Longrightarrow (\exists\ s0 \in SO)(s0 = retrSO(rt))$

It follows immediately from (5) and (6), that for

$s1 = \langle lt,k',d,rt \rangle$

since

(7) $invS1(s1) \Longrightarrow is\text{-}pdisj(\langle dom\ retrSO(lt), dom\ retrSO(rt), \{k\} \rangle)$

that

(8) $invS1(s1) \Longrightarrow$

$(\exists\ s0 \in SO)(s0 = [k' \longrightarrow d] \cup munion\ \{retrSO(lt), retrSO(rt)\})$ (*C.f.* NOTE 3.1)

that is

(9) $invS1(s1) \Longrightarrow (\exists\ s0 \in SO)(s0 = retrSO(s1))$

(C) Adequacy

The rule to be proved is:

(1) $(\forall\ s0 \in S0)(\exists\ s1 \in S1)(invS1(s1)\ \textit{and}\ s0 = retrS0(s1))$

Proof: By induction on *dom* s0

BASIS

(2) *dom* s0 = [ ]

In this case

(3) s0 = [ ]

(4) s1 = *nil* $\in$ S1 and invS1(s1)

(5) s0 = retrS0(s1) = [ ] which concludes the basis

INDUCTIVE HYPOTHESIS

Assume that if

dom s0 $\neq$ [ ] and k $\in$ *dom* s0 then

(6) $(\forall\ s0' \in \{s \in S0 \mid dom\ s \subseteq (dom\ s0 - \{k\})\})\ ((\exists\ s1' \in S1)(invS1(s1')\ \textit{and}\ s0' = retrS0(s1')))$

It now remains to be shown that s0 can be represented.

Let

{ls,{k},rs} be a partition of *dom* s0 such that

$(\forall\ e \in ls)(ls < k)$ *and* $(\forall\ e \in rs)(e > k)$

then under the induction hypothesis (5):

s0 $\restriction$ ls can be represented by lt $\in$ S1 such that

(7) invS1(lt) *and* s0 $\restriction$ ls = retrS0(lt)

s0 $\restriction$ rs can be represented by rt $\in$ S1 such that

(8) invS1(rt) *and* s0 $\restriction$ rs = retrS0(rt)

From this it follows that

s0 can be represented by s1 = <lt,k,s0(k),rt> and

(9) $(\forall\ lk \in xks(lt))(lk < k)$ *and* $(\forall\ rk \in xks(rt))(k < rk)$

and invS1(lt) *and* invS1(rt)

and s0 = [k$\rightarrow$d] $\cup$ *munion* {retrS0(lt), retrS0(rt)}

which is exactly

(10) invS1(s1) *and* s0 = retrS0(s1), and this concludes the proof.

(D) Domains Rule

The rule to be proved is:

(1) (∀ s1 ∈ S1)(invS1(s1) *and* pre-APPLY(retrSO(s1),k) ⟹ pre-APPLY1(s1,k))

Rewriting this using the definitions of pre-APPLY and pre-APPLY1 gives:

(2) (∀ s1 ∈ S1)(invS1(s1) *and* k ∈ *dom* retrSO(s1) ⟹ k ∈ xks(s1))

Proof: By structural induction on s1

BASIS

(3) s1 = *nil*

In this case (2) becomes

(4) k ∈ { } ⟹ k ∈ { } which is obviously true.

INDUCTIVE HYPOTHESIS

If s1 = <lt,k',d,rt> then assume that

(5) invS1(lt) *and* k ∈ *dom* retrSO(lt) ⟹ k ∈ xks(lt) and

(6) invs1(rt) *and* k ∈ *dom* retrSO(rt) ⟹ k ∈ xks(rt)

It now remains to be shown that (2) is true of

s1 = <lt,k',d,rt>

(7) invS1(s1) ⟹ invS1(lt) *and* invS1(rt)

(by definition of invS1)

(8) invS1(s1) ⟹ is-pdis)(< {k'}, xks(lt), xks(rt)>)

(9) k ∈ *dom* retrSO(s1) =

k ∈ {k'} ∪ *union* (*dom* retrSO(lt), *dom* retrSO(rt))

(10) k ∈ xks(s1) ⟺ k ∈ {k'} ∪ *union* (xks(lt), xks(rt))

Under the induction hypothesis (5) and (6),

11) invS1(s1) *and* k ∈ *dom* retrSO(s1) ⟹ k ∈ xks(s1) (by (7) and (8))

(E) Results Rule

The rule to be proved is:

(1) $(\forall s1 \in S1)(invS1(s1)$ *and* pre-APPLY1$(s1,k)$ *and* post-APPLY1$(s1,k,s1',d)$
$\Longrightarrow$ post-APPLY$(retrSO(s1),k,retrSO(s1'),d))$

Expanding this gives:

(2) $(\forall s1 \in S1)(invS1(s1)$ *and* $k \in xks(s1)$ *and* $s1'=s1$ *and* $d=applyf1(s1,k)$
$\Longrightarrow retrSO(s1')=retrSO(s1)$ *and* $d=retrSO(s1)(k))$

It follows immediately from
$s1'=s1$
that
$retrSO(s1')=retrSO(s1)$
so what must be proved is:

(3) $(\forall s1 \in S1)(invS1(s1)$ *and* $k \in xks(s1) \Longrightarrow applyf1(s1,k) = retrSO(s1)(k))$

Proof: By structural induction on $s1$
BASIS
(4) $s1 = \langle nil, k', d, nil \rangle$
In this case,
(5) $invS1(s1)$ and $k \in xks(s1) \Longrightarrow k'=k$,
hence
(6) $applyf1(s1,k) = retrSO(s1)(k) = d$ which proves the basis.

INDUCTIVE HYPOTHESIS
If $s1 = \langle lt,k',d,rt \rangle$ assume
(7) $invS1(lt)$ *and* $k \in xks(lt) \Longrightarrow applyf1(lt,k) = retrSO(lt)(k)$ and
(8) $invS1(rt)$ *and* $k \in xks(rt) \Longrightarrow applyf1(rt,k) = retrSO(rt)(k)$

Now for $s1 = \langle lt,k',d,rt \rangle$
Case 1:
(9) $k = k'$ *and* $invS1(s1)$ *and* $k \in xks(s1)$
As for the basis, $applyf1(s1,k) = retrSO(s1)(k) = d$ (by $invS1(s1)$)
Case 2:
(10) $k < k'$ *and* $invS1(s1)$ *and* $k \in xks(s1)$
In this case $k \in xks(lt)$ (by $invS1(s1)$)
(11) $applyf1(s1,k) = applyf1(lt,k)$ and
(12) $retrSO(s1)(k) = retrSO(lt)(k)$
under the induction hypothesis (7) and (8) these are equivalent.
Case 3:
(13) $k > k'$ *and* $invS1(s1)$ *and* $k \in xks(s1)$
This case is proved similarly to Case 2, which completes the proof of the results rule.

This level of refinement models an array, and in effect maps a binary tree onto linear storage.

REPRESENTATION 2

```
S2    : ROOT:[Ptr]  ARRAY:Ptr ⟶ Node2
Node2 : [Ptr]  Key  Date  [Ptr]


invS2: S2 ⟶ Bool
invS2(s2) ≜ (let pm = pmap(ARRAY(s2)) in
             if not (union rng pm ⊆ dom pm) then false else
             keys-are-ordered(ROOT(s2), ARRAY(s2)) and has-no-loops(pm))

where keys-are-ordered: [Ptr]  (Ptr ⟶ Node2) ⟶ Bool
      pre-keys-are-ordered(ptr,m) ≜ union rng pm ⊆ dom pm
      keys-are-ordered(ptr,m)     ≜ ptr = nil or
                                    (let <lp,k,d,rp> = m(ptr) in
                                    (∀ lk ∈ xks2(lp,m))(lk < k) and
                                    (∀ rk ∈ xks2(rp,m))(rk > k) and
                                    keys-are-ordered(lp,m) and
                                    keys-are-ordered(rp,m)

where xks2: [Ptr]  (Ptr ⟶ Node2) ⟶ Key-set
      pre-xks2(ptr,m) ≜ union rng pm ⊆ dom pm
      xks2(ptr,m)     ≜ if ptr = nil
                        then { }
                        else let <lp,k,d,rp> = m(ptr) in
                             {k} ∪ union {xks2(lp,m), xks2(rp,m)}

where pmap: (Ptr ⟶ Node2) ⟶ (Ptr ⟶ Ptr-set)
      pmap(m) ≜ [p ↦ (let <lp, , ,rp> = m(p) in
                {lp,rp} - { nil })| p ∈ dom m]

where has-no-loops: (Ptr ⟶ Ptr-set) ⟶ Bool
      pre-has-no-loops(pm) ≜ union rng pm ⊆ dom pm
      has-no-loops(pm)     ≜ (∀ p ∈ dom pm)
                             (p ∉ union {reachable(ptr)| ptr ∈ pm(p)})

where reachable: Ptr  (Ptr ⟶ Ptr-set) ⟶ Ptr-set
      reachable(p,pm) ≜ build-set(p,pm,{ })

where build-set: Ptr  (Ptr ⟶ Ptr-set)  Ptr-set ⟶ Ptr-set
      build-set(ptr,pm,ps) ≜ if pm(ptr) = { } or ptr ∈ ps
                             then ps else
                             union {build-set(p,pm,{ptr} ∪ ps)| p ∈ pm(ptr)}
```

```
retrS1: S2 ⟶ St
retrS1(s2) ≅ retrnS1(ROOT(s2), ARRAY(s2))

where retrnS1: [Ptr]  (Ptr ⟶ Node2) ⟶ [Bt1]
      retrnS1(ptr,m) ≅ if ptr = nil
                       then nil
                       else let <lp,k,d,rp> = m(ptr) in
                            <retrnS1(lp,m),k,d,retrnS1(rp,m)>
```

```
APPLY2
States: S2
Type:   Key ⟶ Data
pre-APPLY2(s2,k) ≅ k ∈ xks2(ROOT(s2), ARRAY(s2))
post-APPLY2(s2,k,s2',d) ≅ s2' = s2 and
                           d = applyf2(ROOT(s2),ARRAY(s2),k)

where applyf2 Ptr  (Ptr ⟶ Node2)  Key ⟶ Data
      pre-applyf2(ptr,m,k) ≅ k ∈ xks2(ptr,m)
      applyf2(ptr,m,k) ≅ let <lp,k',d,rp> = m(ptr) in
                          if k = k'
                          then d
                          else if k < k'
                               then applyf2(lp,m,k)
                               else applyf2(rp,m,k)
```

The proofs of the correctness of Representation 2 with respect to Representation 1 are not shown here. The purpose of this example of abstract maps on binary trees is only to illustrate the data refinement method which is used in specifying B^+-trees, in which more complicated data refinement proofs are done.

3.4 Realization

This gives the PASCAL code, including assertions, corresponding to APPLY2. The implementation of the ARRAY: Ptr ⟶ Node2 component of S2 is implicit in the pointer type variables of PASCAL. An invariant, invS3, corresponding to the invariant invS2, must be true of the pointers.

APPLY3

The following data definitions are required for the procedure apply3:

```
TYPE data_type = ... ;
     ptr = ^node;
     node = RECORD
              key:integer;
              data:data_type;
              lptr,rptr:ptr
            END;
VAR root:ptr;
    k:integer;
    res:data_type;
```

The procedure, apply3, is:

```
PROCEDURE apply3 (root:ptr; k:integer; res:data_type);
VAR p:ptr;
   node_key:integer;
BEGIN  {APPLY3}                    {invS3(s3) and k ∈ xks3(root)}
 p:=root;
 if p = nil
   then terminate-error
   else node_key:=p^.key;
 while node_key <> k do
   begin                           {node_key = p^.key and invS3(s3) and
                                    k ∈ xks3(p) and node_key ≠ k}
     if k < node_key
       then p:=p^.lptr
       else p:=p^.rptr;
     if p = nil
       then terminate-error
       else node_key:=p^.key
   end;                            {node_key = p^.key and invS3(s3) and
                                    k ∈ xks3(p) and node_key = k}
 res:=p^.data
END;   {APPLY3}                    {res = applyf2(retrS2(root),k)}
```

Termination follows from depth3(p) decreasing on each iteration of the loop (cf. has-no-loops).

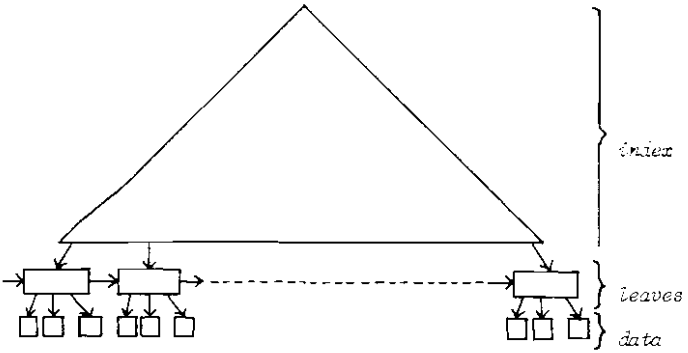
4. REPRESENTATION OF ABSTRACT MAPPINGS AS B+-TREES

4.1 Description of B-trees [1] [6] [7]

A B-tree is a useful structure for storing large mappings from keys to data, where the keys are unique and have some natural order. A B-tree is a generalization of a binary tree, and a B^+-tree is a special form of B-tree in which all the keys and data reside in the leaves. A B-tree of order m has at least m and at most 2m keys at each non-leaf node other than the root, and one more pointer to a descendant node than key in each of these nodes (i.e. between m+1 and 2m+1 pointers). The leaves must contain from m to 2m keys and, in the case of a B^+-tree, the same number of pointers to data as keys. Thus the nodes are always at least half full. A B-tree always remains balanced - all the leaves occur at the same depth. Unless the root is a leaf, it must contain at least 1 key and 2 pointers.

A B-tree can allow the following operations to be performed:
(assume k_i denotes the ith key and d_i the associated data)
find: retrieve data d_i associated with a given key k_i
insert: add a key, k_i and its data, d_i to the mapping, provided that k_i is unique
delete: remove data d_i associated with a given key k_i
next: retrieve d_{i+1} given that d_i has just been retrieved

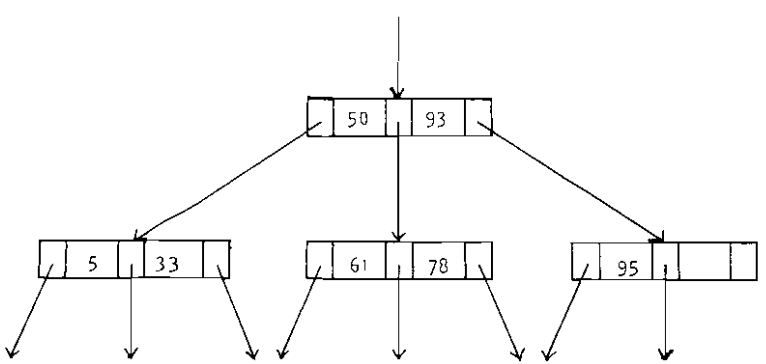
A B^+-tree is organized into an index of non-leaf nodes, and a sequence set of leaf nodes, which may be linked sequentially from left to right as depicted, which facilitates the next operation which is laborious in an ordinary B-tree.



A B^+-tree

The algorithms for insertion and deletion ensure that the B-tree always remains balanced. The find, insert and delete operations are described below for a basic B-tree, with examples, and the differences in the algorithms for B^+-trees are then given in each case.

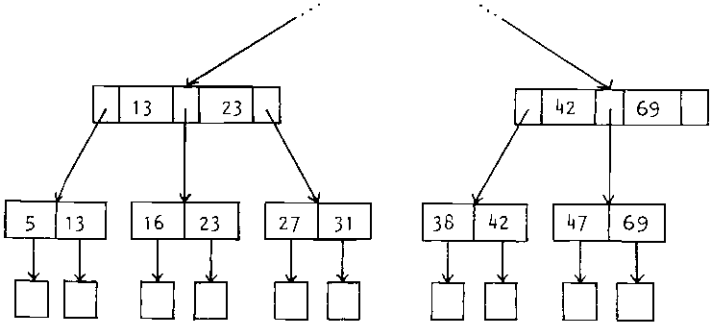
The find operation



A B-tree of order 3

Suppose that key 67 is to be found. The search starts at the root and 3 possible paths may be taken. For keys $\leq$ 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 an exact match is found or a leaf is reached - which denotes that the key has not been found.

For a B^+-tree, a find operation 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 as these nodes do not contain data.



A B^+-tree of order 1

The insert operation

This occurs 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 with the smallest m keys and the associated data in one node,

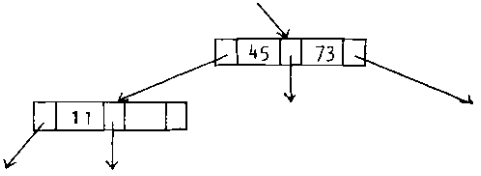
the largest m keys and data in a second node, and the middle key is inserted into the keylist of the parent node to become a separator. If the parent node is not full, the key can be added and the insertion process completed. If the parent node is full, it must be split in a similar manner. If the splitting 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.

In the case of a B^+-tree, the insertion algorithm is similar, with the only difference occurring when a leaf node is split. Then, instead of the middle key being promoted to the parent node, only a copy of this key is promoted, as all the keys must reside in the leaves. Otherwise the insert operation works in the same way as for B-trees.

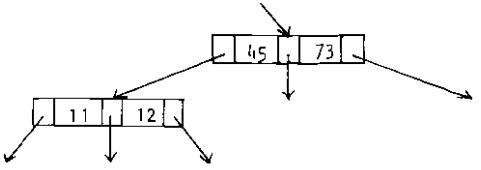
Examples

Insertion in a B-tree of order 1

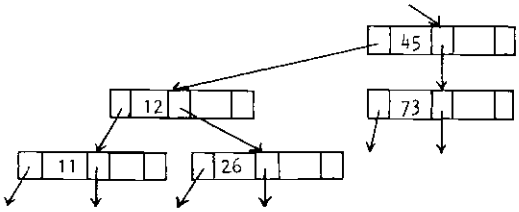
Starting with a B-tree of the form:



Insertion of the key 12 would yield:

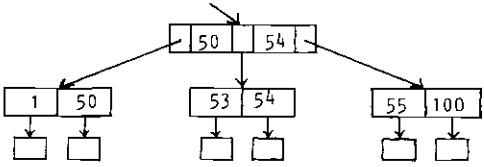


and insertion of the key 26 would produce:

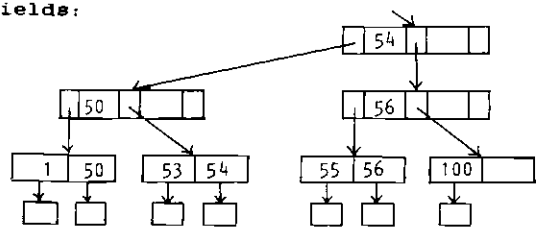


Insertion in a B^+-tree of order 1

Insertion of the key 56 into:



yields:



The delete operation

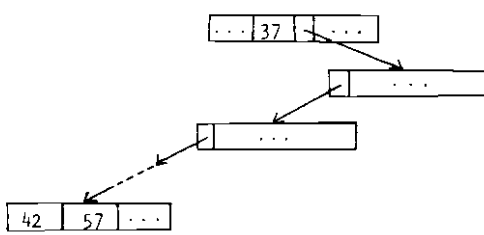
Deletion also occurs in two stages, starting with a find to locate the node containing the key to be deleted. If this key does not reside in a leaf, an adjacent key has to be found in a leaf and put into the position of the deleted key - this moves the empty position to the leaf. An adjacent key is obtained from the leftmost leaf of the right subtree of the deleted key position. If the leaf then has less than m keys, balance must be restored. If the sum of the keys of a neighbouring leaf and the leaf in question is 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 replaced (*redistribution*). If the sum of the keys is less than 2m, the nodes are *merged* (the opposite of splitting) and 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.

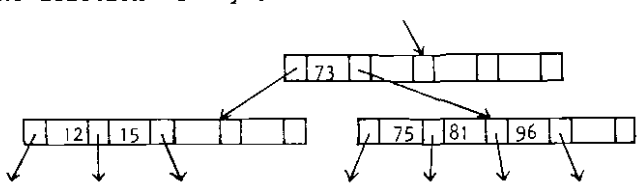
Deletion is simpler in a B$^+$-tree, as non-key values may be left in the non-leaf nodes, and the key to be deleted will reside in a leaf. If a redistribution of two leaf nodes occurs, the separator key in the parent node must be overwritten with a copy of the middle key of the two nodes concerned. If a merge occurs in two leaf nodes, the separator key in the parent node is discarded. In the rest of the tree the delete operation works in exactly the same manner as in a basic B-tree.

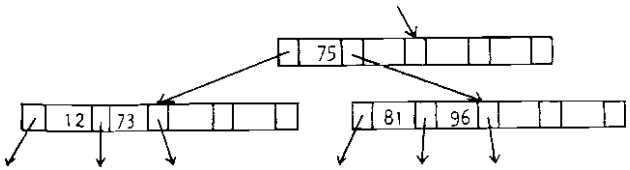
Examples
Deletion in a B-tree of order 2

The deletion of key 37 in the tree depicted below, requires that the next sequential key, 42, be found and put into the empty slot. The key to swap into position is found in the leftmost leaf of the subtree on the right of the empty slot.

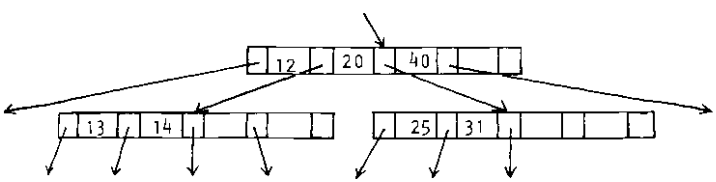


(a) Redistribution
The deletion of key 15 in

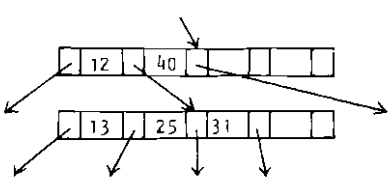




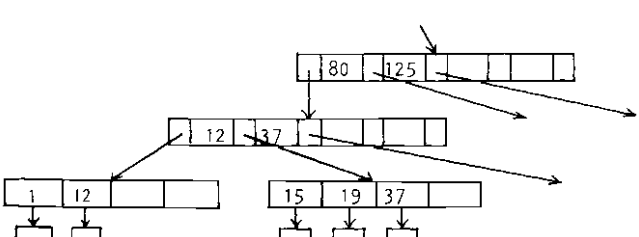
(b) Merging
Deletion of key 14 in



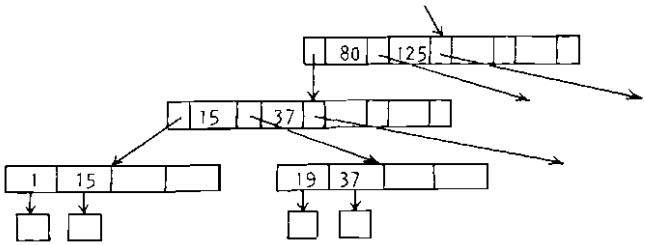
causes merging



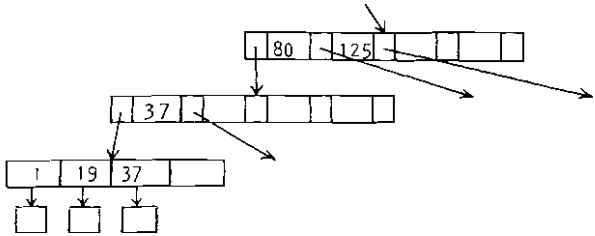
Deletion in a B^+-tree of order 2
(a) Redistribution



Deleting key 12 produces:



(b) Merging
Now, deleting key 15 produces:



For non-leaf nodes the deletion processes of redistribution and merging are identical to those processes acting on a B-tree, as shown in the preceeding examples.

4.2 Specification

The specification uses the same abstract data type as was used for binary trees, but it has been extended by the definition of two further operations, insert and delete.

SPECIFICATION

M = Key $\longrightarrow$ Data

FIND
States: M
Type: Key $\longrightarrow$ Data
pre-FIND(m,k) $\cong$ k $\in$ *dom* m
post-FIND(m,k,m',d) $\cong$ m' = m *and* d = m(k)

INSERT
States: M
Type: Key Data $\longrightarrow$
pre-INSERT(m,k,d) $\cong$ k $\notin$ *dom* m
post-INSERT(m,k,d,m') $\cong$ m' = m $\cup$ [k$\longrightarrow$d]

DELETE
States: M
Type: Key $\longrightarrow$
pre-DELETE(m,k) $\cong$ k $\in$ *dom* m
post-DELETE(m,k,m') $\cong$ m' = m\{k}

4.3 Representation 1

This stage uses as a representation a tree structure in the form of nested sets. This data type was chosen because it captures the essential properties of the node splitting, merging and redistributing of the B^+-tree algorithms for insertion and deletion. However the details of the actual organization of keys in the index part of the tree does not have to be specified.

The main difficulty in the refinement proofs for this level was in proving that the insert and delete operations preserve the invariant.

4.3.1 Data Structure and Data Type Invariant

```
B-tree :: ORDER: Nat  TREE: Node

Node  = Inode | Tnode
Inode = Node-set
Tnode = Key ⟶ Data

inv: B-tree ⟶ Bool
inv(t) ≜ invr(ORDER(t), TREE(t))

where invr: Nat  Node ⟶ Bool
      invr(m,n) ≜ common-inv(n) and size-invr(m,n)

where common-inv: Node ⟶ Bool
      common-inv(n) ≜ cases of n
                      n ∈ Inode: keysets-are-ordered(n) and balanced(n) and
                                 (∀ sn ∈ n)(common-inv(sn))
                      n ∈ Tnode: true
                      end

where size-invr: Nat Node ⟶ Bool
      size-invr(m,n) ≜ cases of n
                       n ∈ Inode: 2 ≤ size(n) ≤ 2*m+1 and
                                  (∀ sn ∈ n)(size-inv(sn))
                       n ∈ Tnode: size(n) ≤ 2*m
                       end

where size-inv: Nat  Node ⟶ Bool
      size-inv(m,n) ≜ cases of n
                      n ∈ Inode: m+1 ≤ size(n) ≤ 2*m+1 and
                                 (∀ sn ∈ n)(size-inv(m,sn))
                      n ∈ Tnode: m ≤ size(n) ≤ 2*m
                      end
```

NOTE 4.1: The size invariant of an Inode is related to the number of descendants it can have. $m+1$ has been used so that the definitions will hold for a tree of *any* order ≥ 1. The size invariant of a Tnode specifies how many keys can occur in the node.

```
where size: Node ⟶ Nat
      size(n) ≜ cases of n
                n ∈ Inode: card n
                n ∈ Tnode: card dom n
                end
```

```
where keysets-are-ordered: Node → Bool
      keysets-are-ordered(n) ≜ cases of n
                                 n ∈ Inode: (∀ s1,s2 ∈ n)((s1 = s2) or
                                          (collect-keys(s1) << collect-keys(s2))
                                          or
                                          (collect-keys(s1) >> collect-keys(s2)))
                                 n ∈ Tnode: true
                                 end

where collect-keys: Node → Key-set
      collect-keys(n) ≜ cases of n
                          n ∈ Inode: union {collect-keys(sn) | sn ∈ n}
                          n ∈ Tnode: dom n
                          end

where <<: Nat-set  Nat-set → Bool
      s1 << s2 ≜ (∀ e1 ∈ s1)(∀ e2 ∈ s2)(e1 < e2)

where >>: Nat-set  Nat-set → Bool
      s1 >> s2 ≜ (∀ e1 ∈ s1)(∀ e2 ∈ s2)(e1 > e2)

where union: (X-set)-set → X-set
      union(ss) ≜ {e | (∃ s ∈ ss)(e ∈ s)}

where balanced: Node → Bool
      balanced(n) ≜ card depths(n) = 1

where depths: Node → Nat-set
      depths(n) ≜ cases of n
                    n ∈ Inode: union {depths(sn) | sn ∈ n} ++ 1
                    n ∈ Tnode: {1}
                    end

where ++: Int-set  Nat → Int-set
      s ++ i ≜ {e+i | e ∈ s}
```

4.3.2 The Retrieve Function

```
retr: B-tree → (Key → Data)
retr(t) ≜ retrn(TREE(t))

where retrn: Node → (Key → Data)
      retrn(n) ≜ cases of n
                   n ∈ Inode: munion {retrn(sn) | sn ∈ n}
                   n ∈ Tnode: n
                   end

where munion: (Key → Data)-set → (Key → Data)
      pre-munion(ms) ≜ (∀ m1,m2 ∈ ms)(dom m1 ∩ dom m2 = { } or m1 = m2)
      munion(ms)     ≜ [k → d | (∃ m ∈ ms)(k ∈ dom m and d = m(k))]
```

```
FIND1
States: B-tree
Type:   Key ⟶ Data
pre-FIND1(t,k)     ≜ k ∈ collect-keys(TREE(t))
post-FIND(t,k,t',r) ≜ t' = t and r = find(k,TREE(t))

where find: Key  Node ⟶ Data
      pre-find(k,n) ≜ k ∈ collect-keys(n)
      find(k,n)     ≜ cases of n
                        n ∈ Inode: find(k,select(n,k))
                        n ∈ Tnode: n(k)
                        end

where select: Inode  Key ⟶ Node
      pre-select(n,k)  ≜ k ∈ collect-keys(n)
      post-select(n,k,r) ≜ r ∈ n and k ∈ collect-keys(r)


INSERT1
States: B-tree
Type:   Key  Data ⟶
pre-INSERT1(t,k,d)     ≜ k ∉ collect-keys(TREE(t))
post-INSERT1(t,k,d,t') ≜ ORDER(t') = ORDER(t) and
                         TREE(t') = insertr(ORDER(t),TREE(t),k,d)

where insertr: Nat  Node  Key  Data ⟶ Node
      pre-insertr(m,n,k,d) ≜ k ∉ collect-keys(n)
      insertr(m,n,k,d)     ≜ let rn = insertn(m,n,k,d)
                             if size(rn) = 1
                               then element(rn)
                               else rn

where insertn: Nat  Node  Key  Data ⟶ Inode
      pre-insertn(m,n,k,d) ≜ k ∉ collect-keys(n)
      insertn(m,n,k,d)     ≜ cases of n
                             n ∈ Inode: let cn = selectc(n,{k})
                                        let cns = insertn(m,cn,k,d)
                                        let rn = n - {cn} ∪ cns
                                        if size(rn) ≤ 2*m+1
                                          then {rn}
                                          else split(rn)
                             n ∈ Tnode: let rn = n + [k⟶d]
                                        if size(rn) ≤ 2*m
                                          then {rn}
                                          else split(rn)
                             end

where element: Node-set ⟶ Node
      pre-element(ns)  ≜ card ns = 1
      post-element(ns,r) ≜ r ∈ ns

where selectc: Inode  Key-set ⟶ Node
      post-selectc(n,ks,r) ≜ (∃ ins,hns ⊂ n)(is-division(n,ins,hns,{r}) and
                                             is-ordered(ins,collect-keys(r) ∪ ks,hns))
```

```
where is-division: Inode  Node-set  Node-set  Node-set ⟶ Bool
      is-division (n,ins,hns,mns) ≙ is-pdisj(<ins,hns,mns>) and
                                    (ins ∪ mns ∪ hns = n)          (cf. section 3.2)

where is-ordered: Node-set  Key-set  Node-set ⟶ Bool
      is-ordered(ins,ka,hns) ≙ (collect-keys(ins) << ka) and
                               ka << collect-keys(hns))

where split1: Inode ⟶ Inode-set
      pre-split1(n)     ≙ size(n) ≥ 2 and keysets-are-ordered(n)
      post-split1(n,ns) ≙ union ns = n and card ns = 2 and
                          keysets-are-ordered(ns) and let {n1,n2} = ns in
                          (size(n1) = size(n) div 2 and size(n2) = size(n) -
                           size(n1))

where split1t: Tnode ⟶ Tnode-set
      pre-split1t(n)    ≙ size(n) ≥ 2
      post-split1t(n,ns) ≙ munion ns = n and card ns = 2 and
                           keysets-are-ordered(ns) and let {n1,n2} = ns in
                           (size(n1) = size(n) div 2 and size(n2) = size(n) -
                            size(n1))


DELETE1
States: B-tree
Type:   Key ⟶
pre-DELETE1(t,k)     ≙ k ∈ collect-keys(TREE(t))
post-DELETE1(t,k,t') ≙ ORDER(t') = ORDER(t) and
                       TREE(t') = deleten(ORDER(t),TREE(t),k)

where deleter: Nat  Node  Key ⟶ Node
      pre-deleter(m,n,k) ≙ k ∈ collect-keys(n)
      deleter(m,n,k)     ≙ let rn = deleten(m,n,k)
                           if rn ∈ Inode and size(rn) = 1
                             then element(rn)
                             else rn

where deleten: Nat  Node  Key ⟶ Node
      pre-deleten(m,n,k) ≙ k ∈ collect-keys(n)
      deleten(m,n,k)     ≙ cases of n
                           n ∈ Inode: let cn = select(n,k)
                                      let rn = deleten(m,cn,k)
                                      if size(rn) ≥ minimum-size(m,rn)
                                        then n - {cn} ∪ {rn}
                                        else
                                        let nn = neighbour(n,cn)
                                        if size(rn) + size(nn) ≥
                                            2*minimum-size(m,rn)
                                          then
                                            n - {cn,rn} ∪ redistribute(rn,nn)
                                          else n - {cn,nn} ∪ merge(rn,nn)
                           n ∈ Tnode: n\{k}
                           end

where minimum-size: Nat  Node ⟶ Nat
      minimum-size(m,n) ≙ cases of n
                          n ∈ Inode: m+1
                          n ∈ Tnode: m
                          end
```

```
where neighbour: Inode  Node ⟶ Node
      pre-neighbour(n,sn)      ≜ sn ∈ n and card n ≥ 2
      post-neighbour(n,sn,rn) ≜ (∃ lns,hns ⊂ n)
                                   (is-division(n,lns,hns,{rn,sn}) and
                                    is-ordered(lns,collect-keys({rn,sn}),hns))

where redistribute: Node  Node ⟶ Inode
      pre-redistribute(n1,n2) ≜ ((n1 ∈ Inode and n2 ∈ Inode) or
                                  (n1 ∈ Tnode and n2 ∈ Tnode)) and
                                   size(n1) + size(n2) ≥ 2 and
                                   keysets-are-ordered({n1,n2}) and
                                 keysets-are-ordered(n1) and keysets-are-ordered(n2)
      redistribute(n1,n2) ≜ cases of n1,n2
                             n1,n2 ∈ Inode: split1 ∘ element ∘ merge(n1,n2)
                             n1,n2 ∈ Tnode: split ∘ element ∘ merge(n1,n2)
                             end
```

NOTE 4.2 This might have been better defined in terms of a post-condition for the purpose of writing the corresponding program code.

```
where merge: Node  Node ⟶ Inode
      pre-merge(n1,n2) ≜ (n1 ∈ Inode and n2 ∈ Inode) or
                         (n1 ∈ Tnode and n2 ∈ Tnode)
      merge(n1,n2)     ≜ cases of n1,n2
                         n1,n2 ∈ Inode: {n1 ∪ n2}
                         n1,n2 ∈ Tnode: {n1 + n2}
                         end
```

4.3.4 Proofs of the Correctness of Representation 1

4.3.4.1 Data Type Proofs

(B) Totality of the Retrieve Function

The rule to be proved is:

(1) (∀ t ∈ B-tree)(inv(t) ⟹ (∃ kdm ∈ (Key ⟶ Data))(kdm = retr(t)))

On substituting definitions, this may be written as:

(2) (∀ t ∈ B-tree)(invr(ORDER(t),TREE(t)) ⟹
(∃ kdm ∈ (Key ⟶ Data))(kdm = retrn(TREE(t))))

Proof:
Case 1: TREE(t) ∈ Tnode
In this case, (2) becomes
(3) size(TREE(t)) ≤ 2*ORDER(t) ⟹ (∃ kdm ∈ (Key ⟶ Data))(kdm = TREE(t))
which can immediately be seen to be true.

Case 2: TREE(t) ∈ Inode
In this case the proof is by structural induction on TREE(t)
BASIS (∀ sn ∈ TREE(t))(sn ∈ Tnode)
Equation (2) becomes
(4) keysets-are-ordered(TREE(t)) *and* balanced(TREE(t)) *and*
2 ≤ size(TREE(t)) ≤ 2*ORDER(t)+1 *and*
(∀ sn ∈ TREE(t))(ORDER(t) ≤ size(sn) ≤ 2*ORDER(t) ⟹
(∃ kdm ∈ (Key ⟶ Data))(kdm = *munion* {sn | sn ∈ TREE(t)}))
which can be seen to be true because keysets-are-ordered guarantees that the domains of the mappings to be united are disjoint.

Now suppose
(∀ sn ∈ TREE(t))(sn ∈ Inode)
INDUCTIVE HYPOTHESIS
(5) (∀ sn ∈ TREE(t))(common-inv(sn) *and* size-inv(sn) ⟹
(∃ kdm ∈ (Key ⟶ Data))(kdm = retrn(sn)))

In this case (2) becomes
(6) keysets-are-ordered(TREE(t)) *and* balanced(TREE(t)) *and*
(∀ sn ∈ TREE(t))(common-inv(sn) *and* size-inv(sn)) *and*
2 ≤ size(TREE(t)) ≤ 2*ORDER(t)+1 ⟹
(∃ kdm ∈ (Key ⟶ Data))(kdm = *munion* {retrn(sn) | sn ∈ TREE(t)})
which follows from the Inductive Hypothesis, because balanced ensures that there are no nodes with mixed Inodes and Tnodes as subnodes.

NOTE 4.3. The above proof could not be performed simply by induction on Node because of the special case of a tree consisting only of a single Tnode.

(C) Adequacy

The rule to be proved is:

(1) (∀ kdm ∈ (Key ⟶ Date))(∃ t ∈ B-tree)(*inv*(t) *and* kdm = *retr*(t)))

On substituting definitions, this may be written as

(2) (∀ kdm ∈ (Key ⟶ Date))(∃ t ∈ B-tree)(*invr*(ORDER(t),TREE(t)) *and*
kdm = *retrn*(TREE(t))))

Proof: This is shown by an informal argument.
Clearly, any mapping,
kdm such that *card dom* kdm ≤ 2*ORDER(t)
can be represented by
TREE(t) = kdm
and this satisfies
invr(ORDER(t),TREE(t)) *and* kdm = *retrn*(TREE(t))

For a mapping, kdm, such that
card dom kdm > 2*ORDER(t)
it is possible to order the domain of the mapping and then to partition kdm into a set of disjoint mappings, each of which has a domain of size between ORDER(t) and 2*ORDER(t), and which form the terminal nodes of the tree. The resulting set will have cardinality ≥ 2 and satisfies common-inv and size-invr (as ORDER(t) ≥ 1). Should the set have cardinality > 2*ORDER(t)+1, it can be partitioned in such a way that each resulting set has cardinality between ORDER(t)+1 and 2*ORDER(t)+1 and the partition has cardinality ≥ 2 and satisfies common-inv and size-*invr*. This partitioning process can continue until a set results which has cardinality ≤ 2*ORDER(t)+1 and what has resulted is TREE(t), such that
invr(ORDER(t), TREE(t)) *and* kdm = *retrn*(TREE(t)).

Adequacy can be proved formally by induction on *dom* kdm, since the proof that INSERT1 is a total operation implies that the representation of any kdm can be generated using INSERT1.

NOTE 4.4 The special case of the root size invariant is precisely required to permit the representation of small mappings. This special case will necessitate the separation of properties several times in the proofs that follow.

4.3.4.2 Operation Proofs

Operation FIND1

(A) Preservation of the invariant

As FIND1 is an identity operation on the B-tree, it follows immediately that the invariant is preserved.

Proof that FIND1 models FIND

(D) Domains Rule

What must be shown is:

(1) (∀ t ∈ B-tree)(inv(t) *and* pre-FIND(retrn(t),k) ⟹ pre-FIND1(t,k))

On substituting definitions, this becomes

(2) (∀ t ∈ B-tree)(inv(t) *and* k ∈ *dom* retrn(TREE(t)) ⟹
k ∈ collect-keys(TREE(t)))

Proof:
The first step is to prove that
(3) (∀ m ∈ Nat, n ∈ Node)(common-inv(n) *and* k ∈ *dom* retrn(n) ⟹
k ∈ collect-keys(n))
This is proved by structural induction on n
BASIS
(4) n ∈ Tnode (2) becomes
(5) common-inv(n) *and* k ∈ *dom* n ⟹ k ∈ *dom* n
which follows immediately.

INDUCTIVE HYPOTHESIS n ∈ Inode
assume
(6) (∀ sn ∈ n)(common-inv(sn) *and* k ∈ *dom* retrn(sn) ⟹
k ∈ collect-keys(sn))

(7) common-inv(n) *and* k ∈ *dom* retrn(n) (Hypothesis)
(8) (∀ sn ∈ n)(common-inv(sn)) and (common-inv)
(9) (∃ sn ∈ n)(k ∈ *dom* retrn(sn)) (retrn)
(10) k ∈ collect-keys(sn) for such an sn (by (6))
Thus
(11) k ∈ collect-keys(n) (collect-keys)
from which
(∀ t ∈ B-tree)(inv(t) *and* k ∈ *dom* retrn(TREE(t)) ⟹
k ∈ collect-keys(TREE(t)))
follows at once.

What must be shown is:

(1) (∀ t ∈ B-tree)(inv(t) *and* pre-FIND1(t,k) *and* post-FIND1(t,k,t',d) ⟹
post-FIND(retr(t),k,retr(t'),d))

On substituting definitions, this becomes

(2) (∀ t ∈ B-tree)(inv(t) *and* k ∈ collect-keys(TREE(t)) *and* t'=t *and*
d=find(k,TREE(t)) ⟹ retr(t')=retr(t) *and* d=retrn(TREE(t))(k))

It follows immediately from
t'=t that retr(t')=retr(t),

so what must be shown is

(3) (∀ t ∈ B-tree)(inv(t) *and* k ∈ collect-keys(TREE(t)) ⟹
find(k,TREE(t)) = retrn(TREE(t))(k))

Proof:
The first step is to prove that
(3) (∀ m ∈ Nat, n ∈ Node)(common-inv(n) *and* k ∈ collect-keys(n) ⟹
find(k,n) = retrn(n)(k))

This is proved by structural induction on n
BASIS
(4) n ∈ Tnode (3) becomes
(5) common-inv(n) *and* k ∈ *dom* n ⟹ n(k) = n(k)
which follows immediately.

INDUCTIVE HYPOTHESIS n ∈ Inode
assume
(6) (∀ sn ∈ n)(common-inv(sn) *and* k ∈ collect-keys(sn) ⟹
find(k,sn) = retrn(sn)(k))

(7) common-inv(n) *and* k ∈ collect-keys(n) (Hypothesis)
(8) (∀ sn ∈ n)(common-inv(sn)) and (common-inv)
(9) (∃ sn ∈ n)(k ∈ collect-keys(sn)) (by (7) and collect-keys)
(10) find(k,sn) = retrn(sn)(k) for such a sn (by (6))
Thus
(11) find(k,n) = retrn(n)(k) (definitions of find and retrn)
from which
(∀ t ∈ B-tree)(inv(t) *and* k ∈ collect-keys(TREE(t)) ⟹
find(k,TREE(t)) = retrn(TREE(t))(k))
follows at once.

Operation INSERT1

A function which is of use in the following proofs is:

not-in: Key Node → Bool
not-in(k,n) ≙ k ∉ collect-keys(n)

(A) Preservation of the Invariant

The rule to be proved is

(1) pre-INSERT1(t,k) *and* inv(t) *and* post-INSERT1(t,k,t') ⟹ inv(t')

This may be expanded to

(2) not-in(k,TREE(t)) *and* invr(ORDER(t),TREE(t)) ⟹
invr(ORDER(t),insertr(ORDER(t),TREE(t),k,d))

The proof of (2) has been done by decomposing what has to be proved into a number of lemmas and proving those which are not immediately apparent. These lemmas and their proofs then combine to provide the proof that INSERT1 preserves the invariant.

The lemmas for insertion are listed below, and are followed by the necessary proofs from which the truth of Theorem 1.12 and thus equation (2) follows:

Insertion Lemmas for Tnodes

Assuming
m ∈ Nat *and* n ∈ Tnode *and* not-in(k,n) *and* n'=insertn(m,n,k,d)

for the following 4 lemmas:

L1.1 size-inv(m,n) ⟹ n' ∈ Tnode-set *and* 1 ≤ size(n') ≤ 2

L1.2 keysets-are-ordered(n') *and* collect-keys(n') = collect-keys(n) ∪ {k}

L1.3 balanced(n')

L1.4 size-inv(m,n) ⟹ (∀ sn' ∈ n')(size-inv(m,sn'))

Assuming
m ∈ Nat *and* n ∈ Tnode *and* not-in(k,n) *and* size-invr(m,n) *and*
n'=insertr(m,n,k,d)

L1.5 size-invr(m,n)

Insertion Lemmas for general nodes

Assuming
m ∈ Nat *and* n ∈ Node *and* common-inv(n) *and* not-in(k,n) *and*
n'=insertn(m,n,k,d)

for the next 4 lemmas

L1.6 size-inv(m,n) ⟹ n' ∈ Inode *and* 1 ≤ size(n') ≤ 2
(from L1.1, split and insertn)

L1.7 keysets-are-ordered(n') *and* collect-keys(n') = collect-keys(n) ∪ {k}

L1.8 balanced(n') *and* (∀ sn' ∈ n')(depths(sn') = depths(n))

L1.9 common-inv(n')

Assuming
m ∈ Nat *and* n ∈ Node *and* size-inv(m,n) *and* not-in(k,n) *and*
n'=insertn(m,n,k,d)

L1.10 (∀ sn' ∈ n')(size-inv(m,sn'))

Assuming
m ∈ Nat *and* n ∈ Node *and* size-invr(m,n) *and* not-in(k,n) *and*
n'=insertr(m,n,k,d)

L1.11 size-invr(m,n')

Theorem 1.12 invr(m,n) *and* not-in(k,n) *and* n'=insertr(m,n,k,d) ⟹
invr(m,n')
(from L1.9 and L1.11)

Proof of Lemma 1.7

To prove:

(1) n ∈ Node *and* common-inv(n) *and* not-in(k,n) *and* n'=insertn(m,n,k,d) ⟹
keysets-are-ordered(n') *and* collect-keys(n')=collect-keys(n) ∪ {k}

Proof: By structural induction on n

BASIS

(2) n ∈ Tnode

which follows immediately from L1.2.

INDUCTIVE HYPOTHESIS n ∈ Inode

assume

(3) (∀ sn ∈ n)(common-inv(sn) *and* not-in(k,sn) *and* sn'=insertn(m,sn,k,d)
⟹ keysets-are-ordered(sn') *and* collect-keys(sn')=collect-keys(sn) ∪ {k})

(4) common-inv(n) *and* not-in(k,n) (Hypothesis)

(5) *let* cn = selectc(n,{k})

(6) cn ∈ n (selectc)

(7) common-inv(cn) ((3), (4), (6) *and* common-inv)

(8) not-in(k,cn) ((4), (6), collect-keys)

(9) *let* cns = insertn(m,cn,k,d)

(10) keysets-are-ordered(cns) *and* collect-keys(cns)=collect-keys(cn) ∪ {k}
(by (7), (8), (9), (3))

(11) *let* rn = n - {cn} ∪ cns

(12) keysets-are-ordered(rn) (by (10), (11), selectc *and* (4))

(13) collect-keys(rn) = collect-keys(n) ∪ {k} (by (10) and (11))

(14) n' = *if* size(rn) ≤ 2**m *then* {rn} *else* split(rn)

The lemma holds in both cases, from the definitions of split, collect-keys and keysets-are-ordered.

Proof of Lemma 1.8

This is an important property of a B-tree, which is not obvious.

To prove:

(1) $n \in$ Node *and* common-inv(n) *and* not-in(k,n) *and* n'=insertn(m,n,k,d) $\Longrightarrow$
balanced(n') *and* ($\forall$ sn' $\in$ n')(depths(sn') = depths(n))

Proof: By structural induction on n

BASIS

(2) $n \in$ Tnode

which follows immediately from L1.3 and L1.1

INDUCTIVE HYPOTHESIS n $\in$ Inode

assume

(3) ($\forall$ sn $\in$ n)(common-inv(sn) *and* not-in(k,sn) *and* sn'=insertn(m,sn,k,d)
$\Longrightarrow$ balanced(sn') *and* ($\forall$ ssn' $\in$ sn')(depths(ssn') = depths(sn')))

(4) common-inv(n) *and* not-*in*(k,n) (Hypothesis)

(5) ($\forall$ sn $\in$ n)(common-inv(sn) *and* not-in(k,sn)) ((4), common-inv *and* not-in)

(6) *let* cn = selectc(n,[k])

(7) cn $\in$ n (selectc)

(8) common-inv(cn) *and* not-in(k,cn) ((5) and (7))

(9) balanced(cn) (by (8))

(10) ($\forall$ sn $\in$ n)(depths(sn)=depths(cn))
(n $\in$ Inode *and* balanced(n) $\Longrightarrow$($\forall$ s1,s2 $\in$ n)(depths(s1)=depths(s2))

(11) depths(n) = *union* {depths(sn) | sn $\in$ n} ++ 1 = depths(cn) ++ 1

(12) *let* cns = insertn(m,cn,k,d)

(13) balanced(cns) *and* ($\forall$sn' $\in$ cns)(depths(sn') = depths(cn)) (by (3))

(14) *let* rn = n - {cn} $\cup$ cns

(15) ($\forall$ srn $\in$ rn)(depths(srn) = depths(cn)) (by (10) and (13))

(16) depths(rn) = depths(*n*) (by (10), (11) and (15))

(17) balanced(rn) (*card* depths(rn)=*card* depths(n)=1 by (16))

(18) n' = *if* size(rn) $\leq 2^{**}$m *then* {rn} *else* splitl(rn)

The lemma holds in both cases, from (16), (17) and the definition of splitl.

Proof of Lemma 1.9

To prove:

(1) n ∈ Node *and* common-inv(n) *and* not-in(k,n) *and* n'=insertn(m,n,k,d) ⟹ common-inv(n')

Proof: By structural induction on n

BASIS

(2) n ∈ Tnode

(3) keysets-are-ordered(n') (by L1.2)

(4) balanced(n') (by L1.3)

(5) n' ∈ Tnode-set (by L1.1)

(6) (∀ sn' ∈ n')(common-inv(sn'))

(7) common-inv(n') (by (3), (4), (6))

INDUCTIVE HYPOTHESIS n ∈ Inode

assume

(8) (∀ sn ∈ n)(common-inv(sn) *and* not-in(k,sn) *and* sn'=insertn(m,sn,k,d) ⟹ common-inv(sn'))

(9) common-inv(n) *and* not-in(k,n) (Hypothesis)

(10) keysets-are-ordered(n') (by L1.7)

(11) balanced(n') (by L1.8)

(12) *let* cn = selectc(n,{k})

(13) cn ∈ n (selectc)

(14) common-inv(cn) *and* not-in(k,cn) ((9), (13), common-inv and not-in))

(15) *let* cns = insertn(m,cn,k,d)

(16) common-inv(cns) (by (14), (15), (8))

(17) *let* n' = n - {cn} ∪ cns

(18) (∀ sn' ∈ n')(common-inv(sn')) (by (9), (13), (16) and common-inv)

(19) common-inv(n') (by (10), (11) and (16))

NOTE 4.5. Induction is required for this proof because common-inv is recursive.

Proof of Lemma 1.10

To prove:

(1) n ∈ Node *and* size-inv(m,n) *and* not-in(k,n) *and* n'=insertn(m,n,k,d) ⟹
(∀ sn' ∈ n')(size-inv(m,sn'))

Proof: By structural induction on n

BASIS

(2) n ∈ Tnode

which follows immediately from L1.4.

INDUCTIVE HYPOTHESIS n ∈ Inode

assume

(3) (∀ sn ∈ n)(size-inv(m,sn) *and* not-in(k,sn) *and* sn'=insertn(m,sn,k,d)
⟹ (∀ ssn' ∈ sn')(size-inv(m,ssn')))

(4) size-inv(m,n) *and* not-in(k,n) (Hypothesis)

(5) (∀ sn ∈ n)(size-inv(m,sn) *and* not-in(sn)) ((4), size-inv *and* not-in)

(6) *let* cn = selectc(n,{k})

(7) cn ∈ n (selectc)

(8) size-inv(m,cn) *and* not-in(k,cn) ((5) and (7))

(9) *let* cns = insertn(m,cn,k,d)

(10) (∀sn' ∈ cns)(size-inv(m,sn')) (by (3))

(11) 1 ≤ size(cns) ≤ 2 (by L1.5)

(12) *let* rn = n - {cn} ∪ cns

(13) size(n) ≤ size(rn) ≤ size(n) + 1

(14) n' = *if* size(rn) ≤ 2**m *then* {rn} *else* split(rn)

Case (a):

(15) size(rn) ≤ 2**m

(16) (∀ sn' ∈ n')(size-inv(m,sn'))

follows immediately from

size(n) ≤ size(rn) ≤ 2**m *and* size-inv(m,n)

Case (b):

(17) size(rn) > 2**m

(18) size-inv(m,n) *and* (13) ⟹ size(rn) = 2**m+1

(19) (∀ sn' ∈ n')(size-inv(m,sn'))

follows immediately from the definition of split.

Proof of Lemma 1.11

To prove:

(1) n ∈ Node *and* size-invr(m,n) *and* not-in(k,n) *and* n'=insertr(m,n,k,d)
⟹ size-invr(m,n')

Proof:

For n ∈ Inode

This case follows immediately from L1.5.

For n ∈ Inode

(2) size-invr(m,n) *and* not-in(k,n) (Hypothesis)
(3) (∀ sn ∈ n)(size-inv(m,sn) *and* not-in(k,sn)) ((2), size-invr and not-in)
(4) *let* cn = selectc(n,{k})
(5) cn ∈ n (selectc)
(6) size-inv(m,cn) *and* not-in(k,cn) ((3) and (5))
(7) *let* cns = insertn(m,cn,k,d)
(8) (∀sn' ∈ cns)(size-inv(m,sn')) (by L1.10)
(9) 1 ≤ size(cns) ≤ 2 (by L1.6)
(10) *let* rn = n - {cn} ∪ cns
(11) size(n) ≤ size(rn) ≤ size(n) + 1
(12) (∀ sn ∈ rn)(size-inv(m,srn)) (by (3), (5), (8) and (10))
(13) n' = *if* size(rn) ≤ 2*m *then* {rn} *else* split(rn)

Case (a):

size(rn) ≤ 2*m
n' = rn and size-invr(m,n') (by (11), (12) and size-invr)

Case (b):

size(rn) > 2*m
n' = split(rn) and size-invr(m,n') (by (12), split)

Proof that INSERT1 models INSERT

(D) Domains Rule

What must be shown is:

(1) (∀ t ∈ B-tree)(inv(t) *and* pre-INSERT(retr(t),k,d) ⟹
pre-INSERT1(t,k,d))

On substituting definitions, this becomes

(2) (∀ t ∈ B-tree)(inv(t) *and* k ∈ *dom* retrn(TREE(t)) ⟹
k ∈ collect-keys(TREE(t)))

Proof:

This proof is identical to the proof of the domains rule for operation FIND1, except that all occurrences of k ∈ . . . are replaced by k ∈ . . .

(E) Results Rule

The rule to be proved is:

e(1) (∀ t ∈ B-tree)(inv(t) *and* pre-INSERT1(t,k,d) *and* post-INSERT1(t,k,d,t') ⟹ post-INSERT(retr(t),k,d,retr(t'))

Expanding this gives

(2) (∀ t ∈ B-tree)(inv(t) *and* k∉ collect-keys(TREE(t)) *and* ORDER(t') = ORDER(t) ⟹
retrn(insertr(ORDER(t),TREE(t),k,d)) = retrn(TREE(t)) ∪ [k ⟶ d]}

Proof: By structural induction on TREE(t)

BASIS

(3) TREE(t) ∈ Tnode

Equation (2) becomes

(4) inv(t) *and* k ∈ collect-keys(TREE(t)) *and* ORDER(t') = ORDER(t) ⟹
retrn(ORDER(t),TREE(t),k,d) = TREE(t) + [k ⟶ d]

(5) *let* rn = TREE(t) + [k ⟶ d]

(6) insertn(ORDER(t),TREE(t),k,d) =
if size(rn) ≤ 2*ORDER(t) *then* {rn} *else* splitt(rn)

Case 1:

size(rn) ≤ 2*ORDER(t)

(7) insertr(ORDER(t),TREE(t),k,d) = rn = TREE(t) + [k ⟶ d]

(8) retrn(rn) = TREE(t) + [k ⟶ d]

Case 2:

size(rn) > 2*ORDER(t)

(9) insertr(ORDER(t),TREE(t),k,d) = splitt(rn) = splitt(TREE(t) + [k ⟶ d])

(10) retrn(splitt(rn)) = *munion* {retrn(sn) | sn ∈ splitt(TREE(t)+[k⟶d])}
= TREE(t) + [k ⟶ d]

Now if TREE(t) ∈ Inode

INDUCTIVE HYPOTHESIS Assume

(11) (∀ sn ∈ TREE(t))(inv(t) *and* k ∉ collect-keys(sn) ⟹
(retrn(insertn(ORDER(t),sn,k,d)) = retrn(sn) + [k ⟶ d]))

(12) *let* cn = selectc(TREE(t),{k})

(13) cn ∈ TREE(t) (selectc)

(14) *let* cns = insertn(ORDER(t),cn,k,d)

(15) retrn(cns) = retrn(cn) + [k ⟶ d] (by (11))

(16) *let* rn = TREE(t) - {cn} ∪ cns

(17) insertn(ORDER(t),TREE(t),k,d) =
if size(rn) ≤ 2*ORDER(t)+1 *then* {rn} *else* split(rn)

Case (a):

size(rn) ≤ 2*ORDER(t)+1

(18) insertr(ORDER(t),TREE(t),k,d) = rn = TREE(t) - {cn} ∪ cns

(19) retrn(rn) = retrn(TREE(t)) - retrn(cn) ∪ retrn(cn)+[k⟶d]
=retrn(TREE(t))+[k⟶d] (cn ∈ TREE(t))

Case (b):

size(rn) > 2*ORDER(t)+1

(20) insertr(ORDER(t),TREE(t),k,d) = split(rn) = split(TREE(t) - {cn} ∪ cns)

(21) retrn(split(rn)) = *munion* {retrn(sn) | sn ∈ split(TREE(t)-{cn} ∪ cns)}
= retrn(TREE(t)) - retrn(cn) ∪ retrn(cn) + [k ⟶ d]
= retrn(TREE(t))+[k⟶d] (cn ∈ TREE(t))

which completes the proof.

Operation DELETE1

A function which is of use in the following refinement proofs is:

is-in: Key Node → Bool
is(k,n) ≙ k ∈ collect-keys(n)

(A) Preservation of the invariant

The rule to be proved is:

(1) pre-DELETE1(t,k) *and* inv(t) *and* post-DELETE1(t,k,t') ⟹ inv(t')

This may be expanded to:

(2) is-in(k,TREE(t)) *and* invr(ORDER(t),TREE(t)) ⟹
invr(ORDER(t),deleter(ORDER(t),TREE(t),k))

Once again, the proof has been done by drawing up a list of Lemmas for Deletion (and proving those which are not obviously true), from which the truth of (2) can be deduced. These lemmas correspond to the Lemmas for Insertion in form.

The lemmas are listed below and are followed by selected proofs of those lemmas which are not immediately apparent and do not follow the lines of the proof of the corresponding lemma for insertion.

Deletion Lemmas for Tnodes

Assuming
m ∈ Nat *and* n ∈ Tnode *and* is-in(k,n) *and* n'=deleten(m,n,k)

for the following 3 lemmas:

L1.13 size-inv(m,n) ⟹ n' ∈ Tnode *and* size(n') = size(n)- 1

L1.14 keysets-are-ordered(n') *and* collect-keys(n') = collect-keys(n) - {k}

L1.15 balanced(n')

Assuming
m ∈ Nat *and* n ∈ Tnode *and* is-in(k,n) *and* size-invr(m,n) *and*
n' = deleter(m,n,k)

L1.16 size-invr(m,n')

Deletion Lemmas for general nodes and Inodes

Assuming
m ∈ Nat *and* common-inv(n) *and* is-in(k,n) *and* n' = deleten(m,n,k)

for the following 4 lemmas

L1.17 n ∈ Inode *and* size-inv(m,n) ⟹ n' ∈Inode *and* size(n') ≤ size(n)
(from merge, redistribute and delete)

also assuming that
n ∈ Node
for the next 3 lemmas:

L1.18 keysets-are-ordered(n') *and* collect-keys(n') = collect-keys(n) - {k}

L1.19 balanced(n') *and* depths(n') = depths(n)

L1.20 common-inv(n')
(proof of this follows the proof of L1.9, and is done by induction using Lemmas 1.13, 1.14, 1.15, 1.18 and 1.19)

Assuming
m ∈ Nat *and* n ∈ Inode *and* size-inv(m,n) *and* is-in(k,n) *and*
n' = deleten(m,n,k)

L1.21 (∀ sn' ∈ n')(size-inv(m,sn'))

Assuming
m ∈ Nat *and* n ∈ Inode *and* size-invr(m,n) *and* is-in(k,n) *and*
n' = deleter(m,r,n)

L1.22 size-invr(m,n') (from L1.16 and L1.17)

Theorem 1.23: invr(m,n) *and* is-in(k,n) *and* n' = deleter(m,n,k) ⟹ invr(n')
(from L1.20 and L1.22)

Proof of Lemma 1.18

To prove:

(1) n ∈ Node *and* common-inv(n) *and* is-in(k,n) *and* n'=deleten(m,n,k) ⟹
keysets-are-ordered(n') *and* collect-keys(n') = collect-keys(n) - {k}

Proof: By structural induction on n

BASIS

(2) n ∈ Tnode

which follows immediately from L1.14.

INDUCTIVE HYPOTHESIS n ∈ Inode

assume

(3) (∀ sn ∈ n)(common-inv(sn) *and* is-in(k,sn) *and* sn'=deleten(m,sn,k)
⟹ keysets-are-ordered(sn') *and* collect-keys(sn')=collect-keys(sn)-{k})

(4) common-inv(n) *and* is-in(k,n) (Hypothesis)

(5) (∀ sn ∈ n)(common-inv(sn)) *and* ((4) and common-inv)

(6) (∃ sn ∈ n)(is-in(sn)) ((4) and is-in)

(7) *let* cn = select(k,n)

(8) cn ∈ n and is-in(k,cn) (select)

(9) common-inv(cn) ((5) and (8))

(10) *let* rn = deleten(m,cn,k)

(11) keysets-are-ordered(rn) *and* collect-keys(rn)=collect-keys(cn)-{k}
(by (8), (9) and (3))

Case 1:

size(rn) ≥ minimum-size(m,rn)

(12) n' = n - {cn} ∪ {rn}

(13) keysets-are-ordered(n') *and* collect-keys(n')=collect-keys(n)-{k} (by (11))

Case 2:

size(rn) < minimum-size(m,rn)

(14) *let* nn = neighbour(n,cn) (defined because (m+1) ≥ 2)

(15) nn ∈ n (neighbour)

Case 2(a)

size(rn) + size(nn) ≥ 2*m

(16) collect-keys(redistribute(rn,nn))=collect-keys(rn) ∪ collect-keys(nn)

(17) keysets-are-ordered(redistribute(rn,nn)) (by definition of redistribute)

(20) n' = n - {cn,nn} ∪ redistribute(rn,nn)

(21) keysets-are-ordered(n') *and* (by (4), (11), (15), (16) and (17))

(22) collect-keys(n')=collect-keys(n) - {k} (by (11), (15), (20), (16) and collect-keys)

Case 2(b)

size(rn) + size(nn) < 2*m

(23) collect-keys(merge(rn,nn)) = collect-keys(rn) ∪ collect-keys(nn)

(24) n' = n - {cn,nn} ∪ merge(rn,nn)

(25) keysets-are-ordered(n') *and*
(by (4), (11), (23), merge, keysets-are-ordered and neighbour)

(26) collect-keys(n') = collect-keys(n) - {k}
(by (11), (23), (24) and collect-keys)

Proof of Lemma 1.19

To prove:

(1) $n \in$ Node *and* common-inv(n) *and* is-in(k,n) *and* n'=deleten(m,n,k) $\Longrightarrow$
balanced(n') *and* depths(n') = depths(n)

Proof: By structural induction on n

BASIS

(2) $n \in$ Inode

which follows immediately from L1.13 and L1.15.

INDUCTIVE HYPOTHESIS $n \in$ Inode

assume

(3) $(\forall sn \in n)$(common-inv(sn) *and* is-in(k,sn) *and* sn'=deleten(m,sn,k)
$\Longrightarrow$ balanced(sn') *and* depths(sn')=depths(sn))

(4) common-inv(n) *and* is-in(k,n) (Hypothesis)

(5) $(\forall sn \in n)$(common-inv(sn)) *and* ((4) and common-inv)

(6) $(\exists sn \in n)$(is-in(k,sn)) ((4) and is-in)

(7) *let* cn = select(k,n)

(8) cn $\in$ n and is-in(k,cn) (select)

(9) common-inv(cn) ((5) and (8))

(10) $(\forall sn \in n)$(depths(sn) = depths(cn))
$(n \in$ Inode *and* balanced(n) $\Longrightarrow (\forall s1,s2 \in n)$(depths(s1)=depths(s2)))

(11) depths(n) = *union* {depths(sn) | sn $\in$ n} $\mapsto$ 1
= depths(cn) $\mapsto$ 1

(12) *let* rn = deleten(m,cn,k)

(13) balanced(rn) *and* depths(rn) =depths(cn) (by (8), (9) and (3))

Case 1:

size(rn) $\geq$ minimum-size(m,rn)

(14) n' = n - {cn} $\cup$ {rn}

(15) $(\forall sn' \in n')$(depths(sn') = depths(cn)) (by (10) and (13))

(16) depths(n') = depths(n) (by (10), (11), (15))

(17) balanced(n') (*card* depths(n)=*card* depths(n')=1 by (4))

Case 2:

size(rn) < minimum-size(m,rn)

(18) *let* nn = neighbour(n,cn) (defined because $(m+1) \geq 2$)

(19) nn $\in$ n (neighbour)

(20) depths(nn) = depths(cn) = depths(rn) (by (10) and (13))

Case 2(a)

size(rn) + size(nn) $\geq 2*$minimum-size(m,rn)

(21) depths(redistribute(rn,nn)) = depths(rn) = depths(cn) (by 20)

(22) n' = n - {cn,nn} $\cup$ redistribute(rn,nn)

(23) $(\forall sn' \in n')$(depths(sn') = depths(cn)) (by (20) and (21))

(24) depths(n') = depths(n) (by (10) and (23))

(25) balanced(n') (by *card* depths(n)=1 = *card* depths(n'), (24) and (4))

Case 2(b)

size(rn) + size(nn) < $2*$minimum-size(m,rn)

This follows similar arguments to those of Case 2(a), and uses the property of merge:

depths(rn)=depths(nn) *and* balanced(rn) *and* balanced(nn) $\Longrightarrow$
balanced(merge(rn,nn))

Proof of Lemma 1.21

To prove:

(1) n ∈ Inode *and* size-inv(m,n) *and* is-in(k,n) *and* n'=deleten(m,n,k) ⟹
(∀ sn' ∈ n')(size-inv(m,sn'))

Proof: By structural induction on n

(2) size-inv(m,n) *and* is-in(k,n)) (Hypothesis)
(3) (∀ sn ∈ n)(size-inv(m,sn)) ((2) and size-inv)
(4) (∃ sn ∈ n)(is-in(k,n)) ((2) and collect-keys)

BASIS

(5) (∀ sn ∈ n)(sn ∈ Tnode)
(6) *let* cn = select(k,n)
(7) cn ∈ n and is-in(k,cn) (select)
(8) cn ∈ Tnode *and* size-inv(m,cn) ((2), (3) and (7))
(9) *let* rn = deleten(m,cn,k)
(10) rn = cn\{k} (deleten)
(11) size(rn) = size(cn) - 1 (by L1.13)

Case 1:

size(rn) ≥ minimum-size(m,rn)
(12) size-inv(m,rn) (size(rn) < size(cn) *and* size-inv(cn) by (8))
(13) n' = n - {cn} ∪ {rn}
(14) (∀ sn' ∈ n')(size-inv(m,sn')) (by (2), (7), (12))

Case 2:

size(rn) < minimum-size(m,rn)
(15) *let* nn = neighbour(n,cn) (defined because (m+1) ≥ 2)
(16) nn ∈ n *and* size-inv(m,nn) (neighbour and (3))

Case 2(a)

size(rn) + size(nn) ≥ 2*minimum-size(m,rn)
(17) n' = n - {cn,nn} ∪ *redistribute(rn,nn)*
Immediate by definition of redistribute and (8), (11) and (16).

Case 2(b)

size(rn) + size(nn) < 2*minimum-size(m,rn)
(18) n' = n - {cn,nn} ∪ *merge*(rn,nn)
Immediate by definition of merge and (8), (11) and (16).

(∀ sn ∈ n)(sn ∈ Inode)
INDUCTIVE HYPOTHESIS
assume
(19) (∀ sn ∈ n)(size-inv(m,sn) *and* is-in(k,sn) *and* sn'=deleten(m,sn,k)
⟹ (∀ ssn' ∈ sn')(size-inv(m,ssn')))

(20) *let* cn = select(k,n)
(21) cn ∈ n and is-in(k,cn) (select)
(22) (∀ sn ∈ cn)(size-inv(m,sn)) (by (3), (21) and size-inv)
(23) *let* rn = deleten(m,cn,k)
(24) (∀ sn ∈ rn)(size-inv(m,sn)) (by (21), (22), and (19))

Case 1:

size(rn) ≥ minimum-size(m,rn)
(25) n' = n - {cn} ∪ {rn}
(26) (∀ sn' ∈ n')(size-inv(m,sn')) (by (21), (22) and (24))

Case 2:

size(rn) < minimum-size(m,rn)

(27) let nn = neighbour(n,cn) (defined because (m+1) ≥ 2)

(28) nn ∈ n (neighbour)

(29) (∀ sn ∈ nn)(size-inv(m,sn)) (by (28) and (3))

Case 2(a)

size(n) + size(nn) ≥ 2*minimum-size(m,rn)

(30) n' = n - {cn,nn} ∪ redistribute(rn,nn)

Immediate from the definition of redistribute, (22), (24) and (29).

Case 2(b)

size(n) + size(nn) < 2*minimum-size(m,rn)

(31) n' = n - {cn,nn} ∪ merge(rn,nn)

Immediate from the definition of merge, (22), (24) and (29).

This concludes the proof of L1.21.

NOTE 4.6: As L1.21 is concerned with Inodes, the basis cannot simply be a Tnode.

The lemma states that after DELETE1 has operated on an Inode, the size invariant is preserved for all its subnodes.

Proof that DELETE1 models DELETE

(D) Domains Rule

What must be shown is

(1) (∀ t ∈ B-tree)(inv(t) *and* pre-DELETE(retr(t),k) ⟹ pre-DELETE1(t,k))

On substituting definitions, this becomes

(2) (∀ t ∈ B-tree)(inv(t) *and* k ∈ *dom* retrn(TREE(t)) ⟹
is-in(k,TREE(t)))

Proof:
This proof is identical to the proof of the domains rule for operation FIND1.

(E) Results Rule

The rule to be proved is

(1) (∀ t ∈ B-tree)(inv(t) *and* pre-DELETE1(t,k) *and* post-DELETE1(t,k,t')
⟹ post-DELETE(retr(t),k,retr(t')))

Expanding this gives

(2) (∀ t ∈ B-tree)(inv(t) *and* is-in(k,TREE(t)) *and* ORDER(t') = ORDER(t)
⟹ retrn(deleter(ORDER(t),TREE(t),k)) = retrn(TREE(t))\{k})

Proof: By structural induction on TREE(t)
BASIS
(3) TREE(t) ∈ Tnode
equation (2) becomes
(4) inv(t) *and* k ∈ *dom* TREE(t) *and* ORDER(t') = ORDER(t) ⟹
retrn(deleter(ORDER(t),TREE(t),k)) = TREE(t)\{k}
This follows immediately as
(5) deleter(ORDER(t),TREE(t),k) = deleten(ORDER(t),TREE(t),k) = TREE(t)\{k}
so
(6) retrn(deleter(ORDER(t),TREE(t),k)) = TREE(t)\{k}

If TREE(t) ∈ Inode
INDUCTIVE HYPOTHESIS Assume
(7) (∀ sn ∈ TREE(t))(inv(t) *and* is-in(k,sn) ⟹
retrn(deleten(ORDER(t),sn,k) = retrn(sn)\{k})

(8) *let* cn = select(TREE(t),k)
(9) cn ∈ TREE(t) *and* is-in(k,cn) (select)
(10) *let* rn = deleten(ORDER(t),cn,k)
(11) retrn(rn) = retrn(cn)\{k} (by (9) and (7))

Case 1:

size(rn) ≥ minimum-size(ORDER(t),rn)

(12) deleten(ORDER(t),TREE(t),k) = TREE(t) - {cn} ∪ {rn}

(13) size(deleten(ORDER(t),TREE(t),k)) = size(TREE(t)) (from (12))

(14) deleter(ORDER(t),TREE(t),k) = TREE(t) - {cn} ∪ {rn} (from(13))

(15) retrn(deleter(ORDER(t),TREE(t),k)) =

retrn(TREE(t)) - retrn(cn) ∪ retrn(cn)\{k} (by (9), (11) and (14))

= retrn(TREE(t))\{k}

Case 2:

size(rn) < minimum-size(ORDER(t),rn)

(16) *let* nn = neighbour(TREE(t),cn) (defined because (m+1) ≥ 2)

(17) nn ∈ TREE(t) (neighbour)

Case 2(a)

size(m) + size(nn) ≥ 2*minimum-size(ORDER(t),rn)

(18) deleten(ORDER(t),TREE(t),k) = TREE(t) - {cn,nn} ∪ redistribute(rn,nn)

(19) size(deleten(ORDER(t),TREE(t),k)) = size(TREE(t)) (by (9), (17), (18) and redistribute)

(20) deleter(ORDER(t),TREE(t),k) = TREE(t) - {cn,nn} ∪ redistribute(rn,nn)

(21) retrn(deleter(ORDER(t),TREE(t),k)) =

retrn(TREE(t)) - retrn(cn) - retrn(nn) ∪ retrn(redistribute(rn,nn))

= retrn(TREE(t)) - retrn(cn) - retrn(nn) ∪ retrn(rn) ∪ retrn(nn) (redistribute)

= retrn(TREE(t)) - retrn(cn) -retrn(cn)\{k} ∪ retrn(nn) (by (11))

= retrn(TREE(t))\{k}

Case 2(b)

size(rn) + size(nn) < 2*minimum-size(ORDER(t),rn)

(22) deleten(ORDER(t),TREE(t),k) = TREE(t) - {cn,nn} ∪ merge(rn,nn)

(23) size(deleten(ORDER(t),TREE(t),k)) = size(TREE(t)) - 1 (by (22) and merge)

if size(deleten(ORDER(t),TREE(t),k)) = 1

(24) *then* TREE(t) = {cn,nn}

(25) deleter(ORDER(t),TREE(t),k) = rn ∪ nn (merge)

(26) retrn(deleter(ORDER(t),TREE(t),k)) = retrn(rn) ∪ retrn(nn)

= retrn(cn)\{k} ∪ retrn(nn) (by (11))

= retrn(TREE(t))\{k} (by (24))

else

if size(deleten(ORDER(t),TREE(t),k)) > 1

(27) deleter(ORDER(t),TREE(t),k) = deleten(ORDER(t),TREE(t),k) (deleter)

= TREE(t) - {cn,nn} ∪ merge(rn,nn) (by (22))

which again gives (by a similar argument to that used in Case 2(a)):

retrn(deleter(ORDER(t),TREE(t),k)) = retrn(TREE(t))\{k}

4.4 Representation 2

This stage uses as a representation a tree structure which is defined by using lists, and the actual representation of the keys in the index part of the tree is specified. An Inode now consists of a list of keys, as well as a list of nodes, and the data type invariant states in what manner these lists must be ordered.

As the data type, BP-tree, is a more concrete form of the B-tree used in Representation 1, its structure closely resembles the structure of a B-tree and many of the functions correspond closely to the functions defined for Representation 1. This facilitates the refinement proofs for this level, as they have a similar form to the corresponding proofs which were done for Representation 1.

REPRESENTATION 2

4.4.1 Data Structure and Data Type Invariant

BP-tree :: ORDERP: Nat TREEP: Nodep

Nodep = Inodep | Tnodep
Inodep :: KEYL: Key-list TREEL: Nodep-list
Tnodep = Key $\xrightarrow{m}$ Data

invp: BP-tree $\rightarrow$ Bool
invp(t) $\triangleq$ invrp(ORDERP(t), TREEP(t))

where invrp: Nat Nodep $\rightarrow$ Bool
invrp(m,n) $\triangleq$ common-invp(n) *and* size-invrp(m,n)

where common-invp: Nodep $\rightarrow$ Bool
common-invp(n) $\triangleq$ *cases of* n
n $\in$ Inodep: keysets-are-orderedp(n) *and* balancedp(n)
and ($\forall$ i $\in$ [1 . *lenTREEL*(n)])
(common-invp(TREEL(n)(i)))
n $\in$ Tnodep: *true*
end

```
where size-invrp: Nat Nodep ⟶ Bool
      size-invrp(m,n) ≜ cases of n
                        n ∈ Inodep: 2 ≤ sizep(n) ≤ 2**m+1 and
                                    (∀ i ∈ {1 . . len TREEL(n)})
                                                (size-invp(m,TREEL(n)(i)))
                        n ∈ Tnodep: sizep(n) ≤ 2**m
                        end

where size-invp: Nat  Nodep ⟶ Bool
      size-invp(m,n) ≜ cases of n
                       n ∈ Inodep: m+1 ≤ sizep(n) ≤ 2**m+1 and
                                   len KEYL(n) = sizep(n) - 1 and
                                   (∀ i ∈ {1 . . len TREEL(n)})
                                   (size-invp(m,TREEL(n)(i)))
                       n ∈ Tnodep: m ≤ sizep(n) ≤ 2**m
                       end

where sizep: Nodep ⟶ Nat
      sizep(n) ≜  cases of n
                  n ∈ Inodep: len TREEL(n)
                  n ∈ Tnodep: card dom n
                  end

where keysets-are-orderedp: Nodep ⟶ Bool
      keysets-are-orderedp(n) ≜ cases of n
                                n ∈ Inodep: is-ordered(KEYL(n)) and
                                      (∀ i ∈ {1 . . len KEYL(n)})
                                      (collect-keysp(TREEL(n)(i)) <<=
                                      {KEYL(n)(i)} <<
                                      collect-keysp(TREEL(n)(i+1)))
                                n ∈ Tnodep: true
                                end

where is-ordered: Key-list ⟶ Bool
      is-ordered(kl) ≜ (∀ i ∈ {1 . . (len kl - 1)})(kl(i) < kl(i+1))

where collect-keysp: Nodep ⟶ Key-set
      collect-keysp(n) ≜ cases of n
                         n ∈ Inodep: union {collect-keysp(TREEL(n)(i)) |
                                                1 ≤ i ≤ len TREEL(n)}
                         n ∈ Tnodep: dom n
                         end

where <<=: Nat-set  Nat-set ⟶ Bool
      s1 <<= s2 ≜ (∀ e1 ∈ s1)(∀ e2 ∈ s2)(e1 ≤ e2)

where balancedp: Nodep ⟶ Bool
      balancedp(n) ≜ card depthsp(n) = 1

where depthsp: Nodep ⟶ Nat-set
      depthsp(n) ≜ cases of n
                   n ∈ Inodep: union {depthsp(TREEL(n)(i)) |
                                          1 ≤ i ≤ len TREEL(n)} ++ 1
                   n ∈ Tnodep: {1}
                   end
```

4.4.2 The Retrieve Function

```
retrp: BP-tree ⟶ B-tree
retrp(t) ≜ <ORDERP(t),retrnp(TREEP(t))>

where retrnFp: Nodep ⟶ Node
      ratrnp(n) ≜ cases of n
                  n ∈ Inodep: [retrnp(sn) | an ∈ elems TREEL(n)]
                  n ∈ Tnodep: n
                  end
```

4.4.3 The Operations

```
FINDP
States: BP-tree
Type:   Key ⟶ Data
pre-FINDP(t,k)      ≜ k ∈ collect-keysp(TREEP(t))
post-FINDP(t,k,t',r) ≜ t' = t and r = findp(k,TREEP(t))

where findp: Key  Nodep ⟶ Data
      pre-findp(k,n) ≜ k ∈ collect-keysp(n)
      findp(k,n)     ≜ cases of n
                       n ∈ Inodep: let i = index(k,KEYL(n))
                                   findp(k,TREEL(n)(i))
                       n ∈ Tnodep: n(k)
                       end

where index: Key  Key-list ⟶ Nat
      index(k,kl) ≜ if kl = < >
                    then 1
                    else if k ≤ hd kl
                         then 1
                         else 1 + index(k, tl kl)


INSERTP
States: BP-tree
Type:   Key  Data ⟶
pre-INSERTP(t,k,d)      ≜ k ∈ collect-keysp(TREEP(t))
post-INSERTP(t,k,d,t') ≜ ORDERP(t') = ORDERP(t) and
                         TREEP(t') = insertrp(ORDERP(t),TREEP(t),k,d)

where insertrp: Nat  Nodep  Key  Data ⟶ Nodep
      pre-insertrp(m,n,k,d) ≜ k ∈ collect-keysp(n)
      insertrp(m,n,k,d)     ≜ let rn = insertnp(m,n,k,d)
                              if sizep(rn) = 1
                              then TREEL(rn)(1)
                              else rn

where insertnp: Nat  Nodep  Key  Data ⟶ Inodep
      pre-insertnp(m,n,k,d) ≜ k ∈ collect-keysp(n)
      insertnp(m,n,k,d)     ≜ cases of n
                              n ∈ Inodep: let il,ik,ck,cn = selectp(n,k)
```

```
                                    let cnl = insertnp(m,cn,k,d)
                                    let sn =<< >,<cn>>
                                    let rn = replace(n,an,cnl,it,it)
                                    if sizep(rn) ≤ 2*m+1
                                       then << >,<rn>>
                                       else splitnp(m,rn)
                          n ∈ Tnodep: let rn = n + [k⟶d]
                                    if sizep(rn) ≤ 2*m
                                       then << >,<rn>>
                                       else splittp(m,rn)
                          end

where selectp: Inodep  Key ⟶ Nat  Nat  Key  Nodep
      post-selectp(n,k,rit,rik,rk,rn) ≜ rit = index(k,KEYL(n))
                                        if rit > len KEYL(n)
                                          then rik = rit -1
                                          else rik = rit
                                        rk = KEYL(n)(rik)
                                        rn = TREEL(n)(rit)

where replace: Inodep  Inodep  Inodep  Nat  Nat ⟶ Inodep
      pre-replace(i1,i2,i3,nk,nt) ≜ is-subnode(i1,i2) and
                                    ((KEYL(i2)=< > and nk=nt) or
                                    (nk=position(KEYL(i2)(1),KEYL(i1))))
                                    and nt=position(TREEL(i2)(1),TREEL(i1))
      replace(i1,i2,i3,nk,nt)     ≜ <alter(KEYL(i1),KEYL(i2),KEYL(i3),nk),
                                     alter(TREEL(i1),TREEL(i2),TREEL(i3),nt)>

where is-subnode: Inodep  Inodep ⟶ Bool
      is-subnode(i1,i2) ≜ if KEYL(i2) = < >
                          then len TREEL(i2)=1 and TREEL(i2)(1) ∈
                               elems TREEL(i1)
                          else is-sublist(KEYL(i1),KEYL(i2)) and
                               is-sublist(TREEL(i1,i2)) and
                               position(KEYL(i2)(1),KEYL(i1)) =
                               position(TREEL(i2)(1),TREEL(i1))

where is-sublist: X-list  X-list ⟶ Bool
      pre-is-sublist(l1,l2) ≜ 0 < len l2 ≤ len l1
      is-sublist(l1,l2)     ≜ (∃ i ∈ {1 . . len l1}(i=position(l2(1),l1)
                              and (∀ j ∈ {1 . . len l2}(l2(j)=l1(i+j-1))

where position: X  X-list ⟶ Nat
      pre-position(x,xl) ≜ x ∈ elems xl
      position(x,xl)     ≜ if x = hd xl
                           then 1
                           else 1+position(x, tl xl)

where alter: X-list  X-list  X-list  Nat ⟶ X-list
      pre-alter(l1,l2,l3,i) ≜ (l2=< > and 1 ≤ i ≤ len l1 + 1) or
                               (is-sublist(l1,l2) and i=position(l2(1),l1))
      alter(l1,l2,l3,i)     ≜ front(i-1,l1) || l3 || back(len l1-len l2-i+1,l1)
```

NOTE 4.7: The length of the preceeding five functions suggests that they could be restructured. An alternative approach would have been to return only a pair of indices to positions in the key and tree lists from selectp, and then to use only these indices as arguments in the function replace.

```
where front: Nat0  X-list ──→ X-list
      front(n,l) ≜ if n=0
                   then < >
                   else if n > len l
                        then l
                        else <hd l> ⌢ front(n-1,tl l)

where back: Nat0  X-list ──→ X-list
      post-back(n,l,rl) ≜ if  n = 0 or n > len l
                          then rl = < >
                          else len rl = n and
                               (∀ i ∈ {1 . . n})rl(i) = l(len l - n + i)


where splitip: Nat  Inodep ──→ Inodep
      pre-splitip(m,n) ≜ len KEYL(n) = 2*m+1 and len TREEL(n) = 2*m+2
      splitip(m,n)   ≜ <<KEYL(n)(m+1)>,<<front(m,KEYL(n)),front(m+1,TREEL(n))>,
                                        <back(m,KEYL(n)),back(m+1,TREEL(n))>>>


where splittp: Nat  Tnodep ──→ Inodep
      pre-splittp(m,n)      ≜ sizep(n) ≥ 2
      post-splittp(m,n,r)   ≜ let sk,gk = halve(dom n)
                              r = <<max(sk)>,<[k ──→ n(k) | k ∈ sk],
                                              [k ──→ n(k) | k ∈ gk]>>


where halve: Key-set ~> Key-set  Key-set
      pre-halve(ks,sks,gks) ≜ card ks ≥ 2
      post-halve(ks,sks,gks) ≜ (sks ∪ gks) = ks and (sks << gks) and
                               (card sks = card gks + 1)


where max: Nat-set ──→ Nat
      pre-max(s)   ≜ s ≠ { }
      post-max(s,r) ≜ r ∈ s and {r} >>= s


where >>=: Nat-set  Nat-set ──→ Bool
      s1 >>= s2 ≜ (∀ e1 ∈ s1)(∀ e2 ∈ s2)(e1 ≥ e2)


DELETEP
States: BP-tree
Type:   Key ──→
pre-DELETEP(t,k)     ≜ k ∈ collect-keysp(TREEP(t))
post-DELETEP(t,k,t') ≜ ORDERP(t') = ORDERP(t) and
                       TREEP(t') = deleterp(ORDERP(t),TREEP(t),k)


where deleterp: Nat  Nodep  Key ──→ Nodep
      pre-deleterp(m,n,k) ≜ k ∈ collect-keysp(n)
      deleterp(m,n,k)     ≜ let rn = deletenp(m,n,k)
                            if rn ∈ Inodep and sizep(rn) = 1
                              then TREEL(rn)(1)
                              else rn
```

```
where deletenp: Nat  Nodep  Key ⟶ Nodep
      pre-deletenp(m,n,k) ≜ k ∈ collect-keysp(n)
      deletenp(m,n,k)     ≜ cases of n
                              n ∈ Inodep: let it,ik,ck,cn = selectp(n,k)
                                          let rn = deletenp(m,cn,k)
                                          if sizep(rn) ≥ minimum-sizep(m,rn)
                                          then let sn = << >,<cn>>
                                               replace(n,sn,<< >,<rn>>,it,it)
                                          else
                                              let nn = neighbourp(n,it)
                                              let sn = <<ck>,<cn,nn>>
                                              if sizep(rn) + sizep(nn) ≥
                                                  2*minimum-sizep(m,rn)
                                               then
                                          replace(n,sn,redistributep(rn,nn,ck),ik,it)
                                               else
                                               replace(n,sn,mergep(rn,nn,ck),ik,it)
                              n ∈ Tnodep: n\{k}
                              end

where minimum-sizep: Nat  Nodep ⟶ Nat
      minimum-sizep(m,n) ≜ cases of n
                           n ∈ Inodep: m+1
                           n ∈ Tnodep: m
                           end

where neighbourp: Inodep  Nat  ⟶ Nodep
      pre-neighbourp(n,i)  ≜ i ≤ len TREEL(n)
      neighbourp(n,i)      ≜ if i = len TREE(n)
                             then TREEL(n)(i-1)
                             else TREEL(n)(i+1)

where redistributep: Nodep  Nodep  Key ⟶ Inodep
      pre-redistributep(n1,n2,k) ≜ (((n1 ∈ Inodep and n2 ∈ Inodep) and
                                    (len KEYL(n1) + len KEYL(n2) > 2*m-1) and
        (len TREEL(n1)=len KEYL(n1)+1 and len TREEL(n2)=len KEYL(n2)+1))
        or (n1 ∈ Tnodep and n2 ∈ Tnodep and len TREEL(n1)+len TREEL(n2)
        > 2)) and
        ((collect-keysp(n1) <<= {k} << collect-keysp(n2)) or
        (collect-keysp(n2) <<= {k} << collect-keysp(n1)))

      redistributep(n1,n2,k) ≜ cases of n1,n2
                               n1,n2 ∈ Inodep: splitip ∘ TREEL(mergep(n1,n2,k))(1)
                               n1,n2 ∈ Tnodep: splittp ∘ TREEL(mergep(n1,n2,k))(1)
                                 end

where mergep: Nodep  Nodep  Key ⟶ Inodep
      pre-mergep(n1,n2,k) ≜ (n1 ∈ Inodep and n2 ∈ Inodep) or
                            (n1 ∈ Tnodep and n2 ∈ Tnodep)
      mergep(n1,n2,k)     ≜ cases of n1,n2
                            n1,n2 ∈ Inodep:
          if collect-keysp(n1) << collect-keysp(n2)
          then << >,<KEYL(n1) || <k> || KEYL(n2),TREEL(n1) || TREEL(n2)>>
          else << >,<KEYL(n2) || <k> || KEYL(n1),TREEL(n2) || TREEL(n1)>>


                            n1,n2 ∈ Tnodep: << >,<n1 + n2>>
                            end
```

4.4.4 Proofs of the correctness of Represenlation 2

4.4.4.1 Data Type Proofs

(B) Totality of the retrieve function

The rule to be proved is:

(1) (∀ t2 ∈ BP-tree)(invp(t2) ⇒ (∃ t1 ∈ B-tree) (t1 = retrp(t2) *and* inv(retrp(t2)))

On substituting definitions, this becomes

(2) (∀ t2 ∈ BP-tree)(invrp(ORDERP(t2),TREEP(t2)) ⇒
(∃ t1 ∈ B-tree)(t1 = <ORDERP(t2),retrnp(TREEP(t2))>
and invr(ORDERP(t2),retrnp(TREEP(t2)))))

Proof:
When retmp is applied to an element of Tnodep, it is an identity function, and if it is applied to an element of Inodep, it relies only on the operator *elems* being applied to a list to be a total function. Since TREEL(n), where n ∈ Inodep, is always a list, and since the conditions of invr are weaker than those of invrp, it follows that the retrieve function retrp is total. This can be proved formally by structural induction on TREEP(t2).

(C) Adequacy

The rule to be proved is

(1) (∀ t1 ∈ B-tree)(invr(t1) ⇒ (∃ t2 ∈ BP-tree)(invp(t2) *and*
t1 = retrp(t2)))

On substituting definitions, this becomes

(2) (∀ t1 ∈ B-tree)(invr(ORDER(t1),TREE(t1)) ⇒
(∃ t2 ∈ BP-tree)
(invrp(OROERP(t2),TREEP(t2)) *and* t1 = <ORDERP(t2),retrnp(TREEP(t2))>))

Proof:
This is easiest shown by informal argument.

If t1 ∈ B-tree and TREE(t1) ∈ Tnode and invr(ORDER(t1),TREE(t1)) is true, then t1 can be represented by t2 =<ORDER(t1),TREE(t1)> and this clearly satisfies
invrp(ORDERP(t2),TREEP(t2)) *and* t1 =<ORDERP(t2),retrnp(TREEP(t2))>

For any t1 ∈ B-tree such that TREE(t1) ∈ Inode and which satisfies invr(ORDER(t1),TREE(t1)) a representation t2 can be produced as follows· ORDERP(t2) = ORDER(t1). Each n ∈ Tnode of t1 can be represented by the identical node in t2 and will satisfy
ORDERP(t2) ≤ sizep(n) ≤ 2*ORDERP(t2)

because this invariant is identical to ORDER(t1) ≤ size(n) ≤ 2*ORDER(t1).

For each $n1 \in$ Inode of t1, the elements of $n1$ can be laid out to form a list of nodes in ascending order of the key set that can be collected from each element (cf. keysets-are-orderedp) - this list forms TREEL(n2). The maximum key in each of the key sets of the elements of $n1$ can be determined and all but the largest laid out in ascending order to form KEYL(n2).

The fact that t1 satisfies invr(ORDER(t1),TREE(t1)) implies that each $n1 \in$ Inode will satisfy common-inv(n1) *and* size-inv(ORDER(t1),n1) and this ensures that the corresponding node $n2 \in$ Inodep obtained as described will satisfy common-invp(n2) *and* size-invp(ORDERP(t2),n2) (cf. keysets-are-orderedp). Thus the resulting $t2 \in$ BP-TREE will satisfy invrp(ORDERP(t2),TREEP(t2)) *and* t1 = <ORDERP(t2),retrnp(TREEP(t2))>.

4.4.4.2 Operation Proofs

Operation FINDP

(A) Preservation of the invariant

As FINDP is an identity operation on the BP-tree, it follows immediately that the invariant is preserved.

Proof that FINDP models FIND1

(D) Domains Rule

What must be shown is

(1) $(\forall t \in$ BP-tree)(invp(t) *and* pre-FINDP(retrp(t),k) $\Rightarrow$ pre-FINDP(t,k))

On substituting definitions this becomes

(2) $(\forall t \in$ BP-tree)(invp(t) *and* $k \in$ collect-keysp(retrnp(TREEP(t)))
$\Rightarrow k \in$ collect-keysp(TREEP(t)))

Proof:
It follows that (2) is true from the fact that the structure of collect-keys is the same as that of collect-keysp (the extra key list present in each $n \in$ Inodep is ignored). Equation (2) can be proved formally to be true by structural induction on TREEP(t).

(E) Results Rule

The rule to be proved is

(1) (∀ t ∈ BP-tree)(invp(t) *and* pre-FINDP(t,k) *and* post-FINDP(t,k,t',d)
⟹ post-FIND1(retrp(t),k,retrp(t'),d))

Expanding this gives

(2) (∀ t ∈ BP-tree)(invp(t) *and* k ∈ collect-keysp(TREE(t)) *and* t' = t
and d = findp(k,TREEP(t)) ⟹
retrp(t') = retrp(t) *and* d = find(k,retrnp(TREEP(t))))

It follows immediately from t' = t that

retrp(t') = retrp(t) so what must be shown is:

(3) (∀ t ∈ BP-tree)(invp(t) *and* k ∈ collect-keysp(TREEP(t)) ⟹
findp(k,TREEP(t)) = find(k,retrnp(TREEP(t))))

Proof:
Equation (3) can easily be shown to be true for elements of Tnodep. When
TREEP(t) ∈ Inodep, it may be written:
let n = TREEP(t)
and writing only the relevant part of the invariant:
keysets-are-orderedp(n) *and*
k ∈ *union* {collect-keysp(TREEL(n)(i)) | 1 ≤ i ≤ *len* TREEL(n)} ⟹

findp(k,TREEL(n)(index(k,KEYL(n)))) =
find(k,select({retrnp(sn) | sn ∈ *elems* TREEL(n)},k))

The invariant, keysets-are-orderedp, guarantees that the function index causes that element of TREEL(n) to be selected that corresponds to the "retrieved" element of TREEL(n) that is chosen by the function select. So (4) can be seen to be true by the definitions of the functions index and select.

The results rule can be proved formally by structural induction on TREEP(t) of a lemma:
(∀ n ∈ Tnodep)(invp(n) ⟹
let <kl,tl> = n *in* tl(index(k,kl)) = select(retrnp(n)(kl)))

Operation INSERTP

(A) Preservation of the invariant

The invariant of Representation 2 corresponds closely to that of Representation 1, the only real differences occurring in the definitions of size-invp and keysets-are-orderedp, and being introduced by the fact that an Inodep contains a list of keys which is ordered, and the ordering of the Nodep-list is related to this key list, as is its length, whereas an Inode contains only a Node-set.

The structure of INSERTP is the same as that of INSERT1, and for all the functions used by INSERT there are corresponding functions of similar structure which are used by INSERTP. The only additional function which is used by INSERTP is the function *replace*, and this can be related to Representation 1 as follows:

Lemma 224.

i1,i2,i3 ∈ Inodep and n1=retrnp(i1) and n2=retrnp(i2) and n3=retrnp(i3) and is-subnode(i1,i2) and within-bounds(i1,i2,collect-keysp(i3)) and keysets-are-orderedp(i1) and keysets-are-orderedp(i3) and ((KEYL(i2)=< > and nk = nt) or (nk = position(KEYL(i2)(1),KEYL(i1)))) and nt = position(TREEL(i2)(1),TREEL(i1)) ⟹

n2 ⊂ n1 and retrnp(replace(i1,i2,i3,nk,nt)) = n1 - n2 ∪ n3 and keysets-are-ordered(retrnp(replace(i1,i2,i3,nk,nt)))

where within-bounds: Inodep Inodep Key-set ⟶ Bool

```
pre-within-bounds(i1,i2,ks) ≜ is-subnode(i1,i2)
within-bounds(i1,i2,ks)     ≜ let j=position(TREEL(i2)(1),TREEL(i1))
                              if KEYL(i2) = < >
                                then let k = j
                                else
                                let
                                k=position(TREEL(i2)(len TREEL(i2)),
                                          TREEL(i1))
                                if j=1 and k=len TREEL(i1)
```

```
    then true
    else
    if j=1
      then ks << {KEYL(i1)(k)}
      else
      if k=len TREEL(i1)
        then ks >> {KEYL(i1)(j-1)}
        else
{KEYL(i1)(j-1)} << ks <<= {KEYL(i1)(k)}
```

Therefore, a list of Lemmas for Insertion for Representation 2 can be drawn up, which matches Lemmas 1.1 to 1.11 of Representation 1 but refers to elements of lists rather than of sets. The only Lemmas which differ in form from those of Representation 1 will be those corresponding to L1.1 and L1.6; i.e. L2.1 and L2.6. These will become:

L21

n ∈ Tnodep *and* not-inp(k,n) *and* size-invp(m,n) *and* n' = insertnp(m,n,k,d)
⟹
n' ∈ Inodep *and* (∀ sn' ∈ *elems* TREEL(n'))(sn' ∈ Tnodep) *and*
1 ≤ sizep(n') ≤ 2 *and len* KEYL(n') = sizep(n') - 1

L26

n ∈ Inodep *and* common-invp(n) *and* not-inp(k,n) *and* size-invp(m,n) *and*
n' = insertnp(m,n,k,d) ⟹
n' ∈ Inodep *and* 1 ≤ sizep(n') ≤ 2 *and len* KEYL(n') = sizep(n')-1

With these two lemmas, the proof of Theorem 2.12 will follow the lines of the proof of Theorem 1.12 (to which it corresponds).

The only lemma for which the proof is somewhat different for Representation 2 is L2.7, and so it is proved below. L2.7 is concerned with the preservation of keysets-are-orderedp on insertion, which is satisfied provided the precondition of the following lemma is satisfied:

L2.25

```
i1,i2,i3 ∈ inodep and is-subnode(i1,i2) and
within bounds(i1,i2,collect-keys(i3)) and keysets-are-orderedp(i1) and
keysets-are-orderedp(i3) and
(∀ i ∈ [1 . len TREEL(i3)])(depths(TREEL(i3)(i)) = depths(i2)) and
(((KEYL(i2)=< > and nk=nt) or (nk=position(KEYL(i2)(1),KEYL(i1)))) and
nt=position(TREEL(i2)(1),TREEL(i1))) ⟹

let r = replace(i1,i2,i3,nk,nt) in
keysets-are-orderedp(r) and
(∀ i ∈ [1 . len TREEL(r)])(depths(TREEL(r)(i) = depths(i1))
```

The proof that INSERTP preserves the invariant invp therefore models the proof that INSERT1 preserves the invariant inv.

Proof of Lemma 2.7

To prove:

(1) $n \in$ Nodep *and* common-invp(n) *and* not-inp(k,n) *and* n'=insertnp(m,n,k,d)

$\Rightarrow$

keysets-are-orderedp(n') *and* collect-keysp(n')=collect-keysp(n) $\cup$ {k}

Proof: By structural induction on n

BASIS

(2) n $\in$ Tnodep

which follows immediately from the equivalent of L1.2 for Representation 2.

INDUCTIVE HYPOTHESIS n $\in$ Inodep

assume

(3) $(\forall i \in \{1 \ldots len\ TREEL(n)\})$

let sn = TREEL(n)(i) *in*

common-invp(sn) *and* not-inp(k,sn) *and* sn'=insertnp(m,sn,k,d)

$\Rightarrow$

keysets-are-orderedp(sn') *and* collect-keysp(sn')=collect-keysp(sn) $\cup$ {k})

(4) common-invp(n) *and* not-inp(k,n) (Hypothesis)

(5) $(\forall i \in \{1 \ldots len\ TREEL(n)\})$(common-invp(TREEL(n)(i)) *and* not-inp(k,TREEL(n)(i)))

(by (4), common-invp *and* not-inp)

(6) *let* it,lk,ck,cn = selectp(n,k)

(7) cn $\in$ elems TREEL(n) (selectp)

(8) common-invp(cn) *and* not-inp(k,cn) ((5) and (7))

(9) *let* cnl = insertnp(m,cn,k,d)

(10) $(\forall i \in \{1 \ldots len\ TREEL(cnl)\})$(depths(TREEL(cnl)(i) = depths(cn))

(by L2.8 - equivalent of L1.8)

(11) keysets-are-orderedp(cnl) (by (7), (8) and (3))

(12) collect-keysp(cnl) = collect-keysp(cn) $\cup$ {k} (by (7), (8) and (3))

(13) *let* sn = << >,<cn>>

(14) is-subnode(n,sn) (by (7), (13) and is-subnode)

(15) collect-keysp(sn) = collect-keysp(cn) (by (13) and collect-keysp)

(16) within-bounds(n,sn,{k}) (by (14) and selectp)

(17) within-bounds(n,sn collect-keysp(cnl)) (by (16), (12), and (15))

(18) *let* rn = replace(n,sn,cnl,it,lt)

(19) keysets-are-orderedp(rn) ((4), (10), (11), (14), (17), selectp, replace, L2.25)

(20) collect-keysp(rn) = collect-keysp(n) $\cup$ {k} ((12), (15), replace L2.25)

(21) n' = *if* size(rn) $\leq 2^*m$ *then* {rn} *else* splitip(rn)

The lemma holds in both cases, from the definitions of splitip, collect-keysp, keysets-are-orderedp, (19) and (20).

Proof that INSERTP models INSERT

(D) Domains Rule

What must be shown is

(1) (V t ∈ BP-tree)(invp(t) *and* pre-INSERT1(retrp(t),k,d) ⟹
pre-INSERTP(t,k,d))

On substituting definitions this becomes

(2) (V t ∈ BP-tree)(invp(t) *and* k ∈ collect-keys(retrnp(TREEP(t)))
⟹ k ∈ collect-keysp(TREEP(t)))

Proof:
This follows the proof of the domains rule for operation FINDP except that all occurrences of k ∈ . . . are replaced by k ∈ . . .

(E) Results Rule

The rule to be proved is

(1) (V t ∈ BP-tree)(invp(t) *and* pre-INSERTP(t,k,d) *and*
post-INSERTP(t,k,d,t') ⟹ post-INSERT1(retrp(t),k,d,retrp(t')))

Expanding this gives

(2) (V t ∈ BP-tree)(invp(t) *and* k ∈ collect-keysp(TREEP(t)) *and*
ORDERP(t') = ORDERP(t) *and* TREEP(t') = insertrp(ORDERP(t),TREEP(t),k,d)
⟹
ORDERP(t')=ORDERP(t) *and*
TREEP(t')=insertrp(ORDERP(t),retrnp(TREEP(t)),k,d))

Proof:

INSERTP is identical to INSERT1 when applied to Tnodep (as a Tnodep is identical to a Tnode), and L2.24 and L2.25 show that INSERTP, when applied to an Inodep, achieves the same effect as INSERT1 when applied to an Inode.

As INSERTP preserves the invariant invp on BP-trees, and it has been shown that FINDP models FIND1, this implies that the results rule (2) is true. It can be proved formally to be true by structural induction on TREEP(t).

Operation DELETEP

(A) Preservation of the invariant

As the invariant of Representation 2 is of a form similar to the invariant of Representation 1 and the structure of DELETEP corresponds to that of DELETE1, the proof that DELETEP preserves the invariant, invp, can be modelled on the proof that DELETE1 preserves the invariant, inv.

As in the case of insertion, a list of Lemmas for Deletion for Representation 2 can be drawn up, which matches L1.13 to Theorem 1.23 of Representation 1, but refers to elements of lists rather than sets. Also, the only additional function used by DELETEP is the function replace.

Because of the way in which replace can be related to Representation 1, (L2.24 and L2.25), it follows that the proof of Theorem 2.23 has the same structure as the proof of Theorem 1.23, to which it corresponds, and it can be modelled on the preceeding proof.

Proof that DELETEP models DELETE1

(D) Domains Rule

What must be shown is

(1) (∀ t ∈ BP-tree)(invp(t) *and* pre-DELETE1(retrp(t),k) ⟹
pre-DELETEP(t,k))

On substituting definitions this becomes

(2) (∀ t ∈ BP-tree)(invp(t) *and* k ∈ collect-keys(retrnp(TREEP(t))) ⟹
k ∈ collect-keysp(TREEP(t)))

Proof:
This proof is identical to the proof of the domains rule for operation FINDP.

(E) Results Rule

The rule to be proved is

(1) (∀ t ∈ BP-tree)(invp(t) *and* pre-DELETEP(t,k) *and* post-DELETEP(t,k,t')
⟹
post-DELETE1(retrp(t),k,retrp(t')))

Expanding this gives

(2) (∀ t ∈ BP-tree)(invp(t) *and* k ∈ collect-keysp(TREEP(t)) *and*
ORDERP(t') = ORDERP(t) *and* TREEP(t') = deleterp(ORDERP(t),TREEP(t),k)
⟹
ORDER(t') = ORDER(t) *and* TREE(t') = deleter(ORDERP(t),retrnp(TREEP(t)),k))

Proof:
As a Tnodep is identical to a Tnode, DELETEP is identical to DELETE1 when applied to a Tnodep. It also follows, from L2.24 and L2.25, that DELETEP, when applied to an Inodep has a similar effect to DELETE1 applied to an Inode.

DELETEP has been shown to preserve the invariant, invp, on BP-trees and as it has been shown that FINDP models FIND1, it follows that (2) is true. It can also be proved to be true formally by using structural induction.

4.5 Realization

This stage consists of the program code corresponding to the FIND, INSERT and DELETE operations for B^+-trees. The system has been implemented in PASCAL, using the UCSD system on a DEC LSI-11.

The code for FIND and INSERT contains weakest pre-condition type assertions as an indication of its correctness. Assertions could be added to the code for DELETE in a similar manner.

The mapping of the B^+-tree onto linear storage is embodied in the pointer type variables of PASCAL and a form of has-no-loops (cf. section 3.3) for B^+-trees corresponding to that for binary trees should be true of these pointers.

When the height of the tree increases, a new node is allocated by using the new command of PASCAL. As the version of the UCSD system which was used (Version II.0) actually implements the heap as a stack, a corresponding release command could not be used to free nodes if the height of the tree should decrease, because the system cannot be relied upon to discard nodes in the order in which they were acquired.

As a function in PASCAL can only return a result which is of a simple type, the function INSERTN could not return an actual Inode. A globally allocated node, r_node, is therefore used to hold intermediate results produced by INSERTN, and the function returns a pointer to this node at each instance.

The layout of the node was chosen following the form of representation used in [7], as this shortens sections of the program code considerably. (The implementation was originally

based on a node containing separately defined lists of different lengths so an actual comparison of code lengths was made). The additional pointer in the case of an Inode is stored in the node variable , pmmi. In the case of a Tnode, this variable will be used to hold the sequential link to the following Tnode when the NEXT operation is implemented.

The program listing appears in the Appendix.

5. CONCLUSION

The most important aspect in the development of the representation of abstract mappings as B^+-trees, was the choice of correct development stages. It was crucial that each stage should capture just the right aspects and properties to allow the steps between successive stages to be made easily and smoothly. The correct choice of representation at each level makes the refinement process a natural progression.

The main difficulty in the refinement proofs was in proving that the operations for insertion and deletion preserve the invariant. This problem was solved by decomposing the rule to be proved into a series of lemmas in each case, from which the preservation of the invariant follows logically. The separation of lemmas into a clear sequence was not an easy task, but the resulting sets of lemmas greatly simplify the proofs of validity.

Another problem is how to relate error-handling in the actual program code to pre-conditions which occur in the specification. This could be done by appropriate sets of assertions in the program code. However in this development a comment has simply been inserted in the code where such checking occurs.

In conclusion, possible future extensions to this work should

be considered. Briefly, a few of these are:

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

The implementation of B^+-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.