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

Elizabeth Fielding

APPENDIX

Technical Monograph PRG-18
September 1980

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

APPENDIX

A compiler listing of the program code is included here. This listing contains the assertions as comments in the program text.

The listing has been separated and reordered in an attempt to make the program more readable. The main program and procedures appear first and then the procedures which are local to these are given in the order in which they occur textually in the program. The points at which local procedures must be included are marked with comments to this effect.

```
{$S+}


{ NOTE:                                                                 }
{       COMMENT BRACKETS OF THE FORM (* ... *) ARE USED TO ENCLOSE THE }
{       ASSERTIONS WHICH APPEAR IN THE PROGRAM.                         }


PROGRAM B_TREE(input,output);

    CONST
         MaxPromptLength = 255;

    TYPE  ScreenCommand = (Clear, Leave);
         PromptString = String[MaxPromptLength];

    FUNCTION  Capital(Ch : Char) : Char;
    FUNCTION  LowerCase(Ch : Char) : Char;
    FUNCTION  Prompt(P : PromptString; Command : ScreenCommand) : Char;
    PROCEDURE ClearLine;
    PROCEDURE ClearScreen;

uses promptunit;
                {NOTE: The routine promptunit is a USCD unit which was      }
                     {written at the PRG and provides an interactive menu   }
                     {interface.                                            }
                     { The first of its parameters is the string which is to }
                     {be used to prompt the user. The string must be supplied }
                     {in the form of a list of menu items, separated by     }
                     {commas. Within each item a ( must appear and the      }
                     {character immediately preceeding this is used as the  }
                     {selector.                                             }
                     { The second parameter specifies whether or not the    }
                     {screen is to be cleared before the prompt is displayed. }
                     { When the routine is called, it displays the string and }
                     {does not return control to the program until the user  }
                     {has responded with a valid reply (which corresponds to }
                     {one of the selectors). The lower case of the reply     }
                     {character is returned to the program.                 }
```

```
CONST
  m = 3;     {ORDER OF THE B_TREE}
  mm = 6;    {2 TIMES m}
  mm1 = 7;   {2 TIMES m + 1}
  tmm1 = 15; {A VALUE > mm1}

TYPE
  OUTKIND = (OCTAL,OCTBYTE);
  destination = (console,fyle);
  count_range = 0..mm;
  key_range = 1..mm;
  tree_range = 1..mm1;
  i_or_t = (inode,tnode);
  node_ptr = ^node;
  data_ptr = ^data_type;
  key_type = integer;
  data_type = integer;


  node_item = record
                key:key_type;

                case i_or_t of
                inode: (tree:node_ptr);
                tnode: (data:data_ptr)
              end;


  node_lists = array [1..mm] of node_item;


  node = record
           key_count:count_range;
           pmm1:node_ptr;

           case node_type:i_or_t of
           inode: (i_lists:node_lists);
           tnode: (t_lists:node_lists)
         end;

{NOTE: pmm1 is used to hold the additional pointer in the case of an Inode }
{      in the case of a Tnode it will be used to hold the pointer to the
{      following sequential Tnode when the NEXT command is implemented
```

```
VAR
  j:integer;
  k1,kn,ki :key_type;
  k :key_type;
  root,r_node :node_ptr;  {root is the root of the whole B-tree              }
                          {r_node is the globally declared node in which     }
                          {intermediate results of the function insertn      }
                          {are stored as this function returns the pointer }
                          {to r_node at each stage, as PASCAL functions      }
                          {can only return simple types                      }
  d :data_type;
  res_ptr:data_ptr;
  res :data_type;
  x,cd :char;
  level:integer;
  f :text;
  filename :string;


{-----------------------------------------------------------------------------}
{ The following routines are included at this point: print_octal            }
{                                                     index                   }
{                                                     locate                  }
{                                                     select                  }
{-----------------------------------------------------------------------------}
```

```
{$I b_find.text}

FUNCTION FIND (K:KEY_TYPE; TREE_PTR:NODE_PTR):DATA_PTR;
VAR
  i:tree_range;
BEGIN  {FIND}
      (*----------------------------*)
      (* k E collect_keys3(tree_ptr^) *)
      (*----------------------------*)
  with tree_ptr^ do
  begin
    if node_type = inode
    then
      (*--------------------------------------------------------------*)
      (* k E collect_keys3(tree_ptr^) and retrn3(tree_ptr^) E Inodep *)
      (*--------------------------------------------------------------*)
    begin
      i:=index (k,i_lists,key_count);
      (*-----------------------------------*)
      (* i = index(k,KEYL(retrn3(tree_ptr^))) *)
      (*-----------------------------------*)
      if i > mm
      then find:=find (k,pmm1)
      else find:=find (k,i_lists [i].tree)
      (*---------------------------------------------*)
      (* find = findp(k,retrn3(tree_ptr^.i_lists[i].tree) *)
      (*      = findp(k,TREEL(retrn3(tree_ptr^))(i)        *)
      (*---------------------------------------------*)
    end
    else
      (*--------------------------------------------------------------*)
      (* k E collect_keys3(tree_ptr^) and retrn3(tree_ptr^) E Tnodep *)
      (*--------------------------------------------------------------*)
    begin
      i:=locate (k,t_lists,key_count);
      if i > key_count
      (*--------------------------------------------------------------*)
      (* this checks the pre-condition: k E collect_keys3(tree_ptr^) *)
      (*--------------------------------------------------------------*)
      then find:=nil
      (*-----------------------------------------------*)
      (* ERROR (not shown in earlier stages of development) *)
      (*-----------------------------------------------*)
      else find:=t_lists [i].data
      (*---------------------------*)
      (* find = retrn3(tree_ptr^)(k) *)
      (*---------------------------*)
    end
  end
      (*--------------------------------*)
      (* find = findp(k,retrn3(tree_ptr^) *)
      (*--------------------------------*)
END;   {FIND}

{$I b_find.text}
```

```
PROCEDURE INSERTR (M:INTEGER; VAR N:NODE_PTR; K:KEY_TYPE; D:DATA_TYPE);


FUNCTION INSERTN (M:INTEGER; VAR N:NODE_PTR; K:KEY_TYPE; D:DATA_TYPE):NODE_PTR;
VAR
  i :tree_range;
  it :tree_range;
  ik :key_range;
  ck :key_type;
  cn :node_ptr;
  data_item :data_ptr;
  item :node_item;

{------------------------------------------------------------------------}
{ The following routines are included at this point: addtomap           }
{                                                    splitt             }
{                                                    i_replace          }
{                                                    spliti             }
{------------------------------------------------------------------------}
```

```
BEGIN {INSERTN}
      (*----------------------*)
      (* k ^E collect-keys3(n^) *)
      (*----------------------*)
  with n^ do
  begin
    if node_type = inode
    then
      (*--------------------*)
      (* retrn3(n^) E Inodep *)
      (*--------------------*)
    begin
      select (n,k,it,ik,ck,cn);
      (*----------------------------------*)
      (* it,ik,ck,cn = selectp(retrn3(n^),k) *)
      (*----------------------------------*)
      r_node:=insertn (m,cn,k,d);
      (*--------------------------------------------*)
      (* insertnp(m,retrn3(cn^),k,d) = retrn3(r_node^) *)
      (*--------------------------------------------*)
      if r_node^.key_count <> 0
      then                        { then split has occurred at the level below }
      begin
        if n^.key_count < mm
        then
      (*--------------------------------------------------------------*)
      (* let rn = replace(retrn3(n^),<< >,<retrn3(cn^)>>,retrn3(r_node^)) in *)
      (* sizep(rn) <= 2*m+1                                           *)
      (*--------------------------------------------------------------*)
            i_replace (n,it,r_node)
      (*--------------------------------------------------------------*)
      (* let rn = replace(retrn3(n^),<< >,<retrn3(cn^)>>,retrn3(r_node^)) in *)
      (* retrn3(r_node^) = << >,<rn>>                                 *)
      (*--------------------------------------------------------------*)
        else
      (*--------------------------------------------------------------*)
      (* let rn = replace(retrn3(n^),<< >,<retrn3(cn^)>>,retrn3(r_node^)) in *)
      (* sizep(rn) >2*m+1                                             *)
      (*--------------------------------------------------------------*)
            spliti (n,it,r_node)
      (*--------------------------------------------------------------*)
      (* let rn = replace(retrn3(n^),<< >,<retrn3(cn^)>>,retrn3(r_node^)) in *)
      (* retrn3(r_node^) = splitip(m,rn)                              *)
      (*--------------------------------------------------------------*)
        end
        { no split occurred at the level below so this node remains unaltered }
      else r_node^.i_lists [1].tree:=n
    end
    else
      (*--------------------*)
      (* retrn3(n^) E Tnodep *)
      (*--------------------*)
    begin
      r_node^.key_count:=0;
      new (data_item);                 { this allocates a location for a data }
      data_item^:=d;                   { item, and stores the data item and   }
      item.key:=k;                     { obtains a pointer to this location   }
      item.data:=data_item;
```

```
      i:=index (k,t_lists,key_count);
      (*-----------------------------*)
      (* i = index(k,KEYL(retrn3(n^))) *)
      (*-----------------------------*)
      if i <= key_count
      then
      (*--------------------------------------------------*)
      (* this checks the pre-condition: k ~E collect-keys3(n^) *)
      (*--------------------------------------------------*)
        begin
        if t_lists [i].key = k
        then
        begin
          writeln ('key ',k,' already exists');
          r_node^.key_count:=0;
          r_node^.i_lists [1].tree:=root;
          insertn:=r_node;
          exit (insertn)
        {ERROR: the tree is left unchanged }
        end
      end;
      if key_count < mm
      then
      (*--------------------------------*)
      (* sizep(retrn3(n^) + [k->d]) <= 2*m *)
      (*--------------------------------*)
           addtomap (n,i,item,r_node)
      (*--------------------------------*)
      (* let rn = retrn3(n^) + [k->d] in *)
      (* << >,<rn>> = retrn3(r_node^)    *)
      (*--------------------------------*)
      else
      (*------------------------------*)
      (* sizep(retrn3(n^) + [k->d] >2*m *)
      (*------------------------------*)
           splitt (n,i,item,r_node)
      (*-------------------------------*)
      (* let rn = retrn3(n^) + [k->d] in *)
      (* splitip(m,rn) = retrn3(r_node^) *)
      (*-------------------------------*)
    end
  end;
  insertn:=r_node
      (*-----------------------------------------*)
      (* insertnp(m,retrn3(n^),k,d) = retrn3(r_node^) *)
      (*-----------------------------------------*)
END;   {INSERTN}
```

```
BEGIN  {INSERTR}
        (*----------------------*)
        (* k "E collect-keys(n^) *)
        (*----------------------*)
  n:=insertn (m,n,k,d);
  if n^.key_count = 0
  then
        (*----------------------*)
        (* sizep(retrn3(n^)) = 1 *)
        (*----------------------*)
  n:=n^.i_lists [1].tree
  else
        { the tree has grown one level }
  begin
    new (n);
    n^:=r_node^
  end
        (*-------------------------------------*)
        (* retrn3(n^) = insertrp(m,retrn3(n^),k,d) *)
        (*-------------------------------------*)
END;   {INSERTR}
```

```
PROCEDURE DELETER (M:INTEGER; VAR N:NODE_PTR; K:KEY_TYPE);


FUNCTION DELETEN  (M:INTEGER; VAR N:NODE_PTR; K:KEY_TYPE):NODE_PTR;
VAR
  i,it :tree_range;
  ik :key_range;
  ck :key_type;
  cn,rn,nn :node_ptr;

{-----------------------------------------------------------------------------}
{ The following routines are included at this point: neighbour               }
{                                                     delfrommap              }
{                                                     order                   }
{                                                     merge                   }
{                                                     redistribute            }
{                                                     d_replace               }
{-----------------------------------------------------------------------------}
```

```
BEGIN  {DELETEN}
  if n^.node_type = inode
  then
  begin
    select (n,k,it,ik,ck,cn);
    rn:=deleten (m,cn,k);
    if rn^.key_count < m
    then
    begin
      nn:=neighbour (n,it);
      if nn^.key_count + rn^.key_count >= mm
      then
      begin
        redistribute (rn,nn,ck,it,n,r_node);
        d_replace (n,ik,it,r_node)
      end
      else
      begin
        merge (rn,nn,ck,it,n,r_node);
        d_replace (n,ik,it,r_node)
      end
    end
                            {if rn^.key_count >= m then the node remains}
                            {unchanged as it satisfies size-invn        }
  end
  else
  begin
    i:=locate (k,n^.t_lists,n^.key_count);
    if i > n^.key_count
    then writeln ('key ',k,' does not exist')
    else delfrommap (n,i)
  end;
  deleten:=n
END;  {DELETEN}
```

```
BEGIN  {DELETER}
  n:=deleten (m,n,k);
  if (n^.node_type = inode) and (n^.key_count = 0)
  then n:=n^.i_lists [1].tree
END;   {DELETER}
```

```
{---------------------------------------------------------------------------}
{ The following routine is included at this point: print_tree             }
{---------------------------------------------------------------------------}

PROCEDURE INITIALIZE;
BEGIN  {INITIALIZE}
  new (r_node);
  r_node^.key_count:=0;
  r_node^.node_type:=inode;
  new (root);
  root^.key_count:=0;
  root^.node_type:=tnode;
END;   {INITIALIZE}
```

```
BEGIN  {B_TREE}
  initialize;
  x:=prompt ('B-TREE: P(rint, F(ind, I(nsert, D(elete, Q(uit, G(en', leave);
  writeln;
  clearscreen;
  unitclear(1);
  while x <> 'q' do
  begin
    if (root^.key_count = 0) and (x <> 'i') and (x <> 'g')
    then writeln ('THE TREE IS EMPTY')
    else
    begin
      case x of
      'f': begin
             write ('TYPE KEY: ');
             readln (k);
             writeln;
             res_ptr:=find (k,root);
             if res_ptr = nil
             then writeln ('KEY ',k,' DOES NOT EXIST')
             else writeln ('DATA = ',res_ptr^);
             writeln
           end;
      'i': begin
             write ('TYPE KEY: ');
             readln (k);
             writeln;
             write ('TYPE DATA: ');
             read (d);
             writeln;
             insertr(m,root,k,d)
           end;
      'd': begin
             write ('TYPE KEY: ');
             readln (k);
             writeln;
             deleter (m,root,k)
           end;
      'p': begin
             level:=0;
             write('output to what file <cr> for console:? ');
             readln(filename);
             writeln;
             if length(filename) = 0 then filename := 'console:';
             {$I-}
             rewrite(f,filename);
             {$I+}
             if ioresult <>0 then exit(program);
             print_tree (root);
             close(f,lock)
           end;
      'g': begin
             cd:=prompt ('TYPE C(reate OR D(estroy',leave);
             writeln;
             clearscreen;
             write ('TYPE NUMBER OF KEYS: ');
             readln (kn);
             write ('TYPE START KEY: ');
```

```
            readln (k1);
            write ('TYPE INCREMENT: ');
            readln (ki);
            for j:=1 to kn do
            begin
              if cd = 'c' then insertr (m,root,k1,k1+1000)
              else deleter (m,root,k1);
              k1:=k1+ki
            end
          end
     end
    end;
    x:=prompt ('B-TREE: P(rint, F(ind, I(nsert, D(elete, Q(uit, G(en', leave);
    clearscreen;
    writeln
  end
END.
```

{NOTE: The routine print_octal was supplied by Jim Kaubisch}

```
{$I b_octprint.text}

PROCEDURE PRINT_OCTAL (NBR : node_ptr; WHICH_KIND :OUTKIND; DEST: DESTINATION);

TYPE
  KIND = (DECIMAL,OCTALNBR,OCTBITE);
  OCT_RANGE = 0..7;
  HEX_RANGE = 0..15;

VAR
  CONVERT :PACKED RECORD
           CASE KIND OF
             DECIMAL : (DEC :node_ptr);
             OCTALNBR: ( OCT :PACKED  RECORD
                                        LO5: 0..7;   {HI: 0..1;}
                                        LO4: 0..7;   {LO1: 0..7;}
                                        LO3: 0..7;   {LO2: 0..7;}
                                        LO2: 0..7;   {LO3: 0..7;}
                                        LO1: 0..7;   {LO4: 0..7;}
                                        HI: 0..1;    {LO5: 0..7;}
                                      END
                       );

             OCTBITE : (OCTB :PACKED  RECORD
                                   BHI3: 0..7;   {LO3: 0..7;}
                                   BHI2: 0..7;   {LO2: 0..7;}
                                   BHI1: 0..3;   {LO1: 0..3;}
                                   BLO3: 0..7;   {HI3: 0..7;}
                                   BLO2: 0..7;   {HI2: 0..7;}
                                   BLO1: 0..3;   {HI1: 0..3;}
                                 END
                       )

          END; {CONVERT}

  FUNCTION CONVERT_DIGIT ( NBR :HEX_RANGE): CHAR;
  BEGIN
    IF NBR < 10 THEN CONVERT_DIGIT := CHR(ORD('0')+NBR)
    ELSE CONVERT_DIGIT := CHR (ORD('A') + NBR - 10);
  END; {CONVERT_DIGIT}

  BEGIN
    CONVERT.DEC := NBR;
    CASE WHICH_KIND OF
      OCTAL    : BEGIN
                   WITH CONVERT.OCT DO
                   IF DEST = CONSOLE THEN
                     WRITE(HI:1,LO1:1,LO2:1,LO3:1,LO4:1,LO5:1)
                   ELSE WRITE(f,HI:1,LO1:1,LO2:1,LO3:1,LO4:1,LO5:1);
                 END;
      OCTBYTE : BEGIN
                 END;
    END; {CASE}
  END; {PRINT_OCTAL}

{$I b_octprint.text}
```

```
FUNCTION INDEX (K:KEY_TYPE; VAR LISTS:NODE_LISTS; KEY_COUNT:COUNT_RANGE):
                TREE_RANGE;
VAR
  j:tree_range;
  found:boolean;
BEGIN  {INDEX}
  j:=1;
  found:=false;
      (*-----------------*)
      (* t = key_count+1-j *)
      (*-----------------*)
  while
      (*------------------------------------------------------------------*)
      (*(found => j <= key_count and k <= lists[j].key)  and j <= key_count+1*)
      (*------------------------------------------------------------------*)
        (j <= key_count) and (not found) do
  begin
    if k > lists [j].key then j:=j+1 else found:=true
  end;
      (*---------------------------------------------------------*)
      (* (j <= key_count and k <= lists[j].key) or j = key_count+1 *)
      (*---------------------------------------------------------*)
 index:=j
      (*--------------------------------------------------*)
      (* let kl = (THE UNIQUE list)(len kl = key_count and *)
      (* (FORALL i E {1..key_count})(kl(i) = lists[i]) in  *)
      (* index = indexp(k,kl))                             *)
      (*--------------------------------------------------*)
END;   {INDEX}


FUNCTION LOCATE (K:KEY_TYPE; VAR LISTS:NODE_LISTS; KEY_COUNT:COUNT_RANGE):
                 TREE_RANGE;
VAR
  j:tree_range;
  found:boolean;
BEGIN  {LOCATE}
      (*---------------------------------*)
      (*k E {lists[i]| 1 <= i <= key_count} *)
      (*---------------------------------*)
  j:=1;
  found:=false;
      (*-----------------*)
      (* t = key_count+1-j *)
      (*-----------------*)
  while
      (*------------------------------------------------------------------*)
      (*(found => j <= key_count and k <= lists[j].key) and j <= key_count+1*)
      (*------------------------------------------------------------------*)
        (j <= key_count) and (not found) do
  begin
    if k <> lists [j].key then j:=j+1 else found:=true
  end;
      (*--------------------------------------------------------*)
      (* (j <= key_count and k = lists[j].key) or j = key_count+1 *)
      (*--------------------------------------------------------*)
```

```
locate:=j
      (*-------------------------------------*)
      (* locate = (THE UNIQUE i)(k = lists[i].key) *)
      (*-------------------------------------*)
END;   {LOCATE}

{$I b_indloc.text}
```

```
{$I b_select.text}

PROCEDURE SELECT (N:NODE_PTR; K:KEY_TYPE; VAR IT:TREE_RANGE;
                  VAR IK:KEY_RANGE; VAR CK:KEY_TYPE; VAR CN:NODE_PTR);
BEGIN  {SELECT}
  with n^ do
  begin
    it:=index (k,i_lists,key_count);
    if it > key_count then ik:=it-1 else ik:=it;
    ck:=i_lists [ik].key;
    if it = mm1 then cn:=pmm1 else cn:=i_lists [it].tree
  end
      (*---------------------------------*)
      (* it,ik,ck,cn = selectp(retrn3(n^),k) *)
      (*---------------------------------*)
END;  {SELECT}

{$I b_select.text}
```

```
PROCEDURE ADDTOMAP (N:NODE_PTR; I:TREE_RANGE; ITEM:NODE_ITEM; R_NODE:NODE_PTR);
VAR
  j:tree_range;
BEGIN  {ADDTOMAP}
      (*----------------------------*)
      (* m <= sizep(retrn3(n^)) < 2*m *)
      (*----------------------------*)
  with n^ do
  begin
    if i <= key_count
    then for j:=key_count+1 downto i+1 do t_lists [j]:=t_lists [j-1];
    t_lists [i]:=item;
    key_count:=key_count+1
      (*--------------------------------*)
      (* retrn3(n^) = retrn3(n^) + [k->d] *)
      (*--------------------------------*)
  end;
  r_node^.i_lists [1].tree:=n
      (*------------------------------------------------------------*)
      (* retrn3(r_node^) = << >,<retrn3(r_node^.i_lists[1].tree)>> *)
      (*                 = << >,<retrn3(n^) + [k->d]>>             *)
      (*------------------------------------------------------------*)
END;  {ADDTOMAP}

{$I b_addmap.text}
```

```
PROCEDURE SPLITT (N:NODE_PTR; I:TREE_RANGE; ITEM:NODE_ITEM; R_NODE:NODE_PTR);
VAR
  j:key_range;
  sn :node_ptr;
BEGIN  {SPLITT}
      (*--------------------------*)
      (* sizep(retrn3(n^)) >= 2*m *)
      (*--------------------------*)
  new (sn);
  sn^.node_type:=tnode;
  if i <= m+1
  then
      (*----------------------------------------*)
      (* let sk,sk = halve(dom retrn3(n^) U {k}) in *)
      (* k E sk                                  *)
      (*----------------------------------------*)
  begin
    for j:=1 to m do sn^.t_lists [j]:=n^.t_lists [m+j];
    sn^.key_count:=m;
      (*-----------------------------------------*)
      (* let sk,sk = halve(dom retrn3(n^) U {k}) in  *)
      (* retrn3(sn^) = [k'->retrn3(n^)(k')| k' E sk] *)
      (*-----------------------------------------*)
    if i <= m
    then for j:=m+1 downto i+1 do n^.t_lists [j]:=n^.t_lists [j-1];
    n^.t_lists [i]:=item;
    n^.key_count:=m+1
      (*---------------------------------------------------------*)
      (* let sk,sk = halve(dom retrn3(n^) U {k}) in               *)
      (* retrn3(n^) = [k'->retrn3(n^)(k')| k' E sk - {k}] + [k->d] *)
      (*---------------------------------------------------------*)
  end
  else
      (*----------------------------------------*)
      (* let sk,sk = halve(dom retrn3(n^) U {k}) in *)
      (* k E sk                                  *)
      (*----------------------------------------*)
  begin
    n^.key_count:=m;
      (*----------------------------------------*)
      (* let sk,sk = halve(dom retrn3(n^) U {k}) in *)
      (* retrn3(n^) = [k'->retrn3(n^)(k')| k E sk]  *)
      (*----------------------------------------*)
    for j:=1 to i-m-1 do sn^.t_lists [j]:=n^.t_lists [j+m];
    sn^.t_lists [i-m]:=item;
    if i <= mm
    then for j:=i to mm do sn^.t_lists [j-m+1]:=n^.tlists [j];
    sn^.key_count:=m+1;
      (*----------------------------------------------------------*)
      (* let sk,sk = halve(dom retrn3(n^) U {k}) in                *)
      (* retrn3(sn^) = [k'->retrn3(n^)(k')| k' E sk - {k}] + [k->d] *)
      (*----------------------------------------------------------*)
  end;
  r_node^.key_count:=1;
  r_node^.i_lists [1].key:=n^.t_lists [n^.key_count].key;
```

```
  r_node^.i_lists [1].tree:=n;
  r_node^.i_lists [2].tree:=sn
      (*-----------------------------------------------------*)
      (* let sk,gk = halve(dom retrn3(n^) U {k}) and         *)
      (* let rn = retrn3(n^) + [k->d] in                     *)
      (* retrn3(r_node^) = <<max(sk)>,<[k->rn(k)| k E sk],   *)
      (*                              [k->rn(k)| k E gk]>>   *)
      (*-----------------------------------------------------*)
END;    {SPLITT}


{$I b_splitt.text}
```

```
{$I b_ireplace.text}


PROCEDURE I_REPLACE (N:NODE_PTR; I:TREE_RANGE; R_NODE:NODE_PTR);
VAR
  j:tree_range;
BEGIN  {I_REPLACE}
      (*---------------------------------------------------*)
      (* 1 <= i <= 2*m =>                                  *)
      (* (let sn = << >,<retrn3(i_lists[i].tree)>> in      *)
      (* is-subnode(retrn3(n^),sn) and KEYL(sn) = < > and  *)
      (* nk = nt = i = position(TREEL(sn)(1),retrn3(n^)))) *)
      (*---------------------------------------------------*)
  with n^ do
  begin
    i_lists [i].tree:=r_node^.i_lists [2].tree;
    if key_count = mm-1
    then pmm1:=i_lists [mm].tree
    else i_lists [key_count+2].tree:=i_lists [key_count+1].tree;
    if i <= key_count
    then for j:=key_count+1 downto i+1 do i_lists [j]:=i_lists [j-1];
    i_lists [i]:=r_node^.i_lists [1];
    key_count:=key_count+1;
      (*------------------------------------------------------------*)
      (* retrn3(n^) = alter(TREEL(retrn3(n^),retrn3(i_lists[i].tree), *)
      (*                    TREEL(retrn3(r_node^)),i)                *)
      (*------------------------------------------------------------*)
    r_node^.key_count:=0
  end;
  r_node^.i_lists [1].tree :=n
      (*-----------------------------------------------------------------------*)
      (* retrn3(i_replace(n,i,r_node) =                                        *)
      (*replace(retrn3(n^),<< >,<retrn3(i_lists[i].tree>>,retrn3(r_node^),i,i)*)
      (*-----------------------------------------------------------------------*)
END;  {I_REPLACE}


{$I b_ireplace.text}
```

```
PROCEDURE SPLITI (N:NODE_PTR; I:TREE_RANGE; R_NODE:NODE_PTR);
VAR
  j:tree_range;
  sn:node_ptr;
BEGIN  {SPLITI}
      (*-----------------------------------------------------------------------*)
      (* let rn = replace(retrn3(n^),TREEL(retrn3(n^))(i),retrn3(r_node)) in *)
      (* len KEYL(rn) = 2*m+1 and len TREEL(rn) = 2*m+2                       *)
      (*-----------------------------------------------------------------------*)
  new (sn);
  sn^.node_type:=inode;
  if i < mm1
  then n^.i_lists [i].tree:=r_node^.i_lists [2].tree
  else n^.pmm1:=r_node^.i_lists [2].tree;
  if i <= m+1
  then
  begin
    for j:=1 to m do sn^.i_lists [j]:=n^.i_lists [j+m];
    sn^.i_lists [m+1].tree:=n^.pmm1;
    sn^.key_count:=m;
      (*-------------------------------------------*)
      (* retrn3(sn^) = <back(m,KEYL(retrn3(n^))),    *)
      (*                back(m+1,TREEL(retrn3(n^)))> *)
      (*-------------------------------------------*)
    if i <= m
    then for j:= m+1 downto i+1 do n^.i_lists [j]:=n^.i_lists [j-1];
    n^.i_lists [i]:=r_node^.i_lists [1];
    n^.key_count:=m
      (*-----------------------------------------------------------------------*)
      (* let rn = replace(retrn3(n^),TREEL(retrn3(n^))(i),retrn3(r_node)) in *)
      (* retrn3(n^) = <front(m,KEYL(rn)),front(m+1,TREEL(rn))>               *)
      (*-----------------------------------------------------------------------*)
  end
  else
  begin
    n^.key_count:=m;
      (*-------------------------------------------*)
      (* retrn3(n^) = <front(m,KEYL(retrn3(n^))),    *)
      (*               front(m+1,TREEL(retrn3(n^)))> *)
      (*-------------------------------------------*)
    if i > m+2
    then for j:=m+2 to i-1 do sn^.i_lists [j-m-1]:=n^.i_lists [j];
    sn^.i_lists [i-m-1]:=r_node^.i_lists [1];
    if i < mm1
    then for j:=i to mm do sn^.i_lists [j-m]:=n^.i_lists [j];
    sn^.i_lists [m+1].tree:=n^.pmm1;
    sn^.key_count:=m;
      (*-----------------------------------------------------------------------*)
      (* let rn = replace(retrn3(n^),TREEL(retrn3(n^))(i),retrn3(r_node)) in *)
      (* retrn3(sn^) = <back(m,KEYL(rn)),back(m+1,TREEL(rn))>                *)
      (*-----------------------------------------------------------------------*)
  end;
  r_node^.key_count:=1;
  r_node^.i_lists [1].key:=n^.i_lists [m+1].key;
  r_node^.i_lists [1].tree:=n;
```

```
  r_node^.i_lists [2].tree:=sn
      (*------------------------------------------------------------*)
      (* retrn(r_node^) = <<KEYL(rn)(m+1)>,<retrn3(n^),retrn3(sn^)>> *)
      (*------------------------------------------------------------*)
END;   {SPLITI}


{$I b_spliti.text}
```

```
FUNCTION NEIGHBOUR (N:NODE_PTR; I:TREE_RANGE):NODE_PTR;
BEGIN  {NEIGHBOUR}
  with n^ do
  besin
    if i < mm
    then neishbour:=i_lists [i+1].tree
    else if i = mm
         then neishbour:=pmm1
         else neishbour:=i_lists [mm].tree
  end
END;   {NEIGHBOUR}

{$I b_neish.text}
```

```
{$I b_delmap.text}

PROCEDURE DELFROMMAP (N:NODE_PTR; I:KEY_RANGE);
VAR
  j:tree_range;
BEGIN  {DELFROMMAP}
  with n^ do
  begin
    if key_count > 1
    then for j:=i to key_count-1 do t_lists [j]:=t_lists [j+1];

{NOTE: if key_count = 1, the tree consists of a Tnode containing a single }
{      key, data pair, so there is no point in shifting the items of the  }
{      node, all that has to be done is to set the key_count to zero.     }
{      Also, as j is of type tree_range, the range variables of the for   }
{      loop must lie in this subrange.                                    }

    key_count:=key_count-1
  end
END;  {DELFROMMAP}

{$I b_delmap.text}
```

```
PROCEDURE ORDER (VAR N1:NODE_PTR; VAR N2:NODE_PTR; I:TREE_RANGE; N:NODE_PTR);
VAR
  temp1,temp2 :node_ptr;
BEGIN  {ORDER}
  if i > n^.key_count
  then
  begin
    if i=mm1
    then temp1:=n^.pmm1 else temp1:=n^.i_lists [i].tree;
    temp2:=n1;
    n1:=n2;
    n2:=temp2
  end
END;   {ORDER}

{$I b_order.text}
```

```
{$I b_merge.text}

PROCEDURE MERGE (VAR N1:NODE_PTR; VAR N2:NODE_PTR; K:KEY_TYPE; I:TREE_RANGE;
                N:NODE_PTR; R_NODE:NODE_PTR);
VAR
  j :tree_range;
  l1,l2 :key_range;
  temp:node_ptr;


PROCEDURE PRE_MERGE (VAR N1:NODE_PTR; VAR N2:NODE_PTR);
BEGIN  {PRE_MERGE}
  if n1 = n2
  then
  begin
    writeln ('attempt to merge identical nodes');
    exit (merge)
  end
  else
  if n1^.node_type <> n2^.node_type
  then
  begin
    writeln ('attempt to merge nodes of different type');
    exit (merge)
  end
END;   {PRE_MERGE}



BEGIN  {MERGE}
  pre_merge (n1,n2);
  order (n1,n2,i,n);
  l1:=n1^.key_count;
  l2:=n2^.key_count;
  if n1^.node_type = inode
  then
  begin
    n1^.i_lists [l1+1].key:=k;
    for j:=1 to l2 do n1^.i_lists [l1+j+1]:=n2^.i_lists [j];
    if l2 = mm
    then temp:=n2^.pmm1
    else temp:=n2^.i_lists [l2+1].tree;
    if l1+l2 < mm-1
    then n1^.i_lists [l1+l2+2].tree:=temp
    else n1^.pmm1:=temp;
    n1^.key_count:=l1+l2+1
  end
  else
  begin
    for j:=1 to l2 do n1^.t_lists [l1+j]:=n2^.t_lists [j];
    n1^.key_count:=l1+l2
  end;
  r_node^.key_count:=0;
  r_node^.i_lists [1].tree:=n1
END;   {MERGE}

{$I b_merge.text}
```

```
PROCEDURE REDISTRIBUTE (VAR N1:NODE_PTR; VAR N2:NODE_PTR; K:KEY_TYPE;
                        I:TREE_RANGE; N:NODE_PTR; R_NODE:NODE_PTR);
VAR
  temp_node : array [1..tmm1] of node_item;
  j,b :integer;
  l1,l2,nl1,nl2 :key_range;



PROCEDURE PRE_REDISTRIBUTE (VAR N1:NODE_PTR; VAR N2:NODE_PTR);
BEGIN  {PRE_REDISTRIBUTE}
  if n1 = n2
  then
  begin
    writeln ('attempt to redistribute identical nodes');
    exit (redistribute)
  end
  else
  if n1^.node_type <> n2^.node_type
  then
  begin
    writeln ('attempt to redistribute nodes of different type');
    exit (redistribute)
  end
END;   {PRE_REDISTRIBUTE}



BEGIN  {REDISTRIBUTE}
  pre_redistribute (n1,n2);
  order (n1,n2,i,n);
  l1:=n1^.key_count;
  l2:=n2^.key_count;
  if n1^.node_type = inode
  then
  begin                            {merge}
    for j:=1 to l1 do temp_node [j]:=n1^.i_lists [j];
    temp_node [l1+1].key:=k;
    if l1 = mm
    then temp_node [l1+1].tree:=n1^.pmm1
    else temp_node [l1+1].tree:=n1^.i_lists [l1+1].tree;
    for j:=1 to l2 do temp_node [l1+j+1]:=n2^.i_lists [j];
    if l2 = mm
    then temp_node [l1+l2+2].tree:=n2^.pmm1
    else temp_node [l1+l2+2].tree:=n2^.i_lists [l2+1].tree;
    nl1:=(l1+l2) div 2;
    nl2:=(l1+l2) - nl1;
                                   {split}
    for j:=1 to nl1+1 do n1^.i_lists [j]:=temp_node [j];
    n1^.key_count:=nl1;
    for j:=1 to nl2+1 do n2^.i_lists [j]:=temp_node [nl1+j+1];
    n2^.key_count:=nl2;
    r_node^.i_lists [1].key:=temp_node [nl1+1].key
  end
  else
  begin
```

```
    if l1 < l2
    then
    begin
      b:=(l2-l1) div 2;
      for j:=1 to b do n1^.t_lists [l1+j]:=n2^.t_lists [j];
      for j:=1 to l2-b do n2^.t_lists [j]:=n2^.t_lists [j+b];
      n1^.key_count:=n1^.key_count+b;
      n2^.key_count:=n2^.key_count-b
    end
    else
    begin
      b:=(l1-l2) div 2;
      for j:=l2+b downto 1 do n2^.t_lists [j]:=n2^.t_lists [j-b];
      for j:=1 to b do n2^.t_lists [j]:=n1^.t_lists [l1-b+j];
      n1^.key_count:=n1^.key_count-b;
      n2^.key_count:=n2^.key_count+b;
    end;
    r_node^.i_lists [1].key:=n1^.t_lists [n1^.key_count].key
  end;
  r_node^.key_count:=1;
  r_node^.i_lists [1].tree:=n1;
  r_node^.i_lists [2].tree:=n2
END;   {REDISTRIBUTE}

{$I b_redistr.text}
```

```
{$I b_dreplace.text}

PROCEDURE D_REPLACE (N:NODE_PTR; IK:KEY_RANGE; IT:TREE_RANGE;
                     R_NODE:NODE_PTR);
VAR
  j:tree_range;
BEGIN  {D_REPLACE}
  with n^ do
  begin
    if r_node^.key_count = 0
    then
    begin
      if ik < key_count
      then for j:=ik to key_count-1 do i_lists [j]:=i_lists [j+1];
      if key_count = mm
      then i_lists [key_count].tree:=pmm1
      else i_lists [key_count].tree:=i_lists [key_count+1].tree;
      i_lists [ik].tree:=r_node^.i_lists [1].tree;
      key_count:=key_count-1
    end
    else
    begin
      i_lists [ik]:=r_node^.i_lists [1];
      if it = mm1
      then pmm1:=r_node^.i_lists [2].tree
      else i_lists [it+1].tree:=r_node^.i_lists [2].tree
    end
  end
END;   {D_REPLACE}

{$I b_dreplace.text}
```

```
{$I B_print.text}

PROCEDURE PRINT_TREE (N:NODE_PTR);
VAR
  leaves :boolean;
  depth, level :integer;




PROCEDURE PRINT_NODE (N:NODE_PTR; LEVEL:INTEGER; DEPTH:INTEGER
                      VAR LEAVES:BOOLEAN);
VAR
  kind :char;
  j :tree_range;
BEGIN  {PRINT_NODE}
  with n^ do
  begin
    if depth = level
    then
    begin
     if node_type = inode then kind:= 'I' else kind:='T';
   writeln (f,'------------------------------------------------------------------');
      writeln(f);
      writeln (f,'LEVEL ',level:5,':');
      writeln(f);
      write (f,'NODE ');
      print_octal(n,octal,fyle);
      write(f,'     TYPE: ',kind);
      write (f,'      NUMBER OF KEYS: ', key_count);
      writeln(f);
      writeln(f);
      write (f,'KEYS: ');
      if node_type = inode
      then
      begin
        for j:= 1 to key_count do write (f,i_lists [j].key:8);
        writeln(f);
        write (f,'TREES: ');
        for j:=1 to key_count do
        begin
          print_octal(i_lists [j].tree,octal,fyle);
          write(f,' ');
        end;
        if key_count = mm then
        begin
          print_octal(pmm1,octal,fyle);
          write (f,' ')
        end
        else
        begin
          print_octal (i_lists [key_count+1].tree,octal,fyle);
          write (f,' ')
        end;
        writeln(f);
        writeln(f)
      end
      else
```

```
        begin
          for j:=1 to key_count do write (f,t_lists [j].key:8);
          writeln(f);
          write (f,'DATA: ');
          for j:=1 to key_count do write (f,t_lists [j].data^:8);
          writeln(f);
          writeln(f)
        end
      end
      else
      begin
        if node_type = inode
        then
        begin
          for j:=1  to key_count do
               print_node (i_lists [j].tree,level,depth+1,leaves);
          if key_count = mm then print_node(pmm1,level,depth+1,leaves)
          else print_node (i_lists [key_count+1].tree,level,depth+1,leaves)
        end
        else leaves:=true
      end
    end
  END;    {PRINT_NODE}




  BEGIN  {PRINT_TREE}
    with n^ do
    begin
      if key_count <> 0
      then
      begin
        leaves:=false;
        level:=0;
        while not (leaves) do
        begin
          depth:=1;
          level:=level+1;
          print_node(n,level,depth,leaves)
        end;
      writeln (f,'-------------------------------------------------------------------');
      end
    end
  END;    {PRINT_TREE}

  {$I B_print.text}
```