Access Functions for Packed Scatter Tables

Bruce E. Martin

Systems and Software Division
Institute for Computer Sciences and Technology
National Bureau of Standards
Washington, D.C. 20234

August 1978
ACCESS FUNCTIONS FOR PACKED SCATTER TABLES

Bruce E. Martin

Systems and Software Division
Institute for Computer Sciences and Technology
National Bureau of Standards
Washington, D.C. 20234

August 1978
ACCESS FUNCTIONS FOR PACKED SCATTER TABLES

Bruce E. Martin

Systems and Software Division
Institute for Computer Sciences and Technology
National Bureau of Standards
Washington, D.C. 20234

August 1978
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUMMARY OF THE ACCESS METHOD</td>
<td>2</td>
</tr>
<tr>
<td>USE OF THE ROUTINES</td>
<td>5</td>
</tr>
<tr>
<td>Declarations</td>
<td>6</td>
</tr>
<tr>
<td>Initializations</td>
<td>6</td>
</tr>
<tr>
<td>Node (De)allocation</td>
<td>7</td>
</tr>
<tr>
<td>APPENDIX A: Documentation of PASCAL code</td>
<td>8</td>
</tr>
<tr>
<td>APPENDIX B: Listings of access routines</td>
<td>16</td>
</tr>
<tr>
<td>APPENDIX C: Sample results</td>
<td>21</td>
</tr>
</tbody>
</table>
DISCLAIMER

The table functions described in this report have been written and tested carefully. The possibility of improper application requires that NBS expressly disclaim any and all consequences of using the functions. A particular warning must be issued on attempts to speed table access by storing indices: such schemes are guaranteed failure, since items move with new insertions.
Access Functions for Packed Scatter Tables

Bruce E. Martin

Three PASCAL access routines are given for packed scatter tables. INSERT packs tables of integer keys; FIND retrieves the keys; DELETE deletes keys. While the routines currently access integer keys (for use in performance testing with pseudo-random integers), they can easily be converted to access other data types -- character strings in a symbol table, for example. To enhance portability, the code is straightforward PASCAL without any input-output capabilities. Appendices contain specific comments on the routines, listings, and sample results.

SUMMARY OF THE ACCESS METHOD

Hashing techniques allow insertion and retrieval of keys by computing a function h(key) and storing the key in T[h(key)], where T is an indexed table. Since the function h(key) generally does not compute unique addresses, different keys may "collide". Simple collision-resolving methods search for an alternate location to store the key.
being inserted. But such methods can cause slow retrieval of keys. More sophisticated methods search for alternate locations for other keys as well as for the key being inserted. INSERT uses a generalized collision-resolving method that recursively considers table rearrangement. The user controls the method to be used with the parameter DEPTH. Lyon gives a more detailed discussion of the algorithm in [1]. A summary of the method appears here. Specific comments on the PASCAL code appear in appendix A.

Integer keys are inserted in the scatter table by first calculating a table address using a primary hash function (key mod table size). If the slot in the table addressed by the primary hash function is not occupied, the key is inserted. Otherwise, the cost of displacing the contents of the slot is calculated by calling the function DISPLACE. Next, the cost of displacing the key is calculated by a second call of DISPLACE. DISPLACE returns a stack that indicates how the table should be rearranged. The table is rearranged using the stack returned by the call of DISPLACE with the least displacement cost.

The function DISPLACE makes probes into the table by increments of the secondary hash step (key mod (table size-2) +1) until an open slot is found. Each slot probed during the search for an open slot is considered for displacement by recursive calls of DISPLACE. The recursion terminates
when the deepest level of recursion, as specified by the user with the parameter DEPTH, is reached or whenever displacement of further slots cannot possibly find a better solution. The deepest call of DISPLACE returns the additional penalty to probe to the free slot; that is, PENALTY(number of probes to free slot) minus PENALTY(number of probes to table address), where PENALTY is a forcing function defined by the user (see [1]). The rearrangement stack returned by the deepest call of DISPLACE consists of the table address of the item being displaced and the address of the free slot. At higher levels, the total displacement cost of each subsequent slot is calculated to be the additional cost to probe to the slot plus its displacement cost, and the minimum is returned. The stack at higher levels of DISPLACE consists of the address of the item being displaced and the stack returned from the chosen call of DISPLACE.

A table of counters of search lengths is maintained for faster rejection of keys not in the table. As keys are inserted, the number of probes to find each key is recorded by incrementing a counter in the search length counter table, KICKOUT, where the search length is the index of KICKOUT. Since key insertion possibly causes other keys to be displaced, thus changing each displaced key's search length, old and new search lengths of displaced keys are also returned on the rearrangement stack to keep KICKOUT updated.
and maintained. When a key is deleted, the appropriate search length counter is decremented.

Keys are retrieved by first calculating their original primary hash function and secondary hash step. Next, the table is probed until the key or an open slot is found or the number of slots probed equals the maximum search length. Generally, the performance of retrieving keys in the table does not improve as keys are deleted. For example, a table half filled performs much better than a table that is first completely filled and then has half of its keys deleted. Rejecting keys not in the table improves as keys are deleted provided deletions cause the maximum search length to decrease. Again, the half-filled table performs better than the table that is half-deleted. However, as keys are reinserted, both rejection and acceptance improve because insertion of keys causes the table to be rearranged more optimally. See Appendix C for measurements that are typical of correctly executing functions: Testing of the routines on a new system should give similar results.
USE OF THE ROUTINES

Declarations

The following must be declared by the calling program:

```pascal
const
tablesize   {the size of the table into which keys
               will be inserted. Must be prime!}
maxreal     {largest real for particular installation.}
kicksize    {size of the search counter table}*
type
  table=array[0..tablesise-1] of integer;
kicktab= array[-1..kicksize+1] of integer;
stkptr = ^stkelmnt;
stkelmnt = record
  ind, oldlen, newlen: integer;
  next: stkptr
end;

var
  oldnodes: stkptr;    {for node (de)allocation}
```

* Kicksize should be the expected longest probe for a
  particular depth, penalty function and table filling. If the estimate of the maximum search length
  is too small, the rejection performance of the table may deteriorate. Therefore, a generous estimate of
  kicksize is desirable.

Initializations

The following must be initialized by the calling program:

<table>
<thead>
<tr>
<th>VARIABLES</th>
<th>WHEN</th>
<th>INITIALIZED TO</th>
</tr>
</thead>
<tbody>
<tr>
<td>of type TABLE</td>
<td>for new table</td>
<td>-1</td>
</tr>
<tr>
<td>OLDNODES</td>
<td>first use of routines</td>
<td>nil</td>
</tr>
</tbody>
</table>
of type KICKTAB for new table as follows:

kickout[-1]:=0;  {no overflow with new table}
kickout[0]:=1;  {maximum probes with new table}
kickout[1..kicksize+1]:=0;

Node (De)allocation

Since node (de)allocation differs from one PASCAL installation to the next, the INSERT routine, in the interest of portability, explicitly controls node (de)allocation for its rearrangement stack via procedures GETNODE and FREENODE. The procedures use a global variable OLDNODES, which points to a linked list of nodes that grows and shrinks during execution.

REFERENCES


APPENDIX A: Documentation of PASCAL code

**Procedure insert**

**PARAMETER TYPE**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tab</td>
<td>table</td>
<td>the table in which keys are inserted.</td>
</tr>
<tr>
<td>key</td>
<td>integer</td>
<td>the integer to be inserted.</td>
</tr>
<tr>
<td>depth</td>
<td>integer</td>
<td>depth of recursion for displacement.</td>
</tr>
<tr>
<td>kickout</td>
<td>kicktab</td>
<td>table of counters of search lengths. KICKOUT[1..KICKSIZE] are counters of search lengths, where the length is the index in KICKOUT. KICKOUT[0] = longest search length. KICKOUT[-1] = longest search length if an overflow occurs. KICKOUT[KICKSIZE+1] is counter of overflow search lengths. KICKOUT[0] traps to KICKSIZE+1 if overflow occurs.</td>
</tr>
</tbody>
</table>

**VARIABLE TYPE**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>integer</td>
<td>primary hash index. TAB[INDEX] is considered for displacement if a collision occurs.</td>
</tr>
<tr>
<td>temp</td>
<td>integer</td>
<td>stores the contents of TAB[INDEX] while displacement of the new key is being considered.</td>
</tr>
<tr>
<td>lenl</td>
<td>integer</td>
<td>length of the longest search returned when TAB[INDEX] is considered for displacement.</td>
</tr>
</tbody>
</table>
len2 integer
length of the longest search returned when the key is considered for displacement.

cost1 real
cost of displacing TAB[INDEX].

cost2 real
cost of displacing the key.

stk1 stkptr
rearrangement stack returned when TAB[INDEX] is considered for displacement.

stk2 stkptr
rearrangement stack returned when the key is considered for displacement.

LINE NUMBER(S)
142..144

The primary hash index is calculated. Another key has search length of 1 so KICKOUT[1] is incremented. If the slot is empty or marked deleted, the key is inserted and INSERT is exited. -1 indicates empty slots; -2 marks deleted slots;

148..149

The cost of displacing TAB[INDEX] is calculated provided DEPTH > 0. DEPTH=0 means the key should be inserted in the first free slot and no displacements occur.

150..152

The cost of displacing the key is calculated by temporarily storing TAB[INDEX] in TEMP. The key is inserted in tab[index] and DISPLACE is called. This was designed so that DISPLACE would have the table address as an parameter (necessary for recursive calls) and so INSERT would have the key as a parameter, making table addresses invisible to the user. Note that with the second call of DISPLACE, COST1 is the actual parameter corresponding to the DISPLACE
formal parameter MAX. This keeps the second call of DISPLACE from considering any displacements that are more costly than the displacement found by the first call of DISPLACE.

153..162

If both the key and TAB[INDEX] were considered for displacement, the table is rearranged by REARRANGE according to the stack returned by the call of DISPLACE returning the lower cost. If only the key was considered for displacement, the table is rearranged by STK2. Finally, after table rearrangement, both stacks are deallocated by FREENODE.

procedure getnode
7..15

A node is allocated from OLDDNODES or by the pascal function NEW.

procedure freenode
16..27

A linked list is walked and deallocated to OLDDNODES.

procedure rearrange
30

If the search length passed to it is greater than the current maximum in KICKOUT[0], KICKOUT[0] is updated.

31..47

Each node in the stack has four fields: IND, OLDLEN, NEWLEN and NEXT. For each node: a) OLDLEN and NEWLEN are tested for overflow. If so, KICKOUT[0] traps to the overflow counter. b) KICKOUT[NEWLEN] is incremented and KICKOUT[OLDLEN] is decremented. c) the contents of TAB[NEXT`.IND] are moved to TAB[IND]. d) the next node of the stack is used. When the last node is encountered, the key is moved to TAB[IND].
function displace

PARAMETER TYPE

index integer
address of item to be considered for displacement.

depth integer
depth of recursion for which displacement should be considered.

max real
MAX is an upper limit on cost for displacement consideration. It is the best solution found so far at higher levels of recursion.

rjstack stkptr
contains indices of slots rejected at higher levels of recursion. A slot which is rejected at a higher level of recursion will not lead to a better solution at a deeper level.

stack stackptr
returns the rearrangement stack for best solution at a given level. (var parameter)

length integer
returns the length of the longest search. (var parameter)

VARIABLE TYPE

ind integer
used to calculate subsequent slots in the table.

probetoind integer
number of probes to hash to TAB[INDEX].

probetofree integer
number of probes to the first free slot.

counter integer
slot counter. Used in calculating subsequent locations to probe
step integer
equal to the secondary hash function for probing.

hashl integer
equal to the primary hash function for probing.

next integer
address of the next slot.

srcrlen integer
longest search from deeper levels of recursion.

uplim real
the upper limit on displacement cost at a given level. UPLIM is the minimum of the cost to move TAB[INDEX] to a free slot and MAX (the least cost found at higher levels).

totcost real
additional cost of probing to next slot plus cost of displacing next slot.

pentonext real
additional cost of probing to next slot, that is PENALTY(probes to next slot) - PENALTY(probetoinid).

thisnode stkptr
pointer to node pushed on stack. It is used to update the new search length.

savrj stkptr
saves a copy of rjstack upon first execution of DISPLACE.

bestack stkptr
saves the stack returned by DISPLACE returning the lowest cost. BESTACK is in turn returned to higher levels of DISPLACE.

tstack stkptr
temporary stack for calls of
The primary and secondary hash functions are calculated. Slots are probed to find the number of probes to INDEX and a free slot.

INDEX, number of probes to index, number of probes to free slot are pushed on STACK as IND, OLDLEN and NEWLEN, respectively. NEWLEN may have to be updated later if a better solution is found. Therefore, THISNODE saves the node. So far, the best solution found is to move TAB[INDEX] to a free slot so BESTACK is set to STACK. RJSTACK is saved.

The tentative longest search is PROBETO-FREE so LENGTH defaults to PROBETO-FREE. The index of the free slot and two dummy constants are pushed on BESTACK. UPLIM is the additional cost of probing to the free slot.

If DEPTH=0, no subsequent slots are to be considered for displacement. The recursion has terminated. The stack with INDEX and the address of the free slot is returned. The value of DISPLACE is UPLIM. Otherwise, subsequent slots are considered for displacement.

If a better solution than UPLIM was found from a higher level of recursion then UPLIM is updated. The primary hash location is the first to be considered for relocation. The cost to probe to the first location is PENALTY(1) - PENALTY(probetoind).

While the cost to probe to the next slot is greater than UPLIM, the following is done:
If the next slot has not already been considered, it is considered for relocation. If the total cost, TOTCOST, is lower than the current lowest cost UPLIM, a better solution has been found and UPLIM, BESTACK and LENGTH are updated to TOTCOST, TSTACK and SRCHLEN, respectively. Otherwise, the slot being considered for relocation is pushed on the reject stack, RJSTACK. The next slot to be considered and the additional cost to probe to it are calculated.

The value of DISPLACE returned is the UPLIM. BESTACK is returned as the rearrangement stack.

**procedure push**

PUSH gets a new node and pushes the arguments I, OLDLEN, NEWLEN onto the stack.

**function member**

MEMBER returns true if its integer argument is a member of the stack. Otherwise it returns false.

**function penalty**

PENALTY is a forcing function defined by the user. Currently it is linear; it returns its argument. To force insertion of keys in a different manner the user must change the forcing function.

**function find**

FIND tries to find the key in the table. If found, its table location is returned, otherwise -1 is returned. The variable probes returns the number of probes to find (or reject) the key.

**procedure delete**

DELETE finds the key in the table by calling function FIND, deletes the key
and updates KICKOUT accordingly. A slot is marked deleted by setting it to -2.
APPENDIX B: Listings of access routines

procedure insert (var tab: table; key, depth: integer;
  var kickout:kicktab);

var
  index, temp, len1, len2: integer;
  cost1, cost2: real;
  stk1, stk2: stkptr;

procedure getnode(var p:stkptr);
begin
  if oldnodes=nil then new(p)
  else begin
    p:=oldnodes;
    oldnodes:=oldnodes^.next;
  end;
  p^.next:=nil
end;

procedure freenode (first,last:stkptr);
var
  x:stkptr;
begin
  if first<>last then begin
    x:=first;
    while first^.next<>last do first:=first^.next;
    first^.next:=oldnodes;
    oldnodes:=x
  end;
end;

procedure rearrange (stack: stkptr; key, length: integer);
begin
  if length > kickout[0] then kickout[0]:=length;
  repeat
    if stack^.newlen > kicksize then begin
      if kickout[-1] < stack^.newlen then begin
        kickout[-1]:=stack^.newlen;
        kickout[0]:=kicksize+1; { trap to overflow counter }
        stack^.newlen:=kicksize+1
      end;
    end;
  end;
end;
function displace (index, depth: integer; max: real; rjstack: stkptr;
var stack: stkptr; var length: integer): real;

var
  ind, probetoind, probetofree, counter, step, hashl,
  next, srchlen: integer;
  uplim, totcost, pentonext: real;
  thisnode, savrj, bestack, tstack: stkptr;

procedure push (i, oldlen, newlen: integer; var stack: stkptr);
var
  node: stkptr;
begin
  getnode(node);
  node^.ind:=i;
  node^.newlen:=newlen;
  node^.oldlen:=oldlen;
  node^.next:=stack;
  stack:=node
end;

function member (i: integer; stk: stkptr): boolean;
var
  found: boolean;
begin
  found:=false;
  while (stk <> nil) and (not found) do
    if stk^.ind=i then found:=true
    else stk:=stk^.next;
  member:=found
end;

function penalty (i: integer): real;
{to be defined as desired; currently linear}
begin
  penalty:=float(i)
end;

begin { function displace }
  step:=(tab[index] mod (tablesize-2))+1;
  hashl:=tab[index] mod tables;
  probetoind:=0;
  repeat

ind:=(hashl + probetoind * step) mod tablesiz;
probetoind:=probetoind + 1
until ind=index;
probetofree:=0;
repeat
ind:=(hashl + probetofree * step) mod tablesiz;
probetofree:=probetofree + 1
until (tab[ind]=-1) or (tab[ind]=-2);
push(index,probetoind,probetofree,stack);
thisnode:=stack;
tstack:=stack;
bestack:=stack;
savrj:=rjstack;
length:=probetofree;
uplim:=penalty(probetofree)-penalty(probetoind);
push(ind,1,1,bestack);
if depth > 0 then
begin
if uplim > max then uplim:=max;
counter:=0;
next:=hashl;
pentonext:=penalty(1) - penalty(probetoind);
while uplim > pentonext do
begin
if (not member(next,stack)) and (not member(next,rjstack))
then
begin
totcost:=pentonext + displace(next, depth-1, uplim-pentonext, 
rjstack, tstack, srchlen);
if totcost < uplim then
begin
uplim:=totcost;
freenode(bestack,stack);
bestack:=tstack;
thisnode^.newlen:=counter+1;
if 1+counter > srchlen then length:=l+counter
else length:=srchlen;
end
else begin
push(next,1,1,rjstack);
freenode(tstack,stack)
end;
tstack:=stack;
end;
counter:=counter+1;
next:=(hashl + counter * step) mod tablesiz;
pentonext:=penalty(l+counter) - penalty(probetoind)
end;
freenode(rjstack,savrj)
end;
stack:=bestack;
displace:=uplim
end;

begin  { procedure insert }
index:=key mod tablesizel;
kickout[l]:=kickout[l]+1;
if (tab[index]=-1) or (tab[index]=-2) then tab[index]:=key
else begin
    stk1:=nil;
    stk2:=nil;
    if depth>0 then
        costl:=displace(index, depth-1, maxreal, nil, stk1, lenl);
    temp:=tab[index];
    tab[index]:=key;
    cost2:=displace(index, depth, costl, nil, stk2, len2);
    if (depth=0) or (cost2<costl) then rearrange(stk2, temp, len2)
    else begin
        tab[index]:=temp;
        rearrange(stk1, key, lenl)
    end;
    freenode(stk1,nil);
    freenode(stk2,nil)
end
end;
{ procedure insert }
function find(var tab: table; key: integer; var kickout: kicktab;
    var probes: integer): integer;

var
hashl, step, index, limit: integer;

begin
probes := 0;
hashl := key mod tablesiz;
step := key mod (tablesiz - 2) + 1;
if kickout[0] = kicksize + 1 then limit := kickout[-1]
else limit := kickout[0];
repeat
index := (hashl + probes * step) mod tablesiz;
probes := probes + 1
until (tab[index] = key) or (tab[index] = -1) or (probes = limit);
if tab[index] = key then find := index
else find := -1
end;
{ function find }

procedure delete(var tab: table; key: integer; var kickout: kicktab);

var
where, probes: integer;

begin
where := find(tab, key, kickout, probes);
if where = -1 then writeln(output, key, ' not found')
else begin
    tab[where] := -2;
    if probes > kicksize then probes := kicksize + 1;
    kickout[probes] := kickout[probes] - 1;
    while (kickout[0] <> 1) and (kickout[kickout[0]] = 0)
        do kickout[0] := kickout[0] - 1
end
end;
{ procedure delete }
The table was 98% filled. The following was done for \( DEPTH = 0,1,2,3,4,10 \): 4899 random integers were generated from a linear-congruential formula \( i := 3309 \times i + 885321 \pmod{4194304} \). The same 4899 keys were retrieved from the table to calculate retrieval performances. 4899 keys not in the table were generated and rejected to calculate rejection performances. This was repeated 18 times.

**TABLESIZE:** 4999  
**NUMBER OF KEYS INSERTED:** 4899  
**PERCENT OF TABLE FILLED:** 98  
**DEPTH OF RECURSION:** 0  
**PENALTY FUNCTION USED:** LINEAR

<table>
<thead>
<tr>
<th>TRIAL</th>
<th>LONGEST PROBE</th>
<th>MEAN PROBES</th>
<th>MEAN REJECTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>179</td>
<td>4.05429</td>
<td>48.17125</td>
</tr>
<tr>
<td>2</td>
<td>154</td>
<td>3.96142</td>
<td>47.80567</td>
</tr>
<tr>
<td>3</td>
<td>266</td>
<td>4.00204</td>
<td>49.21779</td>
</tr>
<tr>
<td>4</td>
<td>181</td>
<td>3.93957</td>
<td>48.19289</td>
</tr>
<tr>
<td>5</td>
<td>158</td>
<td>4.06940</td>
<td>48.14002</td>
</tr>
<tr>
<td>6</td>
<td>407</td>
<td>3.84057</td>
<td>48.72545</td>
</tr>
<tr>
<td>7</td>
<td>210</td>
<td>3.92712</td>
<td>49.15839</td>
</tr>
<tr>
<td>8</td>
<td>131</td>
<td>3.84057</td>
<td>45.55154</td>
</tr>
<tr>
<td>9</td>
<td>171</td>
<td>3.92733</td>
<td>47.68626</td>
</tr>
<tr>
<td>10</td>
<td>170</td>
<td>3.84751</td>
<td>48.78914</td>
</tr>
<tr>
<td>11</td>
<td>144</td>
<td>3.96366</td>
<td>47.68279</td>
</tr>
<tr>
<td>12</td>
<td>155</td>
<td>3.92998</td>
<td>47.07409</td>
</tr>
<tr>
<td>13</td>
<td>220</td>
<td>4.03184</td>
<td>50.97244</td>
</tr>
<tr>
<td>14</td>
<td>249</td>
<td>3.96529</td>
<td>48.20759</td>
</tr>
<tr>
<td>15</td>
<td>142</td>
<td>3.82384</td>
<td>45.78485</td>
</tr>
<tr>
<td>16</td>
<td>264</td>
<td>4.08389</td>
<td>50.35047</td>
</tr>
<tr>
<td>17</td>
<td>195</td>
<td>4.03123</td>
<td>48.02571</td>
</tr>
<tr>
<td>18</td>
<td>169</td>
<td>3.89957</td>
<td>48.48152</td>
</tr>
<tr>
<td><strong>MEAN</strong></td>
<td><strong>198.05</strong></td>
<td><strong>3.95217</strong></td>
<td><strong>48.22322</strong></td>
</tr>
</tbody>
</table>
TABLESIZE: 4999  
NUMBER OF KEYS INSERTED: 4899  
PERCENT OF TABLE FILLED: 98  
DEPTH OF RECURSION: 1  
PENALTY FUNCTION USED: LINEAR

<table>
<thead>
<tr>
<th>TRIAL</th>
<th>LONGEST PROBE</th>
<th>MEAN PROBES</th>
<th>MEAN REJECTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td>2.14696</td>
<td>16.64258</td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td>2.12880</td>
<td>13.07817</td>
</tr>
<tr>
<td>3</td>
<td>26</td>
<td>2.12247</td>
<td>20.30557</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td>2.13574</td>
<td>16.49112</td>
</tr>
<tr>
<td>5</td>
<td>21</td>
<td>2.11614</td>
<td>17.29679</td>
</tr>
<tr>
<td>6</td>
<td>18</td>
<td>2.16431</td>
<td>15.17922</td>
</tr>
<tr>
<td>7</td>
<td>20</td>
<td>2.13247</td>
<td>16.63400</td>
</tr>
<tr>
<td>8</td>
<td>25</td>
<td>2.14778</td>
<td>19.72320</td>
</tr>
<tr>
<td>9</td>
<td>20</td>
<td>2.11859</td>
<td>16.67462</td>
</tr>
<tr>
<td>10</td>
<td>20</td>
<td>2.12267</td>
<td>16.68748</td>
</tr>
<tr>
<td>11</td>
<td>20</td>
<td>2.14227</td>
<td>16.52582</td>
</tr>
<tr>
<td>12</td>
<td>21</td>
<td>2.14206</td>
<td>17.05817</td>
</tr>
<tr>
<td>13</td>
<td>19</td>
<td>2.13655</td>
<td>15.84547</td>
</tr>
<tr>
<td>14</td>
<td>19</td>
<td>2.12288</td>
<td>15.99224</td>
</tr>
<tr>
<td>15</td>
<td>27</td>
<td>2.15554</td>
<td>20.89773</td>
</tr>
<tr>
<td>16</td>
<td>19</td>
<td>2.15819</td>
<td>16.01285</td>
</tr>
<tr>
<td>17</td>
<td>21</td>
<td>2.15839</td>
<td>17.33210</td>
</tr>
<tr>
<td>18</td>
<td>18</td>
<td>2.14472</td>
<td>15.43233</td>
</tr>
<tr>
<td>MEAN</td>
<td>20.50</td>
<td>2.13870</td>
<td>16.87830</td>
</tr>
</tbody>
</table>
TABLESIZE: 4999
NUMBER OF KEYS INSERTED: 4899
PERCENT OF TABLE FILLED: 98
DEPTH OF RECURSION: 2
PENALTY FUNCTION USED: LINEAR

<table>
<thead>
<tr>
<th>TRIAL</th>
<th>LONGEST PROBE</th>
<th>MEAN PROBES</th>
<th>MEAN REJECTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>13</td>
<td>1.89855</td>
<td>11.64564</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>1.90936</td>
<td>10.04123</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>1.90304</td>
<td>10.67605</td>
</tr>
<tr>
<td>4</td>
<td>15</td>
<td>1.89895</td>
<td>13.08532</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>1.92284</td>
<td>10.01408</td>
</tr>
<tr>
<td>6</td>
<td>13</td>
<td>1.89548</td>
<td>11.58522</td>
</tr>
<tr>
<td>7</td>
<td>11</td>
<td>1.91181</td>
<td>9.95264</td>
</tr>
<tr>
<td>8</td>
<td>12</td>
<td>1.91202</td>
<td>10.74402</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>1.92304</td>
<td>9.93794</td>
</tr>
<tr>
<td>10</td>
<td>14</td>
<td>1.91712</td>
<td>12.29271</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>1.89712</td>
<td>10.78138</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>1.87711</td>
<td>11.53031</td>
</tr>
<tr>
<td>13</td>
<td>12</td>
<td>1.91855</td>
<td>10.76546</td>
</tr>
<tr>
<td>14</td>
<td>13</td>
<td>1.91835</td>
<td>11.58236</td>
</tr>
<tr>
<td>15</td>
<td>11</td>
<td>1.92876</td>
<td>10.05409</td>
</tr>
<tr>
<td>16</td>
<td>14</td>
<td>1.89875</td>
<td>12.42600</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>1.91345</td>
<td>9.99183</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>1.90814</td>
<td>10.75648</td>
</tr>
</tbody>
</table>

MEAN 12.27 1.90847 10.99237
TABLESIZE: 4999
NUMBER OF KEYS INSERTED: 4899
PERCENT OF TABLE FILLED: 98
DEPTH OF RECURSION: 3
PENALTY FUNCTION USED: LINEAR

<table>
<thead>
<tr>
<th>TRIAL</th>
<th>LONGEST PROBE</th>
<th>MEAN PROBES</th>
<th>MEAN REJECTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td>1.83710</td>
<td>9.95407</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>1.84486</td>
<td>8.35211</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>1.83445</td>
<td>9.95733</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td>1.81812</td>
<td>10.00489</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>1.81771</td>
<td>9.12349</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>1.80608</td>
<td>8.28373</td>
</tr>
<tr>
<td>7</td>
<td>10</td>
<td>1.84323</td>
<td>9.17962</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>1.81853</td>
<td>9.17350</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>1.82629</td>
<td>9.95611</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>1.84996</td>
<td>9.94284</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>1.85772</td>
<td>9.97958</td>
</tr>
<tr>
<td>12</td>
<td>9</td>
<td>1.83996</td>
<td>8.37170</td>
</tr>
<tr>
<td>13</td>
<td>10</td>
<td>1.81833</td>
<td>9.16084</td>
</tr>
<tr>
<td>14</td>
<td>11</td>
<td>1.83568</td>
<td>9.92794</td>
</tr>
<tr>
<td>15</td>
<td>9</td>
<td>1.81853</td>
<td>8.28965</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>1.83200</td>
<td>9.14064</td>
</tr>
<tr>
<td>17</td>
<td>10</td>
<td>1.81975</td>
<td>9.17228</td>
</tr>
<tr>
<td>18</td>
<td>8</td>
<td>1.81363</td>
<td>7.42416</td>
</tr>
<tr>
<td>MEAN</td>
<td>10.05</td>
<td>1.82955</td>
<td>9.18858</td>
</tr>
</tbody>
</table>
TABLESIZE: 4999  
NUMBER OF KEYS INSERTED: 4899  
PERCENT OF TABLE FILLED: 98  
DEPTH OF RECURSION: 4  
PENALTY FUNCTION USED: LINEAR

<table>
<thead>
<tr>
<th>TRIAL</th>
<th>LONGEST PROBE</th>
<th>MEAN PROBES</th>
<th>MEAN REJECTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>1.80894</td>
<td>7.42559</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>1.79975</td>
<td>9.11900</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>1.77873</td>
<td>7.45621</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>1.80366</td>
<td>8.31006</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>1.79199</td>
<td>8.26495</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>1.80261</td>
<td>9.11696</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>1.78362</td>
<td>7.44886</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>1.81036</td>
<td>8.28026</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>1.79159</td>
<td>9.16411</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>1.80159</td>
<td>9.17452</td>
</tr>
<tr>
<td>11</td>
<td>9</td>
<td>1.80547</td>
<td>8.28271</td>
</tr>
<tr>
<td>12</td>
<td>9</td>
<td>1.80016</td>
<td>8.27434</td>
</tr>
<tr>
<td>13</td>
<td>9</td>
<td>1.79056</td>
<td>8.28699</td>
</tr>
<tr>
<td>14</td>
<td>9</td>
<td>1.78648</td>
<td>8.29067</td>
</tr>
<tr>
<td>15</td>
<td>8</td>
<td>1.79567</td>
<td>7.41192</td>
</tr>
<tr>
<td>16</td>
<td>8</td>
<td>1.80159</td>
<td>7.43702</td>
</tr>
<tr>
<td>17</td>
<td>8</td>
<td>1.82159</td>
<td>7.46540</td>
</tr>
<tr>
<td>18</td>
<td>8</td>
<td>1.79465</td>
<td>7.47846</td>
</tr>
</tbody>
</table>

MEAN 8.83 1.79809 8.14933
TABLESIZE: 4999  
NUMBER OF KEYS INSERTED: 4899  
PERCENT OF TABLE FILLED: 98  
DEPTH OF RECURSION: 10  
PENALTY FUNCTION USED: LINEAR

<table>
<thead>
<tr>
<th>TRIAL</th>
<th>LONGEST PROBE</th>
<th>MEAN PROBES</th>
<th>MEAN REJECTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>1.75199</td>
<td>6.62482</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>1.76546</td>
<td>6.58685</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>1.76484</td>
<td>7.40579</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>1.75688</td>
<td>6.56195</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>1.77709</td>
<td>6.59951</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>1.74321</td>
<td>6.59420</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>1.76811</td>
<td>6.60318</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>1.76219</td>
<td>6.59154</td>
</tr>
<tr>
<td>9</td>
<td>7</td>
<td>1.77342</td>
<td>6.56031</td>
</tr>
<tr>
<td>10</td>
<td>7</td>
<td>1.76260</td>
<td>6.58562</td>
</tr>
<tr>
<td>11</td>
<td>7</td>
<td>1.75750</td>
<td>6.62318</td>
</tr>
<tr>
<td>12</td>
<td>7</td>
<td>1.74239</td>
<td>6.60073</td>
</tr>
<tr>
<td>13</td>
<td>7</td>
<td>1.76423</td>
<td>6.58215</td>
</tr>
<tr>
<td>14</td>
<td>7</td>
<td>1.76832</td>
<td>6.58603</td>
</tr>
<tr>
<td>15</td>
<td>7</td>
<td>1.77995</td>
<td>6.62237</td>
</tr>
<tr>
<td>16</td>
<td>7</td>
<td>1.75423</td>
<td>6.59889</td>
</tr>
<tr>
<td>17</td>
<td>8</td>
<td>1.75729</td>
<td>7.48867</td>
</tr>
<tr>
<td>18</td>
<td>7</td>
<td>1.76382</td>
<td>6.58154</td>
</tr>
<tr>
<td>MEAN</td>
<td>7.11</td>
<td>1.76186</td>
<td>6.68874</td>
</tr>
</tbody>
</table>
The table was filled to 98% loading. All the keys in the table were retrieved to calculate retrieval performances. An equal number of keys not in the table were rejected to calculate rejection performances. 49% of the keys were deleted. Retrieval and rejection statistics were again calculated. The table was filled back to a 98% loading. Retrieval and rejection statistics were again calculated. These are the mean results of 18 trials:

TABLESIZE: 4999
DEPTH OF RECURSION: 4
PENALTY FUNCTION USED: LINEAR

<table>
<thead>
<tr>
<th>ACTION</th>
<th># OF KEYS IN TABLE</th>
<th>% TABLE LOADED</th>
<th>LONGEST PROBE</th>
<th>MEAN PROBES TO ACCEPT/REJECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSERT</td>
<td>4900</td>
<td>97.9</td>
<td>9.06</td>
<td>1.80268 AC</td>
</tr>
<tr>
<td>LOOKUP</td>
<td>4900</td>
<td>97.9</td>
<td>9.06</td>
<td>1.80268 AC</td>
</tr>
<tr>
<td>LOOKUP-4900</td>
<td>4900</td>
<td>97.9</td>
<td>9.06</td>
<td>1.80268 AC</td>
</tr>
<tr>
<td>DELETE</td>
<td>2450</td>
<td>48.9</td>
<td>8.61</td>
<td>8.35276 RJ</td>
</tr>
<tr>
<td>LOOKUP</td>
<td>2450</td>
<td>48.9</td>
<td>8.61</td>
<td>8.35276 RJ</td>
</tr>
<tr>
<td>LOOKUP-2450</td>
<td>2450</td>
<td>48.9</td>
<td>8.61</td>
<td>8.35276 RJ</td>
</tr>
<tr>
<td>INSERT</td>
<td>2450</td>
<td>48.9</td>
<td>8.61</td>
<td>8.35276 RJ</td>
</tr>
<tr>
<td>LOOKUP</td>
<td>2450</td>
<td>48.9</td>
<td>8.61</td>
<td>8.35276 RJ</td>
</tr>
<tr>
<td>LOOKUP-2450</td>
<td>2450</td>
<td>48.9</td>
<td>8.61</td>
<td>8.35276 RJ</td>
</tr>
</tbody>
</table>

AVERAGES OF EIGHTEEN TRIALS

-27-
ACCESS FUNCTIONS FOR PACKED SCATTER TABLES

Three PASCAL access routines are given for packed scatter tables. INSERT packs tables of integer keys; FIND retrieves the keys; DELETE deletes the keys. While the routines currently access integer keys (for use in performance testing with pseudo-random integers), they can easily be converted to access other data types -- character strings in a symbol table, for example. To enhance portability, the code is straightforward PASCAL without any input/output capabilities. Appendices contain specific comments on the routines, listings, and sample results.