PARTIAL SUPPORT USING A “PARTIAL SUPPORT TREE”

Frans Coenen

Department of Computer Science
The University of Liverpool
Liverpool, L69 ZF, UK.
email: frans,@csc.liv.ac.uk
tel: +44 (0)151 794 3698
fax: +44 (0)151 794 3715

1 INTRODUCTION

In [4] an algorithm for producing a linked list of structures each of which contained the code and support for unique rows in a transaction table was described. The suggested advantage was that this was a way of determining association rules using only one pass of the database but without requiring the entire database to be stored in memory. In this paper we present a second algorithm which has a similar aim except in this case the partial supports are contained in a tree structure. As before the nodes in this tree represent unique rows in the table, however in this case the nodes are arranged lexiconically and according to whether one is a subset of another. A child node is a superset of its parent node while a sibling nodes are ordered lexiconically. Example are given later in the text.

The tree commences with a single node which has a *sibling branch* and a *child branch*. New nodes are placed into the tree by traversing the tree until an appropriate location is found. This is achieved using the following “rules” (where R represents the bit pattern of the row in the table currently under consideration and B a bit pattern attached to an existing node in the tree):

Rule 1: If $R = B$ (i.e. an identical node is found) simply increment the support associated with the node and return.

Rule 2: If $R < B$ and $R \subset B$ create a new node for R and place the existing node associated with B on the new node's child branch. The new node is then placed either as a new root node, or is added to the child or sibling branch of a previously investigated node (see examples given below).

Rule 3: If $R < B$ and $R \not\subset B$ create a new node for R and place the existing node associated with B on the new node's sibling branch. The new node is then placed either as a new root node, or is added to the child or sibling branch of a previously investigated node (again, see examples given below).

Rule 4: If $R > B$ and $R \supset B$ then:

- If node associated with B is a leaf node create a new node for R and add this to the existing node's child branch.
- Otherwise proceed down child branch and repeat.

Rule 5: If $R > B$ and $R\perp \supset B$ then:

- If node associated with B is a leaf node create a new node for R and add this to the existing node's sibling branch.
- Otherwise proceed down sibling branch and repeat.

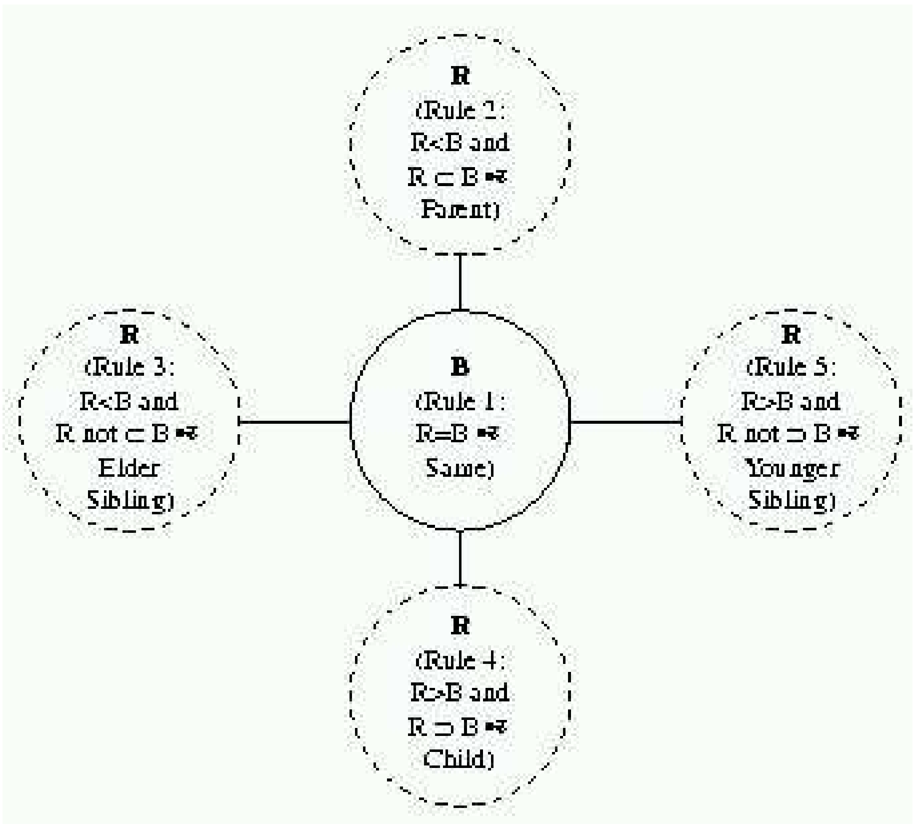

Figure 1: Possible relationships between R and B

Thus given a bit Pattern R and a bit pattern B associated with an existing node their are five possible outcomes. The possible relationships between R and B are given in Figure 1. Rules 4 and 5 above are fairly straight forward, however to achieve a better understanding of rules 2 to 3, and to supply some additional considerations, it is useful to give some illustrations of their operation. This is done in the following two sub-sections.

1.1 Rule 2: $R < B$ and $R \subset B$

The rule is entered either with the start node or as part of a search down a child or sibling branch. Consequently we have three possible outcomes (Figure 2).

However, there is an additional test we must make. Consider the tree given in Figure 3(a). If we wish to add node B to the tree, applying the above rules gives the result shown in Figure 3(b). This is the desired result. However if we now consider the tree given in Figure 3(c) and add node B to this tree, through the application of the above rules we get the result shown in Figure 3(d). This is not the desired result. To resolve this situation we need to check the sibling branch of BC and if this is not a superset of the new node move it up so that it becomes a sibling of the new node.

This still does not necessarily resolve the situation entirely because not all node in the sibling branch are necessarily all supersets or not supersets of the node R. For example in Figure 4(a) we commence with a five node tree structure, we than add the node A with the result as shown in Figure 4(b). Node AC is in the right place, however nodes B and C need to be moved up so that they become siblings of A. This task is not too computationally expensive in that (where some nodes are to be moved up and others should remain) we only need to find the3 point in the sibling branch where to "split the tree".

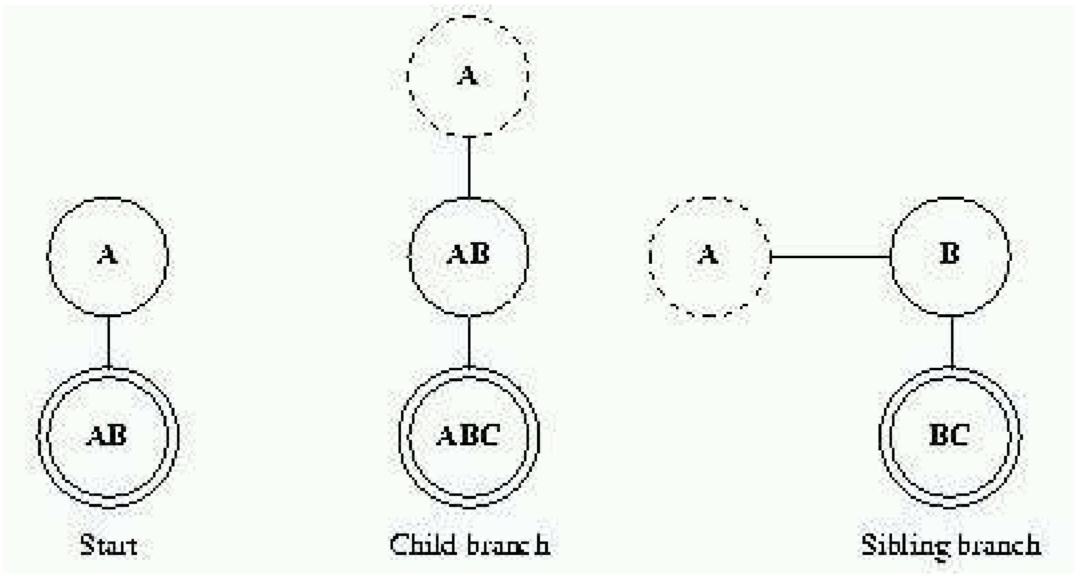

Figure 2: Possible outcomes from the application of Rule 2 (R single circle, B double circle, other nodes dashed circle)

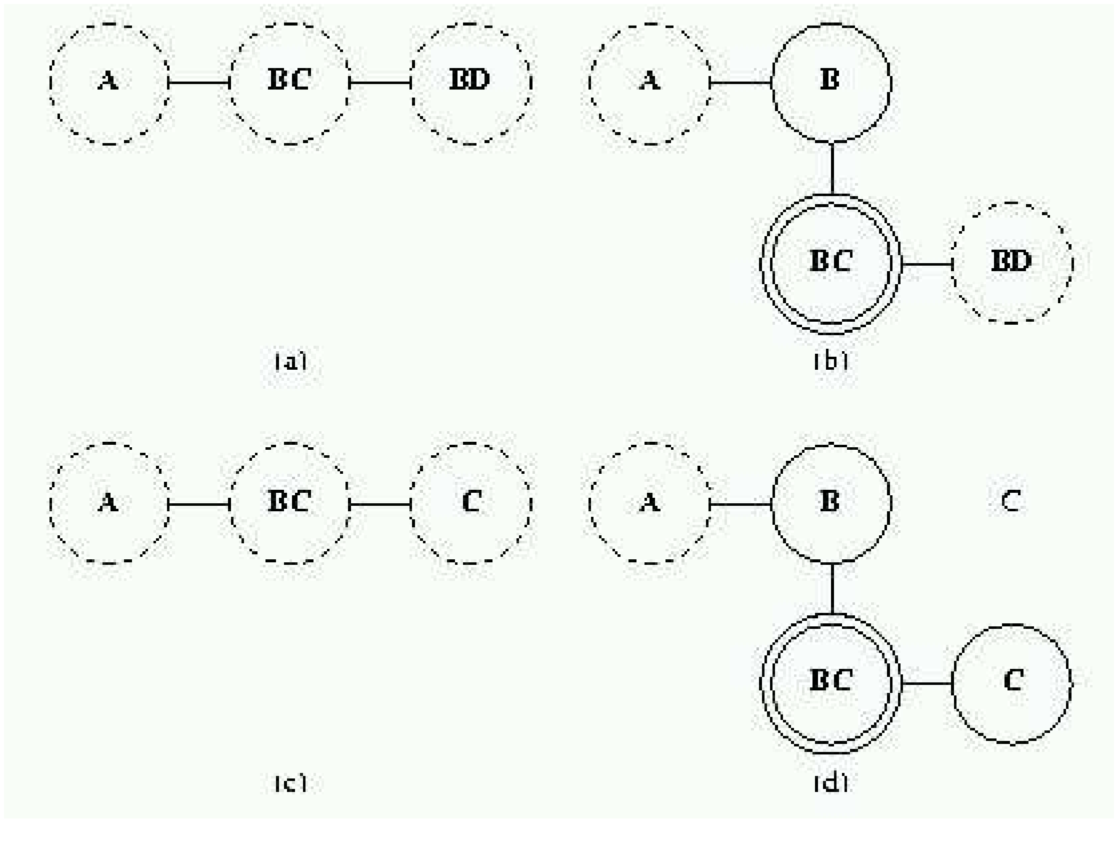

Figure 3: Additional considerations when applying Rule 2 (R single circle, B double circle, correct position indicated for C, other nodes dashed circle)

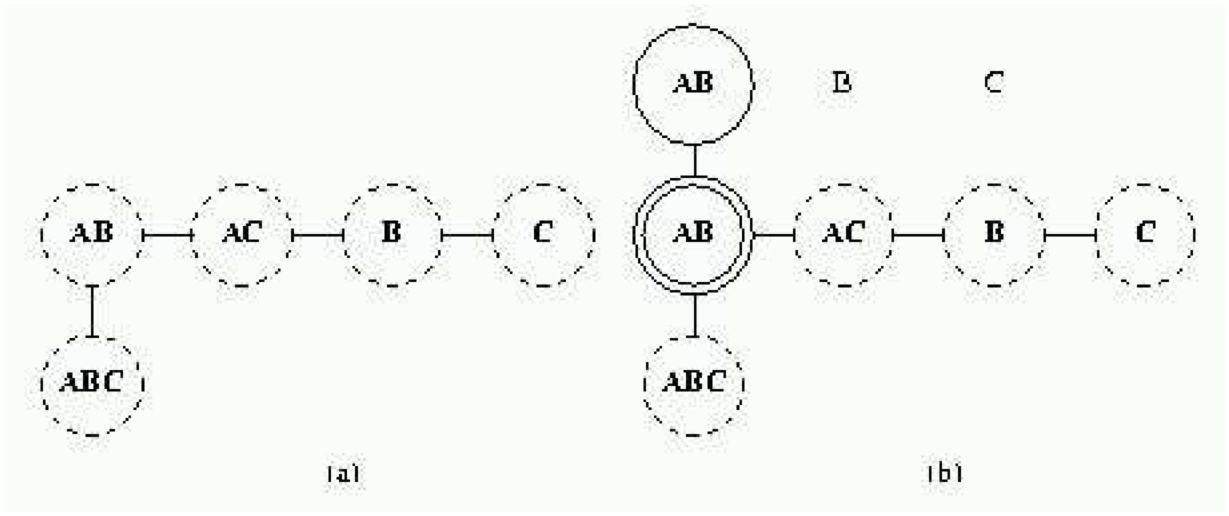

Figure 4: Further considerations when applying Rule 2 (R single circle, B double circle, correct position indicated for C, other nodes dashed circle)

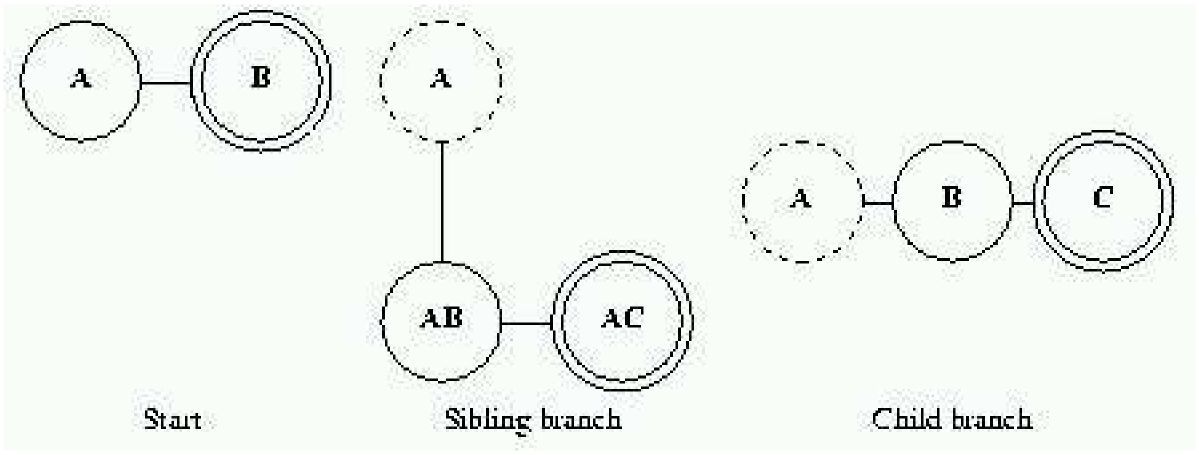

Figure 5: Possible outcomes from the application of Rule 3 (R single circle, B double circle, other nodes dashed circle)

1.2 Rule 3: $R < B$ and $R \not\subset B$

As with Rule 2 described above this rule is entered either with the start node or as part of a search down a child or sibling branch. Consequently we have three possible outcomes as shown in Figure 5. In this case there are no special considerations that need to be taken into account when applying the rule (unlike Rule 2).

2 THE DATA STRUCTURE

As in early work described in [2, 3] use is again made of bit codes to represent the bit patterns, in particular we use the array encoding described in [3]. The data structure used is therefore defined (in C) as follows:

```
typedef struct partSupport{
        int code*;
        int support;
        struct partSupport *siblingPtr;
        struct partSupport *childPtr;
        } PARTSUPPORT, PARTSUPPORTPTR;
```

This definition is stored in a header file called *support.h*.

3 INPUT

The algorithm is designed to be linked in to a “main” program. For example:

```
#include "support.h"
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

/* ------ FUNCTION PROTOTYPES ------ */

extern void partailSupport(char *);
extern void outputPartSupportTree(void);

/* ---------------------- */
/*                        */
/*         MAIN           */
/*                        */
/* ---------------------- */

void main(int argc, char *argv[])
{
/* Check Input. */

if (argc < 2 ) {
        printf("INPUT ERROR - no file name.\n");
        exit(1);
        }

/* Create partial support file. */

partialSupport(argv[1]);

/* Output result (testing only). */

outputPartSupportTree();
}
```

Where the function *partialSupport* instigates the calculation and storage of the partial supports, and the *outputPartSupportLinkedList* procedure outputs the result (the latter used for testing purposes only).

The above is invoked as follows:

```
./mainpro <INPUTFILE>
```

where the input (as before) is in the form of a file, for example ($n = 8$ and $m = 10$):

```
 Row   Interpre-  Bit
in DB   tation    Code
011      BC        6
110      AB        3
011      BC        6
101      AC        5
100      A         1
010      B         2
001      C         4
001      C         4
111      ABC       7
101      AC        5
```

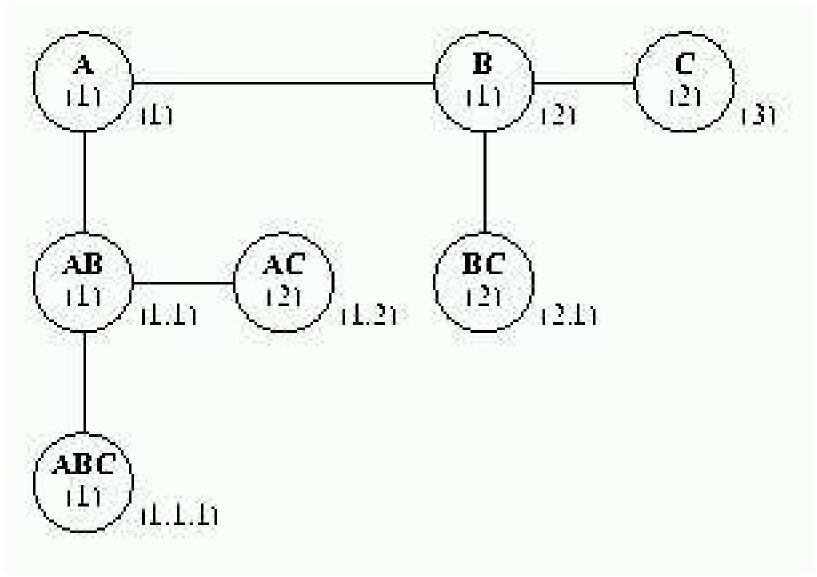

Figure 6: Partial support tree

Given this input we can expect (using the output routines used for testing) output of the form:

```
(1) 1 support = 1
(1.1) 3 support = 1
(1.1.1) 7 support = 1
(1.2) 5 support = 2
(2) 2 support = 1
(2.1) 6 support = 2
(3) 4 support = 2
```

where the bracketted sequence of integers represents a node identifier and the following integer the bit code associated with each row. This would produce a tree structure a shown in Figure 6. Note the node numbering convention used (again for testing purposes only).

4 ALGORITHM OVERVIEW

Broadly the algorithm operates as follows:

- Read first line in the table describing a transaction.
- From this line determine the number of columns/attributes in the table and then use this information to generate the first structure in the linked list.
- Read the remainder of the table in an iterative manner. On each iteration:
  - If the row is already contained in the tree structure increment the support variable in the appropriate structure.
  - Otherwise create a new node in the structure (according to the rules identified above).

5 DETAILED DESIGN OF ALGORITHM

The partial support algorithm is implemented in C using the following functions/procedures:

1. partialSupport

2. processInput
3. addSupport
4. addSupport2
5. checkSiblingBranch
6. checkCodes
7. equalityCheck
8. subsetCheck
9. beforeCheck
10. generateBitPattern
11. createStructure
12. outputPartSupportLinkedList
13. outputPartSupport1
14. outputPartSupport2

The *partialSupport* procedure is used to open and close the input file and instigate the processing and is identical to that described in [4]. The processing is actually carried out by the *processInput* procedure which loops through the table. The *addSupport* and *addSupport*2 procedures are used to instigate the calculation of the bit patterns (codes) associated with each line in the table and either (a) create a new structure in the tree or (b) update the support for an existing structure. Code comparisons are carried out by the *checkCodes* function which in turn accesses the *equalityCheck*, *subsetCheck* and *beforeCheck* functions. Bit patterns are generated by the *generateBitPattern* function which is roughly the same as the function of the same name described in [4]. Where necessary new structures are created by the *createStructure* function. Finally the *outputPartSupportTreet*, *outputPartSupport*1 and *outputPartSupport*2 procedures are used to output the resulting linked list of partial supports and are used for testing purposes only.

The code includes four global variables:

NAME	TYPE	DESCRIPTION
startPtr	PARTSUPPORTPTR	Global pointer to start of linked list structure (the data type PARTSUPPORTPTR is defined in support.h).
arraySize	Integer	Global variable indicating size of bit code storage array.
totalRows	Integer	Global variable in which the total number of rows in the input table is stored.
totalCols	Integer	Global variable in which the total number of columns in the input table is stored.

In the following sub-sections the detailed design of the *addSupport*, *addSupport*2, *checkSiblingBranch*, *checkCodes*, *equalityCheck*, *subsetCheck*, *beforeCheck* and *createStructure* functions/procedures are described. Details concerning the *processInput* procedure and *generateBitPattern* functions can be found in [4]. The output procedures are not described as these are not an integral part of the algorithm. The designs are given in the form of Nassi-Shneiderman charts. Readers who are not interested in the implementational details of the algorithm can skip this section and go straight to Section 5 — "Analysis". All code is listed in Appendix A and test scenarios in Appendix B.

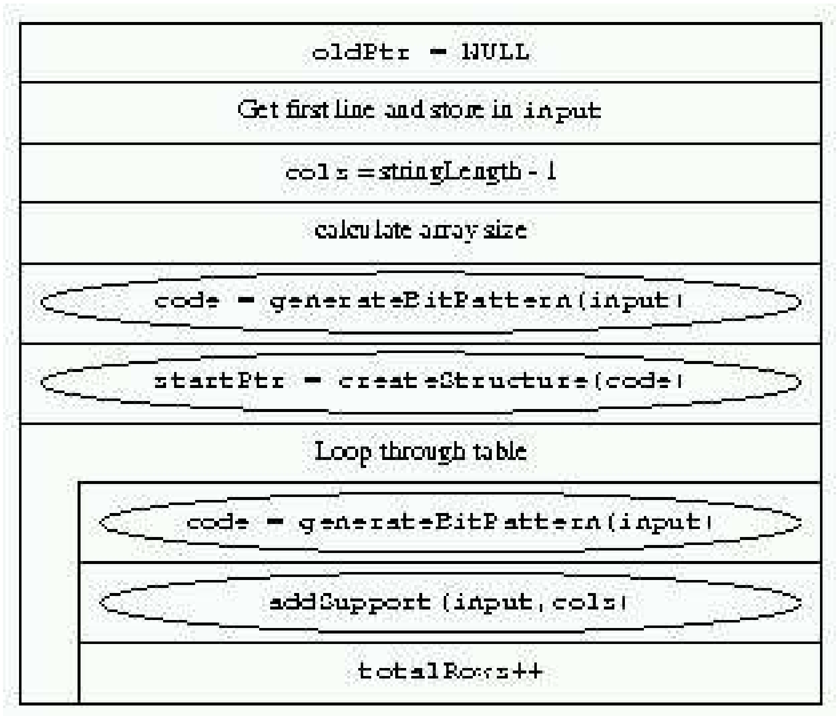
oldPtr = NULL
Get first line and store in input
cols = stringLength - 1
calculate array size
code = generateBitPattern(input)
startPtr = createStructure(code)
Loop through table
code = generateBitPattern(input)
addSupport(input, cols)
totalRows++

Figure 7: Nassi-Shneiderman chart for *processInput* procedure

5.1 PROCESS INPUT

The *processInput* procedure is the principal procedure for calculating support values. The procedure operates as follows:

- Read first line in table and from this line determine the number of columns in the table and consequently the size of the unsigned integer array required to represent the bit codes.
- Create a structure to represent the first line (the initial root node for the tree).
- Process the rest of the table.

The procedure includes the following data items:

NAME	TYPE	DESCRIPTION
in_file	FILE pointer	Formal parameter, pointer to current location in input file.
input	512 character string	Local variable in which current input line is stored as a string (32 is the maximum size of the string).
cols	Integer	Local variable containing the number of columns in the input table.
code	Unsigned integer array pointer	Local variable to hold bit codes.
oldPtr	PARTSUPPORTPTR	Local variable. Pinter to temporarily hold previous location in data structure when calling *addSupport* procedure. This points to NULL initially.

A detailed design for the *processInput* procedure is presented in Figure 7.

5.2 ADD SUPPORT

The *addSupport* procedure is called for each line in the table. This is then processed according to the rules presented in Section 1 above with respect to the nodes in the tree structure as created "sofar". There are five alternatives:

- The current node and the row are the same therefore increment the support associated with the current node (Rule 1).
- The new node is a parent node of the current node (Rule 2).
- The new node is an elder sibling of the current node (Rule 3).
- The new node is located somewhere on the child branch of the current node (Rule 4).
- The new node is located somewhere on the sibling branch of the current node (Rule 5).

Selections are made according to the result returned by the *checkCodes* function (see below). The procedure includes the following data items:

NAME	TYPE	DESCRIPTION
flag	Integer	Formal parameter describing the type of branch currently under consideration: 0 = root, 1 - child, 2 = sibling.
linkPtr	PARTSUPPORTPTR	Formal parameter to control loop through linked list structure.
oldPtr	PARTSUPPORTPTR	Formal parameter to hold temporary pointers to structures.
rowCode	Unsigned integer array Pointer	Formal parameter describing the code/bit pattern for the current line in the table (*inputString*)
newPtr	PARTSUPPORTPTR	Local variable to hold pointer to newly created structure (if any).
newSiblingPtr	PARTSUPPORTPTR	Local variable to hold pointer to new sibling pointer where required.

A detailed design for the *addSupport* procedure is presented in Figure 8.

5.3 ADD SUPPORT 2

The *addSupport* procedure is called where Rule 2 or Rule 3 is implemented, i.e. where the code associated with a row in the table under consideration is "before" the node in the tree currently under consideration. In this case the row must be inserted into the tree so that the tree node is a sibling or a child of the row as appropriate, and the row itself represents either a new root node or it is attached as a child or sibling to a previous node. The latter is implemented according to a flag set to 0 for a new root node, 1 for a child branch and 2 for a sibling branch. The procedure includes the following data items:

NAME	TYPE	DESCRIPTION
flag	Integer	Formal parameter describing the type of branch currently under consideration: 0 = root, 1 - child, 2 = sibling.
newPtr	PARTSUPPORTPTR	Formal parameter to hold pointer to newly created structure (if any).
oldPtr	PARTSUPPORTPTR	Formal parameter to hold temporary pointers to structures.

A detailed design for the *addSupport*2 procedure is presented in Figure 9.

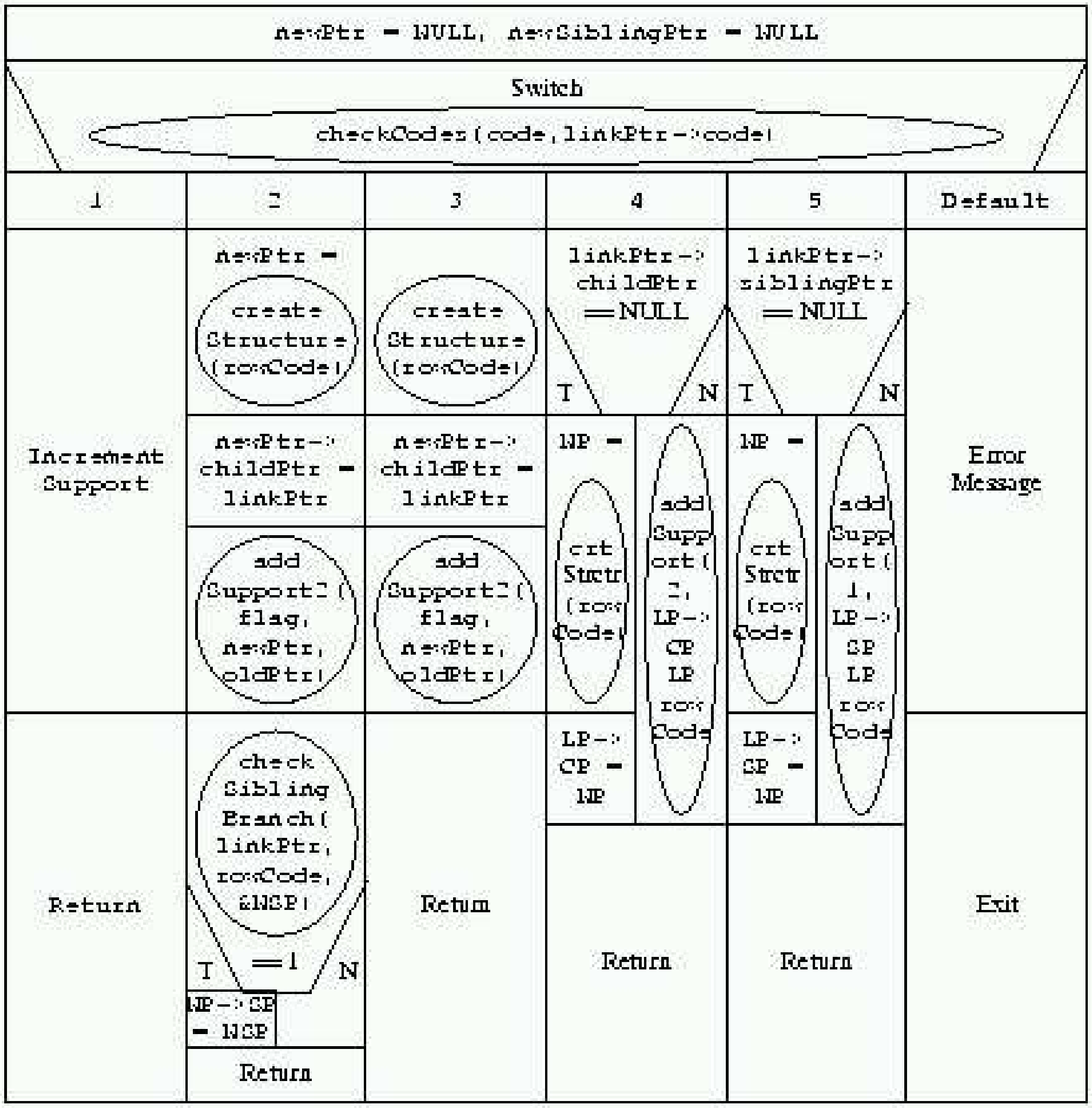
newPtr = NULL, newSiblingPtr = NULL
Switch
checkCodes(code, linkPtr->code)
1
2
3
4
5
Default
newPtr =
create Structure (rowCode)
create Structure (rowCode)
linkPtr-> childPtr ==NULL
linkPtr-> siblingPtr ==NULL
T
N
T
N
Increment Support
newPtr-> childPtr = linkPtr
newPtr-> childPtr = linkPtr
NP =
crt Structr (rowCode)
add Support (2, LP->CP LP rowCode)
NP =
crt Structr (rowCode)
add Support (1, LP->SP LP rowCode)
Error Message
add Support2 (flag, newPtr, oldPtr)
add Support2 (flag, newPtr, oldPtr)
LP-> CP = NP
LP-> SP = NP
Return
check Sibling Branch(linkPtr, rowCode, &NSP)
==1
T
N
NP->SP = NSP
Return
Return
Return
Return
Exit

Figure 8: Nassi-Shneiderman chart for *addSupport* procedure ($LP = linkPtr$, $NP = newPtr$, $CP = childPtr$, $SP = siblingPtr$ and $NSP = newSiblingPtr$) (e.g. $OP- > N$) and $LP = linkPtr$)

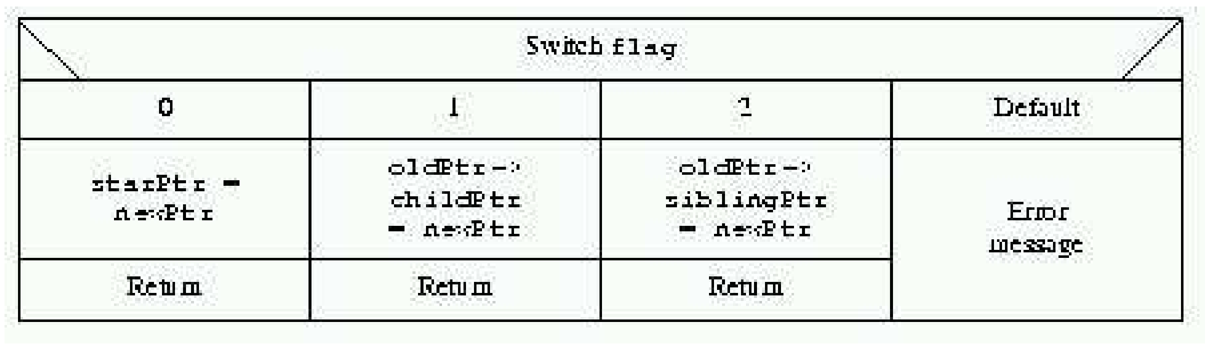

Figure 9: Nassi-Shneiderman chart for *addSupport*2 procedure

5.4 CHECK SIBLING BRANCH

The *checkSiblingBranch* function is called in the special case where Rule 2 has been applied (new node inserted as a parent to an existing node) and it is possible that some nodes in the sibling branch of the existing node in the tree may have to be moved to become siblings of the newly inserted node. The function returns 1 if adjustment is required, and 0 otherwise. Only those nodes in the sibling branch (as illustrated in Sub-section 1.1) which are not supersets of the newly inserted node need to be moved. As the nodes are ordered any nodes that do need to moved will be stored in sequence after those nodes that do not need to be moved. There are four possibilities:

1. Sibling branch is empty therefore no nodes can be move (return 0).
2. The first node in the sibling branch is not a superset of new node therefor move the entire branch to become a sibling branch of the new node (return 1).
3. All nodes in sibling branch are supersets of row there for move none of them (return 0).
4. First section of the sibling branch contains one or more nodes that need to be moved up, will the remainder comprises nodes which are supersets of the new node. Therefore move those that are not supersets up to become siblings of the new node (return 1).

Selections are made using a called to the *subsetCheck* function described below. Where some nodes are to be moved up and some are not we need to find the “cut-off” point in the branch where the “change-over” takes place, i.e. we need to iterate along the branch. The function includes the following data items

NAME	TYPE	DESCRIPTION
linkPtr	PARTSUPPORTPTR	Formal parameter to control loop through linked list structure.
rowCode	Unsigned integer array Pointer	Formal parameter describing the code/bit pattern for the current line in the table (*inputString*).
newSiblingPtr	PARTSUPPORTPTR	Formal parameter to hold pointer to new sibling pointer where required.
localLinkPtr	PARTSUPPORTPTR	Local variable to hold pointer to location in tree structure during processing.
markergPtr	PARTSUPPORTPTR	Local variable to hold pointer to act as a place marker in tree as it is processed.

A detailed design for the *checkSiblingBranch* procedure is presented in Figure 10.

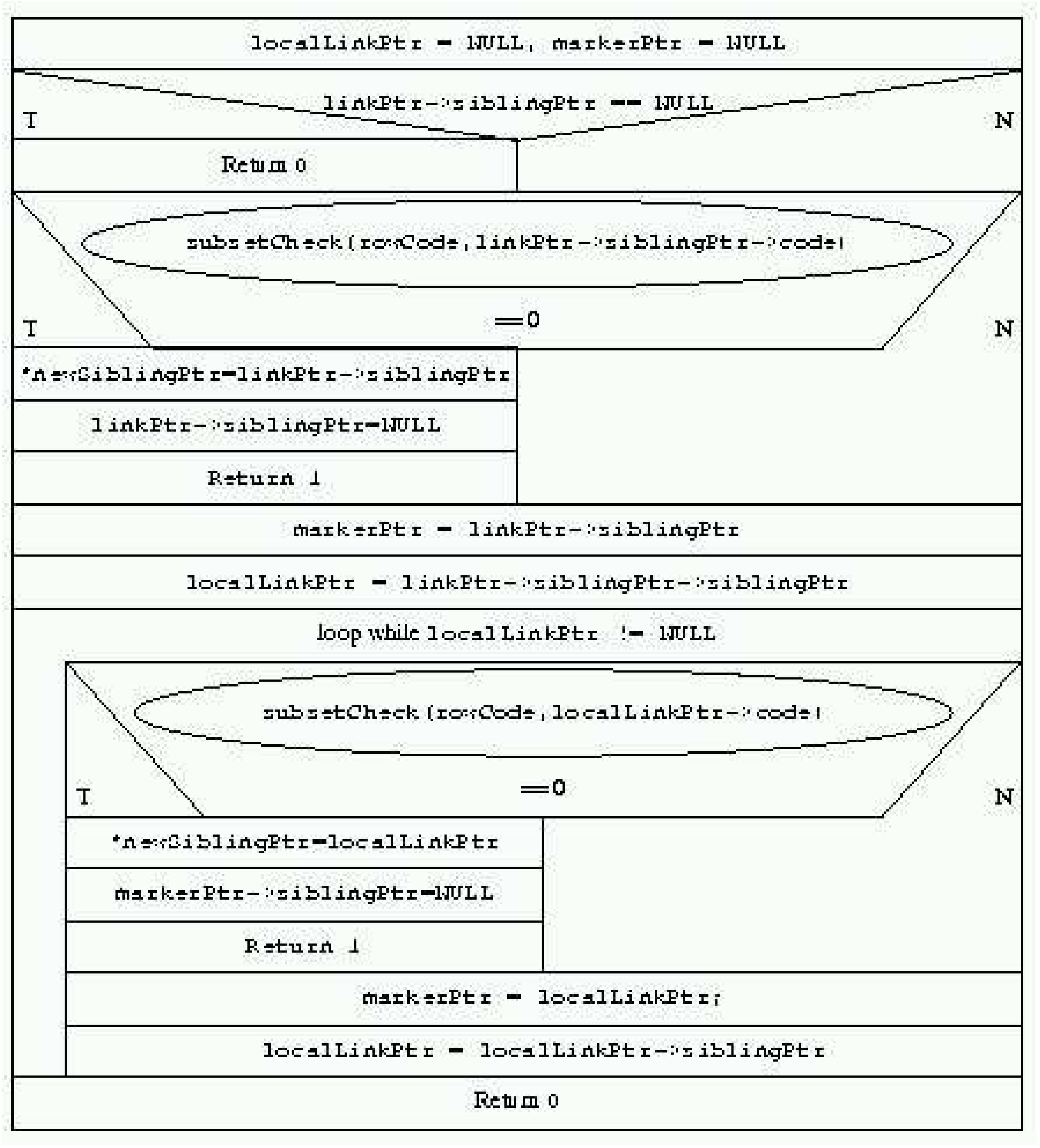
localLinkPtr = NULL, markerPtr = NULL
linkPtr->siblingPtr == NULL
T
N
Return 0
subsetCheck(rowCode,linkPtr->siblingPtr->code)
==0
T
N
*newSiblingPtr=linkPtr->siblingPtr
linkPtr->siblingPtr=NULL
Return 1
markerPtr = linkPtr->siblingPtr
localLinkPtr = linkPtr->siblingPtr->siblingPtr
loop while localLinkPtr != NULL
subsetCheck(rowCode,localLinkPtr->code)
==0
T
N
*newSiblingPtr=localLinkPtr
markerPtr->siblingPtr=NULL
Return 1
markerPtr = localLinkPtr;
localLinkPtr = localLinkPtr->siblingPtr
Return 0

Figure 10: Nassi-Shneiderman chart for *checkSiblingBranch* function)

5.5 CHECK CODES

The *checkCodes* function is used to compare a current bit pattern/code to that held in a particular structure contained in the tree. As described in section 1 there are five possibilities:

- Same (return 1).
- Before and subset (return 2).
- Before and not subset(return 3).
- After and superset (return 4).
- After and not superset(return 5).

Specific checks are carried out by the *equalityCheck*, *beforeCheck* and *subsetCheck* functions described below. The *checkCodes* function uses two data items as follows:

NAME	TYPE	DESCRIPTION
rowCode	Unsigned integer pointer	Formal parameter giving bit code for current line in table.
currentCode	Unsigned integer pointer	Formal parameter giving bit code for current structure.

A detailed design for the *checkCodes* procedure is presented in Figure 11.

5.6 EQUALITY CHECK

Function to determine whether to codes are identical or not. Returns 1 if so, and 0 otherwise. The function uses three data items:

NAME	TYPE	DESCRIPTION
*code*1	Unsigned integer pointer	Formal parameter for first bit code.
*code*2	Unsigned integer pointer	Formal parameter for second bit code.
index	Integer	Local variable index with which to step through codes.

A detailed design for the *equalityCheck* procedure is presented in Figure 12.

5.7 BEFORE CHECK

Similar function to the *equlityCheck* function described above, but returns 1 if row is "before" current node, and 0 otherwise. There is however an added complication in that by "before" we mean lexiconically before and not numerically before. For example AB is before B, although the code associated with AB (3) is numerically after the code associated with B (2). Processing is carried out using a case statement the selector for which is made up of the remainder of two divisions by 2 (the second multiplied by 2). Thus:

$$selector = (number1\ rem\ 2)\ +\ 2\,(number2\ rem\ 2)$$

Thus if $number1 = 3$ and $number2 = 2$ then:

$$selector = (3\ rem\ 2)\ +\ 2\,(2\ rem\ 2) = 1 + 2\,(0) = 1$$

This is then used to test the least significant bit of each code in an iterative manner during which the numbers are adjusted so that a new most "significant" bit is tested on each loop. As a result of each test there are four possibilities:

selector = 0 Both codes have a 0 bit n therefore continue.

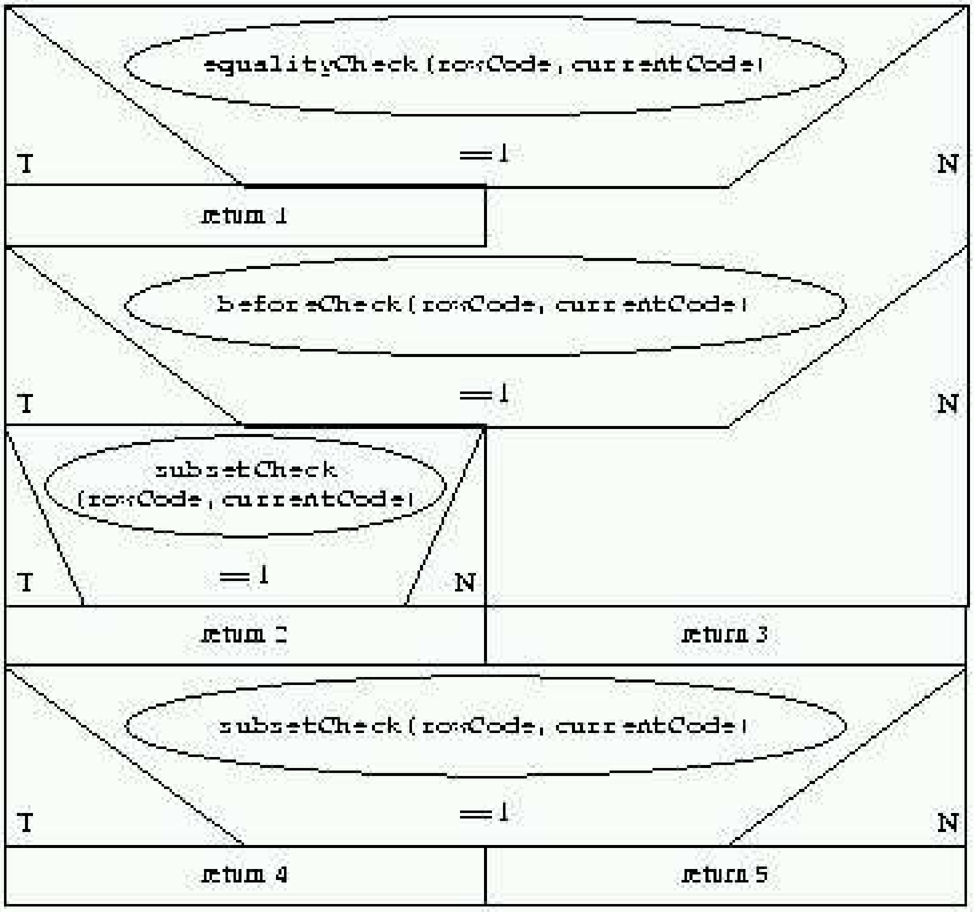
equalityCheck(rowCode, currentCode)
==1
T
N
return 1
beforeCheck(rowCode, currentCode)
==1
T
N
subsetCheck
(rowCode, currentCode)
==1
T
N
return 2
return 3
subsetCheck(rowCode, currentCode)
==1
T
N
return 4
return 5

Figure 11: Nassi-Shneiderman chart for *checkCodes* procedure

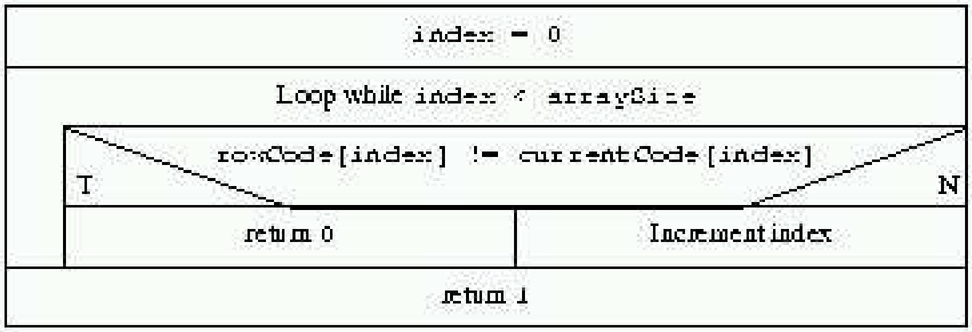
index = 0
Loop while index < arraySize
rowCode[index] != currentCode[index]
T
N
return 0
Increment index
return 1

Figure 12: Nassi-Shneiderman chart for *equalityCheck* procedure

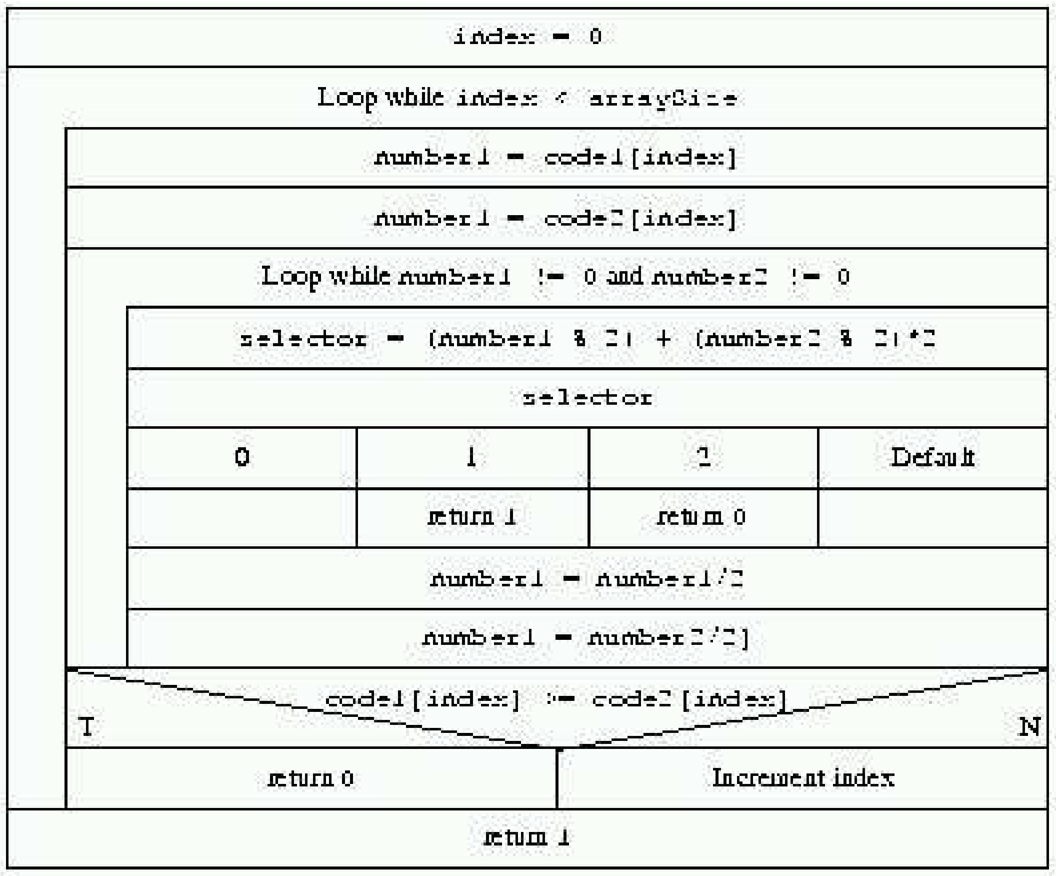

Figure 13: Nassi-Shneiderman chart for *beforeCheck* procedure

selector = 1 Bit n in first number is 1 and in second number is 0, therefore before.

selector = 2 Bit n in second number is 1 and in first number is 0, therefore after.

selector = 3 Both codes have a 1 bit n therefore continue.

The function uses six data items:

NAME	TYPE	DESCRIPTION
*code*1	Unsigned integer pointer	Formal parameter for first bit code.
*code*2	Unsigned integer pointer	Formal parameter for second bit code.
index	Integer	Local variable index with which to step through codes.
*number*1	Integer	Local variable in which to store elements of *code*1.
*number*2	Integer	Local variable in which to store elements of *code*2.
selector	Integer	Selector for case statement.

A detailed design for the *beforeCheck* procedure is presented in Figure 13.

5.8 SUBSET CHECK

The *subsetCheck* function, in form, is very similar to the *equalityCheck* function described above. However, comparisons are made using the “and” logical operator. Given two codes, (say) $X = 0011$

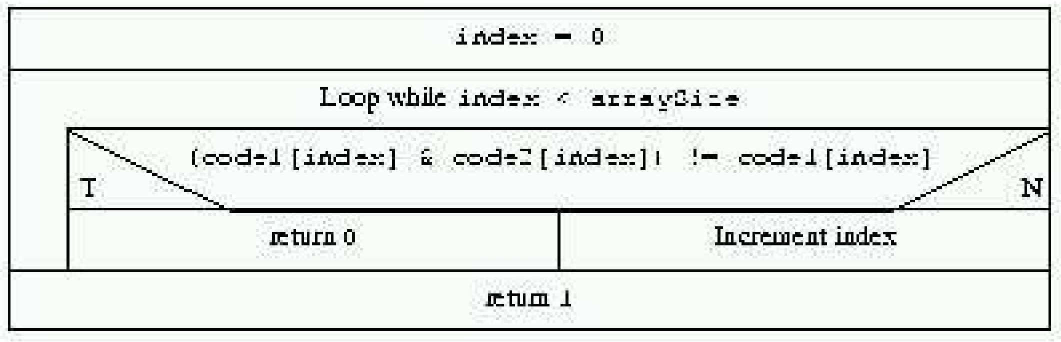
index = 0
Loop while index < arraySize
(code1[index] & code2[index]) != code1[index]
T
N
return 0
Increment index
return 1

Figure 14: Nassi-Shneiderman chart for *subsetCheck* procedure

and $Y = 1011$, and we wish to determine whether the X is a subset of Y we determine the result of a logical "and" operation between the two codes:

$$0011 \ \& \ 1011 = 0011$$

and if the result is equal to X (which it is in the above example) then $X \subset Y$ is true. By reversing the arguments we can of course determine if one is a superset of the other.

As with the *equalityCheck* function the function uses three data items:

NAME	TYPE	DESCRIPTION
*code*1	Unsigned integer pointer	Formal parameter for first bit code.
*code*2	Unsigned integer pointer	Formal parameter for second bit code.
index	Integer	Local variable index with which to step through codes.

A detailed design for the *subsetCheck* procedure is presented in Figure 14.

5.9 CREATE TREE STRUCTURE

The *createStructure* function simply creates a structure for a node in the tree given an appropriate bit code. Assuming that the structure is successfully created the function then returns a pointer to that structure. The function includes the following data items:

NAME	TYPE	DESCRIPTION
code	Unsigned Integer Pointer	Formal parameter giving the bit code for the current in the table
newPtr	PARTSUPPORTPTR	Local variable pointing to start of the newly created structure.

A detailed design for the *createStructure* procedure is presented in Figure 15

6 ANALYSIS

The storage required for the algorithm is as follows:

$$K(12 + \mathit{number\ of\ array\ elements})\ \mathit{bytes}$$

where K is the number of unique rows in the table, and *number elements* is calculated from:

$$\mathit{number\ of\ array\ elements} = \frac{n}{32} + \frac{31}{32}(\mathit{rounded\ up\ to\ the\ nearest\ integer})$$

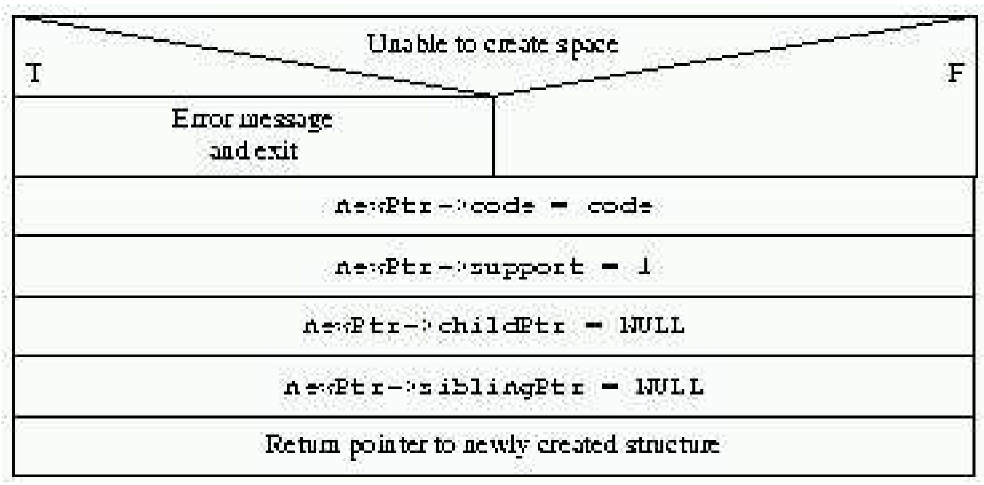

Figure 15: Nassi-Shneiderman chart for *createStructure* procedure

In the worst case K will be equal to m (the number of rows in the table), in the best case K will equal 1 (all rows are the same). If we assume a normal distribution then we can be sure of duplication if:

$$m => 2^n$$

The number of support increments is dependent on the number of unique rows in the database (K) and can be calculated using the identity:

$$support\ increments = m - k$$

The number of comparisons required is largely dependent on the number of duplicates in the table which in turn is dependent on the ratio of M to N. Table 1 gives some statistical data comparing the linked list partial support algorithm described in [4] with that described here. The table assumes that M is a constant 20000 rows. Both algorithms benefit from advantages gained by storing duplicates. However, this advantage is lost after M overtakes 2^N (at about $N = 17$). When M reaches 22 and beyond the number of duplicates and subsets is such that we can say that the number of Nodes is equivalent to the number of rows (i.e. $k = M$).

A graph indicating the storage requirements for the two algorithms using the data presented in Table 1 is given in Figure 16. From the graph (and Table 1) it can be seen that the tree algorithm requires more storage than the linked list. This is not surprising because each linked list node has one branch, while each tree node has two branches; consequently each tree node will require four more bytes of storage than the equivalent linked list node.

A second graph is presented in Figure 17 which compares the number of "tests" associated with each of the algorithms. From this graph we can see that the tree algorithm works well while the likelihood of subsets is high ($N \leq 15$ where $M = 20000$), but as this likelihood decreases the tree becomes "flat", i.e. it degenerates into what is effectively a linked list. However, for each row, checks are still made for the possibility of adding child branches; i.e. three checks are made using the tree algorithm (i) an equals check, (ii) a lexiconically "before/after" check and (iii) a subset/superset check, corresponding to only one check in the linked list algorithm.

Overall the advantages offered by the tree algorithm is dependent on the "added value" offered by the tree organisation — the precise nature of this added value has yet to be determined. It may be that using a simple binary tree is the most appropriate middle ground to adopt for the two approaches.

Test Table					**Linked List Partial Support Alg.**		**Tree Partial Algorithm**	
M	N	Permu-tations	Number Duplicates	Number Subsets	Storage	Number Comparisons	storage	Number Comparisons
20000	10	1024	18976	512	12288	9729164	16384	604459
20000	12	4096	15986	1986	48168	31591916	64224	1569538
20000	14	16384	9696	4153	123648	62966092	164864	24265263
20000	16	65536	6210	3288	165480	84694435	220640	107168424
20000	18	262144	1469	2741	222372	94845802	296496	204898149
20000	20	1048576	522	1291	233736	97839324	311648	264267326
20000	22	4194304	144	540	238272	99720345	317696	284314054
20000	24	16777216	71	193	239148	100254196	318864	293961661
20000	26	67108864	13	74	239844	100223287	319792	297884972
20000	28	268435456	5	27	239940	100113417	319920	298516547
20000	31	2147483648	0	4	240000	99820147	320000	300251079
20000	33		0	2	320000	100728688	400000	298497358
20000	63		0	0	320000	100648950	400000	301493040
20000	65		0	0	400000	99830442	480000	299624325
20000	78		0	0	400000	100207943	480000	301411839
20000	95		0	0	400000	99546592	480000	299424942
20000	97		0	0	480000	100583022	560000	298474803

Table 1: Comparison of "linked-list" partial support algorithm and "tree" partial support algorith

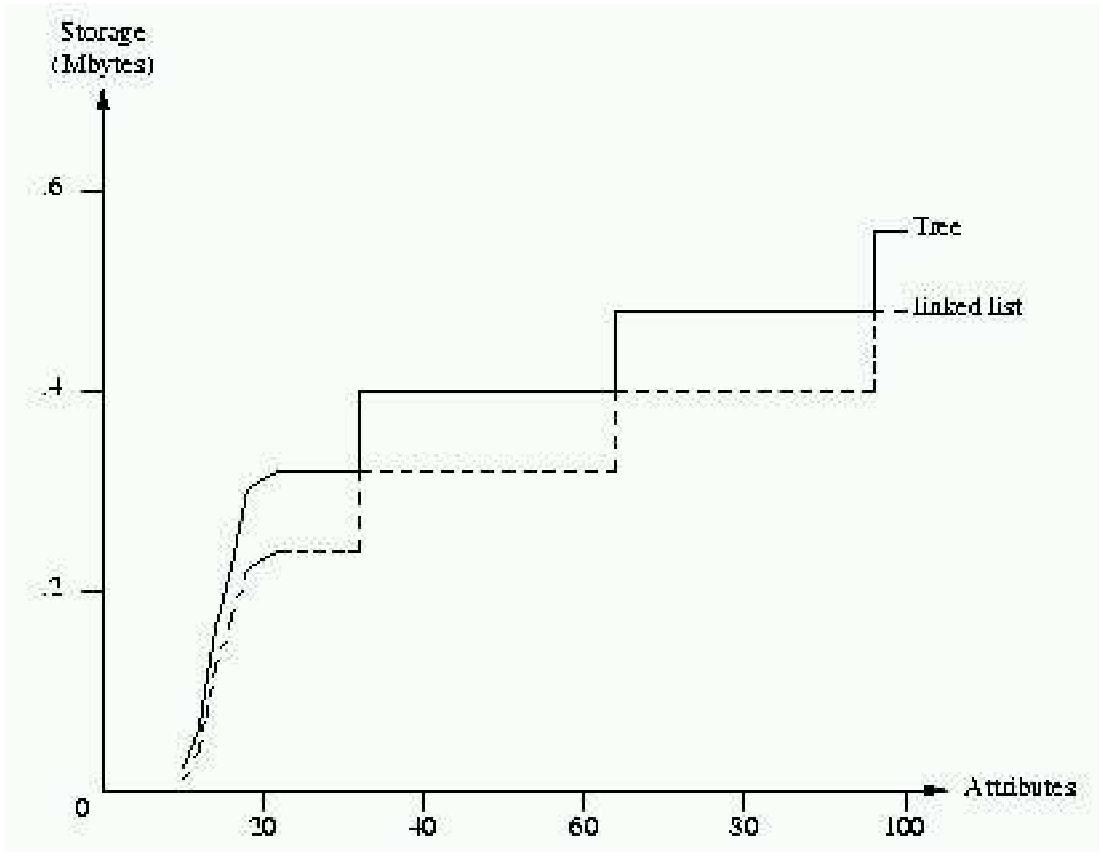

Figure 16: Comparison of storage requirements for linked list and tree algorithms ($M = 20000$)

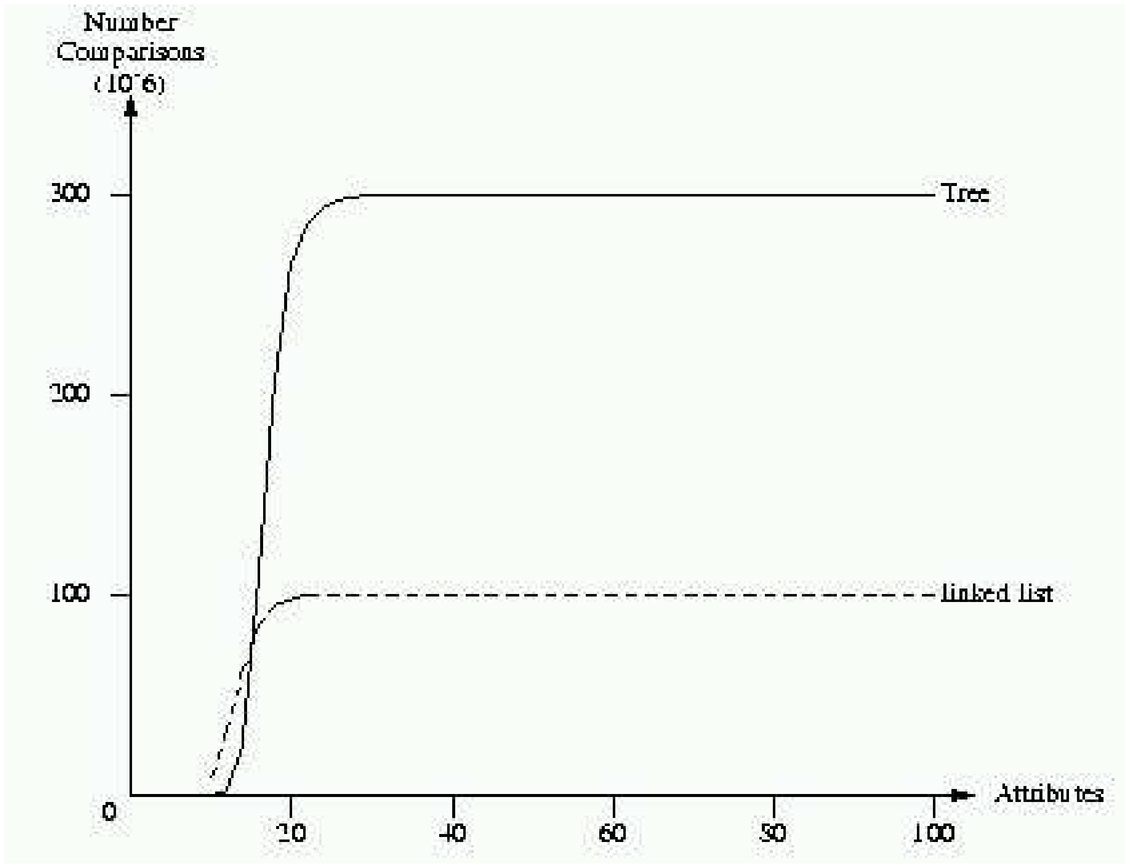

Figure 17: Comparison of number of "test" for linked list and tree algorithms ($M = 20000$)

References

[1] Agrawal, R. and Srikant, R (1994). Fast Algorithms for Mining Association Rules. Proceedings 20th VLDB conference.

[2] Coenen, F. (1998). A Brute force Algorithm for "Basket Analysis" (GCL-BF1). Dept of Computer Science, University of Liverpool. Working Paper 2.

[3] Coenen, F. (1999). A Second Brute force Algorithm for "Basket Analysis" (GCL-BF2). Dept of Computer Science, University of Liverpool. Working Paper 3.

[4] Coenen, F. (1999). Partial Support. Dept of Computer Science, University of Liverpool. Working Paper 4.

APPENDIX A — IMPLEMENTATION

```
/* -------------------------------------------------------------------------- */
/*                                                                            */
/*            P A R T I A L   S U P P O R T   A L G O R I T  H M              */
/*                                                                            */
/*                             Frans Coenen                                   */
/*                                                                            */
/*                           10 February 1999                                 */
/*                                                                            */
/* -------------------------------------------------------------------------- */

/* Program to generate partial supports for lines in a "basket analysis"
table. The code is designed to form part of other programs and not to act
alone. */

/* to compile "gcc -c support.c" */

/* ------ INCLUDE STATEMENTS ------ */

#include "support.h"
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

/* Example test file:

00100011001111100101000110
11001001000000010111001010
00010000001010111001010001
10110110101110010001001010
10100000110111001000011000
01010111101101110101111101
00010001001010010010010101
01000011101011110100001111
00100100010111010010000010
11001111100111001001011011

/* ------ DATA STRUCTURES ------ */

/*  Linked list storage structure to store partial supports. Structure includes:

1) code       = Array of unsigned integers describing the row.
2) support    = Partial support for the rows.
3) childPtr   = Pointer to child structure.
4) siblingPtr = Pointer to sibling structure.

defined in support.h. */

PARTSUPPORTPTR startPtr = NULL;

/* ------ GLOBAL VARIABLES ------ */

/* arraySize = Size of bit code storage array.
   totalRows = Number of rows in table (initialised to 1 for first row).
   totalCols = Number of columns in table. */

int arraySize, totalRows = 1, totalCols;

/* ------ FUNCTION PROTOTYPES ------ */

void partialSupport(char *);
void processInput(FILE *);
void addSupport(int, PARTSUPPORTPTR, PARTSUPPORTPTR, unsigned int *);
void addSupport2(int, PARTSUPPORTPTR, PARTSUPPORTPTR);
int checkSiblingBranch(PARTSUPPORTPTR, int *, PARTSUPPORTPTR *);
int checkCodes(unsigned int *, unsigned int *);
int equalityCheck(unsigned int *, unsigned int *);
int subsetCheck(unsigned int *, unsigned int *);
int beforeCheck(unsigned int *, unsigned int *);
unsigned int* generateBitPattern(char *);
PARTSUPPORTPTR createStructure(unsigned int *code);

/* Next three prototypes for testing only. */

void outputPartSupportTree(void);
void outputPartSupport1(PARTSUPPORTPTR, char *, int);
void outputPartSupport2(PARTSUPPORTPTR);

/* ------------------------------- */
/*                                 */
/*         PARTIAL SUPPORT         */
/*                                 */
/* ------------------------------- */

void partialSupport(char *filename)
{
FILE *in_file;

/* Open file and process. */

if ((in_file=fopen(filename,"r")) == NULL) {
        printf("Unable to open '%s' for reading\n",filename);
        exit(1);
        }
processInput(in_file);

/* Close file. */

fclose(in_file);
}

/* ------ PROCESS INPUT ------ */

/* Note: fgets(stringname,n,filename) - function to read from a file until
either n-1 characters or a newline character is encountered, and store the
characters in the given stringname. */

void processInput(FILE *in_file)
{
char input[512];        /* Largest row has 512 columns! */
unsigned int *code = NULL;
PARTSUPPORTPTR oldPtr = NULL;

/* Get first row, calculate number of columns.  */

fgets(input,512,in_file);
totalCols = strlen(input)-1;
arraySize = (int) (totalCols/32.0 + 31.0/32.0);

/* process first line in table. */

code = generateBitPattern(input);
startPtr = createStructure(code);

/* process rest of table */

while (fgets(input,512,in_file) != '\0') {
        code = generateBitPattern(input);
        addSupport(0,startPtr,oldPtr,code);
        totalRows++;
        }
}

/* ------ ADD SUPPORT ------ */
/* Add current row to linked list structure as follows:

code = 1, Increment support
code = 2, Row before and subset of current node (parent)
code = 3, Row before and not subset of current node (elder sibling)
code = 4, Row after and superset of current node (child)
code = 5, Row before and not superset of current node (younger sibling)

Codes generated by call the checkCode function. */

void addSupport(int flag, PARTSUPPORTPTR linkPtr, PARTSUPPORTPTR oldPtr,
                unsigned int *rowCode)
{
PARTSUPPORTPTR newPtr = NULL, newSiblingPtr = NULL;

/* Process comparison code */

switch (checkCodes(rowCode,linkPtr->code)) {
        case 1:         /* Rule1: Same */
                linkPtr->support = linkPtr->support+1;
                return;
        case 2:                 /* Rule2: Before and subset */
                newPtr = createStructure(rowCode);
                newPtr->childPtr = linkPtr;
                addSupport2(flag,newPtr,oldPtr);
                if (checkSiblingBranch(linkPtr,rowCode,&newSiblingPtr) == 1)
                                newPtr->siblingPtr = newSiblingPtr;
                return;
        case 3:                 /* Rule 3: Before and not subset */
                newPtr = createStructure(rowCode);
                newPtr->siblingPtr = linkPtr;
                addSupport2(flag,newPtr,oldPtr);
                return;
        case 4:         /* Rule 4: After and superset */
                if (linkPtr->childPtr == NULL) {
                        newPtr = createStructure(rowCode);
                        linkPtr->childPtr = newPtr;
                        }
                else addSupport(1,linkPtr->childPtr,linkPtr,rowCode);
                return;
        case 5:                 /* Rule 5: After and not superset */
                if (linkPtr->siblingPtr == NULL) {
                        newPtr = createStructure(rowCode);
                        linkPtr->siblingPtr = newPtr;
                        }
                else addSupport(2,linkPtr->siblingPtr,linkPtr,rowCode);
                return;
        default:        /* Default: Error */
                printf("ERROR: Unidentified bit comparison code in addSupport\n");
                exit(1);
        }
}

/* ------ ADD SUPPORT 2 ------ */

/* Add new node where "before and subset". The flag argument indicates which
type of branch is currently under consideration: = = root, 1 = child,
2 = sibling. */

void addSupport2(int flag, PARTSUPPORTPTR newPtr, PARTSUPPORTPTR oldPtr)
{
switch (flag) {
        case 0:
                startPtr = newPtr;
                return;
        case 1:
                oldPtr->childPtr = newPtr;
                return;
        case 2:
                oldPtr->siblingPtr = newPtr;
                return;
        default:
                printf("ERROR: Unidentified flag in addSupport\n");
        }
}

/* ------ CHECK SIBLING BRANCH ------ */

/* Check sibling branch to determine whether it is superset of the current row
or not. If so return 0 (do nothing), and 1 otherwise (in which case some of the
sibling nodes must be moved up. There are four possibilities:

1. Sibling branch is MT (return 0).
2. No nodes in sibling branch are supersets of row there for move them all up
   (return 1).
3. All nodes in sibling branch are supersets of row there for move none of them
   (return 0).
4. Some nodes in sibling branch are supersets of row others are not therefore
   move those that are not (return 1).        */

int checkSiblingBranch(PARTSUPPORTPTR linkPtr, int *rowCode,
                PARTSUPPORTPTR *newSiblingPtr)
{
PARTSUPPORTPTR localLinkPtr = NULL, markerPtr = NULL;

/* Check for empty sibling branch, if so return 0 (do nothing). */

if (linkPtr->siblingPtr == NULL) return(0);

/* Check if first node in sibling branch points to a superset of row. If not
move the entire branch up, return 1. */

if (subsetCheck(rowCode,linkPtr->siblingPtr->code) == 0) {
        *newSiblingPtr = linkPtr->siblingPtr;
        linkPtr->siblingPtr = NULL;
        return(1);
        }

/* Check rest. Branch starts of with supersets of row (which are OK where they
are), but we must find the point where this is no longer the case, i.e. the
part of the branch that needs to be moved up (if any). */

markerPtr = linkPtr->siblingPtr;
localLinkPtr = linkPtr->siblingPtr->siblingPtr;
while (localLinkPtr != NULL) {
        if (subsetCheck(rowCode,localLinkPtr->code) == 0) {
                *newSiblingPtr = localLinkPtr;
                markerPtr->siblingPtr = NULL;
                return(1);
                }
        markerPtr = localLinkPtr;
        localLinkPtr = localLinkPtr->siblingPtr;
        }

/* No nodes to be moved up. Return 0 (do nothing). */

return(0);
}

/* ---------------------------------- */
/*                                    */
/*      CODE COMPARISON ROUTINES      */
/*                                    */
/* --------------------------------- */

/* ------ CHECK CODE ----- */

/* Check row code with current node code: 1 = same, 2 = parent,
3 = before, 4 = child, and 5 = after. */

int checkCodes(unsigned int *rowCode, unsigned int *currentCode)
{
/* Check if same */

if (equalityCheck(rowCode,currentCode) == 1) return(1);

/* Check whether before or after and subset/superset. */

if (beforeCheck(rowCode,currentCode) == 1) {
        if (subsetCheck(rowCode,currentCode) == 1) return(2);
        else return(3);
        }
if (subsetCheck(currentCode,rowCode) == 1) return(4);
return(5);
}

/* ------ EQUALITY CHECK ------ */

/* Return 1 if row code is equal to current code, and 0 otherwise. */

int equalityCheck(unsigned int *code1, unsigned int *code2)
{
int index = 0;

while (index < arraySize) {
        if (code1[index] != code2[index]) return(0);
        else index++;
        }

/* Codes the same. */

return(1);
}

/* ------ BEFORE CHECK ------ */

/* Returns 1 if first codes is less than (before) second code and 0 otherwise.
Note that before here is not numerical but lexical, i.e. AB is before B
although the code associated with AB (3) is numerically after the code
associated with B (2). The selector used in the case statement is made up of the
remainder of two divisions by 2 (the second multiplied by 2). Thus:

0 = Both codes have a 0 bit n therefore continue.
1 = Bit n in first number is 1 and in second number is 0, therefore before.
2 = Bit n in second number is 1 and in first number is 0, therefore after.
3 = Both codes have a 1 bit n therefore continue.                  */

int beforeCheck(unsigned int *code1, unsigned int *code2)
{
int index = 0, selector;
unsigned int  number1, number2;

while (index < arraySize) {
        number1 = code1[index];
        number2 = code2[index];
        while (number1 != 0 && number2 != 0) {
                 selector = (number1 % 2) + (number2 % 2)*2;
                 switch(selector) {
                        case 0:
                                break;
                        case 1:
                                return(1);
                        case 2:
                                return(0);
                        default:
                        }
                number1 = number1/2;
                number2 = number2/2;
                }
        if (code1[index] >= code2[index]) return(0);
        else index++;
        }

/* Codes the same. */

return(1);
}

/* ------ SUBSET CHECK ------ */

/* Return 1 if row code is a subset of current code, and 0 otherwise. */

int subsetCheck(unsigned int *code1, unsigned int *code2)
{
int index = 0;

while (index < arraySize) {
        if ((code1[index] & code2[index]) != code1[index]) return(0);
        else index++;
        }

/* Row code is a subset of the given code. */

return(1);
}

/* ------------------- */
/*                     */
/*      UTILITIES      */
/*                     */
/* ------------------- */

/* ------ GENERATE BIT PATTERN ------ */

/* Generate an integer bit pattern according to 0 and 1 in table row (least
significant bit to the left). e.g. 1 1 1 0 0 = 7 */

unsigned int* generateBitPattern(char *input)
{
int count1 = 0, count2 = 0, stringIndex=0, arrayIndex=0;
unsigned int *arrayPtr;
unsigned int number, increment;

/* Create space (Malloc returns NULL if no space available). */

arrayPtr = (unsigned int *) malloc(sizeof(unsigned int)*arraySize);

/* Add values to array. */

while (arrayIndex < arraySize) {
        number = 0;
        increment = 1;
        count1 = 1;
        while (count1 <= 32 && count2 < totalCols) {
                if (input[stringIndex] == '1') number = number + increment;
                increment = increment*2;
                stringIndex++;
                count1++;
                count2++;
                }
        arrayPtr[arrayIndex] = number;
        arrayIndex++;
        }

return(arrayPtr);
}

/* ------------------------------------------------------------ */
/*                                                              */
/*          "LINKED LIST OF PARTIAL SUPPORTS" UTILITIES         */
/*                                                              */
/* ------------------------------------------------------------ */

/* ------ CREATE TREE STRUCTURE------ */

/* Create a set structure of the type PARTIAL SUPPORT given the code for the
item. */

PARTSUPPORTPTR createStructure(unsigned int *code)
{
PARTSUPPORTPTR newPtr = NULL;

if ((newPtr = (PARTSUPPORTPTR)(malloc(sizeof(PARTSUPPORT))))==NULL) {
        printf("Insufficient storage space\n");
        exit(1);
        }

newPtr->code = code;
newPtr->support = 1;
newPtr->childPtr = NULL;
newPtr->siblingPtr = NULL;

return(newPtr);
}

/* ------------------------ */
/*                          */
/*          OUTPUT          */
/*                          */
/* ------------------------ */

/* ------ OUTPUT PARTIAL SUPPORT STRUCTURE LINKED LIST ------ */

/* If more then 8 columns it becomes difficult to output node identifiers. For
8 columns, in the worst case, we need a character string of 256 elements. */

void outputPartSupportTree(void)
{
if (totalCols <= 8) outputPartSupport1(startPtr,"start",1);
else outputPartSupport2(startPtr);
}

/* ------ OUTPUT PARTIAL SUPPORT 1 ------ */

void outputPartSupport1(PARTSUPPORTPTR linkPtr,char *node, int counter)
{
int index=0;
char newNode[256];

if (linkPtr != NULL) {
        if (strcmp(node,"start") == 0) sprintf(newNode,"%d",counter);
        else sprintf(newNode,"%s.%d",node,counter);
        printf("(%s) ",newNode);
        while (index < arraySize) {
                printf("%u ",linkPtr->code[index]);
                index++;
                }
        printf("support = %d\n",linkPtr->support);
        outputPartSupport1(linkPtr->childPtr,newNode,1);
        counter = counter+1;
        outputPartSupport1(linkPtr->siblingPtr,node,counter);
        }
}

/* ------ OUTPUT PARTIAL SUPPORT 2 ------ */

/* No node identifiers. */

void outputPartSupport2(PARTSUPPORTPTR linkPtr)
{
int index=0;

if (linkPtr != NULL) {
        while (index < arraySize) {
                printf("%u ",linkPtr->code[index]);
                index++;
                }
        printf("support = %d\n",linkPtr->support);
        outputPartSupport2(linkPtr->childPtr);
        outputPartSupport2(linkPtr->siblingPtr);
        }
}
```

APPENDIX B - TESTING

Here follows a set of “basket analysis” tables designed to execute all aspects of the code wherever a new node is to be inserted into the tree structure.

Rule 2: Before and subset — new root node (start) to tree.

- No siblings to check. Input (add A to $\{AB, ABC\}$):

```
110
111
100
```

Output:

```
(1) 1 support = 1
(1.1) 3 support = 1
(1.1.1) 7 support = 1
```

- All siblings to be moved up to become siblings of new node (root). Input add A to $\{AB, B, C, ABC, BC, CD\}$):

```
1100
0100
0010
1110
0110
0011
1000
```

Output:

```
(1) 1 support = 1
(1.1) 3 support = 1
(1.1.1) 7 support = 1
(2) 2 support = 1
(2.1) 6 support = 1
(3) 4 support = 1
(3.1) 12 support = 1
```

- No siblings to be moved up. Input (add A to $\{AB, AC, AD, ABC, ACD, ADE\}$):

```
11000
10100
10010
11100
10110
10011
10000
```

Output:

```
(1) 1 support = 1
(1.1) 3 support = 1
(1.1.1) 7 support = 1
(1.2) 5 support = 1
(1.2.1) 13 support = 1
(1.3) 9 support = 1
(1.3.1) 25 support = 1
```

- Some siblings to be moved up. Input (add A to $\{AB, AC, B, ABC, ACD, BC\}$):

```
1100
1010
0100
1110
1011
0110
1000
```

Output:

```
(1) 1 support = 1
(1.1) 3 support = 1
(1.1.1) 7 support = 1
(1.2) 5 support = 1
(1.2.1) 13 support = 1
(2) 2 support = 1
(2.1) 6 support = 1
```

Rule 2: Before and subset — new node on child branch.

- No siblings to check. Input (add AB to $\{A, ABC, ABCD\}$):

```
1000
1110
1111
1100
```

Output:

```
(1) 1 support = 1
(1.1) 3 support = 1
(1.1.1) 7 support = 1
(1.1.1.1) 15 support = 1
```

- All siblings to be moved up to become siblings of new node (root). Input (add AB to $\{A, ABC, AC, AD, ABCD, ACD, ADE\}$:

```
10000
11100
10100
10010
11110
10110
10011
11000
```

Output:

```
(1) 1 support = 1
(1.1) 3 support = 1
(1.1.1) 7 support = 1
(1.1.1.1) 15 support = 1
(1.2) 5 support = 1
(1.2.1) 13 support = 1
(1.3) 9 support = 1
(1.3.1) 25 support = 1
```

- No siblings to be moved up. Input (add AB to $\{A, ABC, ABD, ABE, ABCD, ABDE, ABEF\}$):

```
100000
111000
110100
110010
111100
110110
110011
110000
```

Output:

```
(1) 1 support = 1
(1.1) 3 support = 1
(1.1.1) 7 support = 1
(1.1.1.1) 15 support = 1
(1.1.2) 11 support = 1
(1.1.2.1) 27 support = 1
(1.1.3) 19 support = 1
(1.1.3.1) 51 support = 1
```

- Some siblings to be moved up. Input (add AB to $\{A, ABC, ABD, AD, ABCD, ABDE, ADE\}$):

```
10000
11100
11010
10010
11110
11011
10011
11000
```

Output:

```
(1) 1 support = 1
(1.1) 3 support = 1
(1.1.1) 7 support = 1
(1.1.1.1) 15 support = 1
(1.1.2) 11 support = 1
(1.1.2.1) 27 support = 1
(1.2) 9 support = 1
(1.2.1) 25 support = 1
```

Rule 2: Before and subset — new node on sibling branch.

- No siblings to check. Input (add B to $\{A, BC, BCD\}$):

```
1000
0110
0111
0100
```

Output:

```
(1) 1 support = 1
(2) 2 support = 1
(2.1) 6 support = 1
(2.1.1) 14 support = 1
```

- All siblings to be moved up to become siblings of new node (root). Input (add B to $\{A, BC, BD, BE, BCD, BDE, BEF\}$):

```
100000
011000
010100
010010
011100
010110
010011
010000
```

Output:

```
(1) 1 support = 1
(2) 2 support = 1
(2.1) 6 support = 1
(2.1.1) 14 support = 1
(2.2) 10 support = 1
(2.2.1) 26 support = 1
(2.3) 18 support = 1
(2.3.1) 50 support = 1
```

- No siblings to be moved up. Input (add B to $\{A, BC, C, D, BCD, CD, DE\}$):

```
10000
01100
00100
00010
01110
00110
00011
01000
```

Output:

```
(1) 1 support = 1
(2) 2 support = 1
(2.1) 6 support = 1
(2.1.1) 14 support = 1
(3) 4 support = 1
(3.1) 12 support = 1
(4) 8 support = 1
(4.1) 24 support = 1
```

- Some siblings to be moved up. Input (add B to $\{A, BC, BD, D, BCD, BDE, DE\}$):

```
10000
01100
01010
00010
01110
01011
00011
01000
```

Output:

```
(1) 1 support = 1
(2) 2 support = 1
(2.1) 6 support = 1
(2.1.1) 14 support = 1
(2.2) 10 support = 1
(2.2.1) 26 support = 1
(3) 8 support = 1
(3.1) 24 support = 1
```

Rule 3: Before and not subset — new root node (start) to tree. Input (add A to $\{B\}$):

```
01
10
```

Output:

```
(1) 1 support = 1
(2) 2 support = 1
```

Rule 3: Before and not subset — new node on child branch. Input (add A to $\{AB, BC\}$):

```
110
011
100
```

Output:

```
(1) 1 support = 1
(1.1) 3 support = 1
(2) 6 support = 1
```

Rule 3: Before and subset — new node on sibling branch. Input (add A to $\{B, C\}$):

```
010
001
100
```

Output:

```
(1) 1 support = 1
(2) 2 support = 1
(3) 4 support = 1
```

Rule 4: After and superset — empty child branch. Input (add C to $\{A\}$):

```
100
001
```

Output:

```
(1) 1 support = 1
(2) 4 support = 1
```

Rule 4: After and superset — proceed down child branch. Input (add C to $\{A, B, AB\}$):

```
100
010
110
001
```

Output:

```
(1) 1 support = 1
(1.1) 3 support = 1
(2) 2 support = 1
(3) 4 support = 1
```

Rule 5: After and not superset — empty sibling branch. Input (add ABC to $\{A\}$):

```
100
111
```

Output:

```
(1) 1 support = 1
(1.1) 7 support = 1
```

Rule 5: After and not superset — proceed down sibling branch. Input (add ABC to $\{A, B, AB\}$):

```
100
010
110
111
```

Output:

```
(1) 1 support = 1
(1.1) 3 support = 1
(1.1.1) 7 support = 1
(2) 2 support = 1
```