AN ALGORITHM FOR PROCESSING STEM ANALYSIS DATA AND SAMPLING INTENSITIES FOR IMMATURE JACK PINE GROWTH

by

Tiemin Sheng ©

A thesis submitted for the degree of Master of Science in Forestry

Lakehead University

Thunder Bay, Ontario

1994

ISBN 0-315-97066-9

ABSTRACT

Tiemin Sheng. 1994 An algorithm for processing stem analysis data and sampling intensities for immature jack pine growth 176 pp.

Major advisor: Dr. H. G. Murchison

Key Words: Stem analysis, computer algorithm, two-stage sampling, computer simulation.

This study examined two topics. In the first, a computer algorithm was developed to process stem analysis data produced by Tree Ring Increment Measure (TRIM) system. The algorithm developed not only processed TRIM data for cumulative increment of volume, height, and dbh by one-year intervals for individual trees, but also calculated annual volume increment per unit area (vol./ha) by one-year intervals for stands. A hashing technique with a linked list data structure was used in the algorithm. The advantages of the algorithm are to process stem analysis and manage outputs efficiently and to provide a user with quick access to any processed stem analysis tree records. In the second, sampling intensities on both plot and tree levels were investigated. Two forms of two-stage sampling strategies were employed. The study indicated that subsampling using Probabilities Proportional to Size (PPS) could produce reliable estimates for an annual growth. The study suggested that over 91 percent of precision of mean growth estimate can be obtained with the sample plot intensities of 66 percent at the first stage and with the sample tree intensities of 2.1 percent at the second stage at the 95 percent confidence level. The study also showed that subsampling with PPS was superior to that with simple random subsampling.

ACKNOWLEDGEMENTS

I wish to sincerely acknowledge my supervisor, Dr. H. G. Murchison, for kindly offering me valuable suggestions and advice, providing me with the data set, and the time he gave to read the preceding drafts, without which this thesis and research would have been impossible.

I would like to extend my warmest thanks to both Dr. K. M. Brown and Dr. A. J. Kayll, the members of my committee, for their suggestions and time to read the drafts.

A special thank goes to my wife, Jieping, for providing me with devotion and encouragement.

CONTENTS

TABLES

FIGURES

1. INTRODUCTION

The Ontario Ministry of Natural Resources through District Offices and Technological Development Units have conducted studies on stem analysis of individual trees in a number of stands (Murchison and Kavanagh, 1989). These studies were intended to investigate the effects of treatments such as drainage, fertilization, thinning, etc. (Murchison and Kavanagh, 1989). In the study, extensive stem analysis data have been collected and a stem analysis database system was developed. Under the system, named Tree Ring Increment Measure (TRIM) and developed by the Ontario Tree Improvement and Forest Biomass Institute, the database includes extensive information on individual stem analysis trees (Murchison and Kavanagh, 1989). In the locally developed TRIM database, individual tree data are stored in four separate files: the first file named *.OUT.*, the second file named RAD*., the third file named AD*.*, and the last file named ANNV*.*.

A major defficulty in processing this TRIM database in growth and yield purposes is that there are several thousands of data files which are managed by using directory management. With this technique, four different types of files concerning an individual tree

were stored separately in directories. Although TRIM files were managed into separate directories there were still several hundred files of the same type which appeared in a subdirectory. The first question raised was how to process all the TRIM data with only one execution of a computer program to obtain the growth attributes by one-year intervals for all stem analyzed trees. The second question was how to develop a computer algorithm to store the processed data, and manipulate them so that a TRIM data user can have an efficient way to retrieve such stored information.

With the present TRIM methodology, the TRIM sample unit is a 20 m by 20 m plot with an inner plot of 10 m by 10 m. All trees in the inner plot are felled and then stem analyzed to reconstruct growth development and yield (Murchison and Kavanagh, 1989). This is a labour-intensive and time consuming activity. A lot of financial and labour input must be made to collect stem analysis data. Therefore, any alternatives, which are cost effective, will be beneficial to those intending to investigate stand growth and yield of immature jack pine in northern Ontario. As an alternative to TRIM methodology, two two-stage sampling rules are examined in this study in order to get reliable information about jack pine growth with reduced cost and time. At present, no results are available on how many TRIM plots and how many trees are required to produce reliable estimates of volume growth for immature jack pine. Such information may be useful as a guide for foresters and decision makers in evaluating various silviculture treatments and forest resource management plans.

The objectives of this study are two-fold. One is to develop an algorithm to process TRIM data. The other is to investigate two-stage sampling for tree growth using TRIM data.

The first part of the thesis considers the development of computer programs to process TRIM data for the various growth attributes and manage the output efficiently. A hashing technique with the data structure of a linked list is employed.

The second part of this thesis deals with the problem of determining sample intensities for an estimate of volume growth of immature jack pine. Two two-stage sampling rules are investigated.

2. LITERATURE REVIEW

2.1 STEM ANALYSIS COMPUTER PROGRAMS

Stem analysis is a widely used method of studying the past growth of individual trees (Husch et al., 1972). No other method could replace stem analysis completely in reconstructing past growth or development of individual trees. The disadvantage, however, is that the process is time consuming and laborious (Maciver, 1987).

With the advent of annual growth ring measuring equipment such as the ADDO-X system and Holman Digimicrocomputer system and rapid development of advanced computer technology, interest in using stem analysis is high in forest research. But, so far, only a few stem analysis computer programs have been written. With the renewed interest in stem analysis it is becoming apparent that development of an efficient computer algorithm to process stem analysis data is warranted.

The earliest documented simple computer program for processing stem analysis data was developed by Brace and Mager (1968). The program was developed only for plotting the stem analyzed tree profile. The stem analysis data used were recorded in a prespecified

format and involved a data checking procedure. In their research, Brace and Mager (1968) compared the cost of the three computation methods.

Three years later, Pluth and Cameron (1971) developed an algorithm that graphed the derived tree growth parameters of periodic annual increment, and mean annual increment in basal area, height, and total volume. The printed output also includes: average diameter, basal area, and section volume by heights of cutting point and age intervals; total volume increments by age intervals; heights by cumulative age, and cumulative height and total volume by one-year intervals.

A few years later, Herman et al. (1975) developed an algorithm to process stem analysis data. Their algorithm was written specifically for site index research.

Another computer algorithm for plotting stem analysis was developed by Timmer and Verch (1983). The algorithm was developed specifically for forest productivity studies requiring comparisons of tree growth on sites of different productivities. The algorithm generated a set of growth curves showing the development of patterns of trees and computes the growth parameters such as MAI, CAI, height/age, height/dbh, and volume/age.

Fayle et al. (1983) developed a program to graphically display the radial growth pattern of the tree. The input data this program would require were produced by DIGI-MIC tree ring measurer. The advantage of this program is that line graphs of ring widths along

radii of a stem cross section and the average for successive stem sections could be compared visually.

Kavanagh (1983) developed two programs which handled stem analysis data primarily from HOLMAN Digimicrometer data. These two programs served, first, to verify stem analysis data, and second, to produce a set of sequences for the ring width data.

In addition, Kavanagh (1983) reported that there were two unpublished computer programs: one by Wang (1976) and the other by Chapeskie and Fleet (1981). Kavanagh (1983) reported that the algorithm by Wang calculated the periodic annual increment and the mean annual increment for a tree, and the algorithm by Chapeskie and Fleet (1981) was written specifically for handling Holman Digimicrometer data. This algorithm computed estimates of dbh, height, and volume, at the time of cutting and for the previous one- and five-year growth periods.

In summary, all the available algorithms process the stem analysis data from an electronic measuring machine and all outputs of these programs have similar formats. Some algorithms compute MAI, CAI, individual tree height, volume and dbh by age, and some algorithm plots a tree growth profile. All the computer algorithms except one by Kavanagh (1983) are restricted to processing a limited number of stem analysis trees at an execution. An ordered list was the only data structure used in all algorithms. Of all the computer algorithms, only that by Pluth and Cameron (1971) can be used to calculate cumulative height and total volume for a tree by one-year

intervals. However, none of the available algorithms can be used directly to process TRIM data. None of them can be used to calculate volume increment per unit area (m^3/ha) for a given stand by one-year intervals. Nor did they consider using hashing techniques to manage output files efficiently.

2.2 MULTI-STAGE SAMPLING DESIGNS

2.2.1 DESIGN PRINCIPLES

Multi-stage sampling techniques are presented in many text books. The notable books are by Deming (1950), Schumacher and Chapman (1954), Yates (1960), Cochran (1963), Yamane (1967), Sukhatme and Sukhatme (1970), Husch et al. (1972), Barnett (1974), Williams (1978), and de Vries (1986).

Multi-stage sampling is a technique which involves selecting a set of clusters of elements of interest from a target population, using selection rules either simple random sampling (SRS) or probability proportional to size (PPS) sampling, with subsequent subsampling within the selected clusters and measuring the elements at the final stage of selection (Deming, 1950; Cochran, 1963; Yamane, 1967; Sukhatme and Sukhatme, 1970). Sometimes this technique is referred to as subsampling (Cochran, 1963).

The basic procedure with multi-stage sampling is to construct a sample frame. According to Deming (1950), a frame needs to be constructed at every successive stage and the frame should describe all the subsequent sampling units in the population.

In a good design of two-stage sampling, one very important requirement is to delineate and define both primary and secondary sampling units as alike as possible (Deming, 1950). Because when elements in the same unit are alike precision can be gained even with a small number of subsamples (Cochran, 1963).

To avoid producing larger sampling errors in the final results with sample selection rule of equal probability, the populations of the primary sampling units and of the secondary sampling units should be as equal as possible (Deming, 1950). Williams (1978) thought that if first-stage units vary in size a loss in precision may be possible with a selection of units by a simple random selection rule. Selection of subsample with PPS may be the best choice if the units vary in size and sizes are known (Yates, 1960; Cochran, 1963). Because the selection of a subsample with PPS recognizes some inequalities of sample units (Stuart ,1968). Furthermore, this sampling rule may result in smaller Mean Square Error since it has a small contribution from variation between units (Cochran, 1963).

The sample precision of two-stage sampling is very closely related to the distribution of the sample between the two stages (Sukhatme and Sukhatme, 1970). According to these methods, a gain in precision may be realized by three approaches: (i) by selecting more primary sampling units at the first stage and then selecting fewer secondary sampling units at the second stage, as opposed to the other way around in which a small number of primary units is selected at the first stage and then a large number of secondary units at the second stage; (ii) by making primary sample units larger if the

condition of population is that the intra-class correlation within first-stage units is positive and it decreases with an increase of size of primary sampling units; and (iii) by clustering the first stage units that are as heterogeneous as feasible.

2.2.2 ADVANTAGES

Multi-stage sampling has some advantages. Yates (1960) remarked that "it enables existing natural divisions and subdivisions of populations to be utilized as units at the various stages; it permits the concentration of field work of censuses and surveys to cover larger areas; it is very useful for survey of undeveloped areas where no frame exists since only the parts of the population selected at any stage need to be listed for subsampling at the next stage".

To meet a prescribed precision, two-stage sampling is considered to be cheaper because at every successive stage the sampling units become smaller and smaller (Deming, 1950). The other advantage of multi-stage sampling is that laying out a frame for the next stage is needed only in the units which have already fallen into the sample and only the parts of the population selected at any stage need to be listed for subsampling at the next stage (Deming, 1950).

Multi-stage sampling is flexible and could be possibly extended to n-stages as needed according to specific research purposes (Cochran, 1963; Yamane, 1967; Stuart, 1968; Sukhatme and Sukhatme, 1970; Husch et al., 1972; Frayer 1979). The sampling units of multi-stage sampling shrink in size at each step, as opposed to multi-phase sampling in which the sampling units remains the same

size (Lund, 1982). In comparison with random sampling, two-stage sampling may reduce travelling and administration cost (Yamane, 1967) and it could bring a gain in precision compared with one-stage sampling (Stuart, 1968).

2.2.3 DISADVANTAGES

In general, a multi-stage sample is considered to be less precise compared with a sample containing the same number of final-stage units which have been selected by some suitable single-stage process (Yates, 1960).

The other disadvantage is that as we obtain greater flexibility with multi-stage sampling we may have to pay the price of greater complexity in the sampling selection and the analysis of the sample (Stuart, 1968).

In addition, there is also a difficulty in design of multi-stage sampling. According to Yamane (1967), if one is to select primary sampling units and subsampling units with SRS, it may be difficult to have those units roughly equal in size. Fortunately, this defect can be solved by the selection rule of PPS (Yamane, 1967).

2.2.4 ITS APPLICATIONS IN FORESTRY INVENTORIES

A number of forest scientists and researchers have conducted forest inventories using multi-stage sampling techniques. Several forms of multi-stage sampling strategies have been employed. Notably, of all the varied applications of multi-stage sampling, two-

stage sampling schemes have been frequently used by foresters in their research.

The efficiency of two-stage sampling in forest inventories are closely related to distance between sampled items, i.e. travel cost, number of items or plots in sampled first-stage units, and the variation between and within first-stage units (Frayer, 1979; Johnston, 1982).

Cunia (1965) designed two-stage sampling with regression where auxiliary variables are observed and both primary units of plots at stage one and subsample units of trees are selected by simple random sampling.

Farmer et al. (1973) used two-stage sampling to study coniferous standing volume and increments. They clustered the coniferous forest by stands. The stands were selected with replacement by probabilities proportional to size (area) at stage one and the plots were selected by simple random sampling at stage two. They used volume tables and a regression technique to obtain the total volumes and increments for each secondary unit. They found that in all circumstances two-stage sampling was not superior to the simple random sampling because of great variations between stands.

Bonner (1974) derived estimators for a stratified two-stage sampling design and then he described a timber inventory using this sampling design. Aldred and Hall (1975) extended Bonner's sampling design by incorporation of more sample units.

Langley (1975) developed multi-stage sampling theory and reported that estimators under two-stage PPS were unbiased. In his study, he derived estimators for up to four stages of sampling. At each stage of sampling, a level of remotely sensed data was used to generate sampling selection probabilities and trees were observed at the final stage. He applied his four stage sampling design to timber surveys.

Yandle (1977) designed the two-stage sampling which could be used to obtain updated volume growth attributes and additional measurements. Both primary units and secondary units are trees. They were selected with PPS at both stages; selection of primary units with probability proportional to basal area and selection of secondary units with probability proportional to height. He reported that a gain in efficiency was achieved since two variables most correlated to volume were used in two successive stages. Also, he found that two-stage sampling with selection of both primary units and secondary units with equal probability at both stages was inferior to that using PPS sampling at both stages.

Variance estimators were unbiased if SRS was used at both stages (Frayer, 1979; Johnston, 1982). Johnston (1982) suspected that bounds on an unbiased variance estimator might exist if systematic sampling was employed at stage two.

Frayer (1979) derived a set of formulas for multi-stage sampling with applications in remote sensing in forestry. Murchison (1984)

modified one of Frayer's two-stage formulas by modifying the expansion factor at the second stage.

Murchision (1984) used three two-stage sampling schemes as a part of methodology to investigate optimal tree height sampling intensities. The three two-stage sampling schemes resulted from the combinations of simple random sampling or point sampling at stage one and SRS or PPS sampling at stage two. Monte Carlo simulation was performed to compare efficiencies of these three sampling schemes. He generalized that plot based sampling schemes were more advisable than point sample based sampling schemes. With the plot based sampling schemes, he ranked the three sampling schemes in descending order according to their precisions; (i) selection of plot with SRS at stage one and of trees with probability proportional to basal area, (ii) selection of plots and of trees with SRS at both stages, and (iii) point samples at stage one and selection of trees with SRS at stage two. As to their desirable usage, he pointed out that sampling scheme (i) could find its best use in stands with uniformly distributed trees; sampling scheme (ii) was suggested for softwood stands showing clustered spatial distributions.

Murchison and Kavanagh (1989, and1990) conducted research on sampling intensities for yield for two coniferous species using two-stage sampling. They delineated first-stage units by three methods; clustering technique, Pielou's index method and stands. The primary units of plots were selected by SRS at stage one. At stage two, secondary units of trees were selected by two selection rules; SRS and

probability proportional to basal area. They found that two-stage sampling strategy with PPS was more efficient than that of SRS.

In summary, the two-stage sampling strategies implemented so far take forms of such combinations as selection of primary sampling units with either SRS or PPS rules, and selection of secondary sampling units with either SRS or PPS rules. Usually, the sample frame for population is constructed based on the forest area at first stage and characteristics of trees at the second stage. The PPS selection rule, if applicable at the first stage, is performed pertaining to forest area in most cases and, if applicable at the second stage, is usually related to basal area of individual trees. The various estimators with two-stage sampling strategies are unbiased (Frayer, 1979). The varied applications of two-stage sampling strategies in forest inventories have been used more frequently than other multi-stage sampling involving more than two stages. In this sense, two-stage sampling appears to be an important sampling technique in forest inventories.

3. TRIM DATABASE

The TRIM database consisted of all TRIM plot data from both immature and mature jack pine and black spruce stands collected by the OMNR in 1988. In the TRIM database, there were approximtely 10,000 files. A very small portion of the directory listings is shown in Table 1. Under the TRIM data base system, information for each stem analysis tree was stored in four separate files. Any particular record of stem analysis and its contents were identified by their plot number and tree number combined with OUT, or RAD, or AD, or ANNV. In other words, the four separate files stored information which could be used to describe both growth and yield for any stem analyzed trees. The format of these files was as follows:

(1) *.OUT;* ------ which described diameter information. This file was created in Timmins using a Pascal program developed by Domenic Colantonio (Murchison and Kavanagh, 1989). An example of such a file is shown in Table 2.

(2) RD*.;* ------ which described height information. This file and the following two files were created by the TRIM software package (Murchison and Kavanagh 1989).

(3) AD*.;* ------ which also described height information. They might be required when the RD*.;* file was missing or incomplete (Murchison and Kavanagh, 1989). A portion of such a file is shown in Table 3. The data (bold, italics) appeared in a string (one value per line). The remainder of the information in the table is the explanation of the value on that line and is not part of the file. There is a program called SAP in TRIM. The program uses the AD* and RD* ringwidths of data. This program is not available at present.

(4) ANNV*.;* ----- which describes volume information. A portion of such a file is shown in Table 4. The data were the bold, italicized values in string format. The remaining information explained the contents of that line and is not part of the file.

Table 1. A portion of directory listings from a UNIX operating system. For file names with format * K *, K was a plot label followed by a plot number and a tree number and when combined with "OUT", it indicated the diameter information files, with "Ad" indicated the height information files, and ANNV indicated volume information files.

K1_1.OUT
K1_2.OUT
K1_3.OUT
.
K3_1.OUT
K3_2.OUT
K3_3.OUT
.
AD_K1_1.
AD_K1_2.
AD_K1_3.
.
AD_K3_1.
AD_K3_2.
AD_K3_3.
.
ANNV_K1_1.
ANNV_K1_2.
ANNV_K1_3.
.
ANNV_K3_1.
ANNV_K3_2.
ANNV_K3_3.

Table 2. Part of K1_9.OUT file shown as an example of one of *.OUT files. Lines 1 to 3 were the complete information for the disc taken at 2.00 metres up the tree Line 1 contained the following information: species code, the year the tree was harvested, the height (cm) the cross-section was taken from, cross-section age, single bark thickness (mm), pith radius (mm) and the radial distance across the rings (mm). Lines 2 and 3 contained the ring widths (mm) for each ring on the disc.

```
  Pj 1986  200   17    2.350  1.555  63.480
1986 2.190 2.760 3.240 2.520 2.735 3.180 3.325 4.035 4.025
     5.160
1976 5.960 6.100 5.845 5.565 3.980 1.185 1.675
  Pj 1986  150   17    2.620  0.725  65.520
1986 2.205 2.590 2.510 2.360 2.675 2.710 3.135 3.680 3.350
     4.500
1976 4.775 5.290 5.070 5.015 5.270 7.680 2.705
  Pj 1986  130   18    2.155  1.300  65.755
1986 2.570 2.415 2.765 2.345 2.545 2.770 2.995 3.920 3.500
     4.575
1976 5.360 5.940 5.750 6.215 5.000 3.350 2.570 1.170
  Pj 1986  120   18    1.535  1.475  66.820
1986 2.165 2.595 2.665 2.235 2.375 2.750 2.980 3.230 3.210
     4.055
1976 4.370 5.105 5.155 7.500 6.575 5.245 2.925 1.685

                                    . . . . . .

  Pj 1986  0     21    3.695  0.695  84.665
1986 4.510 4.660 4.815 3.405 2.945 3.685 3.875 5.485 4.325
     3.615
1976 4.280 4.670 4.460 5.415 4.895 4.190 4.790 3.740 3.295
     2.565
1966 1.045
```

Table 3. Part of AD_K1_9. file, as an example of AD* file, is shown. Other actual AD* files resembled the information in column 1. The remaining information added by the writer explained the contents of the file and is not part of the file.

3	Ring count 3 (bark and pith included)
5	Ignore this line
1.215	Bark thickness in millimetres
1.265	Ring width in millimetres
1.62	
.	
12	Ring count is 12 and age is 10 years
5	Ignore this line
1.38	Bark thickness in millimetres
3.58	Ring width in millimetres
4.68	
.	
1185	Total height at 21 years in centimetres
073	Height increment of 1 year's growth (cm)
1112	Total height at 20 years (cm)
060	
1052	
.	
045	
0052	
045	
0007	Total height (cm) at 1 year
007	Height growth (cm) in 1 year

Table 4. Part of ANNV_K1_9. file, as an example of one ANNV* file, is shown. Information for the first three cross-sections and the last cross-section of the tree are shown. The other actual ANNV* files resembled column 1. The remaining comments added by the writer explained the contents of such kinds of files and were not part of the ANNV* files.

3	Ring count for disc 1
2	Ring 1 volume (cubic centimetres)
1	Ring 2 volume
0	Ring 3 volume
3	Ring count (bark and pith included)
3	
3	
1	
3	
3	
4	
1	
.	
165	Ring count on last disc
129	
126	
96	
65	
56	
32	
18	
6	
0	
0	

4. DEVELOPMENT OF THE HASHING ALGORITHM

4.1 HASHING TABLE

Hashing is an address calculation technique and is an excellent way to maintain a static or dynamic dictionary in database management (Harrison, 1972; Flores, 1977; VanWyk, 1988). Harrison (1972) described hashing as an ingenious technique which could be used in a number of areas. Stone (1972) thought that, in particular, hashing could find its best use in dealing with large data sets.

A hash table and a key are important components in hashing techniques. According to Standish (1980), a hash table is an aggregate of individual components called records. Distinct records in a hash table contain distinct keys and each record stores information associated with its key. The key is either the name of the entity to which a record pertains, or is chosen to identify a particular record uniquely in a hash table. The basic idea about hashing is that a function, called a hashing function, is applied to an item or its key, and the result, called a hash value, is used as a sort of abbreviation of the item (Harrison, 1972).

According to Aho et al. (1983), in general, there are two forms of hashing. The first is called open or external hashing which allows the set to be stored in a potentially unlimited space. The advantage of this form of hashing is that it places no limit on the size of the set. The second is referred to as closed or internal hashing. This form of hashing limits the size of sets due to using a fixed space for storage.

Several methods of hashing have been developed. One attractive and inexpensive method is to multiply or weight each character of a key (VanWyk, 1988).

Hashing provides a way of finding the target sublist quickly where the record is localized by operating on its key (Flores, 1977). At the price of the small amount of space for pointers, a table of potentially unlimited size can be obtained by hashing methods (Vanwyk, 1988). Aho et al. (1983) thought that hashing was an important and widely useful technique for implementing dictionaries. With regard to the advantages of hashing, Standish (1980) stated: "hashing methods are not only good for tables stored in internal memory; they are also helpful for searching files of records stored on secondary memory devices such as disks and drums. When retrieving records from, say, a disk, whole groups of records can frequently be brought into primary memory at a time. Since it is relatively costly in time to move read/write arms on disks and to wait for rotational delays, it often pays to take care in computing a hash function since the extra cost of hash computation is often repaid by reducing costly mechanical repositioning and rotational delay."

According to Horowitz and Sahni (1983), hash table data structures can be illustrated as in Figure 1. Suppose we have a total number of 8 information records. Each record stores some information about each of 8 provinces in Canada.

Figure 1 An example of a hash table. The hash table is partitioned into 26 slots. Some slots are occupied by provincial information. The rest indicated by 0 are empty.

Assume that the identifiers for the 8 records happen to be the provincial names: Alberta, British Columbia, Manitoba, Nova Scotia,

Ontario, Prince Edward Island, Quebec, and Saskatchewan. We define an integer array TABLE, which is actually a hash table in memory. This hash table is partitioned into 26 slots. The hash function *f* which is to be chosen must map each of the 8 identifiers into one of the numbers 1-26. If the internal binary representation for the letters A-Z corresponds to the numbers 1-26 respectively, then the function *f* defined by: *f* (identifier) = the first character of 8 identifiers; will hash all 8 identifiers into the hash table. The identifiers Alberta, British Columbia, Manitoba, Nova Scotia, Ontario, Prince Edward Island, Quebec, and Saskatchewan will be hashed into slots 1, 2, 13, 14, 15, 16, 17 and 19 respectively by this function.

Now, assume that we want to add two more records into this table, say, to store Newfoundland and New Brunswick information records into the table. The identifiers Newfoundland and New Brunswick will also be hashed into slot 13 by the function (see Figure 2). In Figure 2. we can see that the three identifiers, Nova Scotia, Newfoundland, and New Brunswick are mapped into the same slot. This is called collision according to some text books. This causes a problem because for each slot only one record can be stored. In this study, a linked list data structure must be introduced to solve this problem.

4.2 LINKED LIST

Brillinger and Cohen (1972) defined a linked list as a data structure composed of numerous items called nodes or records, each containing several fields. A field may contain a primitive

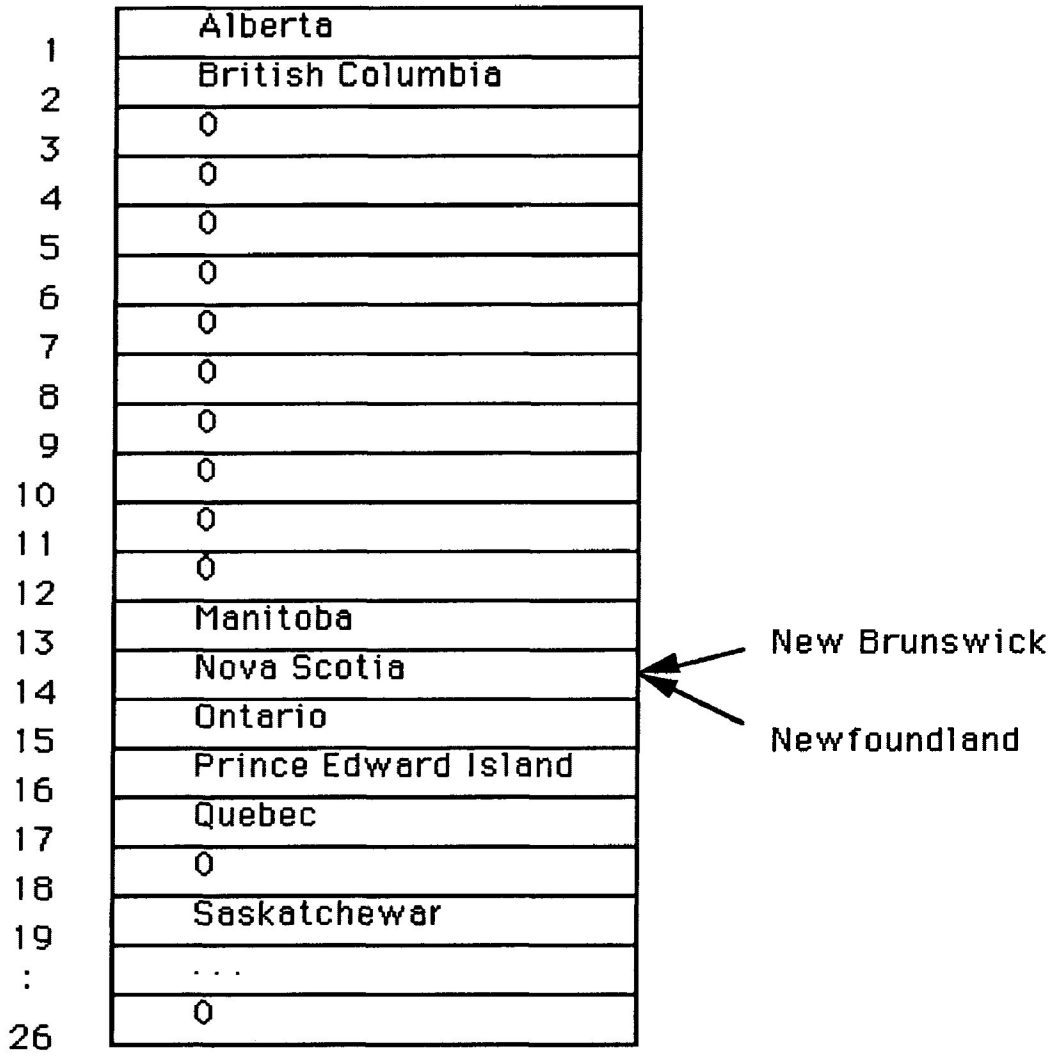


Figure 2. An example of occurrence of a collision. The hash table is partitioned into 26 slots. The three records with the identifiers, Nova Scotia, Newfoundland, and New Brunswick, are hashed into the same slot numbered 13.

data item. According to Flores (1977), these nodes or records were not necessarily physically in consecutive memory locations, but they were logically linked together. He stated that each node contained

a pointer to the next record according to the order relation on their key field(s).

Standish (1980) described a linked list as a method which "... provides a natural way of allocating storage for cyclic and re-entrance lists, and provides allocation for pure lists that conveniently accommodates growth and decay properties, as well as certain natural traversals of the elements".

A linked list was considered to be an excellent solution to the collision problems since the linked list was composed of nodes whose keys hash to the same value, chains never coalesced (VanWyk, 1988). No matter how small we make the hash table, the number of nodes that can be stored in it is limited only by the amount of memory that can be allocated dynamically (Flores, 1977). At the price of some space (the size defined as 4097 in this study) for pointers, we obtain a table of potentially unlimited size that readily supports insertions (VanWyk, 1988).

In comparing to a linked list with an ordered list, Aho et al. (1983) pointed out that ordered list implementation wasted space because it occupied the maximum amount of space independent of the number of nodes actually on the list at any time. In contrast, a linked list implementation employed only as much space as was required for the nodes currently on the list.

On this issue, Horowitz and Sahni (1983) remarked that "by storing each list in a different array of maximum size, storage may be

wasted". They also thought that operation such as insertion on the ordered lists was expensive. They also pointed out that unlike an ordered list where successive items of a list were located a fixed distance apart, in a linked list these items might be placed anywhere in memory.

Flores (1977) thought that a linked list had an advantage over an ordered list: "it is easy to append new records and to delete expired records; it is also easy to search a linked list sequentially.". Furthermore, he pointed out that when insertion operations were needed there would be trouble with an ordered list in which a lot of information movement would need to be done to place a new record into the file.

Figure 3 illustrates Standish's (1980), linked list data structure. Each cell has two fields, an INFO field containing some information which is to be stored, and a LINK field containing an address of another cell. The LINK field of the last cell contains the distinct quantity often defined as zero, which denotes the address of the empty list (END defined by zero here). It should be stressed that all cells are linked logically, not necessarily physically.

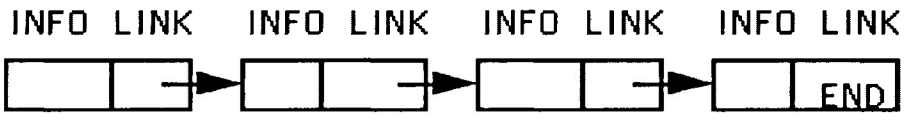


Figure 3. An example of a linked list. Each cell or node consists of two parts, INFO and LINK. INFO's contain information to be stored and LINK's contain addresses indicated by arrows.

4.3 COLLISION-RESOLUTION POLICY

There are three collision-resolution policies: (i) chaining (a synonym for a linked list), (ii) the use of buckets, and (iii) open addressing. Chaining, or a linked list, is viewed as a desirable data structure to solve a collision problem in hashing table methods (Hutchison, 1988), since the linked list is composed of nodes; chains in this way never coalesced (VanWyk, 1988). Standish (1980) also suggested that the chaining method was a better choice.

A linked list data structure can be introduced to solve the collision problem above. The technique of solving the collision is that whenever a collision happens a linked list will be created. In Figure 4, the linked list was created to chain Nova Scotia, Newfoundland, and New Brunswick.

4.4 DEFINITION OF A NODE

The data base file NEWTRIM.DAT consisted of the collection of all the tree nodes or records. Each tree record or node, which was named TREENODE.H file in the program, consists of the specific fields and was presented in the C language syntax in Figure. 5.

In Figure 5, a tree number was defined as *tn* which can be used with a plot label and plot number to retrieve any required records of the stem analysis trees. Tree age was defined as *age*. Age at DBH is defined as *dbh_age*. The last cookie age was defined as *last_ck_age*. The variables above were all declared as integers. DBH outside bark, defined as *dbhob*, and volume outside bark, defined as *vio*, were

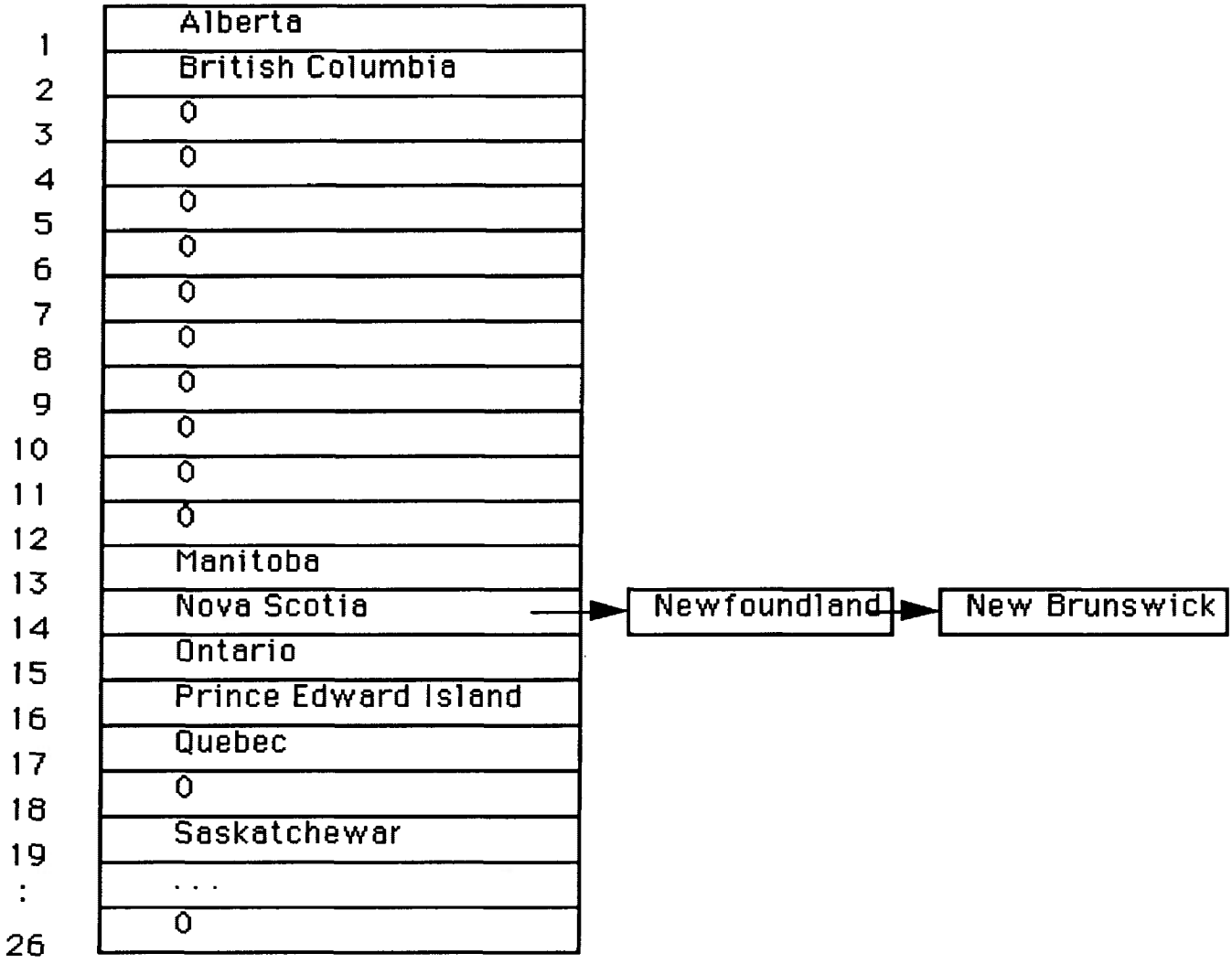


Figure 4. An example of solving a collision problem. The hash table is partitioned into the 26 slots. The three identifiers, Nova Scotia, Newfoundland, and New Brunswick, which are hashed into slot 13, are linked together, that is, the13th slot of Nova Scotia contains the address of Newfoundland which then contains the address of New Brunswick.

```
struct tree_record
{
        char        plt_lab[ LAB_LEN ];
        int         tn;
        char        sp_code[ 3 ];
        int         age;
        int         dbh_age;
        int         last_ck_age;
        double      dbhob;
        double      vio;
        double      dbhs[ MAX_YEAR_NUM ];
        double      hghs[ MAX_YEAR_NUM ];
        double      vols[ MAX_YEAR_NUM ];
        double      s_cor[ 3 ];
        double      c_cor[ 3 ];

        int         next;

};
```

Figure 5. The fields of a node, as a file called TREENODE.H in program, are shown in the C language syntax. The struct tree_record is the syntax name under which the first column is a type and the second column is the names of variables declared.

declared as real variable with double precision. A plot label was defined as *plt_lab[ ]* and a species code was defined as *sp_code[ ]*, both being character arrays. DBH defined as *dbhs[ ]* which would

store a cumulative DBH, tree height defined as *hghs [ ]* which would store a cumulative height, and volume defined as *vols[ ]* which would store a cumulative volume, were declared as real variable with double precision. The spatial locations of trees were stored in *s_cor[ ]* , double precision array for square plots, or *c_cor[ ]* , double precision array for circular plots. The oldest tree, denoted MAX_YEAR_NUM, was defined as *100* . Finally, a linked list pointer was defined as *next* , an integer, which can be used to chain the nodes with the same hash value. Each *Struct tree_record* or each node consumed *2496* bytes in the main memory.

4.5 HASHING FUNCTION

Under the existing TRIM data base system, the way that information for stem analyzed tree records was stored was related to the plot label, plot number, and tree number in the particular plot. For example, the information for tree number 9 within the plot number 1 in Kirkland Lake district was stored in such four separate files K1_9.OUT, AD_K1_9., RAD_K1_9., and ANNV_K1_9.. In this study, a plot label, a plot number, and a tree number were used as a key to calculate a hash value. This value was then used as an index into the hash table, which contained, at this table address, a pointer to the tree record location in the TRIM database. Thus, access to any tree record was intended to be direct. When presented with a plot label and tree number, we just applied the hashing function to create the number associated with that tree record, and proceeded directly to this table address. At this table address we would find a pointer to the tree record in the new TRIM database system.

In the hash function, the plot label, the plot number, and the tree number are simply treated as character strings. The hashing function is applied to sum their ASCII values or internal representations in some fashion. We can simply sum internal representation of plot label, plot number, and tree number to produce the hash value. For example, internal representation for plot label k was 107, internal representation for plot number 1 was 49, and internal representation for tree number 9 was 57. Applying simple hash function to sum their values, we get a hash value of 107 + 49 + 57 = 213. This simple hash function is considered to be a poor hash function, because it results in too many collisions. For example, the ASCII value was 107 for character k, 49 for character 1 and 50 for character 2. By applying the simple hash function to sum ASCII values of file names k1_2 and k2_1, we get hash values of 107 + 49 + 50 = 206 and 107 + 50 + 49 = 206 respectively. Both the file names are hashed into the same slot.

According to Hutchison(1988), when we design hash functions to perform address calculation we must consider that the hash function to be used has less possibility to produce the same hash value. According to VanWyk (1988), we should balance carefully between two desirable properties: a hash function should randomly distribute keys over the table well, and should also be inexpensive to compute.

In this study, the hash function by VanWyk (1988) was used. His hash function has two properties, that is, spreading keys over the hash table well, and also inexpensive to compute.

If w was a string, let $w(i)$ be the i(th) character of w, for $0 <= i < |w|$, and k is a power of $(i + 1)$. The general form of VanWyk's (1988) hash function is described as:

$$h_k(w) = \Sigma_{i=>0} (i + 1)^k * w_i \qquad [4.1]$$

Let $k = 0$, function $h_o(w) = \Sigma_{i=>0} w_i$ simply adds the characters in a string w; this is not likely to be a good hash function, since three letter string of TRIM file names could hash to only a small number of unique values causing a large number of collisions. For example, the hash value for tree number 9 within the plot number 1 in the plot label k was 107 + 49 + 57 = 213, while the hash value for the tree number 8 within the plot number 2 in the plot label k was 107 + 56 + 50 = 213. But if we Let $k = 1$, function $h_1(w) = \Sigma_{i=>0} (i + 1) * w_i$ weighted each character by its position in w; two character strings that are permutations of the same set of letters could get different hash values under $h_1()$. Function $h_2(w) = \Sigma_{i=>0} (i + 1)^2 * w_i$ weighs each character by the square of its position in w.

4.6 LOGIC OF COMPUTER PROGRAMS

Two computer programs written in C language were developed (see APPENDICES I and II). The strategy of development of these two programs was to minimize interface between computer and user, hiding all the intermediate procedures from a user.

The first program called TRIMHASH.C consisted of 29 modules or source files. Each source file is comprised of a certain number of

functions and further, within each function there are sub-functions within which there are sub-sub-functions, and so on. When the program is executed, functions will call their sub-functions and sub-functions will call their sub-sub-functions and so on.

The logic of this program in C language format is as follows:

```
main()
{
      prompt for input of plot label
      prompt for tree number
      for( plot label NOT END )
      {
         for( tree number NOT END )
         {
              file names building
              open necessary files
              read hash table or create hash table file
              process data
              install nodes
              add nodes to TRIM data file
              close files
         }
}
```

What this program expected was just two pieces information; plot label and number of stem analyzed trees in the plot. After such information was given the program would call its 29 modules one by one, within which one function would call another, to process the

TRIM data and would write the results onto the target file, NEWTRIM.DAT.

The logic of this program in the C language format is presented as follows:

```
main()
{
      while( look next tree record NOT END )
      {
            get plot lable and tree number
            open necessary files
            read hash table file
            search for a required record
            print tree record
            get next command from user
      }
      close files
}
```

This program will keep asking you if you would like to view the next record. Each time the program receives responses from the user it will show the record on the screen until the program receives a "No" response.

4.7 IMPLEMENTATION

The hashing algorithm developed in this study would process the TRIM data to calculate cumulative volume growth, cumulative height growth, and cumulative DBH growth for each individual tree by one-

year intervals. To add each tree record to the TRIM data file, called NEWTRIM.DAT, a plot label and a tree number for that tree is first passed through the hashing function, which translated the plot label, the plot number, and the tree number into an offset of the hash table. If this location in the hash table is empty, the tree information and its associated information is added to the end of the TRIM data file NEWTRIM.DAT and its location is placed into the hash table. However, if this location in the hash table is occupied, then another tree record had already been hashed into this table location and a "collision" occurs. In this case, the current pointer in the table is replaced with a pointer to the new structure. The new tree record and associated information is then added to the end of the TRIM data file. The address of the record that had been in the hash table is placed into the next field of the new record. Thus, the linked list will effectively extended by "bumping" all entries down the chain and placing the new tree record at the beginning of the corrected hash list.

When searching for a tree record we proceed to its table address. If the tree record pointed to by the address in the table did not match the target tree record by matching fields which were defined as plot label and tree number, we proceed down the chain until we find the required tree record, or come to the end of the TRIM data base, whichever comes first. It is guaranteed that if the tree record existed in the data base it would be on the selected list.

As an example, the interrelations between the hash table, the database file, and the linked lists are presented in Figure 6. In this

example, the load factor 0.1 was chosen and the hash function [4.1] was used where the power of k was defined as 3.

In Figure 6, for tree number 1 with plot label k1, by applying the hashing function, the hash value was 1822 and the hash value was 1849 for tree number 2. Therefore, the 1822th slot contained the address of the tree number 1 from plot k1 in the database file NEWTRIM.DAT and the 1849th slot pointed to the location of tree number 2 in plot k2 in the database file NEWTRIM.DAT. For the tree number 36 of the plot label n12, by applying the hashing function, the hash value was 47. Unfortunately, by applying the function, the hash value for the tree number 49 of plot label n117 was 47 too. Since both had the same hash value, a "collision" occurred.

Collision is solved as follows: the 47th slot in the hash table pointed to the location numbered 109 (a record number or node number) in NEWTRIM.DAT file; this location not only contained the information of the tree number 49 of the plot n117, but also the location, numbered 96, of the record of the tree number 36 of the plot n12. If more collisions occur, the linked list would be formed exactly in the same way and the nodes with the same hash value would be chained.

4.8 ACHIEVEMENTS

The 543 stem analysis trees, or a total of 543 * 3 = 1629 files, were processed by the hashing algorithm, TRIMHASH.C, with the hashing function [4.1] where the power of the hashing function was set to 3 and the load factor 0.1 was chosen. When executing the

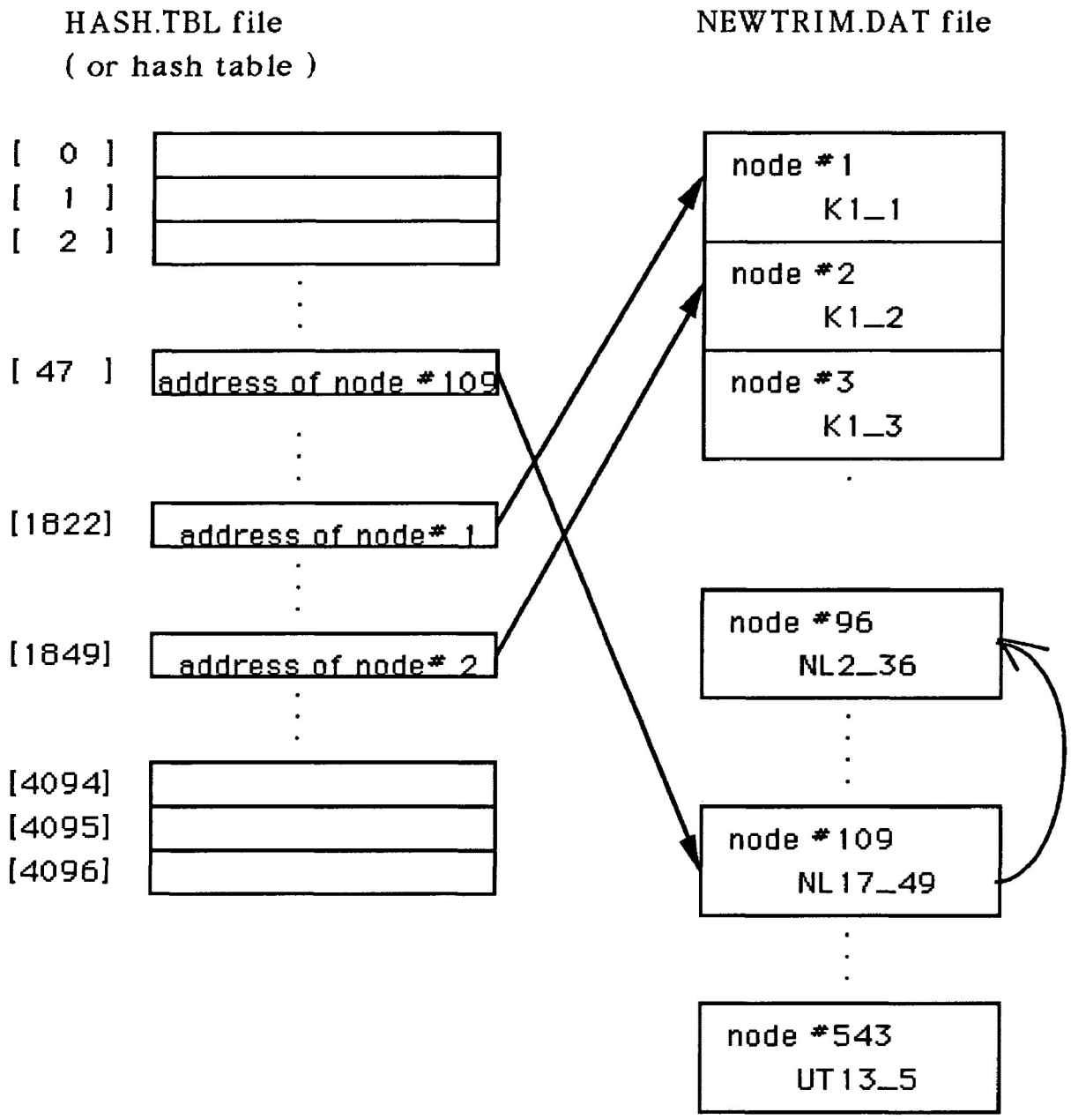


Figure 6. The interrelations between the hash table, the database file, and the linked lists. The left hash table called HASH.TBL is partitioned into 4097 slots.The right called NEWTRIM.DAT is output file. The chain, the 48th slot of the hash table, node# 109 and node No.96 is the linked list. The load factor is 0.1 and the hash function is: $h_3(w) = \Sigma_{i=>0} (i+1)^3 * w_i$ where w (i) is the i(th) character of words w for $0 <= i < |w|$.

program TRIMHASH.C, the user would only need to enter the following information; (1) the plot label, and (2) the total number of trees in that plot. Then, the computer would run the program TRIMHASH.C and would report that the execution was successful. Obviously, this simple executing process greatly reduced the chance of occurrence of errors due to much interface between the user and the computer.

The value of 1.0752 of ALOSS was achieved by this hashing algorithm. It could be interpreted that only one comparison would be made on average to retrieve or visit any of the records of stem analysis trees processed by the hashing algorithm. After processing, any of the records of the stem analysis trees could be retrieved by the other program, PRINT.C. What the PRINT.C program expected was just two pieces of information which the user had to input from the key board: (1) plot label, and (2) tree number which the user would like to retrieve. As an example, the output of one record by the PRINT.C program is shown in Table 5.

After the TRIM data files are processed by the hashing algorithm developed, the working directory no longer necessarily consisted of subdirectories in order to accommodate hundreds of TRIM data files. Instead, there were only two files in a working directory. One was the TRIM database file defined as NEWTRIM.DAT and the other was hash table file defined as HASH.TBL. The directory structures, therefore, were greatly simplified, but the property of direct access to any required tree records was maintained. Both the

Table 5. The example of output of tree number 9 within plot 1 in Kirkland Lake district printed by the PRINT.C program. is shown. The first line includes plot label, tree number, and species code; the second line includes age; the third, fourth, and fifth lines include the cumulations of DBH, height, and volume respectively at the age shown.

Plot label: k1 Tree number: 9 Species code: Pj

Age:	21
DBH(m):	1.341100
Height(m):	11.85000
Volume(m^3):	0.098708

Age	1	2	3	4	5
DBH	0.000000	0.000000	0.026000	0.049400	0.100800
Height	0.070000	0.520000	0.970000	1.400000	2.120000
Volume	0.000050	0.000155	0.000362	0.000777	0.001406

Age	6	7	8	9	10
DBH	0.167800	0.267800	0.392100	0.507100	0.625900
Height	2.370000	2.750000	3.250000	4.750000	5.100000
Volume	0.002565	0.004366	0.006640	0.009833	0.013831

Age	11	12	13	14	15
DBH	0.733100	0.824600	0.894600	0.973000	1.032900
Height	5.750000	6.620000	6.870000	7.750000	8.250000
Volume	0.018572	0.024036	0.031398	0.038513	0.045367

Age	16	17	18	19	20
DBH	1.088300	1.139200	1.186100	1.241400	1.289700
Height	8.750000	9.670000	9.920000	10.520000	11.120000
Volume	0.052790	0.060550	0.070397	0.081352	0.090860

Age	21
DBH	1.341100
Height	11.850000
Volume	0.098708

---

directory structures under the existing TRIM database management and the directory structure after the TRIM data files were processed were shown in Figures 7 and 8.

4.9 ANALYSIS OF EFFICIENCY

In an analysis of efficiency for the hashing algorithm, the hash function [4.1] was used where let k = 1, k = 2, and k = 3 respectively. The two load factors 0.1, and 0.9 were chosen. According to VanWyk (1988), a load factor is a result of a number of items to be processed divided by the size of a hash table. The combinations of three forms of hashing functions with the two load factors were studied with the hashing algorithm to evaluate the efficiency of the hashing algorithm developed. The average length of successful search (ALOSS) by VanWyk (1988) was used to analyze the efficiency of the algorithm.

According to VanWyk (1988), the value of ALOSS measures the quickness of encountering any required records. When the value of the ALOSS is 1 it means that there would be no comparison to be made to encounter any required record. Therefore, any required record could be retrieved directly. When the ALOSS value was 2 or 3 it suggested that there would be 2 or 3 comparisons to be made on average to encounter any required tree record in the data base.

For example, suppose that there are 10,000 tree information records in the database, a value of ALOSS of a hashing algorithm developed is equal to 2. Therefore, there will be only 2 comparisons to be made on average to retrieve any records you would specify. In comparison, the 10,000 comparisons might be made to encounter the

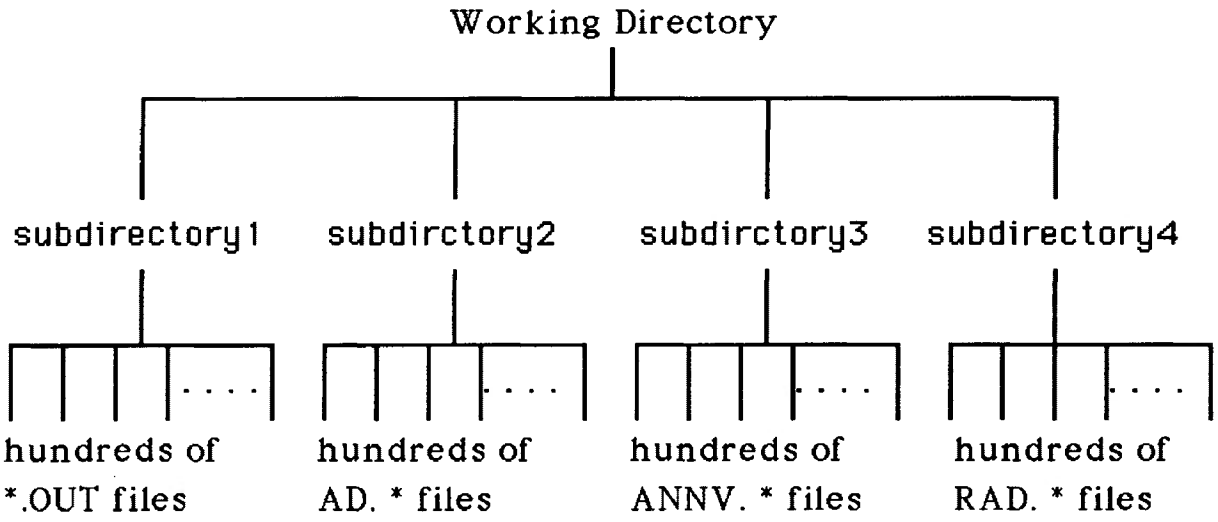


Figure 7. The directory structure under the existing TRIM data base management system.

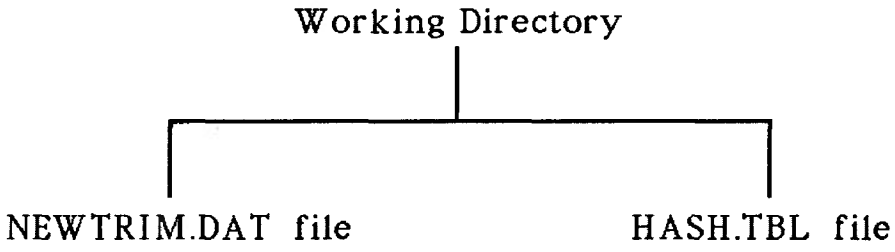


Figure 8. The directory structure after the TRIM data files were processed by the algorithm developed.

record which you would like to retrieve and which happens to be the last record in the database, if you would use the ordinary sequential search algorithm. In general, the bigger the value of ALOSS, the more comparisons would be made on average, or more time would be spent, to encounter any required tree record in the data base.

The statistics of the values of ALOSS are given in Tables 6, 7, 8, 9, 10, and 11. Based upon these statistics, Figures 9, 10, 11, 12 and 13 were plotted.

(i) When the hash table size was fixed.

When the hash table size was fixed or the load factor was set, the effects of the hashing functions on the goodness of performance of the algorithm could be explored. Figures 9 and 10 illustrated the effects of three forms of hashing functions on the goodness of performance of the hashing algorithm with the load factor 0.1 and 0.9 respectively. In Figure 9, when the power k was equal to 1 the values of ALOSS ranged between 3.1250 and 4.7619, the highest curve among the three curves. It suggested that there would be between 3 and nearly 5 comparisons to be made on average to encounter any required tree record. It was also noted that the values of ALOSS were sensitive to the number of trees. Therefore, this was a poor selection method. When the power of k was increased to 2 from 1, the values of ALOSS decreased dramatically. When the number of trees reached the maximum and k was equal to 2, the value of ALOSS was 1.5603, less than 2 comparisons to be made to encounter any tree record on average, compared with over a value of 3 of ALOSS when k was

Table 6. Statistics of hash values for jack pine trees in all plots in the Kirkland Lake District when the load factor is 0.9 and the hashing function is $h_1(w) = \Sigma_{i=>0} (i+1)^* w_i$

Number of trees	Number of unique No.	Average length of successful search
50	16	3.1250
100	21	4.7619
150	36	4.1667
200	49	4.0816
250	66	3.7879
300	81	3.7037
350	90	3.8889
400	101	3.9604
450	104	4.4269
500	109	4.5872
543	123	4.4146

Table 7. Statistics of hash values for jack pine trees in all plots in the Kirkland Lake District when the load factor is 0.9 and the hashing function is $h_2(w) = \Sigma_{i=>0} (i+1)^2 * w_i$

Number of trees	Number of unique No.	Average length of successful search
50	39	1.2821
100	70	1.4286
150	94	1.5957
200	116	1.7241
250	138	1.8116
300	165	1.8182
350	200	1.7500
400	236	1.6964
450	272	1.6544
500	310	1.6129
543	348	1.5603

Table 8. Statistics of hash values for jack pine trees in all plots in the Kirkland Lake District when the load factor is 0.9 and the hashing function is $h_3(w) = \Sigma_{i=>0} (i+1)^3 * w_i$

Number of trees	Number of unique No.	Average length of successful search
50	34	1.4706
100	62	1.6129
150	90	1.6667
200	118	1.6949
250	152	1.6447
300	183	1.6393
350	218	1.6055
400	251	1.5936
450	287	1.5679
500	321	1.5576
543	351	1.5470

Table 9. Statistics of hash values for jack pine trees in all plots in the Kirkland Lake District when the load 0.1 and the hashing function is $h_1(w) = \Sigma_{i=>0} (i+1) * w_i$

Number of trees	Number of unique No.	Average length of successful search
50	26	1.9230
100	47	2.1276
150	52	2.8846
200	80	2.5000
250	99	2.5252
300	122	2.4590
350	153	2.2876
400	175	2.2857
450	175	2.5714
500	186	2.6882
543	195	2.7846

Table 10. Statistics of hash values for jack pine trees in all plots in

the Kirkland Lake District when the load factor is 0.1 and the hashing function is $h_2(w) = \Sigma_{i=>0} (i+1)^2 * w_i$

Number of trees	Number of unique No.	Average length of successful search
50	32	1.5625
100	68	1.4706
150	113	1.3274
200	157	1.2739
250	201	1.2438
300	248	1.2097
350	296	1.1824
400	344	1.1628
450	387	1.1628
500	422	1.1848
543	461	1.1779

Table 11. Statistics of hash values for jack pine trees in all plots in the Kirkland Lake District when the load factor is 0.1 and the hashing function is $h_3(w) = \Sigma_{i=>0} (i+1)^3 * w_i$

Number of trees	Number of unique No.	Average length of successful search
50	41	1.2195
100	88	1.1363
150	137	1.0949
200	187	1.0695
250	235	1.0638
300	285	1.0526
350	334	1.0479
400	380	1.0526
450	427	1.0539
500	470	1.0638
543	505	1.0752

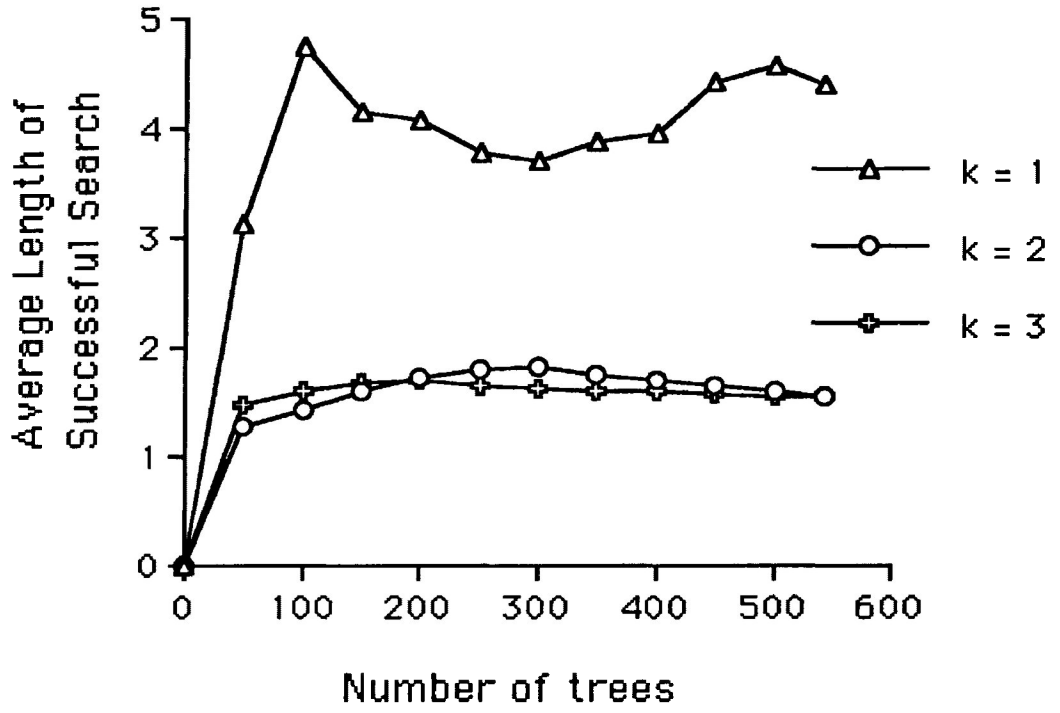


Figure 9. Comparison of the effects of the different forms of hashing functions on the algorithm performance when the load factor was 0.9.

equal to 1. When k was increased to 3, a further increase of 1 more unit, the performance of the hashing algorithm was improved only slightly.

In Figure 10, when the load factor was decreased to 0.1 from 0.9 and k was 1, the same trends were explicitly shown as when the load factor was 0.9 and k was equal to 1. The performance of the hashing algorithm in this case was still not satisfactory, nearly 3 comparisons to be made on average to retrieve any required tree record. It was noted that when k was equal to 2 and k was equal to 3 respectively,

the values of ALOSS were obviously improved and the latter was 1.5603 and the former was 1.5470.

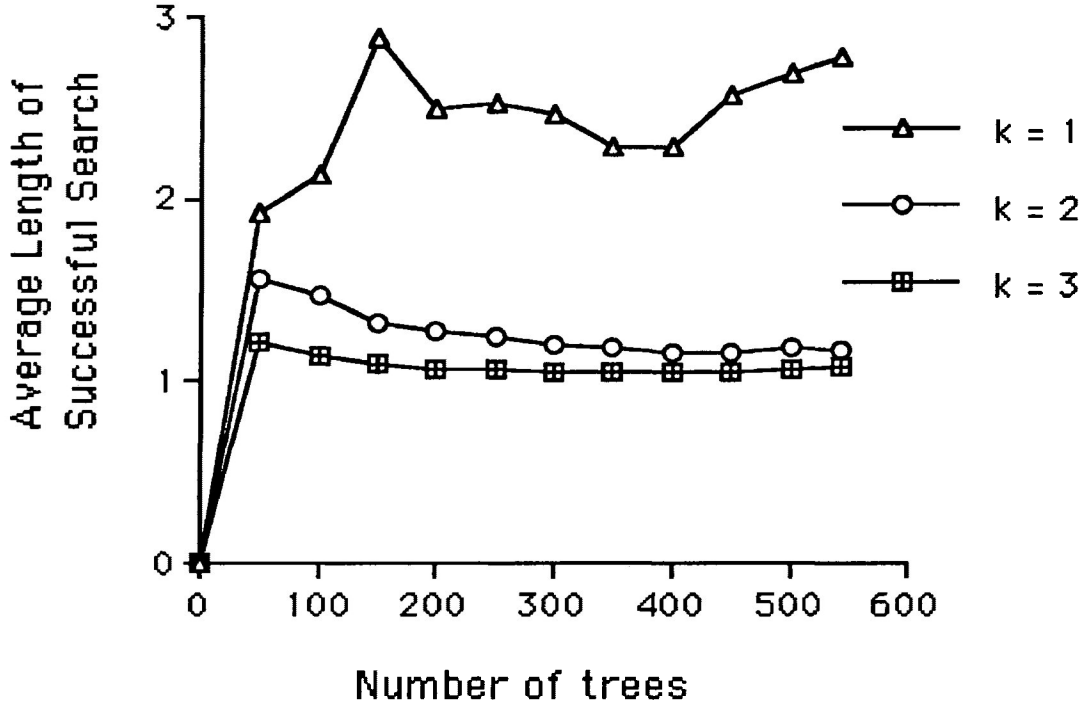


Figure 10.Comparison of the effects of the different forms of hashing functions on the algorithm performance when the load factor was 0.1.

(ii) When the power of the hashing function was fixed

When the power of hashing function was fixed the effects of the load factor on the performance of the hashing algorithm could be investigated. The effects of the two load factors on the goodness of the performance of the hashing algorithm are shown in Figures 11, 12 and 13.

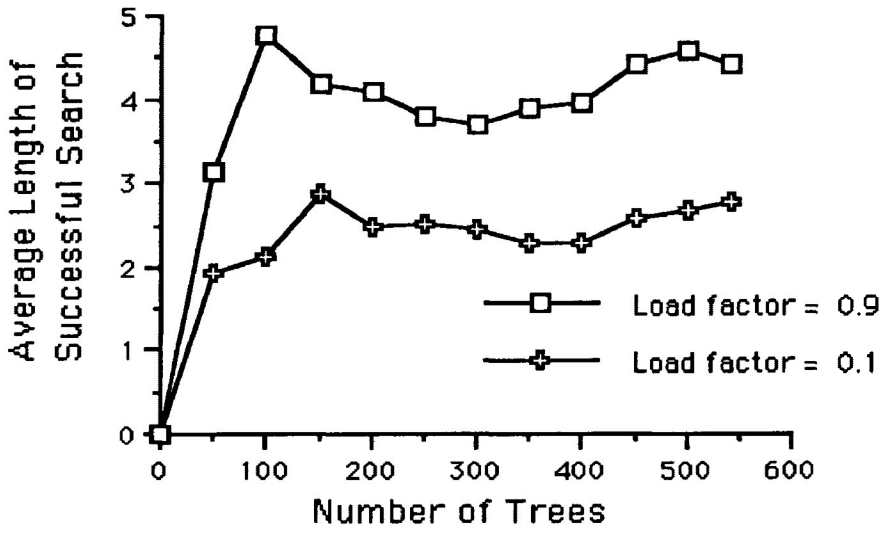


Figure 11. Comparison of the effects of the load factors on the algorithm performance when k was equal to 1.

In Figure 11, when k was equal to 1 the trends of the two lines tended to extend similarly. When the load factor was 0.9 most of the values of ALOSS fell between 4 and 5. While all the values of ALOSS were within 2 and 3, the former value of ALOSS were two units greater than the latter ones on average. It was apparent that the values of ALOSS were sensitive to the number of trees in data base.

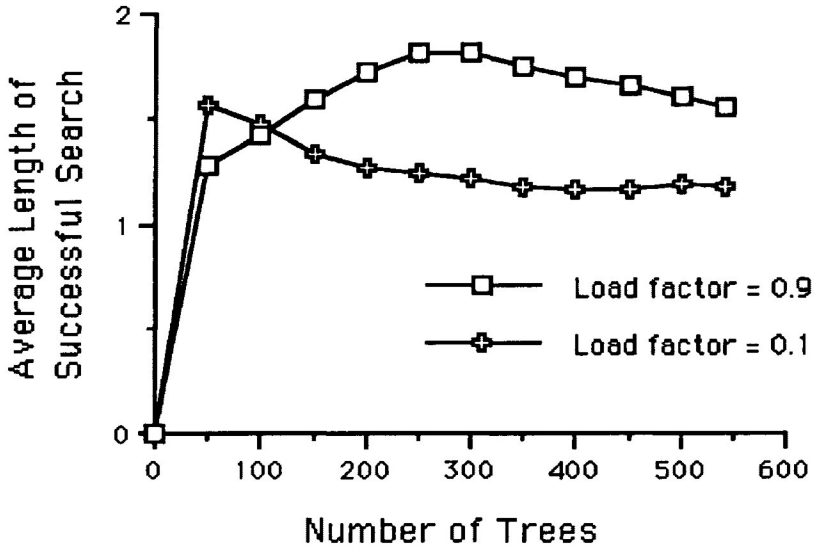


Figure 12. Comparison of the effects of the load factor on the algorithm performance when k was equal to 2.

In Figure 12 when the load factor was 0.9, almost all the values of ALOSS exceeded 1.5. When the load factor was 0.1, almost all the values were below 1.5. In the former case, it was obvious that the values of ALOSS were sensitive to the number of trees in data base. In the latter case, the sensitivity of the values of ALOSS to the number of trees appeared to be insignificant. Overall, there was approximately one unit of ALOSS difference between the two curves.

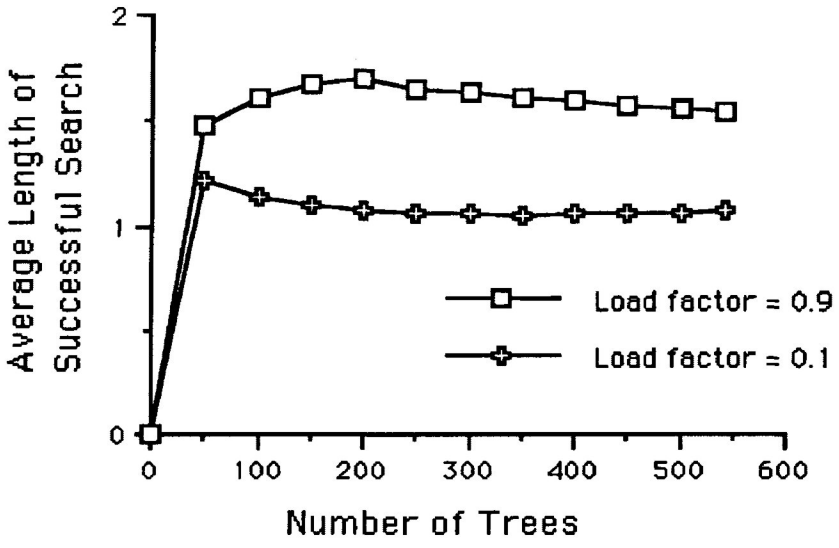


Figure 13. Comparison of the effects of the load factor on the algorithm performance when k was equal to 3.

In Figure 13 the performances of the hashing algorithm were further improved compared with that in Figure 12 in both cases. The values of ALOSS were no longer subject to the number of trees in the data base when the load factor was 0.9. Also, the overall values of ALOSS were lowered a bit compared to these in Figure 12. When the load factor was lowered to 0.1 from 0.9, the overall values of ALOSS were decreased by one unit, falling to about 1.2 on average.

5. TWO-STAGE SAMPLING

This section considers the question of determining what sample intensities of both plot samples and tree samples were required to obtain an accurate estimate of annual volume growth of immature jack pine. The yearly volume growth investigated was limited to the last 10-year period in this study. The two 2-stage sampling techniques were simulated with immature jack pine data collected in stands in the Kirkland Lake District in northeastern Ontario.

5.1 METHODOLOGY

The first sampling scheme was simple random sampling of the primary samples of fixed plots within individual stands and simple random sampling of the subsamples of trees within the plots at the second stage. This sampling scheme would be referred to as the 1st 2-stage sampling rule hereafter.

The second sampling scheme was simple random sampling of the primary samples of the plots within the individual stands and the selection of second-stage subsamples of the trees within the plots using the probability proportional to basal area. This sampling scheme is referred to as the 2nd 2-stage sampling rule hereinafter.

The function for computing total volume based upon the 1st two-stage sampling rule was given by Frayer (1979) and adjusted by Murchison (1984) as follows:

estimate of population mean;

$$VS1 = \overline{BA1} * \left[\sum_{i=1}^{n} M_i * \overline{Vi.} \Big/ \sum_{i=1}^{n} M_i * \overline{BA2i.} \right] \qquad [5.1]$$

where; $\overline{Vi.} = \frac{1}{Mi} * \sum_{i=1}^{Mi} Vij$

$$\overline{BA1} = \frac{1}{N} * \sum_{i=1}^{N} \frac{1}{Mi} * \sum_{j=1}^{Mi} BAij$$

$$\overline{BA2i.} = \frac{1}{Mi} * \sum_{j=1}^{Mi} BA2ij$$

and variance;

$$V(VS1) = \frac{1 - n}{N * n} * \sum_{i=1}^{n} Mi * (\overline{Vi.} - RVBA * \overline{BA2i.})^2/(n-1) + \frac{1}{N * n} * \sum_{i=1}^{n} Mi * (1 - \frac{mi}{Mi})/mi * S^2DV2i \quad [5.2]$$

where;

$$S^2DV2i = \sum_{j=1}^{mi} [DV2ij - \overline{DV2i.}]^2/[mi - 1]$$

$$\overline{DV2i.} = \frac{1}{mi} * \sum_{j=1}^{mi} DV2ij$$

$$DV2ij = Vij - RVBA * BA2ij$$

and $$RVBA = \sum_{i=1}^{n} Mi * \overline{Vi.} / \sum_{i=1}^{n} Mi * \overline{BA2i}$$

total volume estimate;

$$TVS1 = M * VS1$$

and its variance;

$$V(TVS1) = M^2 * V(VS1)$$

Where;

N = the number of primary sample units (plots) in the population (stand or hectare);

n = the number of sample units in the sample;

Mi = the number of sample elements in sample unit i;

mi = the number of elements sampled in sample unit i;

vij = the volume of the tree j in plot i;

bij = the basal area of tree j on plot i;

Vij = the volume of tree j in cluster i.

Vi. = total volume of all trees on secondary cluster i.

BAij = basal area of tree j in cluster i.

$\overline{Vi.}$ = mean volume per tree in second-stage sample for cluster i.

RVBA = the mean tree volume to tree basal area ratio.

$\overline{BA2i}$ = mean basal area of second-stage of cluster i.

Vk = total tree volume per unit area or per primary cluster as estimated by two-stage sample rule k.

DV2ij = the difference between actual tree volume and volume estimated by simulation for tree j on cluster i.

$\overline{DV2i.}$ = mean difference between actual and estimated

tree volume for cluster i.

S^2_{DV2i} = variance of DV2ik.

VSk = total tree volume per unit area or per primary cluster as estimated by two-stage sample rule k.

TVSk = estimated total volume of trees in stand by two-stage method k.

The functions for computing total volume based upon the 2nd two-stage sampling rule were adapted from Murchison (1984) and were given as follows:

estimate of population mean u;

$$u = \sum_{i=1}^{N} \left[\sum_{j=1}^{Mi} BAij/Mi * \sum_{j=1}^{Mi} [\, Vij/BAij \,] \right] / N \qquad [5.3]$$

where;

$$Vi. = \frac{1}{Mi} * \sum_{j=1}^{Mi} [\, Vij * (\sum_{j=1}^{Mi} BAij)/BAij \,]$$

variance for u;

$$V(u) = \frac{N - n}{N * n * (n - 1)} * \sum_{i=1}^{n} \left[\sum_{j=1}^{Mi} BAij/mi * \sum_{j=1}^{mi} Vij/BAij \right] \qquad [5.4]$$

$$- \frac{1}{n} * \sum_{i=1}^{n} \left[\sum_{j=1}^{Mi} BAij/mi * \sum_{j=1}^{mi} Vij/BAij \right]^2$$

$$+ \frac{1}{N * n} * \sum_{i=1}^{n} \left(\sum_{j=1}^{Mi} BAij \right)^2 * \frac{Mi - mi}{Mi * mi * (mi - 1)}$$

$$* \sum_{j=1}^{mi} \left[Vij/BAij - \frac{1}{mi} * \sum_{j=1}^{mi} Vij/BAij \right]^2$$

total volume estimate;

$$TVS2 = N * VS2$$

and its variance;

$$V(TVS2) = N^2 * V(VS2)$$

In this study, the basal areas converted from diameters at breast height were used as the means of selection of the subsamples. Using methodology described by Husch et al. (1972), a cumulative list of tree basal areas by tree number within the plot was made and a random selection of the trees to be subsampled was made from this list. All trees used are located within 10 m by 10 m plots and they were all stem analyzed in accordance with TRIM methodology.

5.2 DATA USED

Stem analyzed tree data from Kirkland Lake District were used to investigate the sampling intensities for jack pine growth. All plots consisted of a 20 m by 20 m measurement plot with a 10 m by 10 m destructively sampled inner plot. All the trees within the inner plot were stem analyzed according to TRIM procedures. The data information provided by Murchison and Kavanagh (1988) are given in Table 12.

5.3 COMPUTER SAMPLING SIMULATION

Simulation is a numerical technique for conducting experiments on a digital computer (Naylor et al., 1966) and it is commonly used by scientists (Kleijnen, 1974). Simulation can serve as a "preservice test" to evaluate the decision rules to avoid running the risk of experiments on the real system (Naylor et al., 1966).

According to Kleijnen (1974) there are two methods of problem solving in general. One is an analytical solution technique which relies on calculus. The other is a numerical solution technique which substitutes numbers for the independent variables and manipulates these numbers. He pointed out that the numerical technique was iterative, i.e., each step in the solution gave a better solution using the results from the previous steps. The numerical technique solved the problem by approximating the real state of nature (Arvanitis, 1966). The Monte Carlo method is a special numerical technique (Kleijnen, 1974).

Table 12. Summary of information for the immature TRIM jack pine plots sampled within the Kirkland Lake District

Plot Label	Sp	Age yr.	Dbh cm	Ht m	Vol m^3/ha	Density No./m^2	Pielou's Index	Site Class
K1	PJ	21	10.3235	10.86	54.151	0.2600	0.831405	1
K2	PJ	20	11.6786	10.22	62.349	0.2000	0.903702	1
K3	PJ	20	11.6352	10.33	62.867	0.1900	0.647512	1
NL2	PJ	19	7.6101	6.24	20.522	0.1591	4.990989	2
NL17	PJ	20	7.9851	7.56	27.731	0.1927	5.596667	2
NL27	PJ	19	6.6207	5.76	13.477	0.1409	4.575950	2
KLD_P1	PJ	21	9.2544	8.76	31.738	0.5100	1.026528	1
KLD_P2	PJ	22	7.9871	7.45	23.907	0.2900	1.206741	2
KLD_P3	PJ	21	7.5219	7.82	31.637	0.4100	1.130969	2
T1	PJ	28	15.2928	13.76	136.039	0.1448	0.772097	1
T2	PJ	28	12.4293	13.02	90.594	0.2641	0.637634	1
T3	PJ	27	15.9670	12.74	146.156	0.1228	0.826934	1
T4	PJ	28	11.0890	12.38	67.307	0.2983	0.797199	2
T5	PJ	28	13.1124	12.81	102.796	0.1739	0.995269	1
T6	PJ	26	14.4276	12.29	115.812	0.1353	0.765531	1
T7	PJ	27	19.2818	14.97	211.714	0.1166	0.790287	1
T8	PJ	28	16.3938	12.33	139.481	0.1589	0.779978	1
UT9	PJ	26	7.0484	7.42	18.323	0.3592	0.657926	3
UT10	PJ	28	13.3050	12.60	99.623	0.2798	0.684092	1
UT11	PJ	28	14.4960	13.23	121.155	0.2577	0.778577	1
UT12	PJ	28	9.2712	11.08	47.822	0.4995	1.481926	2
UT13	PJ	28	14.2218	13.87	127.080	0.1401	0.887776	1

Note: All information was calculated by Murchison and Kavanagh (1989).

The Monte Carlo method is also referred to as the method of statistical trials (Buslenko et al., 1966). Hammersley and Handscomb (1965) defined the Monte Carlo method as "that branch of experimental mathematics which is concerned with experiments on random numbers". Monte Carlo can be any technique for the problem solving using random numbers or pseudorandom numbers (Kleijnen, 1974). Monte Carlo methods have found wide application on digital computers (Buslenko et al., 1966).

In comparing mathematical models with simulation procedures, Ackoff (1962) remarked "that a model represents a phenomenon while simulation imitates it, the first being the 'photograph' and the second the 'motion picture' of the phenomenon in question.". Modelling deals primarily with the relationships between real systems and models; simulation refers primarily to the relationships between computers and models (Zeigler, 1976).

In this study, estimates for the two 2-stage sampling schemes were computed for each cluster of plots within stands. As controls, an estimate of stand yearly volume growth per hectare for the last 10-year period for all trees included on all plots within the stand were computed. The estimate of stand yearly growth volume per hectare was used as controls for comparison of the precision of the population mean estimates for the two 2-stage sampling rules.

For the 1st 2-stage sampling rule, subsampling of trees within plots was performed using simple random sampling for the selection of individual trees within plots and all the possible levels of the

subsampling of trees within the plots were simulated for the last 10-year growth period. The formula to compute the various estimators were given in expressions [5.1] and [5.2].

For the 2nd 2-stage sampling rule, the subsampling of trees within plots was carried out based upon probabilities proportional to tree basal areas. All the possible levels of subsampling of trees were simulated for the last 10-year growth periods. The formula to compute the various estimators were given in expressions [5.3] and [5.4].

In this study, the precisions of the mean estimates for both the unequal probability subsampling rule and the equal probability subsampling rule were computed and were used to evaluate the accuracy of the mean estimates for both sampling rules. A confidence limit of 95 percent was set throughout this study.

According to Statistics by Beijing Forestry University, (1977), the precision may be explained as follows: for example, u is a estimate mean, s is a standard deviation of n samples, t value can be determined given the confident limits, 1.96, for example, for a 95 percent of confidence limit, *estimate error limits* for a mean estimate is defined as t value multiplied by a standard error, $(t*(s/\sqrt{n}))$, *relative mean estimate error* (E) is defined as estimate error limits divided by a mean estimate, $E = (t*(s/\sqrt{n}))/u$. Finally, a precision (P) is defined as 1 minus a relative mean estimate error then multiplied by 100 percent.

$$p = (1 - E)*100\%. \qquad [5.5]$$

It can be interpreted that the higher the value of precision, the more accurate the sample mean estimate. The highest value of precision is 1 or 100 percent.

In this study, because of a small number of stem analyzed jack pine trees in stands T and UT (only 5 stem analyzed trees in each plot except plot T1 which had 13 trees), computer sampling simulation was only performed with three stands K, NL, and KLD_P. The confidence limits were set at 95 percent throughout the analysis of this study.

5.4 SIMULATION PROCESS

The simulation processes can be divided into the two major steps; database preparation and sample rules simulation. The hashing algorithm developed in the earlier section was used to process the TRIM database as the first step preparation of a smaller database. The file NEWTRIM.DAT and the file HASH.TBL produced by the developed algorithm were used as the part of the sampling simulation program. The second step, sample rules simulation, only included one large program, SIMULATION.C. This program consisted of 41 modules or source files which were composed of a total of 78 functions all together under the main program. The main controlling function *main()* coordinated all 78 functions to be executed as designed.

At the very beginning of the simulation, the program expected a plot label and the number of primary sample units, the number of subsample units and the total number of stem analyzed trees in each of the plots. The program after receiving such information would build up the names for all trees. Then, the names were immediately used to calculate their hash values in order to retrieve the required information stored in the TRIM data base file NEWTRIM.DAT.

The minimum subsample number was defined as 2 in the source file DEFINE.H. The maximum number of subsample size was 1 less than the total number of jack pine trees in the plot that had the least number of jack pine trees among the primary sample plots. The source file FINDMAXNUM.H was designed to compute the total number of jack pine trees in each plot and then set the maximum subsample number of trees. To speed up the simulation process the quick sort algorithm and the recursive call were introduced into the the simulation program. Under the computer sampling simulation loop the rule of two-stage random sampling with simple random subsampling was run first and then was followed by the rule of 2-stage random sampling with unequal probability of selection of subsample.

The most outer loop was the yearly volume growth loop. The yearly volume growth to be investigated was limited to the last 10 years. It was defined as MAX_YEAR in the file DEFINE.H. The next enclosed loop was subsample size loop. As mentioned, the maximum subsample was decided by the function FINDMAXNUM. The inner most loop was the sampling simulation loop.

Each plot was repeatedly sampled by each simulator in the following manner. All estimates such as estimate for population mean and estimate for its variance from the current cycle were combined with those of all previous cycles run under the same factors and were averaged at the end of each cycle. The sampling simulation estimates of the volume growth per hectare and standard error estimates for the two two-stage sampling rules were computed from these estimates. The simulator was possibly run up to 2500 times which was defined as SIMULATION_TIMES in the file DEFINE.H or until stable estimates were obtained, whichever came first. The source file STABLETEST.H was designed to evaluate the difference between the current estimates average and the previous estimates average and the standard of evaluating the difference was defined as 0.001. This was defined as ALLOWABLE_ERROR in file DEFINE.H. After the 2500 loops ended or the stability test was satisfied, the results of the sampling rules simulation were written to the external file, SIMULATION.OUT. This process was run for all possible levels of subsampling for TRIM plots within a stand. The results were appended to the external file after each run ended.

5.5 SAMPLING SIMULATION RESULTS

Due to the huge sampling simulation results, the way to present the simulation results is explained as follows: to evaluate both sampling rules and draw reasonable conclusions, only results from the worst cases for the unequal probability subsampling rule and the results from the best cases for the equal probability subsampling rule are presented in this part. The final conclusions are drawn in such a

way that if the results from the worst cases are satisfactory, the method to produce the results can be considered to be adopted; if the results from the best cases are not acceptable, the method which produces the results will be not recommended.

Under each of both primary sample intensities of 100 percent and 66 percent, for the unequal probability subsampling rule, the worst cases were selected from the simulation results with the lowest precision of mean estimates when subsample size was equal to 2, and for the equal probability subsampling rule, the best cases were selected from the simulation results with the highest precision of mean estimate when subsample size was equal to 2.

All the TRIM means lie within the mean estimate ranges produced by both sampling rules with confidence limit of 95 percent. There were no significant differences to be found between TRIM means and the estimate means through the analysis of the simulation results for both sampling rules. Therefore, the focus of the investigation was placed on comparison of precisions of mean estimates produced by both sampling rules.

(1) Under the primary sample intensities of 100 percent

(i) When the subsample intensities were 10 percent both results from the worst case for the unequal probability subsampling rule and results from the best case for the equal probability subsampling rule are presented in Figure 14.

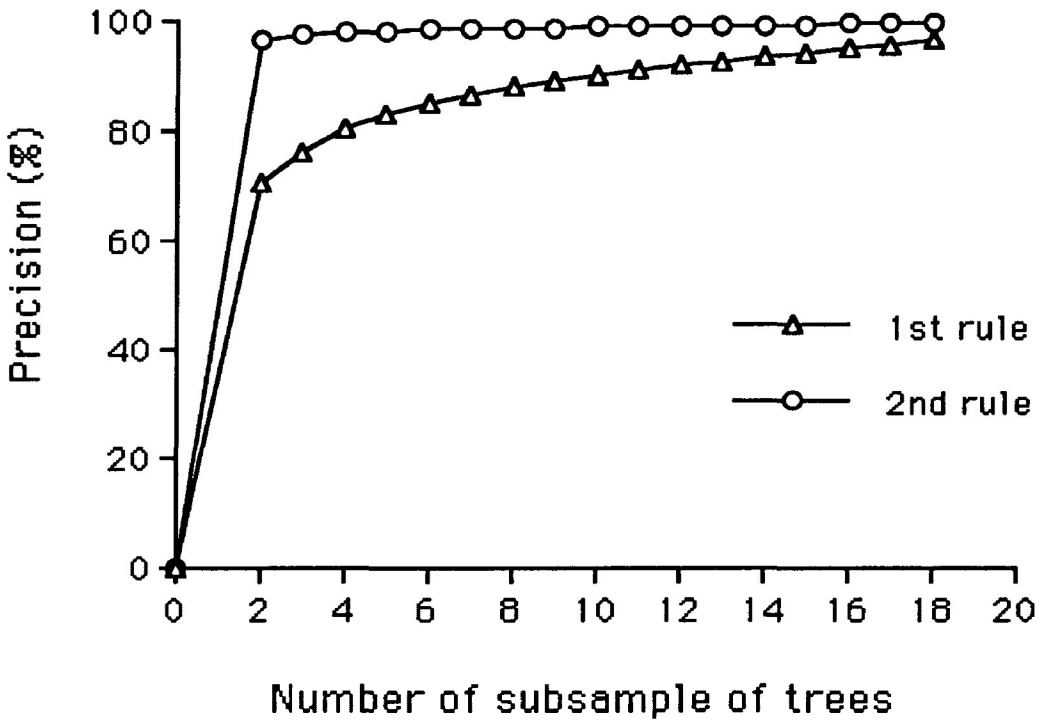


Figure 14. The worst results produced by the unequal probability subsampling rule and the best results produced by the equal probability subsampling rule when the primary sample intensities were 100 percent and subsample intensities were 10 percent.

In this case, for the unequal probability subsampling rule the precision of the mean estimate was 96 percent, while for the equal probability subsampling rule the precision of the mean estimate was only 76 percent. With an increase of the subsample size or subsampling intensities the precision of mean estimate for the equal probability subsampling appeared increasing. For this sampling rule when subsample size was increased to 10 trees from 2 trees the

precision rose up to 91 percent. If the precision of mean estimate for the equal probability subsampling climbed up to the point which can be reached for the unequal probability subsampling rule with only 2 trees, the subsample size should be further increased to 17 trees for the equal probability subsampling.

(ii) When the subsample intensities were 5.8 percent both result from the worst case for the unequal probability subsampling rule and result from the best case for the equal probability subsampling rule are presented in Figure 15.

In this case, the precision of mean estimate for the unequal probability subsampling rule was still high, 97 percent. To obtain over 90 percent of the precision of mean estimate for the equal probability subsampling rule the subsample intensities must be increased to 66 percent.

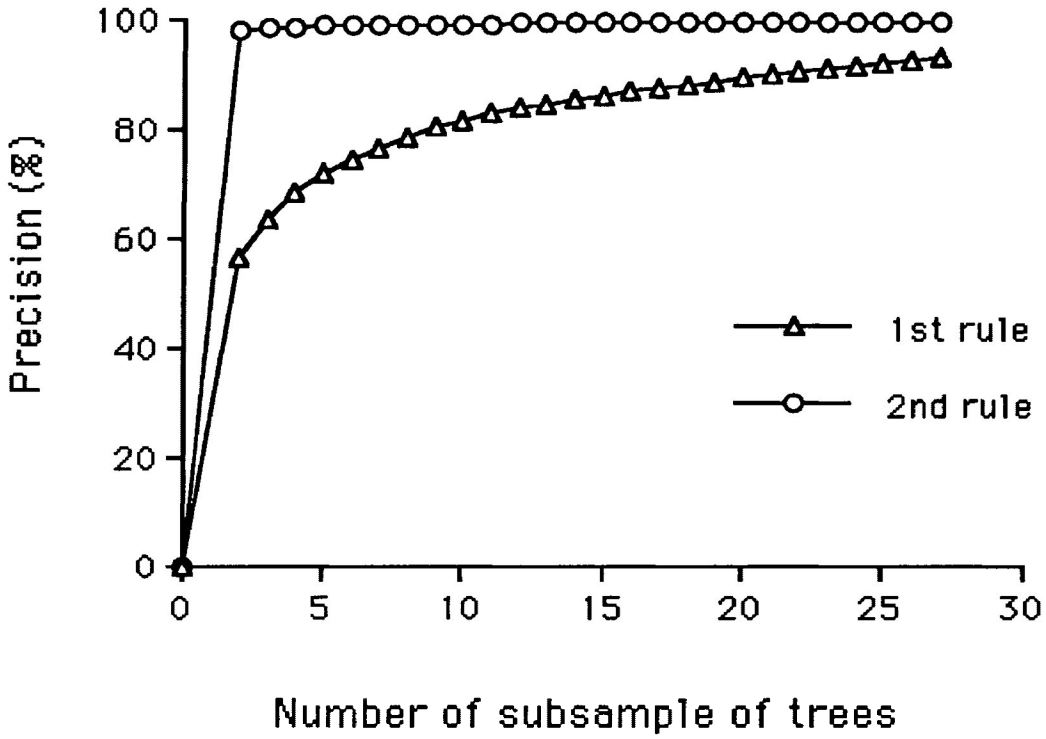


Figure 15. The worst results produced by the unequal probability subsampling rule and the best results produced by the equal probability subsampling rule when the primary sample intensities were 100 percent and subsample intensities were 5.8 percent.

(iii) When the subsample intensity ratio at 3.1 percent both results from the worst case for the unequal probability subsampling rule and result from the best case for the equal probability subsampling rule are presented in Figure 16.

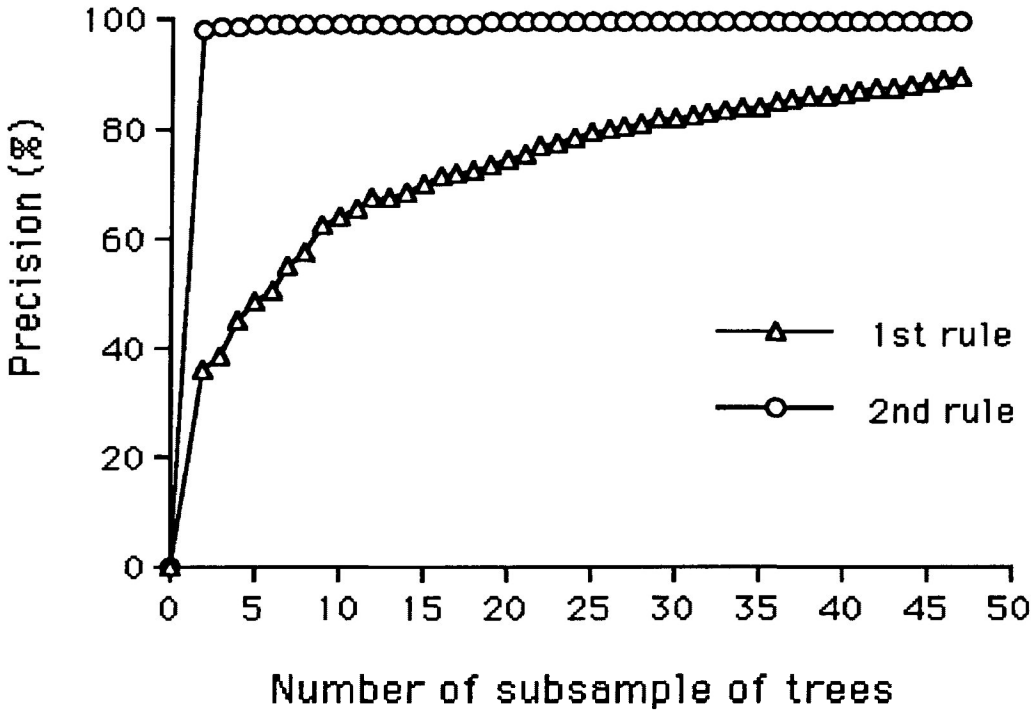


Figure 16. The worst results produced by the unequal probability subsampling rule and the best results produced by the equal probability subsampling rule when the primary sample intensities were 100 percent and subsample intensities were 3.1 percent.

In this case, the precision of mean estimate for the unequal probability subsampling rule remained over 90 percent, compared with only 36 percent of the precision of mean estimate for the equal probability subsampling rule. It is noted that for the equal probability subsampling rule the 90 percent of the precision would still not be obtained even when the subsample intensities rose to 75 percent.

(2) Under the primary sample intensities of 66 percent

(i) When the subsample intensities were 6.7 percent both results from the worst case for the unequal probability subsampling rule and

result from the best case for the equal probability subsampling rule are presented in Figure 17.

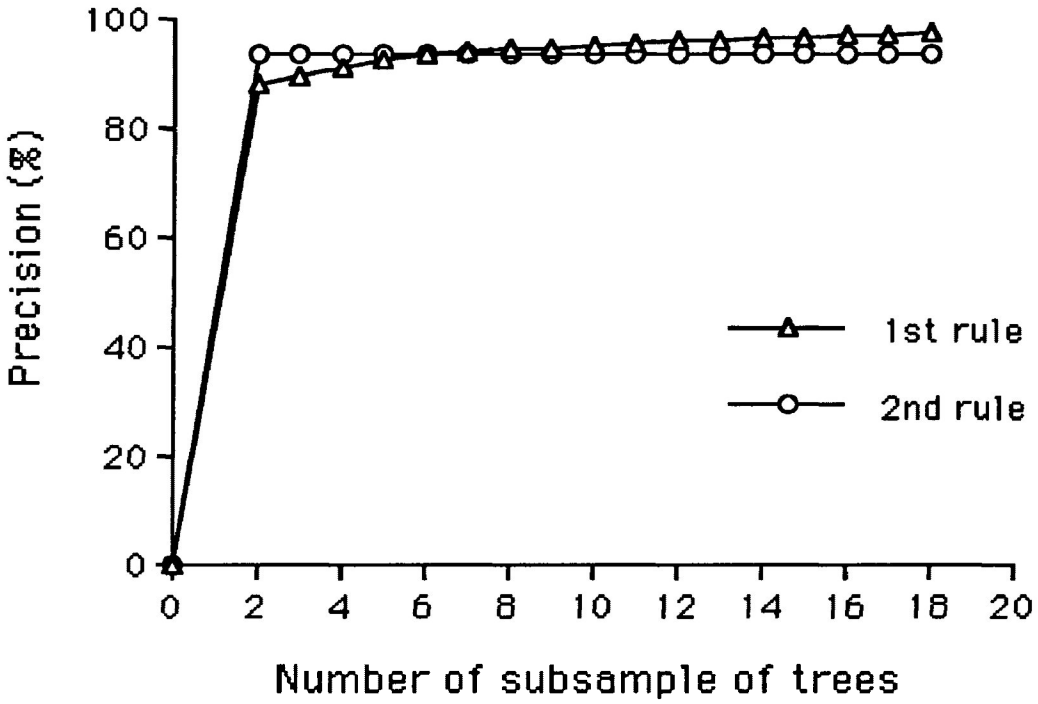


Figure 17. The worst results produced by the unequal probability subsampling rule and the best results produced by the equal probability subsampling rule when the primary sample intensities were 66 percent and subsample intensities were 6.7 percent.

In this case, the precision of mean estimate for the unequal probability subsampling rule remained well over 93 percent. While, the precision of mean estimate for the equal probability subsampling was 88 percent and was sharply increased with an increase of subsample intensities.

(ii) When the subsample intensities were 3.8 percent both results from the worst case for the unequal probability subsampling rule and result from the best case for the equal probability subsampling rule are presented in Figure 18.

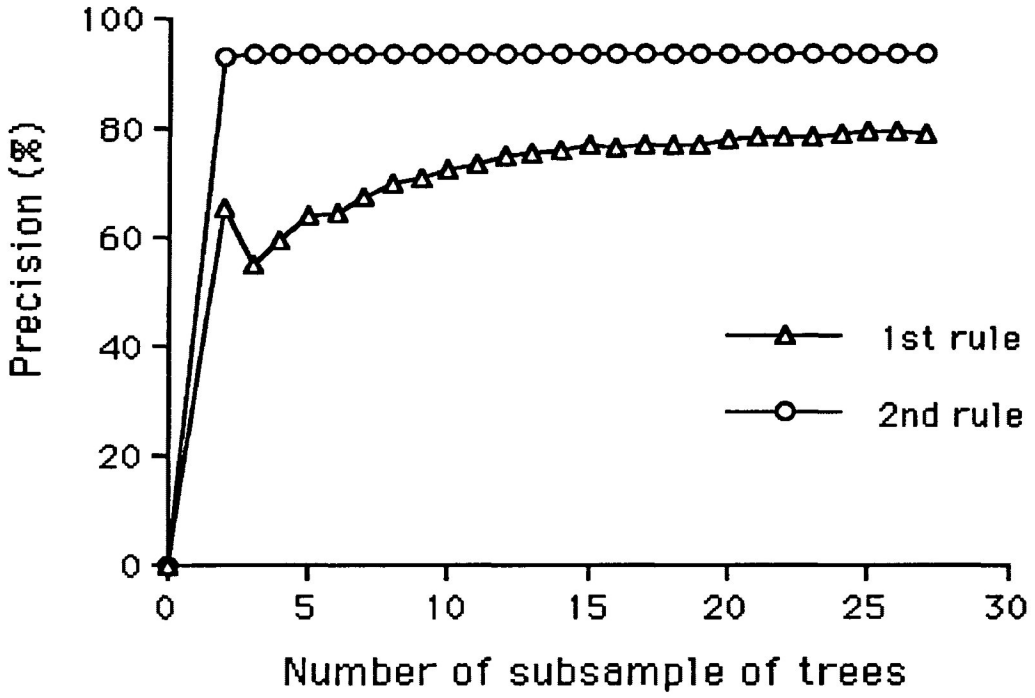


Figure 18. The worst results produced by the unequal probability subsampling rule and the best results produced by the equal probability subsampling rule when the primary sample intensities were 66 percent and subsample intensities were 3.8 percent.

In this case, the precision of mean estimate for the unequal probability subsampling rule was 92 percent, compared with 55 percent for the equal probability subsampling rule. For the equal probability subsampling rule, even when the subsample sizes were increased to the maximum, the precision for this sampling rule was still below 80 percent.

(iii) When the subsample intensities were 2.1 percent both results from the worst case for the unequal probability subsampling rule and result from the best case for the equal probability subsampling rule are presented in Figure 19.

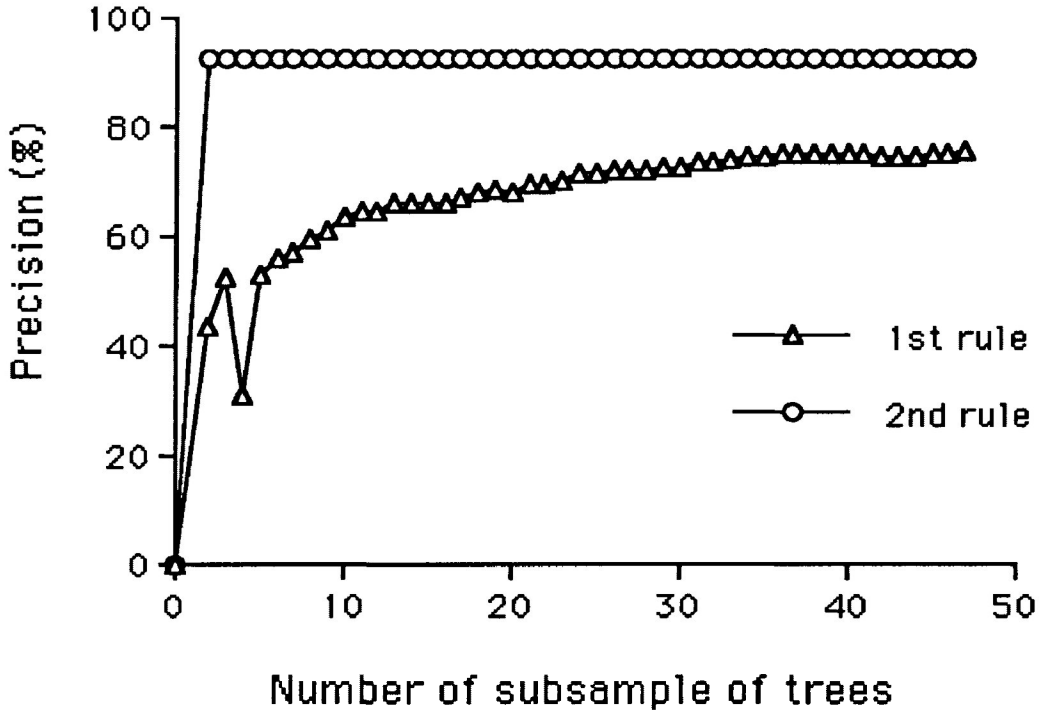


Figure 19. The worst results produced by the unequal probability subsampling rule and the best results produced by the equal probability subsampling rule when the primary sample intensities were 66 percent and subsample intensities were 2.1 percent.

In this case, the precision of mean estimate for the unequal probability subsampling rule was 88 percent, a little below 90 percent. When the subsamples were increased to 3 from 2, or to subsample intensities of 3.1 percent from 2.1 percent , 90 percent of precision was secured. The precision of mean estimate for the equal probability subsampling was very poor, only 43 percent. When the subsample intensities rose to the maximum subsample size, that is, subsample of 47 trees per plot, the precision was still below 76 percent.

6. DISCUSSION

6.1 THE HASHING ALGORITHM

In this study, all the combinations of two hash table sizes with the three forms of hashing function [4.1] were studied to evaluate the efficiency of the hashing algorithm. The results of the value of ALOSS and their trends were consistent throughout the analysis. The combinations of the further increase of the power of the hash function [4.1] and the further decrease of load factor have been attempted with the hashing algorithm. But, the performance of the hashing algorithm was not improved significantly. In addition, when the power of the hashing function [4.1] was set to 3, the performance of the hashing algorithm was tested with the load factors 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, and 0.8 respectively. All of the values of ALOSS fall within 1.0479 and 4.7619.

It could be seen that the performance of the hashing algorithm developed depended upon two factors, load factors and hashing functions. Once the load factor was set, with an increase of the power of hashing function [4.1], the performance of the hashing algorithm was improved, that is, the values of ALOSS were decreased with an increase of insensitivity of ALOSS to the number of trees in the

database. That is because the more weight given to character position of both the plot label and the tree number, the greater spread of the hash values. The best performance of the hashing algorithm resulted from the combination of the load factor 0.1 with the power 3 of hash function [4.1]. It was also apparent that once the hashing function was chosen the performance of the hashing algorithm benefited from the decrease of the load factor.

Although the total of 1629 files including 543 stem analysis trees were processed in demonstration, actually, this hashing algorithm can be used to process all the TRIM data files. It should be pointed out that it is up to the user to balance the performance of the hashing algorithm and the use of the computer memory space. In other words, better performance of this hashing algorithm requires more space. Based on the findings, the better performance of this hashing algorithm can be obtained by changing the load factor. If a user would like to have the best performance of the hashing algorithm the load factor should be set to 0.1 with which a value of 1.2 ALOSS can be expected with the power of 3 of hashing function [4.1]. If the memory space is at a premium, the load factor can be set to 0.9 with which 1.5 ALOSS can be obtained at the expense of a small hash table. Computer memory space consumed by a hash table depends upon the number of trees to be processed and the load factor chosen, both determining the size of a hash table. For example, if there are 10,000 trees to be processed, the power of hashing function [4.1] is set to 3. For performance of value of 1.2 ALOSS, the memory space consumed by the hash table will be a result of a number of trees divided by a

load factor: (10,000/0.1) * 2 = 2,00,000 bytes (because a hash table is declared to be integer which consists of 2 bytes), in comparison to (10,000/0.9) * 2 = 22,222 bytes for a performance of value of 1.5 ALOSS. The later performance of the hashing algorithm can be obtained at expense of a significant reduction in memory space.

The modification of load factor can be done by redefinition of the hash table size in the source file called DEFINE.H (see Table 13.). Specifically, first, open the file DEFINE.H, next, define the hash table size by dividing the number of stem analysis trees by load factor (either 0.1 or 0.9).

Table13. Source file named DEFINE.H.

Syntax name	Variables declared	Values defined
#define	SUCCESS	1
#define	FAILURE	0
#define	END	0
#define	NOT_END_FILE	1
#define	START	1
#define	YES	1
#define	SAME	0
#define	FOUND	1
#define	NOT_FOUND	0
#define	NOT_OPEN	0
#define	OK	0
#define	HASH_FUN_POWER	3
#define	MAX_PLT_LAB	22
#define	LAB_LEN	20
#define	FN_LEN	40
#define	MAX_YEAR_NUM	100
#define	HASH_TBL_SIZE	dividing a total number of trees y either 0.1 or 0.9

6.2 THE TWO-STAGE SAMPLING RULES

The major objective of this investigation was to determine both the minimum primary sample intensities and the minimum subsample intensities required for estimating yearly volume growth of immature jack pine. The worse cases for the unequal probability subsampling rule and the best cases for the equal probability subsampling rule under the different primary sample intensities and the various subsample intensities have already been presented.

For the equal probability subsampling rule, given all the combinations of the primary sample intensities with the subsample intensities there was no significant difference between the estimate mean and the TRIM mean, but, the precisions of mean estimate were all below 88 percent and the precisions were below 60 percent on average. Therefore, this sampling rule is not recommended for use to estimate the yearly volume growth of immature jack pine in northeastern Ontario.

For the unequal probability subsampling rule, when the primary sample intensities were 100 percent, the effect of lowering the subsample intensities on the precision is shown in Table 14. The average of precision of mean estimate was 98.2 percent with the subsample intensities of 10 percent; the average of precision of mean estimate was 97.6 percent with the subsample intensities of 5.8 percent; and the average of precision of mean estimate was 94.6 percent with the subsample intensities of 3.1 percent. When the subsample intensities were reduced 5.8 percent, nearly half of the

first subsample intensities, the precision was only dropped by 0.6 percent on average. When the subsample intensities were further decreased to 3.1 percent the precision was not reduced much, only 3 percent. These results suggest that when the primary sample intensities were set to 100 percent, lowering the subsample number per plot would not reduce the precision significantly. In this case, it means that the rules with the first two higher subsample intensities were indeed unnecessary since the 94.6 precision is close to the true value.

Table 14. The comparison of effects in change of subsample intensities on the precision of mean estimate when the primary sample intensities were 100 percent

Subsample intensities (%)	Average precision (%)
10.0	98.2
5.8	97.6
3.1	94.6

As shown in Table 15, when the primary sample intensities were reduced to 66 percent from 100 percent and the subsample intensities were reduced to 6.7 and 3.8 percent, the precisions of 94.4 and 93.8 on average were obtained respectively. When the subsample intensities were further lowered to 2.1 percent the 91.2 percent of precision on average still can be secured where the precisions ranged from 88.2 percent to 98.8 percent. When comparing the figures in Table 14 with those in Table 15. it can be noted that when the

primary sample intensities were lowered to 66 percent, with the subsample intensities of 6.7 percent, nearly the same precision could be obtained as with the primary sample intensities of 100 percent together with the subsample 3.1 percent.

Table 15. The comparison of effects in change of subsample intensities on the precision of mean estimate when the primary sample intensities were 66 percent

Subsample intensities (%)	Average precision (%)
6.7	94.4
3.8	93.8
2.1	91.2

In this study, the three types of stands may be classified in terms of their trees spatial distributions using the Pielou's index. According to the definition of the Pielou's index the trees in stand K showed uniform spatial distributions, the trees in the stand KLD_P showed random spatial distributions , and those in stand NL showed significant aggregations or clustering. Through the analysis of simulation results, the trees spatial distributions did not appear to influence the precision significantly. This finding is consistent with the conclusion made by Murchison and Kavanagh (1989), that is, "tree spatial distribution as defined by Pielou's Nonrandomness Index appeared to have little influence on sample rule performance."

In summary, throughout the sampling simulations the simulation results, overall, were consistent. In all cases with the same primary sample intensities, with the same subsample intensities, and with the same confidence limit, the precision of the mean estimates for the unequal probability subsampling rule was much higher than that for the equal probability subsampling rule. In most of the cases, the precision of the mean estimates produced by the equal probability subsampling rule was too low to be acceptable even when the subsample sizes were increased to the maximum number allowed by the sampling simulation. In all cases, with all the limited possible combinations of the primary sample intensities together with the subsample intensities the precisions produced by the unequal probability subsampling rule were reliable.

The database in this study were limited to the 10 m by 10 m destructive sample plots for TRIM projects conducted in immature jack pine in northeastern Ontario by the OMNR. The studies were limited to 3 plots per stand. Although the simulation results appeared to be limited by these low numbers of plots, consistent trends appeared and should serve as guidelines. Since the stands and plots were presumably randomly selected, the results should be applicable to the populations from which they were drawn.

7. CONCLUSION

For the hashing algorithm developed in this study, three goals have been achieved:

(1) by using a hashing technique with the data structure of linked list, the TRIM data was processed and all the output were placed into one file which uses a small hash table file, greatly simplifying the directory structures ;

(2) the hashing algorithm can be used to process TRIM data to obtain the various growth attributes (volume cumulative increments, height cumulative increments, and dbh cumulative increment) by one-year intervals for all individual trees;

(3) the hashing algorithm was developed to provide a user with quick access to any required stem analyzed tree record in the output file.

For the computer sampling simulation, based on the findings of this study dealing with the two two-stage subsampling of fixed area plots in immature jack pine stands in northeastern Ontario, it can be concluded:

(1) the subsampling rule using probability proportional to the basal area selection of trees proved to be superior in precision for estimating tree annual volume growth of immature jack pine in northeastern Ontario;

(2) for each stand with the subsampling rule using probability proportional to basal area, with the minimum of primary sample intensity ratio at 66 percent together with the minimum of subsample intensity ratio at 2.1 percent, a precision of 90 percent for the mean estimate of the annual volume growth can be guaranteed with a confidence limit of 95 percent;

(3) given the same confidence limit, for the subsampling rule with the simple random selection of trees, even with larger primary sample intensities of plots together with larger subsample intensities of stem analyzed trees, the reliable estimate could not obtained in this study.

LITERATURE CITED

Ackoff, R. L. 1962. Scientific method: optimizing applied research decisions. John Wiley and Sons, Inc., New York. 464 pp.

Aho, A. V., Hopcroft, J. E., and Ullman, J. D. 1983. Data structure and algorithm. Addision-Wesley Publishing Company. Bell Laboratories, Murray Hill, New Jersey. 48 pp.

Aldred, A. H., and J. K. Hall 1975. Application of large scale photography to a forest inventory. For. Chron. 51(1): 9-15

Arvanitis, L. G. 1966. Decision rules for design of forest sampling system: a contribution to methodology based on computer simulation. Doctoral Dissertation. U. of California, Berkeley. 168 pp.

Barnett, V. 1974. Elements of sampling theory. The English Universities Press Ltd. St Paul's House, Warwick Lane, London. 126 pp.

Beijing Forestry University, 1977. Statistics. China's Forestry Publishing House. 196 pp.

Bonner, G. M. 1974. A forest sampling design for inventories using large-scale aerial photography. State University of New York College of Environmental Science and Forestry. Doctoral dissertation. University Microfilms, a XEROX Company, Ann Arbor, Michigan. 27 pp.

Brace, L. G., and K. M. Mager 1968. Automated computation and plotting of stem analysis data. Can. Dep. For. Rur. Dev., For. Br. Pub. No.1209, 8 pp.

Brillinger, P. C., and Cohen, D. J. 1972. Introduction to data structures and non-numeric computation. Prentice-Hall, Inc., Englewood Cliffs, New Jersey. 126 pp.

Buslenko, N. P., Golenko, D. I., Shreider, Y. A., Sobol, I. M., and Sragovich, V. G. 1966. The Monte Carlo Method. Pergmon Press. New York. 29 pp.

Chapeskie, D., and R. Fleet 1981. Stem analysis program. OMNR. Brockville. (unpublished-cited in Kavanagh 1983)

Cochran, W. G. 1963. Sampling techniques. John Wiley and Sons. New York. 413 pp.

Cunia, T. 1965. Some theory on reliability of volume estimates in a forest inventory sample. For. Sci. 11 (1) : 115-128.

Deming, W. E. 1950. Some theory of sampling. Dover Publications Inc., New York. 517 pp.

de Vries, P. G. 1986. Sampling theory for forest inventory. Springer-Verlag. Berlin Heidelberg, New York. 161 pp.

Farmer, R. A., M. S. Philip and A. R. Sayers. 1973. Some experience with two-stage sampling in a forest survey. Forestry 46(1) : 95-104.

Fayle, D. C. F., D. MacIver and C. V. Bentley. 1983. Computer-graphing of annual ring widths during measurement. For. Chron. 59 : 291-293 pp.

Flores, I. 1977. Data structure and management. 2nd edition. Prentice-Hall, Inc., Englewood Cliffs, New Jersey. 156 pp.

Frayer, W. E. 1979. Multi-level sampling designs for resource inventories. Report RM contract 16-747-CA, CSU Project 31-1470-1468. Rocky Mountain For. and Range Exp. Stn. USDA For. Ser. Ft. Collins, CO. 113 pp.

Hammersley J. M., and Handscomb D. C. 1965. Monte Carlo Methods. John Wiley and Sons Inc. 178 pp.

Harrison, R. V. 1972. Data-structures and programming. Courant Institute of Mathematical Sciences. New York University. 195 pp.

Herman, F. R., D. J. DeMars and R. F. Woollard. 1975. Field and computer techniques for stem analysis of coniferous forest trees. USDA For. Serv. Res. Pap. PNW-194, Pacific Northwest For. and Range Expt. Station, Portland, Oregon. 51 pp.

Horowitz, E., and Sahni, S. 1983. Fundamentals of data structures. Computer Science Press, Inc. Rockville, Maryland, USA. 106 pp.

Husch, B., C. I. Miller and T. W. Beers. 1972. Forest mensuration. 2nd. ed. John Wiley and Sons, New York. 410 pp.

Hutchison, R. C. and Just, S. B. 1988. Programming using the C language. McGraw-Hill Book Company. Formal Systems, Inc. 299 pp.

Johnston, D. C. 1982. Theory and application of selected multi-level sampling designs. Ph. D. Dissertation. Univ. Microfilms International, Ann Arbor, Michigan. 197 pp.

Kavanagh, J. 1983. Stem analysis: sampling techniques and data processing. M. Sc. F. Thesis, Lakehead University, Thunder Bay, Ont. 21 pp.

Kleijnen, J. P. 1974. Statistical techniques in simulation. Marcel DeKKer, Inc., New York. 5 pp.

Langley, P. G. 1975. Multistage variable probability sampling: Theory and use in estimating timber resources from space and aircraft photography. Doctoral dissertation. U. of California, Berkeley, Ca. 92 pp.

Lund, H. G. 1982. In-place resource inventories: principles & practices: Proceedings of a national workshop.
Society of American Foresters.Washington, D. C. 4 pp.

MacIver, D. C. 1987. TRIM compunications. Proc. TRIM users workshop, OMNR, Timmins, Ont. July 9, 1987. 91 pp.

Murchison, H. G. 1984. Efficiency of multi-phase and multi-stage sampling for tree heights in forest inventory. Doctoral dissertation, U. of Minnesota, St. Paul, MN. 158 pp.

Murchison, H. G., and J. Kavanagh. 1988. Evaluation of TRIM database for jack pine. Lakehead Univ. Report submitted to OMNR. 17 pp.

_____ 1989. Sample intensities and cost model for TRIM projects for mature jake pine in northeastern Ontario. 1 pp.

_____ 1990. Sample intensities for TRIM sampling of mature black spruce in northeastern Ontario. Report Submitted To OMNR-NFDG, Timmins, Ontario. 62 pp.

Naylor, T. H., Balintfy, J. L., Burdick, D. S., and Chu. K. 1966. Computer simulation techniques. John Wiley & Sons. Inc., New York. 8 pp.

Pluth, D. J. and D. R. Cameron. 1971. Announcing Fortran IV program for computing and growth parameters from stem analysis. Forest Science 17: 1-102 pp.

Schumacher, F. X., and R. A. Chapman. 1954. Sampling methods in forestry and range management. School of Forestry, Duke Univ. Durham, North Carolina. 95 pp.

Sukhatme, P. V. and B. V. Sukhatme. 1970. Sampling theory and applications. 2nd. ed. Iowa State University Press. Ames, Iowa. 267 pp.

Standish, T. A. 1980. Data structure techniques. Addision-Wesley Publishing Company. Univ. of California, Irvine. 192 pp.

Stone, H. S. 1972. Introduction to computer organization and data structures. Stanford Univ. McGraw-Hill Book Company. 266 pp.

Stuart, A. 1968. Basic ideas of scientific sampling. Charles Griffin & Co. Ltd. London. 77-83 pp.

Timmer, V. R., and B. R. Verch. 1983. SAPP: A computer program for plotting stem analysis. For. Chron. 59 (1): 298 pp.

VanWyk, C. J. 1988. Data structures and C programs. AT & T Bell Laboratories. Murray Hill, New Jersey. 177 pp.

Wang, E. 1976. Stem analysis program. Lakehead University Thunder Bay. (unpublished - cited in Kavangh 1983)

Williams, B. 1978. A sampler on sampling. John Wiley & Sons, New York. 148 pp.

Yamane, T. 1967. Elementary sampling theory. Prentice-Hall, Inc., Englewood Cliffs, New. Jersey. 292 pp.

Yandle, D. O. and F. M. White. 1977. An application of two-stage forestry sampling. Southern Journal of Applied Forestry. vol. 1:3 27 - 32 pp.

Yates, F. R. S. 1960. Sampling methods for censuses and surveys. 3rd Ed. Charles Griffin & Co. Ltd. London. 34 pp.

Zeigler, B. P. 1976. Theory of modelling and simulation. John Wiley & Sons. New York. 3 pp.

APPENDICES

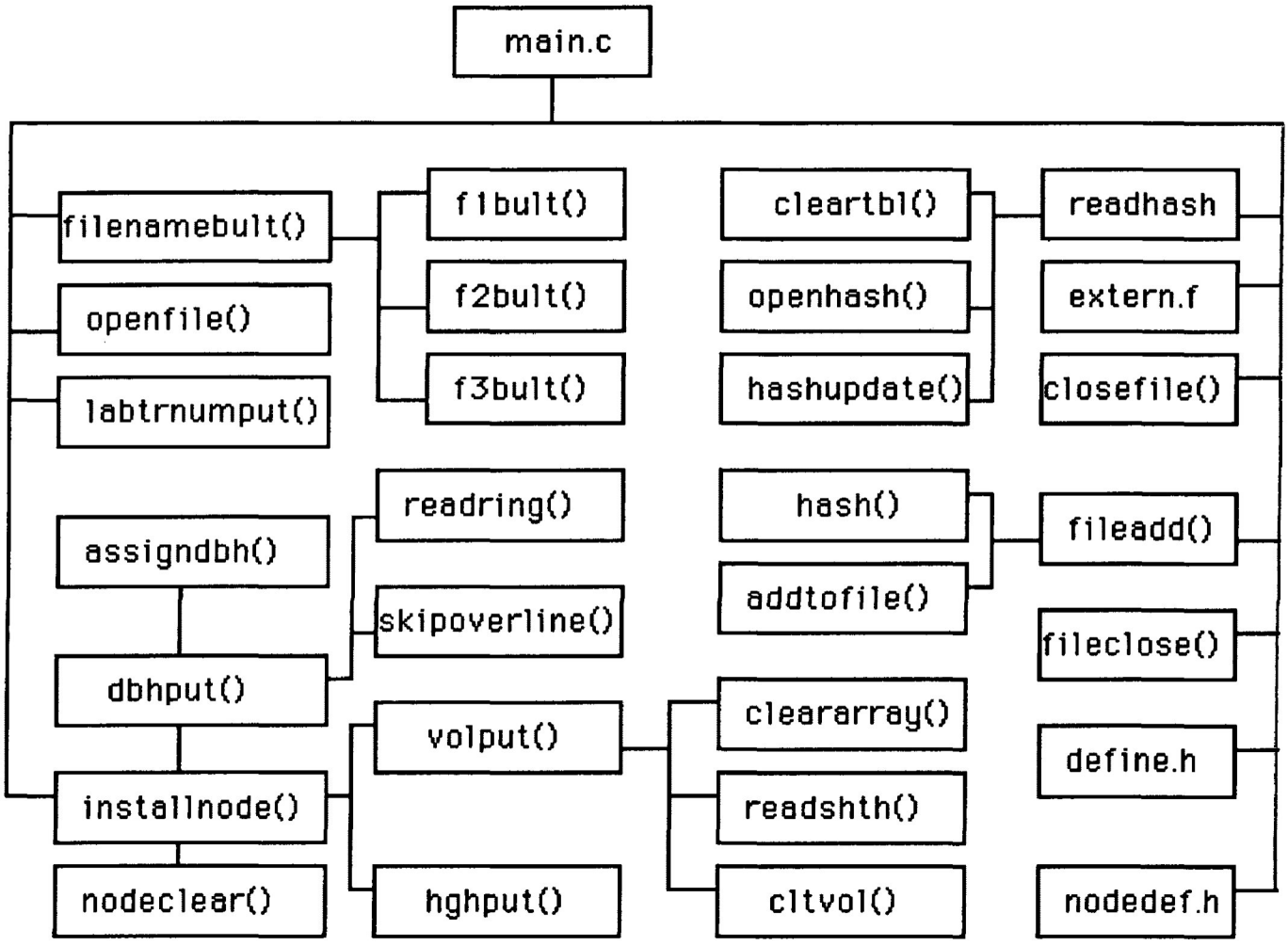
APPENDIX I

This appendix includes the TRIMHASH.C program which was written in C language. This program processed the existing TRIM data files using a hashing function with the data structure of the table pointer, database file, and the linked list. After TRIM data files were proccessed by the program, two files, NEWTRIM.DAT and HASH.TBL, were created. The NEWTRIM.DAT file contained the collection of all processed stem analysis trees' records, or nodes and an associated file HASH.TBL stored the addresses of all the nodes.

When executing the program TRIMHASH.C, the user will be prompted to enter the following information: (1) plot label, and (2) total number of trees in plot. The example for the excution of this program is given in this appendix.

Author: Tiemin Sheng
School of Forestry
Lakehead University
Dec. 1992

Figure I-1. The flow chart of the TRIMHASH.C program:



Here is an example of how to execute the **TRIMHASH.C** program:

```
$cc TRIMHASH.C   -lm < return >
$a.out                    < return >
$ENTER NEXT PLOT LABEL: K1   < return >
$ENTER TOTAL NUMBER OF TREES IN PLOT LABEL<K>: 31 < return >
$ENTER NEXT PLOT LABEL: K2   < return >
$ENTER TOTAL NUMBER OF TREES IN PLOT LABEL<K2>: 22 < return>
$WAIT.....
$SUCCESS
```

Main program: **TRIMHASH.C**

```
#include<stdio.h>
#include "define.h"
#include "trdefine.h"
#include "processdata.h"
#include "fileclose.h"

FILE * fpt_1;
FILE * fpt_2;
FILE * fpt_3;
FILE * hashptr;
FILE * fptr;

char fn_1[ FN_LEN ];
char fn_2[ FN_LEN ];
char fn_3[ FN_LEN ];
long hash_tbl[ HASH_TAB_SIZE ];
char lab[ MAX_PLT_LAB ][ LAB_LEN ];
int  num_tr[ MAX_PLT_LAB ];
struct tree_info temp;

main()
{
   int  i, trnum;

   label_tr_num_input();

   for( i = 0;  i < MAX_PLT_LAB;  i++ )
   {
       for( trnum = 1; trnum <= *(numtr+i); trnum++ )
       {
            process_data( &i,  &trnum );
        }
   }
   file_close();
}
```

Source File: **clear.nod**

```
#include "define.h"
#include "extern.h"

node_clear()
{
    int i, j, k;

    temp.plt_lab[ 0 ] = '\0';
    temp.sp_code[ 0 ] = '\0';
    temp.tn = 0;
    temp.age = 0;
    temp.dbh_age = 0;
    temp.last_ck_age = 0;
    temp.dbhob = 0.0;
    temp.vio = 0.0;

    for( i = 0;  i < 2;  i++ )
    {
        *( temp.s_cor+i ) = 0.0;
        *( temp.c_cor+i ) = 0.0;
    }
    for( j = 0; j < 100; j++ )
    {
        *( temp.dbhs+j ) = 0.0;
    }
    for( k = 0;   k < MAX_YEAR_NUM;   k++ )
    {
        *( temp.vols+k ) = 0.0;
    }
    temp.next = 0;
}
```

Source File: **clear.tbl**

```
#include "define.h"
#include "extern.h"

clear_tbl()
{
     int  i;
     for( i = 0;  i < HASH_TAB_SIZE;  i++ )
     {
          *(hash_tbl + i) = 0;
     }
}
```

Source File: **closefile.h**

```
#include "extern.h"

close_file()
{
    fclose( fpt_1 );
    fclose( fpt_2 );
    fclose( fpt_3 );
}
```

Source File: **dbhput.h**

```
#include<stdio.h>
#include <assert.h>
#include "define.h"
#include "extern.h"

dbhs_put( old )
int  * old;
{
    int      i, yearcut, rc;
   char     sc[ 3 ], line[ 1000 ];
   double  dht, sbt, pith, rad, array[ 100 ];

   while( fgets( line,  1000,  fpt_1 ) != NULL )
   {
      assert( sscanf(line,"%s%d%lf%d%lf%lf%lf", sc,  &yearcut,&dht,
                                        &rc,&sbt,&pith,&rad ) == 7 );
      if( dht == 130.0000 )
      {
         read_ring_wid( &rc,  array );
         assign_dbhs_to_temp( &rc, &sbt, &pith, &rad, array );
      }
      else
      {
        for( i = 0;  i < skip_over_line( &rc );  i++ )
        {
           fgets( line,  1000,  fpt_1 );   /*cast unwanted lines*/
        }
      }
  }
  *old = rc;
  temp.age = rc;
  strcpy( temp.sp_code, sc );

  return( SUCCESS )
}
```

```
skip_over_line( rc )
int  * rc;
{
      int num_line, remainder;
      num_line = 0;
      remainder = *rc;

      if( remainder <= 10 )
      {
        num_line = 1;
       }
      else
      {
        while( remainder > 10 )
        {
            remainder = remainder - 10;
            num_line++;
         }
        if( remainder > 0 )
        {
            num_line = num_line  + 1;
        }
      }
      return( num_line );
}


read_ring_wid( rc, ring_array )
int    * rc;
double  * ring_array;
{
     int  i;
     for( i = 0;   i < *rc;  i++ )
     {
        if( !(i%10) )            /* skip over year, then read data */
        {
          fscanf( fpt_1,  "%*s%lf",  ring_array+i );
        }
        else
        {
          fscanf( fpt_1,  "%lf",  ring_array+i );
```

```
        }
    }
    fscanf( fpt_1,  "\n" );    /* skip over '\n', end of line char */
}


assign_dbhs_to_temp( rc, sbt, pith, rad, ring_array )
int    * rc;
double * sbt;
double * pith;
double * rad;
double * ring_array;
{
      int    i, j;
      double dib, dob, yr_r_wdth;
      yr_r_wdth = 0;
      dib = ( ( *rad + *pith ) * 2.0 );
      dob = ( ( *sbt + *rad +  *pith ) * 2.0 );

      temp.dbh_age = *rc;
      temp.dbhob = dob;
      *(temp.dbhs) = dib;

      for( i = 0;  i < *rc;  i++ )
      {
          yr_r_wdth += (*( ring_array + i ) )* 2.0;
          *( temp.dbhs + i + 1 ) = dib - yr_r_wdth;
      }
}
```

Sourec File: **define.h**

```
#define     SUCCESS                 1
#define     FAILURE                 0
#define     END                     0
#define     NOT_END_FILE            1
#define     START                   1
#define     YES                     1
#define     SAME                    0
#define     FOUND                   1
#define     NOT_FOUND               0
#define     NOT_OPEN                0
#define     OK                      0
#define     MAX_PLT_LAB            22
#define     LAB_LEN                20
#define     FN_LEN                 40
#define     MAX_YEAR_NUM          100
#define     HASH_TAB_SIZE        4097
```

Source File: **extern.h**

```
extern char lab[ MAX_PLT_LAB ][ LAB_LEN ];
extern int  num_tr[ MAX_PLT_LAB ];
extern char fn_1[ FN_LEN ];
extern char fn_2[ FN_LEN ];
extern char fn_3[ FN_LEN ];
extern FILE  * fpt_1;
extern FILE  * fpt_2;
extern FILE  * fpt_3;
extern FILE  * hashptr;
extern FILE  * fptr;
extern long hash_tbl[ HASH_TAB_SIZE ];
extern struct tree_info temp;
```

Source File: **fileclose.f**

```
#include "extern.h"

file_close();
{
    fclose( hashptr);
    fclose( fptr );

    printf( "SUCCESS\n" );
}
```

Source File: **fnamebult.h**

```
#include <assert.h>
#include "extern.h"
#include "define.h"

file_names_bult( i, tr_count )
int * i;
int * tr_count;
{
    assert( f_1_bult( i, tr_count ) == SUCCESS );
    assert( f_2_bult( i, tr_count ) == SUCCESS );
    assert( f_3_bult( i, tr_count ) == SUCCESS );
}


f_1_bult( i, tr_count )
int *i;
int *tr_count;
{
    static int j = 1;
    int k;
    char buff[10];
    char array[ FN_LEN ];

    array[0] = '\0';
```

```
    k = *tr_count;

    sprintf( buff,  "%d",  k );    /* convert int to char */

    strcat( array, lab[ *i ] );
    strcat( array, "_" );
    strcat( array, buff );
    strcat( array, ".out" );
    fn_1[0] = '\0';
    strcpy( fn_1,  array );

    return( SUCCESS );
}


f_2_bult( i,  tr_count )
int  *i;
int  *tr_count;
{
    int  k;
    char buff[ 10 ];
    char array [ FN_LEN ];

    array[0] = '\0';
    k = *tr_count;

    sprintf( buff,  "%d",  k );    /* convert int to char */

    strcpy( array,  "ad_" );
    strcat( array,  lab[*i] );
    strcat( array,  "_" );
    strcat( array,  buff );
    strcat( array,  "." );
    fn_2[0] = '\0';
    strcpy( fn_2,  array );

    return( SUCCESS );
}
```

```
f_3_built( i,  tr_count )
int  *i;
int  *tr_count;
{
    int   k;
    char  buff[ 20 ];
    char  array[ FN_LEN ];

    array[0] = '\0';
    k = *tr_count;

    sprintf( buff,  "%d",  k )     /* convert int to char */

    strcpy( array,  "annv_" );
    strcat( array,  lab[*i] );
    strcat( array,  "_" );
    strcat( array,  buff );
    strcat( array,  "." );
    fn_3[0] = '\0';
    strcpy( fn_3,  array );

    return( SUCCESS );
}
```

Source File: **hash.fun**

```
#include <math.h>
#include "define.h"
#include "extern.h"

hash( i,  tr_count )
int *i;
int *tr_count;
{
     char   cbuff[ 20 ], *cptr;
     double constnt,  position;
```

```
    long    hash_value;

    hash_value = 0;
    position = 0;
    constnt = 3;

    sprintf( cbuff,   "%d",   *tr_count );
    cptr = cbuff;

    while( *( lab[ *i ] + position ) != '\0' )
    {
        hash_value += pow(position+1, constnt)*(*(lab[*i]+position ));
        position++;
    }
    while( *cptr++ != '\0' )
    {
        hash_value += pow(position+1, constnt) * (*(cptr-1));
        position++;
    }
    return( (int) ( hash_value%HASH_TAB_SIZE ) );
}
```

Source File: **hghput.h**

```
#include "extern.h"
#include "define.h"

hghs_put( old )
int  * old;
{
    int   i, j, k, num, start, index;
    double array[ 5000 ];
    k = 0;
    index = 0;

    while( fscanf( fpt_2,  "%lf",  array+index ) == NOT_END_FILE )
    {
```

```
            index++;
        }
        num = (*old) * 2;
        start = index - num;

        for( j = 0;  j < *old;  j++ )
        {
            *( temp.hghs + j ) = *( array + start + k )/100;
            k += 2;
        }
        return( SUCCESS );
}
```

Source File: **install.nod**

```
#include <assert.h>
#include "define.h"
#include "extern.h"
#include "dbhput.h"
#include "hghput.h"
#include "volclt.h"
#include "clear.nod"

install_node( i, tr_indx )
int * i;
int * tr_indx;
{
        int old;
        node_clear();

        strcpy( temp.plt_lab, lab [ *i ] );
        temp.tn = *tr_indx;

        assert( dbhs_put( &old ) == SUCCESS );
        assert( hghs_put( &old ) == SUCCESS );
        assert( vols_clt() == SUCCESS );
}
```

Source File: **numput.h**

```
#include<stdio.h>
#include<string.h>
#include"define.h"
#include"extern.h"

label_tr_num_input()
{
     int  n;
     char *cpointer;

     for( n = 0;  n < MAX_PLT_LAB;  n++ )
     {
          printf( "\nEnter Next Plot Label: " );
          fgets( lab[ n ], LAB_LEN, stdin );
          cpointer = strchr( lab[ n ],  '\n' );
          *cpointer = '\0';

          printf( "\nEnter Total Number Of Trees:" );
          scanf( "%d",  num_tr + n );
          getchar();

          printf( "\n\nWhat you just input are as follows:" );
          printf( "\n\n%-12s%-12s",  "Plot Label:", lab[ n ] );
          printf( "\n%-24s%-8d\n", "Total Number Of Trees:",
                                              *(num_tr+n) );
     }
     printf( "\n\nWAIT.....\n" );
}
```

Source File: **openfile.h**

```
#include <assert.h>
#include "extern.h"
#include "define.h"
```

```
open_files()
{
    if( fptr == NOT_OPEN )
    {
        assert( ( fptr  = fopen( "trim.dat",   "a+" ) ) != FAILURE );
    }
    assert( ( fpt_1 = fopen( fn_1,   "r"  ) )  != FAILURE  );
    assert( ( fpt_2 = fopen( fn_2,   "r"  ) )  != FAILURE  );
    assert( ( fpt_3 = fopen( fn_3,   "r"  ) )  != FAILURE  );
}
```

Source File: **readhash.h**

```
#include <assert.h>
#include "define.h"
#include "extern.h"
#include "clear.tbl"

readhash()
{
    clear_tbl();
    openhash();
    if( fread( hash_tbl, sizeof(long), HASH_TAB_SIZE, hashptr )
                                                < HASH_TAB_SIZE )
    {
        hash_update();
    }
}


openhash()
{
   if( hashptr == 0 )
   {
      assert( ( hashptr = fopen( "hash.tbl",   "r+" ) ) !=  FAILURE );
   }
   else
   {
      fseek( hashptr, (long)0,   0 );
```

```
    }
}
hash_update()
{
    rewind( hashptr );
    assert((fwrite(hash_tbl, sizeof(long), HASH_TAB_SIZE, hashptr))
                                                        != FAILURE );
}
```

Source File: **trdefine.h**

```
#include "define.h"

struct tree_info
{
        char        plt_lab[ LAB_LEN ];
        int         tn;
        char        sp_code[ 3 ];
        int         age;
        int         dbh_age;
        int         last_ck_age;
        double      dbhob;
        double      dbhs[ 100 ];
        double      hghs[ 100 ];
        double      vio;
        double      vols[ MAX_YEAR_NUM ];
        double      s_cor[ 2 ];
        double      c_cor[ 2 ];

        int         next;

};
```

Source File: **volclt.h**

```
#include <stdio.h>
#include <assert.h>
#include "extern.h"
#include "define.h"

vols_clt()
{
    int    max_shth,  tot_disc;
    double  array[ 100 ][ 150 ];

    clear_array( array );
    assert( read_shth_vol( &tot_disc, &max_shth, array )==SUCCESS);
    assert( clt_vol( &tot_disc,  &max_shth,  array ) == SUCCESS );

    return( SUCCESS );
}


clear_array( array )
double (* array)[150];
{
     int  line,  col;
     for( line = 0;  line < 100;  line++ )
     {
         for( col = 0;  col < 150;  col++ )
         {
             *( array[ line ] + col ) = 0.0;
         }
     }
}


read_shth_vol( tot_disc,  max_shth,  array )
int   * tot_disc;
int   * max_shth;
double (* array)[150];
{
```

```
    int r_count, disc_num, col;
    disc_num = 0;
    while( fscanf( fpt_3,  "%d",  &r_count ) == NOT_END_FILE )
    {
        disc_num ++;
        for( col = 0;  col < r_count;  col++ )
        {
            fscanf( fpt_3,  "%lf", (array[disc_num-1]+col) );
        }
    }
    *max_shth = r_count;
    *tot_disc = disc_num;
    temp.last_ck_age = r_count - 2;

    return( SUCCESS );
}


clt_vol( tot_disc, max_shth, array )
int   * tot_disc;
int   * max_shth;
double (* array)[150];
{
    int    i, line_num, col_num;
    double vob,  shth_vol;
    shth_vol = 0;
    for( col_num = *max_shth;  col_num > 0;  col_num-- )
    {
       for( line_num = 0;  line_num < *tot_disc;  line_num++ )
       {
          shth_vol +=  *( array[ line_num ] + col_num - 1 );
       }
       *( temp.vols + col_num - 1 ) = shth_vol/1000000;
                                          /* unit: m^3 */
    }
    return( SUCCESS );
}
```

Source File: **writetofile.h**

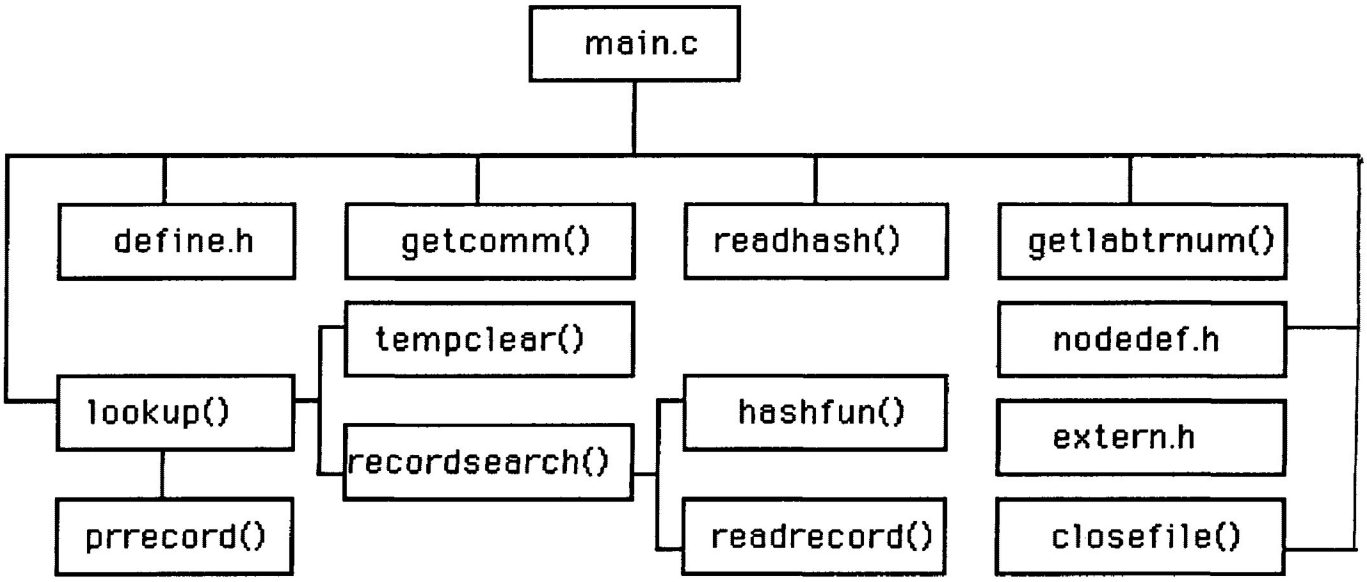
```
#include "define.h"
#include "extern.h"
#include "hash.fun"

file_add( i,  tr_num )
int  * i;
int  * tr_num;
{
     long  hash_value, record;

     hash_value = hash( i, tr_num );
     temp.next = hash_tbl[ hash_value ];
     record = addtofile();
     hash_tbl[ hash_value ] = record + 1;
}


addtofile()                    /* always adds record to end of file */
{
    long  here;

    fseek( fptr,  (long)0,  2 );
    here = ftell( fptr );
    fwrite( (char*) &temp,  sizeof( struct tree_info ),  1,  fptr );

    return( byte_to_record( here ) );
}
```

APPENDIX II

This appendix includes the **PRINT.C** program which was written in C language. This program was developed for a user to print the stored information of any processed stem analyzed trees from **TRIMHASH.DAT** files.

Author: Tiemin Sheng
School of Forestry
Lakehead University
Dec. 1992

Figure II-1. The flow chart of the **PRINT.C** program:



Here is an example of how to execute the **PRINT.C** program :

*$cc **PRINT.C** -lm < return >*
$a.out < return >
$ENTER PLOT LABEL: *K1 < return >*
$ENTER TREE NUMBER: *9 < return >*

(Note: Results printed on computer screen were shown in Table 7.)

$WOULD YOU LIKE PRINT MORE ?
$ENTER < 1 > FOR YES, OR
$ENTER < 0 > FOR NO.
$YOUR CHOICE ?
$0 < return >

Table II-1. The example of results printed by the **PRINT.C** program:

Plot label: k1 Tree number: 9

Species code: Pj
Age: 21
DBH(m): 1.341100
Height(m): 11.85000
Volume(m^3): 0.098708

Year	1	2	3	4	5
DBH	0.000000	0.000000	0.026000	0.049400	0.100800
Height	0.070000	0.520000	0.970000	1.400000	2.120000
Volume	0.000050	0.000155	0.000362	0.000777	0.001406

Year	6	7	8	9	10
DBH	0.167800	0.267800	0.392100	0.507100	0.625900
Height	2.370000	2.750000	3.250000	4.750000	5.100000
Volume	0.002565	0.004366	0.006640	0.009833	0.013831

Year	11	12	13	14	15
DBH	0.733100	0.824600	0.894600	0.973000	1.032900
Height	5.750000	6.620000	6.870000	7.750000	8.250000
Volume	0.018572	0.024036	0.031398	0.038513	0.045367

Year	16	17	18	19	20
DBH	1.088300	1.139200	1.186100	1.241400	1.289700
Height	8.750000	9.670000	9.920000	10.520000	11.120000
Volume	0.052790	0.060550	0.070397	0.081352	0.090860

Year	21
DBH	1.341100
Height	11.850000
Volume	0.098708

The main program **PRINT.C:**

```
#include <stdio.h>
#include <assert.h>
#include "define.h"
#include "trdefine.h"
#include "preadhash.h"
#include "popenfile.h"
#include "pgetnum.h"
#include "pgetcom.h"
#include "plook.up"
#include "pclosefile.h"

FILE  *fptr;
FILE  *hashptr;
FILE  *wfpt;

char  tr_num[ 20 ];
char  lab[ 20 ];
long  hash_tbl[ HASH_TAB_SIZE ];
struct tree_info temp;

main()
{
    int  look_next;

    look_next = START;
    while( look_next == YES )
    {
        get_lab_tr_num();
        open_file();
        readhash();
        lookup();
        get_next_command( &look_next );
    }
    close_file();
}
```

Source File: **processdata.h**

```
#include<stdio.h>
#include "fnamebult.h"
#include "openfile.h"
#include "readhash.h"
#include "install.nod"
#include "write.nod"
#include "closefile.h"

process_data( &i,  &tr_num );
int *i;
int *tr_num;
{
      file_names_bult( &i,  &tr_num );
      open_files();
      readhash();
      install_node( &i, &tr_num );
      file_add( &i,  &tr_num );
      hash_update();
      close_file();
}
```

Source File: **pclear.h**

```
#include "define.h"

extern struct tree_info temp;

temp_clear()
{
    int i, j, k;

    temp.plt_lab[ 0 ] = '\0';
    temp.sp_code[ 0 ] = '\0';

    temp.tn = 0;
    temp.age = 0;
```

```
    temp.dbh_age = 0;
    temp.dbhob = 0.0;
    temp.vio = 0.0;

    for( i = 0;  i < 2;  i++ )
    {
        *( temp.s_cor+i ) = 0.0;
        *( temp.c_cor+i ) = 0.0;
    }
    for( j = 0;  j < 100;  j++ )
    {
        *( temp.dbhs+j ) = 0.0;
    }
    for( k = 0;   k < MAX_YEAR_NUM;   k++ )
    {
        *( temp.vols+k ) = 0.0;
    }
    temp.next = 0;
}
```

Source File: **pclosefile.h**

```
#include "pextern.h"

close_file()
{
    fclose( hashptr );
    fclose( fptr );
    fclose( wfpt );
}
```

Source File: **pextern.h**

```
extern  FILE * fptr;
extern  FILE * hashptr;
extern  FILE * wfpt;
extern  char  lab[ 20 ];
```

```
extern  char  tr_num[ 20 ];
extern  long  hash_tbl[ HASH_TAB_SIZE ];
```

Source File: **pgetcom.h**

```
get_next_command( answer )
int *answer;
{
    printf( "\nLook Next Record ? \n Press< 1 > For Yes\nPress< 0 >
                                                        For No\n");
    printf( "Enter Your Choice: " );
    scanf( "%d", answer );
    getchar();
}
```

Source File: **pgetnum.h**

```
#include "pextern.h"

get_lab_tr_num()
{
   printf( "\nEnter Next Plot Lable:" );
   fgets( lab,  20,  stdin );

   printf( "\nEnter Tree Number:" );
   fgets( tr_num, 20, stdin );
}
```

Source File: **phash.h**

```
#include <math.h>
#include <string.h>
#include "pextern.h"

hash_function()
{
    long   hash_value;
```

```
    char   *cpointer, *cptr;
    double constnt,  position;

    cpointer = strchr( lab, '\n' );
    *cpointer = '\0';

    cpointer = strchr( tr_num, '\n' );
    *cpointer = '\0';

    position = 0;
    hash_value = 0;
    constnt = 3;
    cptr = &tr_num[ 0 ];

    while( *( lab + position ) != '\0' )
    {
        hash_value += pow( position+1, constnt )*( *(lab + position) );
        position++;
    }
    while( *cptr++ != '\0' )
    {
        hash_value += pow(position+1, constnt) *( *(cptr-1) );
        position++;
    }
    return( (int) ( hash_value%HASH_TAB_SIZE ) );
}
```

Source File: **plook.up**

```
#include <string.h>
#include <stdlib.h>
#include "define.h"
#include "phash.fun"
#include "pextern.h"
#include "pprint.h"
#include "pclear.h"

extern struct tree_info temp;
```

```
lookup()
{
    if( record_search() == FOUND )
    {
        print_record();
        temp_clear();
    }
    else
    {
        printf( "\nInvalid Plot Label Or Tree Number.\n" );
        return;
    }
}
```

Source File: **popenfile.h**

```
#include <assert.h>
#include "define.h"
#include "pextern.h"

open_file()
{
    assert( ( fptr = fopen( "trim.dat",  "r+" ) ) != FAILURE );
    assert( ( hashptr = fopen( "hash.tbl", "r+" ) ) != FAILURE );
    assert( ( wfpt  = fopen( "trim.out",   "w" ) ) != FAILURE );
}
```

Source File: **preadhash.h**

```
#include <assert.h>
#include "define.h"
#include "pextern.h"

readhash()
{
   int  n;
```

```
    openhash();

    if( (n = fread(hash_tbl, sizeof(long), HASH_TAB_SIZE, hashptr) )
                                                    < HASH_TAB_SIZE )
    {
          printf( "HASH TABLE CORRUPTED.\n" );
          exit( 1 );
    }
}
```

Source File: **print.h**

```
extern struct tree_info temp;

print_record()
{
    print_header();
    print_content();
}


print_header()
{
  printf( "\n%20s%-12s%-10s", " ", "Plot label:", temp.plt_lab );
  printf( "%-12s%s%-10d\n\n\n",    "Tree number:",   " ", temp.tn );
  printf( "%-15s%s\n",     "Species code:", temp.sp_code );
  printf( "%-15s%d\n",     "Age:",   temp.age );
  printf( "%-15s%f\n",     "DBH(m):",   temp.dbhs[0]/100 );
  printf( "%-15s%f\n",     "Height(m):",   temp.hghs[0] );
  printf( "%-15s%f\n\n\n", "Volume(m^3):", temp.vols[0] );
}


print_content()
{

   int i, j, k, h, p, n, time, old, line, remainder;
```

```
time = 0;
old = temp.age;
remainder = old%5;
line = old/5;

for( i = 0;   i < line;   i++ )
{
    printf( "%-5s",  "Year" );
    for( j = 1;   j <= 5;  j++ )
    {
        printf( "%12d",   j+time );
    }
    printf( "\n\n" );
    printf( "%-8s",  "DBH" );
    for( k = 0;   k < 5;   k++ )
    {
        printf( "%12f",   *(temp.dbhs+old-time-k-1)/100 );
    }
    printf( "\n" );
    printf( "%-8s",  "Height" );
    for( h = 0;   h < 5;   h++ )
    {
        printf( "%12f",   *(temp.hghs+old-time-h-1) );
    }
    printf( "\n" );
    printf( "%-8s",   "Volume" );
    for( p = 0;   p < 5;   p++ )
    {
        printf( "%12f",   *(temp.vols+old-time-p-1) );
    }
    printf( "\n\n" );
    time += 5;
}
if( remainder > 0 )
{
    printf( "%-5s",  "Year" );
    for( j = 1;   j <= remainder;  j++ )
    {
        printf( "%12d",   j+time );
```

```
        }
        printf( "\n\n" );
        printf( "%-8s",  "DBH" );
        for( k = 0;   k < remainder;   k++ )
        {
           printf( "%12f",   *(temp.dbhs+old-time-k-1)/100  );
        }
        printf( "\n" );
        printf( "%-8s",  "Height" );
        for( h = 0;   h < remainder;   h++ )
        {
           printf( "%12f",   *(temp.hghs+old-time-h-1)  );
        }
        printf( "\n" );
        printf( "%-8s",   "Volume" );
        for( p = 0;   p < remainder;   p++ )
        {
           printf( "%12f",   *(temp.vols+old-time-p-1)  );
        }
        printf( "\n\n" );
    }
}
```

APPENDIX III

This appendix includes the SIMULATION.C program which was written in C language. This program was developed to perform two two-stage sampling simulations. The program would use both NEWTRIM.DAT and hash table HASH.TBL. After this program was executed the SIMULATION.DAT file would be created.

Author: Tiemin Sheng
School of Forestry
Lakehead University
Dec. 1992

Figure III-1. The flow chart of the **SIMULATION.C** program:

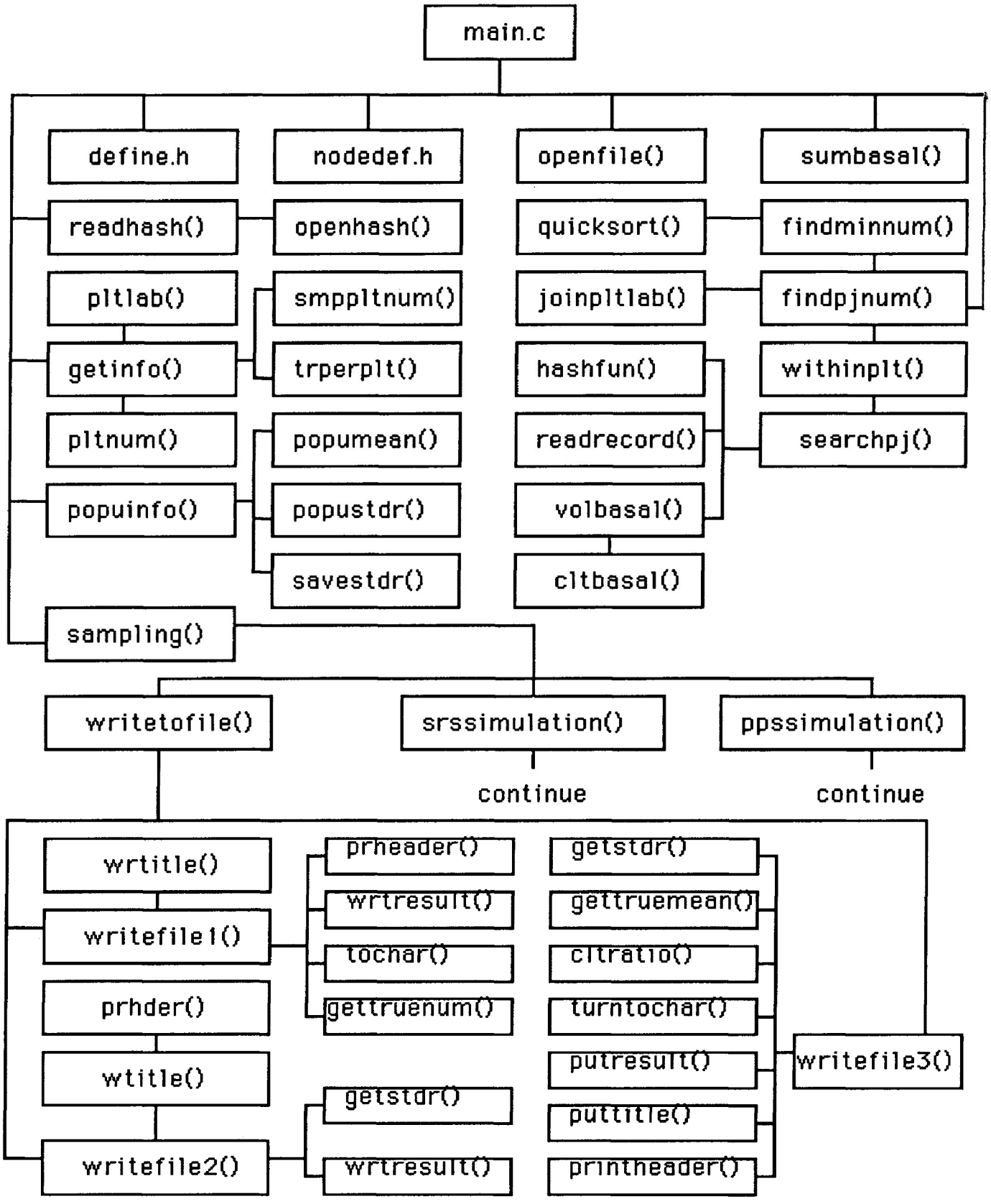
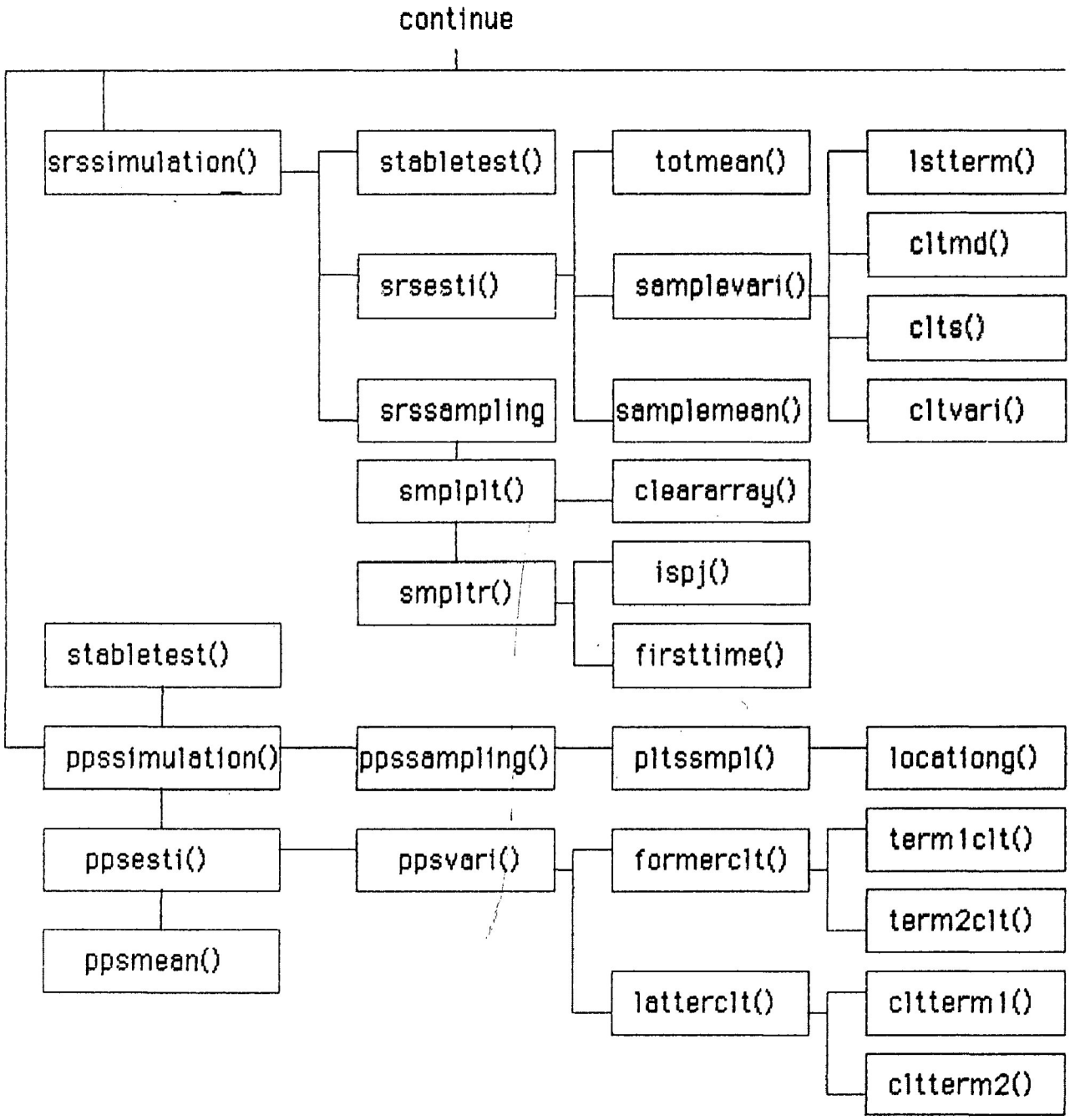


Fig III -1 continued

continue of flow chart of the SIMULATION.C program:



Here is an example how to execute the **SIMULATION.C** program:

```
$cc SIMULATION.C  -lm    < return >
$a.out    < return >
$ENTER LABEL:  K    < return >
$ENTER TOTAL NUMBER OF PLOTS:  3    < return >
$HOW MANY PLOTS WOULD YOU LIKE TO SAMPLE:  2    < return >
$ENTER PLOT NUMBER:  1    < return >
$ENTER PLOT NUMBER:  3    < return >
$ENTER TOTAL NUMBER OF TREES FOR < K1 > :  31    < return >
$ENTER TOTAL NUMBER OF TREES FOR < K3 >:  23    < return >
$WAIT.....
$SUCCESS
```

The simulation main program: **SIMULATION.C**

```
#include <stdio.h>
#include "define.h"
#include "openfile.h"
#include "readhash.h"
#include "getinfo.h"
#include "findpjnum.h"
#include "popuinfo.h"
#include "sampling.h"
#include "closefile.h"
#include "sumbasal.h"

FILE  *rfpt;
FILE  *hfpt;
FILE  *wfpt;
FILE  *tfpt;
FILE  *pfpt;
FILE  *sfpt;
FILE  *cfpt;

int   NP;
int   totplt;
int   totpj;
int   maxsample;
int   pjperplt[ MAX_PLOT ];
int   trperplt[ MAX_PLOT ];
char   pltlab[ MAX_PLOT ];
char   plotno[ MAX_PLOT ][ 6 ];
int   pjnum[ MAX_PLOT ][ MAX_TREE ];
long   hash_tbl[ HASH_TAB_SIZE ];
double volpop[ PJ_NUM ][ MAX_YEAR ];
double baspop[ PJ_NUM ][ MAX_YEAR ];
double volspl[ MAX_SAMPLE_TREE ];
double basspl[ MAX_SAMPLE_TREE ];
double mvolperha[ MAX_YEAR ];
double mbaspertr[ MAX_YEAR ];
double stdrr[ MAX_YEAR ];
double sumbas[ PJ_NUM ][ MAX_PLOT ];
double vppssp[ MAX_SAMPLE_TREE ][ MAX_PLOT ];
double bppssp[ MAX_SAMPLE_TREE ][ MAX_PLOT ];
```

```
main()
{
        int year;

        open_file();
        read_hash();
        get_info();
        find_pj_num();
        popu_info();
        for( year = 0;   year < MAX_YEAR;   year++ )
        {
                sum_basal( year );
                sampling( year );
        }
        close_file();
}
```

Source File: **nodedef.h**

```
struct tree_info
{
        char    plt_lab[ 20 ];
        int     tn;
        char    sp_code[ 3 ];
        int     age;
        int     dbh_age;
        int     last_ck_age;
        double  dbhob;
        double  dbhs[ 100 ];
        double  hghs[ 100 ];
        double  vio;
        double  vols[ 100 ];
        double  s_cor[ 2 ];
        double  c_cor[ 2 ];
        int     next;

};
```

Source File: **openfile.h**

```
#include <assert.h>
#include "define.h"

extern FILE *rfpt;
extern FILE *wfpt;
extern FILE *tfpt;
extern FILE *pfpt;
extern FILE *sfpt;
extern FILE *cfpt;

open_file()
{
    assert( ( rfpt = fopen( "trim.dat",   "r" ) ) != FAILURE );
    assert( ( tfpt = fopen( "true.dat",   "r+" ) ) != FAILURE );
    assert( ( sfpt = fopen( "stdr.dat",   "r+" ) ) != FAILURE );
    assert( (wfpt = fopen( "mean.dat",   "w" )) != FAILURE);
    assert( ( pfpt = fopen( "erre.dat",   "w" ) ) != FAILURE );
    assert( ( cfpt = fopen( "prcn.dat",   "w" ) ) != FAILURE );
}
```

Source File: **basal.h**

```
double basal( prev, next )
double prev;
double next;
{
     double  currarea, prevarea, cbas, pbas, incre,  parameter;
     double  a, b, c;
     c = 2;
     parameter = 15*666.7;
     a = prev/2;
     b = next/2;

     currarea = pow( a, c ) * 3.141592;
     prevarea = pow( b, c ) * 3.141592;
```

```
    cbas = currarea/parameter;
    pbas = prevarea/parameter;
    incre = fabs( cbas - pbas );        /* in case of data error */

    return( incre );
}
```

Source File: **closefile.h**

```
extern FILE *rfpt;
extern FILE *hfpt;
extern FILE *wfpt;
extern FILE *tfpt;
extern FILE *pfpt;
extern FILE *sfpt;
extern FILE *cfpt;

close_file()
{
    fclose( rfpt );
    fclose( hfpt );
    fclose( wfpt );
    fclose( tfpt );
    fclose( pfpt );
    fclose( sfpt );
    fclose( cfpt );

    printf( "SUCCESS\n" );
}
```

Source File: **define.h**

```
#define     SUCCESS                1
#define     FAILURE                0
#define     END                    0
#define     YES                    1
#define     NOT                    0
#define     FOUND                  1
#define     NOT_FOUND              0
#define     DONE                   1
#define     NOT_DONE               0
#define     OK                     0
#define     MINSAMPLE              2
#define     MAX_YEAR              10
#define     MAX_PLOT              20
#define     MAX_TREE             200
#define     PJ_NUM               500
#define     MAX_SAMPLE_TREE      500
#define     MAX_SIMULATION      2500
#define     HASH_TAB_SIZE       4097
#define     ALLOWABLE_ERROR    0.001
```

Source File: **findpjnum.h**

```
#include <string.h>
#include "searchpj.h"

extern  int  totplt;
extern  int  trperplt[ MAX_PLOT ];
extern  int  pjperplt[ MAX_PLOT ];
extern  char pltlab[ MAX_PLOT ];

find_pj_num()
{
    int  i, j;
    char  array[ 20 ];
```

```
    for( i = 0;  i < totplt;  i++ )
    {
        join_plt_lab( i, array );        /* result ex: k1 */
        within_plt( i, array );
    }
    find_min_num();
}


join_plt_lab( i, array )
int  i;
char *array;
{
     char num[ 3 ];
     num[ 0 ] = '\0';
     *array  = '\0';
     strcpy( num,   plotno+i );
     strcat( array, pltlab );
     strcat( array, num );
}


within_plt( i,  array )
int i;
char *array;
{
     int j, indx;

     indx = 0;
     for( j = 1;   j <= trperplt[ i ];   j++ )
     {
         search_pj( i,  array,  j,  &indx );
     }
}


find_min_num()
{
     int i,  temp[ 20 ];

     for( i = 0;  i < totplt;  i++ )
```

```
    {
        temp[ i ] = pjperplt[ i ];
    }
    quick_sort( temp,  temp+totplt-1 );
    maxsample = temp[ 0 ];
}


quick_sort( lower, upper )
int *lower, *upper;
{
    int  partition;
    int  *iptr, *previous_low;

    if( lower < upper )
    {
        partition = *lower;
        previous_low = lower;

        for( iptr = lower+1;  iptr <= upper;  iptr++ )
        {
            if( *iptr < partition )
            {
                previous_low++;
                swap( previous_low, iptr );
            }
        }
        swap( lower, previous_low );
        quick_sort( lower, previous_low - 1 );
        quick_sort( previous_low + 1, upper );
    }
}


swap( left, right )
int *left;
int *right;
{
    int  shelt;
```

```
    shelt = *left;
    *left = *right;
    *right = shelt;
}
```

Source File: **getinfo.h**

```
#include <assert.h>
#include <string.h>
#include "define.h"

extern  char pltlab[ MAX_PLOT ];
extern  int NP;
extern  int totplt;
extern  char plotno[ MAX_PLOT ][ 6 ];
extern  int trperplt[ MAX_PLOT ];

get_info()
{
    assert( get_plt_lab() == SUCCESS );
    assert( get_plot_num() == SUCCESS );
    assert( get_sample_plotno() == SUCCESS );
    assert( get_trperplt() == SUCCESS );

    printf( "WAIT.....\n" );
}


get_plt_lab()
{
    char *cpointer;

    printf( "\n\nENTER LABEL:" );
    fgets( pltlab,  20,  stdin );

    cpointer = strchr( pltlab, '\n' );
    *cpointer = '\0';
    printf( "\n" );
```

```
    return( SUCCESS );
}


get_plot_num()
{
    printf( "HOW MANY PRIMATIVE PLOTS:" );
    scanf( "%d",  &NP );
    getchar();
    printf( "\n" );

    printf( "HOW MANY PLOTS YOU WANT TO SAMPLE:" );
    scanf( "%d",  &totplt );
    getchar();
    printf( "\n" );

    return( SUCCESS );
}


get_sample_plotno()
{
    int  i;
    char *cptr;

    for( i = 0;   i < totplt;   i++ )
    {
        printf( "ENTER SAMPLED PLOT NUMBER:" );
        fgets( plotno+i,  6,  stdin );
        cptr = strchr( plotno+i,   '\n' );
        *cptr = '\0';
        printf( "\n" );
    }
    return( SUCCESS );
}
```

```
get_trperplt()
{
   int  i;

   for( i = 0;  i < totplt;  i++ )
   {
      printf( "ENTER TOTAL NUMBER OF TREES IN %s%s:", pltlab,
                                                      plotno+i );
      scanf( "%d", trperplt+i );
      getchar();
      printf( "\n" );
   }
   printf( "\n" );
   return( SUCCESS );
}
```

Source File: **formerclt.h**

```
#include<math.h>
#include "define.h"

extern int    totplt;
extern int    pjperplt[ MAX_PLOT ];
extern double vppssp[ MAX_SAMPLE_TREE ][ MAX_PLOT ];
extern double bppssp[ MAX_SAMPLE_TREE ][ MAX_PLOT ];
extern double sumbas[ PJ_NUM ][ MAX_PLOT ];

double former_clt( size )
int size;
{
     int  i;
     double  a,  c, item1, item2, result;
     double  item1_clt(), item2_clt();

     c = 2;
     result = 0;
     item2 = item2_clt( size );
     for( i = 0;   i < totplt;   i++ )
```

```
        {
            item1 = item1_clt( size,  i );
            a = fabs( item1 - item2 );
            result += pow( a,  c );
        }
        return( result );
}


double item1_clt( size,  i )
int  size;
int  i;
{
        int    p,  mi,  Mi;
        double sum,  result;

        mi = size;
        sum = 0;
        Mi = pjperplt[ i ];

        for( p = 0;    p < mi;    p++ )
        {
            sum += vppssp[ p ][ i ]/bppssp[ p ][ i ];
        }
        result = ( sumbas[ Mi-1 ][ i ]*sum )/mi;

        return( result );
}


double item2_clt( size )
int  size;
{
        int    j,  p,  Mi;
        double sum,  tot,  mi,  result;

        sum = 0;
        tot = 0;
        mi = size;
        for( j = 0;    j < totplt;    j++ )
```

```
    {
        Mi = pjperplt[ j ];
        for( p = 0;   p < mi;   p++ )
        {
            sum += vppssp[ p ][ j ]/bppssp[ p ][ j ];
        }
        tot += (sumbas[ Mi-1 ][ j ] * sum)/mi;
    }
    result = tot/totplt;

    return( result );
}
```

Source File: **latterclt.h**

```
#include <math.h>

extern int    totplt;
extern int    NP;
extern int    pjperplt[ MAX_PLOT ];
extern double sumbas[ PJ_NUM ][ MAX_PLOT ];
extern double vppssp[ MAX_SAMPLE_TREE ][ MAX_PLOT ];
extern double bppssp[ MAX_SAMPLE_TREE ][ MAX_PLOT ];

double latter_clt( size )
int size;
{
    int    j;
    double item1, item2, sum, n, N, result;
    double clt_item1(), clt_item2();

    sum = 0;
    n = totplt;
    N = NP;
    for( j = 0;   j < totplt;   j++ )
    {
        item1 = clt_item1( size, j );
        item2 = clt_item2( size, j );
```

```
        sum += item1 * item2;
    }
    result = sum/(N*n);

    return( result );
}


double clt_item1( size,  j )
int size;
int j;
{
    int    Mi;
    double c, mi, sumpltbas, sum, squ, result;

    c = 2;
    Mi = pjperplt[ j ];
    mi = size;
    sumpltbas = sumbas[ Mi - 1 ][ j ];
    squ = pow( sumpltbas,  c );
    result = (squ*(Mi - mi))/(Mi*mi*(mi-1));

    return( result );
}


double clt_item2( size,  j )
int size;
int j;
{
    int    p;
    double a, b, c, mi, sum, result;

    c = 2;
    sum = 0;
    for( p = 0;   p < size;   p++ )
    {
        sum += vppsspl[ p ][ j ]/bppsspl[ p ][ j ];
    }
    a = sum/size;
```

```
        b = fabs( sum - a );
        result = pow( b,  c );

        return( result );
}
```

Source File: **getstdr.h**

```
extern FILE  *sfpt;
extern double stdrr[ MAX_YEAR ];

get_stdr( year )
int year;
{
    int i;

    rewind( sfpt );

    for( i = 0;   i < year;   i++ )
    {
        fscanf( sfpt,  "%*s" );
    }
    fscanf( sfpt,  "%lf",  &stdrr[ year ] );
}
```

Source File: **getruemean.h**

```
extern FILE   *tfpt;
extern double  mvolperha[ MAX_YEAR ];

get_true_mean( year )
int year;
{
    int  i;

    rewind( tfpt );

    for( i = 0;  i < year;   i++ )
    {
```

```
        fscanf( tfpt, "%*s" );
    }
    fscanf( tfpt,  "%lf",   &mvolperha[ year ] );
}
```

Source File: **getvolbas.h**

```
#include "define.h"
#include "basal.h"

extern int    totpj;
extern double volpop[ PJ_NUM ][ MAX_YEAR ];
extern double baspop[ PJ_NUM ][ MAX_YEAR ];

get_vol_bas( temp )
struct tree_info temp;
{
     int    i;
     double prev, next;
     double basal();

     for( i = 0;   i < MAX_YEAR;   i++ )
     {
        volpop[totpj][i] = temp.vols[i] - temp.vols[i+1]; /* unit: m^3 */
        prev = temp.dbhs[ i ];                          /* unit: mm */
        next = temp.dbhs[ i+1 ];
        baspop[ totpj ][ i ] = basal( prev, next );     /* mm^2/ha. */

     }
}
```

Source File: **hash.fun**

```
#include <math.h>
#include "define.h"

hash( label,  trnum )
char *label;
```

```
int trnum;
{
      int    num;
      char   cbuff[ 100 ];
      char   *cptr;
      double constnt, position;
      long   hash_value;

      hash_value = 0;
      position = 1;
      constnt = 3;
      num = trnum;

      sprintf( cbuff, "%d", num );
      cptr = cbuff;

      while( *label++ != '\0' )
      {
         hash_value += pow(position, constnt )*( *(label-1) );
         position++;
      }
      while( *cptr++ != '\0' )
      {
         hash_value += pow( position, constnt ) * ( *(cptr-1) );
         position++;
      }
      return( (int) ( hash_value%HASH_TAB_SIZE ) );
}
```

Source File: **ischeck.h**

```
#include "define.h"

extern int pjnum[ MAX_PLOT ][ MAX_TREE ];
extern int pjperplt[ MAX_PLOT ];

isfirsttime( chosennum, array, j )
int chosennum;
```

```
int *array, j;
{
     int p;

     for( p = 0;  p < j;  p++ )
     {
        if( *(array+p) == chosennum )
        {
            return( NOT );
        }
     }
     return( YES );
}


int ispj( i, chosennum )
int i;
int chosennum;
{
     int position;

     for( position = 0;  position < pjperplt[ i ];  position++ )
     {
        if( pjnum[ i ][ position ] == chosennum )
        {
            return( position + 1 );        /*in case zero position */
        }
     }
     return( NOT );
}
```

Source File: **pltsmpl.h**

```
extern int    pjperplt[ MAX_PLOT ];
extern double volpop[ PJ_NUM ][ MAX_YEAR ];
extern double baspop[ PJ_NUM ][ MAX_YEAR ];
extern double sumbas[ PJ_NUM ][ MAX_PLOT ];
extern double vppssp[ MAX_SAMPLE_TREE ][ MAX_PLOT ];
extern double bppssp[ MAX_SAMPLE_TREE ][ MAX_PLOT ];

plt_smpl( k, year, j, count )
```

```
int k;
int year;
int j;
int *count;
{
     int p, end, indx, max;
     double num, randnum;

     end = pjperplt[ j ];
     max = sumbas[ end - 1 ][ j ] * 1000000;/* ! rescale basal area */

     for( p = 0;   p < k;   p++ )
     {
        randnum = rand()%(max+1);  /* number ranging: 0.....max */
        num = randnum/1000000;              /*!scale back */
        indx = locating( j, num );
        vppssp[ p ][ j ] = volpop[ *count+indx ][ year ];
        bppssp[ p ][ j ] = baspop[ *count+indx ][ year ];
     }
     *count += end;                    /* enter next plot data field */
}


locating( j, num )
int   j;
double num;
{
    int h;

    for( h = 0;    h < pjperplt[ j ];    h++ )
    {
       if( num <= sumbas[ h ][ j ] )
       {
          return( h );
       }
    }
}
```

Source File: **popuinfo.h**

```
#include <assert.h>
#include "define.h"
#include "popumean.h"
#include "popustdr.h"

extern int totplt;
extern int NP;

popu_info()
{
    double aver[ MAX_YEAR ];

    assert( popu_mean( aver ) == SUCCESS );
    if( totplt == NP )
    {
       assert( popu_stdr( aver ) == SUCCESS );
       assert( save_stdr() == SUCCESS );
    }
}
```

Source File: **popumean.h**

```
#include <math.h>
#include "define.h"

extern FILE   *tfpt;
extern int    totplt;
extern int    totpj;
extern double mvolperha[ MAX_YEAR ];
extern double mbaspertr[ MAX_YEAR ];
extern double volpop[ PJ_NUM ][ MAX_YEAR ];
extern double baspop[ PJ_NUM ][ MAX_YEAR ];

popu_mean( aver )
double *aver;
{
```

```
    int    i, j;
    double vincre, bincre, aveperplt, parameter;

    parameter = 15*666.7/100;
    vincre = 0;
    bincre = 0;

    for( i = 0;   i < MAX_YEAR;   i++ )
    {
       for( j = 0;   j < totpj;    j++ )
       {
          vincre += volpop[ j ][ i ];
          bincre += baspop[ j ][ i ];
       }
       aveperplt = vincre/totplt;            /* average vol/plot */

       if( totplt == NP )
       {
          mvolperha[ i ] = aveperplt*parameter; /* mean vol/ha. */
         *( aver+i ) = vincre/totpj;              /* mean vol/tree */
       }
       mbaspertr[ i ] = bincre/totpj;   /* average basal area/tree */
                                        /* unit: mm^2 of a tree/ha.*/
       vincre = 0;
       bincre = 0;
    }
    if( totplt == NP )
    {
       save_true_mean();
    }
    return( SUCCESS );
}


save_true_mean()
{
   int  i;
   char string[ 20 ];

   for( i = 0;   i < MAX_YEAR;   i++ )
```

```
    {
        sprintf( string,  "%lf",   mvolperha[ i ] );
        fprintf( tfpt,    "%s\n",  string );
    }
}
```

Source File: **popustdr.h**

```
#include <math.h>
#include "define.h"

extern FILE  *sfpt;
extern int   totpj;
extern double stdrr[ MAX_YEAR ];
extern double volpop[ PJ_NUM ][ MAX_YEAR ];

popu_stdr( aver )
double *aver;
{
    int    i, j;
    double c, diff, squ, sum;
    double trv, totv, hacv, hase;
    sum = 0;
    c = 2;

    for( i = 0;   i < MAX_YEAR;   i++ )
    {
        for( j = 0;   j < totpj;   j++ )
        {
            diff = fabs( volpop[j][i] - *(aver+i) );
            squ = pow( diff, c );
            sum += squ;
        }
        trv = sum/totpj;
        totv = pow( (double)totpj,  c ) * trv;
        hacv = totv * 15*666.7/(3*100);
        stdrr[ i ] = sqrt( hacv );
        sum = 0;
    }
```

```
    return( SUCCESS );
}
save_stdr()
{
    int  i;
    char string[ 20 ];

    for( i = 0;   i < MAX_YEAR;   i++ )
    {
        sprintf( string,  "%lf",  stdrr[ i ] );
        fprintf( sfpt,    "%s\n", string );
    }
    return( SUCCESS );
}
```

Source File: **ppsesti.h**

```
#include <assert.h>
#include "define.h"
#include "ppsmean.h"
#include "ppsvari.h"

pps_esti( n,  m,  v )
int   n;
double *m;
double *v;
{
    assert( pps_mean( n,   m ) == SUCCESS );
    assert( pps_vari( n,   v ) == SUCCESS );

    return( SUCCESS );
}
```

Source File: **ppsmean.h**

```
#include "define.h"
```

```
extern int   totplt;
extern int   totpj;
extern int   pjperplt[ MAX_PLOT ];
extern double vppssp[ MAX_SAMPLE_TREE ][ MAX_PLOT ];
extern double bppssp[ MAX_SAMPLE_TREE ][ MAX_PLOT ];
extern double sumbas[ PJ_NUM ][ MAX_PLOT ];

pps_mean( n,  m )
int   n;
double *m;
{
    int    i, p, mi, Mi;
    double sum, parameter, plotvol, totvol, avepltvol;

    sum = 0;
    totvol = 0;
    mi = n;
    parameter = 15*666.7/100;

    for( i = 0;   i < totplt;   i++ )
    {
        Mi = pjperplt[ i ];
        for( p = 0;   p < mi;   p++ )
        {
            sum += vppssp[ p ][ i ]/bppssp[ p ][ i ];
        }
        plotvol = ( sumbas[ Mi-1 ][ i ]*sum )/mi;
        totvol += plotvol;

        sum = 0;
    }
    avepltvol = totvol/totplt;              /* mean plot volume */
    *m = avepltvol*parameter;                 /* estimated vol/ha. */

    return( SUCCESS );
}
```

Source File: **ppssampling.h**

```
#include "define.h"
```

```
#include "pltsmpl.h"


extern int    totplt;
extern int    pjperplt[ MAX_PLOT ];
extern double totbas;
extern double baspop[ PJ_NUM ][ MAX_YEAR ];
extern double sumbas[ PJ_NUM ][ MAX_PLOT ];

pps_sampling( k,  year )
int k;
int year;
{
     int j, count;

     count = 0;
     for( j = 0;    j < totplt;    j++ )
     {
        plt_smpl( k, year, j, &count );
                                        /* select sample within plot */
     }
     return( SUCCESS );
}
```

Source File: **ppssimu.h**

```
#include <assert.h>
#include "define.h"
#include "ppssampling.h"
#include "ppsesti.h"

pps_simu( n, year, mean, vari )
int  n;
int  year;
double *mean;
double *vari;
{
     int    p;
     double prevsum, currsum, vsum;
```

```
        prevsum = 0;
        currsum = 0;
        vsum = 0;
        for( p = 0;   p < MAX_SIMULATION;   p++ )
        {
            srand( n+p+98 );            /* set seed starting from 100 */
            assert( pps_sampling( n, year ) == SUCCESS );
            assert( pps_esti( n, mean, vari ) == SUCCESS );
            if( stable_test(p, mean, vari, &vsum, &prevsum, &currsum)
                                                        == YES )
            {
                break;              /* terminate loop */
            }
        }
}
```

Source File: **ppsvari.h**

```
#include <math.h>
#include "define.h"
#include "formerclt.h"
#include "latterclt.h"

extern int totplt;
extern int NP;

pps_vari( size,  v )
int   size;
double *v;
{
    double n, N;
    double c, former, latter, varperplt,  parameter;
    double former_clt(), latter_clt();

    n = totplt;
    N = NP;
    parameter = 15*666.7/(3*100);
```

```
    former = former_clt( size );
    latter = latter_clt( size );
    varperplt = ((N-n)/(N*n*(n-1))) * former + latter;  /*
variance/plot */
    *v = pow( NP,  c )*varperplt*parameter;

    return( SUCCESS );
}
```

Source File: **readhash.h**

```
#include <assert.h>
#include "define.h"

extern  FILE  *hfpt;
extern  long  hash_tbl[ HASH_TAB_SIZE ];

read_hash()
{
   open_hash();

   if( fread( hash_tbl,  sizeof(long),  HASH_TAB_SIZE,  hfpt )
                                          < HASH_TAB_SIZE )
   {
        fprintf( stderr,  "%s\n",  "Hash Table Corrupted" );
        exit( 1 );
   }
}


open_hash()
{
   if( hfpt == 0 )
   {
      assert( ( hfpt = fopen( "hash.tbl",  "r" ) ) !=  FAILURE );
   }
   else
   {
      fseek( hfpt,  (long)0,  0 );
   }
```

```
}
```

Source File: **samplemean.h**

```
#include "define.h"

extern int  totplt;
extern int  totpj;
extern double mbaspertr[ MAX_YEAR ];

sample_mean( year, vsum, bsum, mean )
int    year;
double vsum;
double bsum;
double *mean;
{

     double  parameter, tot;

     parameter = 15*666.7/100;
     tot = mbaspertr[ year ]*(vsum/bsum)*totpj;
     *mean = (tot*parameter)/totplt;          /* average volum/ha. */

     return( SUCCESS );
}
```

Source File: **samplevari.h**

```
#include <assert.h>
#include <math.h>
#include "define.h"

extern int   NP;
extern int   totplt;
extern int   totpj;
extern int   pjperplt[ MAX_PLOT ];
extern double volspl[ MAX_SAMPLE_TREE ];
```

```
extern  double basspl[ MAX_SAMPLE_TREE ];
extern  double mbaspertr[ MAX_YEAR ];



sample_vari( k, vsum, bsum, vari, yave, xave )
int   k;
double vsum, bsum;
double *vari, *yave, *xave;
{
     int  j;
     double  r, term;
     double  md[ MAX_PLOT ], s2[ MAX_PLOT ];
     double  d[ MAX_SAMPLE_TREE ];

     r = vsum/bsum;

     assert( lst_term( r, &term, yave, xave ) == SUCCESS );
     assert( clcl_md( k, r, md, d ) == SUCCESS );
     assert( clcl_s( k, s2, md, d ) == SUCCESS );
     assert( clcl_v( k, s2, term, vari ) == SUCCESS );

     return( SUCCESS );
}


lst_term( r, term, yave, xave )
double  r;
double  *term, *yave, *xave;
{
     int   i, np;
     double a, b, c, f1, n, sum, Mi, Msqu, y_rx;

     n = totplt;
     c = 2;
     sum = 0;
     f1 = (double)totplt/(double)NP;

     for( i = 0;   i < n;   i++ )
     {
```

```
        Mi = pjperplt[ i ];
        Msqu = pow( Mi, c );
        b = *( yave+i ) - ( *(xave+i)*r );
        y_rx = fabs( b );
        a = pow( y_rx, c );
        sum += (Msqu * a)/(n - 1 );
    }
    *term = ( (1 - f1)*sum )/n;

    return( SUCCESS );
}


clcl_md( k, r, md, d )
int   k;
double r;
double *md;
double *d;
{
    int    j,  p, line;
    double sum, vincre, bincre, dtemp;

    line = 0;
    sum = 0;

    for( j = 0;   j < totplt;   j++ )
    {
        for( p = 0;   p < k;   p++ )
        {
            vincre = volspl[ line + p ];
            bincre = basspl[ line + p ];
            dtemp = fabs( vincre - (r * bincre) );
            *( d + line + p ) = dtemp;
            sum += dtemp;
        }
        *( md+j ) = sum/k;
        line += k;
        sum = 0;
    }
```

```
    return( SUCCESS );
}



clcl_s( k, s2, md, d )
int   k;
double *s2;
double *md;
double *d;
{
      int    j, p, line;
      double a, b, c, h, squ,  what;

      line = 0;
      squ = 0;
      c = 2;

      for( j = 0;   j < totplt;   j++ )
      {
         for( p = 0;   p < k;   p++ )
         {
            a = *(d+line+p);
            b = *(md+j);
            h = fabs( a-b );
            squ += pow( h, c );
         }
         *( s2+j ) = squ/(k-1);
         line += k;
         squ = 0;
      }
      return( SUCCESS );
}


clcl_v( k,  s2, term, vari )
int   k;
double *s2;
double term;
double *vari;
```

```
{
    int    i;
    double b, c, f1, f2, Mi, Mi2, Mo, Mo2, n, n2, N;
    double mvar, sum, convert, temp;

    sum = 0;
    c = 2;
    Mo = totpj;
    N = NP;
    n = totplt;
    f1 = n/N;
    n2 = pow( n, c );
    Mo2 = pow( Mo, c );
    convert = 15*666.7/100;

    for( i = 0;  i < totplt;  i++ )
    {
        Mi = pjperplt[ i ];
        Mi2 = pow( Mi, c );
        f2 = k/Mi;
        temp = (Mi2*(1-f2)*(*(s2+i)));
        sum += temp;
    }
    mvar = term + (f1/n2)*(sum/k);          /* variance for mean */
    mvar = ( Mo2 * mvar );              /* total variance for <totplt> */
    *vari = (mvar*convert)/totplt;    /* converted to: variance/ha. */
    *vari = *vari/10;

    return( SUCCESS );
}
```

Source File: **sampling.h**

```
#include "define.h"
#include "srssimu.h"
#include "ppssimu.h"
#include "writetofile.h"
```

```
extern int maxsample;


sampling( year )
int year;
{
     int    n;
     double msrs, vsrs, mpps, vpps;

     for( n = MINSAMPLE;   n < maxsample;    n++ )
     {
        srs_simu( n, year, &msrs, &vsrs );
        pps_simu( n, year, &mpps, &vpps );
        write_to_file( n, year,  msrs,  vsrs, mpps, vpps );
     }
}
```

Source File: **searchpj.h**

```
#include "define.h"
#include "nodedef.h"
#include "hash.fun"
#include "getvolbas.h"

extern FILE *rfpt;
extern int  totpj;
extern int  maxsample;
extern int  pjperplt[ MAX_PLOT ];
extern long hash_tbl[ HASH_TAB_SIZE ];
extern int  pjnum[ MAX_PLOT ][ MAX_TREE ];

search_pj( i,  array,  j, indx )
int i;
char *array;
int j;
int *indx;
{
```

```
    struct tree_info temp;
    int     record;

    for( record = hash_tbl[ hash(array, j) ]; record != END &&
            r_record( record, &temp ); record = temp.next )
    {
        if( !strcmp(temp.plt_lab, array) && temp.tn == j &&
                            !strcmp( temp.sp_code, "Pj" ) )
        {
            pjnum[ i ][ *indx ] = j;      /* remember tree # which is pj */
            *indx += 1;
            pjperplt[ i ] += 1;          /* remember how many pj trees */
                                         /* in each secondary plots */
            get_vol_bas( temp );
            totpj++;

            return;
        }
    }
}


r_record( record, temp )
int record;
struct tree_info *temp;
{

   if( fseek( rfpt, (long)(( record-1 )*sizeof(struct tree_info)), 0) !=
OK )
   {
        printf( "\nSEEK ERROR\n" );

        return( FAILURE );
   }
   if( fread( temp, sizeof( struct tree_info ), 1, rfpt ) == FAILURE )
   {
        printf( "\nRECORD NOT FOUND\n" );

        return( FAILURE );
   }
   return( SUCCESS );
}
```

Source File: **smplplt.h**

```
#include <assert.h>
#include "define.h"
#include "smpltr.h"

smpl_plt( n,  year,  i,  count,  line )
int  n;
int  year;
int  i;
int  *count;
int  line;
{
     int  j;
     int  array[ 100 ];

     clear_arr( array );

     for( j = 0;  j < n;  j++ )
     {
        smpl_tr( year,  i,  count,  line,  j,  array );
     }
}


clear_arr( array )
int *array;
{
    int  i;

    for( i = 0;  i < 100;  i++ )
    {
       *( array+i ) = 0;
    }
}
```

Source File: **smpltr.h**

```
#include "define.h"
#include "ischeck.h"

extern int    trperplt[ MAX_PLOT ];
extern double volpop[ PJ_NUM ][ MAX_YEAR ];
extern double baspop[ PJ_NUM ][ MAX_YEAR ];
extern double volspl[ MAX_SAMPLE_TREE ];
extern double basspl[ MAX_SAMPLE_TREE ];

smpl_tr( year, i, count, line, j, array )
int year;
int i;
int *count;
int line;
int j;
int *array;
{
   int max, chosennum, indx;
   int isfirsttime(), ispj();

   max = trperplt[ i ];

   while( chosennum = rand() )
   {
      if( chosennum )          /* if it happens to be 0, excluding it */
      {
         if( chosennum = chosennum%(max+1) )
                                    /* number range: 1...max */
         {
            if( isfirsttime( chosennum, array, j )&&
                                    (indx = ispj( i, chosennum )) )
            {
               *( array+j ) = chosennum;
               indx = indx - 1;                  /* restore its value */
               volspl[ *count ] = volpop[ indx + line ][ year ];
               basspl[ *count ] = baspop[ indx + line ][ year ];
               *count += 1;
```

```
                    break;
                }
            }
        }
    }
}
```

Source File: **srsesti.h**

```
#include <assert.h>
#include "define.h"
#include "totmean.h"
#include "samplemean.h"
#include "samplevari.h"

srs_esti( k, year, mean, vari )
int k;
int year;
int *mean;
int *vari;
{
    double vsum, bsum, yave[ MAX_PLOT ], xave[ MAX_PLOT ];

    vsum = 0;
    bsum = 0;

    assert( tot_mean( k, &vsum, &bsum, yave, xave ) == SUCCESS );
    assert( sample_mean( year, vsum, bsum, mean ) == SUCCESS );
    assert(sample_vari(k, vsum, bsum, vari, yave, xave)==SUCCESS );

    return( SUCCESS );
}
```

Source File: **srssampling.h**

```
#include "define.h"
#include "smplplt.h"
```

```
extern int totplt;
extern int pjperplt[ MAX_PLOT ];

srs_sampling( n,  year )
int n;
int year;
{
    int i, count, line;

    count = 0;
    line = 0;

    for( i = 0;   i < totplt;   i++ )
    {
        smpl_plt( n, year, i, &count, line );
        line += pjperplt[ i ];
    }
    return( SUCCESS );
}
```

Source File: **srssimu.h**

```
#include <assert.h>
#include "define.h"
#include "srssampling.h"
#include "srsesti.h"
#include "stabletest.h"

srs_simu( n, year, mean, vari )
int   n;
int   year;
double *mean;
double *vari;
{
     int    p;
     double prevsum, currsum, vsum;

     prevsum = 0;
```

```
        currsum = 0;
        vsum = 0;
        for( p = 0;   p < MAX_SIMULATION;   p++ )
        {

            srand( n+p-1 );              /* set seed starting from 1 */
            assert( srs_sampling( n, year ) == SUCCESS );
            assert( srs_esti( n, year, mean, vari ) == SUCCESS );
            if( stable_test(p, mean, vari, &vsum, &prevsum, &currsum)
                                                          == YES )
            {
                 break;              /* terminate loop */
            }
        }
}
```

Source File: **sumbasal.h**

```
extern int   totplt;
extern int   pjperplt[ MAX_PLOT ];
extern double baspop[ PJ_NUM ][ MAX_YEAR ];
extern double sumbas[ PJ_NUM ][ MAX_PLOT ];

sum_basal( year )
int year;
{
     int i, p, count;
     double sum;

     count = 0;
     sum = 0;

     for( i = 0;   i < totplt;   i++ )
     {
        for( p = 0;   p < pjperplt[ i ];   p++ )
        {
            sum += baspop[ count + p ][ year ];
            sumbas[ p ][ i ] = sum;
        }
        sum = 0;
        count += p;
```

```
    }
}
```

Source File: **totmean.h**

```
#include "define.h"

extern  int    totplt;
extern  int    pjperplt[ MAX_PLOT ];
extern  double volspl[ MAX_SAMPLE_TREE ];
extern  double basspl[ MAX_SAMPLE_TREE ];

tot_mean( k, vsum, bsum, yave, xave )
int   k;
double *vsum, *bsum;
double *yave, *xave;
{
    int    j, p, line;
    double vincre, bincre, vaver, baver;

    line = 0;
    vincre = 0;
    bincre = 0;

    for( j = 0;   j < totplt;   j++ )
    {
        for( p = 0;   p < k;   p++ )
        {
            vincre += volspl[ line + p ];
            bincre += basspl[ line + p ];
        }
        vaver = vincre/k;
        baver = bincre/k;
        *(yave+j) = vaver;
        *(xave+j) = baver;
        *vsum += vaver * pjperplt[ j ];
        *bsum += baver * pjperplt[ j ];
        vincre = 0;                    /* reseted for next plot */
        bincre = 0;
        line += k;
    }
    return( SUCCESS );
```

```
}
```

Source File: **writetofile.h**

```
#include "writefile1.h"
#include "writefile2.h"
#include "writefile3.h"

write_to_file( k, year, msrs, vsrs, mpps, vpps )
int   k, year;
double msrs, vsrs;
double mpps, vpps;
{
     write_file1( k, year, msrs, mpps );
     write_file2( k, year, vsrs, vpps );
     write_file3( k, year, msrs, vsrs, mpps, vpps );
}
```

Source File: **writefile1.h**

```
#include <math.h>
#include "gettruemean.h"

extern int    totplt;
extern int    NP;
extern int    maxsample;
extern FILE   *wfpt;
extern double mvolperha[ MAX_YEAR ];
extern char   pltlab[ MAX_PLOT ];
extern char   plotno[ MAX_PLOT ][ 6 ];


write_file1( k, year, msrs, mpps )
int   k,   year;
double msrs, mpps;
{
    static int interval = 0;

    double a;
    char   ms[ 20 ], mp[ 20 ];
```

```
    char  size[ 20 ], time[ 20 ], truemean[ 20 ];

    if( totplt < NP )
    {
       get_true_mean( year );
    }
    a = mvolperha[ year ];
    to_char(msrs, mpps, a,  ms, mp, truemean, k, year, size, time );

    if( interval == year )
    {
       write_title( time );
       write_result( k,  size,  truemean,  ms,   mp );

       interval += 1;
    }
    else
    {
      write_result( k,  size,  truemean,   ms,  mp );
    }
}


to_char( msrs, mpps, a, ms,  mp, truemean, k, year, size, time )
double msrs, mpps, a;
char  *ms, *mp, *truemean;
int   k,  year;
char  *size, *time;
{
     year = year + 1;

     sprintf( ms,   "%lf",   msrs );
     sprintf( mp,   "%lf",   mpps );
     sprintf( truemean,  "%lf",   a );
     sprintf( size,  "%d",    k );
     sprintf( time,  "%d",    year );
}


write_title( time )
char *time;
```

```
{
    int  i,  j;

    pr_header1();

    for( i = 0;    i < totplt;    i++ )
    {
       fprintf( wfpt,  "%s%s%s",  pltlab,  plotno+i,  "    " );
    }
    fprintf( wfpt,  "\n\n\n%8s%-11s",  " ", "Year:" );
    fprintf( wfpt,  "%s%s\n",     "-", time );
    fprintf( wfpt,  "%8s%s\n\n",    " ",  "(backward)" );

    fprintf( wfpt,  "%8s%-14s",  " ", "Number of" );
    fprintf( wfpt,  "%-17s",  "TRIM" );
    fprintf( wfpt,  "%-17s",  "Mean for" );
    fprintf( wfpt,  "%-14s\n", "Mean for" );

    fprintf( wfpt,  "%8s%-14s",  " ", "Trees in" );
    fprintf( wfpt,  "%-17s",  "Mean for" );
    fprintf( wfpt,  "%-17s",  "Unequal Pr" );
    fprintf( wfpt,  "%-14s\n",  "Equal Pr" );

    fprintf( wfpt,  "%8s%-14s",  " ", "Subsample" );
    fprintf( wfpt,  "%-17s",  "Stand" );
    fprintf( wfpt,  "%-17s",  "Subsampling" );
    fprintf( wfpt,  "%-14s\n",  "Subsampling" );

    fprintf( wfpt,  "%22s%-17s",  " ", "(m^3/ha.)" );
    fprintf( wfpt,  "%-17s",  "(m^3/ha.)" );
    fprintf( wfpt,  "%-14s\n",  "(m^3/ha.)" );

    fprintf( wfpt, "%8s",  " " );
    for( j = 0;    j < 59;    j++ )
    {
       fprintf( wfpt, "%s", "-" );
    }
    fprintf( wfpt, "%s",  "\n\n" );
}
```

```
write_result( k, size, truemean, ms, mp )
int k;
char *size;
char *truemean;
char *ms;
char *mp;
{
    int i, j;

    i = maxsample - 1;

    fprintf( wfpt,  "%13s",  size );
    fprintf( wfpt,  "%18s",  truemean );
    fprintf( wfpt,  "%17s",  ms );
    fprintf( wfpt,  "%17s\n",  mp );

    if( i == k )
    {
       fprintf( wfpt,  "%s",   "\n" );
       fprintf( wfpt,  "%8s",  " " );
       for( j = 0;   j < 59;   j++ )
       {
          fprintf( wfpt,  "%s",  "-" );
       }
       fprintf( wfpt,  "%s",  "\n\n\n\n\n\n" );
    }
}


pr_header1()
{

  fprintf( wfpt, "%14s%s\n", " ", "Accuracy of estimates for  mean
                                          volume per hectare" );
  fprintf( wfpt, "%14s%s\n",   " ",     "for both equal and unequal
                                        probability subsampling" );
  fprintf( wfpt, "%14s%s%s",   " ", "rules for plots:", "  " );
}
```

Source File: **writefile2.h**

```
#include <math.h>
#include "getstdr.h"

extern  FILE  *pfpt;
extern  int   totplt;
extern  int   NP;
extern  double stdrr[ MAX_YEAR ];
extern  char  pltlab[ MAX_PLOT ];
extern  char  plotno[ MAX_PLOT ][ 6 ];

write_file2( k, year, vsrs, vpps )
int   k, year;
double vsrs, vpps;
{

     static int ring = 0;
     double se;
     char   eqv[ 20 ], uqv[ 20 ];
     char   size[ 20 ], time[ 20 ], strse[ 20 ];

     if( totplt < NP )
     {
        get_stdr( year );
     }
     se = stdrr[ year ];
     convt_to_char(eqv, uqv, strse, size, time, vsrs, vpps, se, k, year);

     if( ring == year )
     {
        wrt_title( time );
        wrt_result( k, size, eqv,  uqv, strse );

        ring += 1;
     }
     else
     {
        wrt_result( k, size, eqv, uqv, strse );
```

```
    }
}

convt_to_char( eqv, uqv, strse, size, time, vsrs, vpps, se, k, year )
char  *eqv, *uqv;
char  *strse, *size, *time;
double vsrs, vpps, se;
int   k,   year;
{
    double uqroot, eqroot;

    year = year+1;

    vpps = fabs( vpps );
    uqroot = sqrt( vpps );
    eqroot = sqrt( vsrs );

    sprintf( eqv,  "%lf",  eqroot );
    sprintf( uqv,  "%lf",  uqroot );
    sprintf( size,  "%d",  k );
    sprintf( time,  "%d",  year );
    sprintf( strse, "%lf", se );
}


wrt_title( time )
char *time;
{
    int i, j;

    print_header2();

    for( i = 0;   i < totplt;   i++ )
    {
       fprintf( pfpt,  "%4s%s",  pltlab,  plotno+i );
    }
    fprintf( pfpt,  "\n\n\n%11s%-11s",  " ", "Year:" );
    fprintf( pfpt,  "%s%s\n",  "-",  time );
    fprintf( pfpt,  "%11s%s\n\n",   " ", "(backward)" );
```

```
    fprintf( pfpt,  "%11s%-15s",  " ", "Number of" );
    fprintf( pfpt,  "%-15s", "S. E. for" );
    fprintf( pfpt,  "%-15s",  "S. E. for" );
    fprintf( pfpt,  "%-15s\n", "S. E. for" );

    fprintf( pfpt,  "%11s%-15s", " ", "Trees in" );
    fprintf( pfpt,  "%-15s",  "TRIM plots" );
    fprintf( pfpt,  "%-15s",  "Unequal Pr" );
    fprintf( pfpt,  "%-15s\n", "Equal Pr" );

    fprintf( pfpt,  "%11s%-15s",  " ", "Subsample" );
    fprintf( pfpt,  "%-15s",  "Stand" );
    fprintf( pfpt,  "%-15s",  "Subsampling" );
    fprintf( pfpt,  "%-15s\n", "Subsampling" );

    fprintf( pfpt,  "%26s%-15s", " ", "(m^3/ha.)" );
    fprintf( pfpt,  "%-15s",   "(m^3/ha.)" );
    fprintf( pfpt,  "%-15s\n", "(m^3/ha.)" );

    fprintf( pfpt,  "%11s",  " " );
    for( j = 0;   j < 56;   j++ )
    {
       fprintf( pfpt,   "%s",  "-" );
    }
    fprintf( pfpt,   "%s",   "\n\n" );
}


wrt_result( k, size, eqv, uqv, strse )
int  k;
char *size, *eqv;
char *uqv, *strse;
{
    int i, j;

    i = maxsample - 1;

    fprintf( pfpt,  "%16s",  size );
    fprintf( pfpt,  "%19s", strse );
```

```
    fprintf( pfpt,  "%15s", uqv );
    fprintf( pfpt,  "%15s\n", eqv );
    if( i == k )
    {
      fprintf( pfpt,  "%s",  "\n" );
      fprintf( pfpt,  "%11s", " " );

      for( j = 0;   j < 56;   j++ )
      {
         fprintf( pfpt,  "%s",  "-" );
      }
      fprintf( pfpt,  "%s",  "\n\n\n\n\n\n" );
    }
}


print_header2()
{
   fprintf( pfpt, "%11s%s\n", " ", "Precision of estimates for standard
                                                    errer of mean volume" );
   fprintf( pfpt, "%11s%s\n",  " ", "per hectare for both equal and
                                                      unequal pobability" );
   fprintf( pfpt, "%11s%s",   " ", "subsampling rules for plots:" );
}
```

Source File: **writefile3.h**

```
#include <math.h>

extern  FILE  *cfpt;
extern  int   totplt;
extern  int   NP;
extern  double stdrr[ MAX_YEAR ];
extern  double mvolperha[ MAX_YEAR ];
extern  char  pltlab[ MAX_PLOT ];
extern  char  plotno[ MAX_PLOT ][ 6 ];

write_file3( k, year, msrs, vsrs, mpps, vpps )
int   k, year;
double msrs, vsrs;
```

```
double mpps, vpps;
{
     static int count = 0;

     double serr, aver;
     char   eqm[ 20 ], eqe[ 20 ], uqm[ 20 ], uqe[ 20 ];
     char   size[ 20 ], time[ 20 ];

     if( totplt < NP )
     {
        get_stdr( year );
        get_true_mean( year );
     }
     serr = stdrr[ year ];
     aver = mvolperha[ year ];
     compute_ratio( aver, serr, &msrs, &vsrs, &mpps, &vpps );
     turn_to_char( eqm, eqe, uqm, uqe, size, time,  msrs, vsrs,
                                              mpps,vpps,k,year  );

     if( count == year )
     {
        put_title( time );
        put_result( k, size, uqm, uqe, eqm, eqe );

        count += 1;
     }
     else
     {
        put_result( k, size, uqm, uqe, eqm, eqe );
     }
}


compute_ratio( aver, serr, msrs, vsrs, mpps, vpps )
double aver, serr;
double *msrs, *vsrs;
double *mpps, *vpps;
{
     double eqstdr, uqstdr;

     *vpps = fabs( *vpps );
```

```
    eqstdr = sqrt( *vsrs );
    uqstdr = sqrt( *vpps );
    *vsrs = (eqstdr/serr)*100;
    *vpps = (uqstdr/serr)*100;
    *msrs = (*msrs/aver)*100;
    *mpps = (*mpps/aver)*100;
}


turn_to_char( eqm, eqe, uqm, uqe, size, time, msrs, vsrs, mpps, vpps,
k, year )
char  *eqm, *eqe, *uqm, *uqe;
char  *size, *time;
double msrs, vsrs, mpps, vpps;
int   k,   year;
{
    year = year+1;

    sprintf( eqm,  "%lf",  msrs );
    sprintf( eqe,  "%lf",  vsrs );
    sprintf( uqm,  "%lf",  mpps );
    sprintf( uqe,  "%lf",  vpps );
    sprintf( size, "%d",   k );
    sprintf( time, "%d",   year );
}


put_title( time )
char *time;
{
   int i, j;

   print_header3();

   for( i = 0;   i < totplt;   i++ )
   {
      fprintf( cfpt,  "%5s%s%s",   pltlab, plotno+i, "  " );
   }
   fprintf( cfpt,  "\n\n%-11s",  "Year:" );
   fprintf( cfpt,  "%s%s\n",  "-",  time );
```

```
    fprintf( cfpt,  "%s\n\n",    "(backward)" );

    fprintf( cfpt,  "%-15s",  "Number of" );
    fprintf( cfpt,  "%-15s",  "Mean for" );
    fprintf( cfpt,  "%-17s",  "S. E. for" );
    fprintf( cfpt,  "%-15s",  "Mean. for" );
    fprintf( cfpt,  "%-15s\n", "S. E. for" );

    fprintf( cfpt,  "%-15s",  "Trees in" );
    fprintf( cfpt,  "%-15s",  "Unequal Pr" );
    fprintf( cfpt,  "%-17s",  "Unequal Pr" );
    fprintf( cfpt,  "%-15s",  "Equal Pr" );
    fprintf( cfpt,  "%-15s\n", "Equal Pr" );

    fprintf( cfpt,  "%-15s",  "Subsample" );
    fprintf( cfpt,  "%-15s",  "Subsampling" );
    fprintf( cfpt,  "%-17s",  "Subsampling" );
    fprintf( cfpt,  "%-15s",  "Subsampling" );
    fprintf( cfpt,  "%-15s\n", "Subsampling" );

    fprintf( cfpt,  "%19s%-15s",  " ", "(%)" );
    fprintf( cfpt,  "%-17s",    "(%)" );
    fprintf( cfpt,  "%-15s",    "(%)" );
    fprintf( cfpt,  "%-15s\n",    "(%)" );

    for( j = 0;    j < 73;    j++ )
    {
       fprintf( cfpt,    "%s",   "-" );
    }
    fprintf( cfpt,    "%s",    "\n\n" );
}


put_result( k, size, uqm, uqe, eqm, eqe )
int   k;
char *size;
char *uqm, *uqe;
char *eqm, *eqe;
{
```

```
    int i, j;

    i = maxsample - 1;

    fprintf( cfpt,  "%5s",   size );
    fprintf( cfpt,  "%20s",  uqm );
    fprintf( cfpt,  "%15s",  uqe );
    fprintf( cfpt,  "%17s",  eqm );
    fprintf( cfpt,  "%15s\n", eqe );

    if( i == k )
    {
      fprintf( cfpt,  "%s",  "\n" );

      for( j = 0;   j < 73;   j++ )
      {
         fprintf( cfpt,  "%s",  "-" );
      }
      fprintf( cfpt,  "%s",  "\n\n\n\n\n\n" );
    }
}


print_header3()
{

   fprintf( cfpt,    "%15s%s\n",    " ",    "Accuracy and precision of
                                                estimates for mean" );
   fprintf( cfpt, "%15s%s\n", " ", "volume per hectare, and standard
                                                    error, for " );
   fprintf( cfpt, "%15s%s\n", " ", "both equal and unequal probability
                                                  subsampling " );
   fprintf( cfpt, "%15s%s",   " ", "rules for plots:" );
}
```