DOI: 10.5281/zenodo.5040181

IMPLEMENTING DIFFERENT TYPES OF DATA IN ECONOMIC APPLICATION

Dănuţ-Octavian SIMION, PhD Associate Professor
Athenaeum University, Bucharest, Romania
danut_so@yahoo.com

Emilia VASILE, PhD Professor
Athenaeum University, Bucharest, Romania
rector@univath.ro

Abstract: *The paper presents the implementation of different types of data in economic application. The management information system ensures the obtaining and provision of the information requested by the user, using IT means, to substantiate the decisions regarding a certain field within the company. Current management information systems are integrated systems. They are characterized by the application of the principle of single data entry and multiple processing in accordance with the specific information needs of each user. The integrated IT Accounting System is characterized by a unique data entry, taken from primary documents that update a single accounting database that will be subsequently exploited to ensure both specific works of financial accounting and those specific to management accounting responding - thus the processing requirements of all users. An information system is an intercorrelated set of IT subsystems designed to manage the human, material and financial resources of a company or public institution. Information systems are open systems, which work closely with the company's partners (customers, suppliers, public institutions and financial-banking organizations). Inhomogeneous data is present in every company, so the information systems need to integrate those into different types of structures and objects that are implemented in modules specific to business flows. On-line applications need automated processes that will extract transform and load inhomogeneous data provided by current transactions that appear in every day operations, so the appropriate way to use these data is to load them into data structures that can be implemented into many business applications.*

Keywords: *data structures, data graphs, data trees, economical data, information processes, business flows, application modules, anlysis of data, data algorithms*
JEL Classification: *C23, C26, C38, C55, C81, C87*

1. Introduction

The architecture of the information system represents the generic solution regarding the data processing processes that must be performed and the way of integrating the data and processing. In other words, architecture is the „constructive solution" of the IT system and reflects the strategic managerial vision of how the organization (company) works.

The company's global IT system is broken down into subsystems, each of which covers a distinct field of activity. In turn, each subsystem is broken down into applications, each covering a distinct activity within the field. For example, the IT subsystem for the commercial field will be broken down into distinct applications for each of the following activities: supply, sales, marketing. The decomposition process continues and in the next step for each application will be defined procedures performing distinct functions within the application (example: procedures for directing processing, procedures for updating the database, procedures for consulting the database).

In turn, the procedures are broken down into modules. These comprise code sequences performing a distinct function within the procedure. For example, a database update procedure will include: a module for adding records, a module for modifying records, a module for deleting records.

Data are facts collected from the real world based on observations and measurements. They are any message received from a receiver in a certain form. Data in automatic data processing terminology is defined as a model for representing information in a computer-accessible format. From a logical point of view, the date is defined by: identifier, attribute and value. The data collection is a set of data organized according to certain criteria (Subero, 2021; Edappanavar, 2019).

Data structures are data collections between which a series of relationships have been established that lead to a certain mechanism for selecting and identifying its components.

Depending on the storage medium, the data structures can be:

- in the internal memory of the computer (during data processing), defined by the notions: list, queue, stack;
- on storage media for further processing, defined by: file and database.

An efficient data structure that can do the following operations as quickly as possible: Insert, Search and Delete. The idea behind hashing is to store an item in a table or list, depending on its key. On average, all these operations require one (1) time.

The elements are placed in a statically allocated array on their key positions. By direct addressing, an element with the key k will be stored in location k. All 3 operations are extremely simple (requires only a memory access), but the disadvantage is that this technique „eats" a lot of memory: O (| U |), where U is the universe of keys (Subero, 2021; Nilsson, 2020).

2. The hash table used to store and model data in applications

Search algorithms that use hashes consist of two distinct parts. The first part is to calculate a hash function that turns the search key into a table address. Ideally, different keys would map to different addresses, but often two or more different keys can be distributed to the same table address. The second part of a search algorithm that uses hashing is a collision resolution process. One of the methods of resolving collisions that we will study uses chained lists, and is therefore immediately usable in dynamic situations where the number of search keys is difficult to predict in advance.

Hashing is a good example of a space-time compromise. If there is no memory limitation, then we could do any search with just a memory access simply by using the key as a memory address, just like in indexed search. This ideal often cannot be achieved because the amount of memory required is prohibitive when the keys are long. On the other hand, if there were no time limit, then we could get the answer with only a minimal amount of memory using a sequential search method.

Hashing provides a way to use both a reasonable amount of memory and time to keep a balance between these two extremes. In particular, we can achieve any balance we choose, simply by adjusting the size of the hash table, not by rewriting the code or choosing different algorithms (Edappanavar, 2019; Rana, 2021).

Hashing is a classic problem in computer science: the various algorithms have been thoroughly studied and are widely used. We will see that, based on generous assumptions, it is not unreasonable to expect search and insertion operations in the symbol table to take place in constant time, regardless of the size of the table. This expectation is the theoretically optimal performance for any implementation of the symbol table, but hashing is not a universal panacea, for two main reasons:

- driving time depends on the length of the key, which can be a responsibility in practical applications with long keys.
- hash does not provide efficient implementations for other symbol table operations, such as select or sort.

The first step in solving the memory problem is to use O (N) memory instead of O (| U |), where N is the number of elements added to the hash. What we need to address is the calculation of the hashing function, which turns the keys into table addresses.

Thus, an element with the key k will not be stored in the location k, but in h (k), where h: U -> {0, 1, ..., N-1} - a randomly chosen but deterministic function (h (x) will always return the same value for a given x while running a program). This arithmetic calculation is normally simple to implement, but care must be taken to avoid the various subtle pitfalls.

The hashing function depends on the key type. Strictly speaking, we need a different hashing function for each type of key that could be used. For efficiency, explicit conversion is generally avoided, striving instead for a return to the idea of considering the binary representation of the key in a machine word as a whole that we can use for arithmetic calculations. It was a common practice on early computers to see a key value at one time as a string and at another as a whole (Subero, 2021; Schilasky, 2020).

In some high-level languages it is difficult to write programs that depend on how the keys are represented on a particular computer, because such programs, by their nature, are machine-dependent and therefore not portable. The hash functions are generally dependent on the process of transforming the key into integers, so the independence and efficiency of the machine are sometimes difficult to achieve simultaneously in hashing implementations. We can usually turn simple whole keys or floating point keys with just one machine operation, but string keys and other types of compound keys require more attention and more attention to efficiency (Schilasky, 2020; Edappanavar, 2019).

Key transformation methods:

- String variables can be converted to numbers in base 256 by replacing each character with its ASCII code.
- Data type variables can be converted to integers by the formula: X = A * 366 + L * 31 + Z where A, L and Z are respectively the year, month and day of the considered date. In fact, this function approximates the number of days elapsed since the beginning of the 1st century.
- Analogue, hour variables can be converted to integers with the formula: X = (H * 60 + M) * 60 + S where H, M and S are respectively the hour, minute and second considered, or with the formula X = ((H * 60 + M)

* 60 + S) * 100 if hundreds of seconds are taken into account. This time, the function is surjective (any integer in the range 0 - 8,639,999 uniquely corresponds to one hour).

- In most cases, data are structures that contain numbers and strings. A good conversion method is to paste all this data and convert it to base 256. The characters are converted by simply replacing them with the corresponding ASCII code, and the numbers by converting to base 2 and cutting into "pieces" of eight bit. The result is multi-digit numbers (too many even for the long long type), which are subjected to a division operation with remainder. How? Why? Simplified example: Suppose we have a table with 101 positions and the key AKEY = 00001 01011 00101 11001 (5-bit code) = 4421710 ≡80 (mode 101) Base 32 (signs) => AKEY = $1 * 32^3 + 11 * 32^2 + 5 * 32^1 + 25 * 32^0$

Very often used scattering functions 1. Subtraction method The hash function is: h (x) = x mode M where M is the number of entries in the table. The problem is to choose M as best as possible, so that the number of collisions for any of the inputs is as small as possible. Also, M must be as large as possible, so that the average number of keys assigned to the same input is as small as possible. However, experience shows that not every value of M is good. The hash functions return a number between 0 and M-1, where M is the maximum size of the hash table. It is recommended that M be chosen as a prime number and avoid choosing $M = 2^k$.

Example 1:

Suppose we have a table with 32 positions and the key
Key1 = 00001 01011 00101 11001 (5-bit code) = 4421710 ≡80 (mode 101)

Base 32 (signs) =>

Key2 = $1 * 32^3 + 11 * 32^2 + 5 * 32^1 + 25 * 32^0$

But if

Key3 =

10110001011001011001011000111101110001110101100101110 01 =

$22 * 32^{10} + 5 * 32^9 + 18 * 32^8 + 25 * 32^7 + * 32^6 + 15 * 32^5 + 14 * 32^4 + 7 * 32^3 + 11 * 32^2 + 5 * 32^1 + 25$ = (((((((((22 * 32 + 5) 32 + 18) 32 + 25) 32 + 12) 32 + 15) 32 + 14) 32 + 7) 32 + 11) 32 + 5) 32 + 25 - the value of mod32 would always be the value of the last letter in the key.

Example 2:

For the same reasons, choosing a value like 1000 or 2000 is not very inspiring, because it only takes into account the last 3-4 digits of the decimal representation.

The method of multiplication

The hash function is h (x) = [M * {x * A}] 0 <A <1, and by {x * A} is meant the fractional part of x * A, ie x * A - [x * A].

Example:

If we choose M = 1278 and A = 0.3, and x = 2005, then we have h (x) = [1278 * {601.5}] = [1278 * 0.1] = 127. It is observed that the function h produces numbers between 0 and M- 1. Indeed $0 \leq \{x * A\} < 1$ $0 \leq M * \{x * A\} < M$.

The value of M is no longer very important. M can be as large as we like, possibly a power of 2. In practice, it has been observed that the dispersion is better for some values of A and worse for others;

If the chosen function behaves as close as possible to a random number generation, the elements will be "scattered" in the table evenly. For each input, each output should be in a certain sense, just as likely. Ideally, each item should be stored alone in its location. However, this is not possible, because N <| U | and, therefore, many times more items will be distributed in the same location. This phenomenon is called collision (Rana, 2021; Subero, 2021).

3. Methods of implementation for hash table data

Collision resolution methods:

- Chain
- Static lists
- Open addressing
- Double hashing

Chaining - In each position in the table we keep a chained list; insert, delete and search go through the whole list. In a purely theoretical case, all N elements could be distributed in the same location, but in practical cases the average length of the longest chain is lg (N). Variant: instead of the list, trees.

Improved version of the previous method: because the length of a chain is at most lg (N), we can use, instead of chained lists, dynamically allocated vectors of length lg (N) - or lg (N) + 3 - pointers are removed.

```
// implementation by adapting the procedures from the tables
// symbols for M static lists

#include <stdlib.h>
#include "Item.h"
typedef struct STnode* link;
struct STnode { Item item; link next; };
static link *heads, z;
static int N, M;

void STinit(int max)
{ int i;
N = 0; M = max/5;
heads = malloc(M*sizeof(link));
z = NEW(NULLitem, NULL);
for (i = 0; i < M; i++) heads[i] = z;
}
Item searchR(link t, Key v)
{
if (t== z) return NULLitem;
if (eq(key(t->item), v)) return t->item;
return searchR(t->next, v);
}
Item STsearch(Key v)
{ return searchR(heads[hash(v, M)], v); }

void STinsert(Item item)
{
int I = hash(key(item), M);
heads[i] = NEW(item, heads[i]); N++; }
 void STdelete(Item item)
int i = hash(key(item), M);
heads[i]= deleteR(heads[i], item);
}
```

By open addressing, all elements are stored in the scatter table. To perform the required operations, we successively check the scatter table until we either find a free location (in the case of Insert), or we find the searched item (for Search, Delete). However, instead of looking for the scatter table in the order of 0, 1, ..., N-1, the sequence of examined positions depends on the key to be inserted. To determine the appropriate locations, we extend the hashing function so that it also contains the check number as a second parameter h: U * {0, 1, ..., N-1} -> {0, 1, ... , N-1}.

Thus, when we insert an element, we first check the location h (k, 0), then h (k, 1) etc. When we get to check h (k, N) we can stop because the scatter table is full. We use the same method to search; if we reach h (k, N) or an empty position, it means that the element does not exist. However, deletions are made more difficult, because you cannot simply "delete" an element because

it would damage the entire dispersion table. Instead, mark the location to be deleted with a delte value and change the Insert function so that it sees the locations with the delete value as empty positions.

This implementation of a scatter table keeps the items in a table twice the length of the maximum number of items expected to be entered, items initialized with null values (NULL values).

To insert a new item, we calculate its position in the table, and if it is already occupied it looks to the right using the null macro to test if a position is occupied. To search for an item with a given key we calculate its position and then scan to find the match or stop if we have reached an unoccupied position (Nilsson, 2020; Schilasky, 2020).

The STinit function sets M so that we expect the table to be half full, so that the other operations need few tests if the hashing function produces values close to random values.

```
#include<stdio.h>
#include<conio.h>
void main() {
 int a[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
 int n, value;
 int temp, hash;

 clrscr();
 printf("\nEnter the value of n(table size):");
 scanf("%d", &n);
 do {
  printf("\nEnter the hash value");
  scanf("%d", &value);
  hash = value % n;
  if (a[hash] == 0) {
   a[hash] = value;
   printf("\na[%d]the value %d is stored", hash, value);
  } else {

   for (hash++; hash < n; hash++) {
    if (a[hash] == 0) {
     printf("Space is allocated give other value");
     a[hash] = value;
     printf("\n a[%d]the value %d is stored", hash, value);
     goto menu;
    }
   }

 for (hash = 0; hash < n; hash++) {
    if (a[hash] == 0) {
     printf("Space is allocated give other value");
```

```
      a[hash] = value;
      printf("\n a[%d]the value %d is stored", hash, value);
      goto menu;
     }
    }
    printf("\n\nERROR\n");
    printf("\nEnter '0' and press 'Enter key' twice to exit");

   }

   menu:

   printf("\n Enter more values?");
   scanf("%d", &temp);

 }
 while (temp == 1);
 getch();
}
```

Properties for hash tables:

- Separate chaining reduces the number of sequential search comparisons by an M factor (on average), using extra space for M chains.
- In a dispersion table that is less than 2/3 full open addressing requires less than 5 tests.

A big improvement on the scatter table is another scatter chart. We will have 2 tables, each with its own hashing function, and we solve collisions by chaining; when we insert an element, we will add it to the table where it falls into a shorter chain. The search is done in both tables in the locations returned by the 2 hashing functions; deleting as well. The length of the longest chain will be, on average, lg (lg (N)). In practice, the length of such a chain will not exceed 4 elements, because the smallest N for which lg (lg (N)) = 5. Instead of lists we use static vectors of size 32 (Khadda, 2020; Nilsson, 2021).

4. Conclusions

Economic application systems fulfill operational, managerial and strategic support roles in businesses and organizations, and can be grouped into information systems for enterprise functions, operational information systems and managerial information systems (Schilasky, 2020; Khadda, 2020). It is important for a manager to understand that economic applications directly supports the organization's operational and managerial functions in accounting, finance, human resources, marketing and operational management. Operational

economical processes data generated and used in business operations (Khadda 2020; Edappanavar, 2019). Depending on their role, there are several categories: transaction processing systems, record and process data resulting from transactions, update databases and produce a variety of documents and reports; process control systems that provide operational decisions that control physical processes; automated service systems for those that support communications. The mixed data existing in a company may be stored in different types of data, such as hash tables that optimize obsevational and series of values and so these can be processed through aptimal algorithms. An amount of many operations that are necessary upon structured data, make the storage in hash tables a good example of how to get quick results for better representation of economic facts that can change data flows in great access to various details.

References

Edappanavar, Shubham. (2019). *C++ Program to implement hash tables*, [online]. Available at: www.cppsecrets.com, https://cppsecrets.com/user/index.php?uid=665.

Khadda, Sanjay. (2020). *Hashtables chaining with doubly linked lists*, [online]. Available at: www.geeksforgeeks.org.

Nilsson, Per. (2020). *Introduction to hash tables*, [online]. Available at: https://www.codeguru.com/cpp/cpp/algorithms/hash/article.php/c5131/Introduction-to-Hash-Tables.htm.

Rana, Shubham. (2021). *C++ program for hashing with chaining,* [online]. Available at: https://www.geeksforgeeks.org/c-program-hashing-chaining/.

Schilasky, Rex. (2020). *Easy-to-use hash table,* [online]. Available at: https://www.codeguru.com/cpp/cpp/algorithms/hash/article.php/c5101/EasytoUse-Hash-Table.htm.

Subero, Armstrong. (2021). *The codeless guide to hashing and hash tables*, [online]. Available at: https://www.freecodecamp.org/news/the-codeless-guide-to-hash/.