# 1 Theoretical surroundings

***Abstract***

*Firstly in this chapter, we will remember the needed basic mathematical concepts. Special attention will be paid to the Names Sets – mathematical structure which is used further for building models needed for our research. We will use strong hierarchies of named sets to create a specialized mathematical model for new kind of organization of information bases called "Multi-Domain Information Model" (MDIM). The "information spaces" defined in the model are kind of strong hierarchies of enumerations (named sets).*

*At the end, we will remember the main features of hashing and types of hash tables as well as the idea of "Dynamic perfect hashing" and "Trie", especially – the "Burst trie". Hash tables and tries give very good starting point. The main problem is that they are designed as structures in the main memory which has limited size, especially in small desktop and laptop computers. For practical implementation of NLA we need a proper model for database organization and corresponded specialized tools. To achieve such possibilities, we will use "Multi-Domain Information Model" (MDIM) and corresponded to it software tools to realize dynamic perfect hashing and burst tries as external memory structures.*

## 1.1 Basic mathematical concepts

Let remember the basic mathematical concepts needed for this research [Bourbaki, 1960, Burgin, 2010].

$\varnothing$ is the *empty set*.

If $X$ is a set, then $r \in X$ means that r belongs to $X$ or $r$ is a member of $X$.

If $X$ and $Y$ are sets, then $Y \subseteq X$ means that $Y$ is a subset of $X$, i.e., $Y$ is a set such that all elements of $Y$ belong to $X$.

The *union* $Y \cup X$ of two sets $Y$ and $X$ is the set that consists of all elements from $Y$ and from $X$.

The *intersection* $Y \cap X$ of two sets $Y$ and $X$ is the set that consists of all elements that belong both to $Y$ and to $X$.

The *union* $\bigcup_{i\in I} X_i$ of sets $X_i$ is the set that consists of all elements from all sets $X_i$, $i \in I$.

The *intersection* $\bigcap_{i\in I} X_i$ of sets $X_i$ is the set that consists of all elements that belong to each set $X_i$, $i\in I$.

The *difference* $Y\backslash X$ of two sets $Y$ and $X$ is the set that consists of all elements that belong to $Y$ but does not belong to $X$.

If $X$ is a set, then $2^X$ is the *power set* of $X$, which consists of all subsets of $X$. The power set of $X$ is also denoted by $\boldsymbol{P}(X)$.

If $X$ and $Y$ are sets, then $X \times Y = \{(x, y);\ x \in X, y \in Y\}$ is the direct or Cartesian product of $X$ and $Y$, in other words, $X \times Y$ is the set of all pairs $(x, y)$, in which $x$ belongs to $X$ and $y$ belongs to $Y$.

Elements of the set $X^n$ have the form $(x_1, x_2, \ldots, x_n)$ with all $x_i \in X$ and are called $n$-tuples, or simply, tuples.

A fundamental structure of mathematics is *function*. However, functions are special kinds of binary relations between two sets.

A *binary relation* $T$ between sets $X$ and $Y$ is a subset of the direct product $X \times Y$. The set $X$ is called the *domain* of $T$ ($X = \mathrm{Dom}(T)$) and $Y$ is called the *codomain* of $T$ ($Y = \mathrm{CD}(T)$). The *range* of the relation $T$ is $\mathrm{Rg}(T) = \{y;\ \exists\, x \in X\, ((x, y) \in T)\}$. The *domain of definition* of the relation $T$ is $\mathrm{DDom}(T) = \{x;\ \exists\, y \in Y\, ((x, y) \in T)\}$. If $(x, y) \in T$, then one says that the elements $x$ and $y$ are in relation $T$, and one also writes $T(x, y)$.

Binary relations are also called multi valued functions (mappings or maps).

$Y^X$ is the set of all mappings from $X$ into $Y$.

$$X^n = \underbrace{X \times X \times \ldots X \times X}_{\mathrm{n}}.$$

A *preorder* (also called *quasiorder*) on a set $X$ is a binary relation Q on X that satisfies the following axioms:

1. Q is reflexive, i.e. $x\mathrm{Q}x$ for all $x$ from $X$.
2. Q is transitive, i.e., $x\mathrm{Q}y$ and $y\mathrm{Q}z$ imply $x\mathrm{Q}z$ for all $x, y, z \in X$.

A *partial order* is a preorder that satisfies the following additional axiom:

3. Q is antisymmetric, i.e., $x\mathrm{Q}y$ and $y\mathrm{Q}x$ imply $x = y$ for all $x, y \in X$.

A *strict partial order* is a preorder that is not reflexive, is transitive and satisfies the following additional axiom:

4. Q is asymmetric, i.e., only one relation $x\mathrm{Q}y$ or $y\mathrm{Q}x$ is true for all $x, y \in X$.

Equivalence on a set $X$ is a binary relation Q on $X$ that is reflexive, transitive and satisfies the following additional axiom:

5. Q is symmetric, i.e., $x\mathrm{Q}y$ implies $y\mathrm{Q}x$ for all $x$ and $y$ from $X$.

A *function* (also called a *mapping* or *map* or *total function* or *total mapping*) $f$ from $X$ to $Y$ is a binary relation between sets $X$ and $Y$ in which:

- There are no elements from X which are corresponded to more than one element from Y;
- To any element from X, some element from Y is corresponded.

Often total functions are also called everywhere defined functions. Traditionally, the element $f(a)$ is called the image of the element $a$ and denotes the value of $f$ on the element $a$ from $X$. At the same time, the function $f$ is also denoted by $f$: $X \to$ Y or by $f(x)$. In the latter formula, $x$ is a variable and not a concrete element from $X$.

A *partial function* (or *partial mapping*) $f$ from $X$ to $Y$ is a binary relation between sets $X$ and $Y$ in which there are no elements from $X$ which are corresponded to more than one element from $Y$.

Thus, any function is also a partial function. Sometimes, when the domain of a partial function is not specified, we call it simply a function because any partial function is a total function on its domain.

A *multi valued function* (or *mapping*) $f$ from $X$ to $Y$ is any binary relation between sets $X$ and $Y$.

$f(x) \equiv a$ means that the function $f(x)$ is equal to $a$ at all points where $f(x)$ is defined.

Two important concepts of mathematics are the domain and range of a function. However, there is some ambiguity for the first of them. Namely, there are two distinct meanings in current mathematical usage for this concept. In the majority of mathematical areas, including the calculus and analysis, the term "domain of $f$" is used for the set of all values $x$ such that $f(x)$ is defined. However, some mathematicians (in particular, category theorists), consider the domain of a function $f$: $X \to Y$ to be $X$, irrespective of whether $f(x)$ is defined for all $x$ in $X$. To eliminate this ambiguity, we suggest the following terminology consistent with the current practice in mathematics.

If $f$ is a function from $X$ into $Y$, then the set $X$ is called the *domain* of $f$ (it is denoted by Dom$f$) and $Y$ is called the *codomain* of $T$ (it is denoted by Codom$f$). The *range* Rg$f$ of the function $f$ is the set of all elements from $Y$ assigned by $f$ to, at least, one element from $X$, or formally, Rg$f = \{y; \exists\, x \in X\,(f(x) = y)\}$. The *domain of definition* DDom$f$ of the function $f$ is the set of all elements from $X$ that related by $f$ to, at least, one element from $Y$ is or formally, DDom$f = \{x; \exists\, y \in Y\,(f(x) = y)\}$. Thus, for a partial function $f(x)$, its domain of definition DDom$f$ is the set of all elements for which $f(x)$ is defined.

Taking two mappings (functions) $f$: $X \to$ Y and $g$: $Y \to Z$, it is possible to build a new mapping (function) $gf$: $X \to Z$ that is called *composition* or *superposition* of mappings (functions) $f$ and g and defined by the rule $gf(x) = g(f(x))$ for all $x$ from $X$.

An $n$-ary relation $R$ in a set $X$ is a subset of the $n$th power of $X$, i.e., $R \subseteq X^n$. If $(a_1, a_2, \ldots, a_n) \in R$, then one says that the elements $a_1, a_2, \ldots, a_n$ from $X$ are in relation $R$.

## ➢ ***Named sets***

*Named set* **X** is a triple $\mathbf{X} = (X, \mu, I)$ where:

- $X$ is the *support* of **X** and is denoted by S(**X**);
- $I$ is the *component of names* (also called *set of names* or *reflector*) of **X** and is denoted by N(**X**);
- $\mu: X \to I$ is the *naming map* or *naming correspondence* (also called *reflection*) of the named set **X** and is denoted by n(**X**).

The most popular type of named sets is a named set $\mathbf{X} = (X, \mu, I)$ in which $X$ and $I$ are sets and $\mu$ consists of connections between their elements. When these connections are set theoretical, i.e., each connection is represented by a pair $(x, a)$ where $x$ is an element from $X$ and $a$ is its name from $I$, we have a *set theoretical named set*, which is binary relation.

A name $a \in I$ is called *empty* if $\mu^{-1}(a) = \varnothing$.

A named set **X** is called:

- *Normalized* if in **X** there are no empty names;
- *Conormalized* if in **X** there no elements without names;

Named sets as special cases include:

- Usual sets;
- Fuzzy sets;
- Multisets;
- Enumerations;
- Sequences (countable as well as uncountable);

etc.

A lot of examples of named sets we may find in linguistics studying semantical aspects that are connected with applying different elements of language (words, phrases, texts) to their meaning [Burgin & Gladun, 1989; Burgin, 2010].

A named set $\mathbf{Y} = (Y, \eta, J)$ is called *named subset* of named set **X** if $Y \subseteq X$, $J \subseteq I$, and $\eta = \mu |_{(Y,J)}$ ($\eta \subseteq \mu \cap (Y \times J)$). In this case **Y** and **X** are connected by the relation of the inclusion.

An ordered tuple of named sets $\Theta = [\mathbf{X}_1, \mathbf{X}_2, ..., \mathbf{X}_k]$ where for all $i$=1, ..., k-1 the condition $N(\mathbf{X}_i) \cap S(\mathbf{X}_{i+1}) \neq \varnothing$ is fulfilled is called *chain of named sets*.

The number k is called a length of the chain Θ.

A tuple of named sets $\Xi_1 = [\mathbf{X}, \mathbf{Y}_1, \mathbf{Y}_2, ..., \mathbf{Y}_n]$ where for all $i$=1,...,n the condition $N(\mathbf{Y}_i) \cap S(\mathbf{X}) \neq \varnothing$ is fulfilled is called *one level hierarchy of named sets*.

If $N(\mathbf{Y}_i) \cap N(\mathbf{Y}_j) = \varnothing$ and $N(\mathbf{Y}_i) \subseteq S(\mathbf{X})$ for all $i$=1,...,n, $j$=1,...,n than Ξ is a *strong one level hierarchy of named sets*.

A tuple of named sets $\Xi_2 = [\mathbf{X}, \Xi_{1,1}, \Xi_{1,2}, ..., \Xi_{1,m}]$ where *sub-hierarchies* $\Xi_{1j} = [\mathbf{Y}_j, \mathbf{Z}_1, \mathbf{Z}_2, ..., \mathbf{Z}_k]$ , $j$=1,...,m are one level hierarchy of named sets is called *second level hierarchy of named sets*.

If $\Xi_{1j}$, $j$=1,...,m, are strong one level hierarchies of named sets than $\Xi_2$ is a *strong second level hierarchy of named sets*.

A tuple of named sets $\Xi_n = [\mathbf{X}, \Xi_{n-1,1}, \Xi_{n-1,2}, ..., \Xi_{n-1,l}]$ where $\Xi_{n-1,i}$ , $i$=1,...,l are n-1 level hierarchies of named sets than $\Xi_n$ is a *n-th level hierarchy of named sets*..

If all sub-hierarchies of $\Xi_n$ are strong hierarchies of named sets than $\Xi_n$ is a *strong n-th level hierarchy of named sets*.

## 1.2 Hashing

A *set abstract data type* (set ADT) is an abstract data type that maintains a set ***S*** under the following three operations:

1. *Insert(x)*: Add the key *x* to the set.
2. *Delete(x)*: Remove the key *x* from the set.
3. *Search(x)*: Determine if *x* is contained in the set, and if so, return a pointer to *x*.

One of the most practical and widely used methods of implementing the set ADT is with *hash tables* [Morin, 2005].

The simplest implementation of such data structure is an ordinary array, where *k*-th element corresponds to key *k*. Thus, we can execute all operations in O(1). It is impossible to use this implementation, if the total number of keys is large [Kolosovskiy, 2009].

The main idea behind all hash table implementations is to store a set of $n = |\boldsymbol{S}|$ elements in an array (the hash table) *A* of length m. In doing this, we require a function that maps any element *x* to an array location. This function is called a *hash function h* and the value *h(x)* is called the *hash value of x*. That is, the element *x* gets stored at the array location $A[h(x)]$.

The occupancy of a hash table is the ratio $\alpha = n/m$ of stored elements to the length of *A* [Morin, 2005].

We have two cases: (1) m ≥ n and (2) m ≤ n:

- In the first case (m ≥ n) we may expect so called ***perfect hashing*** where every element may be stored in separate cell of the array. In other words, if we have a collection of n elements whose keys are unique integers in (1, m), where m ≥ n, then we can store the items in a direct address table, T[m], where $T_i$ is either empty or contains one of the elements of our collection.
- In the second case (m ≤ n) we may expect so called "***collisions***" when two or more elements have to be stored in the same cell f the array.

If we work with two or more keys, which have the *same hash value*, these keys map to the same cell in the array. Such situations are called *collisions*. There are two basic ways to implement hash tables to resolve collisions:

- Chained hash table;
- Open-address hash table.

In ***chained hash table*** each cell of the array contains the linked list of elements, which have corresponding hash value. To add (delete, search) element in the set we add (delete, search) to corresponding linked list. Thus, time of execution depends on length of the linked lists.

In ***open-address hash table*** we store all elements in one array and resolve collisions by using other cells in this array. To perform insertion we examine some slots in the table, until we find an empty slot or understand that the key is contained in the table. To perform search we execute similar routine [Kolosovskiy, 2009].

The study of hash tables follows two very different lines: (1) integer universe assumption; (2) random probing assumption.

**Integer universe assumption:** All elements stored in the hash table come from the universe $U = \{0,...,u-1\}$. In this case, the goal is to design a hash function $h : U \rightarrow \{0,...,m-1\}$ so that for each $i \in \{0,...,m-1\}$, the number of elements $x \in \boldsymbol{S}$ such that $h(x) = i$ is as small as possible. Ideally, the hash function *h* would be such that each element of $\boldsymbol{S}$ is mapped to a unique value in $\{0,...,m-1\}$.

Historically, the **integer universe assumption** seems to have been justified by the fact that any data item in a computer is represented as a *sequence of bits that can be interpreted as a binary number*.

However, many complicated data items require a large (or variable) number of bits to represent and this make the size of the universe very large. In many applications $u$ is much larger than the largest integer that can fit into a single word of computer memory. In this case, *the computations performed in number-theoretic hash functions become inefficient*. This motivates the second major line of research into hash tables, based on *Random probing assumption*.

**Random probing assumption:** Each element $x$ that is inserted into a hash table is a black box that comes with an infinite random probe sequence $x_0, x_1, x_2, \ldots$ where each of the $x_i$ is independently and uniformly distributed in $\{0, \ldots, m-1\}$.

Both the integer universe assumption and the random probing assumption have their place in practice.

When there is an easily computing mapping of data elements onto machine word sized integers then hash tables for integer universes are the method of choice.

When such a mapping is not so easy to compute (variable length strings are an example) it might be better to use the bits of the input items to build a good pseudorandom sequence and use this sequence as the probe sequence for some random probing data structure [Morin, 2005].

## ➢ *Perfect hash function*

We consider hash tables under the *integer universe assumption*, in which the key values $x$ come from the universe $U = \{0, \ldots, u-1\}$. A hash function $h$ is a function whose domain is $U$ and whose level is the set $\{0, \ldots, m-1\}$, $m \le u$.

A hash function h is said to be a ***perfect hash function*** for a set S ⊆ U if, ***for every x ∈ S, h(x) is unique.***

A *perfect hash function h* for *S* is ***minimal*** if $m = |S|$, i.e., $h$ is a bisection between $S$ and $\{0, \ldots, m-1\}$. Obviously a minimal perfect hash function for $S$ is desirable since it allows us to store all the elements of $S$ in a single array of length $n$. Unfortunately, perfect hash functions are rare, even for $m$ much larger than $n$ [Morin, 2005].

The set of elements, *S*, may be:

– *Static* (no updates);

– *Dynamic* where fast queries, insertions, and deletions must be made on a large set.

*"**Dynamic perfect hashing**"* is useful for the second type of situations. In this method, the entries that hash to the same slot of the table are organized as separate *second-level hash table*. If there are $k$ entries in this set $S$, the second-level table is allocated with $k^2$ slots, and its hash function is selected at random from a universal hash function set so that it is *collision-free* (i.e. a perfect hash function). Therefore, the look-up cost is guaranteed to be O(1) in the worst-case [Dietzfelbinger et al, 1994].

*Perfect hashing* can be used in many applications in which we want to assign a unique identifier to each key without storing any information on the key. One of the most obvious applications of perfect hashing (or k-perfect hashing) is when we have a small fast memory in which we can store the perfect hash function while the keys and associated satellite data are stored in slower but larger memory. The size of a block or a transfer unit may be chosen so that $k$ data items can be retrieved in one read access. In this case we can ensure that data associated with a key can be retrieved in a single probe to slower memory. This has been used for example in hardware routers. Perfect hashing has also been found to be competitive with traditional hashing in internal memory on standard computers. *Recently perfect hashing has been used* ***to accelerate algorithms on graphs*** *when the graph representation does not fit in main memory* [Belazzougui et al, 2009].

For the purposes of Natural Language Addressing (NLA) we need possibility to use *perfect hashing with dynamic and very large (practically – unlimited) set, S, of elements with variable length of strings*. In this case, the computing mapping of data elements onto machine word sized integers is not so easy to compute (we have long strings with variable length). In the same time, we could not use the bits of the input items to build a good pseudorandom sequence and use this sequence as the probe sequence for some random probing data structure, because of very large, unlimited, set, *S*, of elements.

## 1.3 Tries

*"As defined by me, nearly 50 years ago, it is properly pronounced "tree" as in the word "retrieval". At least that was my intent when I gave it the name "Trie". The idea behind the name was to combine reference to both the structure (a tree structure) and a major purpose (data storage and retrieval)".*

*Edward Fredkin, July 31, 2008*

**Trie** is a tree for storing strings in which there is one node for every common prefix. The strings are stored in extra leaf nodes.

A *trie* can be thought of as an *m*-ary tree, where *m* is the number of characters in the alphabet. A search is performed by examining the key one character at a time and using an *m*-way branch to follow the appropriate path in the trie, starting at the root. In other words, in the *multi-way trie* (Figure 5), each node has a potential child for each letter in the alphabet. Below is an example of a multi-way trie indexing the three words BE, BED, and BACCALAUREATE [Pfenning, 2012].

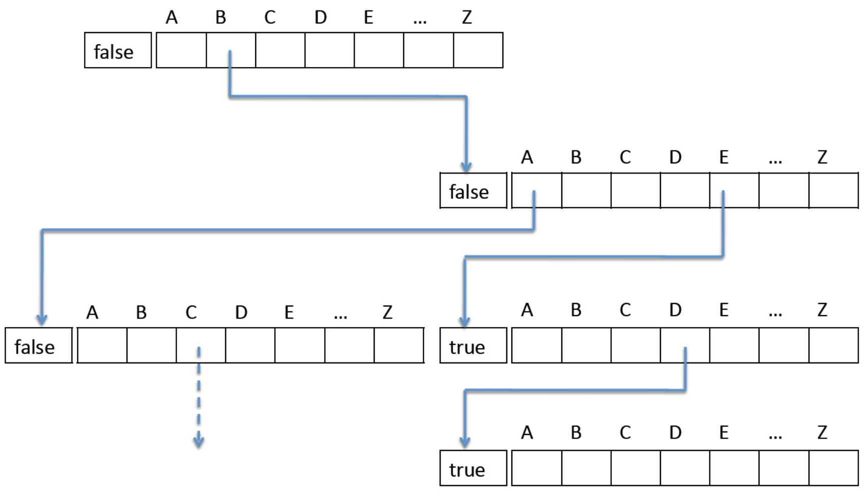

***Figure 5. Example of multi-way trie [Pfenning, 2012]***

*Tries* are distinct from the other data structures because they explicitly assume that the keys are a sequence of values over some (finite) alphabet, rather than a single indivisible entity. Thus tries are particularly well-suited for handling variable-length keys. Also, when appropriately implemented, tries can provide compression of the set represented, because common prefixes of words are combined together; words with the same prefix follow the same search path in the trie [Sahni, 2005].

To illustrate trie [Liang, 1983] had used the set of 31 most common English words (Figure 6):

| | | | |
|---|---|---|---|
| A | FOR | IN | THE |
| AND | FROM | IS | THIS |
| ARE | HAD | IT | TO |
| AS | HAVE | NOT | WAS |
| AT | HE | OF | WHICH |
| BE | HER | ON | WITH |
| BUT | HIS | OR | YOU |
| BY | I | THAT | |

***Figure 6. The 31 most common English words [Liang, 1983]***

Figure 7 shows a linked trie representing this set of words. In a linked trie, the m-way branch is performed using a sequential series of comparisons.

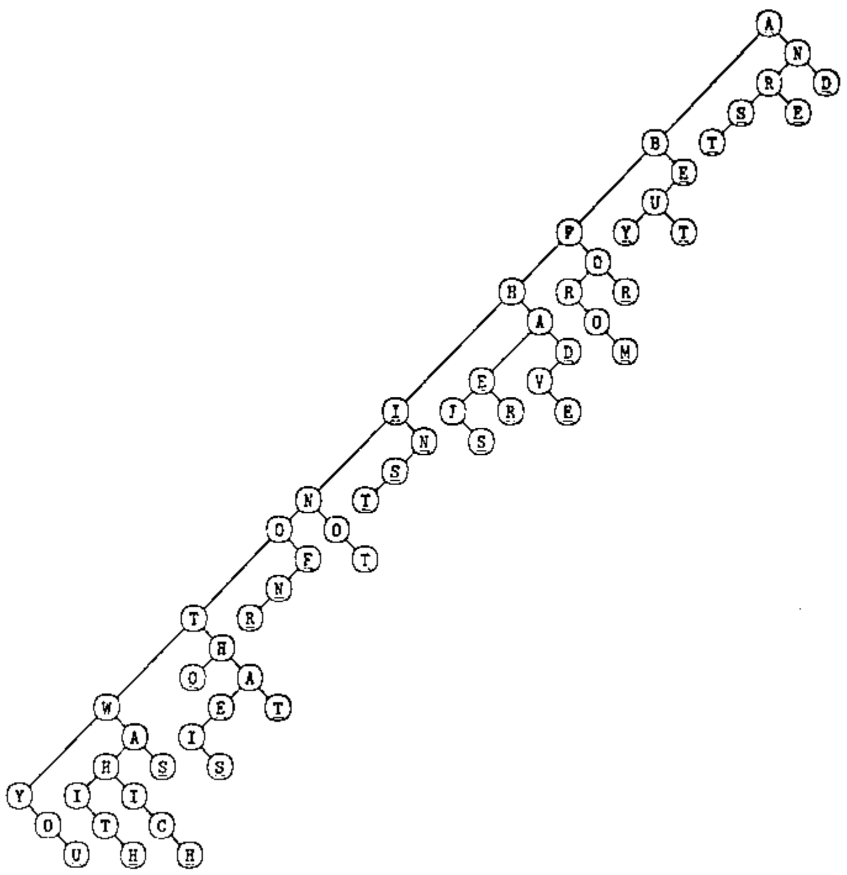

***Figure 7. Linked trie for the 31 most common English words [Liang, 1983].***

Suppose that the elements in our dictionary are student records that contain fields such as student name and social security number (SS#) [Sahni, 2005]. The key field is the social security number, which is a nine digit decimal number. To keep the example manageable, assume we have only five elements.

Table 1 shows the name and SS# fields for each of the five elements in our dictionary.

***Table 1. Five students' records [Sahni, 2005]***

| **Name** | **Social Security Number (SS#)** |
|---|---|
| Jack | 951-94-1654 |
| Jill | 562-44-2169 |
| Bill | 271-16-3624 |
| Kathy | 278-49-1515 |
| April | 951-23-7625 |

To obtain a trie representation for these five elements, we first select a radix that will be used to decompose each key into digits. If we use the radix 10, the decomposed digits are just the decimal digits shown in Table 1. We shall examine the digits of the key field (i.e., SS#) from left to right. Using the first digit of the SS#, we partition the elements into three groups–elements whose SS# begins with 2 (i.e., Bill and Kathy), those that begin with 5 (i.e., Jill), and those that begin with 9 (i.e., April and Jack). Groups with more than one element are partitioned using the next digit in the key. This partitioning process is continued until every group has exactly one element in it (Figure 8) [Sahni, 2005].

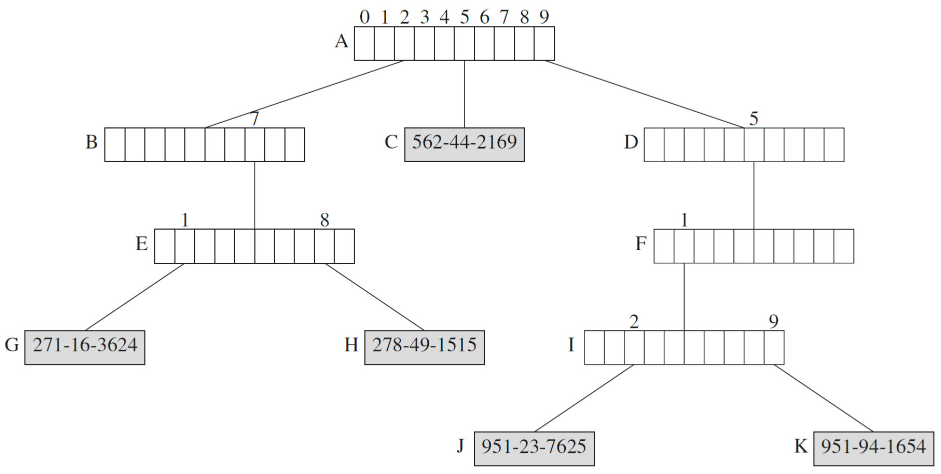

***Figure 8. Trie for the elements of Table 1 [Sahni, 2005]***

The partitioning process described above naturally results in a tree structure that has 10-way branching as is shown in Figure 8. The tree employs two types of nodes:

- Branch nodes;
- Element nodes.

Each branch node has 10 children (or pointer/reference) fields. These fields, child[0 : 9], have been labeled 0, 1, ..., 9 for the root node of Figure 8 ***root.child[i]*** points to the root of a sub-trie that contains all elements whose first digit is ***i***.

In Figure 8, nodes A, B, D, E, F, and I are branch nodes.

The remaining nodes, nodes C, G, H, J, and K are element nodes. Each element node contains exactly one element. In Figure 8, only the key field of each element is shown in the element nodes.

## ➢ *Burst Tries*

The tree data structures compared to hashing have three sources of inefficiency [Heinz et al, 2002]:

- First, the average search lengths is surprisingly high, typically exceeding ten pointer traversals and string comparisons even on moderate-sized data sets with highly skew distributions. In contrast, a search under hashing rarely requires more than a string traversal to compute a hash value and a single successful comparison;
- Second, for structures based on Binary Search Trees (BSTs), the string comparisons involved redundant character inspections, and were thus unnecessarily expensive. For example, given the query string “middle” and given that, during search, “Michael” and “midfield” have been encountered, it is clear that all subsequent strings inspected must begin with the prefix “mi”;
- Third, in tries the set of strings in a sub-trie tends to have a highly skew distribution: typically the vast majority of accesses to a sub-trie are to find one particular string. Thus use of a highly time-efficient, space-intensive structure for the remaining strings is not a good use of resources [Heinz et al, 2002].

These considerations led to the burst trie. A ***burst trie*** is an *in-memory* data structure, designed for sets of records that each has a unique string that identifies the record and acts as a key. Formally, a string ***s*** with length *n* consists of a series of symbols or characters $c_i$ for $i$=0;...;$n$, chosen from an alphabet A of size |A|. It is assumed that |A| is small, typically no greater than 256 [Heinz et al, 2002].

A ***burst trie*** consists of three distinct components (Figure 9): a set of records, a set of containers, and an access trie:

- ***Records***. A record contains a string; information as required by the application using the burst trie (that is, for information such as statistics or word locations); and pointers as required to maintain the container holding the record. *Each string is unique*;
- ***Containers***. A container is a small set of records, maintained as a simple data structure such as a list or a *binary search tree* (BST). For a container at depth *k* in a burst trie, all strings have length at least k and the first k characters of all strings are identical. It is not necessary to store these first k characters. Each container also has a header, for storing the statistics used by heuristics for bursting. Thus a particular container at depth 3 containing “author” and “automated” could also contain “autopsy” but not “auger”;
- ***Access trie***. An access trie is a trie whose leaves are containers. Each node consists of an array *p*, of length |A|, of pointers, each of which may point to either a trie node or a container, and a single empty-string pointer to a record. The |A| array locations are indexed by the characters $c \in$ A. The remaining pointer is indexed by the empty string.

The depth of the root is defined to be 1. Leaves are at varying depths.

A burst trie can be viewed as a generalization of other proposed variants of trie.

Figure 9 shows an example of a burst trie storing ten records whose keys are “came”, “car”, “cat”, “cave”, “cy”, “cyan”, “we”, “went”, “were”, and “west” respectively. In this example, the alphabet A is the set of letters from A to Z, and in addition an empty string symbol ⊥ is shown; the container structure used is a BST. In this figure, the access trie has four nodes, the deepest at depth 3. The leftmost container has four records, corresponding to the strings “came”, “car”, “cat”, and “cave”. One of the strings in the rightmost container is “⊥”, corresponding to the string “we”. The string “cy” is stored wholly within the access trie, as shown by the empty-string pointer to a record, indexed by the empty string [Heinz et al, 2002].

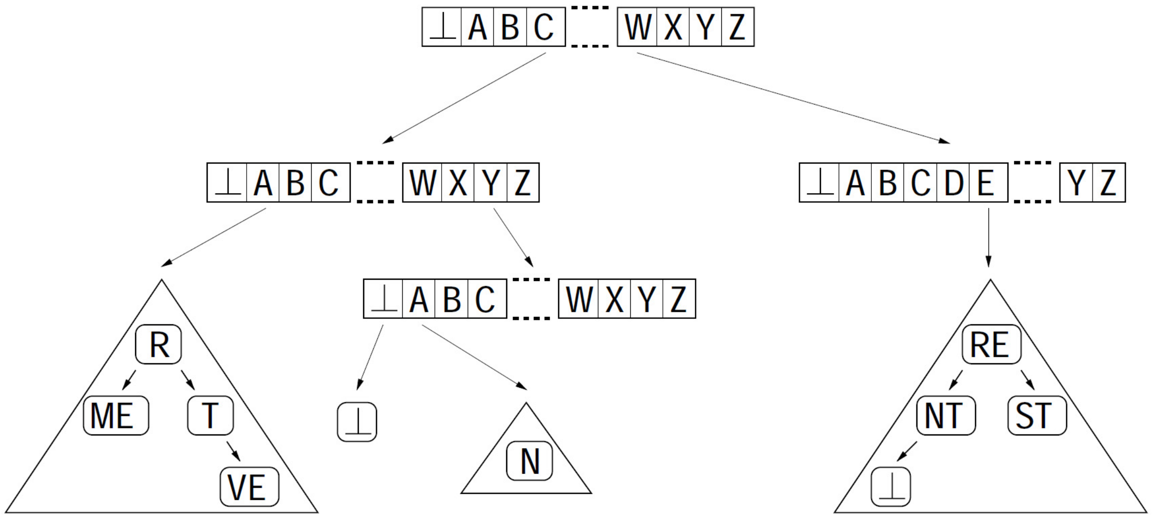

***Figure 9. Burst trie with BSTs used in containers [Heinz et al, 2002]***

***Conclusion of Chapter 1***

*This chapter was aimed to introduce the theoretical surroundings of our work.*

*Firstly in this chapter, we remembered the needed basic mathematical concepts. Special attention was paid to the Names Sets – mathematical structure which we implemented in our research. We used strong hierarchies of named sets to create a specialized mathematical model for new kind of organization of information bases called “Multi-Dmain Information Model” (MDIM). The “information spaces” defined in the model are kind of strong hierarchies of enumerations (named sets).*

*We will realize MDIM via special kind of hashing. Because of this, we remembered the main features of hashing and types of hash tables as well as the idea of “Dynamic perfect hashing” and “Trie”, especially – the “Burst trie”. A* ***burst trie*** *is an in-memory data structure, designed for sets of records that each has a unique string that identifies the record and acts as a key. Burst trie consists of three distinct components: a set of records, a set of containers, and an access trie.*